HEURISTIC SEARCH CONTROL FOR PLAN SYNTHESIS ALGORITHMS AND DYNAMIC CONSTRAINT SATISFACTION PROBLEMS

by

XuanLong Nguyen

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

ARIZONA STATE UNIVERSITY

August 2001

HEURISTIC SEARCH CONTROL FOR PLAN SYNTHESIS ALGORITHMS AND DYNAMIC CONSTRAINT SATISFACTION PROBLEMS

by

XuanLong Nguyen

has been approved

May 2001

APPROVED:

_____, Chair

_____

_____

Supervisory Committee

ACCEPTED:

_____

Department Chair

_____

Dean, Graduate College

ABSTRACT

This thesis investigates heuristic search control for several planning algorithms. It is divided into three closely interconnected parts. The first part of the thesis presents a family of effective and admissible heuristics for state-space planning extracted from planning graph structures. These heuristics are highly informed by exploiting the problem structure existing in the planning graphs. The second part of the thesis develops effective heuristic search control for partial order planners (POP) – a class of highly flexible plan synthesis algorithms that have hither-to lacked effective search control. This approach employs distance-based heuristic estimators, which are inspired from our heuristics for state-space planning, the use of reachability analysis and handling of disjunctive ordering constraints. All these heuristics are empirically shown to improve the efficiency of planning algorithms dramatically. In the last part of the thesis, dynamic constraint satisfaction (DCSP) model is proposed for studying and understanding the heuristics that have been developed for partial order planning and other plan synthesis algorithms. In particular, the thesis investigates a number of heuristics for value and variable ordering for general DCSP search by exploiting the dual view of DCSP and state-space search. The connection and applicability of these heuristics to the ones developed for planning algorithms are also discussed.

To the memory of my mother

ACKNOWLEDGMENTS

I would like to thank my advisor, Prof. Subbarao Kambhampati for his continuing encouragement, support and guidance. The results in this thesis are strongly influenced by his significant contributions to and incredible insights on the field of AI planning. Rao's infectious enthusiasm, pedagogical virtuousity about the subjects and his delightful spontaneity have made the "job" of being his student an invaluable learning experience.

I would also like to thank Dr. Chitta Baral and Dr. Huan Liu for spending time to read this thesis, and for their interesting courses.

My work was benefited by countless feedback from and discussions with members of the AI lab, in particular Terry Zimmerman, Binhminh Do, Biplav Srivastava and Romeo Sanchez. I would like to particularly thank Romeo for a fruitful collaboration on the AltAlt project These people, along with Cenk, Hung, Nam, Nie, Tuan, Ullas, Sreelakshmi and Senthil have made up a strong "AI gang" who are very close to each other. I thank my roommates Tuan, Hung and Pornchai for many good memories during my time at ASU.

I would like to thank Dinh Le Chi for being my best friend, who has helped me through difficult moments. We have shared with each other every little achievement, such as having a paper accepted, landing a job, or being called slimmer after purposefully gaining two more pounds on a junky diet. I hope to have some time to give her a few long-promised lectures on calculus and statistics, although time may be better spent registering for other more advanced and challenging courses she may have to give.

Finally, I would like to dedicate this thesis to the memory of my mother, Tran Thi Dzung, who has been my perennial source of inspiration. And I wish to run home, like a little boy who has just found a solution to a tricky math puzzle, to show this work to my father, Nguyen Xuan Luc, and my little sister, Nguyen Khanh Chi: "Look, this is my solution!". The solution may remain flawed. There may be room for further improvement, or there may even be more interesting things to do in the world, but these are special people in my life who always inspire me to live a meaningful life, and believe that I will be able to get things right by the end of the day.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

The field of AI planning is primarily concerned with building control algorithms that enable an agent to synthesize a plan of actions that will achieve its goal. The plethora of different plan synthesis algorithms developed in the past decades is usually characterized by two "distinct" categories: Planning as search in the space of world states (also known as state-space planning [48]) and planning as search in the space of partially ordered plans (also known as partial-order plan-space planning [62]).

For all their representational and algorithmic differences, the efficiency of each of these types of planners is decided mainly by the effectiveness of the heuristic search control employed in the algorithms. Significant advances have been made in the past few years on the use of heuristic search control for state-space planning algorithms. To date, state-space planners are shown to exhibit a level of efficiency superior to that of other approaches. Not coincidentally, the search control heuristics for state-space planners are also better understood in comparison to partial-order planners through the work by many researchers in the field [48, 6, 5, 20, 21, 4, 32, 33, 27, 12]. Indeed, most search heuristics used in state-space planning algorithms despite being conceptually very simple, are theoretically quite well-founded, and have roots in state-space search theory [51] and constraint satisfaction problems [61].

Consider a goal-directed backward state-space planner such as HSP-R [5], which involves

searching from the goal state G until initial state I is reached. The main search decision is in choosing which state to traverse next. This is typically done by ranking states encountered in the search space using a ranking function $f(S) = g(S) + h(S)$, where g(S) is the distance (in terms of number of actions) from S to G, and $h(S)$ is the *estimate* of the distance h^* from I to state S. According to the theory of admissible state-space heuristics, when an estimate h is admissible (optimistic) (*i.e* $h \leq H$), this ranking ensures the solution plan to be optimal in number of actions. In practice, having a close without being admissible estimate of H is often good enough to synthesize a solution plan of good quality. Therefore, one of the main work of a state-space planning algorithm often lies in having an accurate estimator of the distance based function h^*.

Planning algorithms such as HSP-R are examples of **conjunctive search** state-space planning, where the corresponding set of possible solution plans is essentially split into the search space after each search commitment. Another significant development in state-space planning in the past decade is the **disjunctive search** approach, which essentially keeps the set of possible solution plans in some disjunctive form and then extracts a solution from the disjunctive representation. A typical disjunctive planning algorithm involves compiling a bounded-length plan specification into a planning graph as in Graphplan [4], or satisfiability model as in SATPLAN [32], or a constraint satisfaction problem (CSP) as in GP-CSP [12]. In case of Graphplan, a solution plan is then extracted from the planning graph through graph search. In cases of SATPLAN or GP-CSP, a solution plan is extracted from a solution of the corresponding SAT or CSP problem.[1] The disjunctive approach has also been used in partial-order (plan-space) planning, but with much less success [32].

This thesis is mainly concerned with heuristic search control for conjunctive state-space and partial-order planning.[2] **In the first part of the thesis we will introduce a derivation of a family of highly effective and computationally efficient heuristics that guide the (conjunctive) state-space planning algorithms.** We specifically consider the problem of estimating

[1] From now on, we will use CSP to refer both satisfiability and constraint satisfaction problems, because a SAT problem is essentially a boolean CSP.

[2] For historical reasons, unless noted otherwise in this thesis we will keep referring to conjunctive state-space planning and conjunctive partial-order planning simply as state-space planning and partial-order planning, respectively. Disjunctive state-space and disjunctive partial-order planning are both referred to simply as CSP-based planning.

the number of actions required to reach a state from the initial state. This type of distance-based heuristic estimators have been pursued by several researchers such as McDermott [48], Bonet and Geffner [6, 5], Hoffman [21], Refanidis *et al* [52] in their respective state-space planning algorithms. To make the computation tractable, most of these heuristic estimators make strong assumptions about the independence of subgoals. Because of these assumptions, state-space planners often thrash badly in problems where there are strong interactions between subgoals. Furthermore, these independence assumptions also make the heuristics inadmissible, precluding any guarantees about the optimality of solutions found. In fact, the authors of UNPOP and HSP/HSP-R planners [48, 5] acknowledge that taking the subgoal interactions into account in a tractable fashion to compute more robust and/or admissible heuristics remains a challenging problem [48, 6].

We show that the planning graph, which is used in Graphplan as a disjunctive representation of the set of possible solution plans, is a rich source for deriving effective and admissible heuristics for controlling state space search. Specifically, we describe how a large family of heuristics–with or without admissibility property–can be derived in a systematic manner from a *leveled off* planning graph. These heuristics are then used to control a regression search in the state space to generate plans. The effectiveness of these heuristics is directly related to the fact that they give a better account of the subgoal interactions. We show how the propagation of binary (and higher order) mutual exclusion relations makes the heuristics derived from the planning graph more sensitive to negative subgoal interactions. Positive interactions are accounted for by exploiting structure of the planning graph that explicitly shows when achieving a subgoal also, as a side effect, achieves another subgoal. The heuristic computation is also highly efficient, by relying on the advances made in the past few years on the efficient construction of compact planning graphs. We also provide an empirical study that shows that a state-space planning algorithm armed with our heuristics is very competitive with some of the fastest planning algorithms in the literature.

In the second part of the thesis we will present effective search control for improving the efficiency of partial order planning (POP) algorithms. To date there have

been no known search control heuristics effective enough to carry the POP algorithms to the rank of the fastest state-space planners. Ironically, most existing architectures for integrating planning with execution, information gathering, and scheduling are based on POP algorithms. Several known real-world planning systems such as RAX [22], IxTeT[16] all fall under the POP category. In fact, as argued in [55], POP is widely considered to be the more open planning frameworks. It offers a more promising approach for handling domains with durative actions, and temporal and resource constraints as compared to other planning approaches. All this makes the task of improving the performance of POP algorithms a significant and timely contribution to the AI planning research.

While state-space planners synthesize plans that are totally ordered, partial-order planners search in the space of partially ordered plans. A typical POP algorithm such as UCPOP [62] works in a backward search manner, starting from an initial partial plan made of only a set of subgoals in the goal state. Then the algorithm repeatedly finds an action step to achieve each subgoal, unless such subgoal is already satisfied in the initial state. The addition of a new action (without being totally ordered with other actions) into the current partial plan in turn introduces new subgoals to achieve. The algorithm terminates when there is no more subgoals to achieve. There are several search decisions involved in a POP algorithm. The first search decision is in choosing which flaw (either subgoal or unsafe link, which is a special structure used to ensure the consistency of the plan) to consider. Once a subgoal is considered, the second search decision is in choosing which action step to achieve the subgoal. The third type of search decision is how to resolve an unsafe link, which basically involves in reordering several action steps in the partial plan so that the plan remains consistent.

We will show that the insights and heuristic techniques responsible for the advances in plan synthesis made in the recent years in the context of conjunctive and disjunctive state-space planners are largely adaptable to POP algorithms. In dealing with the search decisions mentioned above in POP algorithms, we present novel methods for adapting distance based heuristics, reachability analysis and disjunctive constraint processing techniques to POP algorithms. Distance-based heuris-

tics are used as the basis for ranking partial plans and as flaw selection methods. The other two techniques are used for efficiently enforcing the consistency of the partial plans–by detecting implicit conflicts and resolving them.

Our methods help scale up POP algorithms dramatically–making them competitive with respect to state space planners, while preserving their flexibility. We present empirical studies showing that POP++, a partial order planner which is enhanced by our ideas, can perform competitively with other existing approaches in many planning domains. In particular, POP++ appears to scale up much better than Graphplan in parallel domains. More importantly, the solutions POP++ generates are generally shorter in length, and provide significantly more execution flexibility. We believe that the scalable versions of the POP algorithm developed in this work will make it feasible to handle domains with temporal and resource constraints, as advocated in [55].

The successful application of state-space and CSP search heuristics to the POP algorithm, while encouraging enough, still leaves us feeling uneasy and puzzled. An inevitable question comes to mind: What if we want to generalize the techniques we developed to dealing with more complex type of constraints, such as those involving time and resources? CSP and state-space provides different facets of a POP algorithm, but they are clearly not the most suitable model for generalizing such techniques. The silver bullet is a computational model that inherits characteristics from both CSP and state-space search. A study of heuristic search control in such model – in addition to being interesting in its own right – will undoubtedly provide invaluable insights into how heuristic search control should be devised in the like of POP algorithms.

The last part of this thesis is set to embark on this ambitious task. **We will use *dynamic constraint satisfaction problem* (DCSP) [36] as a general model based on which our study of search control for POP algorithms is pushed further.** That POP search can be seen as DCSP is not much of a discovery. This connection was first remarked by Kambhampati [25] in the context of refinement search. However, we are not aware of any work that actually exploits this connection to study heuristic search control for POP algorithms. In fact, as far as we know,

while work on both CSP and state space search are plentiful, there were no significant work on search heuristics for the DCSP model, which was first formalized only recently (1990) by Mittal and Falkenhainer [36].

Shortly, a DCSP is a generalization of a CSP instance. In addition to the logical constraints among variables as in CSP, which is to be called **compatibility constraints**, there also is another type of constraint called **activity constraints**. An activity constraint says that given certain true logical conditions, a new variable is activated. A compatibility constraint is valid (applicable) only if all the variables involved in the constraint are active. We will present in this thesis a simple DCSP formulation of a POP algorithm, and point out that with some extension, one can come up with a DCSP formulation that includes time and resource constraints as well. This also demonstrates that the DCSP model appears to be powerful enough to handle a wide variety of constraint-based synthesis tasks, as argued in [36].

While it is possible to simply translate a DCSP instance into a CSP and then solve it using a CSP solver in a manner similar to the disjunctive approach to planning, we would consider heuristic search control for DCSP algorithm that involves incremental activating variables and assigning values for these variables until a solution is found. This algorithm overcomes the memory problem faced by the CSP compilation approach. Furthermore, by discerning the different natures of activity and compatibility constraints, we may be able to come up with more suitable search heuristics for DCSP algorithms.

The heuristic search control in a general DCSP is divided into two main issues similar to that of a CSP: value ordering and variable ordering. The investigation of DCSP value and variable ordering heuristics is based on the close connection of DCSP to both CSP search and state-space search. Briefly, we will point out that the value ordering can be done by a ranking function that ranks DCSP "states" in the search space. This ranking function is closely related to the distance-based function used in state-space search. The main work, however, lies in how to come up with an efficient and accurate approximation of this distance-based function. We present a general heuristic

estimator for approximating the distance-based function based on the ideas of relaxing some of the constraints in the DCSP. The connection between this heuristic estimator and that of the distance-based heuristic estimator developed in the earlier parts of the thesis is also examined. We also examines several variable ordering heuristics for DCSP, which draws ideas from both CSP and state-space, and discuss the connection with POP's flaw selection heuristics.

1.1 Problem Statement and Thesis Contribution

We will now be more precise obout the tasks that we are about to accomplish in this thesis. The first part of the thesis focuses on heuristic search control for state-space planning algorithm.[3]

> *To present a family of effective and admissible heuristics extracted from planning graphs for state-space planning algorithms.*

- We show the weakness of the current state-space heuristic estimators in the literature that make the strong assumption about subgoal independence. We show that ineffectiveness of these heuristics in a number of planning domains is attributed to their failure to account for the complex subgoal interactions inherent in these domains.

- We show that the planning graph structures, with their very useful causal structures and mutual exclusion constraints, can be used to derive a family of highly effective state-space heuristics. We also provide a family of admissible heuristics whose accuracy can be arbitrarily guaranteed at the cost of heuristic computation.

- We provide an empricial study that shows that a state-space planning algorithm armed with our most effective heuristic has a robust performance across a variety of planning domains. This algorithm, which is called *AltAlt*, is also very competitive with some of the best planning algorithms in the literature, such as Graphplan [4] and HSP-R [5].

[3]This work was or to be published in *Proceedings of AAAI-2000* and *Artificial Intelligence Journal* [38, 39].

The second part of the thesis focuses on heuristic search control for partial-order planning, a class of planning algorithms synthesizing highly flexible plans and which have hither-to lacked effective search control.[4]

To present effective search control heuristics that can significantly improve the efficiency of partial-order planning algorithms.

We introduce three main techniques that contribute to the huge improvement of our POP algorithm's efficiency. Interestingly, all these ideas have deep connection to the advances made in the state-space planning algorithm. More specifically,

- We provide an approximation of the distance-based function that measures the number of actions needed to refine a partial plan into a solution plan. The distance-based function is used for ranking partial plans in the search space. The approximation technique is similar to our previous work on search heuristic for state-space planning. Briefly, the approximation is done on the basis of relaxing the negative interactions of actions, and the extraction of actions possibly relevant to the refinements from a planning graph.

- We delay refining threats (unsafe links) using disjunctive ordering constraints. This idea was first introduced by Kambhampati and Yang [26]. From a DCSP's variable ordering perspective, where subgoals are fertile variables, and threats are infertile, this technique reflects a fertile-first variable ordering scheme.

- We apply reachability analysis to discover additional constraints, which are helpful in detecting inconsistent partial plans very early.

- We provide a careful empirical study of the ideas above in a planner called RePOP, which is implemented on top of UCPOP. In our study, RePOP shows some impressive performance for a partial order planner. It is able to solve large planning problem (such as logistics with up to 70 actions plan) in a matter of seconds. In addition to dominating the base planner UCPOP,

[4]This work was published in *Proceedings of IJCAI-01* [40].

REPOP can outperform Graphplan algorithm [4] and is quite competitive with a state of the art state-space planner *AltAlt*[37] in a number of planning domains.

- We also discuss ideas for extending our heuristics to dealing with partially instantiated actions; devising admissible heuristics in terms of both actions and time.

Following the connection of our heuristic ideas to that of state-space search and CSP search theory, we turn our attention to a general DCSP:

To investigate search control heuristics for dynamic constraint satisfaction problems, and their application to partial order planning algorithms.

In addition to clarifying the "POP as DCSP" view, and the dual nature of DCSP as both state-space search and CSP, we introduce a number of value and variable ordering heuristics for DCSP, and discuss their connection to the heuristics developed earlier for our partial order planner REPOP. More specifically,

- Based on earlier work by Mittal and Falkenhainer [36], we introduce a simple DCSP model that also includes a general objective function. We show that a POP algorithm can be formulated as solving a DCSP.

- We show that DCSP can be seen as state-space search, where each DCSP is made up of DCSP variables and the compatibility constraints among these variables. The transformation one state to another is triggered by applying activity constraints to a new a value assignment of a fertile variable in the state.

- We introduce a general distance-based function that can be used to ranking DCSP states in the search space. This ranking provides a basis for value ordering in the DCSP. We also introduce a general technique for estimating the distance-based function through partial constraint relaxation technique, and show the connection of this heuristic estimator to the one used in REPOP for ranking partial plans.

- We introduce the notions of fertile and infertile variables based on which there are a number of different heuristics for variable ordering. In particular, we turn our attention to a promising heuristic that follows these principles: (1) Favoring fertile variables over infertile ones; (2) Using a CSP heuristic for infertile variables; (3) For fertile variables, favor the one whose size of the descendent tree is smallest. We discuss in detail the connection of this heuristic to the one used in REPOP planner.

1.2 Notes to the Reader

Chapter 2, 3 and 4, which make up the bulk of this thesis, can be seen as both complementary and independent. Chapter 2 is very self-contained and exclusively concerned with state-space heuristic estimators. Chapter 3 is exclusively concerned with improving the efficiency of partial order planning algorithm. While it draws several insights from heuristic search control developed for state-space planning in the previous chapter, this chapter is generaly self-contained and can be read independently. Chapter 3 can also be seen as a motivation for the more general task (on DCSP) tacked in Chapter 4. Investigating search heuristics for DCSP is an interesting research problem in its own right. Thus, Chapter 4 is also quite self-contained and can be read independently of the previous chapter, except for the Section 4.3 and subsections 4.8.2 and 4.8.1, which discuss the connection between DCSP model and its search heuristics to POP algorithm and POP's search heuristics. These (sub)sections are where many general ideas proposed for DCSP are validated by the work on POP. Finally, Chapter 5 discusses related work and Chapter 6 concludes this thesis.

Chapter 2

Search Heuristics for State-space Planning

This chapter presents a family of effective and admissible heuristics for state-space planning algorithms. We show that the informedness of the heuristic estimators used in these algorithms can be significantly improved by accounting for the subgoal interactions, which can be captured using planning graphs in a systematic manner. We also provide an empirical evaluation that demonstrates the effectiveness of our heuristics. The ideas presented in this chapter were also implemented in a state of the art planner called *AltAlt* [37, 39][1], which was shown to be very competitive with some of the fastest planning systems that competed at the AIPS-00 planning competition [1].

2.1 Introduction

The last few years have seen a number of attractive and scaleable approaches for solving deterministic planning problems. Prominent among these are "disjunctive" planners, exemplified the Graphplan algorithm of Blum & Furst [4], and heuristic state space planners, exemplified by McDermott's UNPOP [48] and Bonet & Geffner's HSP-R planners [6, 5]. The Graphplan algorithm can be seen

[1] *AltAlt*'s source code is available from *http://rakaposhi.eas.asu.edu/altweb/altalt.html*

as solving the planning problem using CSP techniques. A compact CSP encoding of the planning problem is generated using a polynomial-time datastructure called "planning graph" [27]. On the other hand, UNPOP, HSP, HSP-R are simple state space planners where the world state is considered explicitly. These planners rely on heuristic estimators to evaluate the goodness of children states. As such, it is not surprising that heuristic state search planners and Graphplan-based planners are generally seen as orthogonal approaches [64].

In UNPOP, HSP and HSP-R, the heuristic can be seen as estimating the number of actions required to reach a state (either from the goal state or the initial state). To make the computation tractable, these heuristic estimators make strong assumptions about the independence of subgoals. Because of these assumptions, state search planners often thrash badly in problems where there are strong interactions between subgoals. Furthermore, these independence assumptions also make the heuristics inadmissible, precluding any guarantees about the optimality of solutions found. In fact, the authors of UNPOP and HSP/HSP-R planners acknowledge that taking the subgoal interactions into account in a tractable fashion to compute more robust and/or admissible heuristics remains a challenging problem [48, 6].

In this chapter, we show that the planning graph datastructure computed in polynomial-time as part of the Graphplan algorithm, provides a general and powerful basis for the derivation of state space heuristics that take subgoal interactions into account. In particular, the so-called "mutex" constraints of the planning graph provide a robust way of estimating the cost of achieving a set of propositions from the initial state. The heuristics derived from the planning graph are then used to guide state space search on the problem, in a way similar to HSP-R [5]. Note that this means we no longer use Graphplan's exponential time CSP-style solution extraction phase.

We will describe several families of heuristics that can be derived from the planning graph structure and demonstrate their significant superiority over the existing heuristic estimators. We will provide results of empirical studies establishing that state space planners using our best heuristics easily out-perform both HSP-R and Graphplan planners. Our development focuses both on heuristics

that speedup search without guaranteeing admissibility (such as those currently used in HSP-R and UNPOP), and on heuristics that retain admissibility and thus guarantee optimality. In the former case, we will show that our best heuristic estimators are more robust and are able to tackle many problem domains that HSP-R does poorly (or fails), such as the grid, travel, and mystery domains used in the AIPS-98 competition [47].

While our empirical results are themselves compelling, we believe that the more important contribution of our work is the explication of the way in which planning graph can serve as a rich basis for derivation of families of heuristic estimators. It is known in the search literature that admissible and effective heuristics are hard to compute unless the interactions between subgoals are considered aggressively [41]. Our work shows that planning graph and its mutex constraints provide a powerful way to take these interactions into account. Since mutex propagation can be seen as a form of directed partial consistency enforcement on the CSP encoding corresponding to the planning graph, our work also firmly ties up the CSP and state search views of planning.

The rest of the chapter is organized as follows. Section 2.2 reviews the HSP-R planner and highlights the limitations of its "sum" heuristic. These limitations are also shared to a large extent by other heuristic state search planners such as UNPOP. Section 2.3 discusses how the Graphplan's planning graph can be used to measure the subgoal interactions. Section 2.4 is the heart of the chapter. It develops several families of heuristics, some aimed at search speed and some at solution optimality. Each of these heuristic families are empirically evaluated in comparison to HSP-R heuristic and their relative tradeoffs are explicated. Section 2.5 discusses several ways in which the heuristic computation cost can be vastly improved by exploiting the advances made in efficient construction of planning graph structures. Section 2.6 summarizes the chapter.

2.2 Limitation of the HSP-R's sum heuristic

HSP-R [5] is currently one of the fastest heuristic state search planners. It casts planning as search through the *regression space* of world states [50]. In regression state space search, the states can

be thought of as sets of *subgoals*. The heuristic value of a state S is the estimated cost (number of actions) needed to achieve S from the initial state. It is important to note that since the cost of a state S is computed from the initial state and we are searching backward from the goal state, the heuristic computation is done only once for each state. Then, HSP-R follows a variation of A* search algorithm, called *Greedy Best First*, which uses the cost function $f(S) = g(S) + w * h(S)$, where g(S) is the accumulated cost (number of actions when regressing from goal state) and $h(S)$ is the heuristic value of state S.

The heuristic is computed under the assumption that the propositions constituting a state are strictly independent. Thus the cost of a state is estimated as the sum of the cost for each individual proposition making up that state.

Heuristic 1 (Sum heuristic) $h(S) := \sum_{p \in S} h(p)$

The heuristic cost $h(p)$ of an individual proposition p is computed using a iterative procedure that is run to fix point as follows. Initially, each proposition p is assigned a cost 0 if it is in the initial state I, and ∞ otherwise. For each instantiated action a, let $Add(a)$, $Del(a)$ and $Prec(a)$ be its Add, Delete and Precondition lists. For each action a that adds some proposition p, $h(p)$ is updated as:

$$h(p) := \min\{h(p), 1 + h(Prec(a))\} \tag{2.1}$$

Where $h(Prec(a))$ is computed using the sum heuristic (heuristic 1). The updates continue until the h values of all the individual propositions stabilize. This computation can be done before the backward search actually begins, and typically proves to be quite cheap.

Because of the independence assumption, the sum heuristic turns out to be inadmissible (overestimating) when there are positive interactions between subgoals (i.e achieving some subgoal may also help achieving other subgoals), and less informed (significantly underestimating) when there are negative interactions between subgoals (i.e achieving a subgoal deletes other subgoals). Bonet and Geffner [5] provide two separate improvements aimed at handling these problems to a

certain extent. Their simple suggestion to make the heuristic admissible is to replace the summation with the "max" function.

Heuristic 2 (Max heuristic) $h(S) := \max_{p \in S} h(p)$

This heuristic, however, is often much less informed than the sum heuristic as it grossly underestimates the cost of achieving a given state.

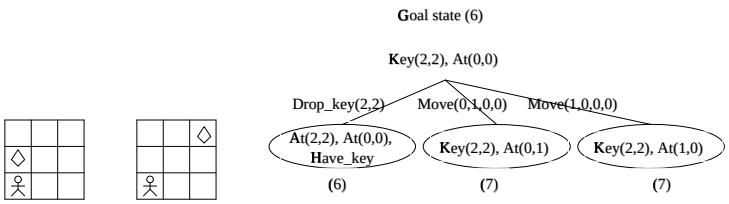
To improve the informedness of the sum heuristic, HSP-R adopts the notion of mutex relations first originated in Graphplan planning graph. But unlike Graphplan, only *static propositional mutexes* (also known as binary invariants) are computed. Two propositions p and q form a static mutex when they cannot both be present in any state reachable from the initial state. Since the cost of any set containing a mutex pair is infinite, we define a variation of the sum heuristic called the "sum mutex" heuristic as follows:

Heuristic 3 (Sum Mutex heuristic)

$h(S) := \infty \; if \; \exists_{p,q \in S} \; s.t. \; mutex(p,q) \; else \; \sum_{p \in S} h(p)$

In practice, the Sum Mutex heuristic turns out to be much more powerful than the sum heuristic and HSP-R implementation uses it as the default.

Before closing this section, we provide a brief summary of the procedure of computing mutexes used in HSP-R[5]. The basic idea is to start with a large set of "potential" mutex pairs and iteratively weed out those pairs that cannot be actually mutex. The set M_0 of potential mutexes is union of set M_A of all pairs of propositions $\langle p, q \rangle$, such that for some action a in A, p in $Add(a)$ and q in $Del(a)$, and set M_B of all pairs $\langle r, q \rangle$, such that for some $\langle p, q \rangle$ in M_A and some action a, r in $Prec(a)$ and p in $Add(a)$. This already precludes from consideration potential mutexes $\langle r, s \rangle$, where r and s are not in the add, precondition and delete lists of any single action. As we shall see below, this turns out to be an important limitation in several domains.



(a) Grid Problem (b) Regression Search

Figure 2.1. A simple grid problem and the first level of regression search on it.

2.2.1 A pathological example that showcases the limitations of sum mutex heuristic

The *sum mutex heuristic* used by HSP-R, while shown to be powerful in domains where the subgoals are relatively independent such as logistics and gripper domains [5], thrashes badly in problems where there is rich interaction between actions and subgoal sequencing. Specifically, when a subgoal that can be achieved early but that must be deleted much later when other subgoals are achieved, the sum heuristic is unable to recognize this interaction. To illustrate this, consider a simple problem from the grid domain [48] shown in Figure 2.1: Given a 3x3 grid. The initial state is denoted by two propositions at(0,0) and key(0,1) and the goal state is denoted by 2 subgoals at(0,0) and key(2,2) (See figure 2.1). Notice the subgoal interaction here: When key(2,2) is first achieved, at(0,0) is no longer true. There are three possible actions: the robot moves from one square to an adjacent square, the robot picks up a key if there is such a key in the square the robot currently resides, and the robot drops the key at the current square. One obvious solution is: The robot goes from (0,0) to (0,1), picks up the key at (0,1), moves to (2,2), drops the key there, and finally moves back to (0,0). This is in fact the optimal 10-action plan. We have run (our Lisp implementation of) HSP-R planner on this problem and no solution was found after 1 hour (generating more than 400,000 nodes, excluding those pruned by the mutex computation). The original HSP-R written in C also runs out of memory (250MB) on this problem.

It is easy to see how HSP-R goes wrong. First of all, according to the mutex computation

procedure described above, we are able to detect that when the robot is at a square, it cannot be in an adjacent square. But HSP-R's mutex computation cannot detect the type of mutex that says that the robot can also not be in any other square as well (because there is no single action that can place a robot from a square to another square not adjacent to where it currently resides).

Now let's see how this limitation of *sum mutex heuristic* winds up fatally misguiding the search. Given the subgoals (at(0,0), key(2,2)), the search engine has three potential actions over which it can regress the goal state (see Figure 2.1b). Two of these– move from (0,1) or (1,0) to (0,0)–give the subgoal at(0,0), and the third–dropping key at (2,2), which requires the precondition at(2,2)–gives the subgoal key(2,2). If either of the move actions is selected, then after the regression the robot would be at either (0,1) or (1,0), and that would increase the heuristic value because the cost of at(0,1) or at(1,0) is 1 (greater than the cost of at(0,0)). If we pick the dropping action, then after regression, we have a state that has both at(0,0) (the regressed first subgoal), and at(2,2) (the precondition of drop key at (2,2) action). While we can see that this is an inconsistent state, the mutex computation employed by HSP-R does not detect this (as explained above). Moreover, the heuristic value for this invalid state is actually smaller compared to the other two states corresponding to regression over the move actions. This completely misguides the planner into wrong paths, from which it never recovers.

HSP-R also fails or worsens the performance for similar reasons in the travel, mystery, grid, blocks world, and eight puzzle domains[47].

2.3 Exploiting the structure of Graphplan's planning graph

In the previous section, we showed the type of problems where ignoring the (negative) interaction between subgoals in the heuristic often lead the search into wrong directions. On the other hand, Graphplan's planning graph, with its wealth of mutex constraints, contains much of such information, and can be used to compute more effective heuristics.

Graphplan algorithm [4] works by converting the planning problem specifications into a

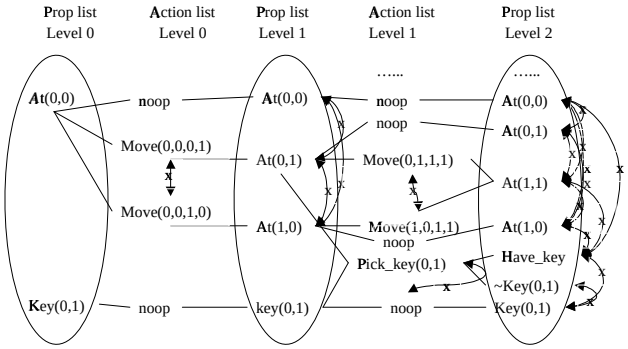


Figure 2.2. Planning Graph for the 3x3 grid problem

planning graph. Figure 2.2 shows part of the planning graph constructed for the 3x3 grid problem shown in Figure 2.1. As illustrated here, a planning graph is an ordered graph consisting of two alternating structures, called "proposition lists" and "action lists". We start with the initial state as the zeroth level proposition list. Given a k level planning graph, the extension of the structure to level $k+1$ involves introducing all actions whose preconditions are present in the k^{th} level proposition list. In addition to the actions given in the domain model, we consider a set of dummy "noop" actions, one for each condition in the k^{th} level proposition list (the condition becomes both the single precondition and effect of the noop). Once the actions are introduced, the proposition list at level $k+1$ is constructed as just the union of the effects of all the introduced actions. Planning-graph maintains the dependency links between the actions at level $k+1$ and their preconditions in level k proposition list and their effects in level $k+1$ proposition list.

The critical asset of the planning graph, for our purposes, is the efficient marking and propagation of mutex constraints during the expansion phase. The propagation starts at level 1, with the actions that are statically interfering with each other (i.e., their preconditions and effects are inconsistent) labeled mutex. Mutexes are then propagated from this level forward by using two simple propagation rules: Two propositions at level k are marked mutex if all actions at level k that support one proposition are pair-wise mutex with all actions that support the second proposition. Two actions at level $k+1$ are mutex if they are statically interfering or if one of

the propositions (preconditions) supporting the first action is mutually exclusive with one of the propositions supporting the second action. Figure 2.2 shows a part of the planning graph for the robot problem specified in Figure 2.1. The curved lines with x-marks denote the mutex relations. The planning graph can be seen as a CSP encoding [30, 64], with the mutex propagation corresponding to a form of directed partial 1- and 2-consistency enforcement [30]. The CSP encoding can be solved using any applicable CSP solving methods (a special case of which is the Graphplan's backward search procedure).

Normally, Graphplan attempts to extract a solution from the planning graph of length l, and will expand it to level $l + 1$ only if that solution extraction process fails. Graphplan algorithm can thus guarantee that the solution it finds is *optimal* in terms of number of steps. To make the optimality hold in terms of number of actions (a step can have multiple actions), we need to start with a planning graph that is **serial** [28]. A *serial planning graph* is a planning graph in which every pair of non-noop actions at the same level are marked mutex. These additional action mutexes propagate to give additional propositional mutexes. A planning graph is said to **level off** when there is no change in the action, proposition and mutex lists between two consecutive levels.

Based on the above mutex computation and propagation rules, the following properties can be easily verified:

1. The number of actions required to achieve a pair of propositions is no less than the number of proposition levels to be expanded until the two propositions both appear and are *not mutex* in the planning graph.

2. Proposition pairs that remain mutex at the level where the planning graph levels off can never be achieved starting from initial state.

3. The set of actions present in the level where the planning graph levels off contains all actions that are applicable to states reachable from the initial state.

The three observations above give a rough indication as to how the information in the planning graph after it levels off, can be used to guide state search planners. The first observation

shows that the level information in planning graph can be used to estimate the cost of achieving a set of propositions. Furthermore, the set of *level-specific* propositional mutexes help give a finer distance estimate. The second observation shows that once the planning graph levels off, all mutexes in the final level are *static* mutexes. The third observation shows a way to extract a finer (smaller) set of applicable actions to be considered by the regression search, since a new action is introduced into a level only if all of its preconditions appear in the previous level and are non-mutexed, and all actions present in a level are also present in the next level.

2.4 Extracting heuristics from planning graph

Before we go on to describing a set of effective heuristics extracted from the planning graph, let us briefly describe how these heuristics are used and evaluated. All the heuristics extracted from the planning graph as well as the HSP-R's sum heuristic are plugged into the *same* regression search engine using a variation of A* search's cost function $f(S) = g(S) + w * h(S)$.

We tested the heuristics on a variety of planning domains. These include several well-known benchmark domains such as the blocksworld, rocket, logistics, 8-puzzle, gripper, mystery, grid and travel. Some of these were used in the AIPS-98 competition [47]. These domains are believed to represent different types of planning difficulty. Problems in the rocket, logistics and gripper domains are typical of those where the subgoals are relatively independent. The grid, travel and mystery domains add to logistic domains the hardness of the "topological" combinatorics, while the blocksworld and 8-puzzle domains also have very rich interactions between actions and subgoal sequencing.

Subsection 2.4.1 is concerned with effective heuristics without consideration of the solution optimality. We set $w = 1$ in all experimental results described in this subsection, except for the parallel domains (e.g rocket, logistics and gripper) where the heuristics work best (in terms of speed) with $w = 5$ [2]. Subsection 2.4.2 is concerned with improving admissible heuristics for finding

[2] See [43] for the role of w in BFS.

optimal solutions. To make the comparisons meaningful, all the planners are implemented in Allegro commonlisp, and share most of the critical data structures. The empirical studies are conducted on a 500 MHz Pentium-III with 512 meg RAM, running Linux. All the times reported include both heuristic computation time and search time, unless specified otherwise.

2.4.1 Extracting effective heuristics

We are now ready to extract heuristics from the planning graph. Unless stated otherwise, we will assume that we have a serial planning graph that has been expanded until it has leveled off (without doing any solution extraction). In this section, we will concentrate on the effectiveness, without insisting on the admissibility of the heuristics.

Given a set S of propositions, denote $lev(S)$ as the index of the first level in the *leveled serial* planning graph in which all propositions in S appear and are non-mutexed with one another. If no such level exists, then $lev(S) = \infty$. Similarly, denote $lev(p)$ as the index of the first level that a proposition p comes into the planning graph. It takes only a small step from the observations made in the previous section to arrive at our first heuristic:

Heuristic 4 (Set-level heuristic) $h(S) := lev(S)$

Consider the set-level heuristic in the context of the robot example in previous section. In the planning graph, the subgoal key(2,2) first comes into the planning graph at the level 6, however at that level this subgoal is mutexed with another subgoal at(0,0), and the planning graph has to be expanded 4 more levels until both subgoals are present and non-mutex. Thus the cost estimate yielded by this heuristic is 10, which is exactly the true cost achieving both subgoals.

It is easy to see that set-level heuristic is *admissible*. Secondly, it can be significantly more informed than the *max heuristic*, because the max heuristic is only equivalent to the level that a single proposition first comes into the planning graph. Thirdly, a by-product of the set-level heuristic is that it already subsumes much of the static mutex information used by the Sum Mutex heuristic. Moreover, the propagated mutexes in the planning graph wind up being more effective in detecting

static mutexes that are missed by HSP-R. In the context of the robot example, HSP-R can only detect that a robot cannot be at squares adjacent to its current square, but using planning graph, we are able to detect that the robot cannot be at any square other than its current square.

Table 2.1 and 2.2 show that the set-level heuristic performs reasonably well in domains such as grid, mystery, travel and 8-puzzle[3] compared to the standard Graphplan [4]. Many of these problems prove intractable for HSP-R's sum-mutex heuristic. We attribute this performance of the set-level heuristic to the way the negative interactions between subgoals are accounted for by the level information.

Interestingly, the set-level heuristic fails in the domains that the *sum heuristic* typically does well, such as rocket world and logistics, where the subgoals are fairly independent of each other. Closely examining the heuristic values reveals that the set-level heuristic remains too conservative and often underestimates the real cost in these domains. A related problem is that the range of numbers that the cost of a set of propositions can take is limited to integers less than or equal to the length of the planning graph. This range limitation leads to a practical problem as these heuristics tend to attach the same numerical cost to many qualitatively distinct states, forcing the search to resort to arbitrary tie breaking.

To overcome these limitations, we pursue two families of heuristics derived by generalizing the set-level heuristic. The first family, called "partition-k" heuristics, attempt to improve the estimate of the cost of a set in terms of costs of its partitions. The second family, called "adjusted sum" heuristics attempt to improve the sum heuristic by considering the interactions among subgoals. These are described in the next two subsections.

2.4.1.1 Partition-k heuristics

To avoid underestimating and at the same time keep track of the interaction between subgoals, we want to partition the set S of propositions into subsets, each of which has k elements: $S =$

[3] 8puzzle-1, 8puzzle-2 and 8puzzle-3 are two hard and one easy eight puzzle problems of solution length 31, 30 and 20, respectively. Grid3 and grid4 are simplified from the grid problem at AIPS-98 competitions by reducing number of keys and grid's size.

[4] Graphplan implemented in Lisp by M. Peot and D. Smith.

Problem	sum-mutex	set-lev	adj-sum
bw-large-c	>500000	>500000	8224
rocket-ext-a	769	>500000	658
att-log-a	2978	>500000	2224
gripper	930	>500000	840
8puzzle-2	1399	51561	1540
8puzzle-3	2899	1047	1384
travel-1	4122	25	40
grid3	>200000	49	1151
grid4	>200000	44	1148
aips-grid1	>200000	108	835
mprime-1	>500000	125	96

Table 2.1. Number of nodes (states) generated by different heuristics, excluding those pruned by mutex computation

$S_1 \cup S_2 ... \cup S_m$ (if k does not divide $|S|$, one subset will have less than k elements). Ideally, we want a partitioning such that elements within each subset S_i may be interacting with each other, but the subsets are independent of each other. Thus we have the following heuristic:

Heuristic 5 (Partition-k heuristic) $h(S) := \sum_{S_i} lev(S_i)$, *where* $S_1, ..., S_m$ *are k-sized partitions of* S.

The question of deciding the partioning parameter k, and how to partition the set S when $1 < k < |S|$, however, is interesting. We find out that this knowledge may be largely domain-dependent. For example, for $k = 1$, the *partition-1* heuristic exhibits similar behavior compared to *sum-mutex* heuristic in domains where the subgoals are fairly independent (e.g gripper, logistics, rocket), and it is clearly better than sum-mutex in all other domains except the blocks world (see table 2.2).

For $k = |S|$, we have the *set-level* heuristic, which is very good in a complementary set of domains, compared with the sum-mutex heuristic.

For $k = 2$, we implemented a simple pairwise partition scheme as follows: The basic idea is, in order to avoid underestimating, we put propositions of greatest levels into different partitions. Given a set $S = \{p_1, p_2, ..., p_n\}$. Suppose $lev(p_1) \leq lev(p_2) \leq ... \leq lev(p_n)$. We partition

$$S = \{p_1, p_n\} \cup \{p_2, p_{n-1}\} \cup ... \cup \{p_{[(n-1)/2]}, p_{[(n+1)/2)]}\}.$$

Problem	Graphplan	Sum-mutex	set-lev	partition-1	partition-2	adj-sum	combo	adj-sum2
bw-large-b	18/ 379.25	18/ 132.50	18/ 10735.48	-	18/ 79.18	22/ 65.62	22/ 63.57	18/ 87.11
bw-large-c	-	-	-	-	-	30/ 724.63	30/ 444.79	28/ 738.00
bw-large-d	-	-	-	-	-	-	-	36/ 2350.71
rocket-ext-a	-	36/ 40.08	-	32/ 4.04	32/ 10.24	40/ 6.10	34/ 4.72	40/ 43.63
rocket-ext-b	-	34/ 39.61	-	32/ 4.93	32/ 10.73	36/ 14.13	32/ 7.38	36/ 554.78
att-log-a	-	69/ 42.16	-	65/ 10.13	-	63/ 16.97	65/ 11.96	56/36.71
att-log-b	-	67/ 56.08	-	69/ 20.05	-	67/ 32.73	67/ 19.04	61/53.28
gripper-20	-	59/ 90.68	-	59/ 39.17	-	59/ 20.54	59/ 20.92	59/38.18
8-puzzle1	31/ 2444.22	33/ 196.73	31/ 4658.87	35/ 80.05	47/ 172.87	39/ 78.36	39/ 119.54	31/ 143.559
8-puzzle2	30/ 1545.66	42/224.15	30/ 2411.21	38/ 96.50	38/ 105.40	42/ 103.70	48/ 50.45	30/ 348.27
8-puzzle3	20/ 50.56	20/ 202.54	20/ 68.32	20/ 45.50	20/ 54.10	24/ 77.39	20/ 63.23	20/ 62.56
travel-1	9/ 0.32	9/ 5.24	9/ 0.48	9/ 0.53	9/ 0.62	9/ 0.42	9/ 0.44	9/ 0.53
grid3	16/ 3.74	-	16/ 14.09	16/ 55.40	16/ 46.79	18/ 21.45	19/ 18.82	16/ 15.12
grid4	18/ 21.30	-	18/ 32.26	18/ 86.17	18/ 126.94	18/ 37.01	18/ 37.12	18/ 30.47
aips-grid1	14/ 311.97	-	14/ 659.81	14/ 870.02	14/ 1010.80	14/ 679.36	14/ 640.47	14/ 739.43
mprime-1	4/ 17.48	-	4/ 743.66	4/ 78.730	4/ 622.67	4/ 76.98	4/ 79.55	4/ 722.55

Table 2.2. Number of actions/ Total CPU Time in seconds. The dash (-) indicates that no solution was found in 3 hours or 250MB.

As Table 2.2 shows, the resulting heuristic exhibits interesting behavior: It can solve many problems that are either intractable by the *sum heuristic* or the *set-level heuristic*.

It would be interesting to have a fuller account of behavior of the family of *partition-k heuristics* with respect to different problem domains. Another related idea is to consider "adaptive partition" heuristics that do not insist on equal sized partitions. For example, p and q are put in the same partition if and only if they are mutexes in the planning graph. We intend to pursue these ideas in future work.

2.4.1.2 Adjusted Sum Heuristics

We now consider improving the sum heuristic by considering both negative and positive interactions among propositions. First of all, it is simple to embed the sum heuristic value into the planning graph. We maintain a cost value for each new proposition. Whenever a new action is introduced into the planning graph, we update the value for that proposition using the same updating rule 1 in Section 2.

We are now interested in estimating the cost $cost(S)$ for achieving a set $S = \{p_1, p_2, ..., p_n\}$. As before, suppose $lev(p_1) \leq lev(p_2) \leq ... \leq lev(p_n)$. Under the assumption that all propositions are independent, we have $cost(S) := cost(S - p_1) + cost(p_1)$. Since $lev(p_1) \leq lev(S - p_1)$, proposition p_1 is possibly achieved before the set $S - p_1$. Now, we assume that there are still no positive interactions, but there are negative interactions between the propositions. Therefore, upon achieving $S - p_1$, subgoal p_1 may have been deleted and needs to be achieved again. This information can be extracted from the planning graph. According to the planning graph, set $S - p_1$ and S are possibly achieved at level $lev(S - p_1)$ and level $lev(S)$, respectively. If $lev(S - p_1) \neq lev(S)$ that means there is some interaction between achieving $S - p_1$ and achieving p_1, because the planning graph has to expand up to $lev(S)$ to achieve both $S - p_1$ and p_1. To take this negative interaction into account, we assign:

$$cost(S) := cost(S - p_1) + cost(p_1) + (lev(S) - lev(S - p_1)) \tag{2.2}$$

Applying this formula to $S - p_1, S - p_1 - p_2$ and so on, we derive:

$$cost(S) := \sum_{p_i \in S} cost(p_i) + lev(S) - lev(p_n)$$

Since $lev(p_n) = \max_{p_i \in S} lev(p_i)$ as per our setup, we have the following heuristic:

Heuristic 6 (Adjusted-sum heuristic)

$h(S) := \sum_{p_i \in S} cost(p_i) + lev(S) - \max_{p_i \in S} lev(p_i)$

Table 2.1 and 2.2 show that this heuristic does very well across *all* different types of problems that we have considered. To understand the robustness of the heuristic, notice that the first term in its formula is exactly the *sum* heuristic value, while the second term is the *set-level heuristic*, and the third *approximately* the *max* heuristic. Therefore, we have

$$h_{adjsum}(S) \approx h_{sum}(S) + h_{lev}(S) - h_{max}(S)$$

It is simple to see that when there is strictly no negative interactions among propositions, $h_{lev}(S) = h_{max}(S)$. Thus, in the formula for $h_{adjsum}(S)$, $h_{sum}(S)$ is the estimated cost of achieving S under the *independence* assumption, while $h_{lev}(S) - h_{max}(S)$ accounts for the additional cost incurred by the *negative* interactions.

Note that the solutions solved by adjusted sum are longer than those provided by other heuristics in many problems. The reason for this is that the first term $h_{sum}(S) = \sum_{p_i \in S} cost(p_i)$ actually overestimates, because in many domains achieving some subgoal typically also helps achieve others. We are interested in improving the *adjusted-sum* heuristic by replacing the first term in its formula by another estimation $cost_p(S)$ that takes into account this type of *positive* interactions while ignoring the negative interactions (which are anyway accounted for by other two terms).

Since there are no negative interactions, once a subgoal is achieved, it will never be deleted again. Furthermore, the order of achievement of the subgoals $p_i \in S$ would be roughly in the order of $lev(p_i)$. Let p_S be the proposition in S such that $lev(p_S) = \max_{p_i \in S} lev(p_i)$. p_S will possibly be the last proposition that is achieved in S. Let a_S be an action in the planning graph that achieves

p_S *in the level* $lev(p_S)$, where p_S first appears. (If there are more than one, none of them would be noop actions, and we would select one randomly.)

By regressing S over action a_S, we have state $S + Prec(a_S) - Add(a_S)$. Thus, we have the recurrent relation (assuming unit cost for the selected action a_S)

$$cost_p(S) := 1 + cost_p(S + Prec(a_S) - Add(a_S)) \tag{2.3}$$

The positive interactions are accounted for by this regression in the sense that by subtracting $Add(a_S)$ from S, any proposition that is co-achieved when p_S is achieved is not counted in the cost computation. Since $lev(Prec(a_S))$ is strictly smaller than $lev(p_S)$, recursively applying equation 3 to its right hand side will eventually reduce to state S_0 where $lev(S_0) = 0$, whose cost $cost_p(S_0)$ is 0.

It is interesting to note that the repeated reductions involved in computing $cost_p(S)$ indirectly extract a sequence of actions (the a_S selected at each reduction), which would have achieved the set S from the initial state if there were no negative interactions. In this sense, $cost_p(S)$ is similar in spirit to (and is inspired by) the "relaxed plan" heuristic recently proposed by Hoffman[21].

Replacing $h_{sum}(S)$ with $cost_p(S)$ in the definition of h_{adjsum}, we get an improved version of adjusted sum heuristic that takes into account both positive and negative interactions among propositions.

Heuristic 7 (Adjusted-sum2 heuristic) $h(S) := cost_p(S) + (lev(S) - \max_{p_i \in S} lev(p_i))$, *where* $cost_p(S)$ *is computed using equation (3).*

Table 2.2 shows that adjusted-sum2 heuristic can solve all types of problem considered. The heuristic is only slightly worse compared with the adjusted-sum in term of speed, but gives a much better solution quality. In our experiments, with the exception of problems in the rocket domains, the adjusted-sum2 heuristic value is usually admissible and often gives optimal or near optimal solutions.

Finally, another way of viewing the adjusted-sum heuristic is that, it is composed of $h_{sum}(S)$, which is good in domains where subgoals are fairly independent, and $h_{lev}(S)$, which is good in a

Problem	Len	max		set-level		w/ memo		GP
		Est	Time	Est	Time	Est	Time	
8puzzle-1	31		-	14	4658	28	1801	2444
8puzzle-2	30	10	-	12	2411	28	891	1545
8puzzle-3	20	8	144	10	68	19	50	50
bw-large-a	12	6	34	8	21	12	16	14
bw-large-b	18	8	-	10	10735	16	1818	433
bw-large-c	28	12	-	14	-	20	-	-
grid3	16	16	13	16	13	16	5	4
grid4	18	10	33	18	30	18	22	22
rocket-ext-a	-	5	-	6	-	11	-	-

Table 2.3. Column titled "Len" shows the length of the found optimal plan (in number of actions). Column titled "Est" shows the heuristic value the distance from the initial state to the goal state. Column titled "Time" shows CPU time in seconds. "GP" shows the CPU time for **Serial Graphplan**

complement set of domains (see table 2.2). Thus the summation of them may yield a combination of *differential* power effective in wider range of problems, while discarding the third term $h_{max}(S)$ may sacrifice the solution quality.

Heuristic 8 (Combo heuristic) $h(S) := h_{sum}(S) + h_{lev}(S)$, *where* $h_{sum}(S)$ *is the sum heuristic value and* $h_{lev}(S)$ *is the set-level heuristic value.*

Surprisingly, as shown in table 2.2 the Combo heuristic is even slightly faster than adjusted-sum heuristic across all type of problems while the solution quality remains comparable.

2.4.2 Finding optimal plans with admissible heuristics

We now focus on admissible heuristics that can be used to produce optimal plans. Traditionally, efficient generation of optimal plans has received little attention in the planning community. In [28] Kambhampati *et. al.* point out that Graphplan algorithm is guaranteed to find optimal plans when the planning graph serial. In contrast, none of the known efficient state space planners [48, 6, 5, 52] can guarantee optimal solutions.

In fact, it is very hard to find an admissible heuristic that is effective enough to be useful across different planning domains. As mentioned earlier, in [6], Bonet et al. introduced the *max heuristic* that is admissible. In the previous section, we introduced the *set-level* heuristic that is

admissible and showed that it is significantly better than the max heuristic. We tested the set-level heuristic on a variety of domains using A* search's cost function $f(S) = g(S) + h(S)$. The results are shown in table 2.3, and clearly establish that set-level heuristic is significantly more effective than max heuristic. Grid, travel, mprime are domains where the set-level heuristic gives very close estimates (see table 2.2). Optimal search is less effective in domains such as the 8-puzzle and blocks world problem. Domains such as logistics, gripper remain intractable under reasonable limits in time and memory.

The main problem once again is that the set-level heuristic still hugely underestimates the cost of a set of propositions. The reason for this is that there are many *n-ary* ($n > 2$) *level-specific* mutex constraints present in the planning graph, that are never marked during planning graph construction, and thus cannot be used by set-level heuristic. This suggests that identifying and using higher-level mutexes can improve the effectiveness of the set-level heuristic.

Propagating all higher level mutexes is likely to be an infeasible idea [4, 28] (as it essentially amounts to full consistency enforcement of the underlying CSP). A seemingly zanier idea is to use a limited run of Graphplan's own backward search, armed with EBL [30], to detect higher level mutexes in the form of "memos". We have done this by restricting the backward search to a limited number of backtracks $lim = 1000$. This lim can be increased by a factor $\mu > 1$ as we expand the planning graph to next level.

Table 2.3 shows the performance of the set-level heuristic using a planning graph adorned with learned memos. We note that the heuristic value (of the goal state) as computed by this heuristic is significantly better than the set-level heuristic operating on the vanilla planning graph. For example in 8-puzzle2, the normal set-lev heuristic estimates the cost to achieve the goal as 12, while using memos pushes the cost to 28, which is quite close to the true optimal value of 30. This improved informedness results in a speedup in all problems we considered (up to 3x in the 8-puzzle2, 6x in bw-large-b), even after adding the time for memo computation using limited backward search.

We also compared the performance of the two set-level heuristics with the serial Graphplan,

Problem	Normal PG	Bi-level PG	Speedup
bw-large-b	22/ 63.57	28/ 20.05	3x
bw-large-c	30/ 444.79	38/ 114.88	4x
bw-large-d	-	44/11442.14	100x
rocket-ext-a	34/ 4.72	34/ 1.26	4x
rocket-ext-b	32/ 7.38	34/ 1.65	4x
att-log-a	65/11.96	64/ 2.27	5x
att-log-b	67/ 11.09	70/ 3.58	3x
gripper-20	59/ 20.92	59/ 7.26	3x
8puzzle-1	39/ 119.54	39/ 20.20	6x
8puzzle-2	48/ 50.45	48/ 7.42	7x
8puzzle-3	20/ 63.23	20/ 10.95	6x
travel-1	9/ 0.44	11/ 0.12	4x
grid-3	19/ 18.82	17/ 3.04	6x
grid-4	18/ 37.12	18/ 14.15	3x
aips-grid-1	14/ 640.47	14/ 163.01	4x
mprime-1	4/ 79.55	4/ 67.75	1x

Table 2.4. Total CPU time improvement from efficient heuristic computation for **Combo** heuristic

which also produces optimal plans. The set-level heuristic is better in the 8-puzzle problems, but not as good in the blocks world problems (See table 2.3). Further analysis is needed to explain these results.

2.5 Improving heuristic computation cost

There are a variety of techniques for improving the efficiency of planning graph construction in terms of both time and space, including bi-level representations that exploit the structural redundancy in the planning graph [45], as well as (ir)relevance detection techniques such as RIFO [49] that ignore irrelevant literals and actions while constructing the planning graph. These techniques can be used to improve the cost of our heuristic computation. In fact, in one of our recent experiments, we have used a bi-level planning graph as a basis for our heuristics. Preliminary results show significant speedups (up to 7x) in all problems, and we are also able to solve more problems than before because our planning graph takes less memory (See table 2.4).

2.6 Chapter Summary

In this chapter, we showed that the planning graph structure used by Graphplan provides a rich source of effective as well as admissible heuristics. We described a variety of heuristic families, that use the planning graph in different ways to estimate the cost of a set of propositions. Our empirical studies show that many of our heuristics have attractive tradeoffs in comparison with existing heuristics. In particular, we provided three heuristics– "adjusted-sum", "adjusted-sum2" and "combo" that are clearly superior to the sum mutex heuristic used by HSP-R across a large range of problems, including those that have hither-to been intractable for HSP-R. State search planners using these heuristics out-perform both HSP-R and Graphplan. We are also one of the first to focus on finding effective *and* admissible heuristics for state search planners. We have shown that the set-level heuristic working on the normal planning graph, or a planning graph adorned with a limited number of higher level mutexes is able to provide quite reasonable speedups while guaranteeing admissibility.

Our approach provides an interesting way of incorporating the strength of two different planning regimes (disjunctive vs. conjunctive search) [29] and views (planning as CSP vs. planning as state search) that have hither-to been considered orthogonal. We use the efficient directed consistency enforcement provided by the Graphplan's planning graph construction to develop heuristics capable of accounting for subgoal interactions. We then use the heuristics to guide a state search engine. In contrast to Graphplan, our approach is able to avoid the costly CSP-style searches in the non-solution bearing levels of the planning graph. In contrast to HSP-R and UNPOP, our approach is able to provide much more informed heuristics that take subgoal interactions into account in a systematic fashion.

Finally, the ideas developed in this chapter were implemented in C programming language by the author and Romeo Sanchez-Nigenda in a state of the art state-space planner called *AltAlt* [39, 37]. *AltAlt* is very efficient in a large number of planning domains, and is shown to be very competitive with the top 4 planning systems that competed at the AIPS-00 planning competition.

Chapter 3

Search Heuristics for Partial Order Planning

In this chapter, we will introduce several novel search control heuristics drawn from state-space and CSP search theory that help speedup a partial order planning algorithm dramatically. These techniques are the use of distance-based heuristic estimator to rank partial plans, the use of disjunctive ordering constraints for handling threat (unsafe links), and the use of reachability analysis for early detection of inconsistent partial plans. Our ideas are implemented in a variant of UCPOP called RePOP[1]. Our empirical results show that in addition to dominating UCPOP, RePOP also convincingly outperforms Graphplan in several "parallel" domains. The plans generated by RePOP also tend to be better than those generated by Graphplan and state search planners in terms of execution flexibility.

[1] UCPOP [62] → UNPOP [48] → RePOP. RePOP's source code is available from *http://rakaposhi.eas.asu.edu/repop.html*.

3.1 Motivation

Most recent strides in scaling up planning have centered around two dominant themes - heuristic state space planners, exemplified by UNPOP[48], HSP-R[5], and CSP-based planners, exemplified by Graphplan[4] and SATPLAN [32] . This is in stark contrast to planning research up to five years ago, when most of the efforts were focused on scaling up partial order planners[35, 62, 25, 56, 23, 17]. Despite such efforts, the partial order planners continue to be extremely slow and are not competitive with the fastest state search-based and CSP-based planners. Indeed, the recent advances in plan synthesis have generally been (mis)interpreted as establishing the supremacy of state space and CSP-based approaches over POP approaches.

Despite its current scale-up problems, partial order planning remains attractive over state space and CSP-based planning for several reasons. The least commitment inherent in partial order planning makes it one of the more open planning frameworks. This is evidenced by the fact that most existing architectures for integrating planning with execution, information gathering, and scheduling are based on partial order planners. In [55], Smith argues that POP-based frameworks offer a more promising approach for handling domains with durative actions, and temporal and resource constraints as compared to other planning approaches. In fact, most of the known implementations of planning systems capable of handling temporal and durative constraints –including IxTET [16] as well as NASA's RAX [22]–are based on the POP algorithms. Even for simpler planning domains, partial order planners search for and output partially ordered plans that offer a higher degree of execution flexibility. In contrast, none of the known state space planners can find parallel plans efficiently [20], and CSP planners such as Graphplan only generate a very restricted class of parallel plans (see Section 3.5).

The foregoing motivates the need for improving the efficiency of POP algorithms. We show in this chapter that the insights and techniques responsible for the advances in plan synthesis made in the recent years in the context of state-based and CSP-based planners are largely adaptable to POP algorithms. In particular, we present novel methods for adapting distance based heuristics,

reachability analysis and disjunctive constraint processing techniques to POP algorithms. Distance-based heuristics are used as the basis for ranking partial plans and as flaw selection methods. The other two techniques are used for efficiently enforcing the consistency of the partial plans–by detecting implicit conflicts and resolving them.

Our methods help scale up POP algorithms dramatically–making them competitive with respect to state space planners, while preserving their flexibility. We present empirical studies showing that REPOP, a version of UCPOP [62] enhanced by our ideas, can perform competitively with other existing approaches in many planning domains. In particular, REPOP appears to scale up much better than Graphplan in the parallel domains we tried. More importantly, the solutions REPOP generates are generally shorter in length, and provide significantly more execution flexibility [55].

This chapter is organized as follows. In the next section we will briefly review the basics of the POP algorithm. Section 3.3 describes how distance based heuristics can be adapted to rank partial plans. Section 3.4 shows how unsafe links flaws can be generalized and resolved efficiently. Section 3.5 reports empirical evaluations of the techniques that have been described.

3.2 Background on Partial Order Planning

In this chapter we consider the simple STRIPS representation of classical planning problems, in which the initial world state I, goal state G and the set of deterministic actions Ω are given. Each action $a \in \Omega$ has a precondition list and an effect list, denoted respectively as $Prec(a), Eff(a)$. The planning problem involves finding a plan that when executed from the initial state I will achieve the goal G.

A tutorial introduction to POP algorithms can be found in [62]. We will provide a brief review here. Most POP algorithms can be seen as searching in the space of partial plans. A **partial plan** is a five-tuple: $\mathcal{P} = (\mathcal{A}, \mathcal{O}, \mathcal{L}, \mathcal{OC}, \mathcal{UL})$, where $\mathcal{A} \subseteq \Omega$ is a set of (ground) actions,[2] $\mathcal{O}$ is a set

[2] Although partial order planners are capable of handling partially instantiated action instances, we restrict our attention to ground action instances.

of ordering constraints over $\mathcal{A}$, and $\mathcal{L}$ is a set of causal links over $\mathcal{A}$.[3] A causal link is of the form $a_i \xrightarrow{p} a_j$, and denotes a commitment by the planner that the precondition p of action a_j will be supported by an effect of action a_i. $\mathcal{OC}$ is a set of open conditions, and $\mathcal{UL}$ is a set of unsafe links. An **open condition** is of the form (p, a), where $p \in Prec(a)$ and $a \in \mathcal{A}$, and there is *no* causal link $b \xrightarrow{p} a \in \mathcal{L}$. Loosely speaking, the open conditions are preconditions of actions in the partial plan which have not yet been achieved in the current partial plan. A causal link $a_i \xrightarrow{p} a_j$ is called **unsafe** if there exists an action $a_k \in \mathcal{A}$ such that (i) $\neg p \in Eff(a_k)$ *and* (ii) $\mathcal{O} \cup \{a_i \prec a_k \prec a_j\}$ is consistent. In such a case, a_k is also said to **threaten** the causal link $a_i \xrightarrow{p} a_j$. Open conditions and unsafe links are also called **flaws** in the partial plan. Therefore a solution plan can be seen as a partial plan with no flaws (i.e., $\mathcal{OC} = \emptyset$ and $\mathcal{UL} = \emptyset$).

The POP algorithm starts with a null partial plan $\mathcal{P}$ and keeps refining it until a solution plan is found. The **null partial plan** contains two dummy actions $a_0 \prec a_\infty$ where the preconditions of a_∞ correspond to the top level goals of the problem, and the effects of a_0 correspond to the conditions in the initial state. The null plan has no causal links or unsafe link flaws, but has open condition flaws corresponding to the preconditions of a_∞ (top level goals).

A **refinement** step involves selecting a flaw in the partial plan $\mathcal{P}$, and *resolving* it, resulting in a new partial plan. When the flaw chosen is an open condition (p, a), an action b needs to be selected that achieves p. b can be a new action in Ω, or an action that is already in $\mathcal{A}$. The sets $\mathcal{OC}$, $\mathcal{O}$, $\mathcal{L}$ and $\mathcal{UL}$ also need to be updated with respect to b. Secondly, when the flaw chosen is an unsafe link $a_i \xrightarrow{p} a_j$ that is threatened by action a_k, it can be repaired by either **promotion**, i.e adding ordering constraint $a_k \prec a_i$ into $\mathcal{O}$, or **demotion**, i.e adding $a_j \prec a_k$ into $\mathcal{O}$.

The efficiency of POP algorithms depends critically on the way partial plans are selected from the search queue, and the strategies used to select and resolve the flaws. In Section 3.3 we present several distance-based heuristics for ranking partial plans in the search queue. Section 3.4 introduces the disjunctive constraint representation for efficiently handling unsafe link flaws, and reachability analysis for generalizing the notion of unsafe links to include implicit conflicts in the

[3] Strictly speaking $\mathcal{A}$ should be seen as a set of "steps", where each step is mapped to an action instance [25].

plan.

3.3 Heuristics for ranking partial plans

In choosing a plan from the search queue for further refinement, we are naturally interested in plans that are likely to lead to a solution with a minimum number of refinements (flaw resolutions). As we handle the unsafe links in a significantly different way than standard UCPOP (see Section 3.4), the only remaining category of flaws to be resolved are open condition flaws. Consequently one way of ranking plans in the search queue is to estimate the minimum number of new actions needed to resolve all the open condition flaws.

Definition 3.1 (h*) *Given a partial plan $\mathcal{P}$, let $h^*(\mathcal{P})$ denote the minimum number of new actions that need to be added to $\mathcal{P}$ to make it a solution plan.*

$h^*(\mathcal{P})$ can be seen as the number of actions that, when executed from the initial state I in some order, will achieve the set of subgoals $S = \{p|(p, a) \in \mathcal{OC}\}$. In this sense, this is similar to estimating the number of actions needed to achieve a state from the given initial state in state search planners [5, 38], but for two significant differences: (i) the propositions in S are not necessarily in the same world state and (ii) the set of actions that achieve S cannot conflict with the set of actions and causal links already present in $\mathcal{P}$.

A well-known heuristic for estimating h^* involves simply counting the number of open conditions in the partial plan [17].

Heuristic 3.1 (Open conditions heuristic) $h_{oc}(\mathcal{P}) = |\mathcal{OC}|$

This estimate is neither admissible nor informed in many domains, because it treats every open condition equally. In particular, it is ineffective when some open conditions require more actions to achieve than others.

We would like to have a closer estimate of h^* function without insisting on admissibility. To do this, we need to take better account of subgoal interactions[38]. Accounting for the negative

interactions in estimating h^* can be very tricky, and is complicated by the fact that the subgoals in S may not be in the same state. Thus we will start by ignoring the negative interactions. This has three immediate consequences: (i) the set of unsafe links $\mathcal{UL}$ becomes empty. (ii) the actions needed in achieving a set of subgoals S will have no conflicts with the set of actions $\mathcal{A}$ and and the causal links $\mathcal{L}$ already present in $\mathcal{P}$. and (iii) a subgoal p once achieved from the initial state can never become untrue. Given these consequences, it does not matter much that the subgoals in S are not necessarily present in the same world state, since the minimum number of actions needed for achieving such a set of subgoals in any given temporal ordering is the same as the minimum cost of achieving a state comprising all those subgoals.

The foregoing justifies the adaptation of many heuristic estimators for ranking the goodness of states in state search planners. Most of the early heuristic estimators used in state search not only ignore negative interactions, but also make the stronger assumption of subgoal independence[5, 48]. A few of the recent ones, [38, 21] however account for the positive interactions among subgoals (while still ignoring the negative interactions). It is this latter class of heuristics that we focus on for use in partial order planning. Specifically, to account for the positive interactions, we exploit the ideas for estimating the cost of achieving a set of subgoals S using a serial planning graph.[4]

Specifically, we build a planning graph starting from the initial state I. Let $lev(p)$ be the index of the level in the planning graph that a proposition p first appears, and $lev(S)$ be the index of the first level at which all propositions in S appear. Let p_S be the proposition in S such that $lev(p_S) = \max_{p_i \in S} lev(p_i)$. p_S will *possibly* be the last proposition in S that is achieved during execution. Let a_S be an action in the planning graph that achieves p_S in the level $lev(p_S)$. We can achieve p_S by adding a_S to the plan. Introduction of a_S changes the set of goals to be achieved to $S' = S + Prec(a_S) - Eff(a_S)$. We can express the cost of S in terms of the cost of a_S and S':

$$cost(S) := cost(a_S) + cost(S + Prec(a_S) - Eff(a_S)) \tag{3.1}$$

[4] We assume that the readers are familiar with the planning graph data structure, which is used in Graphplan algorithm[4].

where $cost(a_S) = 1$ if $a_S \notin \mathcal{A}$ and 0 otherwise. Since $lev(Prec(a_S))$ is strictly smaller than $lev(p_S)$, recursively applying Equation 3.1 to its right hand side will eventually express $cost(S)$ in terms of $cost(I)$ (which is zero), and the costs of actions a_S. The process is quite efficient as the number of applications is bounded by $lev(S)$.

Heuristic 3.2 (Relax heuristic) *$h_{relax}(\mathcal{P}) = cost(S)$, where $S = \{p|(p,a) \in \mathcal{OC}\}$, and $cost(S)$ is computed using the recurrence relation 3.1.*

Given such a heuristic estimate, plans in the search queue are ranked with the evaluation function: $f(\mathcal{P}) = |\mathcal{A}| + w * h(\mathcal{P})$. The parameter w is used to increase the greediness of the heuristic search and is set to 5 by default.

3.4 Enforcing consistency of partial plans

The consistency of a partial plan is ensured through the handling of its unsafe links. In this section we describe two ways of improving this phase. The first involves posting disjunctive constraints to resolve unsafe links. The second involves detecting implicit conflicts (unsafe links) using reachability analysis.

3.4.1 Disjunctive representation of ordering constraints

Normally, an unsafe link $a_i \xrightarrow{p} a_j$ that is in conflict with action a_k is resolved by either promotion or demotion, that is, splitting the current partial plan into two partial plans, one with the constraint $a_k \prec a_i$, and the other with the constraint $a_j \prec a_k$. A problem with this premature splitting is that a single failing plan gets unnecessarily multiplied into many descendant plans poisoning the search queue significantly. A much better idea, first proposed in [26], is to resolve the unsafe link by posting a disjunctive ordering constraint that captures both the promotion and demotion possibilities, and incrementally simplify these constraints by propagation techniques. This way, we can detect many failing plans before they get selected for refinement.

Specifically, an unsafe causal link $a_i \xrightarrow{p} a_j$ that is in conflict with action a_k can be resolved by simply adding a disjunctive ordering constraint $(a_k \prec a_i) \vee (a_j \prec a_k)$ to the plan.

We use the following procedure for simplifying the disjunctive orderings. Whenever an open condition (p, a) is selected and resolved by either adding a new action or reusing an action b in the partial plan, we add a new ordering constraint $b \prec a$ to $\mathcal{O}$, followed by repeated application of the constraint propagation rules below:

- $(a_1 \prec a_2) \in \mathcal{O} \wedge (a_2 \prec a_3) \in \mathcal{O} \Rightarrow \mathcal{O} \leftarrow \mathcal{O} \cup (a_1 \prec a_3)$
- $(a_1 \prec a_2) \in \mathcal{O} \wedge (a_2 \prec a_1) \in \mathcal{O} \Rightarrow$ False
- $(a_1 \prec a_2) \in \mathcal{O} \wedge (a_2 \prec a_1 \vee a_3 \prec a_4) \in \mathcal{O} \Rightarrow$

  $\mathcal{O} \leftarrow \mathcal{O} \cup (a_3 \prec a_4)$

  $\mathcal{O} \leftarrow \mathcal{O} - (a_2 \prec a_1 \vee a_3 \prec a_4)$

The first two propagation rules are already done as part of POP algorithm to ensure the transitive consistency of ordering constraints. The third rule is a unit propagation rule over ordering constraints. This propagation both reduces the disjunction and detects infeasible plans ahead of time. When all the open conditions have already been established and there are still disjunctive constraints left in the plan, the remaining disjunctive constraints are then split into the search space [26].

3.4.2 Detecting and Resolving implicit conflicts through reachability analysis

Although the unsafe link detection and resolution steps in the POP algorithm are meant to enforce consistency of the partial plan, often times they are too weak to detect implicit inconsistencies. In particular, the procedure assumes that a link $a_i \xrightarrow{p} a_j$ is threatened by an action a only if a has an effect $\neg p$. Often a might have an effect q (or precondition r) such that no legal state can have p and q (or p and r) true together. Detecting and resolving such implicit interactions can be quite helpful in weeding out inconsistent partial plans from the search space.

In order to do implicit conflict detection as described above, we need to have (partial) information about the properties of reachable states. Interestingly, such reachability information has played a significant role in the scale-up of state space planners, motivating the development of procedures for identifying mutex constraints, state invariants and memos etc. [4, 18, 15] (we shall henceforth use the term mutex to denote all these types of reachability information). One simple way of producing reachability information is to expand Graphplan's planning graph structure, armed with mutex propagation procedure [4]. The mutexes present at the level where the graph levels off are state invariants [38].

Exploiting the reachability information to check consistency of partial plans requires identifying the feasibility of the world states that any eventual execution of the partial plan must pass through. Although partial order plans normally do not have explicit state information associated with them, it is nevertheless possible to provide partial characterization of the states their execution must pass through. Specifically, we define the general notion of cutsets as follows:

Definition 3.2 (Cutsets) *Pre- and post- cutsets, C^- and C^+ of an action a_k in a plan $\mathcal{P}$ are defined as $C^-(a_k) = Prec(a_k) \cup L(a_k)$, and $C^+(a_k) = Eff(a_k) \cup L(a_k)$, where $L(a_k)$ is the set of all conditions p such that there exists a link $a_i \xrightarrow{p} a_j$ where a_i is necessarily before a_k, and a_j is necessarily after a_k*

The pre- and post-cutsets of an action can be seen as partial description of world states that *must* hold before and after the action a_k. If these partial descriptions violate the properties of the reachable states, then clearly the partial plan cannot be refined into an executable solution.

Proposition 3.1 *If there exists a cutset that contains a mutex, then the partial plan is provably invalid and can be pruned from the search queue.*

While this proposition allows us to detect and prune inconsistent plans, it is often inefficient to wait until the plan becomes inconsistent. Detecting and resolving implicit conflicts is essentially a more active approach that *prevents* a partial plan from becoming inconsistent by this proposition. Specifically, we generalize the notion of unsafe links as follows:

Definition 3.3 *An action a_k is said to have a* **conflict** *with a causal link $a_i \xrightarrow{p} a_j$ if (i) $\mathcal{O} \cup \{a_i \prec a_k \prec a_j\}$ is consistent and (ii) either $Prec(a_k) \cup \{p\}$ or $Eff(a_k) \cup \{p\}$ contains a mutex. A causal link $a_i \xrightarrow{p} a_j$ is* **unsafe** *if it has a conflict with some action in the partial plan.*

These notions of conflict and unsafe link subsume the original notions of *threat* and *unsafe link* introduced in Section 3.2, because $\neg p \in Eff(a_k)$ also implies that $Eff(a_k) \cup \{p\}$ is a mutex. Therefore the generalized notion of unsafe links result in detecting a larger number of (implicit) conflicts (unsafe links) present in a partial plan.

Once the implicit conflicts are detected, they are resolved by posting disjunctive orderings as described in the previous subsection. As we shall see later, the combination of disjunctive constraints and detection of implicit conflicts through reachability information leads to quite robust improvements in planning performance.

3.5 Empirical Evaluation (RePOP Planning Algorithm)

We have implemented the techniques introduced in this chapter on top of UCPOP[62], a popular partial order planning algorithm. We call the resulting planner RePOP. As mentioned in Section 3.2, both UCPOP and RePOP are given ground action instances, and thus neither of them have to deal with variable binding constraints. Both UCPOP and RePOP use the LIFO as the order in which open condition flaws are selected for resolution. Our empirical studies compare RePOP to UCPOP as well as Graphplan[4] and AltAlt[38], which represent two currently popular approaches (CSP search and state space search) in plan synthesis. All these planners are written in Lisp. In the case of Graphplan, we used the Lisp implementation of the original algorithm, enhanced with EBL and DDB capabilities [27]. AltAlt [39] is a state-of-the-art heuristic regression state search planner, that has been shown to be significantly faster than HSP-R [5]. The empirical studies are conducted on a 500 MHz Pentium-III with 256MB RAM, running Linux. The test suite of problems were taken from several benchmark planning domains from the literature. Some of these, including gripper,

Problem	UCPOP	REPOP			Graphplan			AltAlt	
	(time)	Time	#A/ #S	#flex	Time	#A/ #S	#flex	Time	#A
gripper-8	–	1.01	21/ 15	.57	66.82	23/ 15	.69	.43	21
gripper-10	–	2.72	27/ 19	.59	47min	29/ 19	.71	1.15	27
gripper-12	–	6.46	33/ 23	.61	–	–	–	1.78	33
gripper-20	–	81.86	59/ 39	.68	–	–	–	15.42	59
rocket-ext-a	–	8.36	35/ 16	2.46	75.12	40/ 7	7.15	1.02	36
rocket-ext-b	–	8.17	34/ 15	7.29	77.48	30/ 7	4.80	1.29	34
logistics.a	–	3.16	52/ 13	20.54	306.12	80/ 11	6.58	1.59	64
logistics.b	–	2.31	42/ 13	20.0	262.64	79/ 13	5.34	1.18	53
logistics.c	–	22.54	50/ 15	16.92	–	–	–	4.52	70
logistics.d	–	91.53	69/ 33	22.84	–	–	–	20.62	85
bw-large-a(9)	45.78	(5.23) –	(8/ 5) –	(2.75) –	14.67	11/4	2.0	4.12	9
bw-large-b(11)	–	(18.86) –	(11/ 8) –	(3.28) –	122.56	18/ 5	2.67	14.14	11
bw-large-c(15)	–	(137.84) –	(17/ 10) –	(5.06) –	–	–	–	116.34	19
travel1	149.74	(4.32) –	(9/9) –	(0.0) –	0.32	9/ 9	0.0	0.53	9
simple-grid1	56.40	(0.0) –	(6/ 6) –	(0.0) –	0.42	6/ 6	0.0	1.48	6
simple-grid2	–	(2.43) –	(10/ 10) –	(0.0) –	0.95	10/ 10	0.0	1.58	10
simple-grid3	–	–	–	–	3.96	16/ 16	0.0	15.12	16

Table 3.1. "Time" shows *total* running times in cpu seconds, and includes the time for any required preprocessing. Dashed entries denote problems for which no solution is found in 3 hours or 250MB. Parenthesized entries (for blocks world, travel and grid domains) indicate the performance of REPOP when using h_{oc} heuristic. #A and #S are the action cost and time cost respectively of the solution plans. "flex" is the execution flexibility measure of the plan (see below).

rocket world, blocks world and logistics are "parallel" domains which admit solutions with loosely ordered steps, while others, such as grid world and travel world admit only serial solutions.

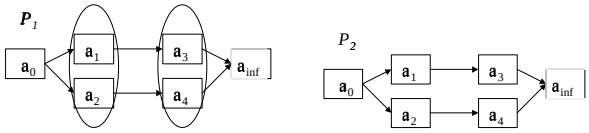
Efficiency of Synthesis: In Table 3.1, we report the total running times for the REPOP algorithm, including the preprocessing time for computing the mutex constraints (using bi-level planning graph structures [45]). Table 3.1 shows that REPOP exhibits dramatic improvements from its base planner, UCPOP, in gripper, logistics and rocket domains–all of which are "parallel domains." For instance, REPOP is able to comfortably generate plans with up to 70 actions in logistics and gripper domains, a feat that has hither-to been significantly beyond the reach of partial order planners. More interesting is the comparison between REPOP and the non-partial order planners. In the parallel domains, REPOP manages to outperform Graphplan. Although REPOP still trails state search planners such as AltAlt, these latter planners can only generate serial plans.

Despite the impressive performance of the REPOP over parallel domains, it remains ineffective in "serial" domains including the grid, 8-puzzle and travel world, which admit only totally ordered plan solutions. We suspect that part of the reason for this may be the inability of our heuristics to adequately account for negative interactions. Indeed, we found that the normal open conditions heuristic h_{oc} is better than our relaxed heuristic on these problems. It may also be possible that the least commitment strategies employed by the POP algorithms become a burden in serial domains, since eventually all actions need to be ordered with respect to each other. One silverlining in this matter is that most of the domains where POP algorithms are supposed to offer advantages are likely to be parallel domains from the planner's perspective–either because the actions will have durations (making the serial/parallel distinction moot) or because we want solution output by the planner to offer some degree of scheduling flexibility.

Plan Quality: We also evaluated the quality of plans generated by REPOP, since plan quality is seen as an important issue favoring POP algorithms. To quantify the quality of plans generated, we consider three metrics: (i) the cumulative cost of the actions included in the plan (ii) the minimum time needed for executing the plan and (iii) the scheduling (execution) flexibility of the plan.

For actions with uniform cost, the action cost is equal to the number of actions in the plan. Table 3.1 shows that REPOP produces plans with lower action cost compared to both Graphplan and AltAlt in all but one problem (*rocket-ext-b*).

We measure the minimum execution time in terms of the *makespan* of the plan, which is loosely defined as the minimum number of time steps needed to execute the plan (taking the possibility of concurrent execution into consideration). Makespan for the plans produced by Graphplan is just the number of steps in the plan, while the makespan for plans produced by AltAlt (and other state space planners) is equal to the number of actions in the plan. For a partially ordered plan P generated by REPOP, the makespan is simply the length of the longest path between a_0 and a_∞. Specifically, $makespan(P) = \max_{a \in P} est(a)$, where $est(a)$ is the earliest start time step for the (instantaneous) action a. To compute est, we can start by initializing est to 0 for all $a \in P$. Next, we repeatedly update them until fixpoint using the following rule: For all $(a_i \prec a_j) \in \mathcal{O}$, $est(a_j) := \max\{est(a_j), 1 + est(a_i)\}$. Table 3.1 shows that the solution plans generated by REPOP are highly parallel, since the makespans of these plans are significantly smaller than the total number of actions. Graphplan's solutions have smaller makespans in several problems, but at the expense of having substantially larger number of actions.



(a) A parallel plan generated by Graphplan

(b) A partially ordered plan

Figure 3.1. Example illustrating the execution flexibility of partially ordered plans over (Graphplan's) parallel plans.

Finally, we measure the execution flexibility of a plan in terms of the number of actions in the plan that do not have any precedence relations among them. The higher this measure, the higher the number of orders in which a plan can be executed ("scheduled"). Figure 3.1 illustrates a parallel plan P_1 and a partially ordered plan P_2, which are generated by Graphplan and REPOP,

respectively. Both plans have 4 actions and a makespan value of 2, but P_2 is noticeably more flexible than P_1, since P_1 implies ordering constraints such as $a_1 \prec a_4$ and $a_2 \prec a_3$, but P_2 does not. To capture this flexibility, we define, for each action a, $flex(a)$ as the number of actions in the plan that *do not* have any (direct or indirect) ordering constraint with a. $flex(P)$ is defined as the average value of $flex$ over all the actions in the plan. It is easy to see that for a serial plan P, $\forall_{a \in P} flex(a) = 0$, and consequently $flex(P) = 0$. In our example in Figure 3.1, $flex(a) = 1$ for all a in P_1, and $flex(a) = 2$ for all a in P_2. Thus, $flex(P_1) = 1$ and $flex(P_2) = 2$. It is easy to see that P_2 can be executed in more ways than P_1. Table 3.1 reports the $flex()$ value for the solution plans. As can be seen, plans generated by REPOP have substantially larger average values of $flex$ than Graphplan in blocks world and logistics, and similar values in gripper. Graphplan produces a more flexible plan in only one problem in the rocket domain.

Problem	UCPOP	+CE	+HP	+HP+CE
gripper-8	*	6557/ 3881	*	1299/ 698
gripper-10	*	11407/ 6642	*	2215/ 1175
gripper-12	*	17628/ 10147	*	3380/ 1776
gripper-20	*	*	*	11097/ 5675
rocket-ext-a	*	*	30110/ 17768	7638/ 4261
rocket-ext-b	*	*	85316/ 51540	28282/ 16324
logistics.a	*	*	411/ 191	847/ 436
logistics.b	*	*	920/ 436	542/ 271
logistics.c	*	*	4939/ 2468	7424/ 4796
logistics.d	*	*	*	16572/ 10512

Table 3.2. Ablation studies to evaluate the individual effectiveness of the new techniques: heuristic for ranking partial plans (HP) and consistency enforcement (CE). Each entry shows the number of partial plans generated and expanded. Note that REPOP is essentially UCPOP with HP and CE. (*) means no solution found after generating 100,000 nodes.

Before ending the discussion on plan quality, we should mention that it is possible to use post-processing techniques to improve the quality of plans produced by state-space and CSP-based planners. However, such post-processing, in addition to being NP-hard in general [3], does not provide a satisfactory solution for online integration of the planner with other modules such as schedulers and executors [16, 55].

Ablation Studies: We now evaluate the individual effectiveness of each of the acceleration

techniques, viz., heuristic functions for ranking partial plans (HP), and consistency enforcement (CE).Table 3.2 shows the number of partial plans generated and expanded in the search when each of these techniques is added into the original UCPOP. We restrict our focus to the parallel domains where REPOP seems to offer significant advantages.

In the logistics and rocket domains, the use of h_{relax} heuristic accounts for the largest fraction of the improvement from UCPOP. Interestingly, h_{relax} fails to help scale up UCPOP even on very small problems in the gripper domain. We found that the search spends most of the time exploring inconsistent partial plans for failing to realize that a left or right gripper can carry at most one ball. This problem is alleviated by consistency enforcement (CE) techniques through detection and resolution of implicit conflicts (e.g. the conflict between $carry(ball1, left)$ and $carry(ball2, left)$). As a result, REPOP can comfortably solve large gripper problems, such as *gripper-20*.

Among the consistency enforcement techniques, both reachability analysis and disjunctive constraint representation appear to complement each other. For instance, in problem *logistics.d*, if only reachability analysis is used with the heuristic h_{relax}, a solution can be found after generating 255K nodes. When disjunctive representation is also used, the number of generated nodes is reduced by more than 15 times to 16K.

3.6 Further Extension

There are several avenues for extending this work.

To begin with, our partial plan selection heuristics do not take negative interactions into account. This may be one reason for the unsatisfactory performance of REPOP in serial domains. We would like to be able to have a better account of subgoal interactions.

Secondly, we have been mostly interested in deriving a heuristic search strategy that can find a partially ordered plan efficiently. Thus we would like to have an admissible heuristic that guarantees the optimality of the solution plan. This of course depends on what type of solution criteria that we consider.

Finally, an interesting area worthwhile for future work involves the derivation of heuristics for more general versions of POP algorithms –including those that handle partially instantiated actions, as well as actions with conditional effects and durations. We will discuss some of these directions in more details in the following subsections.

3.6.1 Admissible Heuristics

As discussed previously, there are at least two different objective criteria for a partially ordered plan. One is in terms of the number of actions, and another is the makespan. In state space planning, our previous work [39] as well as that by Haslum and Geffner[20] provide a good basis for computing a spectrum of admissible distance-based heuristics whose accuracy can be arbitrarily achieved at the cost of the heuristic computation.

It appears more difficult to compute admissible heuristics for POP algorithms. The main problem is, again, that the set of open conditions are not necessarily in the same world state. However, the existence of partial states in a partial plan makes it possible to exploit the admissible heuristics developed in state space planning.

To begin with, let us consider optimizing the number of actions in a plan. In other words, we are interested in an admissible estimate of h^* (Section 3.3). Let $lev(S)$ be the index of the level of the *serial* planning graph in which all propositions in S first appear without mutex. Hence $lev(S)$ is an admissible estimate of the number of actions needed to achieve S when executed from the initial state [39]. To estimate $h^*(\mathcal{P})$, we may consider an existing cutset C in $\mathcal{P}$ that gives the maximum lev value. $lev(C)$ is an optimistic estimate of the cost for achieving C from the initial state, but it may not be admissible estimate for h^*, because it may also count some actions already present in the partial plan. Thus, to make this estimate admissible, we have to subtract it by the number of existing actions in $\mathcal{P}$ that may possibly come before the achievement of C. As a result, we can have the following admissible heuristic.

Heuristic 3.3 (State-based heuristic) $h_{max}(\mathcal{P}) =$

$\max_{a \in \mathcal{A}} \max\{lev(C^-(a)) - |Bef(a)|, lev(C^+(a)) - |Bef(a)| - 1\}$, *where* $Bef(a)$ *is the set of actions* $b \in \mathcal{A}$ *s.t.* $(b \prec a) \cup \mathcal{O}$ *is consistent.*

Notice that $lev(S)$ in a traditionally *parallel* planning graph would give an admissible estimate of the makespan of a plan that can achieve S from the initial state. With a slight modification, we can also derive an admissible heuristic with respect to the makespan criteria as well. In practice, however, these heuristics are typically not informative enough to solve problems of reasonable size.

3.6.2 Capturing subgoal negative interactions

Our work on state space planning [39] suggests that the negative interactions involved in achieving a partial state S from the initial state can be quantified as follows:

$$\Delta(S) = \max_{p,q \in S}(lev(p,q) - \max\{lev(p), lev(q)\}) \tag{3.2}$$

Since the information of partial states does exist in a partial plan, we conjecture that it is possible to adopt a similar mechanism for estimating the negative interactions involved in refining an initial partial plan into a partial plan $\mathcal{P}$. Another promising idea applicable to both this subsection and the previous one is that we can use n-ary state invariants (such as those detected in [15]) to improve our heuristic estimate as well as to detect and resolve more indirect conflicts in the plan.

3.6.3 Heuristics for handling partially instantiated actions

One of the advantages of the POP algorithm has been the ability to find a plan with partially instantiated actions. Dealing with partially instantiated actions may require several significant changes in the POP algorithm. First of all, A partial plan becomes a 6-tuple $\mathcal{P} = (\mathcal{A}, \mathcal{O}, \mathcal{L}, \mathcal{OC}, \mathcal{UL}, \mathcal{B})$, where $\mathcal{B}$ is a set of binding constraints (see [62] for more details). Actions in $\mathcal{A}$ may be partially instantiated, and as a result, the set of open conditions OC may also consist of propositions that are not completely grounded.

Several important issues need to be considered in dealing with these changes, such as how to deal with causal link's conflict and resolve it. However, here we will explore the question of how to rank partial plans, which seems to be one of the most important factors in improving the algorithm efficiency.

One naive way may be to ground a given partial plan $\mathcal{P}$ into a set of all grounded partial plans P_{inst} whose actions are completely instantiated according to the binding constraints. Given the heuristic estimates of completely instantiated partial plans (Section 3.3), we assign

$$h(\mathcal{P}) := \min_{P \in P_{inst}} h(P) \tag{3.3}$$

A serious setback of this method is that there are potentially many grounded instances of a partial plan, making the heuristic computation unjustifiably too costly. Another idea for getting around this problem is that for each partial plan we will keep a set of variable bindings $\mathcal{V}_b$ whose application to the partial plan $\mathcal{P}$ will result in a completely grounded partial plan P deemed to have the smallest h value. This idea, while not guaranteeing optimality, is in fact very similar to how h_{relax} heuristic is computed, in the sense that we actually compute the cost of only one single instantiated partial plan instead of minimizing over the set of all instantiated plans.

Now given a partial plan $\mathcal{P}$, let P be the grounded partial plan resulting from applying the bindings $\mathcal{V}_b$ to $\mathcal{P}$. We assign

$$h(\mathcal{P}) := h(P) \tag{3.4}$$

During the plan refinement, the variable binding set $\mathcal{V}_b$ can be updated incrementally. Given a partial plan $\mathcal{P}$ and its binding set $\mathcal{V}_b$, suppose that $\mathcal{P}$ is refined to become $\mathcal{P}'$ by adding an action a to $\mathcal{P}$. Let $a_1, a_2, .., a_k$ be all instances of a, and $P'_1, P'_2, ..., P'_k$ be the corresponding grounded partial plans of $\mathcal{P}'$. Among $t = 1, 2, ..., k$, let P'_t be a partial plan that has the smallest value of h, and v_b be the bindings involved in refining a to a_t. Thus, the binding set for the partial plan $\mathcal{P}'$ can be

updated as follows:

$$\mathcal{V}'_b := \mathcal{V}_b \cup v_b \tag{3.5}$$

3.7 Chapter Summary

The successes in scaling up classical planning using CSP and state space search approaches have generally been (mis)interpreted as a side-swipe on the scalability of partial order planning. Consequently, in the last five years, work on POP paradigm has dwindled down, despite its known flexibility advantages. In this chapter we challenged this trend by demonstrating that the very techniques that are responsible for the effectiveness of state search and CSP approaches can also be exploited to improve the efficiency of partial order planners dramatically. By applying the ideas of distance based heuristics, disjunctive representations for planning constraints and reachability analysis, we have achieved an impressive performance for a partial order planner, called REPOP, across a number of "parallel" planning domains. Our empirical studies show that not only does REPOP convincingly outperform Graphplan in parallel domains, the plans generated by REPOP have more execution flexibility. This is very interesting for two reasons. First of all, most of the real-world planning domains tend to have loose ordering among actions. Secondly, the ability for generating loosely ordered plans is very important in hybrid methods that involve on-line integration of planning with scheduling.

Chapter 4

Search Heuristics for Dynamic Constraint Satisfaction Problems and their relations to partial order planning

4.1 Motivation

In the previous chapter we have demonstrated the usefulness of several search heuristics in a partial order planning algorithm. These techniques were shown to be inspired from the search heuristics developed in state-space search and constraint satisfaction theories. The successful application of state space and CSP search heuristics to the POP algorithm, while encouraging enough, still leaves us feeling uneasy and puzzled. An inevitable question comes to mind: What if we want to generalize the techniques we developed to dealing with more complex type of constraints, such as those involving time and resources? CSP and state-space provides different facets of a POP algorithm, but they

are clearly not the most suitable model for generalizing such techniques. What is the more suitable model? A study of heuristic search control in such model – in addition to being interesting in its own right – will undoubtedly provide invaluable insights into how heuristic search control should be devised in the like of POP algorithms.

In this chapter we are set to embark on this ambitious task. We will use **dynamic constraint satisfaction problem** (DCSP) [36] as a general model based on which our study of search control for POP algorithms is pushed further. That POP search can be seen as DCSP is not much of a discovery. This connection was first remarked by Kambhampati [25] in the context of refinement search. However, we are not aware of any work that actually exploits this connection to study heuristic search control for POP algorithms. In fact, as far as we know, while work on both CSP and state space search are plentiful, there were no significant work on search heuristics for the DCSP model, which was first formalized only very recently by Mittal and Falkenhainer [36].

Briefly, a DCSP is a generalization of a CSP instance. In addition to the logical constraints among variables as in CSP, which is to be called **compatibility constraints**, there also is another type of constraint called **activity constraints**. An activity constraint says that given certain true logical conditions, a new variable is activated. A compatibility constraint is valid (applicable) only if all the variables involved in the constraint are active. We will present in this thesis a simple DCSP formulation of a POP algorithm, and point out that with some extension, one can come up with a DCSP formulation that includes time and resource constraints as well. This also demonstrates that the DCSP model appears to be powerful enough to handle a wide variety of constraint-based synthesis tasks, as argued in [36].

There are two main ways for solving a DCSP. The first one involves putting a bound on the set of all active variables and compile the DCSP into a CSP. This method has the important advantage of exploiting the extensive theory developed in the CSP literature. However, in many problems it is often impossible to know in advance the size of the DCSP solution, and the CSP approach would end up unnecessarily searching in solutionless bounds many times before finally reaching a solution.

As evidenced from the CSP approach to state-space planning, the CSP compilation may require a large amount of memory that a machine can afford, especially for problems whose solution's size is very large. In addition, the CSP compilation approach essentially ignores the difference between the nature of activity and compatibility constraints existing in the original problem that may have been more gainfully exploited. In this thesis, we follows the second way for solving a DCSP, by incrementally activating variables and assigning values for these variables until a solution is found. This method overcomes the disadvantages presented above for a CSP solving method. In particular, by discerning the different natures of activity and compatibility constraints, we may be able to come up with more suitable search heuristics and algorithms.

The heuristic search control in a general DCSP is divided into two main issues similar to that of a CSP: value ordering and variable ordering. The investigation of DCSP value and variable ordering heuristics is based on the close connection of both CSP search and state-space search. That DCSP is closely connected to CSP is trivial by definition: A DCSP without activity constraints is exactly a CSP. Alternatively, a DCSP without any compatibility constraints can be seen as a state-space search problem. A DCSP search algorithm can be seen as searching in a space of states, where the transformation from one state to another is triggered by the activity constraints. In general, a DCSP with both types of constraint can also be seen as search in a space of states, where each state is a constrained network made up of the DCSP active variables and the compatibility constraints between them.

The dual connections discussed in the foregoing allow us to come up with several general heuristics for value and variable ordering for DCSP. Briefly, we will point out that the value ordering can be done by a ranking function that ranks DCSP "states" in the search space. This ranking function is closely related to the distance-based function used in state-space search. The main work, however, lies in how to come up with an efficient and accurate approximation of this distance-based function. We present a general heuristic estimator for approximating the distance-based function based on the ideas of relaxing some of the constraints in the DCSP. The connection between this

heuristic estimator and that of the POP's distance-based heuristic estimator developed in the first part of this thesis is also examined.

Variable ordering heuristics in a DCSP are also quite interesting, and draw many insight from CSP's variable ordering heuristics. We divide the set of DCSP's variables to two, one consisting of **fertile** variables, and the other **infertile variables**. A variable is called fertile if its value assignment may activate new variables, and infertile otherwise. The difference between DCSP and CSP is that CSP has only infertile variables. Thus we propose to use CSP's variable ordering heuristics to deal with infertile variables in CSP. This characterization of DCSP variables allow us to investigate a number of different variable ordering heuristics. We also are able to show that the techniques used in REPOP (e.g the use of disjunctive ordering constraints and LIFO strategy of subgoal ordering) reflect one of the most promising general variable ordering strategies we have devised for DCSP.

The rest of this chapter is organized as follows. Section 4.2 provides a brief background on DCSP, with an emphasis on the dual view of both CSP and state space search. Section 4.3 presents a DCSP formulation for the partial order planning algorithm. In section 4.4 we present a general algorithm for solving DCSP. The value ordering heuristics are presented in section 4.5 in which we introduce the powerful distance-based function. The details on implementing such a distance-based heuristic is discussed in section 4.6, and the connection to POP's value ordering heuristics are also discussed. Section 4.7 covers the variable ordering heuristics for DCSP, and their relations to POP's variable ordering heuristics. Finally, section 4.9 discusses the limitations and outlines some future work.

4.2 Dynamic Constraint Satisfaction Problem

4.2.1 Definition of DCSP

Our definition of DCSP borrows some notations from [36]. A DCSP is a tuple $dcsp(V, V_I, D, C = C^C \cup C^A)$ such that:

- Set of variables V represents all variable that may potentially become **active** and may appear in a solution (V needs not be explicitly pre-enumerated)[1].
- $V_I = \{v_1, ..., v_k\}$ is a non-empty set of **initial variables**, which is a subset of V.
- D is a set of discrete, finite domains $D_1, ..., D_k, ...$ with each domain D_i representing the set of possible values for each variables $v_i \in V$.
- C^C is a set of **compatibility constraints** on subsets of V limiting the values they may take on. Often, a compatibility on a subset of variables $v_1, v_2, ..., v_n$ is expressed in terms of a logical formulae $P(v_1, v_2, ..., v_n)$. Furthermore, a compatible constraint (or a logical formulae for that matter), is **applicable** if and only if all variables involved in the constraint are active. In other words, for each compatibility constraint $P(v_1, ..., v_n)$, the following is true: $active : v_1 \wedge ... \wedge active : v_n \Rightarrow P(v_1, ..., v_n)$.
- C^A is a set of activity constraints of the form: $P(v_1, ..., v_n) \Rightarrow active : v_j \ \ (v_j \neq v_1, ..., v_n)$. An activity constraint is **applicable** if and only if its left hand side logical formulae is *applicable*.

A solution to a DCSP is an assignment A which meets two criteria: (1) The variables and assignments in A satisfy $C^C \cup C^A$ (2) No subset of A is a solution.

We do not consider *deactivate* constraints in this work. In other words, an active variable will always remain active. It is useful to also have an objective function on the solution of a DCSP. For instance, one may be interested in a solution with a minimum number of variables being activated. A general *linear* objective function of a DCSP solution can be written in the following form:

$$F(A) = \sum_{[v,a] \in A} f([v, a]) \tag{4.1}$$

where [v,a] corresponds to an assignment of value $a \in D_v$ to variable v in the assignment A, and $f(.)$ is a *pre-defined* non-negative cost function on such an assignment. Note that function f is applicable

[1] By definition, V may have infinite number of elements.

to only active variables, e.g $f([v,a]) = 0$ for any non-active $v \in V$. In case of an objective function of the number of (active) variables, $f([v,a]) = 1$ for all active v, and 0 otherwise.

Even if such an objective function is not required in a given DCSP, keeping one in mind can be useful in devising appropriate heuristic for the search, as it will turn out. For the time being, we will limit to DCSP with finite discrete domains.

4.2.2 DCSP as constrained state space search

In this subsection, we will construct a state space view of DCSP. The main idea is that since activity constraints essentially introduce new variables into the problem, the current set of active variables and their constraints can be seen as constituting a state. We will be more precise in the following definitions. In all cases, we are presumably given a DCSP $dcsp(V, V_I, D, C = C^C \cup C^A)$ as defined above.

Definition 4.1 (State) *The underlying* **state** S *of the DCSP consists of a set of all active variables and their (not necessarily complete) assignments in the DCSP. The set of constraints* $C_S = C_S^A \cup C_S^C$, *where* C_S *denotes the subset of constraint set* C *involving only state variables in* S, *is implicitly considered as part of state* S. *A state* S *is* **consistent** *if none of the applicable compatibility and activity constraints in* C^S *is violated.*

Before dwelling into next concepts, we will briefly review a very important notion in the CSP literature, namely, **constraint propagation** [61]. Simply speaking, this is a procedure that aims to reduce the set of possible value assignments (*i.e domain*) for each variable without losing any legal solution of the CSP. This is often done by adding new constraints to the set of constraints, deducing new value assignment and/or reducing the domain size for each variable.

Constraint propagation for compatible constraints in a DCSP is exactly the same as that in a CSP. Constraint propagation for activity constraints is quite simple, and is done in conjunction with the constraint propagation for compatibility constraints. Briefly, it is a repeated application of the following two steps:

1. For each applicable activity constraint of the form

$$P(v_1, v_2, ..., v_n) \Rightarrow active : v_j \ (v_j \neq v_1, ..., v_n)$$

if $P(v_1, ..., v_n)$ is true in S, then v_j is added to the set of active variables in S, which may in turn cause more compatibility constraints to be applicable.

2. Applying compatibility constraint propagation to add more constraints and/or value assignment.

As in CSPs, there are many different ways of enforcing the DCSP's consistency through different constraint propagation procedures, which carry different impacts on the overall performance of a DCSP solver. We do not consider constraint propagation techniques in this work, however. From now on, we will use the term "constraint propagation" to imply a particular constraint propagation $\mathcal{CP}$ that is chosen for our DCSP algorithm.

Assumption 4.1 *Unless otherwise noted in this thesis, by "constraint propagation" we mean any particular constraint propagation procedure, simply named $\mathcal{CP}$, that is chosen for the algorithms in consideration.*

We have seen that through the means of constraint propagation, one can transform the underlying state of a DCSP by changing its set of active variables, value assignments and the set of constraints. A constraint propagation procedure is typically triggered by a new value assignment of a DCSP variable. This leads to the notion of operator defined as follows.

Definition 4.2 (Operator) *An* **operator** *is a value assignment to an active variable, namely, op is of the form assign$[v, a]$. The preconditions of op are active : v, v remains unassigned and $a \in D_v$. The effects of the application of op to state S are new active variables, value assignment and constraints resulting from propagating $v = a$ through state S.*

In addition, the **initial state** of the DCSP is simply $S_0 = V_I$. The **goal state** G_A is a state that has no active variables remaining unassigned, and all constraints in C^C satisfied. In short, $G_A = \{v, [v, a] | [v, a] \in A\}$. It is simple to prove the following.

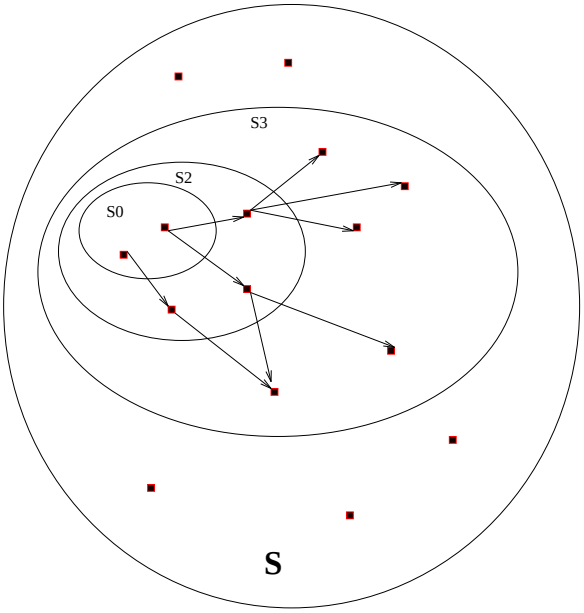


Figure 4.1. Illustration of DCSP's states.

Proposition 4.1 *Let A be a solution (i.e set of value assignments) to a DCSP. There exists at least one sequence of operators of the form $op = [v, a]|[v, a] \in A$ such that when applied to the initial state S_0 will results in the goal state G_A.*

Figure 4.1 provides a simplified example of DCSP states. In the figure, S is the set of all possible state variables, each of which is represented by a squared point. The initial state is S_0. The variables' value assignments, each of which corresponds to applying an value assignment operator of variables in S_0, results in the activation of new state variables that make up state S_1, and so on. For each point $v \in S_i$, a set of arrows pointing to v from points in S_{i-1} represents an activity constraint for variable v. What is not shown in the figure is the compatibility constraints among active variables in each S_i.

4.2.3 Some useful DCSP concepts

In this subsection we introduce several DCSP concepts that will prove useful later on.

1. **Fertile/Infertile variables**: The presense of activity constraints in DCSP allows us to characterize the set of DCSP variables into two categories that we call fertile and infertile variables. A variable is **fertile** in a current DCSP state S if at least one of its value assignment will add one or more new active variables to S through constraint propagation. Otherwise, it is called **infertile** variable. In addition, any variable after assigned a value will also become **infertile**.

2. **Descendent graph**: We call the new active variables propagated from value assignment of a fertile variable v **children** of v. A fertile variable may have one or more children, some of which may be fertile variables and have children themselves and so on. We use the term descendent graph to capture the set of all descendents of a fertile variable v. More precisely, a **descendent graph G** for a fertile variable v is a directed acyclic graph whose nodes are different DCSP variables starting with root node v. Each directed arc (x, y) in G corresponds to a constraint by which some value assignment of x results in the activation of y. To distinguish between two different types of constraint, the arcs corresponding to activity constraints are drawn as solid arrows, and arcs corresponding to compatibility constraints are dotted arrows.

3. **Productive/Unproductive constraints:** A constraint is **productive** if it corresponds to some arc in a descendent graph for a fertile variable. Otherwise, it is **unproductive**. Obviously, all activity constraints, when applicable, are productive. On the other hand, some (applicable) compatibility constraints are and some are not.

Finally, it is important to note that the nature of the variables, constraints and the descendent graph as defined above does depend on the specific constraint propagation procedure $\mathcal{CP}$ being used. Figure 4.2 shows a taxonomy of different types of constraints that are *applicable* to the fertile/infertile variables at the time each of them is assigned some value. These concepts will be illustrated in details in the next section on a DCSP formulation of a POP algorithm.

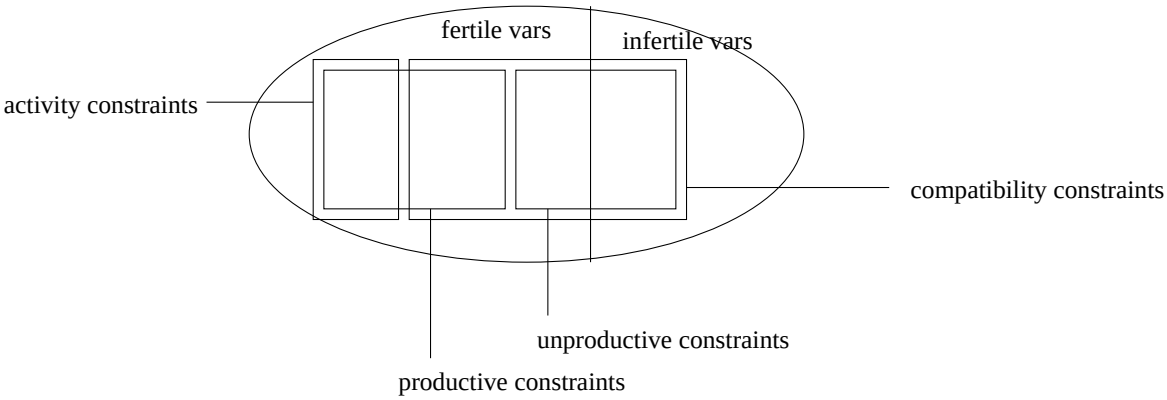


Figure 4.2. A taxonomy of DCSP constraints that are applicable to fertile and infertile variables when these variables are assigned values.

4.3 A DCSP formulation of POP algorithm

Plan synthesis problems are excellent domains for the DCSP model formulated in the previous section can be applied. We will make this connection explicitly in this section.

Consider a planning problem $\mathcal{P}= (I, \Omega, G)$, where I and G are initial and goal state, respectively, and Ω is a set of possible actions. For each action $a \in \Omega$, denote $Prec(a), Eff(a)$ as the set of preconditions and effects of action a, respectively. Denote P as the set of all propositions p and $\neg p$ about the world. The planning problem is concerned with finding a set of (totally or partially) temporally ordered actions such that when executed from the initial state I will achieve goal state G.

All classical planning approaches fall under the category of DCSP. Forward state space search planners [48, 5], as the name indicates, corresponds to DCSP with no compatibility constraints, i.e any new state being propagated already satisfies C^C. Graphplan's[4] backward search was shown to be a simple DCSP instance as well[28, 27]. Here, the presence of mutex constraint makes up the C^C set. The readers are referred to Appendix A for DCSP formulations of both forward and backward state-space search. In the following we will present a DCSP formulation of a partial order planning algorithm.

4.3.1 Partial-order (plan-space) planning as DCSP

We will examine a partial order planning (POP) algorithm, which is most closely resemble to the full-fledged DCSP model. Briefly, POP algorithm searches in the space of partial plans, each of which can be seen as a set of variable and constraints. In each partial plan (i.e "state" in the terminology in section 4.2), variables are the set of open conditions, and the set of actions. Only open conditions are active variables remained to be assigned. An open condition can be assigned by a new action, which in turn activates further open conditions, and propagating more unsafe link induced disjunctive ordering constraints and causal link constraints.

- Set of variables V consist of the following:
  - Set of steps $\{s_{-1}, s_0, s_1, s_2, ...\}$.
  - Set of open conditions $\{openc(s_i, p) | i \geq 0; p \in P\}$, where P is the set of all atomic propositions about the world.
  - Set of precedence constraints $\{(s_i \prec s_j) | i, j = -1, 0, 1, ...\}$.
  - Set of causal links $\{causal(s_i, s_j, p) | i \geq -1, j \leq 0, p \in P\}$.
- Domains:

  For each variable $s_i (i > 0)$, $D_{s_i} = \Omega$. $D_{s_0} = D_{s_{-1}} = \{null\}$.

  For each variable $openc(s_k, p), D_{openc(s_k,p)} = \{s_{-1}, s_0, s_1, ..., s_M\}$, where M is the maximum length of the plan we would like to find.

  For each variable $causal(s_i, s_j, p), D_{causal(s_i,s_j,p)} = \{true, false\}$.

  For each variable $(s_i \prec s_j), D_{(s_i \prec s_j)} = \{true, false\}$
- Initial state: Including the following active variables:

  $s_{-1} = null; s_0 = null$, corresponding to two null first and last step, respectively.

  $openc(s_0, p) = true$ for all $p \in G$.

  $(s_{-1} \prec s_i) = true; (s_i \prec s_0) = true$ for all $i > 0$; $(s_i \prec s_j)$ for $i \neq j$ for all other variables of

this type.

$causal(s_i, s_j, p)$ for all $i \neq j$ and $p \in P$.

- Constraints: For $k = 0, 1, 2, ...$

  - Activity constraints

$$xs_k = a_l \Rightarrow active : openc(s_k, p)\ \forall p \in Prec(a_l) \quad (4.2)$$

$$openc(s_k, p) = s_i \Rightarrow active : s_i \wedge active : causal(s_i, s_k, p) \wedge active : (s_i \prec s_k)$$
$$\forall i = -1, 0, 1, ..., M. \quad (4.3)$$

  - Compatibility Constraints

$$openc(s_k, p) = s_i \Rightarrow (i \neq k) \wedge (s_i \prec s_k) \wedge causal(s_i, s_k, p)$$
$$\forall i = -1, 0, 1, ..., M. \quad (4.4)$$

$$causal(s_i, s_k, p) \Rightarrow \bigvee_{a_l \in \{a \in \Omega | p \in Eff(a)\}} s_i = a_l \vee (i = -1 \wedge p \in I) \quad (4.5)$$

$$causal(s_i, s_j, p) \wedge (s_k = a_l) \wedge \neg p \in Eff(a_l) \Rightarrow (s_k \prec s_i) \vee (s_j \prec s_k)\ \forall p \in P \quad (4.6)$$

$$(s_i \prec s_j) \wedge (s_j \prec s_k) \Rightarrow (s_i \prec s_k) \quad (4.7)$$

$$(s_i \prec s_j) \Rightarrow \neg(s_j \prec s_i) \quad (4.8)$$

    Constraint 4.2, 4.3, 4.4 and 4.5 are concerned with the causal effects of actions. Constraint 4.6 makes sure that the plan is correct in the face of actions' negative effects. Constraints 4.7 and 4.8 take care of basic properties of the ordering relations.

- Objective function is of the form 4.1, where function $f(.)$ is pre-defined as follows: $f([s_k, a_j]) = 1$ for all k, j, and 0 for all other assignments. As will be shown in the next section, an optimal solution to this DCSP corresponds to a solution partial order plan with minimum number of actions.

In the DCSP formulation above, it is simple to see that each "state" of the DCSP instance represents exactly a partial plan in the POP algorithm. Also notice that only activate constraints are

needed to encode the planning problem. It is also simple to see that that other types of constraints such as resource and temporal constraints can be added to this DCSP formulation as well (see Appendix A for more details). For instance, the introduction of (discrete) resource constraints will only result in partial plans with more ordering constraints. The introduction of temporal constraints will likely require effective handling of DCSP with numerical domains.

The readers will notice that the DCSP formulation above is very similar to the causal encoding provided by Kautz and Selman [32]. Essentially, if we translate the *active:v* into just another value for variable v, we will get an encoding like that of Kautz *et al*'s. Again, it is reminded that we are pursuing the direct approach to solving DCSP rather than solving its CSP translation.

Before ending this section, we would like to relate this DCSP formulation to some of the DCSP-specific concepts introduced in subsection 4.2.3.

- **Fertile/Infertile** variables: All variables appear in the left hand side of activity constraints are fertile variables ($s_k, openc(s_k, p)$). The value assignments of these variables are also triggered through constraint propagation by constraint 4.5, whose left hand side is a value assignment of $causal(s_i, s_k, p)$. Thus $causal(s_i, s_k, p)$ also is a fertile variable. The only variables left are of the form $s_i \prec s_j$, which is infertile.

- **Productive/Unproductive** constraints: All activity constraints (4.2, 4.3) are productive. Compatibility constraints whose right hand side consists of value assignments for fertile variables (4.4, 4.5), and thus are productive. The rest (constraints 4.6, 4.7, 4.8) are unproductive constraints.

- **Descendent Graph**: Figure 4.3 illustrates a descendent graph for fertile variable s_i. Of course, the subgraph starting with the root $openc(s_k, p)$ corresponds to a descendent graph for $openc(s_k, p)$, and so on. In each edge of the graph we mark the corresponding constraints, where (1),(2),(3),...,(7) correspond to constraints 4.2, 4.3,..., 4.8, respectively. The constraints most responsible for generating new variables are marked (1),(2),(3) and (4). The other last 3 constraints marked as (5,6,7) do not appear in the graph. Apparently they are unproductive

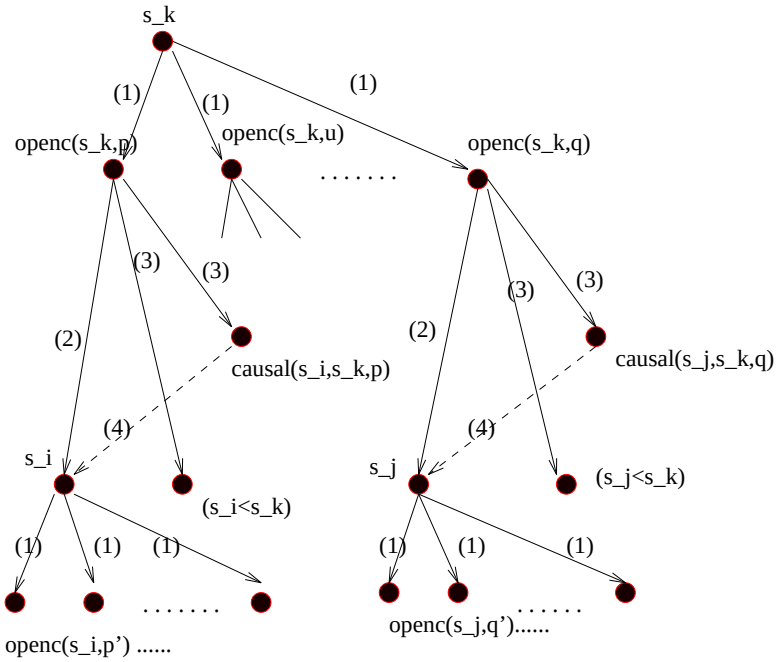


Figure 4.3. Descendent graph for a fertile variable in the DCSP formulation of a POP algorithm.

constraints that do not contribute to the activation of any new variables.

4.4 A general algorithm for solving DCSP

There are two main ways for solving a DCSP. The first one involves putting a bound on the set of all active variables and compile the DCSP into a CSP. This method has the important advantage of exploiting the extensive theory developed in the CSP literature. However, in many problems it is often impossible to guess in advance the size of the DCSP solution, and the CSP approach may end up unnecessarily searching in solutionless bounds many times before finally reaching a solution, or it may search under a bound that is much greater than the optimal size of the solution. In addition, as evidenced by the CSP approach to state-space planning, CSP encodings may require a large amount of memory that a machine cannot afford, especially for problems whose solution's size is very large. In addition, the CSP compilation approach essentially ignores the difference between the nature of activity and compatibility constraints existing in the original problem that

may have been more gainfully exploited. In this thesis, we follows the second way for solving a DCSP, by incrementally activating variables and assigning values for these variables until a solution is found. This method overcomes the disadvantages presented above for the CSP solving method. In particular, by discerning the different natures of activity and compatibility constraints, we may be able to come up with more suitable search heuristics and algorithms. In this section we present a general and simple algorithm that follows this idea.

The algorithm consists of a main *choose/ propagate* cycle. Each choose step selects an active, unassigned variable and assigns it a value that has not been previously ruled out. Each propagate step checks the constraints relevant to the new variable value assignment and propagates their consequences and dependencies.

Algorithm 4.1 (General DCSP algorithm)

1. Let $V = V_I$.

2. Repeat

3. Choose an active variable v remains in V.

4. Choose a value a to assign to v.

5. Repeat

6. Propagate $v = a$ through all activity and compatibility constraint relevant to v

7. Add all new variables and new assignments to V.

8. Until no new active variables or constraints are added.

9. Return NULL if one constraint is violated.

10. Until every active variable in V is assigned a value

11. Return V

A lot of specific details are left out in the algorithm template above. By "choose" we mean a nondeterministic choice is made. In practice, these choices on variable ordering (line 3) and (line 4) play the ultimate role in the algorithm's efficiency. There may be different strategies for dealing

with these choices. Since we generally cannot make a right choice that leads to a solution every time, different backtracking schemes may also have different impacts on the algorithm performance. In addition, different levels of constraint propagation routines (lines 6 and 7) result in different new sets of variables and value assignment. Finally, we haven't talked about how the solution's objective function is taken into account.

In this chapter we will be mainly interested in the variable and value ordering in a DCSP, and somewhat in the account of solution's quality. As before, we will assume that a particular constraint propagation procedure, namely $\mathcal{CP}$, is used in our algorithm.

In traditional CSPs, effective variable ordering strategies are generally thought to reduce the number of backtracks encountered by the search [61]. Often, these heuristic are devised by analyzing the underlying graph that captures the constrain-based relationship among variables. For example, a strategy such as *Minimal width ordering* (MWO) heuristic exploits the topology of the nodes in the underlying graph of the problem to come up with an ordering whose intention is to reduce the need for backtracking. A strategy such as *most constrained variable first* heuristic aims to detect the failure as early as possible for backtracking.

On the other hand, when choosing a value to assign a variable, value ordering strategies in traditional CSPS tend to choose the value that is most likely to succeed, because failure in this case would cause backtracking. There may be many ways to evaluate the likelihood of success of a value. For instance, one heuristic called *min-conflict* heuristic, which basically orders the values according to the conflicts which they are involved with the unassigned variables.

Although it seems that in the CSP literature more attention is paid to finding effective variable ordering strategies in comparison to value ordering strategies, it should be noted that a good value ordering heuristic plays very important role in solving a CSP. To see this, imagine a perfect heuristic for value ordering, which would render any variable ordering strategies irrelevant and unnecessary. One the other hand, a good variable ordering strategy alone does not guarantee that a solution be found quickly. More often, it is impossible to come up with perfect heuristics for

both variable and value ordering, and these two heuristics are likely to complement each other.

4.5 Value ordering heuristic for DCSP

In traditional CSPs, values deemed most likely to succeed will be chosen. For example, the *min-conflict* heuristic states that a value assignment that results in least number of "conflicts" with other unassigned variables will be favored. The number of conflicts are estimated based on the set of (compatible) constraints present in the CSP. In a DCSP, it is not enough to choose a value that is most likely to satisfy all current compatibility constraints. Note that a value assignment may result in activation of one or more new variables. Therefore, it is also important to make sure that these new variables have a high likelihood of being part of a final solution.

This section is concerned with value ordering strategies for DCSP. Given a variable v, we have to choose which value to assign to v. There are two possibilities, v can be either fertile or infertile variable. If v is an **infertile** variable, by definition its value assignment does not activate any new variables. Thus the only measure one can use to evaluate the chance of success of a value is based on the set of compatibility constraints. We can use traditional CSP techniques for dealing with this. For the rest of this section, we will consider value ordering strategies for **fertile** variables.

Once a given fertile variable v is assigned with a value $a \in D_v$, the operator $op = assign[v, a]$ triggers a transformation from a current underlying state of the DCSP to another state made of new variable assignments. Therefore, a value ordering strategy for DCSP can be done on the basis of ranking the underlying states using some ranking function.

For each operator $op = assign[v, a]$, define

$$f_c(op) = \sum_{Assign[u=b] \in Eff(op)} f([u, b]) \tag{4.9}$$

where $Eff(op)$ is the set of value assignments $u = b$ resulting as effects from executing op *and* v was previously unassigned. $f(.)$ is a cost function defined in equation 4.1. Denote $Res(S, op)$ as the resulting state of applying op to S.

Definition 4.3 (Distance-based function) *Given a sequence of operators* $OP = op_1, op_2, ..., op_n$ *that are applied to a state* S_0 *resulting in a sequence of states* $S_1, S_2, ..., S_n$, *let* $Res(S_i, op_i) = S_{i+1}$ *for all* i. *A distance-based function with respect to operator sequence* OP *is defined recursively as:*

$h(S_0, S_0, OP) = 0$, *and*

$h(S_0, S_{i+1}, OP) = h(S_0, S_i, OP) + f_c(op_i)$ *for* $i = 1, 2, ..., n$.

The following equation follows immediately

$$h(S_0, Res(S_0, OP), OP) = \sum_{op_i \in OP} f_c(op_i) \tag{4.10}$$

$h(S_0, Res(S_0, OP), OP)$ is called the distance-based function between two states S_0 and $Res(S_0, OP)$.

Proposition 4.2 *Let* A *be a solution (e.g a set of variable assignments) to a given DCSP. Let* OP *be a sequence of operators corresponding to the assignments in* A, *and* G_A *be a goal state obtained by applying* OP *to the initial state* $S_0 = V_I$, *we have* $F(A) = h(S_0, G_A, OP)$

Proof. Simple by induction on the number of operators n in sequence OP.

Proposition 4.2 makes explicit the connection between DCSP and search in the space of states made of constraint networks. To be specific, it shows that the value of objective function of a solution A is equal to the "distance" – which is measured by the distance-based function h – between the underlying initial state and a goal state of the DCSP.

The distance-based function h provides a powerful basis for heuristic search control of the DCSP. Consider the algorithm template given in [36], which basically starts with the initial state S_0 and systematically finds a sequence of operators OP that leads S_0 to a goal state G_A. Assume that we have an admissible estimate $h^*(S)$ that satisfies the following:

$$h^*(S) \leq H(S) \tag{4.11}$$

where $H(S)$ is defined as the minimum "distance" between S and a goal state G_A

$$H(S) = \min_{OP, G_A} h(S, G_A, OP) \tag{4.12}$$

for any applicable sequence of operators OP that can transform state S to some goal state G_A. We will choose value whose assignment to a given variable would result in a state of smallest value of $\sum f_c(op_i) + h^*(S)$, where op_i are the operators applied so far. The following proposition follows immediately from Proposition 4.2 and the theory of admissible state space heuristic.

Proposition 4.3 *The h^* heuristic function for ranking states is admissible and thus ensuring that the solution found to be optimal in terms of objective function $F(A)$. Furthermore, $F(A) = H(S_0)$.*

When optimality is not insisted on, a close approximation of $H(S)$ can be used to find a solution fast, while the solution's quality is typically close to being optimal.

4.6 Estimating distance-based heuristic function

The general way in approximating $H(S)$ is by constructing a sequence of operators "connecting" S and G_A under the relaxation of certain constraints in $C = C^C \cup C^A$. Since our concentration is on a class of DCSP were C^A is very dominant as well as C^C, we have to selectively account for both types of constraints.

We will present here a distance approximation that, while not being admissible, may yield very close estimate of $H(S)$. The basic idea is simply find *one* single set of operators that can be applied to a state S to become a goal state G_A, under the assumption that some of the constraints in C are relaxed.

Heuristic 4.1 (Constraint-relaxed heuristic $h_c(S)$)

1. *Let C' be a pre-defined subset of C.*
2. *Let $S_t = S$*
3. *Repeat*
    4. *Deterministically choose an active variable v remains in S_t.*
    5. *Deterministically choose an action $op = assign[v, a]$.*
    6. *Let $S_t := Res(S_t, op)$.*

7. Make sure no constraint in C' is violated as

$v = a$ is propagated though C'.

8.$h_c(S) := h_c(S) + f_c(op)$.

9. Until every active variables in S_t is assigned a value

10. Return $h_c(S)$

Obviously, there are two issues that play the most important roles in the effectiveness of this heuristic. The first issue asks which constraints are to be relaxed and which are not (e.g what is C'). The second issue involves finding a specific scheme for selecting variables and value assignments so that the heuristic estimation procedure above will terminate and return good heuristic estimates. We do not have a general answer for the second issue. It is likely that such a scheme has to be derived from the specific problem.

To find a solution for the first answer, an analysis on the set of constraints given in the DCSP is needed. Intuitively, we do not want to relax too much, because the more constraints we relax, the less informed the heuristic estimate will be. On the other hand, if we relax too little, the estimation will probably be almost just as hard as solving the original problem.

The *descendent graphs* (defined in subsection 4.2.3) provide important information on the relative significance of different constraints with respect to the size of the solution (which can be quantified in terms of the total number of active variables making up the final solution). Specifically, since unproductive constraints do not contribute to the activation of variables, it is likely that relaxing these constraints do not lose us any important variables in the correct solution. Thus we have the following heuristic:

Unproductive constraints are relaxed in estimating distance-based function H, namely, C' consists of only productive constraints in C in constraint-relax heuristic $h_c(S)$.

The relaxation of unproductive constraints may come at a price depending on the tightness of these constraints. While productive constraints' main "responsibility" is in producing new variables

for the solution, unproductive constraints' main job is in limiting the set of possible assignments of such variables in the solution. In problem that the set of unproductive constraints are very tight, $h_c(S)$ will be inaccurate unless some of the unproductive constraints are taken into account.

4.7 Variable Ordering for DCSP

In Section 4.2.3 we defined two types of DCSP variables: fertile and infertile variables. It is clear by now that these two types have different characteristics, and it is also likely that they have to be treated differently in our variable ordering strategies. We will have to confront three different heuristic issues:

1. How to order among infertile variables.
2. How to we order among fertile variables.
3. How to order between the 2 sets of infertile variables and fertile variables.

The reader may be quick to point out that, the third issue already precludes a variable ordering heuristic that does not distinguish between fertile and infertile variables. It is likely that such a heuristic would be more effective for certain class of problems where both activity and compatibility constraints interact in very complex ways that it becomes a burden to distinguish between fertile and infertile variables. Fortunately, it seems that in most planning problem specification these two types of constraint are quite loosely coupled. Therefore in this work we will only consider ordering heuristics that exploit the different characteristics of fertile and infertile variables.

We now turn to the second issue first, and for good reasons. For infertile variables, the only applicable constraints are unproductive (compatibility) constraints. Like traditional CSPs which have essentially only infertile variables, the general purpose of variable ordering for infertile variables in a DCSP is to reduce the number of backtracks in the search algorithm. A backtrack occurs only when at least one unproductive constraint is violated. Clearly we may exploit many variable ordering heuristics in the CSP literature to the set of infertile variables here.

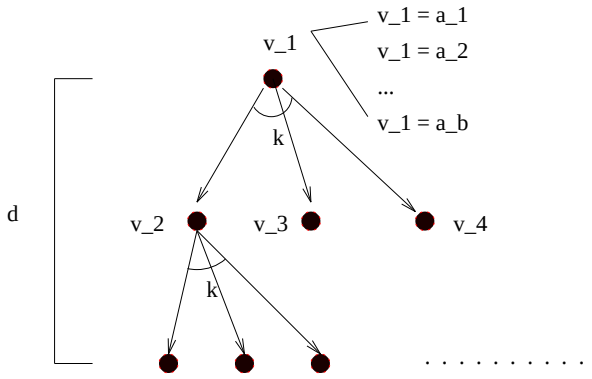


Figure 4.4. A simplified asymptotic evaluation of each fertile variable's search space.

We will now consider ordering strategies for fertile variables. When a fertile variable is chosen and assigned a value, through constraint propagation we will get more active variables into the DCSP state, which may in turn add more active variables, unproductive constraints, and productive constraints to the DCSP state. As discussed before, only productive constraints are responsible for generating new variables. Thus, let us not worry about the productive constraints by assuming that only productive constraints are present in the problem. It is intuitive to favor a fertile variable whose value assignment would activate the smallest number of successor variables that are part of the final solution. This is because the smaller the descendent graph, the less amount of search will be needed until all of the variable's children become infertile.

We will give a simplified asymptotic estimate of the descendent graph's size. By removing all edges in the graph that corresponds to compatibility constraints, we have a graph whose edges are activity constraints only. Since a variable gets activated only once (by the last value assignment of variables in the left hand side of an activity constraint), there are no cycles in this graph. Thus we get a subgraph, which we would call a **descendent tree**, with the same number of nodes as the descendent graph. Consider a fertile variable v. Let d be the average depth of the descendent tree needed to be reached until all successor variables of v become infertile. Let k is the average number of child variables activated when a fertile variable is assigned value. Let b be average number of possible values for each variable, then the search space involved is of the size b^{k^d} (see Figure 4.4.)

According to the foregoing asymptotic analysis, b has the more important role in reducing the search space w.r.t. a fertile variable, followed by k and then d. Estimating b, d and k may be difficult. d can be estimated using a procedure similar to the distance-based function developed in Section 4.12 (with each operator's cost would be 1, since we essentially count only the number of operators). The knowledge of b and k may depend on a particular problem domain. In practice, we may simply want to favor variable that has small value of either b or k or d, assuming the rest are unknown or equal.

Finally, we turn our discussion to the third issue: How to order between the set of fertile variables and infertile variables. One may either favor fertile variables over infertile ones or vice versa. Thus we have the following two heuristics:

1. Always favor fertile variables over infertile ones. Among fertile variables, choose the one whose corresponding b^{k^d} is smallest. When there is no longer fertile variables, traditional variable ordering heuristics for CSP can be applied to infertile variables.

2. Always favor infertile variables over fertile ones. The ordering heuristic among infertile variables may follow any traditional CSP's heuristic, while the ordering among fertile variables may be based on the asymptotic analysis above.

Recall that the value assignment for fertile variables will increase the size of the solution by introducing additional variables, whereas value assignment for infertile variables will narrow the set of possible solution. When the compatibility constraints (among infertile variables) are tight, favoring infertile variables may be a good idea, because the early commitment may give us early "rewards". On the other hand, when the compatibility constraints are loose, the first heuristic would be more helpful.

In the next section, we will show that the search strategy in our planner RePOP is closely resemble to the first strategy, and is empirically proven to be quite effective in a large class of planning problems in comparision to the second strategy.

4.8 Relations to variable and value ordering heuristics in POP algorithms

Having investigated a number of search heuristics for DCSP (Sections 4.5 and 4.7), and discussed the "partial-order planning as DCSP" view (Section 4.3), we are ready to review the heuristics developed in the previous chapter in the context of DCSP, and to discuss possible avenues where the heuristics can be further improved.

In the DCSP formulation of the POP algorithm, we have four types of variables $s_i, \mathit{openc}(s_i, p), \mathit{causal}(s_i, s_j, p), (s_i \prec s_k)$. Among these, variable of the form $\mathit{causal}(s_i, s_j, p)$ is always assigned to be true as soon as it is activated (according to constraints 4.2 and 4.4), so there is no search involved w.r.t this variable. Thus we will only be concerned with 3 types of variables $s_i, \mathit{openc}(s_i, p), (s_i \prec s_k)$.

In a POP algorithm like RePOP, there are several choice points:

1. Flaw selection: A flaw can be an open condition or an unsafe link.

2. Flaw resolution: If a flaw is an open condition, we have to find a step to support that open condition. The step can be any old step, or it can be a new one. If the flaw is an unsafe link, there are two choices (promotion or demotion).

Each flaw in RePOP can be seen as a variable in a DCSP formulation. There are two types of variables, open condition and unsafe link. Thus, flaw selection is directly connected to the issue of variable ordering, while flaw resolution is essentially a matter of value ordering.

One can easily draw a parallel between the RePOP algorithm with that of the DCSP formulation presented in Section 4.3. For instance, dealing with open conditions in RePOP is parallel to dealing with two variables $\mathit{openc}(s_i, p)$ and s_i. That is because a value assignment for $\mathit{openc}(s_i, p)$ immediately activates a new variable s_k whose possible values are limited to the set of actions that can achieve p. Dealing with unsafe link flaw in RePOP is parallel to dealing with ordering variable $s_i \prec s_j$, whose only constraints of disjunctive nature (where search is needed) are

of the form $s_i \prec s_j \vee s_k \prec s_i$.

Lest the DCSP formality obscures the intuition, the following investigation of REPOP heuristics will be done directly within the context of the REPOP algorithm *without* referring to the specific DCSP formulation presented in Section 4.3. After all, our main purpose of studying the DCSP model is to gain a better understanding of its heuristics, and maybe to apply new heuristics to our planning algorithms. It is quite plausible to solve a planning problem by simply generating it into a DCSP models but that is not in our main interest in this work.

4.8.1 Variable ordering for REPOP

Recall that there are two kinds of flaws (variables) in the REPOP algorithm. They are open conditions and unsafe links. Open conditions can be seen as fertile variables, because their assignment (of a new action step) introduce new flaws into the partial plan. On the other hand the resolution of unsafe link flaws do not introduce any new flaw.

In REPOP as long as other open conditions exist, all unsafe link flaws are simply represented in terms of disjunctive ordering constraints $s_i \prec s_j \vee s_k \prec s_i$ without splitting these disjunctions into search space. **This is an application of the fertile-first variable ordering strategy** discussed in Section 4.7.

Now we consider the variable ordering among infertile variables, *i.e* unsafe links. It turns out in our experiment with REPOP that through a simple constraint propagation procedure, all disjunctive constraints $s_i \prec s_j \vee s_k \prec s_i$ disappear by the time all open conditions are assigned an action. This shows that the unproductive (compatibility) constraints wrt infertile variables $s_i \prec s_j$ is not tight in a large number of planning domain.

Of particular interest is the ordering of the set of open conditions (*i.e* fertile variables) in REPOP. A strategy called LIFO [17] is used. Briefly, when a new open conditions are added into the partial plan, they are put into an open condition stack. The last open condition entering the stack will be chosen for resolution. It can be argued that **the LIFO ordering strategy can be seen**

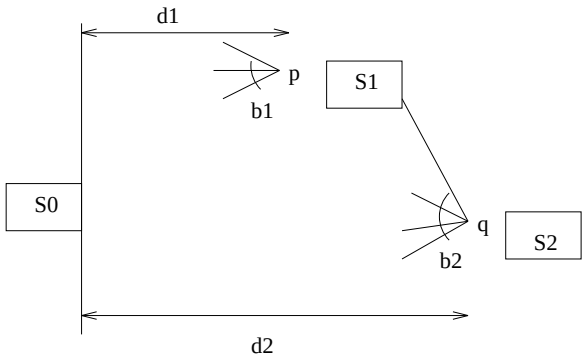


Figure 4.5. LIFO ordering strategy typically favoring fertile variables with small b and d

.

as favoring fertile variables with small b and d, according to the above asymptotic analysis.

Indeed, consider two open conditions p and q whose corresponding steps are s_1 and s_2, respectively. In addition, suppose that p is selected before q according to the LIFO scheme. It can be proven that $s_1 \preceq s_2$ must be true. [2] Now, d_1 and d_2, defined as the depth of the descendent tree for p and q, respectively, can be roughly measured by the distance from the initial state to action step s_1 and s_2. Since s_1 precedes $s2$, $d_1 \leq d_2$ (See Figure 4.5 for an illustration).

To compare the branching factor b_1 and b_2 for p and q, respectively, we would make an assumption that the number of action instances in the given planning domain that can achieve p and q are the same. For an open condition p, b_1 is the number of new action step that can achieve p, plus the number of old steps in the partial plan that can be reused. The same can be said about the branching factor b_2 for q. They are almost the same by our assumption, except for the fact that there may be additional action steps s such that $s_1 \prec s \prec s_2$ that can be reused for achieving q. This implies that $b_1 \leq b_2$.

The above analysis allows us to suggest several possible avenue for further improvement for the variable ordering heuristic. To begin with, we may want to take k (defined as the average number of children for a fertile variables) into consideration. In fact, k was shown to be more

[2] Since p is selected before q according to the LIFO scheme, p must be activated after q was activated. This implies that there exist another open condition q' in the same action step s_2 as q. According to LIFO again, a partial plan that achieves q' must be completely constructed before q can be chosen, because any open condition activated as children of q' must be chosen before q. It follows that p must be one of q''s children or q' itself. This implies that the step that p belongs to must precede or equal the step that q belongs to. Thus $s_1 \preceq s_2$.

important than d. In REPOP, k is the average number of preconditions for each action instance, thus depends directly to the specific problem domain. This suggests that action with fewer number of preconditions may be favored in conjunction with the other heuristics. The derivation above of $b_1 \leq b_2$ is unlikely to hold when assumption involved is no longer true. For example, it is likely that some open condition has more possible achieving action instances than other according to the specific problem domain. This can be taken into account to create a more robust heuristic.

4.8.2 Value ordering heuristics for REPOP

The relax heuristic presented in Section 3.3 for a partial order planning algorithm (REPOP) is a special case of the general heuristic estimator presented in this section. Indeed, in this general heuristic estimator, the current state S_t (line 2) corresponds to the current partial plan in the POP algorithm. Line 4 corresponds to choosing an open condition (subgoal) in the current partial plan. Line 5 corresponds to choosing an action that can support the chosen subgoal. The loop (line 3-9) terminates when there are no more active variables unassigned, corresponding to having no more open conditions. Line 8 corresponds to the recurrent equation 3.1.

Several important pieces are specific to the POP algorithm that need further clarification.

- The first technique specific to POP algorithm's heuristic is the use of planning graphs to guide the selection of of variables and value asignments and to terminate the estimation procedure. A planning graph, which is built from the planning problem's initial state, provides a sort of relevance based analysis, which helps narrow the set of value for assignment, and can be used to influence to choice of variable (see Section 3).

- The choice of which constraints are relaxed in the estimation of H function. Figure 4.6 provides useful information of different types of constraints in the DCSP formulation. It shows that only constraints marked as (1),(2),(3) and (4) (corresponding respectively to constraints 4.2, 4.3, 4.4, 4.5) are productive constraints that are responsible to producing new variables for the solution. According to the *productive constraints-only* strategy (Section 4.5), unproductive

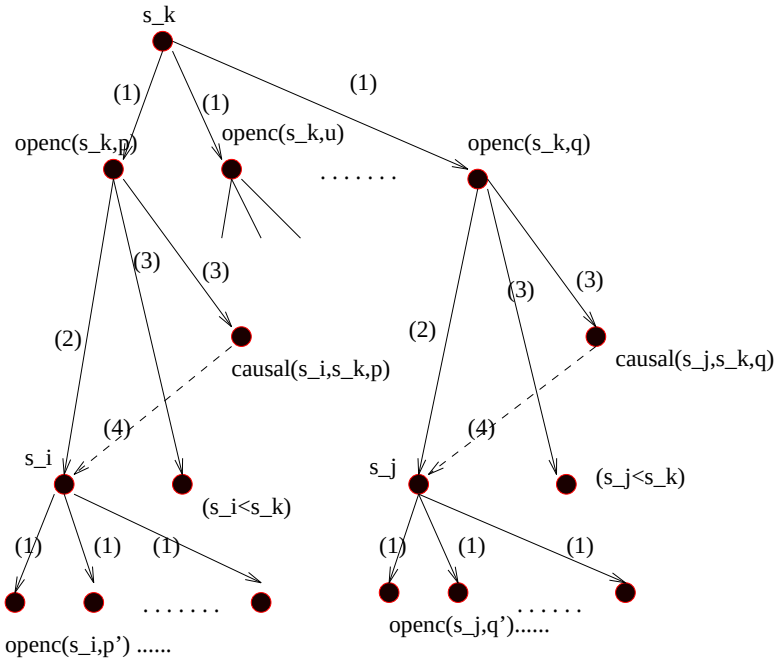


Figure 4.6. Descendent graph for a fertile variable in the DCSP formulation of a POP algorithm.

constraints such as 4.6, 4.7, 4.8 are relaxed in the estimation of H function. These constraints are mainly concerned with the negative effects of actions. This is exactly the type of constraints that get relaxed in the REPOP's relax heuristic.

In domains where the unproductive constraints are tight, relaxing all of them will render the heuristic estimate inaccurate. For instance, domains such as blocks world, grid worlds, 8-puzzle all belong to this type of domain. This suggests that it may be useful to account for certain unproductive constraints in dealing with this type of domain.

4.9 Limitations and Future Work

We have shown the connection between search strategies in partial order planning with that of a dynamic constraint satisfaction problem, and emphasized the importance of studying general search strategies for the general DCSPs. Our presentation of value and variable ordering heuristics for DCSP remains preliminary, however. A more rigorous and comprehensive study of DCSP's search

control heuristics is more desirable but unfortunately out of the scope of this thesis. We believe that such a study would constitute a main direction for future work of this thesis. In particular, it would be worth addressing the following issues.

1. We barely touched the issues of estimating admissible heuristics for a DCSP. Not coincidentally, there is no known effective admissible heuristics for POP up to this writing, either. We would like to have a family of admissible value ordering heuristics for DCSP and partial order planning, whose effectiveness can be traded by the heuristic computation efficiency. The much better understood theory of admissible heuristics for state space planning [39, 20] would probably provide some guidance in this interesting research problem.

2. While the connection between the heuristic estimators developed in Section 4.6 and that of relax heuristics for ranking partial plans (Section 3.3) is clear, much details are missing on how the termination of the *repeat* loop in procedure 4.1 and how certain constraints and variables in the DCSP get relaxed are less than obvious. An investigation into the structure of the DCSP to solve this issue would be worthwhile.

3. A more rigorous development of variable ordering heuristics is clearly needed. It is expected that many insights from the plethora of CSP's variable ordering heuristics that deal with only compatible constraints can be generalized to handle activity constraints as well. (Our notions of fertile/infertile variables would probably continue to be useful).

4. The objective function 4.1 is general enough to capture most types of resource-based constraints and action costs in a planning problem, but it is not natural enough to capture the time (makespan) in planning. *Temporal CSPs*[10] have been devised to deal with time, and we can imagine a framework such as "temporal DCSP" that would be useful for dealing with temporal constraints.

5. More empirical analysis is needed. In addition, good heuristics should also be tested in conjunction with other intelligent backtracking techniques, and constraint propagations and so

on.

Chapter 5

Related Work

5.1 Heuristics for State-space Planning

The work on state-space planning heuristics has strong connections to that in the AI search literature [51]. For example, the "sum" heuristics used in UNPOP [48] and HSP/HSP-R [6, 5] are connected to the Manhattan heuristic in the simple 8-puzzle problem well-known in the search literature. Similarly to the sum heuristic, the Manhattan heuristic also makes the assumption that the cost of achieving a set of subgoals is equal to the sum of the cost of achieving each individual subgoal. Several important developments in the state-space heuristic such as the idea of pattern dababases [8] and linear-conflict [19] also bear some traits of similarity to our family of *set-level* and *adjusted-sum* heuristics, but see [39] for a more thorough analysis.

The set of mutex constraints plays very important role in improving the informedness of our graph-based heuristics. The *level-specific* mutexes can be used to give finer (longer) distance estimates, while *static* mutexes help prune more invalid and/or unreachable states. Thus, our heuristics can be improved by detecting more mutexes. Indeed, more level-specific mutexes can be discovered through more sophisticated mutex propagation rules [11] , while binary and/or higher order static mutexes can be discovered using a variety of different techniques[18, 53, 15].

Several researchers [21, 52] have considered the *positive* interactions while ignoring the negative interactions among subgoals to improve the heuristics in many problem domains. Hoffman [21] uses the length of the first relaxed plan found in a relaxed planning graph (without mutex computation) as the heuristic value. Refanidis [52] essentially extracts the co-achieveness relation among subgoals from the first relaxed plan to account for the positive interactions. These heuristics were reported to provide both significant speedups and improved solution quality.

Concomitant with our work, Haslum & Geffner [20] considered computing admissible state space heuristic based on dynamic programming approach. Interestingly, their most effective *max-pair* heuristic is closely related to our admissible set-level heuristic. Specifically, the heuristic value updating rule in max-pair heuristic has an effect similar to that of the mutex propagation procedure in the planning graph.

Finally, we concentrated on using the heuristics extracted from the planning graph to drive a state search procedure. In contrast, [31] considers the possibility of using such heuristics to drive Graphplan's own backward search. Their results show that some of the same ideas can be used to derive effective variable and value ordering strategies for Graphplan.

5.2 Heuristics for Partial Order Planning

Several previous research efforts have been aimed at accelerating partial order planners (c.f. [23, 24, 17, 26, 56, 57, 16]). While none of these techniques approach the current level of performance offered by RePOP, many important ideas separately introduced in these previous efforts are either related to or are complementary to our techniques.

IxTeT [16] was the first to use distance based heuristic estimates to select among the possible resolutions of a given open condition flaw. Unfortunately, there was no systematic evaluation of the effectiveness of the IxTeT heuristic. It is interesting to note that IxTeT's use of distance based heuristics precedes their independent re-discovery in the context of state-search planners by McDermott [48] and Bonet and Geffner [5]. The main difference our IxTeT's heuristic and ours

is the way the distance-based function is estimated. The first difference is that the IxTeT heuristic makes the subgoal independence assumption, which in effect relaxs more constraints than our heuristic estimate. The second difference is the way the variable and value are chosen in the heuristic estimation. Both heuristics pick open conditions (variables) and then actions (values) to support the open conditions until the initial state is reached. However, we use a planning graph to improve the reachability of the value selection in the heuristic computation, and thus result in arguably more accurate estimate.

The idea of postponing the resolution of unsafe links by posting disjunctive constraints has been pursued by Smith and Peot in [56] as well as by Kambhampati and Yang in [26]. Our work shows that the effectiveness of this idea is enhanced significantly by generalizing the notion of conflicts to include indirect conflicts. The notion of action-proposition mutexes defined in Smith and Weld's work on temporal graphplan [58] is related to our notion of indirect conflicts introduced in Section 3.4.

Finally, there is a significant amount of work on flaw selection strategies (e.g., the order in which open condition flaws are selected to be resolved) [23] that may be fruitfully combined with RePOP. In particular, Joslin and Pollack [23] explicitly considers the issues of flaw selection and flaw resolution as variable and value ordering issues in a CSP. However they completely overlooked the state-space nature of the POP algorithm and the use of distance-based heuristics prevalent in state-space search.

5.3 Dynamic Constraint Satisfaction Problem

Dynamic constraint satisfaction problems was first formulated by Mittal and Falkenhainer [36]. In their paper, the authors consider several variable and value ordering heuristics that are adapted from CSP heuristics in a straightforward manner. For example, the value was ordered simply by the number variables being activated by that choice.

The state-space view of DCSP argued in this paper is closely related to a model called

"constrained heuristic search", which was independently proposed one year earlier. by Fox *et al* [14]. In this paper the authors proposed a number of heuritics that can also be seen as an adaptation from the CSP literature based on the variable tightness, constraint reliance and constraint tightness. Our work departs from both work in emphasizing both the CSP and state-space nature inherent in the DCSP problem. This is done by exploring different roles of compatibility and activity constraints in the DCSP. Other than these two early work, we are not aware of any significant advance in the area of search heuristic. This is perhaps attributed to the relatively lack of research areas where DCSP models can be applied at the time it was introduced.

As mentioned, the connection of search control in partial order planning algorithm to that of CSP was explored by several researchers such as Joslin and Pollack [23], who also adapted CSP style search control heuristics to their DESCARTES planning system. While these ideas were shown to improve the efficiency of their planner significantly, the level of performance by DESCARTES was no where near that of Graphplan [4] which was about to arrive at the scene. Kambhampati was one of the first researchers who remarked the deep connection between DCSP and both POP algorithm and Graphplan's [25, 27]. As the enthusiasm of the field for POP were switched toward the Graphplan algorithm, much research efforts were focused on improving search heuristics for Graphplan search.

Interestingly, the connection of Graphplan's search to DCSP is often construed to its connection to CSP. As a result, most of the heuristics that were considered for Graphplan were simple adaptation of CSP style heuristics [27]. In our more recent work [39], we pointed out the ineffectiveness of these CSP style heuristics, and introduced distance-based heuristic for Graphplan search that is empirically proven to be much more effective.

A different approach to solving DCSP that was not considered in this paper is by solving a CSP instance that was translated from the DCSP by putting a bound on the size of the set of activated variables [36]. In the context of plan synthesis, this approach was pursuied in GP-CSP [12] and to a large extent SATPLAN[32] planners. The attractiveness of this approach is that it can effectively exploit search techniques from the CSP literature without worrying about the synthesis

nature of DCSP. However, these planners tend to run into trouble when the size of the solution is large that the planners are unable to estimate in advance. In addition, the failure to discern the state-space nature from the CSP nature in the underlying DCSP problem implies that the CSP-style heuristics are not always the most effective.

Chapter 6

Conclusion

The main issue that we studied in this thesis is heuristic search control for state-space *and* partial order planning algorithms. The close relation between partial order planning and both CSP and state-space search theory motivated us to study heuristic search control for dynamic constraint satisfaction, with a hope that a good understanding of this model may provide valuable insights into the heuristics for partial order planning, and the possible generalization of these heuristics to the more complex problems.

Our work on search control heuristic estimators for state-space planning algorithms contributed to increased understanding of heuristics for state-space planning. In particular, we showed that the effectiveness of the heuristics can be significantly by taking into account the complex interactions (in the forms of subgoal positive and negative interactions) inherent the problem structure. We showed that the planning graph, for all its causal structures and mutual exclusion constraints, provides a rich source for systematically accounting for such subgoal interactions.

We presented a family of highly effective heuristics extracted planning graphs. Our empirical results showed that a state-space planning algorithm armed with these heuristics is very competitive with some of the fastest planners in the literature. We were also one of the first to focus on finding effective *and* admissible heuristics for state-space planners. We have shown that the *set-*

level heuristic working on the normal planning graph, or a planning graph adorned with a limited number of higher-order mutexes is able to provide quite reasonable speedups while guaranteeing admissibility.

Our work on search control heuristics for a partial order planning algorithm helps bring POP back to the realm of effective planning algorithms. The techniques that we introduced include a novel heuristic estimator for ranking partial plans, using disjunctive representation for ordering constraints, and exploiting reachability analysis for early detection of inconsistent partial plans. The former technique was shown to draw from the connection of POP algorithm with that of state-space search, and the latter techniques draw from heuristic control for CSP. We have implemented these ideas in a variant of UCPOP [62], which we call RePOP. Our empirical studies show that in addition to dominating UCPOP, RePOP also convincingly outperforms Graphplan in parallel domains, and the plans generated by RePOP have more execution flexibility and very good solution quality.

In seeking to have a better understanding and useful generalization of the techniques that we had developed, we focused on the connection between partial order planning algorithm and dynamic contrainst satisfaction Problems. We showed that DCSP is an excellent computational model studying partial order planning as well as other planning algorithms by presenting simple DCSP formulation of these algorithms. We also pointed out that a POP algorithm that handles more complex type of constraints related to time and resource can also be integrated naturally into the DCSP framework.

Several general heuristics for value and variable ordering for DCSP were introduced in this thesis. Value ordering heuristics were tied to the ranking of the DCSP's underlying states using a distance based function. We introduced a general way for approximating such function through partial relaxation of certain contraints. Variable ordering heuristics were tied to a useful characterization of DCSP variables as either *fertile* or *infertile*. We presented a number of different heuristics for dealing with each type of variables.

Finally, we drawed interesting parallels between the heuristic search controls for our planner

REPOP and that of general DCSP. For example, the relax heuristic for ranking partial plans is closely connected to the value ordering heuristics developed for DCSP. In addition, seeming disparate techniques such as the handling of unsafe link through disjunctive ordering constraints and the use LIFO scheme for subgoal selection become different aspects of a single variable ordering heuristic in the DCSP. We also discussed the short coming of the REPOP's heuristics and possible avenues for further improvements.

There remain several limitations in this work that need to be addressed in the future. To summarize, devising admissible heuristics for POP remains a challenge. And so is devising effective heuristics for POP algorithms with time and resources. We believe that such challenging tasks may be benefited from a better understanding of search heuristics for general DCSP, of whom the treatment in this thesis remains preliminary rather than comprehensive, intuition-based rather than theoretically rigorous. Combining different types of value and variable ordering heuristics in DCSPs with respect to the problem differences will continue to be an important area for future research.

References

[1] F. Bacchus. Results of the AIPS 2000 Planning Competition. *URL: http://www.cs.toronto.edu/aips-2000*, April 2000.

[2] F. Bacchus and P. van Run. Dynamic variable ordering in CSPs. In *Proc. Principles and Practice of Constraint Programming (CP-95)*, 1995. Published as Lecture Notes in Artificial Intelligence, No. 976. Springer Verlag.

[3] C. Backstrom. Computational aspects of reordering plans. *JAIR*. Vol. 9. pp. 99-137.

[4] A. Blum and M.L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*. 90(1-2). 1997.

[5] B. Bonet and H. Geffner. Planning as heuristic search: New results. In *Proc. ECP-99*, 1999.

[6] B. Bonet, G. Loerincs, and H. Geffner. A robust and fast action selection mechanism for planning. In *Proc. AAAI-97*, 1997.

[7] B. Bonet and H. Geffner. HSP planner. In *AIPS-98* planning competition, 1998.

[8] J. Culberson and J. Schaeffer. Pattern Databases. *Computational Intelligence*, Vol. 14, No. 4, 1998.

[9] R. Fikes and N. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 1:27–120, 1971.

[10] R. Dechter, I. Meiri and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, Vol. 49, 1991, pp. 61-95.

R. Fikes and N. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 1:27–120, 1971.

[11] M. Do, S. Kambhampati and B. Srivastava. Investigating the effect of relevance and reachability constraints on SAT encodings of planning. In *AIPS-2000*, 2000.

[12] M. Do & S. Kambhampati. Solving Planning Graph by Compiling it into a CSP. In *Proc. AIPS-2000*, 2000.

[13] D. Frost and R. Dechter. In search of the best constraint satisfactions earch. In *Proc. AAAI-94*, 1994.

[14] M.S. Fox, N. Sadeh and C. Baykan. Constrained heuristic search. In *Proc. IJCAI-89*.

[15] M. Fox and D. Long. Automatic inference of state invariants in TIM. *JAIR*. Vol. 9. 1998.

[16] M. Ghallab and H. Laruelle. Representation and control in IxTeT. In *Proc. AIPS-94*, 1994.

[17] A. Gerevini and L. Schubert. Accelerating partial-order planners: Some techniques for effective search control and pruning. *JAIR*, 5:95-137, 1996.

[18] A. Gerevini and L. Schubert. Inferring state constraints for domain-independent planning. In *Proc. AAAI-98*, 1998.

[19] O. Hansson, A. Mayer, and M. Yung. Criticizing solutions to relaxed models yields powerful admissible heuristics. *Information Sciences*, Vol. 63, No. 3, 1992.

[20] P. Haslum and H. Geffner. Admissible Heuristics for Optimal Planning. In *Proc. AIPS-2000*, 2000.

[21] J. Hoffman. A Heuristic for Domain Independent Planning and its Use in an Enforced Hill-climbing Algorithm. Technical Report No. 133, Albert Ludwigs University.

[22] A. Johnson, P. Morris, N. Muscettola and K. Rajan. Planning in Interplanetary Space: Theory and Practice. In *Proc. AIPS-2000.*

[23] D. Joslin and M. Pollack. Least-cost flaw repair: A plan refinement strategy for partial-order planning. In *Proc. AAAI-94.*

[24] D. Joslin, M. Pollack. Passive and active decision postponement in plan generation. Proc. 3rd European Conf. on Planning. 1995.

[25] S. Kambhampati, C. Knoblock and Q. Yang. Planning as Refinement Search: A unified framework for evaluating design tradeoffs in partial-order planning. In *Artificial Intelligence*, 1995.

[26] S. Kambhampati and X. Yang. On the role of Disjunctive representations and Constraint Propagation in Refinement Planning In *Proc. KR-96.*

[27] S. Kambhampati. Planning Graph as a (dynamic) CSP: Exploiting EBL, DDB and other CSP search techniques in Graphplan. *Journal of Artificial Intelligence Research*, 12:1–34, 2000.

[28] S. Kambhampati, E. Lambrecht, and E. Parker. Understanding and extending graphplan. In *Proc. ECP-97*, 1997.

[29] S. Kambhampati. Challenges in bridging plan synthesis paradigms. In *Proc. IJCAI-97*, 1997.

[30] S. Kambhampati. EBL & DDB for Graphplan. *Proc. IJCAI-99.* 1999.

[31] S. Kambhampati and R.S Nigenda. Distance based goal ordering heuristics for Graphplan. In *AIPS-2000*, 2000.

[32] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In *Proc. AAAI-96*, 1996.

[33] H. Kautz and B. Selman. Blackbox: Unifying sat-based and graph-based planning. In *Proc. IJCAI-99*, 1999.

[34] J. Koehler. Solving complex planning tasks through extraction of subproblems. In *Proc. 4th AIPS*, 1998.

[35] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *Proc. AAAI-91.*

[36] S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction problems. In *Proc. AAAI-90*, 1990.

[37] R. Sanchez Nigenda, X. Nguyen and S. Kambhampati. AltAlt: Combining advantages of Graphplan and Heuristic State Search. In *Proc. International Conference on Knowledge-based Computer Systems (KBCS-00)*, Bombay, India, December 2000.

[38] X. Nguyen and S. Kambhampati. Extracting effective and admissible state-space heuristics from the planning graph. In *Proc. AAAI-2000*, 2000.

[39] X. Nguyen, S. Kambhampati and R. Sanchez. Planning Graph as the Basis for Deriving Heuristics for Plan Synthesis by State Space and CSP Search. To appear in *Artificial Intelligence*, 2001.

[40] X. Nguyen and S. Kambhampati. Reviving Partial Order Planning. In *Proc. IJCAI-01*, 2001.

[41] R. Korf and L. Taylor. Finding optimal solutions to the twenty-four puzzle. In *Proc. AAAI-96*, 1996.

[42] R. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. In *Proc. AAAI-97*, 1997.

[43] R. Korf. Linear-space best-first search. *Artificial Intelligence*, 62:41-78, 1993.

[44] R. Korf. Recent progress in in the design and analysis of admissible heuristic functions (Invited Talk). *Proc. AAAI-2000*, 2000.

[45] D. Long and M. Fox. Efficient implementation of the plan graph in STAN. *JAIR*, 10(1-2) 1999.

[46] D. McDermott. A heuristic estimator for means-ends analysis in planning. In *Proc. AIPS-96*, 1996.

[47] D. McDermott. Aips-98 planning competition results. 1998.

[48] D. McDermott. Using regression graphs to control search in planning. *Artificial Intelligence*, 109(1-2):111–160, 1999.

[49] B. Nebel, Y. Dimopoulos and J. Koehler. Ignoring irrelevant facts and operators in plan generation. *Proc. ECP-97.*

[50] N. Nilsson. *Principles of Artificial Intelligence.* Tioga, 1980.

[51] J. Pearl. *Heuristics.* Morgan Kaufmann, 1984.

[52] I. Refanidis and I. Vlahavas. GRT: A domain independent heuristic for strips worlds based on greedy regression tables. In *Proc. ECP-99*, 1999.

[53] J. Rintanen. An iterative algorithm for synthesizing invariants. In *Proc. AAAI-2000*, 2000.

[54] D. Smith. Private Correspondence, August 1999.

[55] D. Smith, J. Frank and A. Jonsson. Bridging the gap between planning and scheduling. In *Knowledge Engineering Review*, 15(1):47-83. 2000.

[56] M. Peot and D. Smith. Threat-removal strategies for partial-order planning. In *Proc. AAAI-93*.

[57] D. Smith and M. Peot. Suspending Recursion Causal-link Planning. In *Proc. AIPS-96*, 1996.

[58] D. Smith and D. Weld. Temporal planning with mutual exclusion reasoning. In *Proc. IJCAI-99*, 1999.

[59] P. Stone, M. Veloso, and J. Blythe. The needs for different domain-independent heuristics. In *AIPS-1994*, 1994.

[60] B. Srivastava, S. Kambhampati and B. Do. Planning the Project Management Way: Efficient Planning by Effective Integration of Causal and Resource Reasoning in RealPlan. ASU CSE Technical Report, 2000.

[61] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, San Diego, California, 1993.

[62] D. Weld. An Introduction to Least Commitment Planning. In *AI Magazine*, 1994.

[63] D. Weld, C. Anderson, and D. Smith. Extending graphplan to handle uncertainty & sensing actions. In *Proc. AAAI-98*, 1998.

[64] D. Weld. Recent advances in AI planning. *AI magazine*, 1999.

APPENDIX A

A.1 DCSP formulation of POP with resource constraints

Consider the planning problem $\mathcal{P} = (I, \Omega, G, R, U)$. I, Ω, G are the same as in classical planning problem. R specifies a list of resources r. For each $r \in R$, denote $c(r)$ as the capacity of resource r. Function U specifies the usages of resources and of the following form: $U : R \times \Omega \rightarrow \Re$, where $U(r, a)$ specifies the amount of resource r used by action a. In addition, one may want to also specify the cost of resources by a cost function $cost : R \times \Re \rightarrow \Re$, wherein $cost(r, x)$ is the cost for using an amount x of resource r.

In addition to the set of constraints specified in Section 4.3, we have to specify constraints related to resources. The main constraint for this type of (sharable-reusable) resource says that no usage can exceed the resource capacity, and can be specified as follows:

- Additional Resource Constraints: For each $r \in R$

  For every *minimal* subset of different action instances $a_{l_1}, a_{l_2}, .., a_{l_n} \in \Omega$ such that $\sum_{a_i} U(a_i, r) > c(r)$:

  $s_{i_1} = a_{l_1} \wedge ... \wedge s_{i_n} = a_{l_n} \Rightarrow \bigvee_{1 \leq k < m \leq n} (s_{i_k} \prec s_{i_m}) \vee (s_{i_m} \prec s_{i_k})$ for any $1 \leq i_1 < i_2 < ... \leq M$.

- Modified Objective function: The introduction of resources may nececitate a new objection function based on the resource usage. The objective function is still of the form 4.1, whereas $f(.)$ is modified such that $f([s, a]) = \sum_{r \in R} cost(r, U(r, a))$, and $f = 0$ otherwise.

A.2 DCSP formulation of POP with temporal constraints

The specification of a temporal planning problem $\mathcal{P}$ has several extension to accommodate temporal constraints. For each action $a \in \Omega$, denote $dur(a)$ as the duration of a. I is the initial state of the world (at time point 0). G is a set of goals g. Different types of temporal goal may be desirable: *Deadline* goal requires different goals g to be true by different time points t; *Maintainance* goal requires g to be true during a specific temporal interval $[t_1, t_2]$, and so on. In this work we will specifically consider the objective of minimizing the makespan of the solution plan. While the encoding is very simple, it requires extending the DCSP model to be able to handle numerical domains.

- Additional variables are et_i for each $i = -1, 0, 1, ...$ for end time of each action s_i.
- For each variable et_i, $0 < et_i < T_{max}$ where T_{max} is the maximum makespan allowed for the solution plan.
- Additional (temporal) constraints

$$(s_i \prec s_j) \wedge (s_j = a_l) \Rightarrow et_j - et_j \geq dur(a_l), (\forall i \neq j, l \neq 0) \tag{A.1}$$

$$et_{-1} = 0. \tag{A.2}$$

$$(s_i = a_l) \Rightarrow et_0 - et_i \leq dur(a_l)(\forall i > 0) \tag{A.3}$$

- Modified objective function: $F(A) = et_0$.

A.3 Forward state-space search planning as DCSP

- Variables:

  Set of steps $s_1, s_2, s_3, ...$ Domain for each variable is Ω.

  $p_k(k = 0, 1, 2, 3, ...)$ for each $p \in P$. Domain for each variable is $\{true, false\}$.

- Initial state:

  $p_0 = true$ for all $p \in I$, and $false$ otherwise.

  $active : s_0$.

- Activity Constraints: For $k = 0, 1, 2, ...$

$$\bigvee_{p \in P} (p_k = true \wedge \neg p \in G) \vee (p_k = false \wedge p \in G) \Rightarrow active : s_k \quad (A.4)$$

$$s_k = a_l \Rightarrow \bigwedge_{p \in P} active : p_{k+1} \wedge \bigwedge_{p \in Prec(a_l)} p_k = true \wedge \bigwedge_{\neg p \in Prec(a_l)} p_k = false$$
$$\wedge \bigwedge_{p \in Eff(a_l)} p_{k+1} = true \wedge \bigwedge_{\neg p \in Eff(a_l)} p_{k+1} = false \wedge \bigwedge_{p_k = true \wedge \neg p \notin Eff(a_l)} p_{k+1} = true$$
$$\wedge \bigwedge_{p_k = false \wedge p \notin Eff(a_l)} p_{k+1} = false. \quad (A.5)$$

- Objective function: Same as in form 4.1, where $f([s_k, a_l]) = 1$ and $f(.) = 0$ for the rest.

In this formulation, each state of the DCSP is the set of assignments for $\{p_k | p \in P\} \cup \{s_i | 0 \leq i \leq k-1\} \cup \{active : s_k\}$ for a given k.

A.4 Backward State search planning as DCSP

- Variables:

  Set of steps $s_1, s_2, s_3, ...$ Domain for each variable is Ω.

  $p_k (k = 0, 1, 2, 3, ...)$ for each $p \in P$. Domain for each variable is $\{true, false\}$.

- Initial state:

  $p_0 = true$ for all $p \in G$

  $p_0 = false$ for all $\neg p \in G$. $active : s_0$.

- Activity Constraints: For $k = 0, 1, 2, ...$ (for indexing the steps backward)

$$\bigvee_{p \in P} (p_k = true \wedge \neg p \in I) \vee (p_k = false \wedge p \in I)$$
$$\Rightarrow active : s_k \quad (A.6)$$

$$s_k = a_l \Rightarrow \bigwedge_{p \in Eff(a_l)} p_k = true \wedge \bigwedge_{\neg p \in Eff(a_l)} p_k = false \wedge$$
$$\bigwedge_{p \in Prec(a_l)} (active : p_{k+1} \wedge p_{k+1} = true) \wedge \bigwedge_{\neg p \in Prec(a_l)} (active : p_{k+1} \wedge p_{k+1} = false) \wedge$$
$$\bigwedge_{p_k = true \wedge \neg p \notin Eff(a_l)} (active : p_{k+1} \wedge p_{k+1} = true) \wedge$$
$$\bigwedge_{p_k = false \wedge p \notin Eff(a_l)} (active : p_{k+1} \wedge p_{k+1} = false) \quad \text{(A.7)}$$

- Objective function: Same as in form 4.1, where $f([s_k, a_l]) = 1$ and $f(.) = 0$ for the rest.

In this formulation, each state of the DCSP is the set of assignments for $\{p_k | p \in P\} \cup \{s_i | 0 \leq i \leq k-1\} \cup \{active : s_k\}$ for a given k.

Related Publication

[1] X. Nguyen and S. Kambhampati. Extracting effective and admissible state-space heuristics from the planning graph. In *Proc. National Conference on Artificial Intelligence (AAAI-00)*, 2000.

[2] R. Sanchez Nigenda, X. Nguyen and S. Kambhampati. AltAlt: Combining advantages of Graphplan and Heuristic State Search. In *Proc. International Conference on Knowledge-based Computer Systems (KBCS-00)*, Bombay, India, December 2000.

[3] X. Nguyen, S. Kambhampati and R. Sanchez. Planning Graph as the Basis for Deriving Heuristics for Plan Synthesis by State Space and CSP Search. ASU CSE Technical Report, 2001. To appear in *Artificial Intelligence*.

[4] X. Nguyen and S. Kambhampati. Reviving Partial Order Planning. In *Proc. International Joint Conference on Artificial Intelligence (IJCAI-01)*, 2001.