Decentralized Control of Multiple Collaborating Agents

by

Henry Wong

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Bachelor of Science in Computer Science and Engineering
and Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2001

Author .....
Department of Electrical Engineering and Computer Science
May 18, 2001

Certified by .....
Dr. Michael Cleary
Senior Member of Technical Staff, Draper Laboratory
Thesis Supervisor

Certified by .....
Leslie Pack Kaelbling
Professor, MIT Artifical Intelligence Laboratory
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Decentralized Control of Multiple Collaborating Agents

by

Henry Wong

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2001, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Computer Science and Engineering
and Master of Science in Computer Science and Engineering

Abstract

This thesis investigates the effect of various factors on multi-agent collaboration in a simulated fire-fighting domain. A simulator was written that models fires and fire-fighting agents in an area of a few square city blocks. Different scenarios were constructed to test the effects of limiting the agents' knowledge and to compare the effects of coordinating the agents through a single entity with the results obtained through independent decision making. This research demonstrates that the amount of information available to each agent and the agents' ability to act on this information are typically much more important factors than the use of a complex planning mechanism; as long as the agents are aware of each other and take minimal steps to coordinate their actions they are able to achieve results that are nearly as good as those achieved by much more complicated algorithms.

Thesis Supervisor: Dr. Michael Cleary
Title: Senior Member of Technical Staff, Draper Laboratory

Thesis Supervisor: Leslie Pack Kaelbling
Title: Professor, MIT Artifical Intelligence Laboratory

Acknowledgments

I'd like thank my supervisor, Michael Cleary, for his invaluable guidance and technical advice. I'd also like to thank Micheal Cleary and Mark Abramson for making a Draper Fellowship available to me, for giving me a topic to explore and for all of their support and enthusiasm towards my thesis.

I'd also like to thank my MIT advisor, Dr. Leslie Kaelbling, for explaining some of the harder parts of the mathematics to me and for providing constant advice on how I could improve my thesis and what steps I should take in order to accomplish the most in the given amount of time.

This thesis was supported by The Charles Stark Draper Laboratory Inc., under IR&D 13024.

.....

Henry Wong, June 2001

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Multi-agent collaboration, in which multiple entities cooperate to accomplish a shared goal, can be found throughout life. Research into automating such collaboration has targeted everything from soccer [11] to industry [6]. However, finding the optimal behavior for each agent in a group is a difficult task.

Constructing an optimal set of behaviors for every entity requires that every agent know the state of each of its peers and attempt to predict what actions they will take. This information must then be factored into the agents' decision making process. Suboptimal solutions, though more tractable, are of questionable value unless their degree of suboptimality is well characterized. This research identifies and quantifies the effects of various factors on the ability of a group of simulated fire fighting agents to extinguish a set of fires throughout a city. The following paragraphs discuss optimal and suboptimal approaches to multi-agent collaboration, the algorithms and comparison methods used in this work and the fire fighting domain these algorithms were tested in.

There are two common approaches to this problem, both of which use centralized planning. One approach is to assign optimal behaviors to each agent *a priori* [23], allowing each agent to carry out its plan independently of the other agents. Issuing *a priori* commands requires detailed

knowledge of the problem before a solution can be constructed. In addition, this style of multi-agent collaboration is extremely inflexible; if circumstances deviate at all from their expected state the entire plan may become useless. Another approach that has been taken towards multi-agent collaboration is to have some form of central coordination that issues commands to every agent in the group as the situation unfolds[5], requiring constant communication between group members.

If a central coordination mechanism with global knowledge issues commands to and receives information from every agent then the problem can be reduced to finding the optimal set of orders that the central coordinator should issue, given the state of the world. In this type of problem there is no real collaboration; only one entity makes any decisions. One method for calculating the optimal set of orders to be issued in any given world-state is through the use of Markov decision processes [10, 17] (MDPs). Appendix A provides a brief description of MDPs and the value iteration algorithms used herein.

Despite the attractiveness of optimal solutions, there are several drawbacks to the centralized approach. Centralized planning requires perfect communication between the coordinating entity and all of the agents and is hampered by damage to the coordinating agent, which is a single point of failure. In addition, development of an optimal multi-agent policy typically requires detailed knowledge of the problem as a whole, which is likely to be infeasible for realisitically complex problems.

Since agents in an urban environment typically do not have global knowledge of the entire city, and often have limited communications with each other or with a central coordinator. This research examines the effects of various factors on agent performance. These factors are the presence or absence of global knowledge, the presence or absence of central coordination, variations in the communication and vision range and the speed at which agents travel around the city. Five algorithms were developed to support review of the various factors. An optimal algorithm was designed

and implemented. This algorithm relies on global knowledge and perfect communications and is very computationally intensive for all but the most trivial problems. A near-optimal approximation algorithm was used in order to solve problems involving larger numbers of agents and fires. In order to examine the effects of central coordination, a third algorithm was developed that did not rely on central coordination but still allowed the agents global knowledge of the city. A fourth algorithm was also designed that required neither global knowledge nor central coordination; comparing this algorithm to the previous one helps to demonstrate the effects of global knowledge on agent effectiveness. Finally, a fifth algorithm was developed which did not rely on global knowledge or coordination and also imposed more realistic constraints on the rate at which agents could travel around the city. The effects of these constraints are discussed as well.

Since solving Markov decision problems is computationally intensive, the size of tractable problems that can be solved optimally is prohibitively small. As a result, a three-level comparison was performed between the optimal solutions (for problems of small size), near-optimal solutions created using an approximation algorithm discussed below and the algorithms discussed in this work.

For comparison purposes, all of the algorithms were run in situations for which an optimal, centralized solution could be constructed. In addition, the algorithms were run in situations for which construction of an optimal solution is computationally intractable; these iterations and their results are discussed in Chapter 6.

The agents' performance increased quickly as their range of knowledge increased, although it plateaued after a certain range. Agents with global knowledge but no centralized control performed nearly as well as centrally controlled agents with global knowledge. The results for problems of various sizes are discussed in Chapter 6.

Meuleau et al. designed an algorithm that approximates the optimal solution to an MDP [16].

This algorithm has been adapted for use in this thesis and compared with the results of the optimal algorithm. The results are discussed in Chapter 6. This approximation is used as a baseline for judging the results of this project on problems which are too large to solve optimally using MDPs.

This work deals with the fire fighting domain. Although prior research has already developed algorithms for use in this domain [2, 5, 12], these algorithms rely on either centralized control or knowledge of the entire world. This project uses a simple simulation that models a few city blocks; the simulator models the increase in intensity of the fires throughout the city as well as the efforts of the fire fighters to extinguish them. This simulator is used to compare the effectiveness of the various distributed algorithms, the near-optimal approximation and the optimal MDP solution. The RoboCup Rescue project [12] has developed a similar simulator, but it is unsuitable for this research for reasons that will be detailed in Chapter 6.

The remainder of this thesis is laid out as follows: Chapter 2 describes prior research done in this field including the requirements for the creation of an optimal solution and previous work aimed at developing near optimal solutions that bypass these requirements. It also further describes the goals and motivations behind this work. Chapter 3 discusses the simulator that is used in this research as well as the particulars of how fires and fire fighters are modeled. Chapter 4 discusses the stepping stone algorithm, which is used to calculate the cost of moving a set of agents between fires. Chapter 5 discusses the algorithms implemented during this research and the situations in which they are applicable. Chapter 6 discusses the tests performed on the various algorithms and the results that were obtained. Chapter 7 summarizes the thesis and presents an interpretation of the results as a whole. Chapter 8 discusses some of the possible future extensions to this work. Appendix A discussed the Markov Decision Problems used to model the simulation and the value iteration algorithm used to provide optimal agent behavior.

Chapter 2

Related Work in Fire Suppression

One application for which multi-agent collaboration is currently being explored is cooperative fire-fighting [2, 5, 12], where simulated fire fighting agents work together to either contain [5] or extinguish [2, 12] fires. The following sections describe related work in modeling fire spread and fire suppression.

2.1 Previous Fire Models

Much of the previous work in fire simulation has focused on creating a realistic fire model that is simple enough computationally to be used in a simulation involving many fires spreading over a wide area of land. Several of these approaches are documented below.

2.1.1 A Simple Grid of Cells

One widely used approach is to model the fire as a lattice of cells. The cells are either on fire or not (there is no intensity) and burn through their fuel supply at a predetermined rate. Individual cells have their own fuel supply. At every time step there is some probability that the fire will spread to nearby cells based on factors such as wind, hill gradient, etc. This model treats each cell

as a separate fire.

Previous work using this model has represented fire suppression in various ways. Cohen has the fire-fighting agents contain the fire and wait for it to run out of fuel [5]. Since Cohen was simulating large scale forest fires and the agents were often far apart from each other, he relied on a centralized "fire boss" to coordinate the agents when they were far apart.

Andrade models fire suppression efforts by reducing the amount of fuel that a particular cell has, thus decreasing the time before it burns out [2]. Andrade's model requires agents to reduce the fuel for each cell individually. Andrade modeled fire fighting in individual rooms on a ship. Thus, each agent had complete knowledge of the room it was in, including the location of each of the fires.

2.1.2 Rothermel's Work

Many fire simulations (e.g., [8, 15]) which model the spread of fires in the wilderness are derived from work developed by Rothermel in 1972 [18]. These systems model a fire as a collection of cells, which spreads by igniting nearby (non-burning) cells. Unlike the simple grid of cells discussed previously, individual cells can have different intensities which affect how likely the fire is to spread to neighboring cells. The intensity of a cell is based on the type of fuel being burned and is not time dependent. Each cell is described by its fuel type, amount of fuel, current state (i.e., on fire or not), susceptibility to being lit on fire by other fires and many other factors. Additionally there are global factors such as terrain, wind conditions and moisture in the air that affect all of the cells. Fire suppression is not typically studied using this type of simulation; these models are generally used to study how a fire will spread unchecked through a forest [7].

2.1.3 The RoboCup Rescue Simulator

The RoboCup Rescue project [12] has developed a simulation that will allow multiple agents to cooperatively fight fires with only local knowledge and limited communications. This promises to be a very useful environment for future research. However, the current efforts of the RoboCup Rescue development team are focused on improving the simulator, rather than designing or implementing the agents' strategies, making it less applicable to this thesis. Since the RoboCup Rescue project has just begun [20], the algorithms used in the project are not very sophisticated. For example, each agent simply heads towards the closest fire, regardless of how many other agents are heading towards or already at that same fire.

Several characteristics of the current implementation of the RoboCup Rescue simulator make it unsuitable for the current research. They include the inability to reset the simulator state, the lack of distance and line-of-sight constraints on agent vision and agent communication, and the need for a more realistic fire fighting model.

The fire models discussed in this chapter are summarized in table 2.1.

	Simple Grid	Rothermel's Model	RoboCup Rescue
Same intensity across fire	Yes	No	No
Intensity changes over time	No	No	Yes
Fires consume fuel	Yes	Yes	Yes
Used to model fire suppression	Yes	No	Yes

Table 2.1: Fire Models

Chapter 3

The Simulator and Fire Model

This chapter discusses the simulator used in this research (in section 3.1) and describes the way that fire and fire suppression were modeled (in section 3.2). Each of the agents in the simulator determines and executes its plan of action.

3.1 The Simulator Engine

The simulator used in this research is a simple Java program that models fires and fire fighting agents within a few city blocks. The city is represented as a grid of nodes of three types ("road", "fire" or "building"). "Building" nodes can catch on fire. Fires that are extinguished are still treated as "fire" nodes, but with zero intensity. Each fire has its own intensity, discussed in section 3.2. The simulator models constraints such as vision and communications bounded by line-of-sight and distance.

The simulator consists of a single main loop that first updates all of the fires (increasing or decreasing their intensity if appropriate), then provides the agents with the information available to them (about the entire world if the scenario allows global knowledge, otherwise only about the

applicable portions of the world) and runs each agent's decision making routine. As part of the decision making process, each agent decides which fire to fight or, if central coordination is present, is assigned to a fire. Figure 3-1 shows the simulator display. The black squares forming a grid through the picture indicate roads. The magenta circle at the intersection is a fire fighting agent. The two light gray squares above and to the right of the agent are low intensity fires. The other squares represent unburned ground.

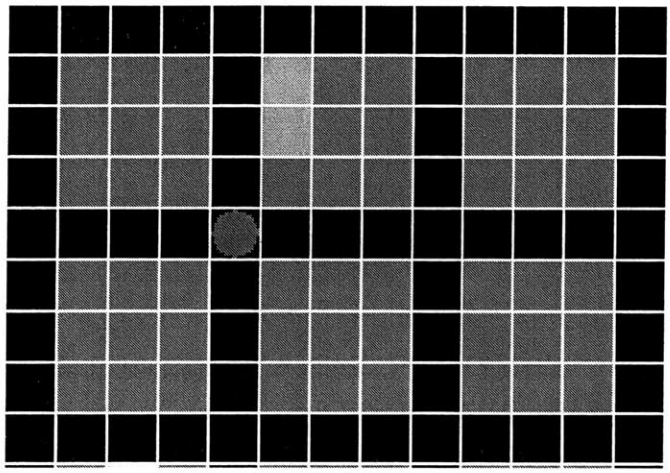

Figure 3-1: The Simulator display

3.2 The Hybrid Fire Model

This section describes the model used to represent fires in this research. This hybrid model retains many of the characteristics of the previously discussed fire models, but is simplified for use with MDPs.

Each fire has one of five intensity levels; as the fire reaches each new intensity level it does more damage. For example, a level one fire might scorch the paint off walls, while a level four fire might cause total destruction. Level zero is used to denote fires that have been extinguished. There is an *a priori* probability that a fire at one particular level of intensity will jump to the next level

of intensity; this probability is based only on the fire's current level. The probability of the fire increasing in intensity goes up as the fire intensity increases. For example, level-three fires have a 60% chance of becoming level-four fires in the next time step whereas level one fires have only a 15% chance of becoming level-two fires in the next time step. Table 3.1 lists the odds of fires at various intensities increasing to the next level.

Intensity of Fire	Probability of Increase
0	0
1	.15
2	.35
3	.6
4	0 (Maximum intensity)

Table 3.1: Probability of fire growth

There is also some probability that a fire will decrease by one level of intensity. This probability is a function of the number of fire fighters fighting the fire and the current intensity of the fire. The effectiveness of each additional fire fighter decreases as the number of fire fighters assigned to the fire increases. For example, having one fire fighter fight a particular fire might provide a 10% chance that the fire will decrease in intensity while having two fire fighters might only provide a 17% chance that fire's intensity will decrease. Table 3.2 summarizes the properties of the hybrid fire model.

	Hybrid Fire Model
Same intensity across fire	Yes
Intensity changes over time	Yes
Fires consume fuel	No
Used to model fire suppression	Yes

Table 3.2: Hybrid Fire Model

The hybrid model is very simple, but it has several important characteristics that make it

useful to this project. Fires that are not extinguished will eventually increase in intensity and, more importantly, will continue to increase in intensity at a faster and faster rate. This forces the agents to attempt to fight every fire they have a realistic chance of stopping, because fires that grow past a certain threshold will be very hard to control. In addition, the decreasing effectiveness of additional fire-fighters means that the agents need to spread out their efforts for maximum gain while still concentrating their efforts enough to fight fires that are on the verge of becoming uncontrollable.

It is very important that the fire model be kept as simple as possible in order to increase the size of the problems that are optimally solvable via MDPs. As a basis for comparison, it took a Sun UltraSparc 20 with a 333 MHz processor one hour to solve the MDP corresponding to four fires, each using the hybrid fire model, and seven fire fighting agents. It took the same computer six hours to solve a two fire, four agent problem when the problem used the RoboCup Rescue fire model because the RoboCup Rescue fire model allows the same fire to have different intensities at different points across its surface and also keeps track of the fuel consumption of each fire.

Chapter 4

The cost of moving agents during a single action

In this research, the "actions" evaluated by a Markov Decision Process (MDP) are simply assignments of agents to fires. At any given time the agents are already at various locations. Thus, the reward for a given action has to take into account the cost of moving the agents from their current locations to the ones designated by the MDP. This chapter describes an algorithm used by the simulator to quickly compute these costs.

For example, consider a situation with four fires (A, B, C, D) and two agents (a, b), where agent "a" is at fire "A" and agent "b" is at fire "B". The cost to assign an agent to "C" and an agent to "D" is the minimum of $C(A,C)+C(B,D)$ and $C(A,D)+C(B,C)$, where C(X, Y) is the cost of moving an agent from location X to location Y. Figure 4-1 illustrates this problem.

This type of problem is a transportation problem [21]; it is a special case of a weighted bipartite matching problem which is in turn a special case of a min-cost flow problem. For computational efficiency reasons, the algorithm implemented in this research is Charnes and Cooper's "stepping

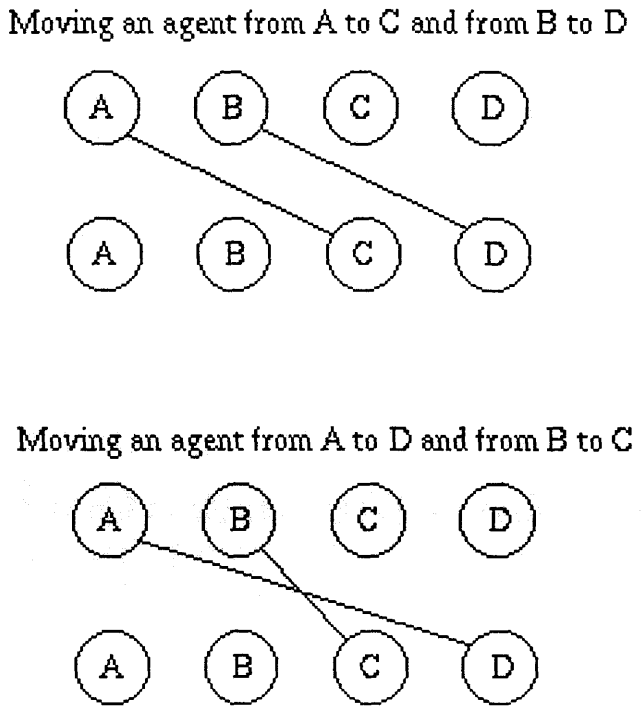

Figure 4-1: Illustration of sample transportation problem

stone" algorithm [4], which is described in detail below.

The stepping stone algorithm is based on the network simplex method, and thus has a similar running time. Although there is no known method that will make the simplex method run in polynomial time [19], in practice the network simplex method, and thus the stepping stone algorithm, run very quickly.

Since the transportation problem is a special case of a min-cost flow problem, it is possible to use min-cost flow algorithms to solve it. However, an average problem uses very few iterations of the stepping stone algorithm and thus runs considerably faster than most min-cost flow algorithms. For example, if the number of nodes in a graph is n and the number of edges if m, the running time of the algorithm described by Galil [9] grows with $O(n^2(m + n\ log\ n)\ log\ n)$. By comparison, a single iteration of the stepping stone algorithm grows in $O(m + n)^2$ time. The stepping stone algorithm is used to calculate the cost of moving a set of agents from their current locations to a

set of new locations. This cost is then factored into the instantaneous reward associated with a given action in the MDP.

4.1 Overview of the stepping stone algorithm

The stepping stone algorithm minimizes the cost of transporting agents from m sources to n destinations along mn direct routes from source to destination. Each path from a source to a destination has a cost associated with it. That cost is applied to each agent that travels along the route. The algorithm is based on the simplex method of solving linear programming problems; it begins by creating a feasible solution to the problem and then refining that solution until an optimal solution is reached.

Before the stepping stone algorithm is called, the simulator determines which locations have more agents than are needed and which locations require additional agents. The simulator assumes that each fire either "supplies" agents or "receives" agents, but not both. Since the travel cost is linearly proportional to the distance traveled, it is always possible to construct an optimal solution that does not require an agent to leave a location and a different agent to enter that location. Although this isn't neccessary, it speeds up the calculation of the final solution considerably because it means that the stepping stone algorithm only needs to consider the number of agents either "supplied" or "received" at each fire, rather than the total number of agents. Figure 4-2 demonstrates one such reduction. The pre-reduction circles indicate the number of agents at each location for a given time and the post-reduction circles indicate only the agents that the stepping stone algorithm will consider.

The stepping stone algorithm begins by generating a table correlating the various costs, supplies and demands. Figure 4-3 shows a sample table. Each row represents a source location; the number

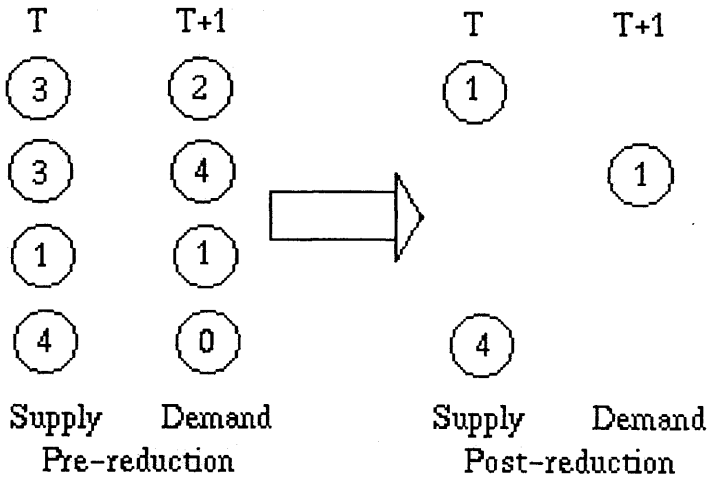

Figure 4-2: Reduction in the number of agents considered

of available agents is shown in parentheses next to the location name. Each column represents a destination and lists its total demand in parentheses.

Source (Supply) \ Destination (Demand)	D1(4)	D2(2)	D3(1)	D4(6)
S1(2)	2	1 (2)	3	4
S2(5)	3	7	5 (1)	8 (4)
S3(6)	2 (4)	6	1	3 (2)

Figure 4-3: Cost/Supply/Demand table ("Cost Table")

For example, the cell (S1, D2) in figure 4-3 shows the cost of transferring an agent from *S1* to *D2* (the cost is 1) and the number of agents being sent from *S1* to *D2* (2 agents are being moved). In order for a solution to be feasible, the values in each row must add up to the total supply available at that location and the values in each column must add up to the total demand

at that location. For example, $(S2, D3) + (S2, D4) = S2_{Supply} = 5$. Figure 4-3 shows a sample table with costs, supplies, demands and cell values. Such tables will be referred to as "cost tables."

Sources with no surplus agents and destinations requiring no additional agents are dropped from the table. The cost of transporting an agent from a source to a destination (the cost of the cell) is indicated in the upper left corner of the corresponding cell. The number of agents being transported along that route (the value of the cell) is also shown. For the purposes of clarity, cells that have not been marked as "basic" (see below) do not have their value displayed in any of tables in this section; each of these "non-basic" cells has a value of zero. Figure 4-4 shows a sample cell with cost 3 and value 4; the cell is marked as "basic".

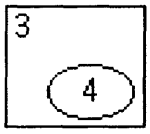

Figure 4-4: A sample cell

A "path" of cells is a sequence of cells in the cost table. The cells in a path do not have to be adjacent in the cost table but each cell must share either a row or column with the both previous and next cells along the path. Figure 4-5 shows a path through various cells of the cost table. The path starts at (S1, D1) and includes (S3, D1), (S3, D3), (S2, D3), (S2, D4) and (S3, D4). Note that the path does not include (S2, D1) or (S3, D2). A path cannot include the same cell more than once except when it forms a cycle, as noted below. Thus (S2, D1), (S2, D2), (S2, D1), (S3, D1) is not a valid path. In all diagrams in this section, the arrow heads will denote cells actually included in the path.

A "cycle" is a path of cells that starts and ends at the same cell. Cycles "originate" at the first cell in the path. Figures 4-6 and 4-7 show cycles in a table; both cycles originate at cell (S3, D1).

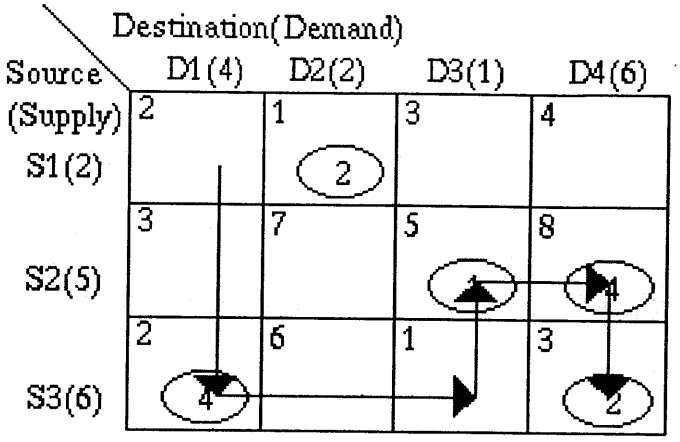

Figure 4-5: Path

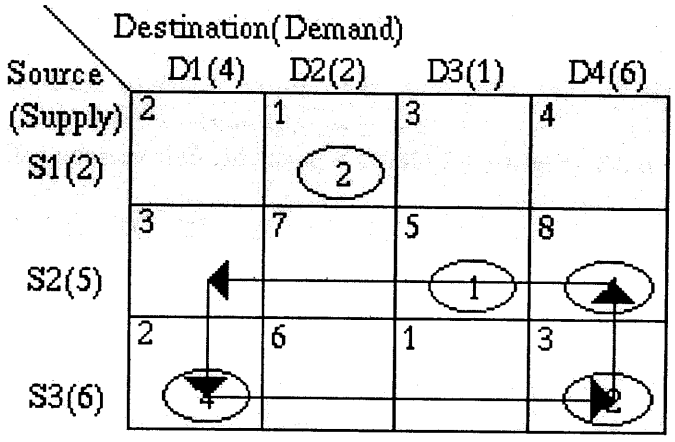

Figure 4-6: A sample cycle

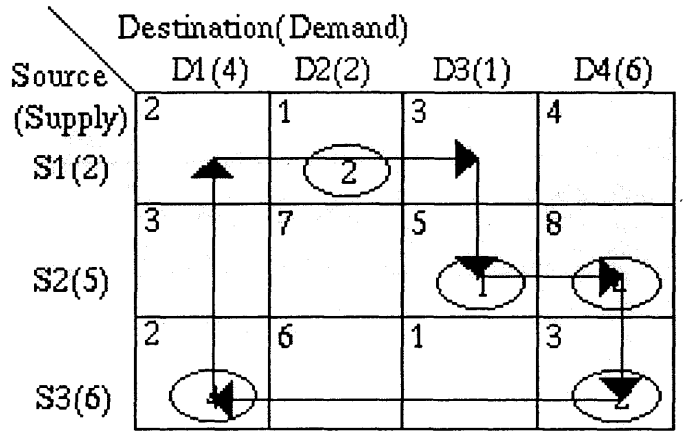

Figure 4-7: Another sample cycle

A set of "connected" cells is an unordered set of cells such that there is a path between any two cells in the set made up entirely of cells within the set.

A "tree" of cells is an set of connected cells such that there are no cycles made up of cells in the tree. Figure 4-8 shows a tree of cells. The tree contains the cells (S2, D1), (S1, D3), (S2, D3), (S3, D3), (S3, D4). For example, (S2, D1), (S2, D3), (S3, D3) is a path between (S2, D1) and (S3, D3) composed entirely of cells in the tree.

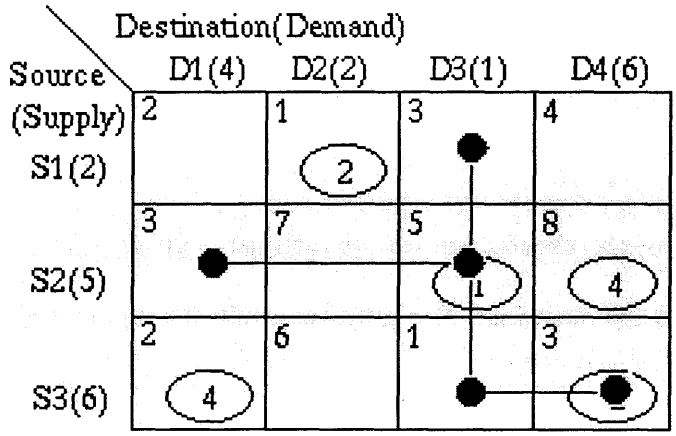

Figure 4-8: A tree of cells

A "spanning tree" is a tree which contains a cell in every row and every column of the cost

table. Figure 4-9 shows a spanning tree of cells.

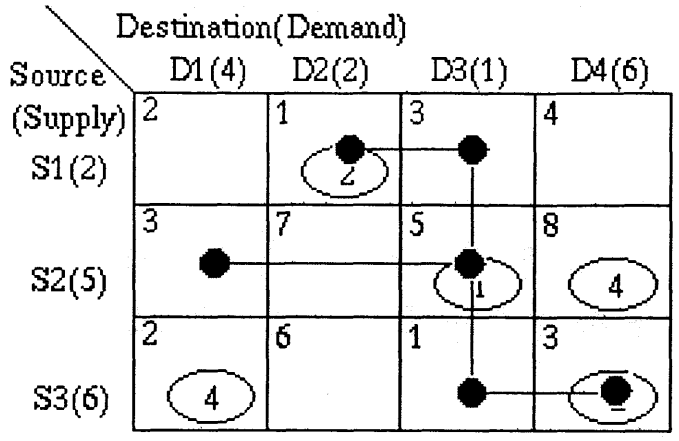

Figure 4-9: A spanning tree of cells

4.1.1 Step 1 - Initial Solution

The stepping stone algorithm uses a greedy approach to create the initial feasible solution. The cell with the lowest cost among those whose supply and demand constraints are both unfulfilled (greater than zero) is chosen and enough agents are assigned to that cell to fulfill the smaller of the two constraints on that cell. If two cells with unfulfilled constraints have the same cost than either of them may be used. As agents are assigned to each cell the remaining supply and demand totals on that cell are reduced. This process is repeated until every supply and demand constraint has been satisfied; proof that all of the constraints will be satisfied is given below. Figures 4-10, 4-11, 4-12 and 4-13 show this process occuring. The grey bars indicate supply and/or demand constraints that have already been fulfilled and the grey numbers represent the value of the reduced constraints.

Note in figure 4-11 that the *S3* constraint has been reduced to 5, but not yet fulfilled.

In figure 4-12, cell (S3, D1) was chosen because it has the lowest cost among the cells with both constraints unfulfilled. Although cell (S3, D3) has lower cost, one of its constraints has already

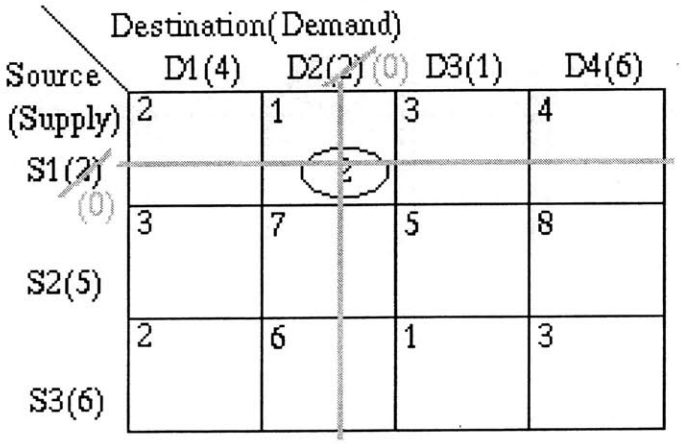

Figure 4-10: Beginning the greedy allocation

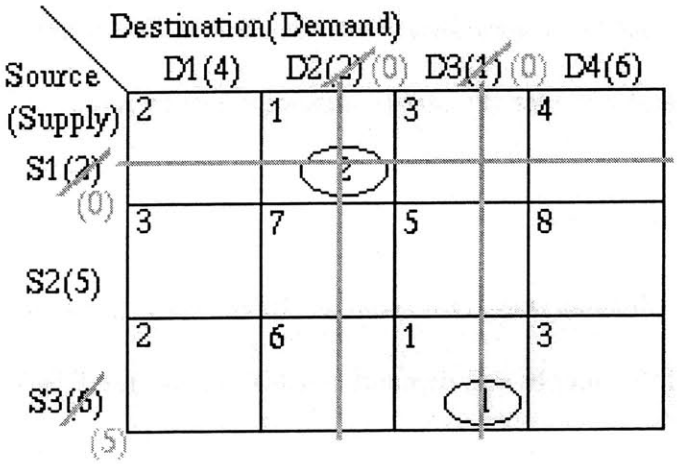

Figure 4-11: After two allocations

been fulfilled.

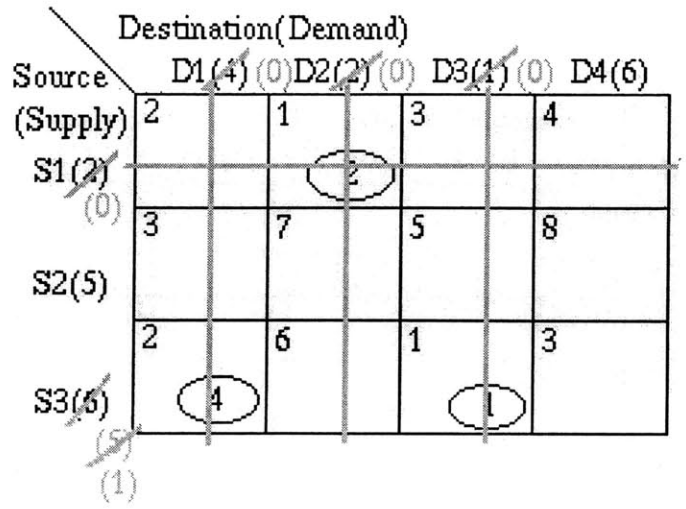

Figure 4-12: After three allocations

Assigning a value to a cell must fulfill at least one constraint, and the last cell must fullfill two constraints since the total supply and total demand are the same. Since the allocation finishes when all of the constraints are fulfilled, the greedy algorithm requires filling at most $m + n - 1$ possible cells where m is the number of sources and n is the number of destinations. In the above example, the first allocation fulfilled two constraints (both the *S1* and *D2* constraints) so only $m + n - 2$ cells were required. Pseudocode for the greedy allocation can be found in figure 4-14.

4.1.2 Step 2 - Basic Cells

The cells chosen in the previous step are marked as "basic" cells. Since each basic cell is chosen on the condition that both its supply and demand constraints are unfulfilled, the basic cells will not form a cycle. If there are less than $m + n - 1$ basic cells, then additional cells are marked as basic until there are $m + n - 1$ basic cells. The additional cells are chosen in such a way that there are no cycles among the basic cells. Proof that these additional cells always exist, as well as a method to pick them, is discussed below.

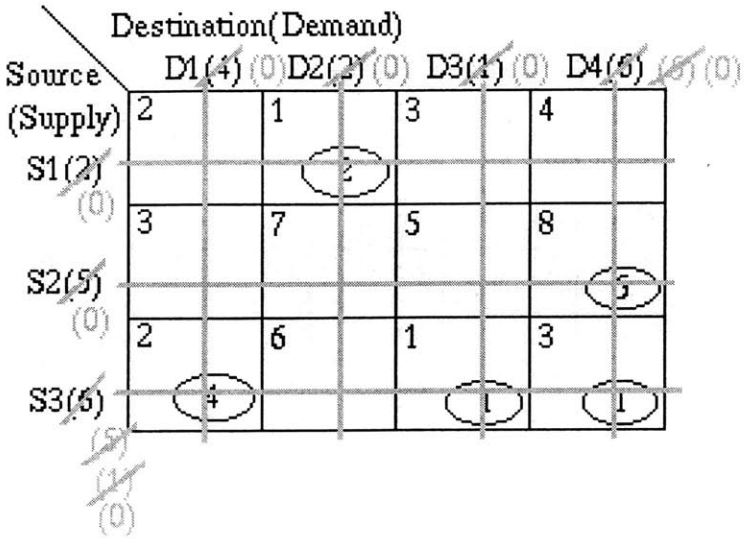

Figure 4-13: The final allocation

```
notDone = true

// loop until no more unfulfilled constraints
while notDone
      do notDone = false
         min_i = 0
         min_j = 0

         // find minimum cost cell among cells with
         // two unfulfilled constraints                                    10
         for i = 0 to length[columns]
             do for j = 0 to length[rows]
                    do if table[i][j].cost < table[min_i][min_j].cost and
                          columns[i].constraint > 0 and
                          rows[j].constraint > 0
                          then min_i = i
                               min_j = j
                               notDone = true

         if notDone                                                       20
            // allocate agents to that cell
            then table[i][j].value = minimum(columns[i].constraint, rows[j].constraint)
                 columns[i].constraint = columns[i].constraint - table[i][j].value
                 rows[j].constraint = rows[j].constraint - table[i][j].value
```

Figure 4-14: Psuedocode for the greedy allocation

Figure 4-15 shows cell (S1, D1) being labeled as basic. The value of the cell is still zero.

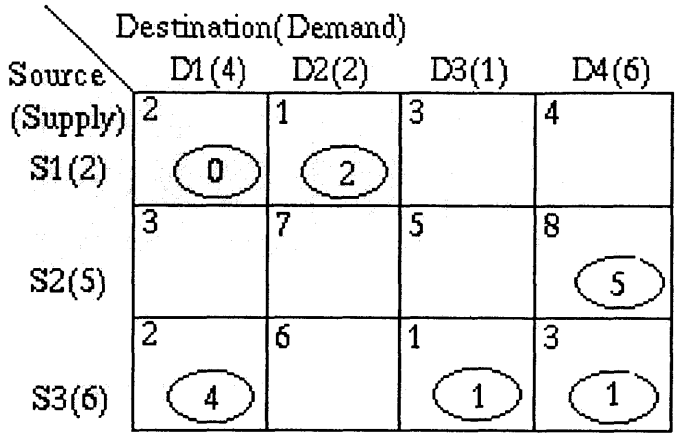

Figure 4-15: Labelling an additional cell as basic with value zero

4.1.3 Step 3 - Dual Costs

Improving the initial solution requires calculating the "dual" costs for every cell in the cost table. As in the Network Simplex Method, the dual cost of a cell reflects the potential change in the value of a solution if that cell were included. Including cells with a negative dual cost would improve the solution while including cells with a positive dual cost would degrade the solution. Cells already included in the solution have a dual cost of zero, as do cells whos inclusion into the solution would not change the value of the solution.

Before the dual costs for the cost table cells can be calculated, the "dual values" for the various rows and columns must first be computed. These values will be used to compute the dual costs of the individual cells, but have no meaning on their own. Calculating the dual values for the various rows and columns can be done by using the cost equations for the basic cells. The cost equations read:

$$U_i + V_j = C_{ij} \tag{4.1}$$

Here C_{ij} is the given cost for the basic cell, U_i is the dual value for row i and V_j is the dual value for column j. Since there are $m + n - 1$ dual equations (one for each of the basic cells) and $m + n$ values to be computed, U_1 is set to zero.

Once the dual costs for the rows and columns are calculated, the reduced cost for non-basic cell (i, j) (RC_{ij}) is calculated as follows:

$$RC_{ij} = C_{ij} - (U_i + V_j) \tag{4.2}$$

The reduced cost for a non-basic cell represents the change in the value of the solution which results from making cell ij basic and making adjustments around the cycle of basic cells thus created in the solution. There were originally $m + n - 1$ basic cells, marking an additional cell as basic creates $m + n$ basic cells. Proof that any collection of $m + n$ cells must include a cycle can be found below.

If all of the reduced costs for non-basic cells are non-negative then the solution is optimal; including additional cells in the solution will not improve the solution at all. Proof of this is discussed below, under the heading "Optimality Conditions" in the "Proof of Correctness" section. If one of the non-basic cells has a negative reduced cost, it can be added into the solution in the following manner.

4.1.4 Step 4 - Negative Reduced Cost Cycles

Consider a cycle of cells that involves a non-basic cell with a negative reduced cost and various basic cells. The $m + n - 1$ basic cells were chosen such that they do not form a cycle. Since it is

impossible to pick $m+n$ cells such that there are no cycles (see section 4. 4.1.1) it must be possible to choose such a cycle. This cycle is refered to as a "negative cost cycle." Figure 4-16 shows the cycle formed from basic cells and a single cell with a negative reduced cost. Note that the cell values used in figure 4-16 were not generated using the greedy algorithm but do represent a feasible solution to the transportation problem. Numbers in gray represent reduced values. Cell (S3, D2) has been marked as basic (with value zero) in order to ensure that there are $m+n-1$ basic cells.

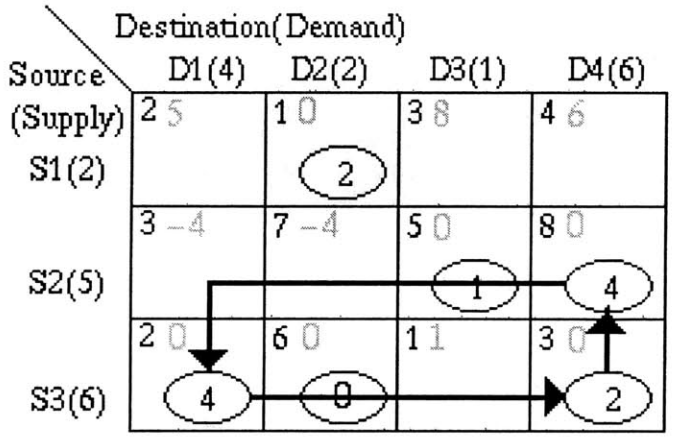

Figure 4-16: A cycle formed from a cell (S2, D1) with negative reduced cost

The first cell in the cycle (the cell with negative reduced cost; (S2, D1) in the example) is labeled "positive." The other cells are alternately labeled "positive" or "negative." The "flow" around this cycle is defined as the value of the smallest "negative" cell. In figure 4-16 the flow would have value 4.

The flow around this negative cost cycle is then subtracted from every negative cell and added to every positive cell. The first positive cell is then labeled basic, and one of the negative cell which now has zero value is labeled non-basic. Figure 4-17 shows the cost table after the flow around the cycle has been added to every positive cell and subtracted from every positive cell; cell (S2, D1) has been made basic and cell (S3, D1) has been made non-basic. The dual values for each row

and column are recomputed using the new set of basic cells and the dual costs for each cell are recomputed using the new dual values. The process is then repeated until none of the non-basic cells have a negative reduced cost. Note that the combined cost of the four cells in the cycle in figure 4-16 is $C(S2, D1)V(S2, D1) + C(S3, D1)V(S3, D1) + C(S2, D4)V(S2, D4) + C(S3, D4)V(S3, D4) = 3 \cdot 0 + 2 \cdot 4 + 3 \cdot 2 + 8 \cdot 4 = 46$ where *C(i, j)* is the cost of cell (i, j) and *V(i, j)* is the value of cell (i, j). The cost of the same four cells after cell (S2, D1) was added to the solution and cell (S3, D1) was removed from the solution (in figure 4-17) is $3 * 4 + 8 \cdot 0 + 3 \cdot 6 + 2 \cdot 0 = 30$. Psuedocode for adjusting a solution to handle a negative cost cycle can be found in figure 4-18. *start_cell* is the negative reduced cost cell found earlier.

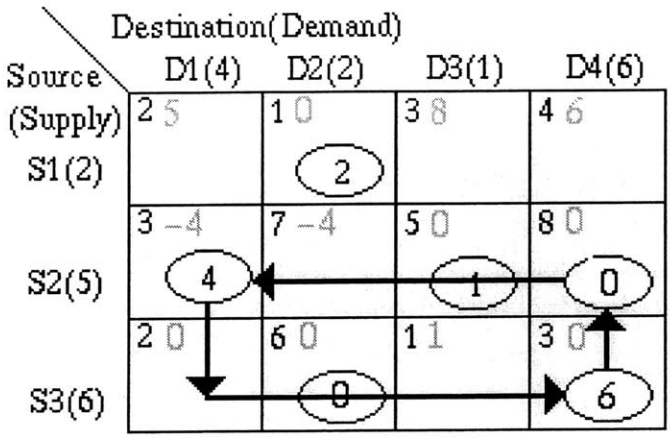

Figure 4-17: An improved solution.

The *search* procedure mentioned on line 2 returns an array of the cells found in the cycle created by the basic cells and the negative reduced cost cell. Proof that this cycle can always be found is given below; a simple depth-first search algorithm can be used to find the particular cells. If any three or more cells in the cycle all share the same row or the same column then only the first and last cells in that row or column are included in the cycle. Figures 4-19 and 4-20 demonstrate this

```
// get cells in cycle formed from basic cells and start_cell
cycle = search(start_cell)

// find minimum value among "negative" cells
min_negative = positive_infinity
min_index = -1
for i = 0 to length[cycle]
    do if i mod 2 == 0 and                    // it's a "negative" cell
          min_negative < cycle[i].value
          then min_negative = cycle[i].value                                  10
               min_index = i

// increment/decrement values
start_cell.value = start_cell.value + min_negative
for i = 0 to length[cycle]
    do if i mod 2 == 0
          then cycle[i].value = cycle[i].value - min_negative
          else cycle[i].value = cycle[i].value + min_negative

// relabel start_cell as basic                                                20
start_cell.basic = true

// relabel the minimum negative cell as non-basic
cycle[min_index].basic = false
```

Figure 4-18: Psuedocode for handling a negative cost cycle

process.

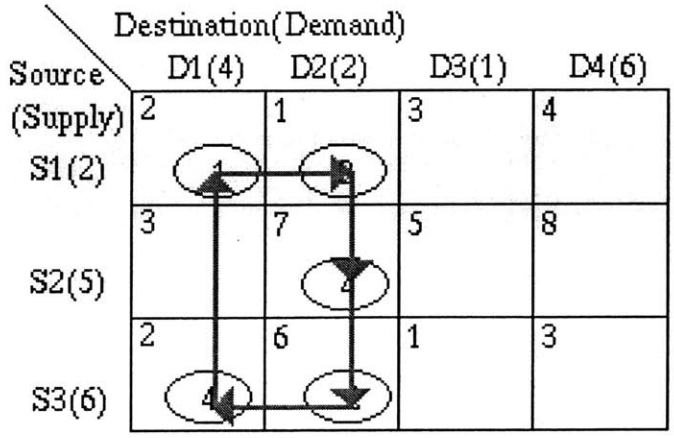

Figure 4-19: A possible cycle

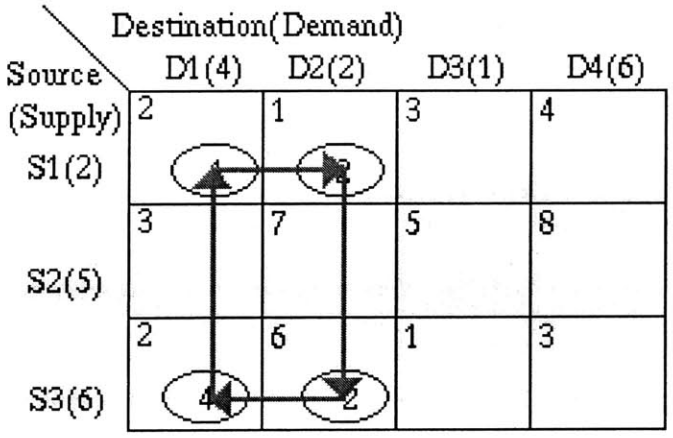

Figure 4-20: A cycle with fewer cells

Removing cells in this fashion can never result in the negative-reduced cost cell (the non-basic cell) being removed, because doing so would form a cycle of only basic cells. The set of basic cells must be acyclic. Proof of this is given in below.

This process results in a feasible solution with lower total cost unless the flow around the negative cost cycle is zero, in which case the total cost will not change. If the flow is zero and another non-basic cell has a negative reduced cost then the same procedure is applied to it. If the

only cell with negative reduced cost is in a cycle with zero flow then the solution is optimal.

If the flow was not zero (and the algorithm has not terminated) then the cells are labeled basic or non-basic as appropriate and the algorithm is repeated. If more than one cell in the cycle has been reduced to zero flow only one of them is labeled non-basic.

This process continues until the reduced cost for each of the non-basic cells is non-negative. Proof that the algorithm eventually terminates and that it produces an optimal solution can be found below.

4.2 Proof of correctness of the stepping stone algorithm

This proof is divided up into sections detailing various stages of the stepping stone algorithm. [4]

4.2.1 Constructing the initial solution

Theorem 1 *At most $m + n - 1$ cells are chosen by the greedy allocation.*

Each basic cell chosen during the greedy allocation is chosen such that it fulfills either the supply or demand constraint for its row or column. Since the sum of all of the supply constraints equals the sum of all of the demand constraints and the value of each basic cell is subtracted from both a supply and a demand constraint, the final basic cell must satisfy two constraints. Thus, there must be at most $m + n - 1$ cells.

Theorem 2 *A set of $m + n$ or more spanning cells must form a cycle.*

A set of spanning cells includes a cell in every row and every column of the cost table. The cells can be matched up with the rows and columns in a manner similar to that used to create

the initial greedy solution; one of the cells is matched up with both its row and column. Any cell who's column or row has been matched to a different cell can then be matched to its remaining, unmatched dimension. Figure 4-21 shows a set of $m + n$ cells. Cell (S3, D1) has been matched with both its row and its column.

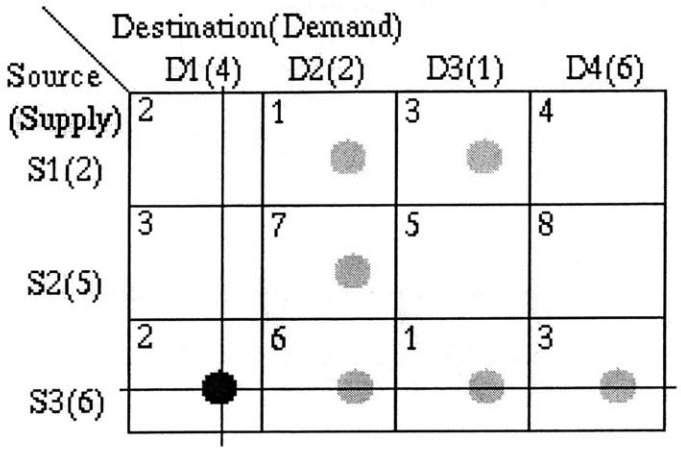

Figure 4-21: One cell matched to two dimensions

Figure 4-22 shows cell (S3, D2) matched with its column. (S3, D2) was chosen because it was one of the cells in the set with only one unmatched dimension.

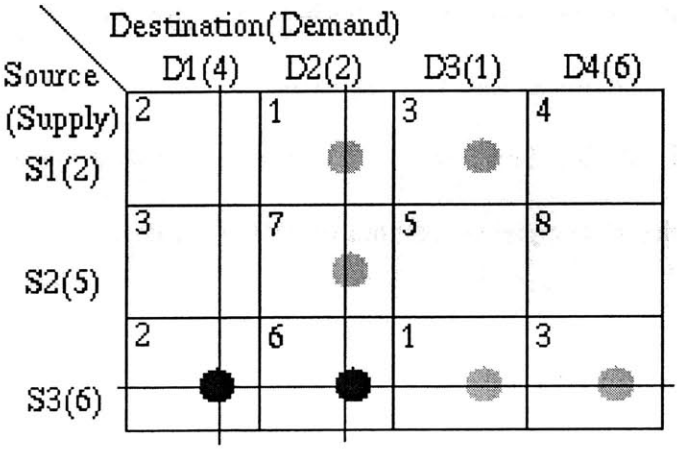

Figure 4-22: (S3, D2) matched to its column

The first cell was matched with both its row and its column. Since there are $m + n$ cells and $m + n$ rows and columns, one cell, X, will not be matched with either its row or its column. Figure 4-23 shows all of the rows and columns matched. Cell (S1, D2) was not matched with a row or column.

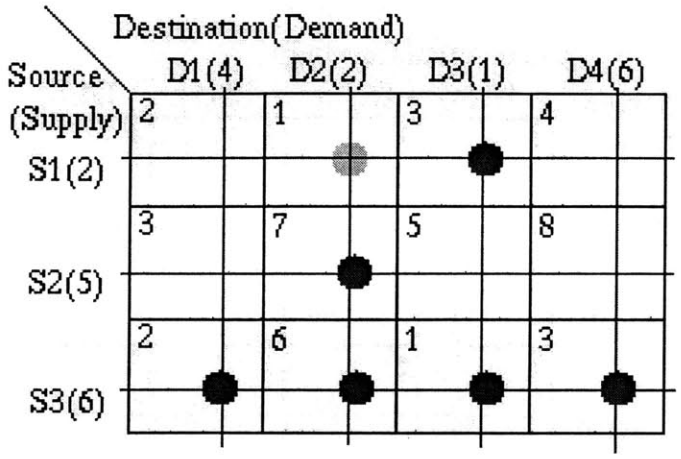

Figure 4-23: (S1, D3) is unmatched

This cell is part of a cycle. Since X was not matched to its row, but its row was matched up with a cell, there must be another cell Y in the same row as X that was matched with the row X is on. In figure 4-23, X is cell (S1, D2) and Y is cell (S1, D3).

Y was matched with either its column or with both its row and column. If Y was matched with only its row then that means that there must be another cell in the same column as Y that was matched with that column. This new cell, Y', is connected to X by the path Y'->Y->X and is similarly matched with either just its column or with both its row and column. In figure 4-23, cell (S3, D3) is Y'.

This path can be continued until it reaches Z, the cell which was originally matched against both its row and its column. In figure 4-23, Z is cell (S3, D1). Z was matched up with the row that Y'' is on, thus the path goes from (S1, D2) to (S1, D3) to (S3, D3) to (S3, D6). Figure 4-24

shows the complete path.

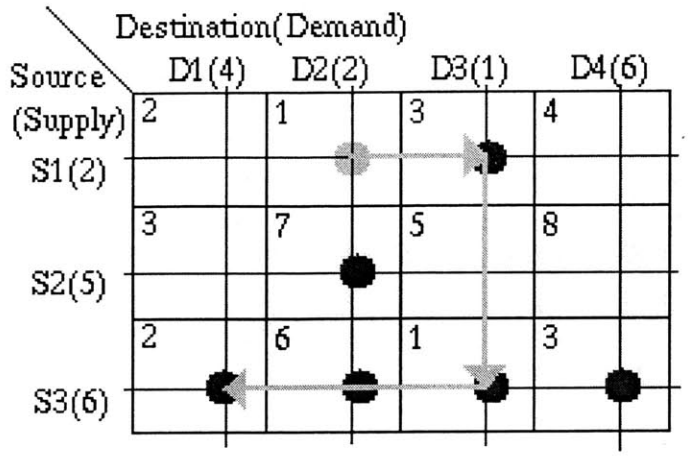

Figure 4-24: A path from X to Z

A similar path can be constructed beginning with the cell in the same column as X. This path can also be extended until it reaches Z. There are now two distinct paths connecting X to Z; reversing the direction of one of these paths will form a cycle among the $m + n$ cells. Figure 4-25 shows the second path from X to Z.

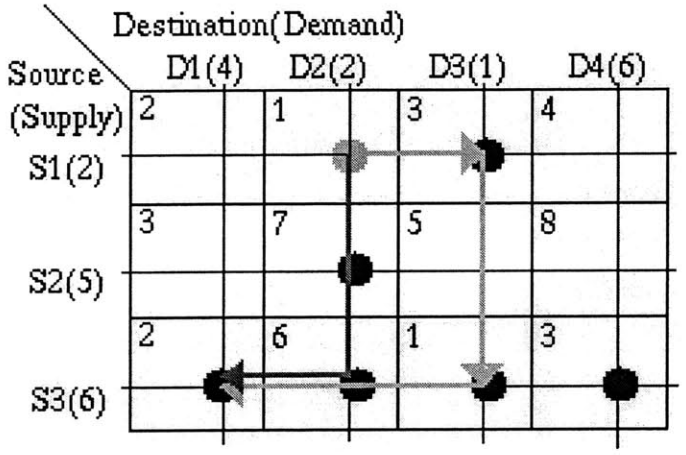

Figure 4-25: A cycle in m+n cells

Thus, a set of $m + n$ cells in a cost table must form a cycle.

The next several proofs in this section will lead to a proof that, *given a set of N cells where $N > m + n - 1$, it is always possible to pick $(m + n - 1) - N$ additional cells such that the entire set of cells is acyclic.*

Theorem 3 *Given a set of N cells where $N > m+n-1$, it is always possible to pick $(m+n-1)-N$ additional cells such that the entire set of cells is acyclic.*

Lemma 1 *Two distinct trees of basic cells can be joined together to make a single (acyclic) tree composed entirely of basic cells by marking a single additional cell as basic.*

Since the two trees, A and B, are distinct, there are no paths in the cost table that include only basic cells and that include cells in both trees. A cell c in the same row as a cell from one of the trees and in the same column as a cell in the second tree can be marked as basic to create a single tree. Figure 4-26 shows two distinct trees being joined by a single cell. Tree A is composed of the three cells in the upper right of the cost table, while tree B is made up of the cells in the lower left of the cost table. Cell c is (S2, D2), and marking it as basic will join the two trees together along the dashed lines.

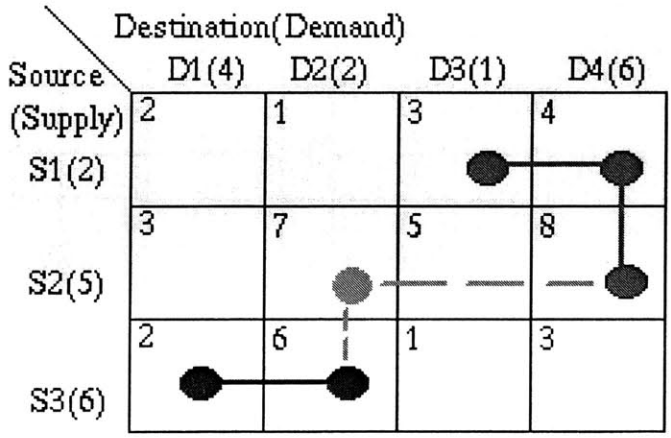

Figure 4-26: Two trees joined by a single cell

Since there were no paths made up entirely of basic cells that included cells from both trees before the trees were joined, any path made up of basic cells that includes cells from both A and B must include cell c. Thus, marking cell c as basic can only create a cycle if marking cell c as basic creates a cycle within one of the two original trees.

For example, if cell c were chosen to be (S2, D3) as in figure 4-27 (ignoring the fact that cell (S2, D3) does not join the two trees together), a cycle would be formed by the cells four cells in the upper right hand corner.

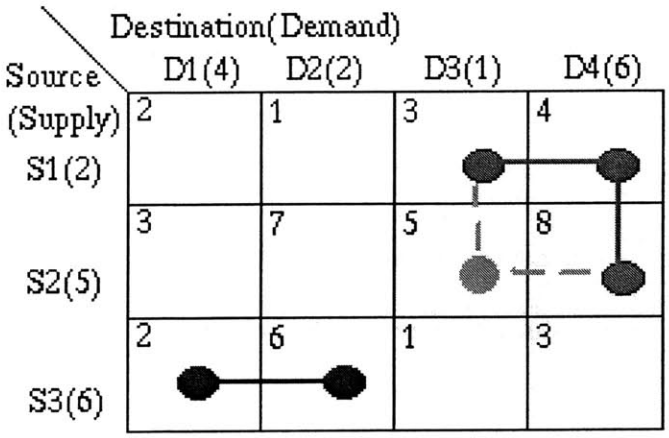

Figure 4-27: A cycle formed within a tree

Cell c was chosen because it is in the same row as a cell in tree A and is in the same column as a cell in tree B. Since there was no path made up of basic cells from the cells in tree A to the cells in tree B before c was marked basic, there can not be a cell in tree B in the same row as cell c and there can not be a cell in tree A in the same column as cell c.

For example, in figure 4-26 cell (S2, D1) can not be a member of the tree in the lower left because if it were, there would be a path connecting the two trees from (S2, D1) to (S2, D4). Since the trees are distinct, there is no such path.

Thus, marking cell c as basic can not create a cycle among the cells in either of the two original trees. Thus, any two trees can be joined into a single (acyclic) tree by marking a single cell as basic.

Lemma 2 *A acyclic, spanning (i.e.: at least one cell in every row and column) set of* $m + n - 1$ *cells must form a single tree.*

If it were possible for $m + n - 1$ cells to form more than one tree, a cell could be added that would join the two trees together. This would create a spanning tree composed of $m+n$ cells. Since $m + n$ cells always form a cycle, as demonstrated above, this is impossible. Since the $m + n - 1$ are chosen because they are spanning and acyclic, they must therefore form a single spanning tree.

Lemma 3 *A single spanning tree cannot be composed of fewer than* $m + n - 1$ *cells.*

In order for there to be a single spanning tree there must be a basic cell in every row and every column. Since the cells form a single tree, one of the basic cells can be paired up with both its row and column and every other basic cell can then be matched up with either its row or its column in the same way that cells were matched up in the proof that $m + n$ cells must form a cycle (see above). Since there is a path from every cell in the tree to every other cell in the tree, every cell will be matched with a single dimension except for the first cell, which will be matched with two dimensions. Since there are $m+n$ rows and columns, there must be $m+n-1$ cells in the spanning tree.

Theorem 4 *The stepping stone algorithm will always choose* $m + n - 1$ *basic cells that form a single spanning tree.*

Since it is impossible to create a single spanning tree with fewer than $m + n - 1$ cells, it is impossible for the greedy algorithm to assign values to a set of $m + n - 2$ or fewer cells such that

all of the supply and demand constraints are satisfied and a single tree is formed from the chosen cells.

Since it is always possible to join two distinct trees together to form a single tree without forming a cycle, it is always possible for the stepping stone algorithm to ensure that there are $m + n - 1$ basic cells without forming a cycle, regardless of which cells the initial greedy allocation chooses to label basic. The stepping stone algorithm can do this by simply choosing cells that join the various trees together and labeling them as basic.

Theorem 5 *A set of $m + n - 1$ basic, connected, spanning cells must be acyclic.*

If a cycle exists among the basic cells, it can be reduced as above, in figures 4-19 and 4-20. The basic cells can be matched up with their rows and/or columns in the same way that cells were matched up in section 2, beginning with one of the cells that remains in the cycle after the reduction. All of the basic cells (including the ones dropped from the cycle) are included in this matching process. Since the set of cells is connected, every cell will eventually be reached by the matching process. Since there are $m + n - 1$ cells and $m + n$ dimensions and the cells are spanning, each cell will be matched up with one of its dimensions, except for the first cell, which will be matched up with both of its dimensions. The reduced cycle must include the cell that was matched up with both of its dimensions, as well as several cells that were matched up with only a single dimension. Two paths can thus be "traced" outward from the original cell as was done in section 2. However, since every cell other than the original cell was matched to at least one dimension, the two paths can never meet in a single cell. Thus, there can not be a cycle.

Theorem 6 *The process of labeling a cell as basic and labeling another cell as non-basic in the stepping stone algorithm never forms a cycle of basic cells and never creates a set of basic cells that is non-spanning.*

Since the cell that was labeled as non-basic, X, was part of a cycle, one of the basic cells (including the cell just labeled as basic) must be on the same row as X and another must be on the same column as X. Thus, the set of basic cells must still be spanning. Additionally, any basic cells that were on either the same row or the same column as X must still be on the same row or same column as another basic cell. Thus, the set of basic cells must still be connected.

Since the set of $m+n-1$ basic cells is spanning and connected, it must also be acyclic as proved above.

4.2.2 Improvement of the solution

Each improvement of the solution requires first calculating the dual cost of each of the rows and then calculating the reduced cost of each of the cells in the table. This portion of the proof demonstrates that any cell with a negative reduced cost is part of a cycle around which the values of the cells can be adjusted to improve the cost of the solution as a whole.

Theorem 7 *Any cell with a negative reduced cost is part of a cycle around which the values of cells can be adjusted to improve the overall solution.*

Construct a graph G from the cost table such that each "supply" constraint in the transportation problem corresponds to a node, each "demand" constraint corresponds to a node, each cell in the table corresponds to an arc between a "supply" node and a "destination" node, and arcs connect a source node to the supply nodes and a sink node to the destination nodes with capacities corresponding to the supply and demand constraints. Figure 4-28 demonstrates the network flow problem derived from a cost table with four rows and four columns. For example, the arc leading from the source node to the node labeled S_1 would have a capacity equal to the S_1 supply constraint. The arc leading from S_1 to D_1 would have a cost equal to the cost of moving an agent from S_1 to

D_1.

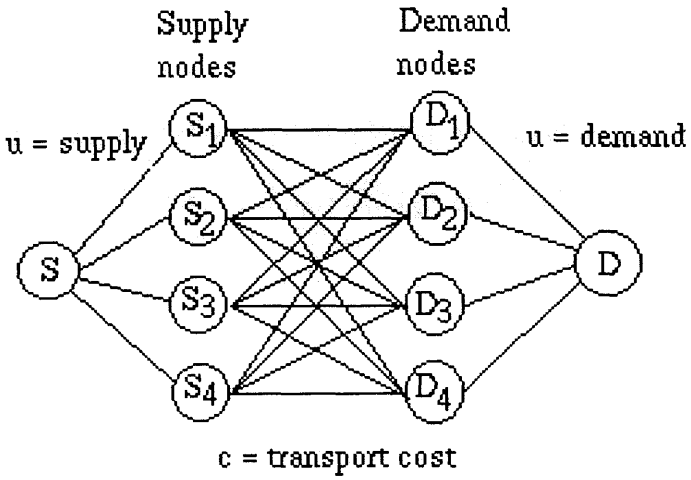

Figure 4-28: A network flow problem derived from a cost table

Thus, any solution to the original transportation problem is also a feasible flow in G. If the costs of all of the arcs from the source node to the supply nodess are negative numbers with large magnitudes, the optimal solution to the transportation problem is the optimal solution to the corresponding min cost flow problem in G. In this graph, the dual cost of each row and column corresponds to the node potential of the appropriate node in the graph and the reduced cost of the nodes in the table correspond to the reduced cost of the arcs in G. According to the complementary slackness optimality conditions outlined in *Network Flows* [1]:

> A feasible solution x^* is an optimal solution of a min cost flow problem if and only if for some set of node potentials the reduced cost and flow values satisfy the following complementary slackness optimality conditions for every arc $(i, j) \in G$:

$$\text{if } c_{ij}^{\pi} > 0, \text{ then } x_{ij}^* = 0 \tag{4.3}$$

$$\text{if } 0 < x_{ij}^* < u_{ij}, \text{ then } c_{ij}^{\pi} = 0 \tag{4.4}$$

$$\text{if } c_{ij}^{\pi} < 0, \text{ then } x_{ij}^* = u_{ij} \tag{4.5}$$

where c_{ij}^{π} is the reduced cost of arc *(i, j)*, u_{ij} is the capacity of the arc *(i, j)* and x_{ij}^{*} is the flow through a particular arc.

This theorem can be mapped back to the original transportation problem as follows:

If each node in the graph is assigned an arbitrary potential, the solution to the minimum cost flow problem is optimal if and only if the reduced costs of the arcs follow the properties outlined above. The reduced cost for an arc is simply the change in the potential between the two nodes that it connects.

Each of the supply and demand cells corresponds to a row or column in the original cost table; the arcs that connect supply cells to demand cells correspond to cells in the original table. The dual cost for each node corresponds to the dual cost of the associated supply or demand constraint and the dual costs of the arcs correspond to the reduced costs for each cell in the cost table. The reduced costs for the basic cells are always zero since the dual costs for the rows and columns are chosen to ensure this. The flow through all of the non-basic cells is zero, so every cell with a positive reduced cost has zero flow. This means that in a non-optimal solution, some cell must have a negative reduced cost and flow below its capacity (which is infinite). This implies that more agents can be sent through this cell in order to improve the solution and that as a result, the arc can be made basic.

4.2.3 Termination

Theorem 8 *The stepping stone algorithm always terminates*

Since the stepping stone algorithm is initialized using only integers and is solved using only mathematical operations that are closed for the set of integers, the cost of any intermediate solution must always be integral. Thus, any improvement to the initial solution must decrease its cost by at least one.

The stepping stone algorithm always generates an initial solution. Since the optimal solution has a specific cost, the initial solution has a specific cost and each iteration improves the cost of the initial solution by at least one, the algorithm must terminate.

4.2.4 Optimality conditions

Theorem 9 *The stepping stone algorithm terminates with an optimal solution.*

A feasible solution to the transportation problem is optimal if and only if there are no negative cost cycles in the cost table whos flow is greater than zero. If such a cycle exists then the solution can be improved by moving as much flow as possible around this cycle, and thus the solution is not optimal.

If there are no such cycles then the solution must be optimal for the following reasons. Let x^* be a feasible solution that has no negative cost cycles and let x^o be an optimal solution to the same problem such that $x^* \neq x^o$. Since x^o is optimal, $c(x^*) \geq c(x^o)$ where $c(x)$ is the cost of the solution x.

Consider the graph G created from the cost table in the previous section of the proof. The Augmenting Cycle theorem [1] states that any two solutions to a network flow problem x^1 and x^2 can be decomposed such that x^1 equals x^2 plus the flow on at most m directed cycles where m is the number of edges. Since x^o and x^* are solutions to the original transportation problem, they are also feasible flows in G. Thus, x^o can be decomposed into x^* plus the flow around some set of cycles in G. It follows from this that x^o can be represented as x^* plus some "flow" around a set of cycles in the original cost table. Since there are no negative-cost cycles in x^*, this implies that $c(x^o) \geq c(x^*)$ so the two flows must have the same cost and x^* is another optimal solution.

4.3 Run time analysis of the stepping stone algorithm

Although many algorithms exist which can solve the transportation problem and which have a better worst case running time than the stepping stone algorithm, the stepping stone algorithm performs very well in practice due to the low overhead required and the fact that most problems require very few iterations to solve.

The original reduction of the current assignment and a future assignment to a set of "supply" and "demand" values takes $O(N)$ time where N is the number of fires in the city. This results in m supply cells and n demand cells. Both m and n are less than N. Generating the cost table, the supply column and the demand row takes $O(mn)$ time. Generating the initial greedy solution can be done in two ways. The first is to sort all of the cells by cost and then assign the first $O(m+n)$ of them to be basic cells. This takes $O(mn\ log(mn))$ time, but involves a fair amount of overhead for the sort. Another way to do it which I found to be faster for problems involving small numbers of fires is to simply iterate through the list $O(m+n)$ times and pick out the smallest one each time. This requires $O(mn(m+n))$ time, but is very fast in practice. All of these time costs are associated with initializing the algorithm and are only performed once.

Once everything has been initialized, the algorithm iterates until the solution is complete. Each iteration requires the following steps:

1. Calculating the dual costs for the rows and columns takes $O(m^2n^2)$ time since you have to search through the basic cells to find one that has only one of either its row's or column's dual cost set.

2. Calculating the reduced cost for the non-basic cells requires $O(mn)$ time, since that's how many non-basic cells there are.

3. Finding a negative cost cell requires $O(mn)$ time.

4. Finding the cycle composed of basic cells and a negative cost cell simply requires a depth first search which takes $O(m+n)$ time.

5. Adjusting the cycle by the minimum flow found requires $O(m+n)$ time since that is the length of the cycle.

In practice the stepping stone algorithm typically requires very few iterations to terminate, so the average case running time is OCm^2n^2 where C is the number of iterations. Although the worse case number of iterations is the number of integral cost values between the starting and optimal costs, in practice the stepping stone algorithm typically requires only two or three iterations to reach an optimal solution.

Chapter 5

Agent Decision Algorithms

The purpose of this research is to examine the issues surrounding loosely coupled, distributed collaboration and to compare it to collaboration in which a central agent coordinates the actions of all of the other agents. In order to model the best possible behavior for the central coordinating agent, the fire-fighting problem is cast as a Markov decision problem. Solving the Markov decision problem will produce a policy mapping every possible state of the world to the best possible action in that state. The central coordinating agent simply needs to "look up" the action corresponding to the current world state. The state of world is represented by the states of the various fires and the locations of the agents, and the actions in each state are represented by possible assignments of fire fighters to fires. For example, one action for a given state involving two fires might be to assign two fire fighters to fire one and one fire fighter to fire two. These Markov decision problems are solved using the value iteration method [14]. Appendix A provides a brief overview of Markov Decision problems and the value iteration algorithm.

Four scenarios are examined in this thesis, and algorithms to handle each situation are discussed below. The following scenarios were chosen to test the effects of various factors on agent performance. The first scenario allows the agents both global knowledge and perfect communica-

tions with a single coordinator. This represents the best case scenario for the agents. The second scenario also allows the agents global knowledge of the city, but requires each agent to make its own decisions without central coordination. The third scenario limits each agent's knowledge to areas within its line of sight and requires that each agent make its own decisions. The final scenario also includes limited knowledge and communications and, in addition greatly reduces the speed at which agents move around the city.

Table 5.1 lists the average damage done to the city over 2000 trials of each scenario. Each problem was run with ten fires and fifteen agents.

	global knowledge	local knowledge
Central coord.	197.455	XXX
Distrib. coord.	216.326	294.939
Reduced speed	XXX	1047.152

Table 5.1: Damage caused to the city during each of the four scenarios

For the scenario in which a central coordinating agent is present, the agents are allowed unlimited communication with a single entity that makes all of the decisions. In situations that allow only local knowledge, any two agents are only allowed to communicate with each other when they are within a certain distance of each other and there is an unobstructed line of sight between them. The five algorithms used in this study are described next.

1. Optimal algorithm - The optimal algorithm is a solution to the MDP used to model the world. The solutions generated by this algorithm will be used as a benchmark for optimality in the rest of the study. This algorithm requires a central agent with global knowledge and global control over the fire fighters but, due to computational restraints, can only be applied to fairly small problems.

2. Approximation algorithm - The approximation algorithm is an estimation of the optimal

solution to the MDP. This algorithm can be applied to problems that are too large to solve via the optimal algorithm and was used as a benchmark to measure later algorithms on larger problems. The results of the approximation algorithm were compared with optimal solutions to a series of small problems and found to be nearly optimal in most cases. The approximation and optimal algorithms are both intended for the same type of problem; both algorithms require global knowledge and centralized control over the agents.

3. Decoupled algorithm - The decoupled algorithm deals with a different situation than the previous two algorithms. Agents are still provided with global knowledge but now must make all decisions locally. For small problems, the results of this algorithm were compared with the results of the optimal algorithm; for larger problems the results of the approximation algorithm were used as the benchmark.

4. Fully collaborative algorithm - The fully collaborative algorithm deals with situations in which neither global knowledge nor centralized control is present; agents must communicate with each other to share knowledge and must make all decisions locally. The results of this algorithm were compared with the results given by the decoupled algorithm to determine the effects of global knowledge versus local knowledge. Additionally, the results were compared with those of the optimal and approximation algorithms to determine the optimality of the algorithm.

5. Constrained collaborative algorithm - The constrained collaborative algorithm deals with a scenario in which additional constraints on agent travel are imposed. These constraints increase the state space to the point where finding an optimal solution is computationally infeasible even for very small problems. The results of this algorithm were compared with the results of the fully collaborative algorithm to determine the effect of these additional

constraints.

The decoupled, fully collaborative and constrained collaborative algorithms all rely on a Boltzman distribution to guide the actions of the individual agents. These distributions are controlled by parameters as described in section 5.3. The values of these parameters were tuned over repeated trials by a simple machine learning algorithm implementing gradient descent. The machine learning algorithm is discussed in further detail in section 5.3.

Agents in the various scenarios used different mechanisms to travel around the city. In the optimal, approximation and decoupled algorithms, the agents are allowed to travel instantly from one fire to another, although "teleporting" around the city may have a cost associated with it. Since the agents have global knowledge, there is no need for them to travel anywhere else in the city. In the fully collaborative algorithm, the agents can still teleport around the entire city but no longer have perfect knowledge of their surroundings. At times they are forced to teleport to randoms locations in order to discover new fires. In the constrained collaborative algorithm, agents are no longer allowed to "teleport" around the map. Instead, each agent has a set walking speed that it used as it followed the city streets.

Table 5.2 summarizes the type of problem that each of the algorithms is designed to solve; more detail is given in the following sections where each algorithm is described.

Characteristics	Algorithms				
	optimal	approx.	decoupled	fully collab.	constrained collab.
optimal	Yes	No	No	No	No
requires global knowledge	Yes	Yes	Yes	No	No
requires global control	Yes	Yes	No	No	No
scalable	No	Yes	Yes	Yes	Yes
allows "teleportation"	Yes	Yes	Yes	Yes	No
uses machine learning	No	No	Yes	Yes	Yes

Table 5.2: Algorithms to be Compared

5.1 Optimal algorithm

The optimal algorithm is designed to solve problems in which both global knowledge and centralized organization are present. This version of the problem relies on a single entity that knows the state of the world and can communicate perfectly with each of the fire fighters. The goal of this problem is to find the optimal behavior for the central coordinator; the orders that the central coordinator gives to an agent provide the agent with the optimum behavior that the agent can perform at that time.

This problem is framed as a Markov decision problem (MDP) designed to generate the behavior of the coordinating entity. The MDPs are solved through the value iteration method described in Appendix A.

Two versions of the optimal algorithm were run on problems of various sizes to test the algorithm's scalability. Simulations that took over a day to run were deemed computationally intractable and were halted. In the "cost free" version, agents were allowed to teleport around the map at no cost. The largest problem of this type that the computer could handle in a single day involved four fires and seven fire fighters. This MDP required a total of 625 states with 120 actions per state and took about four hours to solve on a PII 300 MHz computer. Larger problems proved to be computationally intractable; it took over a day to do five iterations of the value iteration algorithm for a five fire, eight agent problem. Since the algorithm typically takes between fifty and seventy iterations to converge the calculation was halted.

A second version of the algorithm, in which agents were penalized for teleporting large distances, was also tested. This "travel cost" version of the optimal algorithm could solve problems involving three fires and five agents in around 10 minutes on the above mentioned computer. However, a four fire, five agent problem required about 48 hours to solve.

The cost-free version of the problem has one important trait: the effectiveness of a particular assignment of fire fighters to fires depends only on the state of the fires, not on the current locations of the agents. The effectiveness of an assignment in the travel-cost version of the problem relies on both the current states of the fires and the current locations of the various agents. For example, consider a city with two fires, A and B, and a single fire fighter F. The effectiveness of assigning F to fire A is independent of F's current location in the cost free problem, but depends on F's location in the travel cost problem. Suppose having F fight fire A is more useful than having a single agent fight fire B. There is no reason not to assign F to fire A in the cost free problem. However, if agent F is already at fire B, the cost of traveling between the two fires may overshadow the gain of having F fight fire A instead of fire B.

The fact that the utility of an assignment of fire fighters to fires is independent of the agents' locations means that the state of the world in the cost-free problem can be represented by the states of the various fires alone. In the travel cost problem, the state of the world must reflect both the states of the fires and the agents' locations. As discussed above, this reduces the size of the largest feasible problem solvable by the value iteration algorithm.

The value-iteration equation [14] used in the cost free version of the problem computes the value of a given state as the sum of the instantaneous reward for being in the state and the expected value of the best possible action in that state. That is:

$$V(f) = R(f) + (d \max_{p} \sum_{f'} (\Pr(f'|f,p)V(f'))) \tag{5.1}$$

where f is an n-tuple representing the current state of the various fires, p is an n-tuple representing a possible assignment of fire fighters to fires, $R(f)$ is the instantaneous reward for having fires in configuration f and f' represents the possible future configurations of the fires. d is the "discount

factor", in this research it has a value of 0.9. The discount factor alters the solution slightly; it causes the value iteration algorithm to more heavily favor courses of action that will lead to short term gains than it would otherwise have. A useful side effect of discounting is that it also causes the problem to converge more quickly to an equilibrium point.

The term $Pr(f'|f,p)$ is the probability of reaching some particular fire configuration f' from a fire configuration f by performing the assignment p, while $V(f')$ is the value of being in configuration f'. The reward function for being in a given state is simply a measure of how much damage the city will take over the next time step, multiplied by -1. For example, a city with two fires of intensity 1 and one fire of intensity 2 would have a reward of -4. The rewards are multiplied by -1 so that the rewards for a "better" situation is greater than the reward for a "worse" situation.

In the travel-cost version of the problem, each state is composed of a pair of n-tuples. One n-tuple describes the fire states and the other describes the agents' current locations. The equation used to determine the value of a given state is:

$$V(< f,p >) = R(f) + \max_{p'}[C(p,p') + d\sum_{f'} \Pr(f'|f,p')V(< f',p' >)] \tag{5.2}$$

where p describes the current location of the agents, and p' describes a possible assignment of agents to locations. $C(p,p')$ is the minimum cost of moving the agents from assignment p to assignment p' as determined by the Stepping Stone algorithm (see section 4.1).

5.1.1 Collapsing the Agent Location Information

The typical value iteration approach requires iterating over and updating every possible state once, and then repeating the entire process over and over again until the values of the states converge. Since the values of the states asymptotically approach some equilibrium point, the

algorithm ends when the the new value for every state is within some epsilon (ϵ) of the old value. In this research the value of ϵ was set at 0.01. As a basis for comparison, values for states in a three fire problem range from -659.03 to 0.0 and the value difference between the two states closest in value is 1.03. Thus, the value of ϵ was small enough that a change in value by less than ϵ was judged to be insignifigant. Pseudocode for this algorithm is shown in Figure 5-1.

The *odds(x, y, z)* function found in line 18 calculates the probability of reaching the fire configuration x given the current fire configuration y and the agent distribution z. The *reward(x)* function calculates the instantaneous reward for being in state x.

This value iteration algorithm listed in Figure 5-1 runs in $O(|S|^2|A|)$, where $|S|$ is the number of possible states that the MDP might be in and $|A|$ is the number of possible actions in each state.

Since each world state in a travel-cost problem consists of a combination of the states of the fires and the locations of the agents, the number of possible states in the world grows much faster than it does in a similar cost-free problem. For example, a city with 2 fires and 4 agents has 125 possible states for the travel cost problem but only 25 states for the cost-free problem. Similarly, a city with 3 fires and 5 agents has 2625 possible states for the travel cost problem, but only 125 states for the cost-free version.

In the cost-free problem, $|S|is|F|$, the number of possible fire configurations in the city. However, in the travel-cost problem $|S|is|F||P|$, the number of possible fire configurations multiplied by the number of possible assignments of agents to fires.

Thus, solving the cost-free MDP using equation 5.1 requires $O(|F|^2|P|)$ time whereas solving the travel-cost problem using equation 5.1 requires $O(|F|^2|P|^3)$ time. The time required to solve the travel-cost problem can be shortened to $O(|F|^2|P|^2)$ by observing that any particular action will result in only $|F|$ of the $|F||P|$ possible states. For example, if an action assigns three fire fighters to fire A and none to fire B, then every state in which an agent is located at fire B has zero

```
// takes as input an array "states," which includes the states of the world,
// an array "actions," which includes the possible actions in each state and
// the array "fires", which includes a list of all of the possible fires.

// this algorithm modifies the "act" field in each element of the states array
// to reflect the best possible action in that state.

// initialize all values to negative infinity
for i = 0 to length[states]
   do states[i].value = negative_infinity                                    10
      states[i].old_value = negative_infinity

notDone = true
while notDone
   // copy values to old_values for later use
   do for i = 0 to length[states]
         do states[i].old_value = states[i].value

      // create new values
      for i = 0 to length[states]                                            20
         do for j = 0 to length[actions]
               do sum = cost(states[i], actions[j])
                  for k = 0 to length[fires]
                     do sum = sum + d * (<fires[k], actions[j]>.value * odds(fires[k], states[i].fires, actions[j]))

                     // the odds function calculates the odds of reaching the fire configuration
                     // fires[k] given the current fire configuration states[i].fires and the
                     // action actions[j]

                  if sum > states[i].value + reward(states[i])             30
                     then states[i].act = actions[j]
                          states[i].value = sum + reward(states[i])

      // check if values have converged
      notDone = false
      for i = 0 to length[states]
         do if |states[i].old_value - states[i].value| > epsilon
               then notDone = true
                    break
                                                                           40
```

Figure 5-1: Pseudocode for the typical value iteration method

probability of happening.

Despite this optimization, the value iteration algorithm is still too slow to solve problems involving more than two or three fires or fire fighters. However, it is possible to further optimize the value iteration algorithm and solve travel-cost problems in $O(|F|^2|P| + |F||P|^2)$ time by precalculating some of the information used in the MDP and storing it for use throughout the value iteration method. It is possible to speed up the value iteration calculation by noticing that the innermost loop, beginning on line 15 of Figure 5-1, does not rely on the "action" part of *state[i]* (from line 12). This means that when various states have the same fire configuration but different agent distributions, the inner loop will always return the same result. For example, the two following states:

$$< (2,2)(0,2) >$$

$$< (2,2)(1,1) >$$

would both result in the same value being calculated in the inner-most loop. By pre-computing all of the possible values for the inner-most loop, storing them in a hash table and then looking them up as needed, the running time for the value iteration algorithm can be reduced to $O(|F|^2|P|+|F||P|^2)$ time. Pseudocode for the new value iteration method can be found in figure 5-2.

In figure 5-2, *gdp* is a hashtable used to store the calculations. Everything else follows the notation for figure 5-1 as listed above.

Despite this improvement, the travel cost version of the problem is still significantly slower to solve than the cost-free version of the problem. Solving a 4-fire, 7-agent cost-free problem on an Ultra Sparc 20 with 300 MHz processor required about an hour of computation; solving a travel cost problem of the same size on the same computer required approximately 48 hours.

```
// takes as input an array "states," which includes the states of the world,
// an array "actions," which includes the possible actions in each state and
// the array "fires", which includes a list of all of the possible fires.

// this algorithm modifies the "act" field in each element of the states array
// to reflect the best possible action in that state.

// initialize all values to negative infinity
for i = 0 to length[states]
   do states[i].value = negative_infinity                                    10
      states[i].old_value = negative_infinity

notDone = true
while notDone
   // copy values to old_values for later use
   do for i = 0 to length[states]
         do states[i].old_value = states[i].value

      // calculate inner loop for all f, p' and store in hashtable gdp
      // the hashtable will be read later to determine the value for        20
      // the inner loop calculations

      for i = 0 to length[fires]
         do for j = 0 to length[actions]
               do sum = 0
                  for k = 0 to length[fires]
                     do sum = sum + (odds(fires[k], fires[i], actions[j]) * <fires[k], actions[j]>.value)

                     // this loop calculates the expected value of each action. This is
                     // simply the odds that a given state will occur as a result of the   30
                     // action, multiplied by the value of being in that state.

                  gdp.store(<fires[i], actions[j]>, sum)

      // create new values
      for i = 0 to length[actions]
         do for j = 0 to length[fires]
               do for k = 0 to length[actions]
                     do sum = cost(actions[i], actions[k]) +
                              (d * gdp.get(fires[j], actions[k]))              40

                        if sum > states[i].value + reward(states[i])
                           then states[i].act = actions[j]
                                states[i].value = sum + reward(states[i])

      // check if values have converged
      notDone = false
      for i = 0 to length[states]
         do if |states[i].old_value - states[i].value| > epsilon
               then notDone = true                                           50
                    break
```

Figure 5-2: Pseudocode for the improved value iteration method

5.2 Approximation algorithm

This algorithm is designed to deal with the same types of problems as the optimal algorithm, but it can handle much larger problems. Many algorithms have been suggested for finding near optimal solutions to large Markov decision problem. This research adapts a technique laid out by Meuleau *et al.* for solving large, weakly coupled MDPs [16]. Although this technique does not provide optimal solutions, it generates near-optimal results for most problems and can be used on problems involving dozens of agents and fires.

The approximation algorithm takes advantage of the fact that the fires act independently of each other; the growth of the fires in one part of the city does not affect the fires anywhere else in the city. The approximation algorithm calculates the utility of assigning a set number of fire fighters to a fire under the assumption that fire fighters assigned to a fire will remain there until it is extinguished. Given the utilities of assigning various numbers of fire fighters to various fires, the problem is reduced to finding an instantaneous assignment of fire fighters to fires in such a way as to maximize the total utility.

Iterating through every possible combination of utilities and simply choosing the right one would require $O(F^A)$ time, where F is the number of fires and A is the number of agents. However, it is possible to reduce the run time of the approximation to $O(F + A)$ by exploiting one of the characteristics of the fire model used in the simulator.

The fire model used in this research is designed so that adding additional agents to a fire has a decreasing effect. For example, two fire fighters have less than twice the chance of lowering a fire's intensity as one fire fighter. This characteristic of the Hybrid Model means that a "critical mass" of agents is never needed for effective fire fighting and that the utility gained by adding an additional fire fighter to a fire always decreases as the number of fire fighters increases. At each time step the

agents are allocated among the fires in a greedy fashion based on the utility gained per additional fire fighter. The algorithm compares the utility of a given fire with its current allocation of fire fighters to the utility of the same fire with one additional fire fighter. The fire with the largest difference is assigned an additional fire fighter, reevaluated, and again compared to the other fires. The fire with the largest difference (this may or may not be the same fire as the previous iteration) is then assigned a single agent. This continues until all of the fire fighters are assigned.

The approximation algorithm precomputes the utilities of assigning varying numbers of fire fighters to fires of different intensities, but, unlike the optimal algorithm, does not compute a complete solution for every possible occurence. Pre-computing the individual utilities requires $O(fA)$ time, where f is the number of possible states a single fire can be in and A is the total number of fire fighters in the simulation. Evaluating any given situation takes $O(m+n)$ time where m is the number of fires and n is the number of agents.

Travel-cost problems, in which distance costs are taken into account, use a greedy approach to judge the utility of adding agents to various fires. In addition to calculating the utility of having an additional agent at each fire, the cost of moving the nearest unassigned agent to the fire is taken into account. Although this means that agents are moved from one fire to another in a greedy, non-optimal fashion it was too computationally intensive to iterate over all of the possible assignments of agents to fires, calculate the utility and the smallest possible travel cost for creating each assignment and then choose an assignment out of this set. Since the fires didn't typically change much between any two time steps, very few agents were required to move at any given time. Thus, this greedy allocation generally worked very well. (See results in chapter 6.) However, the travel-cost version of this algorithm requires $O(mn)$ time since the algorithm has to compute the distance between every agent and every fire. Although calculating the distance between and agent and a fire is very simple, the large number of these calculations required is the limiting factor on

the scalability of the algorithm.

5.3 Decoupled algorithm

In this algorithm each agent has perfect knowledge of the entire world but now lacks a single central entity to coordinate its actions. Thus, at every time step each agent must view the world and decide which fire it will fight. In the cost-free versions of the problem each agent has no reason to expect an agent to stay in its current location. Because of this, the agents do not attempt to figure out what other agents will do in a given time step. Instead, each agent rates each of the fires. This rating is based on the difference a single fire fighter would make to the expected intensity of the fire; fires which would be more affected by a single fire fighter are given a higher rating. Since this rating is independent of how far the fire is from the agent, every agent will rate the fires the same way. This means that if every agent chose to fight the fire where it would make the most difference, they would all pick from among the fires tied for the highest rating. Since the utility of adding additional fire fighters to a fire decreases as the number of fire fighters increases, this would lead to a lot of wasted effort that could be better spent distributed over other fires. Instead of simply picking the most highly rated fire, the agents use the fires' ratings construct a modified Boltzman distribution [22]:

$$V(X) = e^{(R(X)/C_1)} + C_2 \tag{5.3}$$

$$Pr(X) = V(X) / \sum_X V(X) \tag{5.4}$$

where X is a fire, $R(X)$ is X's rating, C_1 and C_2 are two constants (see below) and $Pr(X)$ is the probability that the agent will choose to fight fire X. The probabilities $Pr(X)$ of all of the fires sum

up to one; at each time step each agent rates each of the fires, generates $Pr(X)$ for every fire, and picks a single fire to fight according to this distribution.

Figures 5-3, 5-4, 5-5, and 5-6 show the probability distributions for a single agent choosing between fires of intensity 1, 2, 3 and 4. The "linear" distribution shows the odds that the agent will pick each of the fires based on the fire's rating. The Boltzman distributions show the effects on this probability distribution of fitting the ratings to Boltzman distributions with various values of C_1 and C_2. For example, "Boltzman (.5/.3)" indicates a Boltzman distribution with $C_1 = .5$ and $C_2 = .3$.

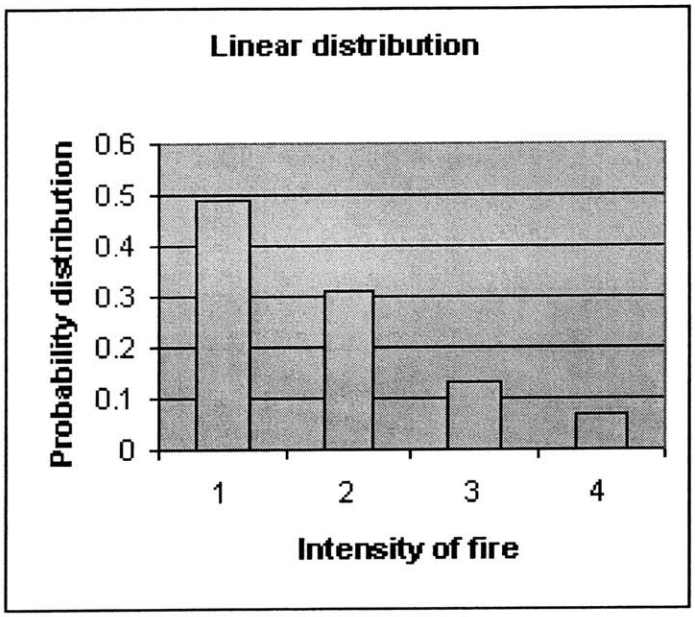

Figure 5-3: A linear probability distribution

A simple gradient descent algorithm was used to tune the values for C_1 and C_2 in order to find the distribution that gave the best results. The gradient descent algorithm evaluated the results for particular values of C_1 and C_2. It then tested the results with a slightly increased value of C_1. If the simulation results improved then the change was kept. If the change worsened the results of the simulation C_1 was tested again with a slightly decreased value. Once C_1 was altered, C_2 was

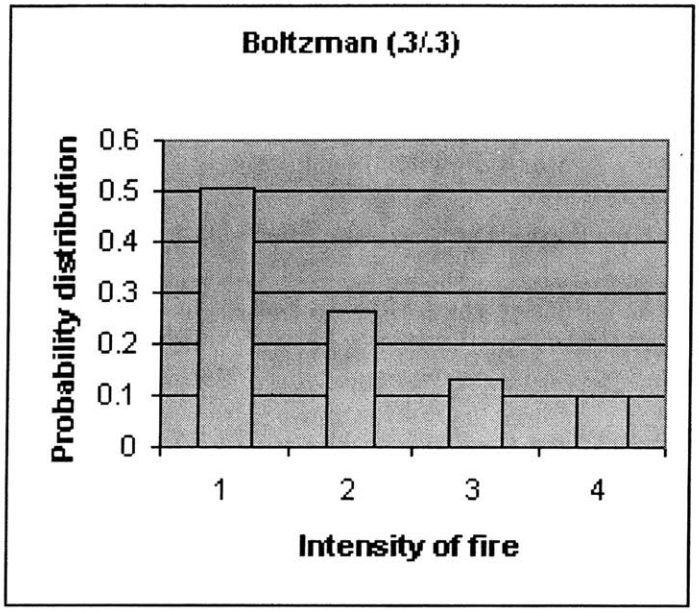

Figure 5-4: A Boltzman distribution with $C_1 = .3$ and $C_2 = .3$

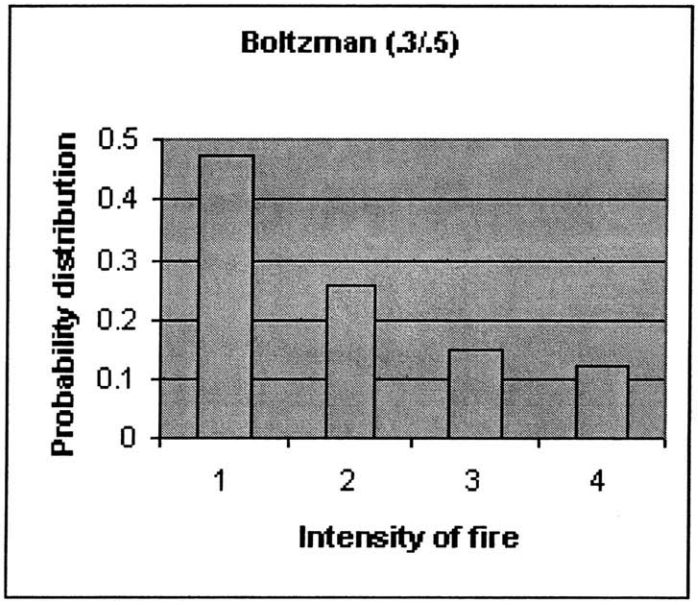

Figure 5-5: A boltzman distribution with $C_1 = .3$ and $C_2 = .5$

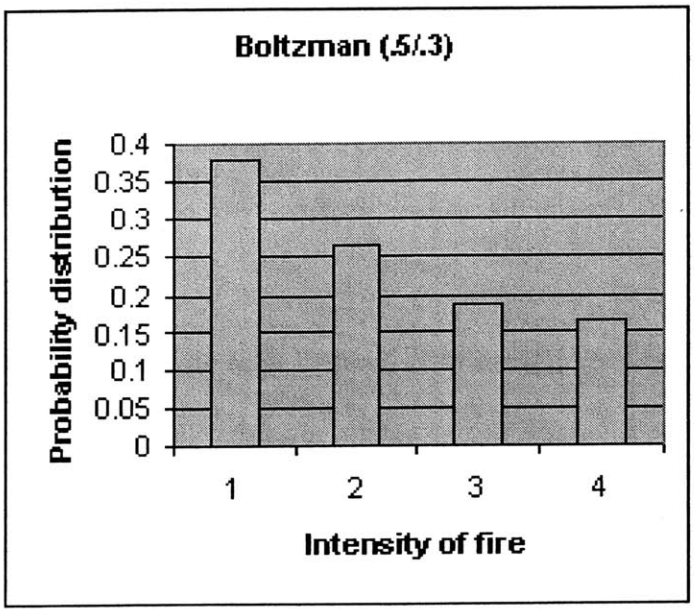

Figure 5-6: A Boltzman distribution with $C_1 = .5$ and $C_2 = .3$

evaluated and altered in the same way and the process was repeated. The incremental changes to C_1 and C_2 began at 10% of their respective values. After C_1 and C_2 were adjusted by this amount 5 times, the size of the incremental change was reduced to 5%. 10 changes of this size were made, followed by 20 changes of 2.5% size. After each set of trials the size of the incremental change was halved and the number of trials in the set was doubled. This continued up until changes of size 0.3125%. The simulation results for any given value of C_1 and C_2 were determined by running 1000 trials and averaging the results. In general, smaller values of C_1 and C_2 tended to produce better results, but reducing the values past a certain threshold cause the agents' effectiveness to suddenly drop off. These experiments, as well as the results, are discussed more thoroughly in the next chapter.

When an agent in a travel-cost version of this problem teleports over a long distance within the city, its ability to fight fires is reduced for the next time step. Agents in a travel-cost problem factor this information into their decisions by estimating the intensity of the fire a time step after

they arrive (and thus after they have paid the cost of teleporting around the city) and using this information in the Boltzman distribution.

Both the travel-cost and cost-free versions of the algorithm take $O(mn)$ time, where m is the number of agents and n is the number of fires, because each agent independently evaluates each of the fires.

Because the agents cannot rely on each other to fight any given fire, compared to a centrally coordinated scheme in which assigning two agents to a fire will always result in both agents fighting that fire, they tend to prefer fighting small fires where an individual agent will have more impact. Thus, the agents in the decoupled algorithm typically do signifigantly worse than those in the optimal or approximation algorithms; they do not work together to fight large fires until all of the small fires have been put out.

5.4 Fully collaborative algorithm

In this scenario, agents are only aware of what they can see of their immediate surroundings and they lack any centralized control. Whenever two or more agents are fighting the same fire, they share their current beliefs about the state of the world and incorporate the information gained from other agents into their individual view of the world.

At any given time the agents are located on the street closest to the fire they are trying to fight. They can "see" up and down the street for several blocks and can also detect fires that are close by but are not within direct line of sight. Agents can also determine the number of agents already fighting any fire they can see. Figure 5-7 shows the area that an agent can see. The thick, red, cross-shaped boundary indicates the area that the agent (represented by the magenta circle at the center of the cross) can see. See chapter 4 for a description of the display.

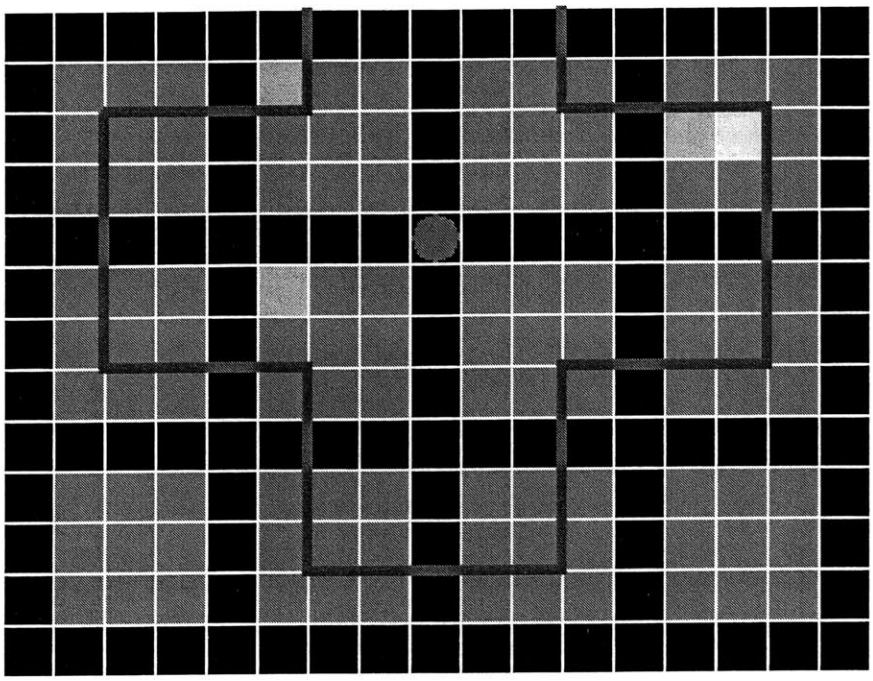

Figure 5-7: Area visible to an agent

Each agent maintains its own "world view," which includes the location and intensity of each fire it knows about and how long ago that information was obtained. When two agents are near each other (that is, they are fighting the same fire), they are assumed to be within communications range of each other and they send each other copies of their current world view. Each agent then uses the most up-to-date observation about each fire to form a new world view and makes its choice about which fire to fight next based on this information. The agents pick which fire they will fight from a Boltzman distribution similar to the one used in the decoupled algorithm but limited to the fires that the agent knows about. The previously described gradient-descent algorithm was used to tune several of the parameters in the Boltzman distribution over many trials.

Since the agents cannot see the entire city, it is possible that the city will contain a fire that none of the agents knows about. In order to help detect and extinguish such fires, the agents will occasionally explore the city rather than fight one of the fires they already know about. At every

time step there is some percentage chance that an agent will choose to explore the city rather than fight a fire. The value of this percentage was also tuned by the machine learning algorithm. When an agent chooses to explore the city, it randomly picks a street in the city and teleports itself there. In the next time step the agent will see any fires that are within its line of sight and will incorporate those fires into its world view. The next time the agent communicates with another agent, it will share this information.

The travel-cost version of this problem incorporates distance costs in the same way that the decoupled algorithm does; the distance an agent teleports affects its ability to fight fires during that time step. Since agents do not fight fires while they are "exploring," distance costs are ignored. For example, agents do not preferentially explore areas closer to themselves even though the travel cost for teleporting to a nearby location is smaller than the cost of traveling a long way. This allows the agents to more easily explore the entire city; having the agents preferentially explore areas close to themselves would prevent distant areas of the city from ever being explored.

As with the decoupled algorithm both versions of the fully collaborative algorithm take $O(mn)$ time because each agent independently evaluates each of the fires.

5.5 Constrained collaborative algorithm

This scenario is similar to the scenario detailed above, except that the agents are unable to teleport around the city. Instead, they travel around the city via the roads and can change their decision about what fire to fight while enroute from location to location. These additions make solving the problem optimally through an MDP nearly impossible. Allowing agents to be located anywhere in the city (instead of simply at a fire) increases the state space of the problem tremendously. Even if the agents had global knowledge of the city (which they do not) solving this problem

optimally would be computationally intractable for all but the most trivial examples. For example, a city comprised of two fires, one agent and a 3 by 3 block of land would require 16 states for a cost-free MDP, 32 states for a travel-cost MDP and 144 states for an MDP capable of solving the constrained collaborative problem.

At every time step the agents gather information about the portion of the city that they can see. The agents use the same "line of sight" code that was used in the fully collaborative algorithm. Agents have full knowledge of the roads in the city; when an agent decides to move from one fire to another it computes the shortest path from its current location to its target. It then follows the city roads along that path to its destination. Agent travel speed was arbitrarily set to allow each agent to move through two cells in a given time step.

Since the agents now spend much of their time traveling from fire to fire, allowing communication only between agents fighting the same fire is unrealistic. Instead, the agents can communicate with other agents within their communication range at any time step. Agents can communicate in straight lines; the agent in figure 5-8 can communicate with any agent in the thick yellow rectangle. The area that the agent can see (the thick, red cross) is also shown in this figure as reference.

Moving from fire to fire requires a considerable investment on the part of the agents and thus happens much less often than it does for the fully collaborative algorithm. Thus, it is much more likely that agents fighting a particular fire will still be fighting the same fire in a few time steps. When an agent sees a fire, it can also see how many people are fighting that fire. It uses this information to help it decide which fire to fight. For example, an agent can see a high intensity fire with several agents around it and a low intensity fire with no agents fighting it. The agent will assume that none of the fire fighters around the high intensity fire will waste several time steps traveling to the low intensity fire and if the distances between the agent and the two fires are comparable, it will head to the low intensity fire. Although doing this results in the agent devoting

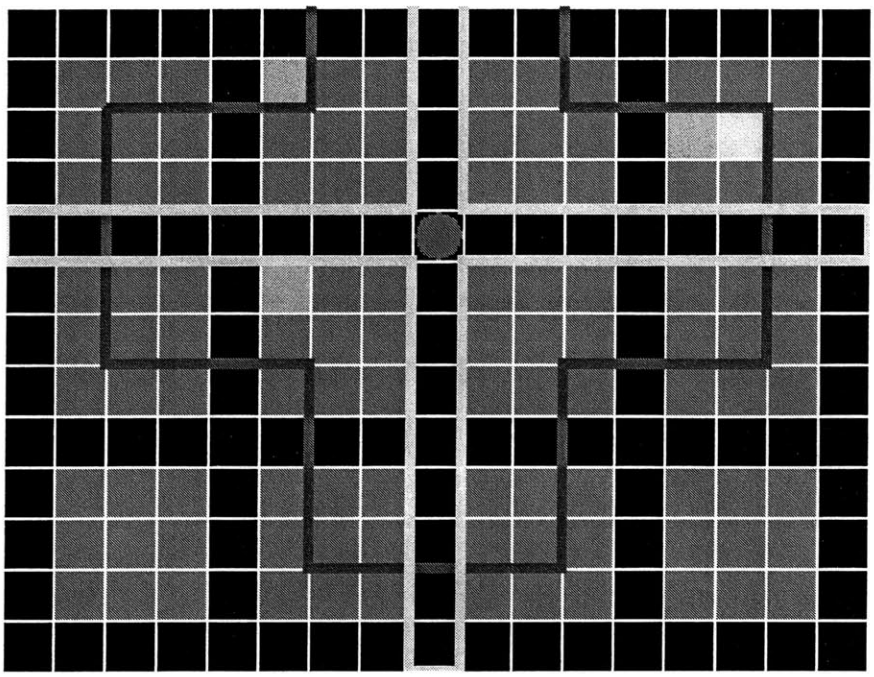

Figure 5-8: Range of communication for an agent

its efforts to a less important fire, fighting a fire with no one around it is more likely to prevent damage to the city because of multiple agents lose effectiveness when fighting the same fire.

In addition to sharing their world views during communication, agents also tell each other their current location, target fire and the number of agents around any fire they can currently see. Agents maintain a world view similar to the one used in the fully collaborative algorithm. The world view does not include information on the number of agents at or heading towards a given fire, other than the information an agent can currently see. Past information about agent locations is not stored in the world view because the agents move around often enough to make this information inaccurate.

For example if two agents, A and B, are heading towards a fire with no fire fighters around it and A is closer to the fire than B, agent A will treat the fire as if there are no fire fighters fighting it. However, agent B treats the fire as though there were already an agent there because it knows that agent A will reach the fire before it does.

Often times an agent enroute to fight a particular fire will see or hear about other fires. This situation is handled the same way agents handle new information at other times; the agent will weigh the utility of continuing on to its original target versus switching to a new target. This calculation is based on the distance between the agent and the various fires, the intensity of the various fires and the number of agents at or heading towards each of the fires. The agent calculates the "expected number" of agents at each of the fires as described above. It then calculates the number of time steps it would take to reach each of the fires and uses this information to determine the expected intensity of the fire assuming the agents currently fighting the fire continue to do so. Finally, the agent calculates the utility of fighting each of the fires given its expected intensity and the same number of fire fighters. The agent will then head towards and attempt to fight the fire with the highest utility.

In summary, at any given time step, regardless of whether an agent is fighting a fire, on its way to a fire, or exploring the city, the agents will access every fire they know about, including fires that they have just seen or just heard about. This assessment attempts to predict the intensity of the fire by taking into account the current intensity, the number of time steps required to reach the fire, and the number of agents already on their way towards the fire. The agent then calculates the utility of fighting each of the fires based on the estimated intensity and the number of agents that will already be at the fire when it arrives. It then heads towards the fire at which it will have the greatest effect.

The four scenarios discussed in this chapter impose various constraints on the agents. In the first scenario the agents can communicate perfectly with a central coordinating mechanism and have perfect knowledge of the city. In the second scenario, the agents retain global knowledge of the city but must now make all decisions locally. In the third scenario, the agents make all of their decisions locally and can only see the portion of the city for which they have a direct line of sight.

In the final scenario, the agents are restricted to local decisions making, local knowledge of the city and are also greatly restricted in their ability to move around the city. The algorithms discussed in this chapter will allow the agents to handle each of these scenarios. However, some of the scenarios are intrinsically more difficult than others; as the number of constraints on the agents increases their ability of quickly extinguish the fires and prevent damage to the city decreases.

Chapter 6

Experiments and Results

The algorithms proposed above were tested in various ways to determine under what conditions the algorithms perform well and under what conditions they perform poorly. The experiments and their results are listed below.

6.1 Comparison of Optimal and Approximation algorithms

Both the optimal and approximation algorithms were run on problems of various sizes to determine the accuracy of the approximation algorithm. The problems are listed as MxN where M is the number of fires and N is the number of agents. Each algorithm was run one thousand times on each problem; the average results for the cost-free problem are shown in table 6.1. A T-Test was used to compare the resulting data sets, the results are also listed in table 6.1.

	3x2	3x5	4x6	4x7
Optimal	84.475	58.383	81.766	76.302
Approx	84.746	58.648	82.956	77.596
T Test	0.8653	0.8432	0.7365	0.6887

Table 6.1: Comparison of optimal and approximation algorithms on cost-free problems

The travel-cost version of the approximation was also run on a series of problems of the same size. The MDP solution to a travel-cost problem is specific to that particular configuration of fires (for example, an MDP that was solved for fires at (1, 1) and (2, 3) would not work for fires at (1, 1) and (10, 10)). Thus, it was not feasible to randomly generate 1000 test cases, and solve the MDP for each of them using the optimal travel-cost algorithm. However, the data shows that on average, the travel-cost version of the approximation algorithm performed signifigantly worse than the cost-free version. This stems from the fact that the algorithm tends to avoid moving agents from one fire to another and because the algorithm uses a simple greedy allocation to move the agents from one fire to another (rather than using the computationally intense Stepping Stone method outlined above). Thus, as the number of agents and fires increases, the agents in the travel-cost problems are less and less effective. Table 6.2 shows these results. A T-Test was also run comparing the results of these experiments with the results of the cost-free version of the approximate algorithm.

As a basis for comparison, the optimal travel-cost algorithm was run on a series of three fire, five agent problems. See section 6.2 for the results.

	3x2	3x5	4x6	4x7
TC Approx	87.758	69.1145	94.33	91.3345
T Test	0.4774	0.2314	0.0424	0.0215

Table 6.2: Effectiveness of approximate travel-cost algorithm

In order to more accurately determine the cases in which the approximation algorithm breaks down, the actions chosen by the approximation algorithm were compared with the actions chosen by the optimal algorithm in various states of the cost-free problem. The approximation algorithm was applied once on every possible state in the four fire, seven agent problem and the action chosen was compared to the MDP solution for the same problem. The approximation algorithm differed from the optimal algorithm in 45 of the 625 states. In all cases, the results differed by at most one

agent. For example, the approximation algorithm might assign three agents to one fire, two agents to another, two to the third fire and none to the fourth while the optimal assignment was three agents to the first fire, two agents to the second and one each to the third and the fourth. Figure 6-1 shows the distribution of states in which the approximation algorithm differed from the optimal algorithm. The intensities of two of the four fires are shown on the X axis while the intensities of the remaining two fires are shown on the Y axis. For example, *44* means that both fires had an intensity of four. The values were sorted along each axis in order of total intensity. For example, *44* is before *34* and *43* which are in turn before *42*, *33* and *24*. The rectangles in figure 6-1 are colored for clarity reasons; rectangles on the same row all share the same color.

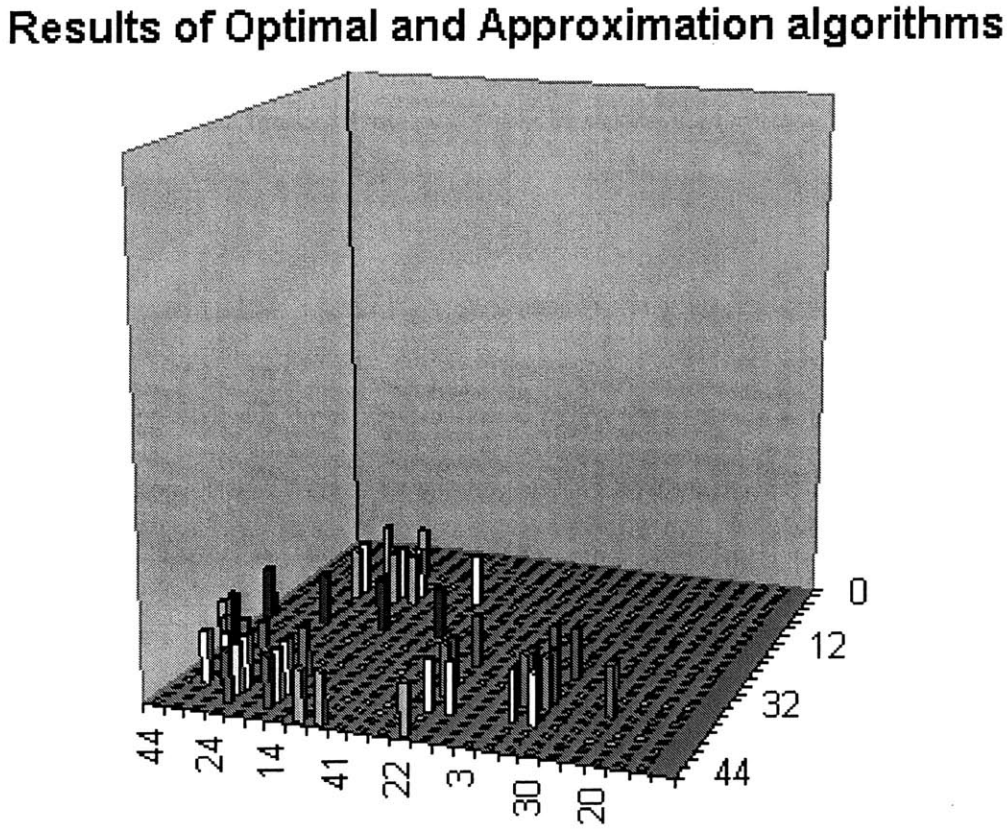

Figure 6-1: Differences between the approximation and optimal algorithms

Figure 6-1 shows that the approximation algorithm differs from the optimal algorithm primarily in cases in which all four of the fires seem to have high intensities. A direct comparison of the choices made by the two algorithms indicates that the approximation algorithm tends to spread its efforts out among several fires rather than concentrating on a single fire as the optimal algorithm does. Table 6.3 shows a representative sampling of the states in the four fire, seven agent problem in which the two algorithms chose different actions. The states are represented as a four-tuple denoting the intensities of the fires while the actions are represented by a four-tuple indicating the number of fire fighters assigned to each fire.

State	Optimal assignment	Approximate assignment
(2 3 4 4)	(2 3 1 1)	(3 3 1 0)
(2 2 4 4)	(3 2 1 1)	(3 3 1 0)
(2 1 4 2)	(2 2 1 2)	(3 2 0 2)
(1 1 4 2)	(2 2 1 2)	(2 2 0 3)
(2 4 2 2)	(2 1 2 2)	(3 0 2 2)

Table 6.3: States in which the Approximation and Optimal algorithms differed

6.2 Effects of fire configuration on optimal solutions

In travel-cost problems the agents' ability to effectively fight the fires is hampered when the fires are far apart. The farther the distance between the fires, the greater the penalty for switching from fire to fire. As a result, the agents will not switch from one fire to another unless fighting a new fire has a much higher utility than fighting the current one. Three different fire configurations were tested to determine the effects of distance on the optimal algorithm's performance. The "Small" fire configuration involved three fires at the coordinates (13, 11), (13, 13) and (13, 14). The "Mix" fire configuration had fires at (3, 3), (13, 11), and (13, 14). The "Far" fire configuration used fires at (3, 11), (9, 14) and (13, 10). The starting intensitities of the fires were varied in addition to the

fire locations. Table 6.4 shows the starting intensities of the various configurations. "Fire 1," "Fire 2," and "Fire 3" refer to the fires in the order they were listed above. For example, "Fire 1" in the "Small" configurations is the fire at coordinates (13, 11).

	Fire 1	Fire 2	Fire 3
FarUneven	1	2	3
FarEven	2	2	2
MixEven	2	2	2
MixDist	3	2	1
MixClose	1	2	3
SmallUneven	1	2	3
SmallEven	2	2	2

Table 6.4: Initial intensities of various fire configurations

Figures 6-2, 6-3 and 6-4 show the three fire configurations. The gray, orange and yellow rectangles represent the fires. The black rectangles arranged in a grid represent the streets. The green rectangles represent unburned ground.

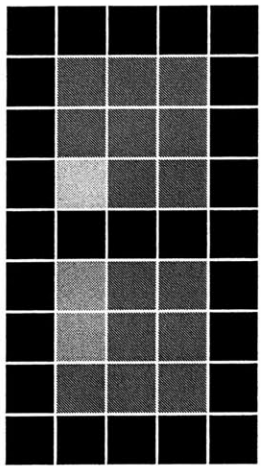

Figure 6-2: The "Small" fire configuration

Each configuration was tested 2000 times and the results were averaged together. In order to test larger distances, a "cost multiplier" was added to the travel costs. A "cost multiplier"' of 10

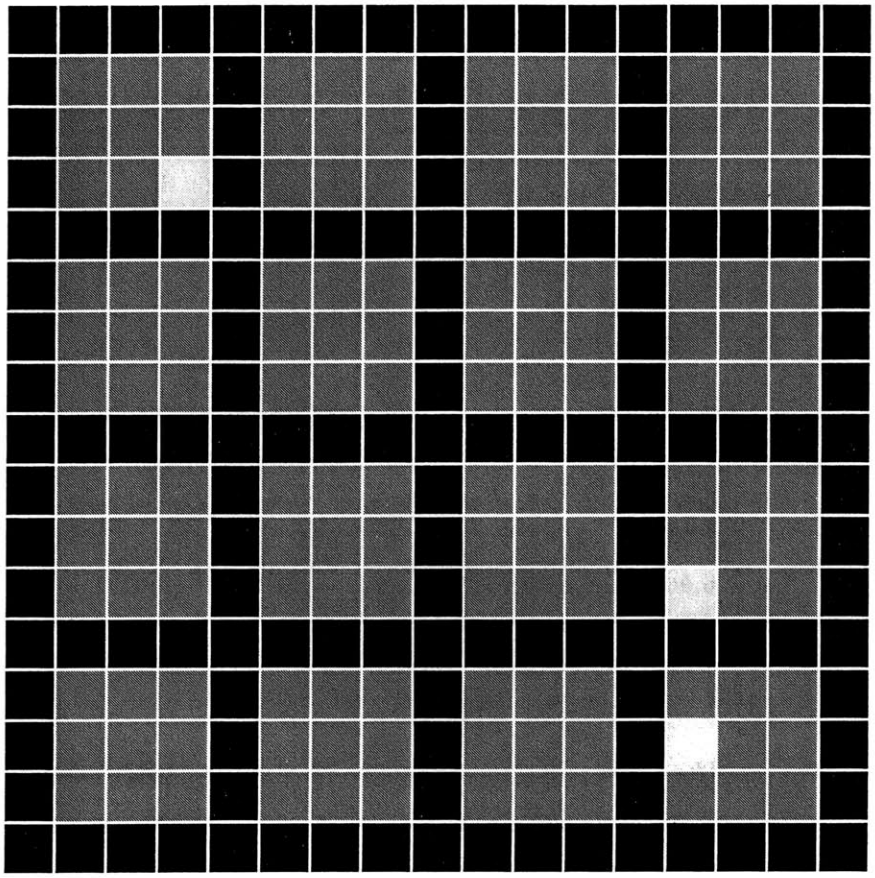

Figure 6-3: The "Mix" fire configuration

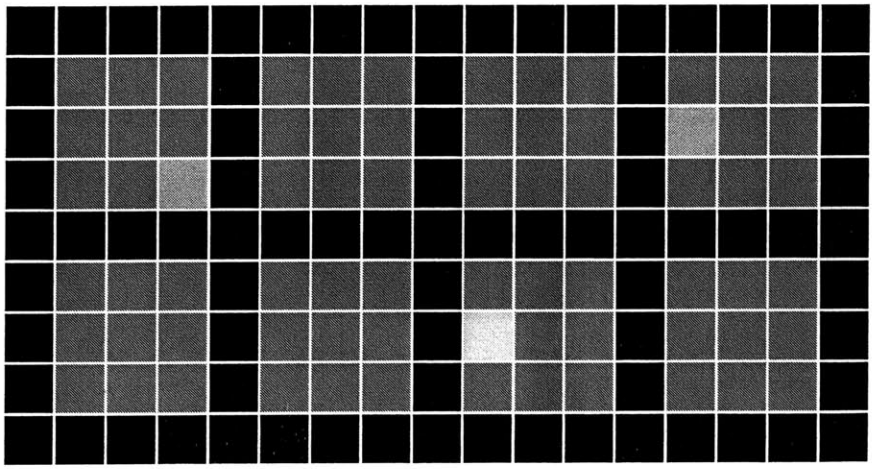

Figure 6-4: The "Far" fire configuration

gives the same results as if the fires had all been ten times as far apart. The intial positions of the agents were randomly determined in every trial. Table 6.5 summarizes the results of these trials. An ANOVA was used to calculate the probability that the results were all drawn from the same random distribution (i.e.; That the results were not signifigantly different.) The ANOVA run on all of the groups combined returned a value of less than .01%. However, an ANOVA run on the tests performed using a cost multiplier of 1 returned a probability of 36.23% that the results were drawn from the same distribution. An ANOVA run on the tests performed using a cost multiplier of 10 resulted in a probability value of 13.02%, while the ANOVA run on the data generated using a cost multiplier of 100 returned a probability of 5.05%.

	Cost multiplier		
	1	10	100
FarUneven	64.649	67.836	100.505
FarEven	64.420	70.674	90.005
MixEven	63.277	74.165	83.518
MixDist	66.085	78.281	86.820
MixClose	62.283	71.834	83.749
SmallUneven	63.295	66.605	78.728
SmallEven	62.283	62.309	74.409

Table 6.5: Effects of distance on optimal travel-cost algorithm effectiveness

The greater the distance cost between two fires, the greater the incentive for an agent not to move from one fire to another. The Small configurations were thus easier for the agents to deal with because they had to pay much smaller penalties to switch from fire to fire. In the Mix configurations the agents could move between the fires at (13, 11) and (13, 14) fairly easily, even with a high cost multiplier. However, the penalty for moving between the fire at (3, 3) and one of the other two fires was very and high and agents tended not to do so. For this reason, the MixDist configuration was generally harder for the agents to handle than the other Mix configurations; agents at either the (13, 11) or (13, 14) fires tended to avoid moving out to the high intensity fire at (3, 3) until the

difference in intensity between the far-off fire and the closer two fires was very high.

When the cost multiplier was raised to 100, the cost of travel between fires was so high that agents typically only moved from one fire to another if the fire they were fighting was nearly extinguished. Thus, the far configurations typically were very hard to deal with; if too few agents started near a fire it was unlikely that other agents would switch fires to help them and the fire would grow to a very high intensity. The agents were much less successful fighting the FarUneven configuration than they were in fighting the FarEven configuration; since one of the fires began with a high intensity it was very likely that not enough agents would be assigned to this fire and it would grow out of control.

The fact that two of the fires in the Mix configuration were close together meant that, even with a high cost multiplier, the agents were able to switch between the two close fires when necessary. However, the third, far off fire suffered from the same problems as the fires in the Far configurations did; agents typically did not arrive at or leave from the fire unless the difference in intensity between the fire at (3, 3) and one of the other two fires was very large.

Similar results were obtained by the approximate travel-cost algorithm. During the execution of the greedy heurisitic to assign agents to fires, the approximation algorithm estimated the cost of moving agents between fires. It later used the stepping stone algorithm to exactly compute this cost, but the initial approximation was often an over estimate of the cost of moving agents around. Thus, the results for the approximate travel-cost algorithm were considerably worse than the results for the optimal algorithm. Table 6.6 demonstrates these results.

A set of ANOVAs was run on this data to help determine the signifigance of the results. The ANOVA run on all of the groups combined again returned a value of less than .01%. The ANOVA run on the tests performed using a cost multiplier of 1 returned a probability of 24.62% that the results were drawn from the same distribution. An ANOVA run on the tests performed using a

cost multiplier of 10 resulted in a probability value of 17.37%, while the ANOVA run on the data generated using a cost multiplier of 100 returned a probability of 2.24%.

	Cost multiplier		
	1	10	100
FarUneven	75.333	79.189	111.215
FarEven	64.460	73.314	94.393
MixEven	65.560	74.233	84.271
MixDist	75.048	92.109	109.960
MixClose	64.416	77.871	83.791
SmallUneven	65.090	63.458	89.305
SmallEven	62.953	75.904	74.581

Table 6.6: Affects of distance on approximate travel-cost algorithm effectiveness

6.3 Parameter tuning in the Boltzman distributions

As mentioned above, the decoupled and fully collaborative algorithms use two parameters to control the Boltzman distribution. A learning algorithm was run to determine the optimal values of C_1 and C_2; this algorithm returned values of $C_1 = .51$ and $C_2 = .34$ for the decoupled algorithm and $C_1 = 0.3246$ and $C_2 = 0.068$ for the fully collaborative algorithm. In order to verify these findings, the algorithm was run on a ten-fire, fifteen agent problem with a variety of values for C_1 and C_2. Each pair of values was repeated 2000 times and the results were averaged together. Table 6.7 summarizes the results for the decoupled algorithm. An ANOVA was run on all of the data sets, it returned a probability less than .01% that the data was all drawn from the same distribution. However, an ANOVA run on the data sets with $C_1 = .5, 1, 1.5$ returned a probability of 32.65% that the data was all drawn from the same distribution.

This chart shows the gradient that the learning algorithm followed; running the algorithm with $C_1 = .51$ and $C_2 = .34$ resulted in 214.25 average "damage points" to the city over 2000 trials.

A similar chart was constructed for the fully collaborative algorithm. However, the minimum

		C_1			
		.00001	.5	1	1.5
C_2	0	477.58	216.73	217.79	219.85
	.5	476.89	217.91	221.50	224.01
	1	475.52	220.37	221.22	221.21
	1.5	474.52	218.42	222.87	222.14

Table 6.7: C_1 and C_2 values in the Boltzman Distribution for the cost-free decoupled algorithm

possible value of C_1 that could be tested was .001. Each agent was aware only of its immediate area and thus it was possible for an agent to not know of any of the fires. Java seemed to occasionally throw floating exceptions when an agent did not know of any fires; this never happened during the decoupled algorithm because the agents could see the entire city. The gradient mapped out using the fully collaborative algorithm was much less consistent and less clear cut than the one mapped out using the decoupled algorithm; repeated tests confirmed that many different values for C_1 and C_2 gave nearly the same result. The optimal values for C_1 and C_2 resulted in an average total damage of 294.9385 over 2000 trials.

An ANOVA run on all of the data sets returned a probability of 68.23% that the data was all drawn from the same original distribution.

		C_1			
		.001	.5	1	1.5
C_2	0	303.928	297.458	301.011	302.650
	.5	304.781	299.982	299.566	302.604
	1	303.997	301.831	295.891	298.133
	1.5	307.397	298.594	300.663	304.718

Table 6.8: C_1 and C_2 values in the Boltzman Distribution for the cost-free, fully collaborative algorithm

The values of C_1 and C_2 were recalculated for travel-cost problems. The optimal values for C_1 and C_2 were found to be .49 and .22. Various values of C_1 and C_2 were tested and the "damage

gradient" was mapped out; the results are shown in table 6.9. The optimal values for C_1 and C_2 were calculated to be .38 and 1.27 respectively.

An ANOVA run on the entire data set returned a probability of less than .01% that the data was all drawn from the same distribution. Running the ANOVA on only the data sets with $C_1 = .5, 1, 1.5$ returned a probability of 24.36% that the data was all drawn from the same distribution.

		C_1			
		.00001	.5	1	1.5
C_2	0	517.30	263.59	277.54	279.34
	.5	521.01	256.48	268.07	267.40
	1	519.33	251.20	256.36	264.08
	1.5	523.49	253.62	257.67	268.62

Table 6.9: C_1 and C_2 values in the Boltzman Distribution for the travel-cost decoupled algorithm

A similar table of data was compiled for values of C_1 and C_2 in the travel-cost, fully collaborative problem. The optimum values for C_1 and C_2 were found to be 0.712 and 0.023. Table 6.10 shows the results.

An ANOVA run on all of the data again returned a probability of less than .01% that the data was drawn from the same distribution2. Excluding the data sets in which $C_1 = .001$ from the ANOVA resulted in a probability of 54.36% that the data was all drawn from the same distribution.

		C_1			
		.001	.5	1	1.5
C_2	0	424.156	323.980	324.412	325.212
	.5	423.633	324.676	324.844	326.437
	1	421.698	323.616	331.148	327.727
	1.5	425.739	324.405	327.888	332.069

Table 6.10: C_1 and C_2 values in the Boltzman Distribution for the travel-cost, fully collaborative algorithm

In general, smaller values of C_1 and C_2 meant that the agents were more likely to fight fires

with higher utilities, regardless of the loss of efficiency due to multiple fire fighters fighting the same fires. In general, this is a useful trait for the agents to have because it means that more important fires will have more agents devoted to them. However, below a certain threshold the agents tend to cluster together too much and efficiency suddenly drops. The gradient descent algorithm was run for a few problems of different sizes, but the optimal values for C_1 and C_2 did not change signifigantly as the problem size changed.

6.4 Comparison of Approximate, Decoupled and Fully Collaborative algorithms

The approximation, decoupled and fully collaborative algorithms were run on a series of ten-fire and fifteen-fire problems with varying numbers of fire fighters. The goal of these tests was to determine the effectiveness of the decoupled and fully collaborative algorithms for various problem sizes. Each algorithm was run two thousand times on each problem size and the results were averaged together. Table 6.11 shows the results for various numbers of agents solving a ten-fire problem. Table 6.12 shows the results of various numbers of agents in a city with fifteen fires. Figures 6-5 and 6-6 show the same data in graph form. The error bars in the graph represent a 95% certainty that a value was drawn from a given distribution.

Algorithm	Number of agents				
	7	10	15	20	25
Approx	246.975	215.392	197.455	174.765	160.838
Decoupled	298.44	257.6275	216.3255	190.828	178.576
Fully col.	353.6585	325.0745	294.9385	275.068	259.1345

Table 6.11: Algorithm results for cost free, ten-fire problems

These charts illustrate some interesting behavior. As more and more agents are added to the

Algorithm	Number of agents				
	10	15	20	25	30
Approx	370.82	326.4	292.9055	272.213	261.3445
Decoupled	453.1465	386.3825	340.7625	313.016	294.137
Fully col.	523.712	484.594	457.022	436.081	411.8205

Table 6.12: Algorithm results for cost free, fifteen-fire problems

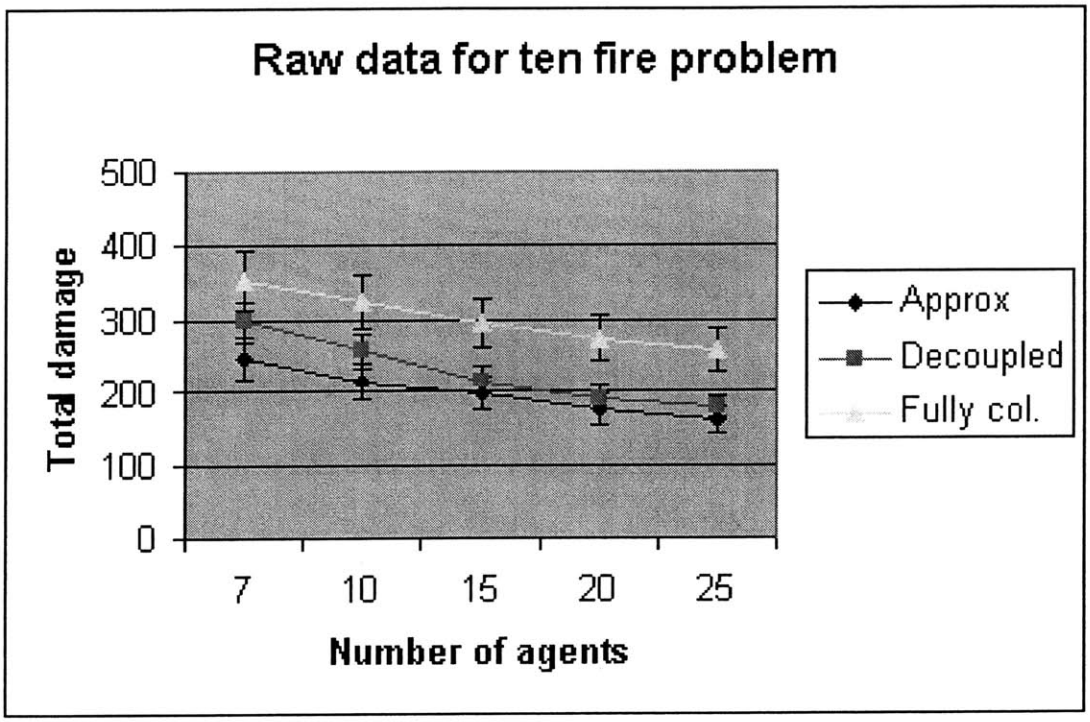

Figure 6-5: Comparison of algorithms in ten-fire, cost free problems.

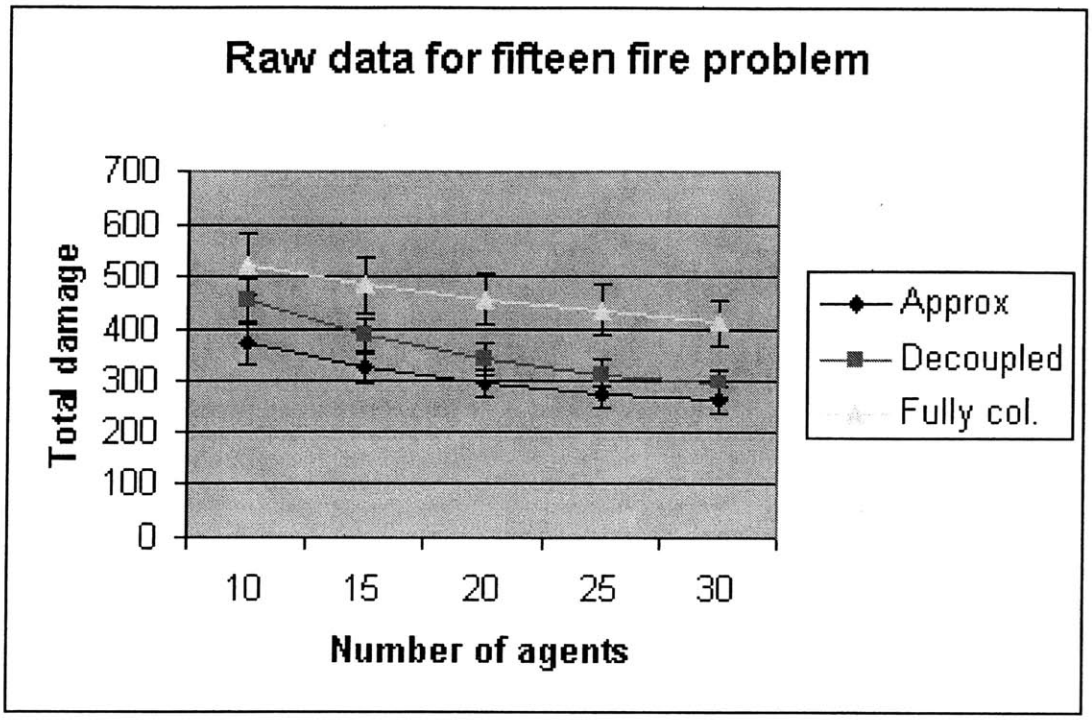

Figure 6-6: Comparison of algorithms in fifteen-fire, cost free problems.

problem, the decoupled algorithm does better and better relative to the approximation algorithm. However, the fully collaborative algorithm seems to do worse and worse (relative to the approximation algorithm) as the number of agents increases. Figures 6-7 and 6-8 show the damage to the city during run-throughs of the decoupled and fully collaborative algorithms as a percentage of the damage done during a run-through of the approximation algorithm.

Since the agents in the decoupled algorithm don't share their plans with each other or coordinate their actions, each agent acts as if it were the only fire fighter in the city. This means that fire fighters in the decoupled algorithm prioritize small fires that can be handled by a single agent very highly as compared to the approximation algorithm.

For example, a four fire, seven agent problem with fires *(1 1 3 4)* resulted in a distribution of *(2 2 3 0)* using the approximation algorithm; 2 agents were assigned to each of the level one fires

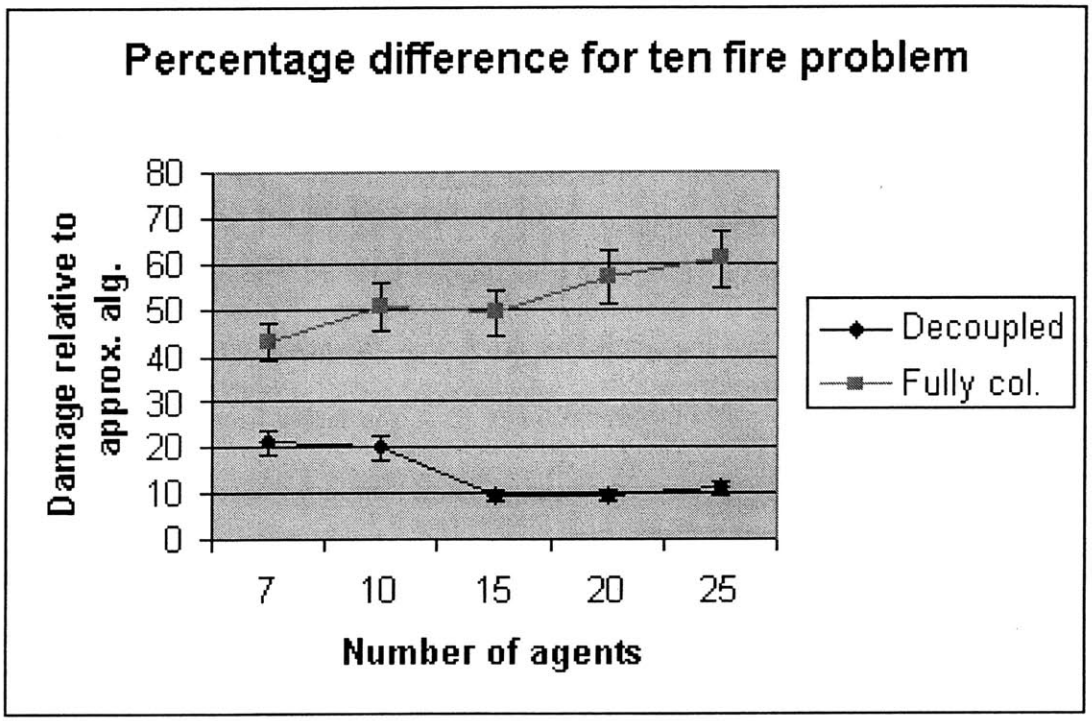

Figure 6-7: Damage relative to approximate algorithm in ten-fire, cost free problems.

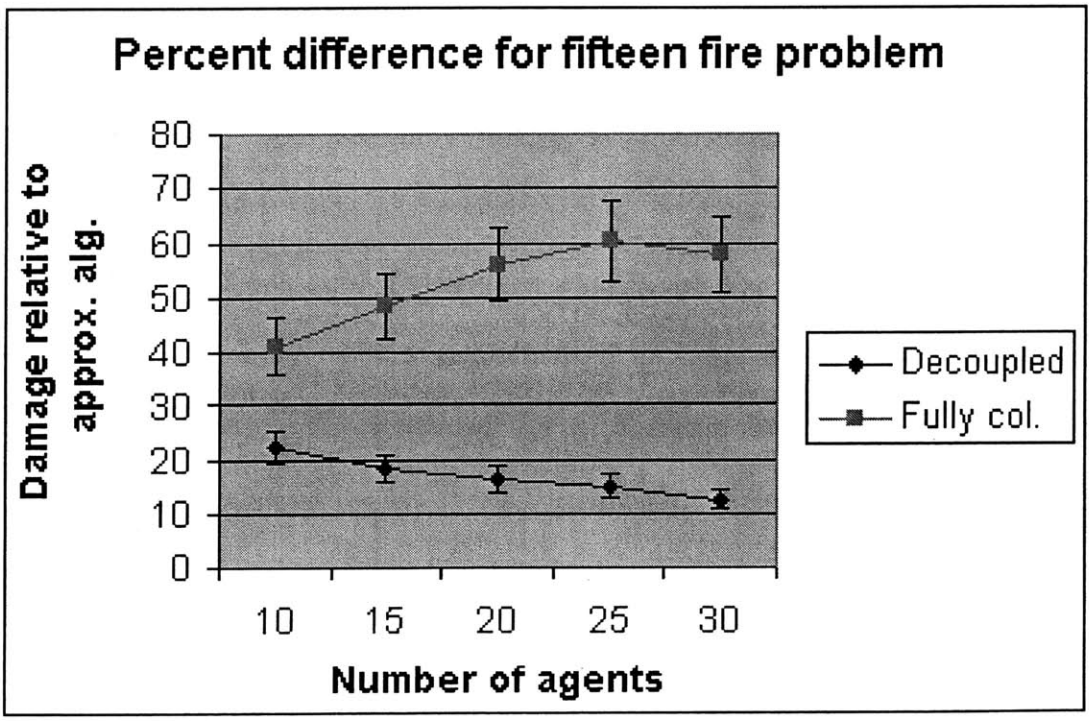

Figure 6-8: Damage relative to approximate algorithm in fifteen-fire, cost free problems.

and three agents were assigned to the level three fire. The decoupled algorithm generates a set of probabilities that the agent will be assigned to any particular fire. The probability distribution generated for this particular state was *(.35 .35 .174 .117)*, which would result in seven agents being distributed as follows: *(2.45 2.45 1.22 .82)*. As more and more agents are added, the decoupled algorithm begins to put out low intensity fires very quickly because it assigns too many agents to these problems. However, because the small fires are being put out so quickly and because there are still a few agents assigned to the higher intensity fires, the larger fires do not typically cause too much damage before the smaller fires are dealt with and more agents are assigned.

The agents in the fully collaborative algorithm are affected by two major factors that agents in the approximation algorithm are not affected by. The first handicap that the agents in the fully collaborative algorithm have is that they must spend a portion of their time exploring the city rather than fighting fires directly. At each time step the agents in the fully collaborative algorithm have a 10% chance of choosing to explore the city rather than fight a particular fire. Since the portion of the city covered in fires is small compared to the total area of city, most exploration does not result in new information and is thus wasted effort. Additionally, at the beginning of each run through of the simulation, a given agent only has information about a few fires in its immediate area. The agent will choose to fight one of those and will only learn about far away fires if it goes exploring and discovers them or meets an agent who has explored and found them. Thus, in the initial stages of the simulation the agents distribute themselves based on very limited information. Because of these two factors, adding in additional agents causes the fully collaborative algorithm to improve at a slower rate than the approximation and decoupled algorithms, as indicated by figures 6-7 and 6-8.

The same set of experiments were conducted using travel-cost versions of all three algorithms. Tables 6.13 and 6.14 summarize the results.

Algorithm	Number of agents				
	7	10	15	20	25
Approx	268.1305	250.1075	229.1075	221.555	215.182
Decoupled	325.54	297.2945	264.2325	233.835	222.3285
Fully col.	394.5895	356.9205	325.036	293.0805	273.0735

Table 6.13: Algorithm results for travel-cost ten-fire problems

Algorithm	Number of agents				
	10	15	20	25	30
Approx	407.862	371.9365	359.2365	344.077	335.9845
Decoupled	486.3785	436.227	400.389	372.0735	343.454
Fully col.	564.1555	507.9245	462.788	433.258	414.7075

Table 6.14: Algorithm results for travel-cost fifteen-fire problems

Figure 6-9 and 6-10 show the same data in graph form. Figure 6-9 shows the performance of the algorithms on ten-fire problems and figure 6-10 shows the performance of the algorithm on fifteen-fire problems.

All of the travel-cost algorithms performed considerably worse than their cost free counterparts because the agents had to factor in the cost of moving from fire to fire into their considerations and thus couldn't always fight the fire at which they would have the most impact. In other respects the travel-cost algorithms behaved very similarly to the cost-free version. As the number of agents increased, the decoupled algorithm began to achieve results that were very similar to the approximation algorithm. The fully collaborative algorithm also improved as the number of agents was increased, but it improved much less quickly than the decoupled algorithm did.

Figures 6-11 and 6-12 show the damage to the city during run-throughs of the decoupled and fully collaborative algorithms as a percentage of the damage done during a run-through of the approximation algorithm.

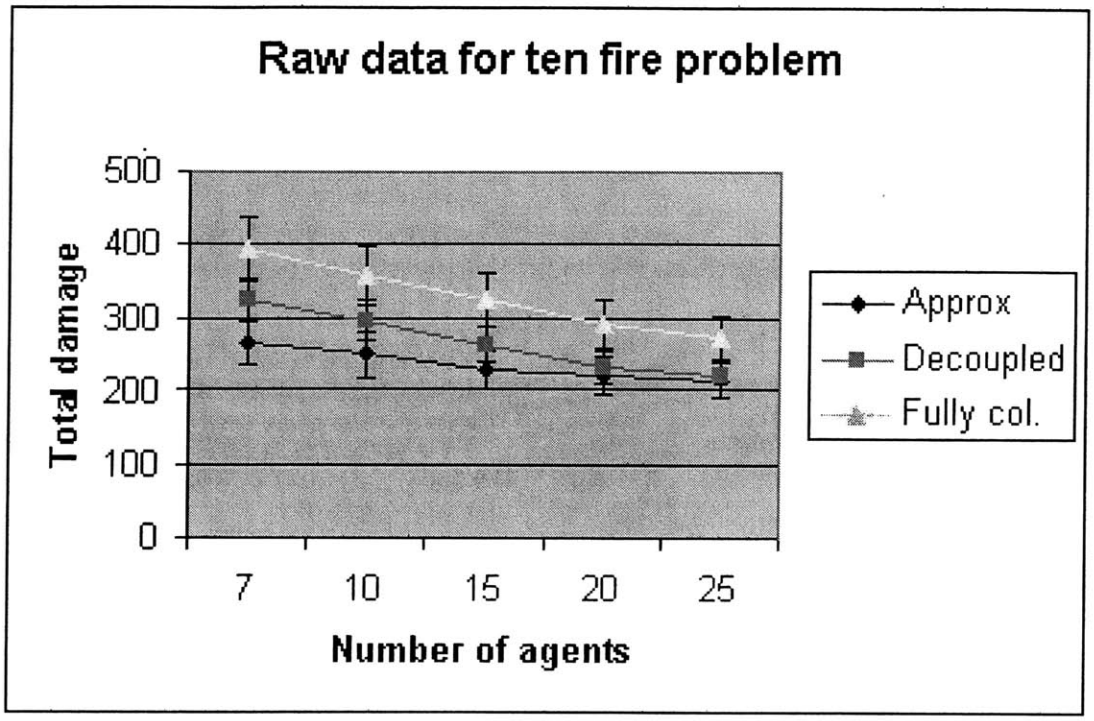

Figure 6-9: Comparison of algorithms in ten-fire, travel-cost problems.

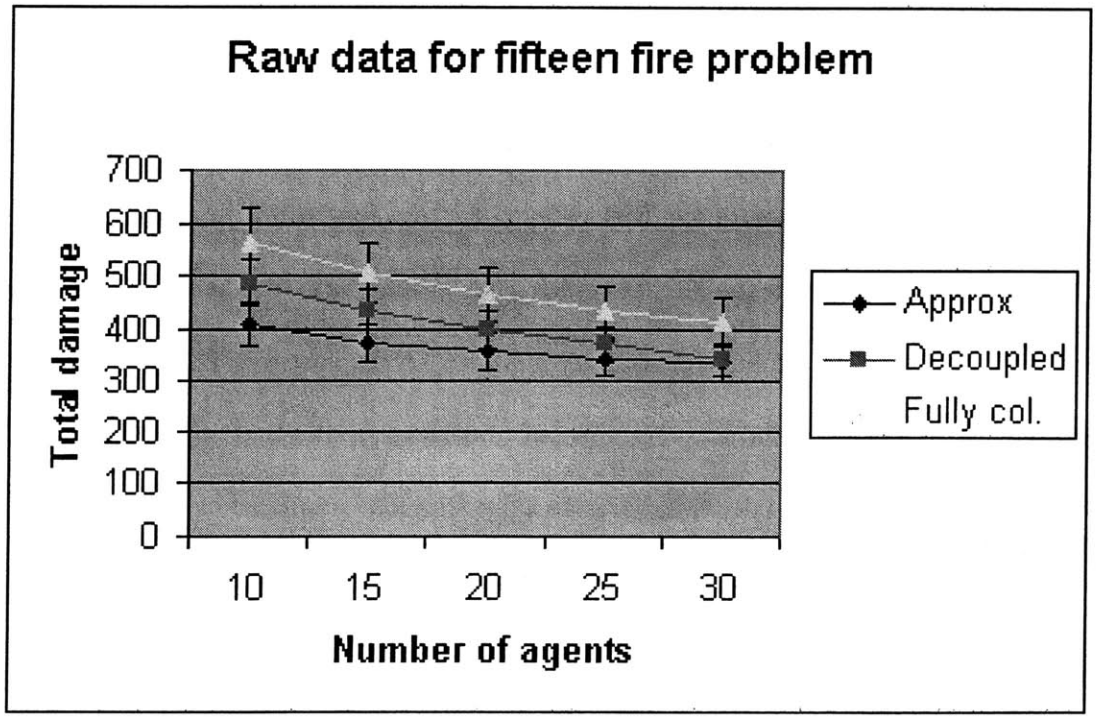

Figure 6-10: Comparison of algorithms in fifteen-fire, travel-cost problems.

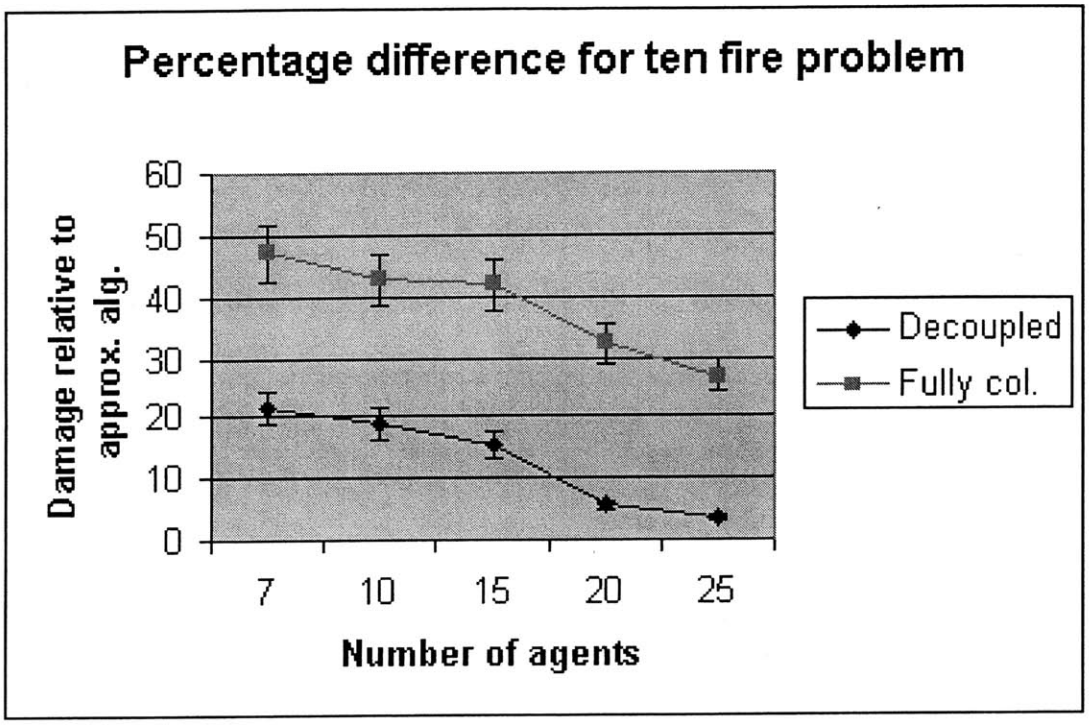

Figure 6-11: Damage relative to approximate algorithm in ten-fire, travel cost problems.

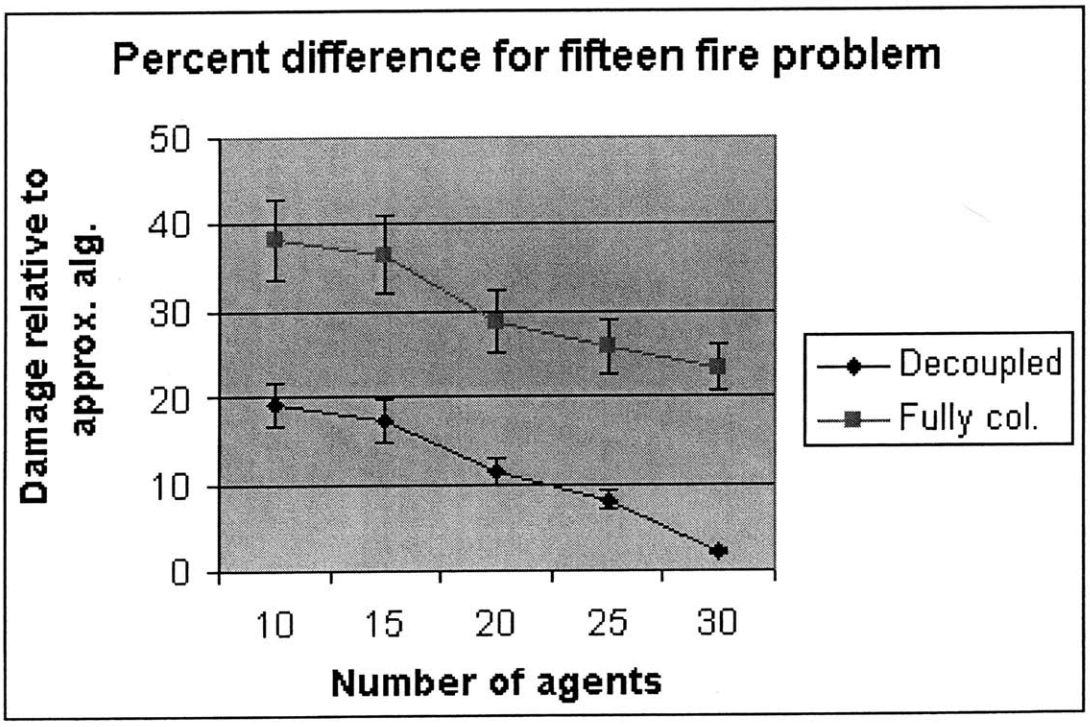

Figure 6-12: Damage relative to approximate algorithm in fifteen-fire, travel cost problems.

6.5 Effects of vision in the fully collaborative algorithm

The area that each agent can "see" in the fully collaborative algorithm is dependent on two different factors: "unobstructed vision" and "obstructed vision." The unobstructed vision parameter controls how far down the road an agent can see while the obstructed vision parameter controls how far "through buildings" the agent can see. Figure 6-13 demonstrates the two factors; the red arrows down the centers of the roads represents the agent's unobstructed vision. The agent's obstructed vision is represented by the magenta arrows leading off of the roads. The total area that the agent can see is enclosed within the thick red cross.

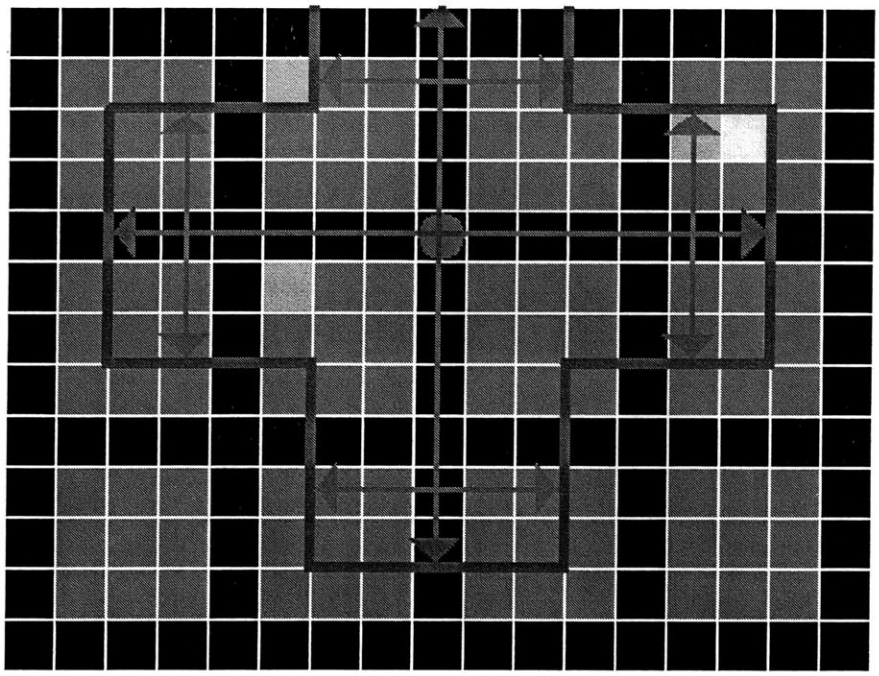

Figure 6-13: Factors controlling agent vision

Altering the two vision controlling parameters changes the amount of information available to the agents at each time step; these effects were mapped out in a series of trials. The table below shows the average values for various combinations of factors. Since an obstructed vision of 2 means that the agents can detect fires about half a block away despite interposed buildings (for example,

the agents look through windows), obstructed vision values greater than 2 are highly unrealistic. Obstructed vision values of 0 prevent the agents from fighting the fires; they can't see off of the road at all and as a result can not detect the fires. Each pair of values was tested 4000 times on a 10 fire, 15 agent problem and the results were averaged together.

		Obstructed Vision	
		1	2
Unobstructed Vision	1	409.442	342.79725
	2	370.8135	328.1255
	3	354.4575	312.21425
	4	338.4675	302.054
	5	322.7125	297.51625
	6	314.997	289.98525
	7	307.4945	288.0105
	8	302.1215	287.95425
	9	297.312	281.636
	10	296.9354	283.6825
	11	295.3885	279.33125
	12	294.40575	280.4715
	13	289.95675	279.28975
	14	289.9015	278.1965

Table 6.15: Effectiveness of various vision parameters

Figure 6-14 demonstrates the same data. The top line represents the effectiveness of the agents for various unobstructed vision ranges when their obstructed vision range was set to 1. The bottom line represents the effectiveness of the agents when their obstructed vision range was set to 2. The Y-axis measures the amount of damage caused to the city before the fires were put out. The error bars represent a 95% of the range of values used to generate each data point.

Increasing the agents' range of vision allowed them to fight the fires more effectively. However, past a certain threshold the advantage to the agents of each increase in vision tapered off; each increase no longer represented a signifigant change (percentage-wise) in the area that the agent could see.

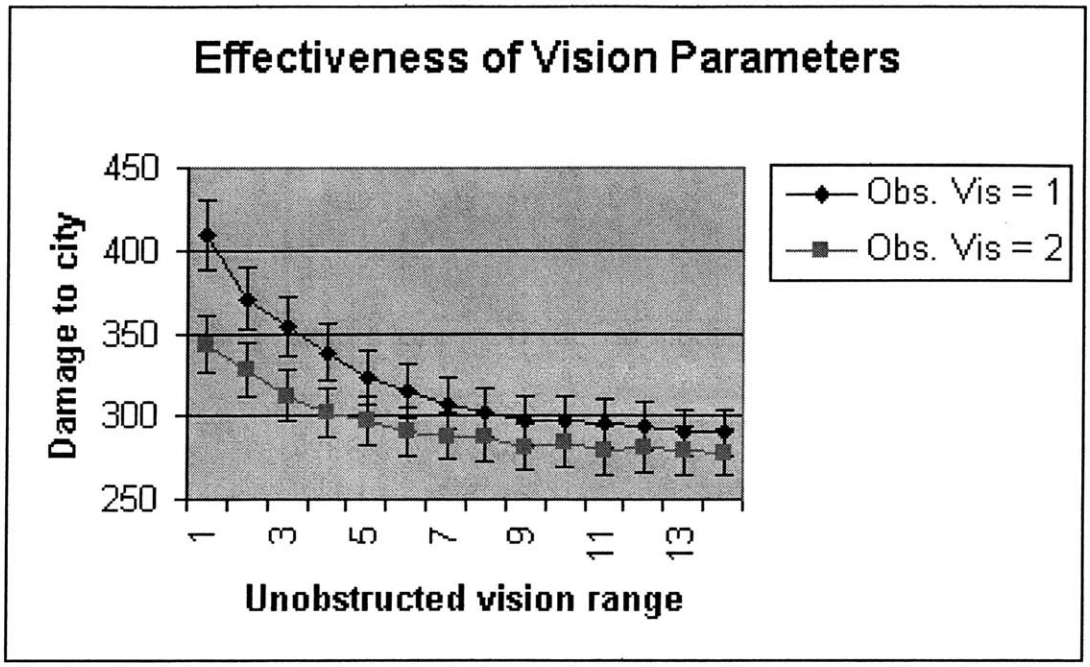

Figure 6-14: Effectiveness of agents with various vision ranges.

Increasing the agent's obstructed vision was more generally more helpful than increasing the agent's unobstructed visions. For example, an agent with an obstructed vision of 1 and an unobstructed vision of 4 can see about as much ground as an agent with an obstructed vision range of 2 and an unobstructed vision range of 2, but agents with higher obstructed vision ranges typically fought fires more effectively. This is largely due to the fact that fires are never located in streets and thus, much of the area that an agent with high unobstructed vision can see will never have a fire in it. Additionally, agents with a high range of unobstructed vision often find their field of view blocked by buildings or by the edge of the city and thus can't fully take advantage of the area that they can see.

6.6 Travel Time of Agents in Constrained Collaborative algorithm

A series of tests were run to determine how much time the agents in the constrained collaborative algorithm spent traversing the city. The simulator was run 2000 times for each of a number of different problem sizes and the results were averaged together. Table 6.16 summarizes the results. Problem sizes are expressed in the format MxN, where M is the number of fires and N is the number of agents. The results represent the amount of time the agents spent walking from place to place as a percentage of the total time the simulator was run.

Problem Size	Percent
4x3	27.512
4x4	30.748
4x6	37.112
4x8	39.029
4x10	41.355
10x7	33.847
10x10	39.527
10x15	39.569
10x20	45.933
10x25	46.341

Table 6.16: Time spent traversing the city in the constrained collaborative problem

The time each agent spent walking around the city decreased as the number of fires grew but increased as the total number of agents increased. As the number of fires increases, the agents have a greater and greater chance of quickly finding fires to fight and need to spend less time wandering around the city. As the number of fire fighters increases, the odds that the agents are unevenly distributed around the city increase. As a result, agents in an overcrowded area will typically seek out fires that are being fought by fewer fire fighters. Although this means that the average agent spends more of its time walking around, it also allows the agents to more evenly distribute their efforts around the city.

6.7 Effectiveness of the Constrained Collaborative algorithm

The constrained collaborative algorithm requires that agents walk around the city instead of teleporting from fire to fire. As a result, the agents have to spend a lot of time traveling around the city instead of fighting fires. In order to guage the effect of this additional constraint, the constrained collaborative algorithm was tested on a series of problems of various sizes. Tables 6.17 and 6.18 summarize the results, each problem was tested 2000 times and the results were averaged together. The results of the fully collaborative algorithm on the same problems are also listed for comparison purposes.

Algorithm	Number of agents				
	7	10	15	20	25
Fully col.	353.6585	325.0745	294.9385	275.068	259.1345
Const col.	1899.214	1322.823	1047.152	760.778	666.864

Table 6.17: Results (in damage points) of the constrained collaborative algorithm on ten-fire problems

Algorithm	Number of agents				
	10	15	20	25	30
Fully col.	523.712	484.594	457.022	436.081	411.821
Const col.	2805.162	2544.182	1962.459	1815.677	1706.526

Table 6.18: Results (in damage points) of the constrained collaborative algorithms on fifteen-fire problems

Figures 6-15 and 6-16 show the same data in graph form; figure 6-15 shows the data for the ten-fire problem and figure 6-16 shows the data for the fifteen-fire problem.

The agents in the constrained collaborative problem performed much worse than the agents in the fully collaborative problem. This was due largely to the fact that the agents had to spend almost half their time on average traveling around the city. Additionally, the fact that traveling from one fire to another required multiple time steps highly discouraged agents from switching

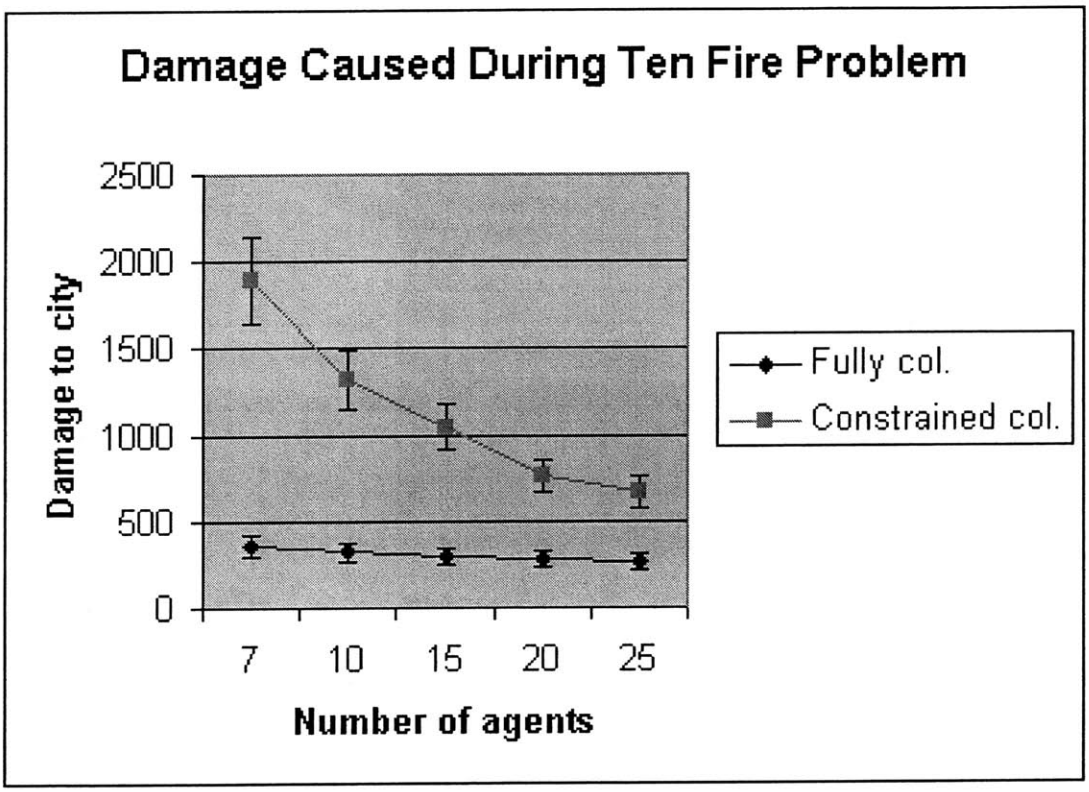

Figure 6-15: Fully collaborative and constrained collaborative algorithm for ten-fire problem

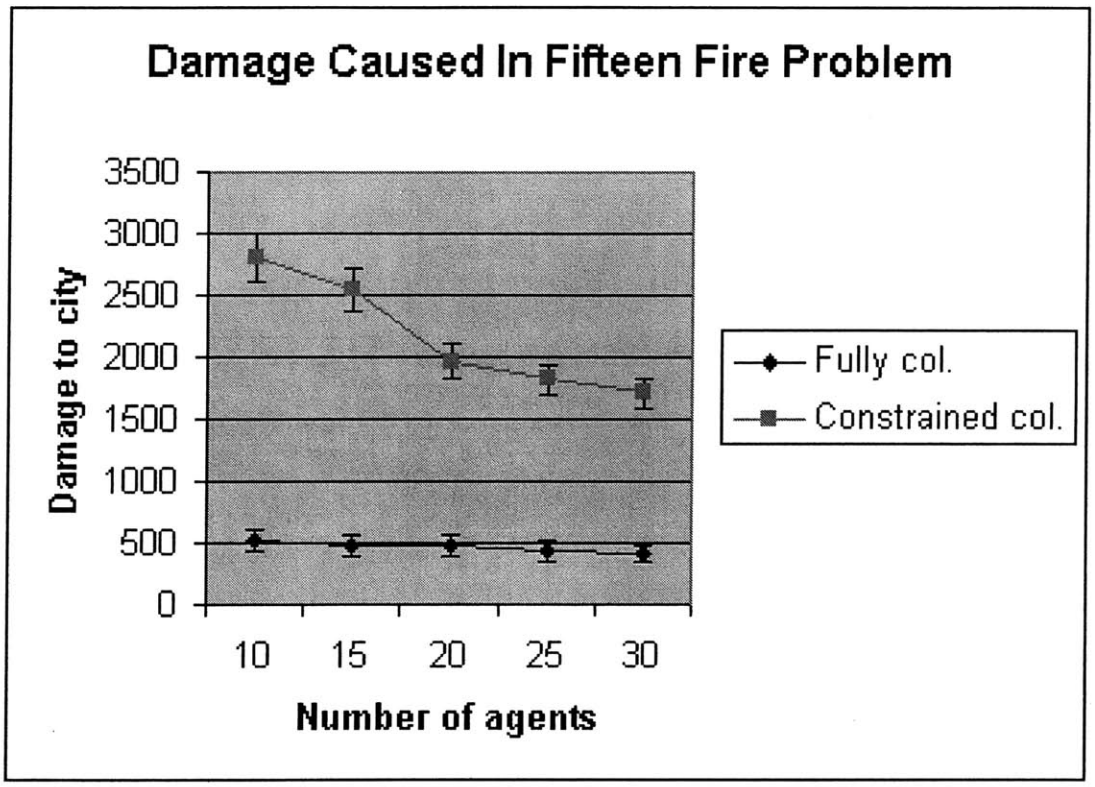

Figure 6-16: Fully collaborative and constrained collaborative algorithm for fifteen-fire problem

between fires; only rarely would an agent leave a fire and to fight another fire that was more than a block or two away.

6.8 Effects of vision range on the constrained collaborative algorithm

The agents in the constrained collaborative algorithm could see the area around them based on their "obstructed vision" and "unobstructed vision." These two factors function exactly the way they do in the fully collaborative algorithm; section 6.5 describes these factors. 4000 trials were conducted for various values of obstructed and unobstructed vision in a 10 fire, 15 agent problem. The average results are presented below, in table 6.19.

		Obstructed Vision	
		1	2
Unobstructed Vision	1	2032.542	1273.819
	2	1811.002	1132.732
	3	1695.266	1050.167
	4	1683.592	1025.398
	5	1699.655	998.239
	6	1660.269	1003.756
	7	1674.990	979.355
	8	1668.745	990.082
	9	1675.276	982.451
	10	1650.274	1025.258
	11	1637.385	997.327
	12	1678.997	975.460
	13	1646.174	982.573
	14	1687.638	986.031

Table 6.19: Effectiveness of vision in constrained collaborative algorithm

Figure 6-17 shows the same data in graph form; the error bars represent 95% certainty with regards to each data set.

The agents in the constrained collaborative algorithm benefit from having vision above a cer-

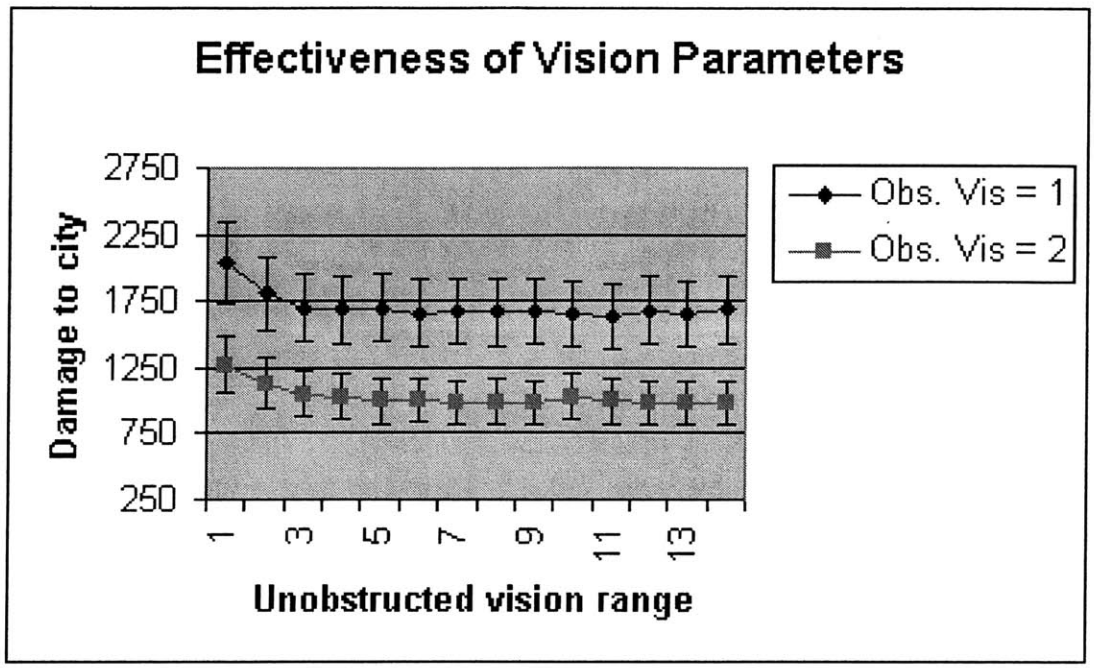

Figure 6-17: Effects of vision parameters on constrained collaborative algorithm

tain threshold. However, since there is a large penalty associated with moving from across large distances, the agents do not benefit much from information about fires that are far away. Unless an agent has just extinguished a fire and is looking for any new fire to fight, it is unlikely that the agents will use information about fires that are more than a block away.

6.9 Effects of non-line-of-sight communications on the constrained collaborative algorithm

The agents in the constrained collaborative algorithm were originally restricted to line of sight communications. This experiment examined the effects of allowing agents to communicate with each other regardless of whether or no there was a clear line of sight between them.

Each agent was allowed to communicate with any other agent within a certain radius, regardless of intervening obstacles. 4000 trials were performed for each communications radius, and the results

are summarized below in table 6.20.

Radius of communications range	Damage to the city
7	1165.2343
10	1117.7045
12	1085.9223
15	942.3124
20	987.1913
25	970.5431

Table 6.20: Effectiveness of communication despite lack of line-of-sight

As with increased vision, increased communications improved the effectiveness of the agents by providing them with information about more and more fires. After a certain point, the effects of increased vision seem to plateau; the cost of moving to a fire that is very far away is so large that the agents can't often take advantage of this additional information. As a basis for comparison, a range of 7 means that the agents could communicate with anyone within a block and a half of themselves, while a range of 25 means that the agents can communicate with anyone within a 5 block radius.

6.10 Effects of non-constant vision on the constrained collaborative algorithm

This experiment explored the effects of a slightly different vision model. Rather than allowing the agents to see all of the fires in a fixed area around them, the agents were able to detect fires at different ranges depending on their intensities. For example fires with intensity 1 might be visible a block away while fires with intensity 4 might be visible two blocks away. In these experiments agents were still required to establish a direct line of sight with a fire before they were able to detect it.

The ranges at which fires were visible were linearly proportional to the intensities of the fires. For example, if a fire of intensity 1 was visible at range 3 then a fire of intensity 2 would be visible at range 6, a fire of intensity 3 would be visible a range 9 and a fire of inensity 4 would be visible at range 12.

4000 trials were conducted for various detection ranges in a 10 fire, 15 agent problem. The ranges listed below reflect the range at which a level four fire could be seen. The range at which lower intensity fires could be see was found by diving the range appropriately and then rounding up. The average results are presented below, in table 6.21.

Range to see level 4 fire	Total damage to the city
1	1423.624
2	1324.423
3	1192.733
4	1024.147
5	1079.373
6	991.139
7	980.029
8	979.240
9	1002.884
10	989.294
11	973.744
12	981.853
13	995.471
14	969.149

Table 6.21: Effectiveness of vision in constrained collab. algorithm

Figure 6-18 shows the same data in graph form.

As with the first set of vision experiments, the effects of increased vision increase greatly at first but then seem to drop off after a certain plateau. Agents are very unlikely to switch to a low intensity fire far away from themselves, so being able to see only high intensity fires at long range provides almost the exact effect as being able to see any fire at long range. Limiting the agent's vision this way also lightens the computational load every agent has to perform before making a

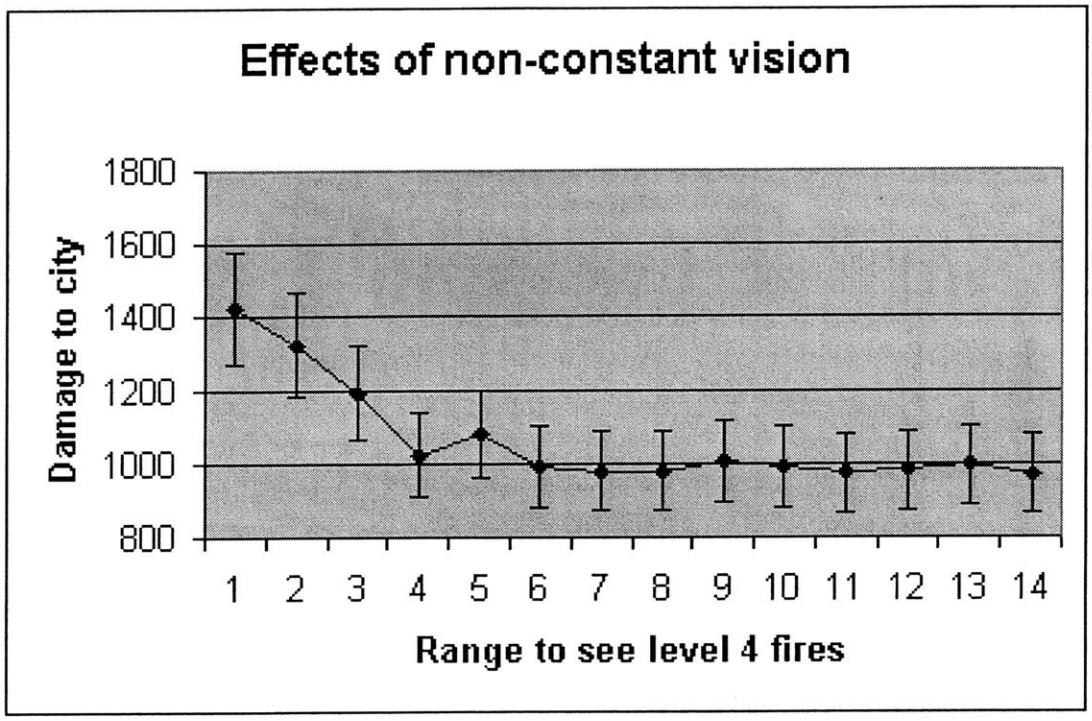

Figure 6-18: Effects of non-constant vision on constrained collaborative algorithm

decision. The simulator used in this research allows each agent unlimited time to make a decision about which fire to fight at any given time step. However, agents acting in a simulator that limited the amount of information they could process at each time step could voluntarily filter out low intensity fires that were long distances away without suffering a large loss in effectiveness.

6.11 Damage caused as a function of time

A series of tests were run to map out the rate at which damage was caused to the city in each of the various scenarios. 500 trials were conducted and the damage done to the city at each time step was stored. The average damage caused at every time step is graphed below; figure 6-19 shows the performance of optimal, approximate, decoupled and fully collaborative algorithms on 4 fire, 7 agent problems. Figure 6-20 shows the performance of the approximate, decoupled and fully

collaborative algorithms on 10 fire, 15 agent problems.

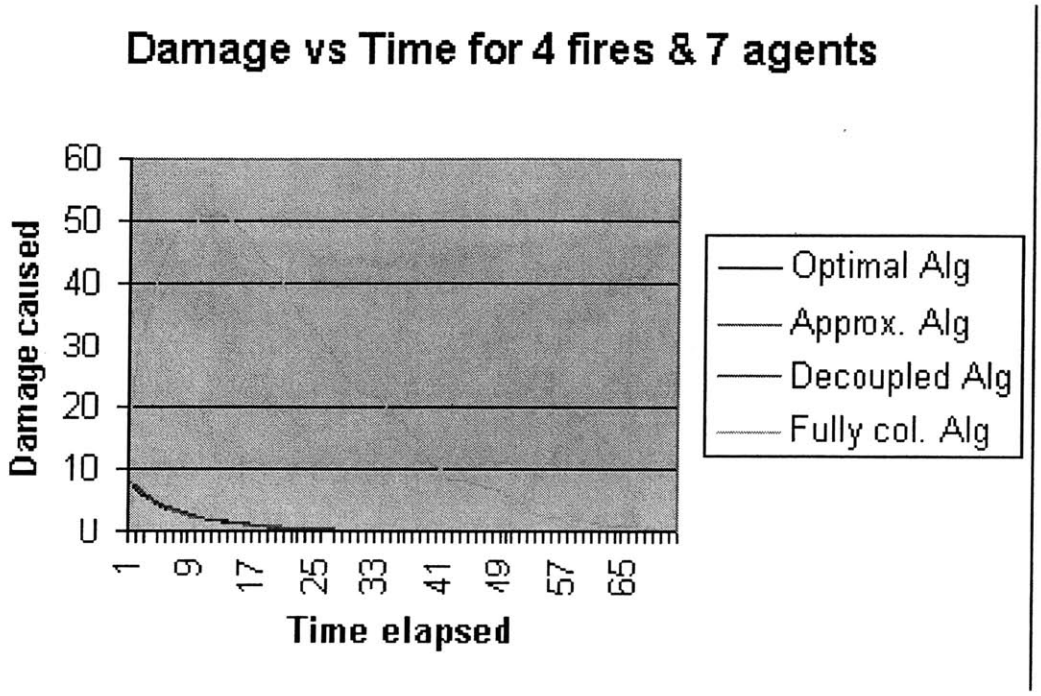

Figure 6-19: Damaged caused as a function of time for 4 fire, 7 agent problems

The performances of the optimal and approximate algorithms followed roughly the same pattern. The largest fires were tackled first, resulting in a quick overall drop in fire intensity. The agents then spread out to fight the low intensity fires; this ensured that none of them would have a chance to rebuild in intensity but meant that portions of the city were on fire for long periods of time. The decoupled algorithm tended to fight smaller fires first, which meant that the initial drop in fire intensity was less rapid as other fires grew in intensity. However, putting out small fires quickly meant that the decoupled algorithm was able to extinguish all the fires in the city much more rapidly than the optimal or approximate algorithms, despite the fact that more total damage was suffered. For example, agents with centralized communication (using the approximation algorithm) in the ten-fire, fifteen agent problem occasionally took 59 time steps to extinguish all the fires in the city, but the city suffered only 148.628 "damage points" on average. Agents making independent

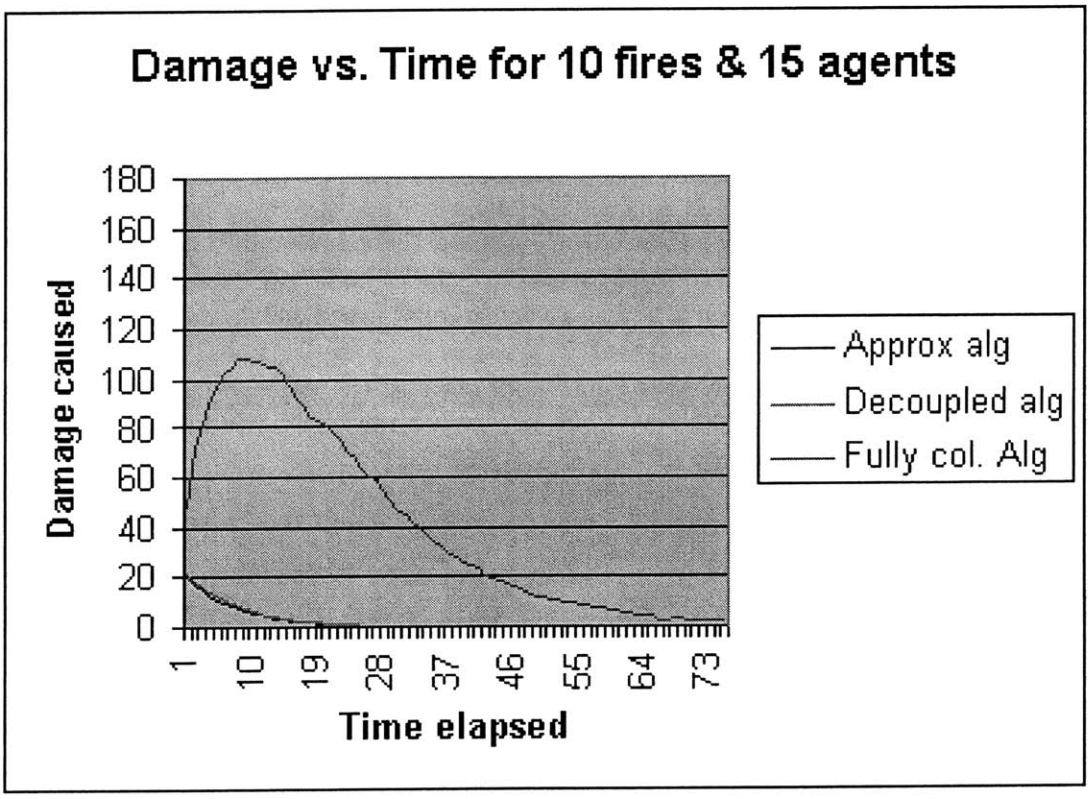

Figure 6-20: Damaged caused as a function of time for 10 fire, 15 agent problems

decisions using the decoupled algorithm never took longer than 49 time steps to put out the fires, but the city suffered an average of 163.196 damage points in the process.

Since the agents executing the fully collaborative algorithm were not given global vision, they typically had a much harder time finding fires to fight initially. Further more, because many fires were not discovered for a long time, the damage to the city per unit time actually increased when those fires began to grow out of control.

Agents with constrained movement and with neither global vision nor global communications were typically unable to put many fires out before they rapidly grew in intensity. Thus, the damage to the city per unit time typically grew much more quickly than it did for the fully collaborative algorithm and only peaked when the fires began reaching their maximum intensity. The graph of damage vs time for agents working on a 10 fire, 15 agent constrained collaborative problem is shown in figure 6-21; the data was not included figure 6-20 because the damage caused in this scenario was so much greater than that of any of the other scenarios.

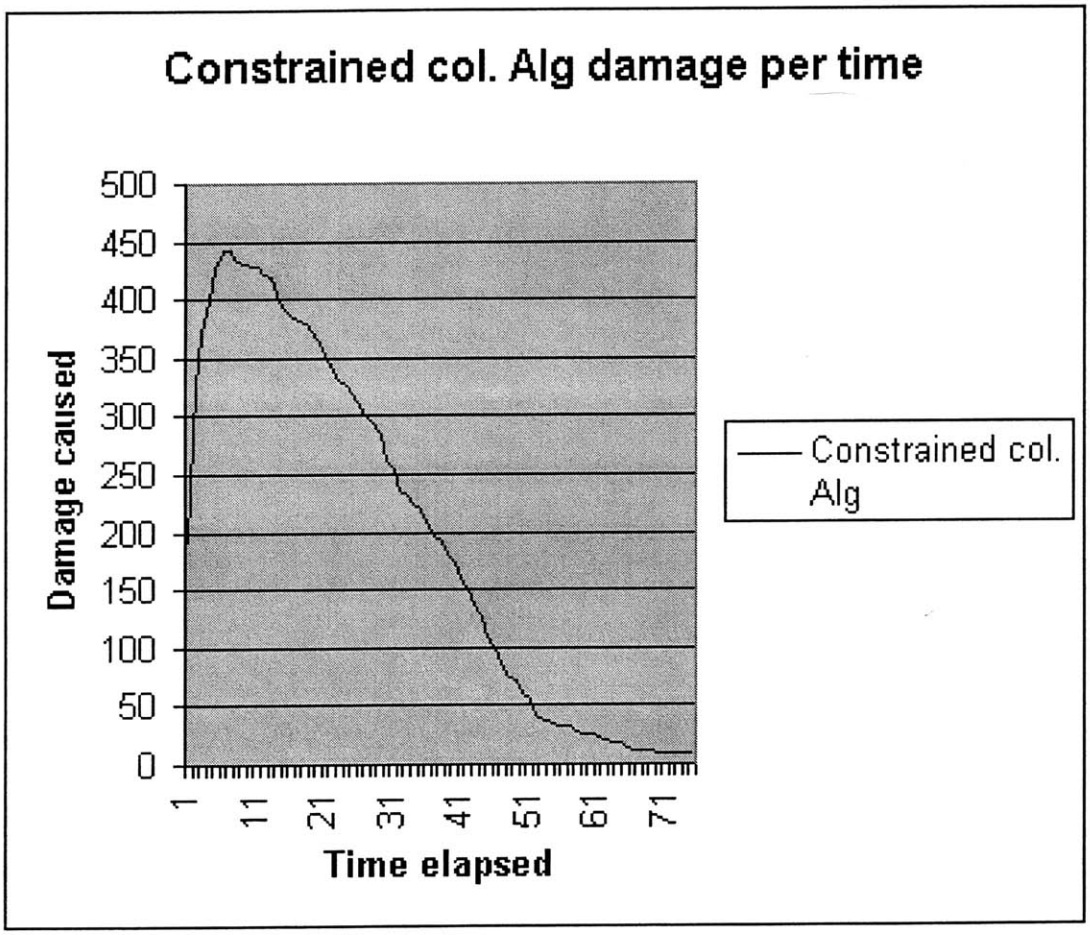

Figure 6-21: Damaged caused as a function of time during a constrained collaborative problem

Chapter 7

Summary and Interpretation of Results

This thesis investigates the effect of various factors on multi-agent collaboration in a simulated fire-fighting domain. A simulator was written that models fires and fire fighting agents in an area of a few square city blocks. Different scenarios were constructed to test the effects of limiting the agents' knowledge and the effects of either coordinating the agents through a single entity or having them make individual decisions. This research demonstrates that the amount of information available to each agent and the agents' ability to act on this information are typically much more important factors than the use of a complex planning mechanism; as long as the agents are aware of each other and take minimal steps to coordinate their actions they are able to achieve results that are nearly as good as those achieved by much more complicated coordination algorithms.

Three algorithms explored scenarios in which agents had global knowledge. Despite the fact that the approximation algorithm uses a very simple greedy agent allocation, it performed almost identically to the optimal algorithm on a variety of cost-free problems and also performed very well

on the travel-cost problems on which it was tested. Agents working without central coordinations also performed well compared to those using the approximation algorithm; despite lacking central coordination the decoupled algorithm was typically able to achieve results within 15 to 20 percent of those obtained by the approximation algorithm.

In contrast, agents working without global knowledge did much worse than those with global knowledge, despite having a slightly more complex coordination mechanism. The agents implementing the fully collaborative algorithm attempted to predict each other's future actions and take this information into account while the agents in the decoupled algorithm simply randomized their choice of fires to fight to avoid making the same choice as every other agent. On the other hand, increasing the vision range of the agents that lacked global knowledge had a fairly large effect in both cost-free problems and travel-cost problems. These agents assume that the area they can see is tiny compared to the total area of the city. Thus, they do not attempt to scale the frequency with which they explore the city based on the portion of the city they can see. As a result, increasing the field of view of agents in the fully collaborative algorithm will never allow them to achieve the same results as were obtained by agents with global vision.

The algorithms that dealt with travel-cost problems were less affected by changes in the agents' vision range than the algoithms designed to solve cost-free problems. The increases in vision allow agents to see fires that are far away, but the agents are penalized for moving to fight those fires. As a result, the gains resulting from additional vision range taper off quickly.

The ability of the agents to act on the information they had was also a key factor in determining the effectiveness of various algorithms. For example, the travel-cost algorithm was much more easily solved when the fires were close together; although the agents had perfect knowledge of the city, the central coordinator was unable to act on this knowledge because of the cost of moving agents from fire to fire.

Similar results were obtained when the agents' fields of view were increased in the constrained collaborative algorithm. Although the increased vision had a large effect initially, the improvement gained from increases to the agents' sight range quickly tapered off. This resulted because the cost to the agents of using this information and switching to a distant fire was so high.

Of the four scenarios investigated in this research, the scenario in which agents had both global knowledge and centralized coordination was by far the easiest to solve. The lack of central coordination caused a signifigant decrease in the agents' ability to put out fires, although the magnitude of this loss decreased as the number of agents increased. Removing the agents' ability to see the entire city resulted in a much larger decrease in fire fighting ability. Unlike the loss of centralized coordination, the performance of agents without global knowledge did not improve relative to those with global knowledge regardless of how many total agents their were, despite there more complex coordination algorithm. Finally, removing the ability of the agents to teleport around the city caused the largest decrease in fire fighting ability, despite the fact that the agents had the most complex coordination mechanism.

Chapter 8

Future work

The purpose of this research was to examine how best to organize and deal with agents with realistic kinds of constraints such as lack of global knowledge and lack of centralized coordination.

There are several possible avenues for future exploration. The hybrid fire model used in the simulator has only four discrete states. The fire model was purposely kept very simple to limit the number of states in the Markov Decision Process; the run time of the value iteration algorithm grows with $(M^N)^3$ where M is the number of possible states in the fire and N is the number of fires. However, the run times of the approximate, decoupled, fully collaborative and constrained collaborative algorithms do not grow with the number of possible states in the fire, they grow with the product of the number of fires and the number of agents. Testing out the algorithms with a more complicated fire model would allow for more thorough verification of the results presented here.

The effectiveness of the approximation algorithm is due largely to the specific characteristics of the hybrid fire model used in this simulation. A more generalized approximation could be developed using neural networks or other function approximation techniques to approximate the behaviour of the MDP-driven optimal solution.

There was no optimal basis against which to compare the constrained collaborative algorithm's performance. Future work might involve developing a centrally coordinated version of the constrained collaborative algorithm that used global knowledge. This algorithm could serve as an "upper bound" against which to measure the performance of the constrained collaborative algorithm. A centrally coordinated version of the algorithm would allow agents to more effectively discover all of the fires in the city and would ensure that agents move to the most useful fire when the fire they are currently fighting is put out.

Many of the benefits of central communications might also be found in a hierarchical organization. In addition, a flexible hierarchical organization would require less communication than single-point central coordination and thus would be less vulnerable to a failure in a single entity. However, it would still provide better interagent coordination than the distributed approaches discussed in this paper. A hypothetical organization of this type might entail each agent attempting to solve its own local goal by dividing the task into several parts for each of its subordinates to handle; agents that have not received a goal from their superiors might simply default to some preselected goal (such as making contact with other agents) or might attempt to determine a useful course of action dynamically.

Chapter 9

Appendix A - Markov Decision Processes

Although creating optimal behavior in a group of agents is a complex task, creating optimal behaviour in a single agent (such as a centralized coordinator) is fairly easy. One common method for determining the optimal behaviour for a single entity is a Markov Decision Processes (MDP). Solving an MDP involves mapping every possible state of the world to the best possible action the entity can take in that state; both the immediate reward for a given action and the possible states that are likely to result because of that action are taken into account when an action is considered.

The definition for a Markov decision process used here closely follows the one used in work done by Littman et al. [14].

A Markov decision process is defined to be a four-tuple of (Ω_S, Ω_A, p, c) where Ω_S is the state space, Ω_A is the action space, p is the state-transition probability distribution function and c is the instantaneous-reward function [14].

The state-transition function indicates the probability of reaching a particular state j given

some action A taken in state i:

$$p_{ij}^{A} = Pr(S_t = j | S_t - 1 = i, A_t = k) \tag{9.1}$$

where S_t indicates the state at time t and A_t denotes the action at time t.

A Markov decision process must satisfy the Markov property, which states that the current state of the world is completely understood without knowledge of past actions and/or their results.

A Markov decision problem is defined as a Markov decision process along with an instantaneous reward function. The reward function computes the reward of performing a given action in a given state. This allows the total reward for any particular state to be computed for a given policy. One example of a Markov decision problem is the following:

A robot is standing in the middle of a floor. The floor is divided up into a grid with the robot at one particular location. The robot wishes to reach a different location and can only move in certain, discrete ways (eg: one unit north, one unit west, etc.). There is a fixed reward associated with moving from one location to another, although in this case the "reward" would most likely be negative (the cost of moving between the cells in the grid).

This problem obeys the Markov property because the state of the world is determined based entirely on the current position of the robot; the world state is not affected by previous locations the robot has been in or previous actions it has taken. A solution to a Markov decision problem creates a policy which maps actions to states. In the robot example used above, the policy would assign a particular movement to each location; whenever the robot was at any particular location it would perform the movement associated with that state.

An optimal solution to an MDP provides a policy that, when followed, allows the goal state to be reached with the least possible total incurred cost. Some MDPs involve more than one goal

state, possibly with varying weights associated with each of them. In this case, the optimal solution defines a set of actions that maximizes the expected reward for any state.

9.1 Value Iteration

Many algorithms have been written to solve Markov decision problems [3, 13]. This research uses the value iteration algorithm devised by Bellman in 1957. This algorithm iterates through the entire state space and calculates the value of each state along with the best action to take in each state. For every iteration, each state is updated as follows:

$$V_{t+1}(i) = \max_{A}[\sum_{S_t}(r(i_t, j_t) + dV_t(j))Pr(j_t|i_t, A))] \tag{9.2}$$

where $V(i_t)$ is the value of state i at time t, d is the discount factor $(0 < d < 1)$, max_A is the maximum over the action space, $r(i_t, j_t)$ is the reward of going from state i to state j and $Pr(j_t|i_t, A)$ is the probability of making a transition to state j from state i at time t given action A.

Each state's value is initialized to zero, then during each iteration of the algorithm the value of the state is updated based on the reward that the robot would receive for moving to other states in view of the likelyhood that a given action would result in the robot moving to a given state. This continues until the change in value of each state falls below some threshold. The algorithm keeps track of the action that produced the greatest value for each state; when the algorithm has terminated, the set of *(state, action)* pairs that were created during the last iteration comprises the solution to the MDP.

The value iteration algorithm often takes a very long time to converge to a single set of values and, depending on the problem, may never converge at all. If the algorithm is simply run for a fixed number of iterations and stopped it is possible that the MDP will not count events beyond

its "horizon". For example, if value iteration is run for 75 iterations and then stopped, it might produce a policy that provides a robot with very good behavior for 75 iterations but leaves the robot with no fuel for the 76th iteration.

Rather than compute the value iteration algorithm out to a fixed horizon, this research uses discounting to ensure that the value iteration converges. Discounting involves multiplying the changes in value by an ever decreasing discount constant. Rather than creating an MDP that simply assumes the simulation will stop after a certain point (in the above example, the MDP assumed that the simulation would only last 75 iterations), discounting reflects an *a priori* probablilty that the simulation will be stopped at any given time step. Although an MDP solved with discounting may produce a slightly different result than an unaltered MDP would, it is also guaranteed to converge to a single set of values and typically does so much more quickly than the non-discounted MDP would have.

Bibliography

[1] R. Ahuja, T. Magnanti, and J. Orlin. *Network Flows.* Prentice Hall, 1993. pages: 309-310.

[2] S. Andrade. *An Intelligent Agent Simulation Of Shipboard Damage Control.* Master's thesis, Brazilian Naval Academy, 2000.

[3] L. Baird. Residual algorithms. In *12th International Conference on Machine Learning*, 1995. Tahoe City, Morgan Kaufman.

[4] A. Charnes and W. Cooper. The Stepping Stone method of explaining linear programming calculations in transportation problems. *Management Science*, 1:49–69, 1954.

[5] P. Cohen, M. Greenberg, D. Hart, and A. Howe. Trial by fire: Understanding the design requirements for agents in complex environments. *AI Magazine*, 10(3):32–48, 1989.

[6] D. Eustace, D. Barnes, and J. Gray. Co-operant robots for industrial applications. *IEEE International Conference on Industrial Electronics and Control*, pages 39–44, 1993.

[7] W. Frandsen. Fire spread through porous fuels from the conservation of energy. *Combustion and Flame*, 16:9–16, 1971. Elsevier Science Inc.

[8] F. Fujioka. Simulating fire spread as spatial stochastic process. In *13th International Conference on Fire and Forest Meteorology*, Agricultural Network Information Center, Lorne Australia, 1996.

[9] Z. Galil and E. Tardos. An $O(n^2(m+nlogn)logn)$ min-cost flow algorithm. In *Journal of the ACM*, volume 35, pages 374–386, April 1988.

[10] R. Howard. *Dynamic Programming and Markov Processes.* MIT Press, Cambridge, 1960.

[11] H. Kitano. Research program of robocup. *Applied Artificial Intelligence*, 12(2), 1998.

[12] H. Kitano, S. Tadokoro, I. Noda, H. Matsubara, T. Takahashi, A. Shinjoh, and S. Shimada. Large-scale disasters as a domain for autonomous agents research. In *IEEE International Conference on Systems, Man and Cybernetics*, volume VI, pages 739–743. Tokyo, 1999.

[13] D. Koller and R. Parr. Policy iteration for factored MDPs. In *16th Annual Conference on Uncertainty in AI.* Association for Uncertainty in AI, 2000.

[14] M. Littman, T. Dean, and L. Kaelbling. On the complexity of solving Markov Decision Problems. In *Eleventh International Conference on Uncertainty in Artificial Intelligence*, Morgan Kaufmann Publishers, Inc, San Francisco, 1995.

[15] M. McCarthy. *Fire Modelling and Biodiversity.* Johnstone Centre, London, 1997.

[16] N. Meuleau, M. Hauskrecht, K. Kim, L. Peshkin, L. Kaelbling, T. Dean, and C. Boutilier. Solving very large weakly coupled Markov Decision Processes. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, AAAI, Madison, Wisconsin 1998.

[17] M. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* Wiley, New York, 1994.

[18] R. Rothermel. A mathematical model for predicting fire spread in wildland fuels. *In USDA Forest Service Resource*, 10, 1972.

[19] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley and Sons, 1998.

[20] S. Tadokoro, H. Kitano, T. Takahasi, I. Noda, H. Matsubara, A. Shinjoh, T. Koto, I. Takeuchi, H. Takahasi, F. Matsuno, M. Hatayama, M. Tayama, T. Matsui, T. Kaneda, R. Chiba, K. Takeuchi, J. Nobe, K. Noguchi, and Y. Kuwata. An approach of AI and Robotics to the emergency response problem in disaster. In *4th International Conference on MultiAgent Systems, RoboCup-Rescue Workshop*, 2000.

[21] T. Tokuyama and J. Nakano. Efficient algorithms for the Hitchcock transportation problem. In *SIAM Journal on Computing*, volume 24, 1995.

[22] J. Varriano. Boltzman distribution.

URL: http://www.cbu.edu/ jvarrian/447/447Boltzman/, Christian Brothers University, Department of Physics, last modified: 3/99.

[23] W. Zhao. *Multiple Autonomous Vehicle Mission Planning and Management*, T-1349. Master's thesis, Massachusetts Institute of Technology, 1999.

Index