STUDIES IN THE USE AND GENERATION OF HEURISTICS

Rina Dechter

September 1985
CSD-850033

UNIVERSITY OF CALIFORNIA

Los Angeles

Studies in the Use and Generation of Heuristics

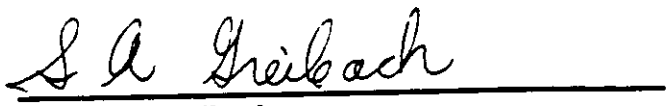
A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in Computer Science

by

Rina Dechter

1985

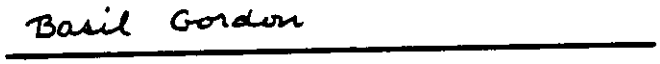
The dissertation of Rina Dechter is approved.



Sheila A. Greibach

Stott D. Parker

James R. Jackson



Basil Gordon

Judea Pearl, Committee chair

University of California, Los Angeles

1985

To my warm family

Table of Contents

List of Figures

page

Acknowledgements

The advice and assistance of J. Pearl have been instrumental in the development of this dissertation. I would like to express my thanks to him for many stimulating discussions, reading numerous drafts, for making many suggestions, and for his continual support.

I would also like to thank the other members of my committee, Shiela Greibach and Stott Parker for their suggestions throughout my work and their comments on this manuscript, James Jackson and Basil Gordon on their encouragements and interest in the topics of this thesis.

Last but not least, I would like to thank my husband Avi, for enriching my ideas through many discussions and helping in the writing process of this manuscript as well as for providing a moral support, understanding, and patience throughout this ordeal.

VITA

August 13, 1950	Born, Natanya, Israel
1973	B.Sc. in Mathematics-Statistics, Israel, Hebrew University of Jerusalem.
1973-1975	M.Sc, in Applied Mathematics at the Weitzman Institute, Israel.
1976-1978	Staff Member, Everyman's University, Tel-Aviv, Israel.
1977-1978	Teaching Assistant, Tel-Aviv University, Israel.
1978-1980	Research Matematician, Perceptronics Inc., Woodland-Hills, California.
1980-1981	Teaching Assistant, Computer Science department, University of California, Los Angeles
1981-present	Research Assistant, UCLA, Computer Science department.
1982-1984	IBM Scientific Center, Los Angeles.

ABSTRACT OF THE DISSERTATION

Studies in

the Use and Generation of Heuristics

by

Rina Dechter

University of California, Los Angeles, 1985

Professor Judea Pearl, Chair

This dissertation is composed of three studies having a common theme: the strengthening of weak methods by improving the advice that guides them.

The first study examines the effects of removing some of the restrictions imposed on A* and, reexamines whether A* is computationally optimal relative to other algorithms that have access to the same heuristic information. It is shown that the wide class of algorithms which, like A*, return optimal solutions when all cost estimates are optimistic, does not contain an optimal algorithm. On the other hand A* is optimal in a somewhat more restricted sense: either (1) relative to the set of instances in which the heuristic estimates are consistent, or (2) relative to the subclass of algorithms which are Best-First.

The second and third studies examine various means of mechanically generating heuristic advice for weak methods, using the paradigm that heuristics are generated by consulting a simplified model of the task domain.

The second study generates advice to help Backtrack solve Binary Constraint Satisfaction Problems. The advice is generated automatically by consulting relaxed models known to yield a backtrack-free solutions. The information retrieved from these models induces a preference order among the choices pending in the original problem. Optimal algorithms for solving easy problems are presented and analyzed, and the utility of using the advice is evaluated experimentally, using a synthetic domain of CSP problem instances.

The last study is devoted to the analysis of optimization problems that are solvable by Greedy algorithms since such easily solved problems are natural targets for simplification. The contribution of this study is in dealing with ordering optimization problems in which a set of n elements should be ordered to minimize a certain cost function. We give several necessary and sufficient conditions characterizing order dependent cost functions which can be optimized by greedy schemes, and distinguish two types of greedily optimized ordering problems; dominant and non-dominant.

CHAPTER 1

INTRODUCTION AND MOTIVATION

Many AI systems rely on weak methods [Newell 1969] for their problem solving power. These are highly general methods that make weak demands on the task environment (and hence their name [Laired 1983]). Examples of weak methods include generate-and-test, hill climbing, and the variety of heuristic search methods (means-end analysis, depth-first search, best-first search, A*, etc.). Despite them being highly general, weak methods are often guided by some shallow knowledge of the domain to improve their performance. This knowledge, however, is used in very specific functions of the method and, therefore, comes in highly prescribed format. For example, the evaluation function used in A* involves domain-specific knowledge, but it is used only to compare states and select the best among them and it is given in a very restrictive format (e.g., $f=g+h$ in A*).

On the other extreme there are what we normally call "strong methods",. which are more strongly dependent upon knowledge of the specific task domain. Most AI expert systems (e.g [Shortliffe 1976].) constitute strong methods because they rely heavily on specific knowledge that is not transferable to other domains.

A weak method becomes "stronger" when supplied with more domain-specific advice (see figure 1.1): for example when A* is supplied with more accurate h estimates. At present, the source of this advice almost always involves human intervention, which has two shortcomings. First, it decreases the generality of the

method by having to rely on human expertise whenever a new domain is encountered and often this expertise is not available. Second, the quality of human-derived information is sometimes questionable: inhomogeneous, ad-hoc and error-prone.

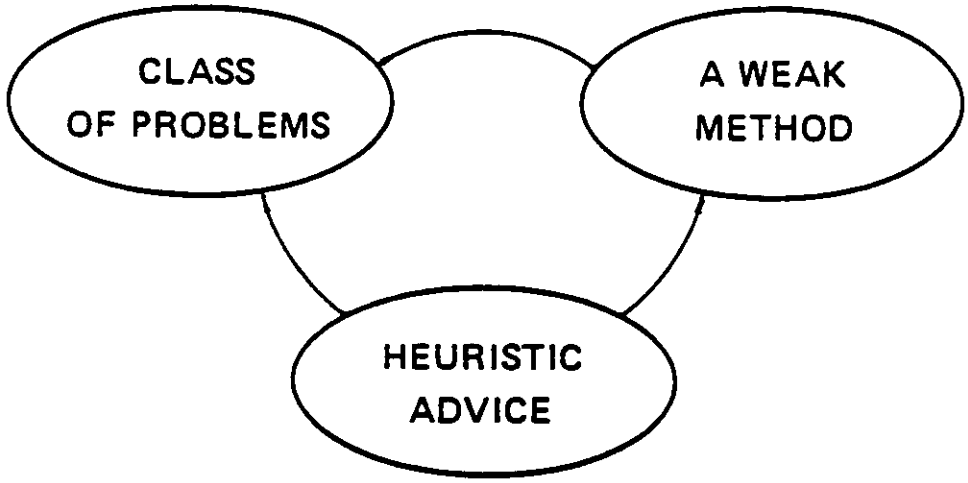


Figure 1.1 - The relation between weak method and heuristic advice.

Understanding how heuristic information is acquired by people, and developing techniques for generating it mechanically should have several benefits. First, it will permit us to strengthen the power of weak methods over a larger variety of problem domains, including new domains where no expertise is available. Second, such techniques may be used to improve the quality of the heuristic information supplied by people. Finally, understanding the processes involved in the generation of heuristics may lead to machines that produce verbal explanations as to how a given heuristic was generated, explicate the assumptions made in the process and argue why the heuristic should work.

This dissertation is composed of three, largely independent, studies all of which have a common theme: the strengthening of weak methods by improving the advice that guides them.

The first study (presented in Chapter 2) examines the effects of removing some of the restrictions imposed on the form of the evaluation function that guides A*. We discuss the generalization of A* into a class of algorithms called **Informed-Best-First**, which, prior to each move, are permitted to examine all the weights (costs and estimates) along each of the exposed search paths. Remarkably, we show that in most cases this added degree of freedom cannot be converted into improved performance.

Specifically, the last section of chapter 2 reexamines whether algorithm A* is computationally optimal relative to other algorithms that have access to the same heuristic information, where optimality is defined in terms of the number of nodes in every applicable problem instance. We show that the wide class of algorithms which, like A*, return optimal solutions (i.e. admissible) when all cost estimates are optimistic (i.e. $h < h^*$), does not contain an optimal algorithm. On the other hand A* is optimal in a somewhat more restricted sense: either (1) relative to the set of instances in which h is consistent, or (2) relative to the subclass of algorithms which are Best-First.

The second and third studies (chapters 3 and 4) examine various means of mechanically generating heuristic advice for weak methods. The paradigm followed is that heuristics are generated by consulting a simplified model of the task domain [Pearl 1983], and involve three steps: simplification, solution, and advice-generation. The implementation of these steps is examined and analyzed relative to two different problem domains, each using a different weak method: (1) Binary Constraint-Satisfaction problems (CSP), searched by informed Backtrack, and (2) Optimization problems, solved by the use of Greedy schemes.

The second study (Chapter 3) investigates extraction of heuristic advice in the context of Binary Constraint Satisfaction Problems i.e., problems that involve the assignment of values to variables subject to a set of constraints. These problems are normally solved by various versions of backtrack search which may be very inefficient. Heuristic advice is needed to guide the order by which the search algorithm assigns values to the variables, so as to minimize the amount of backtracking. The advice is generated automatically by consulting relaxed models of the subproblems created by each value-assignment candidate. The relaxed problems are chosen to yield a backtrack-free solutions, and the information retrieved from these models induces a preference order among the choices pending in the original problem.

We identify a class of CSPs whose syntactic and semantic properties make them easy to solve. The syntactic properties involve the structure of the constraint graph while the semantic properties guarantee some local consistencies among the constraints. In particular, tree-like constraint graphs can be easily solved and are chosen therefore as the target model for the relaxation scheme. Optimal algorithms for solving easy problems are presented and analyzed. A scheme for constructing a "best" constraint-tree approximation to a given constraint graph is introduced and, finally, the utility of using the advice is evaluated in a synthetic domain of CSP problem instances.

The last study (Chapter 4) is devoted to the analysis of Greedy algorithms and optimization problems that are solvable by such algorithms. It is motivated by the need to extend the simplification-based generation of heuristics to optimization problems. In this domain the target normally chosen for the simplification step are easy problems which can be optimally solved by a greedy algorithm, called Greedily optimized. We therefore concentrate on characterization the properties of those

combinatorial problems which render them greedily optimized. Three kinds of optimization problems are treated: selection problems, tree structuring problems, and ordering problems.

A selection problem involves selecting a constrained set of elements with maximum cost when the cost function is symmetric i.e. independent of the order by which the elements were selected. An example is the minimum spanning tree problem. Many of these problems can be modeled as a matroid and the property of being optimally solved by greedy algorithm is explained by the established relationship between greedy algorithms and matroids. This work is presented in [Lawler 1976] and a summary is given in section 4.3. Also presented in this section are problems of structuring an optimal-cost tree such as finding communication codes and merging schemes using the Huffman procedure which is a type of greedy procedure. We give a brief summary of the work in [Parker 1980].

Finally, in section 4.4 we discuss ordering problems, that is, problems in which a set of n elements should be ordered to minimize a certain order dependent cost function. We give several necessary and sufficient conditions characterizing cost functions which are greedily optimized. A compiled list of greedily optimized problems is presented in section 4.5.

The results of this research can be summarized as follows: The first study settles the long time issue surrounding the optimality of A*. We show that the common belief that A* is optimal is true only in very restrictive cases when A* is compared with a narrow class of algorithms and/or relative to a narrow class of problems instances. In general, however, A* is not optimal, nor any other algorithm for that matter. The second study showed that the paradigm of generating heuristic advice by

consulting relaxed models can be useful in the domain of Constraint-Satisfaction problems. We provide the theoretical grounds for this approach and evaluate it experimentally. The experiments, highlight the trade-offs between the strength of the advice and the effort for generating it. Additionally, improved algorithms for solving the classes of easy problems are presented, and analyzed. The main contributions in the third part of this thesis is, first in providing a formal framework for characterizing greedily optimized problems, and second in establishing conditions that delineate two kinds of greedily optimized ordering problems, dominant and non-dominant.

CHAPTER 2

GENERALIZED BEST-FIRST SEARCH STRATEGIES AND THE OPTIMALITY OF A*

This chapter reports several properties of heuristic best-first search strategies whose scoring functions f depend on all the information available from each candidate path, not merely on the current cost g and the estimated completion cost h. We show that several known properties of A* retain their form (with the min-max of f playing the role of the optimal cost) which help establish general tests of admissibility and general conditions for node expansion for these strategies.

Using this framework we then examine the computational optimality of A*, in the sense of never expanding a node that could be skipped by some other algorithm having access to the same heuristic information that A* uses. We define a hierarchy of four optimality types, and consider three classes of algorithms and four domains of problem instances relative to which computational performances are appraised. For each class-domain combination, we then identify the strongest type of optimality that exists and the algorithm achieving it. Our main results relate to the class of algorithms which, like A*, return optimal solutions (i.e., admissible) when all cost estimates are optimistic (i.e., $h \leq h^*$). On this class we show that A* is not optimal and that no optimal algorithm exists, but if we confine the performance tests to cases where the estimates are also consistent, then A* is indeed optimal. Additionally, we show that A* is optimal over a subset of the latter class containing all

best-first algorithms that are guided by path-dependent evaluation functions.

2.1 INTRODUCTION AND PRELIMINARIES

Of all search strategies used in problem solving, one of the most popular methods of exploiting heuristic information to cut down search time is the informed best-first strategy. The general philosophy of this strategy is to use the heuristic information to assess the "merit" latent in every candidate search avenue exposed during the search, then continue the exploration along the direction of highest merit. Formal descriptions of this strategy are usually given in the context of path searching problems (Pearl, 1984), a formulation which represents many combinatorial problems such as routing, scheduling, speech recognition, scene analysis, and others.

Given a weighted directional graph G with a distinguished start node s and a set of goal nodes Γ, the **optimal path problem** is to find a least-cost path from s to any member of Γ where the cost of the path may, in general, be an arbitrary function of the weights assigned to the nodes and branches along that path. A **general best-first** (GBF) strategy will pursue this problem by constructing a tree T of selected paths of G using the elementary operation of **node expansion**, i.e., generating all successors of a given node. Starting with s, GBF will select for expansion that leaf node of T which features the highest "merit", and will maintain in T all previously encountered paths which still appear as viable candidates for sprouting an optimal solution path. The search terminates when no such candidate is available for further expansion, in which case the best solution path found so far is issued as a solution or, if none was found, a failure is proclaimed.

In practice, several short cuts have been devised to simplify the computation of GBF. First, if the evaluation function used for node selection always provides optimistic estimates of the final costs of the candidate paths evaluated, then we can

terminate the search as soon as the first goal node is selected for expansion without compromising the optimality of the solution issued. This guarantee is called **admissibility** and is, in fact, the basis behind the branch-and-bound method [Lawler 1966]. Second, we are often able to purge from T large sets of paths which are recognized at an early stage to be **dominated** by other paths in T [Ibaraki 1977]. This becomes particularly easy if the evaluation function f is **order preserving**, i.e., if for any two paths P_1 and P_2, leading from s to n, and for any common extension P_3 of those paths, the following holds:

$$f(P_1) \geq f(P_2) \qquad f(P_1P_3) \geq f(P_2P_3).$$

Order preservation is a judgemental version of the so called **principle of optimality** in Dynamic Programming (Dreyfus and Law, 1977) and it simply states that if P_1 is judged to be more meritorious than P_2, both going from s to n, then no common extension of P_1 and P_2 may later reverse this judgement. Under such conditions there is no need to keep in T multiple copies of nodes of G; each time the expansion process generates a node n which already resides in T we maintain only the lower f path to it, discarding the link from the more expensive father of n.

These two simplifications are implemented by the following best-first algorithm, a specialization of GBF which we call BF* [Pearl 1984]

Algorithm BF*

1. Put the start node, s, on a list called OPEN of unexpanded nodes.

2. IF OPEN is empty, exit with failure; no solution exists.

3. Remove from OPEN a node, n, at which f is minimum (break ties arbitrarily, but in favor of a goal node) and place it on a list called CLOSED to be used for expanded nodes.

4. If n is a goal node, exit successfully with the solution obtained by tracing back the path along the pointers from n to s, (pointers are assigned in Steps 5 and 6).

5. Expand node n, generating all its successors with pointers back to n.

6. For every successor n' of n:

   a. Calculate $f(n')$

   b. If n' was neither in OPEN nor in CLOSED, then add it to OPEN. Assign the newly computed $f(n')$ to node n'.

   c. If n' already resided in OPEN or CLOSED, compare the newly computed $f(n')$ with that previously assigned to n'. If the new value is lower, substitute it for the old (n' now points back to n instead of to its predecessor). If the matching node n' resided in CLOSED, move it back to OPEN.

7. Go to (2).

By far, the most studied version of BF* is the algorithm A* [Hart 1968] which was developed for **additive cost measures**, i.e, where the cost of a path is defined as the sum of the costs of its arcs. To match this cost measure, A* employs an additive evaluation function $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the currently evaluated path from s to n and h is a heuristic estimate of the cost of the path remaining between n and some goal node. Since $g(n)$ is order preserving and $h(n)$ depends only on the description of the node n, $f(n)$ too is order preserving and one is justified in discarding all but one parent for each node, as in step 6c of BF*. If, in addition, $h(n)$ is a lower bound to the cost of any continuation path from n to Γ,

then $f(n)$ is an optimistic estimate of all possible solutions containing the currently evaluated path, and terminating A* upon the selection of 2 the first goal node (step 4) does not compromise its admissibility[1]. Several other properties of A* can be established if admissibility holds, such as the conditions for node expansion, node reopening, and the fact that the number of nodes expanded decreases with increasing h (results 5, 6, and 7 in [Nilsson 1980a]). These properties are essential for any quantitative analysis of the performance of A* [Pearl 1984].

It has been found, however, that maintaining the admissibility of A* is too restrictive; it limits the selection of heuristics to those which only underestimate costs, and it forces A* to spend disproportionately long time discriminating among roughly equal candidates. As a result, several attempts have been made to execute A* with non-admissible estimates while simultaneously limiting the degree of suboptimality [Pohl 1969, Pohl 1973, Harris 1974, Pearl 1982] and minimizing the computational difficulties that overestimates may induce [Bagchi 1983] However, most of our knowledge regarding the behavior of A* is still limited to additive cost measures and additive evaluation functions.

In this paper, our aim is to examine how the behavior of A* will change if we remove both restrictions. The cost minimization criterion is generalized to include non-additive cost measures of solution paths such as multiplicative costs, the max-cost, (i.e., the highest branch cost along the path) the mode (i.e., the most frequent

[1]Our definition of A* is identical to that of [Nilsson 1980b] and is at variance with [Nilsson 1980a]. The latter regards the requirement $h \leq h^*$ as part of the definition of A*, otherwise the algorithm is called A. We found it more convenient to follow the tradition of identifying an algorithm by how it processes input information rather than by the type of information that it may encounter. Accordingly, we assign the symbol A* to any BF* algorithm which uses the additive combination $f = g + h$, placing no restriction on h, in line with the more recent literature [Barr 1981, Bagchi 1983, Pearl 1984]

branch cost along the path), the range (i.e., the difference between the highest and the lowest branch costs along the path), the cost of the last branch, the average cost, and many others. Additionally, even in cases where the minimization objective is an additive cost measure, we now permit $f(n)$ to take on a more general form and to employ more elaborate evaluations of the promise featured by a given path from s to n, utilizing all the information gathered along that path. For example, one may wish to consult the evaluation function $f(n)=\max_{n'}\{g(n')+h(n')\}$ where n' ranges along the path from s to n. Alternatively, the class of evaluation functions may now include non-linear combinations of g and h in $f=f(g,h)$ and, as a very special example, the additive combination $f=g+h$ with h an arbitrary heuristic function, not necessarily optimistic.

We start by characterizing the performance of the algorithm BF* without assuming any relationship between the cost measure C, defined on complete solution paths and the evaluation function f, defined on partial paths. Later on, assuming a monotonic relationship between f and C on complete solution paths, we establish a relationship between the cost of the solution found and that of the optimal solution (section 2.2). In section 2.3 we establish conditions for node expansion which could be used for analyzing the performance of BF* algorithms. Finally, in section 2.4, we will consider the performance of A* under the additive cost measure and will examine under what conditions A* (employing $f=g+h$) is computationally optimal over other search algorithms which are provided with the same heuristic information h and are guaranteed to find solutions comparable to those found by A*.

For simplicity we shall present our analysis relative to the BF* algorithm with the assumption that f is order preserving. However, all of our results hold for

evaluation functions which are not order preserving if, instead of BF*, GBF is executed, namely, all paths leading to a given node are maintained and evaluated separately.

Notation:

G - directed locally finite graph, $G=(V,E)$

C^* - The cost of the cheapest solution path.

$C(.)$ - the cost function defined over all solution paths

Γ - a set of goal nodes, $\Gamma \subseteq V$

$P_{n_i-n_j}$ - A path in G between node n_i and n_j.

P^s - **a solution path,** i.e., a path in G from s to some goal node $\gamma \in \Gamma$

$c(n,n')$ - cost of an arc between n and n', $c(n,n') \geq \delta > 0$, where δ is a constant.

$f(.)$ - evaluation function defined over partial paths, i.e., to each node n along a given path $P=s_1,n_1,n_2,\ldots,n$ we assign the value $f_P(n)$ which is a shorthand notation for $f(s,n_1,n_2,\ldots,n)$.

$g(n)$ - The sum of the branch costs along the current path of pointers from n to s.

$g^*(n)$ - The cost of the cheapest path going from s to n

$g_P(n)$ - The sum of the branch costs along path P from s to n.

$h(n)$ - A cost estimate of the cheapest path remaining between n and Γ.

$h^*(n)$ - The cost of the cheapest path going from n to Γ

$k(n,n')$ - cost of the cheapest path between n and n'

M - the minimal value of M_j over all j.

M_j - the highest value of $f_P(n)$ along the j^{th} solution path

n^* - a node for which $f(.)$ attains the value M.

s - start node

T - A subtree of G containing all the arcs to which pointers are currently assigned.

For the sake of comparison, we now quote some basic properties of A* [Nilsson 1980a, Pearl 1984] which we later generalize.

Result 1: If $h \leq h^*$, then at any time before A* terminates there exists on OPEN a node n' that is on an optimal path from s to a goal node, with $f(n') \leq C^*$.

Result 2: If there is a path from s to a goal node, A* terminates for every $h \geq 0$ (G can be infinite).

Result 3: If $h \leq h^*$, then algorithm A* is admissible (i.e., it finds an optimal path).

Result 4: If $h \leq h^*$, then any time a node n is selected for expansion by A* it must satisfy: $f_P(n) \leq C^*$ where P is the pointer path assigned to n at the time of expansion.

Result 5: If $h \leq h^*$, then every node in OPEN for which $f(n) < C^*$ will eventually be expanded by A*.

2.2. ALGORITHMIC PROPERTIES OF BEST-FIRST (BF*) SEARCH

In locally finite graphs the set of solution paths is countable, so they can be enumerated:

$$P_1^S, P_2^S, \ldots, P_j^S, \ldots . \tag{1}$$

and correspondingly, we can use the notation $f_j(n)$ to represent $f_{P_j^S}(n)$. Let M_j be the maximum of f on the solution path P_j^S, i.e.:

$$M_j = \max_{n \in P_j^S} \{f_j(n)\} \tag{2}$$

and let M be the minimum of all the M_j's:

$$M = \min_j \{M_j\} . \tag{3}$$

The minmax value M can be interpreted as the level of the "saddle-point" in the network of paths leading from s to Γ. We shall henceforth assume that both the max and the min functions are well defined.

Lemma 1:

If BF* uses an order-preserving f, and P_1 and P_2 are any two paths from s to n such that the pointers from n to s currently lie along P_1 and all the branches of P_2 have been generated in the past, then

$$f_{P_1}(n) \leq f_{P_2}(n). \tag{4}$$

In particular, if P_t is known to be the path assigned to n at termination, then

$$f_{P_t}(n) \leq f_P(n) \quad \forall P \in G_e \tag{5}$$

where G_e is the subgraph explicated during the search, namely, the union of all past traces of T.

Proof:

The lemma follows directly from the order-preserving properties of f and the fact that pointers are redirected towards the path yielding a lower f value. The former guarantees that if at least one node on P_2 was ever assigned a pointer directed along an alternative path, sjperior to the direction of P_2, then the entire P_2 path to n will remain inferior to every alternative path to n, subsequently exposed by BF*.

□

2.2.1 Termination and Completeness

Lemma 2:

At any time before BF* terminates, there exists on OPEN a node n' that is on some solution path and for which $f(n') \leq M$.

Proof:

Let $M = M_j$, i.e., the minmax is obtained on solution path P_j^S. Then at some time before termination, let n' be the shallowest OPEN node on P_j^S, having pointers directed along P_i^S (possibly $i=j$). From the definition of M_j:

$$M_j = \max_{n \in P_j^S} \{f_j(n)\}, \tag{6}$$

therefore,

$$f_j(n') \leq M_j = M. \tag{7}$$

Moreover, since all ancestors of n' on P_j^S are in CLOSED and BF* has decided to assign its pointers along P_i^S, Lemma 1 states

$$f_i(n') \leq f_j(n'). \tag{8}$$

This implies

$$f_i(n') \leq M \tag{9}$$

which proves Lemma 2.

□

Lemma 3:

Let P_j^S be the solution path with which BF* terminates; then

a. any time before termination there is an OPEN node n on P_j^S for which $f(n)=f_j(n)$

b. M is obtained on P_j^S, i.e., $M=M_j$.

Proof:

a. Let n be the shallowest OPEN node on P_j^S at some arbitrary time t, before termination. Since all n's ancestors on P_j^S are closed at time t, n must be assigned an f at least as cheap as $f_j(n)$. Thus, $f_j(n) \geq f(n)$ with strict inequality holding only if at time t n is found directed along a path different than P_j^S. However, since BF* eventually terminates with pointers along P_j^S, it must be that BF* has never encountered another path to n with cost lower than $f_j(n)$. Thus, $f(n)=f_j(n)$.

b. Suppose BF* terminates on P_j^S, but $M_j>M$, and let $n^* \in P_j^S$ be a node where $f_j(.)$ attains its highest value, i.e., $f_j(n^*)=M_j$. At the time that n^* is last chosen for expansion its pointer must already be directed along P_j^S and, therefore, n^{**} is assigned the value $f(n^*)=f_j(n^*)>M$. At that very time there exists an OPEN node n' having $f(n') \leq M$ (Lemma 2), and so

$$f(n') < f(n^*). \tag{10}$$

Accordingly, n' should be expanded before n* which contradicts our

supposition.

□

Theorem 1:

If there is a solution path and f is such that $f_P(n)$ is unbounded along any infinite path P, then BF* terminates with a solution,i.e., BF* is complete.

Proof:

In any locally finite graph there is only a finite number of paths with finite length. If BF* does not terminate, then there is at least one infinite path along which every finite-depth node will eventually be expanded. That means that f must increase beyond bounds and, after a certain time t, no OPEN nodes on any given solution path will ever be expanded. However, from Lemma 2, $f(n') \leq M$ for some OPEN node n' along a solution path, which contradicts the assumption that n' will never be chosen for expansion.

□

The condition of Theorem 1 is clearly satisfied for additive cost measures due to the requirement that each branch cost be no smaller than some constant δ. It is also satisfied for many quasi-additive cost measures (e.g., $\left[\sum_i c_i^r\right]^{1/r}$), but may not hold for "saturated" cost measures such as the maximum cost. In the latter cases, termination cannot be guaranteed on infinite graphs and must be controlled by special means such as using iteratively increasing depth bounds.

We shall soon see that the importance of M lies not only in guaranteeing the termination of BF* on infinite graphs but mainly in identifying and appraising the solution path eventually found by BF*.

2.2.2 Quality of the Solution, and the Admissibility of BF*

So far we have not specified any relation between the cost function C, defined on solution paths, and the evaluation function f, defined on partial paths or solution-candidates. Since the role of f is to guide the search toward the lowest cost solution path, we now impose the restriction that f be monotonic with C when evaluated on complete solution paths, i.e., if P and Q are two solution paths with $C(P) > C(Q)$, then $f(P) > f(Q)$. Since $C(P)$ and $C(Q)$ can take on arbitrarily close values over our space of problem instances, monotonicity can be represented by:

$$f(s,n_1,n_2,\ldots,\gamma) = \psi[C(s,n_1,n_2,\ldots\gamma)] \qquad \gamma \in \Gamma \tag{11}$$

where ψ is some increasing function of its argument defined over the positive reals. No restriction, however, is imposed on the relation between C and f on non-goal nodes, that is, we may allow evaluation functions which treat goal nodes preferentially, for example:

$$f(s,n_1,n_2,\ldots,n) = \begin{cases} \psi[C(s,n_1,n_2,\ldots,n)] & \text{if } n \in \Gamma \\ F(s,n_1,n_2,\ldots,n) & \text{if } n \notin \Gamma \end{cases} \tag{12}$$

where $F(\cdot)$ is an arbitrary function of the path $P = s,n_1,n_2,\ldots n$. The additive evaluation function $f = g+h$ used by A* is, in fact, an example of such goal-preferring types of functions; the condition $h(\gamma)=0$ guarantees the identity $f=C$ oh solution paths, while on other paths f is not governed by C since, in general, h could take on arbitrary values. Other examples of goal-preferring f evolve in branch-and-bound methods of solving integer-programming problems where the quality of a solution path is determined solely by the final goal node reached by that path. Here $f(n)$ may be entirely arbitrary for non-goal nodes, but if n is a goal-node we have $f(n)=C(n)$.

We now give some properties of the final solution path using the relationship stated above.

Theorem 2:

BF* is $\psi^{-1}(M)$–*admissible*, that is, the cost of the solution path found by BF* is at most $\psi^{-1}(M)$.

Proof:

Let BF* terminate with solution path $P_j^S = s,...,t$ where $t \in \Gamma$. From Lemma 2 we learn that BF* cannot select for expansion any node n having $f(n)>M$. This includes the node $t \in \Gamma$ and, hence, $f_j(t) \leq M$. But (11) implies that $f_j(t) = \psi[C(P_j^S)]$ and so, since ψ and ψ^{-1} are monotonic,

$$C(P_j^S) \leq \psi^{-1}(M) \tag{13}$$

which proves the Theorem.

□

A similar result was established by Bagchi and Mahanti [Bagchi 1983] although they used $\psi(C)=C$ and restricted f to the form $f=g+h$.

Theorem 2 can be useful in appraising the degree of suboptimality exhibited by non-admissible algorithms. For example, [Pohl 1973] suggests a dynamic weighting scheme for the evaluation function f. In his approach the evaluation function f is given by:

$$f(n) = g(n) + h(n) + \varepsilon \left[1 - \frac{d(n)}{N}\right] h(n) \tag{14}$$

where $d(n)$ is the depth of node n and N is the depth of the shallowest optimal goal node. Using Theorem 2 we can easily show that if $h(n) \leq h^*(n)$ then BF* will always find a solution within a $(1+\varepsilon)$ cost factor of the optimal, a property first shown

by.

[Pohl 1973] In this case, the requirement $h(\gamma)=0$ for $\gamma \in \Gamma$ and equation (11) together dictate $\psi(C)=C$, and we can bound M by considering $\max_n f_{P^*}(n)$ along any optimal path P^*. Thus,

$$M \le \max_{n \in P^*} f_{P^*}(n) \tag{15}$$

$$\le \max_{n \in P^*} \left[g^*(n)+h^*(n)+\varepsilon h^*(n) \left[1-\frac{d(n)}{N} \right] \right]$$

$$= C^*+\varepsilon h^*(s)$$

$$= C^*(1+\varepsilon)$$

On the other hand, according to Theorem 2, the search terminates with cost $C_t \le M$, hence

$$C_t \le C^*(1+\varepsilon). \tag{16}$$

This example demonstrates that any evaluation function of the form

$$f(n) = g(n) + h(n) \left[1+\varepsilon \rho_P(n) \right] \tag{17}$$

will also return a cost at most $(1+\varepsilon)C^*$, where $\rho_P(n)$ is an arbitrary path-dependent function of node n, satisfying $\rho_P(n) \le 1$ for all n along some optimal path P^*. For example, the functions $\rho_P(n) = \left[\frac{h(n)}{g(n)+h(n)} \right]^K$ or $\rho_P(n) = \left[1+d(n) \right]^{-K}, K > 0$, will qualify as replacements for $[1-d(n)/N]$.

The main utility of Theorem 2, however, lies in studying search on graphs with random costs where we can use estimates of M to establish probabilistic bounds on the degree of suboptimality, $C_t - C^*$. An example of such an option arises

in the problem of finding the cheapest root-to-leaf path in a uniform binary tree of height N, where each branch independently may have a cost of 1 or 0 with probability p and $1-p$, respectively.

It can be shown [Karp 1983] that for $p > 1/2$ and large N the optimal cost is very likely to be near α^*N where α^* is a constant determined by p. Consequently, a natural evaluation function for A* would be $f(n)=g(n)+\alpha^*[N-d(n)]$ and, since it is not admissible, the question arises whether C_t, the cost of the solution found by A*, is likely to deviate substantially from the optimal cost C^*. A probabilistic analysis shows that although in the worst case M may reach a value as high as N, it is most likely to fall in the neighborhood of α^*N or C^*. More precisely, it can be shown (see Appendix 2.1) that, as $N \to \infty$, $P[M \geq (1+\epsilon)C^*] \to 0$ for every $\epsilon > 0$ Thus, invoking Theorem 2, we can guarantee that as $N \to \infty$ A* will almost always find a solution path within a $1+\epsilon$ cost ratio of the optimal, regardless of how small ϵ is.

Theorem 2 can also be used to check for strict admissibility, i.e., $C_t=C^*$; all we need to do is to verify the equality $\psi^{-1}(M) = C^*$. This, however, is more conveniently accomplished with the help of the next corollary. It makes direct use of the facts that 1) an upper bound on f along any solution path constitutes an upper bound on M and, 2) the relation between $f_P(n)$ and C^* is more transparent along an optimal path. For example, in the case of A* with $h \leq h^*$, the relation $f_{P^*}(n) \leq C^*$ is self evident.

Corollary 1:

If in every graph searched by BF* there exists at least one optimal solution path along which f attains its maximal value on the goal node, then BF* is admissible.

Proof:

Let BF* terminate with solution path $P_j^S = s,\ldots,t$ and let $P^* = s,\ldots,\gamma$ be an optimal solution path such that $\max_{n \in P^*} f_{P^*}(n) = f_{P^*}(\gamma)$. By Theorem 2 we know that $f_j(t) \le M$. Moreover, from the definition of M we have $M \le \max_{n \in P_i^S} f_i(n)$ for every solution path P_i^S. In particular, taking $P_i^S = P^*$, we obtain

$$f_j(t) \le M \le \max_{n \in P^*} f_{P^*}(n) = f_{P^*}(\gamma). \tag{18}$$

However, from (11) we know that f is monotonic increasing in C when evaluated on complete solution paths, thus

$$\psi(C(P_j^S)) \le \psi(C^*) \tag{19}$$

implying

$$C(P_j^S) \le C^*, \tag{20}$$

which means that BF* terminates with an optimal-cost path.

□

By way of demonstrating its utility, Corollary 1 can readily delineate the range of admissibility of Pohl's weighted evaluation function $f_w = (1-w)g + wh$, $0 \le w \le 1$ (see [Pohl 1969]). Here $\psi(C) = (1-w)C$ which complies with (11) for $w < 1$. It remains to examine what values of $w < 1$ will force f_w to attain its maximum at the end of some optimal path $P^* = s,\ldots,\gamma$. Writing

$$f_{P^*}(n) \le f_{P^*}(\gamma) \tag{21}$$

we obtain

$$(1-w)g^*(n) + wh(n) \le (1-w)g^*(\gamma) = (1-w)[g^*(n) + h^*(n)], \tag{22}$$

or

$$\frac{1-w}{w} \ge \frac{h(n)}{h^*(n)}. \tag{23}$$

Clearly, if the ratio $\frac{h(n)}{h^*(n)}$ is known to be bounded from above by a constant β, then

$$w < \frac{1}{1+\beta} \tag{24}$$

constitutes the range of admissibility of BF*. Note that the use of $w > 1/2$ may be permissible if h is known to consistently underestimate h^* such that $\frac{h(n)}{h^*(n)} \leq \beta < 1$. Conversely, if h is known to be non-admissible with $\beta > 1$, then the use of $w = \frac{1}{1+\beta}$ will turn BF* admissible.

Another useful application of Corollary 1 is to check whether a given combination of g and h, $f = f(g,h)$, would constitute an admissible heuristic in problems of minimizing additive cost measures. If $h \leq h^*$ and f is monotonic in both arguments, then Corollary 1 states that $f(g,h)$ is guaranteed to be admissible as long as

$$f(g, C-g) \leq f(C,0) \quad \text{for } 0 \leq g \leq C. \tag{25}$$

Thus, for example, $f = \sqrt{g^2+h^2}$ is admissible while $f = (g^{1/2} + h^{1/2})^2$ is not. In general, any combination of the form $f = \phi[\phi^{-1}(g) + \phi^{-1}(h)]$ will be admissible if ϕ is monotonic non-decreasing and concave.

2.3. CONDITIONS FOR NODE EXPANSION

In section 2.3.1 we present separate conditions of necessity and sufficiency for nodes expanded by BF* on graphs. Further restricting the problem domain to trees will enable us to establish in section 2.3.2 an expansion condition which is both necessary and sufficient.

2.3.1 Expansion Conditions for Graphs

Lemma 4:

BF* chooses for expansion at least one node n such that at the time of this choice $f(n)=M$.

Proof: Let BF* terminate with P_j^S and let $n^* \in P_j^S$ be such that $f_j(n^*)=M_j$. From Lemma 3, $M_j=M$. Moreover, at the time that n^* is last expanded, it is pointed along P_j^S. Hence,

$$f(n^*) = f_j(n^*) = M_j = M. \tag{26}$$

□

Theorem 3:

Any node expanded by BF* has $f(n) \leq M$ immediately before its expansion.

Proof:

Follows directly from Lemma 2.

□

Theorem 4:

Let n^* be the first node with $f(n^*)=M$ which is expanded by BF* (there is at least one). Any node which prior to the expansion of n^* resides in OPEN with $f(n)<M$ will be selected for expansion before n^*.

Proof:

$f(n)$ can only decrease through the redirection of its pointers. Therefore, once n satisfies $f(n)<M$, it will continue to satisfy this inequality as along as it is in OPEN. Clearly, then, it should be expanded before n^*.

□

Note the difference between Theorems 3 and 4 and their counterparts, results 4 and 5, for A*. First, M plays the role of C^*. Second, the sufficient condition for expansion in Theorem 4, unlike that of result 5, requires that n not merely reside in OPEN but also enters OPEN before n^* is expanded. For a general f, it is quite possible that a node n may enter OPEN satisfying $f(n)<f(n^*)=M$ and still will not be expanded.

We will now show that such an event can only occur to descendants of nodes n^* for which $f(n^*)=M$, i.e., it can only happen to a node n reachable by an M-bounded path but not by a strictly M-bounded path.

Definition: A path P will be called M-bounded if every node n along P satisfies $f_P(n)\leq M$. Similarly, if a strict inequality holds for every n along P, we shall say that P is strictly M-bounded.

Theorem 5:

Any node reachable from s by a strictly M-bounded path will be expanded by BF*.

Proof:

Consider a strictly M-bounded path P from s to n (M cannot be obtained on s). We can prove by induction from s to n that every node along P enters OPEN before n^* is expanded and hence, using Theorem 4, n will be expanded before n^*.

□

In Section 4 we will use this result to compare the performance of A* to that of other (generalized) best-first algorithms.

The final results we wish to establish now are necessary and sufficient conditions for node expansion which are superior to Theorems 4 and 5 in that they also determine the fate of the descendants of n^*.

Theorem 6:

Let P_j^s be the solution path eventually found by BF* and let n_i be the depth-i node along $P_j^s, i=0,1,\ldots$. A necessary condition for expanding an arbitrary node n in the graph is that for some $n_i \in P_j^s$ there exists an L_i-bounded path from n_i to n where $L_i = \max_{k>i} f_j(n_k)$. In other words, there should exist a path P_{n_i-n} along which

$$f(n') \leq \max_{k>i} f_j(n_k) \qquad \forall n' \in P_{n_i-n} \tag{27}$$

Moreover, a sufficient condition for expanding n is that (27) be satisfied with strict inequality.

Proof:

Assume that n is expanded by BF* and let n_k be the shallowest OPEN node on P_j^S at time t_n when n is selected for expansion (see Figure 2.1).

Since P_j^S is the solution path eventually found by BF* we know (see proof of Lemma 3) that at time t_n n_k is pointed along P_j^S and, therefore,

$$f(n) \leq f(n_k) = f_j(n_k). \tag{28}$$

We are now ready to identify the node n_i on P_j^S which satisfies (27). Let P_{s-n} be the path along which n's pointers are directed at time t_n, and let n_i be the deepest common ancestor of n and n_k along their respective pointer paths P_{s-n} and P_j^S. Since n_i

Figure 2.1 - Used in the proof for node expansion in theorem 6

is an ancestor of n_k we have $i<k$ and so, $f(n) \leq f_j(n_k)$ implies

$$f(n) \leq \max_{k>i} f_j(n_k). \tag{29}$$

We now repeat this argument for every ancestor n' of n along the P_{n_i-n} segment of P_{s-n}. At the time it was last expanded, each such ancestor n' may have encountered a different $n_{k'}$ in OPEN, but each such $n_{k'}$ must have been a descendent of n_i along P_j^S satisfying $f(n') \leq f_j(n_{k'})$. Hence, (27) must be satisfied for all nodes n' along the P_{n_i-n} segment of P_{s-n} which proves the necessary part of Theorem 6.

Sufficiency is proven by assuming that $\max_{k>i} f_j(n_k)$ occurs at some node $n_{k'} \in P_j^S$, $k'>i$. Both n_i and $n_{k'}$ are eventually expanded by BF* and so, if n is not already expanded at the time t' when $n_{k'}$ is last selected for expansion, then P_{n_i-n} should contain at least one OPEN node. We now identify n' as the shallowest OPEN node on P_{n_i-n} at time t', for which we know that

$$f(n') \leq f_{P_{n'-n}}(n'). \tag{30}$$

However, since (27) is assumed to hold with strict inequality for any n' along P_{n_i-n},

we must conclude

$$f(n') \leq f_{P_{s \to n}}(n') < f(n_k) \tag{31}$$

implying that n', not n_k, should have been chosen for expansion at time t', thus contradicting our supposition that n remains unexpanded at time t'.

□

The expansion condition of Theorem 6 plays a major role in analyzing the average complexity of non-admissible algorithms, where f is treated as a random variable [Pearl 1984] This condition is rather complex for general graphs since many paths may stem from P_j^t toward a given node n, all of which must be tested according to Theorem 6. The test is simplified somewhat in the case of trees since (27) need only be tested for one node, $n_i \in P_j^t$, which is the deepest common ancestor of n and γ. Still, the condition stated in Theorem 6 requires that we know which path will eventually be found by the algorithm and this, in general, may be a hard task to determine a-priori. An alternative condition, involving only the behavior of f across the tree, will be given in the next subsection.

2.3.2 Conditions for Expansion on Trees

Here we present a necessary and sufficient condition for node's expansion by algorithm BF* under the assumption that the graph to be searched is a tree and the tie breaking rule is "leftmost-first". This condition, unlike those in Section 2.3.1, do not require knowing the solution path found but invokes only the properties of the search graph and the evaluation function f.

The following notations and definitions will be used:

T_s - a tree rooted at node s. The children of each node are ordered from left to

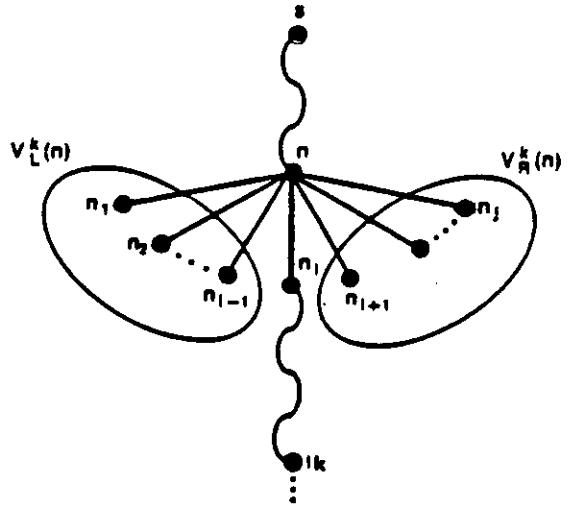


Figure 2.2 - The left and right values of ancestor n of k, w.r.t the path P_{s-k}.

right. N denotes the depth of T_s. Some of its leaf nodes are marked as goal nodes.

T_r - a subtree of T_s rooted at node r, $r \in T_s$.

M_r - the minimax value M associated with solution paths in subtree T_r. If T_r contains no goal nodes we will define $M_r = \infty$.

P_{i-r} - the path from i to r (including, as usual, the two end nodes).

$P_{(i-r]}$ - the path from i to r excluding node i.

$P_{[i-r)}$ - the path from i to r excluding node r.

$P_{(i-r)}$ - the path from i to r excluding nodes i and r.

$A_{i,j}$ - the deepest common ancestor of nodes i and j for which i is not a descendent of j and j is not a descendent of i.

$f(r)$ - the evaluation function $f_{P_{s-r}}(r)$.

For any ancestor node n of a given node k we now define two parameters called **left value** and **right value** to be the minimum among the M values of the left and right subtrees rooted at n, respectively. The terms left and right are defined with respect to path P_{s-k} (see Figure 2.2).

Definition: Let k be a node at depth k $(k \leq N)$ in T_s and let n be a node on $P_{[s-k)}$, with its sons $n_1, n_2, \ldots, n_i, \ldots, n_b$ ordered from left to right where n_i is on path P_{s-k}. **The left value of** n with respect to k, $V_L^k(n)$, is given by

$$V_L^k(n) = \min_{1 \leq j \leq i-1} \left\{ M_{n_j} \right\} \tag{32}$$

Similarly, the **right value of** n with respect to k 11 is

$$V_R^k(n) = \min_{i+1 \leq j \leq b} \left\{ M_{n_j} \right\} \tag{33}$$

Definition: We say that n_i **is to the left of** n_j if some ancestor of n_i is a left sibling of an ancestor of n_j.

Obviously for any two nodes in T_s either one of them is a descendent of the other or one of them is to the left of the other (but not both). This renders the execution of algorithm BF* on T_s unique since, at any time, all nodes in OPEN are totally ordered by both the size of f and the "left of" relationship.

Lemma 5:

Let k be a node in T_s which is expanded by BF* and let n be to the left of k with $n' = A_{n,k}$. If the path $P_{n'-n}$ is bounded above by $f(k)$ then n will be expanded before k (see Figure 2.3a).

Proof:

Assume $P_{n'-n}$ is bounded above by $f(k)$ but k is expanded before n. Immediately before k is expanded there is a node n'' on $P_{n'-n}$ which is on OPEN. Since we have $f(n'') \leq f(k)$ and n'' is to the left of k (since n is), n'' should be selected for expansion and not k which contradicts our supposition.

□

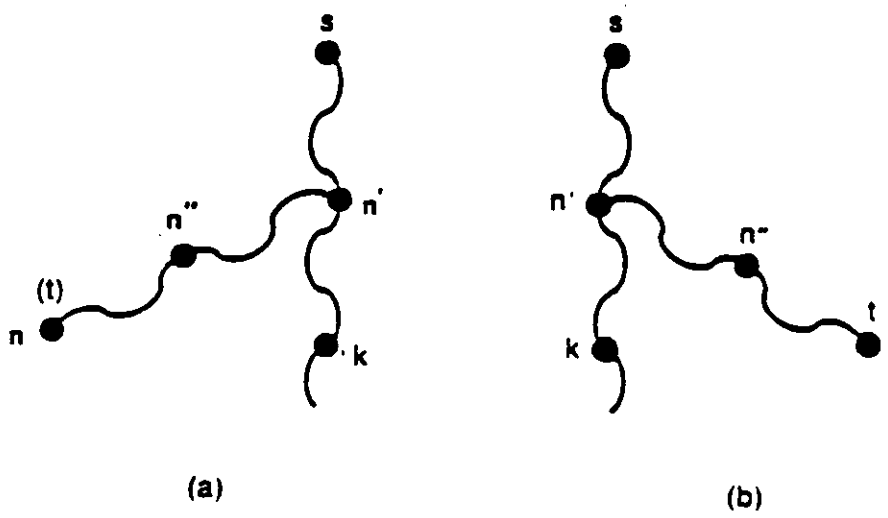


Figure 2.3 - Used in the proof of lemma 5

We are now ready to present the condition for node expansion.

Theorem 7:

A node k at depth k of the search tree T_s will be expanded by BF* if and only if the f value for each ancestor n of k along the path P_{s-k}, is lower than the minimum M values of all subtrees stemming to the left of the path P_{s-n}, and is not higher than the minimum M values of all subtrees stemming to the right of P_{s-n}. Formally:

For every n on P_{s-k}

$$f(n) < \min \left\{ V_L^k(n') \mid n' \in P_{[s-n)} \right\} \tag{34}$$

and

$$f(n) \leq \min \left\{ V_R^k(n') \mid n' \in P_{[s-n)} \right\} \tag{35}$$

Proof:

In part (1) we show that the condition is necessary and in part (2) we prove its sufficiency.

1. Assume to the contrary that node k was expanded but there is a node on $P_{(s-k)}$ which does not satisfy the corresponding inequality. Let n be the first such node with this counter property, that is, either

   $$(a) \quad f(n) \geq \min \left\{ V_L^k(n') \mid n' \in P_{(s-n)} \right\}$$

   or

   $$(b) \quad f(n) > \min \left\{ V_R^k(n') \mid n' \in P_{(s-n)} \right\}$$

   Assume that (a) holds and let n^* be a node on $P_{(s-n)}$, on which the minimum of the right-hand side of (a) is attained, i.e.,

   $$V_L^k(n^*) = \min \left\{ V_L^k(n') \mid n' \in P_{(s-n)} \right\}. \tag{36}$$

   According to the definition of $V_L^k(n^*)$, there is a path P_{n^*-t} from n^* to a goal node t that is situated to the left of n which is bounded by $V_L^k(n^*)$. Since $V_L^k(n^*) \leq f(n)$, the path P_{n-t} is bounded by $f(n)$ and thus, by Lemma 5 t will be expanded before n and the algorithm will halt without expanding n

and k which contradicts our supposition that k is expanded by BF*.

Assume now that (b) holds. Using the same argument we should conclude that exactly before n is chosen for expansion there is a path from an OPEN node to a goal (situated to the right of n) which is strictly bounded below $f(n)$. This path should be expanded before n and the algorithm will terminate without expanding n or k.

2. Let k be a node at depth k that satisfies the condition. We will show that k must be expanded by BF*. Let t be the goal node on which BF* halts. If t is a descendent of k then, obviously k must be expanded. Otherwise, t is either to the left of k or to its right.

case a: t is to the left of k. Let $n' = A_{t,k}$, n' is on the path P_{s-k}. Let M be the max f value on $P_{(n'-t]}$ and $n'' \in P_{(n'-t]}$ with $f(n'') = M$ (see figure 2.3a). Since t is expanded, n' must be expanded, and therefore, before the algorithm expands n''there is always an OPEN node on $P_{(n'-k]}$. From the condition of the theorem

$$\forall\, n \in P_{(n'-k]} \quad f(n) < M = f(n'') \tag{37}$$

and therefore all the nodes on $P_{(n'-k]}$ must be expanded before n''. Knowing that $n''$is expanded implies the expansion of k and all its ancestors.

case b: t is to the right of k. The situation in this case is shown in Figure 2.3b. From the condition of the theorem it follows that the path $P_{(n'-k]}$ is bounded below $f(n'')$ and therefore from lemma 5 k should be expanded before n''.

□

2.4. ON THE OPTIMALITY OF A*

2.4.1 Previous Works and the Notion of Equally-Informed

The optimality of A*, in the sense of computational efficiency, has been a subject of some confusion. The well-known property of A* which predicts that decreasing errors h^*-h can only improve its performance (result 6 in [Nilsson 1980a]) has often been interpreted to reflect some supremacy of A* over other search algorithms of equal information. Consequently, several authors have assumed that A*'s optimality is an established fact (e.g [Nilsson 1980b, Mero 1984].). In fact, all this property says is that some A* algorithms are better than other A* algorithms depending on the heuristics which guide them. It does not indicate whether the additive rule $f=g+h$ is the best way of combining g and h, neither does it assure us that expansion policies based only on g and h can do as well as more sophisticated policies that use the entire information gathered by the search. These two conjectures will be examined in this section, and will be given a qualified confirmation.

The first attempt to prove the optimality of A* was carried out by Hart, Nilsson and Raphael [Hart 1968] and is summarized in [Nilsson 1980b] Basically, Hart et al. argue that if some admissible algorithm B fails to expand a node n expanded by A*, then B must have known that any path to a goal constrained to go through node n is nonoptimal. A*, by comparison, had no way of realizing this fact because when n was chosen for expansion it satisfied $g(n)+h(n) \leq C^*$, clearly advertizing its promise to deliver an optimal solution path. Thus, the argument goes, B must have obtained extra information from some external source, unavailable to A* (perhaps by computing a higher value for $h(n)$), and this disqualifies B from being an "equally informed", fair competitor, to A*.

The weakness of this argument is that it fails to account for two legitimate ways in which B can decide to refrain from expanding n based on information perfectly accessible to A*. First, B may examine the properties of previously exposed portions of the graph and infer that n actually deserves a much higher estimate than $h(n)$. A*, on the other hand, although it has the same information available to it in CLOSED, cannot put it into use because it is restricted to take the estimate $h(n)$ at face value and only judge nodes by the score $g(n)+h(n)$. Second, B may also gather information while exploring sections of the graph unvisited by A*, and this should not render B an unfair, "more informed" competitor to A* because in principle A* too had an opportunity to visit those sections of the graph. Later in this section (see Figure 2.6) we demonstrate the existence of an algorithm B which manages to outperform A* using this kind of information.

Gelperin (1978) has correctly pointed out that in any discussion of the optimality of A* one should also consider algorithms which adjust their h in accordance with the information gathered during the search. His analysis, unfortunately, falls short of considering the entirety of this extended class, having to follow an over-restrictive definition of **equally-informed**. Gelperin's interpretation of "an algorithm B is **never more informed** than A*", instead of just restricting B from using information inaccessible to A, actually forbids B from processing common information in a better way than A does. For example, if B is a best-first algorithm guided by f_B, then in order to qualify for Gelperin's definition of "never more informed than A*," B is forbidden from ever assigning to a node n a value $f_B(n)$ higher than $g(n)+h(n)$, even if the information gathered along the path to n justifies such an assignment.

In our analysis we will use the natural definition of "equally informed," allowing the algorithms compared to have access to the same heuristic information while placing no restriction on the way they use it. Accordingly, we assume that an arbitrary heuristic function $h(n)$ is assigned to the nodes of G and that the value $h(n)$ is made available to each algorithm that chooses to generate node n. This amounts to viewing $h(n)$ as part of the parameters that specify problem-instances and correspondingly, we shall represent each problem instance by the quadruple $I = (G, s, \Gamma, h)$.

We will demand, however, that A* only be compared to algorithms that return optimal solutions in those problem instances where their computational performances are to be appraised. In particular, if our problem space contains only cases where $h(n) \leq h^*(n)$ for every n in G, we will only consider algorithms which, like A*, return least-cost solutions, in such cases. The class of algorithms answering this conditional admissibility requirement will simply be called **admissible** and will be denoted by $\underline{A}_{ad}$. From this general class of algorithms we will later examine two subclasses $\underline{A}_{gc}$ and $\underline{A}_{bf}$. $\underline{A}_{gc}$ denotes the class of algorithms which are **globally compatible** with A*, i.e., they return optimal solutions whenever A* does, even in cases where $h > h^*$. $\underline{A}_{bf}$ stands for the class of admissible BF* algorithms, i.e., those which conduct their search in a best-first manner, being guided by a path-dependent evaluation function as in Section 2.1.

Additionally, we will assume that each algorithm compared to A* uses the primitive computational step of **node expansion**, that it only expands nodes which were generated before, and that it begins the expansion process at the start node s. This excludes, for instance, bi-directional searches [Pohl 1971] or algorithms which

simultaneously grow search trees from several "seed nodes" across G.

2.4.2. Nomenclature and a Hierarchy of Optimality Relations

Our notion of optimality is based on the usual requirement of Dominance [Nilsson 1980a].

Definition: Algorithm A is said to **dominate** algorithm B relative to a set $\underline{I}$ of problem instances iff in every instance $I \in \underline{I}$, the set of nodes expanded by A is a subset of the set of nodes expanded by B. A **strictly dominates** B iff A dominates B and B does not dominate A, i.e., there is at least one instance where A skips a node which B expands, and no instance where the opposite occurs.

This definition is rather stringent because it requires that A establishes its superiority over B under two difficult tests:

1. expanding a subset of nodes rather than a smaller number of nodes

2. outperform B in every problem instance rather than the majority of instances

Unfortunately, there is no easy way of loosening any of these requirements without invoking statistical assumptions regarding the relative likelihood of instances in $\underline{I}$. In the absence of an adequate statistical model, requiring dominance remains the only practical way of guaranteeing that A expands fewer nodes than B, because if in some problem instance we would allow B to skip even one node that is expanded by A, one could immediately present an infinite set of instances where B grossly outperforms A. (This is normally done by appending to the node skipped a variety of trees with negligible costs and very low h).

Adhering to the concept of dominance, the strong definition of optimality proclaims algorithm A **optimal** over a class $\underline{A}$ of algorithms iff A dominates every member of $\underline{A}$. Here the combined multiplicity of $\underline{A}$ and $\underline{I}$ also permits weaker definitions, for example, we may proclaim A **weakly optimal** over $\underline{A}$ if no member of $\underline{A}$ strictly dominates A. The spectrum of optimality conditions becomes even richer when we examine A*, which stands for not just one but a whole family of algorithms, each defined by the tie-breaking-rule chosen. We chose to classify this spectrum into the following four types (in a hierarchy of decreasing strength):

Type 0: A* is said to be 0-optimal over $\underline{A}$ relative to $\underline{I}$ iff in every problem instance $I \in \underline{I}$ every tie-breaking-rule in A* expands a subset of the nodes expanded by any member of $\underline{A}$. (In other words, every tie-breaking-rule dominates all members of $\underline{A}$.)

Type 1: A* is said to be 1-optimal over $\underline{A}$
relative to $\underline{I}$ iff in every problem instance $I \in \underline{I}$ there exists at least one tie-breaking-rule which expands a subset of the set of nodes expanded by any member of $\underline{A}$.

Type 2: A* is said to be 2-optimal over $\underline{A}$
relative to $\underline{I}$ iff there exists no problem instance $I \in \underline{I}$ where some member of $\underline{A}$ expands a proper subset of the set of nodes which are expanded by some tie-breaking-rule in A*.

Type 3: A* is said to be 3-optimal over $\underline{A}$
relative to $\underline{I}$ iff the following holds: if there exists a problem instance $I_1 \in \underline{I}$ where some algorithm $B \in \underline{A}$ skips a node expanded by some tie-breaking-

rule in A*, then there must also exist some problem instance $I_2 \in \underline{I}$ where that tie-breaking-rule skips a node expanded by B. (In other words, no tie-breaking-rule in A* is strictly dominated by some member of $\underline{A}$.)

Type-1 describes the notion of optimality most commonly used in the literature, and it is sometimes called "optimal up to a choice of a tie-breaking-rule". Note that these four definitions are applicable to any class of algorithms, $\underline{B}$, contending to be optimal over $\underline{A}$; we need only replace the words "tie-breaking-rule in A*" by the words "member of $\underline{B}$". If $\underline{B}$ turns out to be a singleton, then type-0 and type-1 collapse to strong optimality. Type-3 will collapse into type-2 if we insist that I_1 be identical to I_2.

We are now ready to introduce the four domains of problem instances over which the optimality of A* is to be examined. The first two relate to the admissibility and consistency of $h(n)$.

Definition: A heuristic function $h(n)$ is said to be **admissible** on (G,Γ) iff $h(n) \leq h^*(n)$ for every $n \in G$

Definition: A heuristic function $h(n)$ is said to be **consistent** (or monotone) on G iff for any pair of nodes, n' and n, the triangle inequality holds:

$$h(n') \leq k(n',n) + h(n) \tag{38}$$

Corresponding to these two properties we define the following sets of problem instances:

$$\underline{I}_{AD} = \left\{ (G,s,\Gamma,h) \mid h \leq h^* \text{ on } (G,\Gamma) \right\} \tag{39}$$

$$\underline{I}_{CON} = \left\{(G,s,\Gamma,h) \mid h \text{ is consistent on } G\right\} \tag{40}$$

Clearly, consistency implies admissibility [Pearl 1984] but not vice versa, therefore, $\underline{I}_{CON} \subseteq I_{AD}$

A special and important subset of $\underline{I}_{AD}$ (and $\underline{I}_{CON}$), called **non-pathological instances**, are those instances for which there exists at least one optimal solution path along which h is not fully informed, that is, $h<h^*$ for every non-goal node on that path. The non-pathological subsets of $\underline{I}_{AD}$ and $\underline{I}_{CON}$ will be denoted by $\bar{I}_{AD}$ and $\bar{I}_{CON}$, respectively.

It is known that if $h \le h^*$, then A* expands every node reachable from s by a strictly C^*-bounded path, regardless of the tie-breaking rule used. The set of nodes with this property will be referred to as **surely expanded** by A*. In general, for an arbitrary constant d and an arbitrary evaluation function f over (G, s, Γ, h), we let $\underline{N}_f^d$ denote the **set of all nodes reachable from s by some strictly d-bounded path** in G. For example, $\underline{N}_{g+h}^{C^*}$ is a set of nodes surely expanded by A* in some instance of $\underline{I}_{AD}$.

The importance of non-pathological instances lies in the fact that in such instances the set of nodes surely expanded by A* are indeed **all** the nodes expanded by it. Therefore for these instances any claim regarding the set of nodes surely expanded by A* can be translated to "the set of all the nodes" expanded by A*. This is not the case, however, for pathological instances in $\underline{I}_{AD}$; $\underline{N}_{g+h}^{C^*}$ is often a proper subset of the set of nodes actually expanded by A*. If h is consistent, then the two sets differ only by nodes for which $h(n) = C^* - g^*(n)$ [Pearl 1984]. However, in cases where h is inconsistent, the difference may be very substantial; each node n for

which $h(n) = C^* - g^*(n)$ may have many descendents assigned lower h values (satisfying $h+g<C^*$) and these descendents may be expanded by every tie-breaking-rule of A* even though they do not belong to $\underline{N}_{g+h}^{C^*}$.

In the following subsection we present several theorems regarding the behavior of competing classes of algorithms relative to the set $\underline{N}_{g+h}^{C^*}$ of nodes surely expanded by A*, and will interpret these theorems as claims about the type of optimality that A* enjoys over the competing classes. Moreover, for any given pair $(\underline{A}, \underline{I})$ where $\underline{A}$ is a class of algorithms drawn from $\left\{ \underline{A}_{ad}, \underline{A}_{bf}, \underline{A}_{gc} \right\}$ and $\underline{I}$ is a domain of problem instances from $\left\{ \underline{I}_{AD}, \underline{I}_{\bar{AD}}, \underline{I}_{CON}, \underline{I}_{\bar{CON}} \right\}$, we will determine the strongest type of optimality that can be established over $\underline{A}$ relative to $\underline{I}$, and will identify the algorithm that achieves this optimality. The relationships between these classes of algorithms and problem domains are shown in Figure 2.4. The algorithm A** is an improvement over A* discussed in Appendix 2.2.

2.4.3 Where and How is A* Optimal?

2.4.3.1 Optimality over admissible algorithms, $\underline{A}_{ad}$

Theorem 8:

Any algorithm that is admissible on $\underline{I}_{AD}$ will expand, in every instance $I \in \underline{I}_{CON}$, all nodes surely expanded by A*.

Proof:

Let $I=(G,s,\Gamma,h)$ be some problem instance in $\underline{I}_{AD}$ and assume that n is surely expanded by A*, i.e., $n \in \underline{N}_{g+h}^{C^*}$. Therefore, there exists a path P_{s-n} such that

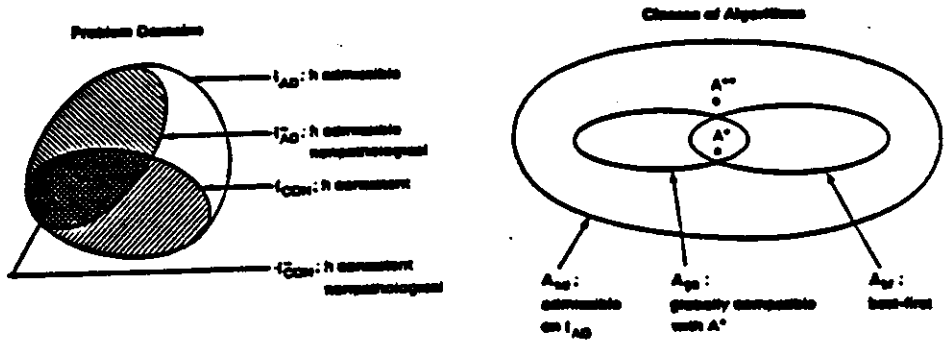


Figure 2.4 - The classes of algorithm and the problem instances for which the optimality of A* is examined.

$$\forall n' \in P_{s-n} \ , \ g(n') + h(n') < C^* \tag{41}$$

Let B be an algorithm compatible with A*, namely halting with cost C^* in I.

Assume that B does not expand n. We now create a new graph G' (see figure 2.5) by adding to G a goal node t with $h(t)=0$ and an edge from n to t with non-negative cost $C=h(n)+\Delta$, where

$$\Delta = 1/2(C^* - D) > 0 \tag{42}$$

and

$$D = \max\left\{ f(n') \mid n' \in \underline{N}^{C^*}_{g+h} \right\} \tag{43}$$

This construction creates a new solution path P^* with cost at most $C^*-\Delta$ and, simultaneously, (due to h's consistency on I) retains the consistency (and admissibility) of h on the new instance I'. To establish the consistency of h in I' we note that since

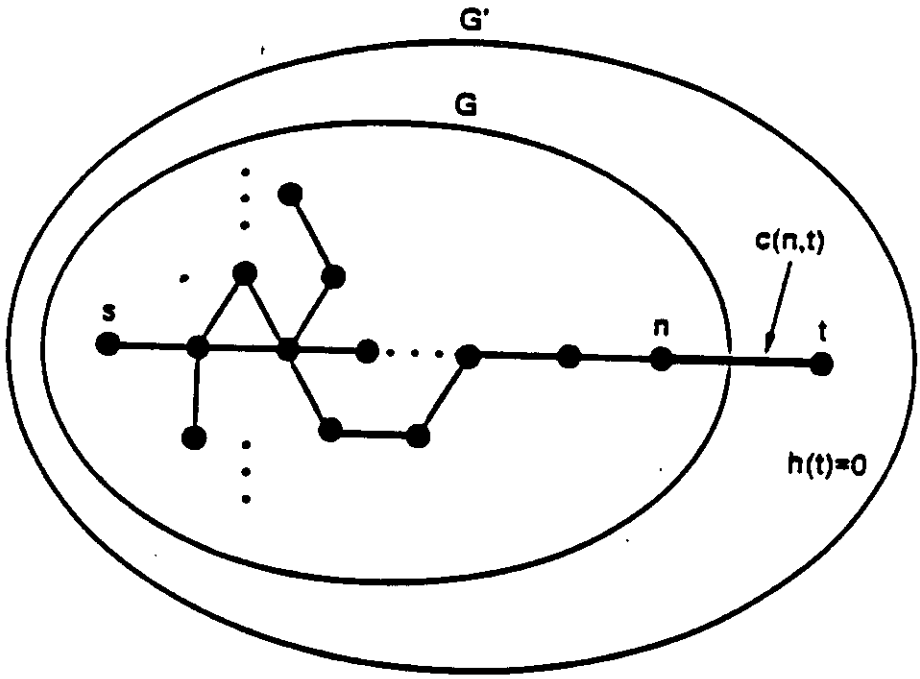


Figure 2.5 - Used in the proof of theorem 8.

we kept the h values of all nodes in G unchanged, consistency will continue to hold between any pair of nodes previously in G. It remains to verify consistency on pairs involving the new goal node t, which amounts to establishing the inequality $h(n') \leq k(n',t)$ for every node n' in G. Now, if at some node n' we have $h(n') > k(n',t)$ then we should also have:

$$h(n') > k(n',n) + c = k(n',n) + h(n) + \Delta \tag{44}$$

in violation of h's consistency on I. Thus, the new instance is also in $\underline{I}_{CON}$.

In searching G', algorithm A* will find the extended path P^* costing C^*-Δ, because:

$$f(t) = g(n) + c = f(n) + \Delta \leq D + \Delta = C^* - \Delta < C^* \tag{45}$$

and so, t is reachable from s by a path bounded by C^*-Δ which ensures its selection. Algorithm B, on the other hand, if it avoids expanding n, must behave the same as in

problem instance I, halting with cost C^* which is higher than that found by A*. This contradicts the supposition that B is both admissible on I and avoids the expansion of node n.

□

The implications of Theorem 8 relative to the optimality of A* are rather strong. In non-pathological cases $I \in \underline{I}_{\overline{CON}}$ A* never expands a node outside $\underline{N}_{g+h}^{C^*}$ and, therefore, Theorem 8 establishes the 0-optimality of A* over all admissible algorithms relative to $\underline{I}_{\overline{CON}}$. In pathological cases of $\underline{I}_{CON}$ there may also be nodes satisfying $f(n)=C^*$ that some tie-breaking-rule in A* expands and, since these nodes are defined to be outside $\underline{N}_{g+h}^{C^*}$, they may be avoided by some algorithm $B \in \underline{A}_{ad}$, thus destroying the 0-optimality of A* relative to all $\underline{I}_{CON}$. However, since there is always a tie-breaking-rule in A* which, in addition to $\underline{N}_{g+h}^{C^*}$, expands only nodes along one optimal path, Theorem 8 also establishes the 1-optimality of A* relative the entire $\underline{I}_{CON}$ domain. Stronger yet, the only nodes that A* expands outside $\underline{N}_{g+h}^{C^*}$ are those satisfying $f(n) = C^*$, and since this equality is not likely to occur in many nodes of the graph, we may interpret Theorem 8 to endow A* with "almost" 0-optimality (over all admissible algorithms) relative to $\underline{I}_{CON}$.

The proof of Theorem 8 makes it tempting to conjecture that A* retains the same type of optimality relative to cases where h is admissible but not necessarily consistent. In fact, the original argument of Hart, Nilsson and Raphael [Hart 1968] that no admissible algorithm equally informed to A* can ever avoid a node expanded by A* (see Section 2.4.1), amounts to claiming that A* is at least 1-optimal relative to $\underline{I}_{AD}$. Similar claims are made by Mero [Mero 1984] and are suggested by the theorems of [Gelperin 1977].

Unfortunately, Theorem 8 does not lend itself to such extension; if h is admissible but not consistent, then after adding the extra goal node t to G (as in Figure 2.5) we can no longer guarantee that h will remain admissible on the new instance created. Furthermore, we can actually construct an algorithm that is admissible on $\underline{I}_{AD}$ and yet, in some problem instances, it will grossly outperform every tie-breaking-rule in A*. Consider an algorithm B guided by the following search policy: Conduct an exhaustive right-to-left depth-first search but refrain from expanding one distinguished node n, e.g., the leftmost son of s. By the time this search is completed, examine n to see if it has the potential of sprouting a solution path cheaper than all those discovered so far. If it has, expand it and continue the search exhaustively. Otherwise,[3] return the cheapest solution at hand. B is clearly admissible; it cannot miss an optimal path because it would only avoid expanding n when it has sufficient information to justify this action, but otherwise will leave no stone unturned. Yet, in the graph of Figure 2.6a, B will avoid expanding many nodes which are surely expanded by A*. A* will expand node J_1 immediately after s ($f(J_1)$=4) and subsequently will also expand many nodes in the subtree rooted at J_1. B, on the other hand, will expand J_3, then select for expansion the goal node γ, continue to expand J_2 and at this point will halt without expanding node J_1. Relying on the admissibility of h, B can infer that the estimate $h(J_1)$=1 is overly optimistic and should be at least equal to $h(J_2)-1$=19, thus precluding J_1 from lying on a solution path cheaper than the path (s, J_3, γ) at hand.

Granted that A* is not 1-optimal over all admissible algorithms relative to $\underline{I}_{AD}$, the question arises if a 1-optimal algorithm exists altogether. Clearly, if a 1-

[3] A simple valid test for skipping a node in $\underline{I}_{AD}$ is that $\max\{g(n') + h(n') \mid n'\}$ on some path P from s to n be larger than the cost of the cheapest solution at hand.

optimal algorithm exists, it would have to be better than A* in the sense of skipping in some problem instances, at least one node surely expanded by A* while never expanding a node which is surely skipped by A*. Note that algorithm B above could not be such an optimal algorithm because in return for skipping node J_1 in Figure 2.6a, it had to pay the price of expanding J_2, and J_2 will not be expanded by A* regardless of the tie-breaking-rule invoked. If we could show that this "node trade-off" pattern must hold for every admissible algorithm and on every instance of $\underline{I}_{AD}$, then we would have to conclude both that no 1-optimal algorithm exists and that A* is 2-optimal relative to this domain. Theorem 9 accomplishes this task relative to $\underline{I}_{\bar{A}D}$.

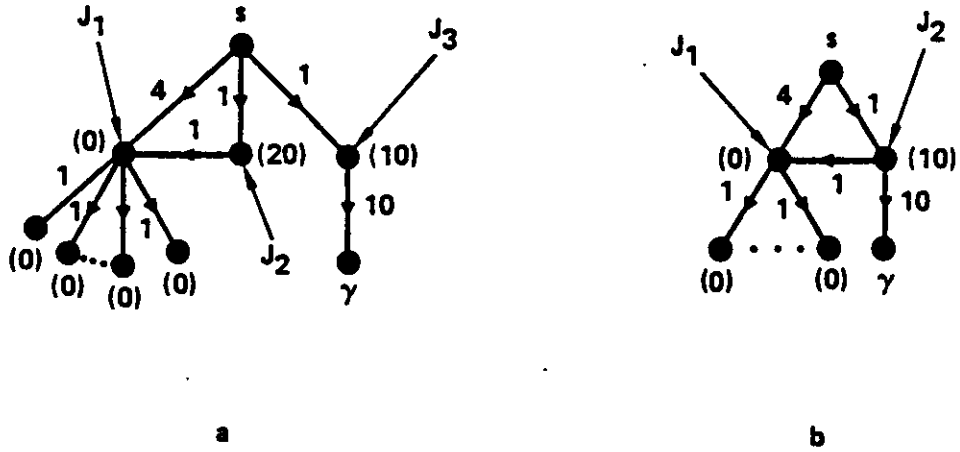


Figure 2.6 - Graphs demonstrating that A* is not optimal.

Theorem 9:

If an admissible algorithm B does not expand a node which is surely expanded by A*

in some problem instance where h is admissible and non-pathological, then in that very problem instance B must expand a node which is avoided by every tie-breaking-rule in A*.

Proof:

Assume the contrary, i.e., there is an instance $I=(G,s,\Gamma,h) \in \underline{I}_{\overline{AD}}$ such that a node n which is surely expanded by A* is avoided by B and, at the same time, B expands no node which is avoided by A*, we shall show that this assumption implies the existence of another instance $I' \in \underline{I}_{AD}$ where B will not find an optimal solution. I' is constructed by taking the graph G_e exposed by the run of A* (including nodes in OPEN) and appending to it another edge (n,t) to a new goal node t, with cost $c(n,t)=D-k_e(s,n)$ where

$$D = \max\left\{f(n') \mid n' \in \underline{N}^{C^*}_{g+h}\right\} \tag{46}$$

and $k_e(n',n)$ is the cost of the cheapest path from n' to n in G_e.

Since G contains an optimal path $P^*_{s-\gamma}$ along which $h(n')<h^*(n')$ (with the exception of γ and possibly s), we know that because ties are broken in favor of goal nodes A* will halt without ever expanding a node having $f(n)=C^*$. Therefore, every nonterminal node in G_e must satisfy the strict inequality $g(n)+h(n)<C^*$.

We shall first prove that I' is in $\underline{I}_{AD}$, i.e., that $h(n') \le h^{*'}(n')$ for every node n' in G_e. This inequality certainly holds for n' such that $g(n')+h(n') \ge C^*$ because all such nodes were left unexpanded by A* and hence appear as terminal nodes in G_e for which $h^{*'}(n')=\infty$ (with the exception of γ, for which $h(\gamma)=h^{*'}(\gamma)=0$). It remains, therefore, to verify the inequality for nodes n' in $\underline{N}^{C^*}_{g+h}$ for which we have $g(n')+h(n') \le D$. Assume the contrary, that for some $n' \in \underline{N}^{C^*}_{g+h}$ we have

$h(n') > h^*_{I'}(n')$. This implies

$$\begin{aligned} h(n') &> k_e(n',n) + c(n,t) \\ &= k_e(n',n) + D - k_e(s,n) \\ &\geq k_e(n',n) + k_e(s,n') + h(n') - k_e(s,n) \end{aligned} \tag{47}$$

or

$$k_e(s,n) > k_e(n',n) + k_e(s,n') \tag{48}$$

in violation of the triangle inequality for cheapest paths in G_e. Hence, I' is in $\underline{\mathbf{I}}_{AD}$.

Assume now that algorithm B does not generate any node outside G_e. If B has avoided expanding n in I, it should also avoid expanding n in I'; all decisions must be the same in both cases since the sequence of nodes generated (including those in OPEN) is the same. On the other hand, the cheapest path in I' now goes from s to n to t', having the cost $D < C^*$, and will be missed by B. This violates the admissibility of B on an instance in $\underline{\mathbf{I}}_{AD}$ and proves that B could not possibly avoid the expansion of n without generating at least one node outside G_e. Hence, B must expand at least one node avoided by A* in this specific run.

□

Theorem 9 has two implications. On one hand it conveys the discomforting fact that neither A* nor any other algorithm is 1-optimal over those guaranteed to find an optimal solution when given $h \leq h^*$. On the other hand, Theorem 9 endows A* with some optimality property, albeit weaker than hoped; the only way to gain one node from A* is to relinquish another. Not every algorithm enjoys such strength. These implications are summarized in the following Corollary.

Corollary 2: No algorithm can be 1-optimal over all admissible algorithms relative to $\underline{\mathbf{I}}_{AD}$, but A* is 2-optimal over this class relative to $\underline{\mathbf{I}}_{AD}$.

The fact that Theorem 9 had to be limited to non-pathological instances is explained by Figure 2.6b, showing an exception to the node-tradeoff rule on a pathological instance. Algorithm B does not expand a node (J_1) which must be expanded by A* and yet, B does not expand any node which A* may skip. This example implies that A* is not 2-optimal relative to the entire $\underline{I}_{AD}$ domain and, again, this begs the questions whether there exists of a 2-optimal algorithm altogether, and whether A* is at least 3-optimal relative to this domain.

The answer to both questions is, again, negative; another algorithm that we shall call A**, turns out both, to strictly dominate A* and to meet the requirements for type-3 optimality relative to $\underline{I}_{AD}$. A** conducts the search in a manner similar to A*, with one exception; instead of $f(n)=g(n)+h(n)$, A** uses the evaluation function:

$$f(n) = \max \left\{ g(n') + h(n') \mid n' \text{ on the current path to } n \right\} \qquad (49)$$

This, in effect, is equivalent to raising the value of $h(n)$ to a level where it becomes consistent with the h's assigned to the ancestors of n. [Mero 1984] A** chooses for expansion the nodes with the lowest f value in OPEN (breaking ties arbitrarily but in favor of goal nodes) and adjusts pointers along the path having the lowest g value. In figure 2.6a, for example, if A** ever expands node J_2 then its son J_1 will immediately be assigned the value $f(J_1) = 21$ and its pointer be directed toward J_2.

It is possible to show (see Appendix 2.2) that A** is admissible and that in non-pathological cases A** expands the same set of nodes as does A*, namely the surely expanded nodes in $\underline{N}^{C^*}_{g+h}$. In pathological cases, however, there exist tie-breaking-rules in A** that strictly dominate every tie-breaking-rule in A*. This immediately precludes A* from being 3-optimal relative to $\underline{I}_{AD}$ and nominates A**

for that title.

Theorem 10:

Let a** be some tie-breaking-rule in A** and B an arbitrary algorithm, admissible relative to $\underline{I}_{AD}$. If in some problem instance $I_1 \in \underline{I}_{AD}$, B skips a node expanded by a** then there exists another instance $I_2 \in \underline{I}_{AD}$ where B expands a node skipped by a**.

Proof:

Let

$$S_A = n_1, n_2, \ldots n_k, J \cdots \tag{50}$$

and

$$S_B = n_1, n_2, \ldots n_k, K \cdots \tag{51}$$

be the sequences of nodes expanded by a** and B, respectively, in problem instance $I_1 \in \underline{I}_{AD}$, i.e., K is the first node in which the sequence S_B deviates from S_A. Consider G_e, the explored portion of the graph just before a** expands node J. That same graph is also exposed by B before it decides to expand K instead. Now construct a new problem instance I_2 consisting of G_e appended by a branch (J,t) with cost $C(J,t) = f(J) - g(J)$, where t is a goal node and $f(J)$ and $g(J)$ are the values that a** computed for J before its expansion. I_2 is also in $\underline{I}_{AD}$ because $h(t)=0$ and $C(J,t)$ are consistent with $h(J)$ and with the h's of all ancestors of J in G_e. For if some ancestor n_i of J satisfies $h(n_i) > h^*(n_i)$ we will obtain a contradiction:

$$\begin{aligned} g(n_i) + h(n_i) &> g(n_i) + h^*(n_i) \\ &= g(n_i) + k_e(n_i, J) + c(J,t) \\ &\geq g(J) + c(J,t) \end{aligned} \tag{52}$$

$$= f(J) \equiv \max_j [g(n_j) + h(n_j)]$$

Moreover, a** will expand in I_2 the same sequence of nodes as it did in I_1, until J is expanded, at which time t enters OPEN with $f(t) = \max [g(J) + c(J,t), f(J)] = f(J)$. Now, since J was chosen for expansion by virtue of its lowest f value in OPEN, and since a** always breaks up ties in favor of a goal node, the next and final node that a** expands must be t. Now consider B. The sequence of nodes it expands in I_2 is identical to that traced in I_1 because, by avoiding node J, B has no way of knowing that a goal node has been appended to G_e. Thus, B will expand K (and perhaps more nodes on OPEN), a node skipped by a**.

□

Note that the special evaluation function used by A** $f(n) = \max \left\{ g(n') + h(n') \mid n' \text{ on } P_{s-n} \right\}$ was necessary to ensure that the new instance, I_2, remains in $\mathbf{I}_{AD}$. The proof cannot be carried out for A* because the evaluation function $f(n)=g(n)+h(n)$ results in $C(J,t) = h(J)$, which may lead to violation of $h(n_i) \leq h^*(n_i)$ for some ancestor of J.

Theorem 10, together with the fact that its proof makes no use of the assumption that B is admissible, gives rise to the following conclusion:

Corollary 3: A** is 3-optimal over all algorithms relative to $\underline{\mathbf{I}}_{AD}$.

Theorem 10 also implies that there is no 2-optimal algorithm over $\underline{\mathbf{A}}_{ad}$ relative to $\underline{\mathbf{I}}_{AD}$. From the 3-optimality of A** we conclude that every 2-optimal algorithm, if such exists, must be a member of the A** family, but figure 6b demonstrates an instance of $\underline{\mathbf{I}}_{AD}$ where another algorithm (B) only expands a proper subset of the nodes expanded by every member of A**. This establishes the desired conclu-

sion:

Corollary 4: There is no 2-optimal algorithm over $\underline{A}_{ad}$ relative to $\underline{I}_{AD}$

2.4.3.2 Optimality over globally compatible algorithms, $\underline{A}_{gc}$

So far our analysis was restricted to algorithms in $\underline{A}_{ad}$, i.e., those which return optimal solutions if $h(n) \leq h^*(n)$ for all n, but which may return arbitrarily poor solutions if there are some n for which $h(n) > h^*(n)$. In situations where the solution costs are crucial and where h may occasionally overestimate h^* it is important to limit the choice of algorithms to those which return reasonably good solutions even when $h > h^*$. A*, for example, provides such guarantees; the costs of the solutions returned by A* do not exceed $C^* + \Delta$ where Δ is the highest error $h^*(n) - h(n)$ over all nodes in the graph [Harris 1974] and, moreover, A* still returns optimal solutions in many problem instances, i.e., whenever Δ is zero along some optimal path. This motivates the definition of $\underline{A}_{gc}$, the class of algorithms globally compatible with A*, namely, they return optimal solutions in every problem instance where A* returns such solution.

Since $\underline{A}_{gc}$ is a subset of $\underline{A}_{ad}$, we should expect A* to hold a stronger optimality status over $\underline{A}_{gc}$, at least relative to instances drawn from $\underline{I}_{AD}$. The following Theorem confirms this expectation.

Theorem 11:

Any algorithm that is globally compatible with A* will expand, in problem instances where h is admissible, all nodes surely expanded by A*.

Proof:

Let $I = (G, s, \Gamma, h)$ be some problem instance in $\underline{\mathbf{I}}_{AD}$ and let node n be surely expanded by A*, i.e., there exists a path P_{s-n} such that

$$g(n) + h(n) < C^* \quad \text{for all } n' \in P_{s-n} \tag{53}$$

Let $D = \max \left\{ f(n') \mid n' \in P_{s-n} \right\}$ and assume that some algorithm $B \in \underline{\mathbf{A}}_{gc}$ fails to expand n. Since $I \in \underline{\mathbf{I}}_{AD}$, both A* and B will return cost C^*, while $D < C^*$.

We now create a new graph G', as in figure 2.5, by adding to G a goal node t' with $h(t') = 0$ and an edge from n to t' with non-negative cost $D - g(P_{s-n})$. Denote the extended path $P_{s-n-t'}$ by P^*, and let $I' = (G', s, \Gamma \cup t', h)$ be a new instance in the algorithms' domain. Although h may no longer be admissible on I', the construction of I' guarantees that $f(n') \leq D$ if $n' \in P^*$, and thus, by Theorem 2, algorithm A* searching G' will find an optimal solution path with cost $C_t \leq M \leq D$. Algorithm B, however, will search I' in exactly the same way it searched I; the only way B can reveal any difference between I and I' is by expanding n. Since it did not, it will not find solution path P^*, but will halt with cost $C^* > D$, the same cost it found for I and worse than that found by A*. This contradicts its property of being globally compatible with A*.

□

Corollary 5:

A* is 0-optimal over $\underline{\mathbf{A}}_{gc}$ relative to $\underline{\mathbf{I}}_{AD}^{-}$.

The corollary follows from the fact that in non-pathological instances A* expands only surely expanded nodes.

Corollary 6:

A* is 1-optimal over $\underline{\mathbf{A}}_{gc}$ relative to $\underline{\mathbf{I}}_{AD}$.

Proof:

The proof relies on the observation that for every optimal path P^* (in any instance of $\underline{I}_{AD}$) there is a tie-breaking-rule of A* that expands only nodes along P^* plus perhaps some other nodes having $g(n)+h(n)<C^*$, i.e., the only nodes expanded satisfying the equality $g(n)+h(n)=C^*$ are those on P^*. Now, if A* is not 1-optimal over A_{gc} then, given an instance I, there exists an algorithm $B \in A_{gc}$ such that B avoids some node expanded by all tie-breaking-rules in A*. To contradict this supposition let A_1^* be the tie-breaking-rule of A* that returns the same optimal path P^* as B returns, but expands no node outside P^* for which $g(n)+h(n)=C^*$. Clearly, any node n which B avoids and A_1^* expands must satisfy $g(n)+h(n)<C^*$. We can now apply the argument used in the proof of Theorem 11, appending to n a branch to a goal node t', with cost $c(n,t')=h(n)$. Clearly, A_1^* will find the optimal path (s,n,t') costing $g(n)+h(n)<C^*$, while B will find the old path costing C^*, thus violating its global compatibility with A*.

□

2.4.3.3 Optimality over best-first algorithms, A_{bf}

The next result establishes A*'s optimality over the set of best-first algorithms (BF*) which are admissible if provided with $h \le h^*$. These algorithms will be permitted to employ any evaluation function f_P where f is a function of the nodes, the edge-costs, and the heuristic function h evaluated on the nodes of P, i.e.

$$f_P \triangleq f(s,n_1,n_2,\ldots,n) = f(\{n_i\},\{c(n_i,n_{i+1})\},\{h(n_i)\} \mid n_i \in P). \qquad (54)$$

Lemma 6:

Let B be a BF* algorithm using an evaluation function f_B such that for every $(G,s,\Gamma,h) \in \underline{I}_{AD}$ f_B satisfies:

$$f_{P_i^\gamma}=f(s,n_1,n_2,\ldots,\gamma)=C(P_i^\gamma)\quad \forall\gamma\in\Gamma. \tag{55}$$

If B is admissible on $\underline{I}_{AD}$, then $\underline{N}_{g+h}^{C^*}\subseteq\underline{N}_{f_B}^{C^*}$.

□

Proof:

Let $I=(G,s,\Gamma,h)\in\underline{I}_{AD}$ and assume $n\in\underline{N}_{g+h}^{C^*}$ but $n\notin\underline{N}_{f_B}^{C^*}$, i.e., there exists a path P_{s-n} such that for every $n'\in P_{s-n}$, $g_P(n')+h(n')<C^*$ and for some $n'\in P_{s-n}$ $f_B(n')\geq C^*$.

Let

$$Q=\max_{n'\in P_{s-n}}\left\{g(n')+h(n')\right\} \tag{56}$$

$$Q_B=\max_{n'\in P_{s-n}}\left\{f_B(n')\right\} \tag{57}$$

Obviously: $Q<C^*$ and $Q_B\geq C^*\rightarrow Q_B>Q$. Define G' to include path P_{s-n} with two additional goal nodes t_1,t_2 as described by figure 2.7. The cost on edge (s,t_2) is $\frac{Q_B+Q}{2}$, the cost on edge (n,t_1) is $Q-g_{P_{s-n}}(n)$, t_1 and t_2 are assigned $h'=0$ while all other nodes retain their old h. $I'=(G',s,\Gamma\cup\{t_1,t_2\},h)\in\underline{I}_{AD}$ since $\forall n'$, $n'\in P_{s-n}$, $g(n')+h(n')\leq Q$ which implies that $h(n')\leq Q-g(n')=h^*_{I'}(n')$.

Obviously the optimal path in G' is P_{s-t_1} with cost Q. However, following the condition of the Lemma, the evaluation function f_B satisfies

$$M_{P_{s-t_1}}=f_B(t_2)=C(P_{s-t_2})=\frac{Q_B+Q}{2}<Q_B \tag{58}$$

Moreover, since $M_{P_{s-t_1}}\geq Q_B$, we have $M_{P_{s-t_1}}<M_{P_{s-t_1}}$, implying that B halts on the suboptimal path P_{s-t_2}, thus contradicting its admissibility.

□

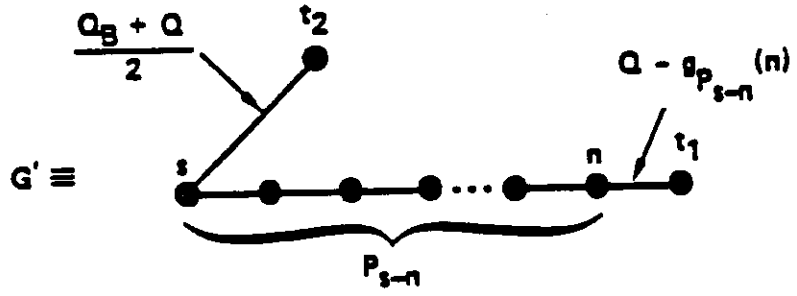


Figure 2.7 - Used in the proof of lemma 6.

Theorem 12:

Let B be a BF* algorithm such that f_B satisfies the property of Lemma 6.

a. If B is admissible over $\underline{I}_{AD}$ then B expands every node in $\underline{N}_{g+h}^{C^*}$.

b. If B is admissible over $\underline{I}_{AD}$ and f_B is of the form:

$$f_{P_{s\to n}}(n) = F(g_{P_{s\to n}}(n), h(n)) \tag{59}$$

then $F(x,y) \le x+y$.

□

Proof:

a. Let M be the min-max value corresponding to the evaluation function f_B. It is easy to see that $M \ge C^*$. From that and from Theorem 11 we get

$$\underline{N}_{g+h}^{C^*} \le \underline{N}_{f_B}^{C^*} \le \underline{N}_{f_B}^{M} \tag{60}$$

and it is implied by Theorem 5 that any node in $\underline{N}_{f_B}^{M}$ is expanded by B.

b. Assume to the contrary that there is a path P and a node $n \in P$ such that

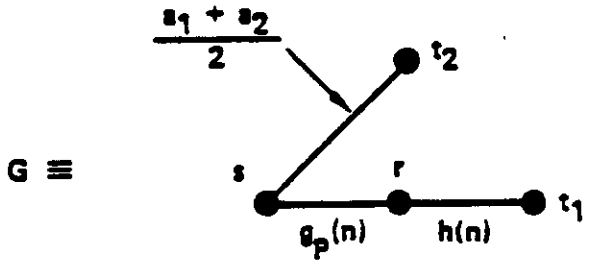


Figure 2.8 - Used in the proof of theorem 12

$$F(g_P(n),h(n)) > g_P(n)+h(n) \tag{61}$$

Let $a_1=F(g_P(n),h(n))$, $a_2=g_P(n)+h(n)$. Obviously, $a_2 < \frac{a_1+a_2}{2} < a_1$. Let G be a graph as shown in Figure 2.8 having nodes s,r,t_1,t_2 and edges (s,r), (r,t_1), (s,t_2) with costs $C(s,r)=g_P(n)$, $C(r,t_1)=h(n)$, $C(s,t_2)=\frac{a_1+a_2}{2}$. Let $I=(G,s,\{t_1,t_2\},h)$ where $h(s)=0$, $h(r)=h(n)$ and $h(t_1)=h(t_2)=0$. Obviously $I \in \underline{I}_{AD}$. However,

$$f(r) = F(g_P(n),h(n)) = a_1 > \frac{a_1+a_2}{2} = c(s,t_2) = f(t_2) \tag{62}$$

implying that B halts on solution path P_{s-t_2}, again contradicting its admissibility.

□

Corollary 7:

A* is 0-optimal over $\underline{A}_{bf}$ relative to $\underline{I}_{\bar{A}D}$ and 1-optimal relative to $\underline{I}_{CON}$. A** is 1-optimal over $\underline{A}_{bf}$ relative to $\underline{I}_{AD}$.

Note that A** is not a member of $\underline{A}_{bf}$; it directs pointers toward the lowest-g rather than the lowest f path as instructed by our definition of BF* (Section 2.1).

□

An interesting implication of Part b of Theorem 12 asserts that any admissible combination of g and h, $h \leq h^*$, will expand every node surely expanded by A*. In other words, the additive combination $g+h$ is, in this sense, the optimal way of aggregating g and h for additive cost measures.

The 0-optimality of A* relative to nonpathological instances of $\underline{I}_{AD}$ also implies that in these instances $g(n)$ constitutes a sufficient summary of the information gathered along the path from s to n. Any additional information regarding the heuristics assigned to the ancestors of n, or the costs of the individual arcs along the path, is superfluous, and cannot yield a further reduction in the number of nodes expanded. Such information, however, may help reduce the number of node evaluations performed by the search (see [Martelli 1977, Bagchi 1983] and [Mero 1984].).

□

2.4.4 Summary and Discussion

Our results concerning the optimality of A* are summarized in Table 1. For each class-domain combination from Figure 2.4, the table identifies the strongest type of optimality that exists and the algorithm achieving it.

The most significant results are those represented in the left-most column, relating to $\underline{A}_{ad}$, the entire class of algorithms which are admissible whenever provided with optimistic advice. Contrary to prevailing beliefs A* turns out not to be optimal over $\underline{A}_{ad}$ relative to every problem graph quantified with optimistic estimates. There are admissible algorithms which, in some graphs, will find the optimal

Domain of Problem Instances		Class of Algorithms: Admissible if $h \leq h^*$ (A_{ad})	Globally Compatible with A^* (A_{gc})	Best-First (A_{bf})
	Admissible I_{AD}	A^{**} is 3-optimal No 2-optimal exists	A^* is 1-optimal No 0-optimal exists	A^* is 1-optimal No 0-optimal exists
	Admissible and nonpathological I^{-}_{AD}	A^* is 2-optimal No 1-optimal exists	A^* is 0-optimal	A^* is 0-optimal
	Consistent I_{CON}	A^* is 1-optimal No 0-optimal exists	A^* is 1-optimal No 0-optimal exists	A^* is 1-optimal No 0-optimal exists
	Consistent nonpathological I^{-}_{CON}	A^* is 0-optimal	A^* is 0-optimal	A^* is 0-optimal

TABLE 1

solution in just a few steps whereas A* (as well as A** and all their variations) would be forced to explore arbitrary large regions of the search graphs (see Fig. 2.6a). In bold defiance of Hart, Nilsson, and Raphael's [Hart 1968] argument for A*'s optimality, these algorithms succeed in outsmarting A* by penetrating regions of the graph that A* finds unpromising (at least temporarily), visiting some goal nodes there, then processing the information gathered to identify and purge those nodes on OPEN which no longer promise to sprout a solution better than the cheapest one at hand.

In nonpathological cases, however, these algorithms cannot outsmart A* without paying a price. The 2-optimality of A* relative to $\underline{I}_{AD}$ means that each such algorithm must always expand at least one node which A* will skip. This means that the only regions of the graph capable of providing node-purging information are regions which A* will not visit at all. In other words, A* makes full use of the information gathered along its search and there could be no gain in changing the order of visiting nodes which A* plans to visit anyhow.

This instance-by-instance node tradeoff no longer holds when pathological cases are introduced. The fact that A* is not 2-optimal relative to $\underline{I}_{AD}$ means that some smart algorithms may outperform A* by simply penetrating certain regions of the graph earlier than A* (A* will later visit these regions), thus expanding only a proper subset of the set of nodes expanded by A*. In fact the lack of 2-optimality in the $(\underline{A}_{ad}, \underline{I}_{AD})$ entry of table 1 means that no algorithm can be protected against such smart competitors; For any admissible algorithm A_1, there exists another admissible algorithm A_2 and a graph G quantified by optimistic heuristic h ($h \leq h^*$) such that A_2 expands fewer nodes than A_1 when applied to G. Mero [Mero 1984] has

recently shown that no optimal algorithm exists if complexity is measured by the number of expansion operations (a node can be reopened several times). Our result now shows that $\underline{A}_{ad}$ remains devoid of an optimal algorithm even if we measure complexity by the number of distinct nodes expanded.

The type-3 optimality of A** over $\underline{A}_{ad}$ further demonstrates that those 'smart' algorithms which prevent A* from achieving optimality are not smart after all, but simply lucky; each takes advantage of the peculiarity of the graph for which it was contrived and none can maintain this superiority over all problem instances. If it wins on one graph there must be another where it is beaten, and by the very same opponent, A**. It is in this sense that A** is 3-optimal, it exhibits a universal robustness against all its challengers.

Perhaps the strongest claim that Table 1 makes in favor of A* is contained in the entries related to $\underline{I}_{CON}$, the domain of problems in which h is known to be not only admissible, but also consistent. It is this domain that enables A* to unleash its full prunning powers, achieving a node-by-node superiority (types 0 and 1) over all admissible algorithms. Recalling also that, under consistent h, A* never reopens closed nodes and that only few nodes are affected by the choice of tie-breaking-rule (see [Pearl 1984]), we conclude that in this domain A* constitutes the most effective scheme of utilizing the advice provided by h.

This optimality is especially significant in light of the fact that consistency is not an exception but rather a common occurrence; almost all admissible heuristics invented by people are consistent. The reason is that the technique people invoke in generating heuristic estimates is often that of **relaxation**; we imagine a simplified version of the problem at hand by relaxing some of its constraints, solve the relaxed

version mentally, then we use the cost of the resulting solution as a heuristic for the original problem [Pearl 1983] It can be shown that any heuristic generated by such a process is automatically consistent, which explains the abandon of consistent cases among human-generated heuristics. Thus, the strong optimality of A* under the guidance of consistent heuristics implies, in effect, its optimality in most cases of practical interest.

APPENDIX 2.1: Finding an ϵ-optimal path in a Tree with Random Costs

Let $C^*(N,p)$ be the optimal cost of a root-to-leaf path in a uniform binary tree of height N where each branch independently has a cost of 1 or 0 with probability p and 1-p, respectively. We wish to prove that for $p > 1/2$

$$P[M \geq (1+\epsilon)C^*(N,p)] \rightarrow 0 \qquad \epsilon > 0$$

where

$$M = \min_j \max_{n \in P_j} \{f(n)\},$$

$$f(n) = g(n) + \alpha^*[N - d(n)],$$

and where α^* is defined by the equation

$$\left(\frac{p}{\alpha^*}\right)^{\alpha^*} \left(\frac{1-p}{1-\alpha^*}\right)^{1-\alpha^*} = 1/2$$

Call a path P (α, L)-regular if the cost of every successive path segment of length L along P is at most αL. Karp and Pearl [Karp 1983] have shown that:

a. $P[C^*(N,p) \leq \alpha N] \rightarrow 0$ for $\alpha < \alpha^*$.

b. If $\alpha < \alpha^*$, one can always find a constant L_α such that the probability that there exists an (α, L_α)-regular path of length N is bounded away from zero. Call this probability $1 - q_\alpha$, $(q_\alpha < 1)$.

Consider the profile of f along a (α, L)-regular path P sprouting from level d_o below the root. Along such a path $g(n)$ satisfies:

$$g(n) \leq d_o + [d(n) - d_o]\alpha + \alpha L$$

and, consequently,

$$f(n) \leq (1-\alpha)d_o + \alpha^* N + \alpha L + d(n)(\alpha - \alpha^*)$$

For $\alpha > \alpha^*$ the expression on the right attains its maximum when d(n) reaches its

largest value of $N-d_o$, and so

$$M \le \max_{n \in P} f(n) \le (1-2\alpha+\alpha^*)d_o+\alpha L+N\alpha$$

Now let d_o be any unbounded function of N such that $d_o=o(N)$ and consider the probability $P[M \ge (1+\delta)\alpha^* N]$. For every α between α^* and $(1+\delta)\alpha^*$ the inequality

$$(1-2\alpha+\alpha^*)d_o(N)+\alpha L_\alpha+N\alpha \le (1+\delta)\alpha^* N$$

will be satisfied for sufficiently large N, hence, choosing $\alpha^* < \alpha < (1+\delta)\alpha^*$, we have

$$P[M \ge (1+\delta)\alpha^* N] \le P[\text{no } (\alpha, L_\alpha)\text{–regular path stems from level } d_o(N)]$$
$$\le \left[q_\alpha\right]^{2^{d_o(N)}} \to 0$$

We can now bound our target expression by a sum of two probabilities:

$$P[M \ge (1+\varepsilon)C^*(N,p)] \le 1-P\left[(1+\varepsilon/2)C^*(N,p) \ge \alpha^* N \ge (1-\varepsilon/2)M\right]$$
$$\le P\left[C^*(N,p) \le \frac{\alpha^* N}{1+\varepsilon/2}\right] + P\left[M \le \frac{\alpha^* N}{1-\varepsilon/2}\right],$$

and, since each term on the right tends to zero, our claim is proved.

APPENDIX 2.2: Properties of A**

Algorithm A** is a variation of A*. It can be viewed as a BF* algorithm (Section 2.1) that uses an evaluation function:

$$f'_{P_{s-n}}(n) = \max \left\{ f(n') = g_{P_{s-n}}(n') + h(n') \mid n' \in P_{s-n} \right\}$$

A** differs from A* in that it relies not only on the $g+h$ value of node n, but also considers the $g+h$ values along the path from s to n. The maximum is then used as a criterion for node selection. Note that A** cannot be considered a BF* algorithm since it uses one function, f', for ordering nodes for expansion (step 3) and a different function g for redirecting pointers (step 6c). Had we allowed A** to use f' for both purposes it would not be admissible relative to $\underline{I}_{AD}$, since f' is not order preserving.

We will now show that A** is admissible over $\underline{I}_{AD}$.

Theorem:

Algorithm A** will terminate with an optimal solution in every problem instance where $h \leq h^*$.

Proof:

Suppose the contrary. Let C be the value of the path P_{s-t} found by A** and assume $C > C^*$.

We will argue that exactly before A** chooses the goal node t for expansion, there is an OPEN node n' on an optimal path with $f'(n') \leq C^*$. If we show that, than obviously A** should have selected n' for expansion and not t, since $f'(n') \leq C^* < C = f'(t)$, which yields a contradiction.

Since A** redirects pointers according to g, the pointer assigned to the shallowest OPEN node n' along any optimal path is directed along that optimal path. Moreover, $h \leq h^*$ implies that such a node satisfies $f'(n') \leq C^*$, and this completes our argument.

□

We next show that A** dominates A* in the following sense:

Theorem:

a) For every tie-breaking-rule of A* and for every problem instance $I \in \underline{I}_{AD}$, there exists a tie-breaking-rule for A** which expands a subset of the nodes expanded by A*. Moreover,

b) There exists a problem instance and a tie-breaking-rule for A** that expands a proper subset of the nodes which are expanded by any tie-breaking-rule of A*.

Proof: Part a

From the definition of f' it is clear that all paths which are strictly bounded below C^* relative to f are also strictly bounded below C^* relative to f'. Therefore, both algorithms have exactly the same set of surely expanded nodes, $\underline{N}_f^{C^*} = \underline{N}_{f'}^{C^*}$, and this set is expanded before any node outside this set. Let n^* be the first node expanded by A* satisfying the equality $f(n^*)=C^*$. Exactly before n^* is chosen for expansion all nodes in $\underline{N}_f^{C^*}$ were already expanded. A**, after expanding those nodes also has n^* in OPEN with $f'(n^*)=C^*$; there exists, therefore, a tie-breaking-rule in A** which will also choose n^* for expansion. From that moment on, A* will expand some sequence $n^*, n_1, n_2, n_3, \cdots t$ for which $f(n_i) \leq C^*$. Since on these nodes

$f'(n_i) = C^*$, it is possible for A** to use a tie-breaking-rule that expands exactly that same sequence of nodes until termination.

Part b

Examine the graph of Figure 2.9.

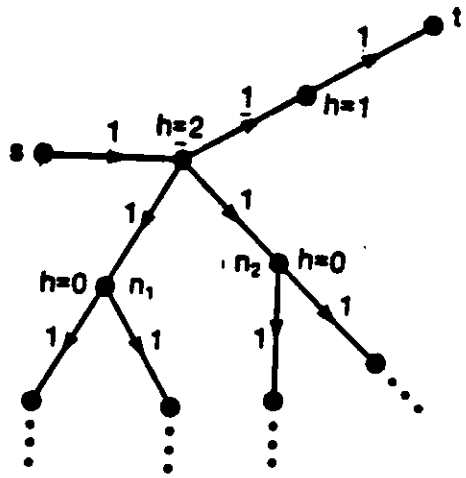


Figure 2.9 - Used in proof of theorem in appendix 2.2

n_1 and n_2 will be expanded by every tie-breaking-rule of A* while there is a tie-breaking-rule for A** that expands only P_{s-t}.

□

CHAPTER 3

GENERATING HEURISTICS FOR CONSTRAINT-SATISFACTION PROBLEMS

3.1. BACKGROUND AND MOTIVATION

3.1.1 Introduction

An important component of human problem-solving expertise is the ability to use knowledge about solving easy problems to guide the solution of difficult ones. Only a few works in AI [Sacerdoti 1974, Carbonell 1983] have attempted to equip machines with similar capabilities. Gaschnig [Gaschnig 1979] Guida et al. [Guida 1979], and Pearl [Pearl 1983] suggested that knowledge about easy problems could be instrumental in the mechanical discovery of heuristics. Accordingly, it should be possible to manipulate the representation of a difficult problem until it is approximated by an easy one, solve the easy problem, then use the solution to guide the search process in the original problem.

The implementation of this scheme requires three major steps: 1. simplification 2. solution 3. advice generation. Additionally, to perform the simplification step, we must have a simple, **a-priori** criterion for deciding when a problem lends itself to easy solution.

This paper uses the domain of constraint-satisfaction tasks to examine the feasibility of these three steps. It establishes criteria for recognizing classes of easy problems, provides special procedures for solving them, demonstrate a scheme for generating good relaxed models, and introduces an efficient method for extracting advice from them. Finally, the utility of using the advice is evaluated in a synthetic domain of problem instances.

Constraint-satisfaction problems (CSP) involve the assignment of values to variables subject to a set of constraints. Understanding three-dimensional drawings, graph coloring, electronic circuit analysis, and truth maintenance systems are examples of CSPs. These are normally solved by some version of backtrack search which may require exponential search time (for example, the graph coloring problem is known to be NP-complete.)

The following paragraphs summarize the basic terminology of the theory of CSP as presented in [Montanari 1974] and extended by [Mackworth 1977] and [Freuder 1982]. Some observations are presented regarding the relationships between the representation of the problem and the performance of the backtrack algorithm.

3.1.2 Definitions and Nomenclature

Formally, the underlying model of a CSP involves a set of n variables $X_1, \ldots, X_n$ each having a set of domain values $D_1, \ldots, D_n$. An **n-ary relation** on these variables is a subset of the Cartesian product:

$$\rho \subseteq D_1 \times D_2 \times, \ldots, \times D_n \,. \tag{1}$$

A **binary constraint** R_{ij} between two variables is a subset of the Cartesian product

of their domain values, i.e.,

$$R_{ij} \subseteq D_i \times D_j \ . \tag{2}$$

A network of binary constraints is the set of variables $X_1, \ldots, X_n$ plus the set of binary constraints between pairs of variables and it represents an n-ary relation defined by the set of n-tuples that satisfy all the constraints. Given a symmetric network of constraints between n variables, the relation ρ represented by it is:

$$\rho = \{(x_1, x_2, \ldots, x_n) \mid x_i \in D_i, and\ (x_i, x_j) \in R_{ij} \text{ for all } ij\} \ . \tag{3}$$

Not every n-ary relation can be represented by a network of binary constraints with n variables, and the issues of finding the best approximation by such network are addressed in [Montanari 1974]. In this paper we will discuss only relations induced by network of binary constraints and henceforth assume that all constraints are binary and symmetric.

Each network of constraints can be represented by a **constraint graph** where the variables are represented by nodes and the non-universal constraints by arcs. The constraints themselves can be represented by the set of pairs they allow, or by a matrix in which rows and columns correspond to values of the two variables and the entries are 0 or 1 depending on whether the corresponding pair of values is allowed by the constraint. Figure 3.1 displays a typical network of constraints (a), where constraints are given using matrix notation (b).

Several operations on constraints can be defined. The useful ones are: union, intersection, and composition. The union of two constraints between two variables is a constraint that allows all pairs that are allowed by either one of them. The intersection of two constraints allows only pairs that are allowed by both constraints. The

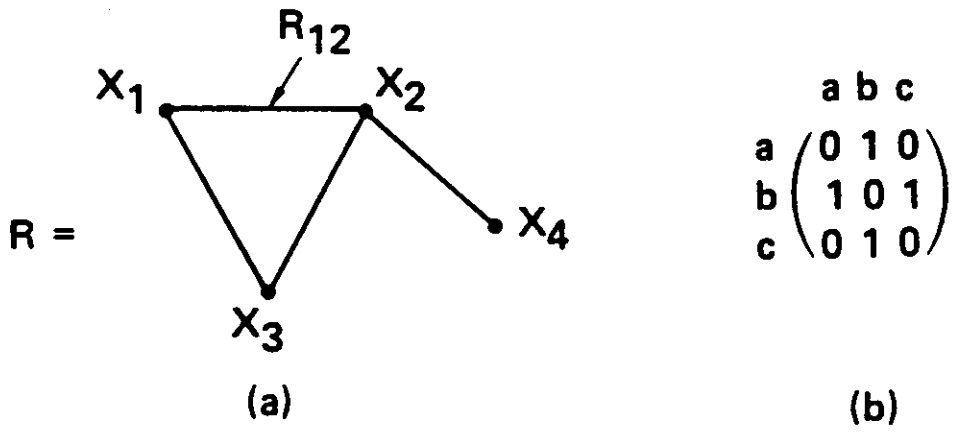


Figure 3.1 - A constraint Network (a) and matrix representation (b)

composition of two constraints, $R_{12}\,R_{23}$ "induces" a constraint R_{13} defined as follows: A pair (x_1,x_3) is allowed by R_{13} if there is at least one value $x_2 \in D_2$ such that $(x_1,x_2)\in R_{12}$ and $(x_2,x_3)\in R_{23}$. If matrix notation is used to represent constraints, then the induced constraint R_{13} can be obtained by matrix multiplication:

$$R_{13} = R_{12} \cdot R_{23} \tag{4}$$

A partial order among the constraints can be defined as follows: $R_{ij} \subseteq R'_{ij}$ iff every pair allowed by R_{ij} is also allowed by R'_{ij} (this is exactly set inclusion). In this case we say that R_{ij} is **smaller** (or stronger) than R'_{ij}. We can also say that R'_{ij} is a relaxation of R_{ij}. The smallest constraint between variables X_i and X_j is the **empty constraint**, denoted Φ_{ij}, which does not allow any pair of values. The largest (i.e. weakest) is the **universal constraint**, denoted U_{ij}, which permits all possible pairs. A corresponding partial order can be defined among networks of constraints having the same set of variables. We say that $R \subseteq R'$ if the partial order is satisfied for every pair of corresponding constraints in the networks.

Finally, we define the notion of equivalence among networks of constraints: two networks of constraints with the same set of variables are equivalent if they represent the same n-ary relation.

Consider, for example, the network of figure 3.2, representing a problem of four bi-valued variables. The constraints are attached to the arcs and are given, in this case, by sets of pairs. The direction of the arcs only indicates the way by which constraints are specified. The constraint between X_1 and X_4, displayed in part (b), can be induced by R_{12} and R_{24}. Therefore, adding this constraint to the network will result in an equivalent network. Similarly, since the constraint R_{21} can be induced from R_{23} and R_{31} it can be deleted without changing the relation represented by the network.

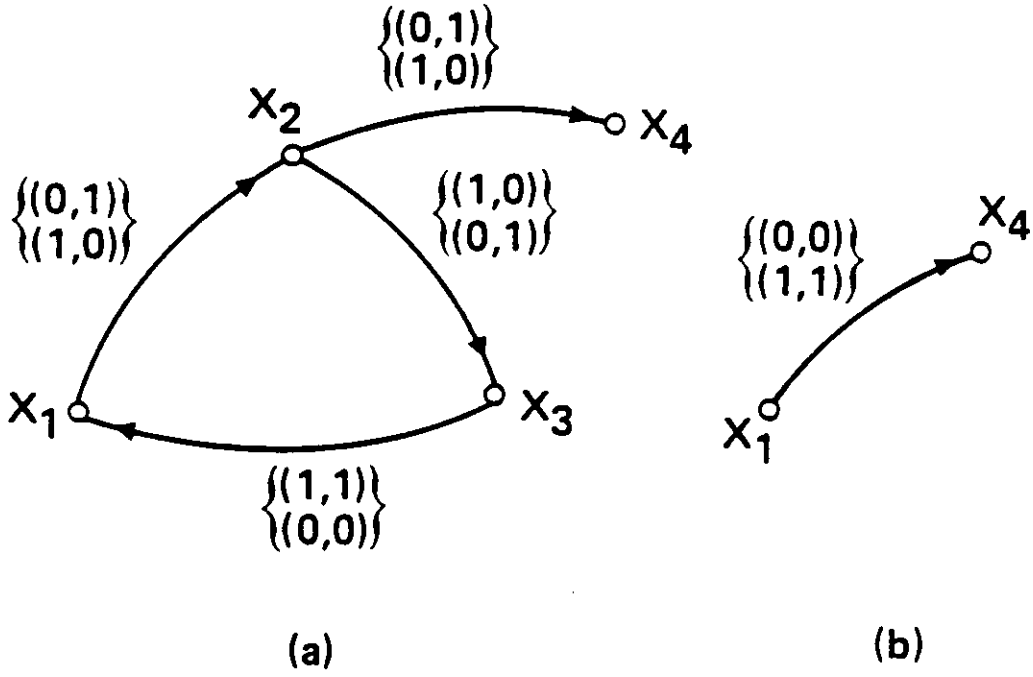


Figure 3.2 - A constraint network, constraints represented by set of pairs.

The process of inducing relations in a given network makes the constraints smaller and smaller, while leaving the networks equivalent to each other. Montanari called the smallest network of constraints which is equivalent to a given network R, **The Minimal Network**. The minimal network of constraints makes the "global" constraints on the network as "local" as possible. In other words, a minimal network of constraints is perfectly explicit.

Every binary-constrained CSP problem can be represented by a network of constraints. A tuple in the relation represented by the network is called a solution. The problem is either to find all solutions, one solution, or to verify that a certain tuple is a solution. The last problem is fairly easy while the first two problems can be difficult and have attracted a substantial amount of research.

3.1.3 Backtrack for CSP

The algorithm mostly used to solve CSP problems is backtrack. Given a vertical order of the set of variables $X_1, X_2, \ldots, X_n$ and a **horizontal order** of values in each domain of a variable $x_{i,1}, x_{i,2}, \ldots, x_{i,k}$, algorithm Backtrack for finding one solution is given below:

Backtrack
Begin
1. Assign $x_{1,1}$ to X_1 (if allowed by a unary constraint)
2. $k=1$
3. while $k \leq n-1$
4. while X_{k+1} has values /* $x_1, x_2, \ldots, x_k$ were already selected*/
5. choose first value $x_{k+1,j}$ s.t. constraints($x_1, x_2, \ldots, x_k, x_{k+1,j}$) = true
6. then erase (temporarily) $x_{k+1,1}, \ldots, x_{k+1,j}$ from domain of X_{k+1}
7. $k=k+1$
8. goto 3
9. end
10. $k=k-1$ (backtrack since no value at (5) exists).
11. If $k=0$ exit , no solution exists.
12. end
13. exit with solution
End.

In line 5 of the algorithm all the constraints between X_{k+1} and previous variables in the vertical order are checked. The value chosen should be consistent with all the previous instantiated values under those constraints. For Backtrack to find all solutions the above algorithm should be modified slightly by adding another outer loop and terminating only when $k=0$.

Montanari considered the question of finding the minimal network M of a given network R as the central problem in CSP, implying that once it is available the problem is virtually solved. The following two lemmas elaborate on this issue by relating the minimal network to the backtrack algorithm.

Lemma 1:

Let R and R' be two equivalent networks such that $R' \subseteq R$. Given the same order for instantiating variables, any sequence of values that is assigned by Backtrack on R' will be assigned also by Backtrack on R when Backtrack looks for all solutions.

Proof:

The order between the networks implies that any sequence of values which is con-

sistent under R' is also consistent under R.

□

Conclusion:

Given a network R and a fixed order of instantiation of variables, Backtrack's performance, when looking for all solutions, is most efficient on the minimal network, relative to all networks which are equivalent to R, since it is contained in all of them.

We now show that when the algorithm seeks only one solution then, using the minimal network, the solution can be found easily in many cases. Some more definitions are required.

Given an n-ary relation ρ, representable by a network with n variables, the projection ρ_S of the relation ρ on a subset S of the variables is not always representable by a network with $|S|$ nodes. If for any subset of variables, S, ρ_S is representable by a network with $|S|$ variables then ρ is said to be a **Decomposable relation**. Given an n-ary decomposable relation ρ, represented by a minimal network M, then for any subset S of variables the subnetwork of M restricted to the nodes in S, is a minimal network of ρ_S. In this case M is also said to be decomposable.

For example, the network in figure 3.3 is minimal but not decomposable. The relation represented by M is:

$$\rho = \{(x_{1,1},x_{2,1},x_{3,1},x_{4,1}),(x_{1,1},x_{2,2},x_{3,2},x_{4,2}),(x_{1,2},x_{2,2},x_{3,1},x_{4,3})\} \qquad (5)$$

Where $x_{i,j}$ denotes the j^{th} value of variable X_i (Note that X_4 is a non-binary variable.)

If $S = \{X_1,X_2,X_3\}$ it can be shown that ρ_s, given by

$$\rho_S = \{(x_{1,1},x_{2,1},x_{3,1}),(x_{1,1},x_{2,2},x_{3,2}),(x_{1,2},x_{2,2},x_{3,1})\}, \qquad (6)$$

cannot be represented by a network with 3 variables. (For more details see [Montanari 1974]).

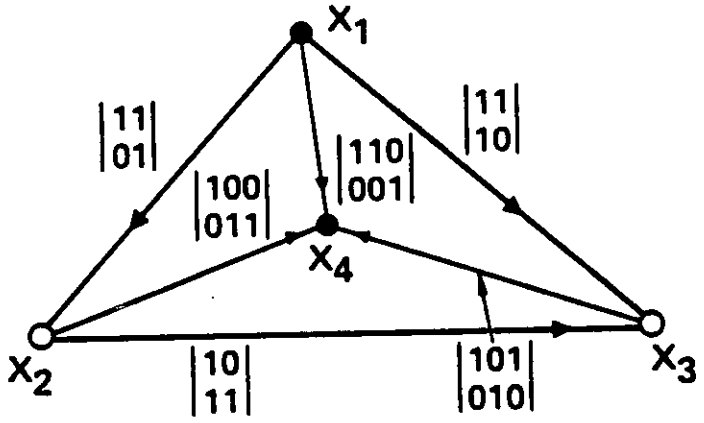


Figure 3.3 - Minimal nondecomposable network of constraints

Lemma 2:

If M is minimal and decomposable network then Backtrack will find one solution without backtracking at all.

Proof:

From M's decomposability it follows that any projection ρ_s has a minimal network which is the subnetwork of M that is restricted to the variables in S. Therefore, any tuple of the variables in S that satisfy all the constraints in the minimal subnetwork is part of an n-tuple in the n-ary relation represented by M, and therefore it can always be extended.

□

The complexity of finding a solution given a minimal and decomposable network M is, therefore, $O(n^2k)$ where n is the number of variables and k is the maximum cardinality of the value domain for all variables. In the previous example of a nondecomposable minimal network Backtrack may explore the path $x_{1,1}, x_{2,2}, x_{3,1}$ and since it cannot be extended to a 4-tuple relation satisfying M the algorithm will have to backtrack. In conclusion we see that solving a CSP in which we are finding all or one solution, is easier when the minimal network is available, but this does not guarantee backtrack-free search unless the network is also decomposable.

Backtrack and its performance on CSPs were extensively discussed in the AI literature. Most researchers have been trying to identify the major maladies in its performance, to provide a corresponding cure, and to analyze the results. These works can be classified along the following dimensions:

1. The problem objectives: finding all or finding one solution

2. control parameters: controlling the order of instantiation of variables, order of inatantiation of values, or manipulating the problem's representation by pruning values or propagating constraints.

3. cure implementation: preprocessing the cures prior to the start of the algorithm, or incorporating them dynamically into the algorithm while it searches for solution(s).

Mentioning only few studies, we start with [Montanari 1974] who considered the task of finding all solutions, and discussed the solution of a problem by propagating the constraints and pruning pairs of values from them. In light of the previous lemmas these methods can be regarded as a preprocessing phase to a backtrack

algorithm although the latter was not mentioned explicitly. Mackworth [Mackworth 1977] extended Montanati's work by introducing consistency checks to cure the maladies of Backtrack. Haralick and Eliot [Haralick 1980] discussed the task of finding all solutions and examined various methods of value pruning including look-ahead mechanisms which are incorporated into the algorithm. Freuder [Freuder 1982] considered the problem of finding one solution to a CSP and provided a procedure to select a good ordering of variables which is performed as a preprocessing to Backtrack. Other works in analyzing the average performance of Backtrack were reported by [Nudel 1983, Purdom 1985] and [Haralick 1980] all estimating the size of the tree exposed by Backtrack while searching for all solutions.

It seems that the only parameter not considered for controlling Backtracks' performance is the order by which values are assigned to variables. Part of the reason can be explained by the following theorem.

Theorem 1:

Given the objective of finding all solutions and given a fixed order for instantiation of variables, the search tree exposed by Backtrack is invariant to the order of selection of values. (All search trees which are identical up to an ordering of branches are considered the same.)

Proof:

Any sequence of values that is explored by Backtrack w.r.t. a specific order of variables is consistent under this subset of variables, and it may or may not lead to a solution. The only way Backtrack can find out if it is extensible to a solution is to continue and explore it. Therefore, Backtrack which tries to find all solutions will have to search this sequence for any order of value assignment.

□

Similarly, Backtrack that looks for one solution in a CSP that has no solution will expose the same search tree under any order of value assignment, given a fixed order of variables. The above theorem states that for the task of finding all solutions value-selection strategies will not improve Backtrack's performance.

In this paper we address the objective of finding a single solution to CSPs. Although this problem is easier it can still be very difficult (e.g. 3-colorability) and it appears frequently. Theorem proving, planning and even vision problems are examples of domains where finding one solution will normally suffice [Simon 1975], and, the order by which values are selected may have in this case a profound effect on the algorithm's performance. In the following section we outline a general approach to devising value selection strategies for finding one solution to CSP.

3.1.4 General Approach for Automatic Advice Generation

Following the model of the A^* algorithm that uses heuristics to guide the selection of the next node for expansion, we now wish to guide Backtrack in selecting the next node on its path. We assume that the order of variables is fixed and therefore the selection of the next node amounts to choosing a promising assignment of values from a set of pending options. Clearly, if the next value can be guessed correctly, and if a solution exists, the problem will be solved in linear time with no backtracking. Backtrack builds partial solutions and extends them as long as they show promise to be part of a whole solution. When a dead-end is recognized it backtracks to a previous variable. The advice we wish to generate should order the candidates according to the confidence we have that they can be extended further to a solution.

Such confidence can be obtained by making simplifying assumptions about the continuing portion of the search graph and estimating the likelihood that it will contain a solution even when the simplifying assumptions are removed. It is reasonable to assume that if the simplifying assumptions are not too severe then the number of solutions found in the simplified version of the problem would correlate positively with the number of solutions present in the original version. We, therefore, propose to count the number of solutions in the simplified model and use it as a measure of confidence that the options considered will lead to an overall solution.

To incorporate the advice generation into the backtrack algorithm, line 5 should be replaced by the following:

5a. eliminate values of X_{k+1} which are not consistent with $x_1, \ldots, x_k$.

5b. /* $x_{k+1,1}, \ldots, x_{k+1,t}$ are the remaining candidates for assignment*/

advise($(x_{k+1,1}, \ldots, x_{k+1,t})$, $(x'_{k+1,1}, \ldots, x'_{k+1,t})$)

5c. assign $x'_{k+1,1}$ to X_{k+1}

The advice procedure takes the set of consistent values of X_{k+1} and orders them according to the estimates of the number of possible solutions stemming from them.

The remaining sections describe the advice-giving algorithm, provide theoretical grounds for it, and report experimental evaluation of its performance. In section 3.2 we establish criteria for recognizing classes of easy CSPs and introduce an efficient method of counting the number of solutions. Section 3.3 describes a process of approximating a given CSP by an easy relaxed one. Section 3.4 evaluates the utility of using the advice using a synthetic domain of CSPs.

3.2. EASY CONSTRAINT-SATISFACTION PROBLEMS

3.2.1 Introduction and background

In general, a problem is considered easy when its representation permits a solution in polynomial time. However, since we are dealing mainly with backtrack algorithms, we will consider a CSP easy if it can be solved by a **backtrack-free** procedure. A backtrack-free search is one in which Backtrack terminates without backtracking, thus producing a solution in time linear in the number of variables.

The discussion of backtrack-free CSPs relies heavily on the concept of constraint graphs. Freuder [Freuder 1982] has identified sufficient conditions for a constraint graph to yield a backtrack-free CSP, and has shown, for example, that tree-like constraint graphs can be made to satisfy these conditions, with a small amount of preprocessing. Our main purpose here is to further study classes of constraint graphs lending themselves to backtrack-free solutions and to devise efficient algorithms for solving them. Once these classes are identified they can be chosen as targets for a problem simplification scheme: constraints can be selectively deleted from the original specification so as to transform the original problem into a backtrack-free one. As already mentioned, we propose to use the "number of consistent solutions in the simplified problem" as a figure-of-merit to establish priority of value assignments in the backtracking search of the original problem. We show that this figure of merit can be computed in time comparable to that of finding a single solution to an easy problem.

Definition: ([Freuder 1982]) **An ordered constraint graph** is a constraint graph in which the nodes are linearly ordered to reflect the sequence of variable assignments executed by the Backtrack search algorithm. The **width of a node** is the number of

arcs that lead from that node to previous nodes, the **width of an ordering** is the maximum width of all nodes, and the **width of a graph** is the minimum width of all the orderings of that graph.

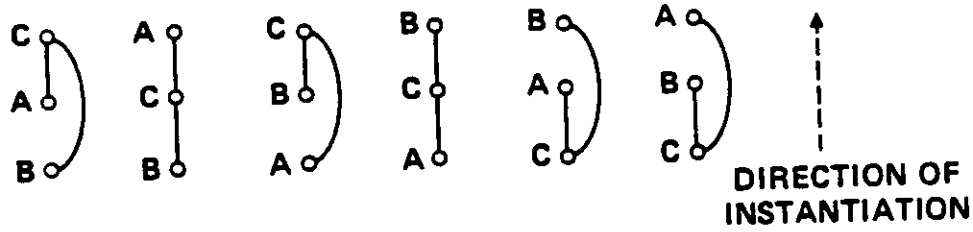


Figure 3.4 - Ordering of a constraint graph

Figure 3.4 presents six possible orderings of a constraint graph. The width of node C in the first ordering (from the left) is 2, while in the second ordering it is 1. The width of the first ordering is 2, while that of the second is 1. The width of the constraint graph is, therefore, 1. Freuder provided an efficient algorithm for finding both the width of a graph and the ordering corresponding to this width. He further showed that a constraint graph is a tree iff it is of width 1.

Montanari [Montanari 1974] and Mackworth [Mackworth 1977] have introduced two kinds of local consistencies among constraints named **arc consistency** and **path consistency**. Their definitions assume that the graph is directed, i.e., each symmetric constraint is represented by two directed arcs.

Let $R_{ij}(x,y)$ stand for the assertion that (x,y) is permitted by the explicit constraint R_{ij}.

Definition: ([Mackworth 1977]): Directed arc (X_i,X_j) is **arc consistent** iff for any value x $\in D_i$ there is a value y $\in D_j$ such that $R_{ij}(x,y)$.

Definition ([Montanari 1974]): A path of length m through nodes $(i_0,i_1,\ldots,i_m)$ is **path consistent** if for any value $x \in D_{i_0}$ and $y \in D_{i_m}$ such that $R_{i_0 i_m}(x,y)$, there is a sequence of values $z_1 \in D_{i_1}, \ldots, z_{m-1} \in D_{i_{m-1}}$ such that

$$R_{i_0 i_1}(x,z_1) \text{ and } R_{i_1 i_2}(z_1,z_2) \text{ and } \cdots R_{i_{m-1} i_m}(z_{m-1},y). \tag{7}$$

$R_{i_0 i_m}$ may also be the universal relation e.g., permitting all possible pairs.

A constraint graph is arc (path) consistent if each of its directed arcs (paths) is arc (path) consistent. Achieving "arc consistency" means deleting certain values from the domains of certain variables such that the resultant graph will be arc-consistent, while still representing the same overall set of solutions. To achieve path-consistency, certain pairs of values that were initially allowed by the local constraints should be disallowed. Montanari and Mackworth have proposed polynomial-time algorithms for achieving arc-consistency and path consistency. In [Mackworth 1984] it is shown that arc consistency can be achieved in $O(ek^3)$ while path consistency can be achieved in $O(n^3k^5)$, where n is the number of variables, k is the number of possible values, and e is the number of edges.

Theorem 2([Freuder 1982]):

a. If the constraint graph has a width 1 (i.e. the constraint graph is a tree) and if it is arc consistent then it admits backtrack-free solutions.

b. If the width of the constraint graph is 2 and it is also path consistent then it admits backtrack-free solutions.

The above theorem suggests that tree-like CSPs (CSPs whose constraint graph are trees) can be solved by first achieving arc consistency and then instantiating the variables in an order which makes the graph exhibit width 1. Since this backtrack-free istantiation takes $O(ek)$ steps, and on trees $e=n-1$, the whole problem can be solved in $O(nk^3)$. The test for this property is also easily verified: to check whether a given graph is a tree can be done by a regular $O(n^2)$ spanning tree algorithm. Thus, tree-like CSPs are easy since they can be made backtrack-free after a preprocessing of low complexity.

The second part of the theorem tempts us to conclude that a width-2 constraint graph should admit a backtrack-free solution after passing through a path-consistency algorithm. In this case, however, the path-consistency algorithm may add arcs to the graph and increase its width beyond 2. This often happens when the algorithm deletes value-pairs from a pair of variables that were initially related by the universal constraint (having no connecting arc between them), and it is often the case that passage through a path-consistency algorithm renders the constraint graph complete. It may happen, therefore, that no advantage could be taken of the fact that a CSP possesses a width-2 constraint graph if it is not already path consistent. We are not even sure whether width-2 suffices to preclude NP-completeness.

In the following section we give weaker definitions of arc and path consistency which are also sufficient for guaranteeing backtrack-free solutions but have two advantages over those defined by [Montanari 1974] and [Mackworth 1977] :

1. They can be achieved more efficiently, and

2. They add fewer arcs to the constraint-graph, thus preserving the graph width

in a larger classes of problems.

3.2.2 Algorithms for achieving directional consistency

a. Case of Width-1

Securing full arc-consistency is more than is actually required for enabling backtrack-free solutions in constraint graphs which are trees. For example, if the constraint graph in figure 3.5 is ordered by (X_1,X_2,X_3,X_4), nothing is gained by making the directed arc (X_3,X_1) consistent.

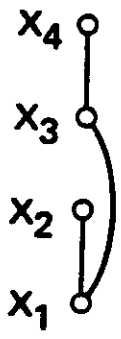


Figure 3.5 - Ordered constraint graph

To ensure backtrack-free assignment, we need only make sure that any value assigned to variable X_1 will have at least one consistent value in D_3. This can be achieved by making only the directed arc (X_1,X_3) consistent, regardless of whether (X_3,X_1) is consistent or not. We, therefore, see that arc-consistency is required only w.r.t a single direction, the one specified by the order in which Backtrack will later choose variables for instantiations. This motivates the following definitions.

Definition: Given an order d on the constraint graph R, we say that R is d-arc-

consistent if all the directed edges which follow the order d are arc-consistent.

Theorem 3:

Let d be a width-1 order of a constraint tree T. If T is d-arc-consistent then the backtrack search along the order d is backtrack-free.

proof:

Suppose that $X_1, X_2, \ldots, X_k$ were already instantiated. The variable X_{k+1} is connected to at most one previous variable (from the width-1 property), say X_i, which was assigned the value x_i. Since the directed arc (X_i, X_{k+1}) is along the order d, its arc-consistency implies the existence of a value x_{k+1} such that the pair (x_i, x_{k+1}) is permitted by the constraint $R_{i(k+1)}$. Thus, the assignment of x_{k+1} is consistent with all previous assignments.

□

An algorithm for achieving directional arc-consistency for any ordered constraint graph is given next (The order $d = (X_1, X_2, \ldots, X_n)$ is assumed)

DAC- d-arc-consistency

```
1. begin
2.   For i=n to 1 by -1 do
3.     For each arc (X_j,X_i); j<i do
4.       REVISE(X_j,X_i)
5.     end
6.   end
7. end
```

The algorithm REVISE(X_j, X_i), given in [Mackworth 1977] , deletes values from the domain D_j until the directed arc (X_j, X_i) is arc-consistent.

REVISE(X_j,X_i)

1. begin
2. For each $x \in D_j$ do
3. if there is no value $y \in D_i$ such that $R_{ji}(x,y)$ then
4. delete x from D_j
5. end
6. end

To prove that the algorithm achieves d-arc-consistency we have to show that upon termination, any arc (X_j,X_i) along d $(j<i)$, is arc-consistent. The algorithm revises each d-directed arc once. It remains to be shown that the consistency of an already processed arc is not violated by the processing of coming arcs. Let arc (X_j,X_i) $(j<i)$ be an arc just processed by REVISE(X_j,X_i). to destroy the consistency of (X_j,X_i) some values should be deleted from the domain of X_i during the continuation of the algorithm. However, according to the order by which REVISE is performed from this point on, only lower indexed variables may have their set of values updated. Therefore, once a directed arc is made arc-consistent its consistency will not be violated.

The algorithm AC-3 [Mackworth 1977] that achieves full arc-consistency is given for reference:

AC-3
1. begin
2. $Q \leftarrow \{ (X_i,X_j) \mid (X_i,X_j) \in$ arcs, $i \neq j \}$
3. while Q is not empty do
select and delete arc (X_k,X_m) from Q
5. REVISE(X_k,X_m)
6. if REVISE(X_k,X_m) caused any change then
7. $Q \leftarrow Q \cup \{(X_i,X_k) \mid (X_i,X_k) \in$ arcs ,i≠k,m$\}$
7. end
8. end

The complexity of AC-3, is $O(ek^3)$, while the directional arc-consistency algorithm

takes ek^2 steps since the REVISE algorithm, taking k^2 tests, is applied to every arc exactly once. It is also optimal, because even to verify directional arc-consistency each arc should be inspected once, and that takes k^2 tests. Note that when the constraint graph is a tree, the complexity of the directional arc-consistency algorithm is $O(nk^2)$.

Theorem 4:

A tree-like CSP can be solved in $O(nk^2)$ steps and this is optimal.

proof:

Once we know that the constraint graph is a tree, finding an order that will render it of width-1 takes $O(n)$ steps. A width-1 tree-CSP can be made d-arc-consistent in $O(nk^2)$ steps, using the DAC algorithm. Finally, the backtrack-free solution on the resultant tree is found in $O(nk)$ steps. Summing up, finding a solution to tree-like CSP's takes, $O(nk) + O(nk^2) + O(n) = O(nk^2)$. This complexity is also optimal since any algorithm for solving a tree-like problem must examine each constraint at least once, and each such examination may take, in the worst case, k^2 steps, especially when no solution exists and the constraints permit very few pairs of values (see Appendix 3.1 for a formal proof of optimality).

□

Interestingly, if we apply DAC w.r.t. order d and then DAC w.r.t. the reverse order we get a full arc-consistency for trees. We can, therefore, achieve full arc-consistency on trees in $O(nk^2)$. Algorithm AC-3, on the other hand, has a worst case performance on trees of $O(nk^3)$ as is shown next. Figure 3.6 illustrates a CSP problem that has a chain-like constraint graph. There are 7 variables, each with k values, constrained as shown in the figure. Each row represent a variable, the points represent values and the connecting lines describe the allowed pairs of values. Since AC-3 does not determine the order in which the arcs enter REVISE, we will impose

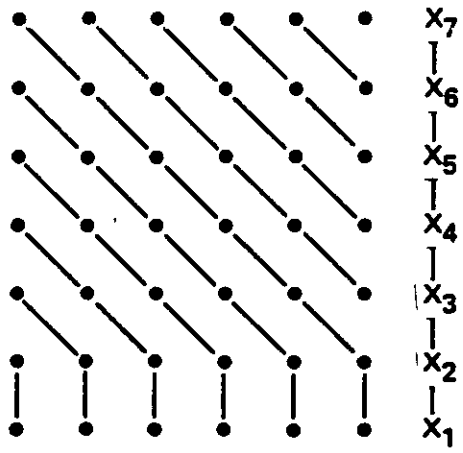


Figure 3.6 - Tree-constraint network showing worst-case of AC-3

an ordering which will be particularly bad for it: in increasing node index, that is, first (X_1,X_2) then (X_2,X_3) etc. However when a new arc is inserted to the queue it is given hte highest priority, because AC-3 employs a last-in-first-out policy. Therefore, Arc (X_1,X_2) will be processed first, then arc (X_2,X_3) and, since a value was deleted from X_2, arc (X_1,X_2) will be inserted back to the queue and processed. No change occurred and, therefore, the next arc to be processed is (X_3,X_4), this will cause processing of (X_2,X_3) again which, in turn, causes the processing of (X_1,X_2) and so on. We see that each arc will be processed $k-1$ times resulting in a total complexity of $O(nk^3)$. It was recently found [Mohr 1985], that using special data structure, the general arc-consistency on graphs can be improved to achieve performance of $O(ek^2)$. This is done by keeping for each value the number of values that match it in each neiboring variable.

Returning to our primary aim of studying easy problems, we now show how advice can be generated using a tree-like approximation. Suppose that we want to solve an n variables CSP using Backtrack with $X_1,X_2,\ldots,X_n$ as the order of

instantiation. Let X_i be the variable to instantiate next, with $x_{i1}, x_{i2}, \ldots, x_{ik}$ the possible candidate values. Ideally, to minimize backtracking we should first try the values which are more likely to lead to a consistent solution but, since this likelihood is not known in advance, we may estimate it by counting the number of consistent solutions that each candidate admits in some approximate, easily solved problem. We generate a relaxed tree-like problem by deleting some of the explicit constraints given, then count the number of solutions consistent with each of the k possible assignments, and finally use these counts as a figure of merit for scheduling the various assignments. In the following we show how counting the number of consistent solutions can be imbedded within the d-arc-consistency algorithm, DAC, on trees.

Any width-1 order, d, on a constraint tree determines a directed tree in which a parent always precedes its children in d (arcs are directed from the parent to its children). Let $N(x_{jt})$ stand for the number of solutions in the subtree rooted at X_j consistent with the assignment of x_{jt} to X_j. Consider a node X_j with all its successor nodes as in figure 3.7. Looking first on the relation between X_j and a specific child node X_c, it is clear that the value x_{jt} can participate in a solution with each value of X_c with which it is consistent, no matter what values are assigned to variables in the subtree rooted at X_c. Therefore the number of partial solutions consistent with x_{jt}, which are restricted to the subtree rooted at X_c, is a sum of the corresponding numbers in the child node. Namely:

$$N(x_{jt} \mid \text{in the tree rooted at } X_c) = \sum_{\{x_{cl} \in D_c \mid R_{jc}(x_{jt}, x_{cl})\}} N(x_{cl}) \qquad (8)$$

Since the partial solutions coming from different successor nodes can be combined in all possible ways, the number of solutions in the subtree rooted at X_j will be their product. Therefore, $N(\cdot)$ satisfies the following recurrence:

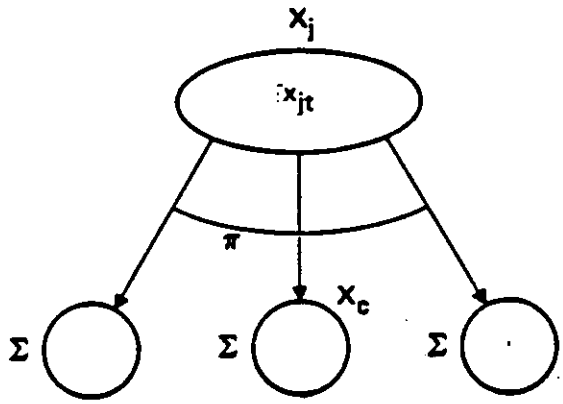


Figure 3.7 - Schematic computation of number of solutions in trees.

$$N(x_{jt}) = \prod_{\{c \mid X_c \text{ is a child of } X_j\}} \sum_{\{x_{cl} \in D_c \mid R_{jc}(x_{jt}, x_{cl})\}} N(x_{cl}) \qquad (9)$$

From this recurrence it is clear that the computation of $N(x_{jt})$ may follow the exact same steps as in DAC: simultaneously with testing that a given value x_{jt} is consistent with each of its children nodes, we simply transfer from each child of X_j to x_{jt} the sum total of the counts computed for the child's values that are consistent with x_{jt}. The overall value of $N(x_{jt})$ will be computed later on by multiplying together the summations obtained from each of the children. The computation starts at the leaves, initialized to $N = 1$, and progresses towards the root. Each variable perform the counting only after all its child's nodes computed their counts as in the counting procedure COUNT that follows.

The COUNT procedure contains the arc-consistency algorithm and REVISE described earlier and performs the calculation according to the recurrence (9) above. The algorithm terminates when the root is assigned counts for all its values. The COUNT procedure is defined for a parent node and all its children.

COUNT($V_p, V_1, V_2, \ldots, V_r$)

1. begin
2. For each (V_p, V_l) do
3. REVISE(V_p, V_l)
4. $n_l(v_{ps}) = \sum_{R_{pl}(v_{ps}, v_{lk})} N(v_{lk})$
5. end
6. For each value $v_{pt} \in D_p$ do
7. $N(v_{pt}) = \prod_{l; V_l \text{ a child}} n_l(v_{pt})$
8. end
9. end

Like REVISE, line 4 takes k^2 steps, therefore, for each parent node, V_j, it takes $k^2 \cdot deg(V_j)$. Thus, the counting for all the variables in the subtree, sums up to $O(nk^2)$ (if n is the number of variables in the tree) , not increasing the complexity of the directional arc-consistency by more then a constant.

b. Case of Width-2

Order information can also facilitate backtrack-free search on width-2 problems by making path-consistency algorithms directional.

Montanari had shown that if a network of constraints is consistent w.r.t all paths of length 2 (in the complete network) then it is path-consistent. Similarly we will show that directional path-consistency w.r.t. length-2 paths is sufficient to obtain a backtrack-free search on a width-2 problems.

Definition: A constraint graph, R, ordered w.r.t. order $d = (X_1, X_2, \ldots, X_n)$, is **d-path-consistent** if for every pair of values (x, y), $x \in X_i$ and $y \in X_j$ such that $R_{ij}(x, y)$ and $i < j$, there exists a value $z \in X_k$, $k > j$ such that $R_{ik}(x, z)$ and $R_{kj}(z, y)$ for every $k > i, j$

Theorem 5:

Let d be a width-2 order of a constraint graph. If R is directional arc- and path-consistent w.r.t d then it is backtrack-free.

Proof:

To ensure that a width-2 ordered constraint graph will be backtrack-free it is required that the next variable to be instantiated will have values that are consistent with previously chosen values. Suppose that $X_1, X_2, \ldots, X_k$ were already instantiated. The width-2 property implies that variable X_{k+1} is connected to at most two previous variables. If it is connected to X_i and X_j, $i, j \leq k$ then directional path consistency implies that for any assignment of values to X_i, X_j there exists a consistent assignment for X_{k+1}. If X_{k+1} is connected to one previous variable, then directional arc-consistency ensures the existence of a consistent assignment.

□

An algorithm for achieving directional path-consistency on any ordered graph will have to manage not only the changes made to the constraints but also the changes made to the graph, i.e., the arcs which are added to it. To describe the algorithm we use the matrix representation for constraints. The matrix R_{ii} whose off-diagonal values are 0, represents the set of values permitted for variable X_i. The algorithm is described using the operations of intersection and composition:
The intersection R_{ij} of R'_{ij} and R''_{ij} is written: $R_{ij} = R'_{ij}$ & R''_{ij}.

Given a network of constraints $R = (V, E)$ and an order $d = (X_1, X_2, \ldots, X_n)$, we next describe an algorithm which achieves path-consistency w.r.t. this order.

DPC-d-path-consistency
begin
(1) $Y^0 = R$
(2) for $k=n$ to 1 by -1 do
  (a) $\forall\ i \le k$ connected to k do
    $Y_{ii} = Y^0_{ii}\ \&\ Y_{ik} \cdot Y_{kk} \cdot Y_{ki}$ /* this is REVISE(i,k)
  (b) $\forall\ i,j \le k$ s.t. $(X_i,X_k),(X_j,X_k) \in E$ do
    $Y_{ij} = Y_{ij}\ \&\ Y_{ik} \cdot Y_{kk} \cdot Y_{kj}$
    $E \leftarrow E \cup (X_i,X_j)$
end
end

Step 2a is the equivalent of the REVISE(i,k) procedure, and it performs the directional arc-consistency. Step 2b starts after 2a is performed for every $i < k$. Step 2b updates the constraints between pairs of variables transmitted by a third variable which is higher in the order d. If X_i,X_j, $i,j<k$ are not connected to X_k then the relation between the first two variables is not effected by X_k at all. If only one variable, X_i, is connected to X_k, the effect of X_k on the constraint (X_i,X_j) will be computed by step 2a of the algorithm. The only time a variable X_k effects the constraints between pairs of earlier variables is when it is connected to both. It is in this case only that a new arc may be added to the graph.

The complexity of the directional-path-consistency algorithm is $O(n^3k^3)$. For variable X_i the number of times the inner loop, 2b, is executed is at most $O((i-1)^2)$ (the number of different pairs less then i), and each step is of order k^3. The computation of loop 2a is completely dominated by the computation of 2b, and can be ignored. Therefore, the overall complexity is

$$\sum_{i=2}^{n}(i-1)^2k^3 = O(n^3k^3) \tag{10}$$

Applying directional-path-consistency to a width-2 graph may increase its width and therefore, does not guarantee backtrack-free solutions. Consequently it is

useful to define the following subclass of width-2 CSP problems.

Definition: A constraint graph is **regular width-2** if there exist a width-2 ordering of the graph which remains width-2 after applying d-path-consistency, DPC.

A ring constitutes an example of a regular-width-2. Figure 3.8 shows an ordering of a ring's nodes and the graph resulting from applying the DPC algorithm to the ring. Both graphs are of width-2.

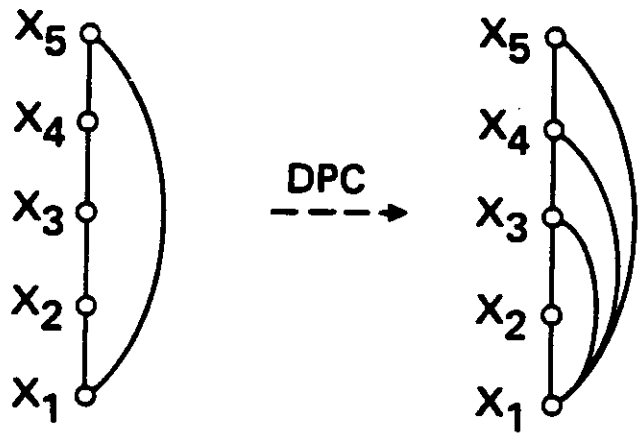


Figure 3.8 - A regular width-2 CSP

Theorem 6:

A regular width-2 CSP can be solved in $O(n^3k^3)$

Proof:

Regular width-2 problem can be solved by first applying the DPC algorithm and then performing a backtrack-free search on the resulting graph. The first takes $O(n^3k^3)$ steps and the second $O(ek)$ steps.

□

The main problem with the preceding approach is whether a regular width-2 CSP can be recognized from the properties of its constraint graph. One promising

approach is to identify nonseparable components of the graph and all its separation vertices [Even 1979].

definition: A connected graph $G(V,E)$ is said to have a **separation vertex** v if there exist vertices a and b, such that all the paths connecting a and b pass through v. A graph which has a separation vertex is called **separable**, and one which has none is called **nonseparable**. A subgraph with no separation vertices is called a **nonseparable component.**

An $O(|E|)$ algorithm for finding all the nonseparable components and the separation vertices is given in [Even 1979]. It is also shown that the connectivity structure between the nonseparable components and the separation vertices has a tree structure.

Let R be a graph and SR be the tree in which the nonseparable components $C_1, C_2, \ldots, C_r$ and the separating vertices $V_1, V_2, \ldots, V_t$ are represented by nodes. A width-1 ordering of SR dictates a partial order on R, d^s, in which each separating vertex precedes all the vertices in its children components of SR.

Theorem 7:

If there exist a d^s ordering on R such that each nonseparable component is regular-width-2 then the total ordering is regular width-2.

Proof:

Let d^s_C be the order induced by d^s on component C, and let P_C be its parent separating vertex. When algorithm DPC reaches a node X, which is not P_C, within this component, then if X is not a separating vertex, it has arcs leading back only to nodes within this component. If the X is a separating vertex, then since it is not the parent of component C, all his children nodes were already processed and, therefore, it has arcs leading back only to nodes within its parent component C. In both cases DPC adds arcs only within the component C. Therefore, if d^s induces an order d^s_C which

is regular-width-2 for all components, then d^s is regular-width-2.

□

As a corollary of theorem 7,we conclude that a tree of simple rings is regular-width-2. In figure 3.9, a graph with 10 nodes identified by its components and its separating vertices is given, with a possible d^s ordering which, in this case, is regular-width-2.

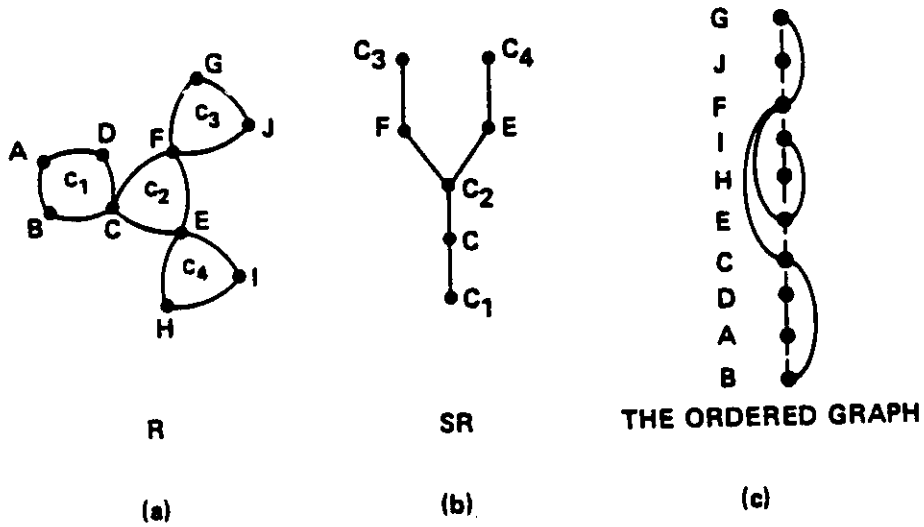


Figure 3.9 - A Graph and its decomposition into nonseparable components

The decomposition of the constraint graph into a tree of nonseparable components and separating vertices can be utilized always to improve Backtrack's performance. Let *SR* be the directed tree associated with a given graph *R*, which induce an ordering on the nodes of the *SR* tree, i.e. parent always precedes its sons. Backtrack, given both *R* and *SR* will work on each component starting from leaf components

towards the root component. The algorithm is described next:

```
Backtrack(R ,SR )
begin
 0. assign TSR  SR . /*TSR stands for temporary SR .*/
 1. If TSR = Φ then generate-solutions(SR ).
 2. let C 1, . . . , C t be the leaf components in TSR .
 3. for each C i a leaf, do
 4.    perform backtrack on this subgraph C i
 5.    associate with the component C i all solutions found, denoted by ρ C i
 6.    delete from the domain of P C i values which do not appear in ρ C i
       (P C i is the parent separating vertex of C i)
 7.    delete C i from TSR .
 8. end.
 9. goto 1.
end.
```

The procedure generate-solution(SR) will generate all the solutions by walking on the SR tree from the root to leaves and concatenating the partial solutions found for each component that can be "joined" w.r.t. the separating vertices. i.e. for two connected components that share the same separating vertex all partial solutions having the same value for that vertex can be joined.

The complexity of the backtrack algorithm as a function of the graph R, denoted by $B(R)$, will satisfy the following:

$$B(R) \leq n \cdot B_{max}(C) \tag{11}$$

where $B_{max}(C)$ is the performance of Backtrack on one of the components in R, which is most costly. If r is the size of the largest component then

$$B(R) = O(k^r) \tag{12}$$

We therefore see that if R has a nice decomposition into an SR then backtrack can utilize this structure and improve its performance.

3.2.3 Summary

This section discuss the following parts of the three main steps involved in the process of generating advice -- simplification, solution, and advice generation.

1. The simplification part: we have devised criteria for recognizing easy problems based on their underlying constraint graphs. The introduction of directionality into the notions of arc and path consistency enabled us to extend the class of recognizable easy problems beyond trees, to include regular width-2 problems.

2. The solution part: using directionality we were able to devise improved algorithms for solving simplified problems and to demonstrate their optimality. In particular, it is shown that tree-structured problems can be solved in $O(nk^2)$ steps, and regular width-2 problems in $O(n^3k^3)$ steps.

3. The advice generation part: we have demonstrated a simple method of extracting advice from easy problems to help Backtrack decide between pending options of value assignments. The method involves approximating the remaining part of a the task by a tree-structured problem, and counting the number of solutions consistent with each pending assignment. These counts can be obtained efficiently and can be used as figures-of-merit to rate the promise offered by each option.

3.3. THE SIMPLIFICATION PROCESS

The previous section suggests that a tree constraint-graph, being associated with an easy CSP, can be made a target for the simplification process from which advice will be extracted. We, therefore, discuss here the issues involved in approximating a network of binary constraints by a tree of constraints. We seek a good approximation since the closeness of the approximation tree to the original network will determine the reliability of the advice generated.

If the network R has an equivalent tree representation we would obviously like to recognize it and find such a representation. This, however may not be explicit in the constraint network; a network may contain many redundant constraints which, if eliminated, would still represent the same overall relation. For example, any one of the arcs in the network of figure 3.10 can be eliminated producing a tree-structured constraint graph representing the same relation. Note that in this figure, and throughout this section, there are multiple arcs between variables which connect values. Two values are connected if they are permitted by the constraint. Another example is given in figure 3.11 in which two 3-nodes networks, R_1 and R_2, are displayed. These two networks are equivalent, because they both represent the equality relation $\rho = \{(0,0,0),(1,1,1)\}$ and, unlike that of figure 3.10, both are maximal, that is, the addition of any pair of values to any one of the constraints (i.e relaxing any specific constraint) will result in a network representing a larger relation. Nevertheless, R_1 can be transformed into R_2 by simultaneously allowing the pair of values (1,0) between (Z,X) and disallowing the pair (0,1) between (X,Y). The question raised by this example is:

What networks have a tree representation and how to perform the transformation into

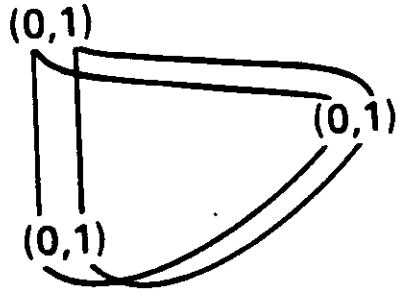


Figure 3.10 - A "redundant" CSP

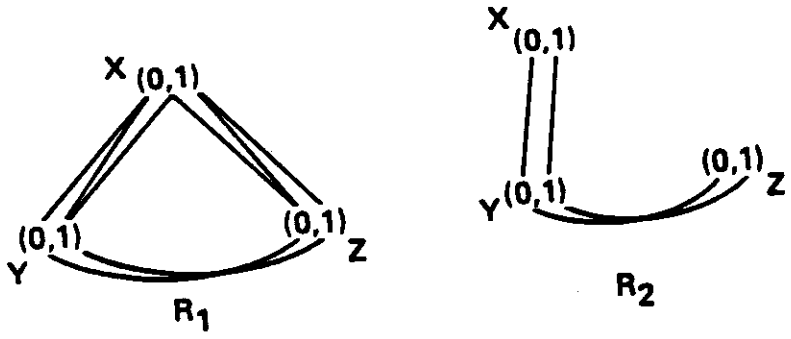


Figure 3.11 - Equivalent network of constraints

a tree?

The two examples given display two levels of operation to be considered in the process of transforming a network into a tree. The first is a macro operation involving the deletion of whole arcs (i.e. total elimination of constraints between a pair of variables) while the second is a micro operation, that merely modifies the arcs

by adding and deleting pairs of values. In our approach we will consider only macro operations of arc deletions; the use of micro transformations introduces a higher level of difficulty to which we will not relate at this point. Considering only arc deletions, a network R can be transformed into an equivalent tree only if some of the arcs are redundant, i.e. they represent constraints that can be inferred from others. This immediately raises the question of testing whether a given constraint is implied by others.

This question is the inverse of the one posed by Montanari [Montanari 1974] who claimed that the central problem in Constraint-Satisfaction Problems is the transformation of the original network R into its minimal representation, M, which is the **most redundant** network that represents the same relation as R. Our interest is the opposite, transforming R into one of its least explicit equivalent network.

Definition:

a. A network R is **maximal** if there is no network R'; $R \subseteq R'$ such that $R \sim R'$ ($\sim$ stands for equivalance)

b. A network R is **arc-maximal** if any arc deletion results in a network representing a larger relation.

A maximal network is arc-maximal but not necessarily vice-versa.

lemma 3:

An arc-consistent constraint tree is maximal.

proof:

In an arc-consistent tree, for any permitted pair of values there is an n-tuple in the relation which contains this pair. Disallowing this pair will eliminate such tuple from the relation, thus making the relation smaller. In other words any arc-consistent

constraint tree is minimal network for that relation.

□

An immediate conclusion is that arc-consistent tree-network is arc-maximal. In general, deleting an arc from a tree-constraint may result in a larger relation even when it is not arc-consistent. Let T_1 and T_2 be the two disconnected subtrees generated from deleting arc (A,B) and let ρ_1 and ρ_2 be the projection of ρ on the variables in T_1 and T_2 respectively. The relation obtained after deleting the arc (A,B) from T is the product of ρ_1 and ρ_2 (i.e. any n-tuple that is the concatenation of a tuple in ρ_1 and a tuple in ρ_2). Therefore if there is a tuple in ρ_1 with $A=a$ and a tuple in ρ_2 with $B=b$ then the relation resulting from deleting arc (A,B) permits the pair (a,b).

In most cases a CSP problem will not be arc-redundant, because if it is posed by humans its specification has already passed through some process of redundancy filtering, and therefore arc deletion will almost always generate larger relations. The third question on which we will focus, therefore, is:
Given a network of constraints, R, what is the spanning tree, T, that will best approximate R?.

To discuss the quality of approximations, the notion of closeness between relations must first be agreed on. Let ρ be the relation represented by R and ρ_a the relation represented by a relaxed network R_a, i.e., $\rho \subseteq \rho_a$. An intuitively appealing measure for the closeness of R to R_a may be:

$$M(R,R_a) = \frac{|\rho|}{|\rho_a|} \tag{13}$$

where $|\rho|$ is the number of n-tuples in ρ. This measure satisfies:

a. $M(\rho,\rho) = 1$

b. If $\rho \subseteq \rho_a \subseteq \rho_b$ then $1 \geq M(R,R_a) \geq M(R,R_b)$

M is a global property of two relations and the task of finding the spanning tree which yields the lowest M is very complex. Instead we propose a greedy approach: at each step delete the least "valuable" arc which leaves the network connected, namely, the arc deleted keeps the resulting network closest to the original one. To pursue this approach we need to define a measure of constraint strength, called weight, for each arc, that will estimate the contribution of that arc to the overall relation. Let R be a network of constraints and R' the network after the arc (X,Y) was eliminated, i.e. the constraint between X and Y becomes the universal constraint. Let l and l' be the size of the relations represented by R and R' respectively. $n'(x_i,y_j)$ is the number of tuples in the relation represented by R' having $X=x_i$ $Y=y_j$, $R'(X,Y)$ is the constraint induced by R' on the pair (X,Y), and $r(X,Y)$ is the local constraint given between X and Y in R. The following is satisfied

$$l = l' - \sum_{(x_i,y_j) \in R(X,Y)-r(X,Y)} n'(x_i,y_j) \tag{14}$$

therefore

$$\frac{l}{l'} = 1 - \sum_{(x_i,y_j) \in R(X,Y)-r(X,Y)} \frac{n'(x_i,y_j)}{l'} \tag{15}$$

Since we have no way of knowing the quantities $n'(x,y)$ and the structure of the induced constraint R'(X,Y), we will estimate them both by a constant,c and R' respectively. This gives:

$$\frac{l}{l'} = 1 - \frac{c}{l'} |R' - r(X,Y)| \tag{16}$$

The only quantity we can actually examine is $r(X,Y)$, therefore to maximize $\frac{l}{l'}$ the

above formula suggests choosing the constraint $r(X,Y)$ with the most number of allowed value-pairs. Our first measure of constraint-weight is, therefore, defined by:

$$m_1(X,Y) = |r(X,Y)| \tag{17}$$

For instance, the weight of the universal constraint is $m_1(X,Y) = k^2$, and if $r(X,Y) = \Phi$ then the weight becomes $m_1(X,Y) = 0$.

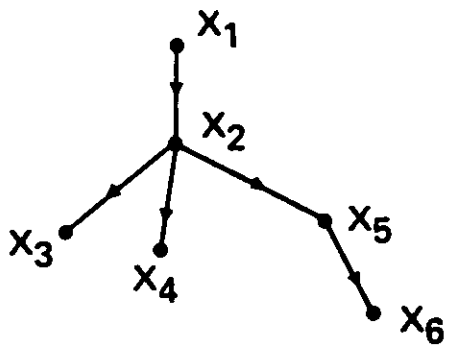
In what follows we develop another measure of constraint strength by adopting notions from probability and information theory and by showing that the problem of finding tree approximation can be partially mapped into the problem of finding a tree-structured joint probability distribution [Chow 1968].

n-ary relations and probability distributions.

Let $P(X)$ be a joint probability distribution of n discrete variables $X_1, X_2, \ldots, X_n$. A product approximation of $P(X)$ is defined to be a product of several lower-order distributions (also called marginal distributions) in such a way that the product is a probability extension of these lower order distributions. A particular class of product approximation considers only second order components where, each variable is conditioned upon at most one other variable. The relationships between the variables can be therefore represented by a tree. Given a directed spanning tree of the variables (the direction is from parents to sons) as in figure 3.12, the distribution function associated with it is given by the product:

$$P(X) = \prod_{X=(x_1,x_2,\ldots,x_n)} P(x_i \mid x_{p(i)}) \tag{18}$$

$p(i)$ is the parent index of variable X_i, and $P(x_0 \mid x_{p(0)}) = P(x_0)$. 0 denotes the root of the tree. Chow [Chow 1968] has shown that if the measure of distance between two probability distributions P and P_a is given by:



$$P(X) = P(X_1) \cdot P(X_2 \mid X_1) \cdot P(X_3 \mid X_2) \cdot P(X_4 \mid X_2) \cdot P(X_5 \mid X_2) \cdot P(X_6 \mid X_5)$$

Figure 3.12 - A tree-dependence distribution

$$I(P, P_a) = \sum_X P(X) \log \frac{P(X)}{P_a(X)} \tag{19}$$

then the closest tree-dependence distribution to P is the one that correspond to the maximum spanning tree when the weight of each arc is $I(X_i, X_j)$. $I(X_i, X_j)$ is Shanon's mutual information between X_i and X_j, defined by:

$$I(X_i, X_j) = \sum_{x_i, x_j} P(x_i, x_j) \log\left(\frac{P(x_i, x_j)}{P(x_i)P(x_j)}\right) \tag{20}$$

$I(P, P_a)$ can be interpreted as the difference of the information contained in $P(X)$ and that contained in $P_a(X)$ about $P(X)$.

Chow's results are remarkable in that a global measure of closeness can be maximized by attending to local measures on individual arcs. We therefore, attempt to adopt Chow's results to our need. Mapping probability distributions to constraints relations, we say that a relation ρ is associated with a distribution function P_ρ if:

$$P_{\rho}(x_1,x_2,\ldots,x_n)=\begin{cases}0 & \text{if } (x_1,x_2,\ldots,x_n)\notin\rho\\ \dfrac{1}{|\rho|} & \textit{otherwise}\end{cases} \tag{21}$$

Let ρ_t be the relation represented by a constraint-tree, t, and let P_{ρ} and P_{ρ_t} be the distributions associated with relations ρ and relation ρ_t, having sizes of l and l_t, respectively. The "distance" between the two distributions:

$$I(P_{\rho},P_{\rho_t})=\sum_{X\in\rho}\frac{1}{l}\log\frac{l_t}{l}=\log\frac{l_t}{l} \tag{22}$$

is a monotone function of $\frac{l_t}{l}$ whose inverse was already proposed as a measure of closeness between two relations (where one contains the other). Accordingly, finding the closest tree dependence distribution P_{ρ_t} to P_{ρ} will result in the closest approximation of a tree relation ρ_t to ρ. Equivalently, in order to minimize $\frac{l_t}{l}$ we need to find the maximum spanning tree w.r.t the measure $I(X_i,X_j)$. From the given mapping between relations and distributions (Eq. (21)) we get that:

$$P(x_i,x_j)=\frac{n(x_i,x_j)}{l} \tag{23}$$

$$P(x_i)=\frac{n(x_i)}{l} \tag{24}$$

where $n(x_i,x_j)$ is the number of tuples in ρ having $X_i=x_i$ and $X_j=x_j$, and $n(x_i)$ is the number of tuples in ρ with $X_i=x_i$. Substituting (23) and (24) in (21) we get

$$I(X_i,X_j)=\sum_{x_i,x_j}\frac{n(x_i,x_j)}{l}log\left(l\frac{n(x_i,x_j)}{n(x_i)n(x_j)}\right) \tag{25}$$

$$=logl+\frac{1}{l}\sum_{x_i,x_j}n(x_i,x_j)\log\frac{n(x_i,x_j)}{n(x_i)n(x_j)} \tag{26}$$

Consequently the appropriate measure of arc weight is:

$$m(X_i,X_j) = \sum_{x_i,x_j} n(x_i,x_j)\log\frac{n(x_i,x_j)}{n(x_i)n(x_j)} \tag{27}$$

The question now is how to obtain the quantities $n(x_i)$, $n(x_i,x_j)$ needed for computing m. To find them accurately, we need to inspect the list of tuples permitted by the global relation which, of course, is unavailable. In the case of probability distributions the marginal probabilities $P(x_i)$, and $P(x_i,x_j)$ are estimated by sampling vectors from the distribution and calculating the appropriate sample frequencies. This cannot be done in our case since finding even one tuple that satisfies the network solve the entire problem. All that we have available is the network of constraints and, therefore, we must approximate the weight $m(X,Y)$ by examining only properties of the arc (X,Y). This leads to approximations:

$$\hat{n}(x_i,x_j) = \begin{cases} 1 & (x_i,x_j)\in r(X_i,X_j) \\ 0 & otherwise \end{cases} \tag{28}$$

$$\hat{n}(x_i) = N_{X_j}(x_i) \tag{29}$$

Where $N_{X_j}(x_i)$ is the number of pairs in the constraint $r(X_i,X_j)$ with $X_i = x_i$, Substituting (28) and (29) in (27) we get:

$$m_2(X_i,X_j) = \sum_{(x_i,x_j)\in r(X_i,X_j)} \log\frac{1}{\hat{n}(x_i)\hat{n}(x_j)} \tag{30}$$

$$= -\sum_{(x_i,x_j)} (\log\hat{n}(x_i) + \log\hat{n}(x_j)) \tag{31}$$

$$= -\sum_{x_i}\hat{n}(x_i)\log\hat{n}(x_i) - \sum_{x_j}\hat{n}(x_j)\log\hat{n}(x_j) \tag{32}$$

The behavior of this measure can be illustrated in some special cases:

a. If the constraint $r(X,Y)$ is the universal constraint (and assuming k values for each variable) then $m_2(u(X,Y)) = -2\sum(k-1)logk = -2k(k-1)logk$

b. If $r(X,Y)$ is the empty constraint $\Phi(X,Y)$ then we define $m_2(\Phi(X,Y)) = 0$

c. If any value of X_i is allowed to go with exactly r values of X_j then $m_2 = -2k \cdot r \log r$. If $r=1$ we get $m_2 = 0$

d. when only one value in one variable is permitted with all the values of the other $m_2 = -k \cdot \log k$

We see that this measure considers not only the number of the pairs allowed but also their distribution over the k^2 slots available. For uniform constraint (like case c) it can be seen that

$$m_2 = -2N \cdot \log r \tag{33}$$

when N is the size of the constraint.

We next give an example showing the behavior of the accurate measure of weight, m, compared with their estimates, m_2.

Consider the relation between 3 binary variables, X,Y,Z, given by:

$$\rho = \{(1,1,1),(1,0,0),(1,1,0),(0,0,0)\} \tag{34}$$

where the order of the variables is (X,Y,Z). A network representing this relation is given in figure 3.13 where the nodes are the variables and the lines correspond to permitted pair of values between pairs of variables. The accurate measures of $n(x_i,x_j)$, and, $n(x_i)$ for the pair (X,Y) are given by:

$n(0,0)=1$, $n(0,1)=0$, $n(1,0)=1$, $n(1,1)=2$, $n(X=0)=1$, $n(X=1)=3$. Therefore, substituting in (29) we get:

$$m(X,Y)=\log\frac{1}{2}+\log\frac{1}{2\cdot 3}+2\log\frac{2}{3\cdot 2} = \log\frac{1}{108}$$

Similarly , for the two other pairs, we obtain:

$$m(X,Z) = \log\frac{4}{729}$$

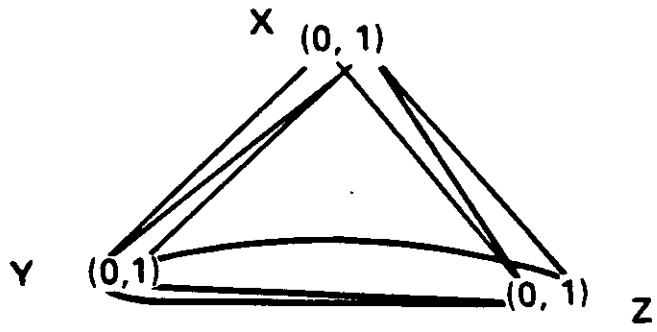


Figure 3.13 - A constraint network

$$m(Y,Z) = \log\frac{1}{108}$$

This suggest that the relation may be best approximated by a tree constituting of the arcs (X,Y) and (Y,Z). Indeed, the elimination of the arc (X,Z) will not change the relation at all whereas it is not possible to express ρ by removing either (Y,Z) or (X,Z) only.

By comparison, the network R and (35) give the weight estimates:

$m_2(X,Y) = -4, m_2(Y,Z) = -4, m_2(X,Z) = -4$

Which, in this case, fail to distinguish between the various constraints.

In conclusion, we suggest to generate tree-approximations for networks using the maximum spanning tree algorithm. Two measures for constraint-strength, to be used by the algorithm, are proposed and justified.

3.4. THE UTILITY OF THE ADVICE-GENERATION SCHEME

We compare here the performance of Advised Backtrack (abbreviated ABT) with that of Regular Backtrack (RBT) analytically, via worst case analysis, and experimentally, on a random constraint problem.

3.4.1 Worst case analysis

An upper bound is derived for the number of consistency checks performed by the algorithms as a function of the problem's parameters and the number of backtracks performed. A consistency check occurs each time the algorithm checks to verify whether or not a pair of values is consistent w.r.t. the corresponding constraint.

Let $\#B_A$ and $\#B_R$ be the number of backtracks, and $N(ABT)$ and $N(RBT)$ the number of consistency checks performed by ABT and RBT, respectively. The problem's parameters are n, the number of variables, and k, the number of values for each variable. Parameters associated with the constraint graph are $|E|$, the number of arcs, and deg, the maximum degree of variables in the graph.

The number of backtracks performed by an algorithm is equal to the number of leaves in the search tree which it explicates. We assume that

$$\text{Number of nodes expanded} = c \cdot \#B \tag{35}$$

approximately holds for some constant c. (This truly holds only for uniform trees where c is the branching factor.) Therefore we use the number of backtracks as a surrogate for the number of nodes expanded. Let $\#C_A$ and $\#C_R$ be the maximum number of consistency checks performed at each node by ABT and RBT, respectively. We have:

$$N \leq \#B \cdot \#C \tag{36}$$

Considering RBT first, the number of consistency checks performed at the i^{th} node in the order of instantiation is less then $k \cdot deg(i)$. That is, each of this variable's values should be checked against the previous assigned values for variables which are connected to it. We get:

$$N(RBT) \leq k \cdot deg \cdot \#B_R \tag{37}$$

The ABT algorithm performs all of its consistency checks within the advice generation. For the i^{th} variable , a tree of size $n-i$ is generated. The consistency checks performed on this tree occur in two phases. In the first phase, for each variable in the tree, the values which are consistent with the already assigned values are determined. The number of consistency checks for a variable v in the tree equal $k \cdot w(v)$, where $w(v)$ is the number of variables connected to v which were already instantiated. Therefore, for all variables in the tree, we have

$$k \cdot \sum_{v \in \text{tree}} w(v) \leq k \cdot |E| \ . \tag{38}$$

The second phase counts the number of solutions. We already showed that the counting takes no more then $(n-i) \cdot k^2$ which is bounded by nk^2. Hence,

$$N(ABT) \leq (k \cdot |E| + n \cdot k^2) \cdot \#B_A \tag{39}$$

We now want to determine the ratio between $\#B_A$ and $\#B_R$ for which it will be worthwhile to use Advised Backtrack instead of Regular Backtrack and, a first approximation, will treat the upper-bounds as tight estimates.

$$N(ABT) \leq N(RBT) \tag{40}$$

does not imply that

$$(k \cdot |E| + n \cdot k^2) \cdot \#B_A \leq k \cdot deg \cdot \#B_R \ , \tag{41}$$

however to get some feelings regarding the comparison between the two algorithms

we will assume that (41) is implied from (40). From (41) we get

$$\frac{\#B_R}{\#B_A} \geq \frac{|E|}{deg} + \frac{nk}{deg} . \tag{42}$$

Since

$$\frac{|E|}{deg} \leq n , \tag{43}$$

(42) will hold if

$$\frac{\#B_R}{\#B_A} \geq n + \frac{nk}{deg} . \tag{44}$$

(44) implies the inequality (41) between the upper bounds of $N(ABT)$ and $N(RBT)$ and we will allow ourselves to assume that it reflects the relationships between the above quantities, namely that (40) is implied from (44) baring in mind that it does not logically follows. Therefore, ABT is expected to result in a reduction in the number of consistency checks only if it reduces the number of backtracks by a factor greater than $(n+\frac{nk}{deg})$. Thus, the potential of the proposed method is greater in problems where the number of backtrackings is exponential in the problem size.

3.4.2 Experimental results

Test cases were generated using a random Constraint-Satisfaction Problem generator. The CSP generator accepts four parameters: the number of variables n, the number of values for each variable k, the probability p_1 of having a constraint (an arc) between any pair of variables, and the probability p_2 that a constraint allows a given pair of values. Two performance measures were recorded: the number of backtrackings ($\#B$) and the number of consistency checks performed. The latter being an indicator of the overall running time of the algorithm. What we expect to see is that the more difficult the problems, the larger the benefits resulting from using

advised Backtrack.

In our experiments we use m_1, the size of the constraint, as the weight for finding the minimal spanning tree. Using the alternative weight, m_2, is not expected to improve the results for two reasons. First, the problems generated were quite homogeneous and we have shown that for such problems both weights are the same. Second, the reduction in the number of backtrackings was so drastic that further improvements due to modified weights seems unlikely.

Two classes of problems were tested. The first, containing 10 variables and 5 values, were generated using $p_1 = p_2 = 0.5$, and the second with 15 variables and 5 values, generated using $p_1 = 0.5$ and $p_2 = 0.6$. 10 problems from each class were generated and solved by both ABT and RBT. The order by which the variables were instantiated was determined, for both algorithms, by the structure of the constraint graph. Namely, variables were selected in decreasing order of their degrees which closely correlate with the criterion of minimum width. [Freuder 1982] The order of value selection is determined by the advice mechanism in ABT and random in RBT. Therefore, while ABT solved each problem instance just once, RBT was used to solve each problem several (five) times to account for the variation in value selection order. When a problem has no solution, the number of backtrackings and consistency checks in RBT is independent on the order of value selection, and in these cases the problem was solved only once by RBT.

Figure 3.14 and figure 3.15 display performance comparisons for both classes of problems. In figure 3.14, the horizontal axis gives the number of backtrackings that were performed by RBT and the vertical axis gives the number of backtrackings performed by ABT for the same problem instance. The darkened circles correspond

to problem instances from the first class while empty circles correspond to instances of the second class. We observe an impressive saving in $\#B$ when advise is used, especially for the second class in which the problems are larger. Figure 3.15 uses the same method to compare the number of consistency checks. Here, we observe that in many instances the number of consistency checks in ABT is larger than in RBT, indicating that in these cases the extra effort spent in "advising" backtrack, was not worthwhile.

These results are consistent with the theoretical prediction of the preceding subsection. If we substitute the parameters of the first class of problems in (44) we get that $\#B_A$ should be smaller than $\#B_R$ by at least a factor of 20 (25 for the second class of problems) to yield an improvement in overall performance. Many of the problems, however, were not hard enough (in terms of the number of backtrackings required by RBT) to achieve these levels. In section 4.3.3 we give an average analysis of RBT performance on this class of problems and indeed we observe that on the average these problems are linear in n,i.e. easy to solve.

Figure 3.16 compares the two algorithms in only those problems that turned out to be difficult. it displays the number of consistency checks in the cases where the number of backtrackings in RBT were at least 70. We see that the majority of these problems were solved more efficiently by ABT than RBT.

Experiments were also performed on the n-queen problem for n between 6 and 15 and on the 3-colorability problem on a set of random graphs. In all cases the number of backtrackings of ABT was smaller than RBT, but the problems were not difficult enough to get a net reduction in the number of consistency checks.

As a result of the above experiments we felt that the "advising" scheme was too good for the tested problems, i.e. it gave very good results with a high cost. We wanted to weaken somehow the quality of advice in order to reduce its costs. The following scheme was devised: instead of generating advice based on the full spanning tree of the remaining problem we trimmed the tree to contain only a fixed number of variables, l. The computation of the number of solutions proceeded in the same way based only on this partial tree. This approach enabled us to test ABT using a whole range of strength of advice starting from a strong and full advice that considers all nodes in the tree, as before, through the weakest advice, having just one node, which is really no advice at all. Figures 3.17 and 3.18 summarize the results of experimenting with this **parametrized advice**. The x-axis display the strength of advice being used, i.e. the parameter l of the number of nodes considered in the tree. The left vertical axis gives the number of consistency checks and the right vertical axis gives the number of backtracking. The entries display the average performance of ABT with the corresponding amount of advice where figure 3.17 displays average w.r.t. problems in class 1 while figure 3.18 gives the results on problems from class 2. Full dot indicate the number of backtrackings, empty dot indicate the number of consistency checks. The cross dot represent the results with no advice, i.e, with RBT. As we see the amount of backtrackings reduces exponentially with l and the amount of consistency checks has a minimum for a relatively weak advice. In the first class of problem the advice based on just two nodes gave the best performance (on the average). In the second class of problems with n=15 an advice based on 3 nodes was best. In both cases the best performance of ABT was indeed better then RBT's performance. We conjecture therefore, that in general, problems in this classes will need only weak advice to enhance their performance considerably.

In conclusion, advice should be invoked with an appropriate strength. One needs therefore, a way of recognizing the difficulty of a problem instance prior to solving it. Knuth [Knuth 1975] has suggested a simple sampling technique that require very small computation to estimate the size of the search tree. These estimates can be used in conjunction with our parametrized advice that will adapt itself according to the expected size of the tree. Namely, smaller problems will be guided by a weaker advice (e.g. based on partial trees) that is obtained more efficiently.

Experiments related to the ones reported here were performed by Haralick et al. [Haralick 1980]. The Forward-Checking lookahead mechanism (reported to exhibit the best performance considering the number of consistency checks) can also be viewed as an automatically generated advice in the sense discussed here. However, since the task was to find all solutions, the results cannot be directly related.

3.4.3 average case analysis

We conclude by giving an upper-bound on the average complexity of regular backtrack on a random CSP.

Given a random CSP generated with the parameters p_1 and p_2, the probability, Q, that a pair of values is consistent is:

$$Q = 1 - p_1 + p_1 p_2 \tag{45}$$

The average number of nodes expanded when backtrack looks for one solution, denoted by $E(B_{one})$ is smaller then the average number of nodes expanded if backtrack looks for all solutions, denoted by $E(B_{all})$. i.e.

$$E(B_{one}) \leq E(B_{all}) \tag{46}$$

We will therefore, find a bound to $E(B_{one})$ by bounding $E(B_{all})$.

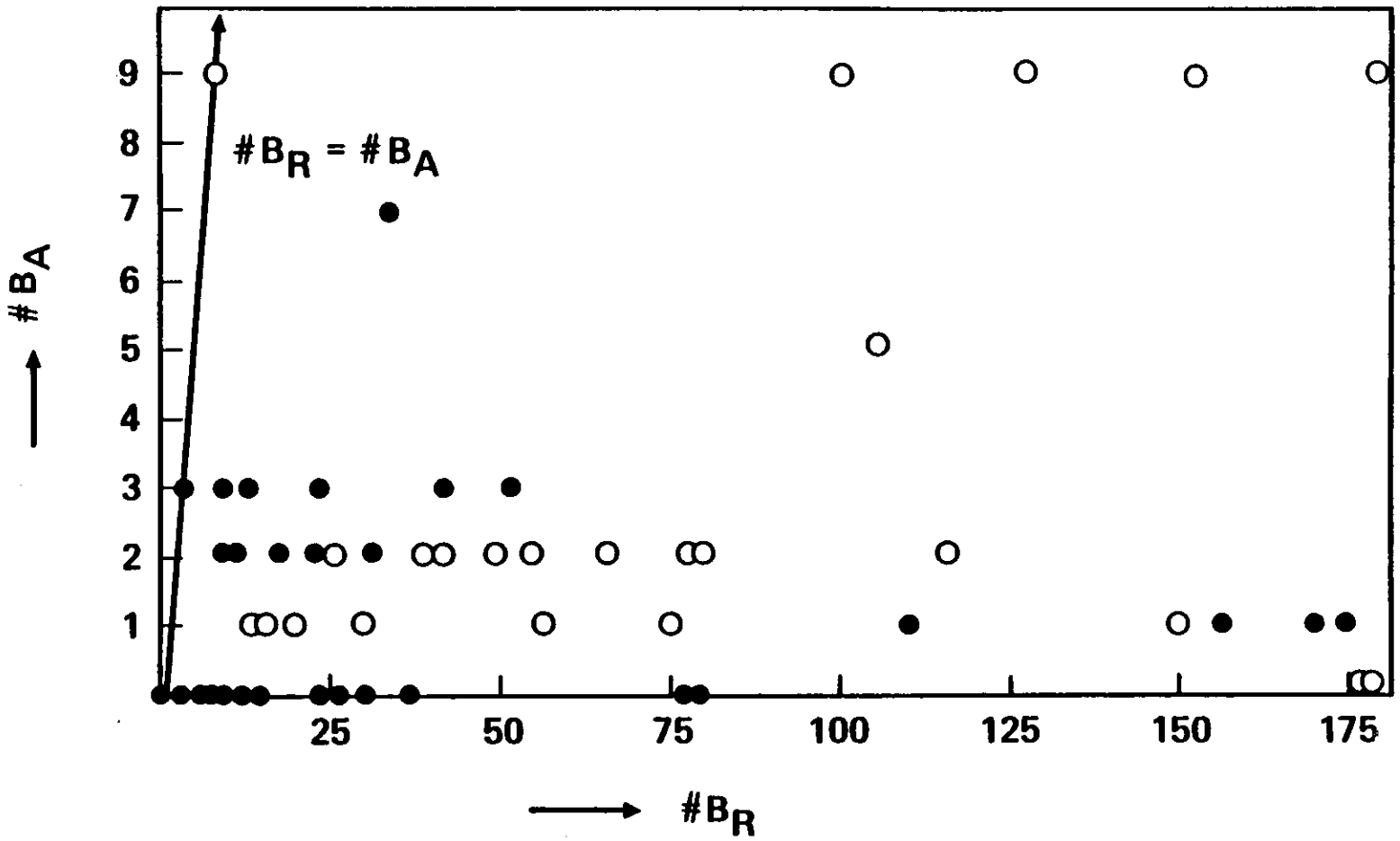


Figure 3.14- Comparison of #backtrackings in ABT and RBT

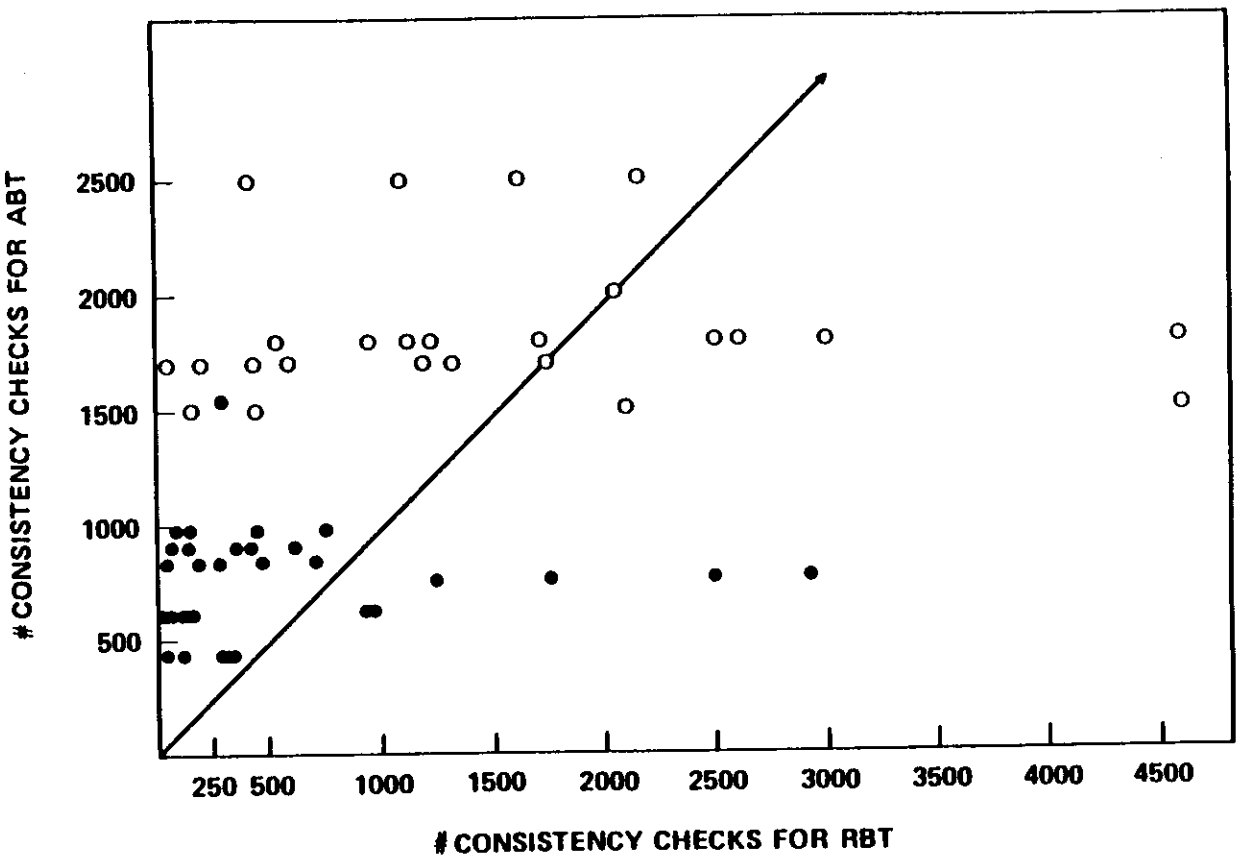


Figure-3.15- Comparison of #consistency checks in ABT and RBT

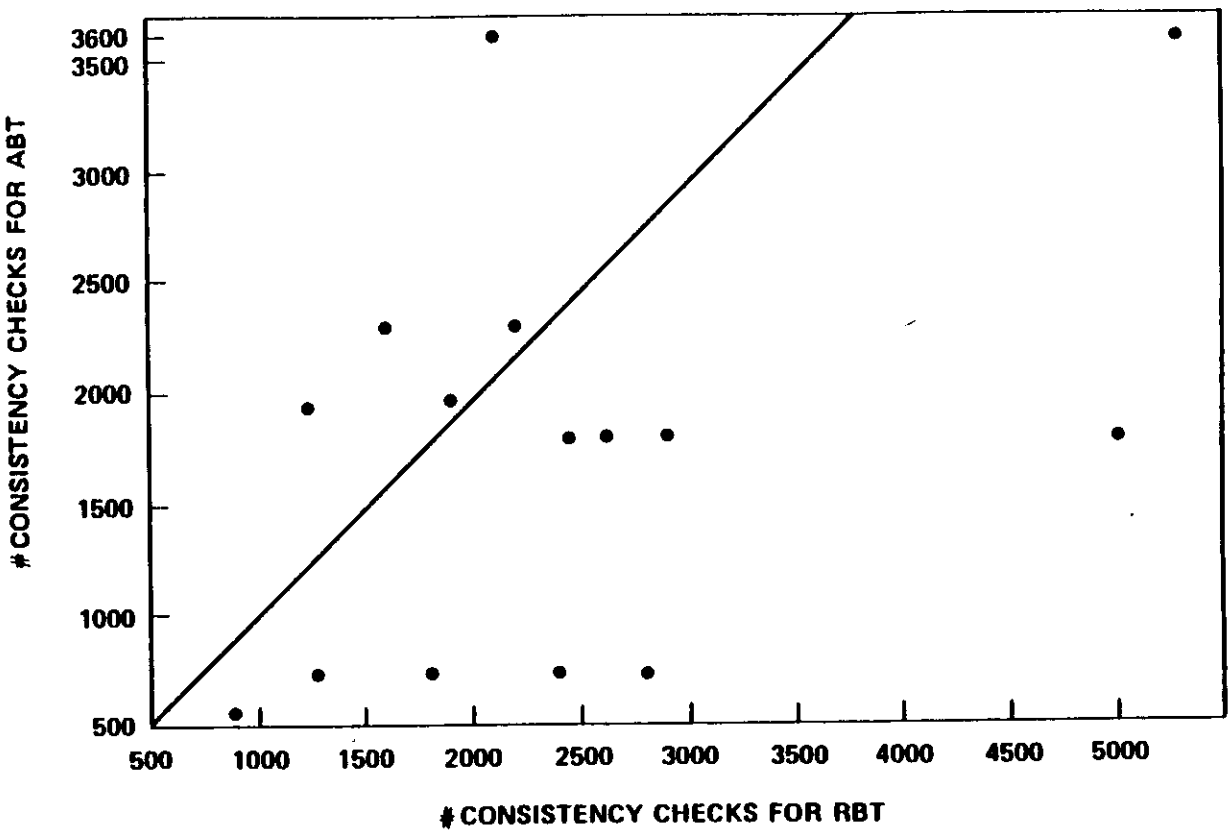


Figure 3.16- Comparing #consistency checks on difficult problems

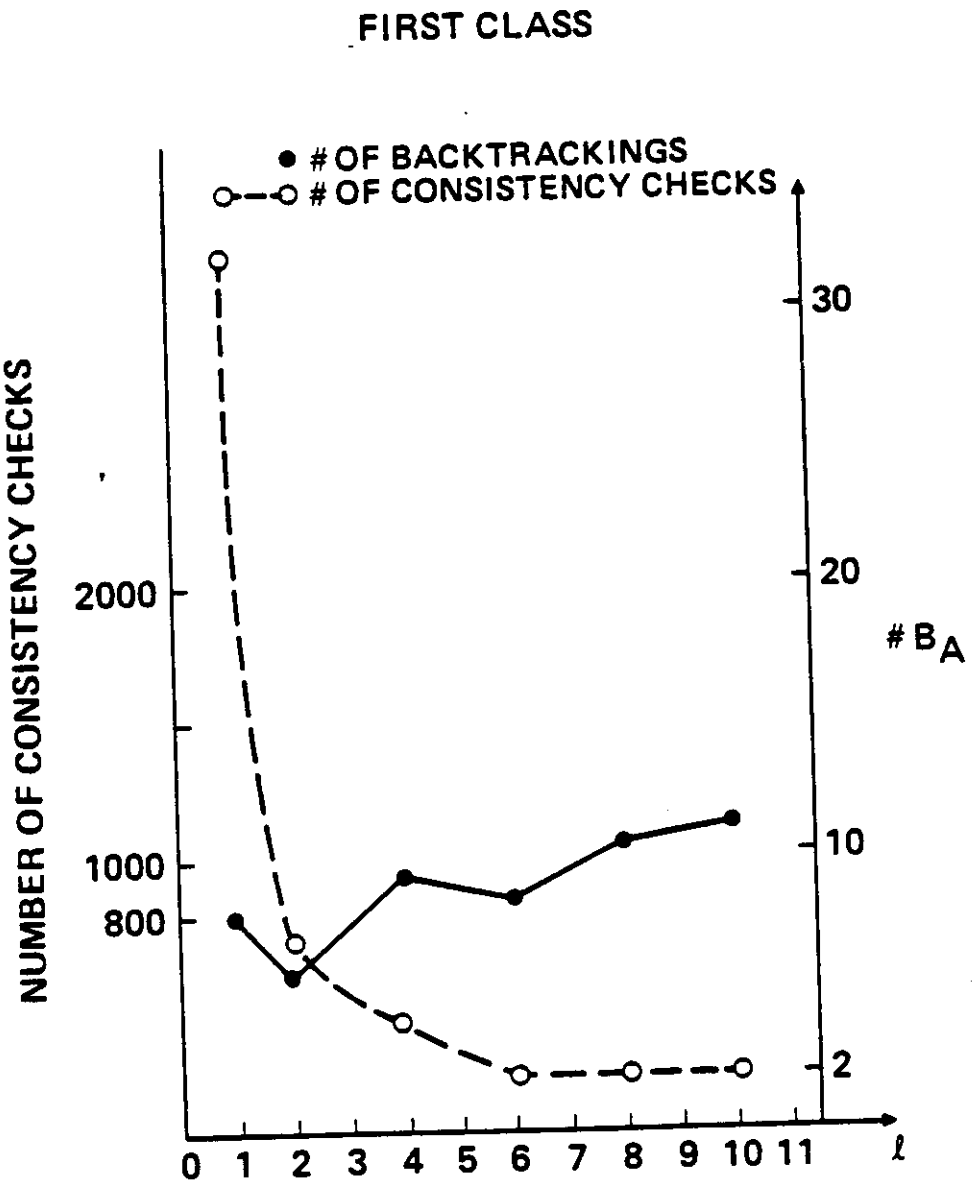


Figure 3.17- #consistency checks and #backtrackings with parametrized advice (first class of problems)

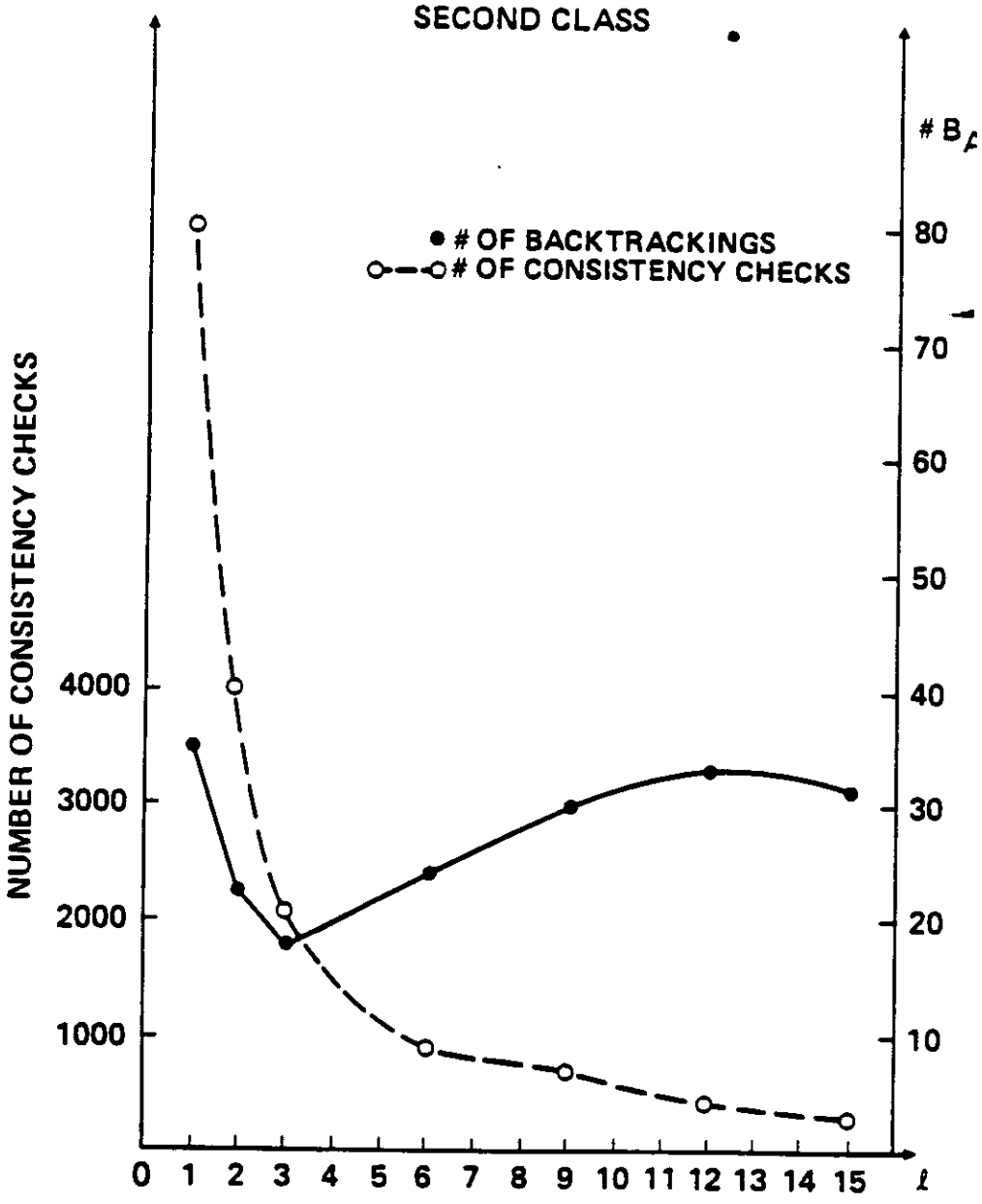


Figure 3.18- #consistency checks and #backtrackings with parametrized advice (second class of problems)

Given a fixed order of variables' instantiation, the probability that the j^{th} value in the sequence is consistent with all the previous $j-1$ values is S^{j-1}. Therefore, the probability that a node at depth l in the search tree will be expanded is: (see also [Haralick 1980]):

$$p(X \geq l) = Q^{\frac{(l-1)l}{2}} \tag{47}$$

where X is a random variable indicating the length of a consistent set of values. Since the number of possible nodes at level l is k^l, we get:

$$E(B_{all}) = \sum_{l=1}^{n} k^l Q^{\frac{(l-1)l}{2}} = \sum_{l=1}^{n} (kQ^{\frac{(l-1)}{2}})^l \tag{48}$$

Let l_{max} be the level in the search tree that has the maximum average number of nodes. Replacing each term in (48) by the highest, we get:

$$E(B_{all}) \leq n \cdot (k \cdot Q^{\frac{(l_{max}-1)}{2}})^{l_{max}} \tag{49}$$

l_{max} can be found by differentiating the function:

$$f(l) = k^l Q^{\frac{(l-1)l}{2}} \tag{50}$$

yielding

$$l_{max} = -\frac{\ln k}{\ln Q} + \frac{1}{2} \tag{51}$$

If $l_{max} \leq n$ we have:

$$E(B_{all}) = O(n \cdot (k^{c(k,Q)} \cdot Q^{(-\frac{1}{2} + c(k,Q))c(k,Q)})) \tag{52}$$

where

$$c(k,Q) = \frac{\ln k}{\ln \frac{1}{Q}} \tag{53}$$

For a fixed k and Q we see that, asymptotically, the average complexity is linear in n. For a fixed Q we have:

$$E(B_{all}) = O(n \cdot k^{c(k,Q)}) = O(n \cdot k^{\frac{1}{\ln\frac{1}{Q}}} \cdot k^{\ln k}) = O(n \cdot k^{\ln k}) \qquad (54)$$

No wonder, therefore, that most instances generated by our model were not too dificult . On the average these problems are linear in n and $O(k^{\ln k})$ in k.

APPENDIX 3.1: Lower bound to the complexity of tree-CSP

We will show that any algorithm that guaranteed to solve any tree-CSP with n variables and k values requires $(n-1)k^2$ steps in the worst case. We assume that the basic step of the search algorithm consists of testing the consistency of a pair of values. The algorithm terminates with the first solution found or by concluding that no solution exist. The proof uses an oracle which, for any given algorithm, creates a problem instance on which that algorithm must spend $(n-1)k^2$ steps.

Consider tree problems that have a star structure as depicted in the figure 3.19.

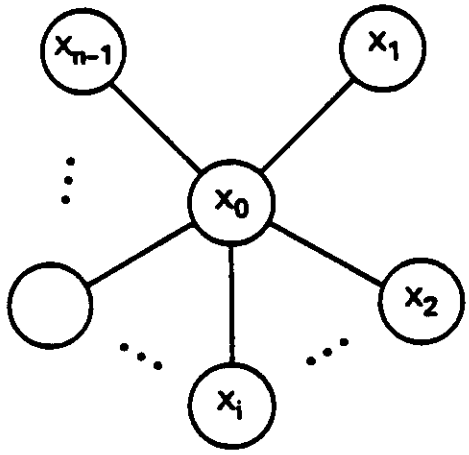


Figure 3.19 - A star constraint-tree

X_0 is the center variable and $X_1, \ldots, X_{n-1}$ are all connected to it. Let v represent an arbitrary value of the center variable. The idea is to have the oracle answer the tests so that, for each v, it will not be possible to conclude the existence of a consistent solution without checking v against all possible values of all the peripheral variables. Since there are $(n-1)k$ such values for each value of X_0, this will prove the claim.

Oracle specification:

At any point of time, the information available to the algorithm (and the oracle) consists of three sets of pairs: positively determined, negatively determined and undetermined. In response to the query $R(X_i,x,X_0,v)$? (testing if the pair (x,v) is consistent) the oracle responds as follows:

a. If for X_i and v there are still undetermined pairs (X_i,y,X_0,v) , $y \neq x$, the answer is "NO", else

b. the response is "YES" unless

c. v in X_0 has a positively determined matching value in each of the other $n-2$ variables (excluding X_i). In which case the answer is again "NO".

This oracle creates a problem instance with no solution so that any algorithm will have to check all pairs of values. Notice that case (b) is executed w.r.t. v and a specific variable X_j, only when all the pairs of a value in X_j and v were determined. After this there is one positive match to v in X_j and all the other pairs are negatively determined. Therefore when case (c) is executed with regard to value v of X_0 all the values which are connected to v are determined and there are exactly one matching value to v in each of the other variables. Part (c) ensures that there will not be a full solution consistent with v.

Suppose that the claim is not true, i.e., that the algorithm halts while **there are still undetermined pairs. Let v be a value of X_0 which participate in one or more of these undetermined pairs. Any variable X_i whose values were all determined w.r.t v must have a positive match with v (due to case (b)). For all other variables we

make the undetermined pairs** positive, thus creating a problem instance that has a solution which the algorithm fails to detect. This yields a contradiction.

□

CHAPTER 4

GREEDY ALGORITHMS AND HEURISTICS

4.1 INTRODUCTION AND MOTIVATION

When one tries to extend the idea that heuristics can be generated by consulting simplified models to optimization problems, one must address the following question: what should be the target of the simplification process? Or put differently: what constitutes an easy optimization problem? Pearl [Pearl 1983] has suggested that a possible criterion for an easy problem is that it is solvable with no backtracking. The easy problems used in Chapter 3 for generating advice for constraint satisfaction problems (i.e., problems with a tree constraint-graph) indeed meet this criterion. In the context of optimization problems this same criterion translates to problems that can be solved (optimally) by a **Greedy algorithm.** We call such problems **greedily optimized.**

Greedy algorithms use an irrevocable search control regime that uses local knowledge to construct a global solution in a "hill climbing" process [Nilsson 1980]. Usually, they involve some real-valued function defined on the states of the search space. The greedy control strategy selects the next state so as to achieve the largest possible increase in the value of this function.

Of the various search control strategies available (e.g., backtracking, best-first, greedy), greedy schemes are probably the closest to explaining human problem-solving strategies because they require only a minimum amount of memory

space and because they often produce adequate results. (Due to the small size of human short-term memory, it is very hard to conceive of a human conducting best-first or even backtracking search, both requiring retention of some properties of previously suspended alternatives.) The study of greedy algorithms and of greedily optimized problems is, therefore, important not only for the enhancement of weak methods but also for understanding human reasoning in general.

The "nearest neighbor" strategy for obtaining solutions for the Traveling Salesman problem, is a well-known example of a greedy scheme that generally does not yield an optimal solution. Nevertheless, this is perhaps the most intuitively appealing and the first choice of most humans faced with this problem. Moreover, the solutions produced by this strategy are often of acceptable quality. For example, Reingold [Reingold 1977] proved that the ratio between the answer found using this method and the optimal answer is less or equal to $lg\,(n+1)/2$ (where n is the number of cities), and empirical evidence [Bentley 1980] suggests that it finds tours that are about 20% worse then optimal on the average when the cities are distributed at random.

There are two ways in which greedy schemes can play a role in the solution of optimization problems. The first is using a "greedy solution" in lieu of a true optimal solution of the problem, as in the example above. The main issue in this case is the choice of the best hill-climbing function, namely, the one resulting in the least expected amount of suboptimization. This is the subject of Section 4.2 where the relative merits of several such functions for the two-constraint knapsack problem are discussed.

The second way is using greedily optimized simplifications of the original problem for stregthening a weak method algorithm (A^*, backtrack). For instance, when A^* is utilized for solving instances of the Traveling Salesman problem, a possible source for the heuristic evaluation function is the optimal value of a Minimum Spanning Tree (MST) problem obtained by relaxing the TS problem. The attractiveness of the MST problem in this case stems to a large extent from it being greedily optimized. For this type of use, and in particular if the generation of heuristic advice is to be mechanized, an issue of utmost importance is that of being able to identify greedily optimized problems and their ranking functions. A necessary step toward this objective is the characterization of greedily optimized problems, which is the subject of Sections 4.3 and 4.4.

Our takeoff point is a list of optimization problems known to be greedily optimized that appear, along with their respective greedy strategy, in the appendix to this chapter. All the problems on the list involve the task of selecting, from a given set of elements, a subset of the elements which satisfy some property, so as to maximize (or minimize) the value of a cost function defined on all possible solutions.

A careful examination of the list of problems reveals that they fall into three categories which we will refer to as **selection-problems**, **ordering-problems**, and **tree construction problems**. If the cost function is not dependent on the order of the elements in the subset of elements that constitute a solution, then a selection problem is at hand. An example is the Minimum Spanning Tree problem (#1 in the list). In an ordering problem the value of the cost function is is dependent on order of the elements as well as on their identity. An example is the Optimal Storage on Tape problem (#4). Tree construction problems fall somewhere between the other two categories. These problems require the generation of a tree which imply a partial

order on the set of elements (see, for example, the Optimal Merge Patterns problem, #6).

The solution of selection and tree-construction problems by greedy algorithms has been studied extensively. Section 4.3 summarizes the relevant results for these categories. Specifically, Section 4.3.1 describes the work of Edmonds and Gale [Lawler 1976] on the relationships between matroid theory and greedy algorithm and Section 4.3.2 summarizes the work of [Parker 1980] which characterizes the set of cost function defined on trees that are minimized using the Huffman algorithm (which is a greedy procedure).

No comparable body of knowledge exists for characterizing greedily optimized ordering problems. An attempt to do so, the Greedoids theory [Korte 1981] fais to capture a large set of ordering problems. A theory intended to feel this gap is presented in Section 4.4, in which we characterize cost functions that permit optimal solutions of ordering problems by a greedy algorithm.

We conclude this introduction by mentioning work on greedy algorithms related only indirectly to the discussion of this chapter. Sahni and Horowitz [Sahni 1978] devoted a chapter in their book to greedy algorithms, containing a collection of problems that can be solved by a greedy algorithm.

A generalization of the Matroid structure (section 4.3), called Greedoids [Korte 1981], represents an attempt to capture an even larger set of problems that can be solved by a greedy algorithm. However, there are greedily optimized ordering problems that cannot be patterned after neither matroid nor greedoid theories (e.g., The "average weighted processing time", in section 4.3).

Some authors had also investigated the conditions under which some classical optimization problems will be solved by greedy algorithms. In particular Dunstan et.al. [Duntsan 1973] discussed linear programming and Magazine et al. [Magazine 1975] discussed a class of knapsack problems.

For NP-Complete problems greedy algorithms were proposed as approximation algorithms and in some cases their performance was analyzed by comparing the cost of the solution obtained and the optimal cost. Garey and Johnson discuss such suboptimal algorithms in their book [Garey 1979]. Probabilistic analysis of the performance of random versus greedy strategies is reported by [Ausiello 1981] for the knapsack and clique problems.

4.2 UTILIZING GREEDY SCHEMES

Consider the following hypothetical story.

Joseph's Dilemma

Joseph (Jacob's youngest and most favored son) wants to bring some food to his brothers. He searches his father's storage and finds six Items: a pair of lamb's legs, a crate of dry fish, a box of pita-bread, a can of olives, a chunk of cheese and a can of milk. The items are clearly marked with their nutritious value (in calories), their volume (in square feet) and their weight (in pounds) as follows:

item	1	2	3	4	5	6
value	4	10	12	2	4	5
volume	2	6	20	10	1	4
weight	16	48	4	8	20	20

Wishing to bring the most valuable combination to his brothers, Joseph would like to take everything, but he is limited as to the total volume he can carry (20 cu. feet) and the total weight he can carry (80 pounds), so some items should be given up.

How should he go about selecting the items so as to maximize his brothers' appreciation?!

Needless to say that Joseph, unequipped with the Tools of Operations-Research and Combinatorial Optimization will not formulate the problem as an optimization problem with two constraints and probably would be unable to come up with an optimal selection within the time allowed by the circumstances. Nevertheless it is not hard to imagine that Joseph, given his natural resourcefulness and common-

sense approach to life would be able to select a fairly good composition. He begins to execute the following strategy: Consider all the items, choose the one that looks most worthy, then add to it the most worthy of the remaining items and so on until either the volume or the weight limit is exhausted. He is not aware (and would probably be offended if he were) of the fact that 3000 years down the road this strategy would be known as the **Greedy strategy.**

The problem Joseph is facing now is how to determine what the "most worthy" item at each step means. Since the final merit of each item lies in its nutritious value he first tries to use this as a selection criterion and thus he picks item 3 as the first choice. He immediately observes that no other item can be added because the entire volume capacity (20 cu. feet) is taken up by this item, yielding a total value of only 12 calories.

Joseph suspects that this highest-value-first criterion is not the best selection strategy (one might say that it is too greedy). Therefore other selection criteria accounting for both size and weight may be tempted, for example, choose the item of lowest size as long as its value is above some threshold. The intention of Joseph in this scheme is to maximize the number of items chosen since he bears in his mind that as more items are selected, the larger will be the total value. Later he realizes that the thing that causes him to regret the selection of a promissing item is the fact that it inhibits the selection of another, equally promissing. Moreover, the feeling of regret occur primarily when the first item consumed a large fraction of either the remaining volume or the remaining weight allowence. Therefore for each item he calculates the two normalized parameters of weight and size $\frac{s_i}{S_r}$ and $\frac{w_i}{W_r}$ where s_i and w_i denote the size and weight of item i. S_r and W_r are the residual total size and

weight, and the criterion indicating the relative amount of resource consumed is:

$$\max\{\frac{s_i}{S_r}, \frac{w_i}{W_r}\}$$

henceforth "index of resource-consumption". His scheme in this case might be: Choose the next item with the smallest index of resource-consumption, which is a parameter updated after each selection, according to the residual available size and weight.

After further pondering with this problem Joseph realizes that the criterion above must also take into account the nutritious value of each item, or else he may hypothetically fill his bag with very small, light, but worthless collection. For each item he calculated the two ratios $\frac{p_i}{s_i/S_r}$ and $\frac{p_i}{w_i/W_r}$ where p_i is the value of item i. Joseph can now try to use each one of the ratios above separately as his selection criterion, or try to combine the two measures by some rule, perhaps the pessimistic rule used before.

What is the mental process involved in the generation and refinement of such scheme and criteria? What knowledge and representations are required for mechanizing such processes? These are the kind of questions that have motivated this study. We will show now that most of the ranking functions suggested in the story, if used by a greedy algorithm, can in fact lead to an optimal solution of some simplified version of the original problem. This suggests that problem simplification plays an important role in such learning processes.

A simplified problem is a problem similar to the original, in which some components are changed to render the problem easy to solve, e.g. a constraint is added or deleted, the cost function is altered etc. The following list displays the ranking

functions suggested in the story:

1. p_i

2. w_i

3. s_i

4. $\max(\frac{s_i}{S_r}, \frac{w_i}{W_r})$

5. $\frac{p_i}{w_i}$

6. $\frac{p_i}{s_i}$

7. $\min(\frac{p_i}{s_i/S_r}, \frac{p_i}{w_i/W_r})$

The main components in the problem are: the cost function, the size constraint, the weight constraint, and the integer constraint. (The latter refers to the constraint that elements cannot be divided, each element is either selected in its entirity or not selected at all.) The ranking functions in the above list yield a greedy algorithm optimal for one of the following simplified problems:

1. If the size and weight constraints are deleted and instead we have the constraint: "The number of elements must be less or equal to N", then this simplified problem is optimally solved using $\{p_i\}$ to rank the selection of elements.

2. If the cost function is simplified and instead of evaluating the combination in

the sack by its total nutritious value we evaluate it by the number of elements included, and if the size constraint is deleted, then this new simplified problem is optimaly solved by function 2 (w_i). Function 3 (s_i) can be explained in a similar way.

3. Function 5, $\frac{p_i}{w_i}$, (resp. 6, $\frac{p_i}{s_i}$) solves optimally the problem in which both the integral constraint and the size (resp. weight) constraint are deleted.

4. The most interesting of all is ranking function 7. It is possible to show that it solves optimally a continuous version of the knapsack problem. In other words, if we can add to the sack any fractional quantity ϵ, from each element, as small as we wish, then adding this quantity from the element suggested by function 7 and updating the remaining size and weight will give an optimal solution (assuming the quantities available from each element exceed the size and weight limits). We conjecture that this strategy is also optimal when the resources available from each element are limited. The simplified problem in this case is obtained by deleting the integral constraints. Notice that the forth ranking function can also be regarded as solving optimally this model with the added constraints on the input instances that the values of all the elements are equal.

The availability of several ranking functions to a greedy strategy of a given problem points to the need of comparing these functions and measuring their "goodness". Intuitively, we expect the merit of a given ranking function to increase with the "closeness" between the original problem and the simplified problem. A partial order w.r.t closeness to the original problem is given in figure 4.1. We say that prob-

lem B is closer to P then problem A if every component in B which differ from P is included in A but not vice-versa.

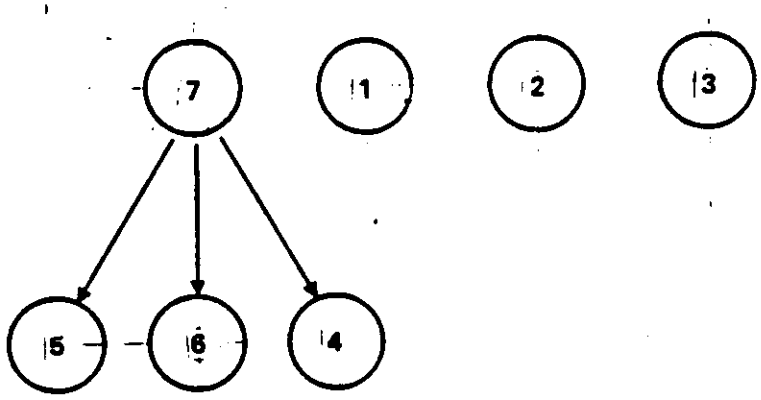


Figure 4.1 - Hierarchy of simplified problems

As we see, the only relationships we could establish in this way is that function 7 is better then 4, 5 and 6. All other ranking functions are incomparable. We do feel, however, that function 4 should be considered better then 2 and 3 since it takes more parameters of the problem into consideration (for that reason we might also feel that function 7 is the best) but, the superiority of 4 cannot be guaranteed for every problem instance, nor does the superiority of ranking function 7.

Another way of using the advice when several uncomparable simplified models induce different ranking functions, is to associate each ranking function with a different classes of instances (of the original problem). For each simplified problem and its ranking function we delineate a class of instances of the original problem

that can be optimally solved by it. For example, the effect of deleting the weight constraint is equivalent to the requirement that the weights of all the elements are equal. Similarly, changing the cost function to the "number of elements" (instead of the "total value") is equivalent to requiring that the values of all elements are equal. We can therefore consider the following classes of problem instances:

1. $\forall i,j\ p_i = p_j$

2. $\forall i,j\ w_i = w_j$

3. $\forall\ i,j\ s_i = s_j$

4. $\forall\ i\ \frac{w_i}{W} < \varepsilon$

5. $\forall\ i\ \frac{s_i}{S} < \varepsilon$

6. $W = S$

The ranking function p_i will optimally solve the original problem on the set of input instances satisfying requirements 2,3, and 6. Function 7 will solve optimally (if ε is small enough) the original problem for problem instances satisfying 4 and 5,etc. Figure 4.2 gives the associations between each ranking function and the set of instances that it solves optimally. The class of instances is denoted by a conjunction of the original problem with several of the requirements on the inputs (denoted by their numbers).

Ranking function	Instances
p_i	P + (2) + (3) + (6)
w_i	P + (1) + (3)
s_i	P + (1) + (2)
$\max\{\frac{s_i}{S}, \frac{w_i}{W}\}$	P + (1) + (4) +(5)
$\frac{p_i}{w_i}$	P + (4) + (3)
$\frac{p_i}{s_i}$	P + (5) + (2)
$\min(\frac{p_i}{s_i/S_r}, \frac{p_i}{w_i/W_r})$	P + (4) + (5)

Figure 4.2 - Mapping ranking functions to sub-domains of instances

This approach suggests to decompose the set of instances to different sub-domains and to associate with each such sub-domain the most appropriate ranking function as in figure 4.2. For example, if in a given problem instance, all the elements are big but do not differ much in size and weight, then function 1 is appropriate. If the elements are all very small, then function 7 will be appropriate.

The main steps of this approach are outlined next and the general idea is illustrated in figure 4.3.

Given a problem P do the following:

1. Check if problem P is greedily optimized. If it is, then apply the appropriate technique for identifying the correct ranking function and stop.

2. If P is not greedily optimized then generate a set of simplified problems which are greedily optimized.

3. For each such problem find the ranking function that optimizes it.

4. Map each simplified problem into a corresponding subset of problem instances of P as explained before.

5. Generate a ranking rule that associate classes of instances with ranking functions.

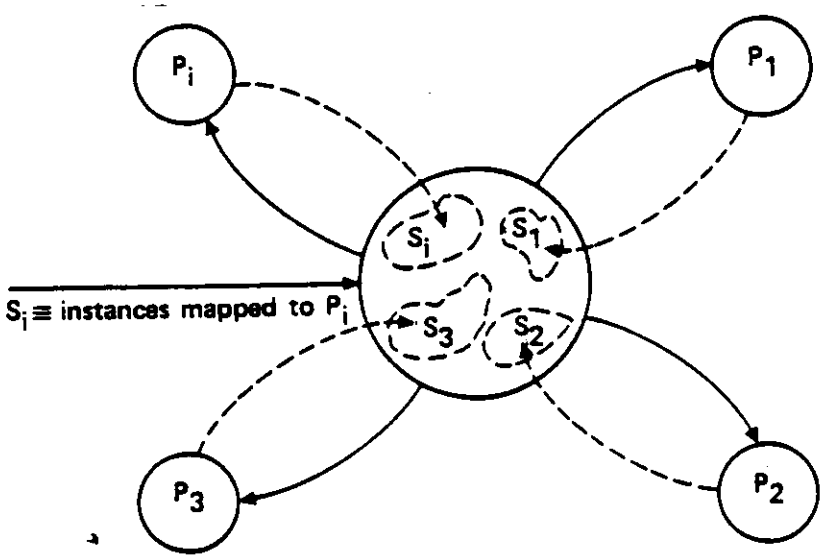


Figure 4.3 - A scheme for generating ranking functions

4.3 GREEDILY OPTIMIZED SELECTION AND TREE-CONSTRUCTION PROBLEMS

In sections 4.3 and 4.4 we present some characteristics of combinatorial optimization problems which can be solved by a greedy algorithm. Among those are the "Minimum spanning tree" problem, the "Optimal storage on tape" problem, "Minimum average flow in one machine job shop scheduling" and more. This set of problems will be called Greedily-optimized problems and a list of such problems is given in section 4.5.

The main source of knowledge to characterize greedily optimized selection-problems is the theory of Matroids. We give a brief summary of the matroid theory which relates matroids with greedy algorithms following with some conclusions.

Matroid theory deals with problems having a finite set of elements, each associated with a real number called weight. The cost of a subset of elements is the sum of their weights. Subsets which satisfy some given constraints are regarded as solutions. A greedy algorithm constructs a solution by choosing the elements in order of decreasing weight as long as the constraint is satisfied. According to the matroid theory, the greedy algorithm will always find a solution with maximum cost if the set of solutions satisfies two properties. The first property is that every subset of a solution is also a solution. The second property is more involved and has to do with the cardinality of a solution (i.e. the number of elements in the set). It requires that if there are two solutions s_i and s_j with cardinality i and j respectively and i>j then it is possible to transfer an element in $s_i - s_j$, to s_j such that the resulting set will be a solution with j+1 elements. A more formal discussion of the matroid theory is given below. It is taken from the book "Combinatorial optimization by Lawler [Lawler

1976]

4.3.1. The theory of matroids and greedy algorithms.

definition: A matroid $M=(E,\Phi)$ is a structure in which E is a finite set of elements and Φ is a family of subsets of elements of E such that:

M.1 $\phi \in \Phi$ and all proper subsets of each set in Φ are in Φ.

M.2 If I_p and I_{p+1} are sets in Φ containing p and $p+1$ elements respectively, then there exists an element e in $I_{p+1} - I_p$ such that $I_p + e$ is in Φ.

The sets of Φ are called independent sets. An independent set is said to be maximal if it is not contained in any other independent set. A maximal independent set is said to be a base of the matroid, and the rank, $r(A)$, of a subset $A \subseteq E$ is the cardinality of a maximal independent subset of A.

Examples:

1. Matroid of matrix A: E is a set of columns of A, and Φ contains all linearly independent subsets of E.

2. Matroid of a graph: E is the set of edges of graph G and Φ contains all cycle-free subsets of edges.

3. Matching matroid: Let $G = (N,A)$ be a graph and E be any subset of nodes in N. Let Φ be the family of all subsets $I \subseteq E$ such that there exist a matching which covers all the nodes in I. $M = (E,\Phi)$ is a matroid called a matching matroid.

Theorem 1: (theorems will be stated without their proofs).

Let Φ be the family of independent sets of a matroid. Then :

T.1 For any $A \subseteq E$, if I and I' are maximal subsets of A in Φ, then $|I| = |I'|$.

Conversely, if $M = (E,\Phi)$ is a finite structure satisfying M.1 and T.1, then M is a matroid.

□

Theorem 1 asserts that for any $A \subseteq E$, all maximal independent sets in A have the same cardinality. The rank function $r(A)$ is thus well defined.

Theorem 2: (Whitney)

Let Ω be the set of bases of a matroid. Then:

T.2 $\Omega \neq \phi$ and no set in Ω contains another properly.

T.3 If B_1 and B_2 are in Ω and e_1 is any element of B_1, then there exists an element e_2 in B_2 such that $B_1 - e_1 + e_2$ is in Ω.

Conversely, If (E,Ω) is a finite structure satisfying T.2 and T.3 then $M = (E,\Phi)$ is a matroid, where $\Phi = \{I \mid I \subseteq B$, for some B in $\Omega\}$

□

Let $M = (E,\Phi)$ be a matroid whose elements have been given weights, $w(e_i) \geq 0$. The triplet (E,Φ,W) in which W is a vector of weights associated with the elements of E, is called a **weighted matroid**, and an independent set with maximum sum of weights is desired. Any weighting of the elements induces a lexicographic ordering on the independent sets, when the elements in each set are ordered decreasingly w.r.t. their weights.

Theorem 3: (Rado, Edmonds)

Let Φ be the family of independent sets of a matroid then:

T.4 For any nonnegative weighting of E, a lexicographically maximum set in Φ has a maximum weight.

Conversely, if $M = (E, \Phi)$ is a finite structure satisfying M.1 and T.4 for any set of weights then M is a matroid.

□

A lexicographically maximal base can be found by the **Matroid Greedy algorithm:** namely, choose the elements of the matroid in order of size, weightiest element first, rejecting an element only if its selection would destroy independence of the set of elements. Note that If B is a lexicographically maximum base and I is any other independent set, then the weight of the k^{th} largest element of B is not smaller then that of the k^{th} largest element of I, for all k. We say that B in Φ is gale optimal in Φ if for any other set I in Φ there exists a one to one mapping $h : I \rightarrow B$ such that $w(e) \leq w(h(e))$ for all e in I.

Theorem 4: (Gale)

Let Φ be the family of independent sets of a matroid. Then:

T.5 For any weighting of the elements in E, there exists a set B which is gale optimal in Φ.

Conversely, if $M = (E, \Phi)$ is a finite structure satisfying M.1 and T.5 then M is a matroid.

□

In the case that some weights are negative and one seeks a maximum-weight independent set, the greedy algorithm can be applied up to the point where only negative elements remain, and all of those are rejected.

For any given matroid $M = (E, \Phi)$, there is a **dual matroid** $M^D = (E, \Phi^D)$ in which each base of M^D is the complement of a base of M, and vice versa.

Theorem 5:

If $M = (E, \Phi)$ is a matroid, then $M^D = (E, \Phi^D)$ is also a matroid.

□

From the above it follows that if A is a lexicographic maximum base of M, then $E - A$ is a lexicographic minimum base of M^D. It is clear that solving a maximization problem for the primal matroid also solves a minization problem for the dual matroid.

Conclusion:

1. From the duality notion of matroids it follows that whenever a matroid has a non empty dual matroid then both maximum and minimum greedy algorithms can be applied to it. The maximum weight independent set can be built either by selecting the elements in order of decreasing weight as suggested before, or by building the minimum independent base of the dual matroid and taking its complement. In practice it means that we find our set by deleting elements from E in order of increasing weight until we are left with a maximal independent set.

2. Given a matroid $M = (E, \Phi)$ and a cost function C defined over Φ then if C can be expressed as:

$$C(S) = F(f_1, f_2, \ldots, f_j)$$

where $f_i = f(e_i)$ and F is a symmetric function, which is also monotonic, then the matroid greedy algorithm with the weights being $\{f_i\}$ will find an independent set with maximum cost.

The first three problems in section 4.3 are selection problems and they can all be modeled as matroids. The "spanning tree" problem is a classic example since a spanning tree is a maximal independent set in a graphic matroid. The "continuous knap-sack" problem can be modeled by the matroid (E,Φ) as follows: each item i is divided into s_i new items with a unit size and a value of $\frac{v_i}{s_i}$. E is the set of all these new ,unit size, elements and a subset belongs to Φ if it does not contain more then B elements. This structure is a matroid and it is easy to see that the greedy algorithm for it will result in the same strategy as the one suggested in section 4.3. The problem of "job sequencing with deadlines" is also greedily optimized and can be modeled and explained by the matroid theory. For details see [Lawler 1976].

An exception; the resource consumption problem.

A simple selection problem which is greedily optimized but cannot be modeled (at least, not in a straight forward way) as a matroid is given next (problem #5). Given a set of n elements, associated with each is a weight w_i and a total weight W, find the largest subset of elements with a total weight less or equal to W.

A greedy optimal strategy for this problem is to select the elements in a non-decreasing order of weight. If we try the straight forward way of modeling the subsets of elements with weight less or equal W as candidate for independent sets, we can see immediately that they do not satisfy property M.2. We therefore consider this problem as a specific model for greedily optimized selection problem that will be referred to as the **resource consumption model.**

4.3.2 Greedily optimized tree-construction problems.

In this subsection we summarize the work by Parker [Parker 1980] on Huffman algorithms and the tree construction problems which are optimally solved by them. Huffman algorithm can be regarded as a greedy algorithm for tree construction problems. An example from this class is the problem "optimal merge patterns" which requires a set files to be merged in a pairwise manner so that the total merge cost will be minimized. The cost of the resulting merged tree is the weighted average path length. Huffman algorithm builds a tree with optimal cost by merging at each step the two files with the smallest weights. The weight of a merged file is the sum of weights of its components files. Another, less common variation of Huffman algorithm is to construct a tree with minimum height. In this case the Huffman algorithm suggests to combine elements with minimal weights and the weight of a merged node is the maximum weight of its sons.

The problems considered here were called by Parker **Binary Tree Construction problems**. Formally, in a problem from this class one is given a set of n+1 leaves having corresponding nonnegative weights $\{w_1, w_2, \ldots, w_i, \ldots, w_n\}$. Construction of a full binary tree on these leaves is effected by n merges of pairs of "available" nodes. Each node in the pair is marked unavailable after a merge and their father (the result of the merge) is marked available, having as his weight some function F of the weights of its sons.

The set of weights for every problem instance are taken from a weight space U, that is an interval from the non-negative reals, and the weight combination function $F : U \rightarrow U^2$ is any symmetric function. F is used to produce the weight of internal nodes generated by a merge operation. A tree cost function, $G : U^n \rightarrow R$, is defined

for all trees having n internal nodes, to be any symmetric mapping from U^n into the real numbers. For such a tree, T, the cost of T will be

$$G(W(T)) = G(W_1(T), W_2(T), \ldots, W_n(T)),$$

where $W_i(T)$ is the i^{th} smallest internal node weight in tree T.

Huffman algorithm for binary tree construction can be stated as follows: given a weight combination function F, merge at each step the two available nodes of smallest weight until only one node is available. For example, if $F(x,y) = x + y$ and G = sum then the cost function is the weighted path length of T. If $F(x,y) = \max\{x,y\} + c$ and G = max then the cost of any tree T is $\max_{1 \le j \le n}(w_j + c\ l_j)$ where l_j is the path length from the leaf w_j to the root of the tree T. This is the tree height measure.

The question is for which other problems will the Huffman algorithm produce an optimal tree and under exactly which cost. Parker identifies a class of weight combination functions which all produce optimal trees with the Huffman algorithm, under corresponding classes of tree cost functions. The following definitions are required:

Definition: A function F is quasilinear if

$$F(x,y) = \phi^{-1}(\lambda\phi(x) + \lambda\phi(y))$$

where λ is a nonzero constant, and $\phi : U \to R$ is invertible.

Definition: Given two weight sequences a $= \{a_1 \le a_2 \le, \ldots, a_n\}$ and b $= \{b_1 \le b_2 \le, \ldots, b_n\}$ we write a $\le$ b if

$$\sum_{i=1}^{k} a_i \le \sum_{i=1}^{k} b_i$$

for every k between 1 and n.

The three main results of this work will be described next.

Results:

1. The root weight in the tree generated by the Huffman algorithm is minimal over all the root weights of trees generated by other procedures, if and only if, the weight combination function F is quasilinear and for which $\lambda \geq 1$ and the ϕ function is unbounded.

   This is an important property since it guarantees a minimal root weight for the Huffman tree (the tree that is generated by the Huffman algorithm). Clearly, if the cost of the tree is determined by its root weight then the resulting Huffman tree is optimal.

2. The second result is an extension of the first one. Here the comparison among different trees is not only via the weight of the root of the trees but also by looking at all the internal nodes' weights of the trees. The result is given by the following theorem:

   Theorem (theorem 5 in [Parker 1980])

   Let $F(x,y) = \phi^{-1}(\lambda\phi(x) + \lambda\phi(y))$ be the weight combination function of the tree construction where ϕ is convex, positive, and strictly monotone and $\lambda \geq 1$. If $W(S)$ and $W(T)$ are the weight sequences of the internal nodes of the trees S and T, constructed respectively by the Huffman algorithm and by any other way, then

   $$W(S) \leq W(T)$$

   (The same results hold if ϕ is concave, negative,and strictly monotone).

3. The third result identifies the cost functions which capture the order

relationship between different weight vectors. That is, whenever the relationship between two trees S and T satisfies $W(S) \leq W(T)$, the cost function, G, satisfies $G(W(S)) \leq G(W(T))$. The set of cost function which have this property are the **Schur concave** functions.

Definition: A function $G : U \rightarrow R$ is Schur concave if

$$(x_i - x_j)(\frac{dG}{dx_i} - \frac{dG}{dx_j}) \leq 0$$

For all $x_i\ x_j \in U$, $i, j \in \{1,2,...,n\}$

Theorem:

$G(a) \leq G(b)$ for all weight sequences $a \leq b$ if and only if G is Schur concave.

We can therefore conclude that if F is quasilinear and satisfies the properties mentioned in result 2, and if G is Schur concave , then the Huffman algorithm produces a tree with minimal cost.

4.4 POPERTIES OF GREEDILY-OPTIMIZED ORDERING PROBLEMS

In this section we attempt to characterize cost functions that permit optimal solution of ordering problems by a greedy algorithm. For obvious reasons, we are particularly interested in properties whose verification does not depend on the knowledge of the optimal solution, but rather can be carried out by simple manipulation of the problem representation.

Instead of discussing a single ordering problem, we focus on an entire class, or family, of problems that are very similar in character. This notion is formalized by that of a **Problem Scheme** P defined as a triplet (E, PAR, C) where:

1. E is a set of elements.

2. PAR is a set of parameters, which are real valued functions defined over the elements of E. Parameters could be single-argument functions, in which case they are denoted by $\alpha(i)$, $\beta(i)$, etc., where i indexes the elements in E. They could also be multiple-argument functions, defined over all sequences of the same number of elements and denoted by $\gamma(i,j)$, $\delta(i,j,k)$, etc.

3. C is a real valued cost function. It associates a cost with any sequence of subsets of elements in E, and is dependent only on the the parameters of the elements and on their order. The cost function is written as $C = C(\sigma)$, where σ denotes a sequence of elements in E, and σ_i is the element in position i in σ.

An instance of a problem scheme P, denoted by P_I is specified by a subset of elements E_I of E along with the values of their parameters. Let n be the cardinality

of E_I. The problem associated with every instance P_I of P is to find a sequencing $(e_1, \cdots, e_n)$ of all the elements of E_I such that $C(e_1,...,e_n)$ is maximal (or minimal) over all permutations of the elements.

For example, the problem of sequencing jobs on a single processor so as to minimize a weighted average of the flow times (problem 14 in Section 4.3) can be formulated in terms of a scheme (E, PAR, C) where E is a set of jobs, PAR contain two parameters, p and u, that associate with each job a processing time and an importance weight, respectively, and C is a cost function defined on any sequence σ of n elements as follows:

$$C(\sigma) = \sum_{i=1}^{n} u_i \sum_{j=1}^{i} p_j \tag{1}$$

where u_i and p_i are the parameters of the i^{th} element in the sequence σ.

A **greedy rule** for P is a sequence of functions $f = \{f_j\}$

$$f_j : \Phi_j \rightarrow R \tag{2}$$

where Φ_j are all sequences of j elements. A greedy procedure for solving any instance P_I of P using f = $\{f_i\}$ is defined as follows (maximization is assumed):

Greedy(P, f):

1. choose $e \in E$ such that $f_1(e) = \max \{f_1(e') \mid e' \in E\}$
2. after choosing $e_1, e_2, \ldots, e_{i-1}$ choose $y \in E - \{e_1, \ldots, e_{i-1}\}$ such that $f_i(e_1, \ldots, e_{i-1}, y) = \max \{f_i(e_1, \ldots, e_{i-1}, x) \mid x \in E - \{e_1, \ldots, e_{i-1}\}\}$

Definition: P is **greedily optimized by** f if for every problem instance P_I having n elements, the sequence $(e_1, \ldots, e_n)$ generated by Greedy(P, f) has a maximum cost over all permutations of the elements.

The above definition of a greedy rule may seem more general then the reader expect. It provides a frame model even though most of the discussion is restricted to the special and familiar cases as described next.

In the balance of this section we restrict our attention to problems with single-argument parameters and (unless specified otherwise) to greedy rules that satisfy

$$f_i(e_1, \ldots, e_i) = f_i(e_i) = f(e_i) . \tag{3}$$

In that case the greedy rule is defined over the set of parameters associated with each element, i.e.,

$$f(e_i) = f(\alpha_i, \beta_i, \ldots) . \tag{4}$$

We will refer to such rules as **ranking functions.** Possible ranking functions for the sequencing problem discussed above are

$$f(u_i, p_i) = u_i \tag{5}$$

and

$$f(u_i, p_i) = p_i u_i . \tag{6}$$

A ranking function induces a weak order among the elements of E and the greedy procedure simply chooses elements in a nonincreasing order of f.

Definition: A ranking function is **optimizing** for some problem scheme P, if for every problem instance, P_I, it generates an optimal order. A problem scheme is said to be **greedily optimized by a ranking function** if it permits an optimizing ranking function. (In the sequel, we use the term **greedily optimized** to mean greedily optimized by a ranking function.)

Our original question is reduced, then, to finding verifiable properties of cost functions that make a problem scheme greedily optimized and to the discovery of optimizing ranking functions. We start by giving a necessary and sufficient condition for a problem scheme to be greedily optimized. The condition is stated for problems having a one-to-one cost function (i.e., no two sequences have the same cost). It should be slightly modified to hold for any cost function but this involves some details that we choose to avoid in this specific theorem.

Theorem 1:

A necessary and sufficient condition for a problem scheme P, having a one-to-one cost function, to be greedily optimized is that for any two elements a and b (characterized by their assigned parameters) and for all problem instances of P in which they both participate, either a precedes b in all optimal sequences or b precedes a in all optimal sequences.

Proof:

Since the cost function is one to one, an optimizing ranking function should also be one to one. The existence of a one-to-one ranking function implies only one internal order between any two elements dictated by the magnitude of f(a) and f(b). Therefore if the condition is not satisfied by a one-to-one cost function, such ordering function cannot exist. If, on the other hand, the condition is satisfied then the relative positions of any pair of elements in the optimal sequences defines an order relationship among the elements. There exists (accept in some pathological cases) a real function f, defined over all the elements which satisfy $f(a)<f(b)$ iff a precedes b in the order (see [Krantz 1971]). The function f is the ranking function that yields an optimal solution.

□

Consider, for example, a problem scheme defined by a set of four elements $\{1,2,3,4\}$ and some cost function. Suppose that the optimal sequence for the instance defined by the set $\{1,2,3\}$ is (3,2,1), and that the optimal sequence for the instance defined by the set $\{1,2,4\}$ is (4,1,2). Then, this schema does not have any optimizing ranking function because elements 1 and 2 do not have the same ordering in the two optimal sequences.

The deficiency with the condition stated in theorem 1 is that it is usually not verifiable from the problem's representation (unless the problem is represented with its optimal solutions, an unlikely situation). We will therefore have to be satisfied with stronger requirements that constitute sufficient but not necessary conditions for greedily optimized problems. One way to do this is to extend the properties required by the theorem on the optimal sequences to all possible sequences.

Definition: Let $P=(E,PAR,C)$ and let f be a ranking function (not necessarily one-to-one) defined over the set of parameters of each element in E. A ranking function f is called **uniform** relative to P, if for every problem instance and for every sequence σ of elements in that problem instance,

$$C(\sigma) \geq C(\sigma^i) \text{ if } f(\sigma_i) > f(\sigma_{i+1}) \tag{7}$$

and

$$C(\sigma) = C(\sigma^i) \text{ if } f(\sigma_i) = f(\sigma_{i+1})$$

for all i, where σ^i is the sequence resulting from the exchange of the i^{th} and $i+1^{th}$ elements in σ.

Theorem 2:

A uniform ranking function f for a problem scheme P is optimizing.

Proof:

We have to show that for every problem instance an ordering generated by the greedy procedure using f is maximal. Let P_I be any problem instance and σ^* be an optimal sequence. We show that if σ^* violates the ordering dictated by f then it could be transformed into such a sequence with no change in the cost. Let i be the first location in σ^* that violates the ordering induced by f, i.e., $f(\sigma_i^*) < f(\sigma_{i+1}^*)$. By exchanging the elements in positions i and $i+1$ the cost of the sequence cannot decrease due to the property of f being uniform ranking function, and it cannot increase due to the assumption that σ^* is an optimal sequence. If now the element σ_{i+1}^* has an f value which is larger then that of the element σ_{i-1}^* then an exchange between them will also not change the value of the cost function. The element σ_{i+1}^* can propagate in this fashion up in the sequence without changing the cost until it reaches its appropriate position according to f. At this point all the first i+1 elements obey the order dictated by f. The next violation of the ordering which appears therefore in a position higher then i will be then located, and the process will continue in the same way. Since the locations of new violations increases, the process is guaranteed to terminate. The result will be an optimal sequence obeying the order dictated by f.

□

A uniform ranking function indicates how to produce an improvement in the value of the cost function from any non-optimal sequence. Also, if a uniform ranking function exists, then it is unique up to a monotonic increasing transformation.

What properties of the cost function guarantee the existence of a uniformly optimizing ranking function? This is the question we address next. Let Σ_{ab} be the set of all the sequences, in all the instances of problem scheme P, for which element

a immediately precedes element b.

Definition: A cost function C is said to be **pairwise preferentially independent** (p.w.p.i.) if either

$$C(\sigma) \geq C(\sigma^a) \quad \forall\, \sigma \in \Sigma_{ab}$$

with strict inequality for at least one sequence, or

$$C(\sigma) \leq C(\sigma^a) \quad \forall\, \sigma \in \Sigma_{ab}\,, \tag{8}$$

with strict inequality for at least one sequence, or

$$C(\sigma) = C(\sigma^a) \quad \forall\, \sigma \in \Sigma_{ab}\,,$$

where σ^a is the sequence resulting by the exchange of any two elements a and b in σ. In the first two cases we say that C prefers a on b (resp. b on a), and denote it by $a >_{p.w.} b$ (resp. $b >_{p.w.} a$). In the third case we say that C is indifferent between a and b and use the notation $a -_{p.w.} b$.

definition: A pairwise preferentially independent cost function C is said to be acyclic if the relation $\geq_{p.w.}$ is transitive, i.e.,

$$\text{if } a \geq_{p.w.} b \text{ and } b \geq_{p.w.} c \text{ then } a \geq_{p.w.} c\,. \tag{9}$$

This last property (i.e., that the relation "C prefers a on b" satisfies transitivity) is required to assure a weak order [Krantz 1971] and does not follow automatically from p.w.p.i. The following example shows that a cost function can be pairwise preferentially independent but not acyclic. Consider a problem instance defined by a set of three elements $\{1,2,3\}$ with a cost function C that creates the following complete order among all different sequences:

$$(321)>(132)>(213)>(312)>(123)>(231)\,.$$

For this instance C is pairwise preferentially independent where 3 is preferred to 2

and 2 is preferred to 1. However, 1 is preferred to 3 thus violating transitivity.

Theorem 3:

A necessary and sufficient condition for a problem scheme $P = (E, PAR, C)$ to have a uniform ranking function is that C is p.w.p.i. and acyclic. Obviously, in that case P is greedily optimized.

Proof:

It is obvious that the existence of a uniform ranking function for a problem scheme P implies that the cost function is p.w.p.i. and acyclic. We will therefore show only the sufficiency part. Since C is p.w.p.i. and acyclic it induces a weak order on all the elements of E, that is, on all the sets of values assigned to parameters. Therefore, there exists [Krantz 1971], a real function f on the elements of E that reflects this ordering, i.e.,

$$a >_{p.w.} b \quad \text{iff} \quad f(a) > f(b) \tag{10}$$

and

$$a -_{p.w.} b \quad \text{iff} \quad f(a) = f(b). \tag{11}$$

Obviously, f is a uniformly optimizing ranking function, and by theorem 2 P is greedily optimized.

□

The following theorem suggests a possible process for identifying a p.w.p.i. cost function and for discovering its optimizing ranking function.

Theorem 4:

Let P be a problem scheme $P=(E, PAR, C)$ and σ any sequence of any subset of the elements in E. If the cost function C satisfies

$$C(\sigma) - C(\sigma^i) = K \cdot (d(\sigma_i) - d(\sigma_{i+1})) \qquad (12)$$

for all i, where K is a nonnegative function defined on σ and d is a function defined on the parameters associated with each element, then d is an optimizing ranking function.

Proof:

It is easily seen that d, satisfying (12), is a uniform ranking function and therefore, by Theorem 2, optimizing.

□

Theorem 4 suggests the following process for discovering a uniform ranking function: perform symbolic manipulation on the cost function and try to express the difference between the cost of an arbitrary sequence and that of the sequence which results from exchanging the i^{th} and $i+1^{th}$ elements. If the expression satisfies condition (12) then an optimizing ranking function is given by d in that expression.

For an example we look again at the single-processor scheduling problem whose cost function is given in (1). Let σ^i be a sequence resulting from the exchange of the i^{th} and $i+1^{th}$ elements. We get that

$$C(\sigma) - C(\sigma^i) = (u_{i+1}p_i - u_i p_{i+1}) = u_{i+1}u_i \left(\frac{p_i}{u_i} - \frac{p_{i+1}}{u_{i+1}}\right). \qquad (13)$$

In this case the ranking function suggested from the above representation is $f(p_i,u_i) = \frac{p_i}{u_i}$. Most of the ordering problems in the list of Section 4.3 have a ranking function satisfying condition (8) (e.g., problems 4,8,9,10,11,12,13,14,15) and many of their cost functions satisfy also condition (12). However, the symbolic manipulation required to demonstrate condition (12) may not be straight forward (e.g. problem 15).

The gap between the sufficient condition for a problem scheme to be greedily optimized, requiring that the cost function be p.w.p.i., and the necessary condition as given in theorem 1, is not as wide as it may seem. For instance, in problem instances of two elements the two conditions coincide, because there are only two possible sequences, one of which is maximal. This observation leads to the following important and uniform way for discovering optimizing ranking functions for any greedily optimized problem not necessarily with a p.w.p.i. cost function.

Theorem 5:

If P is any greedily optimized problem scheme then a ranking function f is optimizing if and only if it agrees with the ordering dictated by the cost function on pairs of elements, that is, for every two elements a and b, if $C(a,b) > C(b,a)$ then $f(a) > f(b)$.

Proof:

If a problem scheme has an optimizing ranking function then it also has to optimaly solve instances of two elements only. To do that the ranking function has to satisfy the above condition, which may induce a complete order on the elements of E. For greedily optimized problems this order should yield an optimal solution for all instances.

□

Notice that the arguments for proving theorem 5 are possible only because we modeled a problem scheme as encompassing all problem instances having different sizes. If we had modeled a problem scheme to be dependent on the size of the problem, n, then the optimizing ranking function could be dependent on n and thus invalidating the argument presented in the proof. All the greedily optimized problems we encountered (specifically, all those in Section 4.3) do comply with our

model of problem scheme, and therefore, we believe we didn't exclude any interesting problems by such a restriction on the model.

Theorem 5 also provides a general way of finding an optimal solution for any instance of a greedily optimized problem scheme including cases where the cost function is not provided in a symbolic form. One should simply sort the elements in the order dictated by applying the cost function to pairs of elements. When a problem is not known a' priori to be greedily optimized, this method could be used to either generate candidate ranking functions or to reject the hypothesis that the problem is greedily optimized (if the pair-wise preference turn out intransitive). When the objective is to find an explicit optimizing ranking function f then, by theorem 5, candidate functions must satisfy $C(a,b)>C(b,a)f(a)>f(b)$ for any two elements a and b of E. It should be noted that this test only gives the ranking function once we know that it is greedily optimized. The general criteria (12) also provides a guarantee that the problem is greedily optimized. Note however that the guarantee can be ascertained by considering only adjacent 4xchanges as opposed to the exchange of any pair of elements.

The above observations imply that cost functions defined on pairs of elements can be used as building blocks for generating cost functions that are greedily optimized. A cost function C_2, defined on pairs of elements with k parameters each, is said to be **transitive** if for every $x,y,z \in R^k$

$$C_2(x,y) \geq C_2(y,x) \text{ and } C_2(y,z) \geq C_2(z,y) \quad C_2(x,z) \geq C_2(z,x). \qquad (14)$$

Theorem 6:

If a cost function C_2 defined on pairs is transitive, then a problem scheme $P=(E,PAR,C)$, where C is given by:

$$C(e_1,e_2,\ldots,e_n)=\sum_{j=1}^{n}\sum_{k=1}^{j}C_2(e_k,e_j) \tag{15}$$

is greedily optimized.

Proof:

The cost function defined in (15) satisfies

$$C(\sigma)-C(\sigma^i)=C_2(e_i,e_{i+1})-C_2(e_{i+1},e_i)\,, \tag{16}$$

and since $C_2(x,y)$ is transitive, then clearly C is p.w.p.i. and acyclic and therefore it is greedily optimized.

□

Theorem 5 points to a way by which the theory of greedily optimized problems, discussed so far, can be extended to encompass greedy rules which are not restricted to be single-argument ranking functions. For instance, suppose we consider greedy rules of the form:

$$f_1(e)=f(e) \text{ and} \tag{17}$$
$$\forall\, i>1 \quad f_i(e_1,\ldots,e_i)=f_i(e_{i-1},e_i)=f(e_{i-1},e_i)\,.$$

Such rules will be called **level-2 greedy rules.** In this case the function for deciding the next element in the sequence may depend on the last element. For instance, if the problem scheme has chosen.one parameter, a positive weight, by which elements are denoted, then the following is a possible level-2 greedy rule:

$$f(x)=x$$
$$f(x,y)=\begin{cases} y & \text{if } x\geq 1 \\ -y & \text{if } x<1 \end{cases} \tag{18}$$

Given any set of elements this greedy rule generates sequences that are a concatenation of two subsequences; a nonincreasing sequence of all elements greater than or equal to 1 followed by a nondecreasing sequence of all elements smaller than 1. It

will optimize any cost function that ranks such sequences as having maximum cost. The following theorem is a natural extension of theorem 5 to level-2 greedy rules.

Theorem 7:

Let $P=(E,PAR,C)$ be a problem scheme. If P is greedily optimized by a level-2 greedy rule then this rule has to agree with the cost on pairs and triplets of elements, that is,

1. For every two elements a and b, $C(a,b) > C(b,a)$ implies $f(a) > f(b)$, and

2. for every three elements $a, b,$ and c, such that $f(a) > f(b)$ and $f(a) > f(c)$, if $C(a,b,c) > C(a,c,b)$ then $f(a,b) > f(a,c)$.

Proof:

The optimizing level-2 greedy rule must solve optimally all problems instances of 2 and 3 elements, implying the first and the second condition, respectively.

□

However, unlike the ranking function in a level-1 rule for problem scheme P, in this case the cost on all pairs and triplets does not determine completely the optimal ordering of the elements (i.e., the set of functions $f(e_1,e_2)$). The argument saying that problem instances of three elements should be solved optimally by a level-2 rule implies something only on the optimal sequence but not on every 3-element sequence. For instance, if the optimal sequence is (a,b,c) the relationship between $f(b,a)$ and $f(c,a)$ is not determined by theorem 7 since b or c do not stand in a first position of the optimal sequence. This situation do not occur in two-element problem instances where there are only two possible sequences one of which has to be optimal. For example the level-2 greedy rule given in (18) satisfies the

requirements of theorem 7 for any cost function which, for pairs and triplets, is given by

$$C(x,y)=x \quad \text{and} \quad C(x,y,z)=xy+z\ , \tag{19}$$

respectively. However, (18) is not necessarily an optimizing greedy rule for cost functions that prescribe more elaborate costs to problem instances of larger size. For instance if we define the scheme cost function to be:

$$C(x_1,\ldots,x_{n-1},x_n)=\prod_{i=1}^{n-1}x_i+x_n \tag{20}$$

then this function is not greedily optimized by the above level-2 rule. For example, if a 4-element problem instance {10,0.5, 0.75, o.25} is considered, rule (18) will generate the sequence (10,0.75,0.25, 0.5) while the optimal sequence is (0.75.0.5,0.25,10). A more promising rule will be:

$$f(x_1,\ldots,x_i)=\begin{cases} x_i & \prod_{j=1}^{i-1}x_j>0 \\ -x_i & \prod_{j=1}^{i-1}x_j<0 \end{cases} \tag{21}$$

This is a greedy rule which is not limited to a specific level. It may provide an optimal solution to a larger set of problem instances but not to all of them. We can conclude therefore that this problem is not greedily optimized even in the broader sense of greedy rule as defined in (2).

The observations made with respect to theorems 5 and 7 can be generalized, in a natural way, to greedy rules of any level. For example, it can be shown that any level-i greedy rule that optimizes a problem scheme P has to comply with the behavior of the cost function on all instances of the problem consisting of 2, 3,..., i elements. However, most greedily optimized problems encountered in the literature,

are solved by level-1 greedy rules, i.e., ranking functions. We, therefore, conclude our brief venture into higher-level greedy rules and return to focus on ranking functions.

4.4.2 Dominant greedy vs. Regular greedy

A popular choice of a greedy rule is the cost function itself, i.e.,

$$f_i(e_1,\ldots,e_i) = C(e_1,\ldots,e_i)\,. \tag{22}$$

Meaning that any point in time a person chooses that element which if it were the last, would yield best cost (i.e. Mayopic policy). This greedy rule is optimal for some problem scheme P if any instance P_I of P have the following property: any subsequence $(e_1,\ldots,e_j)$ of E_I that has a maximal cost over all subsets of size j of E_I can be extended to a sequence of length $j+1$ that has a maximal cost over all subsets of size $j+1$ of E_I. Formally,

$$\forall\ (e_1,\ldots,e_i) \text{ optimal over } \Phi_i\ \ \exists\ e_{i+1} \in E_I - \{e_1,\ldots,e_i\} \tag{23}$$
$$s.t.\ \ (e_1,\ldots,e_i,e_{i+1}) \text{ is optimal on } \Phi_{i+1}\,.$$

When this condition is satisfied, the greedy rule (19) generates an optimal sequence, σ satisfying the following property: any subsequence $(e_1,\ldots,e_j)$ of σ has a maximal cost over all subsets of size j of E_I. An optimal sequence that has this property is called a **dominant sequence**. A greedy rule that generates dominant sequences for every problem instance is said to be **dominant** (we will also say that in this case the cost function is dominant). It is clear then, that a problem scheme that satisfies condition (23) is dominant and its dominant sequences can be obtained using the greedy rule (22).

Not all greedy rules which are optimal are dominant. For example, the cost function

$$C(\sigma) = \sum_{i=1}^{n} u_i \sum_{j=1}^{i} p_j \tag{24}$$

which, as we already know, is greedily optimized by the ranking function $f(u,p) = \frac{p}{u}$, is not dominant. To see this, consider the three-element problem instance, in which each element i is defined by its parameters (u_i, p_i):

$$e_1 = (1,4),\ e_2 = (0.5,3)\ e_3 = (5,10) \tag{25}$$

The values assigned by the ranking function to the elements are: $f(e_1) = 4$, $f(e_2) = 6$, $f(e_3) = 2$, so that the optimal sequence is: (e_2,e_1,e_3). Evaluating sequences of two elements we see that

$$C(e_2,e_1) = 8.5 \tag{26}$$

while

$$C(e_3,e_1) = 105 \tag{27}$$

Obviously, the length-2 subsequence of the optimal sequence is not maximal, and thus the ranking function is not dominant. On the other hand, the following cost function

$$C(\sigma) = \sum_{i=1}^{n} \sum_{j=1}^{i} p_j \tag{28}$$

for which the ranking function $f(p) = p$ is optimizing, is dominant.

In what follows we identify necessary and sufficient conditions for a cost function to be dominantly optimized and discuss the relationships between these conditions and those obtained for regular greedily optimized problems.

The sufficient condition (22) may or may not be verifiable from the cost function if it is given symbolically. This property looks suitable for a proof by induction. Nevertheless, we are interested in identifying other, stronger sufficient conditions which may be easier to verify. Again, we focus on level-1 greedy rules, namely, on the question whether a cost function has a dominant ranking function.

Since a dominant ranking function is optimizing, it is determined by the order imposed by the cost function on pairs of elements and should satisfy the condition of theorem 5, and since the cost function is also defined on sequences of one element, we must have $f(e) = C(e)$ and:

$$\text{if } C(a,b) > C(b,a) \text{ then } C(a) > C(b). \tag{29}$$

To test this property, one should check the behavior of $C(e)$ as a ranking function: if inconsistency is found in the order induced by $C(e)$ and the order induced by the cost on pairs (given that both orders are well defined) then the hypothesis that the problem has a dominant optimizing ranking function may be rejected. (It still may have a non-dominant optimizing ranking function as in (25).)

In order to provide some necessary conditions for a dominant cost function we need the following definitions. The first definition is a weaker version of the p.w.p.i. property in which the order under exchange of adjacent elements should hold only when those elements are at the tail of the sequence.

Definition: A cost function is **tail pairwise preferentially independent** (t.p.w.p.i.) if either

$$C(\sigma) \geq C(\sigma^a) \quad \forall \quad \sigma \in \Sigma^t_{ab} \tag{30}$$

with strict inequality for at least one sequence, or

$$C(\sigma) \leq C(\sigma^a) \quad \forall \quad \sigma \in \Sigma^t_{ab}$$

with strict inequality for at least one sequence, or

$$C(\sigma) = C(\sigma^a) \quad \forall \quad \sigma \in \Sigma^t_{ab}$$

In the first two cases we say that C prefers a on b (resp. b on a) t.p.w., and denote it by $a >_{t.p.w.} b$ (resp. $b >_{t.p.w.} a$). In the third case we say that C is indifferent between a and b t.p.w and use the notation $a -_{t.p.w.} b$. If the relation is acyclic then it induces a weak order on the elements of E.

Definition: A scheme cost function is **order preserving** if

$$\forall\, i \text{ if } C(e_1, \ldots, e_i) \geq C(e_1', \ldots, e_i') \text{ and } C(e_{i+1}) \geq C(e_{i+1}') \qquad (31)$$
$$\text{then } C(e_1, \ldots, e_i, e_{i+1}) \geq C(e_1', \ldots, e_i', e_{i+1}').$$

Let Σ^t_{ab} denote all sequences for which element a immediately precedes b when b is the last element.

Theorem 9:

Let C be a scheme cost function in $P = (E, PAR, C)$ which is both order preserving and t.p.w.p.i. If $C(a) \geq C(b)$ implies that $C(a,b) \geq C(b,a)$, then C is greedily optimized by a dominant ranking function.

Proof:

We prove by induction that the ranking function $f(e) = C(e)$ generates dominant optimal solutions. The induction is on the length, k, of the subsequence under consideration. Let $\sigma = (e_1, e_2, \ldots, e_n)$ be the sequence selected by the greedy algorithm with the ranking function $f(e) = C(e)$.

1. k=1. The first element e_1 is chosen by the criterion that $C(e_1)$ is maximal over all the elements in E. The dominance property obviously holds.

2. $k = 2$. Given e_1 s.t. $C(e_1)$ is maximal, the algorithm chooses e_2 s.t. $C(e_2)$ is maximal over all the elements in $E - e_1$. Let (x,y) be any other possible pair in E.

   a. if $y \neq e_1$ then $C(e_1) \geq C(x)$ and $C(e_2) \geq C(y)$ and therefore from the order preserveness property of C we get $C(e_1,e_2) \geq C(x,y)$

   b. if $y = e_1$ then $C(e_1,e_2) \geq C(e_2,e_1)$ because of the t.p.w.p.i. order which unifies with the order imposed by $C(.)$. Also $C(e_2,e_1) \geq C(x,e_1) = C(x,y)$ because of order preserveness. From the above two inequalities we get $C(e_1,e_2) \geq C(x,y)$.

3. Assume that for all subsequences of length $t \leq k$, $(e_1, \ldots, e_t)$ have maximum cost over Φ_t. We will prove that this holds for $\sigma = (e_1, \ldots, e_k, e_{k+1})$. We have to show that $\forall\ \sigma'$, $\sigma' = (e_1', \ldots, e_{k+1}')$ $C(\sigma) \geq C(\sigma')$. From the induction hypothesis we know that $C(e_1, \ldots, e_k) \geq C(e'_1, \ldots, e'_k)$.

   a. Suppose that $e_{k+1}' \notin \{e_1, \ldots, e_k\}$. In this case $C(e_{k+1}) \geq C(e'_{k+1})$ and from order preserveness it follows that $C(e_1, \ldots, e_k, e_{k+1}) \geq C(e'_1, \ldots, e'_k, e'_{k+1})$.

   b. Assume $e'_{k+1} \in \{e_1, \ldots, e_k\}$. In that case $e'_{k+1} = e_{j_0}$ for $1 \leq j_0 \leq k$.

      We first show that:

$$C(\sigma) = C(e_1, \ldots, e_{k+1}) \geq C(e_1, \ldots, e_{j_0-1}, e_{j_0+1}, \ldots, e_{k+1}, e_{j_0}). \qquad (32)$$

      Then we show that

$$C(e_1, \ldots, e_{j_0-1}, e_{j_0+1}, \ldots, e_{k+1}, e_{j_0}) \geq C(e'_1, \ldots, e'_k, e_{j_0}) = C(\sigma'). \qquad (33)$$

From (32) and (33) it follows that

$$C(\sigma) \geq C(\sigma') . \tag{34}$$

Equation (32) can be shown by propagating the element e_{j_0} to the right side of the sequence σ while always keeping the cost nonincreasing:

$$C(e_1, \ldots, e_{j_\sigma}, e_{j_\sigma+1}) \geq C(e_1, \ldots, e_{j_\sigma+1}, e_{j_\sigma}) \tag{35}$$

since C is t.p.w.p.i. From (35) and the order preserveness of C it follows that:

$$C(e_1, \ldots, e_{j_\sigma}, e_{j_\sigma+1}, e_{j_\sigma+2}) \geq C(e_1, \ldots, e_{j_\sigma+1}, e_{j_\sigma}, e_{j_\sigma+2}) . \tag{36}$$

We can continue in the same manner to propagate e_{j_0} to the right without increasing the cost, thus proving equation (32).

Equation (33) is true since it follows, from the induction hypothesis on the set of elements $E - \{e_{j_0}\}$, that:

$$C(e_1, \ldots, e_{j_\sigma-1}, e_{j_\sigma+1}, e_{k+1}) \geq C(e'_1, \ldots, e'_k) , \tag{37}$$

and, together with the order preserveness property, this implies:

$$C(e_1, \ldots, e_{j_\sigma-1}, e_{j_\sigma+1}, e_{k+1}, e_{j_\sigma}) . \geq C(e'_1, \ldots, e'_k, e_{j_\sigma}) \tag{38}$$

□

It is easy to show that a cost function which is t.p.w.p.i. and order preserving is p.w.p.i. Therefore, a cost function satisfying the conditions of theorem 9 is p.w.p.i. and has a dominant optimizing ranking function. For example, the function

$$\sum_{i=1}^{n} p_1 \cdot p_2 \cdot \cdots \cdot p_i \tag{39}$$

is both p.w.p.i. and order preserving. The ranking function $f(p) = p$ results in a

dominant optimal solution. For C(e) to be an optimizing ranking function, it is important to verify that the order induced by $C(e)$ agrees with that induced by the p.w.p.i. property (i.e. the last condition of theorem 9). For instance, the cost function

$$\sum_{i=1}^{n} p_i{}^i \tag{40}$$

is both p.w.p.i. and order preserving. However, the ordering implied by $C(p)$ results in a decreasing sequence while the (optimal) ordering implied by exchanging adjacent elements is nondecreasing in p. Indeed, this cost function is greedily optimized with $f(p) = -p$, but is not dominant.

A special class of dominant cost functions defined on elements with a single parameter, is presented next. The cost is defined recursively and each element will be identified by the value of its parameter, also called a weight. Given a weight combination function $F : R \times R \rightarrow R$, C is defined as follows:

1. $C(x,y) = F(x,y)$.

2. $C(x,y,z) = F(F(x,y),z)$.

3. Given that C is defined for i elements, $C(e_1,..,e_i,e_{i+1}) = F(C(e_1,..,e_i),e_{i+1})$.

Theorem 10:

Let F be a weight combination function which is monotonic in both arguments, i.e.,

$$F(x,y) \geq F(x,z) \quad y \geq z, \text{ and}$$
$$F(y,x) \geq F(z,x) \quad y \geq z,$$

and path-length monotone, i.e.,

$$F(F(x,y),z) \geq F(F(x,z),y) \quad y \geq z \ .$$

If C is a cost function defined recursively by F, and assuming that on single element $C(e) = e$, then C is dominantly optimized using the ranking function $f(x) = x$. We assume that the path-length monotone property holds also when $x = \Phi$.

Proof:

We will show that C is order preserving and t.p.w.p.i., and that both orderings comply with the ordering dictated by the weights. The monotonicity of F w.r.t. both arguments implies that C is order preserving. From the path-length monotonicity of F it follows that if $e_i \geq e_{i+1}$ then

$$C(e_1, \ldots, e_i, e_{i+1}) = F(F(C(e_1, \ldots, e_{i-1}), e_i), e_{i+1}) \geq \tag{41}$$
$$F(F(C(e_1, \ldots, e_{i-1}), e_{i+1}), e_i) = C(e_1, \ldots, e_{i-1}, e_{i+1}, e_i) \ ,$$

which verifies the required property. Since C is both order preserving and t.p.w.p.i., it is dominantly greedily optimized by the ranking function $f(x) = x$.

□

As an example, the cost function generated recursively by the weight combination function $F(x,y) = x^2 y$ is dominantly optimized by the ranking function $f(x) = x$. The function F is monotone in both arguments and is path-length monotone. The cost function generated this way is given explicitly by:

$$C(e_1, e_2, \ldots, e_i) = e_1^{2i} \cdot e_2^{2(i-2)} \cdot \ldots \cdot e_i \ . \tag{42}$$

Up to now all the "nice" greedily optimized cost function properties we described required the cost function to be p.w.p.i. (or t.p.w.p.i.). Since this property is not a necessary condition it is natural to look for cost functions that are greedily optimized but not p.w.p.i. We show that the p.w.p.i. property may indeed be replaced by another strong property of cost functions.

Definition: A cost function is strong order preserving if $\forall i$ and $\forall x,y \in E$

$$C(e_1,\ldots,e_i) > C(e'_1,\ldots,e'_i) \quad C(e_1,\ldots,e_i,x) > C(e'_1,\ldots,e'_i,y) \quad (43)$$

For example, A lexicographical order among sequence of integers is strong order preserving. The cost function

$$C(e_1,\ldots,e_n) = \sum_{i=1}^{n} |e_i - e_{i+1}| \, 10^{n-i} \quad (44)$$

for $e_i \in [0,10)$,is strong order preserving but not p.w.p.i.

Theorem 11:

If a cost function is strong order preserving then it is dominantly optimized.

proof:

Clearly, the ranking function $f_i(e_1,\ldots,e_i) = C(e_1,\ldots,e_i)$ is dominant when C is strong order preserving.

□

The notions of a dominant cost function and dominant greedy rules are reminiscent of the conditions for greedy optimality of selection problems. This suggests the possibility of combining the selection and sequence problems, for example, finding a maximum cost ordered subset of elements that satisfy given constraints. In the next theorem we enlarge the class of cost functions defined on the independent sets of a matroid to include also order dependent cost functions that can be greedily solved.

Theorem 12:

Let (E,Φ) be a matroid and C be a scheme cost function defined on sequences of elements in E. If the scheme cost function is order-preserving and tail-pairwise preferentially independent, then lexicographically maximum independent set (w.r.t. the

costs of each element as weights) will give a set and an ordering with maximum cost.

Proof:

Let $e = (e_1, e_2, \ldots, e_i)$ be a lexicographically maximum independent set of the matroid, and let e' be any other independent set arranged in some order. Since the cost function is also p.w.p.i. the cost will increase if we reorder the elements in e' nonincreasingly. From matroid theory [Lawler 1976] we know that $\forall i \; e_i \geq e'_i$, and the order preserveness property of the cost function guarantees that $C(e) \geq C(e')$.

□

For example, consider a set of n jobs that have to be processed on two machines. On the first machine they have to meet their deadlines and they all have a unit processing time. Those jobs that meet their deadlines on the first machine will be processed in some sequence on the second machine. Each job is associated on the second machine with a deadline, d', a processing time p' and an importance weight u'. Consider the following task: find a set of jobs that can be finished by their deadline on the first machine and sequence them on the second machine so as to maximize a weighted average of the processing times. The cost function in this case is given in (1). The set of jobs that can all be finished by their deadlines on the first machine, constitutes the independent sets of a matroid [Lawler 1976]. However, the cost function is not dominant and obviously not order preserving and, indeed, it cannot be greedily optimized over the set of jobs that can be finished by their deadlines on the first machine (i.e., theorem 12 cannot be used). Now consider the same problem except that the objective is to maximize a simple average of the processing times. The cost function in this case is:

$$\sum_{i=1}^{n} \sum_{j=1}^{i} p_j$$

This is a dominant cost function, that is also order preserving and thus can be optimally solved using the matroid greedy algorithm as follows: Choose the elements in a decreasing order of their p' as long as they can be ordered to meet their deadlines on the first machine.

We conclude this section by presenting two examples of greedily optimized cost functions and verifying their properties. although all the ordering problems in section 4.3. can be shown to be p.w.p.i. we choose to focus on problems 12 and 15.

Problem 12 requires to order a set of jobs so that the maximum job lateness,

$$\max\{F_i - d_i\}, \tag{45}$$

is minimized. Jackson [Jackson 1955] had shown that the problem is p.w.p.i. and suggested the due-dates as the ranking function. Let $e_1 = (p_1, d_1)$ and $e_2 = (p_2, d_2)$ be a pair of jobs with their processing times and due-dates. Using the process suggested by theorem 7, of identifying ranking functions based on costs of pairs, it can be shown that

$$C(e_1,e_2) > C(e_2,e_1) \rightarrow d_1 > d_2, \tag{46}$$

that is,

$$\max\{p_1 - d_1, p_1 + p_2 - d_2\} > \max\{p_2 - d_2, p_2 + p_1 - d_1\} \rightarrow d_1 > d_2. \tag{47}$$

The cost of one element is

$$C(e_1) = p_1 - d_1. \tag{48}$$

This cost does not provide the same ordering as the due-dates and, therefore, the cost function is greedily optimized but not dominant.

Problem 15 deals with minimizing the maximum flow-time in a two-machine flow-shop. It was shown [Conway 1967] that the maximum flow-time is minimized if job j precedes job $j+1$ when

$$\min\{A_j, B_{j+1}\} < \min\{A_{j+1}, B_j\} . \tag{49}$$

Looking only on two-job problems, it is easily verified that the ordering dictated by (49) coincides with the order determined by costs on pairs. If (A_1, B_1) and (A_2, B_2) are two jobs then the cost function for the sequence (e_1, e_2) is:

$$C(e_1, e_2) = A_1 + \max\{A_2, B_1\} + B_2 . \tag{50}$$

It can be shown that

$$A_1 + \max\{A_2, B_1\} + B_2 \; < \; A_2 + \max\{A_1, B_2\} + B_1 \tag{51}$$

iff

$$\min\{A_1, B_2\} < \min\{A_2, B_1\} \tag{52}$$

This criterion is the one known in the literature. From the transitivity property of the order induced by (52) we know that there exist a ranking function f that induces an individual order. After some manipulation such a ranking function can indeed be formed. It can be shown that if:

$$\min\{A_1, B_2\} < \min\{A_2, B_1\} \tag{53}$$

then

$$\frac{sign(A_1 - B_1)}{\min(A_1 , B_1)} < \frac{sign(A_2 - B_2)}{\min(A_2 , B_2)} . \tag{54}$$

Therefore the function

$$f(A_i, B_i) = \frac{sign(A_i - B_i)}{\min(A_i , B_i)} \tag{55}$$

is a uniform ranking function for the problem. The cost for one element only is $A_1 + B_1$ and it does not coincide with the ranking function (54). We can therefore conclude that this cost function is not dominant.

4.5 A GLOSSARY OF GREEDILY OPTIMIZED PROBLEMS

1. **Minimum (Maximum) Spanning Tree.**

   Given a graph $G = (V,E)$ where V is the set of vertices and E is the set of edges, and a cost function defined on E, Find a spanning tree with minimum cost.

   Greedy strategy: select edges in nondecreasing order of their values as long as they do not create a cycle.

2. **Continuous Knapsack problem.**

   Given a set of items with associated size s_i and value v_i, find a maximum value subset of items or fraction of items with a total size less or equal to B.

   Greedy strategy: Select the elements in nonincreasing order of the ratios $\frac{v_i}{s_i}$

3. **Job sequencing with deadlines.**

   Given n jobs with deadline d_i and profits p_i, and each job takes one unit to process. Find a maximum profit subset of jobs that can be completed by their deadline.

   Greedy strategy: Select the jobs in decreasing order of profit as long as they can be completed by their deadline.

4. **Optimal storage on tape.**

   Given a set of n programs {1,2,...,n}, such that program i has a length l_i, Find the sequence in which they should be placed on a tape such that the average retrieval time is minimized. That is , find a permutation which minimizes

the measure:

$$\sum_{k=1}^{n}\sum_{i=1}^{k} l_i$$

Where l_i is the length of the i^{th} program in the selected permutation.

Greedy strategy: Select the programs in a nonincreasing order of their length.

5. **Storing programs on a limited amount of tape.**

This problem is the same as problem 4 only that the amount of tape available is limited to L, and the goal is to Select a maximum set of programs to be stored on the tape.

Greedy strategy: Select the program in a nondecreasing order of their length. (this is like a 0-1 knapsack with constants values.)

6. **Optimal merge patterns:**

Merge n files in pairwise manner so as the total merging cost will be minimized. Each file has weight w_i. If file j result in the merge of files i and k then the cost of this merge is w_j where

$$w_j = w_i + w_k.$$

Greedy strategy: Merge the two files which have the lowest cost, then add the resultant file to the list and repeat (Huffman procedure).

7. **Single source shortest paths.**

Given a weighted graph, determine the shortest path from a source to all the remaining vertices of the graph.

Greedy strategy: Dijkstra algorithm.

8. **Scheduling sequential search.** [Simon 1975]

An unknown number of chests of spanish treasure have been buried on a random basis at some of n sites, at a known depth of three feet. For each site there is a known unconditional probability p_i, $i=1,2,...,n$, that a chest was buried there, and the cost of excavating site i is q_i. Find a strategy t (a permutation of the integers from 1 to n) that minimize the average cost of the search. The search terminates upon success.

Greedy strategy: select the sites according to increasing order of $\frac{q_i}{p_i}$.

9. **Another search problem.** [Degroot 1970]

Suppose that an object is hidden in one of r locations and let p_i be the prior probability that the object is in location i. Here $\sum_{i=1}^{r} p_i = 1$. Even though the correct location is searched, there may be a positive probability that the object will be overlooked in this search. Let α_i be the probability that the object is overlooked in location i, even though it is there. The cost of each search at location i is c_i. A procedure must be determined that will minimize the expected total cost of the search procedure.

Greedy strategy: Choose the locations according to an increasing order of the ratios

$$\frac{c_i}{\pi_i(k)}$$

where $\pi_i(k)$ denote the probability under any procedure that the object will be found for the first time during the k^{th} search of location i. $\pi_i(k)$ are given by:

$$\pi_i(k) = p_i \alpha_i^{k-1}(1-\alpha_i).$$

10. **Job sequencing with different processing times.**

Given n jobs with processing times p_i for job i find a sequencing that will minimize the mean flow time. That is:

$$\bar{F} = \frac{\sum_{k=1}^{n} \sum_{i=1}^{k} p_i}{n}$$

Where p_i is the processing time of the i^{th} job in the selected strategy and $F_i = \sum_{j=1}^{i} p_j$ is the flow time of the i^{th} job.

Greedy strategy: The mean flow time is minimized by sequencing the jobs in order of nondecreasing processing time:

$$p_1 \leq p_2 \leq \cdots \leq p_n$$

Note that this problem is identical to problem #4.

11. The same as previous problem, only the measure of performance is:

$$\frac{\sum_{i=1}^{n} F_i^{\alpha}}{n}$$

For $\alpha > 0$.

Greedy strategy: sequence jobs in order of nondecreasing processing time:

$$p_1 \leq p_2 \leq \cdots \leq p_n$$

12. **Sequencing Jobs according to due-date.** [Conway 1967] Given n jobs, each associated with a deadline d_i and a processing tim p_i, Find an optimal sequencing that minimized the maximum job lateness. Maximum job lateness is defined by:

$$\max\{F_i - d_i\}$$

Where F_i is the flow time of job i.

Greedy strategy: (Jackson, 1955) The maximum job lateness is minimized sequencing the jobs in order of nondecreasing due-dates.

13. **Sequencing jobs according to slack-times $d_k - p_k$**

Given n jobs with deadlines $\{d_i\}$ and processing times $\{p_i\}$, find an optimal sequencing that maximize the minimum job lateness and the minimum job tardiness. The minimum job lateness is defined by:

$$\min\{F_i - d_i\}$$

Greedy strategy: The minimum job lateness is maximized by sequencing the jobs in a nonincreasing order of slack time: $d_i - p_i$

14. **Sequencing against weighted measure of performance.** [Conway 1967]

The value u_i is given to each job to describe its relative importance. Mean weighted flow time is given:

$$\overline{F_u} = \sum_{i=1}^{n} u_i F_i$$

Greedy strategy : (Smith 1956). The total weighted flow time is minimized by sequencing the jobs in a nondecreasing order of $\frac{p_i}{u_i}$. and is maximized by the antithetical procedure

Many of the previous results can also be generalized to include u_i coefficients: Mean weighted waiting time and mean weighted lateness are minimized sequencing by the ratio $\frac{p}{u}$

15. **Minimizing maximum Flow-time in a two-machine flow-shop.** [Conway 1967]

Let (A_i,B_i) be a pair associated with each job. A_i is the work to perform on the first machine of the shop and B_i is the work to be performed on the second machine. For each i A_i must be completed before b_i can begin.

Given the 2n values: $A_1,A_2,...,A_n$, $B_1,B_2,...,B_n$, Find the ordering of these jobs on each of the two machines so that neither the precedence nor the occupancy constraints are violated and so that the maximum of the F_i is made as small as possible. $(F_i = \sum_{k=1}^{i} p_k)$

Greedy strategy: (Johnson 1954) ; The max flow is minimized if for every j job j precedes job $j+1$ if

$$min(A_j,B_{j+1}) < min(A_{j+1},B_j).$$

Another possible representation for the greedy strategy: sequence according to

$$f_i = \frac{sign(A_i - B_i)}{min(A_i , B_i)}$$

When $A_i \neq B_i$, and otherwise, 0.

16. **Coin changing.**

Let $A_n = \{a_1,a_2,...,a_n\}$ be a finite set of distinct coin types (e.g a_1 = 50c, a_2 = 25c etc.). Each of the a_i's is an integer and that $a_1 > a_2 > ... > a_n$. Each type is available in unlimited quantity. Find a minimal set of coins that sum to C.

Greedy strategy: Use the coin types in an order a_1, a_2, a_n. When the coin type i is being considered as many coins as possible from this type will be

given.

This strategy is optimal when the input set $A_n = \{k^{n-1}, k^{n-2}, \ldots, k^0\}$ for some k>1. When $a_n = 1$ then there is always a solution that can be obtained by the greedy strategy which is not always optimal.

References

[Ausiello 1981] Ausiello, G. and M. Protasi, "Probabilistic analysis of the performance of greedy strategies over different classes of combinatorial problems.," in *Proceedings Fundamentals of computation theory*, Szeged, Hungary: 1981, pp. 24-33. In lecture Notes in Computer-Science #117.

[Bagchi 1983] Bagchi, A. and A. Mahanti, "Search algorithms under different kinds of heuristics - A comparative study.," *Journal of the ACM*, Vol. 30, No. 1, 1983, pp. 1-21.

[Barr 1981] Barr, A. and E.A. Feigenbaum, *Handbook of Artificial Intelligence*, Los Altos, California: William Kaufman Inc., 1981.

[Bentley 1980] Bentley, J. L. and J. B. Saxe, "An analysis of two heuristics for the Eucledean traveling salesman problem.," in *Proceedings Conference of communication, Control, and Computing*, Allerton: 1980.

[Carbonell 1983] Carbonell, J.G., "Learning by analogy: Formulation and generating plan from past experience," in *Machine Learning*, Michalski, Carbonell and Mitchell, Ed. Palo Alto, California: Tioga Press, 1983.

[Chow 1968] Chow, C.K. and C.N. Liu, "Approximating discrete probability distributions with dependence trees," *IEEE Transaction on Information Theory*, 1968, pp. 462-467.

[Conway 1967] Conway, R.W., W.L. Maxwell, and L.W. Miller, *Theory of scheduling*, Reading, Massachusetts: Addison-wesley Publishing company, 1967.

[Degroot 1970] Degroot, A.H., *Optimal statistical Decisions*: Mecgraw-Hill, 1970.

[Duntsan 1973] Duntsan, F.D.J. and D.J.A. Welsh, "A greedy algorithm for solving a certain class of linear programs," *Mathematical programming*, Vol. 5, 1973, pp. 338-353.

[Even 1979] Even, S., *Graph Algorithms*, Maryland, USA: Computer Science Press, 1979.

[Freuder 1982] Freuder, E.C., "A sufficient condition of backtrack-free search.," *Journal of the ACM*, Vol. 29, No. 1, 1982, pp. 24-32.

[Garey 1979] Garey, M.R. and D.S. Johnson, *Computer and Intractability, A guide to NP-Completeness*, San Francisco, California: W.H. Freeman and Company, 1979.

[Gaschnig 1979] Gaschnig, J., "A problem similarity approach to devising heuristics: first results," in *Proceedings 6th international joint conf. on Artificial Intelligence.*, Tokyo, Jappan: 1979, pp. 301-307.

[Gelperin 1977] Gelperin, D., "On the optimality of A*," *Artificial Intelligence*, Vol. 8, No. 1, 1977, pp. 69-76.

[Guida 1979] Guida, G. and M. Somalvico, "A method for computing heuristics in problem solving," *Information Sciences*, Vol. 19, 1979, pp. 251-259.

[Haralick 1980] Haralick, R. M. and G.L. Elliot, "Increasing tree search efficiency for cconstraint satisfaction problems," *AI Journal*, Vol. 14, 1980, pp. 263-313.

[Harris 1974] Harris, L.R., "The heuristic search under conditions of error," *Artificial Intelligence*, Vol. 5, No. 3, 1974, pp. 217-234.

[Hart 1968] Hart, P.E., N.J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEE Trans. System Science and Cybernatics*, Vol. SSC-4, No. 2, 1968, pp. 100-107.

[Ibaraki 1977] Ibaraki, T., "The power of dominance relatios in branch-and-bound algorithms," *Journal of the association for computing Machinery*, Vol. 24, No. 2, 1977, pp. 264-279.

[Jackson 1955] Jackson, J. R., "Scheduling a production line to minimize maximum tardiness," UCLA, Los Angeles, Cal, Tech. Rep. Management Sciences Research Project, 1955.

[Karp 1983] Karp, R.M. and J. Pearl, "Searching for an optimal path in a tree with random costs," *Artificial Intelligence*, Vol. 21, No. 1, 1983.

[Knuth 1975] Knuth, D. E., "Esimating the efficiency of backtrack programs," *Mathematics of computation*, Vol. 29, No. 129, 1975, pp. 121-136.

[Korte 1981] Korte, B. and L. Lovasz, "Mathematical structures underlying greedy algorithms.," in *Proceedings Fundamentals of computation theory*, Szeged, Hungary: 1981, pp. 205-210. Springer Verlang's lecture Notes in Computer-Science #117.

[Krantz 1971] Krantz, D., D. Luce, S. Suppes, and A. Tversky, *Foundations of Measurement, Vol 1.*, New-York and London: Academic Press, 1971.

[Laired 1983] Laired, J. and A. Newell, "A universal weak method.," Carnegy Mellon., Tech. Rep. CMU-CS-83-141, 1983.

[Lawler 1966] Lawler, E.L. and D.E. Wood, "Branch-and-bound methods: A survey.," *Operations Research*, Vol. 14, No. 4, 1966, pp. 699-719.

[Lawler 1976] Lawler, E.L., *Combinatorial optimization, Networks and Matroids*: Holt, Rinehart, and Winston, 1976.

[Mackworth 1977] Mackworth, A.K., "Consistency in networks of relations," *Artifficial intelligence*, Vol. 8, No. 1, 1977, pp. 99-118.

[Mackworth 1984] Mackworth, A.K. and E.C. Freuder, "The complexity of some polynomial network consistancy algorithms for constraint satisfaction problems," *Artificial Intelligence* , Vol. 25, No. 1, 1984.

[Magazine 1975] Magazine, M.J., G.L. Nemhouzer, and L.E.Trotter, "When the greedy algorithm solves a class of knapsack problems.," *Operation research*, Vol. 23, No. 2, 1975.

[Martelli 1977] Martelli, A., "On the Complexity of admissible search algorithms," *Artificial Intelligence*, Vol. 8, No. 1, 1977, pp. 1-13.

[Mero 1984] Mero, L., "A heuristic search algorithm with modifiable estimate," *Artificial Intelligence*, Vol. 23, No. 1, 1984, pp. 13-27.

[Mohr 1985] Mohr, Roger and Thomas C Henderson, "Arc and path consistency revisited," University of Utah, Salt Lake City, Utah, Tech. Rep. UUCS-85-101, 1985.

[Montanari 1974] Montanari, U., "Networks of constraints :fundamental properties and applications to picture processing," *Information Science*, Vol. 7, 1974, pp. 95-132.

[Newell 1969] Newell, A., "Heuristic programming: Ill-structured problems," in *Progress in Operations Research, III.*, Aronofsky J., Ed. New York: Wiley, 1969.

[Nilsson 1980a] Nilsson, N., *Principals of Artificial Intelligence*, Palo Alto, California: Tioga, 1980.

[Nilsson 1980b] Nilsson, N.J., *Problem-Solving Methods in Artificial Intelligence*, Palo Alto , California: New York , McGraw-Hill., 1980.

[Nudel 1983] Nudel, B., "Consistent-Labeling problems and their algorithms: Expected complexities and theory based heuristics.," *Artificial Intelligence*, Vol. 21, 1983, pp. 135-178.

[Parker 1980] Parker, D.S., "Conditions for optimality of the Huffman algorithm.," *SIAM Journal of Computing*, Vol. 9, No. 3, 1980.

[Pearl 1982] Pearl, J. and J.H. Kim, "Studies in semi-admissible heuristics," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 4, No. 4, 1982, pp. 392-399.

[Pearl 1983] Pearl, J., "On the discovery and generation of certain heuristics," *AI Magazine*, No. 22-23, 1983.

[Pearl 1984] Pearl, J., *HEURISTICS: Intelligent Search Strategies for Computer Problem Solving*, reading Mass: Addison-Wesley, 1984.

[Pohl 1969] Pohl, I., "First results on the effect of error in heuristic search," *Machine Intelligence*, Vol. 5, 1969, pp. 219-236.

[Pohl 1971] Pohl, I., "Bi-directional Search," in *Machine Intelligence 6*, B. Meltzer D. Michie, Ed. New York: American Elsevier: 1971, pp. 127-140.

[Pohl 1973] Pohl, I., "The avoidence of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving," in *Proceedings Ijcai*, Stanford Universitym , California: 1973.

[Purdom 1985] Purdom, P.W. and C.A. Brown, *The Analysis of Algorithms*: CBS College Publishing, Holt, Rinehart and Winston, 1985.

[Reingold 1977] Reingold, E.M., J. Nievergelt, and N. Deo, *Combinatorial Algorithms: Theory and practice*, Englewoods Cliffs, N.J.: Prentice-Hall, 1977.

[Sacerdoti 1974] Sacerdoti, E. D., "Planning in a hierarchy of abstraction spaces," *Artificial Intelligence*, Vol. 5, No. 2, 1974, pp. 115-135.

[Sahni 1978] Sahni, S. and E. Horowitz, *Fundamentals of Computer Algorithms*, Rockville, Maryland: Computer Science press, 1978.

[Shortliffe 1976] Shortliffe, E.H., *Computer-based Medical Consultations: MYCIN*, New York: American Elsevier, 1976.

[Simon 1975] Simon, H. A. and J. B. Kadane, "Optimal problem solving search: all or none solutions.," *Artificaial Intelligence*, Vol. 6, 1975, pp. 235-247.