EINDHOVEN
UNIVERSITY OF
TECHNOLOGY

A method for designing decision support systems for operational planning

Citation for published version (APA):
Eiben, A. E. (1991). *A method for designing decision support systems for operational planning*. [Phd Thesis 1 (Research TU/e / Graduation TU/e), Mathematics and Computer Science]. Technische Universiteit Eindhoven. https://doi.org/10.6100/IR354562

DOI:
10.6100/IR354562

Document status and date:
Published: 01/01/1991

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 04. Oct. 2023

A Method for Designing Decision Support Systems for Operational Planning

A. E. Eiben

A Method for

Designing Decision Support Systems

for Operational Planning

A Method for Designing Decision Support Systems for Operational Planning

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof. dr. J.H. van Lint, voor een commissie aangewezen door het College van Dekanen in het openbaar te verdedigen op dinsdag 2 juli 1991 om 16.00 uur

door

ÁGOSTON ENDRE EIBEN

geboren te Budapest

druk: wibro dissertatiedrukkerij, helmond.

Dit proefschrift is goedgekeurd door de promotoren:

Prof. dr. K.M. van Hee

en

Prof.dr. E.H.L. Aarts

Acknowledgements

First of all, I am grateful to Kees van Hee for accepting me as a PhD student back in 1987, as well as for his valuable ideas and suggestions concerning my work.

Similarly, I am grateful to Emile Aarts who has contributed a great deal to the conceptual maturity and the clarity of this thesis.

Many thanks go to Wim Nuijten for his stimulating participation in discussions that have always resulted in a better understanding of the subject.

I also want to thank Arjan de Vet for helping me by reading drafts of my thesis.

I owe special thanks to Fem du Buisson, without whom I would never have gotten through the administrative difficulties of university life.

This research was carried out as an NFI project funded by the Dutch Organization for Scientific Research (NWO). I thank the NWO and the Eindhoven University of Technology for providing all the facilities needed for a successful study.

I wish to express my gratitude to my parents who gave me the first impulses for choosing a scientific career and kept me from leaving the path too early.

Finally, I am more than grateful to my wife Berenice for her patience in times of stress.

Contents

Introduction 1

1 Decision Support Systems 6
1.1 Brief Overview of DSS History and Literature. 6
1.2 Operational Decision Making 8
1.3 Our View on Decision Support Systems 11
1.4 Making Decision Support Systems: DSS Shells, DSS Generators 14

2 Planning 19
2.1 Model of Planning Problems. 20
2.1.1 Static Case: the World as a State 21
2.1.2 Dynamic Case: the World as a Process. 32
2.1.3 The Role of Time. 41
2.2 Examples of Planning Problems 51
2.2.1 Travelling Salesman Problem 53
2.2.2 Travelling Salesman Problem with Time Windows. 55
2.2.3 Precedence Constrained Scheduling Problem. 58
2.2.4 Time Table Problem 61
2.2.5 Ship Loading Problem. 65

3 Search Problems 71
3.1 Model of Search Problems 75
3.2 Relationship between Planning Problems and Search Problems 81
3.3 Examples of Search Problems. 88
3.3.1 Travelling Salesman Problem 88
3.3.2 Precedence Constrained Scheduling Problem. 90
3.3.3 Time Table Problem 91
3.3.4 Ship Loading Problem. 93

4 Search Procedures 94
4.1 Space Search, Graph Search, Local Search 94
4.2 The General Search Procedure 98
4.3 Examples of Search Procedures 109
4.3.1 Genetic Algorithms 109
4.3.2 Simulated Annealing 110
4.3.3 Threshold Accepting, Hill Climbing 111
4.3.4 Depth First Search 111
4.3.5 Breadth First Search 112
4.3.6 Best First Search 112
4.4 Convergence of Stochastic Search Procedures 113

5 Searching by Generalized Genetic Algorithms 129
5.1 Multiparent Production 133
5.2 Multidimensional Genotypes 136
5.3 Selection, Reduction and Evaluation 139

6 Towards a Software Tool for DSS Design 142
6.1 Problem Definition Component 142
6.2 Solution Method Definition Component 147

7 Final Remarks 152

References 156

Samenvatting 162

Introduction

The research described in this thesis constitutes the first phase of a project that is concerned with decision support systems for operational planning. Briefly summarized, the goal of the project is to develop generally applicable software tools to facilitate decision support system design. To give the context of our investigation let us begin with an overview of our motivations.

Operational decision problems, also called operational planning problems, frequently occur in business environment, in particular in production or distribution processes. Regardless of what operational decision problems exactly are, we can already remark that solving such problems requires substantial computational efforts, thus seeking for computer support to manage these problems is a straightforward idea. Software systems providing this support are called decision support systems (DSSs). From the seventies on there were many projects in DSS development directed at different decision problems, following different approaches and resulting in different DSSs. Despite of the diversity of these projects there are several general observations to make about DSS development. Let us here mention the following ones.

a) Developing a DSS for a specific operational decision problem is a time and money consuming activity that is repeated for every specific decision problem.
b) DSSs are mostly relying on Operational Research as theoretical and practical background.
c) The DSSs developed for different operational decision problems exhibit architectural and functional similarities. Apparently there is an underlying common structure of DSSs that might be investigated and made explicit.

It is also a remarkable fact that compared to the number of decision situations in practice there are only few decision support systems that operate supporting a real decision problem.

In this thesis we perform a theoretical investigation to confirm the feasibility of a general method for developing DSSs. The essence of our approach is to make a theoretical model of DSSs and use this theoretical model as a blueprint when constructing a DSS. We also have another intention with a theoretical model: gaining a good insight in the field, explaining related phenomena, and last but not least providing a clear terminology that facilitates further discussion and research. We maintain two requirements with respect to our model.

a) It is general enough to be able to model many different DSSs, that is it should have a broad application domain.
b) It is sufficiently detailed so as to provide a 'high resolution' view of DSSs. This makes it possible to use this model as the basis of constructing DSSs.

Such a model embodies a theoretical skeleton of decision support systems. Using such a general skeleton as a guideline can make DSS development systematic, thus probably less time consuming and erroneous than it is nowadays. This skeleton can also serve as the theoretical basis of a generic software tool that supports DSS development.

Notice that the above requirements are somewhat counteracting each other. On one hand, stressing generality we may loose 'high resolution', i.e. detailed view. Maintaining little structure within a model means putting a few restrictions on the application domain but this often leads to a vague understanding of the modelled phenomenon. Furthermore, if our model gives a 'low resolution' image of DSSs, then there would be a large gap between theory and implementation, i.e. the blueprint would be too rough to use. On the other hand, if we make a highly detailed model with a rich structure then we necessarily include many assumptions about the application domain. Such a detailed model facilitates implementation but by having made many assumptions we may essentially reduce the domain of application.

We are trying to solve this contradiction between our goals by making general models with parameters, that is certain variables with unspecified values. In addition, we want to use high level parameters, i.e. parameters the value of which can be an expression in a high level language with great expressive power. In

such a case the rigidity of using the same model for many different decision problems would be counterbalanced by the flexible parameters that are able to incorporate various information.

Notice that by our approach constructing a decision support system would be reduced to specifying problem dependent values for the parameters of the general framework. Using a DSS development tool based on an abstract DSS model with parameters, DSS development would become 'simple' instantiating, that is supplying input values of the parameters of the tool. In such a way DSS development would require less effort than having to design the whole DSS. The first phase of our project - and hence this thesis - is meant to establish a formal basis for this approach.

We remark that the approach sketched above is deviating from the common DSS approach. According to the commonly practiced method one mostly develops a problem specific decision support system that is applicable under tight conditions only. The software we are aiming at is applicable to a broad range of problems and is flexible by its parameters. Namely, if conditions in and around the decision problem change we can suit our system to the changes by redefining the parameters. The price of this flexibility is that our DSSs will be probably less efficient than those based on a tailored mathematical programming method. This inefficiency, however, should be seen in the light of two other factors. First, a system can be relatively inefficient but still satisfactory if computation times remain within acceptable limits. Second, the development of a sophisticated, highly problem suited system is mostly very expensive which can make it unattractive. As the reader may have already realized, our approach towards DDSs shows certain features that are mostly associated with Artificial Intelligence (AI). In particular, a highly parameterized system, the use of a language with great expressive power are mostly AI attributes. We admit that indeed we are trying to pass the traditional borders of DSS research and study the feasibility of an AI-like methodology for DSSs.

We understand that the modelling-and-instantiating approach has its limitations. No matter how sophisticated our model is, there might be situations where it cannot be applied. This is the case if, for instance the intrinsic structure of the model we give is inappropriate for describing the given situation, or for handling

the situation such problem dependent knowledge should be used that cannot be expressed by our parameters. Nevertheless, if we carry out the investigation thoroughly, it will be clear in advance where we can rely on the general model and what are the points in a DSS where problem specific heuristics are preferable.

During the development of our models we need to make certain choices, assumptions that influence our results. Throughout this thesis we are making these choices explicit. This provides the possibility of making other choices, hereby it helps to choose other courses of investigation.

Our first restriction is that we concentrate on automated decision making within a DSS, disregarding for example interfacing, data and model management aspects. This determines two main subjects of investigation: the problems to be solved by a DSS and the problem solving methods used by a DSS. Keeping these two issues apart we get the explicit freedom to study and apply more solution methods to the same class of problems or to investigate the application domain of a certain problem solving method.

This thesis is organized as follows. In Chapter 1 we give an overview of decision support systems and describe how a generic DSS model - a DSS skeleton - is related to software tools, such as a decision support system, a decision support system shell or a decision support system generator.

In Chapter 2 we develop a formal model of operational planning problems where time is explicitly involved and we distinguish static and dynamic cases depending on the role of time. To test the applicability of our formalism we describe Travelling Salesman Problems (with and without time windows), Precedence Constrained Scheduling Problems, Time Table Problems and Ship Loading Problems in terms of the model.

In Chapter 3 we briefly discuss three global problem solving paradigms: search, automated reasoning, mathematical programming and we choose the search paradigm for further elaboration. We investigate search problems and define how can they be considered as a representation form of planning problems. We introduce a standard manner to transform an arbitrary planning problem to a search problem.

In Chapter 4 we develop a model of search that incorporates space search, graph search and local search explaining their relationship. We define a General Search Procedure (GSP) and describe Genetic Algorithms, Simulated Annealing,

Threshold Accepting, Depth-first Search, Breadth First Search and Best First Search as subtypes of our GSP. For stochastic optimization procedures we prove convergence properties within our model.

Chapter 5 is devoted to genetic algorithms. Beyond the advantage that they generalize other search methods, such as includes simulated annealing, threshold accepting or hill climbing, genetic algorithms show a reasonable performance on a wide class of problems and they can be easily adapted if the problem in question changes. We make a generalization of genetic algorithms and we obtain a type of search procedures where problem dependent (heuristic) components can be clearly located. Hereby we believe to reach a good balance, that is a widely applicable search procedure that is detailed enough to support designing problem oriented instances of it.

In Chapter 6 we give the outlines of a generic software tool facilitating DSS development relying on the previously given models. By the results of Chapter 2 we can sketch the problem definition component based on a language in logical fashion. Based on our view on search, the definition of a problem solving method requires the definition of the constituents of the GSP. Some of these constituents can be defined such that they are applicable for many problems. This reduces the definition of a search procedure to the definition of those components that require problem dependent knowledge, heuristics.

CHAPTER 1

Decision Support Systems

When willing to set guidelines for designing decision support systems, the first straightforward question one has to answer is: what are DSSs? In this chapter we are trying to give our answer to this question.

1.1 Brief Overview of DSS History and Literature

To answer the question about what a DSS is let us have a brief look upon their history. The name decision support system was first used by Gorry and Scott Morton (1971) and has made quite a career since then. Most of the authors of the field, however, do not give a clear definition of what they mean by this term, as it is observed by Sol (1985). Historically, DSSs originate from Electronic Data Processing (EDP) on the practical side, while their theoretical backgrounds lay in Operational Research (OR). In Sol (1985) we find a short description of the software evolution that has lead to DSSs from EDP through Management Information Systems (MIS) in business environments, cf. Burch and Strater (1974), Naylor (1982).

When it comes to the definition of a DSS there are at least two ways to do it: specifying the functions of a DSS or giving a description of its components. The most frequently quoted definition from Keen and Scott Morton (1978) belongs to the first type stating that a DSS

- assists managers in their decision processes in semi structured tasks;
- supports, rather than replaces, managerial judgment;
- improves the effectiveness of decision making rather than its efficiency.

Besides such broad definitions, cf. also Alter (1980), Keen (1986), there are more specific ones like that of Anthonisse, Lenstra and Savelsbergh (1988) who identify DSSs as interactive planning systems. Van Hee and Lapinski (1988) specify a DSS as a system assisting managers in the control of a business process. The functions of a DSS they distinguish are the following:

- performing data management functions;
- evaluating decisions proposed by the user;
- generating decisions satisfying some user defined conditions.

Observe that this definition puts up strong requirements about a DSS. According to this view a DSS can be told about a decision and it can make a decision. On one hand, this leads to a more restricted notion of a DSS than usual, on the other hand it has a big advantage: it is specific enough to be used to decide whether a given software system is a DSS or not.

As for the components of a DSS there is no universally accepted view either. Sprague (1980) distinguishes a data base management system, a model base management system and a user interface called the dialog generation management system within a DSS. Bonczek, Holsapple and Whinston (1981) envisage a language system, a knowledge system and a problem processing system, while the system analysis of Sprague and Carlson (1982) yields four entities for representations, operations, memory aids and control mechanisms.

From the software point of view DSSs are intended to be user friendly and interactive programs. The methods they apply, their architecture and the underlying philosophies are diverse, although an observation in Verbeek (1990) is remarkable: the majority of the literature on DSS belongs to the field of Operational Research. For several authors, e.g. Savelsbergh (1988), DSSs are but an "approach towards the practice of operations research", and even in the abstract framework for research on decision support systems Sprague (1980) mainly considers models of the equational type and methods of optimization based on linear, dynamic or stochastic programming. Recently, another paradigm, Artificial Intelligence (AI) is entering the field of DSS. Bonczek, Holsapple and Whinston (1983) and Van Hee and Lapinski (1988) consider incorporating Artificial Intelligence methods into DSS; the approach of Eiben and van Hee (1990) has a

strong AI accent as well.

Throughout the development of Artificial Intelligence, cf. Nilsson (1982), Winston (1984), Shapiro and Eckroth (1987), many important notions were introduced. Here we mention two important contributions to the theory and practice of computing science: the notions of knowledge representation and symbolic computation. The main lesson we have learnt from knowledge representation, cf. Brachman, Levesque and Reiter (1989), is that the same abstract knowledge can be formulated and stored in entirely different ways, e.g. by equations, formulae, logical frames. Symbolic computation is mostly understood as an alternative to classical numeric computation that is typical for OR methods, in particular mathematical programming. From the "application of theorem proving to problem solving", Green (1969), it has led to using logic as a language for computation, Kowalski (1974), and to logic programming, Lloyd (1987). Nowadays there are many working software systems that are based on automated logical reasoning; the best known members of this family are the so called expert systems, see Waterman (1986). Another important AI feature is the separation of domain knowledge and computation mechanism. This separation advances flexibility in defining and modifying the problem at hand or the applied problem solving method. We believe that incorporating AI methods into DSS research and practice broadens the scope of DSSs and helps better understanding and exploiting of problem solving heuristics in decision support.

1.2 Operational Decision Making

To present our view on decision support systems let us first specify what we mean by a decision situation: in a decision situation a decision maker has to (re)act in an environment in order to preserve or achieve certain conditions. A decision is thus a control action of the decision maker that is meant to influence the environment. In general one can distinguish three classes of decision situations: those concerning strategic, tactic and operational issues.

Example 1.1
When setting up new factories we encounter decisions at different levels. A strategic decision is e.g. to build four factories in four different countries. Such a

decision is to meet very general requirements, like that of being less dependent of local troubles, for instance of natural disasters or political changes. To make such a decision presumes awareness of the phenomenon of 'local trouble' and requires ability of estimating its likelihood. A strategic decision has a long term effect; since complete factories cannot be moved without substantial effort they will probably remain at their locations for decades.
A tactical decision is to determine whether to install assembly lines or flexible production cells in each factory. Choosing between the two can rely on better formulated goals and more solid knowledge than in a strategic issue. Think of the fact that assembly lines are appropriate for a mass production, while flexible production cells are more suitable to order oriented production. Such a decision is easier to withdraw but still at high costs, so it probably will not be reconsidered for years.
Operational decisions need to be taken daily or weekly, for example to determine what to produce to satisfy the costumers orders. Such decisions are triggered by rather strictly formulated goals, e.g. we need to deliver 5000 of item number 2 tomorrow, and might be made day by day.
□

The above classification is of course not strictly formal; whether or not a decision is strategic or tactic is somewhat arbitrary. Nevertheless, in strategic and tactic decision situations there are so many factors to take into account and such an extent of uncertainty that every attempt to model them formally has serious limitations. Therefore, we only deal with operational decision problems along this study, that is we restrict ourselves to problems where
- decisions have a short term impact (several hours to several days);
- a sound model of the decision situation can be given.

Such a restriction about the application domain of a DSS seems to guarantee that we can handle problems within this domain and provide sufficient support to the users. This is, however, a hasty conclusion since even these simplified situations can lead to formal models that yield mathematically intractable problems, cf. Garey and Johnson (1979).

Example 1.2
To illustrate an operational decision problem let us take a time table problem in a

school. The data model of the decision environment consists of the description of the relevant objects under consideration and the relationships between these objects. For instance

- classes, that is groups of students;
- subjects such as mathematics, geography, English, etc;
- teachers;
- classrooms;
- lecture hours;

can be the objects given and the corresponding relations can tell which teacher is qualified to teach mathematics, how many English lessons do the classes need per week, etc.

Furthermore we can formulate conditions that need to be satisfied, e.g.

a) In any classroom at any time there is at most one teacher teaching one subject.

b) The same class should not get the same subject three times a day.

c) All lessons on the same subject for the same class should be given by the same teacher.

Such conditions are called constraints, they are either satisfied or violated and therefore are qualitative.

The goal in this decision situation is to make a weekly schedule for the school such that each group gets every subject it needs in a week. To satisfy this goal a decision maker has to make elementary decisions, i.e. assignments of classes, subjects, teachers, classrooms and lecture hours and has to compose a correct and complete time table from such assignments. A correct time table satisfies all the constraints, a complete time table has all the lectures scheduled, that is each group gets each of the needed subjects.

Besides constraints there can be criteria given. A criterion is a quantitative measure that rates a certain feature, e.g. the amount of idle lecture hours of a class. Criteria are often subjects of optimization, that is one can be aiming at decisions that realize the lowest or highest possible value due to a certain criterion. In a school we might prefer time tables that minimize the number of idle hours of each class. Criteria can also be used to enforce constraints by measuring the rate of violation of a certain constraint. For instance, a modelling decision can be that we delete constraint (b) above, add a criterion that measures how concentrated a subject is scheduled and we aim at a time table that keeps the

value of this criterion low.

□

Since clusters of decisions are often called plans we also use the term planner for a decision maker. On the same grounds we refer to our application domain as operational decision making or operational planning.

It is an important factor in our world view that within the class of operational decision problems we distinguish various problem types. By a problem type we mean a group of problems that are of the same basic character, for instance routing problems or scheduling problems, see Lawler et al. (1989). Strictly speaking, one can consider the notion of *problem type* in two different ways. One possible view is to see a type as the set of all problem instances declared to belong to it. Another way is to consider a problem type as the abstract framework fixing the major outlines of a problem but still having parameters i.e. free variables with unspecified values. Problem instances belonging to a type can be obtained by giving values to these parameters.

It is important to notice that the border between different problem types is arbitrary. For example, we can recognize a crucial difference between a travelling salesman problem and a chinese postman problem, see Garey and Johnson (1979), and describe them as two types. Nevertheless, we can also model them such that they are basically of the same character thus forming subtypes of the same problem type.

1.3 Our View on Decision Support Systems

Having discussed where DSSs can be applied let us give our view on what they are. With respect to their functions we maintain the view presented by Van Hee and Lapinski (1988), Eiben and Van Hee (1990). For maximal clarity let us repeat that in a DSS we distinguish the following basic functions:

- performing data management functions;
- evaluating decisions proposed by the user;
- generating decisions satisfying some user defined constrains and scoring high by possibly given criteria.

Example 1.2 continued
A DSS for time tables can support its user by simply displaying an actual time table or by facilitating the lookup of the lectures of a certain teacher. Such activities fall under the data management functions mentioned above.
Enabling the user to make changes on the actual time table the system can compute and display the effects of these changes, declaring a new time table correct or incorrect according to the given constraints, or calculating the corresponding value of a criterion, e.g. the number of lessons not yet scheduled.
The third, and most sophisticated function mentioned above is that of generating time tables automatically. In this case the DSS computes a complete and correct time table by itself or improves a certain partial time table given by the user.
□

In this thesis we do not consider data management and other related issues (such as user interface, man-machine interaction); here we focus our attention on generating decisions.

Besides the question what a DSS does, there is of course another one: how well it is doing it. There are several quality measures of a DSS. The most frequently considered ones are effectivity, efficiency, robustness and flexibility. Effectiveness is a measure of the obtained solutions with respect to some evaluation criteria, it is the degree of fulfillment of wishes regarding a solution, cf. Verbeek (1990). Robustness concerns the sensitivity of the system, i.e. how sensitive the solutions are for changes in data. Efficiency is the speed rate of the computation, measuring how fast the solutions can be obtained. Here we may distinguish the net speed, the computation time of the DSS, and gross speed regarding the time used by the man-machine combination. Last but not least, flexibility concerns the efforts needed to adapt the DSS to structural changes in the planning environment or changes in the priorities of the planner. These features are not independent, there is for instance a well-known counterbalance between efficiency and flexibility. One of the basic premises of our approach is that we prefer the latter, rather having a highly flexible system applicable to a wide range of problems, than an efficient tool for only a narrow application domain.

Example 1.2 continued
It is typical for practical decision situations that the environment is changing over and over again. In a school it could mean that some teachers become ill or that the management begins to prefer fewer idle hours of classes to well spreading of subjects. This means that the constraints can vary or the criteria may change, which requires that the user can adapt his system easily.
□

With respect to the components of a DSS we basically distinguish three of them: a communication component, a problem description component (PDC) and a problem solving component (PSC). Since we are primarily interested in automated decision generation we direct our attention to the problem description and the problem solving components. Both the PDC and the PSC must be able to receive information from the user. When filled up the PDC should contain a description of the decision or planning situation and the actual problem to be solved. Given the necessary inputs to the PSC it should contain a problem solving method that is suitable to handle the problem specified by the PDC.

Example 1.2 continued
The PDC of a DSS for time table problems should contain the sets of lectures, classrooms, teachers and lecture periods together with the basic relationships, e.g. the qualifications of the teachers. Also the elementary decisions should be given and the construction rules that define how to make a time table from them. Naturally, the constraints determining the correctness of a time table must be represented in the PDC too. At last, the specification of possible criteria belongs to the PDC as well.

We can not say much about the PSC at this point since the way of problem solving is not determined by the problem. For time tables we can apply different solution methods, from a classic OR method, see Even, Itai and Shamir (1976), to genetic algorithms, cf. Colorni, Dorigo and Maniezzo (1991).
□

1.4 Making Decision Support Systems: DSS Shells, DSS Generators

Now that we have a certain view on what a DSS is, let us consider the question of how to make one. Sol (1985) recommends to investigate decision support system generators as DSS design environments that "bridge the gap between general tools and specific DSSs". Such general tools are software systems meant to reduce the efforts and costs of making a DSS. The main assumption behind using such tools is that DSSs have common general features conjoined by application specific ones. Distilling the common features we can construct a theoretical skeleton of DSSs where the application specific constituents are absent. Such a skeleton can be seen as a frame containing parameters where application specific information can be incorporated by specifying values for the parameters. Such a skeleton or frame can be used in two different ways to facilitate DSS development. The first possibility is to build a so called DSS shell, the second one is to build a DSS generator.

A DSS shell is an implementation of the abstract DSS skeleton. Giving specific values to the parameters the shell becomes instantiated, thus by furnishing the shell with application specific information we obtain a complete DSS.

Example 1.2 continued

A DSS shell to support the design of a DSS for time table problems can contain a subshell to specify the problem to solve. Within this problem description subshell we may distinguish further components, for instance:

- a component for data modelling to define the objects under consideration together with their relationships (e.g. the sets of classes, classrooms, teachers and lecture periods and the qualifications of the teachers, weekly needs of classes, etc.);
- a component for constraint description that needs to be filled with the application specific constraints that have to be satisfied.

Goal specification and construction rules of time tables can be hard coded in the system, since they are universal for all time table problems.

The above items can be envisaged as parameters of the problem description subshell. For instance, the object specification sub-subshell may have 5 parameters: G, S, T, C, H standing for the set of groups, subjects, teachers,

classrooms and lecture hours, respectively. This five tuple embodies type specific information about time table problems. To describe a particular time table problem we need to instantiate the shell by giving values for these parameters, i.e. a set of groups, a set of subjects, etc.

□

Let us remark that the notion of a shell is not restricted to DSSs. A system shell is generally meant as an implemented skeleton where only the frame of the whole system architecture is fixed, many parameters are unspecified. These parameters have to be given values in to obtain a complete system. According to a refined view not any system that has parameters is recognized as a shell (think of a program with input values), a shell is mostly seen as having high level parameters. Here we encounter a crucial issue about shells: whether or not a system is a shell depends on the level of its parameters.

Analyzing the basis of high and low level division of information one can notice a strong correlation with the border between problem type and problem instance. Namely, information that defines a particular problem instance within a given problem type is seen as low level one. Information, the modification of which leads to another problem (sub)type is seen as high level information. For easy reference to these two kinds of information let us make a convention using the term data for low level and knowledge for high level information.

The second way of making use of an abstract DSS skeleton is to build a DSS generator based on it. A DSS generator is also based on an implemented DSS skeleton, its input is knowledge specifying a problem type and a solution method type, its output is a DSS as executable software.

Observe the relation between the three notions we use: DSS, DSS shell, DSS generator. We see a DSS as a system with low level input parameters that can only receive data such as the number of classes, etc. These data define a problem instance and an instance of a solution method; having these two defined the DSS can make decisions as output. A DSS shell is a system with both high and low level parameters; incorporating knowledge through the high level parameters the shell becomes a DSS. A DSS generator has only high level parameters receiving problem specific knowledge as input and producing a DSS as output. Notice that the difference between a shell and a generator resembles the difference between

an interpreter and a compiler for a formal language.

After making these distinctions we can be more specific about flexible systems mentioned in the introduction. As it turns out from the foregoing we maintain the vision of having DSSs with low level parameters. These parameters are rigid, in the sense that setting them to new values leads only to instantial changes, thus we cannot expect great flexibility in a DSS. On the other hand DSS shells and DSS generators possess high level parameters for incorporating knowledge. Changing such knowledge (e.g. incorporating new constraints) changes the structure of the problem not only the instance. Thus, this is the level where flexibility can be included, that is strictly speaking we are not aiming at flexible DSSs but at flexible DSS tools - shells and generators.

A big advantage of using general tools like a DSS shell or a DSS generator is that it reduces the efforts and time of DSS design. The shortcoming of such an automated development is that the class of DSSs we can make is previously determined by the tool. Namely, DSS shells and DSS generators can have wide or narrow application domains. This depends on the application domain of the DSS skeleton they are based upon, and after all upon whether the abstract DSS model used is general or not.

Example 1.2 continued
In the DSS shell sketched for time table problems the goal specification and plan construction were built-in features. This indicates that the shell was tailored for time table problems. Nevertheless, in a DSS applied for routing problems these items might be entirely different. Therefore, a highly flexible DSS shell or DSS generator meant to treat time table problems as well as routing problems should be based on a general DSS model, such that specifying both problem types can be done by its parameters.
Likewise, it is quite probable that solving a time table problem requires another solution method than solving a routing problem. Accordingly, in a flexible DSS shell or generator also the PSC should be based on a general model problem solving such that many different solution procedures can be defined by the parameters.

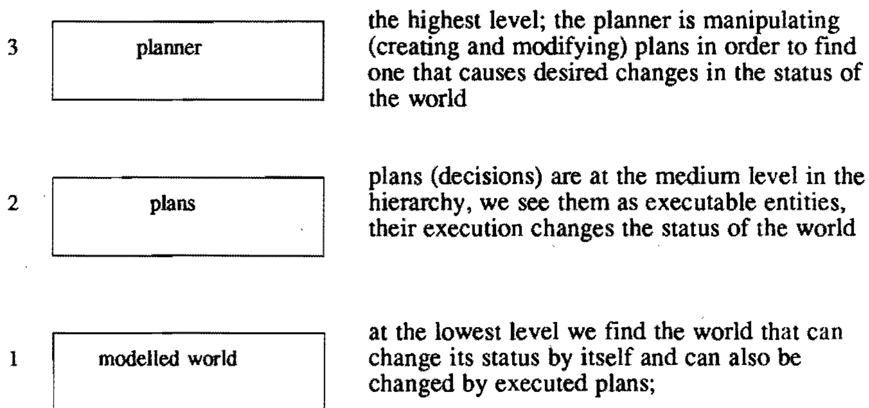
□

The main goal of our project is to make a widely applicable DSS development tool. First we are carrying out a theoretical study. We want to create a DSS skeleton - an abstract framework of a DSS - and establish a method to design DSSs guided by this skeleton. It is straightforward that we wish to have a skeleton with wide application domain. The major guidelines for our research can be summarized as follows:

- we concentrate on the automatic decision making function of a DSS;
- we try to separate domain knowledge about the problem to be solved and procedural knowledge concerning the problem solving method;
- we aim at a widely applicable model of operational planning problems by means of high level parameters;
- we intend to make a widely applicable model of (a class of) solution methods applicable for such problems.

An additional objective of our study is to obtain a clear terminology that supports good understanding of the related phenomena and facilitates further research and discussions.

Before going into detailed studies let us introduce the basic taxonomy of planning.



In this hierarchy entities of a certain level have influence on the ones on one level below: a plan changes the world, a manipulation modifies a plan. For instance *teacher* x *begins a lecture with class* y *at* z *o'clock* is a plan, its

execution brings changes in the world. Executing the command *interchange English and Mathematics in the time table* a plan (the time table) is changed, therefore it is a manipulation.

In these terms an operational planning problem has to do with the first two levels. Therefore, a model of planning problems should contain a model of the world and a model of plans extended by the facility of defining what kind of plan is wanted. We elaborate such a model in the next chapter.

Problem solving can be associated with levels number 2 and 3. A model of solution methods should describe how to perform manipulations in order to achieve a desired plan. This will be investigated in Chapter 3 and Chapter 4.

CHAPTER 2

Planning

Like many other notions of computing science the term *planning* is loaded with an everyday meaning. Such notions - unlike most of the mathematical terms - appear to have a formal meaning even without a definition. (Think, for instance, of the notion of *plan* as opposed to a *consistently complete join semi-lattice*.) This appearance causes a very undesirable effect, namely that relatively little effort is being made to clarify the foundations. The terminology of planning is far from being unambiguous, many interpretations can be given to the same word.

In this chapter we investigate planning and establish our formal interpretation. We present a conceptual model of planning problems. Setting the basic assumptions, identifying the most relevant factors and their relationships we are aiming at

- a coherent planning theory by systematic top-down analysis;
- a good insight in the components of planning problems so that we can derive a method to specify planning problems.

We try to make the choices consciously during the development of the theory, such that the restrictions and their reasons are clear. Hereby the application domain of the theory is visible and so are the possible extensions or restricted versions of our theory. Naturally we do not claim that our interpretation is the only good one, but it forms a sound formal basis for further investigations.

Notice that we did not explicitly mention that we study operational planning problems here but that is the application field we have chosen.

2.1 Model of Planning Problems

According to the hierarchy sketched in Chapter 1 the two layers we are concerned with are

1) the *world* which can change its status either by itself or by *actions* of the planner;
2) actions and *plans* consisting of actions that are executable entities acting on the world; the effect of their execution is that the status of world changes.

A *planning situation* includes the description of the circumstances in the world, the description of plans and their influence on the world. A *planning problem* can be given in the context of a planning situation by giving an initial status of the world and certain goal status. Solving a planning problem the planner wants to act towards changing the status of the world by a plan such that a goal status is reached. Often, there are also requirements about the way a goal status is reached, e.g. it must be done as cheap as possible.

Although this interpretation of a planning problem smoothly matches our intuition, we slightly modify it. In the sequel we assume that solving a planning problem the planner wants to have a plan that can transform the initial status to a goal status. A *solution of a planning problem* is then a plan which, if executed, transforms the world to a goal status. The advantage of this reformulation is a consequent terminology where the solution of a planning problem is a plan and not the fact that a certain world status is reached.

The first obvious step in elaborating our theory is to model the world by introducing *world states* as abstract entities modelling the status of the world at a certain moment. Intuitively we consider world states as snapshots taken of the world at a certain moment.

A crucial characteristic of planning problems is that time is involved. Depending upon the role of time we can distinguish two kinds of circumstances: in a static case the world does not change its status unless an action or plan is executed, in a dynamic one the status of the world can be changed any time without being triggered. In the next sections we develop a detailed formal theory of planning problems extending the results of Eiben (1989).

2.1.1 Static Case: the World as a State

In the static case it is assumed that a world state is maintained until an action takes place, with other words, every change in the status of the world is caused by an action.

Example 2.1.1.1 Blocks-World Problem (BWP)
In the Blocks-World Problem, Nilsson (1982), we have a table, several blocks on the table and a robot arm that is able to put a block onto another one or onto the table. In the beginning the blocks are in an initial configuration. The objective is to make up a sequence of movements of the robot arm that converts the blocks to a given goal configuration. In terms of world states, actions and effect we can describe the BWP as follows.

World: the table, the blocks and the robot arm;
States: block configurations;
In. state: initial configuration of the blocks;
Actions: moving a block onto another one or onto the table;
Effect: the configuration changes, since the position of the moved block changes;
Plans: sequences of actions (for one robot arm);
Goal: blocks are in a given specific configuration.
□

Example 2.1.1.2 Travelling Salesman Problem (TSP)
The travelling salesman problem is well-known in OR, cf. Bellmore and Nemhauser (1968). Its basic version reflects a simple decision situation that is still probably intractable in computational sense, i.e. it is NP-hard, see Garey and Johnson (1979).

There is a number of cities given together with data describing the distances between them. A salesman has to make the shortest possible tour visiting all the cities and returning to his home city.

World: the cities and the moving agent, distances;
States: position of the agent, list of visited cities;

In. state: start position of the moving agent (a city), no cities visited yet;
Actions: moves from one city to another;
Effect: the position of the agent changes, a city becomes visited;
Plans: sequences of actions (for one agent);
Goal: all the cities are visited exactly once, the agent is back to the start position,
the total distance of the tour is minimal.

□

Observe that the items *world*, *initial state* and the first part of *goal* belong to the first layer of our taxonomy, *actions* and *plans* belong to the second one, while *effect* lays the necessary connection between the two layers.

Remark 2.1.1.3
Taking a static or dynamic model for a certain situation is not arbitrary. For instance, let us take time windows into account for TSP, cf. Savelsbergh (1988). If we do not want to incorporate time into the world states (and remain static), then we need to introduce a state *underway* that is maintained for a period of time and then ceases (turns to being at a new city). This actually means that we apply a dynamic model.

□

There are three shortcomings of the scheme of the above examples. This scheme provides a too low resolution view on planning problems in the following sense. First, we cannot distinguish changing constituents of the world from permanent ones, e.g. the actual position of the agent from the distances between cities. Entities of the first type can be changed by an action hence they should be included in the world states. Information of the second kind is characterized by not being changed by actions. Upholding this information by the states is superfluous, hence it should be put as background data. This problem will be discussed further in section 2.2.

The second shortcoming is that we cannot distinguish possible and impossible actions. Namely, actions are not always executable in reality, consequently not every well constructed plan is executable either. *Executable in reality* is a notion with respect to the first layer, telling that something can or cannot be done. We

shall treat it on the second layer defining a predicate *allowed* such that allowed actions and plans are all executable.

Example 2.1.1.4

Reasonable definitions of allowability to the above examples are the following.

BWP : moving block A onto block C is allowed if nothing is on A and nothing is on C;

TSP : move from A to B is allowed if the agent is in A and B is not visited yet.

□

Third, when giving the goal in Example 2.1.1.2 we did not distinguish the condition about the state of the world (all the cities are visited exactly once, the agent is back to the start position) and the evaluation criterion for plans (the total distance of the tour is minimal).

These observations lead to a more detailed vision of planning.

In a planning situation we have the first and the second layer and their connection, that is

1) a set of world states;
2) a set of actions;
3) a predicate *allowed* on actions with respect to states;
4) a function *effect* which assigns a state to the pair of a state and an action.

Furthermore, we need certain general composition rules that specify how to construct plans from actions and how to extend the predicate *allowed* and the function *effect* from actions to plans.

A planning problem contains all the necessary information on what is given and what is wanted, that is it consists of

1) a planning situation;
2) an initial world state of this planning situation;
3) a condition defining the goal states;
4) an evaluation criterion for plans.

A solution of a planning problem is a plan that - when applied to the initial state - leads to a goal state. An optimal solution is a solution if it realizes the minimal

(maximal) value of the given criterion.

In the sequel we convert this detailed, still informal description into a mathematical model.

Definition 2.1.1.5

A *planning universe* is an ordered triple $(S, A, (T,<))$ of non empty sets, where

- S is a set of *world states* with a special element $\square \in S$ called *nil*;
- A is a set of *actions*;
- $(T,<)$ is an linearly ordered set of *time instances*.

□

The ordering $<$ on T often remains implicit, mentioning only T instead of $(T,<)$.

Definition 2.1.1.6

An *operation* is a pair $(a,t) \in A \times T$ and $time : A \times T \to T$ is a projection function, such that

$$\forall\, t \in T : time(a,t) = t$$

holds.

□

An operation (a,t) denotes the action a executed at the time t. Notice that hereby we modify our view assuming that it is an operation that changes a world state. To determine the set of applicable operations with respect to a certain state we introduce the following relation. Let us also remark that in our view operations have no duration.

Definition 2.1.1.7

The *allowability relation* of a planning universe (S, A, T) is a relation

$$\alpha : S \times (A \times T) \to \{true, false\},$$

such that

$$\forall\, s \in S\ \forall\, a \in A\ \forall\, t,\tau \in T : \alpha(s,(a,t)) = \alpha(s,(a,\tau)).$$

If $\alpha(s,(a,t))$ then the operation (a,t) is *allowed* in the state s.

□

Definition 2.1.1.8

Let (S, A, T) be a planning universe, α an allowability relation. An *effect-function* of (S, A, T) is a function

$$e : S \times (A \times T) \to S$$

such that for every $s \in S, a \in A, t,\tau \in T$:

a) $e(\square,(a,t)) = \square$;

b) $\alpha(s,(a,t)) = false \Leftrightarrow e(s,(a,t)) = \square$;

c) $e(s,(a,t)) = e(s,(a,\tau))$.

□

The function e describes the effect of the operations on states, that is $e(s,(a,t)) \in S$ is the state obtained by applying the operation (a,t) to the state s.

Observe that the condition of Definition 2.1.1.7 and (c) in Definition 2.1.1.8 institute a sort of time independence. These conditions immediately follow from our view of static cases as given in the introduction of this chapter. Namely, we presume that without committing an operation a state is not changed, that is it keeps all its features and properties. At this level of abstraction we consider two properties of a state: which operations can be applied to it and what effect the operations have on it. The above definitions establish that for operations the included time instance is irrelevant in determining WHAT happens, it is determining only WHEN it happens. Nevertheless, for plans (introduced later) the time instances will be necessary to determine their effect, that is the WHAT, even in the static case.

The kernel of the concept of *allowed* is enclosed in (b) in Definition 2.1.1.8 that founds the relation between e and α, in fact the relation between *executable* and *allowed*. Notice the role of the *nil* state $\square$ shown in (a). It is a universal absorber state that permits to formulate the effect of unexecutable operations the same way as that of executable ones: as a state transition. Formally, the usage of $\square$ makes it possible to define e as a complete (not partial) function on $S \times (A \times T)$. The notion of an *executable operation* is modelled as an operation that yields a state different from $\square$. The predicate *allowed* is for a "syntactical" characterization of executability, in practice α will be used to specify the real domain of e.

Lemma 2.1.1.9

$\forall\, a \in A\ \forall\, t \in T : \alpha(\square,(a,t)) = \mathit{false}.$

Proof

It is obvious from the definitions.

□

At this point we have all the ingredients needed to formalize the notion of a planning situation.

Definition 2.1.1.10

A (static) *planning situation* is a 5-tuple

$(S, A, T, \alpha, e),$

where the triple (S, A, T) forms a planning universe, α is an allowability relation and e is an effect function on (S, A, T).

□

A planning situation captures the most relevant factors of the world under consideration. To define a planning problem, however, we also need to know what a plan is and what the problem is, i.e. what kind of plan is wanted as a solution.

Definition 2.1.1.11

A *plan* is a finite set of operations. A plan $P \in \mathcal{P}(A \times T)$ is called a *section* iff

$\forall\, o_1, o_2 \in P : \mathit{time}(o_1) = \mathit{time}(o_2).$

Hereby the function *time* can be defined for any non-empty section P by

$\mathit{time}(P) = \mathit{time}(o)$

taking $o \in P$ arbitrarily.

If P is a plan then a non empty section $R \subseteq P$ is called a *maximal section of P* if for every section $Q \subseteq P$ it holds that

$\mathit{time}(R) = \mathit{time}(Q) \Rightarrow Q \subseteq R.$

□

It is easy to see that we can uniquely divide any plan into disjoint maximal sections. Before the next definition recall that for an arbitrary set X with cardinality $n \in \mathbb{N}$ a numbering is a bijection

$\nu : \{1, \ldots, n\} \to X.$

Definition 2.1.1.12

If a plan P is divided into n maximal sections then the *natural numbering* of the sections is a bijection

$\nu : \{1, \ldots, n\} \to \{\, P' \in \mathcal{P}(A \times T) \mid P' \textit{ is a maximal section of } P \,\}$

such that

$\forall\, i,j \in \{1, \ldots, n\} : i < j \;\Leftrightarrow\; time\,(P_i) < time\,(P_j),$

where P_i denotes $\nu(i)$ for the sake of convenience.

□

Proposition 2.1.1.13

For any plan there is one and only one natural numbering.

Proof

It is obvious, the ordering on T implies the existence, unicity follows from the unicity of the maximal sections.

□

Definition 2.1.1.14

$[P_1, \ldots, P_n]$ is the *partition of a plan P* (denoted as $P \sim [P_1, \ldots, P_n]$) if $P_1, \ldots, P_n$ are the maximal sections of P numbered by the natural numbering.

□

The following proposition gives a simple characterization of the partition of a plan.

Proposition 2.1.1.15

For every plan P it holds that $P \sim [P_1, \ldots, P_n]$ iff $P_1, \ldots, P_n$ are sections such that

a) $P = \bigcup_{i=1}^{n} P_i$;

b) $time(P_1) < time(P_2) < \ldots < time(P_n)$.

Proof

The $\Rightarrow$ direction is self-evident by definition.

To prove the $\Leftarrow$ direction let $P_1, \ldots, P_n$ be sections such that (a) and (b) hold. By

(b) we have that $P_1, \ldots, P_n$ are numbered in the natural way, thus all we need to show is that they are all maximal. If they were not, then we had an $i \in \{1, \ldots, n\}$ and an $o \in P \setminus P_i$ such that

$time(o) = time(P_i)$.

But (a) implies that $o \in P_j$ for a section P_j $(j \neq i)$, hence

$time(P_j) = time(o)$,

thus

$time(P_j) = time(P_i)$

with $i \neq j$ which contradicts (b).

□

An important special case within our planning theory is obtained if we restrict ourselves to sequential plans, where every section is a singleton. Due to Proposition 2.1.1.15 such plans can be simply sequentialized by the time of their operations, that is they can be written in a form $\{o_1, \ldots, o_n\}$ with $time(o_1) < \ldots < time(o_n)$.

Now we have everything prepared to extend *effect* and *allowed* from operations to plans. Nevertheless, before we can define the effect of a plan based on the effects of its operations, we have to make a choice about the effect of a section. Hereby we encounter a hard problem: how to handle the effect of more operations at the same time. Here we sketch two ways of treating this problem.

Basically we can either consider mutual influence between equitemporal actions or not. If we do, then we cannot determine the effect of a section from the single effect of its elements. In this case we have to define the effect of every section, thus we have to define an extended effect function as a primitive.

We decide to assume the opposite of the above, that is that equitemporal actions do not interfere. Hence we can decompose the effect of a section which implies that the extended effect function will not be a primitive, it will be computed by the effect function e.

Notice that this decision introduces new difficulties. For instance, if the operations o and o' act in parallel at the same time instance then we can take either $e(e(s,o),o')$ or $e(e(s,o'),o)$ as the effect of $\{o, o'\}$ on the state s. Thus, computing the effect of parallel actions sequentially introduces ambiguity, since a section with k different operations can be numbered in $k!$ different ways that

determine $k!$ different sequences, thus we might have $k!$ different outcomes. To exclude this ambiguity we define the effect of sections such that for ambiguous cases it yields the *nil* element $\square$.

Definition 2.1.1.16

The *extended effect function* of a planning situation (S, A, T, α, e)

$$e' : S \times \mathcal{P}(A{\times}T) \nrightarrow S$$

is defined for any $s \in S$ and finite $P \in \mathcal{P}(A{\times}T)$ in the following way:

if $P = \emptyset$ then $e'(s, P) = s$;

if $P = \{o_1, \ldots, o_n\}$ is a section with an arbitrary numbering of its operations and for every subset $\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$ and every permutation π of $\{i_1, \ldots, i_k\}$

$$e(\ldots e(s, o_{\pi(i_1)}) \ldots, o_{\pi(i_k)}) = e(..e(s, o_{i_1}) \ldots, o_{i_k})$$

holds, then

$$e'(s, P) = e(\ldots e(s, o_1) \ldots, o_n) ;$$

if $P = \{o_1, \ldots, o_n\}$ is a section and there exists a subset $\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$ and a permutation π of $\{i_1, \ldots, i_k\}$ such that

$$e(\ldots e(s, o_{\pi(i_1)}) \ldots, o_{\pi(i_k)}) \neq e(\ldots e(s, o_{i_1}) \ldots, o_{i_k})$$

then

$$e'(s, P) = \square ;$$

if $P \sim [P_1, \ldots, P_n]$ then $e'(s, P) = e'(\ldots e'(s, P_1) \ldots, P_n)$.

$\square$

Notice that the intuitive interpretation of time is formalized right here by using the natural numbering to order the sections of a plan. This establishes that the effect of the operations with a smaller (earlier) assigned time instance precedes the effect of the ones with a larger (later) time instance.

Definition 2.1.1.17

The *extended allowability relation* α' establishes allowability of plans with respect to states. It is a Boolean function

$$\alpha' : S \times \mathcal{P}(A{\times}T) \nrightarrow \{true, false\}$$

such that for any $s \in S$ and finite $P \in \mathcal{P}(A{\times}T)$:

if $P = \emptyset$ and $s = \square$ then $\alpha'(s, P) = false$;

if $P = \emptyset$ and $s \neq \square$ then $\alpha'(s, P) = true$;

if $P = \{o_1,...,o_n\}$ is a section with an arbitrary numbering of its operations, then

$\alpha'(s, P) = \alpha(s,o_1) \wedge \alpha(e'(s,\{o_1\}), o_2) \wedge \ldots \wedge \alpha(e'(s,\{o_1, \ldots, o_{n-1}\}), o_n)$;

if $P \sim [P_1, \ldots, P_n]$ then

$\alpha'(s, P) = \alpha'(s,P_1) \wedge \ldots \wedge \alpha'(e'(\ldots e'(s,P_1) \ldots, P_{n-1}), P_n)$.

We say that a plan P is *allowed* in a state s if $\alpha'(s, P) = true$.

□

By the next proposition we show that *allowed* and *effect* are properly extended from operations to plans.

Proposition 2.1.1.18

Let (S, A, T, α, e) be a planning situation, α' the extended allowability relation and e' the extended effect function. Then for every $s \in S$ and $P \subseteq A \times T$ it holds that

$\alpha'(s, P) = false \Leftrightarrow e'(s, P) = \square$.

Proof

For the sake of convenience we denote $\alpha'(s, P) = false$ by $\neg\, \alpha'(s, P)$.

1) The case of $P = \emptyset$ is trivial by Definition 2.1.17.

2) If P is a section $\{o_1, \ldots, o_n\}$ then

$\neg\, \alpha'(s, P)$

iff (by Definition 2.1.1.17)

$\neg\, \alpha(s,o_1) \vee \neg\, \alpha(e'(s,\{o_1\}), o_2)) \vee \ldots \vee \neg\, \alpha(e'(s, \{o_1, \ldots, o_{n-1}\}), o_n)$

iff (by Definition 2.1.1.8)

$e(s,o_1) = \square \vee e(e'(s,\{o_1\}), o_2)) = \square \vee \ldots \vee e(e'(s, \{o_1, \ldots, o_{n-1}\}), o_n) = \square$

iff (by Definition 2.1.1.16)

$e'(s, P) = \square$

3) If $P \sim [P_1, \ldots, P_n]$ then

$\neg\, \alpha'(s, P)$

iff (by Definition 2.1.1.17)

$\neg\, \alpha'(s,P_1) \vee \ldots \vee \neg\, \alpha'(e'(\ldots e'(s,P_1) \ldots, P_n))$

iff

$\exists\, k \in \{1, \ldots, n\} : \neg\, \alpha'(e'(\ldots e'(s,P_1) \ldots, P_k))$

iff (by (2) above)

$\exists\, k \in \{1, \ldots, n\} : e'(e'(\ldots e'(s,P_1) \ldots, P_{k-1}), P_k) = \square$

iff (by iterating (a) of Definition 2.1.1.8 and Definition 2.1.1.16)

$e'(\ldots e'(s,P_1) \ldots, P_n) = \square$

iff (by Definition 2.1.1.16)

$e'(s, P) = \square$.

□

Definition 2.1.1.19

A *planning problem* is defined by a planning situation (S, A, T, α, e) and a triple (s, γ, κ), where $s \in S$ is the *initial state*, $\gamma : S \to \{true, false\}$ is the *goal condition*, such that $\gamma(\square) = false$ and $\kappa : \mathcal{P}(A \times T) \to \mathbb{R}^k$ is a (multidimensional) *criterion* to minimize.

A plan P is a *solution of the planning problem* if

$\gamma(e'(s, P)) = true$;

it is an *optimal solution of the planning problem* if it is a solution and

$\forall P' \in \mathcal{P}(A \times T) : [\gamma(e'(s, P')) \Rightarrow \kappa(P) \le \kappa(P')]$,

where for $x = (x_1, \ldots, x_k) \in \mathbb{R}^k$ and $x' = (x'_1, \ldots, x'_k) \in \mathbb{R}^k$

$x \le x'$ iff $\forall\, i \in \{1, \ldots, k\} : x_i \le x'_i$.

□

Let us make two remarks with respect to this definition. First, note that by the presence of κ we are not restricted to optimization-like planning problems. If we are not aiming at any optimum then we can define κ constant through $\mathcal{P}(A \times T)$. The second is that κ could be generalized to an arbitrary condition γ' on plans. In this case a planning problem would be (S, A, T, α, e) with (s, γ, γ') and the condition $\forall P' \in \mathcal{P}(A \times T) : [\gamma(e'(s, P')) \Rightarrow \kappa(P) \le \kappa(P')]$ would be a special case of $\gamma'(P)$.

Proposition 2.1.1.20

Every solution of a planning problem is allowed, that is if (S, A, T, α, e), (s, γ, κ) is a planning problem and $P \subseteq A \times T$ is finite then $\gamma(e'(s, P))$ implies $\alpha'(s, P)$.

Proof It is a self-evident corollary of the definition of γ and Proposition 2.1.1.18.

□

2.1.2 Dynamic Case: the World as a Process

Recall the introduction of Chapter 2.1; the characteristic feature in a dynamic situation is that a world state can change without having an action executed.

Example 2.1.2.1 Precedence Constrained Scheduling Problem (PCSP)
We have a finite number of jobs and machines that can perform jobs. Given a predecessor relation on jobs, a description of the abilities of the machines (which job can be done on which machine) and the duration of performance for jobs and machines, we have to schedule the jobs on the machines such that no job is performed before its predecessors have been completed and the total processing time is minimal, cf. Garey and Johnson (1979).

World: machines, jobs, which job can be done on which machine, predecessor relation on jobs, durations;
States: pairs of machines and jobs (machine performing job), list of completed jobs;
In. state: no machine is performing any job, no jobs completed;
Actions: beginning a job on a machine;
Allowed: a free machine is allowed to begin a not completed job if the predecessors of the job have already been completed and the machine has the ability to perform the given job;
Effect: for a period: machine is performing job, later: machine free and job completed;
Plans: sets of operations where more actions can take place at the same time;
Goal: all the jobs are completed;
Criterion: the total processing time is minimal.
□

Observe that in a PCSP one operation triggers two state transitions at two different time instances: first there is a new world state maintained for a period of time (the machine is busy with the job), then there is another state transition that leads to the final result of the given action (the machine becomes free and the job becomes completed). This phenomenon can be viewed as having a change in the world state that is not caused by an operation (at the time of the change). We

consider two possibilities to model dynamic worlds.

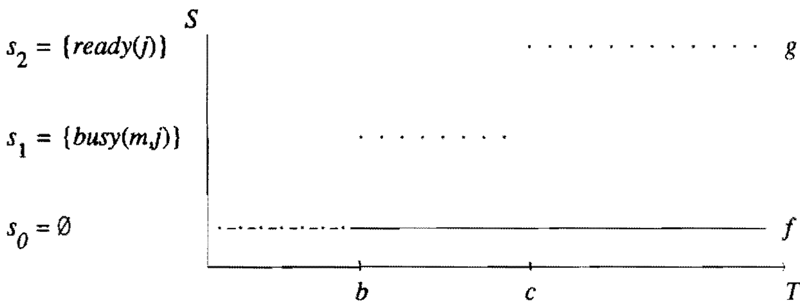
First, we can drop the intuitive basis of world states and give up the view that a state is a snapshot of the world at a certain moment. Introducing entities that rather belong to periods than to moments we can try to capture the problem by these new objects.

The second possibility is to keep the state-snapshot vision but embedding states into a time flow. This would lead to the notion of a *process* and the replacement of the state-operation-new state construction by a process-operation-new process model as the next example illustrates.

Example 2.1.2.2

Let a 4-tuple (m,j,b,c) stand for a machine m, a job j, a beginning time b and completion time c to describe that m is performing j between b and c. Such a 4-tuple gives a partial world description over the period of time between b and c. To obtain a complete view on what happens at a time instance t we have to check the set of all these 4-tuples taking their 'projection' on t. Thus a set of such 4-tuples can be seen as a world description. The tuple (m,j,b,c) can be seen as some mixture of an operation and its effect, that of beginning j on m at b, coupled by telling that m will be occupied between b and c and that j will be finished at c. A set of such 4-tuples thus also can be seen as some kind of plan.

Let us keep states as given in Example 2.1.2.1. Introducing a predicate *ready* and a predicate *busy* we can identify a state by a set of atomic formulae that are true in that state. If we now introduce processes as parameterizations of the world states by time then the operation of beginning j on m at b turns the process f into the process g as exhibited below.



□

After studying the two possibilities we have chosen the second one. The reasons to choose the process based approach are threefold.

- The original interpretation of world states is kept: they can still be interpreted as modelling the world at a certain moment.
- Taking processes as the basic entities changed by operations, we preserve the previous construction of planning problems: the notions *operation*, *allowed* and *effect* can be naturally extended to processes.
- In Example 2.1.2.2 the operations, effects and states are mixed up, something that can be advantageous for an economical representation formalism, but not for a conceptual model that should clarify the matter.

Therefore, in this section we extend the static world of states to a dynamic world by introducing processes as parameterizations of the world states by time. Taking processes as the basic entities to be changed by operations, we introduce *dynamic planning problems* where an *initial process* and certain *goal processes* are given and we want to have a plan that transforms the initial process to a goal process. In the sequel we formalize this concept of planning preserving as much as possible from the static model. Definitions and propositions that are identical to those of section 2.1.1 are not repeated.

Definition 2.1.2.3

A *process* of a planning universe (S, A, T) is a partial function

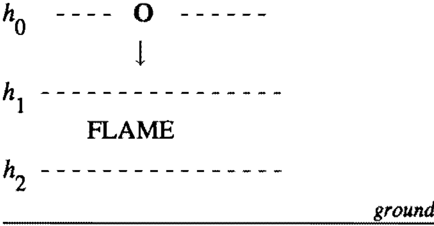
$f : T \nrightarrow S \setminus \{\square\}$. We also introduce a special absorber process that will be denoted as $\hat{\square}$ and called *nil* process. The notation $F_{S,T}$ stands for the *set of all processes* of the universe (S, A, T) extended by the *nil* process.

For notational convenience we shall denote $F_{S,T}$ by simply F if it can not lead to confusion.

□

Example 2.1.2.4

Let us take an example from the so called qualitative physics, see Forbus (1984). A ball is dropped above a flame at the moment t. Falling down it goes through several states eg. falling, falling and being heated and then finally being broken when reaching the ground.



Let us introduce the predicates *falling—at*, *heated* and *broken* and let us again identify a state by a set of atomic formulae that are true in that state. Then the environment can be modeled by the following states and time instances:

$$S = \bigcup_{h \in [0,100]} \{ \textit{falling—at}(h) \} \cup \bigcup_{h \in [0,100]} \{ \textit{falling—at}(h) \wedge \textit{heated} \} \cup \{ \textit{broken} \}$$

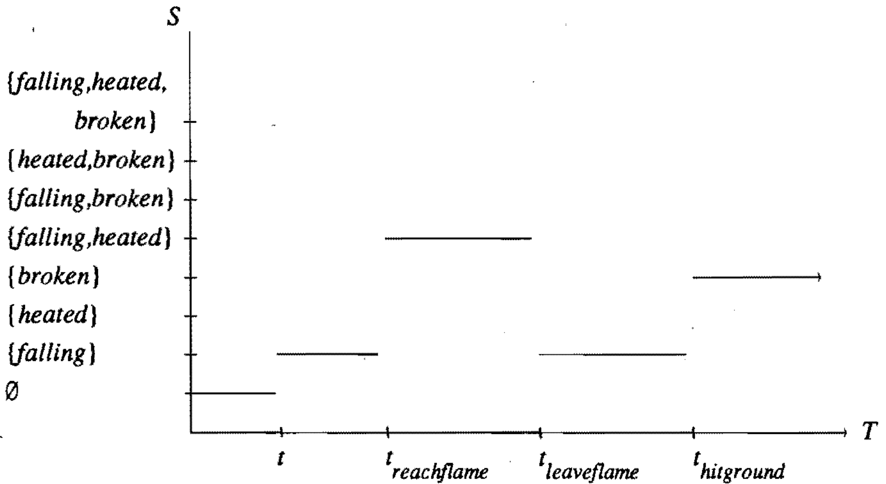
$$T = \mathbb{R}_0^+$$

and the falling ball dropped at time t can be described by the following process f.

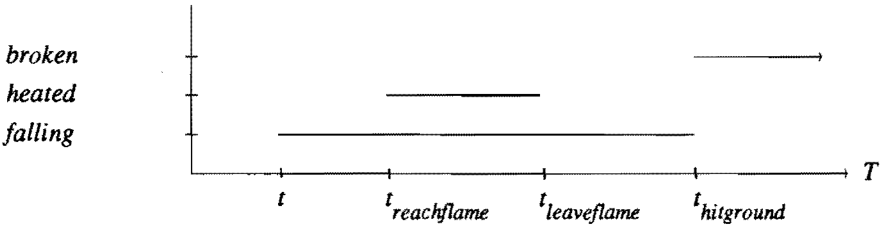
$$f(\tau) = \begin{cases} \{\textit{falling—at}(h_\tau)\} & \text{if } t \leq \tau < t_{reachflame} \\ \{\textit{falling—at}(h_\tau) \wedge \textit{heated}\} & \text{if } t_{reachflame} \leq \tau < t_{leaveflame} \\ \{\textit{falling-at}(h_\tau)\} & \text{if } t_{leaveflame} \leq \tau < t_{hitground} \\ \{\textit{broken}\} & \text{if } t_{hitground} \leq \tau \end{cases}$$

The exact value of h_τ , $t_{reachflame}$, $t_{leaveflame}$ and $t_{hitground}$ can be calculated by the well-known Newtonian laws of mechanics if h_0, h_1, h_2 are given.
□

Notice that a state is again identified by the set of atomic formulae that are true in it. We also maintain the so called Closed World Assumption, stating that if an atom is not contained in a state s then it is not true in s. By this representation of states their inner structure is visible in the example: roughly, states are subsets of the set { *falling, heated, broken* }. This implies that we have two possibilities of depicting a process. The first one is to indicate every state on the vertical axis, that is every element from $\mathcal{P}$({ *falling, heated, broken* }). This leads to 8 items and the following figure.



The other possibility is that we indicate only the 'ingredients' of the states, that is each of *falling, heated* and *broken* and the actual state at a moment t can be obtained by upwards projection. This results in the following picture.



Notice that this is nothing but the well-known Gantt chart representation.

Definition 2.1.2.5

A *dynamic allowability relation* of a planning universe (S, A, T) is a relation

$\hat{\alpha} : F \times (A \times T) \rightarrow \{true, false\}$.

□

Denoting time segments we shall use the following notational conventions: for any $t \in T$

$tT = \{ \tau \in T : t < \tau \}$, $\underline{t}T = \{ \tau \in T : t \leq \tau \}$;

$Tt = \{ \tau \in T : \tau < t \}$, $T\underline{t} = \{ \tau \in T : \tau \leq t \}$.

Definition 2.1.2.6

Let (S, A, T) be a planning universe, $\hat{\alpha}$ be a dynamic allowability relation. A function

$$\hat{e} : F \times (A \times T) \to F$$

is a *dynamic effect-function* if it holds that for every $f \in F$ $a \in A$, $t \in T$:

a) $\hat{e}(\hat{\square},(a,t)) = \hat{\square}$;

b) $\neg\, \hat{\alpha}(f,(a,t)) \Leftrightarrow \hat{e}(f,(a,t)) = \hat{\square}$;

c) if $\hat{e}(f,(a,t)) \neq \hat{\square}$ then $(\hat{e}(f,(a,t))) \restriction T\underline{t} = f \restriction T\underline{t}$.

□

Here is a crucial difference between the static and the dynamic planning model! In the static case Definition 2.1.1.7 and point (c) of the Definition 2.1.1.8 expressed independence from the time instance of an operation. Obviously, this independence would not hold for processes that are essentially meant to describe changing situations. Point (c) of Definition 2.1.2.6 is to require that only the future, and never the past of a process is changed by an operation.

Definition 2.1.2.7

Let (S, A, T) be a planning universe. A *dynamic planning situation* of (S, A, T) is a 5-tuple

$$(F, A, T, \hat{\alpha}, \hat{e})$$

where $F = F_{S,T}$, $\hat{\alpha}$ is a dynamic allowability relation and $\hat{e}$ is a dynamic effect-function of (S, A, T).

□

Definition 2.1.2.8

The *extended effect function* of a dynamic planning situation $(F, A, T, \hat{\alpha}, \hat{e})$

$$\hat{e}' : F \times \mathcal{P}(A \times T) \to F$$

is defined for any $f \in F$ and finite $P \in \mathcal{P}(A \times T)$ in the following way:

if $P = \emptyset$ then $\hat{e}'(f, P) = f$;

if $P = \{o_1, \ldots, o_n\}$ is a section with an arbitrary numbering of its operations and for every subset $\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$ and every permutation π of $\{i_1, \ldots, i_k\}$

$$e(\ldots e(f,o_{\pi(i_1)})\ldots,o_{\pi(i_k)}) = e(\ldots e(f,o_{i_1})\ldots,o_{i_k})$$

holds, then

$$\hat{e}'(f, P) = \hat{e}(\ldots\hat{e}(f, o_1)\ldots, o_n);$$

if $P = \{o_1, \ldots, o_n\}$ is a section and there exists a subset $\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$ and a permutation π of $\{i_1, \ldots, i_k\}$ such that

$$e(\ldots e(f,o_{\pi(i_1)})\ldots,o_{\pi(i_k)}) \neq e(\ldots e(f,o_{i_1})\ldots,o_{i_k})$$

then

$$\hat{e}'(f, P) = \hat{\square};$$

if $P \sim [P_1, \ldots, P_n]$ then $\hat{e}'(f, P) = \hat{e}'(\ldots\hat{e}'(f,P_1)\ldots,P_n)$.

□

Notice that the basic feature of time is the same as for the static case: the effect of the operations with a smaller (earlier) assigned time instance precedes the effect of the ones with a larger (later) one.

Recall point (c) of Definition 2.1.2.6 that can be informally understood as stating that the past of a process cannot be changed by an operation. The question whether this property also holds for plans in general is answered by the following proposition.

Proposition 2.1.2.9 (Past Invariance)
Let $f \in F$, $P \sim [P_1, \ldots, P_n]$, $t_i = time(P_i)$ for every $i \in \{1, \ldots, n\}$. Then $\hat{e}'(f, P) \neq \hat{\square}$ implies $\hat{e}'(f, P) = f$ on $T_{\underline{t}_1}$.

Proof

1) $n = 1$ (P is a section)

Let $P = P_1 = \{o_1, \ldots, o_k\}$, $k > 0$ and $t = time(P)$. Furthermore, let $f_0 = f$ and

$$f_i = \hat{e}(\ldots\hat{e}(f_0,o_1)\ldots,o_i),$$

for $i \in \{1, \ldots, k\}$.

Notice that $\hat{e}'(f, P) \neq \hat{\square}$ implies $f_i \neq \hat{\square}$ for every $i \in \{1, \ldots, k\}$ and then by the iterated application of point (c) of Definition 2.1.2.6 we have that

$$f_i \upharpoonright T_{\underline{t}} = f_{i-1} \upharpoonright T_{\underline{t}} \quad \text{for every } i \in \{1, \ldots, k\},$$

hence

$$f_k \upharpoonright T_{\underline{t}} = \ldots = f_1 \upharpoonright T_{\underline{t}} = f_0 \upharpoonright T_{\underline{t}},$$

thus

$\hat{e}'(f, P) = f \;\; on\; Tt$

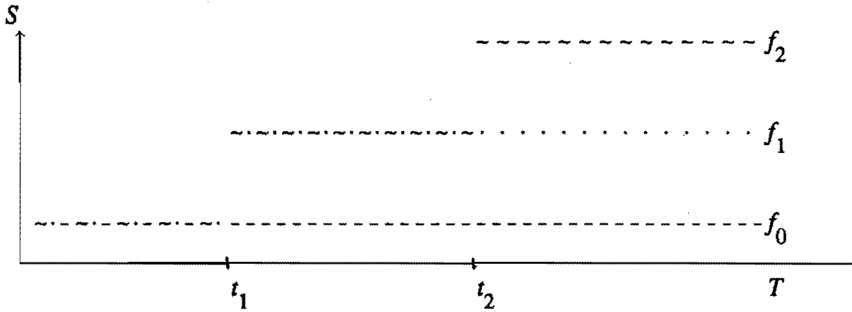
2) $n > 1$

Let $f_0 = f$ and $f_i = \hat{e}(\ldots\hat{e}(f_0,P_1)\ldots,P_i),\; i \in \{1,\ldots,n\}$.

$\hat{e}'(f, P) \neq \hat{\square}$ implies $f_i \neq \hat{\square}$ and then by (1) and Definition 2.1.2.6 we obtain that for every $i \in \{1,\ldots,n\}$

$f_i \upharpoonright Tt_i = f_{i-1} \upharpoonright Tt_i$

as the figure below illustrates it for $n = 2$.



Since $P_1,\ldots,P_n$ are numbered by the natural numbering

$Tt_{i-1} \subseteq Tt_i$ for every $i \in \{1,\ldots,n\}$,

thus

$f_n \upharpoonright Tt_1 = f_0 \upharpoonright Tt_1$

□

Definition 2.1.2.10

The *extended allowability relation* $\hat{\alpha}'$ of a dynamic planning situation $(F, A, T, \hat{\alpha}, \hat{e})$ establishes allowability of plans with respect to processes. It is a Boolean function

$\hat{\alpha}' : F \times \mathcal{P}(A \times T) \rightarrow \{true, false\}$

such that for any $f \in F$ and finite $P \in \mathcal{P}(A \times T)$

if $P = \emptyset$ and $f = \hat{\square}$ then $\alpha'(f, P) = false$;

if $P = \emptyset$ and $f \neq \hat{\square}$ then $\alpha'(f, P) = true$;

if $P = \{o_1,\ldots,o_n\}$ is a section with an arbitrary numbering of its operations then

$$\hat{\alpha}'(f, P) = \alpha(f,o_1) \wedge \hat{\alpha}(\hat{e}'(f,\{o_1\}), o_2) \wedge \ldots \wedge \hat{\alpha}(\hat{e}'(f,\{o_1,\ldots,o_{n-1}\}), o_n);$$

if $P \sim [P_1, \ldots, P_n]$ then

$$\hat{\alpha}'(f, P) = \hat{\alpha}'(f, P_1) \wedge \ldots \wedge \hat{\alpha}'(\hat{e}'(\ldots \hat{e}'(f, P_1) \ldots, P_{n-1}), P_n).$$

We say that a *plan* P *is allowed* w.r.t. a process f if $\hat{\alpha}'(f, P) = true$.

□

Proposition 2.1.2.11

Let $(F, A, T, \hat{\alpha}, \hat{e})$ be a dynamic planning situation, $\hat{\alpha}'$ and $\hat{e}'$ the corresponding extended allowability relation and the extended effect function, respectively. Then for every $f \in F$ and $P \in \mathcal{P}(A \times T)$ it holds that

$$\hat{\alpha}'(f, P) = false \Leftrightarrow \hat{e}'(f, P) = \hat{\square}.$$

Proof

It is analogous to the proof of Proposition 2.1.1.18 with $f \in F$ instead of $s \in S$.

□

Definition 2.1.2.12

A *dynamic planning problem* is defined by a dynamic planning situation $(F, A, T, \hat{\alpha}, \hat{e})$ and a triple $(f_0, \hat{\gamma}, \kappa)$, where $f_0 \in F$ is the *initial process*,

$\hat{\gamma} : F \rightarrow \{true, false\}$ is the *goal condition*, such that $\hat{\gamma}(\hat{\square}) = false$ and
$\kappa : \mathcal{P}(A \times T) \rightarrow \mathbb{R}^k$ is a (multidimensional) *criterion*.

A plan P is a *solution of a dynamic planning problem* iff

$$\hat{\gamma}(\hat{e}'(f_0, P)) = true;$$

it is an *optimal solution* if it is a solution and

$$\forall P' \in \mathcal{P}(A \times T) : [\hat{\gamma}(\hat{e}'(f_0, P')) \Rightarrow \kappa(P) \leq \kappa(P')],$$

where for $x = (x_1, \ldots, x_k) \in \mathbb{R}^k$ and $x' = (x'_1, \ldots, x'_k) \in \mathbb{R}^k$

$$x \leq x' \text{ iff } \forall i \in \{1, \ldots, k\} : x_i \leq x'_i .$$

□

Proposition 2.1.2.13

Every solution of a planning problem is allowed, that is if $(F, A, T, \hat{\alpha}, \hat{e})$, $(f, \hat{\gamma}, \kappa)$ is a dynamic planning problem and $P \subseteq A \times T$ is finite then $\hat{\gamma}(\hat{e}'(f, P))$ implies $\hat{\alpha}'(f, P)$.

Proof

It is straightforward from the definition of $\hat{\gamma}$ and Proposition 2.1.2.11.

□

The structure represented by the five tuple (S, A, T, α, e) or $(F, A, T, \hat{\alpha}, \hat{e})$ can be considered as a model to describe planning situations where the elements of the tuples are the parameters. More precisely, we can specify a planning situation by a 6-tuple (x, S, A, T, α, e), where $x \in$ {*static*, *dynamic*} and the values of S, A, T, α, e must be such that (S, A, T, α, e) forms a static planning situation if x = *static* and $(F_{S,T}, A, T, \alpha, e)$ forms a dynamic planning situation if x = *dynamic*. Defining a planning situation we give a domain description or decision model; defining a triple (s_0, γ, κ), respectively $(f_0, \hat{\gamma}, \kappa)$ determines a problem to solve.

We believe that this framework carries those aspects of the world that are relevant for planning. The examples in section 2.2 justify this belief demonstrating how to use the formalism as a high level description language to specify planning problems. In the meanwhile, by such a practical exercise we gain a more detailed view about how these parameters can be given and what kind of value they can have. In particular, the examples will serve as good illustration of parameters having expressions of a high level language as values. This will bring us closer to outline a method to define a planning problem within a DSS.

2.1.3 The Role of Time

Intuitively it is clear that the dynamic model of planning is a generalization of the static one. More precisely, we envisage that if the dependence on time (the parameterization) is kept constant then we get the equivalent of a static planning model within a dynamic one.

Likewise, says intuition, if we consider a process as one object - a (meta) state - then we can project a dynamic model into a static one. To investigate this question formally we can regard a planning situation as a space of objects (S or F) with a function on it (e or $\hat{e}$) and interpret 'being the equivalent of' by the following double definition.

Definition 2.1.3.1

Let (S, A, T) and (W, A, T) be two planning universes, (S, A, T, α, e) and $(F_{W,T}, A, T, \hat{\alpha}, \hat{e})$ be a static and a dynamic planning situation and let F denote $F_{W,T}$.

We say that (S, A, T, α, e) *has an isomorphic representation in* $(F, A, T, \hat{\alpha}, \hat{e})$ if there exists an injection $I : S \rightarrow F$ such that

$$I(\square) = \hat{\square}$$

and for every $a \in A$, $t \in T$ and $s \in S$

$$I(e(s,(a,t))) = \hat{e}(I(s),(a,t)).$$

$(F, A, T, \hat{\alpha}, \hat{e})$ *has an isomorphic representation in* (S, A, T, α, e) if there exists an injection $J : F \rightarrow S$ such that

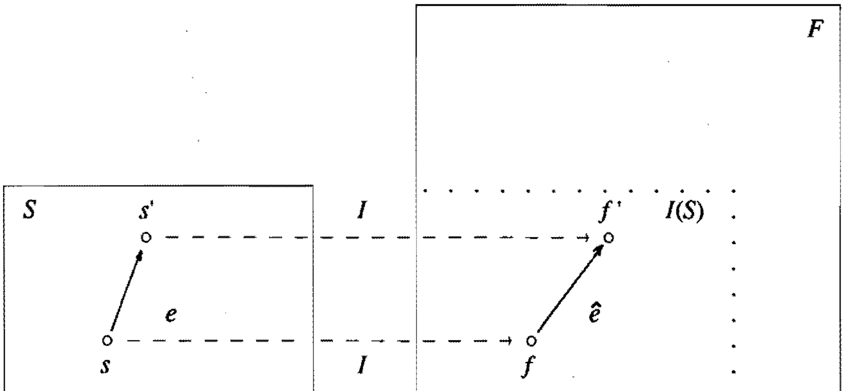
$$J(\hat{\square}) = \square$$

and for every $a \in A$, $t \in T$ and $f \in F$

$$J(\hat{e}(f,(a,t))) = e(J(f),(a,t)).$$

□

To illustrate the meaning of this definition us take the isomorphic representation of a static case in a dynamic one and consider the following figure.



For an isomorphic representation of a static situation in a dynamic one an injection I is required such that I and *effect* commute, i.e. that $I \circ e = \hat{e} \circ I$. Notice that if such an I can be given then the static situation can be handled by a dynamic one.

Namely, we can compute the effect of operations on states by applying

$$e = I^{-1} \circ \hat{e} \circ I.$$

Proposition 2.1.3.2

Let (S, A, T, α, e) be a static planning situation. If there exist $s,z \in S \setminus \{\square\}$, $s \neq z$ and $o \in A \times T$ such that $e(s,o) = z$ then there is no planning universe (W, A, T) and dynamic planning situation $(F_{W,T}, A, T, \hat{\alpha}, \hat{e})$ such that (S, A, T, α, e) has an isomorphic representation in $(F_{W,T}, A, T, \hat{\alpha}, \hat{e})$.

Proof

Let (S, A, T, α, e) be a static planning situation, $s,z \in S \setminus \{\square\}$, $s \neq z$ and $(a,t) \in A \times T$ such that

$$e(s,(a,t)) = z.$$

If (W, A, T) is a planning universe and $(F_{W,T}, A, T, \hat{\alpha}, \hat{e})$ is a dynamic planning situation such that (S, A, T, α, e) has an isomorphic representation in $(F_{W,T}, A, T, \hat{\alpha}, \hat{e})$ by an injection I, then

$$I(s) \neq I(z).$$

On the other hand, point (c) of Definition 2.1.1.8 implies

$$\forall\, \tau \in T : e(s,(a,\tau)) = z$$

and therefore

$$\hat{e}(I(s),(a,\tau)) = I(e(s,(a,\tau))) = I(z) \quad \text{for every } \tau \in T$$

by Definition 2.1.3.1. Observe that by $z \neq \square$ and the properties of I we have

$$\hat{e}(I(s),(a,\tau)) \neq \hat{\square} \quad \text{for any } \tau \in \mathrm{T}.$$

Then by (c) of Definition 2.I.2.6 this latter implies

$$\hat{e}(I(s),(a,\tau)) \upharpoonright T\underline{\tau} = I(s) \upharpoonright T\underline{\tau} \quad \text{for every } \tau \in T.$$

Hence

$$I(s) \upharpoonright T\underline{\tau} = I(z) \upharpoonright T\underline{\tau} \quad \text{for every } \tau \in T$$

thus

$$I(s) = I(z),$$

which is a contradiction.

$\square$

Proposition 2.1.3.3

Let (W, A, T) be a planning universe and $(F_{W,T}, A, T, \hat{\alpha}, \hat{e})$ be a dynamic planning situation. If there exist $f \in F_{W,T} \setminus \{\hat{\square}\}$, $a \in A$ and $t,\tau \in T$ $(t \neq \tau)$ such that

$\hat{e}(f,(a,t)) \neq \hat{e}(f,(a,\tau))$ then there is no planning universe (S, A, T) and static planning situation (S, A, T, α, e) such that $(F_{W,T}, A, T, \hat{\alpha}, \hat{e})$ has an isomorphic representation in (S, A, T, α, e).

Proof

Let $(F_{W,T}, A, T, \hat{\alpha}, \hat{e})$ be a dynamic planning situation, $f \in F_{W,T} \setminus \{\hat{\square}\}$, $a \in A$ and $t,\tau \in T$ $(t \neq \tau)$ be such that $\hat{e}(f,(a,t)) \neq \hat{e}(f,(a,\tau))$. If (S, A, T, α, e) is a static planning situation such that $(F_{W,T}, A, T, \hat{\alpha}, \hat{e})$ has a isomorphic representation in (S, A, T, α, e) by J, then

$$J(\hat{e}(f,(a,t))) \neq J(\hat{e}(f,(a,\tau)))$$

since J is an injection. On the other hand,

$$J(\hat{e}(f,(a,t))) = e(J(f),(a,t))$$

by Definition 2.1.3.1; furthermore by (c) of Definition 2.1.1.8 we have

$$e(J(f),(a,t)) = e(J(f),(a,\tau)).$$

This implies

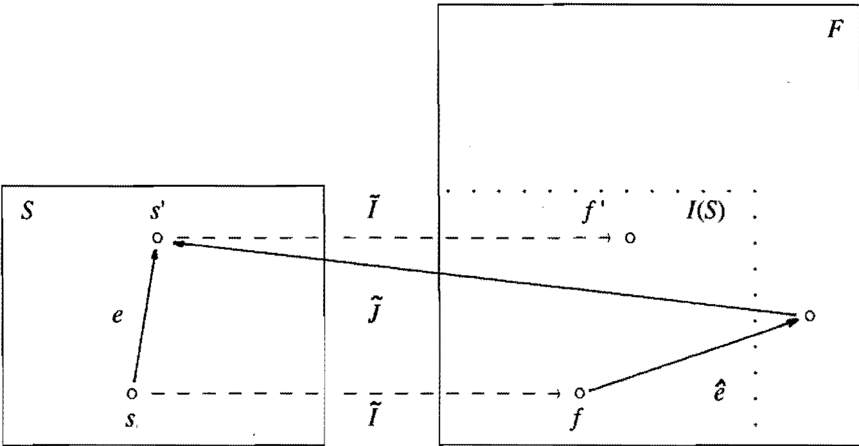
$$J(\hat{e}(f,(a,t))) = J(\hat{e}(f,(a,\tau))),$$

which is a contradiction.

□

These propositions demonstrate that a non trivial static (dynamic) planning situation cannot be isomorphically represented in a dynamic (static) one. The proofs also show the source of this fundamental mismatch: (c) of Definition 2.1.1.8 and (c) of Definition 2.1.2.6 counteract each other. Since these points embody the very nature of the static, respectively the dynamic case, from the above propositions we can conclude that static and dynamic models are deeply different by nature.

The above results imply that maintaining only static (dynamic) models and computing the dynamic (static) effect function through the appropriate mapping $J^{-1} \circ e \circ J$ $(I^{-1} \circ \hat{e} \circ I)$ is impossible. We can, however, construct a mapping $\tilde{I}$ from static to dynamic universes and a corresponding non-injective mapping $\tilde{J}$ from dynamic to static universes such that $e = \tilde{J} \circ \hat{e} \circ \tilde{I}$. This means that - although not isomorphically - we can represent the information about a static case within a dynamic one.



Definition 2.1.3.4

Let $(F, A, T, \hat{\alpha}, \hat{e})$ be a dynamic planning situation. A process $f \in F$ is called *ray-tailed* if there exists a time instance $t \in T$ such that

$$\forall \tau \geq t : f(\tau) = f(t). \qquad (*)$$

The notation $f^{\,t}$ stands for a rail-tailed process with $t \in T$ being the smallest time instance satisfying (*).

□

Definition 2.1.3.5

For a static planning situation (S, A, T, α, e) a *dynamic planning situation* $(F_{S,T}, A, T, \hat{\alpha}, \hat{e})$ *is mirroring* (S, A, T, α, e) if for every $f \in F_{S,T}$, $a \in A$ and $t,\tau \in T$ the relation $\hat{\alpha}$ and the function $\hat{e}$ satisfy

$$\hat{\alpha}(f,(a,t)) = \alpha(f(t),(a,t)),$$

and

$$\hat{e}(f,(a,t))(\tau) = \begin{cases} f(\tau) & \text{if } \tau < t \\ e(f(t),(a,t)) & \text{if } \tau \geq t. \end{cases}$$

The *mirroring mappings* $\tilde{I} : S \rightarrow F$ and $\tilde{J} : F \twoheadrightarrow S$ are defined as follows.

$$\tilde{I}(s)(t) \equiv s \quad \text{for every } s \in S, t \in T$$

and

$\tilde{J}(f^t) = f(t)$ for a ray-tailed process $f^t \in F$.

□

Proposition 2.1.3.6

Let (S, A, T, α, e) be a static planning situation, $(F_{S,T}, A, T, \hat{\alpha}, \hat{e})$, $\tilde{I} : S \rightarrow F$ and $\tilde{J} : F \nrightarrow S$ be a dynamic planning situation and the mappings mirroring (S, A, T, α, e). Then it holds that $e = \tilde{J} \circ \hat{e} \circ \tilde{I}$, that is for every $s \in S$ and $(a,t) \in A \times T$

$$e(s,(a,t)) = \tilde{J}(\hat{e}(\tilde{I}(s),(a,t))).$$

Proof

By definition we have that

$$\tilde{I}(s) \equiv s$$

and

$$\hat{e}(\tilde{I}(s),(a,t))(\tau) = \begin{cases} s & \text{if } \tau < t \\ e(s,(a,t)) & \text{if } \tau \geq t \end{cases}$$

and thus

$$\tilde{J}(\hat{e}(\tilde{I}(s),(a,t))) = e(s,(a,t)).$$

□

Notice, that even if we cannot consider a process of a planning universe as a state of another planning situation, we can regard it as an object of a space that is transformed into another object by a plan. The definition of the transition relation within the space of processes is self-evident.

Definition 2.1.3.7

Let $(F, A, T, \hat{\alpha}, \hat{e})$ be a dynamic planning situation. The *transition relation* » on F is defined for any $f, g \in F$ by

$$f \gg g \quad \text{iff} \quad \exists P \in \mathcal{P}(A \times T) : e'(f, P) = g.$$

□

The relation » is clearly reflexive since

$$\forall f \in F : e'(f,\emptyset) = f$$

holds by definition. With some surprise we realized that transitivity does not hold

for » in general, although intuitively we had expected that if the process f can be turned to g, and g to h then f can be turned to h as well. Next we give a counterexample to show a case when this does not hold.

Example 2.1.3.8
Let us take $(\mathbb{R}_0^+, \{a_1, a_2\}, \mathbb{R}_0^+)$ as a planning universe and let $f,g,h, \in F$ defined by the following. Let $t_1,t_2 \in \mathbb{R}_0^+$ and let α and e be such that:

$\hat{\alpha}(f,(a,t)) = true \;\Leftrightarrow\; [a = a_1 \wedge t = t_1] \vee [a = a_2 \wedge t = t_2]$;

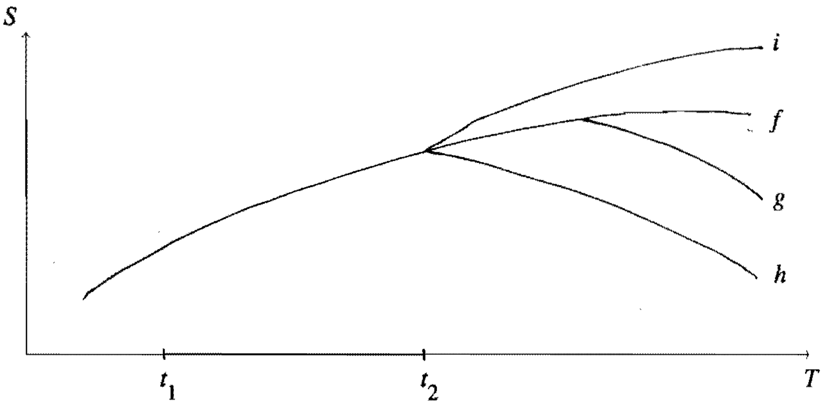
$\hat{\alpha}(g,(a,t)) = true \;\Leftrightarrow\; a = a_2 \wedge t = t_2$;

$\hat{e}(f,(a_1,t_1)) = g$;

$\hat{e}(f,(a_2,t_2)) = i$;

$\hat{e}(g,(a_2,t_2)) = h$.

The next figure illustrates the relationship between the processes f, g, h and i.



On one hand, the general definitions of an allowability relation and an effect function are satisfied here thus this is a possible planning situation.

On the other hand, $\hat{e}(f,(a,t))$ can only be g or i or $\hat{\square}$, which implies

$f \gg g \;\wedge\; g \gg h \;\wedge\; \neg\,(f \gg h)$,

that is transitivity does not hold in this case.

□

Investigating the cause of the subjective 'absurdity' of Example 2.1.3.8 we find it in having $\hat{e}(f,(a_2,t_2)) \neq \hat{e}(g,(a_2,t_2))$ although $f = g$ up to t_2. This discloses that

we intuitively maintain a hidden assumption that is violated here. This assumption informally says that it is only the past and the present that determine a situation, regardless to the future which would have come without external interference. Notice that being in a process, from 'within' we cannot distinguish two processes which have the same history up to now, i.e. at t_2 we cannot tell whether we are in f or g. Therefore we assume that at any moment the set of our possible actions and the effect of the actions is independent from the future.

We admit that one might be reticent about the universal validity of these features. Therefore we do not extend our theory with requiring these properties in general but we formulate them as two assumptions.

Determinative Past Assumption 1 (DPA 1)
Let $(F, A, T, \hat{\alpha}, \hat{e})$ be a dynamic planning situation. The Determinative Past Assumption 1 holds for $(F, A, T, \hat{\alpha}, \hat{e})$ if for any $f,g \in F$ and $t \in T$

$$f \restriction T\underline{t} = g \restriction T\underline{t} \;\Rightarrow\; \hat{\alpha}_{f,t} = \hat{\alpha}_{g,t},$$

where $\hat{\alpha}_{f,t}$ stands for $\{ (a,\tau) \in A \times T \mid \tau = t \wedge \hat{\alpha}(f,(a,\tau)) = true \}$.
□

Determinative Past Assumption 2 (DPA 2)
Let $(F, A, T, \hat{\alpha}, \hat{e})$ be a dynamic planning situation. The Determinative Past Assumption 2 holds for $(F, A, T, \hat{\alpha}, \hat{e})$ if for any $f,g \in F$ and non empty section P such that $time(P) = t$

$$f \restriction T\underline{t} = g \restriction T\underline{t} \;\Rightarrow\; \hat{e}'(f, P) = \hat{e}'(g, P).$$

□

These assumptions are not fully independent as the following proposition indicates.

Proposition 2.1.3.9
For every dynamic planning situation $(F, A, T, \hat{\alpha}, \hat{e})$ DPA 2 implies DPA 1, but the reverse does not necessarily hold.
Proof
Let $(F, A, T, \hat{\alpha}, \hat{e})$ be a dynamic planning situation and assume that DPA 2 holds but DPA 1 does not. Then we can take two processes f, $g \in F$ and $t \in T$ such that

$f \restriction T\underline{t} = g \restriction T\underline{t}$

and

$\hat{\alpha}_{f,t} \neq \hat{\alpha}_{g,t}$.

Without loss of generality we may assume

$\hat{\alpha}_{f,t} \setminus \hat{\alpha}_{g,t} \neq \emptyset$,

thus we can choose an element $(a,t) \in \hat{\alpha}_{f,t} \setminus \hat{\alpha}_{g,t}$. Then for (a,t) we have

$\hat{\alpha}(f,(a,t)) \wedge \neg\, \hat{\alpha}(g,(a,t))$.

Then the $\Rightarrow$ direction of (b) from Definition 2.1.2.6 implies

$\hat{e}(g,(a,t)) = \hat{\square}$

and from the $\Leftarrow$ direction of (b) of Definition 2.1.2.6 it follows that

$\hat{e}(f,(a,t)) \neq \hat{\square}$.

On the other hand DPA 2 implies

$\hat{e}(f,(a,t)) = \hat{e}'(f, P) = \hat{e}'(g, P) = \hat{e}(g,(a,t))$

for $P = \{(a,t)\}$ which is a contradiction.

To see that DPA 1 $\Rightarrow$ DPA 2 does not hold in general let us consider a dynamic planning situation $(F, A, T, \hat{\alpha}, \hat{e})$ for which DPA 1 holds. Let the processes f,g and $t \in T$ be such that $f \neq g$ and

$f \restriction T\underline{t} = g \restriction T\underline{t}$.

it is easy to see that extending $(F\ A, T, \hat{\alpha}, \hat{e})$ by introducing a new action $a' \notin A$ and defining

$\hat{\alpha}(h,(a,\tau)) = \mathit{true} \;\Leftrightarrow\; (h = f \vee h = g) \wedge a = a' \wedge \tau = t$

and

$\hat{e}(f,(a',t)) = g$ and $\hat{e}(g,(a',t)) = f$

results in a planning situation for which DPA 1 does hold but DPA 2 does not.

□

The following proposition proves that DPA 2 is a sufficient condition for the transitivity of the relation ». The proof is constructive, not only stating that there is a plan that turns f into h but also constructing it from the plans that turn f into g and g into h.

Proposition 2.1.3.10

Let $(F, A, T, \hat{\alpha}, \hat{e})$ be a dynamic planning situation. If DPA 2 holds for $(F, A, T, \hat{\alpha}, \hat{e})$ then the relation » on F is transitive.

Proof

Let the processes $f,g,h \in F \setminus \{\hat{o}\}$ and the plans $P,Q \subseteq A \times T$ be such that

$$\hat{e}'(f, P) = g \text{ and } \hat{e}'(g,Q) = h.$$

We show that there exists a plan $R \subseteq A \times T$ such that

$$\hat{e}'(f,R) = h.$$

Let $P \sim [P_1, \ldots, P_n]$ and $Q \sim [Q_1, \ldots, Q_m]$ and let $t_i = time(P_i)$, $i \in \{1, \ldots, n\}$ and
$\tau_i = time(Q_i)$, $i \in \{1, \ldots, m\}$.

i) $\tau_1 \leq t_1$ (Q starts not later than P)

$$\hat{e}'(f, P) = g \Rightarrow f = g \text{ on } T\underline{t}_1$$

by the past invariance proposition (2.1.2.9), so by $\tau_1 \leq t_1$

$$f = g \text{ on } T\underline{\tau}_1$$

is obvious. Then DPA 2 implies that

$$\hat{e}'(f, Q_1) = \hat{e}'(g, Q_1)$$

thus

$$\hat{e}'(f, Q) = \hat{e}'(\ldots \hat{e}'(f, Q_1) \ldots, Q_m) = \hat{e}'(\ldots \hat{e}'(g, Q_1) \ldots, Q_m) = \hat{e}'(g, Q),$$

that is $R = Q$ is satisfactory.

ii) $\tau_1 > t_1$ (Q starts later than P)

Let $f_0 = f$, $f_i = e'(f_{i-1}, P_i)$ for every $i \in \{1, \ldots, n\}$. By the iterative application of the past invariance property (Proposition 2.1.2.9) we obtain

$$f_i = g \text{ on } T\underline{t}_{i+1} \text{ for every } i \in \{1, \ldots, n\text{-}1\}.$$

Let k be such that $k = max\ \{\ i \mid t_i < \tau_1\ \}$.

Then

$$t_k < \tau_1 \leq t_{k+1}$$

and

$$f_k = g \text{ on } T\underline{t}_{k+1}$$

imply that

$$f_k = g \text{ on } T\underline{\tau}_1.$$

Observe that DPA 2 leads to

$$\hat{e}'(f,P_1 \cup \ldots \cup P_k \cup Q) = \hat{e}'(e' \ldots (e'(f, P_1) \ldots, P_k), Q) = \hat{e}'(f_k, Q) = h,$$

that is for $R = P_1 \cup \ldots \cup P_k \cup Q$ we obtain

$\hat{e}'(f, R) = h.$

□

2.2 Examples of Planning Problems

Section 2.1 presents two kinds of models of planning problems. A static model is appropriate if we can assume that any state lasts until an action is committed. If, however, we foresee that there are states that are maintained only for a certain period of time then a dynamic model can describe the case. In section 2.1 states, actions etc. were primitives of the theory. Here in section 2.2 we are going to have a look at the 'inside' of these primitives, that is we give a detailed description of five planning problems.

For any description method it is very important that it is clear enough and relatively simple entities are used to describe a problem. This makes it easy to decide whether the formal description matches the intuitive interpretation of the problem, in other words this makes a formal description a good interface between intuition and formal treatment of the problem. Using the theoretical model as a description framework imposes a certain method of problem specification. Describing a problem through defining $S, A, T, \alpha, e, s, \gamma, \kappa$ implies that we can concentrate on a relatively small aspect of the problem at a time, e.g. what the world states should be like, or which conditions should hold before the application of an action. The whole model is then composed by these relatively simple components.

To characterize a state as a snapshot about the world at a moment it seems to be natural to list all those facts that hold in the world at that moment. Making just one step further we come to the idea of saying that a state IS a set of valid facts. We complete this view with the so called Closed World Assumption, assuming that a state s contains all the valid facts, i.e. if a fact is not contained in s then it is not true in s. Hereby we take a logic-based approach that enables us to handle facts, complex statements and validity with respect to states, cf. Pednault (1987), Treur (1988), van Langen and Treur (1989).

Before we begin the formal work we want to draw the readers attention to certain important aspects of the following examples.

As we have mentioned after Remark 2.1.1.3 there are two kinds of facts: *permanent* ones that are not changed by an action and *temporary* ones that can be modified if an action is committed. Incorporating permanent facts in the states is superfluous, therefore in our construction we distinguish permanent functions and relations and temporary functions and relations.

Permanent relations belong to the background information; we presume that they are stored in a kind of database and can be directly quoted without any reference to the actual state. Therefore, we do not distinguish the relation $R \subseteq A \times B$ and the corresponding relation symbol, but will also use the notation $R(a,b)$ meaning the appropriate Boolean value. Similarly, we assume about permanent functions that they are always computed, that is if $D : A \times B \rightarrow \mathbb{R}$ is a permanent function then an expression of the form $D(a,b)$ denotes a real number.

By their very nature, temporary relations can not be given a truth value without a reference to the actual state. Therefore, for every temporary relation R we introduce a corresponding relation symbol r with the same arity and define a state as a set of ground atoms constructed from these relation symbols. To interpret truth w.r.t. states we assume that a state is a complete collection of facts true in it, i.e. we assume that $r(x) \in s$ if and only if $R(x)$ holds in s.

To exclude infinite features we shall avoid the use of temporary functions. We use function symbols only to denote names of actions, e.g. $to(x,y)$ will be the name of the action of going from x to y. These temporary function symbols are purely syntactic: an expression of the form $to(x,y)$ has no value, it is but a name.

In the sequel, names of function and relation symbols in upper case indicate that we consider them as semantic objects, names in lower case stand for syntactic objects.

Finally, let us make some abbreviations to simplify the notation. As it turns out from the foregoing, we use temporary relation and function symbols for special purposes. They are seen as purely syntactical objects, therefore we shall use an abbreviated way of set construction, namely:

$\{ r(x_1, \ldots, x_n) \mid x_1 \in X_1, \ldots, x_n \in X_n \}$

instead of

$\{ expression \in Expr \mid \exists x_1 \in X_1 \ldots \exists x_n \in X_n : expression = r(x_1, \ldots, x_n) \}$,

where *Expr* denotes the set of all expressions used.

We use two other notational conventions for *there exists one and only one* and

there exists at most one. For the sake of convenience

$\exists!\, x \in X : \varphi(x)$

abbreviates the formula

$[\exists\, x \in X : \varphi(x)] \quad \wedge \quad [\forall\, x,y \in X : \varphi(x) \wedge \varphi(y) \Rightarrow x = y]$,

while

$\exists?\, x \in X : \varphi(x)$

stands for

$\forall\, x,y \in X : \varphi(x) \wedge \varphi(y) \Rightarrow x = y$.

2.2.1 Travelling Salesman Problem

Based on the informal description in Example 2.1.1.2 we distinguish the following relevant entities of the world:

- $Z = \{z_1, \ldots, z_n\}$ is the set of constant symbols denoting the cities;
- $D : Z \times Z \rightarrow \mathbb{R}_0^+$ is a permanent distance function for cities;
- *AT* is a temporary unary relation pointing out the city where the agent is;
- *SEEN* is a temporary unary relation to mark cities that have already been visited.

Observe that no agent or salesman is mentioned in this description. Indeed, if we only presume one agent, he can be simply omitted. We, however, will mention it sometimes as if it was present, just to make explanations easier.

Formalizing the above view on the world we introduce

- *at*, a unary relation symbol corresponding to the relation *AT*;
- *seen*, a unary relation symbol corresponding to the relation *SEEN*;
- *to*, a binary function symbol (name), *to*(*x*,*y*) representing the action of going from city *x to city y*.

Definition 2.2.1.1

A pre-state is a set V

$V \subseteq \{\ at(x) \mid x \in Z\ \} \cup \{\ seen(x) \mid x \in Z\ \}$.

The set of all pre-states will be denoted by S_p.

□

Definition 2.2.1.2

A pre-state $V \in S_p$ is called *correct* if the agent is at one location at a time, that is if

$$\exists!\, x \in Z : at(x) \in V.$$

□

We assume that the position of the agent and the status 'being seen' of a city can only change by performing an action, that is we choose the static model version.

The constituents of the planning universe are :

$S = \{ V \in S_p \mid V \text{ is correct} \};$

$A = \{ to(x,y) \mid x,y \in Z \};$

$T = \mathbb{N}.$

Definition 2.2.1.3

The *allowability relation* for $s \in S, t \in T, x,y \in Z$ is defined by

$\alpha(s, (to(x,y),t)) \Leftrightarrow$

1) $at(x) \in s$ and

2) $seen(y) \notin s$.

□

Definition 2.2.1.4

For every $s \in S, t \in T, x,y \in Z$, if $\alpha(s,(to(x,y),t))$ then

$$e(s, (to(x,y),t)) = (s \setminus \{at(x)\}) \cup \{at(y), seen(y)\}.$$

□

Let us have a look on the role of *seen*. We could have chosen world states describing only the actual position of the agent, that is containing only the *at* predicate. However, restricting ourselves to the static model would then imply that the information about the past (where the agent has already been) would be lost after each state transition. Therefore we incorporated *seen* in the states. Notice that a modelling decision was taken that says that a location becomes seen when arriving at it.

Definition 2.2.1.5

A planning problem describing the travelling salesman problem can be given by the following items.

$s_0 = \{at(z_1)\};$

$\gamma(s) \Leftrightarrow at(z_1) \in s \wedge \forall z \in Z : seen(z) \in s;$

$\kappa(P) = \sum_{i=1}^{m} D(x_i,y_i)$

for any arbitrary $P = \{ (to(x_1,y_1),t_1), \ldots, (to(x_m,y_m),t_m) \}$.

□

It is common for TSPs that not a minimal value of κ is required only a κ value under a certain border $B > 0$. This, however, does not make TSP easier in the sense that it remains NP-complete, cf. Garey and Johnson (1979).

Proposition 2.2.1.6

For any $V \in S_p$, $a \in A$ and $t \in T$, if V is correct and $\alpha(V,(a,t))$, then $e(V,(a,t)) \in S_p$ is correct too.

Proof

By Definition 2.2.1.4 an operation does not change the number of expressions of the form $at(z)$ in a state.

□

2.2.2 Travelling Salesman Problem with Time Windows

Recall Remark 2.1.1.3 where we stated that the extension of TSP with time windows can not be expressed in a static model. Therefore we develop a dynamic model for TSP with time windows, cf. Savelsbergh (1988).

We distinguish the following relevant entities of the world:

- $Z = \{z_1, \ldots, z_n\}$ is the set of constant symbols denoting the cities;
- $D : Z \times Z \to \mathbb{R}_0^+$ is a permanent distance function for cities;
- $v \in \mathbb{R}^+$ is the standard velocity of moving;
- $W : Z \to \mathcal{P}(\mathbb{N} \times \mathbb{N})$, such that for every $z \in Z$ and $w \in W(z)$, $w = (w.1,w.2)$ $w.1 \leq w.2$ holds and the interval $[w.1, w.2]$ is a time window

belonging to z;

- *AT* is a temporary unary relation pointing out the location where the agent is;
- *SEEN* is a temporary unary relation to mark cities that have already been visited;
- *UNDERWAY* is a temporary zeroary relation that stands for being between two cities.

Formalizing the above view on the world we introduce

- *at*, a unary relation symbol corresponding to the relation *AT*;
- *seen,* a unary relation symbol corresponding to the relation *SEEN*;
- *underway*, a zeroary relation symbol corresponding to the relation *UNDERWAY*;
- *to*, a binary function symbol (name), $to(x,y)$ representing the action of going from city x to city y.

Definition 2.2.2.1

A *pre-state* is a set V

$$V \subseteq \{\ at(x) \mid x \in Z\ \} \cup \{\ seen(x) \mid x \in Z\ \} \cup \{underway\}$$

The set of all pre-states is denoted by S_p.

□

Definition 2.2.2.2

A pre-state $V \in S_p$ is called *correct* iff

1) $\neg\ (underway \in V\ \wedge\ \exists\, x \in Z : at(x) \in V)$ and
2) $\exists ?\, x \in Z : at(x) \in V$.

□

The constituents of the planning universe are:

$$S = \{\ V \in S_p \mid V\ is\ correct\ \};$$
$$A = \{\ to(x,y) \mid x,y \in Z\ \};$$
$$T = \mathbb{N}.$$

Let F denote the set of all processes of this planning universe.

Definition 2.2.2.3

A process $f \in F$ is *correct* if $f(t)$ is correct for every $t \in dom(f)$.

□

Definition 2.2.2.4

The *allowability relation* for $f \in F$, $t \in dom(f)$, $x,y \in Z$ is defined by

$\hat{\alpha}(f, (to(x,y),t)) \Leftrightarrow$

1) $at(x) \in f(t)$ *and*
2) $seen(y) \notin f(t)$ *and*
3) $\exists\, w \in W(y) : w.1 < t + D(x,y)/v < w.2$.

□

Definition 2.2.2.5

For every $f \in F$, $t \in dom(f)$, $x,y \in Z$, if $\alpha(f,(to(x,y),t))$ then

$$\hat{e}(f, (to(x,y),t))(\tau) = \begin{cases} f(\tau) & \tau < t \\ [\, f(t) \setminus \{at(x)\}\,] \cup \{underway\} & t \le \tau < t + D(x,y)/v \\ f(t) \cup \{at(y),\, seen(y)\} & t + D(x,y)/v \le \tau \end{cases}$$

□

Definition 2.2.2.6

A planning problem describing the travelling salesman problem with time windows can be given by the following items.

$f_0(t) = \{at(z_1)\}$ for every $t \in T$;

$\gamma(f) \Leftrightarrow \exists\, t \in T : [\, at(z_1) \in f(t) \wedge \forall z \in Z : seen(z) \in f(t)\,]$;

$$\kappa(P) = \sum_{i=1}^{m} D(x_i,y_i)$$

for any arbitrary $P = \{\, (to(x_1,y_1),t_1), \ldots, (to(x_m,y_m),t_m)\, \}$.

□

Proposition 2.2.2.7

For any process f, $a \in A$ and $t \in dom(f)$, if f is correct and $\hat{\alpha}(f,(a,t))$, then $\hat{e}(f,(a,t))$ is correct too.

Proof

It is trivial by Definition 2.2.2.4 and Definition 2.2.2.5.

□

2.2.3 Precedence Constrained Scheduling Problem

The problem we have sketched in Example 2.1.2.1 leads to the following items:
- M and J are sets of constant symbols to denote the machines and the jobs;
- $PRE \subseteq J \times J$, a permanent relation prescribing the precedence between jobs;
- $ABLE \subseteq M \times J$ a permanent relation showing that a machine can perform a job;
- $D : ABLE \rightarrow \mathbb{R}_0^+$ a permanent function that indicates the duration of the performance of a job on a machine;
- $BUSY \subseteq M \times J$ a temporary relation to tell that a machine is working on a job;
- $READY \subseteq J$ a temporary relation to indicate that a job has been completed.

In the formal description we shall use:

- *busy*, a binary relation symbol corresponding to the relation $BUSY$;
- *ready*, a unary relation symbol corresponding to $READY$;
- *begin*, a binary function symbol (name) denoting the action of beginning a job on a machine.

Definition 2.2.3.1

A pre-state is a set V

$$V \subseteq \{ \, busy(m,j) \mid m \in M, j \in J \, \} \cup \{ \, ready(j) \mid j \in J \, \}.$$

The set of all pre-states is denoted by S_p.

□

The intention is clear, world states are the possible snapshots during the job performing process. The situation at a time instance, however, is not fully determined by the ongoing activities. Jobs completed earlier are influencing the situation as well. Hence, a choice needs to be made between either checking the history before decisions, or defining a representative of the relevant aspects of the history and completing the snapshots with it. We have chosen the second possibility, this explains the role of *ready*.

The world states are constructed such that in any state

- a machine is doing a job only if it is able to do that job and
- a machine is doing at most one job and
- a job is being done on at most one machine and
- ready jobs are not being performed.

Definition 2.2.3.2

A pre-state $V \in S_p$ is called *correct* iff

1) $\forall (m,j) \in M \times J : busy(m,j) \in V \Rightarrow ABLE(m,j)$ and
2) $\forall m \in M \exists ? j \in J : busy(m,j) \in V$ and
3) $\forall j \in J \exists ? m \in M : busy(m,j) \in V$ and
4) $\forall j \in J : [ready(j) \in V \Rightarrow \neg\exists m \in M : busy(m,j) \in V]$.

We take a planning universe consisting of:

$S = \{ V \in S_p \mid V \textit{ is correct} \}$;

$A = \{ begin(m,j) \mid m \in M, j \in J \}$;

$T = \mathbb{R}_0^+$.

Furthermore, for reasons discussed in section 2.1.2 the we take a dynamic model and denote the set of all processes corresponding to this universe by F.

The conditions to begin a job j on a machine m at a time instance t are :

- j is not ready yet;
- m has the ability to perform j;
- all the predecessors of j are ready;
- j is free at t;
- m is free at t.

Definition 2.2.3.3

For every $f \in F$, $m \in M$, $j \in J$ and $t \in dom(f)$

$\hat{\alpha}(f, (begin(m,j),t)) \Leftrightarrow$

1) $ready(j) \notin f(t)$ and
2) $ABLE(m,j) \wedge$ and
3) $\forall j' \in J : [PRE(j',j) \Rightarrow ready(j') \in f(t)]$ and
4) $\neg\exists m' \in M : busy(m',j) \in f(t)$ and
5) $\neg\exists j' \in J : busy(m,j') \in f(t)$.

The effect of an allowed operation ***begin***(x,y) is that x performs y and that some of the events that would have happened disappear from the future.

Definition 2.2.3.4

For every $f \in F$, $m \in M$, $j \in J$ and $t \in dom(f)$ if $\hat{\alpha}(f,(begin(m,j),t))$ then

$$\hat{e}(f,(begin(m,j),t))(\tau) = \begin{cases} f(\tau) & \tau < t \\ [f(\tau) \setminus (busy(m,.) \cup busy(.,j) \cup \{ready(j)\})] \cup \{busy(m,j)\} & t \le \tau < t+D(m,j) \\ [f(t) \setminus busy(.,j)] \cup \{ready(j)\} & t+D(m,j) \le \tau \end{cases}$$

where

$busy(m,.)$ abbreviates the set $\{busy(m,j) \mid j \in J\}$

and

$busy(.,j)$ abbreviates the set $\{busy(m,j) \mid m \in M\}$.

Definition 2.2.3.5

A planning problem describing the PCSP problem can be determined by :

$f_0(t) = \emptyset$ for all $t \in T$;

$\hat{\gamma}(f) \Leftrightarrow \exists\, t \in T\ \forall\, j \in J : ready(j) \in f(t)$;

$\kappa(P) = max\ \{\ t + D(m,j) \mid t \in T, m \in M, j \in J, (begin(m,j),t) \in P\ \}$.

□

It is usual for PCSPs that not an optimal plan is required, only a plan that is completed before a certain deadline D, i.e. a plan with $\kappa(P) \le D$, cf. Garey and Johnson (1979).

Proposition 2.2.3.6

If $f \in F$, $(a,t) \in A \times T$ and $\hat{\alpha}(f,(a,t))$ holds then $\hat{e}(f,(a,t)) \in F$ as well.

Proof

By simple case analysis based on Definition 2.2.3.3 and Definition 2.2.3.4.

□

2.2.4 Time Table Problem (TTP)

In a TTP we have to make the weekly schedule of a finite set of teachers, subjects, classrooms and group of students. Here we develop a more complicated model than the classic ones from Even, Itai and Shamir (1976); Garey and Johnson (1979). We divide the week into non overlapping lecture periods with equal length L. Furthermore we know which teachers are qualified to give which subjects and how many lectures of a certain subject does a group of students need. The objective is to assign teachers, lectures, classrooms and time periods over a week such that every subject is given by a teacher qualified for it and every group of students gets the required number of lectures of every subject. Besides to this basic aim we also want to satisfy a didactic and organizational goal, in particular we want a time table that spreads the same subject over the whole week.

Within the world we distinguish
- G, a finite set of groups of students;
- Z, a finite set of subjects;
- D, a finite set of teachers;
- K, a finite set of classrooms;
- $H = \{ h_1, \ldots, h_M \}$, a finite set of not overlapping lecture hours with the same length $L > 0$ numbered consecutively such that if $t_i > 0$ denotes the beginning time of h_i then
  - $h_i = \{ x \in \mathbb{R} \mid t_i \leq x < t_i + L \}$ and
  - $t_i + L < t_{i+1}$;
- $ABLE \subseteq D \times Z$, is a permanent relation to represent which teachers are qualified to give which subjects;
- $N : G \times Z \rightarrow \mathbb{N}$, is a permanent function, $N(g,z)$ denoting the number of lectures of the subject z the group g needs to get in a week;
- $BUSY \subseteq G \times Z \times D \times K$, a temporary relation expressing that a group is receiving a subject from a teacher in a classroom;
- $GIVEN \subseteq G \times Z \times D \times \mathbb{N}$, a temporary relation denoting how many times a group has received a subject from a teacher already.

On this basis we further introduce

- *busy*, a 4-ary relation symbol that corresponds to the relation *BUSY*;
- *given*, a 4-ary relation symbol corresponding to the relation *GIVEN*;
- *begin*, a 4-ary function symbol (name) to denote the action of beginning to give a subject to a group by a teacher in a classroom.

Definition 2.2.4.1

A *pre-state* is a set V

$$V \subseteq \{ busy(g,z,d,k) \mid g \in G,\ z \in Z,\ d \in D,\ k \in K \} \cup \{ given(g,z,d,n) \mid g \in G,\ z \in Z,\ d \in D,\ n \in \mathbb{N} \}.$$

The set of all pre-states is denoted by S_p.

□

We develop a dynamic model, where the world states are to describe ongoing activities such that in every state

- the teacher is qualified to teach the subject he is giving;
- a group gets a subject only if it is needed;
- the same group always gets the same subject from the same teacher;
- no subject is given more times then needed;
- one group is only busy with one thing at one place;
- one teacher is only busy with one thing at one place;
- one classroom is only occupied for one activity.

Definition 2.2.4.2

A pre-state $V \in S_p$ is called *correct* iff

1) $\forall\, (g,z,d,k) \in G \times Z \times D \times K : busy(g,z,d,k) \in V \Rightarrow ABLE(d,z)$ and
2) $\forall\, (g,z,d,k) \in G \times Z \times D \times K : busy(g,z,d,k) \in V \Rightarrow N(g,z) > 0$ and
3) $\forall\, (g,z,d,k) \in G \times Z \times D \times K :$
$busy(g,z,d,k) \in V \Rightarrow (\neg \exists\, d' \in D\ \exists\, n \in \mathbb{N} : d' \neq d\ \wedge given(g,z,d',n) \in V)$ and
4) $\forall\, (g,z,d,k) \in G \times Z \times D \times K : busy(g,z,d,k) \in V \Rightarrow given(g,z,d,N(g,z)) \notin V$ and
5) $\forall\, g \in G\ \exists?\, (z,d,k) \in Z \times D \times K : busy(g,z,d,k) \in V$ and
6) $\forall\, d \in D\ \exists?\, (g,z,k) \in G \times Z \times K : busy(g,z,d,k) \in V$ and
7) $\forall\, k \in K\ \exists?\, (g,z,d) \in G \times Z \times D : busy(g,z,d,k) \in V$.

At (4) recall that N is a permanent function, thus $N(g,z)$ is a real number.

□

The planning universe is then (S, A, T), where

$S = \{ V \in S_p \mid V \text{ is correct } \};$

$A = \{ begin(g,z,d,k) \mid g \in G, z \in Z, d \in D, k \in K \};$

$T = \mathbb{R}_0^+.$

Furthermore, let F denote the set of all processes of this planning universe.

Definition 2.2.4.3

For every $f \in F$, $g \in G$, $z \in Z$, $d \in D$, $k \in K$, $t \in dom(f)$

$\hat{\alpha}(f,(begin(g,z,d,k),t)) \Leftrightarrow$

1) $ABLE(d,z)$ and
2) $N(g,z) > 0,$ and
3) $\neg\exists\, d' \in D\ \exists\, n \in \mathbb{N} : d' \neq d \ \wedge given(g,z,d',n) \in f(t)$ and
4) $given(g,z,N(g,z) \notin f(t)$ and
5) $\neg\exists\, (z',d',k') \in Z \times D \times K : (z',d',k') \neq (z,d,k) \wedge busy(g,z',d',k') \in f(t)$ and
6) $\neg\exists\, (g',z',k') \in G \times Z \times K : (g',z',k') \neq (g,z,k) \wedge busy(g',z',d,k') \in f(t)$ and
7) $\neg\exists\, (g',z',d') \in G \times Z \times D : (g',z',d') \neq (g,z,d) \wedge busy(g',z',d',k) \in f(t)$ and
8) $\exists\, h_i \in H : t = t_i$.

□

Definition 2.2.4.4

For every $f \in F$, $g \in G$, $z \in Z$, $d \in D$, $k \in K$, $t \in dom(f)$ if $\hat{\alpha}(f,(begin(g,z,d,k),t))$ holds then

$$\hat{e}(f,(begin(g,z,d,k),t))(\tau) = \begin{cases} f(\tau) & \tau < t \\ f(\tau) \cup \{busy(g,z,d,k)\} & t \leq \tau < t + L\ , \\ f(\tau)^* & t + L \leq \tau \end{cases}$$

where $f(\tau)^* =$

$$= \begin{cases} f(\tau) \cup \{given(g,z,d,1)\} & \text{if } \neg\exists\, x \in \mathbb{N} : given(g,z,d,x) \in f(t) \\ [f(\tau) \setminus \{given(g,z,d,x)\}] \cup \{given(g,z,d,x+1)\} & \text{if } x \in \mathbb{N} \text{ and } given(g,z,d,x) \in f(t) \end{cases}$$

□

Definition 2.2.4.5

The planning problem that describes the TTP can be determined by the following.

$f_0(t) = \emptyset$ for all $t \in T$;

$\hat{\gamma}(f) \Leftrightarrow \forall g \in G \; \forall z \in Z \; \exists d \in D : given(g,z,d,N(g,z)) \in f(h_M+L)$.

The criterion κ is constructed from two other criteria κ_1 and κ_2 that measure the number of 'double' and 'triple' lectures.

$$\kappa_1(P) = \sum_{g \in G} \sum_{z \in Z} \sum_{d \in D} |\{ i \in \{1,...,M-1\} \mid \exists k,k' \in K : (begin(g,z,d,k),t_i) \in P \;\wedge\; (begin(g,z,d,k'),t_{i+1}) \in P \}|$$

$$\kappa_2(P) = \sum_{g \in G} \sum_{z \in Z} \sum_{d \in D} |\{ i \in \{1,...,M-2\} \mid \exists k,k',k'' \in K : (begin(g,z,d,k),t_i) \in P \;\wedge\; (begin(g,z,d,k'),t_{i+1}) \in P \;\wedge\; (begin(g,z,d,k''),t_{i+2}) \in P \}|$$

$\kappa(P) = \kappa_1(P) + 10 \cdot \kappa_2(P)$.

□

Proposition 2.2.4.6

If $f \in F$ and $\hat{\alpha}(f,(a,t))$ holds for the operation (a,t) then $\hat{e}(f,(a,t)) \in F$ too.

Proof

Let $a = begin(g,z,d,k)$ for some $g \in G$, $z \in Z$, $d \in D$ and $k \in K$, $\tau \in T$ arbitrary. We have to verify that the conditions (1), . . .,(7) of Definition 2.2.4.2 hold for $e(f,(a,t))(\tau)$.

For $\tau < t$ it is obvious, since $e(f,(a,t))(\tau) = f(\tau)$ by Definition 2.2.4.4.

If $t \leq \tau < t + M$ then

$\hat{e}(f,(a,t))(\tau) = f(\tau) \cup \{busy(g,z,d,k)\}$

by definition and it is easy to see that (1), . . .,(7) of Definition 2.2.4.3 are sufficient to

imply (1), . . .,(7) of Definition 2.2.4.2.

If $t + M \leq \tau$ then we only have to check (2) and (4) from Definition 2.2.4.1 since $f(\tau)^*$ only differs from $f(\tau)$ by an atom of the form $given(g,z,y)$. In this case it is enough to notice that

$\forall \; busy(g,z,d,k) \in e(f,(a,t))(\tau) : busy(g,z,d,k) \in f(\tau)$

thus if (2) and (4) would not hold for $e(f,(a,t))(\tau)$ then they would not hold for $f(\tau)$ either.

□

2.2.5 Ship Loading Problem

In this problem we have a ship visiting a set of harbours, loading and unloading containers at each harbour, cf. van Hee (1985). Knowing the trip of the ship, the load- and unload needs of the harbours and assuming the ship is empty at the beginning, we need to make a loading plan for the harbours such that
- all the load- and unload needs of the harbours are met;
- the loading and unloading work is minimal;
- the ship always remains stable.

To make a world description we need
- $H = \{h_1, \ldots, h_n\}$ a set of harbours numbered in the order the ship is visiting them;
- U a finite set of units (containers);
- $W : U \rightarrow \mathbb{R}_0^+$, a permanent weight function on units;
- $X \in \mathbb{N}$, $Y \in \mathbb{N}$, $Z \in \mathbb{N}$, standing for the length, width and height of the block shaped storage depot of the ship;
- $LN : H \rightarrow \mathcal{P}(U)$, a permanent function giving the load needs of the harbours;
- $UN : H \rightarrow \mathcal{P}(U)$, a permanent function giving the unload needs of the harbours, such that

$$\forall i \in \{1, \ldots, n\} : [\, LN(h_i) \cap UN(h_i) = \emptyset \ \wedge \ UN(h_i) \subseteq \bigcup_{j<i} LN(h_j) \,];$$

- $ONSHIP \subseteq U \times \{1, \ldots, X\} \times \{1, \ldots, Y\} \times \{1, \ldots, Z\}$ a temporary relation to describe the position of the units on the ship;
- $INHARBOUR \subseteq U \times H$ a temporary relation to describe units at the harbours;
- $AT \subseteq H$ a temporary relation showing the position of the ship.

To the temporary relations and for the actions we define
- *onship*, a 4-ary predicate symbol corresponding to *ONSHIP*;
- *inharbour*, a binary predicate symbol corresponding to *INHARBOUR*;
- *at*, a unary predicate symbol corresponding to *AT*;

- *move*, a binary function symbol denoting a move from a harbour to another harbour;
- *load*, a 5-ary function symbol denoting the loading of a unit at a harbour to a 3 dimensional position;
- *unload*, a 5-ary function symbol denoting the unloading of a unit at a harbour from a 3 dimensional position.

Definition 2.2.5.1

Pre-states are defined as subsets of the following set:

$\{ at(h) \mid h \in H \} \cup$

$\{ onship(u,x,y,z) \mid u \in U, x \in \{1,...,X\}, y \in \{1,...,Y\}, z \in \{1,...,Z\} \} \cup$

$\{ inharbour(u,h) \mid u \in U, h \in H \}$.

The set of all pre-states is again denoted by S_p.

□

To make the formulae shorter in the sequel we shall use

$Q\ (x,y,z)$:

to abbreviate

$Q\, x \in \{1,...,X\}\ Q\, y \in \{1,...,Y\}\ Q\, z \in \{1,...,Z\}$:

where Q is the quantifier $\forall$, $\exists$, $\exists!$ or $\exists?$.

Definition 2.2.5.2

$V \in S_p$ is *correct* iff

1) $\exists!\ h \in H : at(h) \in V$ and
2) $\forall\ u \in U\ \exists?\ x,y,z : onship(u,x,y,z) \in V$ and
3) $\forall\ x,y,z\ \exists?\ u \in U : onship(u,x,y,z) \in V$ and
4) $\forall\ u \in U\ \forall\ (x,y,z)$:

   $onship(u,x,y,z) \in V \Rightarrow [\ z > 1 \Rightarrow \exists\ v \in U : onship(v,x,y,z\text{-}1) \in V\ ]$ and
5) $\forall\ u \in U\ \forall\ (x,y,z) : onship(u,x,y,z) \in V \Rightarrow [\neg\ \exists\ h \in H : inharbour(u,h) \in V\ ]$ and
6) $\forall\ (u,h) \in U \times H : inharbour(u,h) \in V \Rightarrow [\ \neg\ \exists\ (x,y,z) : onship(u,x,y,z) \in V\ ]$ and
7) $\forall\ u \in U\ \exists?\ h \in H : inharbour(u,h) \in V$.

□

$S = \{ V \in S_p \mid V\ is\ correct \}$;

$A =$
$\{ move(h,h') \mid h,h' \in H \} \cup$
$\{ load(u,h,x,y,z) \mid u \in U, h \in H, x \in \{1, \ldots, X\}, y \in \{1, \ldots, Y\}, z \in \{1, \ldots, Z\} \} \cup$
$\{ unload(u,h,x,y,z) \mid u \in U, h \in H, x \in \{1, \ldots, X\}, y \in \{1, \ldots, Y\}, z \in \{1, \ldots, Z\} \}$;
$T = \mathbb{N}$.

We observe that a state can only be changed by actions, thus we develop a static model. The allowability condition is defined for each action.

Definition 2.2.5.3

a) $\alpha(s,(move(h_i,h_j),t)) \Leftrightarrow$

1) $at(h_i) \in s$ and
2) $j = i + 1$ and
3) $0.8 \leq \dfrac{m_{left}(s)}{m_{right}(s)} \leq 1.2$ and
4) $0.8 \leq \dfrac{m_{for}(s)}{m_{back}(s)} \leq 1.2$,

where the latter two conditions are to guarantee the stability of the ship, having

$$m_{left}(s) = \sum_{y=1}^{Y} \sum_{x=1}^{\lfloor X/2 \rfloor} \sum_{z=1}^{Z} \sum_{\{ u \in U \mid onship(u,x,y,z) \in s \}} w(u) \cdot (\lfloor X/2 \rfloor + 1 - x) + \frac{X \cdot Y \cdot Z}{4};$$

and

$$m_{right}(s) = \sum_{y=1}^{Y} \sum_{x=\lfloor X/2 \rfloor + 1}^{X} \sum_{z=1}^{Z} \sum_{\{ u \in U \mid onship(u,x,y,z) \in s \}} w(u) \cdot (x - \lfloor (X+1)/2 \rfloor) + \frac{X \cdot Y \cdot Z}{4}$$

$$m_{for}(s) = \sum_{x=1}^{X} \sum_{y=1}^{\lfloor Y/2 \rfloor} \sum_{z=1}^{Z} \sum_{\{ u \in U \mid onship(u,x,y,z) \in s \}} w(u) \cdot (\lfloor Y/2 \rfloor + 2 - y) + \frac{X \cdot Y \cdot Z}{4};$$

and

$$m_{back}(s) = \sum_{x=1}^{X} \sum_{y=\lfloor Y/2 \rfloor+1}^{Y} \sum_{z=1}^{Z} \sum_{\{u \in U \mid onship(u,x,y,z) \in s\}} w(u) \cdot (y - \lfloor (Y+1)/2 \rfloor) + \frac{X \cdot Y \cdot Z}{4}$$

where $\lfloor x \rfloor$ stands for the standard entier function.

b) $\alpha(s,(load(u,h,x,y,z),t)) \Leftrightarrow$
   1) $at(h) \in s$ and
   2) $inharbour(u,h) \in s$, and
   3) $\neg\exists\, v \in U : onship(v,x,y,z) \in s$ and
   4) $\exists\, v \in U : onship(v,x,y,z\text{-}1) \in s$.

c) $\alpha(s,(unload(u,h,x,y,z),t)) \Leftrightarrow$
   1) $at(h) \in s$ and
   2) $onship(u,x,y,z) \in s$, and
   3) $\neg\exists\, v \in U : onship(v,x,y,z+1) \in s$.

□

Definition 2.2.5.4

The effect of allowed operations is as follows:

a) $e(s,(move(h_i,h_j),t)) = (s \setminus \{at(h_i)\}) \cup \{at(h_j)\}$,

b) $e(s,(load(u,h,x,y,z),t)) = (s \setminus \{inharbour(u,h)\}) \cup \{onship(u,x,y,z)\}$,

c) $e(s,(unload(u,h,x,y,z),t)) = (s \setminus \{onship(u,x,y,z)\}) \cup \{inharbour(u,h)\}$.

□

Definition 2.2.5.5

A ship loading problem is specified by the above (S,A,T,α,e) and the following s_0, γ and κ.

$s_0 = \{ at(h_1) \}$;

$\gamma(s) \Leftrightarrow \forall\, u \in U\ \forall\, h \in H : u \in LN(h) \Rightarrow inharbour(u,h) \notin s$ and
$\forall\, u \in U\ \forall\, h \in H : u \in UN(h) \Rightarrow inharbour(u,h) \in s$ }.

The objective function κ is to measure the total work done by a plan:

$\kappa(P) = |P|$.

□

Proposition 2.2.5.6
For any $V \in S_p$, $a \in A$ and $t \in T$, if V is correct and $\alpha(V,(a,t))$ then $e(V,(a,t))$ is correct too.

Proof
It is straightforward by case analysis. Let us show here the case of a = *unload*(u,h,x,y,z). Then it holds that for the items of Definition 2.2.5.2

(1) is obvious;

(2), (3) and (5) follow from the fact that *e* deletes an *onship* from the correct *V*;

(4) holds since by (3) of (c) of the Definition 2.2.5.3 we always unload from the top of a stack and *V* is correct;

(6) is guaranteed by (c) of Definition 2.2.5.4;

(7) follows from (2) of (c) of Definition 2.2.5.3 and (5) from Definition 2.2.5.2.

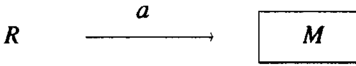
□

After having completed these five examples an articulated method of planning problem definition has arisen. This method of specifying planning problems by means of the model of section 2.1 is

- general, it applies to all our cases, and it seems sound to presume that it will be satisfactory to other planning problems as well;
- facilitating clear understanding of the problem at hand by supporting and also forcing precise analysis;
- modular, that is we can concentrate on one simple aspect at a time and the whole problem description is composed by the general model.

This method and its application in DSS development will be further discussed in Chapter 6. An important result of the above examples is that they provide an insight of the structure of the parameters, for instance how allowability relations look like and how they can be defined. This insight can be the basis of designing a formal language for defining planning problems.

Finally let us mention a special aspect of formal planning problems. Namely, such a planning problem can be viewed as the interface between reality and formal problem solving. It is well-known that the modelling step we make from a real problem *R* to a formal model *M* is of crucial importance.



If the model M does not describe the relevant parts of R appropriately, then all further computational efforts using M can be done for nothing. It is therefore quite an unpleasant fact that the correctness of such a modelling step cannot be rigorously proved, only intuitively justified. The reason for this is trivial: since one end of the arc a is an informal entity, we cannot establish formal relationships along a, i.e. between R and M. Nevertheless, once we have created a formal model M the correctness of any further treatment of M can be rigorously investigated. After having defined search problems in Chapter 3 we return to this question.

CHAPTER 3

Search Problems

Similarly to *planning* the terms *problem* and *problem solving* have many interpretations. Without wanting to open a long discussion about what they 'really' mean, we summarize three general views on problem solving, cf. Simon (1983).

1) Problem solving by search
Based on the intuitive picture of a given problem one determines what kind of entities can be accepted as solutions of the problem, e.g. one can expect a plan, a number, a formula, or a string as solution. Thereafter one defines the set of all entities that are of the same kind as the expected solution, e.g. the set of all plans, the set of real numbers, the well formed formulae of a given language, or the set of all strings over an alphabet. In this case a formal solution is a special element of this set satisfying some requirements, e.g. a plan turning the initial state into a state satisfying the goal condition γ, the smallest real number with a given property, a formula being true in a given semantic model, or a string beginning with a certain prefix. The above set is considered as a space where we search for a solution. The search takes place by transitions in the space; one mostly uses transition operators that, when applied to an element of the space, yield another element. By this paradigm problem solving is starting at an initial element and making successive transitions in attempt to reach a solution.

2) Problem solving by logical reasoning

According to this view, one first has to set up a logical framework with general axioms and deduction rules, together with specific axioms describing the problem. A solution is understood as a formula (a deduction of a formula, a substitution in a formula) in the given logic. Problem solving then consists of making logical derivations until a desired formula (deduction, substitution) is reached. This approach is commonly - although not exclusively - applied within Artificial Intelligence.

3) Problem solving by mathematical programming

By this approach we formalize the intuitive problem by defining a set of variables and a so called objective function on these variables. Thereafter we define a solution as a variable assignment that realizes the lowest (highest) value of the objective function. The characteristic feature of those problems that can be treated by this approach is that the problem originally contains a measure to be optimized, or that such a measure can be defined in a natural way such that the solutions we have in mind can be identified by having a minimal (maximal) value according to this measure. Following this approach, problem solving is mostly done by numerical computation aiming at calculating a variable assignment with a minimal (maximal) objective function value. This approach is mostly associated with Operational Research.

Observe that the above problem solving metaphors are not mutually exclusive as the following example demonstrates.

Example 3.1

Consider a forward reasoning first-order theorem prover aiming at constructing a proof for a theorem φ from some axioms by some inference rules. On one hand, every application of the inference rules is clearly a reasoning step that leads to new information (propositions).

On the other hand, regarding first order formulae - including the axioms - as elements of a space we can consider the inference rules as transition operators. Namely, an inference rule *if A then B*, respectively *if A and B then C* can be viewed as a transition operator that turns the object A to B, respectively $A \wedge B$ to C. The deduction process then becomes searching a path to the desired theorem.

Furthermore, if we can reasonably define 'distance' between formulae, then the working of the theorem prover can also be seen as optimization, i.e. aiming at a minimal distance between the end of the deduction chain and φ.

□

The solution finding phase in a DSS requires problem solving abilities. When choosing among the above paradigms one should consider the following.

Search is a wide spread problem solving concept that has been the subject of many investigations and the basis of several implementations. There is a huge variety of solution finding methods that are characterized as 'search algorithms'. They differ a lot in spirit, application domain and performance. Ahlswede and Wegner (1987) see search as performing a sequence of tests each test cutting the search space; the goal of the search is to identify an object within the space. Aigner (1988) discusses probabilistic search to handle optimization-like problems, in particular applied to game playing. Charniak and McDermott (1985) consider search within artificial intelligence; they depict it as the "theory of guessing" and discuss space search based upon the usage of transition operators. Kanal and Kumar (1988) classify search algorithms for handling discrete optimization problems, while Pearl (1984) focuses on incorporating heuristics formally.

Automated reasoning grew out of classical logic by showing that resolution based theorem proving can be the underlying mechanism of problem solving, cf. Green (1969). It made its breakthrough in the mid seventies by introducing the principle of "using logic as programming language", Kowalski (1974). This idea has led to numerous practical applications and has formed the theoretical ground of the family of logic programming languages, cf. Lloyd (1987), Sterling and Shapiro (1986). The field is still being intensively investigated, Minker (1988). A great advantage of automated logical reasoning methods is that the language of logic has a great expressive power and is easy to read, that is user friendly. A generally experienced disadvantage of automated reasoning systems is their low performance. Automated reasoning as a problem solving paradigm is mostly related to Artificial Intelligence; in practice it often occurs under the names logic programming, deductive databases and is applied in expert systems, cf. Waterman (1986), or knowledge based systems, see Addis (1986), Davis and Lenat (1982), Eiben and Schuwer (1990).

Optimization can be discovered in many algorithms that traditionally belong to Operational Research. Methods that can be globally classified as mathematical programming have been applied to a wide class of problems, see e.g. Kolen and Lenstra (1990), Minoux (1986), Nemhauser and Wolsey (1988), Papadimitriou and Steiglitz (1982). These methods have booked remarkable results, although the theoretical and practical boundaries are also recognized, Garey and Johnson (1979), Hansen (1989). Roughly speaking we can describe mathematical programming methods as efficient but rigid. This means that they perform well under tight conditions, which makes the application domain of a certain algorithm rather limited.

To handle planning problems we have chosen the search paradigm for more reasons. Partly because in this way we expect more flexibility then in OR methods, partly because (heuristic) search is sometimes seen as a possible link between OR and AI, cf. Glover and Greenberg (1989).

To give a detailed, though still informal summary of applying the space search concept for problem solving, let us take planning problems for example.

a) We define a *search space* and the correspondence between the elements of the search space and plans. This latter is to guarantee that having found an element in the search space means something in the planning context.
b) We give *goal conditions* that specify a subspace of the whole search space. A *solution of the search problem* is meant as an element of this subspace; in other words it is an element that satisfies the goal conditions. Every element of the search space can be considered as a candidate for being a solution, therefore we call them *candidates* in the sequel. Obviously, the goal conditions must be given in such a way that solutions of the search problem correspond to solutions of the planning problem.
c) There are *transition operators* or *manipulations* defined on the search space. Applying a transition operator (manipulation) to a candidate results in another candidate.
d) A search problem is solved by traversing the search space by means of the transition operators (manipulations) defined in (c), i.e. by stepping from candidate to candidate. A *search procedure* is a method that prescribes the way the consecutive steps are taken.

By this approach we simplify problem solving in the following sense. If the search space and the manipulations are defined then at any point of the search space we have limited choices: we have to chose a possible manipulation at that point. The 'only' remaining difficulty is to decide which manipulations should be taken in order to reach a solution. It is typical for practical planning problems that the obtained search problem is intractable, cf. Garey and Johnson (1979).

The above points (a), (b), (c) and (d) imply a natural construction hierarchy for designing a search based problem solving method. Logically and chronologically one has to proceed by specifying the following items

1) the search space: where we search;
2) goal conditions specifying solutions within the search space: what we search;
3) the manipulations: the elementary steps by which we search;
4) the search method to prescribe how we search.

There is a natural division of these four points into two groups: (1) and (2) contain ***what*** is needed, while (3) and (4) specify ***how*** we are trying to obtain it. From the viewpoint of planning problems we can also justify this distinction of the two groups. (1) and (2) embody a translation of the planning problem to the search context, remaining at problem specification, while (3) and (4) constitute a method to handle the resulted problem, thus they belong to problem solving. This motivates our terminology: when talking about a *search problem* we roughly mean (1) and (2), the term *search procedure* covers (3) and (4). In the rest of Chapter 3 and in Chapter 4 we give a formal treatment of the search paradigm by investigating these two notions.

3.1 Model of Search Problems

The basis of our view on search is that we are looking for an element in a space. As a consequence, a solution of a search problem is a point of the search space, hence points of the space can be seen as candidate solutions. This formalization looks harmless, though it has consequences that might be counterintuitive at the first glance.

Example 3.1.1

Let us consider a shortest path problem in a graph $G = (N,E)$, Papadimitriou and Steigliz (1982). Since the expected solution of such a problem is a path in the graph G, the candidates of a corresponding search problem should be (partial) paths as well. A natural way of defining a search space is thus defining it as the set of all paths in G. The surprising consequence of this is that the search will take place in the space of all paths and not in the set N of all nodes, the 'natural' space of G.

□

There is another remarkable lesson of this example. Notice that knowing what kind of objects we want as solutions (eg. paths) we have defined a search space that consists of the same kind of objects. This shows that the definition of solutions intuitively precedes the definition of candidates. The formal relationship is, however, reversed: the search space should be defined first and then the goal conditions on it.

Example 3.1.2

Let us consider the planning problems of Chapter 2.2. For all of them we can define candidates as being plans, a solution of the search problem is a candidate (plan) that turns the initial state (or process) into a goal state (or process). Notice that in this case the search terminology perfectly matches the planning terminology: the solutions of the search problems are exactly the solutions of the planning problem.

□

Example 3.1.3

Regarding a theorem prover as a search procedure the candidates of a corresponding search space can be finite sequences of well formed formulae forming a correct deduction from the axioms.

□

In practical cases we have observed a resemblance in the way the candidates are defined.

a) First, one defines elementary objects to construct the candidates from, e.g. edges and nodes to make up a path, operations to build plans from, or formulae that occur in a deduction.
b) Then one specifies a way of construction and defines candidates as complex objects correctly constructed from the elementary objects. We used *path construction* in Example 3.1.1, *set construction* in Example 3.1.2 and *finite sequence satisfying the definition of deduction* in Example 3.1.3.

We do not investigate this regularity in defining the candidates any further. In the sequel candidates and the free search space will be primitives regardless of their inner structure.

Definition 3.1.4
A set C of candidates is called the *free search space*.
□

To define the goal of the search we have to specify which candidates are satisfactory to terminate with.

Definition 3.1.5
A *goal condition* over the free search space C is a Boolean function

$$\varphi_g : C \rightarrow \{true, false\}$$

over candidates. The *goal space* is the set

$$C_g = \{ c \in C \mid \varphi_g(c) = true \}.$$

□

Observe that the free search space defined for a given problem can be too wide, i.e. there can be candidates that we cannot interpret in the terms of the problem. A reason for this can be that the elementary objects and the construction rules to build the candidates are not defined sharp enough: there are meaningless or unwanted constructions that must be filtered out.

Example 3.1.6
Let us consider the TSP (Chapter 2.2.1) with sets of operations as candidates. Obviously, the set $\{ (to(x,y),t), (to(x,z),t) \}$ where $y \neq z$ belongs to an unexecutable plan, therefore it should be excluded as a candidate in the search space.

Restricting ourselves to candidates that belong to allowed plans Proposition 2.1.1.20 guarantees that the considered candidates belong to executable plans.

□

In practice, such a restriction on the free search space is often expressed as a conjunction of more conditions which we shall call *constraints* in the sequel. The restriction in Example 3.1.6 is needed to filter out impossible plans from among the candidates. Such constraints can be considered as *hard constraints* in the sense that they are rooted in the planning problem itself, they are not to express some subjective human wishes. Nevertheless, there can be possible but unwanted candidates depending on the preferences of the planner. Constraints that are used to exclude such candidates are mostly called *soft constraints*. The difference in the usage of hard and soft constraints is that the planner has to satisfy hard constraints while he has the freedom to enforce of reject soft constraints.

Example 3.1.7
A hard constraint for the TSP with the candidates from Example 3.1.2 can be

$\psi \Leftrightarrow$ the plan c is allowed with respect to the initial world state.

Possible soft constraints are for instance:

$\psi_1 \Leftrightarrow$ in the plan c the city z_3 is visited before the city z_5, or

$\psi_2 \Leftrightarrow$ in the plan c the city z_2 is visited last before returning home.

□

At the present level of abstraction we shall not distinguish soft and hard constraints. We melt them together into one feasibility condition according to the following definition.

Definition 3.1.8
A *feasibility condition* over the free search space C is a Boolean function

$\varphi_f : C \rightarrow \{true, false\}$

over candidates; candidates with $\varphi_f(c) = true$ are *feasible*, with $\varphi_f(c) = false$ are *infeasible*.

The *feasible search space* is

$C_f = \{ c \in C \mid \varphi_f(c) = true \}$.

□

Definition 3.1.9

A *search problem* is a 3-tuple $(C, \varphi_f, \varphi_g)$, where C is a set of candidates, the free search space, $\varphi_f : C \rightarrow \{true, false\}$ and $\varphi_g : C \rightarrow \{true, false\}$ are the feasibility condition and the goal condition, respectively. A *solution of a search problem* is a candidate $c \in C$ for which

$$\varphi_f(c) = true \text{ and } \varphi_g(c) = true$$

holds.

□

In practice, soft and hard constraints play a different role: the hard constraint must be satisfied by the planner, while he has the freedom to modify (add or delete) soft constraints. Doing so, he obviously changes the search problem as well, since according to these modifications the feasibility condition changes. From this point of view, hard constraints can be considered as defining condition of the broadest reasonable search problem.

Example 3.1.10

Let us recall Example 3.1.7. With the constraints ψ, ψ_1 and ψ_2 given there we can define three different search problems (C, ψ, φ_g), $(C, \psi \wedge \psi_1, \varphi_g)$ and $(C, \psi \wedge \psi_2, \varphi_g)$, where (C, ψ, φ_g) carries the broadest feasible search space.

□

Practice proves that most of the search procedures restrict the search to the feasible search space. According to this view we could have defined a search problem as a pair (D, φ_g), where D is a set of candidates, φ_g is a goal condition. This puts the role of φ_f in a yet other light: we can consider the free search space C as a preliminary definition of the actual search space and regard φ_f as the completion needed to define this actual search space $\{c \in C \mid \varphi_f(c)\}$. The reasons to define a search problem as a triple are twofold. First, not all the search procedures deal with feasible candidates only. Second, we consider the role and the notion of the feasibility condition so important that we do not want to 'hide' it within the set D from (D, φ_g). For the sake of convenience, however, the term *search space* will be often used as a synonym of the feasible search space. Accordingly, the term *candidate* will often stand for a feasible candidate in the sequel.

There is an important class of problems that are often solved by search methods, therefore we want to model them with our general definition of a search problem.

Definition 3.1.11
An *optimization problem* is either a minimization problem or a maximization problem. A *minimization problem* is a pair (C, f), where C is an arbitrary set, $f: C \to \mathbb{R}$ is the so called *objective function*. The aim in a minimization problem is to find a minimum of f over C, that is a $c \in C$ such that

$$\forall\, d \in C : f(c) \leq f(d).$$

A *maximization problem* can be defined analogously, requiring a maximum of f over C, i.e. a $c \in C$ such that

$$\forall\, d \in C : f(c) \geq f(d).$$

□

By an optimization problem we always mean a minimization problem in the sequel. Notice that this does not lead to any loss in generality, since any maximization problem can easily be transformed to an equivalent minimization problem and vice versa.

Observe that in the definition of a minimum there is a universal quantifier that ranges over the whole C. This means that verifying that a certain $c \in C$ is a minimum can be very difficult even if C is finite. Furthermore, in practice it is not always needed to find an absolute minimum of f. Therefore one often considers a *decision problem*, see Garey and Johnson (1979) or *recognition problem*, cf. Nemhauser and Wolsey (1988) where a candidate $c \in C$ is wanted such that it satisfies

$$f(c) \leq D,$$

with D being a bound given in advance.

Let us remark that such a decision problem is not necessarily easy to solve in a mathematical sense. A great deal of the NP-complete problems listed in Garey and Johnson (1979) are decision problems (recognition problems) in the above sense.

It is clear that both optimization problems and decision problems can be seen as search problems: an optimization problem (C, f) can be expressed as a search problem $(C, \varphi_f, \varphi_g)$, where

$\varphi_g(c) = true \Leftrightarrow \forall d \in C : f(c) \leq f(d)$,

while a decision problem with a given bound D corresponds to

$\varphi_g(c) = true \Leftrightarrow f(c) \leq D$.

According to the high level parameterization concept, we consider the elements of the 3-tuple $(C, \varphi_f, \varphi_g)$ as parameters that need to be set in order to define a search problem.

3.2 Relationship Between Planning Problems and Search Problems

Our objective by investigating search problems is obvious: we want to apply them in a DSS to solve planning problems. To formulate we exactly mean by this we examine the relationship between planning problems and search problems in this section.

Observe that the mathematical models of planning problems in Chapter 2 are 'human friendly'. This means that the formalism, the usage of abstract entities e.g. a plan, effect, etc., facilitates a natural mapping between the model and the real world. This feature supports the construction and the understanding of such models.

A solution method, however, is preferably efficient, which might counteract understandability. The reason is that for the sake of efficiency the candidates should be easy-to-handle by the search procedure, i.e. candidates should have a simple structure. Therefore, in the search problem one probably prefers another representation of reality, a representation that supports computation.

Notice that although a mathematical problem model may imply some preferences for certain forms of the candidates, it is primarily the search procedure that requires a certain form. In principle there can be more 'procedure friendly' representations given to the same mathematical problem model.

Example 3.2.1

Imagine we have a planning problem defined in the abstract terms of Chapter 2. If we have a discrete programming procedure to apply then describing it in terms of 0 - 1 matrices is algorithm friendly. However, if we want to solve the given

problem by SLDNF-derivation, cf. Lloyd (1987), then a Horn-clause representation form is algorithm friendly and 0 - 1 matrices are not.

□

From the above it is obvious that a translation step is needed to establish the correspondence between the objects of a planning problem and the ones used in a search problem. Such a translation should of course not only assign candidates to plans (and vice versa) but should also guarantee that

- feasibility of candidates correctly reflects allowability of plans, and
- solutions of the search problem correspond to solutions of the planning problem.

Next we are going to work out the details how a search problem can be defined to a planning problem. For the sake of convenience we restrict ourselves to dynamic planning problems; Proposition 2.1.3.6 ensures that we do not loose generality by this restriction.

Definition 3.2.2

Let $(F, A, T, \hat{\alpha}, \hat{e})$, $(f_0, \hat{\gamma}, \kappa)$ be a dynamic planning problem and let C be an arbitrary set intended to be the free search space. A *translation function* or *representation function* is a partial function

$$R : \mathcal{P}(A \times T) \nrightarrow C.$$

According to the second name of R, the candidate $R(P)$ is called the *representation of the plan P*.

□

The name *translation function* fits the intuitive view of switching from planning context to search context. The name *representation function* is closer to the conventional AI terminology, where the form of an abstract object is often called its representation.

Definition 3.2.2 gives the formal interpretation of 'the candidate corresponding to the plan P': it is $R(P)$. Also 'the plan corresponding to the candidate c' is defined hereby: it is a plan P satisfying $R(P) = c$. This latter, however, is only uniquely determined if R is injective, i.e.

$$R^{-1}(c) = \{ P \in \mathcal{P}(A \times T) \mid R(P) = c \}$$

is a singleton.

To be able to talk about 'the plan corresponding to the candidate c' even if this latter is not the case we have to define an *interpretation function* in the following sense.

Definition 3.2.3

Let $(F, A, T, \hat{\alpha}, \hat{e})$, $(f_0, \hat{\gamma}, \kappa)$ and C be as before and let R be a translation function. An *interpretation function* corresponding to R is a function

$I : C \nrightarrow \mathcal{P}(A \times T)$

such that

$\forall\ c \in dom(I) : I(c) \in R^{-1}(c).$

The plan $I(c)$ is *the interpretation of the candidate* c (in terms of the planning problem).

□

The relationship between the translation and the interpretation is determined in the definition of the interpretation function that implies that

$\forall\ c \in dom(I) : R(I(c)) = c$

always holds. Nevertheless, it is easy to see that

$\forall\ P \in dom(R) : I(R(P)) = P$

does not necessarily hold in general.

Definition 3.2.4

Let $(F, A, T, \hat{\alpha}, \hat{e})$, $(f_0, \hat{\gamma}, \kappa)$ be a planning problem, C be an arbitrary set intended to be the free search space and let R and I be a representation and an interpretation function between $\mathcal{P}(A \times T)$ and C. Furthermore, let φ_f and φ_g be a feasibility condition and a goal condition over C, respectively. We say that φ_f *fits* $(F, A, T, \hat{\alpha}, \hat{e})$, $(f_0, \hat{\gamma}, \kappa)$, C, R *and* I if for any $c \in C$

$\varphi_f(c) \Rightarrow \hat{\alpha}'(f_0, I(c)).$

We say that φ_g *fits* $(F, A, T, \hat{\alpha}, \hat{e})$, $(f_0, \hat{\gamma}, \kappa)$, C, R *and* I if for any $c \in C$

$\varphi_g(c) \Rightarrow [\ \hat{\gamma}(\hat{e}'(f_0, I(c)))$ and $\forall\ d \in C : \hat{\gamma}(\hat{e}'(f_0, I(d))) \Rightarrow \kappa(I(c)) \leq \kappa(I(d))\ ].$

If φ_f and φ_g fit $(F, A, T, \hat{\alpha}, \hat{e})$, $(f_0, \hat{\gamma}, \kappa)$, C, R *and* I then we say that *the search problem* $(C, \varphi_f, \varphi_g)$ *fits the planning problem* $(F, A, T, \hat{\alpha}, \hat{e})$, $(f_0, \hat{\gamma}, \kappa)$.

□

With the aid of the last definition we can formalize the basis of solving planning problems by search: having defined a planning problem we have to specify a search problem that fits it. Only then can we interpret a solution of the search problem as a plan and obtain a solution of the planning problem by search, thus only then can we solve planning problems by search.

Recall the figure and our remark about formal planning problems at the end of section 2.2. From that point of view we can illustrate the role of search problems by the next figure

where M' denotes a search problem.

As we have mentioned, the correctness of the step a can only be intuitively justified. The relationship between two formal models, however, can be formally defined and this is exactly the purpose of Definition 3.2.4. Let us also remark that the formal definition of a planning problem is sometimes is omitted in practice. This means that M is skipped and one immediately makes M' by defining plans as candidates with the appropriate feasibility and goal conditions. Obviously, this implies that in such a case it is M' that has to be related to R and therefore it is the correctness of M' that is justified on an intuitive ground.

Definition 3.2.5

Let $(F, A, T, \hat{\alpha}, \hat{e})$, $(f_0, \hat{\gamma}, \kappa)$ be an arbitrary planning problem, C a set (the intended search space), R a representation function and I an interpretation function The *feasibility condition derived from* $(F, A, T, \hat{\alpha}, \hat{e})$, $(f_0, \hat{\gamma}, \kappa)$, C, R, I is given by

$$\varphi_f(c) = \begin{cases} \hat{\alpha}(f_0, I(c)) & \text{if } c \in dom(I) \\ \mathit{false} & \text{otherwise} \end{cases}$$

The *goal condition derived from* $(F, A, T, \hat{\alpha}, \hat{e})$, $(f_0, \hat{\gamma}, \kappa)$, C, R, I is defined as

$$\varphi_g(c) = \begin{cases} \mathit{true} & \text{if } c \in dom(I) \text{ and } \hat{\gamma}(\hat{e}'(f_0, I(c))) \text{ and} \\ & \quad \forall\, d \in C : \hat{\gamma}(\hat{e}'(f_0, I(d))) \Rightarrow \kappa(I(c)) \leq \kappa(I(d)) \\ \mathit{false} & \text{otherwise} \end{cases}$$

□

Definition 3.2.6

Let $(F, A, T, \hat{\alpha}, \hat{e})$, $(f_0, \hat{\gamma}, \kappa)$ be a planning problem. The ***natural search problem*** corresponding to $(F, A, T, \hat{\alpha}, \hat{e})$, $(f_0, \hat{\gamma}, \kappa)$ is $(C, \varphi_f, \varphi_g)$ where $C = \mathcal{P}(A \times T)$, $R = id_C$, $I = R^{-1}$, φ_f and φ_g are the feasibility condition and the goal condition derived from this C, R and I.

□

Observe that in the natural search problem corresponding to a planning problem the derived feasibility and goal condition satisfy

$$\varphi_f(P) \Leftrightarrow \hat{\alpha}'(f_0, P)$$

and

$$\varphi_g(P) \Leftrightarrow \gamma(\hat{e}'(f_0, P)) \text{ and } \forall P' \in \mathcal{P}(A \times T) : \hat{\gamma}(\hat{e}'(f_0, P')) \Rightarrow \kappa(P) \leq \kappa(P').$$

Let us remark that in Example 3.1.2 we meant natural search problems but we expressed it informally since we did not have the formal vocabulary yet.

As we have mentioned in the introduction of this chapter a search space is mostly defined with an eye on an intended solution method such that the form of the candidates is suited to the given method. Since the method we have in mind is handling lists or tables rather than sets of operations (see later in Chapter 5) we present a standard way of defining a search space containing tables.

For the sake of convenience let us suppose that every action can be identified by a name and a finite list of parameters. (Recall that in each example of section 2.2 the set of actions A was defined in such a way.) Formally this means that we assume that in a planning problem we have a finite set $AN = \{act_1, \ldots, act_K\}$ of action names each name having a fixed arity n_i. Furthermore, we assume that for every $i \in \{1, \ldots, K\}$ and $j \in \{1, \ldots, n_i\}$ there is a finite set X^i_j given that forms the domain of the j-th parameter of the i-th action name. The set of all actions in this case is

$$A = \bigcup_{i=1}^{K} \{ act_i(x_1, \ldots, x_{n_i}) \mid x_1 \in X^i_1, \ldots, x_{n_i} \in X^i_{n_i} \}.$$

Now, if we further assume that each X_j^i is numbered, then to any plan - a set of operations - we can define a list of operations uniquely through the following steps.

1) Let us denote the standard lexicographic ordering on $X_1^i \times \ldots \times X_{n_i}^i$ by $\prec_i$ for every $i \in \{1, \ldots, K\}$. Then we can define a lexicographic ordering $\ll$ on the set of actions by

$$act_i(x_1, \ldots, x_{n_i}) \ll act_j(y_1, \ldots, y_{n_j}) \iff i < j \vee [i = j \wedge (x_1, \ldots, x_{n_i}) \prec_i (y_1, \ldots, y_{n_i})]$$

2) Based on $\ll$ and the ordering $<$ on T we define an ordering $\lhd$ on $A \times T$ as follows:

$$(a,t) \lhd (a',t') \iff t < t' \vee [t = t' \wedge a \ll a'].$$

Notice that any plan P uniquely determines its $\lhd$-ordered version. Strictly speaking, since $\lhd$ is a linear ordering the following can be easily seen: if P is a plan with n operations then there is a list $P^{\lhd} = [o_1, \ldots, o_n]$ such that

$$\forall o \in A \times T : [\, o \in P \iff o = o_i \text{ for some } i \in \{1, \ldots, n\}\,]$$

and

$$\forall i \in \{1, \ldots, n\text{-}1\} : o_i \lhd o_{i+1}\,.$$

It can be often convenient to write plans in table form instead of a list form, see for example the coming section or Chapter 5. To obtain such a form for a plan with m operations we first have to make the list $P^{\lhd}$ in the form

$$P^{\lhd} = [\,(act_{i_1}(x_1^1, \ldots, x_{n_{i_1}}^1), t_1), \ldots, (act_{i_m}(x_1^m, \ldots, x_{n_{i_m}}^m), t_m)\,].$$

Then the table

$$\mathcal{T}(P) = \begin{bmatrix} act_{i_1} & \cdots & act_{i_m} \\ x_1^1 & \cdots & x_1^m \\ \vdots & & \vdots \\ x_N^1 & \cdots & x_N^m \\ t_1 & \cdots & t_m \end{bmatrix}$$

is uniquely defined, where $N = \max\{ n_1, \ldots, n_K\}$ and if the action name in the j-th column ($j \in \{1, \ldots, m\}$) has an arity $n < N$ then the $(n+1)$-th, . . ., N-th positions of that column are filled up with a special symbol, say *.

Definition 3.2.7

Let $(F, A, T, \hat{\alpha}, \hat{e})$ be a planning situation where

$$A = \bigcup_{i=1}^{K} \{ act_i(x_1, \ldots, x_{n_i}) \mid x_1 \in X_1^i, \ldots, x_{n_i} \in X_{n_i}^i \}$$

The *default search space* corresponding to this planning situation is the set

$$C = \bigcup_{m \in \mathbb{N}} \left[\bigcup_{i=1}^{K} \left[\{act_i\} \times X_1^i \times \ldots \times X_{n_i}^i \times \{*\} \times \ldots \times \{*\} \times T \right] \right]^m$$

and the corresponding *default representation function* is obtained by the above construction, i.e. it is

$R : \mathcal{P}(A \times T) \to C,$

such that

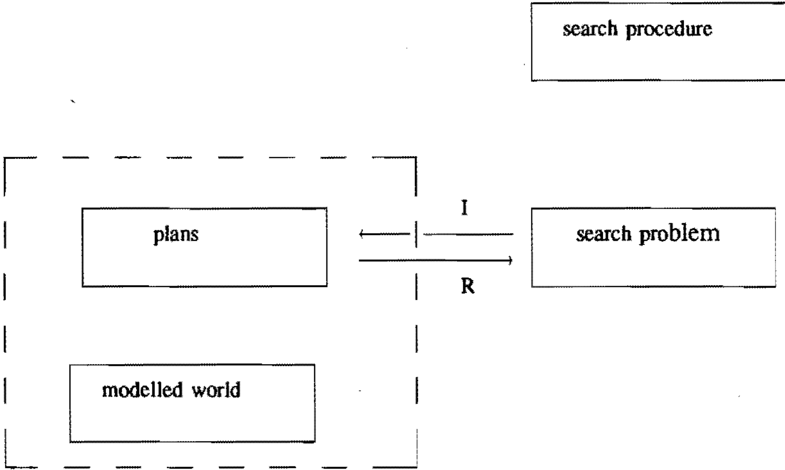
$R(P) = \mathcal{T}(P^{\lhd}).$

The *default interpretation function* is $I = R^{-1}$.

□

Notice that the default feasibility and goal conditions determine a search problem that fits the given planning problem. Nonetheless, it can happen that we want to save the 'roundabout' through R and I and want to define φ_f and φ_g by a 'shortcut', immediately in terms of candidates. In such a case we have to invent conditions φ_f and φ_g such that the truth value of φ_f (c) and $\varphi_g(c)$ can be determined by examining c only, without computing $I(c)$ and the rest to it. In this case we also have to prove that these non default conditions fit the given planning problem, R and I. In section 3.3 we present an illustration of this matter.

Recall the basic taxonomy of planning introduced at the end of Chapter 1. In the light of this chapter it can be refined as the following figure indicates:



3.3 Examples of Search Problems

In this section we present certain search problems that correspond to the planning problems of Chapter 2.2.

3.3.1 Travelling Salesman Problem

Let us take the TSP form section 2.2.1 with the set of cities $Z = \{z_1, \ldots, z_n\}$. Since we only have one action name in this example we can omit the reference to it and define a simplified version of the default search space as

$$C = \bigcup_{m \in \mathbb{N}} (Z \times Z \times T)^m .$$

The set C is thus the set of finite tables with first and second rows consisting of cities and the third row containing time instances.

For a $\triangleleft$-ordered plan $P = [\ (to(u_1,v_1),t_1), \ldots, (to(u_k,v_k),t_k)\ ]$ the default representation function is

$$R(P) = \begin{bmatrix} u_1 & . & . & . & u_k \\ v_1 & . & . & . & v_k \\ t_1 & . & . & . & t_k \end{bmatrix} \qquad (*)$$

and $I = R^{-1}$.

Next we define a non-default feasibility condition. For a candidate $c \in C$ of the form (*)

$$\varphi_f(c) \Leftrightarrow \begin{array}{ll} \forall i \in \{1,\ldots,k\text{-}1\} : t_i \neq t_{i+1} & \text{and} \\ \forall i \in \{1,\ldots,k\text{-}1\} : v_i = u_{i+1} & \text{and} \\ u_1 = z_1. & \end{array}$$

Proposition 3.3.1.1

If $\varphi_f(c) = \mathit{true}$ then $\alpha'(s_0, I(c))$.

Proof

It is obvious that a feasible candidate c has the form

$$c = \begin{bmatrix} z_1 & v_1 & . & . & . & v_{k\text{-}1} \\ v_1 & v_2 & . & . & . & v_k \\ t_1 & t_2 & . & . & . & t_k \end{bmatrix}$$

Then we have to show that

$$\alpha'(\ \{at(z_1)\},\ \{\ (to(z_1,v_1),t_1),\ (to(v_1,v_2),t_1)..\ ,(to(v_{k-1},v_k),t_k)\ \}\ ) = \mathit{true},$$

which follows easily from the definitions of section 2.2.1.

□

We can also define an evaluation criterion for any candidate in the form (*) by

$$\kappa(c) = \sum_{i=1}^{k} D(u_i,v_i),$$

and specify a non-default goal condition by

$$\psi(c) \Leftrightarrow \begin{array}{ll} \varphi_f(c) & \text{and} \\ k = n & \text{and} \\ v_n = z_1\ ; & \end{array}$$

and

$$\varphi_g(c) \Leftrightarrow \begin{array}{l} \psi(c) \quad \text{and} \\ \forall d \in C : [\psi(d) \Rightarrow \kappa(c) \leq \kappa(d)]. \end{array}$$

3.3.2 Precedence Constrained Scheduling Problem

Applying the default method with omitting the reference to the (unique) action name we obtain the following search space, representation function and interpretation function.

$$C = \bigcup_{k \in \mathbb{N}} (J \times M \times T)^k .$$

For a $\lhd$-ordered plan $P = [\ (begin(x_1,y_1),t_1), \ldots,(begin(x_k,y_k),t_k)\ ]$ let

$$R(P) = \begin{bmatrix} x_1 & . & . & . & x_k \\ y_1 & . & . & . & y_k \\ t_1 & . & . & . & t_k \end{bmatrix}$$

and $I = R^{-1}$.

We define a non-default feasibility condition for an arbitrary candidate $c \in C$ of the above form as follows

$\varphi_f(c) \Leftrightarrow$

a) $\forall\, i,l \in \{1, \ldots, k\} : x_i \neq x_l$ and

b) $\forall\, i \in \{1, \ldots, k\} : ABLE(x_i,y_i)$ and

c) $\forall\, i \in \{1, \ldots, k\}\ \forall\, j \in J :$

$[PRE(j,y_i) \Rightarrow \exists\, l \in \{1, \ldots, i\text{-}1\} : y_l = j \wedge t_l + D(x_l,y_l) < t_i]$ and

d) $\forall\, i \in \{1, \ldots, k\}\ \neg\exists\, l \in \{i, \ldots, k\} : [x_l = x_i] \wedge [t_l < t_i + D(x_i,y_i)]$.

Proposition 3.3.2.1

$\forall\, c \in C : \varphi_f(c) \Rightarrow \dot{\alpha}(f_0, I(c))$

Proof

We give the sketch of the proof remarking that the above (a), . . .,(d) imply the conditions (1), . . .,(4) of Definition 2.2.3.3, while (5) of Definition 2.2.3.3 follows from (a).

□

A non-default goal condition for an arbitrary $c \in C$ can be given by

$\psi(c) \Leftrightarrow \varphi_f(c)$ and

$k = |J|$;

and

$$\varphi_g(c) \Leftrightarrow \psi(c) \quad \text{and}$$
$$\forall d \in C : [\psi(d) \Rightarrow \kappa(c) \leq \kappa(d)],$$

where

$$\kappa(c) = max \{ t_i + D(x_i,y_i) \mid i \in \{1, \ldots, k\} \}.$$

It is straightforward that for any candidate with $\varphi_g(c) = true$ $I(c)$ is an optimal solution of the planning problem given in Definition 2.2.3.5.

3.3.3 Time Table Problem

Here again we can omit the reference to the name of the actions obtaining the following.

$$C = \bigcup_{m \in \mathbb{N}} (G \times Z \times D \times K \times T)^m .$$

For a $\lhd$-ordered plan $P = [\ (begin(x_1,y_1,u_1,v_1),q_1), \ldots, (begin(x_m,y_m,u_m,v_m),q_m)\ ]$

$$R(P) = \begin{bmatrix} x_1 & \cdots & x_m \\ y_1 & \cdots & y_m \\ u_1 & \cdots & u_m \\ v_1 & \cdots & v_m \\ q_1 & \cdots & q_m \end{bmatrix} \qquad (*)$$

and $I = R^{-1}$.

We define a non-default feasibility condition for an arbitrary candidate $c \in C$ in the above form as follows:

$\varphi_f(c) \Leftrightarrow$

a) $\forall i \in \{1, \ldots, m\} : ABLE(u_i,y_i)$ and

b) $\forall i \in \{1, \ldots, m\} : N(x_i,y_i) > 0$ and

c) $\forall i \in \{1, \ldots, m\}\ \neg\exists j \in \{1, \ldots, m\} : x_i = x_j \wedge y_i = y_j \wedge u_i \neq u_j$ and

d) $\forall g \in G\ \forall z \in Z : |\{ i \in \{1, \ldots, m\} \mid x_i = g \wedge y_i = z \}| \leq N(g,z)$ and

e) $\forall i \in \{1, \ldots, m\}\ \neg\exists j \in \{1, \ldots, m\} : x_i = x_j \wedge q_i = q_j \wedge y_i \neq y_j \wedge u_i \neq u_j \wedge v_i \neq v_j$ and

f) $\forall i \in \{1, \ldots, m\}\ \neg\exists j \in \{1, \ldots, m\} : u_i = u_j \wedge q_i = q_j \wedge x_i \neq x_j \wedge y_i \neq y_j \wedge v_i \neq v_j$ and

g) $\forall i \in \{1, \ldots, m\} \neg\exists j \in \{1, \ldots, m\} : v_i = v_j \wedge q_i = q_j \wedge x_i \neq x_j \wedge y_i \neq y_j \wedge u_i \neq u_j$ and

h) $\forall i \in \{1, \ldots, m\} : q_i \in \{ t_1, \ldots, t_M\}$.

Proposition 3.3.2.1

$\forall c \in C : \varphi_f(c) \Rightarrow \hat{\alpha}(f_0, I(c))$

Proof

It is obvious, the above points (a), . . .,(h) imply the conditions (1), . . .,(8) of Definition 2.2.4.3, respectively.

□

We give a non-default goal condition for a $c \in C$ in the form (*) by

$$\psi(c) \Leftrightarrow \varphi_f(c) \text{ and } m = \sum_{g\in G} \sum_{z\in Z} N(g,z);$$

and

$$\varphi_g(c) \Leftrightarrow \psi(c) \text{ and } \forall d \in C : [\psi(d) \Rightarrow \kappa(c) \leq \kappa(d)],$$

where

$$\kappa(c) = \kappa_1(c) + 10 \cdot \kappa_2(c)$$

with

$$\kappa_1(c) = \sum_{g\in G} \sum_{z\in Z} \left| \{ n \in \{1, \ldots, L\text{-}1\} \mid \exists\, i,j \in \{1, \ldots, m\} : x_i = g \wedge y_i = z \wedge q_i = t_n \wedge x_j = g \wedge y_j = z \wedge q_j = t_{n+1} \} \right|$$

and

$$\kappa_2(c) = \sum_{g\in G} \sum_{z\in Z} \left| \{ n \in \{1, \ldots, L\text{-}2\} \mid \exists\, i,j,l \in \{1, \ldots, m\} : x_i = g \wedge y_i = z \wedge q_i = t_n \wedge x_j = g \wedge y_j = z \wedge q_j = t_{n+1} \wedge x_l = g \wedge y_l = z \wedge q_l = t_{n+2} \} \right|$$

3.3.4 Ship Loading Problem

We present the default search space to illustrate a case where there are more action names. In the Ship Loading Problem we have $AN = \{ move, load, unload \}$ with $arity(move) = 2$, $arity(load) = arity(unload) = 5$. Here we have to use the *-notation, i.e. apply the extended version of *move* actions of the form

$move(h_i, h_j, *, *, *)$,

where $h_i, h_j \in H$.

The default search space is thus

$$C = \bigcup_{m \in \mathbb{N}} \Big[\{move\} \times H \times H \{*\} \times \{*\} \cup \{load\} \times U \times H \{1, \ldots, X\} \times \{1, \ldots, Y\} \times \{1, \ldots, Z\} \cup \{unload\} \times U \times H \{1, \ldots, X\} \times \{1, \ldots, Y\} \times \{1, \ldots, Z\} \Big]^m$$

with the default representation function:

$$R(P) = \begin{bmatrix} name_1 & \ldots & name_k \\ \tilde{a}_1 & \ldots & \tilde{a}_k \\ \tilde{b}_1 & & \tilde{b}_k \\ \tilde{c}_1 & & \tilde{c}_k \\ \tilde{d}_1 & & \tilde{d}_k \\ \tilde{e}_1 & & \tilde{e}_k \\ t_1 & \ldots & t_k \end{bmatrix}$$

Observe that a candidate can be divided into blocks such that within each block there are only columns where the action name is either *load* or *unload* and different blocks are separated by a column belonging to a *move* action.

CHAPTER 4

Search Procedures

In this chapter we consider search procedures. We are aiming at a general model that captures a large class of search procedures with the same formalism.

4.1 Space Search, Graph Search, Local Search

The term *search* is often extended by certain adjectives, so that one often speaks about space search, graph search, neighbourhood search or local search. In this section we give a brief overview of these types of search and present our vision on their relationship. In particular, we take space search as a basis and claim that all the others can be seen as variants of this one.

Roughly, we can associate space search with a view based on using so called transition operators: a transition operator transforms a candidate (an element of the search space) into another candidate, cf. Charniak and McDermott (1985). In typical graph search methods such as breadth first search or depth first search, Pearl (1984), it is presumed that a set of edges is given between the points of the search space. Accordingly, the search takes place along the edges, that is a step from a candidate to another candidate is possible if there is an edge between them. In neighbourhood search or local search one assumes that the search space is divided into overlapping regions, called neighbourhoods, see Aarts and Korst

(1989), and that steps from a candidate c are only possible to candidates contained in the neighbourhood of c.

Definition 4.1.1

Given a search space C, a *transition operator* or a *manipulation* is a partial function

$$m : C \nrightarrow C.$$

□

To avoid any possibility for confusion between operations (Definition 2.1.1.6) and operators (Definition 4.1.1) we shall use the name manipulation in the sequel.

Example 4.1.2

Let us take an arbitrary planning problem and the natural search problem belonging to it by Definition 3.2.6. To any operation (a,t) we can define a manipulation $m_{(a,t)}$ such that applying $m_{(a,t)}$ to a candidate (plan) P the result is

$$m_{(a,t)}(P) = P \,\Delta\, \{(a,t)\} = \begin{cases} P \cup \{(a,t)\} & \text{if } (a,t) \notin P \\ \\ P \setminus \{(a,t)\} & \text{if } (a,t) \in P \end{cases}$$

Another example can be the shift manipulation m_t defined for any $t \in T$ that delays the actions of a plan. Formally, applying m_t to a candidate P the result is a new candidate

$$m_t(P) = \{ (a,\tau') \in A \times T \mid \tau' = \tau + t, (a,\tau) \in P \}.$$

□

Definition 4.1.3

If we have a set M of manipulations on the set C then the set

$$E_M = \{ (c,m(c)) \in C \times C \mid m \in M, c \in dom(m) \}$$

is regarded as *the set of edges induced by M on the vertices C* and the graph

$$G_M = (C,E_M)$$

is called *the graph induced by M on C.*

□

By the above definition we can naturally envision manipulations as edges between candidates, or rather, we can see the names of manipulations as labels on

edges. The notion of the induced graph helps us to enlighten a source of confusion that often occurs if one loosely speaks about a search graph. Namely, if the original problem is given in terms of a graph, for instance a shortest path problem within a graph $G = (N,E)$, then we actually have two graphs. The original graph G is used to define the search space C where a candidate is a path within G. The induced graph $G_M = (C,E_M)$, however, is used to structure the search space according to a given set M of manipulations. Therefore, simply talking about a search graph can be misleading; for full clarity one should specify whether G or G_M is meant.

We can also model neighbourhood search by manipulations according to the next definition.

Definition 4.1.4

If we have a set M of manipulations on the set C, then *the neighbourhood function induced by M* is

$N_M : C \rightarrow \mathcal{P}(C)$,

such that for every $c \in C$

$N_M(c) = \{ m(c) \in C \mid m \in M, c \in dom(m) \}$.

For any $c \in C$ *the neighbourhood of c induced by M* is the set $N_M(c)$, a candidate $c' \in N(c)$ is a *neighbour of c*.

□

Example 4.1.5

A well-known type of local search algorithms for travelling salesman problems is based on the usage of k-exchanges ($k \in \mathbb{N}$), cf. Lin (1965), Lin and Kernighan (1973). A k-exchange in our terms is an operator that produces a new candidate (tour) from an old one. An important issue of this type of algorithms is how to deal with local optima, where the term *local optimum* in fact refers to a candidate that is optimal in the neighbourhood induced by the set of all possible k-exchanges.

□

There is a remarkable assumption often made in local search or neighbourhood search algorithms. Namely, local search procedures (neighbourhood search

procedures) are often assumed to terminate with a candidate that is optimal in its neighbourhood. This means that we can characterize local search (neighbourhood search) as space search by a set of manipulations M terminating with a candidate c satisfying

$$f(c) = min\{ f(c') \mid c' \in N_M(c) \}.$$

Notice that for defining *min* we need an objective function $f : C \rightarrow \mathbb{R}$ on the set of candidates such that *min* can be defined by the f values of the candidates. Therefore, we associate local search procedures (neighbourhood search procedures) with optimization problems. Observe that the above property of local search methods is crucial for distinguishing them. Namely, if we only keep the stepping-to-a-neighbour property then local search becomes graph search (space search) under another name.

Genetic algorithms, cf. Goldberg (1989), Grefenstette (1985, 1987), Schaffer (1989), form an important class of search procedures and they can not be described by the former notion of a manipulation that turns a candidate into another candidate. Namely, a genetic manipulation typically needs two parents (candidates) to produce a set of children (new candidates). Such a relation between the candidates can be expressed by an extended form of manipulations.

Definition 4.1.6

Given a search space C, a *hyper manipulation* is a partial function

$$m : \mathcal{P}(C) \nrightarrow \mathcal{P}(C).$$

□

Obviously, ordinary manipulations can be seen as hyper manipulations defined on singletons.

To define the hyper graph induced by a set M of hyper manipulations we can either

- maintain the candidate-vertex correspondence and use hyper edges going from set-of-vertices to set-of-vertices, or
- identifying a vertex by a set of candidates, thus use hyper vertices and usual (hyper) vertex-to-(hyper)vertex edges.

We chose the second possibility which leads to the following definition.

Definition 4.1.7
If we have a set M of hyper manipulations on the space C, then *the hyper graph induced by M* is

$$G_M = (\mathcal{P}(C), E_M)$$

where

$$E_M = \{ (x, m(x)) \in \mathcal{P}(C) \times \mathcal{P}(C) \mid m \in M, x \in dom(m) \}.$$

□

Notice that using hyper graphs includes using ordinary graphs: an ordinary graph can be considered as a special hyper graph having only edges between hyper nodes that are singletons.

By having defined a search space and manipulations we know what to search, where and by which steps. Nevertheless, we still have to determine how to search. This means that we have to specify a method that prescribes the consecutive steps of the search process. This method is mostly called the *search strategy, search procedure, search method* or *search algorithm*; we shall mostly use the name *search procedure.* In the rest of this section we identify the most essential components of search procedures and put them together into a General Search Procedure.

4.2 The General Search Procedure

Our General Search Procedure (GSP) is an iterative generate-and-test procedure. For the sake of generality we take a search procedure iterating a set of candidates, called *population,* in each iteration cycle. Generating and testing in this case means that the procedure is creating new populations (candidates) and testing them whether they suffice as a solution.

Scheme of the General Search Procedure

```
Initialize a set of candidates
while not Goal do
  begin
    Select a subset from the set of candidates
    Produce new candidates from the selected ones
    Add the new candidates to the old ones
    Reduce the extended set of candidates
  end
```

Notice where the generate and test components are included in the above scheme. Generation of new candidates is done by *Select* and *Produce*, while testing happens at checking the *Goal* and at applying *Reduce*.

We consider this scheme as the skeleton of our problem solver within a DSS. This skeleton specifies the main outlines of a search method using *Initialize, Goal, Select, Produce* and *Reduce* as parameters. Giving values to these parameters we obtain a complete search procedure. A very important question is: which of the above parameters can be set problem independently and which of them needs to carry problem dependent knowledge or heuristics. If, for instance we can define such values for *Goal, Select* and *Reduce* that can be used over a broad domain of search problems then we reduced the efforts of designing a problem specific search procedure to making *Initialize* and *Produce*.

Before the exact definition of the GSP we make a yet other generalization. In the sequel we also wish to consider methods maintaining extra structure on the actual set of candidates, i.e. on the populations. In particular we want to cover cases of having lists of candidates, not only sets. Therefore we shall use lists of candidates as populations with the standard operations $\in$, $\subseteq$, $\cup$, $\cap$ etc., meaning the straightforward definition of these operations for lists. We introduce C^* to denote the set of all finite lists over C, C^n ($n \in \mathbb{N}$) stands for the set of lists with n elements.

In the sequel we unfold the scheme of the GSP. Our main concern will be the exact identification of the parameters of the search procedure by formalizing the terms *Goal Select, Produce, Reduce* and at the end of this section we have a closer look on *Initialize*. In section 4.4 we investigate which conditions on the parameters imply desirable properties of the GSP, namely convergence.

To model random search we introduce functions with so called random parameters in our procedure. What do we mean by a randomly parameterized function with domain X and range Y? Strictly speaking we take a set of functions $\mathcal{F} \subseteq X \to Y$, a probability space $(\Omega, \mathcal{A}, \mathbb{P})$ and a random variable $\tilde{f} : \Omega \to \mathcal{F}$. Then $\tilde{f}(\omega) : X \to Y$ is a function with the desired signature for any $\omega \in \Omega$. Therefore we call ω a *random parameter* and consider $\tilde{f}$ as a randomly parameterized function from X to Y. For notational simplicity we often write the signature of $\tilde{f} : \Omega \to (X \to Y)$ in an equivalent form of $\tilde{f} : \Omega \times X \to Y$.

To have random *Selection, Production* and *Reduction* we fix a basic manner to obtain random parameters for them. Namely, we introduce three finite sets B, Γ, Δ to provide parameters for *Selection, Production* and *Reduction*, respectively and attach a random variable that delivers the values of these parameters.

Definition 4.2.1

A *basic sample process* is a tuple $(\Omega, \mathcal{A}, \mathbb{P}, B, \Gamma, \Delta, \{Z_n \in \Omega \to B \times \Gamma \times \Delta \mid n \in \mathbb{N}\})$, where

- $(\Omega, \mathcal{A}, \mathbb{P})$ forms a probability space;
- B, Γ, Δ are finite sets;
- $Z_n \in \Omega \to B \times \Gamma \times \Delta$ $(n \in \mathbb{N})$ is a sequence of independent random variables such that for every $n \in \mathbb{N}$
  - also $Z_n.1$, $Z_n.2$ and $Z_n.3$ are independent and
  - $\forall \beta \in B : \mathbb{P}[\ \{\omega \in \Omega \mid Z_n.1(\omega) = \beta\}\ ] > 0$,
  - $\forall \gamma \in \Gamma : \mathbb{P}[\ \{\omega \in \Omega \mid Z_n.2(\omega) = \gamma\}\ ] > 0$,
  - $\forall \delta \in \Delta : \mathbb{P}[\ \{\omega \in \Omega \mid Z_n.3(\omega) = \delta\}\ ] > 0$.

□

Having a basic sample process the triple $Z_n(\omega) \in B \times \Gamma \times \Delta$ is taken for the parameters in the n-th iteration cycle of the GSP. Notice that if we take singletons for B, Γ and Δ then we obtain a deterministic case. In the sequel we assume that a basic sample process $(\Omega, \mathcal{A}, \mathbb{P}, B, \Gamma, \Delta, \{Z_n \in \Omega \to B \times \Gamma \times \Delta \mid n \in \mathbb{N}\})$ is given. The introduction of random parameters requires that we extend the signature of hyper manipulations.

Definition 4.2.2

A *random hyper manipulation* is a partial function with the signature

$$m : \Gamma \times C^* \nrightarrow C^*.$$

For random hyper manipulations the set of all *parent-lists* is the set of all those lists of candidates that are capable of producing offspring:

$$P_M = \{ x \in C^* \mid \exists\, m \in M \; \exists\, \gamma \in \Gamma : (\gamma, x) \in dom(m) \}.$$

□

If it can not lead to confusion then we shall often omit the predicates 'random' and 'hyper' only mentioning manipulations.

From now on we assume that next to the search space C and a basic sample process also a set of manipulations M is given.

Definition 4.2.3

A *selection function* is to select a set of parent-lists from the actual population. It is a partial function $F_s : B \times C^* \nrightarrow \mathcal{P}(P_M)$, where

$$\forall \beta \in B \; \forall x \in C^* \; \forall y \in F_s(\beta, x) : y \subseteq x.$$

□

Definition 4.2.4

A *production function* is to produce the children of a parent-list by the previously given manipulations. It is a function $F_p : \Gamma \times P_M \to C^*$, such that

$$\forall \gamma \in \Gamma \; \forall x \in P_M \; \exists\, m \in M : F_p(\gamma, x) = m(\gamma, x).$$

□

Notice that by this definition we allow that there are more random hyper manipulations that are used alternating in the successive production steps.

Definition 4.2.5

A *reduction function* reduces the set of old and newborn candidates and determines the 'survivors' for the next iteration cycle. Formally it is a function $F_r : \Delta \times C^* \times C^* \to C^*$, such that

$$\forall \delta \in \Delta \; \forall x \in C^* \; \forall y \in C^* : F_r(\delta, x, y) \subseteq x \cup y.$$

□

Definition 4.2.6
An *evaluation function* is to determine whether a population is good to terminate with. Here we shall use it with the signature

$$F_e : C^* \rightarrow \{true, false\}.$$

□

In practice the above functions F_s, F_p, F_r and F_e may depend on some extra parameters too (eg. the number of iterations made) but here we do not denote this dependence.

Let $(\Omega, \mathcal{A}, \mathbb{P}, B, \Gamma, \Delta, \{Z_n \in \Omega \rightarrow B \times \Gamma \times \Delta \mid n \in \mathbb{N}\})$ be a basic sample process. Its incorporation in the GSP happens by drawing an $\omega \in \Omega$ randomly and taking the corresponding realizations of Z_n, that is $(\beta_n, \gamma_n, \delta_n) = Z_n(\omega)$, as the random parameters in the n-th iteration cycle of the procedure.

Furthermore, let $x_{init} \in C^*$ and let F_s, F_p, F_r, F_e be a selection, a production, a reduction and an evaluation function, respectively. The unfolded version of the schema of the GSP is as follows:

General Search Procedure

$x := x_{init}$
WHILE NOT $F_e(x)$ DO
BEGIN
  get β, γ and δ
  $y := F_s(\beta, x)$
  $z := \bigcup_{q \in y} F_p(\gamma, q)$
  $x := F_r(\delta, x, z)$
END
Output the actual population

From now on a search procedure is understood as an instance of the GSP. To specify such an instance one needs to define C, M, F_s, F_p, F_r, F_e and $x_{init} \in C$. These items can therefore be seen as parameters, the values of these parameters determine the procedure. This motivates the following definition.

Definition 4.2.7
Given a basic sample process, a corresponding *stochastic search procedure* is a 7-tuple

$$(C, M, x_{init}, F_s, F_p, F_r, F_e),$$

where C is an arbitrary set, M is a set of hyper manipulations on C, $x_{init} \in C^*$, F_s, F_p, F_r and F_e are a selection, a production, a reduction, and an evaluation function, respectively.
A *deterministic search procedure* is a search procedure that belongs to a basic sample process where the sets Β, Γ and Δ are singletons.
□

When considering deterministic search procedures we shall often omit the reference to the random parameters β, γ and δ. In such cases we use a defective signature of the functions of the GSP leaving out Β, Γ and Δ from their domain.

A search procedure is creating populations successively. This results in a sequence of populations which will be called evolution. For an exact definition of this notion we introduce the transition function.

Definition 4.2.8
The *transition function belonging to a search procedure* is a function

$$F_t : (B \times \Gamma \times \Delta) \times C^* \to C^*$$

to create the next population 'in one go'. Formally it is defined as

$$F_t((\beta,\gamma,\delta),x) = F_r(\delta, x, \bigcup_{q \in F_s(\beta,x)} F_p(\gamma,q)).$$

□

Definition 4.2.9
The *evolution belonging to a search procedure* is a sequence

$$\{X_n(\omega) \in C^* \mid n \in \mathbb{N}\},$$

where

$$X_0(\omega) = x_{init},$$

$$X_{n+1}(\omega) = F_t(Z_n(\omega), X_n(\omega)) \quad \text{for } n \geq 0.$$

□

To provide an easier reading of the formulae we often leave out the symbol ω from the notation, and abbreviate $X_n(\omega)$ by X_n. In such cases $\mathbb{P}[property(X_n)]$ means $\mathbb{P}[\ \{\omega \in \Omega \mid property(X_n(\omega))\}\ ]$.

Notice that we obtain different evolutions for different initial populations. Therefore we use a notation that indicates the dependence on the initial population:

- $\{X_n \mid n \in \mathbb{N}\}_x$ denotes the evolution with $x = x_{init}$ and
- $\mathbb{P}_x[\,..X_n..]$ stands for $\mathbb{P}[\,..X_n.. \mid X_0 = x]$.

The question whether we can apply a (stochastic) search procedure to solve a search problem can be divided into two questions:

- whether the search procedure is suited to the given search problem, and
- whether we can hope that the search procedure finds a solution of the given search problem.

The following definition is to formalize what we mean by "suited to" above.

Definition 4.2.10

A search procedure $(C, M, x_{init}, F_s, F_p, F_r, F_e)$ *fits a search problem* $(D, \varphi_f, \varphi_g)$ if

$$C = D_f = \{\ d \in D \mid \varphi_f(d)\ \}$$

and

$$\forall x \in C^* : [\ (\exists c \in x : \varphi_g(c)) \Rightarrow F_e(x)\ ].$$

□

There are more possibilities to formalize the kernel of the second question above "whether we can hope that". The weakest formal conditions could be for instance

$$\mathbb{P}_{x_{init}}[\ \exists n \in \mathbb{N}\ \exists c \in D_g : c \in X_n] > 0$$

or

$$\exists n \in \mathbb{N} : \mathbb{P}_{x_{init}}[\ \exists c \in D_g : c \in X_n] > 0,$$

where $D_g = \{\ d \in D \mid \varphi_g(d)\ \}$.

The following proposition shows that there is no difference between these two formulations.

Proposition 4.2.11

For every $x \in C^*$, evolution $\{X_n \mid n \in \mathbb{N}\}_x$ and $c \in C$

$$\exists\, n \in \mathbb{N} : \mathbb{P}_x[c \in X_n] > 0 \quad \Leftrightarrow \quad \mathbb{P}_x[\exists\, n \in \mathbb{N} : c \in X_n] > 0.$$

Proof

Let us take an arbitrary $x \in C^*$, $c \in C$ and introduce $A_n = \{\omega \in \Omega \mid c \in X_n(\omega)\}$ as an abbreviation and observe that $\bigcup_{n\in\mathbb{N}} A_n = \{\omega \in \Omega \mid \exists\, k \in \mathbb{N} : c \in X_k(\omega)\}$.

$\Rightarrow$

If $0 < \mathbb{P}_x[A_k]$ for a certain $k \in \mathbb{N}$, then

$$0 < \mathbb{P}_x[A_k] \le \sum_{n\in\mathbb{N}} \mathbb{P}_x[A_n] = \mathbb{P}_x[\bigcup_{n\in\mathbb{N}} A_n].$$

$\Leftarrow$

If $\mathbb{P}_x[\bigcup_{n\in\mathbb{N}} A_n] = 0$ and there is no $k \in \mathbb{N}$ with $0 < \mathbb{P}_x[A_k]$ then

$$\mathbb{P}_x[\bigcup_{n\in\mathbb{N}} A_n] = \sum_{n\in\mathbb{N}} \mathbb{P}_x[A_n] = 0$$

which is a contradiction.

□

Definition 4.2.12

A search procedure $(C, M, x_{init}, F_s, F_p, F_r, F_e)$ *is likely to solve a search problem* $(D, \varphi_f, \varphi_g)$ if

$$(C, M, x_{init}, F_s, F_p, F_r, F_e) \text{ fits } (D, \varphi_f, \varphi_g)$$

and

$$\mathbb{P}_{x_{init}}[\exists\, n \in \mathbb{N}\, \exists\, c \in D_g : c \in X_n] > 0.$$

□

Observe that whether or not a search procedure is likely to solve a search problem is formally dependent on the initial population x_{init}. This is fully conform to our intuition and stresses the importance of having a good method to create an initial population. This, however, is not easy in general; it can be quite difficult to create an element of C, that is - in terms of the search problem - to create a feasible candidate $d \in D_f$.

Notice that as a matter of fact there are two phases within a search procedure, an *initialization phase* and an *iteration phase*. In the initialization phase an initial population is created, that is a set of feasible candidates. In the iteration phase

new feasible candidates are produced repeatedly from the old ones in order to reach a solution.

In general we can not say much about how to construct initial candidates. It is, however, remarkable that the creation of a good initial candidate can often be carried out iteratively. To see how recall the remark after Definition 3.1.3 where we observed that it is common to define candidates as certain constructions based on a set of elementary objects and some construction rules. Next we sketch an iterative way of initialization for a search problem $(D, \varphi_f, \varphi_g)$ where the elements of the free search space D are constructed from a set of elementary objects by a finite set of construction rules. Let us denote the empty construction by ε. Then the construction of a feasible candidate can be performed by the following search procedure.

Scheme of the Iterative Construction (IC) procedure

x = [ε]
WHILE NOT $\exists\, d \in x : \varphi_f(d)$ DO
  BEGIN
    $F_s([d]) = \{[d]\}$
    Produce d' by modifying d according to a construction rule or its inverse
    $F_r([d], [d']) = [d']$
  END

Note that this is a single point search procedure maintaining a population with cardinality 1. Observe that the *Initialization* step, *Goal, Selection* and *Reduction* are defined problem independently here. This means that if we want to perform the initialization phase of a complete search procedure by another search procedure then we can easily apply the IC procedure only having to define its way of *Production*. In other words it is *Production* where the problem dependent heuristics belong.

For the special case when the search problem belongs to a planning problem we can give a more detailed version - a subtype of the IC procedure. Let $(\mathcal{P}(A \times T), \varphi_f, \varphi_g)$ be the natural search problem corresponding to a planning problem, see definition 3.2.6. To create a feasible initial plan we take another

search problem $(\mathcal{P}(A \times T), \tilde{\varphi}_f, \tilde{\varphi}_g)$, where

$\tilde{\varphi}_f(P) \equiv true,$

and

$\tilde{\varphi}_g(P) \Leftrightarrow \varphi_f(P).$

Definition 4.2.13

The *Iterative Plan Construction for Initialization* (IPCI) procedure is for the above search problem; formally it is $(\tilde{C}, \tilde{M}, \tilde{P}_{init}, \tilde{F}_s, \tilde{F}_p, \tilde{F}_r, \tilde{F}_e)$ as defined below.

$\tilde{C} = \mathcal{P}(A \times T).$

Let

$$m^1_{(a,t)}(P) = P \setminus \{(a,t)\}$$

and

$$m^2_{(a,t)}(P) = P \cup \{(a,t)\}$$

for every $(a,t) \in A \times T$ and $P \in \mathcal{P}(A \times T)$ and let us define $\tilde{M}$ as

$$\tilde{M} = \{ m^1_{(a,t)} \mid (a,t) \in A \times T \} \cup \{ m^2_{(a,t)} \mid (a,t) \in A \times T \}.$$

$\tilde{P}_{init} = [\emptyset].$

$\tilde{F}_s([P]) = \{[P]\}.$

$$\tilde{F}_p([P]) = \begin{cases} m^1_{(a,t)}(P) & \text{for an } (a,t) \in \varphi_f^P \quad \text{if } \Phi \\ m^2_{(a,t)}(P) & \text{for an } (a,t) \in P \quad \text{if } \neg\Phi \end{cases}$$

where $\varphi_f^P = \{ (x,y) \in A \times T \mid \varphi_f(P \cup \{(x,y)\}) \}$, Φ stands for a problem dependent condition and also the choice of taking an operation $(a,t) \in \varphi_f^P$ for $m^1_{(a,t)}(P)$ and an operation $(a,t) \in P$ for $m^2_{(a,t)}(P)$ is problem dependent.

$\tilde{F}_r([P], [Q]) = [Q].$

$\tilde{F}_e([P]) = true \Leftrightarrow \varphi_f(P).$

□

Observe that the IPCI procedure is constructing a feasible plan from the empty plan by adding and deleting operations. This means that we could partially automate the construction phase of search procedures applied to planning. Of

course, we do not claim that we hereby answered the question of 'how to construct an initial plan when solving a planning problem by search'. Instead, we presented a framework that permits to sharpen this question to 'by which condition Φ and which way of choosing (a,t) can we construct an initial plan when solving a planning problem by search'. This implies that if one applies the IPCI procedure to a certain problem then defining these items is sufficient to have an initial feasible plan constructed.

In the same spirit we can also apply the IC procedure to solve a whole search problem. Let $(\mathcal{P}(A \times T), \varphi_f, \varphi_g)$ be the natural search problem corresponding to a planning problem. Through defining another search problem $(\mathcal{P}(A \times T), \bar{\varphi}_f, \bar{\varphi}_g)$ by

$$\bar{\varphi}_f(P) \equiv true,$$

and

$$\bar{\varphi}_g(P) \Leftrightarrow \varphi_f(P) \wedge \varphi_g(P).$$

we can obviously apply the IC procedure to construct a solution for $(\mathcal{P}(A \times T), \varphi_f, \varphi_g)$, i.e. to construct a plan P with $\varphi_f(P) \wedge \varphi_g(P)$.

Definition 4.2.14

The *Iterative Plan Construction for Solution* (IPCS) procedure is $(\bar{C}, \bar{M}, \bar{P}_{init}, \bar{F}_s, \bar{F}_p, \bar{F}_r, \bar{F}_e)$ where each component is the same as in the IPCI procedure except that

$$\bar{F}_e([P]) = true \Leftrightarrow \varphi_f(P) \wedge \varphi_g(P).$$

□

Notice again, that we hereby did not answer the question of 'how to solve a planning problem by construction'. We, however, presented a procedure that reduces this question to 'by which condition Φ and which way of choosing (a,t) can we solve a planning problem by construction'. Several methods based on using dispatch rules can be considered as special cases of the above IPCS procedure.

In the sequel we focus our attention on the 'real' search phase of search procedures; in 4.4 we investigate properties of iterative procedures applied for optimization problems.

4.3 Examples of Search Procedures

Recall that by developing the GSP we were aiming at identifying the most essential components of a wide class of search procedures. To justify that we have achieved this aim we specify types within the framework of the GSP that coincide with well-known types of algorithms. These algorithms appear under different labels like heuristic search, graph search, local search, neighbourhood search in the literature, and in this section we show that they all can be considered as special cases of our GSP.

4.3.1 Genetic Algorithms

Genetic algorithms, cf. Goldberg (1989), are approximation algorithms applied to a search problem where φ_g is defined by an objective function $f : C \rightarrow \mathbb{R}$. In a classical genetic algorithm (GA) a candidate $c \in C$ is a finite binary sequence with a fixed length $k > 1$. The standard genetic production methods are crossover of two parents and mutation of single candidates. Genetic crossover takes two sequences $(u_1, \ldots, u_k)$, $(v_1, \ldots, v_k)$, a randomly chosen position $n \in \{1, \ldots, k\}$ and creates two children:

$$(u_1, \ldots, u_{n-1}, v_n, \ldots, v_k),\ (v_1, \ldots, v_{n-1}, u_n, \ldots, u_k).$$

The standard mutation changes one value at a randomly chosen position in a candidate, producing $(u_1, \ldots, 1\text{-}u_n, \ldots, u_k)$ from $(u_1, \ldots, u_k)$.
The typical genetic selection and reduction are based on a survival-of-the-fittest mechanism, preferring candidates with a low objective function value (in case of minimization).
The appropriate, although partial, instantiation of the GSP resulting in such a GA is the following.

$$C = \{0,1\}^k.$$

and let $cross : \Gamma \times C^2 \rightarrow C^2$ and $mut : \Gamma \times C^1 \rightarrow C^1$ stand for the usual crossover and mutation.

$$M = \{\ cross, mut\ \};$$

$$F_p(\gamma, x) = \begin{cases} cross(\gamma, [c,d]) & \text{if } x = [c,d] \\ mut(\gamma, [c]) & \text{if } x = [c] \end{cases}$$

Mostly there is a random *Selection* and *Reduction* mechanism in GAs that is based on the objective function value (fitness) of the candidates. It is typically made such that fitter candidates (with a lower objective function value) have a larger chance to become a parent and to survive.

4.3.2 Simulated Annealing

Just as GAs simulated annealing algorithms, cf. Aarts and Korst (1989), van Laarhoven (1988), are for function optimization where the goal is determined by an objective function $f: C \rightarrow \mathbb{R}$ over the search space C. To obtain a simulated annealing (SA) algorithm we need to take an arbitrary random manipulation

$$m : \Gamma \times C^1 \rightarrow C^1$$

and

$$\Delta \subseteq (0,1],$$
$$M = \{ m \},$$
$$F_s([c]) = \{[c]\},$$
$$F_p(\gamma,[c]) = [m(\gamma,c)],$$
$$F_r(\delta,[c],[d]) = \begin{cases} [d] & \text{if } exp\left[\dfrac{f(c)-f(d)}{p}\right] > \delta \\ [c] & \text{otherwise} \end{cases},$$

where $0 < \delta \leq 1$ by the definition of Δ and $p > 0$ is the so called cooling parameter decreasing along the evolution.

We remark that the usual simulated annealing (SA) terminology uses the notion of neighbourhoods. At the first sight it seems that SA algorithms rely on neighbourhoods independent from the manipulations of the search procedure. Deeper analysis, however, displays that SA people do not presume the presence of neighbourhoods given beforehand; they intuitively refer to the neighbourhoods induced by M as defined in Definition 4.1.4.

Observe that a simulated annealing algorithm can be considered as special GA, where children are produced exclusively by mutation.

4.3.3 Threshold Accepting, Hill Climbing

Threshold accepting (TA), Dueck and Scheuer (1988), is very similar to Simulated Annealing. The essential difference between TA and SA is in the different acceptance mechanisms, i.e. reduction functions. Namely, TA accepts a newly generated candidate if it is not much worse than the old one, while SA does it only with a probability. To describe TA let C, M, F_s and F_p the same as in section 4.3.2. Furthermore let

$\Delta = \{\delta\}$ with $\delta \geq 0$,

$$F_r(\delta,[c],[d]) = \begin{cases} [d] & \text{if } f(d) - f(c) > -\delta \\ [c] & \text{otherwise} \end{cases},$$

A well-known instance of Threshold Accepting is Hill Climbing where $\delta = 0$. From the foregoing it is easy to see that Threshold Accepting and Hill Climbing can be considered as special forms of simulated annealing.

4.3.4 Depth First Search

Depth first search (DFS), cf. Pearl (1984), is generally considered as a tree search algorithm assuming that during the search we are moving between the nodes of a tree along the edges. This feature show that DFS belongs to graph search procedures in the sense described in and after Definition 4.1.2. The name 'depth first' can be understood by observing that a DFS procedure always produces children of the first element of the population (a list) and places the new children in front of the old elements. This indeed can be seen as searching in depth - if only we take the depth of a candidate as the number of its ancestors. A depth first search procedure can be applied to an arbitrary search problem $(D, \varphi_f, \varphi_g)$ with.

$C = D_f$

$m : C \rightarrow C^*$ is arbitrary,

$M = \{ m \}$,

$F_s([c_1, \ldots, c_n]) = [c_i]$,

such that c_i is the first one of $c_1, \ldots, c_n$ with $c_i \in dom(m)$.

$F_p([c]) = m(c)$,

$F_r([c_1, \ldots, c_n], [d_1, \ldots, d_n]) = [d_1, \ldots, d_n, c_{i+1}, \ldots, c_n]$,

such that c_i is the first one of $c_1, \ldots, c_n$ with $c_i \in dom(m)$.

$F_e([c_1, \ldots, c_n]) \Leftrightarrow \exists\, i \in \{1, \ldots, n\} : \varphi_g(c_i)$.

Observe that by this schema a depth first search procedure can be defined by only defining a manipulation $m : C \nrightarrow C^*$ and giving an initial candidate.

4.3.5 Breadth First Search

Breadth first search (BFS), cf. Pearl (1984), is very similar to depth first search only differing in the way the list is reordered after generating the children. In other words it is only the selection function that distinguishes BFS and DFS. To obtain breadth first search as an instance of the GSP let all the components be as in section 4.3.4 except that

$F_r([c_1, \ldots, c_n], [d_1, \ldots, d_n]) = [c_{i+1}, \ldots, c_n, d_1, \ldots, d_n]$,

such that c_i is the first one of $c_1, \ldots, c_n$ with $c_i \in dom(m)$.

Notice that similarly to DFS we can fully define a breadth first search procedure by the applied manipulation and initial candidate.

4.3.6 Best First Search

Best first search (BES), cf. Pearl (1984), requires some measure to define 'best', i.e. we need an objective function $f : C \rightarrow \mathbb{R}$ to define φ_g in $(C, \varphi_f, \varphi_g)$. Furthermore let

$m : C \nrightarrow C^*$ be an arbitrary manipulation and

$M = \{ m \}$,

$F_s(x) \in \{ [c] \subseteq x \mid c \in dom(m) \text{ and } \forall\, d \in x : f(c) \leq f(d) \}$,

$F_p([c]) = m(c)$,

The characteristic behaviour of a BFS procedure is determined by the specific selection function. The reduction function (reordering the list) does not play a crucial role, therefore we omit its specification.

4.4 Convergence of Stochastic Search Procedures

In this chapter we investigate a special type of search: approximation procedures for combinatorial problems, Papadimitriou and Steigliz (1982). In combinatorial optimization the search space C is always finite and the goal of the search is determined by means of an objective function $f : C \to \mathbb{R}$ requiring that the search stops when an optimum (minimum) of f is reached. This objective function is guiding the search, candidates and populations can be compared according to their objective function value. By this feature we can distinguish special class of iterative search procedures. If there is an objective function to be optimized then the evolution is obviously 'trying' to reach better and better populations, therefore the name *improvement procedure* is appropriate. In this section we derive general conditions that imply that improvement procedures lead to an optimum, cf. Eiben, Aarts and van Hee (1991).

Further on in this section we assume that a basic sample process $(\Omega, \mathcal{A}, \mathbb{P}, B, \Gamma, \Delta, \{Z_n : n \in \mathbb{N}\})$ and a search procedure $(C, M, x_{init}, F_s, F_p, F_r, F_e)$ are given.

To formulate our first two lemmas as generally as possible we temporarily introduce a new random variable $Y_n : \Omega \to (C^* \to C^*)$ for every $n \in \mathbb{N}$ such that

$$Y_n(\omega)(x) = F_t(Z_n(\omega),x)$$

and thus

$$X_n(\omega) = Y_n(\omega)(X_{n-1}(\omega))$$

The assumption about the independence of the Z_n's naturally transfers to the Y_n's, i.e. it is assumed for every $n \in \mathbb{N}$ and $F_i \subseteq C^* \to C^*$ $(0 \le i \le n)$:

$$\mathbb{P}[Y_n \in F_n \wedge Y_{n-1} \in F_{n-1} \wedge \ldots \wedge Y_0 \in F_0] = \prod_{i=0}^{n} \mathbb{P}[Y_i \in F_i].$$

The first lemma expresses a simple rewriting rule.

Lemma 4.4.1

$$\mathbb{P}[X_n = y \mid X_{n-1} = z] = \mathbb{P}[Y_{n-1}(z) = y] \quad \forall\, n \ge 1, \forall\, x,y,z \in C^*.$$

Proof

It is trivial, we only remark that the independence of the Y_n's is necessary.

□

By definition each $X_n(\omega)$ is an element of C^* for every $\omega \in \Omega$. Therefore we can consider X_n not only as an abbreviation of $X_n(\omega)$ but also as a random variable $X_n : \Omega \to C^*$. On this basis the question whether an evolution $\{X_n \mid n \in \mathbb{N}\}_x$ is a Markov chain is formally correct.

Lemma 4.4.2
$\{X_n \mid n \in \mathbb{N}\}_x$ is a Markov chain, and if the Z_n's (Y_n's) have the same distribution then the chain is homogeneous.

Proof
Let $n > 0$, $x_i \in C^*$ ($i \in \{1, \ldots, n+1\}$). Then by the independence and Lemma 4.4.1 we get

$$\mathbb{P}[X_{n+1} = x_{n+1} \mid X_n = x_n \wedge \ldots \wedge X_0 = x] =$$
$$\mathbb{P}[Y_n(x_n) = x_{n+1} \mid Y_{n-1}(x_{n-1}) = x_n \wedge \ldots \wedge Y_0(x) = x_1] =$$
$$\mathbb{P}[Y_n(x_n) = x_{n+1}] =$$
$$\mathbb{P}[X_{n+1} = x_{n+1} \mid X_n = x_n],$$

which proves the Markov property.
If the Y_n's have the same distribution then by Lemma 4.4.1

$$\mathbb{P}[X_m = y \mid X_{m-1} = z] = \mathbb{P}[X_n = y \mid X_{n-1} = z]$$

is self-evident for any $y,z \in C^*$ and $m,n \in \mathbb{N}$.
□

The fact that the Z_n's have the same distribution can be seen as 'the way of producing offsprings remains basically the same from generation to generation'. This does not hold, for instance in SA algorithms, where the control parameter p is decreasing, hence the distribution of F_r, and hereby the distribution of F_t is changing.

To establish convergence we have to express formally that the algorithm tends to an optimum. Observe that we defined the search space in general as a set without any norm or distance measure. Therefore we can not expect convergence saying that X_n ($n \to \infty$) is getting close to an optimum. What remains is to require that X_n contains an optimum, or rather, that the chance of containing an optimum is growing to 1.

Let $C_{opt} = \{ c \in C \mid c$ is a minimum of $f \}$.

Definition 4.4.3

The chain $\{X_n \mid n \in \mathbb{N}\}_x$ is *monotone* if

$$\forall n \in \mathbb{N} : min\{ f(c) \mid c \in X_{n+1}\} \le min\{ f(c) \mid c \in X_n\}.$$

Remark 4.4.4

Observe that

$$\forall c_{opt} \in C_{opt}\ \forall n \in \mathbb{N} : c_{opt} \in X_n \Rightarrow c_{opt} \in X_{n+1}$$

is not necessarily true, but

$$\forall n \in \mathbb{N} : X_n \cap C_{opt} \neq \emptyset \Rightarrow X_{n+1} \cap C_{opt} \neq \emptyset$$

always holds for monotone chains.

□

Lemma 4.4.5

If $\{X_n \mid n \in \mathbb{N}\}_x$ is monotone then the following assertions are equivalent:

a) $\mathbb{P}_x[\exists n \in \mathbb{N} : X_n \cap C_{opt} \neq \emptyset] = 1,$

b) $\mathbb{P}_x[\lim_{n\to\infty} X_n \cap C_{opt} \neq \emptyset] = 1,$

c) $\lim_{n\to\infty} \mathbb{P}_x[(X_n \cap C_{opt}) \neq \emptyset] = 1$

Proof

Notice that if $A_n = \{\omega \in \Omega \mid X_n(\omega) \cap C_{opt} \neq \emptyset\}$ and $\{X_n \mid n \in \mathbb{N}\}$ is monotone then the sets $A_1, \ldots, A_n, \ldots$ form a monotone sequence due to Remark 4.4.4. The existence and the equality of $\lim_{n\to\infty} \mathbb{P}_x[A_n]$ and $\mathbb{P}_x[\lim_{n\to\infty} A_n]$ for monotone sequences is a well-known result of elementary measure theory. This implies the equivalence of (b) and (c).

The equivalence of (a) and (b) is straightforward if we consider that in this case $\lim_{n\to\infty} A_n = \bigcup_{n\in\mathbb{N}} A_n$.

□

Definition 4.4.6

For any $x \in C^*$ *the set of all populations that may occur in* $\{X_n \mid n \in \mathbb{N}\}_x$ is

$$succ(x) = \{y \in C^* \mid \exists n \in \mathbb{N} : \mathbb{P}[X_n = y \mid X_0 = x] > 0\}.$$

Furthermore, if $U \subseteq C^*$ then

$$succ(U) = \bigcup_{x \in U} succ(x).$$

□

The next theorem is our basic convergence result. The main idea underlying the proof is to have upper bounds on the probability of taking the wrong way, i.e. making steps in the search space that do not reach any optimum.

Theorem 4.4.7

Let $U \subseteq C^*$ and let the following hold

a) $\{X_n \mid n \in \mathbb{N}\}_x$ is monotone for every $x \in U$,

b) $n_k \in \mathbb{N}$ $(k \in \mathbb{N})$ such that $n_k \to \infty$ $(k \to \infty)$ and

$\varepsilon_k \in (0,1]$ $(k \in \mathbb{N})$ such that $\prod_{k=0}^{\infty} \varepsilon_k = 0$ and

$\forall\, y \in succ(U) : \mathbb{P}[X_{n_{k+1}} \cap C_{opt} = \emptyset \mid X_{n_k} = y] \le \varepsilon_k$ holds for every $k \in \mathbb{N}$.

Then $\mathbb{P}_x [\lim_{n \to \infty} (X_n \cap C_{opt}) \neq \emptyset] = 1$ for every $x \in U$.

Proof

Let us introduce $H = \{ y \in C^* \mid y \cap C_{opt} = \emptyset \}$ and choose an initial population $x \in H \cap U$.

Furthermore let $p_k = \mathbb{P}[X_{n_k} \cap C_{opt} = \emptyset \mid X_0 = x]$ $(k > 0)$.

Then for any $k > 0$ we have

$$p_{k+1} = \sum_{y \in H} \mathbb{P}[X_{n_{k+1}} \cap C_{opt} = \emptyset \mid X_{n_k} = y] \cdot \mathbb{P}[X_{n_k} = y \mid X_0 = x] \le$$

$$\sum_{y \in H} \varepsilon_k \cdot \mathbb{P}[X_{n_k} = y \mid X_0 = x] =$$

$$\varepsilon_k \cdot \sum_{y \in H} \mathbb{P}[X_{n_k} = y \mid X_0 = x] =$$

$$\varepsilon_k \cdot p_k.$$

This implies that

$$p_{k+1} \le \prod_{i=1}^{k} \varepsilon_k .$$

Hence

$$\lim_{k\to\infty} \mathbb{P}[X_{n_k} \cap C_{opt} = \emptyset \mid X_0 = x] = \lim_{n\to\infty} p_k \le \prod_{k=1}^{\infty} \varepsilon_k = 0.$$

Notice that the monotonicity of $\{X_n \mid n \in \mathbb{N}\}_x$ implies that the sequence $\mathbb{P}_x[X_n \cap C_{opt} = \emptyset]$ $(n \in \mathbb{N})$ is non increasing and then from $n_k \to \infty$ $(k \to \infty)$ we have that

$$\lim_{n\to\infty} \mathbb{P}_x[X_n \cap C_{opt} = \emptyset] \le \lim_{k\to\infty} \mathbb{P}_x[X_{n_k} \cap C_{opt} = \emptyset] = 0,$$

consequently

$$\lim_{n\to\infty} \mathbb{P}_x[X_n \cap C_{opt} \ne \emptyset] = 1$$

holds. Then by lemma 4.4.5 we obtain almost sure convergence:

$$\mathbb{P}_x[(\lim_{n\to\infty} X_n \cap C_{opt}) \ne \emptyset] = 1.$$

□

Theorem 4.4.8

Let $x \in C^*$ and the following conditions be satisfied:

a) $\{X_n \mid n \in \mathbb{N}\}_x$ is monotone, and

b) $\{X_n \mid n \in \mathbb{N}\}_x$ is homogeneous, and

c) $\mathbb{P}_x[\,\exists\, n \ge k : X_n \cap C_{opt} \ne \emptyset \mid X_k = y\,] > 0$ for every $y \in succ(x)$ and $k \in \mathbb{N}$.

Then $\mathbb{P}_x[(\lim_{n\to\infty} X_n \cap C_{opt}) \ne \emptyset] = 1.$

Proof

We apply Theorem 4.4.7 with $U = \{x\}$ by constructing a sequence $n_0, n_1, \ldots$ and a sequence $\varepsilon_0, \varepsilon_1, \ldots$ so that they satisfy its condition (b).

Let $y \in succ(x)$ and

$$M_y = min\ \{n \in \mathbb{N} \mid \mathbb{P}[X_n \cap C_{opt} \ne \emptyset \mid X_0 = y] > 0\}$$

be the minimum number of steps required to find an optimum with positive chance when taking y as initial population.

According to (b) and (c) M_y is finite for every $y \in succ(x)$. Then

$$M = max\ \{M_y \mid y \in succ(x)\}$$

is finite because C^* is finite, thus $succ(x)$ is finite. Hence

$$\forall\, y \in succ(x) : \mathbb{P}[X_M \cap C_{opt} \ne \emptyset \mid X_0 = y] > 0$$

holds by the monotonicity, and thus

$$\forall\, y \in succ(x) : \mathbb{P}[X_M \cap C_{opt} = \emptyset \mid X_0 = y] < 1. \qquad \text{(i)}$$

Introducing the abbreviation

$$p_y = \mathbb{P}[X_M \cap C_{opt} = \emptyset \mid X_0 = y]$$

we can define

$$p = max\ \{p_y \mid y \in succ(x)\ \}.$$

Notice that by (i) and the finiteness of $succ(x)$ we have that $p < 1$ and

$$\forall\, y \in succ(x) : \mathbb{P}[X_M \cap C_{opt} = \emptyset \mid X_0 = y] \ \leq\ p.$$

Let $n_k = M \cdot k$ and $\varepsilon_k = p$ $(k \in \mathbb{N})$. Observe that $n_k \to \infty$ $(k \to \infty)$ hold, and so does

$$\prod_{k=0}^{\infty} \varepsilon_k = 0 \ \ since\ p < 1.$$

What remains is to show that

$$\forall\, y \in succ(x) : \mathbb{P}[X_{n_{k+1}} \cap C_{opt} = \emptyset \mid X_{n_k} = y] \ \leq\ \varepsilon_k \quad \text{for every } k \geq 0. \qquad \text{(ii)}$$

By the homogeneity we have that for every $y \in succ(x)$

$$\mathbb{P}[X_{M\cdot(k+1)} \cap C_{opt} = \emptyset \mid X_{M\cdot k} = y] \ =\ \mathbb{P}[X_M \cap C_{opt} = \emptyset \mid X_0 = y] \ \leq\ p$$

holds. This proves (ii), and hereby also the proof of the theorem is complete.

□

Loosely applying Definition 4.2.12 we can consider this theorem as stating: if an optimization procedure is likely to solve a problem and its evolution is homogeneous and monotone then it surely solves the problem.

Definition 4.4.9

The reduction function $F_r : \Delta \times C^* \times C^* \to C^*$ is *conservative* if it always preserves the best f value, that is at least one of the optima. Formally this means that

$$F_r(\delta, x, y) \cap MIN_{xy} \neq \emptyset \quad \text{for every } x,y \in C^* \text{ and } \delta \in \Delta,$$

where

$$MIN_{xy} = \{\ c \in x \cup y \mid \forall\, d \in x \cup y : f(c) \leq f(d)\}$$

contains the minima of $x \cup y$.

□

Lemma 4.4.10

If the reduction function is conservative then the evolution $\{X_n \mid n \in \mathbb{N}\}_x$ is monotone.

Proof

Notice that for any arbitrary $y,z \in C^*$ and $\delta \in \Delta$

$$min\{\, f(c) \mid c \in z \,\} \;\geq\; min\{\, f(c) \mid c \in z \cup y \,\} \;\geq\; min\{\, f(c) \mid c \in F_r(\delta,z,y) \,\}$$

if F_r is conservative. By the definition of the transition function we have

$$X_{n+1} = F_r(\delta_n, X_n, \bigcup_{q \in F_s(\beta_n, X_n)} F_p(\gamma_n, q))$$

which implies

$$min\{\, f(c) \mid c \in X_{n+1}\} \;\leq\; min\{\, f(c) \mid c \in X_n\}.$$

□

Next we come to our original purpose to find restrictions on the functions of a search procedure such that together they imply convergence.

Definition 4.4.11

The set of manipulations M *connects the search space* C if for every $c,d \in C$ the candidate d is reachable from c by manipulations, that is:

$$\exists\, n \in \mathbb{N}\; \exists\, c_1 \in C \ldots \exists\, c_n \in C : c = c_1 \wedge d = c_n \wedge$$
$$\forall\, i \in \{1, \ldots, n\text{-}1\} : ([c_i],[c_{i+1}]) \in E_M,$$

where E_M is the set of edges induced by M.

□

In the following three definitions we define the same predicate for the selection, the production and the reduction function. We shall call them *generous* if they give a positive chance to every candidate, to become a parent, to be born and to survive, respectively.

Definition 4.4.12

If for every $c \in C$ it holds that $[c] \in \bigcup_{m \in M} dom(m)$ then the *selection function is generous* if in every iteration cycle, that is for every $n \in \mathbb{N}$

$$\forall\, x \in C^* \;\forall\, c \in x : \mathbb{P}[\, [c] \in F_s(Z_n.1, x)\,] > 0.$$

□

Definition 4.4.13

The *production function generous* if for every $n \in \mathbb{N}$

$\forall c \in C \; \forall d \in C : ([c],[d]) \in E_M \;\Rightarrow\; \mathbb{P}[\,[d] = F_p(Z_n.2, [c])\,] > 0.$

□

Definition 4.4.14

The *reduction function is generous* if for every $n \in \mathbb{N}$

$\forall x,y \in C^* \; \forall c \in x \cup y : \mathbb{P}[\, c \in F_r(Z_n.3, x, y)\,] > 0.$

□

Remark 4.4.15

We remark that the generousity of F_s, F_p and F_r imply that for every $n \in \mathbb{N}$

a) $\forall x \in C^* \; \forall c \in x \; \exists \beta \in B \;:\; \mathbb{P}[\, Z_n.1 = \beta\,] > 0 \;\wedge\; [c] \in F_s(\beta,x).$

b) $\forall c \in C \; \forall d \in C : ([c],[d]) \in E_M \Rightarrow$

$\exists \gamma \in \Gamma : \mathbb{P}[\, Z_n.2 = \gamma\,] > 0 \;\wedge\; [d] = F_p(\gamma,[c]).$

c) $\forall x,y \in C^* \; \forall c \in x \cup y \; \exists \delta \in \Delta \;:\; \mathbb{P}[\, Z_n.3 = \delta\,] > 0 \;\wedge\; c \in F_r(\delta,x,y).$

Theorem 4.4.16

Let us assume that the drawings Z_n's have the same distribution. Let the selection, the production and the reduction function be generous. Furthermore let the reduction function be conservative and let the given set of manipulations M connect C. Then for any initial population $\mathbb{P}_x\,[\lim_{n\to\infty}(X_n \cap C_{opt}) \neq \emptyset] = 1.$

Proof

The proof goes via Theorem 4.4.8, we show that its conditions (a), (b) and (c) hold for any $x \in C^*$.

a) F_r is now conservative and therefore $\{X_n \mid n \in \mathbb{N}\}_x$ is monotone by Lemma 4.4.10.

b) Since Z_n's have the same distribution $\{X_n \mid n \in \mathbb{N}\}_x$ is homogeneous by Lemma 4.4.2.

c) We show more than necessary, namely we prove

$\forall y \in C^* \; \forall c_{opt} \in C_{opt} : \mathbb{P}_y[\exists\, n \in \mathbb{N} : c_{opt} \in X_n] > 0.$

Let $c_{opt} \in C_{opt}$ and $c_0 \in y$ arbitrary. By the connectivity condition on M we have that there exists an $n \in \mathbb{N}$ and a sequence $c_1, \ldots, c_n$ from C, such that $c_{opt} = c_n$ and

$([c_0],[c_1]) \in E_M \;\wedge\; ([c_1],[c_2]) \in E_M \;\wedge\; \ldots \;\wedge\; ([c_{n-1}],[c_n]) \in E_M.$

Then we have

$$\mathbb{P}_y[c_{opt} \in X_n] \geq \mathbb{P}_y[c_1 \in X_1 \wedge \ldots \wedge c_{n-1} \in X_{n-1} \wedge c_n \in X_n]$$

$$= \mathbb{P}_y[c_1 \in F_t(Z_1,y) \;\wedge\; c_2 \in F_t(Z_2,F_t(Z_1,y)) \;\wedge \ldots \wedge\; c_n \in F_t(Z_n, \ldots F_t(Z_1,y) \ldots)]$$

$$= \sum_{z_1,\ldots,z_n} \mathbb{P}_y[c_1 \in F_t(Z_1,y) \;\wedge \ldots \wedge\; c_n \in F_t(Z_n, \ldots F_t(Z_1,y) \ldots) \;\wedge\; Z_1 = z_1 \;\wedge \ldots \wedge Z_n = z_n]$$

$$= \sum_{(z_1,\ldots,z_n)\in H} \mathbb{P}_y[Z_1 = z_1 \wedge \ldots \wedge Z_n = z_n]$$

$$= \sum_{(z_1,\ldots,z_n)\in H} \prod_{i=1}^{n} \mathbb{P}_y[Z_i = z_i] \qquad (*)$$

where

$$H = \{(z_1,\ldots,z_n) \in (B \times \Gamma \times \Delta)^n \mid c_1 \in F_t(z_1,y) \;\wedge \ldots \wedge\; c_n \in F_t(z_n, \ldots F_t(z_1,y) \ldots)\}.$$

If $H \neq \emptyset$ then (*) is positive by Remark 4.4.15 which proves $\mathbb{P}_y[c_{opt} \in X_n] > 0$. Showing $H \neq \emptyset$ is thus sufficient to prove the theorem. Therefore we need to construct a sequence $z_1,\ldots,z_n$ such that $c_i \in F_t(z_i, \ldots F_t(z_1,y) \ldots)$ holds for any $i \in \{1,\ldots,n\}$. Observe that
the generousity of F_s implies $\exists\, \beta_1 \in B : [c_0] \in F_s(\beta_1,y)$ and
the generousity of F_p implies $\exists\, \gamma_1 \in \Gamma : [c_1] = F_p(\gamma_1,[c_0])$,
hence

$$c_1 \in \bigcup_{q\in F_s(\beta_1,y)} F_p(\gamma_1,q).$$

Then by the generousity of F_r we have that

$$\exists\, \delta_1 \in \Delta : c_1 \in F_r(\delta_1,y, \bigcup_{q\in F_s(\beta_1,y)} F_p(\gamma_1,q)\,).$$

Hereby we proved the existence of a $z_1 = (\beta_1,\gamma_1,\delta_1)$ for which it holds that

$$c_1 \in F_t(z_1,y).$$

In the same way we can construct $z_2 = (\beta_2,\gamma_2,\delta_2)$ such that

$$c_2 \in F_t(z_2,y),$$

and so on until we have $(z_1,...,z_n) \in (B \times \Gamma \times \Delta)^n$ satisfying

$$c_1 \in F_t(z_1,y) \wedge \ldots \wedge c_n \in F_t(z_n, \ldots F_t(z_1,y) \ldots).$$

This verifies that $H \neq \emptyset$ and completes the proof of the theorem.

□

Next we relax the requirements on M at the cost of a further restriction of the reduction function.

Definition 4.4.17

The set of manipulations M *quasi connects* C if there exists a *source element* $\sigma \in C$ such that any $d \in C$ is reachable from σ by manipulations. Formally it means that

$$\exists\, n \in \mathbb{N}\, \exists\, c_1 \in C \ldots \exists\, c_n \in C : [\, \sigma = c_1 \wedge d = c_n \wedge \forall\, i \in \{1, \ldots, n\text{-}1\} : ([c_i],[c_{i+1}]) \in E_M\,],$$

where E_M is the set of edges induced by M.

□

Definition 4.4.18

If the given set of manipulations M quasi connects C and $\sigma \in C$ a source element as given in Definition 4.4.17 then the reduction function is called σ-*preserving* if

$$\forall\, x,y \in C^* \;\forall\, \delta \in \Delta : \sigma \in x \;\Rightarrow\; \sigma \in F_r(\delta,x,y).$$

□

Lemma 4.4.19

Let $x \in C^*$ and $\sigma \in C$ be a source element. If $\sigma \in x$ and the reduction function is σ-preserving, then

$$\forall\, y \in succ(x) : \sigma \in y.$$

Proof The proof is trivial.

□

Theorem 4.4.20

Let us assume that the drawings Z_n's have the same distribution. Let the selection, the production and the reduction function be generous and the reduction function be conservative. Furthermore let the given set of manipulations M quasi connect C, $\sigma \in C$ be a source element and let the reduction function be σ-preserving.

Then for any initial population x with $\sigma \in x$ it holds that

$$\mathbb{P}_x[\lim_{n\to\infty} (X_n \cap C_{opt}) \neq \emptyset] = 1.$$

Proof

Again, the proof is based on Theorem 4.4.8. Let us take $x \in C^*$ such that $\sigma \in x$. The conditions (a) and (b) of Theorem 4.4.8 hold by the same reasoning as in the proof of Theorem 4.4.16.

c) We show that

$$\forall\, y \in succ(x)\ \forall\, c_{opt} \in C_{opt} : \mathbb{P}_y[\,\exists\, n \in \mathbb{N} : c_{opt} \in X_n] > 0.$$

Let $y \in succ(x)$ and $c_{opt} \in C_{opt}$ be arbitrary. Due to $\sigma \in x$ and Lemma 4.4.19 we have that $\sigma \in y$.

Then taking $c_0 = \sigma$ the quasi connectivity condition on M implies

$$\exists\, n \in \mathbb{N}\ \exists\, c_1 \in C \ldots \exists\, c_n \in C : \sigma = c_1 \wedge d = c_n \wedge$$
$$\forall\, i \in \{1, \ldots, n\text{-}1\} : ([c_i],[c_{i+1}]) \in E_M.$$

The rest of the proof is identical to that of Theorem 4.4.18.

□

Notice that for simulated annealing Theorem 4.4.8, Theorem 4.4.16 and Theorem 4.4.20 cannot be applied. As we remarked after Lemma 4.4.2, in a simulated annealing algorithm the control parameter is decreasing, hence the distribution of the transition function is changing. Therefore, it is only Theorem 4.4.7 that we can apply to SA, since in its conditions only monotonicity of the evolution is required, homogeneity is not. At first sight it seems that we can not apply this theorem either, since the evolution of a standard SA algorithm is not monotone. This problem, however, is easy to overcome by slightly modifying standard SA such that it preserves its characteristic features but becomes monotone. In the following definition we present extended SA where we maintain an extra element in the population: a best candidate seen so far.

Definition 4.4.21

We define *extended simulated annealing* (ESA) as the following instance of GSP.

Δ, M and F_p are the same as in section 4.3.2.

Furthermore let

$$F_s(\beta,[c,c_b]) = [c] \quad \text{for every } \beta \in B$$

and

$$F_r(\delta,[c,c_b],[d]) = \begin{cases} [d,c_b'] & \text{if } exp\left[\dfrac{f(c) - f(d)}{p}\right] > \delta \\ [c,c_b'] & \text{otherwise} \end{cases}$$

where

$$c_b' = \begin{cases} c_b & \text{if } f(d) \geq f(c_b) \\ d & \text{if } f(d) < f(c_b) \end{cases} \quad \text{and}$$

$p \in \mathbb{R}$ is the usual control parameter for SA.

□

It is easy to see the the above reduction function is conservative, hence the evolution belonging to an ESA algorithm is always monotone. Notice that for ESA the successive populations of the evolution are all from C^2. From now on a standard population will be denoted by a list $[c,c_b]$ if we want to emphasize the presence of 'the best seem so far' or by a list $[c, .]$ if we concentrate on the 'real' element c and c_b does not play a role. For a perfect matching with the usual SA terminology we take the viewpoint of neighbourhood search, that is instead of using manipulations we shall express ourselves in terms of neighbourhoods. This means that if $N : C \to C^*$ denotes the neighbourhood function induced by M, then we rewrite the production function in the form

$F_p(\gamma, [c]) \in N(c)$.

Furthermore, we adopt the following assumptions from Aarts and Korst (1989).

1) The inhomogeneous Markov chain of the evolution is a sequence of homogeneous Markov chains of the same finite length L. This means that we keep the control parameter constant for L cycli, i.e. in the i-th cycle we use the control parameter p_i ($i \in \mathbb{N}$) which is defined by a sequence p_k' ($k \in \mathbb{N}_0$) as follows:

$p_i = p_k'$ if $k \cdot L < i \leq (k+1) \cdot L$;

2) $|N(c)| = K$ for every $c \in C$;

3) $\mathbb{P}[F_p(Z_n.2, [c]) = [d]] = 1/K$ for every $c \in C$, $d \in N(c)$ and $n \in \mathbb{N}$.

Theorem 4.4.22

Let C be the search space and $f : C \to \mathbb{R}$ be the objective function. Let us consider an ESA algorithm for which the above conditions (1), (2) and (3) hold. Furthermore, we assume that for any evolution $\{X_n \mid n \in \mathbb{N}\}_x$

(4) $\forall\, c,d \in C\ \exists\, n \geq 1\ \exists\, c_0, \ldots, c_n \in C$ such that $c_0 = c$ and $d = c_n$ and
$\mathbb{P}_x[X_{k+1} = [c_{k+1}, .\,] \mid X_k = [c_k, .\,]\,] > 0, \quad k \in \{0, 1, \ldots, n\text{-}1\}$

and

(5) $p'_k \geq \dfrac{(L+1) \cdot \Delta}{log(k+D)} \qquad k \in \mathbb{N}$

hold, where

$D > 0$ is an arbitrary constant,

$\Delta = max\ \{ f(c) - f(d) \mid c \in C, d \in N(c) \}$,

L (the length of the homogeneous subchains) is the maximum of the minimum number of steps required to reach an optimum from y for all $y \in C^2$

are assumed to be finite.

Then $\mathbb{P}_x[\lim_{n\to\infty} X_n \cap C_{opt} \neq \emptyset] = 1$ for any initial $x \in C^2$.

Proof

Let $x \in C^2$ be the initial population.

a) It is easy to see that the evolution belonging to ESA is always monotone due to c_b.

b) We construct $n_k \in \mathbb{N}$ and $\varepsilon_k \in (0,1]$ $(k > 0)$ such that $n_k \to \infty$ $(k \to \infty)$ and

$\prod_{k=1}^{\infty} \varepsilon_k = 0$ and

$\forall\, y \in C^2 : \mathbb{P}_x[X_{n_{k+1}} \cap C_{opt} = \emptyset \mid X_{n_k} = y] \leq \varepsilon_k$ holds for every $k \in \mathbb{N}$.

This construction will be done by the following steps.

b1) We determine $\mathbb{P}_x[X_{i+1} = [c_{i+1}, .\,] \mid X_i = [c_i, .\,]\,]$ depending on the control parameter.

b2) By (b1) and (5) we deduce an upper bound ε_k for
$\mathbb{P}_x[X_{k \cdot L} \cap C_{opt} = \emptyset \mid X_{(k-1) \cdot L} = y]$.

b3) We show that $\prod_{k=1}^{\infty} \varepsilon_k = 0$, then applying Theorem 4.4.7 with $n_k = k \cdot L$ the proof is complete.

b1) $\mathbb{P}_x[\, X_{i+1} = [c_{i+1}, .\,] \mid X_i = [c_i, .\,]\,]$

=

$$\mathbb{P}_x[\, F_s(Z_i, 1, [c_i, .\,]) = [c_i]\,] \cdot \mathbb{P}_x[\, F_p(Z_i, 2, [c_i]) = [c_{i+1}]\,] \cdot$$
$$\cdot\, \mathbb{P}_x[\, F_r(Z_i, 3, [c_i, ., c_{i+1}]) = [c_{i+1}, .]\,]$$

=

$$1/K \times exp\left[\, \frac{(f(c_i) - f(c_{i+1}))^+}{p} \,\right]$$

b2) According to the definition of L, from any $y \in C^2$ we can reach an optimum in not more than L steps. If monotonicity holds then we do not loose optimal objective function values, thus from any $y \in C^2$ we can reach an optimum in exactly L steps as well. This implies that for any $k > 0$, $y \in C^2$ there exist $y_0, \ldots, y_L$ from C^2 such that $y_0 = y$, $y_L \cap C_{opt} \neq \emptyset$ and

$$\mathbb{P}_x[X_{(k-1)\cdot L+i+1} = y_{i+1} \mid X_{(k-1)\cdot L+i} = y_i\,] > 0 \quad \text{for every } i \in \{0, \ldots, L\text{-}1\}.$$

Hence

$$\mathbb{P}_x[X_{k\cdot L} \cap C_{opt} \neq \emptyset \mid X_{(k-1)\cdot L} = y]$$

≥

$$\mathbb{P}_x[X_{k\cdot L} = y_L \mid X_{(k-1)\cdot L} = y]$$

≥

$$\mathbb{P}_x[X_{k\cdot L} = y_L \mid X_{k\cdot L-1} = y_{L-1}] \times \ldots \times \mathbb{P}[X_{(k-1)\cdot L+1} = y_1 \mid X_{(k-1)\cdot L} = y]$$

=

$$\prod_{i=0}^{L} 1/K \times exp\left[\, - \frac{(f(c_i) - f(c_{i+1}))^+}{p'_k} \,\right]$$

where $c_i \in C$ is such $y_i = [c_i, .\,]$ for every $i \in \{0, \ldots, L\text{-}1\}$.
Then by the definition of Δ we have

$$exp\left[\, - \frac{\Delta}{p'_k} \,\right] \leq exp\left[\, - \frac{(f(c_i) - f(c_{i+1}))^+}{p'_k} \,\right]$$

and by the lower bound on p'_k we obtain that

$$exp\left[\, - \frac{log(k+D)}{L+1} \,\right] \leq exp\left[\, - \frac{\Delta}{p'_k} \,\right]$$ holds, hence

$$\prod_{i=0}^{L} 1/K \cdot exp\left[- \frac{(f(c_i)-f(c_{i+1}))^+}{p'_k} \right] \geq \left[1/K \cdot exp\left[- \frac{log(k+D)}{L+1} \right] \right]^L .$$

This leads to

$$\mathbb{P}[X_{k \cdot L} \cap C_{opt} \neq \emptyset \mid X_{(k-1) \cdot L} = y] \geq \left[1/K \cdot exp\left[- \frac{log(k+D)}{L+1} \right] \right]^L .$$

which is equivalent to

$$\mathbb{P}[X_{k \cdot L} \cap C_{opt} = \emptyset \mid X_{(k-1) \cdot L} = y] < 1 - \left[1/K \cdot exp\left[- \frac{log(k+D)}{L+1} \right] \right]^L$$

that is

$$\mathbb{P}[X_{k \cdot L} \cap C_{opt} = \emptyset \mid X_{(k-1) \cdot L} = y] < 1 - \frac{1}{K^L} \cdot \frac{(k+D)^{\frac{1}{L+1}}}{k+D} .$$

b3) Choosing $n_k = k \cdot L$ and $\varepsilon_k = 1 - \frac{1}{K^L} \cdot \frac{(k+D)^{\frac{1}{L+1}}}{k+D}$ we need to prove

$$\prod_{k=1}^{\infty} \varepsilon_k = 0$$

which requires

$$\lim_{n \to \infty} \prod_{k=1}^{n} \varepsilon_k = 0.$$

To prove this recall from mathematical analysis that for any sequence $a_k \in \mathbb{R}$ ($k \in \mathbb{N}$) it holds that

$$\lim_{n \to \infty} \prod_{k=1}^{n} (1 - a_k) = 0 \quad \text{iff} \quad \sum_{k=1}^{\infty} log(1 - a_k) = - \infty$$

Noticing that $log(1 - a_k) \leq - a_k$ is generally true we can conclude that

$$\sum_{k=1}^{\infty} a_k = \infty \quad \text{implies} \quad \lim_{n \to \infty} \prod_{k=1}^{n} (1 - a_k) = 0.$$

Now choosing $a_k = \frac{1}{K^L} \cdot \frac{(k+D)^{\frac{1}{L+1}}}{k+D}$ we have $\varepsilon_k = 1 - a_k$ and

$$\sum_{k=1}^{\infty} a_k = \infty.$$

This latter implies $\lim_{n \to \infty} \prod_{k=1}^{n} \varepsilon_k = 0$ and completes the proof of the theorem.
□

Notice that based on our general convergence results for stochastic search we could prove almost sure convergence for extended simulated annealing in a straightforward way. This is a stronger form of convergence than the stochastic one proved under the same conditions in Aarts and Korst (1989). Besides the new convergence results this approach opens the way to convergence proofs through a general approach considering multicandidate populations.

CHAPTER 5

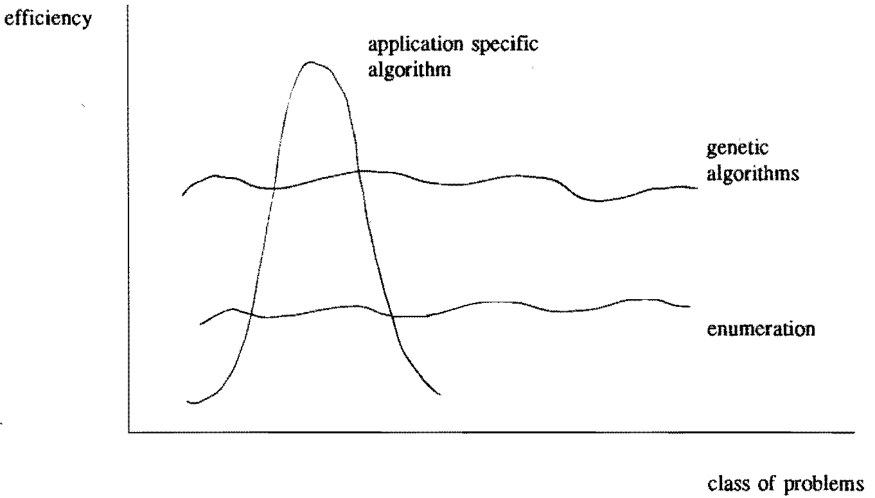
Searching by Generalized Genetic Algorithms

In Chapter 1 we emphasized the flexibility of the problem solving component of a DSS development tool. Therefore, we would like to have a generic algorithm that can be easily set to several different problem solving methods. We know that using the same generic algorithm for many different problem types with only some fine tuning on a given problem leads to a loss in efficiency. Nevertheless, the use of such an algorithm could save much effort when designing the problem solving component of a DSS; moreover it could provide an easy way of adapting it if the former version is not applicable anymore.

The GPS presented in the previous chapter is such a generic procedure. Its generality, however, is also somewhat disadvantageous. Having a procedure where more details are fixed, there are fewer components that one has to make oneself when applying the procedure in a specific case. This means that using a more specific version of the GSP as default search procedure could provide more support in DSS design then the General Search Procedure.

In this chapter we study genetic algorithms (GAs) as possible nominees for embodying a good balance between being general and detailed. Our reasons to choose GAs are threefold.

First, genetic algorithms are more and more recognized as robust problem solvers. Goldberg (1989) illustrates their performance by the following figure



Our second reason was the observation made in Chapter 4 that GAs offer a framework incorporating simulated annealing, threshold accepting and hill climbing. This implies that taking GAs as default search procedures we still have a considerable freedom.

Third, several authors mention the importance of a so called adapter function in decision making, cf. van Hee (1985). Verbeek (1990) describes this function as one "to acquire knowledge from plans already constructed in the form of detecting quality characteristics". This feature can be understood in two ways:

1) trying to gain knowledge from examining several planning sessions and making up new planning heuristics; or
2) regarding one planning session where we evaluate plans we make during planning and try to detect correlations between their structure and quality.

If we take the latter interpretation then we find a striking resemblance with the basic genetic principle, since a GA works by pursuing good quality gene patterns that contribute to a high fitness value.

Genetic algorithms are (stochastic) search methods based on biological principles. Although they have numerous applications in classifier systems and self-learning systems, cf. Grefenstette (1985,1987), Schaffer (1989), primarily they

are approximation algorithms to find the maximum (minimum) of an objective function over a finite search space. The biological analogies have helped a lot with inventing and investigating genetic algorithms, Holland (1975), De Jong (1980), Goldberg (1989). We feel, however, that these analogies form an obstacle in the sense that there are certain GA conventions that could be dropped without dropping the basic principles behind GAs. What are these basic principles? In our opinion, they are the following.

a) GAs are applied to a search space the points of which are finite sequences over an alphabet. The elements of a sequence are called genes, the sequences are seen as individuals or genotypes.
b) The objective function on the search space (or rather, a transformed version of it) is viewed as fitness of the individuals. The goal of a GA is to find individuals with maximal fitness value.
c) In an attempt to find an individual with a maximal fitness value, GAs try to detect and exploit correlation between the positioning of genes in individuals. As we described it in section 4.3.1, GAs perform this by taking two parents as samples and creating offspring from the genes of the parents.

Ad (a).

Determining the syntax of the individuals in a certain problem is often seen as a coding step. Again in biological terminology, one often considers the entities of the original problem, e.g. tours between cities or schedules of a job shop, as fenotypes that are coded to genotypes. Using sequences for genotypes is, however, not always the most natural choice. For instance if we have different types of genes then sequences are not the most appropriate way to structure them. For example, for job shop scheduling the sequence

$$[j_1, \ldots, j_n, m_1 \ldots, m_n, t_1 \ldots, t_n]$$

and the table

$$\begin{bmatrix} j_1 & \cdot & \cdot & \cdot & j_n \\ m_1 & \cdot & \cdot & \cdot & m_n \\ t_1 & \cdot & \cdot & \cdot & t_n \end{bmatrix}$$

can both represent a schedule where the job j_i is performed by the machine m_i beginning at the time t_i. The table, however, seems to be a more natural choice, not mentioning the easiness of distinguishing different types of genes simply by the row they are found in.

Considering classical genotypes as one dimensional ones, tables as two dimensional ones, one can easily figure cubic ones etc, for any dimension $n \in \mathbb{N}$. By such a generalization we remain within the borders of being 'genetic' if only our offspring production shows genetic features. As Suh and van Gucht (1987) state: "the selection of good representations and recombination operators is highly correlated".

Ad (b)

Surprisingly enough classic GAs optimize according to one fitness function although nature certainly judges its creatures by more criteria. This convention of GAs can be dropped too taking different priorities into account by using multiple fitness functions. This extension also allows us to apply different criteria at selecting parents and at choosing the survivors of a population. Since in Eiben and van Hee (1990) we discussed this matter in a broader context we do not go into details here.

Ad (c).

On one hand, restricting the number of parents to two is literally a natural restriction; biological offspring production never exceeds this number. On the other hand, this restriction is odd since most of the GA people are familiar with probability theory, one of the main principles of which is: more samples - more certainty. For GAs it would mean that having more parents one could expectedly increase the certainty of detecting the strong gene configurations. Preserving the GA principles one could take n ($n \geq 2$) parents and define gene recombination operators that produce children of them.

Although such production functions might not be crossover-like, they still should be considered as genetic ones if they are for propagating strong gene patterns. De Jong (1985,1987) addresses the problems of new representations and new gene recombination methods stressing the importance "to invent new operators better suited to the [new] representation".

5.1 Multiparent Production

In this section we define a generalized form of gene recombination based on the classic sequential genotypes where in general n ($n > 0$) parents can produce children. We present a general offspring production procedure that incorporates many known gene recombination operators. Hereby we are aiming at multiple objectives:

1) We explicitly point out the fact that crossover is only one way of creating children and so is our procedure. This may give impetus to inventing other non standard methods.
2) Within the framework of our procedure we identify components that might be problem specific. Hereby we locate where heuristics can be incorporated, with other words where domain knowledge can be used.
3) Our general procedure can be used as a framework that facilitates designing different recombination operators. Hereby it supports one of the crucial steps of creating a GA.

Let V be a finite set, $L \geq 1$ and let us take the search space $C = V^L$. In the genetic terminology a candidate c is called a *genotype*. When interested in the inside of candidates we shall denote them as

$c = (c.1, \ldots, c.L)$,

where $c.i \in V$ ($i \in \{1, \ldots, L\}$) are the genes of c.

Definition 5.1.1

Let $c \in C$ be a genotype. A *marker* is a number $k \in \{1, \ldots, L\}$, the *gene marked* by k in c is $c.k$.

□

To make a child of n parents $c_1 = (c_1.1, \ldots, c_1.L), \ldots, c_n = (c_n.1, \ldots, c_n.L)$ we scan their genes. More precisely, we mark one gene of them each and make the child gene by gene choosing from the marked genes. The hint and the first examples of such production functions is due to Nuijten (1990).

```
PROCEDURE scan
BEGIN
  initialize markers k_1, . . ., k_n
  j := 1
  WHILE j ≤ L DO
    BEGIN
      pick one gene c_i.k_i of the marked genes c_1.k_1, . . ., c_n.k_n of the parents and
      let c_i.k_i be the j-th gene of the child
      update the markers
      j := j + 1
    END
END
```

Notice that *scan* can be seen as a highly parameterized procedure; the main outlines are set but *initialize, pick* and *update* need to be given to obtain different recombination methods. Many known genetic operators can be considered as a special form of scanning with $n = 2$, distinguished by different *initialize, pick* and *update* mechanisms.

Example 5.1.2 (1-point crossover)
The classical 1-point crossover operation (described in section 4.3.1) can be obtained by the following. Let $n = 2$, *initialize* the markers as $k_1 = 1, \ldots, k_n = 1$ and let us choose the j-th gene of the child by

$$pick(\{c_1.k_1, c_2.k_2\}) = \begin{cases} c_1.k_1 & \text{if } 1 \leq j \leq l \\ c_2.k_2 & \text{if } l < j \leq N \end{cases}$$

where $l \in \{1, \ldots, N\}$ is drawn randomly.
Furthermore let us apply a simple *update* shifting the markers to the right by one position in each cycle.
□

Observe that in the 1-point crossover *initialize*, *pick* and *update* are problem independent. There are more sophisticated crossovers known that use domain knowledge when picking the actual gene of the child and also their updating mechanism is tailored.

Example 5.1.3 (Heuristic crossover)

This method, cf. Grefenstette et al. (1985), is elaborated for the Travelling Salesman Problem. The set V is the set of cities and a tour is coded by adjacency representation as a permutation of cities. The method makes use of a function $D : V \times V \rightarrow \mathbb{R}^+$ that represents the distance between the cities.

We can describe heuristic crossover as scanning to produce a child c by:

- $n = 2$;
- taking a random city $c.1 \in V$ as the staring point of the child's tour;
- *initializing* the markers at those genes (cities) that follow $c.1$ in the parents;
- *picking* that city for $c.j$ $(j > 1)$ that provides the shorter edge leaving $c.(j\text{-}1)$ or if the shorter edge would introduce a cycle in the child the picking randomly a one that does not introduce a cycle;
- *update* the markers such that the marked cities follow the last picked city in the given parent.

□

The name of the above method shows how the authors envisage it. Sensing that the presence of a problem dependent factor (using the distances) is characteristic they named their crossover heuristic. Liepins et al. (1987) also studied this crossover calling it greedy. This name shows what is important for them: not the fact that problem dependent domain knowledge is used but the way it is used.

In Mühlenbein (1989) children are produced by a so called *p-sexual voting recombination*. It is a real multiparent method for p parents, although the name *p-sexual* is not a very good one. Namely, the author does not distinguish different sexes among the individuals requiring that one parent of each sex is needed for mating. The other characteristic feature is *voting*. Interpreted in our terms it is a yet another heuristic where *pick* chooses the gene with the highest occurrence among the marked genes if only it occurs more times then a threshold.

In our tests (see later) we used a scanning procedure for PCSP, generating newly ordered sequences of jobs by similar *pick* heuristics.

Example 5.1.4

We use the *scan* procedure to L long sequences of jobs (symbols of an alphabet V). A child of n $(n \geq 2)$ parents is created by:

- *initializing* the markers as $k_1 = 1, \ldots, k_n = 1$;

- for the j-th gene of the child we *pick* that marked gene of the parents that belongs to the job with the longest processing time;
- *update* after creating the j-th gene of the child c happens by setting $k_i = \min\{ l \in \{j, \ldots, N\} \mid c_i.l \notin \{c.1, \ldots, c.j\} \}$ for each parent c_i $(i \in \{1, \ldots, n\})$.

□

Notice also that the application domain of the above *update* is not restricted to PCSP. It can be applied for any problem where the individuals are permutations of the elements of the alphabet V, for instance TSP. In such a case the child of permutations should be a permutation as well and this is exactly what this *update* is taking care of. Hereby we can solve a problem of permutation representation mentioned in Grefenstette et al. (1985), namely that of legal tour generation.

5.2 Multidimensional Genotypes

In this section we study extended genotypes. We observe that several authors, e.g. Mühlenbein (1989), Gerrits and Hogeweg (1991), Colorni, Dorigo and Maniezzo (1991) apply tables as genotypes instead of the usual gene sequences. Formalizing such extensions we introduce n-dimensional genotypes and in particular we investigate the case of 2-dimensions. Hereby we are aiming at making the possibility of using non sequential genotypes explicit.

Definition 5.2.1

Let V be a finite set, the alphabet. An *n-dimensional genotype* can be defined as an n-dimensional matrix over A, where the elements of the matrix have n indices. In particular, a *2-dimensional genotype* of (size $K \cdot L$) is a table

$$\begin{bmatrix} v_{11} & \cdot & \cdot & \cdot & v_{1L} \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ v_{K1} & \cdot & \cdot & \cdot & v_{KL} \end{bmatrix}$$

where $v_{ij} \in V$ for every $i \in \{1, \ldots, K\}$, $j \in \{1, \ldots, L\}$.

□

A bit more sophisticated form of 2-dimensional genotypes is obtained if we distinguish different types of genes and use the spatial relationships to structure them.

Definition 5.2.2

Let $V_1, \ldots, V_K$ be finite sets and let each V_i be interpreted as a type of gene. A *structured 2-dimensional genotype* (with size $K \cdot L$) is a table

$$\begin{bmatrix} v_{11} & \cdot & \cdot & \cdot & v_{1L} \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ v_{K1} & \cdot & \cdot & \cdot & v_{KL} \end{bmatrix}$$

where $v_{ij} \in V_i$ for every $i \in \{1, \ldots, K\}, j \in \{1, \ldots, L\}$.

□

Example 5.2.3

The case of job shop mentioned before belongs to this latter sort of 2-dimensional genotypes. It can be described by having V_1 the set of jobs, V_2 the set of machines and V_3 the set of time instances in consideration. In our tests we used this coding.

□

There are obviously numerous ways to produce children from tables as parents.

- Mühlenbeim (1989) uses a pointwise construction method;
- Colorni, Dorigo and Maniezzo (1991) use a crossover-like method making children from the rows of the parents;
- Gerrits and Hogeweg (1991) apply column exchanges in a problem specific way.

The latter two can be seen as 'Cartesian recombination' between parent tables or as a generalization of 1-dimensional crossover if we consider rows (columns) as meta genes. Next we elaborate a child production method combining Cartesian and genetic features. For the sake of convenience we consider the case of $K = 2$ and row exchanges as a basis.

Let us assume that we have two sets of genes V_1 and V_2 and the individuals are structured two dimensional genotypes of the form

$$\begin{bmatrix} v_{11} & \ldots & v_{1L} \\ v_{21} & \ldots & v_{2L} \end{bmatrix}$$

where $[\,v_{11} \ldots v_{1L}\,] \in V_1^{\,L}$ and $[\,v_{21} \ldots v_{2L}\,] \in V_2^{\,L}$.

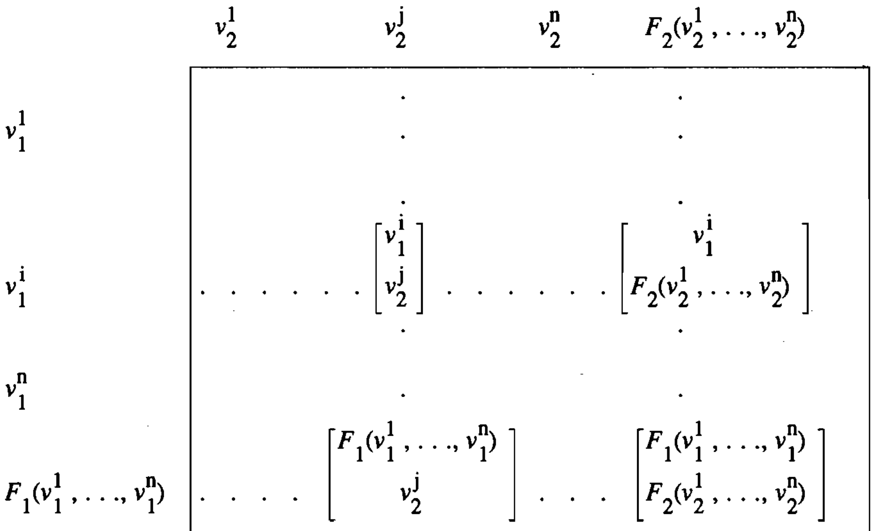
Furthermore, let us assume that we have two one dimensional multiparent production functions F_1 on $V_1^{\,L}$, F_2 on $V_2^{\,L}$ each creating a child of n parents. Let

$$\begin{bmatrix} v_{11}^1 & \ldots & v_{1L}^1 \\ v_{21}^1 & \ldots & v_{2L}^1 \end{bmatrix}, \quad \ldots \quad, \begin{bmatrix} v_{11}^n & \ldots & v_{1L}^n \\ v_{21}^n & \ldots & v_{2L}^n \end{bmatrix}$$

be n individuals for the sake of convenience abbreviated as

$$\begin{bmatrix} v_1^1 \\ v_2^1 \end{bmatrix}, \quad \ldots \quad, \begin{bmatrix} v_1^n \\ v_2^n \end{bmatrix}$$

Below we display the possibilities of obtaining children by row exchanges, F_1 and F_2.



The element $\begin{bmatrix} v_1^i \\ v_2^j \end{bmatrix}$ denotes the child obtained by combining the first row of the i-th parent with the second row of the j-th parent. $F_1(v_1^1, \ldots, v_1^n)$ and $F_2(v_2^1, \ldots, v_2^n)$ denote the one dimensional child obtained by applying the one production functions F_1 and F_2 to the first, respectively second rows of the above tables.

Observe that the n^2 elements at the upper left side are purely 'Cartesian', the $(n+1)^2 - n^2$ elements of the last row and the last column are really genetic ones.

It is easy to see that in general $(n+1)^K$ children can be produced in this way, where n^K are purely Cartesian and $(n+1)^K - n^K$ are really genetic.

The above figure can help to make offspring production methods on ones own. Any set of the children displayed above (for instance the ones in the last row) can be taken as offspring of these parents. The possibilities are still manifold even for $K = 2$, e.g. column exchanges or exchanges of sub-rectangles.

5.3 Selection, Reduction and Evaluation

Besides the set of manipulations and a production function, in the default search procedure we have to define selection, reduction and evaluation functions as well. Below we discuss how these functions can be defined and we give a set of options that are typical in genetic algorithms but can be used in general.

The standard way in a GA is to select (reduce) based on the objective function value of the candidates. However, some natural principles with respect to selection (reduction) presume properties of the objective function that do not always hold. In such a case we have to make use of the objective function indirectly.

1) Let g be is a non negative transformed version of the original objective function f such that candidates with a larger g value are better, i.e. have a smaller f value. This function g is mostly called the *fitness function.* A frequently applied random selection (reduction) method is to select (reduce) randomly giving higher chance to fit candidates, e.g. by a distribution

assigning the probability

$$P(c,n) = \frac{g(c)}{\sum_{d \in X_n} g(d)}$$

to the candidate c in the n-th population X_n. Notice that such a selection (reduction) function is generous in the sense of Definitions 4.4.12 and 4.4.14 if g is positive.

2) Another possibility is a best first like selection (reduction), choosing only *elite candidates*, that is choosing such that for every $\beta \in B$, $\delta \in \Delta$, $x,y \in C^*$

   $\forall c \in \cup F_s(\beta,x) \; \forall d \in (x \setminus \cup F_s(\beta,x)) : g(c) \geq g(d)$,

   or

   $\forall c \in F_r(\delta,x,y) \; \forall d \in (x \cup y \setminus F_r(\delta,x,y)) : g(c) \geq g(d)$.

3) A third way can be to combine the first two ideas and if there must be n ($n > 1$) candidates chosen (for being a parent or to survive) then choosing $k < n$ elite ones and $n - k$ at random.

4) A special way of reduction is to let every newly produced child to survive, i.e. having a reduction function satisfying

   $x \subseteq F_r(\delta,x,y)$

   for every $\delta \in \Delta$, $x,y \in C^*$.

5) In case we have a source element σ in the search space (see Definition 4.4.17), then a σ-preserving reduction function may be needed. In this case the source element σ always 'survives' together with some other candidates that can be chosen according to the above principles.

The evaluation function of a search procedure plays the role of the termination condition. The natural choice of setting F_e equivalent to the goal condition of the given search problem is not always applicable in practice. As we discussed it in section 3.1, this can occur if an optimization is to be solved. In such a case one often applies termination conditions such as for instance:

a) $F_e(X_n) = true \Leftrightarrow min \{ f(c) \mid c \in X_n \} \leq D$

where D is a given bound;

b) $F_e(X_n) = true \Leftrightarrow min \{ f(c) \mid c \in X_{n-1} \} - min \{ f(c) \mid c \in X_n \} \leq D$

where $D > 0$ is a border given in advance;

c) $F_e(X_n) = true \Leftrightarrow n \geq N$
with $N \in \mathbb{N}$ given in advance.

Notice that every option listed above for a selection, reduction and evaluation function is problem independent; they can be applied to optimization problems in general. Their use is not restricted to genetic algorithms, which makes it possible to include these sets of options in a generic DSS development tool. Hereby we can reduce the efforts of DSS development by allowing the DSS designer to chose one of these options when creating a DSS.

CHAPTER 6

Towards a Software Tool for DSS Design

In the previous chapters we developed a model of planning problems and a model of search as problem solving method. Here we sketch how a generic software tool for DSS design can be based on these models. Strictly speaking, we do not consider the design of a complete DSS, we restrict ourselves to the problem definition and problem solving components.

6.1 Problem Definition Component

The module of a DSS tool that facilitates problem definition must be able to receive and interpret information needed for the definition of a planning problem. According to the model elaborated in Chapter 2 this module is receiving as input:
- background data;
- the definition of possible actions and time instances;
- the definition of pre-states;
- the correctness condition defining states;
- allowability conditions and a goal condition;
- the effect description;
- the evaluation criterion;
- the definition of an initial state.

Let us note that for the sake of convenience we use a static planning terminology along this chapter, that is we always refer to a 'state' instead of a 'state or process', etc.

On the above basis we can expect the following functions from this module.
- Accepting a problem description, i.e. the definition of a planning problem.
- Facilitating the modification of the problem description by allowing modification of the data and the above given items (e.g. redefining allowability or the evaluation criterion).

The DSS created in this way should then be able to perform the following tasks.
- Representing and displaying plans.
- Supporting hand made planning by allowing the user to manipulate plans.
- Computing α' and e' automatically from the definitions of α and e.
- Answering queries like
    - is the pre-state s a state (i.e. does it satisfy the correctness condition)?
    - is the operation o (plan P) allowed at state s?
    - which state is obtained if we apply the operation o (plan P) to state s?
    - does the state s satisfy the goal condition?
    - is the plan P a solution of the planning problem?
    - what is the value of plan P according to the given criterion?

As the language of the problem definition module we propose the following.

There must be a data language to define relevant objects, permanent functions and permanent relations that will be used. For the TSP in section 2.2.1 the data should describe the set Z and the function D. Data modelling is an important part of software development but it would go beyond the purpose of this chapter to discuss it in details. Nevertheless, let us remark that relational algebra is advisable for data description since it provides a theoretically and practically proved approach for data modelling and it can be smoothly linked to a logic fashioned language introduced below.

Note that based on the given data and standard arithmetics, we have the following at our disposal:

- constants (objects from the data description and arithmetic constants e.g. the real numbers);
- function symbols denoting permanent functions and standard arithmetic functions;
- relation symbols denoting permanent relations and standard arithmetic relations.

These items determine a first order language such that the truth value of its formulae can be computed by the given data and an arithmetic computation unit.

Next, let us extend the this first order language by adding relation symbols denoting temporary relations and let us denote the resulting first order language by L. Hereby the pre-states are determined as sets of temporary atoms, that is atoms containing only relation symbols denoting temporary relations. Notice that the truth value of a formula of L that contains a temporary atom can be computed with respect to pre-states only. The definition of the truth value of a temporary atom $r(x)$ with respect to a pre-state s can be based on the identification of $r(x) \in s$ and $s \models r(x)$. Thereafter, the truth value of every atom (temporary and permanent) can be determined and the truth of an arbitrary formula of L with respect to a pre-state can be defined by standard formula induction.

A correctness condition that defines states as pre-states satisfying this condition can be given as a well formed formula (wff) in L. Remember that for TSP this formula was:

$\exists! \; x \in Z : at(x).$

Note that the goal condition is also a statement about states, therefore L is also appropriate to express it. In section 2.2.1 we used the formula

$at(z_1) \;\wedge\; \forall z \in Z : seen(z);$

Naturally, we also have to define the names of actions. If the set of time instances is also known - for instance $\mathbb{R}$ by default - then hereby the set of operations becomes defined and so does the set of all plans. To define the allowability of operations we need requirements about the state an operation is applied to. Also these conditions of allowability can be expressed in L, possibly by one condition for each different action. Since in a TSP we had only one action - $to(x,y)$ - we needed only one formula as a condition:

$at(x) \;\wedge\; \neg\, seen(y).$

Observe that states are formally sets of atoms hence their changes are easy to express in terms of $\cup$ and $\setminus$. This implies that we need to surpass the first order language L and introduce an effect language EL in a functional fashion based on set operations. Again, we can expect that to each action there belongs a correct expression of EL, for instance for $to(x,y)$ we gave

$(s \setminus \{at(x)\}) \cup \{at(y), seen(y)\}$

in section 2.2.1.

Finally, evaluation criteria can be defined as arithmetic expressions possibly relying on the given data. The criterion in case of the TSP was

$$\kappa(\{ (to(x_1,y_1),t_1), \ldots, (to(x_m,y_m),t_m) \}) = \sum_{i=1}^{m} D(x_i,y_i).$$

The definition of an initial state requires that we explicitly give a set of atoms of L that satisfies the correctness condition. For the TSP the set

$\{at(z_1)\}$

was given as initial state.

Notice that such a logic fashioned language for problem specification is human friendly in the sense that non-experts without much experience are likely to read and write sentences of such a language. This feature makes it possible that the access to these items be left free after the DSS design phase, that is that even the user of the DSS is allowed to modify these parameters. Observe that hereby the flexibility of the DSS tool can be carried over to the DSS itself.

Next to the definition of a planning problem the DSS tool must also support the definition of a search problem. The search problem $(C, \varphi_f, \varphi_g)$ should be defined such that it fits the given planning problem. If there is no hard argument against it, then we suggest that the default free search space is used together with the default representation and interpretation function (see section 3.2). Note that these can be created automatically from description of the planning problem.

The functions of the submodule facilitating search problem definition can then be listed as follows.

- Taking the default search problem belonging to the given planning problem.

- Adding extra conditions on plans restricting the default feasibility and goal conditions.
- Defining non-default feasibility and goal conditions to a given planning problem.
- Supporting the definition of feasibility and goal conditions on plans without having defined the full planning problem (recall the remarks after definition 3.2.4).

Notice that the necessary data and the set of plans can be defined without formally defining the world states, allowability and the effect function. If we have

- constants (objects from the data description and arithmetic constants e.g. the real numbers);
- function symbols denoting permanent functions and standard arithmetic functions;
- relation symbols denoting permanent relations and standard arithmetic relations

together with the set of actions and time instances, then the set of all plans is determined and so is a first order language L' $\subseteq$ L wherein we can express feasibility and goal conditions for a formal search problem. This permits that we omit the analysis and the specification of states and effects immediately defining plans as candidates and φ_f and φ_g by wff's in L'. Let us remark that in the examples of Chapter 3 the given φ_f and φ_g were expressed in such a manner. For the TSP in section 3.3.1 we had the default form of candidates, that is

$$\begin{bmatrix} u_1 & . & . & . & u_k \\ v_1 & . & . & . & v_k \\ t_1 & . & . & . & t_k \end{bmatrix}$$

where a column

$$\begin{bmatrix} u_i \\ v_i \\ t_i \end{bmatrix}$$

belonged to an operation $(to(u_i,v_i),t_i)$. The feasibility condition for a candidate in the above form was given as

$$\forall\, i \in \{1,\ldots,k\text{-}1\} : t_i \neq t_{i+1} \;\wedge\; \forall\, i \in \{1,\ldots,k\text{-}1\} : v_i = u_{i+1} \;\wedge\; u_1 = z_1$$

The evaluation criterion κ for a candidate in the above form was defined as

$$\sum_{i=1}^{k} D(u_i,v_i),$$

and the goal condition was given as

$\forall i \in \{1, \ldots, k\text{-}1\} : t_i \neq t_{i+1} \ \wedge \ \forall i \in \{1, \ldots, k\text{-}1\} : v_i = u_{i+1} \ \wedge \ u_1 = z_1 \ \wedge \ k = n$
$\wedge \ v_n = z_1$.

Recall that an appropriate search problem is not only depending on the given planning problem but also on the intended solution method. With respect to a solution method we have basically two choices: either an construction or an iteration method can be applied. Choosing between the two the following arguments can be considered:
- for highly constrained problems it can be very difficult to produce feasible offspring of candidates, thus a (stepwise) construction method can be easier than iteration in the space of feasible candidates;
- if there are evaluation criteria involved then we have an optimization problem, in which case iteration, in particular improvement, is the commonly made choice;
- construction may be applied even for optimization; in section 4.2 we presented a general construction method in an iterative fashion, based on the use of heuristics (dispatch rules) to extend the empty plan step by step towards a good plan.

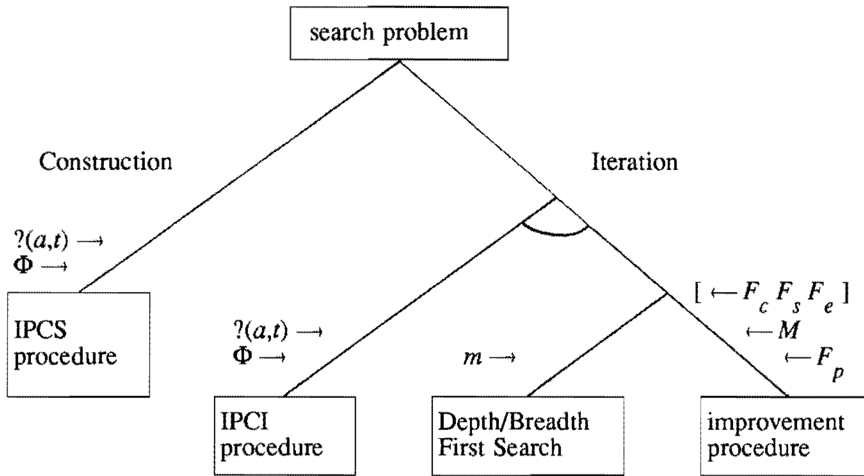
Notice that if the planning problem is defined, then correct feasibility and goal conditions for an appropriate search problem can be derived automatically.

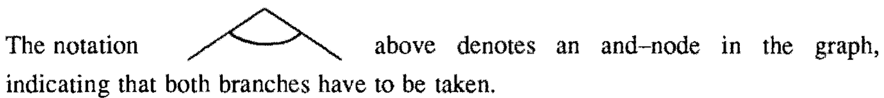
6.2 Solution Method Definition Component

The module of a DSS tool that facilitates the definition of a problem solving method must be able to receive and interpret information needed to define a search procedure. The language to define a search procedure should be appropriate to define manipulations on plans, initialization and the selection, production, reduction and evaluation functions. Two kinds of choices for such a language are:
- an executable specification language, e.g. ExSpect, cf. van Hee, Somers and Voorhoeve (1989);
- an imperative programming language (Pascal, C).

Notice that these options for the specification language of the solution method are lacking the human friendliness of logic fashioned languages. This implies that the user of the DSS would have only limited access to the inside of problem solving mechanism. Nevertheless, we can assemble a library of (parameterized) components of search procedures (e.g. several kinds of selection functions) and support the composition of a search procedure from this library. This library can be used within the DSS tool - thus in the DSS development phase - as well as within the DSS. By allowing the user to (re)compose procedures and by letting him tune the parameters of the components it is possible to provide flexibility for the problem solving section too. This seems to be a promising approach as far as *selection*, *reduction* and *evaluation* are concerned. Other items, however, such as the set of *manipulations* and offspring *production* seem to be too problem dependent to be defined in a generally usable manner. Nevertheless, if we restrict ourselves to the generalized genetic framework discussed in Chapter 5, then we can provide guidelines even for defining manipulations and offspring production. We return to this question later, after discussing other aspects of defining a search procedure.

If we already have a search problem that fits the original planning problem then a search procedure has to be defined. Since the search space is already determined, c_{init}, M, F_s, F_p, F_r and F_e need to be defined, that is the initial candidate(s), the set of manipulations, the selection function, the production function, the reduction function and the evaluation function. By the following figure we give a global illustration of how a search procedure can be defined. By $\leftarrow X$ and $X \rightarrow$ we mean that the item X has to be given by the DSS designer. The interpretation of $[\leftarrow X]$ is that the item X does not necessarily have to be defined, it can be chosen from a set of provided options. The textual explanation is given below.



The notation  above denotes an and-node in the graph, indicating that both branches have to be taken.

If one wants to find a solution by construction then by the Iterative Plan Construction for Solution procedure (Definition 4.2.14) we can offer a reasonable support. Namely in the IPCS procedure $\bar{M}$, $\bar{P}_{init}$, $\bar{F}_s$, $\bar{F}_r$ and $\bar{F}_e$ are already defined. The system designers work is thus reduced to complete the definition of the production function by

- determining the condition Φ that tells whether to extend or shrink the actual plan;
- finding good heuristics (dispatch rules) to choose the operation (a,t) that is added to / subtracted from the actual plan.

If one is willing to apply an iteration (improvement) procedure then an initialization and an iteration part have to be made.

We can offer support for initialization by the Iterative Plan Construction for Initialization procedure as given in Definition 4.2.13. Here again, most of the components are already defined and only the definition of the production function has to be completed by

- determining the condition Φ that tells whether to extend or shrink the actual plan;
- finding good heuristics to choose the operation (a,t) that is added to / subtracted from the actual plan.

If one chooses a non improvement iteration method then we can offer two generic procedures in this spirit: depth first search and breadth first search, defined in section 4.3.4. and 4.3.5. As we remarked it there, these procedures can be hard coded in advance, only leaving two parameters free: the applied manipulation $m : D \nrightarrow D^*$ and an initial candidate.

For having an improvement procedure one has to define each of M, F_s, F_p, F_r, F_e. We pay special attention to this case and discuss it below.

Defining manipulations is a highly problem dependent step where we can offer little support in general. Nonetheless, if one agrees to work within the generalized genetic framework discussed in Chapter 5, then he can rely on the procedure *scan*. Taking it as the general way to describe manipulations the design is better shaped: *init, pick* and *update* have to be given. Since within this framework one can define many different kinds of offspring production methods, we advise to use it, unless the manipulations the designer has in mind do not fit this form.

Designing a selection function to choose the parents from the actual population can be reasonably supported in general. In section 5.3 we listed some generally applicable options, for example:
- select parents fully at random;
- select only elite parents, that is candidates that are better then other candidates according to some criterion;
- select a number of elite parents and some other ones randomly.

Next to such problem independent possibilities one can apply heuristics, that is a problem dependent manner of selecting, relying on the given domain knowledge.

The production function generates the children of the chosen parents. If for every parent-list there is only one manipulation applicable then the production mechanism is determined by M and F_s. If, however, there are more possibilities,

e.g. different mutations possible then the role of F_p is important. Standard ways to handle such cases are choosing yet other manipulations in turn, or defining F_p such that it chooses between the given possibilities randomly according to a given distribution.

Similarly to the selection function we can reasonably support the definition of F_r by options. The basic principles to choose the survivors are very similar to those of selecting the parents as the items (1) - (5) in section 5.3 indicate.

Recall the remark after Definition 3.1.11 about the difficulties to verify optimality. Therefore we present three practically applicable evaluation functions:

- giving a bound $B > 0$ and defining the value of $F_e(x)$ *true* if $f(c) \leq B$ holds for a candidate $c \in x$;
- stopping if the improvement by the last iteration step remains under a certain level;
- stopping if the number of iteration cycli reaches a limit.

There is a conclusion we can draw from the foregoing: the crucial factors in designing a search procedure are the manipulations and the production function. These should be suited to the problem and at the moment we do not see many possibilities to provide automatic support for their definition. Let us point out another step that seems to be crucial: initialization. Although we could present a general construction-initialization procedure, we foresee that it can be advantageous to make the initialization step by a more problem suited algorithm.

CHAPTER 7

Final Remarks

The field of decision support systems is "lacking conceptual research-oriented articles", as reported by Elam, Huber and Hurt (1986). In practice, the development of a DSS is mostly a case-bounded activity, repeated for every new decision problem. In this thesis we describe a mainly theoretical investigation directed at setting the outlines of a generic DSS development tool. Concentrating on the automatic decision generation function within a DSS we distinguish two main issues of interest: problem description and problem solving. We study them both independently and elaborate a theoretical model of planning problems and search procedures. The underlying idea of our approach is to have these models implemented by software that facilitates the definition of instances of the models. By building decision support systems with the aid of such a tool, DSS design could be carried out more systematically and with less effort than by the case specific practice of today.

Based on the notions of world states, actions and time we work out a planning theory. Time is explicitly present in our model of planning problems, which makes it possible to notice and handle difficulties of parallel actions. Discovering the limits of modelling only static world situations we introduce dynamic planning models and clarify their relationship to static ones. Finally, observing conflicts between intuition and the formal model, we discover the importance of the so

called Determinative Past Assumption in dynamic planning situations. By the theory we obtain a clear terminology and a general framework facilitating the definition of specific planning problems. Hereby we also lay the basis of the problem definition component of a generic DSS tool.

As for the problem solving part, we model stochastic space search procedures. The elaborated model incorporates features of graph search and local search methods, enlightening their relationships. By the General Search Procedure we distinguish the most essential components of a wide scale of search methods, ranging from depth first search to genetic algorithms. Modelling successive iteration cycli of the search by Markov chains, we prove general convergence results for methods applied to optimization problems, in particular simulated annealing. Special attention is paid to genetic algorithms. By generalizing classical genetic features we obtain a class of search procedures that can serve as a default problem solving method in DSSs.

To gain early feedback whether our problem solving approach is practicable we made a shell prototype based on the General Search Procedure. This prototype was implemented in C++ and it supported the definition of genetic-like search procedures for different optimization problems, cf. Nuijten (1990). With the aid of the shell we could define a search procedure and obtain an executable program to run evolutions. Plans were represented in a table (list) form as discussed in Chapter 3. By the absence of the problem definition component, feasibility and goal conditions had to be defined in the spirit of section 3.3. A set of options for each of the selection, reduction and the evaluation function was implemented in a problem independent manner as discussed in the foregoing. Support at the definition of a search procedure was thus partly realized by providing the possibility of choosing among these options. Hence, using this shell it was enough to concentrate on the design of the predicted problem dependent components such as initialization, the manipulations and offspring production. For the latter two we applied the generalized genetic framework using *scanning* with the heuristics described in Example 5.1.3 and Example 5.1.4. We have defined mutation and multiparent production as manipulations, the production function applied them randomly with a certain given distribution.

We experienced that it was simple to define search procedures by using our shell. More precisely, we found that it was quite easy to compose a genetic-like search procedure by the given options - if only we have coded manipulations and initialization already. Although the shell is just a prototype we gained feedback about the practicability of our approach to DSS design and confirmed that one can develop the decision generating subsystem of a DSS based on our notions.

The usefulness of such a shell is of course also depending on the quality of the search procedures created with the aid of it. To test it we applied the shell for making improvement algorithms for handling TSP and PCSP. We made runs to test the efficiency and the effectivity of the procedures. For the TSP we ran 500 tests with a 120 city problem from Grötschel (1977) and obtained an average tour length of 7952.7 (14.6% above the optimum). For the PCSP we took the FIS2 instance with 10 machines and 100 jobs from Fischer (1963). After 500 runs the average length of the obtained schedules was 1037.6 (11.6% above the optimum). The efficiency of our procedures was moderate: the computation times were between 10-15 minutes on a Sun SPARC station for each problem.

From a practical point of view, we can say that the aim of a decision generation procedure is not to find a theoretically optimal decision, but to find better decisions than a man would do. To make a rough comparison with human planners, the same problem instances were also given to ten colleagues. We observed that for the TSP they slightly outperformed our procedure by achieving an average tour length of 7738.3 (11.5% above optimum) within about 10-15 minutes, while the figures for the PCSP are 1080.8 (16.2% above the optimum) obtained after 1-3 hours of thinking.

These results have an illustrative value demonstrating that taking search procedures as the basis of problem solving in a DSS is a sound approach. Whether or not this approach is really practicable will, however, only be certain if more realistic (harder) problems can also be handled within acceptable computation times.

Future work has to be directed at mainly practical issues. The implementation of a problem definition component has to be carried out based on the language we sketched in section 6.1. Here we will have to handle questions about data management and interfacing. The prototype of the component supporting the definition of a problem solving method has to be extended such that a good

balance is reached between being free and having restrictions by built in features when composing a search procedure. A library of components of search procedures has to be made and made accessible to the DSS designer as well as the user of the created DSS. Here the issue of man-machine interaction during problem solving has to be treated. At last, the border between the DSS tool and DSSs has to be defined. This means that for both the problem definition and the solution method definition component it needs to be decided which parameters are fixed by the DSS designer and which ones can be modified later by the user of the created DSS.

References

Aarts, E.H.L. and Korst, J., *Simulated Annealing and Boltzmann Machines*, Wiley and Sons, 1989.

Addis, T.R., *Designing knowledge-based systems*, Kogan Page, 1986.

Ahlswede, R. and Wegener, I., *Search Problems*, Wiley and Sons, 1987.

Aigner, M., *Combinatorial Search*, Wiley and Sons, 1988.

Alter, S.L., *Decision support systems: current practice and continuing challenges.* Addison-Wesley, 1980.

Anthonisse, J.M., Lenstra, J.K. and Savelsbergh, M.W.P., Behind the screen: DSS from an OR point of view, *Decision Support Systems* 4, pp. 413-419, 1988.

Bellmore, M., and Nemhauser, G.L., The Traveling Salesman Problem: A Survey, *Operations Research* 17, pp. 538-558, 1968.

Bonczek, R.H., Holsappe, C.W. and Whinston, A.B., *Foundations of Decision Support Systems*, Academic Press, New York, 1981.

Bonczek, R.H., Holsappe, C.W. and Whinston, A.B., Specification of modelling and knowledge in decision support systems, in *Processes and Tools for Decision Support*, ed. Sol, H.G., North-Holland, 1983.

Brachman, R.J., Levesque, H.J. and Reiter, R. (eds.), *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, Morgan Kaufmann, 1989.

Burch, J.C. and Strater, F.R., *Information Systems: Theory and Practice*, Hamilton, 1974.

Charniak, E. and McDermott, D., *Introduction to Artificial Intelligence*, Addison-Wesley, 1985.

Colorni, A., Dorigo, M. and Maniezzo, V., Applying Evolutionary Algorithms to Solve the Time-table Problem, in *Parallel Problem Solving from Nature*, Lecture Notes in Computer Science, vol. 486, eds. Schwefel, H.-P. and Maenner, R., Springer-Verlag, 1991.

Davis, R. and Lenat, D.B., *Knowledge-based Expert Systems in Artificial Intelligence*, McGraw Hill, 1982.

De Jong, K., Adaptive system design: a genetic approach, *IEEE Transactions on Systems, Man and Cybernetics* 10, pp. 566-574, 1980.

De Jong, K., Genetic Algorithms: A 10 Year Perspective, in *Proceedings of the International Conference on Genetic Algorithms*, ed. Grefenstette, J., Lawrence Erlbaum Associates, 1985.

De Jong, K., On Using GAs to Search Problem Spaces, *Proceedings of the 2nd International Conference on Genetic Algorithms*, ed. Grefenstette, J., Lawrence Erlbaum Associates, 1987.

Dueck, G. and Scheuer, T., Threshold Accepting: A general Purpose Optimization Algorithm Superior to Simulated Annealing, manuscript 1988.

Eiben, A.E., Modeling Planning Problems, in *Proceedings of MFDBS 89*, Lecture Notes in Computer Science, vol. 364, eds. Demetrovics, J and Thalheim, B., Springer-Verlag, 1989.

Eiben, A.E., Aarts, E.H.L. and van Hee, K.M., Global Convergence of Genetic Algorithms: A Markov Chain Analysis, in *Parallel Problem Solving from Nature*, Lecture Notes in Computer Science, vol. 486, eds. Schwefel, H.-P. and Maenner, R., Springer-Verlag, 1991.

Eiben, A.E. and van Hee, K.M., Knowledge Representation and Search Methods for Decision Support Systems, in *Data, Expert Knowledge and Decisions*, eds. Gaul, W. and Schader, M., Springer Verlag, 1990.

Eiben, A.E. and Schuwer, R.V., Knowledge-based Systems: a Formal Model, *Proceedings of the Third Dutch Conference on Artificial Intelligence*, 1990. (in Dutch)

Elam, J.J., Huber, G.P. and Hurt, M.E., An examination of the DSS literature (1975-1985), in *Decision Support Systems: A Decade in Perspective*, eds. McLean, E.R. and Sol, H.G., North-Holland, 1986.

Even, S., Itai, A. and Shamir, A., On the Complexity of Timetable and Multicommodity Flow Problems, *SIAM Journal on Computing* 5, pp. 691-703, 1976.

Fisher, H. and Thompson, G.L., Probabilistic Learning Combinations of Local Job-shop Scheduling Rules, in *Industrial Scheduling*, eds. Muth, J.F. and Thompson, G.L., Prentice Hall, 1963.

Forbus, K.D., Qualitative Process Theory, *Artificial Intelligence* 24, pp. 85-168, 1984.

Garey, R.M. and Johnson, D.S., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman and Co., 1979.

Genesereth, M.R. and Nilsson, N.J., *Logical Foundations of Artificial Intelligence*, Morgan Kaufmann, 1987.

Gerrits, M. and Hogeweg, P., A Genetic Algorithm application on the search for minimal mutation phylectic trees, an NP-complete problem, in *Parallel Problem Solving from Nature*, Lecture Notes in Computer Science, vol. 486, eds. Schwefel, H.-P. and Maenner, R., Springer-Verlag, 1991.

Glover, F. and Greenberg, H.J., New approaches for heuristic search: A bilateral linkage with artificial intelligence, *European Journal of Operational Research* 39, pp. 119-130, 1989.

Goldberg, D.E., *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, Reading MA, 1989.

Gorry, G.A. and Scott Morton, M.S., A framework for management information systems, *Sloan Management Review* 13, pp. 55-70, 1971.

Green, C., Application of Theorem Proving to Problem Solving, *Proceedings of the First International Joint Conference on Artificial Intelligence*, North-Holland, 1969.

Grefenstette, J.J. (ed.), *Proceedings of the International Conference on Genetic Algorithms*, Lawrence Erlbaum Associates, 1985.

Grefenstette, J.J. (ed.), *Proceedings of the 2nd International Conference on Genetic Algorithms*, Lawrence Erlbaum Associates, 1987.

Grefenstette, J., Gopal, R., Rosmaita, B. and Van Gucht, D., Genetic Algorithms for the Travelling Salesman Problem, *Proceedings of the 2nd International Conference on Genetic Algorithms*, ed. Grefenstette, J., Lawrence Erlbaum Associates, 1985.

Grötschel, M., *Polyedrische Characterisierungen Kombinatorischer Optimierungsprobleme*, PhD. Thesis, Hain, Meisenheim am Glan, 1977. (in German)

Hansen, P., A short discussion of the OR crisis, *European Journal of Operational Research* 38, pp. 277-281, 1989.

van Hee, K.M., Information systems and decision support, *Informatie* 27, pp. 978-986, 1985. (in Dutch)

van Hee, K.M., Decision support systems for logistics, in *Databases*, ed. J. Paredaens, Academic Press, London, 1987.

van Hee, K.M. and Lapinski, A., OR and AI Approaches to Decision Support Systems, *Decision Support Systems* 4, pp. 447-459, 1988.

van Hee, K.M., Somers, L.J. and Voorhoeve, M., Executable Specifications for Distributed Systems, in *Information System Concepts: An In-depth Analysis*, eds. Falkenberg, E.D. and Lindgreen, P., North-Holland, 1989.

Holland, J.H., *Adaptation in Natural and Artificial Systems*, Univ. of Michigan Press, Ann Arbor, 1975.

Kanal, L. and Kumar, V. (eds.), *Search in Artificial Intelligence*, Springer-Verlag, 1988.

Kanal, L. and Kumar, V., The CDP: A Unifying Formulation for Heuristic Search, Dynamic Programming, and Branch-and-Bound, in *Search in Artificial Intelligence*, eds. Kanal, L. and Kumar, V., Springer-Verlag, 1988.

Keen, P.G.W., Adaptive Design for Decision Support Systems, *Database* 12, 1980.

Keen, P.G.W., Decision Support Systems: The Next Decade, in *Decision Support Systems: A Decade in Perspective*, eds. McLean, E.R. and Sol, H.G., North-Holland, 1986.

Keen, P.G.W. and Scott Morton, M.S., *Decision Support Systems: An Organizational Perspective*, Addison-Wesley, 1978.

Kolen, A.W.J. and Lenstra, J.K., *Combinatorics in operations research*, Report BS-R9024, Centre for Mathematics and Computer Science, Amsterdam, 1990.

Kowalski, R., Predicate Logic as Programming Language, *Proceedings of the IFIP Congress*, North-Holland, 1974.

van Laarhoven, P.J.M., *Theoretical and computational aspects of simulated annealing*, CWI Tracts, Centre for Mathematics and Computer Science, Amsterdam, 1988.

van Laarhoven, P.J.M., Aarts, E.H.L. and Lenstra, J.K., *Job shop scheduling by simulated annealing*, Report OS-R8809, Centre for Mathematics and Computer Science, Amsterdam, 1988.

van Langen, P. and Treur, J., *Representing World Situations and Information States by Many-sorted Partial Models*, Report PE8904, University of Amsterdam, 1989.

Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G. and Shmoys, D.B., *Sequencing and Scheduling: Algorithms and Complexity*, Designing Decision Support Systems Notes, Eindhoven University of Technology, 1989.

Liepins, G.E., Hilliard, M.R., Palmer, M. and Morrow, M., Greedy genetics, *Proceedings of the 2nd International Conference on Genetic Algorithms*, ed. Grefenstette, J., Lawrence Erlbaum, 1987.

Lin, S. Computer Solutions of the Traveling Saleman Problem, *The Bell System Technical Journal* 44, pp. 2245-2269, 1965.

Lin, S. and Kernighan, B.W., An Effective Heuristic Algorithm for the Travelling-Salesman Problem, *Operations Research* 21, pp. 498-518, 1973.

Lloyd., J.W., *Foundations of Logic Programming*, Second Edition, Springer-Verlag, 1987.

Minker, J., *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufman, 1988.

Minoux, M., *Mathematical Programming Theory and Algorithms*, Wiley and Sons, 1986.

Mitra, G. (ed.), *Mathematical Models for Decision Support*, NATO ASI Series, Computer and Systems Sciences, vol. 48, Springer-Verlag, 1988.

Mühlenbein, H., Parallel Genetic Algorithms, Population Genetics and Combinatorial Optimization, *Proceedings of the 3rd International Conference on Genetic Algorithms*, ed. Schaffer, J.D., Morgan Kaufmann, 1989.

Naylor, T.H., Decision Support Systems or what happened to MIS?, *Interfaces* 12, pp. 92-94, 1982.

Nemhauser, G.L. and Wolsey, L.A., *Integer and Combinatorial Optimization*, Wiley and Sons, 1988.

Nilsson, N.J., *Principles of Artificial Intelligence*, Springer-Verlag, 1982.

Nuijten, W.P.M., *Genetic Algorithms and Job Shop Scheduling*, Masters Thesis, Eindhoven University of Technology, 1990. (in Dutch)

Papadimitriou, C.H. and Steiglitz, K., *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, N.J., 1982.

Pearl, J., *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison- Wesley, 1984.

Pearl, J. (ed.), *Search and Heuristics*, North-Holland, 1983.

Pednault, E.P.D., Formulating multiagent, dynamic-world problems in the classical planning framework, in *Proceedings of the 1986 Workshop on Reasoning about Actions and Plans*, eds. Georgeff, M.P. and Lansky, A.L., Morgan Kaufmann, 1987.

Savelsbergh, M.W.P., *Computer Aided Routing*, Ph.D. Thesis, Centre for Mathematics and Computer Science, Amsterdam, 1988.

Savory, S.E. (ed.), *Artificial Intelligence and Expert Systems*, Chichester: Horwood, 1988.

Schaffer, J.D. (ed.), *Proceedings of the 3rd International Conference on Genetic Algorithms*, Morgan Kaufmann, 1989.

Shapiro, S.C. and Eckroth, D. (eds.), *Encyclopedia of Artificial Intelligence*, Wiley, 1987.

Simon, H.A., Search and Reasoning in Problem Solving, *Artificial Intelligence* 21, pp. 7-29, 1983.

Sol, H.G., DSS: Buzzword or OR challenge?, *European Journal of Operational Research* 22, pp. 1-8, 1985.

Sprague, R.H., A framework for research on decision support systems, in *Decision Support Systems: Issues and Challenges*, ed. Fich, G. and Sprague, R.H., Pergamon Press, 1980.

Sprague, R.H., DSS in context, *Decision Support Systems* 3, pp. 197-202, 1987.

Sprague, R.H. and Carlson, E.D, *Building Effective Decision Support Systems*, Prentice-Hall, 1982.

Sterling, L and Shapiro, E., *The Art of Prolog*, MIT Press, 1986.

Suh, J.Y. and van Gucht, D., Incorporating heuristic information into genetic search, *Proceedings of the 2nd International Conference on Genetic Algorithms*, ed. Grefenstette, J., Lawrence Erlbaum Associate, 1987.

Treur, J., *Reasoning about partial models, actions and plans*, Report P8813, University of Amsterdam, 1988.

Turban, E.R. and Watkins, P.R., *Applied Expert Systems*, North Holland, 1988.

Verbeek, P.J., *Learning About DSS: Two Case Studies on Manpower Planning in an Airline*, PhD. Thesis, Rotterdam University, 1990.

Waterman, D.A., *A Giude to Expert Systems*, Addison-Wesley, 1986.

Winston, P.H., *Artificial Intelligence*, Second Edition, Addison-Wesley, 1984.

Samenvatting

In dit proefschrift wordt een onderzoek beschreven dat gericht is op het verkrijgen van generieke softwaregereedschappen die het ontwikkelen van decision support systems (DSS) voor operationele planningsproblemen uit de praktijk makkelijker, sneller en dus goedkoper maken dan de hedendaagse technieken. Wij beperken ons tot het (semi-) automatisch genereren van beslissingen, zodat man-machine interactie, user interfaces, data- en modelmanagement buiten beschouwing worden gelaten. Een theoretisch onderzoek wordt uitgevoerd dat zich richt op het formeel modelleren van planningsproblemen en oplosmethoden.

Eerst wordt in Hoofdstuk 2 een theoretisch model van planningsproblemen uitgewerkt en worden theoretische aspecten van zulke problemen besproken. Daarna worden vijf planningsproblemen gespecificeerd met behulp van de aangeboden theorie. Hierdoor ontstaan richtlijnen voor een wijze waarop een formele beschrijving van een planningsprobleem gegeven kan worden.

Zoekend naar een passende algemene oplosmethode bestuderen wij 'zoeken', 'logische redenering' en 'mathematisch programmeren', waarna het paradigma 'zoeken' gekozen wordt. Wij geven een model van zoekproblemen en onderzoeken de relatie tussen planningsproblemen en zoekproblemen. In Hoofdstuk 4 wordt op zoekmethoden ingegeaan. Hierbij introduceren wij een Algemene ZoekMethode (AZM) en beschrijven enkele bekende typen van algoritmen als specialisaties van de AZM. Vervolgens worden convergentie-stellingen bewezen die aangeven aan welke eisen de componenten van een zoekmethode moeten voldoen om convergentie van het zoeken naar een oplossing te garanderen. Een veelbelovende klasse van zoekmethoden, genetische algoritmen (GAs), wordt in meer detail bestudeerd. Interessante eigenschappen van GAs zijn dat zij in een ruime probleemklasse redelijk presteren en gemakkelijk aangepast kunnen worden als het probleem - binnen die klasse - verandert.

Aan het einde van het proefschrift maken wij een stap in de richting van de volgende onderzoeksfase: het realiseren van generieke software die op basis van de voorafgaande theorie het ontwikkelen van een beslissingsondersteunend systeem vergemakkelijkt. Door de bevindingen van Hoofdstuk 2 beschikken wij

over een geparameteriseerd model dat een grote klasse van planningsproblemen omvat. Doordat het model hoog niveau parameters heeft (d.w.z. parameters die expressies van een taal met een grote expressieve kracht als waarde kunnen hebben) is het specificeren van concrete instantiaties relatief eenvoudig. Voor het model van oplosmethoden gaat deze laatste eigenschap in mindere mate op. Hoewel bij het definiëren van een zoekprocedure men gebruik kan maken van de AZM, kan het geven van een volledige definitie nogal veel werk vereisen. Er zijn echter componenten van de AZM waarvoor een op brede schaal bruikbare invulling kan worden gegeven. Als bovendien de meer beperkte klasse van genetische algoritmen wordt beschouwd, is het mogelijk om richtlijnen te geven voor de ontwikkeling van de overige componenten van een zoekprocedure.

Het door dit proefschrift beschreven onderzoek is theoretisch van aard; het legt de conceptuele basis voor een methode en generieke software voor DSS-ontwikkeling. De echte praktische bruikbaarheid van onze benadering kan niettemin alleen door nadere tests met een volledig uitgebouwd tool vastgesteld worden.

Curriculum Vitae

De schrijver van dit proefschrift werd op 14 juni 1961 geboren te Budapest, Hongarije. Hier behaalde hij in 1979 zijn diploma aan het Árpád Gymnasium. Na de militaire dienst startte hij met de studie wiskunde aan de Eötvös Loránd Universiteit te Budapest in 1980; in 1985 studeerde hij af op een onderzoek over logisch programmeren. Vanaf september 1985 was hij werkzaam bij de Expert Systems Department van Computing Applications and Co. te Budapest als knowledge engineer bij diagnostische systemen en deed als zodanig praktische en theoretische kennis op het gebied van Kunstmatige Intelligentie op. In februari 1987 kwam hij naar Nederland waar hij sinds maart 1987 als onderzoeker in opleiding bij de Sectie Informatiesystemen aan de Technische Universiteit Eindhoven werkt. Het hier verrichte onderzoek word begeleid door prof.dr. K.M. van Hee en maakt deel uit van een NFI project dat gefinancierd wordt door de Nederlandse Organisatie voor Wetenschappelijk Onderzoek. Sinds het aflopen van de door NWO gefinancierde periode is hij in dienst bij de Technische Universiteit Eindhoven.

Stellingen behorend bij het proefschrift

A METHOD FOR DESIGNING DECISION SUPPORT SYSTEMS FOR OPERATIONAL PLANNING

van A.E. Eiben

1. Theorem 1 in [1] stelt dat als een logisch programma P geen interne variabelen heeft en dichotoom is, er een complementair programma voor P bestaat. In deze stelling is echter de conditie van dichotomie overbodig.

   [1] Sato, T. and Tamaki, H., Transformational logic program synthesis, *Proceedings of the International Conference on Fifth Generation Computer Systems*, ed. by ICOT, North-Holland, 1984.

2. Het onderscheid tussen statische en dynamische planningssituaties is essentieel. Doordat de tijd in beide situaties een verschillende rol speelt, zijn de cruciale eigenschappen van die twee situaties onverenigbaar. (zie Hoofdstuk 2 van dit proefschrift)

3. De term graafalgoritme kan op twee verschillende manieren geïnterpreteerd worden. Omdat die twee interpretaties zelden onderscheiden worden, ontstaat er verwarring in het woordgebruik. (zie Hoofdstuk 3 van dit proefschrift)

4. De biologische analogieën die meegeholpen hebben met het funderen van genetische algoritmen, cf. [1,2], zijn belemmerend. Namelijk, door het niet beperken van genotypen tot eindige 0-1 rijen, het toelaten van meer dan twee ouders en het toepassen van niet crossover-achtige genetische recombinatie wordt een ruimere klasse van genetische algoritmen verkregen dan nu wordt gebruikt. (zie Hoofdstuk 5 van dit proefschrift)

   [1] Holland, J.H., *Adaptation in Natural and Artificial Systems*, Univ. of Michigan Press, 1975.
   [2] Goldberg, D.E., *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, 1989.

5. De relatieve inefficiëntie van een algemeen toepasbaar algoritme ten opzichte van een toegesneden algoritme wordt onbelangrijk als de betreffende executietijden onder een bepaalde grens vallen. Daar veel praktische problemen toch van een beperkte omvang zijn en de gebruikte hardware steeds krachtiger wordt, geeft deze observatie bestaansrecht aan algemeen toepasbare maar (nog) trage oplossingstechnieken.

6. Omdat probleemspecificaties de brug vormen tussen reële problemen en de wereld van formele oplosmethoden, valt het formeel niet te bewijzen dat een specificatie correct is. Daarom is de intuïtieve verificatie van de correctheid van een specificatie van dermate groot belang dat specificatiemethoden gebaseerd op een taal waarvan het niveau voldoende hoog is, zonder twijfel zijn aan te bevelen.

7. In [1] definieert Mars kennissystemen als systemen waar "zo goed mogelijk een scheiding is aangebracht tussen toepassingsgebied-onafhankelijke afleidingsregels en toepassingsgebied-specifieke kennis". De aanwezigheid van zo'n scheiding heeft echter zulke grote voordelen, dat het niet beperkt zou moeten blijven tot redeneersystemen binnen AI. Met name zijn systemen aan te bevelen waar zowel binnen de component voor probleembeschrijving als binnen de component voor probleemoplossing een scheiding tussen toepassingsgebied-onafhankelijke en toepassingsgebied-specifieke kennis verwezenlijkt is.

   [1] Mars, N., Onderzoek van niveau: Kennistechnologie in wording, *Informatie*, jrg. 30, nr. 2, pp. 84-90, 1988.

8. Het praktisch bruikbaar maken van logisch programmeren heeft rampzalige gevolgen voor de theorie daarvan. Met name het gebruik van dynamic clauses maakt dat de semantiek van een PROLOG-programma op losse schroeven komt te staan.

9. Een Nederlands gezegde luidt: "kleren maken de man". In het Hongaars wordt echter gezegd: "kleren maken de man niet". Dit laatste zal bij velen minder in de smaak vallen; het roept namelijk de vraag op "wat maakt de man dan wel?".

10. Het streven van Hongarije om zich bij Europa aan te sluiten kan pas serieus genomen worden als het woord cosmopoliet daar niet langer als politiek scheldwoord wordt gebruikt.

11. Het gerecht waarvan in [1] de bereidingswijze wordt gegeven, is ook heel lekker als men truffels door gerookte achterham vervangt.

   [1] Makowsky, J.A., Abstract Embedding Relations, in *Model-Theoretical Logics*, eds. Barwise, J. and Feferman, S., Springer-Verlag, 1985.