Planning Automated Guided Vehicle Movements in a Factory

James Boon Hwee Kwa

PhD
University of Edinburgh
1988

To
Lai Ying
and
Mei-En

Abstract

This dissertation examines the problems of planning automated guided vehicle (AGV) movement schedules in an automated factory. AGVs are used mainly for material delivery and will have an important role in linking "islands of automation" in automated factories. Their employment in this context requires the plans to be generated in a manner which supports temporal projection so that further planning in other areas is possible. Planning also occurs in a dynamic scenario—while some plans are being executed, planning for new tasks and replanning failing plans occur. Expeditious planning is thus important so that deadlines can be met. Furthermore, dynamic replanning in a multi-agent environment has repercussions—changing one plan may require revision of other plans. Hence the issue of limiting the side effects of dynamic replanning is also considered. In dealing with these issues, the goals of this research are: (1) generate movement plans which can be executed efficiently; (2) develop fast algorithms for the recurrent subproblems *viz.* task assignment and route planning; and (3) generate robust plans which tolerate execution deviations; this helps to minimize disruptive dynamic replanning with its tendency to initiate a chain reaction of plan revisions.

Efficient movement plans mean more productive utilization of the AGV fleet and this objective can be realized by three approaches. First, the tasks are assigned to AGVs optimally using an improved implementation of the Hungarian method. Second, the planner computes shortest routes for the AGVs using a bidirectional heuristic search algorithm which is amenable to parallel implementation for further computational time reduction. Third, whenever AGVs are fortuitously predisposed to assist each other in task execution, the planner will generate gainful collaborative plans. Efficient algorithms have been developed in these areas. The algorithms for task assignment and route planning are also designed to be fast, in keeping with the objective of expeditious planning.

Robust plans can be generated using the approach of tolerant planning. Robustness is achieved in two ways: (1) by being tolerant of an AGV's own execution deviations; and (2) by being tolerant of other AGVs' deviant behaviour. Tolerant planning thus defers dynamic replanning until execution errors become excessive. The underlying strategy is to provide more than ample resources (time) for AGVs to achieve various subgoals. Such redundancies aggravate the resource contention problem. To solve this, an iterative negotiation model is proposed. During negotiations, AGVs yield in turn to help eliminate the conflict. The negotiation behaviour of each is governed by how much spare resources each has and tends towards intransigence as the bottom line is approached. In this way, no AGV will jeopardize its own plan while cooperating in the elimination of conflicts. By gradual yielding, an AGV is also able to influence the other party to yield more if it can, therein achieving some fairness. The model has many of the characteristics of negotiation acts in the real world (*e.g.* skilful negotiation, intransigence, selfishness, willingness to concede, nested negotiations).

Acknowledgements

I am grateful to several people who have been a source of help and encouragement. Foremost of all, my wife, Lai Ying, has been my constant companion and has sacrificed much of her own interests to enable me to devote more time and effort to this research. When I was discouraged by lack of progress, her simple yet effective words have reminded me of the need to look again at my objectives and persevere.

I also wish to thank my supervisors, John Hallam and Austin Tate, for their guidance and helpful comments throughout my research. Many of the mistakes and obscurities in earlier drafts have been weeded out due to their sharp-eyed and critical reading. I am also grateful to Jim Howe for his supervision in my first year as a research student. I owe it to him for his suggestion to consider the planning domain for a research project. Mark Drummond and Ken Currie in the AIAI Planning Group have also helped in their suggestions and observations. This research would have taken longer to complete had it not been for the constant encouragement of these people.

Many friends in Edinburgh have helped to make my sojourn in Scotland enjoyable. Looi Chee-Kit and Ng Siew Kheng have also graciously consented to proofread the chapters. I am grateful for their effort and willingness to suggest possible improvements.

The computing facilities donated by Xerox Corporation were used in the software development pertaining to this research and in the composition of this dissertation.

This work was supported by a scholarship from the Government of the Republic of Singapore.

Papers Published

Tolerant Planning and Negotiation in Generating Coordinated Movement Plans in an Automated Factory. *Proc. of the 1st International Conference on Industrial and Engineering Applications of Artificial Intelligence*, 522-529, 1988.

Planning Robust AGV Movements. *Proc. of the European Conference on Artificial Intelligence*, 1988.

On the Consistency Assumption, Monotone Criterion and the Monotone Restriction. *SIGART Newsletter*, **103**, 29-32, 1988.

Declaration

I declare that this dissertation has been composed by myself and describes my own work.

Contents

List of Figures

List of Tables

List of Algorithms

Summary of Notations

Chapter 2

G	a graph
V	set of vertices in a graph
E	set of edges in a graph
P	a path in a graph
M	a matching
SDR	system of distinct representatives
MC	minimal covering
C	a covering
$\oplus$	exclusive or
Tβ	tentative β-columns
Fβ	potential flippable set of β-rows

Chapters 3 and 4

s	start node
t	goal node
$\Gamma_1(n)$	successors of node n in the problem graph
$\Gamma_2(n)$	parents of node n in the problem graph
d	current search direction index; when search is in the forward direction $d=1$, and when in the backward direction, $d=2$
d'	$3-d$; it is the index of the direction opposite to the current search direction
ε	a small positive value
$c_i(m,n)$	positive and finite cost of the direct arc from m to n if $i=1$, or from n to m if $i=2$

$g_i^*(n)$	cost of the optimal path from s to n if $i=1$, or from n to t if $i=2$
$h_i^*(n)$	cost of the optimal path from n to t if $i=1$, or from s to n if $i=2$
$f_i^*(n)$	$g_i^*(n)+h_i^*(n)$; it is the cost of the optimal path from s to t constrained to contain n
$g_i(n)$, $h_i(n)$	estimates of $g_i^*(n)$ and $h_i^*(n)$ respectively
$f_i(n)$	$g_i(n)+h_i(n)$
λ	cost of the optimal path from s to t
L_{min}	cost of the best (least costly) complete path found so far linking s to t
$TREE_1$	forward search tree
$TREE_2$	backward search tree
$OPEN_i$	set of open nodes in $TREE_i$
$\|OPEN_i\|$	number of nodes in $OPEN_i$
$CLOSED_i$	set of closed nodes in $TREE_i$
$p_i(n)$	parent of node n in $TREE_i$
Ω_m	set of nodes in $OPEN_{d'}$ which are descendants of m in $TREE_{d'}$
MeetingNode	node where $TREE_1$ met $TREE_2$ and yielded the best complete path found so far

Chapter 6

s	speed
n	node
ω	waiting time at a node
ω_p	loading time at collection point
ω_z	unloading time at destination
D_{z1}	prescribed arrival time at destination
D_{z2}	prescribed departure time at destination
t_z	planned arrival time at destination
δ	constant value for leeway in varying planned departure time from the starting point
τ	planned tolerance at a node
I	reserved interval
H_1	beginning of reserved interval
H_2	end of reserved interval
T_1	end of front hedge

T_2	beginning of rear hedge
ψ	skill set
R	RToken

Chapter 7

V_x	assisted AGV
V_y, V_z	assisting AGVs
S	starting position
C	collection point
D	delivery destination
N	a node
C', C"	shifted collection points
D'	shifted delivery destination
P1	shortest path from S to C
P2	shortest path from C to D
P1'	shortest path from S to C'
P2'	shortest path from C to D' or from C' to D
P	shortest path from S to D
P'	a collaborative path
T_{Fwd}	forward collaboration tree
T_{Bwd}	backward collaboration tree
T_C	subtree of T_{Fwd} rooted at C
T_D	subtree of T_{Bwd} rooted at D
P_{AC}	assisted collection path/plan
P_{AD}	assisted delivery path/plan

Appendix B

k	node where the fringe of $TREE_1$ meets the fringe of $TREE_2$. Typically, there will be more than one meeting node since fringes change with search and will meet again elsewhere in the search space.
g_{min1}	minimum g_1 value of nodes in $OPEN_1$
g_{min2}	minimum g_2 value of nodes in $OPEN_2$

P_{NIC}	proposition $[(g_{\min1} + g_{\min2}) \geq L_{min}]$ representing Nicholson's terminating condition
P_{BSPA}	proposition $[k \in CLOSED_1 \cap CLOSED_2]$ representing Pohl's terminating condition

Chapter 1

Introduction

Material delivery is a problem which needs to be solved in all factories. Raw materials must be moved into the factory. Work-in-progress items must be moved from one location to another within the factory. Semi-finished or manufactured products must be moved out of the factory. These items can be transported automatically by various means ranging from highly inflexible systems such as conveyor belts, to the most flexible means, the automated guided vehicle (AGV) which is an unmanned truck. Inflexible material delivery systems are more suited to factories which have rather permanent production or transfer lines and which do not require frequent changes, as in the case of manufacturing items in large quantities with long product lifespans. In contrast, AGVs are highly flexible means of transportation which allow greater control over the paths of material delivery. They are better suited to factories employing flexible manufacturing concepts, as in situations characterized by high variety, low volume, demand responsiveness and short product lifespan.

This dissertation examines the problems in automatic planning of AGV movements in a factory. In particular, I shall be considering the problems of task assignment, route planning, traffic coordination, and task collaboration. Since AGV movement planning occurs during execution (*i.e.* some AGVs are already enroute to accomplish their tasks) as well as before, the implications of a dynamic scenario will also be taken into account in examining these problems.

This research on the problem of automatic planning of AGV movements is motivated by the important role of AGVs in modern factories as a vehicle for material delivery as well as for other specialized functions. This motivation will be further explained in the next section. The scope of this research is discussed in section 1.2 and section 1.3 gives a guide for the reader.

1.1 Motivation

The factory of the future will undoubtedly be highly if not fully automated because automation is the key to manufacturing responsiveness, productivity and reliability; these are major determining factors of survivability in a highly competitive trading world. The signs of the automation spur are clearly visible. At the forefront is the use of robots. Also evident is a wave of new manufacturing concepts—computer-integrated manufacturing (CIM), flexible manufacturing systems, flexible assembly systems, manufacturing automation protocol, *etc.* On the eastern front we see the emergence of modern manufacturing practices from Japan: just-in-time or JIT (Burgam, 1984) and Kanban (Gunn, 1982). Many of these concepts are dependent on automation and have enabled significant inventory and cost handling savings. Adding support to this modernization movement, Bradt (1984) states:

> ... we are at the front end of a virtual revolution—not an evolution, but a revolution—in batch manufacturing operations, and an understanding of that revolution is essential for the success and, indeed, the survival of United States manufacturing companies ...

And Critchlow (1985) writes:

> ... it is necessary to use JIT and other techniques—robots, precision tooling, and interfacing transportation systems—to provide higher-quality products, savings through inventory reduction, and the flexibility to fabricate parts as needed. ... there will be an increasing demand for small quantities of a large number of different models of items, ... only through the flexibility of the future factory concept will it be possible to produce them economically and of high quality.

The factory of the future which many envisage as a "lights-out" type of CIM plant with its computers and much of the robots and computer-controlled machinery in communication using a standard specialized protocol, is a concept which has been taken seriously. Governments and corporations are committing huge sums of money in support of it. In an excellent survey article on the factory of the future, Valéry (1987) gives the figures—MITI, Japan: $130 million over the next five years; EEC: $120 million since 1982 and has in the pipeline proposals for CIM research totalling $1.3 billion over the next five years; USA: $50 billion was spent from 1981 to 1986 installing flexible manufacturing tools. There has been some concerted effort on the academic front as well. Only recently, the Institute for Manufacturing Automation Research (IMAR) was established by a consortium of universities and corporations in USA (Schlesinger and Tiersten, 1987). IMAR is a research and development centre for advanced manufacturing technologies with a focus on integration—how to link islands of automation into a CIM system.

AGVs, already in use extensively in modern factories of today, will feature even more prominently in future factories. Between 1960 and 1980, 360 AGV systems (AGVS) with a total of about 3,900 AGVs were in operation in Europe (Müller, 1983). Since AGVs will take on a key role in linking the islands of automation in the future factory (Valéry, 1987), we can expect AGVs to be introduced in greater numbers and with higher levels of sophistication. Apart from automation, new applications engendered by technological progress will also increase the usage of AGVs. For example, AGVs give mobility to a mounted robot and this opens up a vast range of application possibilities. Furthermore, in semiconductor industries, AGVs are suitable for clean rooms where the presence of humans would make it extremely difficult to maintain the very low particulate density essential for the fabrication of VLSI devices with sub-micron feature sizes.

The four main problems which have to be confronted in implementing any AGV system are:

- *Task assignment.* Task assignment concerns the selection of an AGV to undertake a given task. For example, if something at location P needs to be moved somewhere else, the system might simply select the available AGV nearest to P at that time.

- *Route planning.* In route planning, a specific path, usually the shortest trafficable path, is determined for an AGV to move from its starting position to where its load has to be fetched, and from there to the destination. The shortest path can be found using one of several algorithms such as the A* (Hart *et al*, 1968, 1972) and Dijkstra's (Aho *et al*, 1974) algorithms. Whichever is used, it is necessary to have a map representing the route network in some appropriate data structure (*e.g.* a graph) within the computer.

- *Navigation.* Given a route to traverse, the AGV must be able to navigate its way along it. There are several systems of navigation which can be used. Floor-bound AGVs are restricted to following a network of cables or painted lines in/on the floor of the factory. If cables are used, the AGVs are guided inductively; and if painted strips are used, then an optical guidance system is used. For further information, the reader can refer to Müller (1983) and Todd (1986). Free-ranging AGVs unlike their floor-bound cousins have their geographical limits restricted by trafficability alone. These face a more difficult navigation problem and thus rely on more sophisticated systems with some incorporating a multiplicity of means. Typical approaches are: use of beacons (active and passive) for position fixing by triangulation; dead-reckoning using an odometric system; path identification or route following using an optical TV camera (McTamaney, 1987; Tsuji and Jiang, 1987); and sonar-based systems.

- *Traffic control.* Traffic control coordinates the movements of AGVs such that they do not collide into one another. The most common method of coordination is the blocking method (Todd, 1986). In this method, not more than one AGV is allowed in a block (path segment in the network) at any time. This guarantees no collision. Alternatively, or as a supplementary means, contact-sensitive bumpers are installed at both ends of AGVs. When an impending collision is detected, the AGV is abruptly stopped and it remains stationary until reactivated or when the cause of activation is no longer sensed. This also serves as a safety measure to avoid serious mishaps involving humans.

The problem with conventional AGV systems is that they do not fully meet the needs of the automated factory. For example, task assignment and execution may not be optimized from a global perspective. Sub-optimal plans do not utilize AGVs as efficiently as they should. Consequently, a larger fleet of AGVs may be necessary. More serious is the *ad hoc* nature of traffic control; this does not admit temporal projection of when an AGV is expected to complete its delivery. Although simulation may be used to forecast future events, the results can be highly misleading. For instance, if two AGVs approach the same block at the same time according to the simulation, there is no way to tell which will actually be admitted into the block. In reality, one of the AGVs will gain control of the block just before the other, but this cannot be predicted due to limitations of simulation accuracy.

Factory automation requires planning and integration. Plans of subsystems which are expected to work in concert must be dovetailed in order to satisfy various dependency constraints. Hence material delivery plans should tie in with the job shop schedules. A hierarchy of related plans would exist and all are designed to meet production objectives which are defined according to market demands and forecasts. Many of these plans have time constraints, and based on expected deadlines of some preceding operations, further operations can be planned. This means that temporal projection is necessary. It would be near impossible to plan further time-dependent operations in advance if there is no information on when prerequisite operations will complete. Even if possible with whatever scant information is available, the predictions of when future plans will complete can only be made with large margins of uncertainty. Such plans would make poor use of production resources. AGV movement plans are one such type of plans in the hierarchy of plans which have deadline constraints and need to be planned in a manner which admits temporal projection.

1.2 The Scope

This dissertation examines the problems in automatic generation of AGV movements in an automated factory. Specifically, I shall consider the problems of task assignment, route planning, traffic coordination and the construction of collaborative plans. The problem of navigation will not be addressed since this is a major research issue in its own right. Hence, it is assumed that AGVs have some means of navigating from one position to another.

1.2.1 Planning in a Dynamic Context

The approach to solving these problems will be largely influenced by the fact that the factory scenario is dynamic. Changes and new requirements arise as plans are generated and executed. Changes are likely to occur since certain aspects of the real world were inadequately considered or not considered at all in the model used by the planner. Consequently, problems arise when the situation in the real world does not match the expected outcomes in the plan. When this occurs, replanning is necessary to salvage a failing plan. Such replanning situations generate new planning goals. New requirements (*e.g.* from new orders) may also arise during execution. Rather than defer consideration of these new requirements until the current set of plans have run their course and then repeat the plan-and-execute cycle afresh, it may be advantageous to exploit current circumstances and attempt concurrent execution of the new plans.

The implication of planning within a dynamic scenario is that plans must be computed in good time to meet deadlines. Expeditious planning is thus necessary and this calls for the use of fast algorithms, especially for problems which have to be repeatedly solved during planning. Also, since dynamic planning needs to be prompt, it helps if its preoccupation is with new tasks rather than with salvaging failing plans. An approach to meeting this objective is to produce robust plans which tolerate minor execution deviations. Tolerance of execution errors allow replanning to be deferred and reduces the frequency of disrupting the dynamic planner. A larger proportion of the processor's time is then available for planning new tasks. Hence plans for new tasks can be generated more rapidly.

A second factor to consider in dynamic replanning within a multi-agent environment is the effect of plan revision of one AGV on the plans of other AGVs. The modifications of an AGV's plan may impinge on the plans of other AGVs, and the affected AGVs' accommodations may in turn affect other AGVs, and so on. A need thus arises to prevent or at least reduce the likelihood of a chain reaction of plan revisions. Robust plans which reduces the need for dynamic replanning also help in this respect.

1.2.2 Efficient Employment of AGVs

Efficient usage of AGVs means that the effort required to achieve the given tasks should be economical. This is desirable for several reasons. First, less energy is expended if a task is efficiently executed. Since AGVs are battery-powered and recharging is a time-consuming process, it is desirable to have more tasks executed by the AGVs before they turn in for long unproductive periods to recharge their batteries.[†] Second, AGVs can be made available for their next tasks earlier, and this faster turnaround of AGVs implies a smaller AGV cohort than required otherwise.

The effort involved in execution is tied closely to the distance which has to be travelled. Hence the approaches I adopt in aiming for efficient usage of AGVs are centred on distance minimization or reduction. The key question is: how can the overall distance of travel be minimized? Or if minimization is too costly or not possible because of mathematical intractability, what can be done to reduce it as far as possible in a practical way?

Economy of effort will be achieved in three ways:

- *Optimal task assignment.* The assignment of tasks will be optimized from a global perspective. This means that for a given set of tasks, the AGVs will be assigned the tasks individually, one per task, such that the total of the distances travelled to fetch and deliver the loads is minimized. I will propose a fast and novel implementation of the Hungarian algorithm which was originally described for manual application.

[†] A remedy to unproductive recharging spells is to replace the weak batteries instead of recharging them. This requires human intervention or some mechanical means to replace the batteries automatically. As far as I know, recharging and not replacement is the method commonly used.

- *Optimal route planning*. The planner will always select the shortest trafficable routes. Although several algorithms are available to solve the shortest path problem, I shall propose a bidirectional heuristically guided search algorithm which is amenable to parallel implementation. This allows concurrent development of the two search trees and may yield a search time reduction of up to half the time it takes on a uniprocessor. Further time reduction will also be achieved by exploiting routes learned previously. The speed-up from route learning comes from: (1) instant availability of previous solutions which matches the current problem exactly; and (2) use of learnt partial solutions to reduce the search space and hasten termination.

- *Collaborative planning*. The second approach exploits the possibility of task sharing. If one AGV can solve two tasks using the same route as it would take if it set out to solve only one of the tasks, why use two AGVs? Collaborative planning is the problem of generating plans in which AGVs assist each other in task execution when the opportunity arises. Collaborative plans exploit existing movement plans in planning for new tasks such that economy of effort accrues. I shall be proposing efficient algorithms which search for collaborative opportunities from among the set of currently tasked AGVs.

1.2.3 Movement Planning

Movement planning involves generating a set of AGV movement schedules which define the paths and the movement timings of the AGVs. The plans or schedules generated must be safe *i.e.* conflict-free. Temporal projection is straightforward if plans are generated prior to execution. All that is needed for temporal projection is an interpolation of timings specified at waypoints along the route. Often this is not necessary since the information required—when a load collection will occur and when a task can be expected to complete—can be looked up from the schedule of the movement plan.

Besides temporal projection, how can movement planning produce robust plans which will continue to be viable despite minor execution deviations? The approach taken is to generate interval-based schedules instead of the usual point-based schedules. This means that AGVs reserve intervals of time for its presence at various locations along its route. During execution, it will plan to arrive and depart well within the reserved intervals so that adequate leeway for deviations exists. If the actual arrival/departure time differs from the planned time, it is nevertheless possible to arrive/depart within the next downstream interval provided the deviant arrival/departure time remains within the current interval. Hence replanning is not immediately warranted. By not incurring dynamic replanning as often as would be the case if plans are less tolerant of execution deviations, the dynamic planner can be free to devote its time to what really needs its attention—planning for new tasks. This improves the likelihood of generating plans in good time.

Interval-based planning means redundant allocation of resources. In the case of AGV movement planning, the resource is time at a place. Redundant allocation aggravates the conflict resolution problem—resource contention becomes more frequent. Two time intervals which are not far apart in time are more likely to overlap than two time instances. If two intervals overlap, then a potential conflict exists since no more than one AGV can be at the same place at the same time. Resolving such conflicts necessarily entails some compromise. The overlap between the intervals must be eliminated either by shrinking or shifting the intervals. Shrinking reduces plan robustness and shifting moves the planned arrival/departure time away from its initial ideal. The compromises should be made in an equitable manner. If one AGV has to compromise much more than another, then its plan would be relatively brittle and it is more likely to invoke dynamic replanning, possibly leading to a chain of plan revisions. An auxiliary issue in the design of the AGV movement planner is thus to design a conflict resolution scheme which achieves fairness. The iterative negotiation model will be proposed for this purpose. In the model, AGVs are modelled as intelligent agents with negotiation skills and they haggle over disputed resources until the conflict is resolved or in the event of a negotiation impasse, a plan failure is reported and a new plan (*e.g.* using a different route) will be sought.

1.3 Reader's Guide

This section describes the structure of the dissertation and gives a guide for the reader. The chapters are presented in a bottom-up order. Hence the modules of the planner are described first and then the planner itself.

Figure 1-1 gives the reader a guide at a glance. It describes briefly the contents of the chapters and the orders in which chapters may be read. A more detailed description of the contents of the main chapters follows:

- *Chapter 2*. Assigning tasks to AGVs is the first problem to be solved in planning AGV movements. In distributed AI (DAI), this is similar to the task distribution problem. The usual approach to this problem in DAI is based on a greedy algorithm: assigning the tasks one at a time to the agent which can best undertake each task. Although fast and simple, this approach leads to suboptimal assignments. A better alternative is to use polynomially time-bounded optimal assignment algorithms from combinatorial mathematics or graph theory. This chapter describes a novel implementation of one such algorithm based on the Hungarian method. I will show how the techniques of problem transformation, reduction and decomposition can be applied, resulting in improved running time by as much as 60% and yet without incurring extra memory costs. The algorithm meets the planner's requirement of fast computation and efficiency.

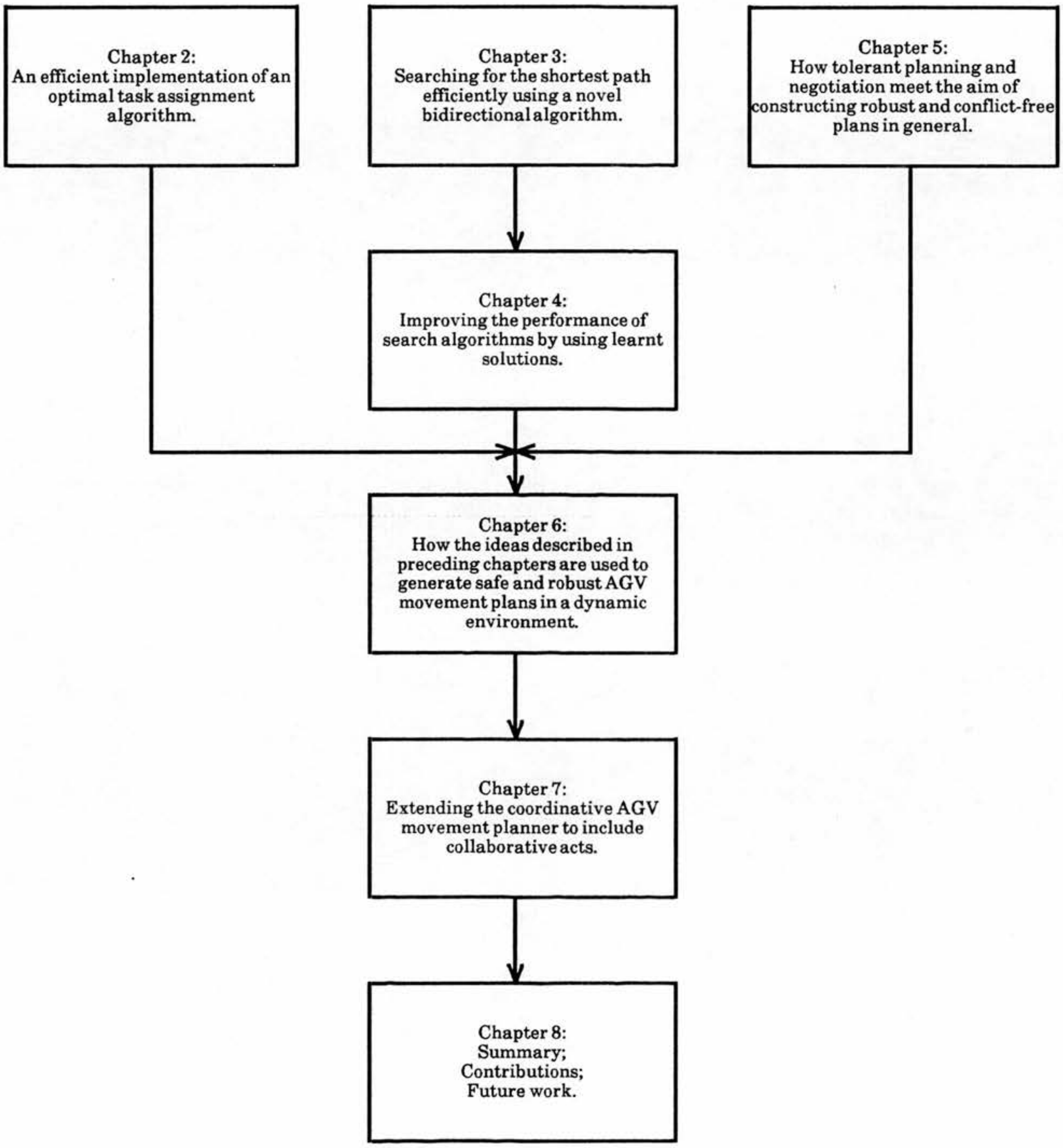

Chapter 2:
An efficient implementation of an optimal task assignment algorithm.
Chapter 3:
Searching for the shortest path efficiently using a novel bidirectional algorithm.
Chapter 5:
How tolerant planning and negotiation meet the aim of constructing robust and conflict-free plans in general.
Chapter 4:
Improving the performance of search algorithms by using learnt solutions.
Chapter 6:
How the ideas described in preceding chapters are used to generate safe and robust AGV movement plans in a dynamic environment.
Chapter 7:
Extending the coordinative AGV movement planner to include collaborative acts.
Chapter 8:
Summary;
Contributions;
Future work.

Figure 1-1. A quick guide for the reader.

- *Chapter 3*. Following task assignment, the planner must compute the best routes to be traversed by the AGVs to achieve their tasks. The main objectives here are to minimize the time and energy required to achieve the tasks. It is assumed that shortest paths meet these objectives. This chapter begins by surveying various shortest path algorithms, discussing their relative merits and demerits. A case is then established for bidirectional heuristic search which has the potential of computing the shortest path fastest. A novel algorithm BS* is then described. BS* is by far the most efficient bidirectional admissible search algorithm. It is also the first search algorithm which prunes the search tree without sacrificing admissibility. Empirical results will be presented to show that it performs better than the previous best algorithm in the same class by about 30% in both time and space. When implemented in a dual processor machine it can be expected to return solutions in half the time A* takes and with only nominal penalty in memory utilization.

- *Chapter 4*. Route planning can be made more efficient by exploiting previously computed shortest paths. In order for exploitation to be possible, known shortest routes must be remembered. Rather than record only the shortest route just computed, all shortest paths embedded in the search trees should be extracted and stored. This chapter describes efficient extraction algorithms and show how learnt information can be used to achieve earlier termination in subsequent searches.

- *Chapter 5*. All planning systems make use of representations (models) of some sort. For pragmatic reasons, the models are incomplete and simplified, and thus not wholly accurate. Consequently, the outcomes during execution may not correspond exactly to prior expectations. Such execution deviations may invalidate the remainder of the plan. The conventional solution is to monitor the execution and perform run-time replanning if necessary. These actions are not only costly but may require a more complicated planning system. In some instances, replanning may not be achieved in time to salvage a failing plan. Tolerant planning is a way of making plans more tolerant of execution deviations, not only of the executing agent itself, but also of other agents in the same environment. This has two important advantages: monitoring may be relaxed and less dynamic replanning may be invoked. However, tolerant planning aggravates the resource contention problem. To counter this, the iterative negotiation model is proposed. It seeks to resolve conflicts fairly by mimicking negotiation acts in the real-world. Its main merit is conceptual and representational simplicity—it does not require agents to model any aspects of other agents, and thus escapes the logical intricacies of such an approach.

- *Chapter 6*. This chapter describes how the AGV movement planner is implemented based on the ideas discussed in preceding chapters. It argues a case for interval-based plans and then describes: (1) how the concept of tolerant planning can be applied to generate robust movement schedules; and (2) how the traffic coordination problem can be solved using the iterative negotiation model.

- *Chapter 7*. The efficient utilization of AGVs can be further improved by using opportunities for currently tasked AGVs to assist in the execution of new tasks. This requires opportunities to be first identified and then realized in the new set of plans. The chapter shows how a subset of these opportunities can be identified by searching a collaboration graph. Patching in a collaborative plan can then be achieved using the same techniques of tolerant planning and negotiation.

Chapter 2

Task Assignment

2.1. Introduction

The task assignment problem is to determine how best to match a set of given tasks to the available AGVs such that each AGV is assigned not more than one task. There are two approaches to solving the task assignment problem in some optimal sense—local and global optimization. This chapter proposes the global optimization approach since it enables the tasks to be achieved with greater economy of effort. The main emphasis of the chapter is on a novel implementation of the Hungarian method to achieve a globally optimal assignment of tasks. I will show how some problem-solving principles—problem transformation, reduction and decomposition (Nilsson, 1980)—can be applied to solve the optimal assignment problem more efficiently than the conventional approach.

Section 2.2 points out the importance of economy of effort in accomplishing tasks in this specific AGV application. Section 2.3 describes a local optimization approach used by Davies and Smith (1983) in their contract net metaphor. Whereas this is a greedy algorithm which seeks to optimize for one task at a time, the other approach is to seek a global optimization (section 2.4) over the set of tasks. The latter can be achieved using the Hungarian method (Kuhn, 1955; Anderson, 1974) described in section 2.4.2, or the network flow method (Carré, 1979; Ford and Fulkerson, 1962). The network flow method will not be described since it is similar in many ways to the Hungarian method and both have similar worst-case orders of complexity. Section 2.4.3 describes briefly the Kuhn-Munkres algorithm (Munkres, 1957) which is the conventional implementation of the Hungarian method. In section 2.4.4, I will describe a novel implementation of the Hungarian method. A comparison is then made along with experimental results in section 2.4.5 to show the superiority of the new implementation over the Kuhn-Munkres algorithm.

2.2 Motivation

In every feasible assignment, a clear objective for the assignment procedure is to minimize the distance which must be traversed to achieve the given tasks. This is important because distance determines largely how much time an AGV must commit to a task. The shorter the distance, the less time it needs to commit and hence can be available earlier for other tasks. By this, the overall utilization of AGVs is made more efficient.

Another reason why shorter distances are desirable is that one can expect less occurrences of movement conflicts with other AGVs than if longer distances are involved. Less conflict means less conflict resolution work for the planner. More importantly, since conflict resolution results in other AGVs compromising their preferred movement timings, shorter routes help AGVs to stick as close as possible to their initial ideal plans. The impact of route length on conflict resolution will become clearer in chapter 6 where conflicts between AGVs' movement plans are addressed.

Efficiency is also improved in another sense. Energy is expended whenever an AGV moves. Moving short distances helps conserve energy. With some AGVs having gross weights of a ton or more it is certainly a plus to conserve energy. Energy conservation also allows an AGV to undertake more tasks before it has to turn in to recharge its batteries.

2.3 The Local Optimization Approach

The local optimization approach seeks to find the best way to assign one task at a time and hence belongs to the class of greedy algorithms. The chief advantage of greedy algorithms is speed, but at the expense of a globally optimal solution. The local optimization algorithm (LOA) can be outlined as:

Algorithm 2-1:

```
        procedure LocalOpt(TaskList, Agents)
        /* Assign the tasks to the candidates using a local optimization criterion.
           */
1.      foreach x in TaskList do
2.          Get the list of candidates from Agents capable of undertaking x.
3.          Assign x to the candidate in the list which is best able to perform x.
        endforeach
        endprocedure
```

A simple way to implement the LOA in a distributed AI context is to adopt the announce-bid-award sequence which Davies and Smith (1983) employ in the contract net metaphor for task distribution. In this metaphor, task distribution begins when an agent, known as the *manager*, has a task which it wants to delegate to one or more agents known as the *contractors*. The manager will typically decompose the task into several subtasks and assign these to the contractors. If task decomposition is unnecessary, it only needs to assign the task to one contractor. Any contractor assigned a task or subtask may in turn repeat the task distribution process by assuming the role of a manager and seeking subcontractors.

The assignment of a task begins with the manager broadcasting an announcement message. This step is analogous to a tender invitation in the business world. A modification of it could be a kind of selective invitation wherein the tender is open to some but not all of the possible contractors. Agents in receipt of the announcement message evaluate their ability to undertake the task and, if possible and desirable, will make a bid for the task by responding with a bid message to the manager. Bids have an associated "cost" tag to indicate how much it would cost the manager to commission that contractor for the task/subtask. When all the bids have been received or when the bidding deadline has expired, the manager evaluates the bids according to some criterion and awards the task/subtask to the best or least costly bidder. The assignment process is completed by an award message sent to the successful bidder.

Greedy algorithms are often employed when the only known algorithms to produce a globally optimal solution are of exponential complexity. For example, the travelling salesman problem and others in the NP-complete class are sometimes solved by greedy algorithms (Aho *et al*, 1983). However, in assignment problems, algorithms to produce globally optimal solutions exist which are polynomially bounded in time and space. Under what circumstances then is an LOA justified? Possible justifications are:

- When it is not possible to implement a globally optimal algorithm (GOA). For example, if memory limited dedicated microprocessor systems are used, it may not be possible to accommodate the more space-consuming GOA.

- When tasks arrive at different times and the assignment of a task cannot be held back till enough tasks have been accumulated for a more global approach. However, this alone is no justification as we shall see.
- The economy of a globally optimal solution is insignificant.
- The time to compute a globally optimal solution is nevertheless excessive.

In the AGV scenario, we can disregard the first point—inadequate memory space—since task assignment should be solved by the central computer. Such a computer can be safely assumed to have sufficient memory to meet the space requirements of a GOA. The second consideration—urgency of tasks—is apparently valid since tasks may arrive in batches as well as singly. But when only one task is to be assigned, a GOA is also applicable: in this case, LOA and GOA return the same result.

AGVs

		1	2	3	4	5
	1	9	9	11	17	19
	2	8	9	18	12	10
Tasks	3	7	19	11	18	14
	4	9	11	13	12	5
	5	14	15	9	16	14

LOA (worst case when task assignment order is (2, 3, 5, 4, 1)):-
Solution: {(1,4) (2, 1) (3,3) (4,2) (5,5)}
Total cost: 61
Computation time: 0.01 sec.

GOA:-
Solution: {(1,2) (2,4) (3,1) (4,5) (5,3)}
Total cost: 42
Computation time: 0.2 sec.

Solution element (i,j) refers to that in the ith row (task) and jth col. (AGV).
Value of each element is the cost of the corresponding task-AGV pair.

Figure 2-1. Comparison of LOA and GOA solutions.

The last two points—insignificant gain and excessive computational time—can be refuted by way of an example. Figure 2-1 compares the solution quality and the time it takes to assign five tasks. The matrix shows the costs of various AGV-task pairings. In applying the LOA, unless task priorities are given, tasks are arbitrarily assigned sequentially. In the example, the worst case assignment sequence is used. The LOA's assignment solution has an overall cost of 61 units (e.g. metres). A GOA based on the author's implementation of the Hungarian method has an overall cost of 42 units and required only 0.2 second to compute. If the problem was scaled up to 10 tasks, the GOA would typically require less than a second. Clearly, the time factor is insignificant, but the cost savings can be significant and more so for larger task sets.

Although global optimization is computationally more expensive than local optimization, the overheads can be easily afforded if implemented by a central computer. Noting that for any assignment problem solved by an LOA, a GOA can solve it at least as well, it is clear that the task assignment problem should be solved to meet the global optimization objective.

2.4 The Global Optimization Approach

The global optimization approach seeks to solve the assignment problem such that the total cost of the solution is minimized. This can be achieved without recourse to the naive way of first enumerating all the possible solutions and then selecting the best. Efficient ways of solving the optimal assignment problem are based on the Hungarian method with its origin in combinatorial mathematics, and the network flow method (Carré, 1979; Ford and Fulkerson, 1962) from graph theory.

2.4.1 Problem Representations

In general, the number of available AGVs differs from the number of tasks to be assigned. The possible pairings of AGVs to tasks can be represented with a bipartite graph (see Figure 2-2). In the graph, one set of nodes represents the available AGVs and the remaining nodes form the other disjoint set representing the tasks to be assigned. An arc linking an AGV node to a task node indicates that the AGV is capable of

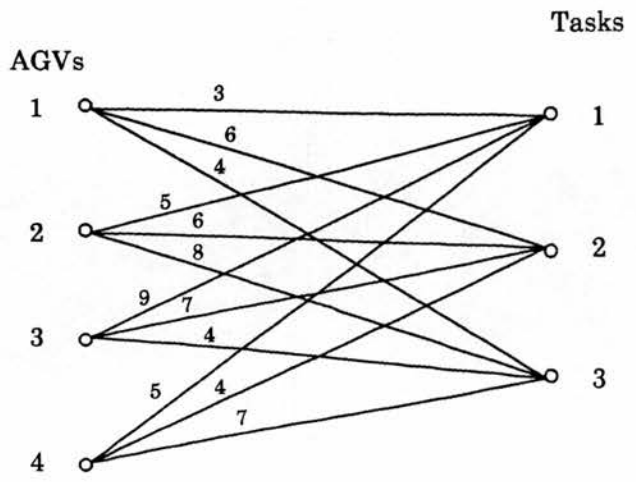


Figure 2-2. A bipartite graph showing AGV-task pairings.

		AGVs 1	2	3	4
Tasks	1	3	5	9	5
	2	6	6	7	4
	3	4	8	4	7

Figure 2-3. Matrix representation of bipartite graph

undertaking the task. Arcs are also labelled with associated costs. Whereas the bipartite graph representation is suited to the network flow method, we shall use a matrix data structure (see Figure 2-3) to represent the bipartite graph in the Hungarian method. In general, there will be some impossible AGV-task pairings. These impossible pairings can be represented in the matrix with a large number much greater than any real arc cost.

2.4.2 The Hungarian Method

The foundations of the Hungarian method can be found in the works of König (1931), Egerváry (1931) and Hall (1935). Interestingly, these early researchers did not address directly the problem of optimal assignment but established theorems in pure mathematics which eventually led to the discovery of efficient algorithms of practical importance. In particular, the first efficient algorithm to solve the optimal assignment problem is credited to Kuhn (1955) who named his algorithm the Hungarian method in honour of König and Egerváry, two Hungarian mathematicians who jointly developed a theorem which played a vital role. Before I describe the Hungarian method, some definitions are needed for clarity and conciseness of elaboration.

2.4.2.1 Preliminary Definitions

G is a *finite graph* $<V,E>$ where V is a finite set of *vertices* and E is a set of *edges* $=\{e_{ij} \mid e_{ij}=(v_i,v_j), v_i, v_j \in V\}$.

G is a *bipartite* graph if V can be partitioned into two disjoint subsets X and Y (i.e. $X \cup Y = V$, $X \cap Y = \emptyset$) and none of the edges in E are such that both vertices of the edge are in X or both in Y.

A *path* P is a sequence of edges.

P is a *simple path* if no vertex occurs more than once in it.

M is a *matching* if $M \subseteq E$ and all vertices in V are incident with at most one edge in M.

The *matching cost* of M is the total cost of the edges in M.

$|M|$ denotes the the number of edges in M i.e. its cardinality.

M is a *maximal matching* if there is not another matching M' such that $|M'|>|M|$.

M is an *optimal assignment* if G is bipartite and M is a maximal matching such that there is not another maximal matching with a lower matching cost.

A *system of distinct representatives* (SDR) in relation to a family of sets $S_1, S_2, ..., S_n$, is a set of elements such that all elements are dissimilar and each element is in at least one set S_i, $i \in [1,n]$. Alternatively, for an nxn matrix, the SDR is any n elements such that no two are in the same row or column. Such elements are also called *independent* elements.

A *line* in a matrix is either a row or a column.

A *minimal covering* (MC) in relation to a matrix with some 0 elements is the smallest set of lines which crosses out all the 0s.

2.4.2.2 Description of the Hungarian Method

In this section, the description of the Hungarian method is based on Anderson's (1974) elegant account of it. The method makes two assumptions: (1) the cost matrix is square; and (2) all elements have positive non-zero cost values. The reason for the matrix being square and a way to deal with rectangular matrices will be explained shortly. The reason for the second assumption will become obvious in the following paragraphs. The second assumption can be easily satisfied—simply increase every element by the same constant value (say k) which makes all elements non-negative. This is allowed because the relative costs of elements are unaltered and hence it does not affect the optimal solution. Its only side-effect is to increase the total cost of the optimal assignment by nk where n is the cardinality of the maximal matching. Knowing this, determining the actual total cost is a trivial step.

Given a nxn cost matrix, a feasible solution to the optimal assignment problem must satisfy two constraints. First, it must be a system of distinct representatives (SDR). In the matrix representation, an SDR is equivalent to a maximal matching M. Hence we may use the terms SDR and maximal matching interchangeably. Viewed another way, the SDR or maximal matching problem is similar to the problem of placing n-rooks on a nxn board such that no rook attacks another. The second constraint is that the matching cost of the SDR representing the solution is not greater than that of any other SDR. (Where the context clearly refers to matching cost, the shorter term "cost" will be used.)

A naive approach to the optimal assignment problem is to exhaustively enumerate the n! different SDRs and select that with the least cost. Fortunately, there are some neat tricks to overcome this combinatorially explosive situation. These tricks become obvious when certain observations are made.

The first observation is that the constraint of no two distinct representatives being on the same line implies that if every element on a line is changed by x, then every SDR's cost will also be changed by x. This relationship holds for both incremental and decremental changes. Since the cost of every SDR is changed by the same amount, a least costly SDR before the change was made retains its least costly status after the change. This *invariant* line operation can be used to increase the number of 0 elements in the matrix.† The way to do this is to take a line with elements which are all greater than zero, and subtract the minimum value in the line from every element in it. This will produce at least one more 0. Applied repeatedly, sufficient 0s will appear such that we can pick n distinct 0 representatives. The selected 0s are also called *independent* 0s. What has happened is that the optimal assignment problem has been reduced to a simpler maximal matching problem which is to find n independent 0s. Since all elements are non-negative, such an SDR having a zero cost is a least costly SDR and thus qualifies as a solution. The actual cost of the optimal assignment is the sum of the original costs of these independent 0s.

The reason why the Hungarian method only works with square matrices should be evident now: if the matrix is not square, the above line operations do not possess the invariant property. In other words, some line operations may alter the cost of an SDR more/less than the other SDRs. Nevertheless, this limitation does not pose a problem for

† In this chapter, "0" refers to an *element* in the matrix with zero value whereas "zero" is used in the numerical sense.

rectangular matrices since these can be augmented with rows/columns to make them square. In doing so, all the elements in the extra rows/columns must have the same value which is greater than any cost value in the original matrix. Obviously, the optimal assignment will contain extra elements from these fictitious rows/columns. Omitting these extraneous elements gives the real optimal assignment. Although this circumvention incurs extra computational effort, the Hungarian method is so fast that the wastage is insignificant.[†]

The Hungarian method has two main phases. Recapitulating, phase I is to produce sufficient 0s to transform the optimal assignment problem into a maximal matching problem, which is then solved in phase II. Phase I will be illustrated by two examples to show how two different cases should be dealt with.

Phase I begins by applying the line reduction operation to every row and column which does not already contain a 0.

In the first example, the initial cost matrix as shown in Figure 2-4a is transformed by row reduction operations to that shown in Figure 2-4b. Since all elements are initially non-zero, all five rows were reduced. Examining the columns next reveals that only the second and last columns are free of 0s. Reducing these columns gives the final result shown in Figure 2-4c. By inspection, we see that it has a set of 0s from which an SDR (indicated by the bold 0s) can be chosen.

In the second example a different initial cost matrix is used (Figure 2-5a). Following the initial row reductions (Figure 2-5b), only the second column should be reduced. When all applicable lines have been reduced (Figure 2-5c), we find that there are still insufficient 0s to obtain an SDR.

[†] It takes on the average, about 13 seconds to work on a 50x50 matrix using the NewHungarian procedure (section 2.4.4.9) implemented in Interlisp-D in a Xerox 1186 workstation.

		AGVs				
		1	2	3	4	5
	1	12	19	7	20	11
	2	11	10	5	13	10
Tasks	3	6	14	10	6	20
	4	9	15	11	11	18
	5	18	8	17	7	12

Figure 2-4a. Initial cost matrix.

5	12	0	13	4
6	5	0	8	5
0	8	4	0	14
0	6	2	2	9
11	1	10	0	5

Figure 2-4b. After row reductions.

5	11	0	13	**0**
6	4	**0**	8	1
0	7	4	**0**	10
0	5	2	2	5
11	**0**	10	0	1

Figure 2-4c. After row and column reductions.

		AGVs				
		1	2	3	4	5
	1	12	19	7	20	11
	2	11	10	5	13	10
Tasks	3	6	14	10	6	20
	4	9	10	9	11	18
	5	19	11	18	11	8

Figure 2-5a. Initial cost matrix.

5	12	0	13	4
6	5	0	8	5
0	8	4	0	14
0	1	0	2	9
11	3	10	3	0

Figure 2-5b. After row reductions.

5	11	0	13	4
6	4	0	8	5
0	7	4	0	14
0	0	0	2	9
11	2	10	3	0

Figure 2-5c. After row and column reductions.

1	7	**0**	9	0
2	**0**	0	4	1
0	7	8	**0**	14
0	0	4	2	9
11	2	14	3	**0**

Figure 2-5d. After matrix modification.

Obviously, more 0s must somehow be introduced. Having exhausted the line reduction technique, a different procedure must be attempted. Such a procedure must also possess the invariant property *i.e.* applying it does not change the status of the least costly SDRs inherent in the initial matrix. This procedure shall be called the *matrix modification procedure*. Its first step is to derive a minimal covering (MC) of the matrix because the MC is related to the maximum number of independent 0s which one can select from the matrix. This fact comes from the König-Egerváry max-min theorem which states:

Theorem 2-1:

> In a m×n matrix K, the maximum number of independent 0s which can be selected from K is equal to the minimum number of lines (rows or columns) which together cover all the 0s.

For a proof of this theorem, the reader is referred to Anderson (1974). The theorem gives us a decision procedure for the question of when a matrix is *saturated* with sufficient 0s to produce an SDR, or when the matrix has been sufficiently reduced to the simpler maximal matching problem. Note that it does not tell us how to compute an MC in the first instant. It assumes that one can find an MC by inspection. We shall return to the minimal covering problem in section 2.4.4.1.

If an MC is found to be inadequate, then it can be used to generate more 0s via the matrix modification procedure. This procedure has three main steps:

Algorithm 2-2:

```
        procedure ModifyMatrix(K)
        /* Generate more 0s in the matrix K. Assumes that the MC is accessible
           from a global variable. */
    1.  Get the smallest number, s, not in the MC.
    2.  Subtract s from all the uncrossed columns.
    3.  Add s to all the crossed rows.
```

```
endprocedure
```

In step 1, s will be greater than 0 since all values in K are non-negative and all 0s are within the MC. Step 2 will introduce negative values at affected 0s. Observing that these 0s lie on crossed rows, we can return the negative values to 0 by applying a row increment operation to these rows. This is what step 3 does. Alternatively, if in step 2, the uncrossed rows are decremented by s, then in the step 3 the crossed columns should be incremented by s.

The matrix modification procedure also possesses the invariant property because steps 2 and 3 which modify the matrix are invariant line operations. The effects of the procedure are:

- All *uncrossed* elements are decremented by s.
- All elements *crossed* out *once* in the MC are unchanged.
- All elements *crossed* out *twice* in the MC are incremented by s.

Progress has been made in generating more promising 0s in the matrix since at least one new 0 has been introduced somewhere outside the previous MC. Notice that 0s which were doubly crossed have been lost. This is not a regressive event because such 0s are redundant—removing them does not change the MC required to contain the remaining singly crossed 0s. The new 0s now allow the possibility of increasing the size of the MC, which brings it a step closer to the final solution.

Returning to the second example, an MC for the matrix after row and column reductions is shown by the dashed lines in Figure 2-5c. The minimum value of the elements outside the MC is 4. Applying steps 2 and 3, we obtain the resultant matrix (see Figure 2-5d) which now has enough well disposed 0s to yield an SDR as shown by the bold 0s. We shall refer to any MC which covers an SDR as a *complete* MC.

We can now outline the Hungarian method more formally as follows:

Algorithm 2-3:

```
        procedure Hungarian(K)
        /* Given a nxn matrix K, return an optimal assignment. */
        vars N, C; /* local variables */
    1.  Let N be the number of pairings in the optimal assignment.
    2.  Reduce all applicable rows and columns.
        /* Phase I. */
    3.  Obtain a minimal covering C.
    4.  if |C| < N then
    5.      ModifyMatrix(K)
    6.      Go to step 3.
    7.    else Compute an SDR. /* Phase II. */
        endif
    8.  Return the SDR
        endprocedure
```

Without defining the computational procedures for steps 3 and 7, one can only apply the Hungarian method manually with inspection. Even so, it would be difficult to inspect large matrices. In fact, the conventional implementation of the Hungarian method does not follow exactly the steps just outlined.

2.4.3 Conventional Implementation of the Hungarian Method

First, a few more definitions are needed. These are relative to a matching M.

A *matched vertex* is one which is in some edge of M; otherwise it is an *unmatched* or *free vertex*.

A *matched edge* is an edge in M; otherwise it is an *unmatched edge*.

P is a *simple path* if no vertex occurs more than once in it.

P is an *alternating path* if it is simple and for every pair of consecutive edges in P, only one of them is matched.

P is an *augmenting path* if it is an alternating path and its first and last edges are unmatched.

The *augmented matching* $M' = M \oplus P$ where $\oplus$ is the symmetric difference operator. $\oplus$ is also the exclusive or operator.

The conventional implementation of the Hungarian method is attributed jointly to Kuhn (1955) and Munkres (1957). Originally developed to solve the optimal assignment problem for bipartite graphs, it was later extended by Edmonds (1965) to cope with general graphs. The following outline of the bipartite graph version is a variation of the Hungarian theme:

Algorithm 2-4:

```
     procedure Kuhn-Munkres(K)
     /*  Given a nxn matrix K, compute an optimal assignment. */
     vars N, M, P; /* local variables. */
1.   Let N be the number of pairings in the optimal assignment.
2.   Reduce all applicable rows and columns in K.
3.   Obtain arbitrarily an initial matching M.
4.   if |M| = N
5.           then return M as the solution; exit
     endif
6.   Find an augmenting path P relative to M.
```

```
7.  if P is found
8.       then M ← M⊕P
9.            Go to step 4.
         else /* the process of finding an augmenting path also enables an
                 MC to be identified. */
10.           Apply the matrix modification procedure using the MC found.
11.           Go to step 6.
    endif
    endprocedure
```

Whereas the original Hungarian theme emphasizes the development of a complete MC which specializes the original problem to a maximal matching instance, the Kuhn-Munkres procedure emphasizes the use of augmenting paths to boost the matching size.

The procedure maintains a matching at every iteration of the main loop (steps 4 to 11). At each iteration, an augmenting path is sought. If found, the matching is augmented and the termination test is applied. Otherwise, the matrix modification procedure is applied to generate more promising 0s. (It happens that the labelling procedure (Ford and Fulkerson, 1962) for identifying augmenting paths also yields an MC if no augmenting path can be found. This is one advantage of the conventional approach; it does not require a separate procedure for finding an MC.) Following this, the search for an augmenting path is resumed afresh.

The complexity of the Kuhn-Munkres procedure is $O(n^2v^2)$ (Bondy and Murty, 1976, pg. 90).† The more general algorithm of Edmonds has a complexity of $O(v^4)$. Gabow (1973) gives a $O(v^3)$ algorithm applicable to general graphs. It is also based on

† In this chapter, complexity terms such as O(nv) refer to time complexity, with n and v denoting the cardinality of the maximal matching and the number of vertices in the graph respectively.

augmenting paths. However, Gabow's algorithm does not guarantee that the solution will be of maximal cardinality although its overall cost is optimal. This is a serious limitation for the AGV movement planner—some tasks may be left unassigned!

2.4.4 A Novel Implementation of the Hungarian Method

In the preceding two sections, we have seen how the Hungarian method can be manually applied to derive an optimal assignment. The conventional computational implementation (the Kuhn-Munkres procedure) was also outlined. A point was emphasized that the conventional approach does not *strictly* follow the main steps of the Hungarian procedure (Algorithm 2-3).

The absence of a strict implementation for the original elegant form of the Hungarian method motivated me to seek an efficient implementation for it. This section describes an original contribution to this quest.

A strict implementation of the dual phase Hungarian method can take advantage of the strategy of *problem transformation*—the original optimal assignment problem is translated to a simpler maximal matching problem. The novel implementation described in this section follows this course. It emphasizes the use of an *ideal* or *preferred* MC in phase I. Such an MC enables the application of the *problem decomposition* strategy so that the ensuing maximal matching problem in phase II can be solved more efficiently. Instead of subjecting a matrix of size n×n to the maximal matching problem, the ideal MC yields two submatrices from the original matrix which can then be examined more expeditiously for their component SDRs. A third submatrix can also be identified as non-contributory to any SDR being sought. Narrowing the search space in this way is an example of the strategy of *problem reduction*.† Another

† Problem reduction differs from problem decomposition. The latter refers to the case of a problem *P* which can be split into two or more subproblems which can then be solved independently; their solutions constitute *P*'s solution. The former refers to the case of a problem *P* which can be scaled down to a smaller problem *P*', and solving *P*' solves *P* as well.

feature in this implementation is that a nonempty subset of the independent 0s is often found in phase I in the course of computing an ideal MC. Consequently, phase II only needs to search for the remaining independent 0s instead of all of them—another manifestation of the problem reduction strategy.

I will first elaborate in section 2.4.4.1 on the notions of ideal and preferred MCs. These play a central role in the new implementation. Section 2.4.4.2 explains the matrix marking scheme used in the course of finding independent 0s and lines forming an MC. Section 2.4.4.3 shows how to identify an obvious subset of the independent 0s, a step which narrows the scope of the problem. Sections 2.4.4.4 and 2.4.4.5 explain the use of α- and β-lines to form an MC. A procedure to construct an MC is described in section 2.4.4.6. In section 2.4.4.7, I will show how the procedure can be modified to construct a preferred MC. Section 2.4.4.8 shows how a preferred MC enables the remaining independent 0s to be found more efficiently by splitting the problem into smaller independent subproblems. It also outlines two maximal matching algorithms, either of which can be used to find the remaining independent 0s. Finally, section 2.4.4.9 presents the main algorithm.

2.4.4.1 Ideal and Preferred Minimal Covering

In general, for a given disposition of 0s in a matrix, several MCs exist. The number of elements (0s and non-0s) within one MC may be different from that of another. Since the maximal matching phase has to seek out independent 0s from among these elements, the smaller the number of elements the less will be the search effort. This suggests that we should use an *ideal* MC which has the least number of elements. The term *spread* will be used to refer to the number of *elements* which are crossed out only *once* in the MC. Doubly crossed elements are excluded because they cannot contribute to any SDR (inherent in the MC) and thus need not be searched (section 2.4.4.8). Note that spread differs from the *size* of the covering which refers to the number of *lines* in the MC.

In order to compute an ideal MC, its composition must be quantified. This is given in the following theorem:

Theorem 2-2:

For a n×n matrix, a complete minimal covering of r rows and c columns is ideal if $r = \lfloor n/2 \rfloor$.†

Proof:

Since the minimal covering is complete, $r + c = n$.

Let S be the spread of the covering.

The MC is ideal if S is minimal.

$$\begin{aligned} S &= nr + nc - 2rc \\ &= n^2 - 2rc \\ &= n^2 - 2r(n - r) \\ dS/dr &= 4r - 2n \\ &= 0 \text{ when } r = n/2. \end{aligned}$$

Therefore, S is minimal when $r = n/2$.

Note that r must be an integer.

When n is even, an integer value for r which minimizes S is $n/2 = \lfloor n/2 \rfloor$.

Consider next the case of n being odd.

n/2 must take an integer value which minimizes S.

Since S is monotonically decreasing with r when $r < n/2$, and S is monotonically increasing with r when $r > n/2$, the two integer values for r which can possibly minimize S are $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$.

Since $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$,

when $r = \lfloor n/2 \rfloor$, $c = n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$.

and when $r = \lceil n/2 \rceil$, $c = \lfloor n/2 \rfloor$.

† $\lfloor\ \rfloor$ and $\lceil\ \rceil$ are the integer floor and ceiling functions respectively.

Either way, $rc = \lfloor n/2 \rfloor \lceil n/2 \rceil$, and the minimal $S = n^2 - 2\lfloor n/2 \rfloor \lceil n/2 \rceil$. [1]

Hence either $r = \lfloor n/2 \rfloor$ or $r = \lceil n/2 \rceil$ minimizes S.

Corollary 2-2-1:

The spread of an MC is minimal if:

$r = c$ when n is even;

or $|r - c| = 1$ when n is odd.

The economy of search using an ideal MC is highly significant. This is evident from the ratio $S_{min}:n^2$ where S_{min} is the spread of an ideal MC and n^2 is the total number of elements which would have to be searched without the benefit of an MC to constrain the search, as would be the case in the conventional application of a maximal matching algorithm. From [1] in the preceding proof, it is clear that $S_{min}:n^2 \simeq 0.5$. Hence we can expect about 50% reduction in the time phase II takes. If an ideal MC is indeed obtainable, the reduction is even greater when some of the independent 0s have already been found in phase I.

$$\begin{array}{c|c|c|c|c|}
 & 1 & 2 & 3 & 4 \\ \hline
1 & 0 & & & 0 \\ \hline
2 & 0 & & 0 & 0 \\ \hline
3 & & & 0 & \\ \hline
4 & 0 & 0 & & \\ \hline
\end{array}$$

Ideal MC requires 2 rows and 2 columns to be crossed.
But crossing any 2 rows leaves 0s in at least 3 columns.
This disposition of 0s does not yield an ideal MC.
However, a preferred MC (rows 1, 2, 4 and col. 3) exists.

Figure 2-6. Non-existence of an ideal minimal covering.

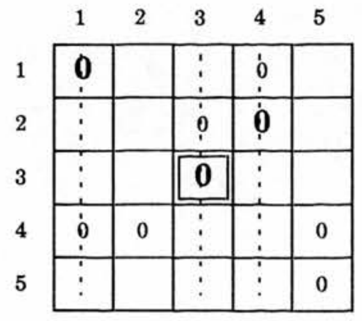
In practice, an ideal MC may not exist because the set of MCs is constrained by the disposition of the 0s. For example, the 0s in Figure 2-6 cannot be contained in any ideal MC comprising 2 row and 2 column crossings. (In most of the matrices illustrated in this chapter, only the 0s are shown.) The best recourse then is to aim for an MC which is closest to the ideal. We shall refer to such an MC as a *preferred* MC. It can be found by selecting an MC comprising r rows and c columns, and satisfying the criterion: min(|r–c|).

In subsequent sections, a preferred MC should be taken to mean a preferred if not ideal MC.

2.4.4.2 Marking Scheme

Initially, all rows, columns and 0s in the matrix are labelled *unmarked.* A 0 element can be selected as a member of an SDR only if it is unmarked. When an unmarked 0 is selected, it along with other 0s in the same row and column are *marked.* We also mark that row and column. This marking scheme is necessary to ensure that no two selected 0s appear in the same row/column.

2.4.4.3 Primary and Secondary Singular 0s



Singular 0s are shown as big bold 0s.
The only primary singular 0 is boxed.
Secondary singular 0s are unboxed.
Partial MC defined by the α-lines depicted by dashed lines.

Figure 2-7. Primary and secondary singular 0s.

The possibility of phase I terminating with the discovery of some independent 0s which must be in a feasible solution depends on the presence of *singular* 0s. A singular 0 is one which appears as the *only unmarked* 0 in a line. Note that even if an unmarked 0 is alone in the row it occupies but not so in the occupied column, it is considered singular. Singular 0s can also be classified either as *primary* or *secondary.* The former are those which appear alone in a line; no other 0s, marked or unmarked share that line. All other singular 0s are secondary; they are the result of marking the other 0s sharing the line. For example, the singular 0s in Figure 2-7 are shown in bold with the primary singular 0 shown boxed as well.

Singular 0s are important for two reasons. First, they form the set of independent 0s which is a partial SDR at the end of phase I—if row (column) i has only one unmarked 0, there is no alternative but to take that 0 as the row's (column's) representative. Second, an initial partial covering for the singular 0s can be trivially found, reducing the scope of the minimal covering problem. Thus phase I can be split into two subphases: Ia, which is to compute the set of singular 0s and its partial covering C_1, followed by Ib, which is to compute the remaining or complementary minimal covering C_2. The complete minimal covering C is $C_1 \cup C_2$. Phase Ia can be implemented along the following lines:

Algorithm 2-5:

```
        procedure Singular0s(K)
        /* Phase Ia: Given a matrix K, compute the set SingZ of singular 0s and its
           partial cover C1 which is represented by RowCover1 and ColCover1 */
        vars SingZ;   /* local variable */
1.      Label all 0s, rows and columns as unmarked.
2.      SingZ ← RowCover1 ← ColCover1 ← { }   /* RowCover1 and ColCover1 are
                                                  global variables */
3.      until no line has a single unmarked 0 in it do
4.          Choose an unmarked 0 which appears alone in a line;
5.          SingZ ← SingZ∪{(i, j)}     /* i and j are the row and column positions of
                                          the unmarked 0. */
6.          if it is the only unmarked 0 along its row
7.              then ColCover1 ← ColCover1∪{j}
8.              else RowCover1 ← RowCover1∪{i}
```

```
            endif
9.          Mark that 0 and all other 0s in row i and column j.
10.         Mark row i and column j.
        enduntil
11.     Return SingZ.
        endprocedure
```

In steps 6 to 8, it is the line orthogonal to the singular direction which is selected as a member of the covering. The reason is that for a covering to be *minimal*, any line which is constrained to pass through a singular 0 must eliminate the *maximum* possible number of 0s. For example, if row 1 in Figure 2-8, being the direction of singularity, is included in the covering the other 0 must be crossed out by another line. This covering comprising 2 lines is not minimal since column 2 alone suffices.

$$\begin{array}{c|c|c|}
 & 1 & 2 \\
\hline
1 & & 0 \\
\hline
2 & & 0 \\
\hline
\end{array}$$

Figure 2-8. Minimal covering is simply column 2.

2.4.4.4 α- and β-lines

The lines in the partial covering C1 found in phase Ia are called *α-lines* to distinguish them from the *β-lines* which will be found in phase Ib to complete the MC. This distinction is useful because the remaining unknown members of the SDR are determined from the β-lines. It is unnecessary to search among the α-lines because these have yielded the partial SDR represented by the variable SingZ.

For example, the α-lines in Figure 2-7 are columns 1, 3 and 4 and SingZ={(1,1), (2,4), (3,3)}.

Note that at the end of phase Ia, the elements outside the marked lines form a submatrix of size (n-|SingZ|)x(n-|SingZ|). This is the resultant submatrix after deleting from the original matrix the rows and columns marked in phase Ia. We shall refer to this submatrix as Kβ. Note also that all its lines have at least two 0s. There cannot be any 0-free line because the preliminary row and column reduction operations prior to phase Ia guarantee that every line has at least one 0. There cannot be any line with a single 0 in it because it would have been marked in phase Ia and therefore excluded from the submatrix. The good news is that instead of searching for a preferred MC over a matrix of size nxn, we now have a smaller submatrix to work on in phase Ib—the strategy of *problem reduction.* Correspondingly, the final phase II only needs to search among a smaller set of β-lines.

2.4.4.5 Flippable Sets of β-lines

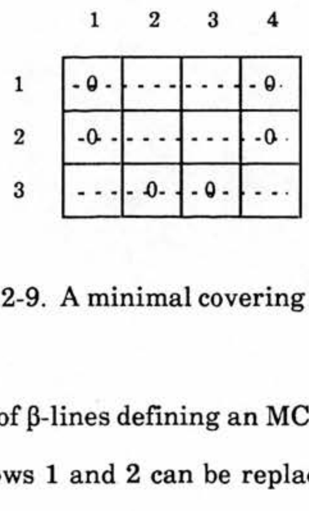


Figure 2-9. A minimal covering of β-lines.

Suppose we have the set of β-lines defining an MC for the submatrix as shown in Figure 2-9. The β-lines along rows 1 and 2 can be replaced by β-lines along columns 1 and 4 without affecting the size of the MC. (The terms *β-row* and *β-column* will be used to refer to the horizontal and vertical β-lines respectively.) These two sets of β-lines are said to be *flippable* since they are interchangeable. The importance of this notion will become clear in the next section.

2.4.4.6 Constructing Minimal Coverings Recursively

The algorithm which derives an MC is based on the recursive construction of MCs—the strategy of *problem decomposition.* To find the MC of a nxn matrix K, we split K into two submatrices K1 and K2 with dimensions 1xn and (n–1)xn respectively. Repeat the procedure on K2 until the base case is reached in which the new K2 is just a row. The problem then is twofold: (1) determine an MC for the base case; and (2) determine an MC for K given that of K2 and the disposition of 0s in K1.

The base case can be solved easily. Since there are at least two 0s in that row (recall that the submatrix passed on to phase Ib has at least two 0s in every line), the MC for the base case must be the row itself.

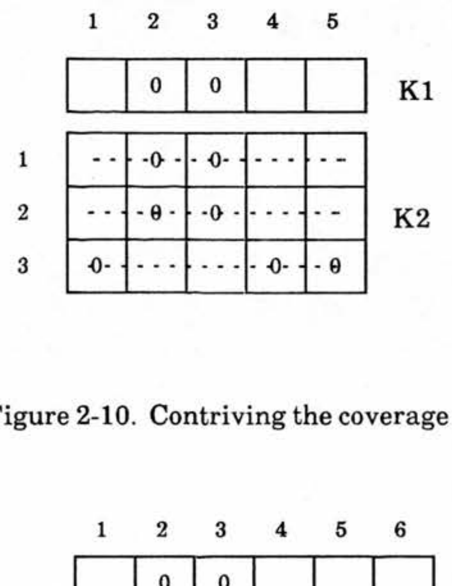
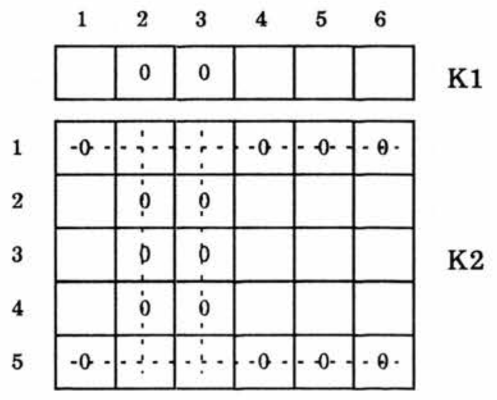


Figure 2-10. Contriving the coverage of 0s in K1.



β-columns 2 and 3 also cover the 0s in K1.

Figure 2-11. Fortuitous covering of 0s in K1.

The construction of an MC for K is more subtle. Either a new β-line is needed to cover the 0s in K1, or a fortuitous event may happen such that the 0s of K1 are covered by K2's β-columns. The latter event may also be contrived by flipping a set of β-rows in K2. As long as fortuity and contrivance are allowed to pre-empt the introduction of a new β-line, the construction process will guarantee that K's covering is minimal. Examples of the contrived and fortuitous cases are shown in Figures 2-10 and 2-11 respectively.

The strategy of recursive construction begs the question of how to determine when a new β-line is unnecessary, or when will fortuity or contrivance be pre-emptive in the way described above. The first step is to mark all 0s in K1 which are crossed by K2's β-columns. If no unmarked 0s are left in K1 then fortuity has taken its course.

If the situation is not fortuitous, we have to look for a flippable set of β-rows in K2 which if flipped will cross out the remaining unmarked 0s. If such a set exists, then we contrive a fortuitous state by flipping these β-rows. The problem of finding a flippable set is best solved by a goal-directed approach using Algorithm 2-6.[†] It is more intuitive and elegant than the alternative approach of examining K2's β-rows directly.

Algorithm 2-6:

```
    procedure FindFlipSet(K1,K2)

    /* Given row K1 and matrix K2, compute the set Tβ of tentative β-columns
       and the set Fβ of β-rows made redundant by the tentative β-columns.
       Returns a stack containing the results. */

    vars Tβ, Fβ;          /* local variables */

1.  Tβ ← Fβ ← { }
2.  foreach column c of K1 containing an unmarked 0 do
3.          Tβ ← Tβ∪{c}
```

[†] In the following algorithms, it is assumed that the sets of α- and β-lines are globally accessible.

```
        endforeach
    4.  foreach r in the β-rows of K2 until |Tβ| = |Fβ| do
    5.      if all the singly crossed 0s in r are covered by Tβ
    6.          then Fβ ← Fβ∪{r}
            endif
        endforeach
    7.  Return { |Tβ|–|Fβ|, Fβ, Tβ }
        endprocedure
```

Having set the stage, the recursive algorithm to compute an MC can now be defined:

Algorithm 2-7:

```
        procedure GetMinCover(KMat)
        /* Given a matrix KMat, find a set of β-lines which comprises a minimal
           covering for KMat. */
        vars K1, K2, Stack, N, Fβ, Tβ, SubCoverSize;  /* local variables */
    1.  if KMat is a row  /* base case */
    2.     then Add the new β-row which crosses out KMat's 0s.
           else
    3.        Arbitrarily split KMat into a row K1 and a rectangular matrix K2.
    4.        GetMinCover(K2)
    5.        Mark every 0 in K1 which is crossed by a β-column.
    6.        if all 0s in K1 are marked
    7.            then exit procedure  /* fortuitous event */
              endif
              /* K1 has some unmarked 0s. We should then check whether a
                 flippable set of β-rows can be flipped to cover the unmarked 0s of K1.
                 */
    8.        Stack ← FindFlipSet(K1,K2)
```

```
9.      N ← pop(Stack)    /* N denotes the shortfall of β-rows to form a
                             flippable set. */
10.     Fβ ← pop(Stack)   /* Fβ denotes the potential flippable set of β-rows. */
11.     Tβ ← pop(Stack)   /* Tβ denotes the tentative β-columns */
12.     if N = 0
13.         then   /*  Fβ and Tβ are flippable sets and a fortuitous state can
                       be contrived by flipping Fβ. */
14.                Delete all β-rows in Fβ.
15.                Confirm as β-columns those in Tβ.
16.         else Add the new β-row which crosses out K1's 0s.
        endif
    endprocedure
```

An example illustrating the recursive construction of an MC is shown in Figure 2-12.

Instead of splitting the matrix KMat along a row, we could alternatively split along a column. However, the procedure requires that every recursion uses the same direction of split.

2.4.4.7 Constructing Preferred Minimal Coverings

This section describes how procedure GetMinCover can be modified to find a preferred MC.

The main idea is to increase whenever possible and desirable the number of β-columns so that the difference between the number of β-rows and the number of β-columns in the MC is minimized. Recall that this is the condition for an ideal MC which has to be searched for independent 0s (see end of section 2.4.4.1). This deliberate increase of the number of β-columns supplements the compulsory β-columns (due to fortuity and contrivance) so

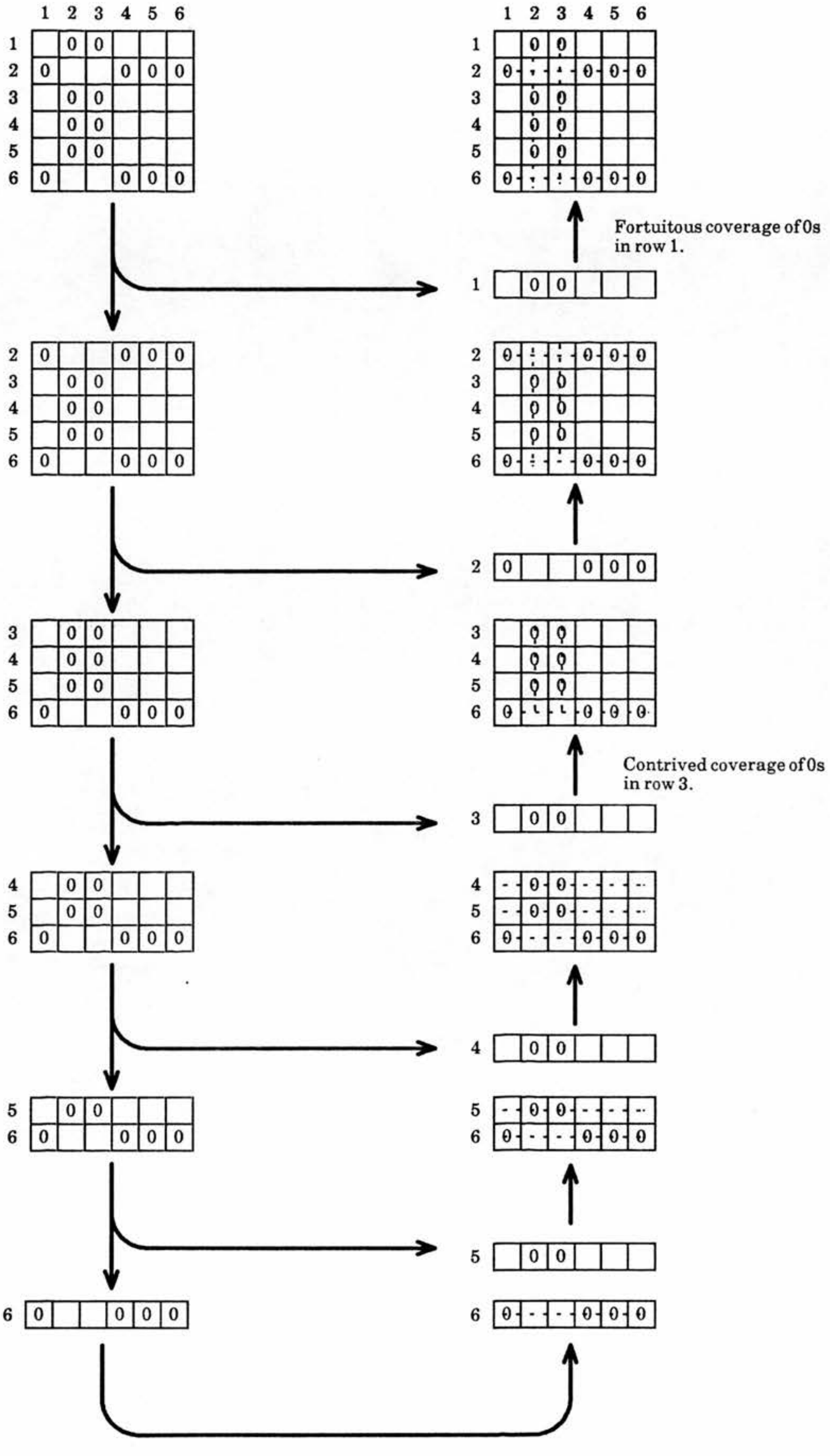
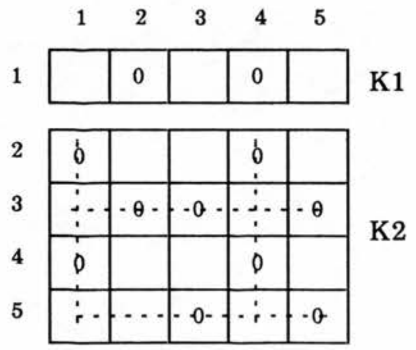

Fortuitous coverage of 0s in row 1.
Contrived coverage of 0s in row 3.

Figure 2-12. Recursive construction of an MC.

that a better proportion of β-rows and β-columns is obtained. For example, an ideal MC for a 4x4 submatrix Kβ has two β-rows and two β-columns. If at some intermediate stage of the GetMinCover procedure, the number of β-rows found thus far is two, and there is a choice of a new β-line being either a row or a column, it is better to use a β-column since this gives a preferred (in fact, ideal) MC.

The strategy to construct preferred MCs poses two questions: (1) when is it desirable to increase the number of β-columns apart from necessity due to fortuity and contrivance? (2) when is it possible? The question of desirability has been answered in the preceding paragraph when the rationale for the strategy was explained. The other question of possibility needs to be tackled under two circumstances.

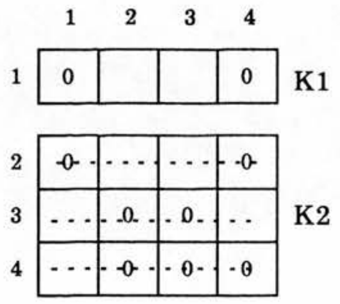


K1's 0 at (1,4) is covered by a β-column.
Its only unmarked 0 at (1,2) can be covered
by a new β-row at row 1
or a β-column at column 2.

Figure 2-13. Possibility of a new β-column at column 2.

The first is when there remains only one unmarked 0 in K1 *i.e.* it is the only 0 not covered by the existing β-columns. Since this 0 can be crossed by either a β-row or β-column passing through it, an opportunity thus exists to increase the number of β-columns by one. Figure 2-13 shows an example.

Secondly, even when there is more than one unmarked 0, it is still possible to increase the number of β-columns if the addition of q new β-columns at the positions of the unmarked 0s of K1 makes q – 1 β-rows in K2 redundant *i.e.* the 0s which these β-rows cover, are now covered by the new β-columns instead. The event occurs when at the end of procedure FindFlipSet, $|T\beta| - |F\beta| = 1$. This point is expanded in the proof of the following theorem and also by way of an illustration in Figure 2-14.



When procedure FindFlipSet returns,
the set of tentative β-columns in Tβ is {1, 4},
the set of β-rows in Fβ is {2}.
The condition $|T\beta| - |F\beta| = 1$ allows an MC to be formed by adding the β-columns in Tβ and deleting the redundant β-row in Fβ.

Figure 2-14. Possibility of new β-columns at columns 1 and 4.

Theorem 2-3:

If at the end of procedure FindFlipSet, $|T\beta| - |F\beta| = 1$, then the new set of β-lines, formed by deleting the β-rows in Fβ, and adding the β-columns in Tβ, constitutes an MC for KMat.

Proof:

Suppose n is the cardinality of the MC of K2 and $|F\beta| = k$.

Since procedure GetMinCover invoked FindFlipSet, a fortuitous event has not occurred (see lines 6 to 8 of Algorithm 2-7).

A fortuitous event cannot be contrived since this requires $|F\beta| - |T\beta| = 0$ (lines 12 and 13 of Algorithm 2-7).

Since fortuity and contrivance did not pre-empt the need for a new β-line, the cardinality of KMat's MC must be $n + 1$.

If the tentative β-columns in Tβ are accepted, then the $k - 1$ redundant β-rows in Fβ can be deleted without leaving any 0 in K2 uncovered.

Since the 0s of K1 are covered by the new β-columns, all the 0s in KMat are therefore covered.

Since the new β-columns exceed the deleted β-rows by 1, there are now $n+1$ β-lines which thus constitute an MC for KMat.

Theorem 2-3 gives a decision procedure for introducing a new β-column in place of a β-row across K1. Another way of viewing this is to observe that when $|T\beta| - |F\beta| = 1$, a β-row across K1 teams up with those in Fβ to form a flippable set and it is this set which has been flipped.

Theorem 2-3 is central to the algorithm which finds not just an MC but one which is also preferred. It gives us a way to avoid increasing the number of β-rows which would otherwise increase the spread of the MC of submatrix Kβ.

Finally, here is how procedure GetMinCover can be modified to find a preferred MC. Simply replace line 12 in Algorithm 2-7 with the following:

```
if  N=0 or (N=1 and increasing the number of β-columns, by
    including those in Tβ and deleting β-rows in Fβ, reduces the
    spread of the MC of the submatrix Kβ)
```

The modified procedure will always attempt to find a preferred MC. But this is not necessary because it is the final *complete* MC which should be preferred. Intermediate MCs serve only to generate more 0s via the matrix modification procedure. So it seems that efficiency has been impaired by prematurely looking for the preferred MC. The solution, then, is to apply the unmodified version of GetMinCover until a complete MC surfaces. The β-lines are then reconstructed, this time using the modified version of GetMinCover. Thus there is a trade-off: in an attempt to avoid unnecessary search for a preferred MC, an additional construction of β-lines is necessary. The worth of it depends on the problem size. With large matrices we can expect more intermediate coverings will be computed before reaching phase II and the changes will be more worthwhile.

2.4.4.8 Phase II: Maximal Matching

When a preferred MC of sufficient size is found to guarantee (according to Theorem 2-1) that enough 0s exist for an SDR to be found, phase II is entered. Recall that phase Ia will typically have found some of the independent 0s which must be in any SDR. Phase II only needs to look for the remaining independent 0s among the β-lines specially constructed to give the least possible spread so that the search effort in this phase is minimized.

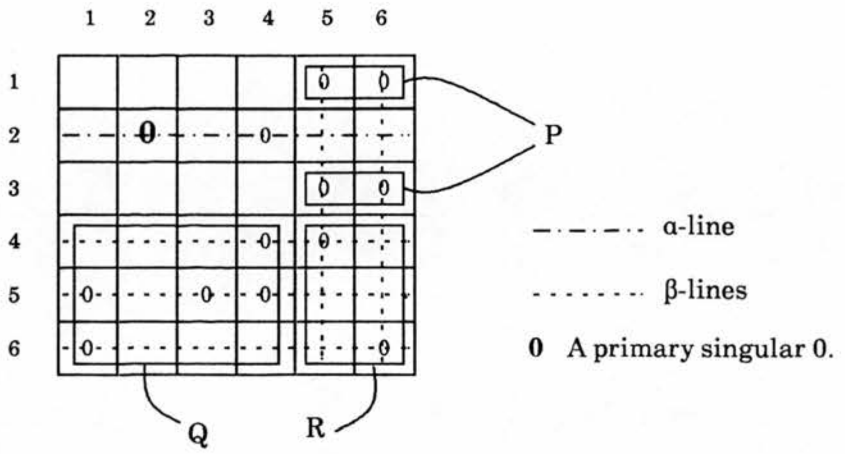


Figure 2-15. Problem reduction and decomposition via β-lines

There is another important advantage of this special combination of β-lines: it gives another opportunity for the strategies of problem decomposition and reduction to be applied within phase II. Figure 2-15 illustrates this point. The β-lines are rows 4, 5 and 6, and columns 5 and 6. The elements covered can be represented with the 3 submatrices P, Q and R. 0s in R are doubly crossed; picking any one will mean that no other 0s along the two β-lines through it can be selected. If we are to have as many independent 0s as there are β-lines, any selected 0 must "consume" only one β-line. Hence phase II can disregard 0s in R, thus *reducing* the scope of its problem to P and Q. Now observe that picking any 0 in P does not affect selection of 0s in Q (and vice-versa) since P and Q have no common rows or columns. P and Q are thus independent and phase II can be applied twice, once to P and then to Q. Effectively, the original problem has been *decomposed* into two smaller independent problems which can be solved more readily.

The final step is to compute the component SDRs of P and Q. Their union with SingZ (the component SDR from C_1, the covering of α-lines discovered in phase Ia) is the complete SDR. We can use either of two existing methods for computing an SDR given any 0-saturated matrix which has sufficient 0s to guarantee the existence of a solution.

The first method is attributed to Hall (1935). It is based on a constructive proof of his theorem on systems of distinct representatives which is more commonly known as the marriage theorem (given a set of men and women, and each man has a set of women he finds acceptable, then every man can be married to an acceptable woman if and only if for any k sets, their union contains at least k distinct women). The proof is constructive because it derives a maximal matching as a result of the proof. Here is the algorithm to compute a maximal matching:

Algorithm 2-8:

```
        procedure MaxMatch(KMat)
        /*  Compute a maximal matching (SDR) for KMat.  */
        vars M, P; /* local variables */
1.      M ← {} /* any matching can be used initially. */
2.      Find an augmenting path P relative to M. /* see Christofides (1975) */
3.      M ← M⊕P  /* augmented matching now has one more member */
4.      if M is not maximal
5.              then Go to step 2.
        endif
6.      Return M.
        endprocedure
```

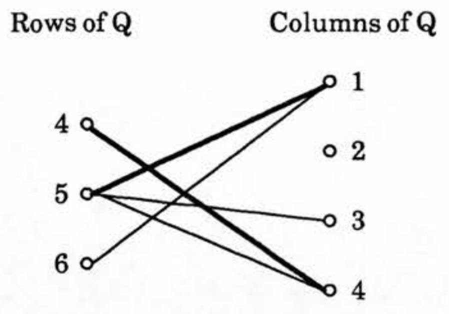


Figure 2-16a. A matching M for Q.

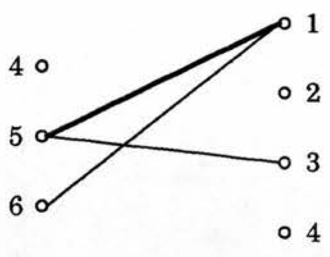


Figure 2-16b. Augmenting path P relative to M.

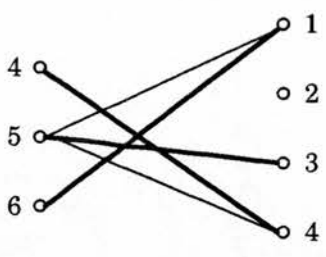


Figure 2-16c. Augmented match M⊕P.

The key to procedure MaxMatch is to find an augmenting path relative to a matching M. If M is not maximal then such a path must exist according to a theorem later established by Berge (1957). The significance of the augmenting path is that it can be used to increase the current matching by an additional member. For example, Figure 2-16a shows the bipartite graph which is represented by the submatrix Q (Figure 2-15). 0s in the matrix appear as edges in the graph. A current matching is shown as a set of edges in bold. An augmenting path relative to it is shown in Figure 2-16b. Applying the symmetric difference operator ⊕ means that the new matching is constructed by deleting the edges in M which are in P and adding the unmatched edges of P to M (see Figure 2-16c). Since P's unmatched edges exceed the matched edges by one, the cardinality of the new matching is increased by one. Repeating the augmenting process, the maximal matching is bound to be found eventually.

The second algorithm also relies on augmenting paths. Developed by Hopcroft and Karp (1973), it uses a maximal set of vertex-disjoint shortest augmenting paths relative to M, instead of just any one augmenting path. It can be implemented by replacing steps 2 and 3 in Algorithm 2-8 with:

```
2.   Let L be the length of the shortest augmenting path relative to M.
     Find a maximal set of augmenting paths {$P_1$, $P_2$, ..., $P_r$} such that:
         a. $\forall i \in [1,r], |P_i| = L$.
         b. all the $P_i$'s are vertex-disjoint.
3.   $M \leftarrow M \oplus P_1 \oplus P_2 \oplus \ldots \oplus P_r$
```

Although this modification has a better complexity order—$O(n^{2.5})$ against $O(n^3)$ of the original algorithm—it is more complicated. Computing the maximal set of vertex-disjoint shortest augmenting paths is not at all a trivial step. Its *practical* efficiency remains to be gauged by empirical evidence. This is important because efficiency also depends on the complexity's constant of proportionality. If this is large, theoretically less efficient algorithms with smaller constants of proportionality may turn out to be more efficient in reality (Aho *et al*, 1983, pp. 20-21).

2.4.4.9 The New Implementation

Here is the outline of how the new implementation of the Hungarian method is coded:

Algorithm 2-9:

```
     procedure NewHungarian(K)
     /* Given a matrix K of cost values, compute an optimal assignment.
        Assumes α- and β-lines are globally accessible. */
     vars N, Kβ, P, Q, SingZ, S1, S2;      /* local variables */
1.   Let N be the number of pairings in the optimal assignment.
2.   Reduce all applicable rows and columns in K.
```

```
    /* Phase Ia: compute the initial partial MC and SDR defined by the α-lines
       and SingZ respectively. */
3.  SingZ ← Singular0s(K)
4.  if number of α-lines < N
        then /* Phase Ib: compute rest of the MC. */
5.          Form submatrix Kβ by removing from K all rows and columns
            marked in phase Ia.
6.          GetMinCover(Kβ)  /* GetMinCover computes the β-lines for Kβ
            */
7.          if total no. of α- and β-lines < N
                then   /* matrix is not saturated with independent 0s. */
8.                     ModifyMatrix(K)
9.                     Go to step 6.
            endif
            /* Phase II: complete MC has been found; proceed to find the
               other independent 0s. */
10.         Construct independent submatrices P and Q from Kβ.
            /* P and Q contain the elements in Kβ which are crossed only
               once by a β-column and β-row respectively. */
11.         S1 ← MaxMatch(P)
12.         S2 ← MaxMatch(Q)
    endif
13. Return S1 ∪ S2 ∪ SingZ.
    endprocedure
```

As far as I can ascertain, there is no other computational implementation *strictly* following the main steps of Algorithm 2-3; NewHungarian is the first. The conventional

Kuhn-Munkres implementation described in section 2.4.3 is a variation of the original Hungarian theme.

The reader can refer to appendix A for a proof of NewHungarian's termination and that it has a complexity of $O(n^4)$.

2.4.5 Comparison of the NewHungarian and Kuhn-Munkres Procedures

The main differences are:

- NewHungarian being a strict implementation of the Hungarian theme has two main phases; Kuhn-Munkres has only one phase. A single phase implementation loses the intuitive clarity of the Hungarian theme, but it gains two advantages: (1) it is easier to code—for example, it does not require code to compute a maximal matching as in phase II; and (2) it does not require a special procedure to find an MC since this can be derived from the same labelling procedure for finding an augmenting path.

- NewHungarian makes use of augmenting paths in phase II *after* a complete MC has been found whereas Kuhn-Munkres uses augmenting paths *before* such an MC has been found. NewHungarian is thus able to derive an SDR with zero or more successive augmentations of the suboptimal matching. In Kuhn-Munkres, augmentation will typically involve several interludes of the matrix modification procedure.

- The matrix modification procedure serves different roles. In NewHungarian, it serves to increase the size of the MC. In Kuhn-Munkres, it serves to produce an augmenting path.

- NewHungarian emphasizes the identification of a preferred MC as its first objective whereas Kuhn-Munkres emphasizes augmentation of its suboptimal matching as its sole objective; the MC appears as an incidental by-product in its pursuit of augmenting paths.

- NewHungarian's complexity is $O(n^4)$ whereas it is $O(v^2n^2)$ for the Kuhn-Munkres algorithm.

NewHungarian and Kuhn-Munkres are similar in that minimal coverings, augmenting paths, and matrix modification feature in both, although in different order and emphasis.

Given the above differences and similarities, it would appear that the conventional implementation should be more efficient since it does not involve a separate maximal matching phase. However, experimental data bear out the opposite. Both algorithms were implemented in Interlisp-D in a Xerox 1186 workstation. Randomly generated square cost matrices with size ranging from 10 to 200 at increments of 10 were used. For each size, a set of 20 different matrices were generated. The running times (with garbage collection time excluded) for NewHungarian and Kuhn-Munkres to solve each set were measured accurately using a system-provided function.

Figure 2-17 shows a plot of running times against the matrix sizes. It reveals that NewHungarian is more efficient and its superiority improves with increasing problem size. A more quantitative illustration of this point is given in Figure 2-18 which shows the running times of NewHungarian as a proportion of those of Kuhn-Munkres. Note that the reduction of running times rapidly exceeds 40%. We can also derive the expected time complexity by estimating the complexity function using a least squares fit on the function kn^x. In the case of NewHungarian, the estimated function is $0.14n^{1.96}$, and for Kuhn-Munkres, it is $0.08n^{2.24}$.

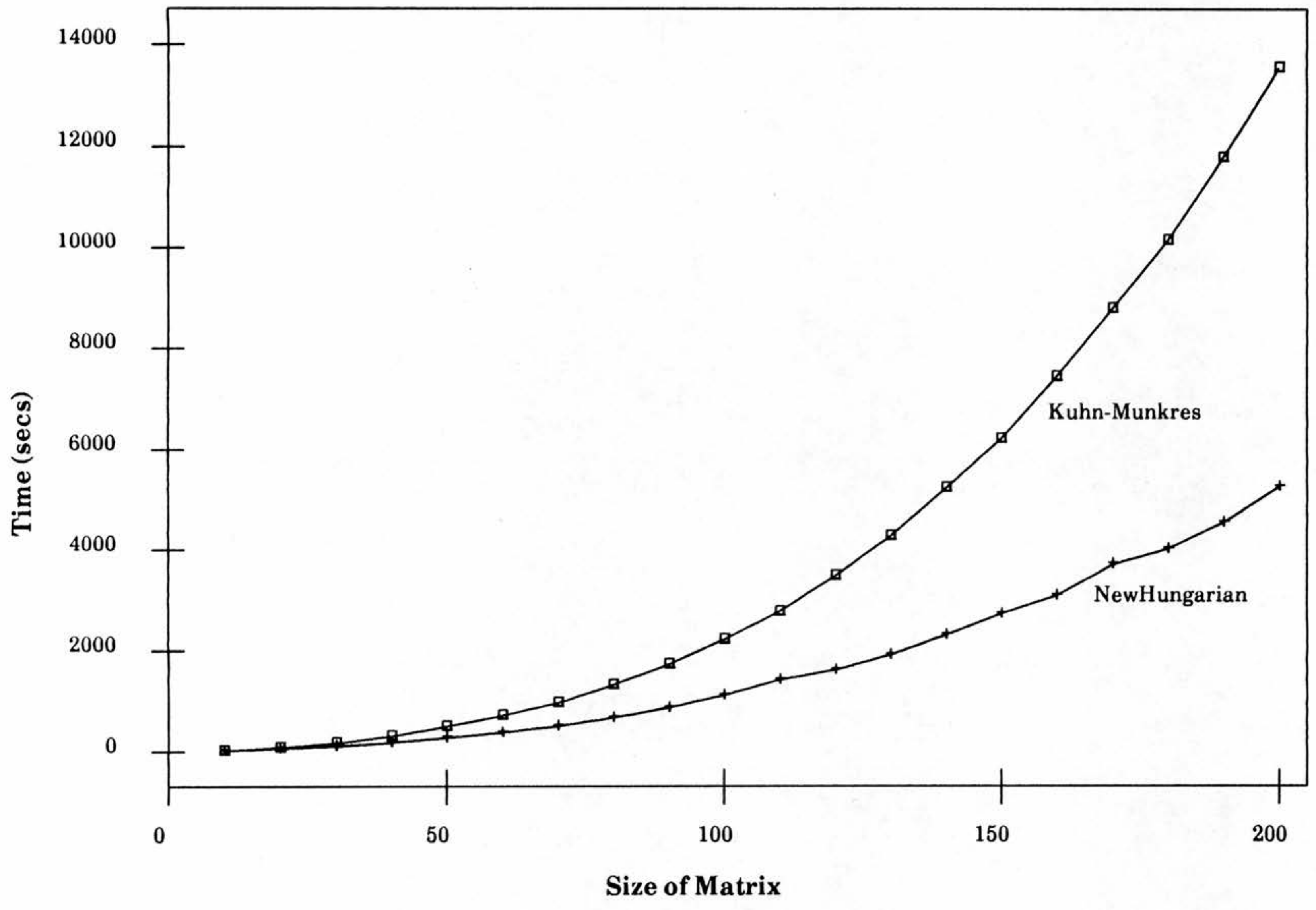


Figure 2-17. A comparison of running times with increasing problem size.

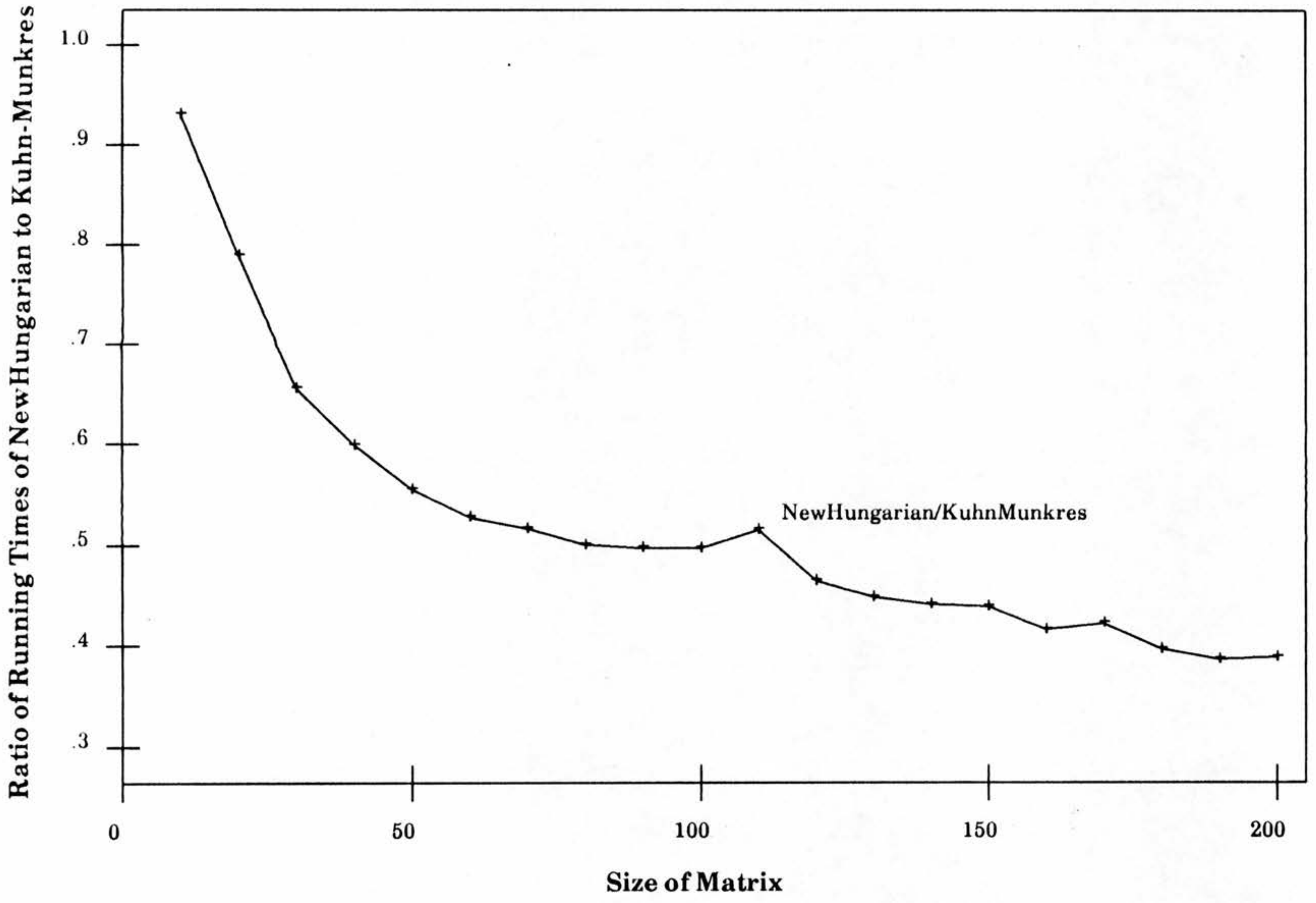


Figure 2-18. Efficiency of NewHungarian compared with Kuhn-Munkres.

The difference in the constants of proportionality is due to the greater overheads of NewHungarian and it explains why, for small problem sizes, NewHungarian's superiority is not highly significant. But as the problem size increases, the strategies of problem transformation, reduction and decomposition which NewHungarian adopts, by seeking preferred MCs, begin to be highly effective in improving efficiency.

2.5 Summary

In this chapter, I have argued that AGV-task assignment should meet the global optimality criterion since the benefits—better overall utilization of AGVs; more tasks achieved between recharging; less conflict resolution; and less compromising of AGVs' plans—far outweigh the nominal computational demands. It is shown that a global optimal assignment algorithm—the Hungarian method—yields significantly better overall assignment costs than a local optimization algorithm; and yet with extremely short computation time.

The main contribution of the chapter is a novel implementation of the Hungarian method which has been found to be more efficient than the conventional implementation. The conventional approach emphasizes the use of augmenting paths to increase the size of a suboptimal matching while the cost matrix is unsaturated with independent 0s. This approach is a variation of the original Hungarian theme. In contrast, the new implementation follows the dual phase Hungarian method strictly and thus preserves its elegance and intuitive clarity.

The dual phase approach first modifies the cost matrix till it is saturated with independent 0s. This transforms the optimal assignment problem to the simpler maximal matching problem which is solved in the second phase. A difficulty in the original approach has been the computation of the minimal covering which is central to the matrix modification procedure. I developed an algorithm which directly computes the minimal covering by decomposing the minimal covering problem to two subproblems—finding the obvious set of α-lines, followed by the not-so-obvious set of β-lines. The latter is derived by an elegant recursive decomposition technique. I have also shown how a preferred MC can be found which provides the key to improving search efficiency. The preferred MC achieves its effect by yielding (in general) three submatrices, one of which can be disregarded, thereby reducing the scope of the maximal matching problem to the remaining two submatrices. These two submatrices can then be searched independently and more expeditiously for their contributions to the final solution. Besides contributing a new and more efficient algorithm to the optimal assignment problem, I have also shown how certain general principles of problem solving can be applied to good effect.

The motivation for this novel implementation initially came as a challenge to develop a strict implementation. Along the way, an important observation was made that the structure of the minimal covering affects greatly the subsequent effort to find the set of distinct representatives. This observation led to the discovery of the ideal minimal covering. In the elaboration of the new implementation, I have tried to highlight the underlying problem solving strategies rather than a mere exposition of the contents. This approach of reflecting on how a problem is solved while focussing on the solution, is both interesting and helpful in revealing problem solving insights.

Chapter 3

Route Planning

3.1. Introduction

In motivating the need for optimal assignment of tasks (section 2.2) the advantages of economy of effort—better overall utilization of AGVs; more tasks performed between recharging; less conflict to resolve; and less compromise of individual AGV plans—were established. Optimal assignment satisfies the objective of economy of effort by minimizing the total distance which has to be traversed to achieve a set of tasks. The same objective also requires that every AGV minimize the route taken to achieve its assigned task. Basically, the shortest path problem needs to be solved.

The shortest path problem occurs frequently and in different circumstances in the AGV planner. For example, whenever a new fetch-and-deliver task appears, every available AGV must plan two routes for each of the alternative pick-up points. If a route segment is obstructed during execution, another shortest route must be computed, this time with new restrictions in force. The problem also appears as a subproblem in collaborative planning (see Chapter 7), and in the maximal matching subproblem in Chapter 2 if one wishes to use the theoretically more efficient Hopcroft-Karp algorithm. Its frequent occurrence means that an efficient algorithm for it is desirable. Efficiency in this context means computational speed and conservative memory requirements. Speed is especially important during dynamic replanning; an alternative plan should be found as fast as possible if matters are not to be overtaken by failing circumstances. Hence an efficient and admissible algorithm for the shortest path problem is needed for the AGV movement planning application.

The theme of this chapter is that admissible AGV routes can be found efficiently using bidirectional heuristic search. We first review the many existing search algorithms and show how these fall short in the AGV application. The review uncovers several key issues in search theory which are examined in section 3.3. A better and novel algorithm (BS*) which is bidirectional, admissible and informed is then described in section 3.4. Section 3.5 compares its performance with algorithms in the same class, as well as with A*. BS*'s performance is shown to be superior in its class. BS* is also comparable to A*, but BS* has an important advantage over A*—it is amenable to parallel implementation which means that its running time can be potentially reduced further by as much as 50%.

3.2. Review of Search Algorithms

3.2.1 Search Characterization

Graph searching algorithms abound in the literature. The algorithms can be characterized broadly by three aspects: search direction; admissibility; and use of heuristics. Search direction can be *unidirectional* from the start node to the goal node or vice-versa. If there is a function for retrieving the parents of a node and the goal node is known *a priori*, then the search can be *bidirectionally* pursued starting from the start and goal nodes. A search algorithm is *admissible* if it returns the optimal solution if a solution exists. Otherwise it is non-admissible. Search algorithms may or may not make use of *heuristic* information to guide their search. Heuristically-guided or *informed* search considers the estimated remaining path cost of unexpanded nodes in the search tree. This information is included in the overall rating of a node's promise in leading toward the goal node. Algorithms which are not heuristically guided are classified as *uninformed*. Under certain conditions, heuristic search algorithms may also be admissible.

Search algorithms are also classified as breath-first, depth-first, best-first or some combination of these; a good account of these can be found in Winston (1984). Of these, an admissible algorithm must follow the best-first policy in selecting the next node to expand.

3.2.2 Unidirectional Algorithms

3.2.2.1 Moore's Algorithm

The earliest admissible search algorithm was developed by Moore (1959). It assumed that all arcs in the graph have the same cost. It labels the start node with the value 0 and then *expands* it, giving the first generation of nodes whose members are its immediate successors. Until termination, search proceeds by choosing arbitrarily an unexpanded node from the most recent generation, say the i-th generation. The immediate successors which have not been generated before are labelled with the integer $i+1$. Nodes generated also have an associated pointer to their parent. The search terminates when the goal node is found or no nodes can be further expanded. When the goal node is found, its label gives the minimum cost in terms of the number of arcs which must be traversed to reach the goal from the start node. The shortest path can be constituted from a trace of the parent pointers starting from the goal node. Where the real arc costs may differ and the true shortest path is required, Moore's algorithm is inapplicable.

3.2.2.2 Dijkstra's Algorithm

Dijkstra (1959) formulated an algorithm which computes the shortest paths from a node to all the other nodes in a graph with positive arc costs. Hence the first shortest path algorithm can be credited to Dijkstra. His algorithm uses two disjoint lists of nodes: the list S of expanded nodes and V of unexpanded nodes in the graph. It begins by placing the start node, labelled 0, in S and all the other nodes in V. The nodes which are not successors of the start node are labelled with infinity or a number greater than the largest arc cost. Successor nodes are labelled with their respective arc cost from the start node. Until V is empty, the algorithm selects the node in V with the smallest label and transfers it to S. Ties are resolved arbitrarily. The labels of the remaining nodes in V are then revised (if necessary) to reflect the shortest path from the start to the corresponding node in V subject to the constraint that the path passes only through nodes in S.

Dijkstra's algorithm can be adapted to compute the shortest path to a particular goal node by terminating the search when the goal node is placed in S. A more efficient implementation is to restrict the consideration of nodes during the label updating phase to the unexpanded successors of the node just transferred to S. This is correct because only such nodes can possibly be updated with a lower value label. In fact, this modified Dijkstra algorithm is the same as the popular A* algorithm (section 3.2.2.5) when the *h* function returns 0 always.

Whereas Moore's algorithm follows the breadth-first search strategy, Dijkstra's follows the best-first strategy. Both are instances of uninformed search. If all arc costs are the same or deemed as such, then Dijkstra's algorithm is similar to Moore's.

Dijkstra's algorithm has the advantage of easy implementation, but if a particular shortest path is required, it is inefficient. Lacking heuristics to guide its search, it tends to be wasteful in exploring unpromising avenues.

3.2.2.3 Floyd's Algorithm

Floyd (1962) developed an uninformed search algorithm (a 9-line Algol program) to compute the shortest path between every distinct pair of nodes in a graph. It is more elegant than the obvious alternative of repeating Dijkstra's algorithm using a different start node each time. However, its simplicity incurs an inefficiency in having to examine all nodes in the graph during each label update phase. It can be improved by considering only the remaining unexpanded nodes. An even better modification is to consider only the unexpanded successors of the node just expanded—as in the improved Dijkstra algorithm.

Floyd's algorithm is appropriate if all the shortest paths in the problem graph should be pre-computed. A particular shortest path can subsequently be found by looking up a matrix of pointers along with its associated distance. However, this is impractical for large graphs. Besides, many of the shortest paths will not be required. Pre-computed paths may also be invalidated by path segments which later become impassable for some reason. Hence much of its computation along with the memory to hold information may be useless.

3.2.2.4 Graph Traverser

Doran and Michie's Graph Traverser (1966) is the progenitor of heuristic search algorithms. In it, every expanded node has an *estimate* of the *remaining* path cost from it to the goal node. These cost estimates are used to guide the search. In considering which node to expand next, it does not take into account the path cost *so far* (*i.e.* from the start node to the current node). Consequently, it cannot be guaranteed to be admissible. This is acceptable if the search objective is not to find the best solution path from the *initial* to the goal state, but to find what seems to be the best path from the *current* to the goal state. For example, in some computer games such as chess, the history of moves is irrelevant† and what matters is the best way of forcing a checkmate from the current state of play.

3.2.2.5 A and A*

These algorithms developed by Hart *et al* (1968) overcome the weaknesses of the Graph Traverser by a more comprehensive consideration of the estimated total cost of a path constrained to pass through a node. Both algorithms use an evaluation function $f(n)$ for each node n which is the sum of two terms: $g(n)$, the minimum cost so far from the start node to n; and $h(n)$, which is the estimated remaining cost from node n to the goal node. A* differs from A in that $h(n)$ is a *lower bound* to the *true* remaining cost in A* but not so in A. The composite evaluation function $f(n)$ combines a breadth-first influence *via* $g(n)$ with a depth-first influence *via* $h(n)$, and pursues search in a best-first manner.

A significant contribution of A* is its supporting proof that its admissibility is guaranteed by using underestimated remaining path costs. Because its performance is generally superior to previous uninformed admissible search algorithms, it has become one of the most popular search algorithms in AI. This explains why latter informed algorithms, both unidirectional and bidirectional, are based on A*. It thus deserves a closer examination, which will be given in sections 3.3.6 and 3.3.7.

3.2.2.6 HPA and HPA+

Pohl's (1970, 1977) HPA+ (heuristic path algorithm) is yet another A-like algorithm. It differs from A only in that the g and h terms in the f function have non-negative weight coefficients which add to 1. In A, these coefficients are always 1. HPA+ belongs to the class of informed unidirectional search algorithms. Its admissibility depends on the weightings and the h function.

† This assumes that a player does not consider the likely moves of the opponent on the basis of apparent tactics induced from previous moves.

Pohl's account of HPA is inconsistent. In Pohl (1970), HPA is described as a simplified variant of HPA+ in that nodes visited (open or closed) are ignored. In Pohl (1977), he allows for open nodes to be revisited. Whichever version is used, HPA is generally non-admissible.

Pohl showed that in the absence of error in the *h* estimates, the search effort in terms of the number of nodes expanded is minimized when the *g* term is weighted less than the *h* term. In fact, only nodes along the shortest path are expanded. There is little practical significance in this result because it is unrealistic to expect perfect *h* estimates in most problem domains.

Pohl also suggested a dynamic weighting strategy wherein the weights varied with the node as well as the state of the search. The idea behind this is to have a greater depth-first influence initially and when the goal is near, breadth-first search begins to dominate. This is similar to the manner in which a blind person might attempt to find an object. Knowing the general direction to the destination, he could take large steps forward boldly and when in the vicinity of the destination, begin groping around for the object sought after. Conceptually, the idea is sound, but often it is difficult to find good domain parameters to tune the weights dynamically.

HPA+ and HPA are interesting as generalizations of A but they have not significantly improved on A*. Hence their merit for admissible search remains to be established.

3.2.2.7 B and B'

Martelli's (1977) B algorithm has better worst-case performance than A* when the consistency assumption (see section 3.3.6) does not hold. For a graph of N nodes, B's running time is $O(N^2)$ against A*'s running time of $O(2^N)$. B uses a global variable F which holds the current maximum of the f values of expanded nodes. In choosing which node to expand next, the candidate nodes are divided into two disjoint sets. Candidate nodes with *f* values below *F* are in one set S1 and the rest in the other set S2. If S1 is empty, B selects the node in S2 which has the minimum *f* value, as A* would. Otherwise, B takes the node from S1 which has the minimum *g* value. Ties are resolved in favour of the goal node.

The rationale for this is that *h* values, being inconsistent, may be misleading and should therefore be used only when necessary. Since every node with an *f* value below *F* must be expanded sooner or later (a lemma proved by Hart *et al*, 1968), such nodes should be expanded as soon as possible. Nodes in S1 are deemed to have unreliable *h* estimates, which explains why selection from S1 is based only on the *g* value.

When the consistency assumption holds, *f* values of expanded nodes are monotonically nondecreasing and S1 is always empty. In this case, B is identical to A*. The B algorithm was later modified to B' by changing the *h* estimates in the course of search (Mérõ, 1981, 1984). B' performed better than B by expanding no more nodes than B would and its worst case behaviour is at least twice as good as B. B and B' are thus better than A* when the consistency assumption does not hold.

3.2.3 Bidirectional Algorithms

3.2.3.1 Nicholson's Algorithm

The first bidirectional admissible search algorithm was introduced by Nicholson (1966). Search in both directions is based on Dijkstra's uninformed best-first algorithm. It terminates when the minimal cost of the complete paths found hitherto is less or equal to the sum of the minimum cost of the incomplete paths found from the start and the goal nodes. Until then, the node to be expanded next is that which has the least cost so far. Nicholson's algorithm is superior to any unidirectional uninformed search algorithm, in terms of number of nodes expanded and running time. However, it cannot match A*'s superiority (see Figure 3-1).

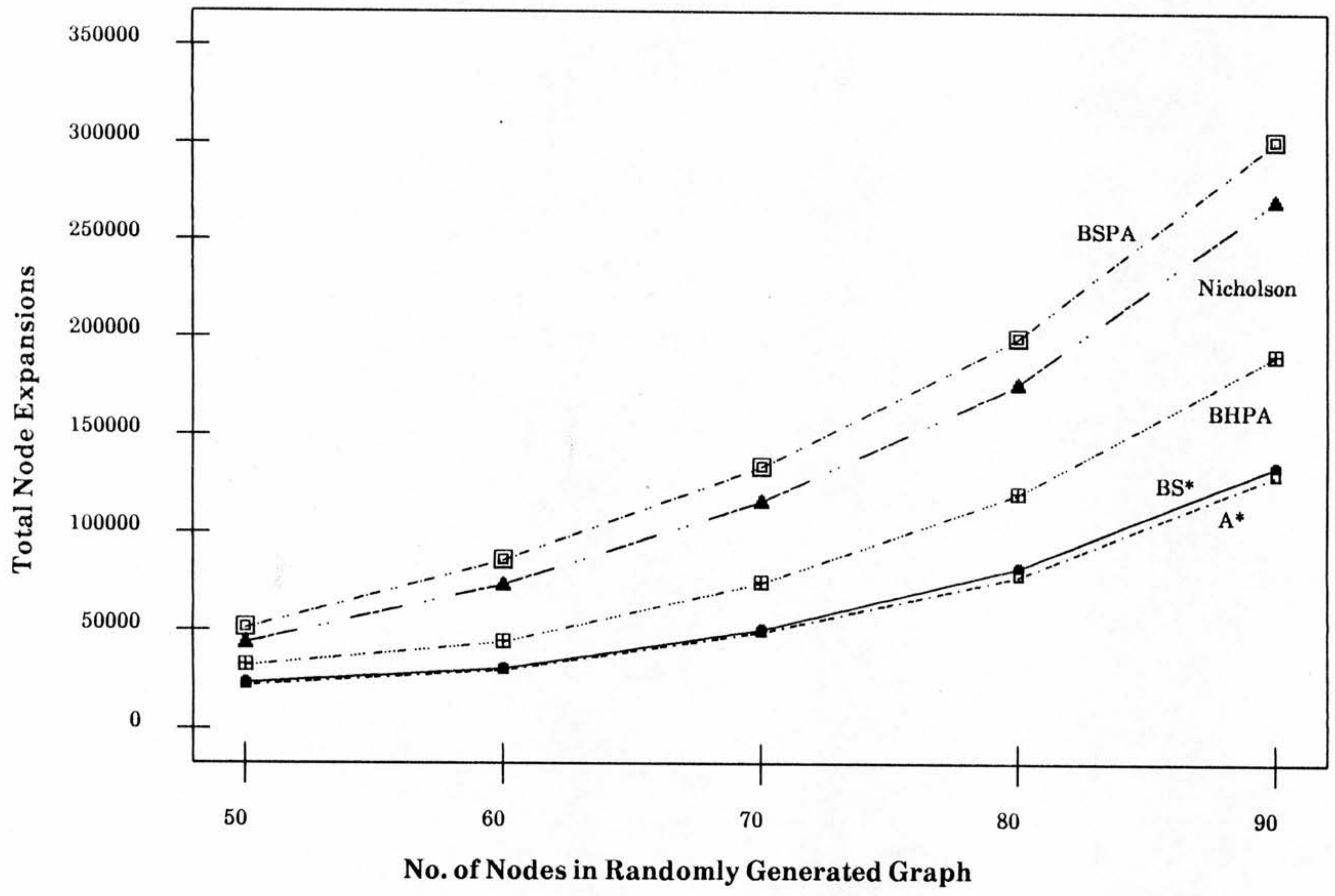


Figure 3-1. Comparison of total node expansions of admissible algorithms.

A non-admissible version can be easily adapted by terminating the search once a complete path is found linking the start to the goal node. This terminating condition can be used for any bidirectional algorithm if the shortest path is not required.

3.2.3.2 BSPA and VGA

Pohl's (1971) BSPA (bidirectional shortest path algorithm) based on Dijkstra's algorithm is another bidirectional admissible uninformed algorithm. It was originally documented as VGA (very general algorithm) in Pohl (1969). BSPA differs from Nicholson's algorithm in two ways. First, the terminating conditions differ. In BSPA, admissible search terminates when a node chosen for expansion has already been expanded by the search effort from the *other* direction. The least cost path is then the best complete path with *all* nodes along it expanded from at least one direction.

The second modification in BSPA has to do with what Pohl called the *cardinality comparison principle*. The principle states that the next node to expand should be chosen from the search direction which has a lesser number of candidate nodes (open nodes). Having determined the search direction, the corresponding node with the smallest cost is selected for expansion. Pohl showed theoretically and empirically that making progress in a sparser region is a better strategy than Nicholson's strategy of moving equidistantly (*i.e.* select the open node with the smallest g value) from the start and goal nodes.

Apparently, Pohl (1969) misinterpreted Nicholson's terminating condition and actually meant BSPA to differ only on the principle which determined the next search direction. It is nevertheless true that the cardinality comparison principle is superior, but only if the comparison is based on the same terminating condition. In fact, even with the inferior equidistant principle, Nicholson's algorithm performed better (see Figure 3-1) because its terminating condition is far superior to Pohl's version. My results contradict Pohl's conclusion that BSPA is superior; understandably so, now that the misinterpretation has been pointed out. Since Nicholson's termination condition can likewise be used for other admissible bidirectional algorithms, its superiority should be proven. This is done in appendix B.

3.2.3.3 BHPA and VGHA

Following Doran's (1966) bidirectional Graph Traverser, Pohl (1971) formulated a bidirectional informed search algorithm—BHPA (bidirectional heuristic path algorithm)—which is based on his HPA. BHPA is a specialized admissible version of VGHA (very general heuristic algorithm) which is described in Pohl (1969). Whereas BHPA is a bidirectional implementation of A*, VGHA is based on HPA+.

In both algorithms, the wavefronts are developed *independently* of each other. Each aims toward the opposite terminal node. A major problem with independent *informed* bidirectional search is that the wavefronts may not meet near the middle of the search space. This tends to occur when several comparably good paths exist and the focus of search differs in the forward and backward search perspectives. Consequently, the overall search effort may exceed that of a unidirectional informed search or even that of a bidirectional uninformed search. This phenomenon is often enough to rule out BHPA and VGHA as suitable for application. A further weakness of both is that their termination conditions are unnecessarily strong, as in the case of BSPA.

3.2.3.4 BHFFA and BHFFA2

Champeaux and Sint's (1977a, 1977b) bidirectional heuristic front-to-front algorithm (BHFFA) is the first implemented search algorithm in which the two search efforts are *mutually dependent*. Instead of evaluating the remaining path cost estimates (h) relative to the terminal (start/goal) nodes, the h estimates are computed relative to *all* the *open* nodes on the opposite wave front. The idea is that a shortest path is likely to be the pair of incomplete paths (one in the forward search tree and the other in the backward tree) which when bridged directly is the shortest among all bridged pairs. This approach is known as *wave-shaping* because using such h estimates to determine which node to expand next has the effect of drawing the wavefronts together.

Wave-shaping is computationally demanding as Champeaux and Sint (1977a) admitted: "the large disadvantage of our algorithm is the very time consuming calculation of the distance estimator". Besides having to compute for every open node as many h estimates as there are open nodes in the opposite wavefront, BHFFA has to recompute the order of merit of open nodes on both wavefronts each time a new open node is generated. As time-saving measures, Champeaux and Sint suggested restricting the h estimates of a node to a subset of more promising open nodes on the opposite wavefront, and to ignore the order of merit revision.

These measures obviously violate admissibility. However, contrary to the original claim, BHFFA is not admissible in the first place. The error of non-admissibility was later discovered and led to BHFFA2 (Champeaux, 1983)—a substantially modified version. BHFFA2 retains the expensive h estimate computation and maintenance of sort orders, but has a stronger terminating condition. Hence BHFFA2 can be expected to be even more computationally demanding.

Although a proof of admissibility is given for BHFFA2, I have found it to be unconvincing for the following reason. In BHFFA2, the remaining path cost estimate (h) is defined with the constraint that the remaining path must pass through a node on the opposite wavefront. Its value thus depends on the g value of such a node. Since g values of open nodes may be reduced by subsequent expansions, the h and hence f values of nodes may require updating. BHFFA omits a necessary case in its updating procedures: if $g_1(n)$ (or $g_2(n)$) is updated and n is closed in the opposite tree,[†] $h_2(n)$ (or $h_1(n)$) should be updated and n reconsidered for expansion in the opposite tree by changing its pointer, f_2 value, and placing n in the open list.

[†] The subscripts 1 and 2 denote the forward and backward search directions.

Whereas experimental data was presented for BHFFA, none was given for BHFFA2. However, BHFFA was only matched against Pohl's unidirectional HPA using the 15-puzzle domain and no running time data were presented. Although it was found that BHFFA generally solved more problems than HPA and produced shorter solution paths, the tests do not show how well BHFFA fared against BHPA in overcoming the main problem of wavefronts not meeting until each has covered a large part of the search space. We shall refer to this as the elusive wavefront problem. BHFFA's unconvincing and BHFFA2's unsubtantiated advantages, along with their confirmed disadvantage of being extremely time consuming, do not commend them for application.

3.2.3.5 D-node Retargetting

D-node retargetting (Politowski and Pohl, 1984) is a non-admissible wave-shaping bidirectional search algorithm based on Pohl's HPA. It avoids the computational extravagance of BHFFA and BHFFA2 by not exhaustively computing all relevant *h* estimates for the open nodes. Instead, the *h* estimates are only computed relative to a chosen node in the opposite tree. This chosen node (d-node) is the open node furthest from the root of the search tree. However, d-nodes are not always updated whenever a new open node is generated, since this would mean a revision of all *h* estimates on the opposite wavefront—a taxing procedure. Instead, the d-nodes are changed only after an arbitrary number of node expansions. In other words, the target nodes are initially the root nodes of the two search trees (as is always the case in non-wave shaping algorithms), but are periodically reset to the current furthest tip nodes of the search trees. We shall examine more closely the test results reported by the originators in section 3.3.3. Suffice it for now to note that d-node retargetting is non-admissible and hence does not meet our requirement.

3.3. Issues in Search Theory

3.3.1 Uninformed vs Informed

Uninformed searches such as Moore's and Dijkstra's algorithms do not make use of information suggesting how close a node is to the goal. An informed or heuristic search will make use of such information to guide the selection of the next node to expand. Which should be used depends on whether only one path is required or all paths from a node. It also depends on whether shortest paths are required and whether search is unidirectional or bidirectional.

If only one path is required, informed search generally finds it with less search effort. Whether it also takes a shorter time depends on how costly it is to compute the h estimates. But if the solution must also be optimal and the h estimates cannot be guaranteed not to exceed their true values, then uninformed algorithms are the only recourse. Uninformed search is unadventurous and always considers the closest node first. When the search space is large, uninformed search will be sooner defeated by the combinatorial explosion problem.

If all paths from a particular node are required, uninformed searches are more efficient. Informed searches need a specific goal node on which to base their estimates. When the goal node changes, the search tree has to be regenerated from scratch. In contrast, uninformed algorithms only need to grow the search tree once until all nodes in the problem graph have been expanded.

3.3.2 Unidirectional vs Bidirectional

When a particular path is required, it can be searched from one or two directions. Obviously, bidirectional search is only applicable when a specified goal state exists, implying only one path is required. Otherwise, search must be unidirectional.

There is little doubt that bidirectional search is better than unidirectional search when heuristics are not used. The same is not true when heuristics are involved, although there is the potential for even greater reduction in the number of nodes expanded. The problem preventing this potential from being realized is the elusive wavefront problem. In general, without wave-shaping techniques, bidirectional informed search algorithms (BHPA and VGHA) are disappointing compared with a unidirectional equivalent (A* or HPA). Nevertheless, they can be better than uninformed searches, both unidirectional and bidirectional. This is substantiated by the test results shown in Figure 3-1.

Another advantage common to any bidirectional search is that maintaining the sort order of the two smaller lists of open nodes is less expensive than that of one large list of a unidirectional search. This should go some way towards improving the running time.

3.3.3 Independent vs Wave-shaping

In the case of bidirectional informed search, the two wavefronts can be developed either independently or in a mutually dependent manner (wave-shaping). The latter is a common panacea to the elusive wavefront problem, but exacts an extreme penalty in running time. This is evident from test results in Politowski and Pohl (1984).

First, these test results revealed that d-node retargetting typically ran 10 to 20 times faster than BHFFA, in spite of time-saving pruning operations used in the BHFFA implementation. Second, its time overheads were somewhat higher than BHPA, in spite of expanding half as many nodes as BHPA. Politowski and Pohl did not say whether the stronger admissible terminating criterion of BHPA was used. Presumably, since d-node retargetting searches for non-admissible solutions, a cheaper non-admissible criterion for BHPA was used. If it is not so, the overheads of d-node retargetting compared to a non-admissible BHPA would be even higher. Third, the parameters for HPA in one of the tests meant that it was essentially A*, and for this test, d-node retargetting was more than 10 times slower than A*, and BHFFA was 20 times slower than d-node retargetting! These results also show that even with certain time saving steps—restricted computations of the *h* estimates in d-node retargetting; and staging in BHFFA—running times were nevertheless excessive. Hence when time is important and the greater memory costs can be afforded, the independent approach should be adopted.

The independent approach has another advantage over wave-shaping techniques. Noting that bidirectional search is amenable to twice as much parallelism in implementation as unidirectional search, the independent approach allows about 50% reduction of running time if full parallelism is exploited. The same cannot be said for wave-shaping since node expansions on different wavefronts cannot occur concurrently.

A variation of the independent approach is to postulate an intermediate node (Pohl, 1971; Chakarabarti *et al*, 1986) which is contained in a solution. This decomposes the problem into two subproblems which can then be solved more readily. Obviously, both can be solved unidirectionally or bidirectionally. However, it rests on the assumption that an island node can be identified *a priori*, and hopefully, it will be near the middle of the search space where the decomposition is most effective. In practice, this assumption is hard to satisfy, especially when an optimal solution is required.

3.3.4 Equidistance vs Cardinality Comparison Principle

In a uniprocessor implementation of a bidirectional search algorithm, a choice has to be made as to whether a node on the forward or backward wavefront should be expanded next. The *equidistance principle* selects the most promising open node. It works reasonably well only in uninformed search (Figure 3-1). If the searches are guided by heuristics, the cardinality comparison principle—choose from the smaller wavefront—works better (Pohl, 1969). An intuitive understanding of how it achieves its effect can be gained by noting that it strives to equate the sizesof the two sets of open nodes. This state is likely to exist when the two wavefronts meet ideally at the middle of the search space. Note however that in a dual-processor implementation, with two wavefronts developed concurrently, the cardinality comparison principle is irrelevant.

3.3.5 Terminating Conditions

The terminating condition for bidirectional *admissible* search is not as straightforward as in the non-admissible case. In the latter, termination occurs when the first complete path is found. Admissible search has to continue beyond this stage. The general idea is to keep track of the best complete path P found so far along with its cost k. If the minimum total cost estimate f from either direction is not less than k, then P is an optimal solution and search can be terminated. This is not obvious, but it has to do with the fact that, before termination, there is always a node in one of the wavefronts with an f estimate below the optimal path cost (for proof, see Pohl (1971)).

A common mistake made in BHPA and BHFFA2 is to restrict unnecessarily the updating of P to paths which have *all* nodes closed. This is probably due to Pohl's misinterpretation (see section 3.2.3.2 and appendix B) of the terminating condition in Nicholson's algorithm, an interpretation imported into BSPA and BHPA. It is also found in BHFFA2. This original discovery is important because it means that these algorithms can be terminated earlier if P is updatable by *any* complete path found—an improvement which is exploited in the new BS* algorithm described in section 3.4.2.1.

3.3.6 Admissibility and the Consistency Assumption

An admissible heuristic search algorithm always gives the optimal solution if any. One of the most significant contributions of Hart *et al* (1968) is the condition for admissibility—if the total path cost estimate $f(n)=g(n)+h(n)$, then admissibility is assured when h never exceeds its true value, h^*.

For bidirectional searches, admissibility requires a stronger condition—the *consistency assumption* (Hart *et al*, 1968). It states that *h* estimates consistently improve along any partial path developed by A*, or in more precise terms, the error of the *h* estimate must be monotonically non-increasing along these paths. For example, in route planning, an obvious $h(n)$ function which satisfies both the lower bound condition and the consistency assumption is the straight line distance between a node n and goal node. This condition for admissible bidirectional search is not evident because it only creeps out in the proof given for admissibility. The originators often cite the weaker condition of $h \leq h^*$ which tends to mislead the reader into thinking that this is all that is required.

There have since been two other minor variations of the consistency assumption, *viz.* monotone restriction (Nilsson, 1980) and monotone criterion (Pohl, 1977). These were proposed with the view that the consistency assumption was unnecessarily strong. Contrary to this, Kwa (1988) shows that the three terms are equivalent. Furthermore, all three guarantee admissibility *i.e.* the condition of $h \leq h^*$ is subsumed.

There are three important consequences of the consistency assumption:

- When A* terminates, not only is the shortest path from the start node to the goal node known, but also from the start node to all the other expanded nodes.

- A* and bidirectional algorithms based on A* can be simplified and made more efficient. Since *g* values of expanded nodes are true minima, any successor node generated which has already been expanded can be ignored. (This will be referred to as the *once-closed-always-closed* property.) This differs from the general A* algorithm which includes a procedure to propagate improved *g* values and a revision of parental pointers when a better route to a previously expanded node is found. The simplification cannot be applied to BHFFA2 however. The reason is that, in BHFFA2, the *h* values (and hence *f* values) of closed nodes can be reduced in subsequent expansions. In fact, this disadvantage is inherent in all wave-shaping algorithms since they all share the characteristic of variable h values.

- Prior to termination, the computed f values are monotonically non-decreasing and always less than the cost of the optimal path. This last observation may seem unimportant, but it can be gainfully exploited to constrain the search effort—a feature which does not appear in the existing algorithms. Use will be made of it in section 3.4.2 when the novel BS* algorithm is described. In the next chapter, its role in hastening termination in a modified A* which uses learnt solutions will be explained.

3.3.7 Optimality

The optimality of a search algorithm differs from the notion of admissibility. Whereas admissibility refers to an algorithm's assuredness of always yielding a minimal cost solution, optimality refers to the search effort involved—the fewer nodes expanded, the better. Hence, optimality as used in this sense refers to search *efficiency*, and the reader should not confuse it with the notion of a minimal cost criterion as is usually meant elsewhere.

The notion of optimality originated from Hart *et al* (1968) where it was shown that A* is optimal in the sense that it expanded no more nodes than any less informed heuristic algorithm K. By "less informed", it is meant that K's h estimate for any node is less than A*'s. The proof given was later shown to be erroneous by Gelperin (1977).

More definite conclusions were established by Dechter and Pearl (1982). They pointed out that the monotone restriction condition of A* only substantiated the superiority of some A* algorithms over other less informed A* algorithms. In other words, if A1* and A2* are two instances of the A* class of algorithms and A1* is more informed than A2* then A1* will not expand more nodes than A2*. This agrees with intuition. A* is also proved to be optimal in the class of best-first algorithms which also use h(n) as an underestimate of the true remaining path cost. These findings suggest that A* is a good basis for a heuristic bidirectional algorithm and the heuristic estimates should be as accurate as possible. The last point is also suggested by Pohl's theorem for HPA which proved that if the estimates are perfect, only nodes along the optimal path are expanded.

3.3.8 Summary so far

The following summarizes the main points made during the preceding discussion of search issues:

- If all the paths from one or more nodes are required, uninformed search fares better than informed search.
- If only one path is required, informed search is more space efficient.
- Heuristic search does less searching, at the expense of time in computing the h estimates. Often the time saved in searching a smaller space offsets the overhead time.
- Without the benefit of heuristics, bidirectional search is more efficient than unidirectional search. The same need not be true if the searches are informed.
- Bidirectional informed search has the potential to search a much smaller space than its unidirectional equivalent. However, this is rarely realized because of the elusive wavefront problem.
- The sort order of the lists of open nodes of a bidirectional search is more easily maintained than the single large list of a unidirectional search.
- Bidirectional search is amenable to greater parallelism in implementation than unidirectional search.
- Wave-shaping overcomes the elusive wavefront problem, but at the expense of greatly extending the running time.
- Despite time-saving measures which violate admissibility, wave-shaping remains computationally expensive.

- In bidirectional search, the independent approach allows concurrent node expansions on different wavefronts, whereas wave-shaping does not. The former can be expected to reduce the running time (compared to its unidirectional equivalent) by about 50%.
- The cardinality comparison principle is superior to the equidistance principle only when search is non-concurrent and heuristically guided.
- Termination conditions for admissible bidirectional search are more difficult to establish. Except for Nicholson's algorithm, admissible bidirectional algorithms have used an overly strong condition for updating the search cut-off parameter. Consequently, search termination is unnecessarily delayed.
- Contrary to earlier beliefs, the three terms—consistency assumption, monotone restriction and monotone criterion—are equivalent. Each also subsumes the condition that h underestimates the true remaining path cost.
- Using underestimated remaining path costs guarantees admissibility in A*. In the case of bidirectional search, the monotone restriction must be satisfied.
- The consistency assumption is useful: (1) the optimal path is known for all closed nodes; (2) the once-closed-always-closed property simplifies A*; and (3) monotonically nondecreasing f estimates along a path can be gainfully exploited.
- A* is optimal in the class of best-first search algorithms which use underestimates of the remaining path cost.
- The unidirectional basis for a heuristic bidirectional algorithm should depend on whether optimal solutions are required. If so, A* should be used, otherwise B'. This is because, in the admissible case, the consistency assumption must be satisfied and B' is appropriate only when this assumption does not hold.

3.4. BS*—A Novel Admissible Bidirectional Search Algorithm

BS* is a novel bidirectional search algorithm which is admissible, informed and develops its wavefronts independently. It has the distinctive feature of using staging operations (Nilsson, 1980) to contain the search trees and yet does not violate admissibility.

BS*'s motivation derives from certain key observations:

- The wave-shaping approach should be ruled out because it is computationally extravagant, and remains so even when time-saving staging operations are used. Moreover, these staging operations violate admissibility.
- Termination of admissible bidirectional search relies on a cut-off parameter which is dynamically updated throughout the search process. It was observed that current bidirectional algorithms do not make use of all opportunities to update this parameter. Consequently, earlier termination is often missed.
- None of these algorithms exploit the information arising from search to eliminate unpromising search avenues. It was observed that when information is pooled from both search trees, situations can be identified which allow staging operations to be performed without violating admissibility. This has several useful spin-offs: more accurate guidance of search control, early exposure of non-promising nodes and reduced book-keeping overheads.

From these observations, it seemed that an improved version of BHPA could be implemented which may overcome its earlier setback—too many node expansions due to the elusive wavefront problem. BS* is such an algorithm.

Section 3.4.1 details the notations used to describe BS*. Section 3.4.2 shows how several refinements can be made to BHPA, transforming it into BS*. (Henceforth, unless stated otherwise, BHPA refers to its admissible version.) Section 3.4.3 outlines the BS* algorithm. Formal proof of BS*'s admissibility is then presented in section 3.4.4. Section 3.4.5 summarizes the advantages of BS*.

3.4.1 Notations

s	start node.
t	goal node.
$\Gamma_1(n)$	successors of node n in the problem graph.
$\Gamma_2(n)$	parents of node n in the problem graph.
d	current search direction index; when search is in the forward direction $d=1$, and when in the backward direction, $d=2$.
d'	$3-d$; it is the index of the direction opposite to the current search direction.
ε	a small positive value.
$c_i(m,n)$	positive and finite cost of the direct arc from m to n if $i=1$, or from n to m if $i=2$.
$g_i^*(n)$	cost of the optimal path from s to n if $i=1$, or from n to t if $i=2$.
$h_i^*(n)$	cost of the optimal path from n to t if $i=1$, or from s to n if $i=2$.
$f_i^*(n)$	$g_i^*(n)+h_i^*(n)$; it is the cost of the optimal path from s to t constrained to contain n.
$g_i(n)$, $h_i(n)$	estimates of $g_i^*(n)$ and $h_i^*(n)$ respectively.
$f_i(n)$	$g_i(n)+h_i(n)$.
λ	cost of the optimal path from s to t.
L_{min}	cost of the best (least costly) complete path found so far linking s to t.
$TREE_1$ ($TREE_2$)	the forward (backward) search tree.
$OPEN_i$	the set of open nodes in $TREE_i$.
$\|OPEN_i\|$	number of nodes in $OPEN_i$.
$CLOSED_i$	the set of closed nodes in $TREE_i$.
$p_i(n)$	parent of node n in $TREE_i$.

Ωm	the set of nodes in $OPEN_{d'}$ which are descendants of m in $TREE_{d'}$.
MeetingNode	node where $TREE_1$ met $TREE_2$ and yielded the best complete path found so far.

3.4.2 Genesis of BS*

This section explains how several novel improvements transform BHPA into BS*. It is thus proper to begin with the salient features of BHPA which are amenable to improvement. For an outline of BHPA, see appendix C.

BHPA is basically a dual A* with a global variable amin set to record the cost of the best complete path passing through a node closed in both trees. BHPA attempts to update amin only when the node chosen for expansion happens to be closed in the opposite search tree. amin is tested for termination in each iteration of the select-node-for-expansion loop. Search terminates when amin is less than or equal to all of the *f* values of nodes in either open sets. Nodes placed in the open sets remain there until they are expanded.

3.4.2.1 Earlier Termination.

It may appear that since BHPA tests for termination after every round of expansion, there cannot be any modification to the algorithm which can hasten termination. However, termination depends on the value of amin; the faster amin approaches its minimal value, the earlier the algorithm terminates. An improvement is possible if hitherto neglected opportunities are used to update amin. These opportunities arise when a new complete path is found. They exist whenever a successor node is generated within the expansion loop and placed in the open set. They are not only more numerous but occur earlier than those used in BHPA *i.e.* they are more opportune. The suggested opportunities are valid because what matters is that amin records the best complete path found so far. Whether or not this path passes through a doubly closed node is immaterial. In fact, the suggested test subsumes the original amin-update test since the latter occurs after the former and cannot yield a better complete path which has not been discovered earlier by the former. The obvious advantage of earlier termination is a saving of further search effort.

3.4.2.2 Nipping and Pruning

In BHPA, when a node is selected for expansion it will definitely be expanded. This is not always necessary. If it is known that the optimal path passes through the selected node, or that the node cannot possibly be on the optimal path, then it makes no sense to expand the node. Instead, such a node should be merely closed and not expanded—a process called *nipping*.

Selected nodes (for expansion) which should be nipped are those which are already closed in the opposite search tree. There are two types of *closed* nodes: *good* and *bad*. Belonging to the good type are those nodes n which have $g_i(n) = g_i^*(n)$. All other closed nodes belong to the bad type. A nipped node can be either good or bad. In what follows, informal justifications will be given to show that nipping both types of nodes does not invalidate admissibility.

A consequence of the monotone restriction is that when a node is closed the optimal path to it is already known *i.e.* $g_i(n) = g_i^*(n)$. All closed nodes should therefore be good. This is true in A* and BHPA but not always true in BS* because of its unusual practice of discarding certain nodes from the open sets. Discarding nodes has the effect of foreclosing the optimal path to certain nodes which can only be found by searching beyond a discarded node. Consequently, when a node is closed the path to it may be suboptimal.

Good nipped nodes are those for which the optimal path constrained to pass through the node is already known. None of its descendants can improve on this path. Rightfully it should be nipped, but more should be done. A nipped node, being a closed node in the opposite tree, may have descendants already generated and existing in the opposite open set. These descendants are also useless and ought to be removed from the open set. The process of removing them is called *pruning*. Pruning is thus similar to retrospective nipping.

Bad nodes are due to nipping and pruning. Later, we shall see another cause—trimming. Since bad nodes by nature do not offer any possible improvement to the current best complete path, nipping bad nodes and pruning the descendants of bad nodes are not only innocuous but check further spawning of bad nodes. Note that BS* cannot distinguish bad nodes from the good ones. This does not matter since neither nipping nor pruning involving both node types violates admissibility.

3.4.2.3 Trimming

Trimming like pruning involves removal of certain nodes from the open sets. It too has the effect of foreclosing the discovery of optimal paths to some nodes. Nodes which should be trimmed are those with f values at least equal to L_{min}. Trimming opportunities arise whenever a better L_{min} value is found.

The justifications are tricky to establish. We need to consider three types of *open* nodes—*good*, *bad* and *possibly bad*. Good open nodes are those with $g_i(n)=g_i^*(n)$. Bad and possibly bad open nodes have $g_i(n)>g_i^*(n)$. What distinguishes the two is that a possibly bad open node has a chance of becoming a good open node before it is closed, whereas a bad open node will (if closed) become a bad closed node .

If an open node n is good and meets the trimming condition, the cost of an optimal path through it or any of its potential descendants cannot be less than L_{min}. Since L_{min}

cannot be improved by expanding *n*, the node should be trimmed away. If the node is bad, then it must be the case that some earlier nipping, pruning or trimming operation caused it to be bad, and its goodness was foreclosed when it was found that neither it nor its potential descendants could have led to an improvement. Trimming a bad node is thus justified. Finally, for the case of the possibly bad node, the fact that a chance remains for it to become good means that at least one of its contemporaries on the same open set may lead to a better path to it. If the possibly bad node remained and a better path to it is found, then its associated values would be updated. If it is removed, it will be reinserted when a better path emerges. Although trimming a possibly bad node is unnecessary, it does no harm.

BS* again is unable to distinguish the three categories of open nodes. The subtle point is that in trimming what it ought to trim, it could have trimmed what need not be trimmed. However, the latter event is safe since it does not foreclose finding the optimal path nor degrade the efficiency of the algorithm.

3.4.2.4 Screening

Screening is the process of placing in the open sets only those hitherto ungenerated successor nodes with f values below the L_{min} threshold. The justifications for trimming apply similarly here since screening is a form of trimming (think of it as pre-emptive trimming). The advantage of screening nodes is that the computational cost of insertion and subsequent trimming is avoided.

3.4.3 The BS* Algorithm

The improvements described in the preceding section can be implemented to give the following BS* algorithm:

Algorithm 3-1:

```
        procedure BS*($s$, $t$)

        /* Compute the shortest path from $s$ to $t$. */

1.      $L_{min} \leftarrow \infty$ ; $g_1(s) \leftarrow g_2(t) \leftarrow 0$; $f_1(s) \leftarrow f_2(t) \leftarrow h_1(s)$.

2.      Put $s$ in $OPEN_1$ and $t$ in $OPEN_2$.

3.      until $OPEN_1$ or $OPEN_2$ is empty do

            /* Determine the search direction index. */

4.          if $|OPEN_1| \leq |OPEN_2|$

5.              then $d \leftarrow 1$ else $d \leftarrow 2$

            endif

6.          $d' \leftarrow 3 - d$. /* Set the opposite search direction index. */

7.          Transfer node $m$ in $OPEN_d$ with the lowest $f_d$ value into $CLOSED_d$.

8.          if $m$ is closed in the opposite search tree $TREE_{d'}$

                then /* nip $m$ in $TREE_d$ and prune $TREE_{d'}$ */

9.                  • Close $m$ without expanding it.

10.                 • Identify the set $\Omega_m$ comprising nodes in $OPEN_{d'}$ which
                      are also descendants of $m$ in $TREE_{d'}$.

11.                 • Remove from $OPEN_{d'}$ those nodes which are members of
                      $\Omega_m$.

                else /* expand $m$ */

12.                 $TrimFlag \leftarrow false$

13.                 foreach $n$ in $\Gamma_d(m)$ which is not closed in $TREE_d$ do

14.                     $g \leftarrow g_d(m) + c_d(m,n)$; $f \leftarrow g + h_d(n)$

15.                     if $f < L_{min}$ and $n$ is not in $OPEN_d$

                            then /* insert $n$ into $OPEN_d$ */

16.                             • Place $n$ in $OPEN_d$

17.                             • $g_d(n) \leftarrow g$; $f_d(n) \leftarrow f$
```

```
18.                     • $p_d(n) \leftarrow m$. /* Set $n$'s parent pointer. */
19.                 elseif $f < L_{min}$ and $n$ is in $OPEN_d$ and $g < g_d(n)$
                    then /* update $n$ in $OPEN_d$ */
20.                     • $g_d(n) \leftarrow g$; $f_d(n) \leftarrow f$
21.                     • $p_d(n) \leftarrow m$
                endif
22.             if $n$ is in $TREE_{d'}$ and $g_1(n) + g_2(n) < L_{min}$
                    then /* update $L_{min}$ */
23.                     • $L_{min} \leftarrow g_1(n) + g_2(n)$; $MeetingNode \leftarrow n$
24.                     • $TrimFlag \leftarrow true$
                endif
            endforeach
25.         if $TrimFlag$
                then /* trim the open lists */
26.                 Remove from $OPEN_1$ and $OPEN_2$ those nodes with $f$
                    values $\geq L_{min}$ and which are not source nodes (for
                    $OPEN_1$ the source node is $s$; for $OPEN_2$ it is $t$).
            endif
        endif
    enduntil
27. if $L_{min} = \infty$
28.     then no path exists
29.     else the optimal path cost is $L_{min}$ and the optimal path can be
             determined by tracing the forward and backward parent pointers
             from $MeetingNode$.
    endif
    endprocedure
```

The preceding algorithm can be modified to maintain child pointers as well as parent pointers *i.e.* when a node is expanded, its list of successors is recorded. We may want to do this for faster identification of Ω_m (line 10). Otherwise, Ω_m can be found only by re-expanding the non-tip nodes of the subtree rooted at *m*; a more costly procedure. Although storing child pointers incurs memory overheads, the overheads may be incurred in the first place in some problem domains (*e.g.* route planning) where the problem graphs are represented using adjacency lists of child pointers (Aho *et al*, 1983, pp. 87-88). In any case, when child pointers are stored, the space complexity of BS* will be at least $O(|N|x|E|)$ where N and E are the sets of nodes and edges (respectively) defining the problem graph. Parent pointers alone give a lower bound of $O(|N|)$. Whether BS*'s space complexity is in fact $O(|N|x|E|)$ depends on the the data structures used for *OPENi, CLOSEDi, etc.*

3.4.4 Admissibility of BS*

Final validation of the admissibility of BS* has to rest on a formal proof which is presented here. It follows the A* proof outline in Nilsson (1980) but with substantial modifications. By admissibility, it is meant that when a path exists from *s* to *t* and the consistency assumption is satisfied, BS* will terminate with the optimal solution. The first step of the proof is to show that BS* will terminate when a path exists from *s* to *t*.

Lemma 3-1:

BS* terminates for finite graphs.

Proof:

In every iteration of the main loop (steps 3 to 26) of BS*, a node is removed from *OPENd* and a finite number of nodes is added to *OPENd* (nodes in the graph are assumed to have a finite number of adjacent nodes). Since closed nodes are never reopened, there will eventually be no more new nodes to be added to *OPEN*1 or *OPEN*2. *OPEN*1 or *OPEN*2 will eventually be empty and BS* then terminates.

Before proving that when a path exists from *s* to *t*, BS* will terminate even if the graph is infinite, we need to establish a few more lemmas.

Lemma 3-2:

s is always in *TREE*1.

Proof:

Node *s* did appear in *TREE*1 since it was initially placed in *OPEN*1 (step 2). Node *s* can only subsequently disappear from *TREE*1 when it is disowned by *TREE*1 in either of two events:

A. *s* was in *OPEN*1 with an ancestor node in *TREE*1 from which a pruning operation was performed (steps 10, 11).

B. *s* was trimmed away from *OPEN*1 (step 26).

Event A cannot occur since *s* has no ancestor in *TREE*1. Neither can event B occur since any trimming of *OPEN*1 ignores *s*. Node *s* thus remains in *TREE*1.

Lemma 3-3:

t is always in *TREE*2.

Proof:

Use a similar proof as for lemma 3-2 with *s*, *OPEN*1 and *TREE*1 replaced by *t*, *OPEN*2 and *TREE*2 respectively.

Lemma 3-4:

If *n*1 and *n*2 are both in *OPEN*1 and *n*1 as it stands offers a better path from *s* to *n*2, BS* will not expand *n*2 before *n*1.

Proof:

Node n_1 as it stands offers a better path to n_2 means that

$$g_1(n_1) + h_1^*(n_1,n_2) < g_1(n_2).$$

According to the consistency assumption,

$$h_1(n_1) - h_1(n_2) \leq h_1^*(n_1,n_2).$$

Hence $g_1(n_1) + h_1(n_1) \leq g_1(n_1) + h_1^*(n_1,n_2) + h_1(n_2)$

$$< g_1(n_2) + h_1(n_2).$$

Since $f_1(n_1) < f_1(n_2)$, BS* will select n_1 before n_2.

Corollary 3-1:

Along an optimal path P from s to t, if n_1 precedes n_2 along P and both are in *OPEN*1 and all of n_1's ancestors in *TREE*1 lie on P, BS* will not expand n_2 before n_1 when searching in the forward direction.

Lemma 3-5:

If n_1 and n_2 are both in *OPEN*2 and n_1 as it stands offers a better path from n_2 to t, BS* will not expand n_2 before n_1.

Proof:

Use a similar proof as for lemma 3-4 with appropriate changes.

Corollary 3-2:

Along an optimal path P from s to t, if n_2 precedes n_1 along P and both are in *OPEN*2 and all of n_1's ancestors in *TREE*2 lie on P, BS* will not expand n_2 before n_1 when searching in the backward direction.

Theorem 3-1:

Before BS* finds a complete path which is also an optimal path P, there exist n_i and n_j in *OPEN*1 and *OPEN*2 respectively such that n_i and n_j are along P.

Proof:

Let P_1 and P_2 be maximal subpaths of $P=(n_1=s, n_2, \dots, n_k=t)$ in $TREE_1$ and $TREE_2$ respectively.

$P_1=(n_1, n_2, \dots, n_i)$ and $P_2=(n_j, n_{j+1}, \dots, n_k)$.

P_1 and P_2 are not null paths (i.e. $1 \le i$ and $j \le k$) since n_1 and n_k are in $TREE_1$ and $TREE_2$ respectively (by lemmas 3-2 and 3-3).

Furthermore, $i < j$. Otherwise, the optimal path would have been found.

We wish to show that node n_i (n_j) is in $OPEN_1$ ($OPEN_2$) i.e. not closed.

Node n_i (n_j) is closed only when one of four events occurred:

A. Node n_i (n_j) was nipped in $TREE_1$ ($TREE_2$) (step 9).

B. Node n_i (n_j) has an ancestor node n_x (n_y) in $TREE_1$ ($TREE_2$) which was nipped in $TREE_2$ ($TREE_1$) (step 9).

C. Node n_i (n_j) had a successor in $TREE_1$ ($TREE_2$) and on P and the successor was trimmed from $OPEN_1$ ($OPEN_2$) (step 26). For example, P_1 was $(n_1, n_2, \dots, n_i, n_{i+1})$ with n_{i+1} in $OPEN_1$. A trimming event led to the removal of n_{i+1} from $OPEN_1$ (recall that trimming applies only to open nodes). Consequently, P_1 is now $(n_1, n_2, \dots, n_i)$ with n_i in $CLOSED_1$.

D. Node n_i (n_j) was expanded but had no successors which could be added to $OPEN_1$ ($OPEN_2$).

For event A to occur, n_i (n_j) must have been chosen for expansion in preference to n_j (n_i) while searching in the backward (forward) direction. Similarly, for event B to occur, n_x (n_y) must have been chosen for expansion in preference to n_j (n_i) while searching in the backward (forward) direction. According to corollaries 3-1 and 3-2 these events are impossible.

Suppose event C occurred i.e. node n_{i+1} (n_{j-1}) was trimmed from $OPEN_1$ ($OPEN_2$).

Then it must be that $f_1(n_{i+1}) \ge L_{min}$. [1]

Since n_{i+1} and its ancestors lie on the optimal path,

$$g_1(n_{i+1}) = g_1^*(n_{i+1}).$$

$$f_1(n_{i+1}) = g_1(n_{i+1}) + h_1(n_{i+1})$$

$$\leq g_1^*(n_{i+1}) + h_1^*(n_{i+1}) \text{ since } h_1(n_{i+1}) \leq h_1^*(n_{i+1})$$

$$= f_1^*(n_{i+1}) = \lambda.$$

Since BS* did not find an optimal path, $\lambda < L_{min}$.

Therefore $f_1(n_{i+1}) < L_{min}$, contradicting [1].

Likewise it can be shown that $f_2(n_{j-1}) < L_{min}$. Thus event C cannot occur.

Finally, to show that event D cannot occur, we note that the inequality $i<j\leq k$ implies that n_i (n_j) has at least one successor n_{i+1} (n_{j-1}) to consider when searching forward (backward). Since the best path to n_{i+1} (n_{j-1}) is via n_i (n_j), n_{i+1} (n_{j-1}) will be added to *OPEN*1 (*OPEN*2) should n_i be expanded, contradicting event D.

Since none of the events are possible, it follows that node n_i (n_j) is not closed and must be in *OPEN*1 (*OPEN*2).

Corollary 3-3:

Before BS* terminates, there exist nodes n_i and n_j in *OPEN*1 and *OPEN*2 respectively where $f_1(n_i) \leq \lambda < L_{min}$ and $f_2(n_j) \leq \lambda < L_{min}$.

Corollary 3-4:

If a path exists from s to t, BS* will not terminate before finding an optimal path.

Theorem 3-2:

If an optimal path exists, BS* will terminate.

Proof:

If the graph is finite, BS* will terminate according to lemma 3-1.

Suppose the graph is infinite.

Let $d^*(n)$ be the length (number of arcs) in the best partial path so far in *TREE*1.

n is the tip node of this path.

Since the cost of each arc is at least a small positive value ε,

$$g_1(n) \geq d^*(n)\,\varepsilon$$

$$f_1(n) = g_1(n) + h_1(n) \geq d^*(n)\,\varepsilon + h_1(n).$$

If BS* does not terminate, $d^*(n)$ will tend towards infinity and so will $f_1(n)$ since $h_1(n)$ is finite. Similarly, it can be shown that even the smallest of the f_2 values of nodes in $OPEN_2$ will tend to infinity. Since this contradicts corollary 3-3, BS* must terminate.

Theorem 3-3:

If a path exists from s to t, BS* will terminate with the optimal solution *i.e.* BS* is admissible.

Proof:

Corollary 3-4 tells us that BS* will not terminate with a suboptimal path. Theorem 3-2 says that BS* must terminate. Hence BS* must terminate with the optimal path *i.e.* $L_{min} = \lambda$.

3.4.5. Advantages of BS*

Even without the nipping, pruning, trimming and screening operations, the first suggested improvement of using all opportunities to update L_{min} will hasten termination. The four operations further expedite termination by eliminating unfruitful search whenever possible using information gathered during search. Bad open nodes due to these operations become more prone to trimming because of their inflated g and f values. This is certainly a desirable effect. The operations serve not only to avoid search being misled, but save unnecessary and burdensome additional book-keeping which would be incurred otherwise. Unlike BHPA, when L_{min} is updated BS* re-examines the potential of open nodes for contributing to a better solution, discarding those found worthless. In this way, BS*'s open sets are kept lean. Consequently, maintaining the order of the open sets is computationally less demanding. Perhaps more important is that the open sets should be able to guide search control more accurately using the cardinality comparison principle. This is apparent when BS*'s termination condition is examined. Trying to nullify the current smaller open set, which is what the cardinality comparison principle does, corresponds to the choice which is more likely to satisfy the termination condition (step 3).

A point made earlier is that bidirectional algorithms are more amenable to parallel computations and this enables its running time to be further reduced significantly. This is apparent when we examine the time intensive steps which can be concurrently executed. In a unidirectional search algorithm such as A*, only one node can be expanded at a time, but once the successor nodes are identified, the *f* values of these successors can be computed concurrently. In contrast, BS* can develop the two search trees concurrently *i.e.* the two most promising nodes, one from each search tree, can be expanded simultaneously. Furthermore, the *f* values of both sets of successors can also be evaluated in parallel. The obvious but telling point is that BS* allows more concurrent computations than A*, and if the number of processors is not a limiting factor, as in massively parallel architectures, then one can expect BS* to run significantly faster than A*. However, if A* is able to make full use of the processors available to BS*, then BS*'s speed gain is only significant when generation of successor nodes is more costly than computation of *f* values. Another point to note is that there are communication and shared variable access control overheads involved in concurrent computations. However, such overheads are highly machine dependent and their vitiating effects are unlikely to be serious in non-wave-shaping algorithms which permit both search trees to be developed independently.

3.5. Comparison of BS* Against Some Other Algorithms

Experiments were conducted to compare the performance of BS* against A* and other members of BS*'s class *viz.* Nicholson's algorithm, BSPA, and BHPA. All were coded in Interlisp-D on the Xerox 1186.

The test domain used was route planning since this is indeed a relevant application (*e.g.* in mobile robotics work) which would give a good indication of how the various algorithms would perform in a real world problem. Moreover, an admissible heuristic function which also satisfies the monotone restriction can be easily found in this domain. Graphs representing a network of geographically dispersed nodes were randomly generated by first generating the coordinates of n nodes and then randomly determining which of each node's neighbours are accessible from it. The number of nodes, n, ranged from 50 to 90 in increments of 10. Arc costs are the straight line distances between nodes. The heuristic function used gives the straight line distance between a node and the terminal node. For each graph, the $n \times (n-1)$ optimal paths for all distinct ordered node pairings were computed. The grand totals of node expansions were measured. In the case of the heuristically guided algorithms, the running times with garbage collection time excluded were also measured. These tests were selected to reflect overall performance *i.e.* the performance range over problems yielding the shortest to the longest optimal paths.

Figure 3-1 reveals that, on the average, heuristically guided search, both unidirectional and bidirectional, grew smaller search trees than the uninformed algorithms. BS*'s overall performance curve is only slightly inferior to A*'s, but it is a significant improvement on BHPA's—about 30% improvement for each of the five graphs. Figure 3-2 shows that the running times of the informed algorithms bear a similar relationship to their performance curves in Figure 3-1. The mean ratios of ordinates (with reference to A*'s ordinates) in Figures 3-1 and 3-2 are shown in Table 3-1. It shows that for an average 4% increase in node expansions over A*, BS*'s running time increased by only 8%. This compares well against BHPA's corresponding figures of 50% and 70%.

Finally, I compared the relative superiority of BS* and A* as a function of path length. For this, a 120-node graph was used. The 119x120 solutions were partitioned according to path length and for each partition, I examined the proportion of solutions for which one algorithm out-performed the other in terms of node expansions. For the sake of

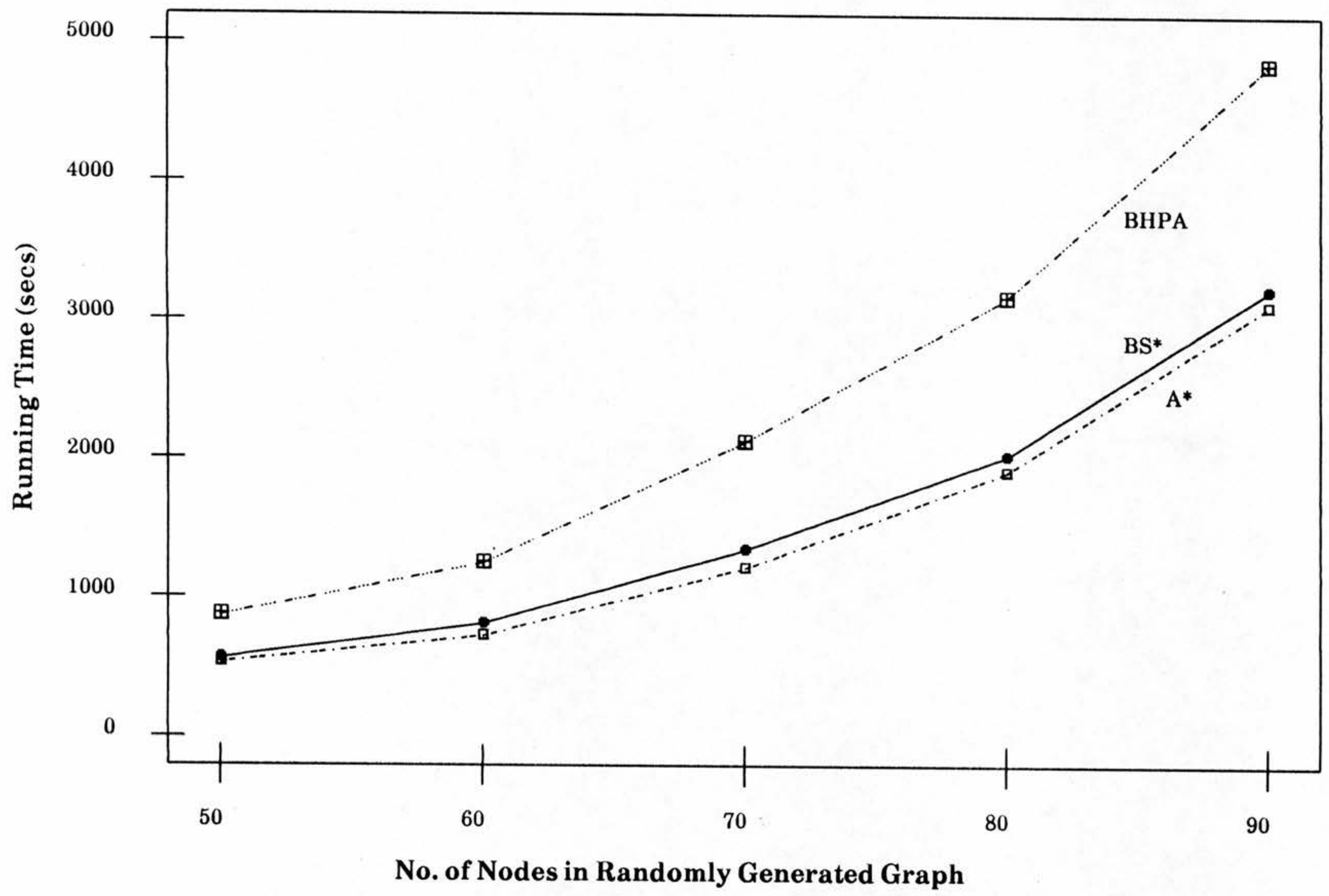


Figure 3-2. Running times of some admissible algorithms.

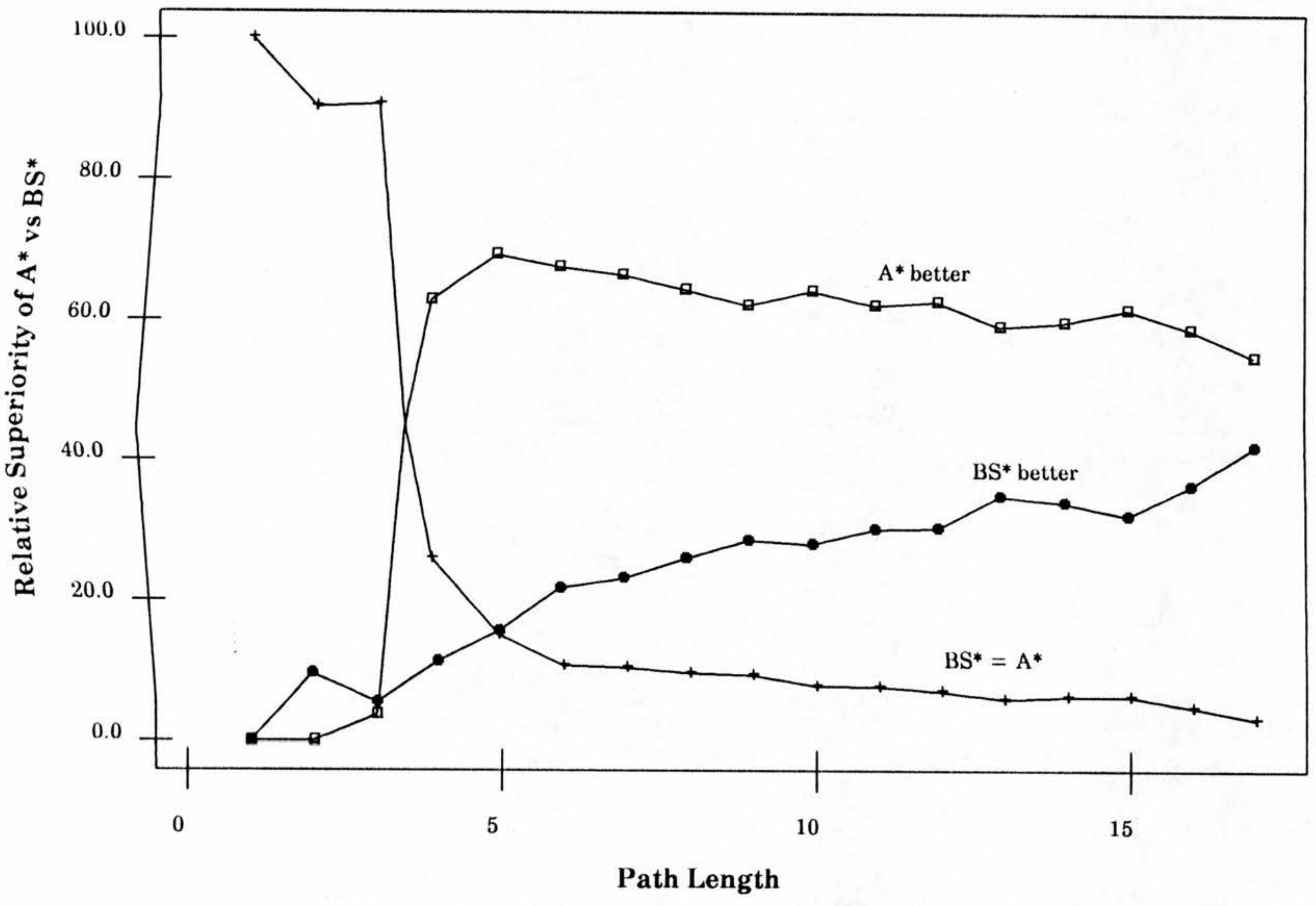


Figure 3-3. Relative superiority by percentage of solutions for various path lengths.

Table 3-1. Mean ratios with reference to A*.

	BHPA	BS*
Mean ratio of expansions	1.5	1.04
Mean ratio of running time	1.7	1.08

statistical accuracy, partitions with less than 50 members were ignored. The results plotted in Figure 3-3 show that even though A* is marginally better than BS* from an overall performance point of view, BS* does perform better than A* in some problem instances, and the frequency of such occurrences increases with solution path length. The same pattern was observed when the experiment was repeated with other graphs. A possible reason why BS*'s superiority does not dominate all the time is that the cardinality comparison principle, although efficacious, is nevertheless imperfect. The improving superiority of BS* with path length may be explained by the fact that unidirectional search algorithms, such as A*, are more vulnerable to the effects of combinatorial growth of search trees with increasing depth of search.

The experiments did not include comparisons with bidirectional algorithms of the wave-shaping class. Among these, BHFFA and d-node retargetting, being non-admissible, cannot be compared on an admissible basis. A comparison on a non-admissible basis would be uninteresting since a non-admissible version of BS* is similar to Pohl's non-admissible version of BHPA. As for BHFFA2, its admissibility is questionable. Even in its present form, it is unworthy of comparison because it would take more time than its predecessor (BHFFA) which itself has been found to be very computationally expensive despite certain time-saving staging operations.

3.6. Summary

Route planning is a frequent problem in AGV planning, occurring before and during execution. Its frequent occurrence, which at times needs to be solved in near real-time, warrants an admissible and efficient algorithm.

The shortest path problem can be solved more efficiently if unidirectional search is guided heuristically. Efficiency may be further improved by searching bidirectionally from the start and goal nodes. This latter approach was inspired by the superiority of bidirectional uninformed algorithms over their unidirectional counterparts. However, admissible bidirectional heuristic search algorithms are not generally superior. They either search too much of the search space or take far too much time figuring out which node to expand next.

From the review of previous algorithms, three key observations were made. First, it was evidently clear that wave-shaping and its variants are infeasible since the computational demands are extravagant. Second, it was noted that all the admissible bidirectional algorithms following Pohl's BSPA unknowingly adopted BSPA's unnecessarily strong terminating condition, which was apparently due to Pohl's misinterpretation of Nicholson's algorithm. Relaxing the termination condition achieves earlier termination and hence less searching. Third, and the most important, it was observed that information from both the search trees can suggest that certain nodes are unpromising and should not be developed further. This exploitation of information collected in the course of search did not occur in previous algorithms, and could also explain their tendency to explore a greater part of the search space than their unidirectional equivalents. These observations suggest that an independent approach for a bidirectional heuristic algorithm may yet surpass A*, at least for some of the time. Hence the idea of BS* was born. BS* was also motivated by the potential for significantly reducing the running time through parallel implementation.

BS* is essentially a turbo-BHPA. Its added power comes from using all opportunities to achieve early termination and from exploiting information available during search to eliminate unpromising nodes. In particular, it was shown how the search curtailment parameter (*L_{min}*) can be updated without missing any opportunity as previous algorithms did. Also, it was shown how the search trees can be curtailed such that unpromising branches are never grown.

BS* is by far the best (in terms of time and space) bidirectional admissible search algorithm in the non-wave-shaping class. It is also distinctive in being the first search algorithm which employs staging or search reduction operations (nipping, pruning, trimming and screening) and yet preserves admissibility.

In assessing the merit of BS*, I suggested as a possible criterion the maximum possible time reduction (implying multiprocessor usage) compared to a unidirectional implementation, and this time reduction is to be achieved without significantly expanding more nodes. Using this criterion, BS* has the potential to compute optimal solutions significantly faster than A*. Although this will incur on average a nominal penalty in memory requirement (about 4%), there will be instances when BS* will be superior both in time and space even without the benefit of concurrent computations. Experiments also indicate that such instances will occur with greater frequency as the solution path length increases.

Chapter 4

Route Learning

4.1. Introduction

In chapter 3, it was pointed out that route planning is a recurrent activity for the AGV movement planner and at times must be solved under near real-time constraints, implying that fast computation is essential. It was shown how BS*, an admissible bidirectional heuristic search algorithm, met this requirement. This chapter shows how another technique—route learning—can be incorporated into search algorithms to further reduce the time and effort it takes to compute a path.

Route learning or path learning is the process of extracting and remembering useful information found in the course of searching for a particular solution path. Routes learned can be used as instant solutions to future searches or, if a search is necessary, learnt routes may reduce the search effort involved. Either way, route learning can shorten the time it takes to find a path.

Our motivation for route learning is that it helps to achieve the objective of a fast AGV movement planner by: (1) avoiding redundant search; and (2) learning from experience to yield future solutions more readily. For example, if during a search for a path from A to Z, a path from C to K is incidentally uncovered and remembered, a fresh search is avoided if the latter path is subsequently required. For that matter, all paths uncovered should be remembered if there is no telling which is useful and which is not. To see how route learning can reduce search effort, consider the case when a path A to Z is required and the search has so far found a path from A to N. If a path from N to Z has been learnt from a previous search, then the search can terminate immediately. But if a shortest path is required, then the search must continue. Even so, the search effort can be reduced.

This chapter describes the novel use of route learning in graph searching in the following manner. First, I will show in section 4.2 that much can be learnt (or otherwise wasted) from the search tree grown in the course of looking for a particular path. If the graph is undirected, the amount of learning from a tree is double that in the directed case. The immense knowledge which can be extracted implies the need for representation using efficient data structures. This is the subject of section 4.3 where a representation structure of $O(n^2)$ space complexity is described for a graph of n nodes, along with path storage and retrieval procedures of $O(n)$ time complexity. Section 4.4 details the learning algorithms for extracting the information of all the subpaths embedded in a search tree. In section 4.5, we will see how the learnt paths can be used in both admissible and non-admissible search to produce solutions with less time and effort. BSL* (based on BS*), an algorithm which incorporates the learning algorithms, is also outlined. Section 4.6 presents empirical results comparing the performance of BSL* with BS*. The evidence shows that route learning is an effective technique in reducing the time and effort in a search problem.

4.2. Solutions Galore

When a graph is searched for a solution path, one or two search tree(s) is/are gradually grown, depending on whether the search is unidirectional or bidirectional. Much can be learnt from a search tree about solution paths linking other states (apart from the start and goal states) appearing in the tree. In fact, the tree embodies all solutions between states I and J if a path can be traced in it from I to J. The number of such incidental solutions embedded in a search tree can be enormous and it grows exponentially with the depth of the tree, more so than the number of nodes in it.

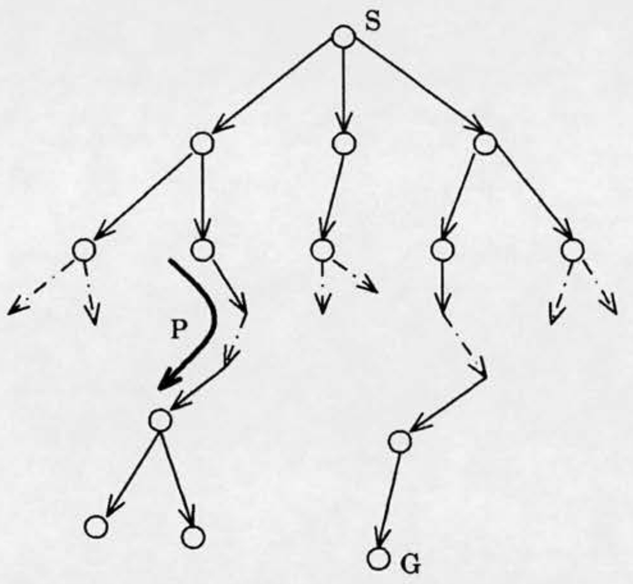


Figure 4-1. A search tree T and a typical embedded path P.

Consider for example, a directed graph which is searched for a solution path from the start state S to the goal state G. Without loss of generality, let us assume that non-optimal solutions are acceptable and a unidirectional search algorithm is used. Suppose the search tree grown is T (Figure 4-1). Normally, one is only interested in the solution path from S to G which T reveals; but in route learning, all paths embedded in T are of interest.

A typical path P embedded in T can be represented by the sequence $<n_0, n_1, \dots , n_k>$. Every subsequence of P is also an embedded path. It is easy to show that P contains $k(k+1)/2$ subsequences including itself. Since there are usually several root-to-tip paths (*i.e.* paths which begin at the root and end at a tip node) in T , it should be apparent that many subpaths are embedded in the tree. However, some subpaths appear in several root-to-tip paths as Figure 4-2 illustrates. This indicates that the total number of distinct subpaths is bounded below by $k'(k'+1)/2$ and above by $\Sigma_{i\in[1,q]}k_i(k_i+1)/2$ where q is the number of root-to-tip paths, k_i denotes the number of arcs in the i-th root-to-tip path and $k' = \min(k_1, k_2, \dots, k_q)$.

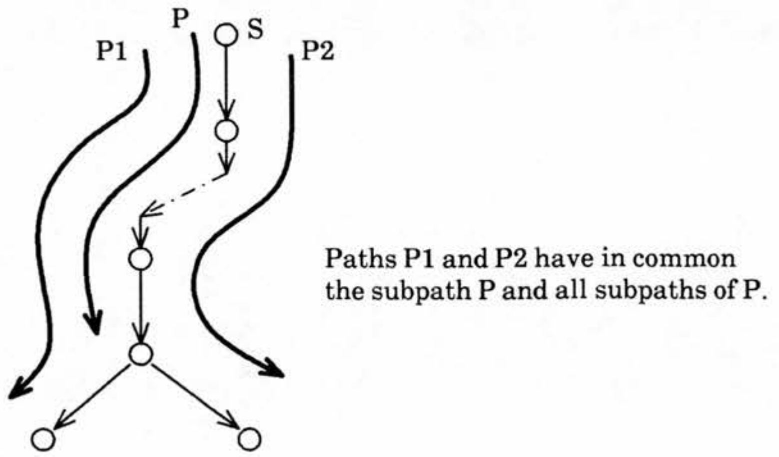

P1
P
S
P2
Paths P1 and P2 have in common the subpath P and all subpaths of P.

Figure 4-2. Common subpaths.

For an accurate quantitative appreciation of the total number of subpaths in T, suppose T has a constant branching factor B (*i.e.* every non-tip node has B immediate successors) and T is fully grown to a depth of n (*i.e.* all the tip nodes are at depth n). Let Σ_i ($i \leq n$) be the total number of distinct subpaths in the subtree T_i which include all nodes of T from the root node to the nodes at depth i. If all the subpaths in T_i have been learnt, and Δ_{i+1} more new paths can be learnt by extending T_i to T_{i+1}, then $\Sigma_{i+1} = \Sigma_i + \Delta_{i+1}$. Here is how Δ_{i+1} can be determined.

The number of nodes at depth $i+1$ is B^{i+1}. Thus there are B^{i+1} distinct root-to-tip paths which end at a node at depth $i+1$. Consider one such path $P = \langle n_0, n_1, \dots, n_{i+1} \rangle$. Any subpath of P which ends before depth $i+1$ is embedded in T_i. Since all subpaths embedded in T_i have been learnt, only the subpaths of P which end at n_{i+1} are new and should be gleaned. These are the subpaths $<n_x, n_{x+1}, ..., n_{i+1}>$ with $x=0, 1, ... , i$. Since P gives $i+1$ new subpaths, $\Delta_{i+1}=(i+1)B^{i+1}$. Σ_i can thus be computed from the series:

$$B+2B^2+3B^3+ ... +iB^i.$$

For an idea of the exponential growth of Σ_i, see Table 4-1 which shows the values of Σ_i for $B=2$ as the depth increases to 10.

Table 4-1. Exponential growth of Σ.

Depth	**Σ**
1	2
2	10
3	34
4	98
5	258
6	642
7	1538
8	3586
9	8194
10	18434

In general, a search tree will not be as regular as that depicted in our example to derive an expression for Σ_i. Not all the tip nodes will be at the same level and the branching factor varies from node to node. Nevertheless, we can expect exponential growth of the total number of subpaths as the tree grows.

The vast store of incidental solutions embedded in a search tree has two implications. First, route learning can be worthwhile if the effort to glean the incidental solutions is not forbiddingly taxing. Second, learnt paths should not be stored explicitly as a sequence (*e.g.* as a list) of states. An economical representation scheme which also permits quick path retrieval is required.

4.3. Efficient Data Structures

We need to record, for each path found, both its length and the sequence of nodes defining the path. This section shows how these can be recorded using matrices.

For a graph with nodes indexed 1, 2, ... , N, we can use a NxN matrix L to record the length information. L(i, j) stores the length of the path from node i to node j in the graph. If only the shortest paths are learnt and the graph is undirected, L(i, j) = L(j, i). In this case, L is symmetric and this allows a more efficient triangular matrix to be used.

The path sequences can also be stored using a NxN matrix Q. A naive method is to set Q(i, j) to point at the path sequence <i, ..., j> which is itself a data structure which consumes memory. A more efficient method (Aho *et al*, 1983) is to use a matrix of back pointers. In this method, Q(i, j) is set to the node preceding j in the path from i to j. For example, suppose N = 4 and the graph is directed; then the path P = <1, 3, 2, 4> can be recorded as shown in Figure 4-3 using Algorithm 4-1.

$$\begin{array}{c|c|c|c|c|}
 & 1 & 2 & 3 & 4 \\ \hline
1 & - & 3 & 1 & 2 \\ \hline
2 & & - & & \\ \hline
3 & & & - & \\ \hline
4 & & & & - \\ \hline
\end{array}$$

Row 1 of Q updated from path <1, 3, 2, 4>.

Figure 4-3. Matrix of back pointers (directed graph)

Algorithm 4-1.

```
       procedure StorePath(Path)
       /* Store the path sequence Path in the globally accessible matrix Q. The
          function pop returns the first node and sets its input variable to what is
          left of the sequence. */
       vars FirstNode, ToNode; /* local variables */
1.     FirstNode $\leftarrow$ pop(Path)
2.     StorePathAux(FirstNode, Path)
       endprocedure

       procedure StorePathAux(BPointer, Path)
3.     ToNode $\leftarrow$ pop(Path)
4.     Q(FirstNode, ToNode) $\leftarrow$ BPointer
5.     if Path is not the empty sequence
6.         then StorePathAux(ToNode, Path)
       endif
       endprocedure
```

The method of back pointers is more efficient because elements in **Q** store an integer value and the extra overhead of storing the corresponding sequences explicity as in the naive method is not incurred. Furthermore, in storing path $P = <n_0, n_1, \dots, n_k>$, we have also stored all subpaths of P which begin at n_0. Hence no additional effort and storage is necessary to store these subpaths of P as would be required with the naive method.

However, path retrieval is now more involved. The path from node 1 to 4 is retrieved by tracing the back pointers until the start node is found. The back pointers will be traced in the order 2, 3, and 1. Reversing this order and putting the terminal node (*i.e.* 4) at the end gives the path. Algorithm 4-2 gives a recursive path retrieval procedure. Clearly, the method of back pointers incurs more time to retrieve a path compared to the naive method. If the path has k nodes, k–1 matrix access operations are now needed instead of just one. Nevertheless, this time penalty is well worth the storage overhead saved.

Algorithm 4-2.

```
    procedure RetrievePath(FromNode, ToNode)
    /* Returns the path from FromNode to ToNode. */
    vars BPointer, Path; /* local variables */
1.  BPointer ← Q(FromNode, ToNode)
2.  if BPointer = FromNode
            then Return <FromNode>
    endif
3.  Path ← RetrievePath(FromNode, BPointer)
4.  Return the concatenation of Path and <ToNode>.
    endprocedure
```

	1	2	3	4
1	-	3	1	2
2		-		
3			-	
4	3	4	2	-

Rows 1 and 4 of Q updated from path <1, 3, 2, 4> and its reverse respectively.

Figure 4-4. Matrix of back pointers (undirected graph)

If the graph is undirected, any path found means that its reverse is also known. Hence we can also apply StorePath to the reverse of <1, 3, 2, 4> giving the matrix Q as shown in Figure 4-4. The number of learnt paths from the discovery of just one path has now doubled from three to six, showing the greater degree of route learning in undirected graphs.

Note that even if shortest paths are recorded in an undirected graph, $Q(i, j) \neq Q(j, i)$. Hence a more economical triangular matrix structure cannot be used for Q unlike the length matrix L.

Instead of storing the backward pointers, we could instead store the forward pointers. In this case, Q(i, j) indexes the node immediately after node i in the path sequence $P = <i, \dots, j>$. A forward pointer version of StorePath will record not only P, but all subpaths of P ending at j. Since there is no intrinsic advantage in doing so, we shall use the backward pointers in our algorithms.

4.4. Route Learning Algorithms

It seems that with the large number of paths embedded in a search tree, path extraction and learning will be a tedious and time consuming procedure. Indeed this is true if all the paths in the tree are enumerated and the StorePath procedure is applied to every path. Such an approach will incur an enormous number of matrix update operations, many of which will be redundant. A judicious and efficient learning strategy is required which avoids redundant learning. This section describes how such a strategy can be implemented both for unidirectional and bidirectional searches.

4.4.1 Enumerating the Set of Distinct Subpaths

The set of distinct subpaths in a tree T_k can be enumerated incrementally using a method similar to that described in section 4.2 to compute Σ (the number of distinct subpaths in a tree). The main idea is to rebuild T_k incrementally starting from T_0, the tree with just the root node n_0. This rebuilding process generates a series of trees T_0, T_1, ... , T_k such that T_{i+1} is built from T_i by adding a new node to T_i. As long as the node added is a successor of a tip node in T_i, T_{i+1} will be a subtree of T_k and the series will eventually terminate at T_k. Figure 4-5 illustrates this process.

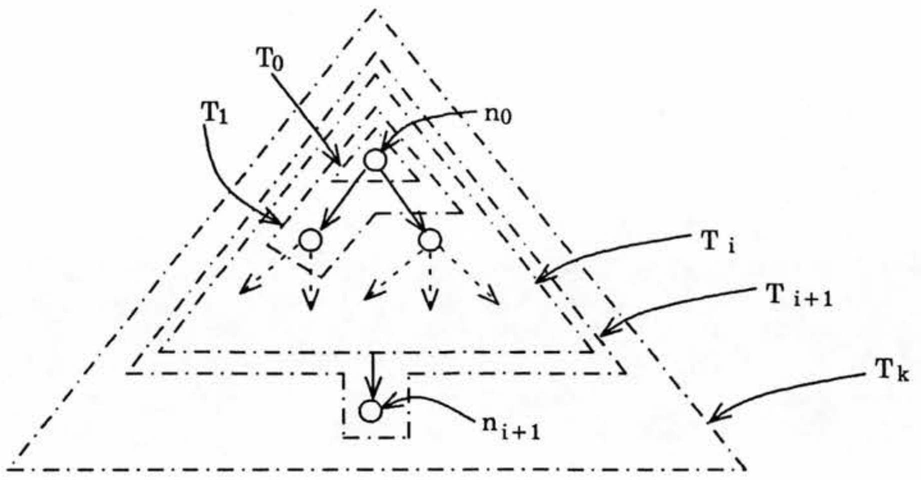


Figure 4-5. An incremental series of trees.

Suppose all the subpaths in T_i have been enumerated. The subpaths in the next subtree T_{i+1} and which are not in T_i are those which end at node n_{i+1}. The longest of these is $<n_0, n_1, \ldots, n_i, n_{i+1}>$ and the others are $<n_x, \ldots, n_{i+1}>$ $(x=1, 2, \ldots i)$. Altogether, $i+1$ new subpaths can be enumerated. These can be learnt starting from the longest and ending with the shortest, or in the reverse sense. We shall see (when the notion of learning cut-off is introduced) that it is advantageous to learn the longest subpath first.

Clearly, enumerating the new subpaths from T_{i+1} is straightforward once all the subpaths embedded in T_i have been enumerated. Since T_0 is the degenerate tree with no path in it, subpaths in T_1 can easily be enumerated. Having done this, T_2 is worked on next, and so on until T_k has been processed. In this way, all the distinct subpaths in T_k can be systematically enumerated and learnt.

4.4.2 Avoiding Redundant Learning

There are two ways in which redundant learning can occur. The first is when a path has been learnt during a previous search *i.e.* from a different search tree. Thus, even though the above enumeration procedure ensures that the next subpath (say P) identified has not already been enumerated from previous subtrees in the same series, it may be that P has been learnt already. If P is subjected to StorePath again, then one redundant matrix update operation will ensue for each arc in P. Obviously, we should first check whether P has been learnt before attempting to record it. A simple way to do this is to initialize all elements of the matrix Q to zero (or any integer not in the index set of the nodes) and if subsequently, $Q(i, j) > 0$, then the path from node i to node j has been learnt.

The second way redundant learning can occur is more subtle and recognising it allows a simplified and more efficient version of procedure StorePath. Whereas StorePath typically records several pointers, the simplified procedure merely records one. To see how this is possible, suppose $P = <n_x, \dots, n_{i+1}>$ is the next path enumerated when T_{i+1} is built. If P is subjected to StorePath, $i+1-x$ pointers will be recorded. But $i-x$ of these pointers correspond to subpaths $<n_x, \dots, n_z>$ ($z = x+1, x+2, \dots, i$). These subpaths are in T_i and have been learnt, implying that these $i-x$ pointers already exist in Q. This means that to learn P, all that is needed is to record one pointer *i.e.* set $Q(n_x, n_{i+1})$ to n_i. Hence, if learning proceeds chronologically according to the subpath enumeration procedure, a substantial time-saving measure is possible.

4.4.3 Learning Cut-off

We have seen that whenever a subtree T_{i+1} is built in the series, $\overset{+1}{\underset{\wedge}{i}}$ new subpaths can be learnt, either starting with the longest or with the shortest. Learning cut-off is the act of aborting the learning process before all the $\overset{+1}{\underset{\wedge}{i}}$ new subpaths have been recorded, yet without sacrificing any learning otherwise possible. In doing so, we save some effort in checking several subpaths. This is possible if there is a way to conclude at some stage of the learning process that the remaining subpaths have already been learnt. The clue to this is to observe that the strategy of learning as subpaths are enumerated has the property of total learning *i.e.* when a path has been previously learnt, so have all its subpaths. Couple this with the observation that among the $i+1$ new subpaths arising from T_{i+1} a subpath P_1 which is shorter than P_2 must be a subpath of P_2, and we see that if P_2 has been learnt, so have all its subpaths (*i.e.* those shorter than P_2). Hence if learning the i+1 new subpaths begins with the longest and works its way towards the shortest, and prior to recording a subpath, it is checked and found to have been learnt already, then the learning process associated with T_{i+1} can cease immediately.

4.4.4 Forward Learning

Here is how all that has been said fits together in the following learning procedure (FwdLearn). FwdLearn can be invoked whenever a node is closed (or opened for the first time) in the forward search tree (for example, the search tree of A* or $TREE_1$ of BS*). By using FwdLearn as an attached procedure (some refer to such a procedure as a demon) which is activated whenever a node is closed, we circumvent the need for a separate procedure to rebuild the search tree after it has been fully developed.

When learning is incorporated into BS*, there is another advantage in dynamic invocation of FwdLearn. Since only the optimal solutions are learnt, learning must be abandoned when the guarantee that a closed node has an optimal path to it from its root is invalidated. This invalidation happens when the first staging operation occurs. Identifying the moment of invalidation is easy during the search, but is far more complicated if the search trees are rebuilt after the search has terminated.

Algorithm 4-3.

```
    procedure FwdLearn(FromNode, ToNode, FromToDist, Pointer)
    /*  Learns the set of subpaths (of the path from FromNode to ToNode)
        which terminate at ToNode, beginning with the longest. Aborts when
        an intermediate subpath has been learnt already. Assumes
        FromNode ≠ ToNode. It is also assumed that elements of L were
        initialized to zero. Pointer is the node preceding ToNode along Path.  */
    vars Path; /* local variable */
1.  Set Path to the path from FromNode to ToNode.
    /*  The conditional part in the next line is the learning cut-off test.  */
```

```
2.  foreach N in Path until (Q(N, ToNode) > 0 or N = ToNode) do
3.      Q(N, ToNode) ← Pointer        /* this replaces the more involved
                                         StorePath procedure. */
4.      L(N, ToNode) ← FromToDist – L(FromNode, N)
    endforeach
    endprocedure
```

4.4.5. Backward Learning

When a bidirectional search algorithm is used, a backward search tree (*e.g.* *TREE*2 in BS*) is also generated. This provides another source for route learning. The backward learning algorithm (BwdLearn) is likewise an attached procedure which is activated whenever a node is closed (or opened for the first time) in *TREE*2. BwdLearn adopts the main points of FwdLearn—storing only one pointer per path; beginning with the longest subpath; and aborting learning when the cut-off condition is true. However, the inverted nature of the backward search tree calls for a different design.

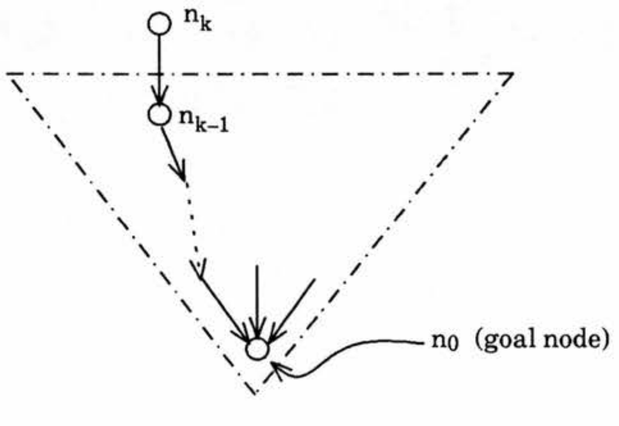


Figure 4-6. Extending the backward search tree by closing another node.

Figure 4-6 shows the state of *TREE*2 when node n_k is the next node closed. BwdLearn has to examine in turn, beginning with the longest, all the new subpaths in <n_k, n_{k-1},

... , n_0>. Unless the cut-off condition is true at some intermediate stage, k pointers will be stored corresponding to the subpaths <n_k, ... , n_x> (x=k–1, k–2, ... , 0). Unlike FwdLearn which repeatedly stores the same pointer (at different elements of Q), all these k pointers will be different. Implementing BwdLearn is thus less straightforward than FwdLearn. Nevertheless, a lucid implementation can take the following recursive form:

Algorithm 4-4.

```
       procedure BwdLearn(FromNode, GoalNode, FromtoGoalDist)
       /*  Learns the set of subpaths (of the path from FromNode to GoalNode)
           which begin at FromNode, starting with the longest. Aborts when an
           intermediate subpath has been learnt already. Assumes
           GoalNode ≠ FromNode. */
       vars Path, ToNode; /* local variables */
1.     Set Path to the reverse of the path from FromNode to GoalNode.
2.     ToNode ← pop(Path)
3.     BwdLearnAux(Path, ToNode)
       endprocedure

       procedure BwdLearnAux(Path, ToNode)
       vars Pointer; /* local variable */
4.     if Q(FromNode, ToNode) > 0
           then /* cut-off learning */ exit
       endif
5.     Pointer ← pop(Path)
6.     Q(FromNode, ToNode) ← Pointer
7.     L(FromNode, ToNode) ← FromtoGoalDist – L(ToNode, GoalNode)
8.     if |Path| > 1
           then    /* there are more new subpaths to consider */
```

```
9.                  BwdLearnAux(Path, Pointer)
        endif
        endprocedure
```

4.4.6 Final Learning

If a unidirectional search algorithm is used, then FwdLearn is the only algorithm which is needed. In the case of a bidirectional search, BwdLearn is also required. Even so, at the termination of search, there will still be opportunities for learning. This section shows how the final gleaning is performed.

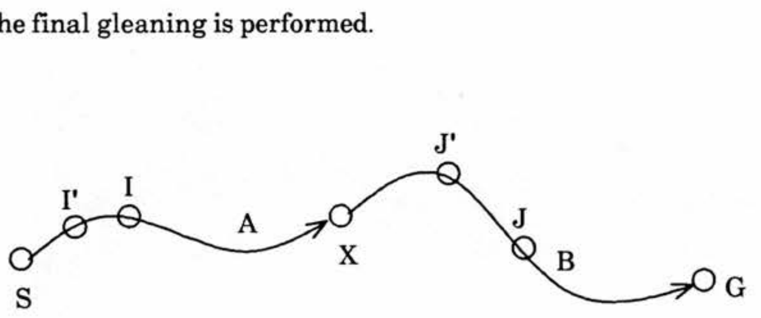


Figure 4-7. Solution path from S to G is P=A∪B.

Suppose the solution path found by a bidirectional search algorithm is P (see Figure 4-7). A conventional bidirectional search keeps track of the best complete path found thus far by recording its length and the node X at which the forward search tree (*TREE*1) met the backward search tree (*TREE*2) and produced the best complete path (see section 3.4.3). When search terminates, P is found by concatenating the path A (from S to X) in *TREE*1 with the path B (from X to G) in *TREE*2. Two observations can be made which suggest how further learning is possible.

First, node X is not necessarily closed in both search trees. This is a consequence of algorithms such as Nicholson's and BS* using all opportunities to find the best complete path. Other algorithms such as BHPA are more restrictive in this respect: the updating of the variables which represent the best complete path found thus far occurs only when X is closed in both trees. Another reason why X may not be closed becomes clear in section

4.5.2. There we see that X may not even be in one of the search trees because a learnt subpath (from a previous search tree) linked with a partial path (instead of two partial paths meeting) to yield the best complete path. Hence, following search termination, X should be checked to see if it is open in both search trees. If it is open in *TREE*1, then FwdLearn should be invoked. Likewise, BwdLearn should be invoked if it is open in *TREE*2.

Second, P presents more of its own subpaths which have not been considered previously because they are neither embedded in *TREE*1 nor *TREE*2. These are the subpaths which begin before X and terminate after X. They can be learnt simply by iterating along path B and applying FwdLearn repeatedly as shown in the following procedure:

Algorithm 4-5.

```
     procedure Glean(Start, XNode, Goal, StoGDist)
     /* Completes the learning process upon search termination. The first three
        input arguments correspond to S, X, and G in Figure 4-7. */
     vars PathB, Pointer, Distance;
1.   if XNode is not closed in TREE1 then FwdLearn(Start, XNode) endif
2.   if XNode is not closed in TREE2 then BwdLearn(Goal, XNode) endif
3.   PathB ← RetrievePath(XNode, Goal)
4.   Pointer ← pop(PathB)    /* popped node is XNode which has been learnt
                                and should be the pointer for the first node
                                which invokes FwdLearn. */
5.   foreach ToNode in PathB do
6.        Distance ← StoGDist – L(ToNode, Goal)
7.        FwdLearn(Start, ToNode, Distance, Pointer)
8.        Pointer ← ToNode
     endforeach
     endprocedure
```

A more complicated and efficient version of procedure Glean can be designed. What follows are the underlying ideas, but the implementation is left to the interested reader. Referring again to Figure 4-7, we see that FwdLearn can be invoked in the opposite direction (from G to X along path B) to that used above. This reverse direction happens to be advantageous because learning cut-off can be used to trim the path which FwdLearn has to work on. For example, if FwdLearn is invoked at node J, FwdLearn will first find the path from S to J and then proceed to learn all subpaths in this path. If learning is cut-off (for the first time in this final learning phase) at node I because it was found that the path from I to J has been learnt, then there is no need to learn any subpaths contained in <I, ... , X, ... , J>. Thus, in a later invocation of FwdLearn by node J' (lying between X and J along P), we can be sure that I will again be a cut-off point if FwdLearn has not been cut-off before I. This suggests that FwdLearn should use the shorter path from S to I instead of the longer path from S to J' as it normally would. Obviously, this requires that the cut-off node which is found nearest to S be recorded.

The procedure Glean applies only when learning is not abandoned because of staging operations which invalidate the condition of closed nodes having optimal paths. This point can be understood more clearly with an example. Referring to Figure 4-7 again, suppose the first staging operation was performed at node I and the final optimal path *via* X is due to a partial path A and a learnt path B. Since learning was abandoned at I, the path from I to X and its subpaths may not be learnt. Applying Glean as above would ensure that all paths beginning in A and ending in B are learnt. But our learning algorithms are dependent on the total learning property which is now violated because of some unlearnt subpaths between I and X. Unless procedure Glean is modified to preserve the total learning property, the book-keeping will be erroneous. Essentially, the modification of Glean is to ensure that all partial paths in the final path are totally learnt before the original steps in Glean are applied.

4.4.7 When Learning Should Occur

Dynamic application of FwdLearn and BwdLearn (*i.e.* learn in unison with search) can occur in two situations: (1) when a node is closed; and (2) when a node is opened for the first time. The choice depends on whether learning is confined strictly to optimal paths.

If only the optimal paths are learnt, then learning should be invoked when a node is closed, but on condition that the consistency assumption holds. This condition is necessary because only then are the paths from the root of the search tree to the closed nodes guaranteed to be optimal. Otherwise, only the final solution is guaranteed to be optimal (assuming that an admissible algorithm is used). In this case, dynamic learning cannot be used and any learning of optimal paths must be based on the final solution path alone. Obviously, this occurs after the search has ended and the amount of learning is also severely restricted.

If learning suboptimal paths is acceptable, then dynamic learning should be activated when a node is opened for the first time. The idea is to learn a new path whenever possible, and if a path is found to be better than one which has been learnt, then a decision has to be made on whether relearning should follow. However, relearning is more complicated than it seems. For example, suppose paths A and B have been learnt and A is a subpath of B, and if a better path linking the terminal nodes of A is found, then A and some of its subpaths can be revised, along with their associated cost values. Unfortunately, these changes will invalidate the costs of B and its subpaths which are dependent on A. Extending the revisions to the affected paths in B is a non-trivial matter. The implication is that relearning may not be worthwhile.

4.5. Application to Search Theory

The previous section described how routes/paths can be learnt during a search process. In this section, we will see how learning can be used to reduce the time and effort in finding a solution.

The most obvious way learning helps is when the search algorithm which is required to find a path (say from node S to another node G) can be preempted because the path has been learnt already. This occurs when $Q(S, G) > 0$, implying that the path exists in memory. All that is required is to invoke RetrievePath and return the solution path along with its cost in the L matrix.

Even when the search is not preempted because an instant solution does not exist, learnt solutions can nevertheless be useful. Besides the preemptive role, learnt paths can complete partial paths in the search tree and enable earlier search termination. This second role is similar to the use of two partial paths, one from each tree of a bidirectional search, to find complete paths. Hence when learning is applied, there are two ways of forming complete paths—using partial paths only; and using a learnt path as well. Whichever is used, the best of these complete paths is maintained and its cost acts as a search cut-off parameter. An example of a search cut-off parameter is L_{min} in BS* (section 3.4). How learnt paths expedite search cut-off depends on the modes of search—directionality and admissibility. Since each of these has two possibilities, four combinations exist. For the two non-admissible combinations, search cut-off is immediate once a learnt subpath is available to link up with the partial path(s) to yield a complete path. But when optimal solutions are required, the search cut-off condition has to be more carefully considered. We shall examine this condition in detail in the following two subsections.

4.5.1 Unidirectional Admissible Search

Examples of unidirectional admissible search algorithms are A*, B, B' and Dijkstra's best-first search (section 3.2.2). The discussion in this section will be based on A* and its variants (B and B') since these are more general—setting $h = 0$ reduces these algorithms to the Dijkstra uninformed algorithm.

In the conventional application of A*, A* will expand several nodes sequentially until the goal node is chosen for expansion, at which point the optimal solution is known and A* terminates. Suppose the sequence of nodes expanded (or chosen for expansion) is $<n_0, n_1, \dots, n_k>$. If the consistency assumption holds (section 3.3.6), then $f(n_{i+1}) \geq f(n_i)$ for $i = 0, 1, \dots, k-1$. Furthermore, $f(n_k) = f^*(n_k)$. The proofs for these statements can be found in Nilsson (1980).

If learnt routes are used to cut-off search, we would maintain a parameter such as L_{min} to record the length of the best complete path found so far. Suppose node n_i is expanded and looking up $Q(n_i, n_k)$ tells us that the optimal path from n_i to n_k has been learnt, thus completing the partial path ending at n_i. If this new complete path gives a better value for L_{min}, then L_{min} is updated with $g(n_i) + L(n_i, n_k)$. If subsequently, node n_j is chosen for expansion and $f(n_j) \geq L_{min}$, then by transitivity, we can predict that the f values of the nodes after n_j in the above sequence, will also be at least equal to L_{min}, implying $f^*(n_k) \geq L_{min}$. Hence the complete path found by linking up with the learnt subpath at n_i cannot be inferior to the optimal path which A* would conventionally find. This complete path is thus the optimal path and A* can abort the search prematurely. Unfortunately, the condition $f(n_j) \geq L_{min}$ rarely holds before n_k is chosen. This is because f, in this case, is based on h being admissible (*i.e.* $h \leq h^*$) and unless $h(n) = h^*(n)$ at some intermediate node n, $f(n)$ will be less than $f^*(n)$ (which is the cost of the optimal path). Since $L_{min} \geq f^*(n)$ by definition, $f(n)$ will be less than L_{min} and cut-off will not occur.

What if the consistency assumption does not hold? In this case, the *f* values of the nodes in the expansion sequence is not necessarily monotonically nondecreasing. The cut-off condition in the preceding paragraph cannot guarantee an optimal solution. In fact, learning optimal solutions from the search tree has to be severely restricted to the final solution path. As explained in section 4.4.7, the FwdLearn procedure cannot be invoked when a node is closed because there is now no guarantee that when a node is closed, its optimal path has been found.

Although learnt subpaths cannot play a useful cut-off role when the consistency assumption does not hold, they can nevertheless play another role which has the same effect of expediting the discovery of an optimal solution. For this, we must turn to algorithms B and B'. Recall that these algorithms maintain a global variable F which records the maximal f value of the nodes which have been closed. F is used to identify the set of open nodes which definitely have inconsistent h estimates and thus are more likely to mislead the search. Nodes in this set are considered for expansion first, and only when this set is empty will the open nodes with consistent h estimates be considered. If L_{min} is maintained as described, a node with a consistent h estimate will, when chosen for expansion, have an f value below L_{min}. Conversely, if its f value is above L_{min}, then we can surmise that its h estimate is inconsistent. In this case, it should likewise be relegated to the set of first resort. By helping to identify the candidate nodes with unreliable ratings, there should be less incidents of search being misled. Effectively, the optimal solution can be found with less effort.

Summarizing, there is not much opportunity to apply learning to reduce search in A*. The main benefit of learning is in preempting search by instant look-up solutions. When the consistency assumption does not hold, learning can nevertheless reduce the search effort of algorithms B and B'. However, the opportunities for learning are severely curtailed.

4.5.2 Bidirectional Admissible Search

Unlike unidirectional admissible searches, in bidirectional admissible searches we only need to consider the case when the consistency assumption holds. This is because informed bidirectional admissible search requires the consistency assumption to be satisfied. It is easy to show how learning can help to cut-off search.

All bidirectional admissible algorithms maintain a form of cut-off parameter in the first place. For example, BHPA uses a_{min} and in BS*, it is L_{min}. Learnt subpaths which can complete the partial paths in the search trees can be used to tighten the cut-off parameter and thereby enable earlier termination. There are two ways partial paths can be linked into a complete path using learnt subpaths.

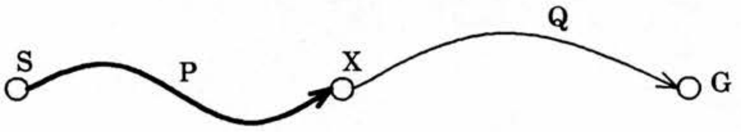


Figure 4-8. Completing a forward partial path.

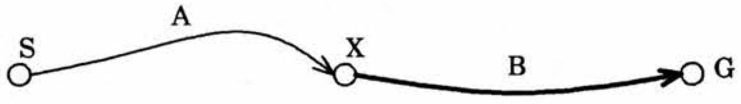


Figure 4-9. Completing a backward partial path.

First (see Figure 4-8), either a forward partial path P is linked with a learnt path Q from node X of P to the goal node G, or (see Figure 4-9) a backward partial path B is linked with a learnt path A from the start node S to node X of B.

Second (see Figure 4-10), a learnt subpath Q can bridge the non-terminal end nodes X and X' of partial paths P and P' in the forward and backward search trees. Finding a learnt path Q in this case is computationally expensive since the algorithm must inspect, whenever a new open node is generated, all the nodes in the opposite search tree for possible link-ups. This overhead increases exponentially as the search trees grow and easily adds more time than can be saved by the technique of search cut-off.

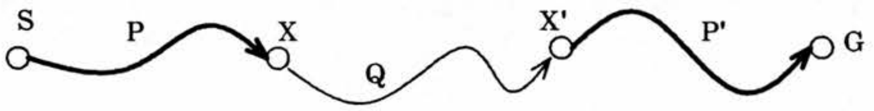


Figure 4-10. Bridging a forward and a backward partial path.

In contrast, the first method requires only one inspection for every new open node and is likely to save more time than the time added by its overhead. Hence only this method should be used.

Here is how BS* can be modified to learn by experience and to use learnt information to help find a solution:

Algorithm 4-6:

```
    procedure BSL*(s, t)
    /* Compute the shortest path from s to t. All variables except matrices Q and
        L are local. */
1.  if Q(s, t) > 0
        then    /* path has been learnt; preempt search. */
2.                  return RetrievePath(s, t) and L(s, t); exit
    endif
3.  ShouldLearn ← true
```

```
4.  $L_{min} \leftarrow \infty$ ; $g_1(s) \leftarrow g_2(t) \leftarrow 0$; $f_1(s) \leftarrow f_2(t) \leftarrow h_1(s)$
5.  Put $s$ in $OPEN_1$ and $t$ in $OPEN_2$.
6.  until $OPEN_1$ or $OPEN_2$ is empty do
        /* Determine the search direction index using the cardinality
           comparison principle. */
7.      if $|OPEN_1| \le |OPEN_2|$
8.          then $d \leftarrow 1$ else $d \leftarrow 2$
        endif
9.      $d' \leftarrow 3 - d$. /* Set the opposite search direction index. */
10.     Transfer node $m$ in $OPEN_d$ with the lowest $f_d$ value into $CLOSED_d$.
11.     if ShouldLearn
12.         then Apply the forward or backward learning procedures
                 accordingly
        endif
13.     if $m$ is closed in the opposite search tree $TREE_{d'}$
            then /* nip $m$ in $TREE_d$ and prune $TREE_{d'}$ */
14.              • Close $m$ without expanding it.
15.              • Identify the set $\Omega_m$ comprising nodes in $OPEN_{d'}$ which
                   are also descendants of $m$ in $TREE_{d'}$.
16.              • Remove from $OPEN_{d'}$ those nodes which are members of
                   $\Omega_m$.
17.              • ShouldLearn $\leftarrow$ false
            else /* expand $m$ */
18.              TrimFlag $\leftarrow$ false
19.              foreach $n$ in $\Gamma_d(m)$ which is not closed in $TREE_d$ do
20.                  $g \leftarrow g_d(m) + c_d(m,n)$; $f \leftarrow g + h_d(n)$;
21.                  if $f < L_{min}$ and $n$ is not in $OPEN_d$
```

```
                then /* insert n into OPENd */
22.                 • Place $n$ in $OPEN_d$
23.                 • $g_d(n) \leftarrow g; f_d(n) \leftarrow f; p_d(n) \leftarrow m$
24.                 • LearntLinkUp?( )
25.             elseif $f < L_{min}$ and $n$ is in $OPEN_d$ and $g < g_d(n)$
                then /* update n in OPENd */
26.                 • $g_d(n) \leftarrow g; f_d(n) \leftarrow f; p_d(n) \leftarrow m$
27.                 • LearntLinkUp?( )
                else /* n is screened */
28.                 • ShouldLearn $\leftarrow$ false
              endif
29.           if $n$ is in $TREE_{d'}$ and $g_1(n) + g_2(n) < L_{min}$
                then /* update and set TrimFlag */
30.                 • Update $L_{min}$ and MeetingNode
31.                 • TrimFlag $\leftarrow$ true
32.                 • ShouldLearn $\leftarrow$ false
              endif
                /* Trim open sets if a better Lmin was found */
            endforeach
33.         if TrimFlag
34.           then  Remove from $OPEN_1$ and $OPEN_2$ those nodes with
                    $f$ values $\geq L_{min}$ and are not source nodes (for
                    $OPEN_1$ the source node is $s$; for $OPEN_2$ it is $t$).
            endif
        endif
    enduntil
35. if $L_{min} = \infty$
```

```
36.         then no path exists
            else
37.               • Apply the modified Glean to learn remaining subpaths from
                    the solution path
38.               • Return RetrievePath(s, t) and L(s, t)
        endif
        endprocedure


        procedure LearntLinkUp?( )
39.     if learnt subpath exists and yields a better complete path
            then
40.               • Update $L_{min}$ and MeetingNode
41.               • TrimFlag ← true
        endif
        endprocedure
```

Note that in BSL*, learning is permitted as long as no staging operations (nipping, pruning, trimming and screening) have been performed. This is because once a staging operation occurs, there is no longer a guarantee that when a node is closed, its optimal path with respect to its root node is known.

BSL* remains admissible because the proof of BS* applies similarly. The reason for this is threefold: (1) the application of learnt subpaths affects the search only *via* L_{min}; (2) essentially, the definition of L_{min} remains unchanged; and (3) as far as the proof is concerned, L_{min} records the cost of the best complete path found so far and it is irrelevant whether this is due to learning or due solely to the algorithm's search information.

4.5.3 Related Work

There is little previous work on applying learning to search theory. The earliest scheme can be found in Michie and Ross (1969) which described GT4, an adaptive version of Doran's GT2 (1968).† GT4 implements learning from experience by modifying the numerical parameters used in the heuristic evaluation function, and by re-ordering the list of operators used to develop the search tree. The experimental results reported (based on the 8- and 15-puzzle domains) suggest the effectiveness of these two techniques in improving search performance; more problems were solved and with less effort involved. However, the methods are not applicable to admissible search and may not lead to an improvement in computation time.

The only other reference I have come across is Pohl's (1969) suggestion that the weights w_i in the general evaluation function $f(n)=g(n)+\Sigma_i w_i h_i(n)$ be tuned dynamically according to some learning strategy. However, Pohl did not describe any learning scheme. This learning approach differs from the concept of route learning: learning is used only to alter the evaluation function whereas in route learning, the f function is not modified. Instead of honing the guidance mechanism *via* the f function, the method of route learning uses previously discovered solutions explicitly to preempt $\overset{\text{search}}{\wedge}$ or prune the search space.

4.6. Empirical Results

In this section, empirical evidence is presented to support the claim that route learning can expedite the discovery of optimal solutions using a bidirectional admissible search algorithm. We will see a reduction in the time and effort it takes to find solutions as learning increases. After some initial learning, subsequent computational time is less

† Doran's GT2 differs from the original Graph Traverser (Doran and Michie, 1966; Doran, 1967) by partially expanding/developing selected nodes instead of fully expanding them. GT2 was written in Algol. GT3 (Marsh, 1969) is merely a POP-2 version of GT2.

than if no learning is involved.

The results presented are based on the mean data of 20 repetitions of an experiment (run with BS* and BSL*), each using a different set of 1000 pairs of start and goal nodes. Each set was randomly generated and referred to a 90-node graph also randomly generated. At the beginning of each experiment, the L and Q matrices were initialized *i.e.* nothing had been learnt. For BS*, the experiment measured two parameters T and E. For BSL*, two additional parameters were measured: I and L. These parameters are defined below and are in relation to the i-th lot (comprising 50 paths) for i = 1, 2, ... , 20.

T	computation time
E	number of nodes expanded
I	number of instant solutions from previous learning
L	number of paths learnt.

Figure 4-11 shows the learning curve as a percentage of the total learning possible. Since the generated graph is such that any two distinct nodes are connected, there are altogether 89x90 routes to learn. The learning curve shows diminishing marginal increase as lot number increases. The reason is that the probability of discovering (by chance) an unlearnt route decreases as more routes are learnt.

Figure 4-12 shows the time BSL* takes for a lot as a percentage of BS*'s time. The unbroken curve for BSL* shows that the computation time per lot decreases as the lot number increases (*i.e.* as more paths are learnt). Since the time reduction is due to two factors—instant solutions and more effective search cut-off—a second plot (the dashed BSL* curve) is given to show just the effect of the second factor. For this, the timings due to non-instant solutions were estimated by subtracting from T the time due to instant solutions, and normalizing the result for a similar lot size (*i.e.* if T' is the time for I instant solutions in the lot, then the normalized time is $(T-T')\text{x}50/(50-I)$). A similar trend as the un-normalized curve is observed, showing that learning does hasten search termination. Note that in both BSL* plots, the time initially exceeded BS*'s time. This

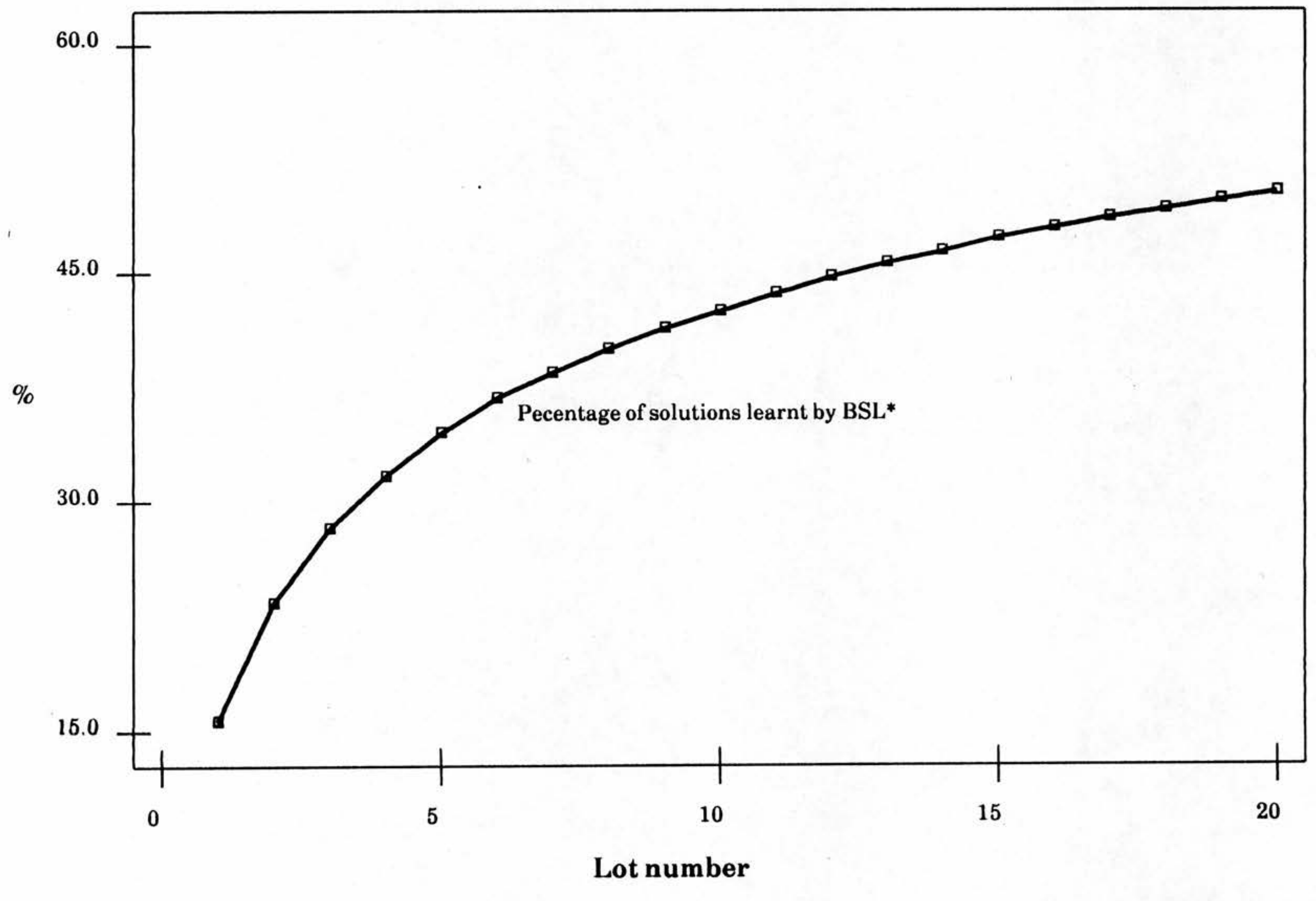


Figure 4-11. Learning curve of BSL*.

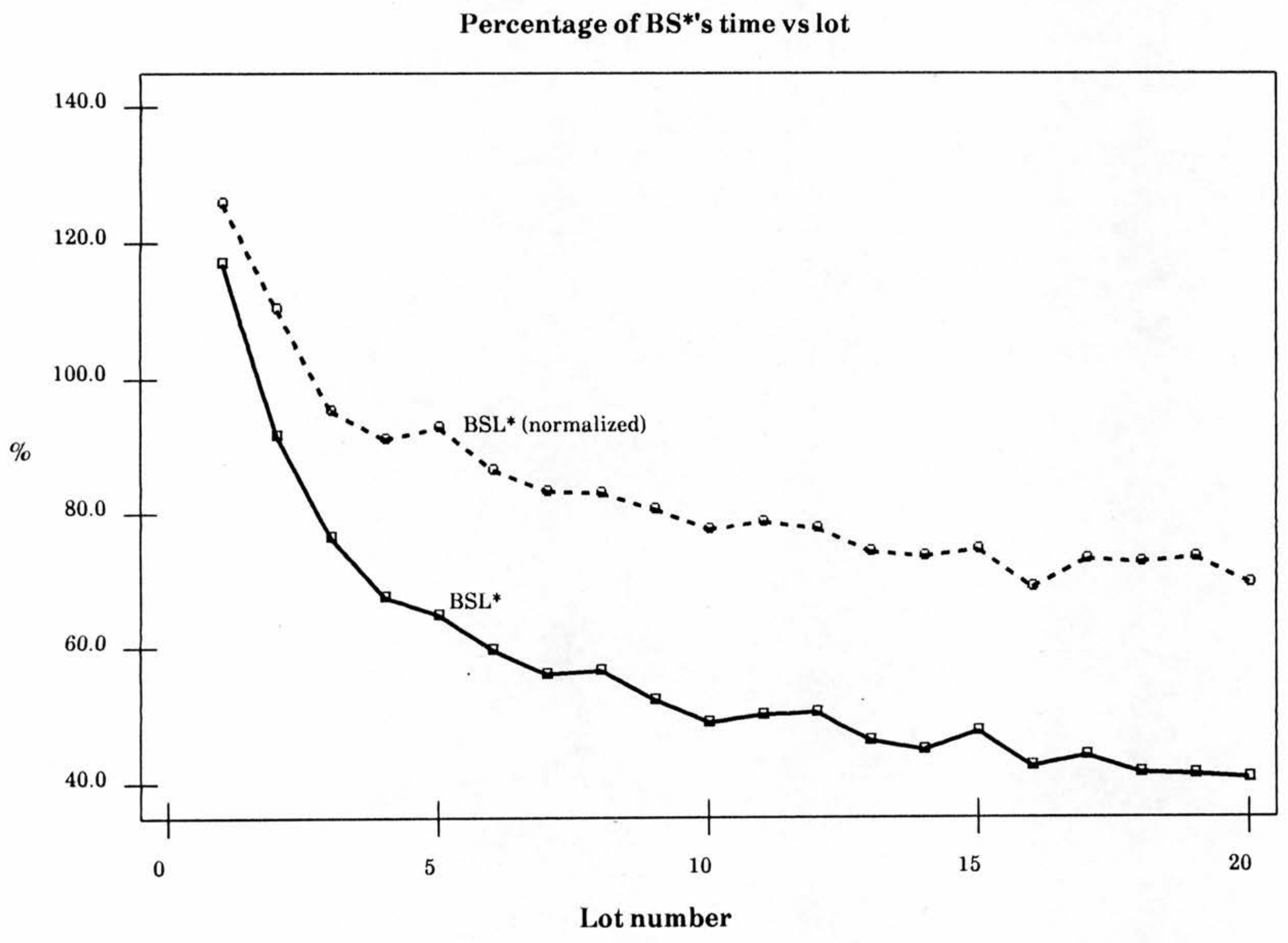


Figure 4-12. Effectiveness of learning in reducing search time.

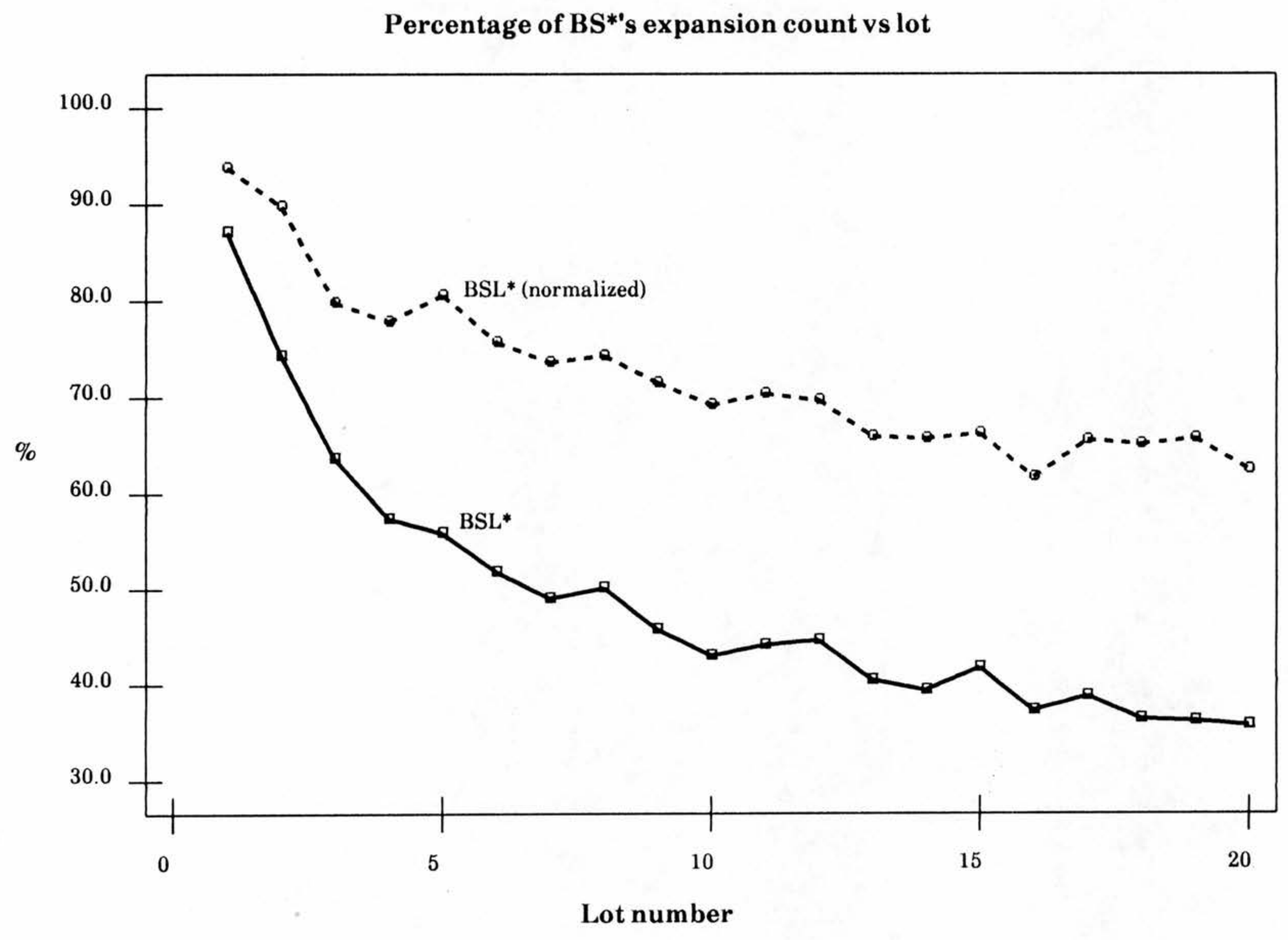


Figure 4-13. Effectiveness of learning in reducing search effort.

is because much time was initially spent learning and there was insufficient learning to make a significant impact on the search effort. But after about 30 path computations, learning begins to pay off.

Figure 4-13 compares the relative search effort involved in terms of number of nodes expanded in each lot. Again, two plots are shown so that the impact of learning on search cut-off can be observed. Both show a positive effect of learning in reducing search effort, with greater effect as more learning is attained.

4.7. Summary

There is abundant incidental knowledge embedded within search trees developed in the course of solving a particular problem. The embedded knowledge is potentially useful in two ways. First, it can provide instant solutions to later problems. Second, it can be used to prune the search trees of subsequent searches by weeding out some of the unpromising nodes. Consequently, solutions can be found more rapidly. In order to reap these advantages of learning, three problems must be solved:

- how to represent and retrieve the knowledge efficiently;
- how to extract the embedded knowledge;
- how to apply learnt knowledge during search.

The first problem can be solved by using two matrices L and Q. L holds the costs of the paths associated with its indices and Q holds the back pointers. By tracing these pointers, learnt paths can be retrieved. In general, both must be square, but if the graph is undirected and only optimal solutions are learnt, then L is symmetrical and can be stored in a triangular array.

The second problem can be solved by three learning algorithms (FwdLearn, BwdLearn and Glean). These algorithms work efficiently by avoiding redundant learning and prematurely aborting the learning process when it concludes that the remaining information is not new. All three algorithms are applicable in bidirectional searches, but only FwdLearn is applicable when search is unidirectional. In certain circumstances, these algorithms can be invoked dynamically in unison with search or within a separate procedure after the search terminates. Both methods are applicable when learnt solutions need not be optimal. Dynamic invocation is more efficient but it requires the consistency assumption to hold if learning is restricted to optimal paths. If the consistency assumption does not hold and learning is confined to optimal paths, then the post-search method is necessary and even so, learning must be based solely on the final optimal solution.

Concerning the third problem, it was shown how learning serves two useful roles: preemption and cut-off. Implementing preemption is trivial but hastening search termination is trivial only if suboptimal solutions are acceptable. In this case, all ends well when any partially developed path can be completed using a learnt path. If optimal solutions are required, the matter is less straightforward, and it depends on the directional mode of search. If search is unidirectional, search cut-off hardly occurs in A*, but can occur more frequently in B and B'. However, in bidirectional admissible search, there is more scope of search cut-off being advanced by considering learnt paths. For this reason, the application of learning benefits bidirectional search more than unidirectional search.

The ideas are generally applicable to various search algorithms. However, learning is useful only when: (1) search is recurrent; and (2) suboptimal solutions are acceptable, or if only optimal solutions are accepted, then a bidirectional admissible algorithm is used (implying also that the consistency assumption holds). It happens that these conditions are met in our AGV application.

Finally, to demonstrate the worth of learning, the BSL* algorithm was introduced and tested. Based on BS*, it incorporates the three learning algorithm as well as the two learning application techniques. Empirical data substantiate the hypothesis that learning can reduce both the search time and effort.

Chapter 5

Tolerant Planning and Negotiation

5.1. Introduction

Planning systems need to consider the difficulties of successful execution to be useful in solving real world problems. Merely generating a logically correct order of actions to achieve a goal only solves the real problem in part. If it over-optimistically assumes that the states of the world will change as expected according to its model of the world, then there is a likelihood that the plan will fail. The problem of achieving the goal then remains. Besides logical correctness, plans must also have a degree of executability in the real world. By "executability", I mean the ability to cope with real world issues during execution; issues which may not have been accounted for during plan generation, either not at all or only partially. For example, time is often an important factor, especially when a plan is part of a higher schedule of events. Other real world issues a planner may need to consider are: unexpected environmental changes; uncertainties in action outcomes; resource limitations; and unforeseen interference from other agents in the same environment.

The importance of plan executability can be seen in recent work in AI planning research which augments earlier work focusing mainly on logical correctness. Early AI planners evolved over two generations. The first generation of planners (e.g. Fikes and Nilsson's STRIPS, 1971) produced plans which were totally ordered sequences of primitive actions. Observing that some actions can be executed in parallel, Sacerdoti and Tate developed the second generation of nonlinear and hierarchical planners: NOAH (Sacerdoti, 1975) and NONLIN (Tate, 1977). These are characterized by the partial order of plan structures and the use of abstract search operators or action schemas to successively refine the plan structure. Following this, the focus shifted onto real world issues of practical importance. Vere built DEVISER (Vere, 1983), a system based on NONLIN which could handle time constraints and timed events in the real world. Other work related to planning involving time can be found in Allen (1983), Bell and Tate (1984, 1985), Cheeseman (1983), Dean (1985), Miller *et al* (1985), Tsang (1986) and Vilain (1982). Efforts have also been made in designing planners to manage resources (Bell, 1985; Wilkins, 1982), monitor the progress of plan execution and incorporate the use of sensory systems into plans to guide plan execution dynamically (Finger, 1982; Wilkins, 1985). Another area which has attracted considerable attention is multi-agent planning (Genesereth *et al*, 1986; Georgeff, 1984, 1986; Rosenchein, 1982, 1985; Rosenchein and Genesereth, 1985) where coordination and cooperation are the main research issues.

Plan executability warrants more than higher sophistication in the planner to anticipate real world issues prior to execution. It also points to the need for a dynamic replanning capability.[†] This need is due to a practical limit as to what a planner can consider prior to execution; a limitation which means that surprises during execution can nevertheless happen. However, dynamic replanning can be expensive or is liable to be futile if it fails to complete in time, Hence it is desirable to minimize or defer dynamic

[†] Dynamic replanning also implies the means to establish the circumstances in which replanning is performed. This means that a planner should have a plan execution and monitoring component (Currie, 1985; Wilkins, 1985). One could say that the third generation of planners are characterized by their dynamic replanning capability.

replanning as much as possible. The problems of dynamic replanning are more severe in multi-agent domains where replanning for one agent may have repercussions for other agents. These observations suggest that a worthy objective of a planner is to generate plans which are not prone to require repair during execution.

The theme of this chapter is that *tolerant planning* imparts executability by allowing leeway for execution errors. By tolerating execution deviations, dynamic replanning need not be invoked as often or as immediately as would be the case with a less tolerant plan. The approach in designing tolerant plans is to allow for redundancies in the requirements (usually resources) for execution. While this approach is feasible, it raises another problem—more conflicts must be resolved during planning. This conflict resolution problem can be solved using a novel model of *iterative negotiation* for multi-agent coordination. The purpose of this chapter is to discuss, in general terms, these concepts of tolerant planning and iterative negotiation. This will add another tool to the repertoire described in previous chapters with which an AGV movement planner can be designed to generate coordinated movement plans (described in chapter 6).

We begin by describing in section 5.2 some major problems in multi-agent planning. This provides the motivation for the concept of robust planning which is introduced in section 5.3. The intent is to show how tolerant planning is a means to a higher aim of plan robustness. We then consider some strategies for tolerant planning in section 5.4. In section 5.5, a computational model of iterative negotiation is described to show how it resolves conflicts between agents as each strives to generate its own tolerant plan.

5.2. Problems in Multi-agent Planning

In multi-agent planning, the plans for every agent must be constructed such that each agent is able to achieve its goal, and at the same time, does not prevent other agents from achieving their goals. This constraint requires two kinds of conflicts to be resolved: (1) intra-agent conflicts which are local in nature, pertaining to the interactions between steps within an agent's plan; and (2) inter-agent conflicts which pertain to interactions between the steps of two or more agents' plans. The conflict resolution problem is thus more complicated in multi-agent planning compared to single agent domains where only intra-agent conflicts are involved.

Besides a more difficult conflict resolution problem, there is also a problem of extensive dynamic replanning during execution. When an agent deviates from its own plan during execution, corrections to its remaining plan may be needed, and this in turn may set off a chain of changes upon other agents' plans. If plan modifications are excessive the fixes may not be achieved in time to meet near real-time performance which is essential for a dynamic replanning system. Hence an important objective is to contain plan modifications, minimizing them as far as possible.

The problems mentioned are not unrelated. The way in which multiple interdependent plans are generated influences the extent to which replanning is necessary when execution deviations occur. If the system is able to produce plans which are tolerant of execution deviations, then in most instances when deviations are within limits, the system avoids immediate plan revisions with their potential for ensuing chain revisions. This is particularly important when execution deviations are likely, dynamic replanning is computationally expensive and the plan revisions may not be produced in time to avert failures.

The point addressed is that strong interdependency between multiple agents' plans can bring forth plans which are brittle. Traditional approaches in solving just the conflict resolution issue without regard to its influence on dynamic replanning are liable to produce brittle plans. A solution to this is tolerant planning which aims to produce robust plans which continue to be executable despite execution errors and environmental changes.

5.3. Robust Plans

A robust plan permits the goal to be achieved in spite of events which work against its successful execution. Plan robustness is one of the factors which contributes to plan executability. In contrast, a brittle plan would be almost immediately invalidated when an adverse event occurs *e.g.* hardware failure; an unexpected change in environmental conditions; an action which did not quite achieve the intended effects. The choice of the term "robust" instead of the more obvious term "tolerant" is to make a distinction between tolerant planning as a means and the kind of plan it produces as an end. This distinction is important because robust plans can be achieved by other means. An alternative approach will be described later in this section.

The concept of robust systems is not new. Many fault-tolerant systems exist today. With satellites costing astronomical sums, it makes good sense to have several redundant subsystems on board to enable uninterrupted operation in the event of hardware failure. The US space shuttles also have several redundant flight management computers to ensure a high degree of safety. Critical computer-based systems which cannot tolerate system stoppage (e.g. air defence control systems) also have hot standby systems ready to take over when the current system fails.

From the above examples, we can note that a key strategy in designing fault-tolerant or robust systems is to provide for redundant resources. The redundant resources need not be restricted to physical resources. Taking a more general view, time and physical space can be considered as resources as well. Bell (1985) went even further in his classification of resources and proposed that truth assertions about the world can be treated as resources.

The same approach can be used to build robust plans. Instead of merely ensuring that the bare resources required are available for each plan step, redundant resources can be allocated as reserves. For example, if an aircraft normally needs 800 gallons of fuel to cover a certain route without stopping enroute, and it is possible that a stronger head wind than normally encountered may force an undesirable detour to an intermediate refuelling point, then it would be wise to plan for more gallons. Another example involving multiple agents can be taken from the movement coordination of several mobile robots. Considering the space-time constraint that no two robots can be at the same place at the same time, a plan to schedule the loiter times of two robots which require to be at the same position to perform some task might require robot X to be there from 11:00 to 11:08 a.m. followed by robot Y to be there from 11:08 to 11:15 a.m. The problem is that should robot X arrive late or loiter longer than planned, then the plan for robot Y must be modified to avoid an impending collision. A more tolerant plan will take into account the possible delays and grant more loiter time for both X and Y. The planner may even ensure that a time gap separates the loiter periods of X and Y just in case X's loiter time exceeds the tolerance provided.

Tolerant planning is not the only way to build robust plans. There is a twin of tolerant planning—contingent planning. The difference is that whereas tolerant planning imparts robustness by granting redundant resources, contingent planning takes the approach of having redundant knowledge of how to accomplish the goal. The recent work of Drummond *et al* (1987) on C-plans (based on net theory) to explicitly represent disjunctive actions in the plan structure is an example of contingent planning. C-plans allow the plan execution monitoring component to infer an alternative way to achieve the goal if one option is not possible, but in determining the alternative, it does not need to modify the plan structure. Disjunction can hence be considered as an approach to imparting robustness to plans through causal tolerance.

There is a trade-off between ease of plan generation and plan robustness. In tolerant planning, the minimal plus the redundant resources for a plan step form a set of resource choices to achieve the subgoal. As a plan develops, more resources are reserved and other agents are denied their use. From the perspective of other agents, the more resources are reserved by one agent, the more restricted are the options open for their consideration and it becomes increasingly difficult to find a feasible plan. In order to counter the greediness of agents as they reserve redundant resources, a mechanism is required to ensure that agents cooperate by releasing some of the redundant resources which are needed by other agents. I shall address this problem in section 5.5 by way of the iterative negotiation model.

5.4. Strategies for Tolerant Planning

The preceding discussion on tolerant planning suggests two main strategies for tolerant planning: setting tolerances for execution deviations, and imposing buffers or hedges between the plans of multiple agents. These are ways of realizing the principles of greediness and wariness which seem useful in designing tolerant plans.

5.4.1 Setting Tolerances

If the achievement of subgoals must be satisfied precisely then the planner must deem any deviation from the expected outcomes as possibly leading to failure. Consequently, it has to take some kind of plan repair action. The problem is that dynamic replanning can be expensive, may take more time than is available, or initiate a chain reaction among other agents whose plans are not independent. The weakness of plans which must be executed with precision is that such plans are brittle and vulnerable to changes.

Precise satisfaction of subgoals also require strict execution control not only of the agent but of the environment so that the subgoal can indeed be precisely satisfied with confidence. Unfortunately, strict execution control entails more complex and expensive monitoring systems. A recourse is to postpone dynamic replanning by allowing subgoals to be satisfied imprecisely or within certain tolerances. So long as deviations from the ideal subgoal are within limits, the same plan structure should permit continuance towards satisfying the next subgoal. Although the manner of execution of a predetermined plan step may have to change, there should be no immediate need to modify the plan structurally—a more radical remedial action requiring dynamic replanning.

5.4.2 Hedging

Hedges are safety margins imposed between plans of several agents working in the same environment. One can visualize this in terms of requirement or resource spaces. Every agent reserves a subspace of the resource space. The resource space defines what is needed by the agent at a particular time and hence is dynamic. Whenever two spaces overlap, a conflict exists. When the spaces touch, there is no conflict, but the situation can be described as critical since the slightest change in resource requirement leads to an overlap. If the resource subspaces are allocated during planning such that they are adequately spaced apart, then there is some leeway for requirement changes to occur without bringing about an overlap immediately. In this way, plans of other agents are tolerant of another agent's execution deviation.

Whereas setting tolerances allows an agent to postpone dynamic replanning otherwise warranted immediately by its own execution deviations, hedging allows agents to postpone dynamic replanning due to the deviant behaviour of other agents. Hedging thus helps to prevent or contain a chain reaction of plan alterations.

5.5. Negotiation as a Conflict Resolution Mechanism

Tolerant planning requires the allocation of redundant resources and this can rapidly reduce the availability of resources. When available resources are over demanded by several agents, they must come to an agreeable compromise in which the total demand on resources does not exceed availability. These conflicts can be resolved by an agent yielding some of its redundant resources for the sake of another, or several may compromise their initial demands. Whichever the case, the actual conflict resolution process can take either of two forms.

The first approach is to submit the dispute to a higher authority; a third party acting as an arbitrator solely decides how to apportion the disputed resources between the plaintiffs. The second approach is to let the two agents in conflict settle the dispute between themselves.

Good arbitration requires consideration of all the relevant factors involved. The factors will in general be case-dependent. This makes arbitration difficult to implement—it must be knowledge-rich, knowing how to resolve disputes judiciously for each different case. Its more extensive and intensive consideration of the circumstances surrounding each case means that arbitration requires extended time and effort.

The second option is similar to a haggling process. Neither party involved needs to know the plans, intents and requirements of the other party. This approach is conceptually simple and yet can be effective. Let us consider an example from the business world for a better understanding of how negotiation works.

When a customer wishes to procure some item from a company, each party (in general) would offer a different price. The customer would hope to secure the item at as low a price as possible, while the company would aim to fatten its profit margin by closing the sale at as high a price as possible. Both have to come to a point of mutual agreement without the benefit of an arbitrator. What normally happens is an iterative process of negotiation with offers followed by counter-offers and so on till the price differential is eliminated. Otherwise, when neither is prepared to compromise further to strike a deal the negotiation breaks down. Note that for each agent, there is an element of selfishness along with a willingness to concede. We can also observe that over a series of business transactions, when there is a choice to make a deal or break off negotiation, customers and businessmen are generally satisfied in the end. Admittedly, this conclusion drawn from a simplified illustration excludes extraneous circumstances (*e.g.* economic recession; mismanagement; fraud; labour disputes) in the real world. Fortunately, such factors are irrelevant to most applications for which a negotiation metaphor can be useful.

This general overall satisfaction by virtue of freedom of negotiation suggests that the concept can be imported to resolve conflicts between agents in a problem solving situation. Another advantage is that by not involving a higher authority, a heterarchical organization more amenable to parallel processing is possible. Vámos (1983) has also argued that heterarchical organizations are more suited than hierarchical organizations in evolving cooperation and in general work more efficiently. Apparently, this point is also evident in contrasting political and economic systems (*e.g.* capitalism *vs* socialism; decentralization *vs* centralization; privatization/deregulation *vs* nationalization).

5.5.1 Characteristics of Negotiation

In designing an effective computational model of negotiation which mimicks the concept described, we need to examine the characteristics of negotiation and seek an understanding of how each can help in conflict resolution. We examine these as follows:

- *Conflict knowledge.* An agent knows with which other agent(s) it is in conflict with and what the conflict(s) is/are. Unless this is so, conflicts will be overlooked to the detriment of subsequent plan execution. The knowledge of existing conflicts can be self realized or discovered by another agent which then informs the agent(s) involved.

- *Willingness to negotiate.* Between any two agents in dispute, at least one must be willing to attempt to resolve the conflict. Otherwise, a negotiation process cannot even begin. In the absence of capital gains, we can assume that this motivation comes from altruism or benevolence imposed by the system designer. However, willingness does not mean that an agent must settle conflicts at all cost, or even at some cost. As in reality, a conflict can be resolved when only one agent makes the concessions. Also, when the bottom line is reached, an agent can break off negotiation. If all parties break off negotiations, then an impasse is reached; the conflict is not entirely resolved. Sometimes, an impasse may be temporary. After a lapse of time, the circumstances may have changed somewhat, and an agent may then be willing to make further concessions.

- *Knowing the negotiables.* An agent knows its own bottom line of negotiation, although not necessarily that of other agents. This implies that the objects in conflict can be partitioned into two sets: the negotiable and the non-negotiable with the latter defining the bottom line.

- *Yielding and Substitution.* An agent may compromise its position by either yielding or substituting some of the contended objects with other objects which may or may not be free of contention. Substitution is actually yielding made up in part or whole with further object reservation. Since yielding decreases redundancy more than substitution, the latter should be attempted first whenever possible.

- *Agents are selfish.* Although agents are willing to negotiate, they are in a way selfish in trying to minimize what they can give up to settle the conflict. Each would hope that the gap of dispute can be narrowed by the other party's effort rather than its own. In the context of multi-agent planning, the initial tolerant plan of each agent (*i.e.* before any redundant resources has been yielded to resolve a conflict with another agent) is the "optimal" tolerant plan in the sense that any yielding reduces its robustness and hence executability. Thus every agent should want to practise the policy of being selfish *i.e.* sticking as close as possible to its initial plan.

- *Negotiation is an iterative process.* A point of mutual agreement is usually reached after several rounds of concessions. This follows from the selfish characteristic since minimal compromise dictates that an agent should yield in small steps. Depending on the gap of dispute, several small steps by both agents will have to occur before an agreement can be reached.

- *Intransigence.* Sometimes, when an agent believes that the other has not conceded sufficiently before it makes its next concession, it may elect to concede nothing *i.e.* appear to be intransigent. This induces the other agent to make another offer, and possibly, successively further offers to keep the negotiation process going. Such behaviour is also observed when one agent has a relatively smaller set of negotiable objects to concede away.

- *Skilful negotiation.* Every agent has a set of negotiation techniques to use. The set used by one agent can differ from the other agent's, as should be the case since their perspectives (of what they desire to achieve ultimately) differ. Furthermore, agents know at each stage of the negotiation which technique to apply next. This implies that they know the order of techniques to use and which of the negotiable objects to consider next.

- *Nested negotiation.* Consider the case of agents A1 and A2 contending over object X. The resolution may require that A2 gives up X completely, but it cannot do so without some substitution in order to maintain its position above the bottom line. A2 then finds that X can be substituted with object Y. However, Y is held by agent A3. A2 decides to negotiate with A3 over Y before it resolves its conflict with A1. Then again, A3 may have to negotiate with yet another agent in order to yield Y, and so on as the case may be. We see that although a single negotiation act initially involved only two agents, it may incur further nested negotiation acts within it. For a more concrete example, suppose Pegasus Aircraft Co. is the main manufacture for the PX1 aircraft, and it uses Eagles Co. as a subcontractor for the wings. In a business negotiation, the customer insists on a reduction of \$10m for the PX1 aircraft. Pegasus Aircraft Co. then tries to meet this by asking Eagles Co. to reduce its price for the wings by \$3m. Eagles Co. finds that it is nevertheless a profitable deal to lower its price by \$3m, but it will try to minimize its profit squeeze by negotiating with T-Lium Co., which supplies the titanium material for making the wings, for a reduction of \$1m for the material.

5.5.2 A Computational Negotiation Model

From the above characteristics, we see that there are many degrees of freedom which one can exploit in a computational model. The good news is that a variety of negotiation behaviours can be modelled, offering a richness and flexibility to meet a wide range of situations. The bad news is that choice brings its own problem—how to determine the optimal choice. In the extreme case when a plethora of possibilities are available, it becomes a combinatorial optimization problem. Unless the problem is amenable to formal analysis, one can seldom hope to find an optimal choice. Without the benefit of formal analysis, the recourse is to make a choice judiciously, implement and observe, and modify the implementation through trial and error.

After some experimentation, I have found that the model defined by the following algorithms mimicks all of the characteristics of iterative negotiation. The algorithms implement a control structure for iterative negotiation, but leave much of the details (*e.g.* negotiation techniques) for the user to define. This is necessary since certain details are dependent on the application domain. We will see an example of how this general model can be specialized to deal with conflicts in planning AGV movements in Chapter 6.

Algorithm 5-1:

```
    procedure Plan(Agent, Task)
    /* Construct a conflict-free plan. */
    vars P, pi, MainConflictSet, Residue1, pi', P';
1.  Generate plan P independently for Agent to achieve Task.
    /* P is feasible from the viewpoint of Agent, but may involve steps which
    conflict with other agents. In general, P is a partially ordered graph of
    primitive plan steps. */
2.  until all plan steps in P have been considered do
3.      Choose a plan step pi not considered before.
4.      Identify MainConflictSet, the set of agents in conflict with pi.
5.      if MainConflictSet is empty then go to step 13 endif
6.      Residue1 ← ResolveConflict(Agent, pi, MainConflictSet).
7.      if Residue1 = 0 then go to step 13 endif
        /* pi cannot be made conflict-free. */
8.      Find an alternative plan step pi' for pi.
9.      if pi' exists then pi ← pi'; go to step 4 endif
        /* P is infeasible */
10.     Find an alternative plan P' for Agent to achieve Task.
11.     if P' exists then P ← P'; repeat step 2 afresh endif
12.     signal failure and exit  /* Task cannot be achieved by Agent. */
13. enduntil
    endprocedure
```

Algorithm 5-2:

```
    procedure ResolveConflict(Initiator, Action, ConflictSet)
    /* Attempt to eliminate all conflicts associated with Action; returns the
       residue. */
    vars Residue2, Respondent, C;
```

```
1.  Residue2 ← 0
2.  until Residue2 > 0 or all members of ConflictSet have been considered do
3.      Choose from ConflictSet, an agent Respondent not considered before.
4.      Compute conflict C between Initiator and Respondent with respect to
        Action.
5.      Residue2 ← Negotiate(Initiator, Respondent, C)
    enduntil
6.  Return Residue2.
    endprocedure
```

Algorithm 5-3:

```
    procedure Negotiate(Initiator, Respondent, Differential)
    /* Iteratively negotiate to eliminate Differential; returns the residue. */
1.  Differential ← ApplyTechnique(Initiator, Differential).
2.  if Differential = 0 then exit, returning 0 endif
3.  Differential ← ApplyTechnique(Respondent, Differential).
4.  if Differential = 0 then exit, returning 0 endif
5.  if  no new techniques can be applied to reduce Differential
6.          then exit, returning Differential   /* Negotiation impasse occurred. */
    endif
7.  Go to step 1.
    endprocedure
```

Algorithm 5-4:

```
    procedure ApplyTechnique(Agent, Differential)
    /*  Respond by applying a technique in the current negotiation round;
    returns the residue. */
    vars T, SubAgents, Residue3;
1.  Find technique T appropriate for Agent to reduce Differential.
```

```
2.  if prior to T's application, negotiation with Agent's subsidiary agents is
        required,
3.      then  Identify SubAgents, the set of Agent's subsidiary agents involved.
4.            Residue3 ← ResolveConflict(Agent, T, SubAgents)
    endif
5.  Apply T to reduce Differential as far as Residue3 would allow.
6.  Return Differential.
    endprocedure
```

In the main procedure Plan, an agent attempts to resolve all conflicts associated with every step in its plan. If a step cannot be made conflict-free despite negotiation, a substitute step is sought. If this is also futile then a new plan must be constructed. The agent attempts to resolve the conflict by first identifying the set of agents with which it is in conflict with and then invoking the procedure ResolveConflict. Within ResolveConflict, the planning agent (initiator) negotiates in turn with each member (respondent) of the conflict set by calling Negotiate. The Negotiate procedure iteratively eliminates the conflict with the initiator and respondent each taking turns to reduce the conflict by calling ApplyTechnique. As its name suggests, ApplyTechnique selects and applies a technique to reduce the conflict. If necessary, ApplyTechnique will initiate its own conflict resolution process (thus capturing the nested negotiation behaviour in the model) before it returns a residual conflict.

Defining the negotiation techniques is a critical part of the modelling process since effective conflict resolution depends on rational negotiation behaviour of agents. Having defined the techniques, the next step is to rank them in increasing order of preference. Naturally, the most preferred involves the least compromise. The outcome of ordering is a list $L = (T_1\ T_2\ \ldots\ T_n)$ where T_1 is the first technique to be tried and T_n is the technique of last resort. Defining the list L (also called the skill set) captures the skilful negotiation characteristic of an agent. The various techniques are domain-specific and are thus left to the user to define during implementation. Besides enabling a variety of negotiation behaviours to be modelled, this simple list representation also offers the flexibility of adapting the skill set to suit the current circumstances since the list can be manipulated dynamically.

A technique is chosen by taking the first element in L. When the chosen technique is no longer applicable, L is reset to its tail. Applying a chosen technique has three effects: (1) the conflict is totally eliminated—in this case, no further negotiation is necessary; (2) the conflict is partially reduced; and (3) the conflict is not reduced at all. The last effect is not necessarily a result of means being exhausted; it can be used to simulate intransigent behaviour. In the second case, the technique may or may not have exhausted its means. If it has exhausted its means (*i.e.* no further redundant resources can be yielded by reapplying the same technique), then a different technique, if any, should be attempted next by the same agent.

The main algorithm Plan can be modified to invoke an arbitration procedure if there is still more residual conflict (Residue1) after attempting resolution at step 6. Although it has been pointed out that arbitration can be computationally expensive, its inclusion here may have the advantage of eliminating the residual conflict by decree, and therein avoid local replanning (step 8) or better still, avoid total replanning (step 10). If these replanning steps are more costly than arbitration, then arbitration will be worthwhile. On the other hand, there is a risk—arbitration may be in vain, in which case, the wasted arbitration effort is an added burden since the replanning step(s) would not be circumvented.

The design of an arbitration procedure is somewhat open-ended. In general, it should have a higher perspective of the circumstances. Thus it should have access to the repository of knowledge in the agents in conflict, as well as the organizational goals/policies. It subjects the relevant information to its decision procedure and the output should be a resolution plan or an indication of non-resolution. In the latter case, it means that the residual conflict has to be eliminated by altering at least one agent's plan.

5.5.3 Related Work

Davis and Smith (1981) developed a contract negotiation metaphor the name of which suggests great similarity with our negotiation mechanism. However, there are significant differences. The contract negotiation metaphor has been used for task distribution and dissemination of control in a multi-agent problem solving environment. When a task needs to be solved by an agent, it is typically decomposed into subtasks and announced to agents in the same environment. Agents which are able to support an announced subtask make a bid which includes certain information specified in the announcement message. The main agent (manager) decides which bidder to award the subtask to and sends out the award message to the successful agent (contractor). In assuming the subtask, the contractor may in turn become a manager and decompose the subtask even further and initiate another announce-bid-award contract cycle. Even if a task is not decomposed, an agent may delegate it to another instead of itself undertaking it. Davis and Smith's negotiation metaphor is suited to distributed problem solving and is not designed to accomplish conflict resolution.

A more closely related work is Goldstein's (1975) which schedules the activities of individuals over a particular time period in such a way as to allow all if not most of the activities to occur. Individual activities are initially specified with associated general requirements and preferences, leaving a great deal of freedom for the system to instantiate the period of occurrence for each activity. Conflicts may arise from activities of one or more individuals. The system first constructs a possibility space of periods for the activities of each individual. It then resolves conflicts between activities of the same individual. Finally, it fixes conflicts between activities of different individuals. Every resolution step narrows the possibility space.

Goldstein's scheme of conflict resolution relies on two types of bargaining techniques. Resource-driven and purpose-driven techniques form the first and second type respectively. The two types differ in that resource-driven techniques only make local adjustments of preferences whereas purpose-driven techniques can modify requirements or goals. Being more radical in behaviour, purpose-driven techniques are invoked to fix a conflict only when the resource-driven techniques have failed. In general, a conflict can be fixed by more than one technique. The system will invoke all the applicable techniques to build a search tree in the same way as search operators are used to generate a search tree in STRIPS (Fikes and Nilsson, 1971). At any stage, the system chooses the most promising node based on the sum of satisfied preferences to work on next until all conflicts have been resolved. Goldstein's use of bargaining techniques differs from our negotiation techniques in the following ways:

- Goldstein's system applies all possible bargaining techniques to build a search tree. Our negotiation techniques are linearized and applied one after another in a predetermined order and search is not involved.
- The motivations differ. Goldstein applied his system to plan activity calendars of individuals and did not seek to build schedules which allowed activities to overrun without warranting immediate changes to affected activities. In our case, we aim at generating robust plans which tolerate to some extent the deviant behaviours of other agents.
- By attempting to debug each conflict in a single step, Goldstein's system is non-iterative. It resembles more the case of every conflict being resolved by the arbitrator with no initial haggling between the agents in dispute.
- Fixes in Goldstein's system can work in a discontinuous manner. For example, a meeting initially scheduled from 9 to 10 a.m. may be shifted to 3 to 4 p.m. In our case, conflicts are resolved in a gradual manner and this has the effect of preventing the final set of plans from deviating too far from their initial ideals.

5.6. Summary

Tolerant planning is a feasible approach to building robust plans which have better executability than brittle plans. Robust plans are important and useful in several real-world situations, particularly when dynamic replanning needs to be minimized and when execution monitoring should be implemented economically. These organisational objectives can be achieved by allowing the goals of plans to be satisfied imprecisely yet innocuously during execution. The key strategy underlying tolerant planning is to provide more than sufficient means (resources) to achieve the goals, in much the same vein as existing engineering systems are designed to be fault-tolerant by virtue of redundant hardware.

While robustness is the benefit gained, the cost of tolerant planning is an increased demand for resources. Each agent's tolerant plan will need to reserve more resources than would be the case otherwise. Resource reservation by one agent denies the same resources to other agents. Since overall availability of resources is limited, resource contention becomes more acute. A mechanism is hence needed to allocate available resources fairly and in accordance with operational requirements.

The iterative negotiation algorithm is proposed to solve this problem. It relies on agents having a set of negotiation skills to participate intelligently during negotiation with another agent. Intelligent negotiation requires the exercise of intransigence when one has little to give away and tactical sense which seeks to induce the other party to meet one most of the way in settling the dispute. When all agents negotiate rationally, an approximately equitable state of compromise should result. Negotiation thus achieves a semblance of rational self-organisation, not in structural configuration, but in terms of agents finding their own level of plan robustness as conflicts are resolved.

The main advantage of this model is conceptual and representational simplicity. Unlike some multi-agent planning models, there is no need to include within the representation of agents models of other agents' beliefs, intents, plans and inference mechanism. Rather than having to grapple with the logical complications of such an approach, the approach of iterative negotiation merely requires a simple form of rational interaction.

Chapter 6

Planning Coordinated AGV Movements

6.1 Introduction

Coordination of AGV movements is the main problem in AGV movement planning for the obvious reason of avoiding collisions. The problem can be solved during plan execution as is the approach in current AGV systems. This is feasible when deadlines need not be strictly satisfied since *ad hoc* coordination does not guarantee that AGVs can meet their deadlines.

An AGV movement planner which is able to guarantee conditionally that task deadlines can be met is not only necessary when plans must meet both logical and temporal constraints, it also has the important advantage of permitting further planning on the basis of such guaranteed information. Otherwise, further planning must be either curtailed or made precariously.

Planning with a deadline constraint requires the coordination problem to be solved during planning and not during execution. This enables plans to proceed with greater certainty that if no unexpected interference occurs, then execution will proceed unhindered and the deadlines will be met. Besides meeting deadlines, the approach which resolves all anticipated conflicts prior to execution has another advantage—the sophisticated and costly monitoring and control systems for real time coordination which are necessary for current AGV systems may be substituted with simpler and cheaper systems.

This chapter describes how the coordination problem can be solved during AGV movement planning *i.e.* given a task, how can an AGV construct a conflict-free plan prior to execution. It also exemplifies chapter 5's themes of tolerant planning as an approach towards building plans which have better executability, and the iterative negotiation metaphor as a model of conflict resolution.

In section 6.2, the movement coordination problem will be given the more general resource-based perspective. This gives an appreciation of the similarities in the principles underlying various solutions to coordination problems at large. Section 6.3 examines the traditional methods of coordinating AGV movement. We will see that there is a serious drawback in not allowing accurate temporal projections which are essential in planning with deadline constraints. Section 6.4 motivates the need for interval-based schedules and shows how the intervals should be defined according to the required tolerance against execution deviations. Section 6.5 describes the negotiation techniques and shows how these are organized into skill sets which can be selected according to the negotiation strategy used. Section 6.6 explains the implementation of the iterative negotiation model as a conflict resolution mechanism in the AGV movement planner. We will also see two examples: one showing how a negotiation process can resolve a conflict, and the other a non-trivial problem scenario to demonstrate the capability of the model. Section 6.7 discusses various issues related to a realistic implementation and points out some limitations and areas for further research. Section 6.8 acquaints the reader with related work.

6.2 The Coordination Problem

Coordination problems occur frequently in many application domains. They exist whenever there is a contention over limited means. For example, computing resources such as printers, tape drives, *etc.*, can be accessible to one process at a time, and it is a function of an operating system to coordinate access.

Coordination is also necessary in several transportation domains. Air, road and rail traffic control systems are obvious examples. Whereas tangible resources can be identified in the operating system example, the objects of contention may not be tangible in the transportation examples. Nevertheless, it is possible to take a general view of coordination problems; a view in which *resources*, not necessarily tangible, have to be allocated to meet certain constraints. The advantage of an abstract view is that general principles or strategies can be identified from solved problem instances, and these general strategies may be brought to bear on new problems in different domains.

Correlating a problem instance with the general view is straightforward when the objects of contention are tangible, but can be less apparent otherwise. For instance, many of the resources in the operating system example fit the general picture readily. In the case of movement coordination, the constraint that no two physical objects can be at the same place at the same time, suggests that the relevant resource is a space-time entity. For example, suppose T is the time interval in which every conceivable event of interest occurs. Imagine that every relevant position in the domain has its own interval T. This interval is the resource associated with the position. Safe coordination requires that only disjoint portions of the interval are allocated to objects which need to be at the position.

6.3 The Traditional Approach to Movement Coordination

The most common technique for coordinating the movements of AGVs is the blocking method, an idea copied from the railway domain. In the blocking method (Müller, 1983; Todd, 1986), the route network is partitioned into blocks (segments) and a block control system ensures every block has at most one AGV in it.

The block control system makes use of devices (sensors and activators) fixed along the routes as well as on the AGVs. These devices work in concert to stop an AGV just before a block, or allow an AGV to proceed along a block. The traffic control is guaranteed to be sound according to some logical constraints. Often, the governing logic is hardwired into the AGV system.

Another approach is to allow AGVs to move so long as no impending collision is detected. Collisions can be anticipated using sonar transceivers, contact sensitive bumpers, or some other proximity sensors. Coordination is achieved by stopping the AGV prior to collision, and enabling the AGV to proceed only when the cause of stoppage disappears. The advantage of this approach is higher throughput—blocks can accommodate more than one AGV at a time.

Traditional methods can solve the coordination problem effectively as has been proven by their long established use in industry. However, there is a serious shortcoming—they do not admit temporal projection unless a complete simulation is carried out. Even so, the predictions may not correspond with the actual outcomes during execution. The reason is that a certain non-determinism is involved in concurrently moving AGVs.

Suppose, two AGVs, X and Y, approach the same junction at about the same time and at the next time instant, both could have arrived at the junction according to the simulation. One must give way to the other if collision is to be avoided. If both stop to give way to the other, then a starvation type of deadlock occurs. Since AGV systems are engineered not to give rise to deadlocks and collisions, either X or Y but not both will stop, depending on whichever first gains control of the block. This is where the non-determinism arises. Inherent inaccuracy in any simulation model cannot predict which of the two AGVs will gain control first. If the simulation grants the block to X, then the ensuing sequence of events will be very different from the case if Y was granted control instead.

Temporal projection is nevertheless important since it serves to confirm whether AGV movement plans will meet the temporal constraints (deadlines) of a higher schedule of events. The conclusion is that AGV movement plans must be planned with complete temporal detail if temporal projection is to be possible. This departs from the traditional approaches which plans the routes with only some temporal detail (*e.g.* start times) and leaves spatial-temporal coordination to be realized in an *ad hoc* manner during execution.

6.4 Interval-based AGV Movement Plans

In this section, I will describe how AGV movement plans can be generated such that temporal projection is possible without simulation. I will also show how these plans meet the objective of executability by virtue of robustness; an issue which was addressed in chapter 5.

6.4.1 Temporal Projection via Scheduling

Temporal projection is the forecast of arrival and departure times of an AGV along a route. It is trivial once a movement schedule can be determined prior to execution since such a schedule defines when an AGV will arrive at and depart from nodes along its route. Elsewhere along the route, the arrival/departure time can be computed by interpolation or by solving the equations of motion.

The usual kind of movement schedule is point-based. A point-based schedule defines exactly the time instant when an AGV will arrive at a position. It is often illustrated using a graph of position versus time. For example, Figure 6-1 shows that an AGV departing from position A at time t_a, moving at constant speed towards B, will arrive at B at time t_b. It will remain at B until time $t_{b'}$ (perhaps to pick up a load) before proceeding to C at a slower speed, arriving there at t_c.

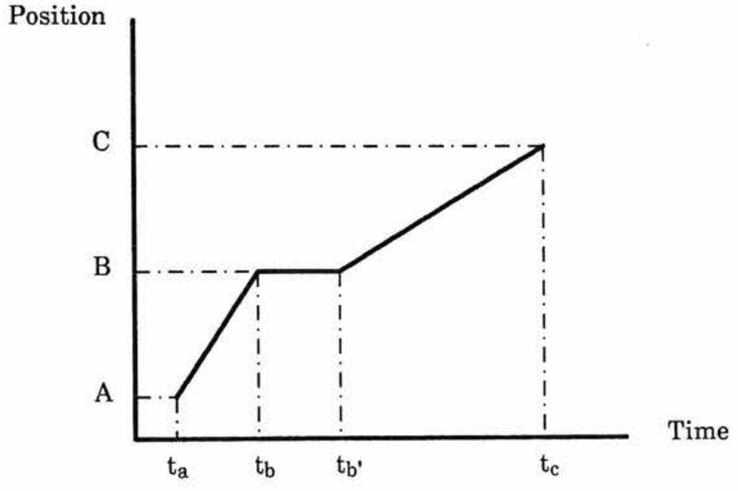


Figure 6-1. Graph of a point-based schedule.

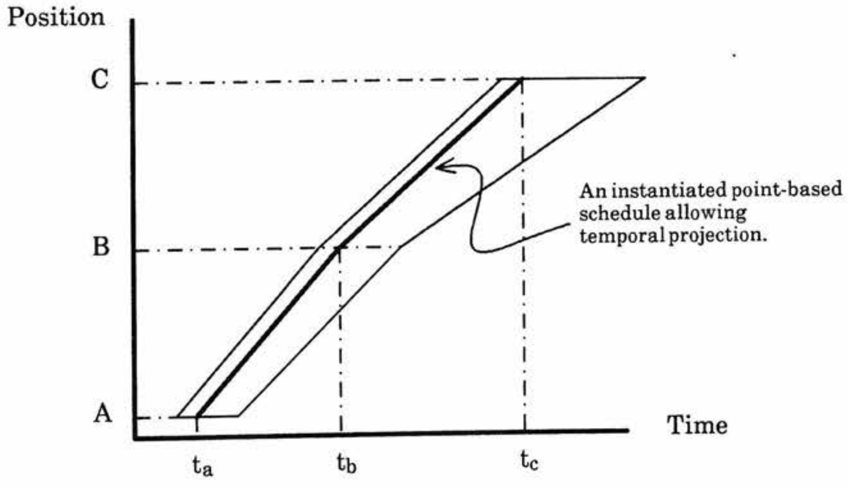


Figure 6-2. Graph of an interval-based schedule.

Alternatively, an interval-based specification for schedules can be used. An interval defines the range of time when an AGV is given sole access to a position. The interval-based specification is more general since it encompasses a set of point-based schedules. Figure 6-2 illustrates this. Note that the interval widths may vary. This has to do with providing an allowance for late arrival. Such allowances should reflect the difficulty of maintaining punctuality. Since an AGV is more likely to be late when a greater distance has to be traversed, the interval width should be monotonically non-decreasing with respect to distance travelled.

Temporal projection can be precise even with interval-based schedules. All that is required is a reference point in one of the intervals. Effectively, this reference point instantiates a point-based schedule within the set defined by the intervals. For example, if it is known that an AGV will arrive at A at time t_a (see Figure 6-2), then the arrival times at succeeding intervals can be predicted, assuming that the speed and waiting times along the route are also known.

6.4.2 Advantages of Interval-based Movement Plans

The justification for an interval-based representation stems from two objectives: (1) to reduce the frequency of plan revalidations during execution; and (2) to build robust movement plans which are less susceptible to replanning and have better immunity against replanning repercussions.

Suppose for the sake of argument, point-based schedules are used. Assuming that the planner had constructed a conflict-free set of schedules, there will be no collision as long as execution is perfect. Unfortunately, ideal execution is impossible in reality since execution deviations are bound to occur. Actual arrival times will differ at least minutely from the planned arrival times. Every time a deviation occurs, the remaining schedule must be revised and the system must verify that the new schedule is safe. This requires the examination of the movement plans of every AGV which will also be at points along the route associated with the revised schedule. Such frequent exhaustive revalidation has a vitiating effect on the near real-time capability of a dynamic replanner.

Use of interval-based schedules avoid the problem of frequent exhaustive revalidation. An interval-based schedule has time intervals at every check point along the route of an AGV. If these intervals are assigned such that all schedules are conflict-free, then no revalidation is necessary as long as the arrival and departure times of an AGV remain within interval.

Since continued safety (*i.e.* the plans remain conflict-free) is guaranteed when execution deviations remain within bounds, dynamic replanning is also unnecessary. Thus, interval-based movement plans are robust. The relation with tolerant planning by way of assigning redundant resources is clear. An interval at a point has redundancies since only a part of it will be "consumed" during execution. The robust characteristic is realized by tolerating deviant arrival and departure times which remain within the intervals.

6.4.3 Composition of an Interval

A rational scheme for defining the intervals at various check points (nodes) along every route is needed. The considerations for such a scheme should include the following:

- Tolerating self deviations: this should be proportional to the likelihood of arriving late. Since punctuality is increasingly hard to attain with longer routes, distance from the starting point is a good metric to use. For better realism, the topology of the route should also be considered (*e.g.* corners). We shall refer to the interval established to accommodate self deviations as the *tolerance* interval.

- Tolerating deviations of other AGVs: this can be a safety buffer of arbitrary constant size positioned at the front and rear ends of the time interval. Although constant, the size can be changed in later planning to reflect more accurately its role based on the incursions experienced. These buffers correspond to the *hedges* discussed in section 5.4.2.
- Meeting the delivery time constraints: the initial task specification will include an interval denoting the desired period during which unloading at the destination should occur.
- Using a suitable speed for movement: each AGV has a maximum speed depending on its gross weight, state of its battery, *etc*. The planned movement speed should be set slightly below the maximum speed so that there is an option to speed up the AGV should the need arises. The speed and delivery time constraints determine the absolute position of the reserved intervals.

In general, a tasked AGV will move at approximately constant average speed s_1 from an initial position n_0 to a pick-up point n_p, spend a certain amount of time (say ω_p) picking up the load, then proceed at approximately constant average speed s_2 to its destination n_z where it spends some time (say ω_z) unloading. The speed s_2 is normally less than s_1 since the AGV's gross weight is increased after picking up its load.

One of the constraints which the AGV will have to satisfy is to unload within a prescribed interval $[D_{z1}, D_{z2}]$ at the destination. $[D_{z1}, D_{z2}]$ is part of the initial task specification given to the planner, and it is assumed that $\omega_z \leq (D_{z2}-D_{z1})$ or else the AGV planner should report an inconsistent specification to the higher planner. Given these information and the considerations above, how should the intervals be defined?

The first step is to decide on the planned arrival time t_z at the destination. One way is to set it at D_{z1}, the beginning of the prescribed interval. Alternatively, t_z can be set such that ω_z is centred within $[D_{z1}, D_{z2}]$ *i.e.* $t_z = D_{z1} + (D_{z2}-D_{z1}-\omega_z)/2$. In this case, the maximum amounts by which t_z can be advanced or regressed are equal.

Having fixed tz, a point-based schedule can be easily determined by backward projection using the speeds and waiting time along the path. This point-based schedule defines the *planned* arrival and departure times at various check points. An example is shown in Figure 6-3.

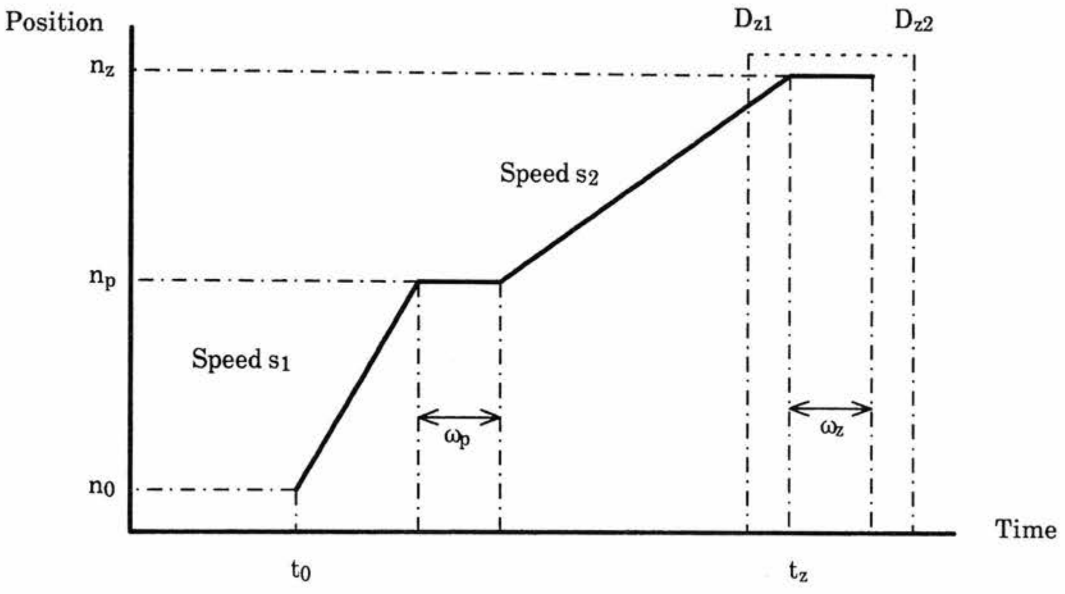


Figure 6-3. A preliminary point-based schedule.

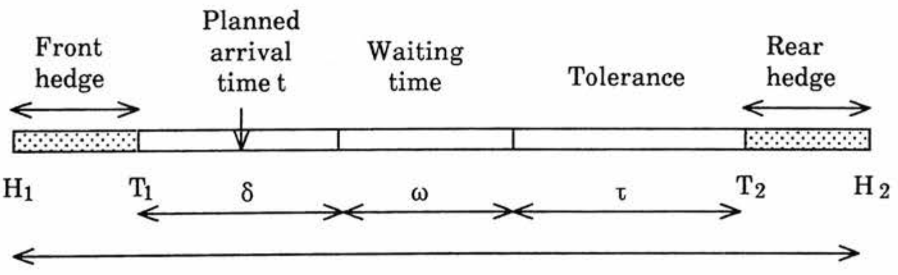


Figure 6-4. Components of a reserved interval.

Finally, the *reserved* intervals for an AGV to be present at the check points along its route can be defined to have the following components (see Figure 6-4):

- A constant initial interval δ which gives some leeway for the planner to change the planned departure time from the start point n_0. This interval exists at all nodes along the path. T_1 denotes the beginning of δ.
- Immediately following δ is a waiting interval ω during which loading or unloading can occur. At most nodes, no waiting interval is required.
- ω is followed by a tolerance interval τ with a width reflecting the likelihood of maximum delay. For example, for every metre travelled from n_0, add 5 seconds, and for every corner turned, add another 5 seconds. At n_0, no tolerance interval exists. T_2 denotes the end of the tolerance interval.
- Two constant width hedges enclosing the juxtaposed intervals defined above; these hedges are defined for all nodes and complete the definition of the reserved interval. H_1 and H_2 denote the beginning and end of the reserved interval respectively.

6.5 Coordinating AGV Movements

The main task in generating a conflict-free set of plans is to ensure that no two AGVs (which pass a common node in such a way that they may collide) have overlapping intervals reserved at the same node. Essentially, three subproblems must be solved:

- Determining whether a collision is possible given the two trajectories.
- Identifying the conflict if the two trajectories may result in a collision.
- Resolving the conflict.

Checking two trajectories for a collision possibility depends on the junctions[†] and the

[†] In the route map, every junction is a node. Other nodes are positions where AGVs can stop for loading, unloading, charging, servicing, and for control purposes.

cornering precision of AGVs. So as not to digress from the current focus on conflict resolution, I shall assume that a collision possibility always exists and defer a more realistic consideration to section 6.7.4.

Adopting the resource-based viewpoint of coordination, a conflict exists when there is contention for the same resource *i.e.* time at a check point. This means that when two reserved intervals overlap, a conflict exists since there is a likelihood of the two AGVs being at the check point at the same time. Note that the overlap may be such that one interval contains the other. Furthermore, a conflict may involve more than two intervals. A resolution scheme must cope with these unusual cases besides the common case of a simple overlap involving only two intervals.

6.5.1 Principal Modes of Conflict Resolution

Any conflict in the form of overlapping intervals can be reduced by either yielding (contracting at one end) or shifting the intervals, or a combination of both. These two basic modes are illustrated in Figures 6-5a and 6-5b.

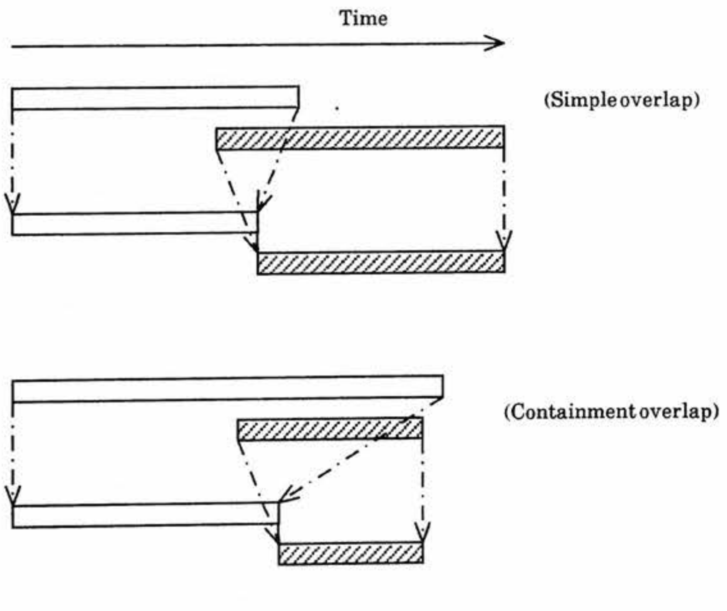


Figure 6-5a. Conflict resolved by yielding.

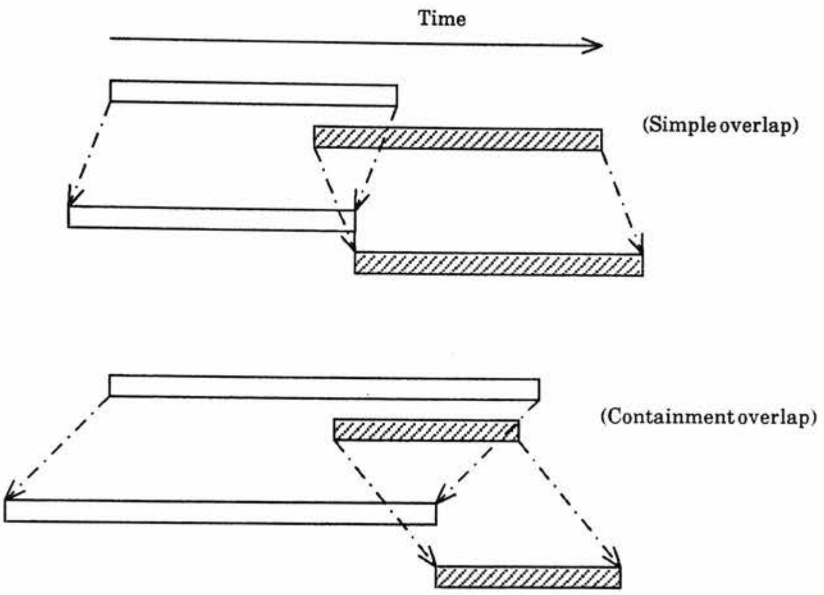


Figure 6-5b. Conflict resolved by shifting enblock.

Yielding helps to preserve the preferred arrival time at the expense of robustness whereas shifting preserves robustness by compromising the preferred arrival time. The preservation of preferred arrival time and robustness are conflicting objectives and hence a judicious combination of the two resolution modes should be used. In any case, the important objective is to maintain a balance: AGVs should compromise their preferred arrival times and plan robustness as equitably as possible. Otherwise, there will be some AGVs with plans more brittle than others and therefore more likely to invoke dynamic replanning.

Observe that in the containment case, resolution can be achieved at either the left or the right end of the containing interval since the conflict can be defined at either end (see Figure 6-6). Obviously, the end with the smaller conflict should be the focus of resolution. Note also that intervals remain unbroken after conflict resolution. An alternative resolution scheme is to split the containing interval into two subintervals. However, splitting is not recommended because it does not reflect the continuous nature of execution deviations. Moreover, it complicates subsequent conflict resolution as well as the scheme which ensures consistency between intervals of the same AGV.

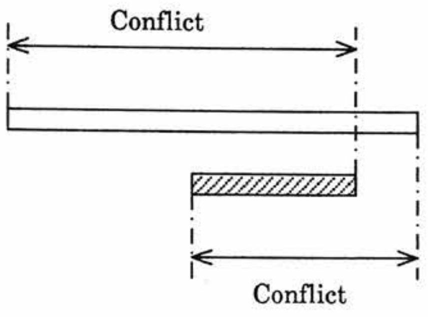


Figure 6-6. Alternative conflict viewpoints.

6.5.2. Implementing the Negotiation Model

A conflict resolution scheme should achieve its effect without causing large disparities in plan robustness. All resultant plans should have their robustness reduced by more or less the same degree as reflected by the average yielding.

An approximately equitable state of compromise can be achieved using the iterative negotiation model described in chapter 5. The model defines the control structure or protocol of negotiation, and in it, every AGV is modelled as an agent with its own set of negotiation techniques. The AGV also has a bottom line of compromise which defines the negotiable parts of the reserved intervals. These negotiable parts can be prioritized so that certain less important parts are sacrificed first.

In general, AGVs can have different sets of negotiation techniques, bottom lines, and priorities of negotiable resources. This may be desirable when tasks have differing priorities. For simplicity of illustration, I will assume that all AGVs are modelled similarly.

6.5.2.1 Bottom Line of Negotiation

The bottom line of negotiation has two parts: (1) the maximal shift allowed; and (2) the maximal yielding allowed. These ensure that compromises do not jeopardize the delivery deadline constraint or overly reduce plan robustness.

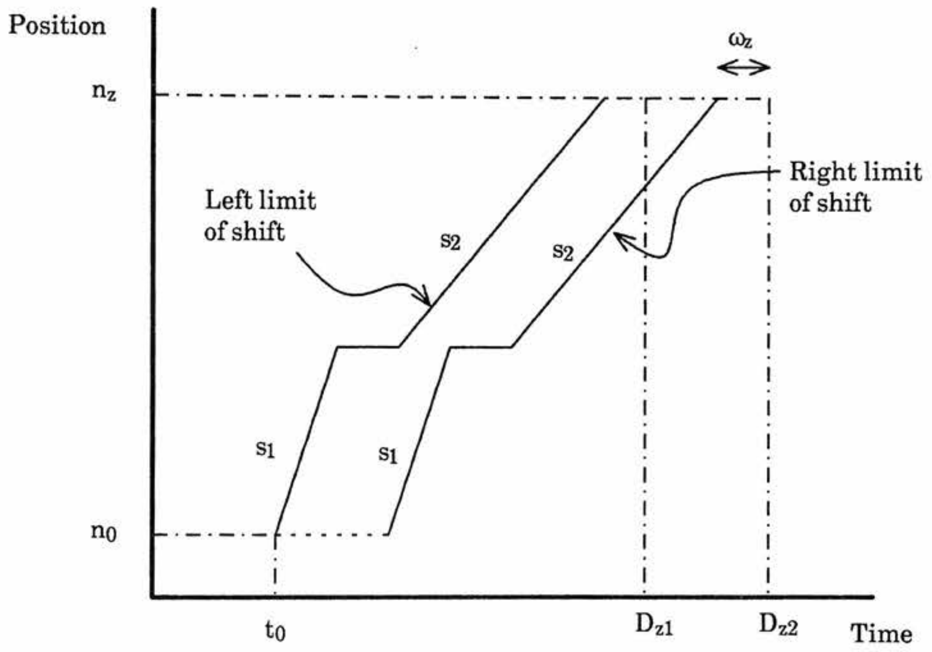


Figure 6-7. The limits of shift permitted.

The maximum shift allowed is determined by a fixed range within which the reserved interval must lie. Suppose the earliest departure time for the AGV from its start position n_0 is t_0; the prescribed unloading period at the destination n_z is $[D_{z1}, D_{z2}]$, the unloading time is ω_z, and s_1 and s_2 denote the maximum unloaded and loaded speeds respectively. As shown in Figure 6-7, the left limit of the range is defined by forward projection from the earliest departure time at n_0. The line shows the earliest possible arrival time at points before the destination. The right limit is defined by backward projection from $D_{z2}-\omega_z$ at n_z. This line shows the latest possible departure time at various points along the route so that unloading at the destination can be completed by

Dz2. If the prescribed unloading period need not be satisfied strictly, then the right limit can be extended further. On the other hand, a more conservative policy might set the limits more tightly.

The maximum yield permitted is an arbitrary fraction of the reserved interval. A simple guideline is to set it at 50% of its initial tolerance interval plus both the front and rear hedges. In setting the bottom line of yield, the consideration should be avoidance of an overly conservative policy which is likely to cause negotiation impasse when the AGVs in dispute refuse to concede further to eliminate a residual conflict.

6.5.2.2 Liberal and Conservative Policies

During conflict resolution, an AGV may shift/yield all or a portion of the maximal amount possible (and required) in a single step. The former follows a liberal policy whereas the latter is more in keeping with a conservative policy.

A liberal policy has the advantage of eliminating the conflict without protracted haggling, but may result in highly unbalanced negotiation. On the other hand, a conservative policy has the advantage of better equity but tends to be protracted and hence time consuming. It is not obvious what the ideal should be, but it should be a moderately sized quantum of compromise which does not entail too many iterations. This can be based on the amount of conflict and the approximate number of iterations tolerable. For example, if the conflict is 200 seconds and about 10 rounds of negotiations should occur, then the maximum quantum of compromise can be set at 20 seconds. In practice, the number of iterations required to resolve the conflict will be more than 10 because there may be rounds when less than 20 seconds are conceded, or even nothing if intransigence is manifested.

6.5.2.3 The Negotiables

The bottom line of yield demarcates the parts of the reserved interval which can be conceded. Recall that, in general, a reserved interval has four components initially. These are: (1) the constant size interval δ for flexibility in instantiating the arrival time; (2) the waiting period ω; (3) the tolerance interval τ to accommodate delays along the way; and (4) the front and rear hedges to help contain a chain reaction of plan revisions.

Suppose that the bottom line of negotiation is as defined in section 6.5.2.1. If a conflict exists at the left (front) end of an interval, then only the front hedge can be yielded and every yield reduces it at most by the computed quantum. If the conflict is at the right (rear) end of the interval, then the components which can be yielded are the rear hedge and 50% of the allocated tolerance interval. Since the yieldable amount is greater at this end, more iterations can occur here before all is depleted.

6.5.2.4 Negotiation Techniques

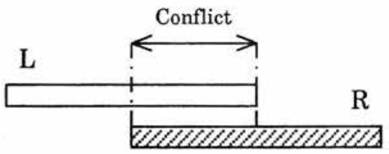


Figure 6-8. The L(eft) and R(ight) intervals

Each agent (AGV) has at least one ordered set of negotiation techniques representing its negotiation skill repertoire. Before describing the basic techniques from which the negotiation skill sets can be constructed, an explanation of the terms used is necessary. The superscripts L and R refer to intervals to the left and right of the conflict respectively (see Figure 6-8). "Left" and "right" refer to temporal relationships on the time line between intervals belonging to *different* agents. I will also use the terms "below" and "above" to refer to spatial relationships between intervals belonging to the *same* agent. For example, if an AGV uses a route (n_0, n_1, n_2, ... n_z), then its interval at node n1 is above that at n0, whereas the interval at n1 is below that at n2. The terms H1, H2, T1 and T2 are defined as in Figure 6-4.

The following lists a possible set of techniques with respect to an interval:

ShiftLeft	Reduce the overlap on its right by shifting $[H1, H2]^L$ to the left without reducing its width.
ShiftRight	Reduce the overlap on its left by shifting $[H1, H2]^R$ to the right without reducing its width.
YieldFrontHedge	Without shifting $H2^R$, reduce the left overlap by yielding as much of the front hedge (increasing $H1^R$) as is required and possible.
YieldEndHedge	Without shifting $H1^L$, reduce the right overlap by yielding as much of the rear hedge (decreasing $H2^L$) as is required and possible.
YieldFrontHedge&Shift	Reduce the right overlap by shifting to the left as much as is required and possible. The shift is made after yielding the front hedge by the amount of shift ($T1^L$, $T2^L$ and $H2^L$ are decreased).
YieldEndHedge&Shift	Reduce the left overlap by shifting to the right as much as is required and possible. The shift is made after yielding the rear hedge by the amount of shift ($H1^R$, $T1^R$ and $T2^R$ are increased).
YieldTolerance	Without shifting $H1^L$, reduce the right overlap by yielding as much of the rear of the tolerance interval as is required and possible ($T2^L$ and $H2^L$ are reduced).
YieldTol&Shift	Reduce the left overlap by shifting to the right as much as is required and possible. The shift is made after yielding the rear of the tolerance interval by the amount of shift ($H1^R$ and $T1^R$ are increased).

MoveLeft&Below	Reduce the right overlap by shifting to the left by the required amount after intervals to the left and below have modified their tolerances and/or hedges to allow such a shift.
MoveRight&Above	Reduce the left overlap by shifting to the right by the required amount after intervals to the right and above have modified their tolerances and/or hedges to allow such a shift.

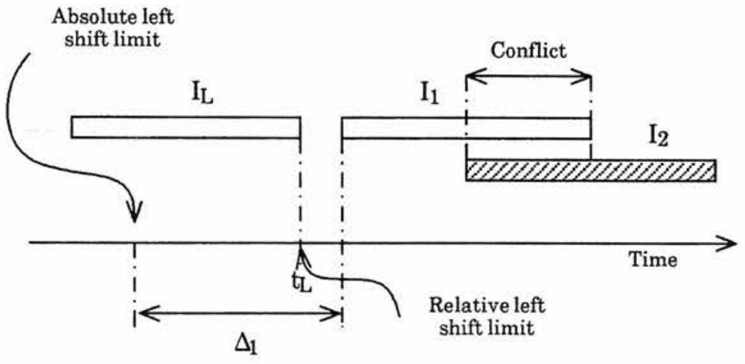


Figure 6-9. Local and absolute left shift limits

Besides a maximum limit by which an interval can be shifted (explained in section 6.5.2.1) the actual shift also depends on other intervals adjacent to the two overlapping intervals. For example, Figure 6-9 shows two intervals I_1 and I_2 in conflict. Although I_1 can be shifted to the left by as much as Δ_1, the ShiftLeft technique will only shift as far as t_L, a limit due to an adjacent interval I_L. Any shifting beyond this requires interval I_L to yield or shift. Changing I_L entails a secondary negotiation process which may lead to further repercussions. This should be avoided if the conflict can be resolved locally by adjusting only the two intervals involved.

For the same reason, the other techniques (with the exception of MoveLeft&Below and MoveRight&Above) which involve shifting are defined such that secondary negotiation processes will not occur. However, in spite of all the yielding and shifting that can be conceded locally, the conflict may not be completely resolved. Only then are the more drastic techniques (MoveLeft&Below and MoveRight&Above) used.

MoveLeft&Below and MoveRight&Above involve other intervals besides the two intervals in conflict. These techniques move the left/right adjacent interval to accommodate a greater shift of the intervals in conflict. Such accommodations entail secondary negotiation processes.

Shifting an interval also affects the arrival and departure speeds which must remain within the permissible range. Otherwise, the intervals above or below it have to be shifted as well. This is another instance of secondary negotiation.

6.5.2.5 Negotiation Skills

The negotiation skills of agents in the model can be represented as *ordered* lists of the negotiation techniques. The ordering corresponds to the order of application such that any compromise begins with the least important resource which can be conceded. When the technique at the head of the list has exhausted its means, it is dropped from the list and the next in line will be applied.

From the ten primitive techniques defined in section 6.5.2.4, several skill sets can be composed. For example, the eight sets used in the AGV planner are:

Ψ_{1a}: (ShiftLeft YieldEndHedge YieldFrontHedge&Shift YieldTolerance MoveLeft&Below)

Ψ_{2a}: (ShiftRight YieldFrontHedge YieldEndHedge&Shift YieldTol&Shift MoveRight&Above)

Ψ_{3a}: (ShiftLeft YieldFrontHedge&Shift MoveLeft&Below)

Ψ_{4a}: (ShiftRight YieldEndHedge&Shift YieldTol&Shift MoveRight&Above)

Ψ_{1b}: (YieldEndHedge YieldFrontHedge&Shift YieldTolerance ShiftLeft
MoveLeft&Below)

Ψ_{2b}: (YieldFrontHedge YieldEndHedge&Shift YieldTol&Shift ShiftRight
MoveRight&Above)

Ψ_{3b}: (YieldEndHedge&Shift ShiftLeft MoveLeft&Below)

Ψ_{4b}: (YieldEndHedge&Shift YieldTol&Shift ShiftRight MoveRight&Above)

The variety of skills defined permits the modelling of:

- Negotiation according to conflict type—left or right relative to the interval.
- Secondary negotiation to accommodate shifting beyond the relative limits.
- A tolerance preservation strategy which aims to maintain robustness by applying shifting techniques first before the yielding techniques. The lists with an "a" in the subscripts of their labels collectively model this strategy.
- A strategy which aims to preserve the preferred arrival time by doing the opposite *i.e.* apply the yielding techniques before the shifting techniques. The lists with a "b" in the subscripts of their labels collectively model this strategy. As in the other strategy, the techniques which initiate secondary negotiations are of last resort in keeping with the objective of minimizing repercussions.

Figure 6-10 depicts the appropriate usage of these skill sets for different intervals involved in the negotiation process. Note that all intervals at the same check point and which are left of the conflict use either Ψ_{1a} or Ψ_{1b}, and those to the right of the conflict use either Ψ_{2a} or Ψ_{2b}. Secondary negotiations below and above an interval should use Ψ_{3a}/Ψ_{3b} and Ψ_{4a}/Ψ_{4b} respectively.

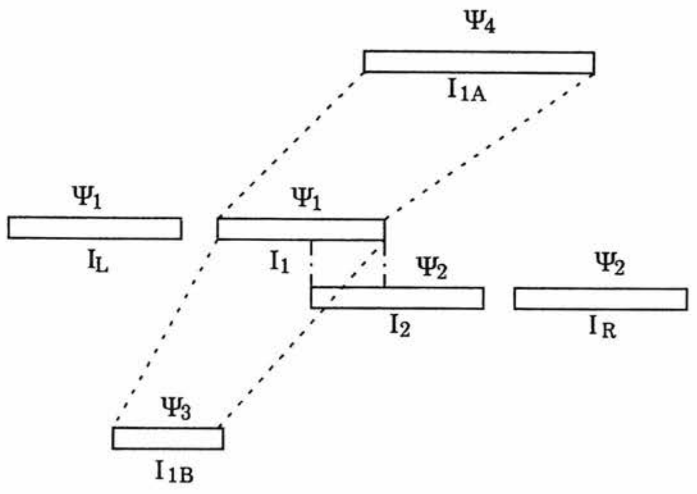


Figure 6-10. Usage of negotiation skill sets.

6.6 Implementation

This section describes the implementation of the AGV movement planner. The implementation has been written in the object-oriented language Loops (Bobrow and Stefik, 1983; Stefik and Bobrow, 1986; Gittins, 1987) and runs in a Xerox 1186 workstation. Examples will also be given to show how the negotiation model works.

6.6.1 Architecture

Each AGV has a team of experts to assist in generating a robust conflict-free movement plan. As shown in Figure 6-11, these are:

- Route Planner

  The route planner is invoked when the AGV is polled with a task. In general, a fetch-and-deliver type of task may have more than one supply point to choose from. The shortest path for each alternative point is computed using the BSL* algorithm (see chapter 4); and the best of these is returned as the bid value associated with the task.

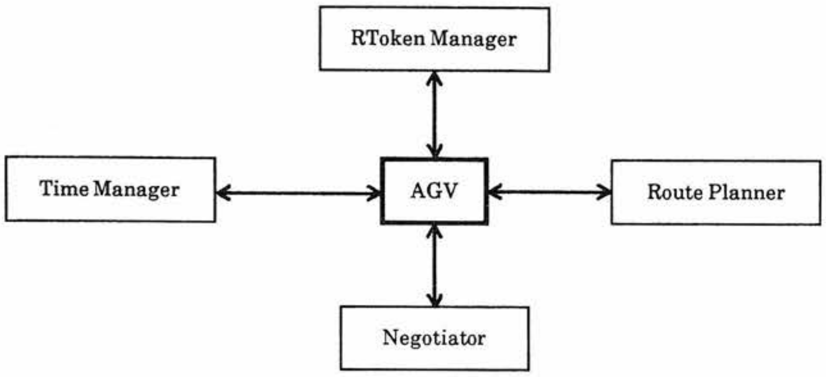


Figure 6-11. Main modules of AGV.

Note that any path computed must be checked for blockages (*e.g.* due to an AGV breaking down). It is assumed that information on any blocked route segments is available. If any path segment is blocked, then route learning is deactivated and the BS* algorithm must be used instead.

- Time Manager

  When the AGV is polled with a task, the time manager identifies all the free periods of the AGV in which there is sufficient time to perform the task. The free periods are found by complementing the current time line (the interval from current time to the end of planning time) with the time periods of the tasks pending execution in the AGV's assigned task queue.

  Since the AGV's positions at the beginning of the free periods usually differ, the distance, and hence the time required to perform the task, will also differ among the free periods. The period which allows the shortest distance is selected as a basis for bidding.

  When the AGV is assigned a task to plan, the time manager defines the reserved intervals at various nodes along the path. During negotiation, the time manager computes the time constraints involved (*e.g.* absolute and relative shift limits) and updates the components of the reserved interval. It also provides the information to guide the application of negotiation techniques.

- RToken Manager

  An RToken is an object (record) which represents the reserved interval. Every node along the path has an associated RToken created by the RToken manager. The RToken manager also has access to a global database of all RTokens established in the course of planning and hence is able to spot conflicts.

- Negotiator

  The function of the negotiator is to resolve conflicts identified by the RToken manager. In general, the skill set adopted by the negotiator will depend on the AGV, task, negotiation policy and conflict type. The policy determines whether a tolerance or arrival time preservation skill set should be used.

6.6.2 Examples

Here is a simple example to show how conflicts can be resolved using the iterative negotiation model. Figure 6-12 shows two simple plans involving two AGVs X and Y. X's plan is to move along the path (9 10 11) and Y's plan is to move along (12 10 13). X planned first and did not require any conflict resolution since no other intervals existed at nodes along its path. The values of its RTokens are:

R1	[198.75, 258.75]
R2	[217.5, 289.0]
R3	[248.33, 311.33]

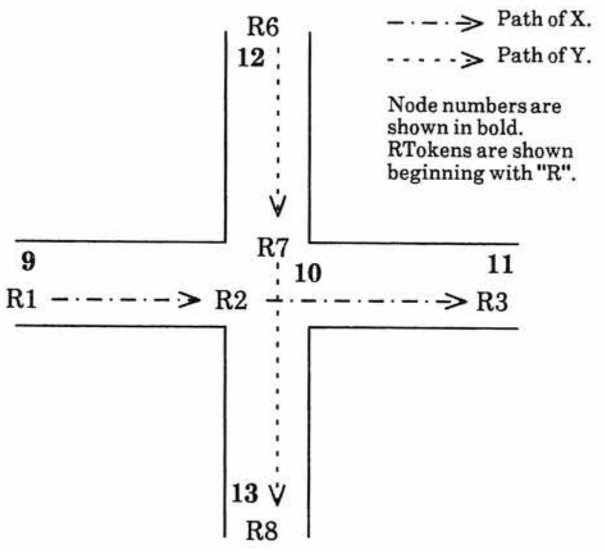

R6
12
R7
10
9
11
R1
R2
R3
13
R8
Path of X.
Path of Y.
Node numbers are shown in bold.
RTokens are shown beginning with "R".

Figure 6-12. Disposition of RTokens.

As Y developed its plan next, it established the following RTokens:

R6	[240.42, 301.92]
R7	[259.17, 332.17]
R8	[290.0, 364.5]

Y's RToken manager checked the nodes along its path for a possible conflict. It found that the only co-located RTokens are X's R2 and its R7. Since these RTokens' intervals overlapped, Y's negotiator initiated a negotiation process with X's negotiator. A trace of the negotiation process as generated by the planner is shown in Figure 6-13. Both negotiators used shifting techniques before the yielding techniques since both had selected the tolerance preservation strategy*i.e.* strategies Ψ_{1a} and Ψ_{2a} were used by R2 and R7 respectively. Observe that before R7 could apply YieldEndHedge&Shift, R8 had to shift to the right so that Y's departure speed from node 10 could remain within the speed limits. This is an instance of nested (secondary) negotiation. Furthermore, since R8 lies downstream from R7, its relevant tolerance preservation strategy is Ψ_{4a} (see Figure 6-10).

```
R2 (217.5 289.0) conflicts with R7 (259.17 332.17)
     Amt of conflict: 29.83
At node 10, R7 will negotiate with R2 on its left.
R7    ShiftRight:
        Conceded:  4.17    (259.17 332.17) -> (263.33 336.33)
         Residue:  25.67
R2     ShiftLeft:
        Conceded:  2.08    (217.5 289.0) -> (215.42 286.92)
         Residue:  23.58
R7    YieldFrontHedge:
        Conceded:  10.0    (263.33 336.33) -> (273.33 336.33)
         Residue:  13.58
R2    YieldEndHedge:
        Conceded:  10.0    (215.42 286.92) -> (215.42 276.92)
         Residue:  3.58
R7    YieldEndHedge&Shift:
R7 will move R8 above.
[Move Above R8] ShiftRight
        Conceded:  3.58    (273.33 336.33) -> (276.92 336.33)
         Residue:  0.0
```

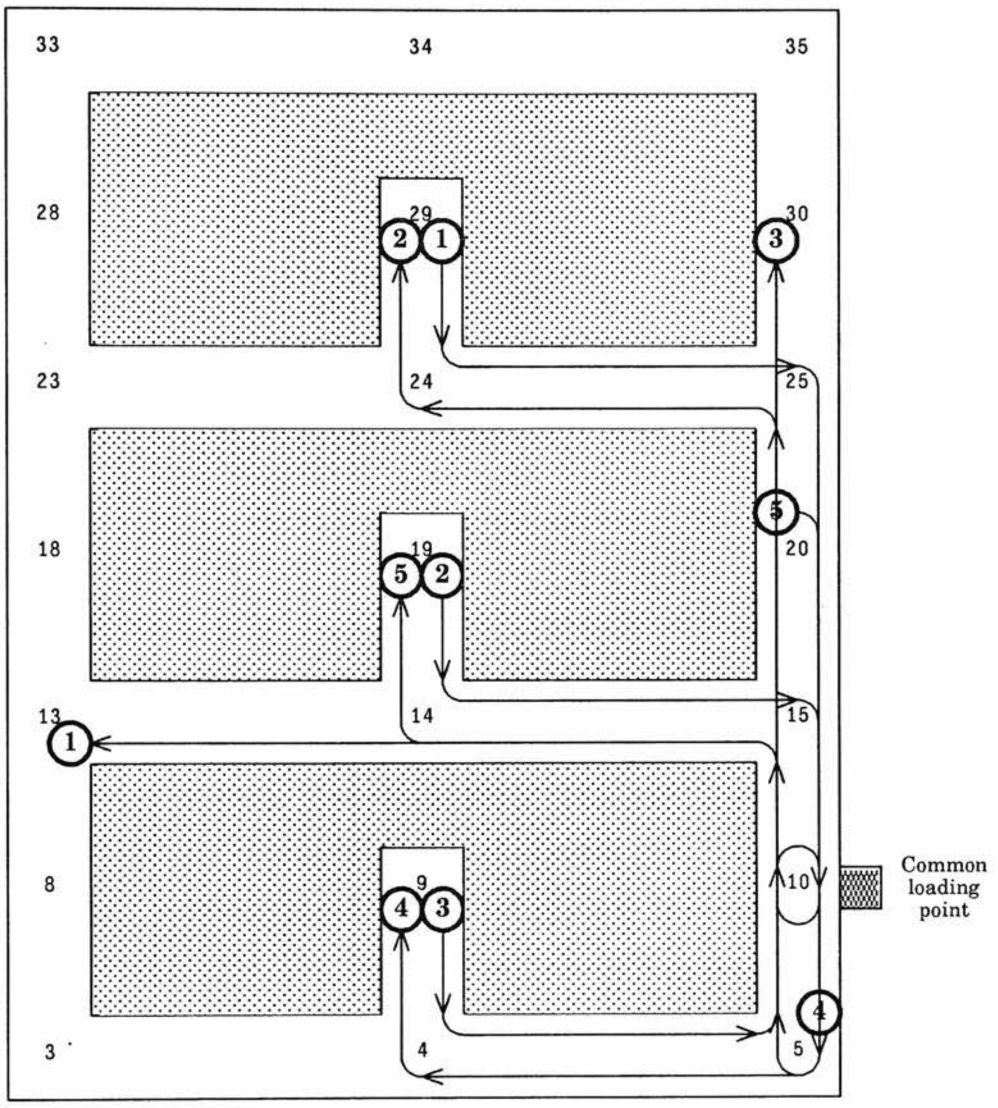
Figure 6-13. Trace of a negotiation process.

In another example, we have the following problem scenario (see also Figures 6-14 and 6-15):

Tasks to perform:

	Destination	Unloading period
Task1	9	[05:00, 06:00]
Task2	19	"
Task3	29	"
Task4	30	"
Task5	13	"

This non-trivial example was contrived such that all the AGVs had to proceed to the same loading point at about the same time, a situation in which many conflicts had to be resolved. As evident from the complete negotiation trace (see appendix D), later planning entailed more nested negotiations as more competing intervals were reserved at the nodes. As expected, strategies which employ yielding techniques first tend not to incur as many instances of nested negotiation as those which invoke shifting techniques first. This is because shifting techniques have to reclaim some time at one end of an



Uncircled numbers are the nodes defining the route map.
Bold circled numbers denote the AGVs

Paths of AGVs:

AGV1 ... (29 24 25 20 15 10 15 14 13)
AGV2 ... (19 14 15 10 15 20 25 24 29)
AGV3 ... (9 4 5 10 15 20 25 30)
AGV4 ... (5 10 5 4 9)
AGV5 ... (20 15 10 15 14 19)

Figure 6-14. Trajectories of five AGVs.

interval after yielding at the other end and this reclaiming may impinge upon other intervals, resulting in secondary negotiations. Despite many rounds of negotiation required, all the tasks could be achieved within the delivery time constraints.

	AGV1	AGV2	AGV3	AGV4	AGV5
Initial position:	29	19	9	5	20
Task assigned:	Task5	Task3	Task4	Task1	Task2

All the tasks required the AGVs to pick up the loads at node 10.

Figure 6-15. Initial disposition of AGVs.

Another observation is that negotiators which have made previous concessions tend to behave intransigently. The intransigence is exhibited when a technique is selected and all of its associated means have been previously conceded. In this case, the negotiator makes no concession (in the trace, this is shown by the word "unchanged") and passes the responsibility of conceding next to the other agent. It may happen that the other agent is also somewhat depleted of resources which it had conceded earlier in the negotiation process. In this case, both agents will behave intransigently for a few rounds until different techniques are invoked with available negotiable resources. In this way, fairness is achieved to some extent: an agent close to its bottom line will not be forced to concede unless the other is similarly impoverished.

All in all, the planner took about 54 seconds to compute the set of movement plans shown in appendix E. These movement plans are conflict-free as was verified by observing an animated execution using a simulator module developed as part of the planner.

6.7 Discussion

6.7.1 Passage Time

Having a finite length, an AGV will take some time to pass a node. For greater modelling accuracy, this passage time should not be ignored. It is measured from the moment the front of the AGV reaches a node to the time its rear leaves the node. Hence, if s is the length of the AGV, u is the initial speed at which the front of the AGV leaves the node, and a is its acceleration (assumed constant), then the passage time t is determined by solving the motion equation $s = ut + at^2/2$.

In some applications, an AGV may haul a "train" of unpowered trailers and its effective length is the total length of the train including the AGV. The same equation applies in computing its passage time except that the effective length is used instead.

The way to account for the passage time t is to extend the interval by pushing back the departure time without changing the arrival time. It can thus be viewed as a kind of non-negotiable waiting time.

Note that a train might occupy more than one node at the same time and consequently, reserved intervals within a single plan would overlap. However, this does not affect the way the planner works since such overlaps (being at different locations) do not pose a resource contention problem.

6.7.2 Soundness

The movement plans generated are sound if no collisions will occur (theoretically) at any point. This can be checked by examining the safety conditions at junctions and between junctions.

Safety at a junction is guaranteed since the intervals reserved at the junction are disjoint. However, this does not imply safety along a path segment. For example, Figure 6-16 shows two pairs of reserved intervals of AGVs X and Y at nodes n1 and n2. X arrives at and departs from n1 before Y but arrives at n2 after Y. Clearly, these intervals are conflict-free at the nodes. However, because the order of the arrival is reversed in going from n1 to n2, somewhere between n1 and n2, Y will run into X unless Y can overtake X safely. On the other hand, if the order of arrival of any two AGVs moving in the same direction along the same path segment remains invariant, then overtaking is unnecessary and safety can also be guaranteed along the path segment.

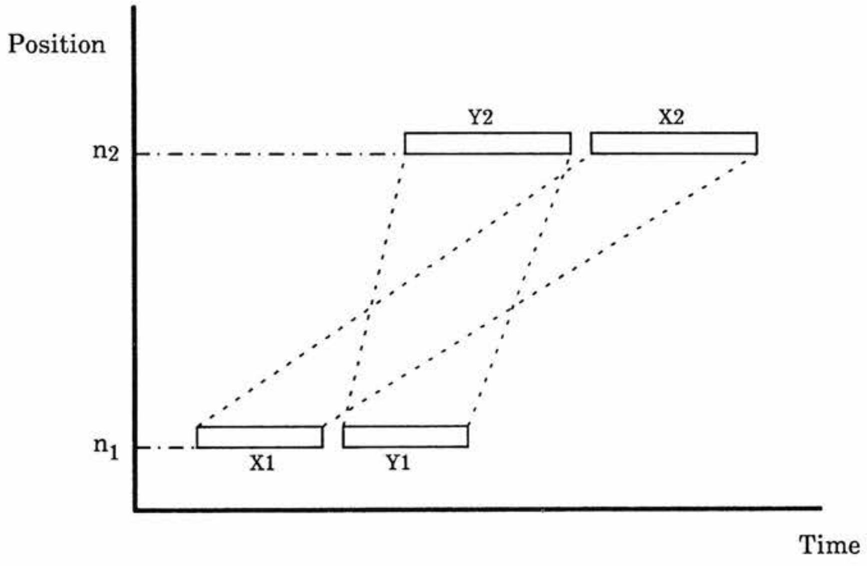


Figure 6-16. Collision along a path segment.

Complete safety thus depends on two conditions: (1) non-overlapping intervals where trajectories may interfere; and (2) invariant arrival order in any two pairs of intervals of two AGVs proceeding in the same direction along a path segment. The planner ensures both are satisfied and thus guarantees that the movement plans generated are sound.

6.7.3 Arbitration

Arbitration may be useful in the event of negotiation impasse. Recall that according to the top level negotiation control structure (see Algorithms 5-1 to 5-4), when conflict resolution fails, local replanning is first attempted and if this also fails, then an entire alternative plan is sought. If this is also in vain, then the planner can look for another AGV to plan for the task; an AGV which may be better predisposed to find a feasible plan for the task. All these can be at much computational expense and undesirable if arbitration could have overcome the impasse in the first place. However, there is no guarantee that arbitration will always succeed, and if it fails, then the arbitration effort would have been an added burden.

The essential characteristic of arbitration is that it dictates settlement terms to the plaintiffs. In this application, arbitration involves forcing one or both AGVs to concede beyond their bottom lines. The disadvantage of this is that it weakens the robustness of plans.

6.7.4 False Conflicts

Conflict resolution is achieved at the expense of computation time and plan robustness. The time it takes can be significant when nested negotiations are necessary. It achieves its goal of safety by trading off plan robustness. Since these are costs not to be taken lightly, any opportunity to avoid resolution without sacrificing safety should be exploited.

This is possible if certain trajectories can be identified as safe (under certain assumptions) even though the intervals overlap at a node. For example, Figure 6-17 shows some combinations of trajectories passing through a junction which are safe regardless of the associated time intervals. The assumptions which must hold are: (1) there is enough room for safe passage by two AGVs; (2) the AGVs have the requisite turning precision; and (3) the AGVs obey a standard turning convention *e.g.* turning in a clockwise direction with a certain turning radius. Although these assumptions may not be realized in practice, some combinations of trajectories can nevertheless be accepted as collision-free (*e.g.* combination B).

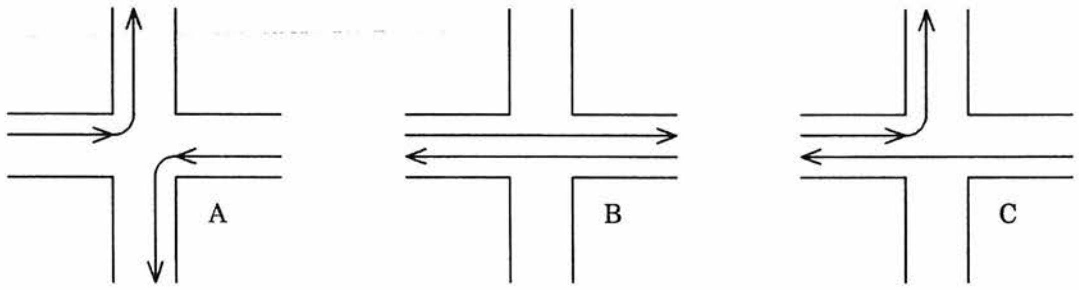


Figure 6-17. Safe trajectories.

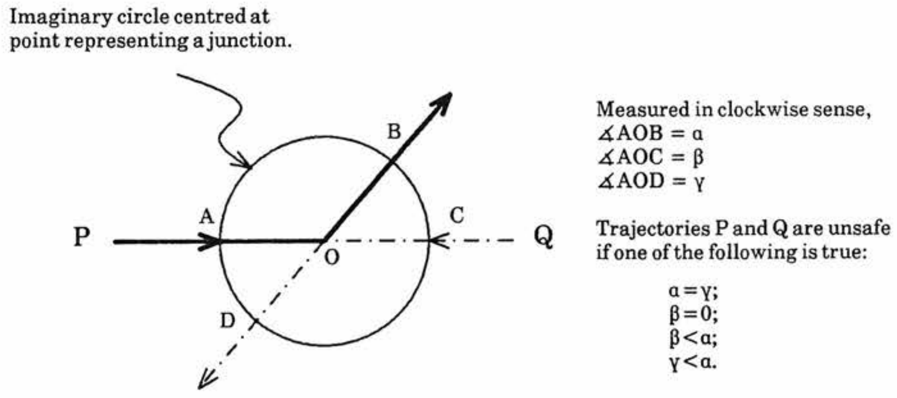

Imaginary circle centred at point representing a junction.
P
A
B
C
Q
O
D
Measured in clockwise sense,
∡AOB = α
∡AOC = β
∡AOD = γ
Trajectories P and Q are unsafe if one of the following is true:
α = γ;
β = 0;
β < α;
γ < α.

Figure 6-18. Trajectories at a point representation of a junction.

Since there may be many types of junctions of different topology and number of branches, it is useful to have a computational procedure for determining safety given any two trajectories. A procedure which achieves this objective rests on the critical constraint that the two trajectories must not touch or intersect each other. Clearly, this is sufficient to guarantee safety regardless of the associated reserved intervals.

Informally, the procedure is to select arbitrarily one trajectory, say P (see Figure 6-18), set its approach direction as the reference line from which the angles of approach/departure paths are measured in the clockwise direction. Imagine there is a circle centred at the node denoting the junction. The angles of the approach/departure paths are angular coordinates of the points where the approach/departure paths meet the circumference of the circle. The angles α, β, and γ must be in the range [0, 360]. If any one of the relations shown in Figure 6-18 is true, then the trajectories are not safe.

6.7.5 Dynamic Replanning

Although interval-based movement plans are robust, the need for dynamic replanning cannot be excluded totally. Hence provision must be made for a dynamic replanning capability.

Dynamic replanning is not much different from planning prior to execution. In both, a new plan must be constructed in the context of existing plans and conflicts are similarly resolved. This suggests that the movement planner can be adapted for replanning as well.

In general, a planner has more than one task to plan and it may choose to generate the plans for these tasks in a parallel manner in which it works out a bit of one plan, switches its attention to another, and so on until all the plans have been completed. A reason favouring such an approach is that it is amenable to the principle of tackling the more difficult parts of a problem first.

An alternative approach is to plan for one task at a time. In this approach planning is always within the context of a set (possibly null) of completely planned tasks. This sequential approach is simpler to implement and conceptually easier to appreciate. Its advantage is that a planner based on it is also capable of both dynamic planning and replanning[†] since the planning circumstances are essentially similar *i.e.* a new/revised plan must be constructed in the context of existing plans.

Since the control strategy of the AGV movement planner takes the sequential approach, it readily meets the replanning role as well. As a conventional planner, it is activated by the task manager. As a dynamic replanner, the activation originates from the execution manager instead. Notwithstanding this difference, the planning functions and conflict resolution mechanism remain the same.

During execution, dynamic replanning will begin benignly before it calls for more radical actions which may require replanning on the part of other AGVs. When the execution monitor detects or anticipates a different arrival time, the initial remedial action is to change the speed such that subsequent deviations from the schedule are minimized. Speed variation does not alter any of the intervals and hence preserves the safety of all the plans. If in spite of speed modifications, the arrival time at a later node is expected to be outside the reserved interval, then the interval has to be shifted. This is where dynamic replanning makes use of the same negotiation mechanism if the shifted interval intrudes into another interval.

If speed variation obviates any shifting of the interval but nevertheless leaves the AGV behind schedule, then an opportunity arises for shrinking the reserved intervals at later check points. By doing so, reserved resources which are no longer required are relinquished and made available to other AGVs during subsequent planning.

Another event which warrants replanning is an AGV breaking down during execution. The remedy for this is either to replan afresh for the affected task or modify the task such that another AGV takes over the load from the AGV which came to grief. The latter event is dependent on the fact that the AGV broke down while carrying the load and

[†] Whereas dynamic planning refers to the construction of a new plan for a task received during execution, dynamic replanning refers to the modification of a defunct plan for an existing task during execution.

that it is possible to effect load transfers between AGVs.

6.7.6 Termination

In distributed agent environments with possibility of conflict, questions of divergence and deadlock must be examined to be sure that computational processes will terminate. Here, we will see that neither divergence nor deadlock will occur and hence negotiation processes will definitely terminate.

6.7.6.1 Divergence

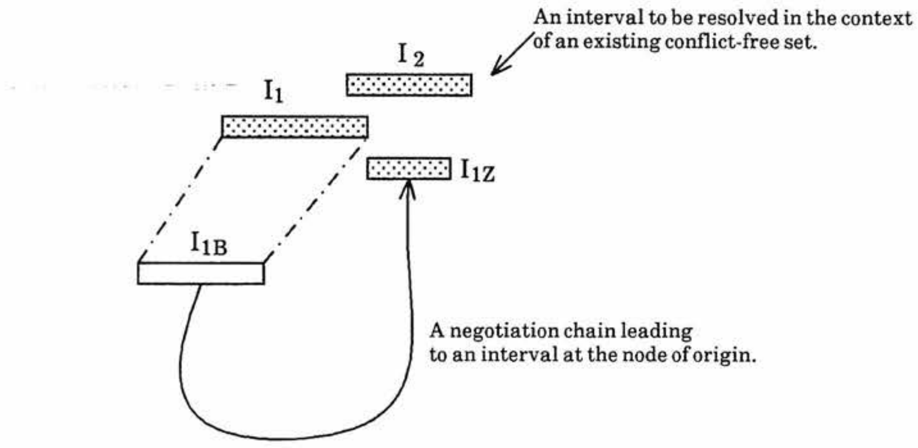
Divergence here refers to the case of a negotiation process always increasing a conflict rather than reducing it. If this happens, the conflict will not be resolved, and the process can only be terminated if its divergent behaviour is detected and aborted.

An approach which avoids divergence altogether rests on the observation that divergence can only occur if there is a technique which increases the conflict instead of reducing it. Hence if all techniques in a skill set are defined never to increase a conflict, no divergence can possibly occur.

Referring to Figure 6-10, divergence will not occur if I_1 does not apply either ShiftRight or MoveRight&Above—two techniques which moves the right end of an interval to the right which would increase the conflict between I_1 and I_2. This restriction is enforced by excluding the forbidden techniques from the skill sets I_1 will ever use. Similarly, any interval such as I_2 with a conflict on its left to resolve, must use skill sets which exclude the techniques ShiftLeft and MoveLeft&Below. Since these conditions are satisfied in the definitions of the skill sets (section 6.5.2.5) used in the movement planner, divergence is avoided altogether and schemes to detect and subsequently abort the process are unnecessary.

6.7.6.2 Deadlock

Deadlock refers to an interval waiting for another to concede before it makes a concession, but the concession for which it is waiting for cannot occur until it concedes. This cyclic wait is the general characteristic of the deadlock phenomenon. It is a problem which has been well investigated by operating systems pundits (Peterson and Silberschatz, 1985; Finkel, 1986).


An interval to be resolved in the context of an existing conflict-free set.
I_2
I_1
I_{1Z}
I_{1B}
A negotiation chain leading to an interval at the node of origin.

Note: The shaded intervals are at the same node.

Figure 6-19. A hypothetical deadlock situation.

A hypothetical example of deadlock in the AGV movement planning context is shown in Figure 6-19. A conflict occurs between I_1 and I_2. Suppose I_1 can only shift to the left if I_{1B} is shifted by some amount to the left. I_{1B} may initiate further negotiation processes if its shift is conditional on another interval's concession. Suppose the negotiation chain leads back to an interval I_{1Z} at the same node as I_1. Before I_{1Z} can shift to the left, it requires I_1 to do likewise. A deadlock results since each interval along the chain is perpetually waiting for the next interval to concede first. Similarly, a hypothetical case can be contrived in which all intervals in a chain are waiting for the next to shift right before it can do likewise.

Fortunately, these deadlock situations will never occur in the AGV movement planner. Informally, the proof of this can be established by noting that any interval required to shift left has a planned arrival time which is to the left of the initiator's. Proceeding transitively along the chain of initiations, I_{1Z}'s arrival time must be to the left of I_1's. Since I_{1Z} is initially conflict free in relation to I_1, I_{1Z} must be to the left of I_1, contradicting the deadlock condition illustrated in Figure 6-19. Since I_{1Z} is on the left side of I_1, a shift by I_{1Z} to the left will never require I_1 to do likewise. By the same line of reasoning, it can be proved that a right shifting type of deadlock can never occur.

Note that a negotiation impasse in which neither is able to concede further is not a deadlock situation since waiting states in a deadlock imply that concessions are possible. Unlike deadlocks, negotiation impasse can occur. However, Algorithm 5-3 detects an impasse and ensures that the planner continues with the next step.

6.7.7 Limitations

The present planner has several limitations which are addressed below. Overcoming some of these limitations is largely a matter of further programming effort to improve the planner beyond its present feasibility demonstration level.

- Narrow bidirectional segments

  It is assumed that narrow bidirectional segments do not exist. These are segments which cannot accommodate two AGVs abreast but allow an AGV to pass in either direction. In order to include these segments, the planner must ensure that the entire segment is reserved for the duration in which any part of the AGV is along it. In this case, resources are tied to entire segments rather than just the two nodes defining the segment. Effectively, a block control strategy is required.

A possible way to deal with this is to treat the segment as a special node at which a mandatory waiting time is imposed, this being equal to the time of occupation along the segment.

- Communication bandwidth

  The chattiness of iterative negotiation poses a serious communication problem if the negotiator and RToken manager modules are modelled within a computer on board the AGV. This would require frequent transmissions which not only has a bandwidth limitation, but is also subject to interference. These problems can be overcome by modelling the AGV within a central host computer, permitting a faster and completely reliable method of message passing to be used in place of the transmissions during a negotiation process. Instead of making frequent transmissions, only the final plan data/changes need to be transmitted to the physical AGVs.

- Overtaking

  The system does not permit overtaking between nodes. If an AGV is allowed to overtake another AGV which is moving slower or has come to grief, then extensions are needed to plan a safe trajectory for overtaking. This is a non-trivial task since collision possibilities in both the direction of travel and the opposite direction must be considered. It also assumes that the overtaking AGV is of the free-ranging category with the requisite manoeuvrability.

- U-turns

  It is assumed that AGVs can execute "U-turns" at nodes along broad (*i.e.* wide enough to accommodate two AGVs abreast) bidirectional segments. U-turns may be necessary at loading and unloading points. Although AGVs may have a turning radius small enough to execute a successful U-turn, it is unlikely that a train of unpowered trailers will follow the turn safely.

A possible solution where U-turns are not allowed is to impose a constraint on the route planner such that the successor nodes expanded during the search do not impose a U-turn.

- Congestion

  The instances of conflict can be reduced if routes are chosen which avoid congested segments or junctions. Another advantage of less contention is better executability of plans: if an AGV is the sole user of a segment, any replanning will not involve other AGVs which will be at the same location.

  It can be envisaged that a planner which considers traffic congestion will maintain an account of resource contention at various nodes. The optimal criteria of routes will include traffic congestion as well as distance. Two approaches are possible: (1) use a composite heuristic function; and (2) retain the distance heuristic function but reject the shortest path found if it is considered too congested, and continue to look for the next shortest and so on until a shortest acceptable route is found. The advantage of the former is that search terminates once a route is found. Its disadvantage is that it is not clear how a composite heuristic function satisfying the monotone restriction can be defined. This constraint is necessary for BS* and BSL* to be applicable.

- Backtracking

  The current planner does not implement the limited backtracking at the top level of the planning algorithm (lines 8 and 10 of Algorithm 5-1). Backtracking at this level would first modify the route locally (*i.e.* segments where the conflicts could not be resolved are substituted); and if this recourse is also in vain, then a new route is selected. Implementing this backtracking scheme requires AGVs to make tentative commitments during negotiations. This means that any shifting or yielding of intervals are only confirmed when a plan has been successfully negotiated; otherwise, the AGVs involved must retain the state of their reserved intervals prior to negotiations. Furthermore, the substitution of segments or route should proceed on condition that the increase in distance must not exceed a threshold (*e.g.* the distance which the best alternative AGV would have to travel). If this threshold is exceeded, then it might be better to re-assign the task.

- Completeness

  The planner does not possess the completeness property *i.e.* it does not guarantee that a solution will be found if it exists. It fails to be complete because it does not examine all possibilities of conflict resolution. In fact, this is practically impossible since there are infinite possibilities for reducing an overlap between two intervals.

6.8 Related Work

6.8.1 Operations Research Methods

Much work has been done in the operations research community to solve vehicle routing and scheduling (VRS) problems. In its most general form, the VRS problem is: given m depots, n delivery points, a fleet of k vehicles with known capacities, find a set of routes and vehicle assignments to satisfy an overall objective *e.g.* minimum total cost. Variations of the VRS problem differ in the constraints imposed (*e.g.* maximum travel time, maximum distance allowed per vehicle).

Several heuristic methods have been developed which aim to produce efficiently, feasible but not necessarily optimal solutions. Lin and Kernighan (1973) proposed an algorithm which incrementally improves an initial feasible solution for the single-depot case which can be recast as a travelling salesman problem (TSP). Clarke and Wright (1963) developed a cost-saving approach which has since spawned a few variations with different savings measures (Gaskell, 1967; Yellow, 1970; Holmes and Parker, 1976).

The basic idea is to have all delivery points assigned a separate primitive route and iteratively merge routes which accrue savings.

Gillet and Miller (1974) proposed the SWEEP algorithm which generates the routes sequentially instead of concurrently as in the above methods. Delivery points are first clustered by sweeping a line anchored at the depot. The points swept are assigned to a vehicle until a constraint is reached (*e.g.* capacity exceeded). When this happens, subsequent delivery points swept are assigned to the next vehicle and so on. This method follows from the observation that in many cases, the single-depot VRS problem has a petal-like solution with little or no overlapping of adjacent circuits.

Methods which seek optimal solutions such as the branch-and-bound algorithms (Christofides *et al*, 1981a, 1981b) are computationally demanding and limited to problems with simple constraints. Many of the methods addressing the single-depot case have been extended to handle multiple depots (Wren and Holliday, 1972; Tillman and Cain, 1972). All of these approaches nevertheless have limited scope. Each can only be applied for a particular situation. Some difficulties which these methods do not address are:

- Vehicles may not be depot-based *i.e.* their starting and finishing positions are not at the depot and need not be fixed. This requires a solution to an additional vehicle-depot assignment problem.
- Detailed movement timings are not considered nor planned for in the vehicle schedules since the main objective is efficient distribution. The problem of collision avoidance during movement is assumed irrelevant with manned vehicles.
- The problem is also assumed to be time-invariant and hence solutions are supposed to be valid at all times. The methods address only routine situations rather than dynamic problem scenarios.

The lack of generality in these methods was noted recently by Bott and Ballou (1986) who also pointed out the shortcomings in these traditional approaches and argued for a generalized vehicle routing and scheduling methodology which can address many of the real world restrictions.

6.8.2 Coordination

6.8.2.1 Wesson's Work

Wesson (1977, 1981) developed a planner which generates a conflict-free sequence of air traffic control instructions to be issued to aircraft in the jurisdiction of an enroute air traffic controller. The sequences are generated by simulating the movements of all aircraft in the environment for the next 20 minutes to uncover dangerous events. For each such event spotted, beginning with the one of highest priority, all possible responses culled from a set of event-response production rules, are simulated separately, sprouting new branches in the search tree. After a certain depth of search, the best state, evaluated according to some criteria, is chosen and the corresponding sequence of instructions to reach this state constitutes the air traffic control plan. The plan is then executed until an unforeseen dangerous event occurs or 10 minutes have elapsed when the planning process is repeated.

Wesson's planner works more like a replanner than a planner since it fixes flight plans which are established *a priori*. Given these, it determines the air traffic control instructions to impose the necessary flight plan changes whenever dangerous events are anticipated. Its anticipative capability is limited to a 20 minutes look-ahead and presumes that ample time is available for corrective action to be implemented.

Since its schedules are point-based and are not necessarily conflict-free prior to execution, dynamic replanning can be expected to be frequent. Furthermore, its approach of planning by simulation is limited to a chronological order of conflict consideration and is unable to exploit possible advantages of an examination according to criticality of conflict.

6.8.2.2 AUTOPILOT

Thorndyke *et al* (1981) designed AUTOPILOT which is a planner for a single aircraft working in a multi-agent environment. Similar AUTOPILOTs exist for other aircraft. No central air traffic control entity is involved. The objective of AUTOPILOT is to design a plan to navigate the aircraft safely across an airspace.

AUTOPILOT first retrieves a set of candidate plans from a library of plans, using as keys the origin and destination in the airspace. These are then examined and evaluated in relation to the plans of other aircraft. The best conflict-free plan is chosen for execution. If none is found, the plan with least conflict is chosen for patching. AUTOPILOT will attempt to patch its own plan first, using a set of predetermined patches as in Wesson's system.

AUTOPILOT has four alternative strategies for fixing a flawed plan. Two of these involve only self-modification and other agents are not required to modify their plans. The first of these does not involve communication since its candidate plans are examined in relation to the inferred plans of other aircraft. In the second strategy, instead of inferring others' plans, the aircraft is informed about these plans after asking. Hence there is no risk of misinterpretation as in the first strategy.

The other two strategies require other aircraft in conflict to patch their plans if self-modification fails to resolve all conflicts. In the third strategy, based on the best plan, aircraft in conflict are requested sequentially to modify their plans. Along with the request, plans of the requesting and previously requested aircraft are forwarded, enabling modification to proceed with up-to-date information. The fourth strategy differs in that all the initial candidate plans which remains flawed after self-modification are communicated to the aircraft in the conflict set. Plan patching then proceed simultaneously and independently, and results are reported back to the requesting aircraft along with the planning assumptions. The initiator is responsible for monitoring the amendments, and will halt further patching efforts once a conflict-free plan emerges which does not jeopardize the plans of other aircraft.

Note that AUTOPILOT does$\overset{\text{not}}{\wedge}$generate its initial plans. Instead, these are retrieved by looking up a library of predetermined plans. Apparently, this is because AUTOPILOT's main research focus is dynamic replanning. As in Wesson's case, the planning occurs during execution and not before. By not generating in the first instance a set of conflict-free plans, it too will have to do more fixing during plan synthesis. It also makes no mention of the important issue of timeliness: how can dynamic replanning be performed as fast as possible to avert impending disasters.

6.8.2.3 Organizational Structuring

Corkill and Lesser (1979, 1983) proposed the concept of organizational structuring for coordinating problem solving in a distributed network. Each problem solving node in the network uses a compound blackboard architecture with three or four blackboards.

There is a conventional data blackboard on which intermediate and final hypotheses are posted as in the HEARSAY II system (Erman *et al*, 1980). A second blackboard containing the system's own goals and communicated goals of other nodes is used by the planning component of the node. The third is the organizational blackboard with data structures which can be externally set to influence the node's problem-solving behaviour and to define its relationships with other nodes. A fourth blackboard may also be used to help focus the behaviour of the local node. This permits organizational roles to be evaluated for acceptance or rejection in view of suggestions arising from local processing and interaction with other nodes.

The network architecture can take various forms as decided by the designer. For example, one could have a lateral architecture in which all nodes can communicate directly with one another. Alternatively, in a hierarchical structure, there will be supervisor-worker links which define the communication possibilities. The advantage of Corkill and Lesser's approach is that the architecture chosen can be adapted to the problem structure by defining the data structures appropriately in the organizational blackboards.

Although the blackboard approach provides a general problem-solving paradigm, its success is highly dependent on proper design of the control module and the encapsulation of problem-solving expertise into knowledge sources. Generally, it is more appropriate for domains in which diverse and competing expertise exist and must be brought to bear on the problem.

6.8.3 Planning with Time Constraints

6.8.3.1 DEVISER

Vere's DEVISER (1983) is the first hierarchical planner which considers time constraints. It uses two types of time constraints tagged to activities, events and goals: an interval (window) to represent the permissible range of start times and a single-valued duration. Plan generation follows the link-expand-resolve cycle as in Sacerdoti's NOAH (1975) and Tate's NONLIN (1977). Node expansion or ordering, and variable instantiation cause an outward propagation of window compressions across the partially ordered network. During this process, updated windows are checked for validity *i.e.* earliest start time $\leq$ latest start time. If this constraint is not satisfied, the changes are aborted and DEVISER backtracks to the last choice point. Temporal constraints are also useful for pruning the search space. The original DEVISER has since been upgraded to DEVISER II (Vere, 1985) which incorporates two new features—temporal scope of assertions; and window cutoff—to improve planning time substantially.

6.8.3.2 Bell and Tate's Work

Bell and Tate (1984, 1985) extended Vere's use of temporal constraints to include intervals for finish times, durations and inter-sequence wait times. The constraints are reformulated as a longest path linear programming problem.

If a temporally valid solution exists, it gives the set of earliest/latest start/finish times delimiting the intervals of each activity. The richer set of constraints extends the representational repertoire, giving greater applicability to real world problems. Another advantage of incorporating more aspects of temporal constraints is that invalid plans can be uncovered earlier, hence reducing wasteful search.

As in Vere's DEVISER, the plans generated normally pertain to a single agent. Complex issues of interleaving multi-agent plans safely and with a view towards robust execution are not addressed.

6.8.3.3 FORBIN

In Miller (1985) and Dean (1985a, 1985b), FORBIN is described as a planner which generates a sequence of movements and actions for a mobile robot in a semi-automated factory. It is essentially a hierarchical planner with a Time Map Manager (TMM) and a Time Optimizing Scheduler (TOS).

The TMM serves three main functions: (1) detect contradictory assertions which overlap in time and suggest a remedy when this occurs; (2) monitor the continued warrant for a subplan or action; and (3) evaluate relative advantages of disjunctive branches in the partially ordered plan net.

The TOS computes a schedule for each expansion choice, the best of which is chosen for further expansion with the schedule serving as a guide on the order of expansion.

Like other planners with a temporal reasoning component, FORBIN uses time constraints to control search. By ensuring that expansion choices satisfy deadlines and other time constraints, lesser mistakes are made and hence less backtracking occurs. The emphasis of FORBIN is synthesizing a plan for a robot's activities which includes travel. It is restricted to a single agent and does not solve the problem of synchronizing the plans of multiple mobile agents which may interact.

6.8.3.4 Temporal Logic

Several other developments (Vilain, 1982; Allen, 1983; Tsang, 1986) in temporal planning use a form of temporal logic based on thirteen primitive interval relations. Vilain extended this set to twenty-six relations which include time points. Cheeseman (1983) proposed a predicate calculus representation of time which does not entail a separate temporal reasoning mechanism since temporal assertions, like other assertions can be manipulated by the same Prolog-like backward reasoning mechanism.

The main concern of temporal logic is how to infer the temporal relation between two assertions which are indirectly linked by other temporally related assertions. Controlling inferences to avoid making many possible but useless inferences is a problem faced by these approaches.

Temporal logic being symbolically-oriented is useful for finding the truth value of relations between intervals which are not instantiated numerically, but it does not suggest how numerical intervals should be modified and propagated to maintain consistency in the course of plan generation. Problems with intervals numerically instantiated need not rely on temporal logic since the real line provides a common basis to infer directly the relation between any two known intervals.

6.8.4 Dynamic Replanning

Not many planners have incorporated a dynamic replanning capability. Those that do often employ a data structure which maintains dependency information. For example, the plans generated by STRIPS are supervised during execution by PLANEX (Fikes *et al*, 1972) using a triangular table. Rows and columns of the triangular table record respectively the preconditions and effects of operators used in the plan. With this record of the plan structure, good and bad surprises can be identified during execution. Good surprises are serendipitous developments which enable some future operators to be skipped because they are no longer necessary. Bad surprises occur when the state of the world has changed such that additional or old plan steps must be spliced into the current plan to achieve the original goals.

NOAH's procedural net (Sacerdoti, 1975) is also used by its rather simplistic execution monitor. The net is viewed as a hierarchy of actions and by asking the apprentice questions in an order determined from the action hierarchy, NOAH can pinpoint the erroneous action and replan accordingly.

In Ward and McCalla's ELMER route planning system (1982), there are two error detection strategies: use of check points and error transitions. The latter are identifiable instances which suggest a deviation from the prescribed route. The methods for error recovery are: (1) backtrack to the last point on the planned route; (2) replan from the current position to the same destination; and (3) when the current position is unknown, explore until an identifiable place is found and then follow either of the first two methods. ELMER's approach to dynamic replanning differs from the other planners in not using a dependency data structure.

Hayes (1975) also worked on a journey planner which produced a travel plan using pre-stored time-tables of the operating schedules of various transportation means. His main contribution was the use of a decision graph in conjunction with a goal tree to fix travel plans.

In all of the above, dynamic replanning is in the context of a single agent and the importance of timeliness is again not addressed. Also, no attention is paid to the issue of reducing the vulnerability of plans to dynamic replanning.

6.9 Summary

Unlike traditional AGV systems which employ hardware to coordinate movements during execution, the AGV planner described solves the coordination problem prior to execution. By generating a set of conflict-free schedules, it meets the requirement of temporal projection. Furthermore, since conflicts are resolved in advance, it is possible to accommodate more than one AGV in a path segment: implying that a higher throughput than that of the traditional block control system is possible.

Although point-based schedules suffice for the coordination problem, interval-based schedules are used instead because these meet the objectives of plan robustness and preservation of the near real-time capability of a dynamic replanner. Robustness is achieved by permitting execution deviations without immediate replanning; and if replanning is necessary, a benign speed alteration remedy can be attempted first without requiring replanning on the part of other AGVs. Tolerance of execution deviations also obviates frequent revalidation and this further reduces the likelihood of the dynamic replanner being bogged down by too much work.

The trade-off in resorting to interval-based schedules is that more conflict resolution is needed during planning. Resolution requires intervals to be modified and should be made in an equitable manner to maintain a comparable level of robustness among the affected plans. This should reduce the sensitivity of the set of plans to replanning repercussions. These aims can be met using the iterative negotiation model. Its implementation requires the definition of sets of negotiation skills and a demarcation of the negotiable components of the intervals.

An AGV movement planner has been implemented in LOOPS on a Xerox 1186 workstation, using the various algorithms described in previous chapters. Although the planner does not possess the property of completeness, it has the properties of soundness and definite termination. Tests on its coordination capability have been encouraging, even for trying scenarios.

Chapter 7

Planning Collaborative AGV Movements

7.1 Introduction

The sequential nature of planning for one task at a time means that whenever a task is being planned, there already exists a set of movement plans associated with the tasks planned previously. Instead of an AGV (say X) planning for the new task purely in a coordinative manner based on the route already worked out (as described in chapter 6), it may be possible to exploit some of the existing plans. Such exploitation may yield either of two advantages: (1) a better route for X is then possible; or (2) X may be made redundant because its task can be undertaken by some of the currently tasked AGVs. Planning the involvement of other AGVs in executing a task otherwise solely executed by X is known as *collaborative planning*. The goal of collaborative planning is to exploit existing movement plans so that economy of effort accrues.

Collaborative planning expands further the theme of constructing efficient AGV movement plans. This theme was developed in previous chapters which examined optimal task assignment, computation of shortest routes and route learning. Continuing the development of this theme, this chapter shows how economy of effort in executing tasks can be further improved by planning collaborative AGV movements.

Section 7.2 describes with the aid of examples the types of collaboration which is the concern of this chapter. It pays particular attention to the characteristics of the different modes of collaboration since these characteristics serve as clues to the identification of collaborative opportunities. Section 7.3 defines the conditions for collaboration to be possible. These conditions are used to prune the search tree of collaborative opportunities to a more manageable size. Section 7.4 elaborates on the computational procedures for identifying collaborative opportunities. An example showing the effectiveness of collaboration is presented in section 7.5. Section 7.6 examines some related work from operations research and AI which tackle the cooperation problem from a more general perspective.

7.2 Forms of Collaboration

AGVs which have plans pending execution can collaborate in many ways to help achieve a new task. One or more AGVs may be involved and their routes may or may not have to be changed. In any case, a *collaborating* AGV[†] must vary its movement schedule since assistance in the form of fetching another AGV's load while enroute necessarily entails commitment of time.

The various collaboration schemes can be classified into two categories. The first category does not require any of the collaborating AGVs to modify their routes. In the second category, at least one of the collaborating AGVs must make a detour in order to render assistance.

Figures 7-1 to 7-3 show contrived examples of collaboration schemes belonging to the

[†] The terms "collaborating AGV", "assisting AGV" and "collaborator" are used synonymously in this chapter.

first no-detour category. For the sake of simplicity, Figures 7-1 and 7-2 show only one assisting AGV involved. In general, the illustrated forms of collaboration may involve one or more assisting AGVs.

In all the figures, Vy, Vy' and Vy" are AGVs which have already planned to achieve their tasks along the paths depicted by broken lines. In this chapter, the AGVs with plans pending execution will also be referred to as the *currently tasked* AGVs. Vx is an AGV tasked to perform a new task.

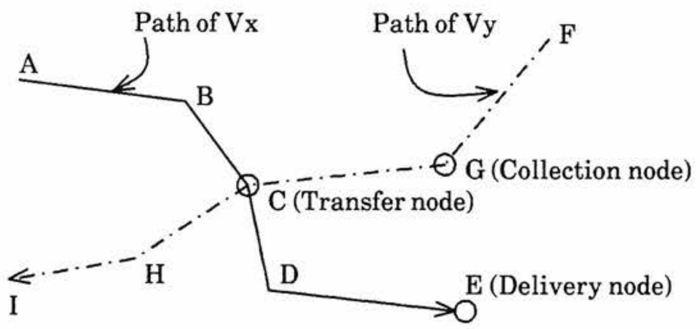


Figure 7-1. Assisted collection.

Figure 7-1 illustrates the case of collaboration by assisting in the collection of the load for Vx's task. Without collaboration, Vx will have to take the path (A B C G C D E). Since Vy and Vx have a common path segment (G C) in their plans, a possibility for collaboration exists in which Vy does not alter its path. Vy can assist Vx by moving Vx's load from G to C, leaving it there for Vx to collect or transferring it directly† over to Vx. The advantage is that Vx can now travel a shorter path (A B C D E) to achieve its task. The two essential features of the *assisted collection* form of collaboration are: (1) the collection node lies on a collaborator's path and (2) the collection point can be shifted such that Vx's execution path is shortened.

† This is possible only if (1) direct transfer of loads between AGVs can be effected; and (2) the place of transfer can accommodate both AGVs.

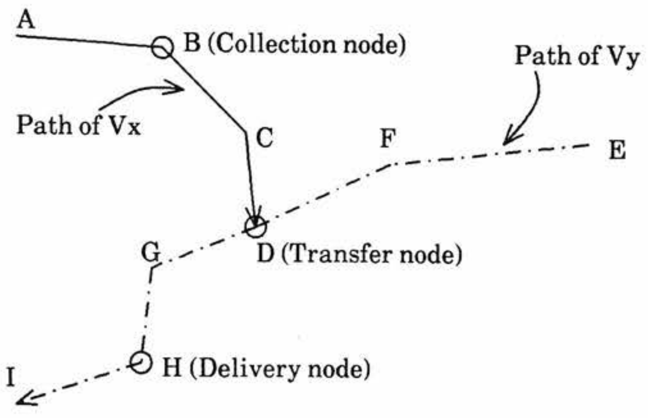


Figure 7-2. Assisted delivery.

Figure 7-2 gives an example of a form of collaboration known as *assisted delivery*. Here we see the assisting AGV, Vy, taking over the load from Vx and delivering it while enroute to its own destination. As far as Vx is concerned, its delivery node is at D where it relinquishes its load for good. Without collaboration, Vx will have to take a longer path (A B C D G H), but with collaboration, its path is reduced to (A B C D). Again, gainful collaboration is possible because of fortuitous commonality of plan segments. In this case, the segment in common is (D G H). The key features of the assisted delivery form of collaboration are: (1) the delivery node of Vx lies on the collaborator's path; and (2) a better delivery point for Vx can be found along it. Unlike assisted collection which shifts the collection point, it is the delivery point which is shifted in this case.

Although the commonality of plan segments has been highlighted in both the preceding examples, it is not meant as a necessary condition for collaboration. In fact it is an unnecessarily strong constraint since what is required is two nodes in common—the first and last of the segments—rather than all nodes along the segment. Complete commonality was highlighted only as an obvious indication of collaborative opportunities.

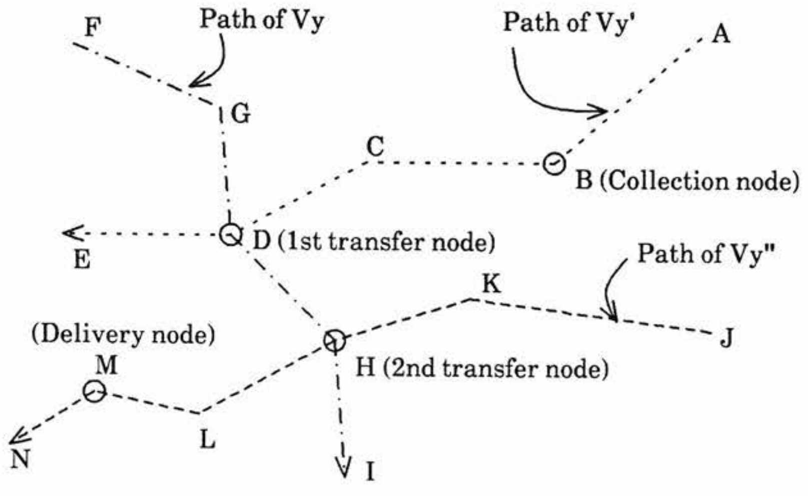


Figure 7-3. Task subsumption.

In the *task subsumption* form of collaboration (see Figure 7-3) in which assisting AGVs do not alter their routes, the new task can be undertaken by some of the currently tasked AGVs working in concert, and Vx which was originally tasked becomes redundant. Without collaboration, Vx must proceed to pick up its load at B and deliver it at M. It so happens that a path from B to D can be constituted from the existing plans of AGVs Vy, Vy' and Vy", allowing the new task to be achieved within its deadline constraint and yet without jeopardizing the delivery constraints of the assisting AGVs. Note that in this form of collaboration, the number of tasks assigned exceeds the number of AGVs involved. The key feature of this mode of collaboration is that both the pick up and delivery nodes lie on the path of collaboration.

Figure 7-4 shows a simple scenario in which any form of collaboration must involve a detour on the part of at least one of the currently tasked AGVs. At least one detour is necessary since neither the collection nor delivery node lies on the path of any of the currently tasked AGVs. Any appropriate collaborative plan which can be found thus belongs to the second category.

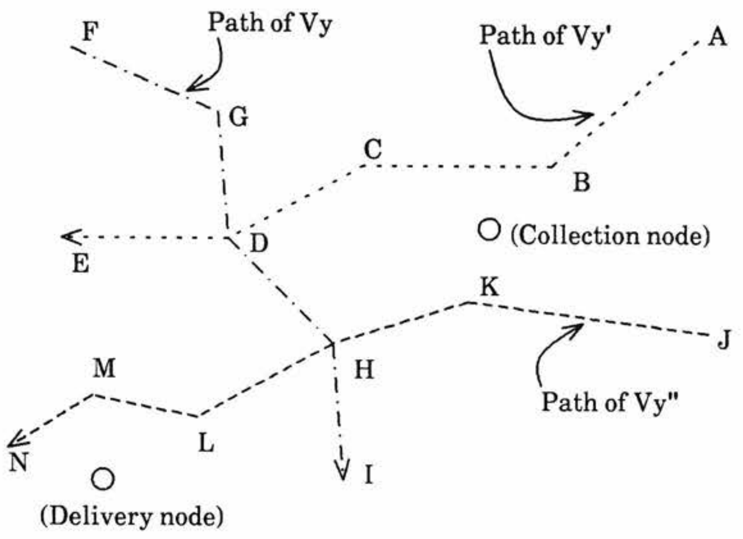


Figure 7-4. Detours necessary.

The problem of finding appropriate detours such that economy of effort accrues is non-trivial. Many detour possibilities exist and have to be enumerated and checked. Since it is not clear how an efficient strategy can be designed to tackle this problem, it is left as a subject for future research. Instead, this chapter will focus on the more tractable problems belonging to the first category *i.e.* assisted collection, assisted delivery and task subsumption.

7.3 Conditions for Collaboration

The identification of collaboration opportunities is the first problem to be solved in a collaborative planner. This raises two questions: (1) when is it plausible for some form of collaboration to be established? and (2) if it seems possible, when is it desirable? This section answers these questions by examining the conditions for plausible and gainful collaboration.

7.3.1 Feasibility and Plausibility

When is collaboration of the no-detour category apparently possible? The answer depends on the movement schedules of the AGVs involved, their routes, physical characteristics of the routes, and the load-bearing capacities of the AGVs.

Considering first temporal feasibility, an obvious condition is that the deadline constraints of the participating AGVs must be satisfied. However, satisfaction can be assured only when the collaborative plan is made conflict-free by the coordination mechanism described in chapter 6. Thus feasibility cannot be ascertained until a plausible collaborative plan has been constructed and subjected to conflict resolution. In other words, in collaborative planning, only plausible plans are found and a phase of coordinative planning must follow, just as in the case of purely coordinative planning in which a locally consistent plan is first found and then made globally consistent by means of the iterative negotiation mechanism. Hence the conditions for collaboration only suggest plausibility and they do not guarantee feasibility.

In collaborative plans involving two or more AGVs, the load is transferred, either directly or indirectly, from one to the other. If n AGVs are involved, there will be n–1 transfers. In any transfer, obviously the fetching AGV or "transferror" must off-load before the collecting AGV or "transferree" departs with the collected load. This temporal constraint is satisfied if the computed departure time of the transferror is before the arrival time of the transferree. Hence a plausible transfer can be determined on this basis. If timings at transfer points satisfy this constraint, then another route segment can be added to the plausible collaborative plan. The advantage of using this condition is that the part of the transferree's plan before the transfer point need not be modified on account of the transfer.

Even when the constraint in the preceding paragraph is not satisfied, plausibility can be established by forcing the transferree to arrive later than its current plan suggests.

However, this has the disadvantage of changing the transferree's plan before it collects the load; a change which needs further validation and resolution, steps which reduce the likelihood of the plan being found feasible. Furthermore, such forceful ordering of departure times is likely to cause a chain reaction of plan changes, adversely affecting the dynamic performance of the planner and lowering the tolerance of existing plans. For these reasons, the satisfaction of the departure time constraint is left to fortuitous circumstances *i.e.* coerced retardation of the transferree is forbidden.

The route constraint follows from the restriction of no detours. This means that any shifting of collection or delivery points must occur along the currently planned paths of the assisting AGVs. As pointed out in section 7.2, this requires either the collection or delivery points or both lie(s) on some such paths in the first place.

The load-bearing constraint is easy to check. Any currently tasked AGV can be a collaborator only if it is able to carry the load along the way. This should take into account the possibility that it may have to carry its own load as well during the period of assistance.

7.3.2 Desirability

Plausible collaborative plans need to be checked for desirability as well. The criterion for desirability is tied to the objective of the movement planner *i.e.* economy of effort. Again, distance happens to be a convenient and natural metric to use in assessing when economy of effort accrues from a collaborative plan.

Since the assisting AGVs do not change their planned paths, we only need to consider the change in distance of Vx, the assisted AGV which was originally tasked to perform the new task alone. At the start of collaborative planning, Vx has a locally consistent plan with a known route and distance *d*. A collaborative plan is worthwhile only if it can reduce *d*. If several plausible and desirable collaborative plans exist, then the planner selects the best *i.e.* the plan which reduces *d* most. If the best is subsequently found to be infeasible during coordinative planning, then the next best is selected and so on until a feasible collaborative plan is found; failing which the original plan of Vx involving no collaboration is used.

7.4 Planning Collaborative Movements

This section describes how collaborative plans of the no-detour category can be found. The methods outlined do not search the space of collaborative opportunities exhaustively as it is confined to those which are more readily and efficiently identified. Hence the methods to be described implement a kind of lazy collaboration.

In lazy collaboration, the search for plausible and desirable collaborative plans is simplified considerably by making two restrictions. The first has been mentioned: search is based on the no-detour category. The second restriction is to confine the search for a better collection/delivery point along the current route of the assisted AGV Vx. Both these restrictions have been made for the sake of efficiency. Otherwise, the search will involve considerable route planning and testing of the collaborative conditions. Furthermore, a larger search tree will have to be maintained with exponentially compounded computational costs of time and space.

7.4.1 Task Subsumption and Assisted Collection

Figure 7-5 shows the path which AGV Vx would take to execute a new task without collaboration. Vx's path is made up of two shortest subpaths P1 and P2 from S (starting location) to C (collection point) and from C to D (delivery point) respectively. Suppose P' is a *collaborative path* (not shown in the figure) constructed from the plans of some of the currently tasked AGVs and this path begins at C (a method for finding collaborative paths will be explained in section 7.4.4). P' offers the possibility of shifting the load of Vx to an alternative collection point C'.

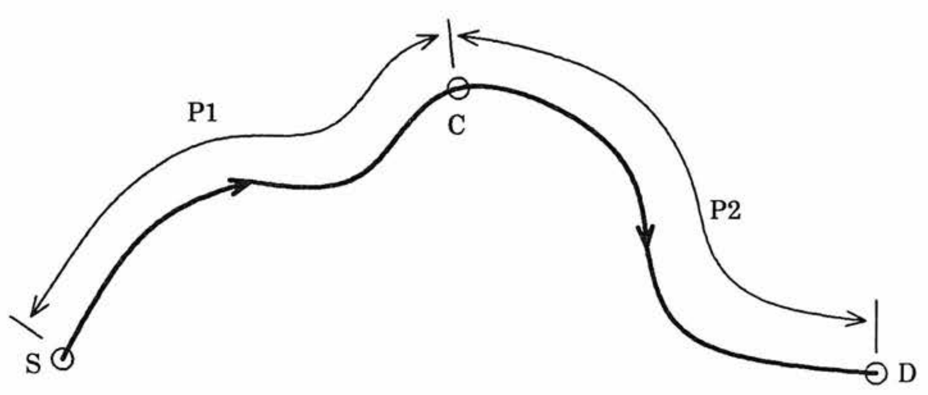


Figure 7-5. Pre-collaboration path of an AGV Vx.

If C' coincides with D, then the best form of collaboration—task subsumption—has been found. Otherwise, a simple and efficient method of determining if C' leads to a shortening of Vx's path is to check whether C' lies on P1 or P2.

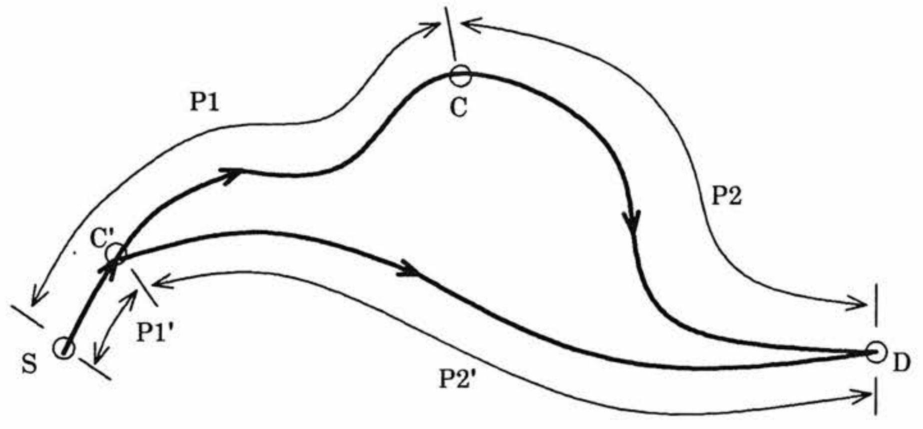


Figure 7-6. A better path via C' on P1.

Suppose C' lies on P1 (see Figure 7-6). The new path for Vx *via* C' using shortest paths P1' and P2' from S to C' and from C' to D respectively, cannot be longer than the path comprising P1 and P2. (The proof of this statement is trivial and is omitted.) Likewise, if

C' lies on P2 (see Figure 7-7), then the new path *via* C' cannot be longer than the path based on P1 and P2.

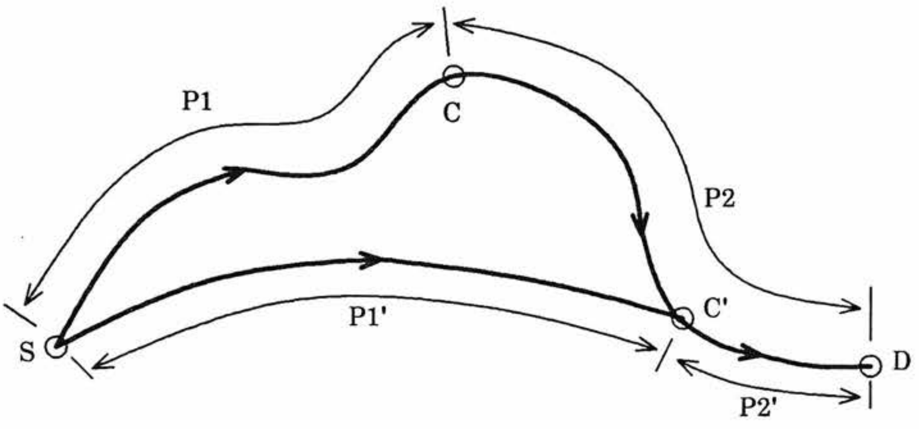


Figure 7-7. A better path via C' on P2.

Suppose that no assisted delivery and task subsumption plans are available. Then it may seem that the best collaborative plan to take is that which establishes C' as close as possible to S or D since this brings the new path for Vx closest to the shortest path from S to D which is the ideal. But collaborative planning should not cease at this juncture because further improvement is possible. Given C' and the new shortest subpaths from S to C' and C' to D respectively, the search procedure can be applied recursively to find a collection/delivery point if not a task subsumption plan. The recursion terminates when no more collaborative opportunities exist.

In recursive collaborative planning, a variation within the recursive search can be made to improve search efficiency. Suppose C' has been shifted to a better position along P1, then the next search for a new collection point can be confined to P2'. Search along P1' can be omitted since P1' has been explored in the preceding search while examining P1. Likewise, if C' has been shifted along P2', then the next recursive search for a new collection point can be confined to P1'.

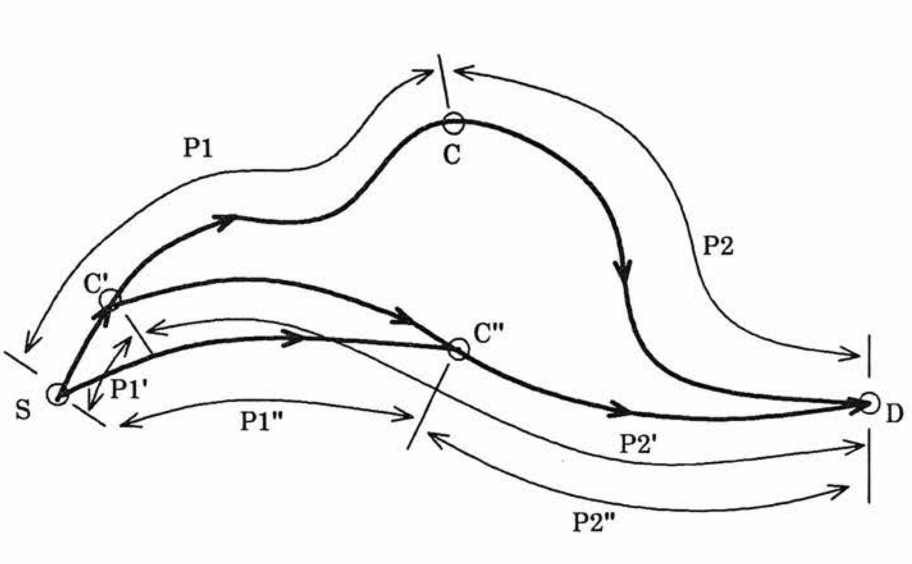


Figure 7-8. Recursive searching enables C" to be established away from P1 and P2.

The advantage of recursive searching is that it enables a larger part of the search space to be explored while still using the same efficient method for determining desirability. Without recursive searching, the search space for a collaborative plan is confined to the original paths P1 and P2 of Vx. With recursive searching, a better collection/delivery point can be established elsewhere as well. Figure 7-8 illustrates this point. The first search invocation establishes a better collection point C' along P1. Based on this, Vx plans a new shortest delivery path P2'. A second search invocation along P2' finds a better collection point at C" which does not lie on the original P1 or P2. Recursive searching thus overcomes to some extent the search space restriction without compromising search efficiency.

7.4.2 Assisted Delivery

In any assisted delivery plan, the assisted AGV Vx will travel along P1 to C where it collects the load, moves along part of P2 and unloads at a point before its original delivery destination D. Some other collaborating AGV will pick it up and either complete the delivery at D or pass it on to another collaborating AGV and so on until the load reaches D. Thus the aim is to find a collaborative path P' which effectively shifts (from the viewpoint of Vx) the delivery point from D to a better location D' on P2. Among the assisted delivery plans based on D and P2, the best is that which establishes D' closest C. Again, search should not terminate at this juncture since a recursive application may uncover better collaborative plans which are based on assisted collection or task subsumption.

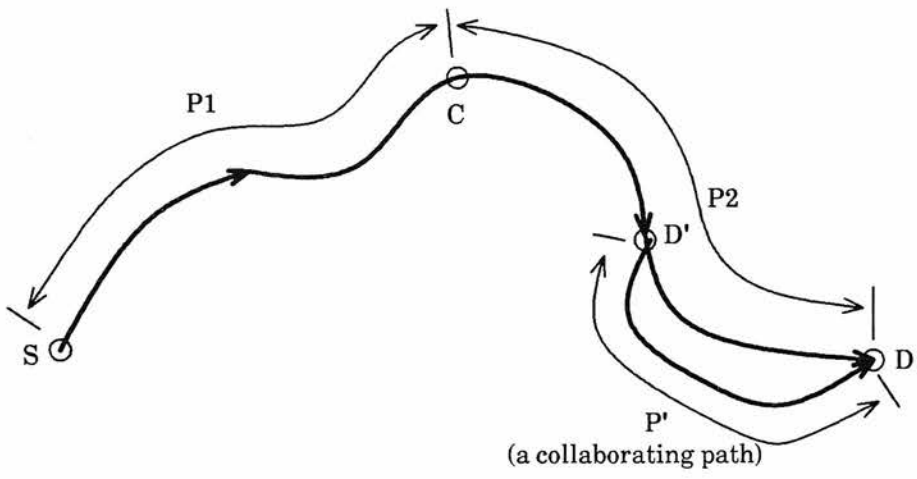


Figure 7-9. An assisted delivery plan.

Figure 7-9 shows an example of an assisted delivery plan. A few points should be mentioned. First, unlike assisted collection in which both P1 and P2 are searched, search in this case does not involve P1. Second, in the next recursive search invocation, there is no need to search for a new delivery point along the new subpath P2' since such points have already been uncovered in the current search. Third, the collaborative path need not be a subpath of P2; all that is required is that it begins somewhere along P2 and ends at D. If P' begins at C, then P' offers a task subsumption plan.

7.4.3 Assiduous Search

Unlike lazy collaboration, an assiduous search strategy for identifying collaboration opportunities would not restrict the search along the current subpaths of the assisted AGV Vx. This offers the advantage of possibly uncovering collaborative plans not otherwise found, or which are superior to those found by the restrictive search methods described above. Although the implemented collaborative planner does lazy collaboration only, I shall briefly describe how a more comprehensive search can be implemented for the benefit of those who may wish to investigate its feasibility.

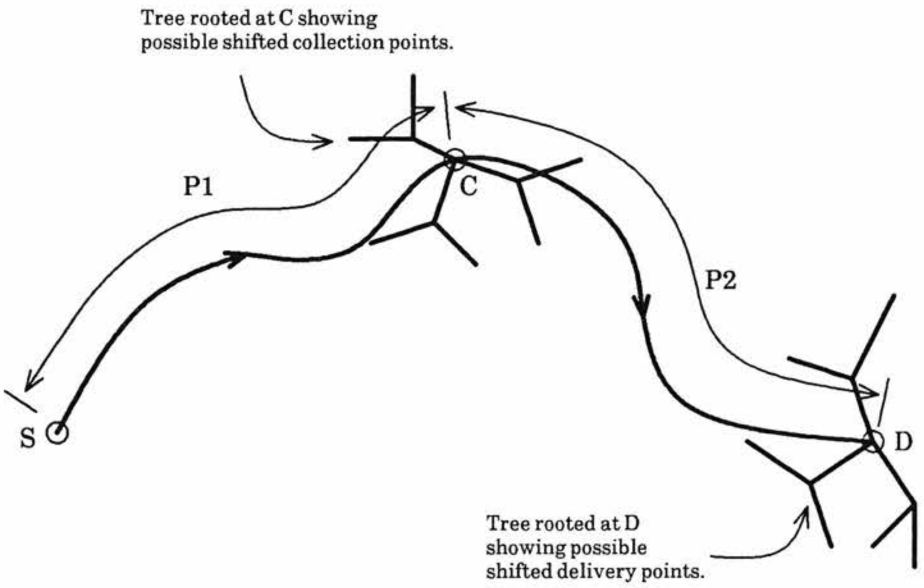


Figure 7-10. Assiduous searching grows trees of shifted collection/delivery points.

Essentially, two search trees are developed which are rooted at the original collection and delivery points (see Figure 7-10). The tree rooted at C is used to find assisted collection plans and that rooted at D for assisted delivery plans. The nodes in these trees represent shifted collection/delivery points and the corresponding collaborative paths can be found by tracing the arcs back to the root of the tree. Not all shifted collection/delivery points lead to a path reduction for Vx. All combinations of shifted collection and delivery points have to be checked for desirability by computing the new P1' and P2' shortest subpaths. The combination which has the shortest total distance yields the best collaborative plan.

The disadvantage is that many combinations have to be checked and every check for desirability now requires the computation of two shortest paths. This is far more costly than the previous approach which requires two simple tests to determine if the shifted point lies on the current subpaths P1 and P2.

7.4.4. Collaborative Paths and the Collaboration Graph

Collaborative paths are composed of path segments of currently tasked AGVs based on their existing plans. Their vital role in the search for collaborative plans has been elaborated upon above. This section explains how collaborative paths are found.

The key to the problem of discovering collaborative paths is to maintain a data structure such that it is possible to determine how a collaborative path can be extended by another currently tasked AGV. This extension occurs whenever a transfer point exists. A data structure which meets this requirement is the *collaboration graph*. Figure 7-11 shows part of such a graph. In it, some of the path segments of three currently tasked AGVs (V1, V2 and V3) are shown. For example, the path segments of V1 shown are V1a, V1b, V1c and V1d.

Nodes in the collaboration graph have a list of pointers to the RTokens (as explained in section 6.6.1, RTokens contain information such as ownership and the components of the time interval reserved at a node) of the currently tasked AGVs. These RTokens are those which are associated with the node. For example, if V1 and V2 have RTokens R_{V1a} and R_{V2a} at node N1, then N1 will have an associated list structure containing the

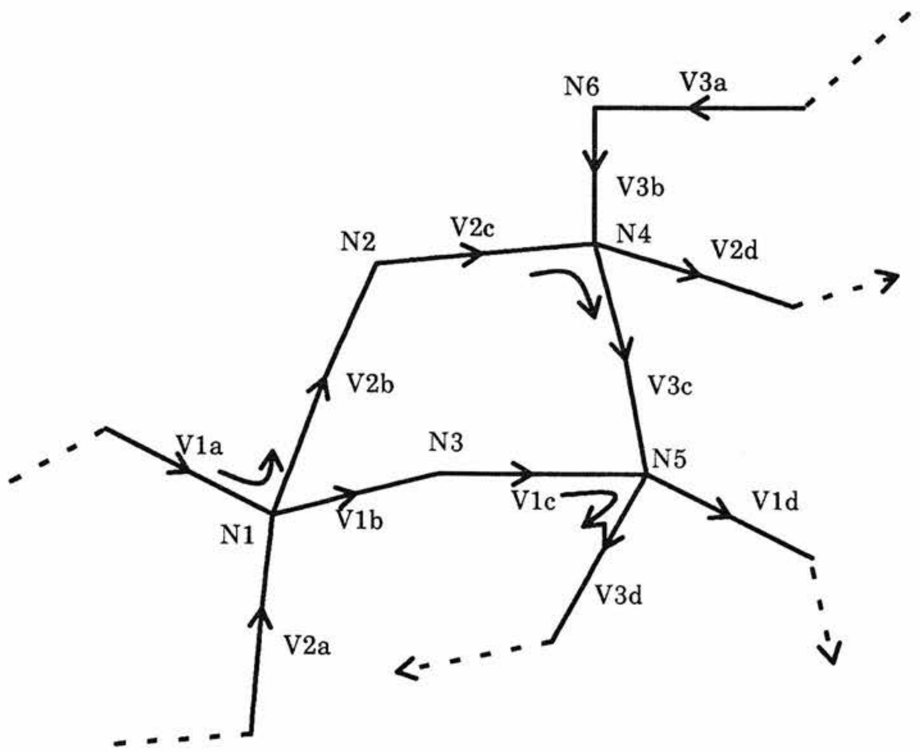


Figure 7-11. Part of a collaboration graph.

pointers to R_{V1a} and R_{V2a}. The significance of these pointers is that they provide the information for determining whether the node is a transfer point from a temporal perspective.

The collaboration graph is a dynamic data structure. It changes during planning and execution. Whenever a new plan is established, RTokens are added to existing and newly created nodes in the graph. During execution, defunct RTokens are removed, shrinking the list of pointers at some nodes. Nodes in the graph with empty pointer lists are removed.

Suppose V1a (see Figure 7-11) is the last segment of a collaborative path being constructed. How can it be extended? One definite possibility is to add on V1b which means that V1 does not transfer the load to another AGV at N1 but carries the load past N1 instead. The other possibility is to transfer it to V2 if the constraints for a transfer (see section 7.3.1) are satisfied. The curved arrow annotated next to N1 in Figure 7-11 shows that a transfer from V1 to V2 at N1 is possible. Conversely, the absence of such an arrow indicates that the transfer conditions are not satisfied. Hence, if a collaborative path along V2a is being developed, a transfer from V2 to V1 at N1 is impossible and the only way to extend it is along V2b.

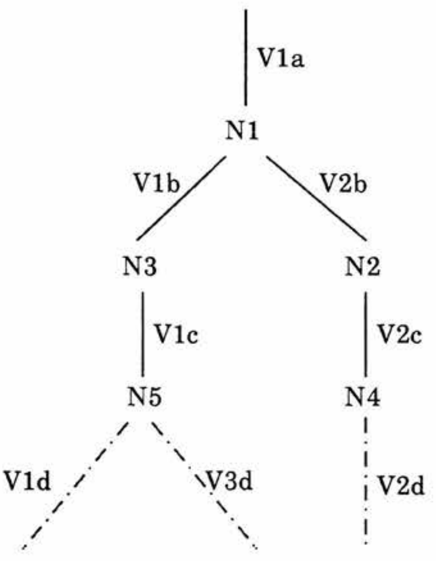


Figure 7-12. A sub-tree of collaborative paths.

Continuing the development of the collaborative path at V1a, the subtree of collaborative paths which can be grown is shown in Figure 7-12. Every node in this tree defines a unique collaborative path which is reconstructed by tracing the path in the tree from the root to the node. Note that although V3c offers another collaborative path ending at N5, it is discarded because the collaborative path *via* N3 is shorter.

Two kinds of trees can be developed using the collaboration graph. The example in Figure 7-12 shows part of a *forward* tree of collaborative paths. The forward tree is rooted at the original collection point and contains the assisted collection paths. The other type of tree is the *backward* tree of collaborative paths. Its construction follows the same principles except that the arrows in the collaboration graph are traced in the backward direction. The backward tree is rooted at the original delivery node and contains the assisted delivery paths.

It may seem that the tree of collaborative paths can be quite large and thus extensive searching will slow down the planner's performance. Fortunately, the tree can be pruned substantially using the constraints for transferability and delivery deadline.

7.4.5 The Algorithms

The following algorithms implement the recursive collaborative planning procedures as described informally above. These procedures search for a set of collaborative plans indexed by the pair of collection and delivery points in the forward and backward collaboration trees respectively. The collaborative plan may be composite in nature *i.e.* made of an assisted collection and an assisted delivery plan. The set of collaborative plans is found by invoking CollaborateAux recursively.

Following exit of CollaborateAux within the main Collaborate procedure, the set of plans are sorted in decreasing order of desirability. Hence task subsumption plans will appear in front of the list followed by plans which require the assisted AGV to move the shortest paths. From this sorted list, the first plan is popped off and is subjected to the conflict resolution process. If it is found infeasible, then it will be discarded and the next in line is selected. This process is repeated until the collaborative planner finds a feasible plan. If no collaborative plan exists, CollaborateAux will nevertheless return a plan based on the original collection and delivery points.

Algorithm 7-1:

```
        procedure Collaborate(S, C, D, P1, P2)
        vars  TFwd, TBwd, CPlans, Planned, C', D', PAC, PAD, PAGV; /*local variables */
        /*  P1 and P2 are the shortest paths from S to C and from C to D respectively. */
1.      Grow forward collaboration tree TFwd rooted at C.
2.      Grow backward collaboration tree TBwd rooted at D.
3.      CPlans ← CollaborateAux(C, D, P1, P2, true, true, true)
4.      Sort CPlans in order of decreasing desirability.
5.      Planned ← false
6.      foreach Best in CPlans until Planned do
7.          C' ← collection point of Best
8.          D' ← delivery point of Best
9.          if C' ≠ C
10.             then  Construct the assisted collection plan PAC by tracing the path from
                      C to C' in TC.
            endif
11.         if D' ≠ D
12.             then  Construct the assisted delivery plan PAD by tracing the path from
                      D' to D  in TD.
            endif
13.         if D' ≠ C'
                then  /* task subsumption has not occurred. */
14.                   Construct plan PAGV of assisted AGV based on the shortest paths
                      from S to C' and C' to D'.
            endif
15.         Resolve conflicts in PAC, PAD and PAGV (if these exist) by iterative
            negotiation.
16.         if all plans can be resolved then Planned ← true endif
        endforeach
```

```
endprocedure
```

Algorithm 7-2:

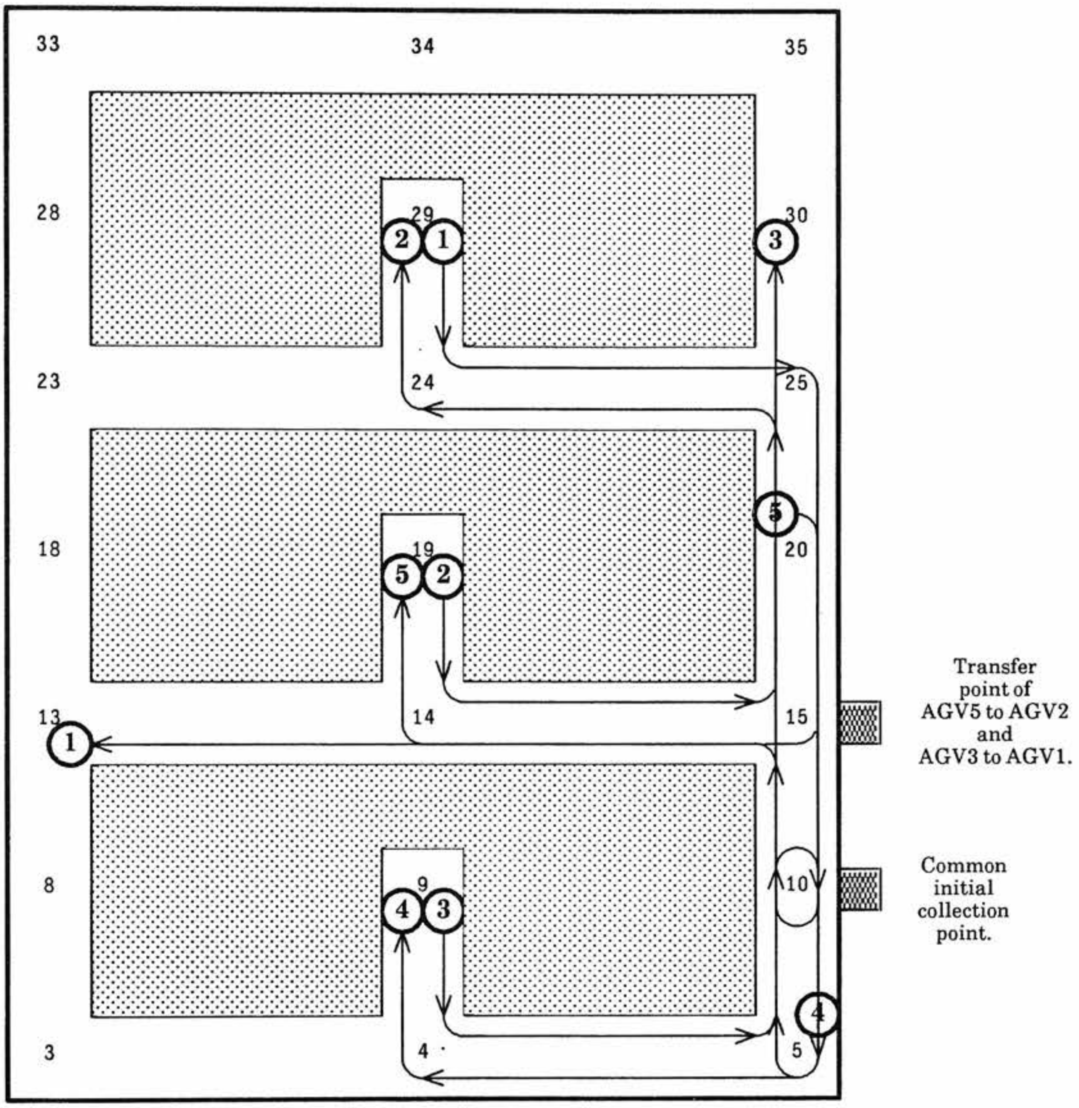
```
    procedure CollaborateAux(C, D, P1, P2, ShiftDFlag, ShiftCAlongP1Flag,
                                  ShiftCAlongP2Flag)
      /* Returns the combination of collection and delivery points found by a
         recursive search. S is a global variable defined within Collaborate. */
    vars CPlans, P1', P2', TC, TD; /* local variables */
1.  CPlans ← {{C, D}}
2.  if D is in TC  /* TC is the subtree of TFwd rooted at C. */
        then /* a task subsumption plan has been found. {C, D} is included in
                case the subsumption plan is found infeasible. */
3.          CPlans ← {{C, C}, {C, D}}
        else /* look for collaborative plans based on assisted delivery and
                collection plans. */
4.          if ShiftDFlag
                then /* consider assisted delivery plans. TD is the subtree of
                        TBwd rooted at D. */
5.                  foreach N in P2 when (N ≠ D and N is in TD) do
6.                      P2' ← subpath of P2 from C to N;
7.                      CPlans ← CPlans ∪ CollaborateAux(C, N, P1, P2',
                                                  false, true, true)
                    endforeach
            endif
8.          if ShiftCAlongP1Flag
                then /* look for collaborative plans based on assisted
                        collection plans which shift C along P1.*/
9.                  foreach N in P1 when (N ≠ C and N is in TC) do
10.                     P1' ← subpath of P1 from S to N;
```

```
11.                    P2' ← shortest path from N to D;
12.                    CPlans ← CPlans ∪ CollaborateAux(N, D, P1', P2',
                                                   *true, false, true*)
                 **endforeach**
           **endif**
13.        **if** ShiftCAlongP2Flag
             **then** /* look for collaborative plans based on assisted
                        collection plans which shift C along P2. */
14.              **foreach** N **in** P2 **when** ($N \neq C$ **and** N is in TC) **do**
15.                  P1' ← shortest path from S to N;
16.                  P2' ← subpath of P2 N to D;
17.                  CPlans ← CPlans ∪ CollaborateAux(N, D, P1', P2',
                                                   *true, true, false*)
                 **endforeach**
           **endif**
     **endif**
18.  Return CPlans
     **endprocedure**
```

Note that within procedure CollaborateAux, in the loop beginning at line 14, if $N = D$, then N cannot be in TC; otherwise, a task subsumption plan exists and this would have been detected in line 2, in which case, the procedure will skip lines 4 to 17. This explains why the more complete expression ($N \neq C$ **and** $N \neq D$ **and** N is in TC) is not used in line 14.

7.5 An Example

We can observe an improvement in economy of effort in task execution by applying the collaborative planner to the same problem scenario (see section 6.6.2), involving five


33
34
35
28
29
30
23
24
25
18
19
20
13
14
15
8
9
10
3
4
5
1
2
3
4
5
Transfer point of AGV5 to AGV2 and AGV3 to AGV1.
Common initial collection point.

Uncircled numbers are the nodes defining the route map.
Bold circled numbers denote the AGVs

Paths of AGVs:

AGV1 ... (29 24 25 20 15 14 13)
AGV2 ... (19 14 15 20 25 24 29)
AGV3 ... (9 4 5 10 15 20 25 30)
AGV4 ... (5 10 5 4 9)
AGV5 ... (20 15 10 15 14 19)

Figure 7-13. Better paths for AGV1 and AGV2 due to collaboration.

AGVs, which was used to demonstrate the feasibility of the purely coordinative AGV movement planner (described in chapter 6). However, the same delivery deadlines would have imposed an overly restrictive set of constraints for the planner to find feasible collaborative plans. Hence, for the purpose of demonstrating collaboration at work, the delivery deadlines have been relaxed from [05:00, 06:00] to [05:00, 07:30]; a relaxation which permits some plausible collaborative plans to be found feasible.

Time	AGV	Activity	Task	Position
03:11	4	Loading	1	10
03:32	5	Loading	3	10
03:47	5	Loading	2	10
04:03	3	Loading	5	10
04:18	3	Loading	4	10
04:19	5	Unloading	3	15
04:40	3	Unloading	5	15
05:01	4	Unloading	1	9
05:04	1	Loading	5	15
05:37	2	Loading	3	15
05:42	3	Unloading	4	30
05:44	5	Unloading	2	19
06:26	1	Unloading	5	13
07:10	2	Unloading	3	29

Figure 7-14. Loading and unloading sequence of events

The conflict-free set of movement plans generated by the collaborative planner is illustrated in Figure 7-13. Figure 7-14 lists, in chronological order, the sequence of loading and unloading events according to the plans. The effect of collaboration is that AGV1 and AGV2 can be assisted by AGV3 and AGV5 respectively by shifting their collection points from node 10 to 15. This saves both the assisted AGVs from traversing the path segment (15 10 15) which would have been necessary without the benefit of collaboration.

Observe that AGV5 nevertheless has to fetch its own load at node 10 even though it could have been similarly assisted by AGV3. No collaborative plan was generated to assist AGV5 because it was the first AGV to plan the execution of its task and hence did not have the benefit of any currently tasked AGVs to assist it. This reveals a limitation of the planner: retrospective collaboration is not exploited to improve the plans of currently tasked AGVs using the latest plan pertaining to the new task.

Running in compiled mode, the collaborative planner required about 140 seconds to plan for the five tasks, compared to 54 seconds required by the purely coordinative planner. Despite the restrictions imposed to uncover collaborative opportunities more readily, the time overheads are high but may be affordable if planning is relatively well ahead of the time when execution begins. If time is a limiting factor, it may be necessary to curtail the depth of recursive search and work on whatever candidate plans can be found within the search time permitted.

7.6 Related Work

The only work I have come across, which deals with some kind of collaboration in the transportation domain, relates to the transhipment problem (Taha, 1971). The transhipment problem is to determine how items should be distributed from the sources to the destinations in an optimal manner. In the transhipment case, movements of items are allowed to pass through other sources and destinations.

There are two limitations in the transhipment solution which deny its applicability to the AGV domain. First, the transfer locations are limited to the set of sources and destinations. This severe limitation rules out many collaborative opportunities. Second, movement timings and deadlines are not represented in the transhipment model. Hence no means exist to check the satisfaction of temporal constraints.

In distributed AI (DAI), there is an important research issue which concerns mechanisms or schemes for facilitating cooperation between multiple agents in the same environment. The rest of this section describes the main approaches and systems designed to deal with the cooperation problem in a more general context.

Davies and Smith's contract net metaphor (1983), reviewed in section 2.3, models cooperative problem-solving by task sharing in a benevolent way. Recall that the manager decomposes a task into subtasks which are then distributed to some of the contractors which have volunteered to solve the subtasks. It assumes that a task can be decomposed into independent subtasks. This departs from the nature of AGV movement collaboration in which parts of the collaborative plan are interdependent.

Another mode of cooperation is result-sharing. This is featured in the Distributed Vehicle Monitoring Testbed (Durfee *et al*, 1987). Using the same structure of multiple-blackboards as described in section 6.8.2.3, agents organize their problem solving tasks and eventually communicate the results of the monitoring activities to another agent which pools the information for further processing. This agent may in turn pass its solution to yet another agent to piece together a larger portion of the puzzle, and so on until the complete solution is obtained. Cooperation is achieved by communicating the same high level goals to the agents.

A recurrent theme in cooperation is the formation of groups of agents which are able to solve a problem more effectively. This is possible in the contract net metaphor. It is also observed in the EXCHANGE (Doran and Corcoran. 1986) and CONTRACT (Doran, 1986) systems.

Based on TEAMWORK2 (Doran, 1985; Ambros *et al*, 1986) which is an experimental shell for implementing multi-agent nonlinear hierarchical planners, EXCHANGE models a group of socio-economic entities (villages) to investigate the evolution of economic structures and activities as a result of individual interests, functions, resources and limitations. It demonstrates that structure stems from function and hence hierarchical organizations can evolve to a form which exploits the environment in a more efficient manner. EXCHANGE achieves cooperation by self-organization such that the measure of overall goal achievement is improved.

In CONTRACT, a lattice of contracts provide the basis of a search for an optimal contract which best exploits the environment. It relates to EXCHANGE in that a contract stems from a beneficial pact between actors. Further pacts at a higher level may be formed recursively. Eventually, a hierarchy of contracts is formed in which each contract is made up of lower level contracts. The problem-solving objective is then to search for the best contract given a scenario. Although the design of CONTRACT was motivated by its relevance to sociocultural systems in which a similar pact formation phenomenon is apparent, CONTRACT may also be used as a framework for investigating DAI problem-solving.

Game theory is another approach to the formulation of cooperative strategies. Developed by von Neumann (von Neumann and Morgenstern, 1944), it was first applied to the analysis of political, economic and military strategies in a competitive context: how to maximize one's gain with minimal risks due to the actions of adversaries. The cooperative context—how agents can select options which are mutually beneficial—was investigated by Axelrod (1984) who found that cooperation can be induced if uncooperative agents can be punished *e.g.* tit-for-tat. Such retribution requires that there must be a sequence of "moves" in the game. If the game involves only one move, then there is no opportunity for retribution.

Genesereth *et al* proposed a game-theoretic model of cooperation between intelligent agents acting rationally, but without the benefit of communication. The model captures the different types of interaction (*e.g.* prisoner's dilemma; battle of the sexes) between two agents using a pay-off matrix. It assumes: (1) that such a matrix is completely known by both agents; (2) that agents behave rationally and will seek to maximize their pay-offs; and (3) that agents act simultaneously in each game. Hence game exploitation to the other's disadvantage does not occur as may happen if each took turns in choosing its course of action. Genesereth *et al* identified various decision procedures appropriate for some of the interaction types. If these decision procedures are adopted by the agents, mutually beneficial cooperation will ensue. The major difficulty in this approach lies with the definition of the pay-off matrix. This assumes that accurate utility estimates can be found for all the enumerated combinations of actions. A further shortcoming is that it does not solve the resolution problem which is inevitable when agents' plans are not independent.

7.7 Summary

Collaborative planning is another means to improve the economy of effort in task execution. Its key idea is to exploit existing movement plans of currently tasked AGVs which may be fortuitously predisposed to assist in the execution of a new task.

Collaboration can occur only if certain conditions—order of arrival/departure times; delivery deadlines; load bearing capacity; and space feasibility—are satisfied. However, these conditions determine plausibility and not feasibility. Final feasibility is ascertained when the selected collaborative plan is subjected to the conflict resolution mechanism.

The search space for collaborative opportunities can be large and exhaustive searching may undermine the speed of the planner. An efficient approach is to search along the subpaths of the AGV being assisted, and to restrict the opportunities to those which do not entail detours on the part of the assisting AGVs. This enables the desirability of plausible collaborative plans to be verified readily. Another important reason for restricting the search to the no-detour category is to minimize the changes to existing plans; these changes are confined to the temporal dimension since the paths of collaborating AGVs do not change.

Three forms of collaboration—assisted collection, assisted delivery and task subsumption—have been defined and algorithms for identifying them have been outlined. The algorithms implement the search recursively and use the conditions for collaboration to prune the search tree. Such pruning improves search efficiency without overlooking collaborative opportunities which could have been found otherwise.

An experimental collaborative planner has been implemented to demonstrate that collaboration is indeed possible and beneficial. When presented with the same scenario (involving five AGVs) which was used to demonstrate the feasibility of the purely coordinative planner, the collaborative planner was able to improve on two of the AGVs' movement plans.

Chapter 8

Summary and Conclusion

8.1 The Research Issues

The main problem which this research focusses on is the automatic generation of AGV movement plans in a way which is an improvement over current AGV systems. Current AGV systems produce movement plans which are sub-optimal and coordinated in an *ad hoc* manner. By assigning tasks to AGVs optimally, better economy of effort during execution is attained and this brings with it the benefits of energy saving and more productive utilization of AGVs. Moreover, *ad hoc* coordination has the serious disadvantage of not admitting temporal projection which is essential for further planning of operations in the factory. This disadvantage can be overcome by planning the movement schedules of the AGVs prior to execution.

An important consideration in the design of the AGV movement planner is that planning occurs before and during execution. Run-time planning is required because of the quasi-continuous nature of factory operations—new tasks are generated as operations proceed. Furthermore, in reality, the movement timings of AGVs do not correspond exactly to the planned timings *i.e.* execution deviations are likely. This means that plans may require modification during execution. Hence this dissertation also addresses the problems of dynamic planning and replanning. Since dynamic planning and replanning are in the context of other movement plans which already exist, and there is relatively less time for the planning process to complete before execution must begin, dynamic planning/replanning needs to be expeditious with few side-effects on existing plans.

Summarizing, the research issues are:

- Optimality: how can economy of effort in the execution of AGV movement plans be realized?
- Temporal projection: how can event timings be forecasted to permit further planning?
- Dynamic planning/replanning: how can a task be planned in the context of existing plans?
- Dynamic performance: how can planning be made as fast as possible so as to increase the probability of beginning execution early enough to accomplish the new goals?

8.2 Addressing the Issues

8.2.1 Optimality

The main approach is to use shortest paths for AGVs individually and as a group. Individually, every AGV will use the shortest possible path to move from one location to another. As a group, the total distance of all the AGVs' paths is minimized by assigning the tasks optimally. These two measures guarantee that a group of AGVs, in which each has its own task and does not share the burden of other AGVs, has the least cost of effort (measured in distance) required to discharge the tasks.

A second approach is to reduce the overall effort involved *via* task sharing. If AGVs are able to share the burden of another AGV, then AGVs can collaborate in performing a task in a more economical manner *e.g.* by exploiting commonality of plan segments.

8.2.2 Temporal Projection

Temporal projection is possible once a set of conflict-free movement schedules have been found. Determining this set is the outcome of movement planning with a conflict resolution scheme. Two versions of an experimental AGV movement planner—the purely coordinative version and the collaborative version—have been implemented to meet this need. Both make use of the iterative negotiation model of conflict resolution to ensure that plans are conflict-free.

8.2.3 Dynamic Planning/Replanning

Two alternative approaches are possible. The first is to design a pre-execution planner which functions separately from the dynamic planner/replanner. The second is to design a planner which meets both the roles of planning before and during execution. I have chosen the second since it offers the important advantage of code sharing.

A pre-execution planner which is able to plan dynamically can be implemented by adopting a sequential planning approach in which tasks are planned completely one at a time. Hence whenever a task is being planned, there already exists (in general) some plans of other AGVs. Since this is exactly the scenario for dynamic planning/replanning, the planner meets the dyamic role as well.

Another advantage in this approach is that some exising plans are completely defined to permit exploitation of collaborative opportunities. The sequential planning approach is thus congenial to collaborative planning which contributes further to the optimality objective.

8.2.4 Dynamic Performance

Dynamic performance demands that the planning process be as fast as possible since there is no guarantee of a lower bound on the time available. Expeditious planning relies on two approaches: (1) use of fast algorithms, especially those which are often invoked during planning; and (2) generating robust plans which lessen the computational burden of the dynamic replanner by reducing the incidents of dynamic replanning required to preempt impending failures. Robust plans achieve this effect by tolerating an AGV's own execution deviations as well as those of other AGVs'. Replanning is then unnecessary since the plan continues to be safe as long as the deviations are not excessive. Robust movement plans are thus somewhat innocuous to deviant behaviours and this also helps to contain a chain reaction of plan revisions when replanning becomes necessary.

8.3 Contributions

In addressing the research issues, the original contributions are as follows:

- NewHungarian—an optimal task assignment algorithm which is a novel computational implementation of the Hungarian method originally described for manual application. NewHungarian is faster than the Kuhn-Munkres algorithm which solves the same problem using the augmenting path approach.
- BS*—a new algorithm belonging to the class of bidirectional admissible heuristic search algorithms. BS* is superior in this class both in terms of computational time and space and has the potential to run twice as fast as the fastest unidirectional search algorithm (A*) when implemented in a multiprocessor machine. BS* is also the first staged search algorithm which preserves admissibility.

- Efficient route learning algorithms for extracting all other optimal solutions embedded in search trees. These algorithms can be incorporated into most admissible search algorithms to improve the time and effort in searching for an optimal solution. BSL*, based on BS*, is one such enhanced algorithm. Significant speed-up is made possible by the availability of instant solutions, or if unavailable, the use of learnt solutions to achieve earlier termination.
- The concept of tolerant planning for generating robust movement plans which have better executability by being tolerant of extraneous execution deviations as well as the AGVs' own deviations.
- The iterative negotiation model for conflict resolution which allows AGVs to compromise their initial ideal plans in an equitable manner without jeopardizing safety.
- An efficient recursive algorithm which searches for collaborative opportunities.
- An experimental AGV movement planner to demonstrate the feasibility of the concepts of tolerant planning, iterative negotiation and collaboration.

8.4 Further Research

8.4.1 Coordinative Planning

For greater planning speed, the use of a parallel computer, in which the processor nodes represent RTokens created during planning, should be investigated. Links between processor nodes would represent the relations between RTokens. Such a network would support the propagation of conflict resolution characteristic of our iterative negotiation model. The identification of sub-processes which can be computed concurrently, and the relationship between procedural control and the characteristics of plans produced in concurrent negotiation acts are possible research issues.

A form of dependency-directed backtracking mechanism permitting a more comprehensive search of the solution space may be useful. Possibly, this will reduce the number of task rejections due to failure to find a feasible plan *i.e.* the completeness of planner is improved. Furthermore, if accurate heuristics can be found to guide the search, the final solution choice will be closer to some optimal criterion. Such a solution may be more robust and thus less liable to incur dynamic replanning.

8.4.2 Collaborative Planning

Although the collaborative planner exploits existing movement plans to find an even more efficient plan for a new task, exploitation in the reverse form does not occur. This suggests that retrospective collaboration based on the set of plans which includes that which has just been collaboratively planned, may yield further opportunities for improving on previous plans.

Another avenue for improvement is to make a preliminary analysis of the set of tasks to be planned. The aim is to determine the order of task planning which gives the most opportunities for collaboration.

A third area is a liberal scheme for identifying collaborative opportunities. Unlike the present scheme which works within the no-detour class, a liberal scheme will allow detours on the part of assisting AGVs. The advantage of this can be noted from the example scenario in which a slight detour on the part of an assisting AGV enables a task to be subsumed.

Appendix A

Some Theorems Related to the Hungarian Method of Optimal Assignment.

This appendix proves that the NewHungarian procedure (Algorithm 2-9) terminates and that its time complexity is $O(n^4)$ where n is the cardinality of the maximal matching.

Lemma A-1:

Before phase I of procedure NewHungarian terminates, in any MC obtained, there are always more uncrossed elements than doubly crossed elements.

Proof:

For a $n \times n$ matrix, the cardinality of the maximal matching is n.

Before phase I terminates, suppose the current MC has r rows and c columns.

Since phase I has not terminated,

$r + c < n$ [1]

Let α be the number of elements in the matrix which are crossed out twice by the MC.

$\alpha = rc$

Let β be the number of elements crossed out by the MC.

$\beta = nr + nc - \alpha$

Let γ be the number of elements which are not in the MC.

$$\gamma = n^2 - \beta = n^2 - n(r+c) + \alpha$$

$$\gamma - \alpha = n^2 - n(r+c) \qquad [2]$$

From [1] and [2], $\gamma > \alpha$.

Theorem A-1:

Procedure NewHungarian terminates.

Proof:

When $\gamma > \alpha$ and phase I tries to find a larger MC by subtracting from each of the γ uncrossed elements a positive value (equal to the minimum of these elements) and adding the same value to each of the α doubly crossed elements, the total sum of the elements in the matrix must be reduced in every iteration of phase I. Noting that all elements are always non-negative, it follows that a finite number of applications of the ModifyMatrix procedure (Algorithm 2-2) will eventually reduce the sum of the matrix elements to zero. In this case, all elements are 0s and the set of distinct representatives can then be trivially found. Phase I must therefore terminate.

Phase II also terminates by virtue of Berge's theorem (Berge, 1957) and the fact that every augmentation definitely increases the matching.

It follows that procedure NewHungarian terminates in a finite number of steps.

Theorem A-2:

The time complexity of procedure NewHungarian is $O(n^4)$.

Proof:

In general, before phase II is entered, there will be several iterations of phase I. In deriving an MC, each row/column obtained requires a scan of n elements in it. For an MC of r rows and c columns, the number of elements scanned is $n(r+c)$. Since $r + c \leq n$, the time complexity of the MC computation is $O(n^2)$.

If the MC is incomplete, members of a subset of the matrix elements are decremented and members of a disjoint subset are incremented. Since the number of elements whose values are changed is at most n^2, the time complexity of the matrix modification procedure is also $O(n^2)$.

Since the MC computation and matrix modification procedure are in sequence, each iteration of phase I is $O(n^2)$.

Next, we will show that the maximum number of iterations in phase I is bounded above by kn^2, where k is a constant.

At each iteration of phase I, let p and q be respectively the minimum and maximum non-zero values in the matrix.

Clearly, the sum S of element values $\leq n^2q$.

Suppose the minimum uncrossed element value is s.

Since more elements are decremented than incremented in each iteration (by Lemma A-1), S must be decremented by at least s.

As $s \geq p$, the number of iterations to reduce S to 0 must be less than $\lceil n^2q/p \rceil$.

It follows that the time complexity of NewHungarian is $O(n^4)$.

Appendix B

Superiority of Nicholson's Terminating Condition

Notations:

k	A node where the fringe of $TREE_1$ meets the fringe of $TREE_2$. Typically, there will be more than one meeting node since fringes change with search and will meet again elsewhere in the search space.
g_{min1}	The minimum g_1 value of nodes in $OPEN_1$.
g_{min2}	The minimum g_2 value of nodes in $OPEN_2$.
P_{NIC}	The proposition $[(g_{min1} + g_{min2}) \geq L_{min}]$ representing Nicholson's terminating condition (for proof see Nicholson (1966)).
P_{BSPA}	The proposition $[k \in CLOSED_1 \cap CLOSED_2]$ representing Pohl's terminating condition (for proof see Pohl (1969)).

(further notations appearing in this appendix are defined in section 3.4.1.)

PNIC and PBSPA are two different terminating conditions which have been proposed for uninformed best-first bidirectional searches based on Dijkstra's algorithm. Here, we prove that PNIC is better than PBSPA in the sense that PNIC can detect a least path cost solution earlier than PBSPA, if not at the same time. The proof also tells us that PNIC subsumes PBSPA: in other words, whenever PBSPA is true, so is PNIC but not *vice-versa*. Hence the disjunctive terminating condition $P_{NIC} \vee P_{BSPA}$ which some might suggest (in the hope of capturing terminating states otherwise undetected by PNIC alone), is superfluous.

The proof of the superiority of Nicholson's terminating condition can be established by showing that there exists an instance in which Nicholson's algorithm terminates before BSPA, and Nicholson's algorithm terminates no later than BSPA.

Lemma B-1:

PNIC is tested before or at the same time as PBSPA.

Proof:

A k node must exist before *Lmin* can be assigned a value. Since PNIC contains the *Lmin* term, it can first be tested only when the first k node is found. Subsequently, *Lmin* is updated when a new k node is found or when a better path to/from an old k node is found. A k node can be in one of the four states shown in Figure B-1. Since a node must be open before it can be closed, state A must precede states B and C, and, states B and C must precede state D. The precedence relationships of these four states are shown in Figure B-2. Clearly, PNIC can be tested in any of these four states but PBSPA can be true only in state D.

State	**Status of node k**	
A	□□	Open in both trees.
B	□▨	Open in forward tree and closed in backard tree.
C	▨□	Closed in forward tree and open in backard tree.
D	▨▨	Closed in both trees.

Figure B-1. The four states of node k

$$\begin{tikzcd}
 & \overset{B}{\square\blacksquare} \arrow[dr] & \\
\overset{A}{\square\square} \arrow[ur] \arrow[dr] & & \overset{D}{\blacksquare\blacksquare} \\
 & \overset{C}{\blacksquare\square} \arrow[ur] &
\end{tikzcd}$$

Figure B-2. Precedence relationships of states of k

<u>Lemma B-2</u>:

$\forall m \in CLOSED_1,\ g_{\min 1} \geq g_1(m)$.

Proof:

Suppose O is the set of nodes in $OPEN_1$, and node m is chosen from O for expansion. In general, expanding m enlarges $OPEN_1$ to $(O - \{m\}) \cup S$, where S is the set of m's successor added to $OPEN_1$. Clearly, $g_1(m) \leq g_1(n)$ for all n in O or else m would not have been chosen for expansion. Moreover, for every node n in S, $g_1(n) = g_1(m) + c_1(m,n) \geq g_1(m)$ since $c_1(m,n) \geq 0$. Hence $g_1(m)$ is at least equal to the smallest of g_1 in the new $OPEN_1$.

Lemma B-3:

$\forall m \in CLOSED_2$, $g_{\min 2} \geq g_2(m)$.

Proof:

Same as the proof for lemma B-2 but with the search direction reversed.

Lemma B-4:

PNIC is true in state D.

Proof:

Let k be the node which first presents a state D. Since k is considered as a possible contributor to the best value assigned to L_{min}, $g_1(k)+g_2(k) \geq L_{min}$. Since $k \in CLOSED_1 \cap CLOSED_2$, $g_{\min 1} \geq g_1(k)$ (by lemma B-2) and $g_{\min 2} \geq g_2(k)$ (by lemma B-3), it follows that $[(g_{\min 1}+g_{\min 2}) \geq L_{min}]$ (*i.e.* PNIC) is true.

This proof also establishes the correctness of PBSPA if the correctness of PNIC is true. By correctness, we mean that when the terminating condition is satisfied, then the least cost solution has indeed been found.

Theorem B-1:

PNIC is true before or at the same time as PBSPA.

Proof:

PBSPA can only be true when the first state D occurs (by definition of PBSPA). When this occurs, PNIC is also true (by lemma B-4). But PNIC is tested not only before a state D (by lemma B-1), it can in fact be true in states earlier than D (by the example depicted in Figure B-3 which shows a problem instance when PNIC is true in state A).

Legend: ■ A node in state D.

Path(n): IJ...M The best complete path so far is IJ...M and has length n.

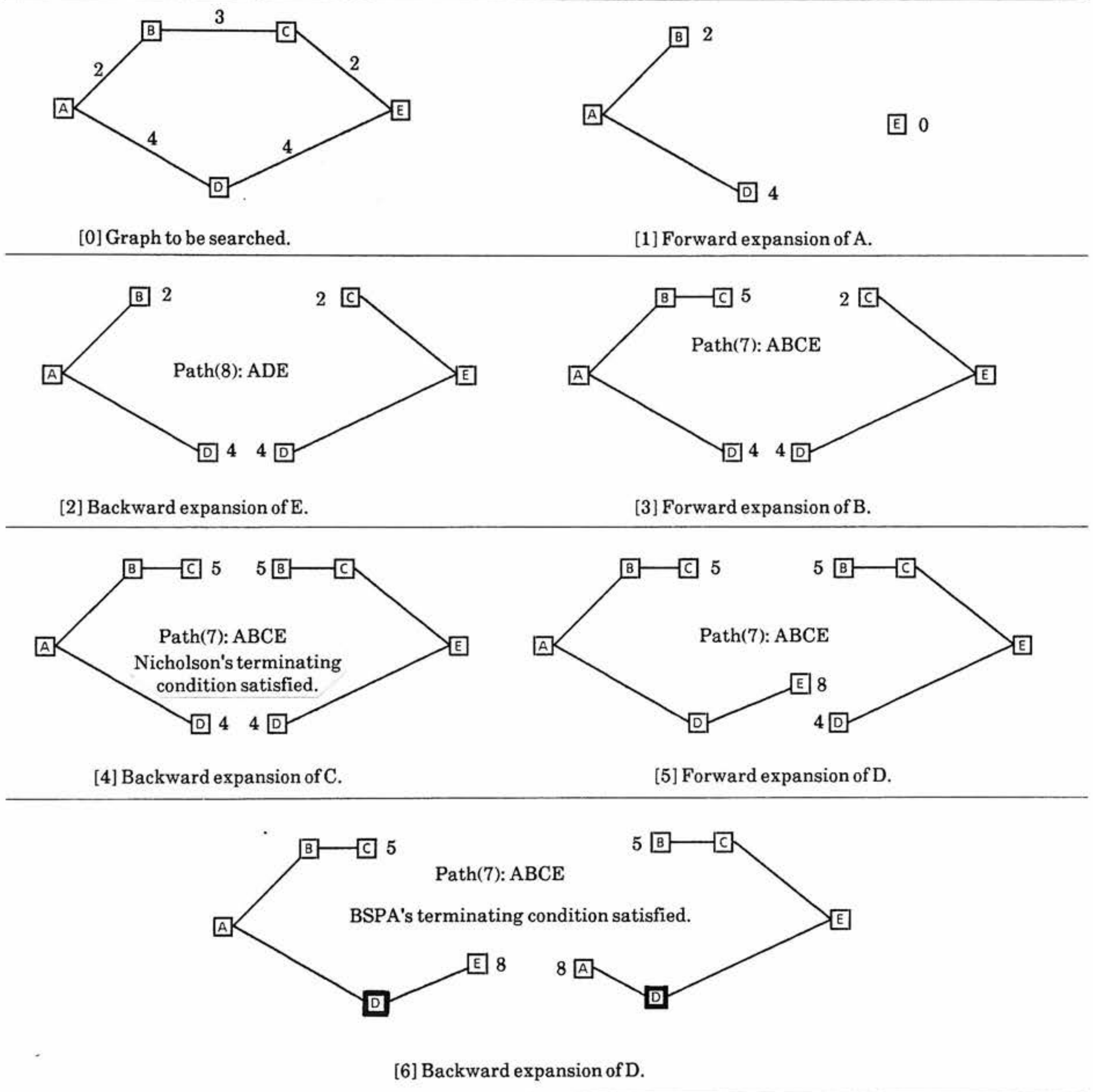

[0] Graph to be searched.
[1] Forward expansion of A.
[2] Backward expansion of E.
Path(8): ADE
[3] Forward expansion of B.
Path(7): ABCE
[4] Backward expansion of C.
Path(7): ABCE
Nicholson's terminating condition satisfied.
[5] Forward expansion of D.
Path(7): ABCE
[6] Backward expansion of D.
Path(7): ABCE
BSPA's terminating condition satisfied.

Figure B-3. An instance of Nicholson's algorithm terminating before BSPA.

Appendix C

Pohl's BHPA Algorithm

(Notations used are defined in section 3.4.1.)

<u>Algorithm C-1</u>:

```
    procedure BHPA( )
    /* Compute the shortest path from $s$ to $t$. $amin$ is the search cut-off
       parameter. */
1.  $amin \leftarrow \infty$ ; $g1(s) \leftarrow g2(t) \leftarrow 0$; $f1(s) \leftarrow f2(t) \leftarrow h1(s)$.
2.  Put $s$ in $OPEN1$ and $t$ in $OPEN2$.
3.  until  $OPEN1$ is empty
        or $OPEN2$ is empty
        or $amin <$ smallest $f1$ value in $OPEN1$
        or $amin <$ smallest $f2$ value in $OPEN2$
    do
4.      Determine search direction index $d$.
5.      Transfer node $m$ in $OPENd$ with the lowest $fd$ value into $CLOSEDd$.
        /* Expand $m$. */
6.      foreach $n$ in $\Gamma d(m)$ do
7.          $g \leftarrow gd(m) + cd(m,n)$; $f \leftarrow g + hd(n)$.
8.          if $n$ is in $OPENd$ and $f < fd(n)$
```

```
9.            then    $f_d(n) \leftarrow f$; $p_d(n) \leftarrow m$
10.           elseif  $n$ is in $CLOSED_d$ and $f < f_d(n)$
11.           then    $f_d(n) \leftarrow f$; $p_d(n) \leftarrow m$
12.                   Transfer $n$ from $CLOSED_d$ to $OPEN_d$.
13.           else    $f_d(n) \leftarrow f$; $p_d(n) \leftarrow m$
14.                   Place $n$ in $OPEN_d$.
           endif
        endforeach
15.     if $m \in CLOSED_1 \cap CLOSED_2$ and $a_{min} > g_1(m) + g_2(m)$
16.        then    $a_{min} \leftarrow g_1(m) + g_2(m)$; $MeetingNode \leftarrow m$
        endif
    enduntil
17. if $a_{min} = \infty$
18.     then no path exists
19.     else   the optimal path cost is $a_{min}$ and the optimal path can be
               determined by tracing the forward and backward parent
               pointers from $MeetingNode$.
    endif
    endprocedure
```

Appendix D

A Complete Negotiation Trace Involving Five AGVs

```
Selected strategy: Tolerance preservation i.e. shift before yielding.

AGV4 planning Task1 ....
***** DONE ******

AGV5 planning Task2 ....

R2 (186.25 . 232.75) conflicts with R8 (186.25 . 234.25)
     Amt of conflict: 46.5
At node 10, R8 will negotiate with R2 on its left.
R8     ShiftRight:
         Conceded:  4.69    (186.25 . 234.25) -> (190.94 . 238.94)
          Residue:  41.81
R2      ShiftLeft:
         Conceded:  0.0     (186.25 . 232.75) unchanged.
          Residue:  41.81
R8     YieldFrontHedge:
         Conceded:  10.0    (190.94 . 238.94) -> (200.94 . 238.94)
          Residue:  31.81
R2     YieldEndHedge:
         Conceded:  10.0    (186.25 . 232.75) -> (186.25 . 222.75)
          Residue:  21.81
R8     YieldEndHedge&Shift:
R8 will move R9 above.
[Move Above R9] ShiftRight YieldEndHedge&Shift
R9 will move R10 above.
[Move Above R10] ShiftRight
         Conceded:  10.0    (200.94 . 238.94) -> (210.94 . 238.94)
          Residue:  11.81
R2     YieldFrontHedge&Shift:
R2 will move R1 below.
[Move Below R1] ShiftLeft
         Conceded:  10.0    (186.25 . 222.75) -> (186.25 . 212.75)
          Residue:  1.81
R8     YieldTol&Shift:
R8 will move R9 above.
[Move Above R9] ShiftRight YieldEndHedge&Shift
R9 will move R10 above.
[Move Above R10] ShiftRight
         Conceded:  1.81    (210.94 . 238.94) -> (212.75 . 238.94)
          Residue:  0.0
***** DONE ******

AGV2 planning Task3 ....

R15 (139.375 . 188.375) conflicts with R2 (186.25 . 212.75)
```

```
     Amt of conflict: 2.13
At node 10, R15 will negotiate with R2 on its right.
R15    ShiftLeft:
        Conceded:  0.0      (139.38 . 188.38) unchanged.
          Residue:  2.13
R2     ShiftRight:
        Conceded:  0.0      (186.25 . 212.75) unchanged.
          Residue:  2.13
R15    YieldEndHedge:
        Conceded:  2.13     (139.38 . 188.38) -> (139.38 . 186.25)
          Residue:  0.0
***** DONE ******

AGV3 planning Task4 ....

R25 (229.6875 . 264.6875) conflicts with R9 (231.5 . 268.5625)
     Amt of conflict: 33.19
At node 15, R25 will negotiate with R9 on its right.
R25    ShiftLeft:
        Conceded:  4.69     (229.69 . 264.69) -> (225.0 . 260.0)
          Residue:  28.5
R9     ShiftRight:
        Conceded:  0.0      (231.5 . 268.56) unchanged.
          Residue:  28.5
R25    YieldEndHedge:
        Conceded:  10.0     (225.0 . 260.0) -> (225.0 . 250.0)
          Residue:  18.5
R9     YieldFrontHedge:
        Conceded:  10.0     (231.5 . 268.56) -> (241.5 . 268.56)
          Residue:  8.5
R25    YieldFrontHedge&Shift:
R25 will move R24 below.
[Move Below R24] ShiftLeft YieldFrontHedge&Shift
R24 will move R23 below.
[Move Below R23] ShiftLeft YieldFrontHedge&Shift
R23 will move R22 below.
[Move Below R22] ShiftLeft
        Conceded:  8.5      (225.0 . 250.0) -> (225.0 . 241.5)
          Residue:  0.0

Resolving at both ends: (R2) -> R24 -> (R8)

R2 (186.25 . 212.75) conflicts with R24 (196.25 . 232.75)
     Amt of conflict: 16.5
At node 10, R24 will negotiate with R2 on its left.
R24    ShiftRight:
        Conceded:  0.0      (196.25 . 232.75) unchanged.
          Residue:  16.5
R2     ShiftLeft:
        Conceded:  0.0      (186.25 . 212.75) unchanged.
          Residue:  16.5
R24    YieldFrontHedge:
        Conceded:  1.5      (196.25 . 232.75) -> (197.75 . 232.75)
          Residue:  15.0
R2     YieldEndHedge:
        Conceded:  0.0      (186.25 . 212.75) unchanged.
          Residue:  15.0
R24    YieldEndHedge&Shift:
R24 will move R25 above.
[Move Above R25] ShiftRight YieldEndHedge&Shift YieldTol&Shift MoveRightOrAbove
R25 moves the following right: ((5.0 . #.($ R9)))
[Move Right R9] ShiftRight YieldFrontHedge YieldEndHedge&Shift
R9 will move R10 above.
[Move Above R10] ShiftRight YieldEndHedge&Shift
R10 will move R11 above.
[Move Above R11] ShiftRight
```

```
        Conceded:  10.0    (197.75 . 232.75) -> (207.75 . 232.75)
         Residue:  5.0
R2    YieldFrontHedge&Shift:
        Conceded:  0.0     (186.25 . 212.75) unchanged.
         Residue:  5.0
R24   YieldTol&Shift:
R24 will move R25 above.
[Move Above R25] ShiftRight YieldEndHedge&Shift YieldTol&Shift MoveRightOrAbove
R25 moves the following right: ((5.0 . #.($ R9)))
[Move Right R9] ShiftRight YieldFrontHedge YieldEndHedge&Shift
R9 will move R10 above.
[Move Above R10] ShiftRight YieldEndHedge&Shift
R10 will move R11 above.
[Move Above R11] ShiftRight YieldTol&Shift
R9 will move R10 above.
[Move Above R10] ShiftRight YieldEndHedge&Shift
R10 will move R11 above.
[Move Above R11] ShiftRight
        Conceded:  5.0     (207.75 . 232.75) -> (212.75 . 232.75)
         Residue:  0.0

R24 (212.75 . 232.75) conflicts with R8 (212.75 . 238.9375)
     Amt of conflict: 20.0
At node 10, R24 will negotiate with R8 on its right.
R24    ShiftLeft:
        Conceded:  0.0     (212.75 . 232.75) unchanged.
         Residue:  20.0
R8    ShiftRight:
        Conceded:  10.0    (212.75 . 238.94) -> (222.75 . 248.94)
         Residue:  10.0
R24   YieldEndHedge:
        Conceded:  0.0     (212.75 . 232.75) unchanged.
         Residue:  10.0
R8    YieldFrontHedge:
        Conceded:  0.0     (222.75 . 248.94) unchanged.
         Residue:  10.0
R24   YieldFrontHedge&Shift:
        Conceded:  0.0     (212.75 . 232.75) unchanged.
         Residue:  10.0
R8    YieldEndHedge&Shift:
        Conceded:  0.0     (222.75 . 248.94) unchanged.
         Residue:  10.0
R24   YieldTolerance:
        Conceded:  0.0     (212.75 . 232.75) unchanged.
         Residue:  10.0
R8    YieldTol&Shift:
R8 will move R9 above.
[Move Above R9] ShiftRight YieldEndHedge&Shift YieldTol&Shift
R9 will move R10 above.
[Move Above R10] ShiftRight YieldEndHedge&Shift
R10 will move R11 above.
[Move Above R11] ShiftRight YieldTol&Shift
R10 will move R11 above.
[Move Above R11] ShiftRight
        Conceded:  4.69    (222.75 . 248.94) -> (227.44 . 248.94)
         Residue:  5.31
R24   MoveLeftOrBelow:
R24 moves the following left: ((5.31 . #.($ R2)))
[Move Left R2] ShiftLeft YieldEndHedge YieldFrontHedge&Shift YieldTolerance
        Conceded:  5.31    (212.75 . 232.75) -> (207.44 . 227.44)
         Residue:  0.0

R1 (157.5 . 192.5) conflicts with R23 (177.5 . 204.0)
     Amt of conflict: 15.0
At node 5, R23 will negotiate with R1 on its left.
R23   ShiftRight:
        Conceded:  9.69    (177.5 . 204.0) -> (187.19 . 213.69)
         Residue:  5.31
R1      ShiftLeft:
        Conceded:  5.31    (157.5 . 192.5) -> (152.19 . 187.19)
         Residue:  0.0
***** DONE ******

AGV1 planning Task5 ....

R16 (172.8125 . 212.8125) conflicts with R35 (198.4375 . 233.4375)
     Amt of conflict: 14.38
At node 15, R35 will negotiate with R16 on its left.
R35    ShiftRight:
        Conceded:  6.56    (198.44 . 233.44) -> (205.0 . 240.0)
         Residue:  7.81
R16     ShiftLeft:
        Conceded:  4.69    (172.81 . 212.81) -> (168.13 . 208.13)
         Residue:  3.13
R35    YieldFrontHedge:
        Conceded:  3.13    (205.0 . 240.0) -> (208.13 . 240.0)
         Residue:  0.0

Resolving at both ends: (R15) -> R34 -> (R2 R24)

R15 (139.375 . 186.25) conflicts with R34 (165.0 . 210.0)
     Amt of conflict: 21.25
At node 10, R34 will negotiate with R15 on its left.
R34    ShiftRight:
        Conceded:  0.0     (165.0 . 210.0) unchanged.
         Residue:  21.25
R15     ShiftLeft:
        Conceded:  0.0     (139.38 . 186.25) unchanged.
         Residue:  21.25
R34    YieldFrontHedge:
        Conceded:  10.0    (165.0 . 210.0) -> (175.0 . 210.0)
         Residue:  11.25
R15    YieldEndHedge:
        Conceded:  7.87    (139.38 . 186.25) -> (139.38 . 178.38)
         Residue:  3.38
R34    YieldEndHedge&Shift:
        Conceded:  3.38    (175.0 . 210.0) -> (178.38 . 210.0)
         Residue:  0.0

R34 (178.375 . 210.0) conflicts with R2 (186.25 . 207.4375)
     Amt of conflict: 23.75
At node 10, R34 will negotiate with R2 on its right.
R34     ShiftLeft:
        Conceded:  0.0     (178.38 . 210.0) unchanged.
         Residue:  23.75
R2     ShiftRight:
        Conceded:  0.0     (186.25 . 207.44) unchanged.
         Residue:  23.75
R34    YieldEndHedge:
        Conceded:  6.62    (178.38 . 210.0) -> (178.38 . 203.38)
         Residue:  17.13
R2     YieldFrontHedge:
        Conceded:  0.0     (186.25 . 207.44) unchanged.
         Residue:  17.13
R34    YieldFrontHedge&Shift:
        Conceded:  0.0     (178.38 . 203.38) unchanged.
         Residue:  17.13
R2     YieldEndHedge&Shift:
        Conceded:  0.0     (186.25 . 207.44) unchanged.
         Residue:  17.13
R34    YieldTolerance:
        Conceded:  5.0     (178.38 . 203.38) -> (178.38 . 198.38)
```

```
           Residue:  12.13
R2     YieldTol&Shift:
          Conceded:  .44     (186.25 . 207.44) -> (186.69 . 207.44)
           Residue:  11.69
R34    MoveLeftOrBelow:
R34 moves the following left: ((11.69 . #.($ R15)))
[Move Left R15] ShiftLeft YieldEndHedge YieldFrontHedge&Shift
R15 will move R14 below.
[Move Below R14] ShiftLeft YieldFrontHedge&Shift
R14 will move R13 below.
[Move Below R13] ShiftLeft YieldTolerance
R34 will move R33 below.
[Move Below R33] ShiftLeft YieldFrontHedge&Shift
R33 will move R32 below.
[Move Below R32] ShiftLeft YieldFrontHedge&Shift
R32 will move R31 below.
[Move Below R31] ShiftLeft
          Conceded:  11.69   (178.38 . 198.38) -> (166.69 . 186.69)
           Residue:  0.0

Resolving at both ends: (R14) -> R33 -> (R7)

R14 (113.125 . 148.625) conflicts with R33 (146.25 . 172.9375)
     Amt of conflict: 2.38
At node 15, R33 will negotiate with R14 on its left.
R33    ShiftRight:
          Conceded:  0.0     (146.25 . 172.94) unchanged.
           Residue:  2.38
R14     ShiftLeft:
          Conceded:  0.0     (113.13 . 148.63) unchanged.
           Residue:  2.38
R33    YieldFrontHedge:
          Conceded:  1.69    (146.25 . 172.94) -> (147.94 . 172.94)
           Residue:  .69
R14    YieldEndHedge:
          Conceded:  .69     (113.13 . 148.63) -> (113.13 . 147.94)
           Residue:  0.0

R33 (147.9375 . 172.9375) conflicts with R7 (167.5 . 204.0)
     Amt of conflict: 5.44
At node 15, R33 will negotiate with R7 on its right.
R33     ShiftLeft:
          Conceded:  0.0     (147.94 . 172.94) unchanged.
           Residue:  5.44
R7     ShiftRight:
          Conceded:  5.44    (167.5 . 204.0) -> (172.94 . 209.44)
           Residue:  0.0

R32 (123.75 . 157.9375) conflicts with R6 (148.75 . 183.75)
     Amt of conflict: 9.19
At node 20, R32 will negotiate with R6 on its right.
R32     ShiftLeft:
          Conceded:  0.0     (123.75 . 157.94) unchanged.
           Residue:  9.19
R6     ShiftRight:
          Conceded:  9.19    (148.75 . 183.75) -> (157.94 . 192.94)
           Residue:  0.0
***** DONE ******
```

Appendix E

Movement Plans (see Figure 6-14)

(Based on tolerance preservation strategy *i.e.* shift before yield)

Movement plan for AGV1:

Node	Arrival Time	Wait(secs)	RToken
29	01:02	0	R29
24	01:21	0	R30
25	01:57	0	R31
20	02:12	0	R32
15	02:27	0	R33
10	02:46	10	R34
15	03:35	0	R35
14	04:15	0	R36
13	05:10	20	R37

Movement plan for AGV2:

Node	Arrival Time	Wait(secs)	RToken
19	01:14	0	R12
14	01:30	0	R13
15	02:00	0	R14
10	02:19	10	R15
15	02:58	0	R16
20	03:26	0	R17
25	03:49	0	R18
24	04:36	0	R19
29	05:00	20	R20

Movement plan for AGV3:

Node	Arrival Time	Wait(secs)	RToken
9	02:11	0	R21
4	02:29	0	R22
5	03:08	0	R23
10	03:27	10	R24
15	04:01	0	R25
20	04:23	0	R26
25	04:46	0	R27
30	05:10	20	R28

Movement plan for AGV4:

Node	Arrival Time	Wait(secs)	RToken
5	02:42	0	R1
10	03:06	10	R2
5	03:49	0	R3
4	04:36	0	R4
9	05:00	20	R5

Movement plan for AGV5:

Node	Arrival Time	Wait(secs)	RToken
20	02:47	0	R6
15	03:02	0	R7
10	03:47	10	R8
15	04:16	0	R9
14	04:53	0	R10
19	05:12	20	R11

Bibliography

Abbreviations:

ACM	Association for Computing Machinery
AIAI	Artificial Intelligence Applications Institute
ECAI	European Conference on Artificial Intelligence
IEEE	Institute of Electrical and Electronics Engineers
IJCAI	International Joint Conference on Artificial Intelligence
NCAI	National Conference on Artificial Intelligence
SIAM	Society for Industrial Applications of Mathematics
SIGART	Special Interest Group on Artificial Intelligence
SLAC	Stanford Linear Accelerator Centre
SRI	Stanford Research Institute

Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1983) *Data Structures and Algorithms.* Addison-Wesley, Reading, MA.

Allen, J. F. (1983) Maintaining knowledge about temporal intervals. *Communications of the ACM*, **26**, 832-843.

Ambros-Ingerson, J. A., Doran, J. E., Steel, S. W. D. & Trayner, C. (1986) *TEAMWORK 2 — the system so far.* Cognitive Studies Centre Memorandum 23, Univ. of Essex, Colchester.

Anderson, I. (1974) *A First Course in Combinatorial Mathematics.* Clarendon Press, Oxford.

Axelrod, R. (1984) *The evolution of cooperation.* Basic Books, New York.

Bell, C. E. (1985) *Resource management in automated planning.* Technical Report AIAI-TR-8, AIAI, Univ. of Edinburgh.

Bell, C. E. & Tate, A. (1984) *Using temporal constraints to restrict search in a planner.* Technical Report AIAI-TR-5, AIAI, Univ. of Edinburgh.

Bell, C. E. & Tate, A. (1985) *Use and justification of algorithms for managing temporal knowledge in O-Plan.* Technical Report AIAI-TR-6, AIAI, Univ. of Edinburgh.

Berge, C. (1957) Two theorems in graph theory. *Proc. National Academy of Science*, **43**, 842-844.

Bobrow, D. G. & Stefik, M. (1983) *The Loops Manual.* Intelligent Systems Laboratory, Xerox Corporation, Palo Alto.

Bondy, J. A. & Murty, U. S. R. (1976) *Graph theory with applications.* MacMillan, London.

Bott, K. & Ballow, R. H. (1986) Research perspectives in vehicle routing and scheduling. *Transportation Research*, **20A**, 239-243.

Bradt, L. J. (1984) The Automated Factory. *Robotics World*, **2**, 18-19.

Burgam, P. M. (1984) JIT: On the move and out of the aisles. *Manufacturing Engineering*, **92**, 65-71.

Carré, B. (1979) *Graphs and Networks.* Oxford Univ. Press, Oxford.

Chakarabarti, P. P., Ghose, S. & Desarkar, S. C. (1986) Heuristic search through islands. *Artificial Intelligence*, **29**, 339-347.

de Champeaux, D. & Sint, L. (1977a) An improved bidirectional heuristic search algorithm. *Journal of the ACM*, **24**, 177-191.

de Champeaux, D. & Sint, L. (1977b) An optimality theorem for a bidirectional heuristic search algorithm. *Computer Journal*, **20**, 148-150.

de Champeaux, D. (1983) Bidirectional heuristic search again. *Journal of the ACM*, **30**, 22-32.

Cheeseman, P. (1983) *A representation of time for planning.* SRI Technical Note 278, Palo Alto.

Christofides, N. (1975) *Graph Theory—An Algorithmic Approach.* Academic Press, London.

Christofides, N., Mingozzi, A. & Toth, P. (1981a) Exact algorithm for the vehicle routing problem, based on spanning tree and shortest path relaxation. *Mathematical Programming*, **20**, 255-282.

Christofides, N., Mingozzi, A. & Toth, P. (1981b) State space relaxation procedures for the computation to bounds to routing problems. *Networks*, **11**, 145-164.

Clarke, G. & Wright, J. (1963) Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research*, **11**, 568-581.

Corkill, D. D. (1979) Hierarchical planning in a distributed environment. *Proc. IJCAI-79*, 168-175.

Corkill, D. D & Lesser, V. R. (1983) The use of meta-level control for coordination in a distributed problem-solving network. *Proc. IJCAI-83*, 784-756.

Critchlow, A. J. (1985) *Introduction to Robotics*. MacMillan, London.

Davies, R. & Smith, R. G. (1983) Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, **20**, 63-109.

Dean, T. (1985a) Temporal reasoning involving counterfactuals and disjunctions. *Proc. IJCAI-85*. 1060-1062.

Dean, T. (1985b) *Temporal Imagery: An approach to reasoning with time for planning and problem solving*. PhD Dissertation, Yale University.

Dechter, R. & Pearl, J. (1982) The optimality of A* revisited. *Proc. NCAI-82*, 95-99.

Dijkstra, E. W. (1959) A note on two problems in connexion with graphs. *Numerische*

Doran, J. E. (1966) Doubletree searching and the Graph Traverser. Memorandum EPU-R-22, Department of Artificial Intelligence, University of Edinburgh.

Doran, J. E. (1967) An approach to automatic problem solving. In *Machine Intelligence 1* (eds. Collins, N. L. and Michie, D.) Oliver and Boyd, Edinburgh.

Doran, J. E. (1968) New developments of the Graph Traverser. In *Machine Intelligence 2* (eds. Dale, E. and Michie, D.) Oliver and Boyd, Edinburgh.

Doran, J. E. (1985) The computational approach to knowledge, communication and structure in multi-actor systems. In *Social Action and Artificial Intelligence* (eds. Gilbert, N. and Heath, C. C.) Gower, London.

Doran, J. E. & Michie, D. (1966) Experiments with the Graph Traverser program. *Proc. Royal Society*, **294(A)**, 235-259.

Doran, J. E. and Corcoran, G. (1986) A computational model of production exchange and trade. In *To Pattern and the Past* (eds. Voorrips, A. and Loving, S.). Special issue of *Journal of the European Study Group on Physical, Chemical and Mathematical Techniques Applied to Archaeology (PACT)*, **11**, 349-359, 1986.

Doran, J. E. (1986) A contract-structure model of sociocultural change. In *Computer Applications in Archaeology* (ed. Laflin, S.) Univ. of Birmingham Computer

Drummond, M. E., Currie, K. & Tate, A. (1987) *Contingent plan structures for spacecraft.* Technical Report AIAI-TR-22, AIAI, University of Edinburgh.

Durfee, E. H., Lesser, V. R. & Cor
a distributed problem solving network. In *Distributed Artificial Intelligence.* (ed. Huhns, M. N.) Pitman, London.

Edmonds, J. (1965) Paths, trees and flowers. *Canadian J. Math*, **17**, 449-467.

Egerváry, J. (1931) Matrixok kombinatorikus tulajdonságairól. *Mat. és Fiz. Lapok*, **38**, 16-28. Translation by Kuhn, H. W. On Combinatorial Properties of Matrices. *George Washington University Logistics Papers*, **11**, 1955.

Elliot R. J. & Lesk, M. E. (1982) Route finding in street maps by computers and people. *Proc. NCAI*-82, 258-261.

Erman, D. L., Hayes-Roth, F., Lesser, V. R. & Reddy, D. R. (1980) The HEARSAY-II speech understanding system: Integrating knowledge to resolve uncertainty. *ACM Computing Survey*, **13**, 213-253.

Fikes, R. E., Hart, P. E. & Nilsson, N. J. (1972) Learning and executing generalized robot plans. *Artificial Intelligence*, **3**, 251-288.

Fikes, R. E. & Nilsson, N. J. (1971) STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, **2**, 189-208.

Finkel, R. A. (1986) *An operating systems vade mecum.* Prentice-Hall, London.

Floyd, F. W. (1962) Algorithm 97: shortest path. *Comm. of the ACM*, **5**, 345.

Ford, L. & Fulkerson, D. R. (1962) *Flows in Networks*. Princeton University Press, Princeton.

Garbow, H. N. (1973) *Implementations of algorithms for maximum matching on nonbipartite graphs*. PhD dissertation, Computer Science Dept., Stanford University.

Gaskell, T. (1967) Bases for vehicle fleet scheduling. *Operational Research Quarterly*, **18**, 281-295.

Gelperin, D. (1977) On the optimality of A*. *Artificial Intelligence*, **8**, 69-76.

Genesereth, M. R., Ginsberg, M. L. & Rosenchein, J. S. (1986) Cooperation without communication. *Proc. NCAI-86*, 51-57.

Georgeff, M. P. (1984) A theory of action for multiagent planning. *Proc. NCAI-84*, 121-125.

Georgeff, M. P. (1986) The representation of events in multiagent domains. *Proc. NCAI-86*. 70-75.

Gilett, B. & Miller, L. (1974) A heuristic algorithm for the vehicle dispatch problem. *Operations Research*, **22**, 340-349.

Gittins, M. (1987) LOOPS: A multi-paradigm environment. In *Artificial Intelligence Programming Environments* (ed. Hawley, R.) Ellis Horwood, Chichester.

Goldstein, I. P. (1975) Bargaining between goals. *Proc. IJCAI-75*, 175-180.

Gunn, T. G. (1982) The Mechanization of Design and Manufacturing. *Scientific American*, September issue, 86-108.

Hall, P. (1935) On representatives of subsets. *J. London Math. Soc*, **10**. 26-30.

Hart, P. E., Nilsson, N. J. & Raphael, B. (1968) A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Sys. Sci. Cybernetics*, **SSC-4**, 100-107.

Hart, P. E., Nilsson, N. J. & Raphael, B. (1972) Correction to: A formal basis for the heuristic determination of minimum cost paths. *ACM SIGART Newsletter*, **37**, 28-29.

Hayes, P. J. (1975) A representation for robot plans. *Proc. IJCAI-75*, 181-188.

Holmes, R. A. & Parker, R. G. (1976) A vehicle scheduling procedure based upon savings and a solution perturbation scheme. *Operational Research Quarterly*, **27**, 83-92.

Hopcroft, J. E. & Karp, R. M. (1973) An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Computing*, **2**, 225-231.

König, D. (1931) Graphs and matrices. (Hungarian) *Mat. és Fiz. Lapok*, **38**, 116-119.

Kuhn, H. W. (1955) The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, **2**, 83-97.

Kwa, J. B. H. (1988) On the consistency assumption, monotone criterion and monotone restriction. *ACM SIGART Newsletter*, **103**, 29-31.

Lin, S. & Kernighan, B. W. (1973) An effective heuristic for the Travelling Salesman Problem. *Operations Research*, **21**, 498-505.

Marsh, D. L. (1969) LIB Graph Traverser. *Multi-POP Program Library Documentation*, Dept. of Machine Intelligence and Perception, Edinburgh University.

Martelli, A. (1977) On the complexity of admissible search algorithms. *Artificial Intelligence*, **8**, 1-13.

McTamaney, L. S. (1987) Mobile robots—real-time control. *IEEE Expert*, **2**, 55-68.

Mérõ, L. (1981) Some remarks on heuristic search algorithms. *Proc. IJCAI-81*, 572-574.

Mérõ, L. (1984) A heuristic search algorithm with modifiable estimate. *Artificial Intelligence*, **23**, 13-27.

Michie, D. & Ross, R. (1969) Experiments with the adaptive Graph Traverser. In *Machine Intelligence 5* (eds. Meltzer, B. and Michie, D.) Edinburgh Univ. Press, Edinburgh.

Miller, D., Firby, R. J. & Dean, T. (1985) Deadlines, travel time, and robot problem solving. *Proc. IJCAI-85*, 1052-1054.

Moore, E. (1957) The shortest path through a maze. *Proc. of the International Symposium on Theory of Switching*, Part II.

Müller, T. (1983) *Automated guided vehicles*. Springer-Verlag, Berlin.

Munkres, J. (1957) Algorithms for the assignment and transportation problems. *J. of SIAM*, **5**, 32-38.

von Neumann, J. and Morgenstern, O. (1944) *Theory of games and economic behaviour*. Princeton University Press, Princeton.

Nicholson, T. A. J. (1966) Finding the shortest route between two points in a network. *Computer Journal*, **9**, 275-280.

Nilsson N. J. (1980) *Principles of Artificial Intelligence*. Tioga, Palo Alto.

Politowski, G. & Pohl, I. (1984) D-node retargeting in bidirectional heuristic search. *Proc. NCAI-84*, 274-277.

Peterson, J. L. & Silberschatz, A. (1985) *Operating system concepts*. Addison-Wesley, Reading, MA.

Pohl, I. (1969) *Bidirectional and heuristic search in path problems.* SLAC Report No. 104, Stanford, Palo Alto.

Pohl, I. (1970) Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, **1**, 193-204.

Pohl, I. (1971) Bidirectional search. In *Machine Intelligence 6* (eds. Meltzer, B. and Michie, D.) Edinburgh Univ. Press, Edinburgh.

Pohl, I. (1977) Practical and theorectical considerations in heuristic search algorithms. In *Machine Intelligence 8* (eds. Elcock,. E. W. and Michie, D.) Edinburgh Univ. Press, Edinburgh.

Rosenchein, J. S. (1982) Synchronisation of multiagent plans. *Proc. NCAI-82*, 115-119.

Rosenchein, J. S. (1985) *Rational interaction: Cooperation among intelligent agents.* Research Memorandum STAN-CS-85-1081, Stanford University.

Rosenchein, J. S. & Genesereth, M. R. (1985) Deals among rational agents. *Proc. IJCAI-85*, 91-99.

Sacerdoti, E. (1975) *A structure for plans and behaviour.* SRI AI Centre Technical Note No. 109, Palo Alto.

Schlesinger, R. J. & Tiersten, S. (1987) The competitive edge. *Hardcopy*, **7**, 78.

Stefik, M., & Bobrow, D. G. (1986) Object-oriented programming: Themes and variations. *The AI Magazine*, **6**, 40-62.

Taha, H. A. (1971) *Operations Research—An Introduction.* MacMillan, New York.

Tate, A. (1977) Generating project networks. *Proc. IJCAI-77*, 888-893.

Thorndyke, P. W., McArthur, D. & Cammarata, S. (1981) AUTOPILOT: A distributed planner for air fleet control. *Proc. IJCAI-81*, 171-177.

Tillman, F. A. & Cain, T. (1972) An upper bounding algorithm for the single and multiple terminal delivery problem. *Management Science*, **18**, 664-682.

Todd, D. J. (1986) *Fundamentals of Robot Technology.* Kogan Page, London.

Tsang, E. P. K. (1986) Plan generation in a temporal frame. *Proc. ECAI-86*, 479-493.

Tsuji, S. & Jiang, Y. Z. (1987) Visual path planning by a mobile robot. *Proc. NCAI-87*, 1127-1130.

Valéry, N. (1987) Factory of the future. *The Economist*, 30 May - 5 June.

Vámos, T. (1983) Cooperative systems—an evolutionary perspective. *IEEE Control Systems Magazine*, **3**, 9-14.

Vere, S. A. (1983) Planning in time: Windows and durations for activities and goals. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. **PAMI-5**, 246-267.

Vere, S. A. (1985) Temporal scope of assertions and window cutoff. *Proc. IJCAI-85*, 1055-1059.

Vilain, M. B. (1982) A system for reasoning about time. *Proc. NCAI-82*, 197-201.

Ward, B. & McCalla, G. (1982) Error detection and recovery in a dynamic planning environment. *Proc. NCAI-82*, 172-175.

Wesson, R. B. (1977) Planning in the world of the air traffic controller. *Proc. IJCAI-77*, 473-479.

Wesson, R. B. (1981) Network structures for distributed situation assessment. *IEEE Transactions on Systems, Man and Cybernetics*, **SMC-11**, 5-23.

Wilkins, D. (1985) *Recovering from execution errors in SIPE*. SRI Technical Note No. 346, Palo Alto.

Wilkins, D. (1982) *Domain independent planning: Representation and plan generation*. SRI Technical Note No. 266, Palo Alto.

Winston, P. H. (1984) *Artificial Intelligence*. Addison-Wesley, Reading, MA.

Wren, A. & Holliday, A. (1972) Computer scheduling of vehicles from one or more depots to a number of delivery points. *Operational Research Quarterly*, **23**, 333-344.

Yellow, P. C. (1970) A computational modification to the savings method of vehicle scheduling. *Operational Research Quarterly*, **21**, 281-283.