Integrating landmarks in partial order planners

Master's Thesis

Bram Ridder

Integrating landmarks in partial order planners

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Bram Ridder
born in Hoorn, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

University of Strathclyde
Department of Computer and Information Sciences
Glasgow, United Kingdom
www.cis.strath.ac.uk

Integrating landmarks in partial order planners

Author: Bram Ridder
Student id: 1276263
Email: bram.ridder@cis.strath.ac.uk

Abstract

In this document we will continue a line of research which focusses on reviving partial order planning. We will look at the latest pair of partial order planners, RePOP[31] and VHPOP[43] which both use techniques developed for state-space planners in an attempt to make partial order planning competitive with state of the art state-space planners. We focus on recent advances in utilizing landmarks[35] as exemplified by LAMA[37]. We inquire two lines of research, one is to integrate landmarks within the heuristic as done by LAMA and the other is to utilize landmarks to split a planning problem into a set of subproblems. We will detail on additional techniques derived and used and present novel flaw selection strategy. Our aim is to revive partial order planning by taking VHPOP as our base planner and incorporate techniques from FF[21], Fast-Downward[17], STeLLa[39], LAMA[37], and the integration of landmarks in FF[22]. We use the planning problems presented at the 3rd international planning competition[26] and compare the results of our approaches to the original VHPOP.

Thesis Committee:

Chair:	Prof. Dr. C. Witteveen Faculty EWI, TU Delft
University supervisor:	Dr. M.M. de Weert, TU Delft
Company supervisor:	Prof. Dr. D. Long, Department Computer and Information sciences, University of Strathclyde, UK
Committee Member:	Dr. P. Kluit, TU Delft

Preface

First of all I would like to thank Cees Witteveen for all the advice and help and allowing me to go to the University of Strathclyde to finish my master. I would like to thanks both of my supervisors Maria Fox and Derek Long for their feedback and discussions. I would also like to thank Haakan Younes for creating VHPOP (and his excellent coding style) and giving me valuable feedback. Amanda and Andrew Coles to give feedback and useful tips throughout the project and Alex Coddington for suggestion useful ideas.

Bram Ridder
Glasgow, United Kingdom
January 7, 2010

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the course of the last decades, AI and more specifically domain independent planning has come a long way. A broad range of planners have been developed, as have languages. Applications for planning have steadily increased both due to more complex systems that have been developed requiring more reasoning to be controlled and the advances in the field of planning itself. Previously planning was confined to a very strict subset of problems which could only describe a finite set of state transitions and made assumptions which made it only applicable to a very small range of systems, if any. Over the years the assumptions have been lifted one by one which allowed planning to be applied to a broader range of problems and integration in real-life systems was finally possible (STMP[42], O-Plan[10]). Successes have been documented with domain specific planners which enjoy manually written rules and heuristic[1][29]. In this work we will solely be looking at domain independent planners which can, in principle, be applied to any kind of problem provided that the problem can be formalized in the given modeling language. In this work we will use domain files which have been written in the language PDDL[16] and we assume the reader is familiar with this language.

The extensions and evolution of the languages used in planning have come a long way as well, from STRIPS[12], to ADL[32], and finally into PDDL and SAS^+[18]. Domains can now be handled which require temporal reasoning as well as handling numerical fluents. Applications planning can and is applied to include autonomous vehicles[25], voltage power stations[2], etc[28]. In situations where domain independent planners are used, both in applications and in planning competitions, we see a break from the trend of only a decade ago, then research was mainly focussed on hierarchal planners and partial order planners. Nowadays the planning landscape is dominated by forward state-space planners and rapid advances are made in this area while other planning approaches do not enjoy as much attention as they used to. Although there are some good reasons for this shift in focus, there are some merits in plan-space planning which are not shared by forward state-space planners.

In this work we continue a line of work which began with the revival of partial order planning by a planner called RePOP[31] published in 2001, which challenged the pessimism about the performance of partial order planners by integrating state-of-the-art state-space planning techniques into partial order planners and showed dramatic improvements. More recent work saw VHPOP[43] entering the IPC-3[26] planning competition, again making

use of state-of-the-art improvements in state-space planning, and which was able to tackle temporal domains which it won the best newcomer prize. In the last few years some new and interesting developments have been made in state-space planning. For example, the Causal Graph heuristic[17], landmark integration[22, 36] and many other techniques have been developed which have made considerable impact on performance. To continue the line of work in partial order planning we set out to evaluate the latest advances in planning and incorporate and improve upon these techniques in a partial order planning context.

In this work we focus on integrating landmarks into the latest version of VHPOP and utilize this information both to improve the existing heuristics (much like LAMA[37] did) and to split the planning problem into smaller subproblems that are (we hope) easier to solve than the original problem (as has been previously been explored in STeLLa[39]). The novelty in both approaches is the application to partial order planners instead of state-space planners and in using a different landmark generation process that also deals with disjunctive landmarks. In addition to these main approaches we consider additional ways to exploit information by using domain analysis to create a novel new flaw selection strategy as well as other policies which speed up the search process. We expect that these approaches will show a decrease in the search space that needs to be explored by VHPOP, due to a more informed and guided search process, but we anticipate a decrease in the quality of the plans (in term of plan length) due to the greedy nature of our approaches.

Our hope is that this renewed effort will revive the interest in partial order planning and present a showcase for its merits. VHPOP will be used as the base planner to build upon and central to this work are the following state-space planners in no particular order: Fast-Forward[21], Fast-Downward[17], STeLLa[39] and LAMA[37]. In the first two sections we briefly introduce classical and partial order planning and give the formal models of both which is used throughout this document. Next we look at recent work done in partial order planning and then turn to recent advances in state-space planning which are relevant to this work. Once we have laid down the foundation and the background relevant to our planner, we introduce our planner and discuss how and which techniques from other planners have been integrated into VHPOP and discuss some novel contributions to partial order planning. Finally, we consider several configurations and test their effectiveness using the planning problems from the 3rd international planning competition[26] and compare it to the original VHPOP. We end our work with our conclusions and future work.

Chapter 2

Classical planning

One of the earliest fields of research within the planning community made some very strict assumptions regarding the problems they tried to tackle. A classical planning problem is fully observable, deterministic, finite, static (things only happen when an action is executed) and discrete (in time, action, objects, and effects) domains[30]. Thus we are dealing with restricted state-transition systems. Although this set of assumptions allows for systematic search algorithms to tackle the problem, a quick glance at typical problems tackled by the planning community shows that problems quickly become infeasible to solve. In fact, even with these limitations in place, determining whether there is a solution is PSPACE-complete[6].

People in the planning community have come up with several methods to solve larger than trivial planning problems. One of these can be found in the way languages are constructed to formalize planning problems. One of the earlier language to formalize classical planning problems is STRIPS[12] and in a hope to make planning algorithms simpler and more efficient various restrictions were imposed without making it too hard to describe problem domains. In STRIPS the representation of the world is decomposed into logical conditions and states are represented as a conjunction of positive first-order literals. A later extension to this language is ADL[32] and the defacto standard language in use now is PDDL[16].

2.1 Formal model

Before we go on to describe actual planners which are able to tackle classical planning problem we give a formal model of a classical planning problem.

We denote the whole set of all possible states in the world as S. A planning problem P is given as the triple $\langle s_0, s_g, O \rangle$, where

- $s_0 \in S$ is the initial state.
- $s_g \in S$ is the goal state.
- O is the set of operators.

We can navigate through the state-space of the world by applying actions. The set of all actions we can execute is formulated by the set of operators O. Every operator $o \in O$ is described by the triple $o = \langle name, prec, effects \rangle$, where

- name, the name of the operator, is a syntactic expression of the form $n(x_1, \ldots, x_k)$, where n is a symbol called an operator symbol, $x_1, \ldots, x_k$ are all of the variable symbols that appear anywhere in o, and n is unique (i.e., no two operators in O can have the same operator symbol).
- prec, the preconditions of o, this condition describes which set of literals must hold in any state $s \in S$ before it can be executed.
- effects, the effects of o, after the preconditions have been met and the action is executed this field describes how the state from which the action was executed will be affected. It contains an add-list of positive literals which will be added to the state from which the action is executed (literals which are already true are ignored) and a delete-list of negative literals which will be deleted from the state (literals which are not true in the current state are ignored).

In effect every operator can thus be instantiated into several actions based on the values passed to its parameters. An action which is fully instantiated is called a *grounded action*.

A solution to a planning problem is a sequence of actions $\langle a_1, a_2, \ldots, a_n \rangle$, corresponding to a sequence of state transitions $\langle s_0, s_1, \ldots, s_n \rangle$ such that $s_{n+1} = \gamma(s_n, a_n)$ and s_0 is the initial state and s_n is the goal state. $\gamma(s_x, a_x)$ denotes a transition from state s_x by applying action a_x to this state. An action can only be applied if all its preconditions are satisfied by the state we want to apply it to: $\forall_{p \in prec(a_x)} p \in s_x$. The new state after applying an action is defined as: $\gamma(s_x, a_x) = (s_x \setminus effects^-(a_x)) \cup effects^+(a_x)$, note that the delete effects are handled first before the add effects.

With the definition of the planning model complete, we are now able to define the domain of a classical planning domain. Given a planning problem P, its domain D is specified as the tuple: $\langle S, A, \gamma \rangle$:

- S is the set which contains all possible states. $|S| = 2^{all\ ground\ atoms\ of\ L}$.
- A is the set of all ground actions of O
- $\gamma(s,a) = (s \setminus effects^-(a)) \cup effects^+(a)$ iff $\forall_{p \in prec(a)} p \in s$

Do note that $\gamma(s,a)$ is closed under S, that is to say, the literals added to arrive at a new state are already part of the first-order language L.

2.2 Planning approaches

In order to tackle problems which can be described by the STRIPS language various solutions have been proposed, some more successful than others. All these planners can be categorized into two major families, state-space and plan-space planning. The former will be discussed in this section and the latter in the next section.

2.2.1 State-Space planning

The one most straightforward of the two is state-space planning which is the most common type of search algorithm used in research nowadays. States in STRIPS are defined as a conjunction of literals, so it is easy to work out which actions can be applied to any given state by looking at their preconditions. We define the neighbourhood of a given state $s \in S$ as $N = \bigcup_{a \in A} \gamma(s,a)$. This form of search is an iterative process in which we hold the neighbourhood set, initially instantiated as $N_0 = \{s_0\}$, select a state from it and expand its neighbourhood and add these into the neighbourhood set until the neighbourhood satisfies the goal state: $N_i = \{N_{i-1} \cup \bigcup_{a \in A}(\gamma(s,a)) \setminus s | s \in N_{i-1}\}$ This method is called *forward state-space search* (or forward-chaining).

Another way of doing state-space search is by starting at the goal state and working backwards until we find a state which satisfies the initial state. This method is called *backward state-space search* and operates very much the same, but instead of looking for actions which are applicable in a state we search for actions which achieve the atoms in a given state and the resulting state is constructed by removing all effects and adding all preconditions. However, we must make sure that a given action does not undo any of the atoms in the goal state, if an action satisfies this restriction it is called consistent. So what we are looking for is a sequence of consistent actions which will bring us from the goal state to the initial state. Thus we start with the goal state, $N_0 = \{s_g\}$ and the neighbourhood is defined as $N = \{\bigcup_{a \in A} s \setminus \mathit{effects}(a) \cup \mathit{prec}(a) | \mathit{consistent}(a)\}$.

Both forward- and backward-chaining have individual strengths and weaknesses and their usability depend on the problem domain and the heuristics used. For instance with forward state-space search when we check every possible action from any given state it is clear that the branch factor can be enormous, for example when we are in the library and want to pick up a book there may be thousands of possible pick up actions we can execute for every possible book. A lot of actions can be executed which are irrelevant to the actual goal we try to achieve. On the other hand doing backward state-planning the branching factor will generally be lower since every action we select will be relevant to the goal in some way. But again the same problem arises as we now need to find a way to backtrack to the initial state. Consider buying a plane ticket to destination X, while we can assert that any plane ticket which will get us to X is relevant to the goal but if we have no clue how easy we can get to the departure location from the initial location it still is not helping a lot. From both examples we can conclude that merely selecting a search strategy will get us nowhere in most cases, if we want to do have any chance of finding a plan at all we will need some guidance to direct state-space planners.

It is interesting from a historical perspective that whilst forward state-space planning is now considered state-of-the-art, before it was assumed that this method was to inefficient to be practical[42]. In the next chapter we will discuss the most important heuristics which have made forward state-space planning efficient for problem solving.

Heuristics So far we have established that both forward- and back-chaining search algorithms alone are not enough to tackle interesting planning problems. In order to do that we need good heuristics, or more specifically, once we have found a neightbourhood for a

given state we want to be able to determine which of these gets us closest to the goal. In the context of classic planning we are interested in the state which requires the fewest number of actions to get to the goal state. Unfortunately, finding the exact number of actions which need to be added before a goal is reached is as hard as planning itself. Suppose that that one such an heuristic exists; If we simply choose the state from our neighbourhood with the lowest value we will solve the planning problem in polynomial time - assuming that the optimal plan is polynomial in length - which is impossible. However, some approaches have been found which gives us a reasonable estimate without requiring to much computation. However, the strength of each heuristic depends on the structure of the domain; There is no silver bullet when it comes to heuristics, some work very well in some domains but are worthless in others, some planners like Fast-Downward[17] try to circumvent this issue by using several heuristics while others employ randomness in their search and allow restarts like LPG[14]. This section will describe the most important and most used heuristics and planners in classical planning.

The aim of heuristics is to provide guidance for the planner in an attempt to speed up the planning process. So it is no use to have an heuristic function which takes a long time to complete because we might be better of without an heuristic at all! So an heuristic estimate must be computed promptly yet still give good guidance. The informativeness of an heuristic tells us how 'informed' an heuristic is, this is an estimate of how much guidance we expect an heuristic to give us. Coupled with this is the notion of admissibility, we say that an heuristic is admissible if every estimate it gives from any given state to the goal is *less* or *equal* to the actual value. The latter is important if we want to do optimal planning because if we explore the best state - as defined by an admissible heuristic - in our neighbourhood first the first solution we find is guaranteed to be the optimal solution. It is very easy to come up with an admissible heuristic - for example assign 0 as an estimate for every state - the challenge is to come up with an heuristic which is both informed as well as admissible. We do note, however, that choosing a good heuristic will not be enough to tackle all problem domains. Not even if the heuristic is almost perfect, i.e. the heuristic only differs a constant value from the true heuristic[20]. Additional techniques which prune the search space are needed.

The traditional way of constructing a heuristics for problems is to relax the problem and try to solve the relaxed problem. In state-space planners there are 3 common heuristics: h_{max}, h_{add}, and h^k.

- h_{max} This heuristic assumes that if the planner solves the most difficult subgoal, that all other goals are automatically satisfied as well. This heuristic is admissible but not very informed.

- h_{add} This heuristic assumes complete goal independence, that is to say there will be no interference when trying to solve every subgoal independently. This heuristic is not admissible but better informed than h_{max}.

- h^k This heuristic is a generalization of h_{max}, instead of trying the solve the hardest subgoal we try to solve the hardest set of k subgoals. h_{max} is thus equal to h^1. If the number of subgoals in the problem is equal to k this heuristic will actually solve the

original problem. This heuristic is admissible and the higher k the better informed it is, however the computation cost increases polynomial as k increases[19].

However, trying to find values with the above heuristics in the original planning problem is equally difficult as planning in the first place. Consider any given planning problem with a given goal state s_g, next we introduce a new action which has as its preconditions all atoms in the goal state and produces a new literal 'Finished' and make this the new goal state for our problem. Given that there is only one goal atom to satisfy all above heuristics will be the same, but finding a solution for out new problem is at least as difficult as the original problem.

Relaxation To make the heuristics applicable in planning we must further relax the given problem. One of the reasons why planning is hard is because of conflicts between actions. For example, if we want to visit two people on a day we cannot visit both at the same time. Driving to the first person is in direct conflict with our goal to visit the other person. We can relax the planning problem further by removing these types of conflicts by ignoring the delete effects of an action. This means that by applying any given action, the resulting state will be a superset of the previous state. However, there is one little caveat we need to take care of, what if the goal state requires a negative effect? While this is not a problem with the STRIPS language (it only allows positive literals in states), extensions of this language do need to take care of this. The solution is simple, when we apply the heuristic we translate all negative effects and negative atoms in the goal state and operators to a new positive atom. We will now define how every heuristic works as described in[5].

h_{add} The estimated cost of achieving a set of goal atoms p from a given state s are obtained by solving the functional equation:

$$g(p;s) \stackrel{def}{=} \begin{cases} 0 & \text{if } p \in s \\ min_{a \in O(p)}[1 + g(prec(a);s)] & \text{otherwise} \end{cases} \tag{2.1}$$

for all atoms p by means of a Bellman-Ford type of algorithm. Where $g(prec(a);s)$ stands for the estimated cost of achieving the set $prec(a)$ from state s and $O(p)$ stands for all sets of operators which, combined, 'add' p. In this algorithm the measure $g(p;s)$ are updated as:

$$g(p;s) := \min_{a \in O(p)}[g(p;s), 1 + g(prec(a);s)] \tag{2.2}$$

starting with $g(p;s) = 0$ if $p \in s$ and $g(p;s) = \infty$ otherwise, until they do not change.

For the additive heuristic h_{add} the cost $g(C;s)$ of sets of atoms C is defined as the sum of the costs $g(p;s)$ of the individual atoms r in C. We denote such additive costs as g_{add}:

$$g_{add} \stackrel{def}{=} \sum_{r \in C} g(r;s) \tag{2.3}$$

The heuristic h_{add} is then defined as:

$$h_{add}(s) \stackrel{def}{=} g_{add}(G,s) \tag{2.4}$$

h_{max} Instead of taking the sum of all goal atoms we can define the h_{max} heuristic as:

$$g_{max}(C;s) \stackrel{def}{=} \max_{r \in C} g(r;s) \tag{2.5}$$

Thus $h_{max}(s) \stackrel{def}{=} g_{max}(G;s)$.

h^k The last heuristic to define is the h^k heuristic in which we search for the most expensive subset to satisfy. The size of the subset is defined by k. The cost $g^m(C;s)$ is characterized by the equation:

$$g^m(C;s) \stackrel{def}{=} \begin{cases} 0 & \text{if } C \subseteq s, \text{ else} \\ \min_{B \in R(C)}[1 + g^m(B;s)] & \text{if } |C| \leq m \\ \max_{D \subset C, |D|=m} g^m(D;s) & \text{otherwise} \end{cases} \tag{2.6}$$

where $B \in R(C)$ if B is the result of regressing the set of atoms C through some action a. Thus $h^k(s) \stackrel{def}{=} g^m(C;s)$.

Do note however that given a relaxed problem, finding an optimal plan is still NP-complete[19]. So we must resort to finding an estimate, after introducing partial order planners we will look at recent advances in planning and check which search strategies and heuristics they employ to solve problems.

Chapter 3

Partial order planners

The other family of planners takes a different approach to planning and can be argued to be more elegant. The previous discussed search algorithm only concerns itself with a strict total-ordered sequence of actions from the initial state to the goal state and was only able to reason about grounded actions directly applicable to the state under consideration. Plan-space planning takes a different approach that offers some advantages over state-space planning.

Instead of deliberating over single states as nodes we construct partial specified plans to reason about and instead of selecting actions to advance planning we use plan refinement operations to solve flaws in a partial plan. The reason why a node in a search space is called 'partial specified' is because we can reason about ungrounded (or lifted) actions. For example, if our goal is to get to destination X we might add an ungrounded action which will get us a plane ticket to location X but does not specify yet where to depart from. This is called the *least commitment principle* and gives a planner more control over the search strategy as we are no longer bound to specify all details up front, committing ourselves to grounded actions as in state-space planning and in effect greatly reduces the branching factor. Moreover, unlike state-space planning where we commit ourself to a total-ordered sequence, we do not need to commit ourselves to any specific ordering when choosing an action. This gives the planner the opportunity to work on several goal atoms independently before deciding which one is achieved first or resolve any kind of mutex relations due to the ordering. Despite these flexibilities there are some big challenges to overcome when using plan-space planning.

In the next section we give a formal definition of partial plans and discus how planning problems are generally solved in that context.

3.1 Formal model

A (partial) plan can be represented by a tuple $\langle A, L, O, B \rangle$, where A is a set of operators, L a set of causal links, O a set of ordering constraints defining a partial order on the set A, and B a set of binding constraints on the action parameters ($B = \emptyset$ if ground actions are used). Each action a is an instance of some operator A in the planning domain and a plan

can contain multiple instances of the same operator. A causal link, $a_i \xrightarrow{q} a_j$ represents a commitment by the planner that precondition q of action a_j is to be fulfilled by an effect of action a_i.

When given a planning problem, an initial partial plan is generated by creating two additional actions: a_0 which contains as effects all literals in the initial state s_0 and a_∞ which has as preconditions all literals of the goal state s_g. The partial plan is now generated by adding these two actions and by ordering a_0 before a_∞: $\langle\{a_0, a_\infty\}, \emptyset, \{a_0 \prec a_\infty\}, \emptyset\rangle$.

Instead of defining a neighbourhood of possible actions which can be executed given a certain state, a refinement planner works by adding elements to a plan in order to remove flaws in the plan. Fixing a flaw is called plan refinement. A flaw can either be an open condition $\xrightarrow{q} a_i$, which represents precondition q of an action a_i which is not yet supported by another action, or an unsafe link (or threat) $a_i \xrightarrow{q} a_j$, whose condition q can unify with the negation of an effect of an action a_k that could possibly be ordered between a_i and a_j. There are 3 different solutions to this problem: 1) Either a_k is ordered before a_i (demotion); 2) a_k is ordered after a_j (promotion); 3) or a binding constraint is introduced so that the effect of a_k cannot unify with q (separation). Once we find a plan without any flaws we have found a solution, if on the other hand we cannot refine any plan further and we have not found a solution yet the problem is unsolvable.

During the planning process a partial order planner keeps track of its plan-space in the set P. During every iteration a plan $p \in P$ is selected and then a flaw is selected to be resolved in p. All possible refinements resolving the flaw are returned and added to P, until either P is empty (denoting that no solution has been found) or a plan without flaws is found (a solution).

3.2 Plan selection strategies

During each iterations two important choices need to be made, first of all which partial plan to work on and secondly which flaw to select to resolve. In order to achieve good performance we must again rely on good heuristics, but whereas the search space in state-space is finite this is *not* the case in plan-space, this is due to the fact that we do not represent the states explicitly in plan-space planning. While this gives us a lot of freedom on how to plan we also lose information available to us in state-space planning. In this and next section we will discuss how this translates into search procedures and heuristics we can apply.

Like in state-space planning we want to base our decision how to advance constructing a plan based on the notion of distance to the goal. While in a state-space representation we had direct access to the state we are working on the state of a partial plan is ambiguous at best. Because we adhere to the principle of least commitment and because we do not need to select the actions of a plan in any particular order it is unclear how a partial plan can be translated into a state. This gives us problems when we try to estimate the number of actions needed to find a solution, the heuristics used in state-space planning cannot directly be used in partial order planning, so we need to find new methods to deal with this.

Lacking an explicit state representation, partial order planners have originally resorted to counting the number of flaws in a plan to estimate the work to be done before reaching

a solution. Because we are interested in a plan with as few actions as possible the heuristic function is:

$$f(s) = g(s) + h(s) \tag{3.1}$$

This equation can be used in a best-first search algorithm to select which plan to extend next. $g(s) = |A|$ (the number of actions) and $h(s)$ is the estimate of work still to be done. Possible heuristics include counting all flaws or only the number of open conditions. While this gives us some guidance in general these notions do not give a clear idea of far we are from a goal nor is this heuristic admissible, because multiple open flaws could be resolved by one single action. But even if there is only one open condition, the above heuristics do not take into consideration how much more work needs to be done to satisfy this open condition. This is the main reason why partial order planners have fallen behind the performance of state-space planners recently. However, in later chapters we will describe some new developments in partial order planners which are closely related to the heuristics for state-space planners.

3.3 Flaw selection strategies

After we have selected a plan from our plan-space based on an heuristic we need to identify the flaws in this plan and choose which one to resolve first. First of all we must identify that there are several types of flaws which can occur (see above) and that solving one type before another might enhance the search process. In the literature a lot of strategies have been proposed in which order flaws should be selected and fixed. Unfortunately so far no iron clad rule has been found which works best in every situation and it is not clear that one will ever be found. However, based on some benchmark problems some strategies are considered better than others[34]. In this section we introduce the basic notation.

Cost of repairing a flaw An inherent property of a flaw in a partial plan is that it will not go away unless we explicitly work on it. For example, in state-space planning we might add an action to satisfy a goal atom, but the same action could satisfy another goal atom as well. In partial order planning every causal link between an effect of an action and a precondition of goal atom is made explicitly by adding one. Of course we can, by adding an action, use the same action to solve multiple open conditions by adding causal links. This allows us to look at every single flaw independently and check how many options we have to solve this single flaw. As discussed before, all flaws need to be handled separately, the ordering in which these flaws are resolved can lead to a significant reduction to the search space.

To do flaw selection we use the notion of ‘most constrained variable’ which is extensively used in CSPs (constraints satisfaction problems). When we have to decide which variable to resolve first it is an good idea to assign a value to the most constrained variable, that is the variable with the smallest domain of values to choose from. Consider for example solving a SuDoKu puzzle, one way of solving this puzzle is:

1. Scan the puzzle row per row and select the first empty spot found and assign it a number which is unique in its row, column, and block.

2. Repeat (1) until we either find an empty spot which we cannot assign a number or solved the puzzle. In the former, clear the last assigned block and repeat (1), in the latter case we are done!

It is clear to see while this algorithm will solve any SoDoKu puzzle it is not the most efficient way to do so. If would be much faster to concentrate on the spots which have very few choices, the obvious example being one. If a spot only has one possible choice it must take that choice and by propagating this effect through the puzzle we might able to solve the puzzle without doing any search at all! A proper SuDoKu puzzle has only one possible solution to its problem so if we have a spot with 9 possible numbers 8 of these will be false. If we choose to make this variable the start of our search tree we could potentially end up searching 8 different branches while if we start with a variable with only two choices there is less chance to take a wrong branch and even if we do the other branch is guaranteed to succeed. Choosing the most constrained variable will also allow us to detect dead ends faster as we have less alternatives to try in the case of failure.

In the case of flaw selection, based on the flaw there are a number of possible resolutions to this flaw:

- Open conditions:
  An open goal g in plan P can be solved by the following resolutions:
  - The number of literals in the initial state which can unify with g;
  - The number of action effects of actions already in the partial plan which can unify with g;
  - The number of action effects of new actions that can unify with g.

- Threads:
  Threads can be distinguished in two cases:
  - Non-separable threads, in this case there is a causal link between two actions $a_i \xrightarrow{q} a_j$, whose condition q unifies with the negation of an effect of an action a_k that could possibly be ordered between a_i and a_j. In this case we cannot separate the negation and causal link. So our only option is to demote or promote a_k, giving us two options.
  - Separable threads, in this case we have a similar situation but this time we can separate the negation of the causal link by adding a binding constraint. However if we are dealing with a predicate of k variables we only need to make sure one of these is unequal to the causal link. Thus adding k choices to the possible resolutions.

Preference strategies Depending on the most constrained variable heuristic alone proves not to be sufficient to yield an effective planner. To improve upon this heuristic authors have looked a possible orderings in which flaws should be resolved given a partial plan[34]. Some of these strategies depend on the cost analysis outlined above, but others only consider the type and order in which the flaws were introduced to the partial plan. We will use the following notation to describe various strategies:

- o: open conditions
- n: non-separable threads
- s: separable threads
- t: static open condition
- l: local open condition
- u: unsafe open condition

The first three we have already detailed. A static open condition is a condition which occurs in the initial state of a problem but cannot be affected by any actions, i.e. it does not appear in the effects of any action. A local open condition is the subset of open conditions which is related to the most recently added action to the plan that still has open conditions. And lastly, an unsafe open condition flaw is an open condition what would be threatened if we would add a causal link.

The different order or tie-breaking strategies are:

- LIFO (last in first out)
- FIFO (first in first out)
- LC (lowest cost)
- R (random)
- New (achieves an open conditions by using a newly created action)

Given this notation we can now denote strategies as follows:

- $\{o\}LIFO$ - Always prefer open conditions, and always prefer the most recent added subgoals.
- $\{o\}LIFO/\{n\}LC/\{s\}R$ - Prefer open conditions in LIFO order, followed by non-separable threads in cost order, followed by separable threads in random order.
- $\{n\}_{0-1}/\{o\}LIFO/\{n,s\}R$ - Look for forced non separable threads, that is open conditions which either have only 1 solution or none (this would indicate a dead end), followed by open conditions in LIFO order, followed by random selections of either non-separable or separable threads.

When discussing actual planners in the next chapter we will detail which strategy they use.

Chapter 4

Previous work on partial order planners

Recent years have seen a renewed interest in partial order planning with planners like REPOP[31] and VHPOP[40] which are both based on UCPOP[33]. Research interest in partial order planners had previously waned after intense research up until the first half of the last decade. While partial order planning is an attractive planning framework it suffers from some serious problems which researchers up to this point have failed to solve.

On the positive site, partial order planners allow for very flexible planning and execution when compared to forward chaining state-space planners. Where the latter only allows, given a current state, to select a transition to move to the next state, the former allows to work on several aspects of a given plan by refining the plan by solving a flaw. One could even argue that this approach is actually more akin planning compared to state-space approaches. Another benefit is execution flexibility, once a partial order planner has found a solution it need not be a total ordered set of actions which need to be executed. Rather it is a partial plan which can be executed in several ways as the plan can be ordered in several ways.

Despite all its merits, partial ordered planning has been lacking behind state-space planners like FF[21], LPG[14], LAMA[37], which show consistently better results in planning benchmarks. The main reason why partial order planners have been falling behind is because of the lack of proper search guidance. Where state-space planners have an explicit state description to work from and to derive an heuristic from, it is not clear how such an explicit state can be derived from a partial plan. For example, if we take a look at the plan selection heuristic for UCPOP which is defined as: $|S| + |OC| + |UC|$ as the default, this algorithm is alike a A* heuristic as analyzed by Gerevini and Schubert[15], but not informative nor admissible.

More recent planners have improved both plan and flaw selection heuristics by adapting techniques from state-space planners and integrating them into the context of partial order planners. In this section we will look at two recent advances in partial planners, REPOP and VHPOP.

4.1 RePOP

One of the first attempts to revive partial order planning was RePOP[31]. In their paper, Nguyen and Kambhampati challenges, what they call, the prevailing pessimism about the scalability of partial order planning. They observed that the techniques developed for state-space planners could also be implemented for partial order planners and set out to see if the performance of partial order planners could be improved. In this section we will briefly describe the techniques used to gain better performance which allows RePOP to compete with the GraphPlan[3] planner.

4.1.1 Plan selection heuristics

To select which plan to refine first, RePOP takes a different approach than UCPOP to estimate the amount of actions still needed to find a plan. It does not rely on counting the number of flaws, but adopts a technique very similar to the h_{add} heuristic, although it does account for positive interactions between actions. To do this, RePOP makes use of a serial planning graph. It builds a planning graph from the initial state s_0. Let $lev(p)$ be the index of the level in the planning graph that a proposition p first appears, S_{OC} be the subset of all propositions which make up the open conditions, and $lev(S)$ be the index of the first level at which all propositions in S_{OC} appear. Let p_s be the proposition in S_{OC} such that $lev(p_s) = \max_{p_i \in S_{OC}} lev(p_i)$. It is assumed that p_s is possibly the last proposition in S_{OC} 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. Given this, RePOP defines its cost heuristic for the set of open conditions:

$$cost(S_{OC}) = cost(a_s) + cost(S \bigcup prec(a_s) \setminus effects(a_s)) \tag{4.1}$$

where $cost(a_s) = 1$ if $a_s \notin A$ and 0 otherwise. The final heuristic is: $f(s) = |A| + w * cost(S_{OC})$, where w is set to 5.

4.1.2 Additional improvements

Although RePOP does not introduce new flaw selection strategies, it does does conduct reachability analysis to prune unattainable partial plans and tries to postpone commitment to solving threats as long as possible by allowing for disjunctive ordering constraints. So instead of solving a threatened causal link $a_i \xrightarrow{p} a_j$, which is threatened by action a_k, by demoting or promoting it. RePOP adds a disjunctive ordering constraint $(a_k \prec a_i) \vee (a_j \prec a_k)$ to the plan. Next a constraint propagation rule is applied every time a new ordering constraint is added to the plan. Basically when we add an ordering constraint which matches one of the ordering constraints in a disjunctive binding we discard the other possible orderings in the disjunctive ordering constraint. For example, if an ordering $(a_k \prec a_i)$ is added to the orderings O and $(a_k \prec a_i) \vee (a_j \prec a_k) \in O$, we redefine O as: $O = O \bigcup (a_k \prec a_i) \setminus (a_k \prec a_i) \vee (a_j \prec a_k)$. What this technique hopes to achieve is when faced with an unsafe link, instead of choosing a definite solution directly (i.e. promote or demote) postpone this decision until we can not longer ignore it or we are forced to make an ordering which will solve

this outstanding flaw as well. Effectively it tries to reduces the search space as we do not need to branch directly when faced with this flaw.

The other technique concerns itself with mutal exclusive prepositions. While most partial order planners will only consider a causal link threatened when we are faced with an action which negates the specific proposition the link achieves. There are more threats which are not directly obvious. Sometimes it is possible that two proposition are mutal exclusive and cannot appear at the same time in the plan. An easy way to find such mutal exclusive propositions is by looking at the level-off point of the plan graph, any mutexes present at that level are state invariants.

4.2 VHPOP

The 3rd international planning competition[26] in 2002 saw a revival of a partial order planning by a planner called VHPOP[43]. The aim of the authors was to incorporate the advances which had been made in CPS-based planning algorithms and state-space planning as heuristics search into partial order planners.

VHPOP is a partial order planner which is loosely based on UCPOP[33] and incorporates quite a number of interesting plan and flaw selectiong strategies. But most important of all, while we claimed that one of the biggest challenges of partial order planners is to find good heuristics as it lacks a explicit state description VHPOP defines a variation of h_{add} which can be used in its search while still accounting for positive interactions. On top of that is uses a tie-breaking heuristic in case it gets stuck on a plateau.

As we have claimed earlier, there is no silver bullet when it comes to defining heuristics for domain independent planners. VHPOP takes the approach by running multiple planners concurrently with different flaw selection strategies.

4.2.1 Temporal reasoning

To handle temporal domains VHPOP uses an STN with a constraint-based interval approach[40], in other words it uses the STN to record the temporal constraints and during search queries this STN to check for consistencies of plan refinements. STN stands for Simple Temporal Network and is used to define temporal constraints between pairs of end and start points of actions. Given that we can use an STN to record the particular ordering of a partial plan, there is no real need for the set of partial orders O. To record the start and end points of a durative action, every such action is split up into two nodes t_{2i-1} (start time) and t_{2i} (end time). To allow for a compact representation the STN is represented by a d-graph[11]. The d-graph is a complete directed graph, where each edge $t_i \rightarrow t_j$ is labeled by the shortest temporal distance, d_{ij}, between the two time nodes t_i and t_j (i.e. $t_j - t_i \leq d_{ij}$). Internally such a graph is represented by a matrix which records the difference in time between any two start and end points. To account for the start point, an additional time point t_0 is added which represent time zero.

Constraints are added to the STN whenever new actions are added, open conditions are resolved and ordering constraints are imposed. The duration, δ_i, of a durative action a_i is specified as a conjunction of simple duration constraints $\delta_i \bowtie c$, where c is a real-

valued constant and $\bowtie \in \{=, \leq, \geq\}$. Each simple duration constraint gives rise to temporal constraints between the time nodes $t_{2i-1|}$ and t_{2i} when adding a_i to a partial plan. The temporal constraints, in terms of the minimum distance d_{ij} between two time points, are as follows:

$$\delta_i = c \quad d_{2i-1,2i} = c \wedge d_{2i,2i-1} = -c \tag{4.2}$$

$$\delta_i \leq c \qquad d_{2i-1,2i} \leq c \tag{4.3}$$

$$\delta_i \geq c \qquad d_{2i,2i-1} \leq -c \tag{4.4}$$

The semantics of PDDL dictates that every action be scheduled strictly after time zero. To ensure this we allow for a minimal time gap ε between these two time points, i.e. $\forall_{i>0} d_{2i-1,0} \leq -\varepsilon$. This is also the general method to encode that a time point should precede another time point.

Every time a temporal constraint is added to VHPOP all the shortest paths that could have been affected are updated. This operation can be executed in $O(|A|^2)$ time, thus it is quite a costly action to perform.

Once a plan without flaws has been found we still need to decide when to execute which action. Given the STN we have any number of possible schedules in which to execute the plan because the STN gives us the tightest constraints. So any ordering that does not violate these constraints is valid. Since we are trying to minimize the makespan we choose the times as early as possible as our final schedule for the plan.

4.2.2 Plan selection heuristics

One of the other important features of VHPOP is that it bridges the gap between plan-state planning and heuristics used in state-space planning. Instead of relying on the number of flaws to select a plan to work on it uses an adapted version of the h_{add} heuristic where it tries to account for positive interaction between actions. The problem in defining such an heuristic remains that we are dealing with partial plans and not with explicit state representations. Instead we look for potential actions which could unify with any open conditions in a partial plan, or more formal:

Given a literal q, let $GA(q)$ be the set of ground actions having an effect that unifies with q. The cost of the literal q can then be defined as:

$$h_{add}(q) = \begin{cases} 0 & \text{if } q \text{ unifies with a literal that holds initially} \\ \min_{a \in GA(q)} h_{add}(a) & \text{if } GA(q) \neq \emptyset \\ \infty \text{otherwise} & \end{cases} \tag{4.5}$$

A possitive literal q holds initially if it is part of the initial conditions. A negative literal $\neg q$ holds initially if q is not part of the initial conditions. The cost of an action a is defined as:

$$h_{add}(a) = 1 + h_{add}(prec(a)) \tag{4.6}$$

Which follows the same definition of h_{add} in state-space planners. The cost to handle other language constructs is given in the next list:

$$\text{Existentially quantified variables} \qquad h_{add}(\exists x.\phi) = h_{add}(\phi) \tag{4.7}$$

$$\text{Conjunctions} \qquad h_{add}(\bigwedge_i \phi_i) = \sum_i h_{add}(\phi_i) \tag{4.8}$$

$$\text{Disjunction} \qquad h_{add}(\bigvee_i \phi_i) = \min_i h_{add}(\phi_i) \tag{4.9}$$

The additive heuristic for a partial plan π with open condition set $OC(\pi)$ can now be defined as follows:

$$h_{add}(\pi) = \sum_{\xrightarrow{q} a_i \in OC(\pi)} h_{add}(q) \tag{4.10}$$

Like our previous discussion of the h_{add} heuristic this algorithm is not admissible and will not perform very well for strong interconnected problems. VHPOP tries to account for the possible positive interaction by looking at already existing actions in the plan which produce an effect that could be unified with the given literal. This can only be the case if the given action can be ordered before the action requiring the literal. Or more formally:

$$h^r_{add}(\pi) = \sum_{\xrightarrow{q} a_i \in OC(\pi)} \begin{cases} 0 & \text{if } \exists_{a_j} \in A \text{ s.t. an effect of } a_j \text{ unifies with } q \text{ and } a_i \prec a_j \notin O \\ h_{add}(q) & \text{otherwise} \end{cases} \tag{4.11}$$

Note that this algorithm is weaker than h_{ff}[21] as it only takes actions into account which are already part of the plan and not action which are added later. Nonetheless this heuristics is a big step up from the previous partial order planners which only rely on counting flaws for plan selection.

4.2.3 Flaw selection strategies

VHPOP used a combination of 4 flaw strategies during the IPC-3 competition, these flaw selection strategies are:

- MW-Loc: $\{n,s\}LR/\{o\}MW_{add}$
- MW-Loc-Conf: $\{n,s\}LR/\{u\}MW_{add}/\{l\}MW_{add}$
- LCFR-Loc: $\{n,s,u\}LR/\{o\}LR$
- LCFR-Loc-Conf: $\{n,s,u\}LR/\{l\}LR$

VHPOP uses all flaw selection strategies in unison by starting 4 planning procedures at the same time. Every planning procedure runs for 1000 iterations before switching to the next. The planning process stops once the first solution is found. The reason for doing so is that every flaw selection strategy has its own strengths and weaknesses and by using multiple at the same time hopefully at least 1 will be suitable for the problem at hand.

Chapter 5

Recent advances in forward-state planning

Having discussed the last advancements in partial order planning we will now turn to the recent advances in state-space planning and in particular forward-state planning. In our discussion we will only limit ourself to the planners which are relevent to our work. Other planners like LPG[14], for example, while relevant in planning in general will not be covered in this work. The focus on this section is mainly on the heuristics employed by these planners as we will not adopt their search algorithms.

5.1 Fast forward

The introduction of Fast-Forward(FF)[21] in 2001 was a big step up from the HSP planner[5][4]. It combines some clever search techniques and a better informed heuristic than used so far. FF is a forward-chaining search-based planner, uses the relaxed planning graph heuristic to guide its search and two different search strategies to find a solution to a planning problem. FF has performed very well in the AIPS-2000 planning competition where it won two "outstanding performance" awards for its performance in the "fully automatic" track. It is noted that FF finds solutions very quickly although the quality of the plans (i.e. plan length) tends to vary with respect to the degree of optimality. FF is considered a big milestone in planning and a lot of subsequent planning systems were build based on FF or used the FF heuristic h_{ff}.

5.1.1 Heuristic

The main aspect of FF is the heuristic it applies to planning problems. It adapts the h_{add} heuristic from HSP by using a GraphPlan-style algorithm. It still relaxes the plan by ignoring delete effects, but for every state it builds a relaxed planning graph to the goal state which only contains positive effect and solves this problem in polynomial time. A solution to a relaxed plan graph is given as $P' = \langle O_0, \ldots, O_m - 1 \rangle$ where O_i is the set of actions selected in action layer i, and m is the number of the fact layer first containing all goal atoms. The heuristic is computed as follows:

$$h(s) = \sum_{i=0,\ldots,m-1} |O_i| \tag{5.1}$$

When searching for a plan in the relaxed plan graph it is often best to find the shortest plan as possible. To accomplish this the following heuristics are used in how to solve the relaxed plan. When faced with a choice which action to select to support a fact, if there is a no-op available to make this happen always go with the no-op. If this it not the case then we select the action whose preconditions seems the easiest to achieve:

$$\mathit{Difficulty}(o) = \sum_{p \in \mathit{prec}(o)} \min\{i | p \text{ is a member of fact layer number } i\} \tag{5.2}$$

Furthermore the actions are linearised in the order in which they get selected. Finding an optimal linear action linearisation for a parallel set of achievers O_i is NP-complete. This solution will be found in polynomial time.

The given heuristic is not admissible but informed which is a great step up from the h_{max} and h_{add} heuristics described before. Because we actually construct a relaxed planning graph and solve this one we take both all goal atoms and positive interactions between actions into account.

5.2 Fast-Downward

Another take on domain analysis was taken in 2004 with the introduction of Fast-Downward[17], which won the ICAPS 2004's classical track. One of its features is that it does not use PDDL as the planning language directly but first translates it into a language called SAS^+[18]. In this representation values are not proportional, but multi-valued, e.g. a truck that is associated with a location is represented in PDDL with one proposition for every location, with only true in any given state. In SAS^+ a variable is created for the location of the truck and it can have exactly one value amongst the locations. Clearly the SAS^+ representation is more intuitive to grasp and it also prunes the number of possible states which can be represented. Giving the last example, if we have n locations the number of states representable in PDDL is 2^n while in SAS^+ there are exactly n states.

Before some actual search can take place we must translate a PDDL representation into the according SAS^+ representation. While we will not discuss the actual method used in Fast-Downward the idea is that if we find a collection of propositional values which can be represented by a single SAS^+ variable we know that only one of these propositions can be true at any time. This is an invariant condition of the domain that exactly one of these propositions is true or in Graphplan terminology all pairs of these propositions must be mutex.

Although SAS^+ does not allow us to define domains which cannot be defined in PDDL it does uncover some hidden constraints of the domains which could be used to our advantage. This is exactly what Fast-Downward does.

5.2.1 SAS^+ representation

A concise SAS^+ representation of a planning task can be generated from a typical PDDL representation automatically[18].

Definition 5.2.1 *A SAS^+ planning task is a tuple $\Pi = \langle V, O, s_0, s_g \rangle$ where:*

- *V is a finite set of multi-valued* state variables*, each with a finite domain D_v.. A fact is a pair $\langle v, d \rangle$ (also written $v \mapsto d$), where $v \in V$ and $d \in D_v$. A partial variable assignment s is a set of facts, each with a different variable. (We use set notation such as $\langle v, d \rangle \in s$ and function notation such as $s(v) = d$ interchangeably.) A state is a partial variable assignment defined on all variables V.*
- *O is a set of operators, where an operator $o \in O$ is a tuple $\langle name, prec, effects \rangle$ of partial variable assignments.*
- *s_0 is a state called the initial state.*
- *s_g is a partial variable assignment called the goal.*

An operator $o = \langle name, prec, effects \rangle \in O$ is applicable in state s if $prec \subseteq s$. In that case, it can be applied to s, which produces the state s' with $s'(v) = effects(v)$ where $effects(v)$ is defined and $s'(v) = s(v)$ otherwise. We write $s[o]$ for s'. For operator sequences $\pi = \langle o_1, \ldots, o_n \rangle$, we write $s[\pi]$ for $s[o_1] \ldots [o_n]$ (only defined if each operator is applicable in the respective state). The operator sequence π is a plan iff $s_g \subseteq s_0[\pi]$.

Each state variable of a SAS^+ planning task has an associated directed graph which captures the ways in which the value of the variable changes through operator application[23].

Definition 5.2.2 *The domain transition graph(DTG) of a state variable $v \in V$ of an SAS^+ task $\langle V, O, s_0, s_g \rangle$ is the digraph $\langle D_v, A \rangle$ which includes an arc $\langle d, d' \rangle \iff d \neq d'$ and there is an operator $\langle name, prec, effects \rangle \in O$ with $prec(v) = d$ or $prec(v)$ undefined, and $effects(v) = d'$.*

5.2.2 The causal graph heuristic

Once we have created the DTGs for the variables of a problem, we know how the value of a variable can change and which transitions are possible between the different values. However, a DTG does not stand alone and for most problems DTGs will have external dependencies with other variables to make its transitions. These external dependencies are captured in a causal graph(CG).

Definition 5.2.3 *Let Π be a SAS^+ planning task with variable set V. The causal graph of Π ($CG(\Pi)$), is the directed graph with vertex set V containing an arc (v, v') if $v \in V \wedge v' \in V \wedge v \neq v'$ and one of the following conditions is true:*

- *The domain transition graph of v' has a transition with some condition on v.*

- *The set of affected variables in the effect list of some operator includes both v and v'.*

In the first case, we say that an arc is induced by a transition condition. In the second case we say that it is induced by co-occurring effects.

The computation of the causal graph heuristic is done by using a path finding algorithm the find the shortest paths through the DTGs of every goal atom independently. While traversing the DTG related to a goal atom we try to find the shortest possible path from the initial value of that variable. To account for extra work which might need to be done (e.g. we need to move a truck to the same location as the package before it can be loaded), every time we hit a precondition which is not satisfied we solve the DTGs related to the unsatisfied literals in similar fashion and take the number of steps necessary as the weight attached to traversing an edge in the DTG. A major difference between this heuristic and heuristics utilized by FF and HSP is that it actually takes the interactions into account of DTGs directly connected by the causal graph and updates the value of the variables according to the effects of the transitions. Thus it does not allow for multiple value of the same variable to hold true at the same time.

For example take the following problem from the same domain. A package is located at location *A*, a truck is located at location *D* which is also the goal location of the package and the graph is constructed like depicted in figure 5.1. Using the FF heuristic it will realize it needs to drive the truck to A, load the package and unload it again at D. But because it does not take into consideration the delete effects, when it reaches *A* it assumes that the truck can be at any location in the graph and thus simply unloads the package. The heuristic value is 5. But suppose the truck has moved to location *B* now, using the FF heuristic we will again get 5 as the heuristic. In other words the whole driving sequence to the package is a plateau. If, on the other hand, we use the causal graph heuristic it will record the fact that the truck needs to move to location *A* and once it has arrived there that it needs to drive back. In this case even the causal graph heuristic will give us the true heuristic of 8.

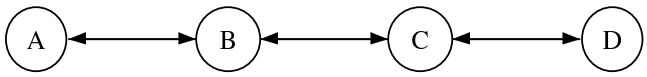

Figure 5.1: Simple road network.

This is one of the fundamental strengths of the causal graph heuristic, if the causal graph does not contain any cycles, is that it will always return the true heuristic for achieving a goal atom. Unfortunately this is hardly ever the case. If there are cycles in the causal graph there is no way to identify the order in which to evaluate the costs for transitions in the DTGs and propagate them upwards. The approach taken by Fast-Downward is to cut cycles in the causal graph by removing dependencies which cause the cycles, edges which affect the least preconditions are removed first. This means that dependencies which affect the most preconditions are preserved. With the cycles removed we can again use a pathfinder algorithm to find the shortest path within every DTG linked to a goal atom.

The heuristic can be formalized as:

$$h_{cg} = \sum_{v \in s_g} cost_v(v_s, v_g) \tag{5.3}$$

Where $cost_v(d, d')$ stands for solving the subproblem $\Pi_{v,d,d'}$ with the value of v in the initial state is d ($s[v] = d$) and the goal value is d'. Per goal atom, only that particular variable and its parents (as defined in the CG) are considered. Because we assume an acyclic graph we can apply a Dijkstra algorithm to calculate the transition costs in the DTGs from every node to every other node. We start with the variables which have no dependancies according to the CG, we can simply apply Dijstra to calculate the distance between every node pair $d, d' \in D_v$, resulting in the optimal cost $cost_v(d, d')$ for every pair. Next we move to the variables which have dependencies on other variables which have already been computed. We modify the original Dijkstra algorithm to take into account the cost of solving the parent variables from their current value e to the goal value e'. To calculate the transition cost $cost(d, d')$ we simply add all costs for achieving the parent variables from the current state to the precondition as defined by the transition arc. So if we have a transition arc with preconditions for a set of variables: $v_1 = e_1, \ldots, v_n = e_n$, where all $e_i, e_i' \in D_v$ and $cost_{e_i,e_i'}$ already computed, we define the cost function as follows:

$$cost_v(d, d') = 1 + \sum_{i=1,\ldots,n} cost_{e_i,e_i'} \tag{5.4}$$

Note that with every transition made during the computation of the algorithm, we keep track of our state. Thus if we find a transition which decreases the cost from some $d \in D_v$ to $d' \in D_v$ by following an arc with some edge z we will change the state accordingly: $s[v] = z$. By doing this we are able to take advantage of the context of a variable and get better heuristic values as highlighted in the example above.

While the causal graph heuristic is very good at domains with no or few cycles in the causal graph, if the interdependencies between variables are very strong the algorithm will break down as we need to delete a lot of information by removing dependencies in the causal graph. One of these problems is the blocks world domain in which a planner must find a sequence of actions to stack and unstack blocks on a table to reach some goal state. We are only able to interact with the top most block of each tower, so there is a very tight interconnection between the different blocks. In order to circumvent this weakness Fast-Downward uses h_{ff} as a second heuristic to solve the problem in parallel in case the causal graph heuristic fails.

5.2.3 ADCG

In 2008 a version of the causal graph heuristic was introduced called additive-disjunctive causal graph heuristic(ADCG)[27] which does not require to break the cycles in the CG. Effectively it does not even require the CG at all to compute the heuristic. In order to do this the ADCG heuristic assumes subgoal independence and will not take positive interactions into account, this allows it to calculate an heuristic value without breaking cycles in the CG.

The way in which we can do this is quite straight forward. Informally, given a start value v_s and a goal value v_g such that both are part of the same variable V. We search for the shortest path in the DTG and solve external dependencies to other DTGs independently. The additive-disjuncitve causal graph heuristic is given as:

$$h_a^c(s) \stackrel{def}{=} \sum_{x \in s_g} h(x|x_s) \tag{5.5}$$

Where x_s is defined as the value of the variable x in s_0. Now to calculate the actual heuristic we start from the values of all variables as defined in the initial state and try to find the shortest path for all variables defined in the goal state. Because we treat every subgoal independently, actions applied to satisfy a certain transaction will have no effect on the state of other search branches. The function is defined as:

$$h(x''|x') \stackrel{def}{=} \begin{cases} 0 \text{ if } x'' = x', \text{else} \\ min_{o \in O | x \in o_{prec} \wedge x'' \in o_{effects}} [1 + h(x|x') + \sum_{x_i \in o_{prec} | x_i \neq x} h(x_i|x_i')] \end{cases} \tag{5.6}$$

where we look for the minimal path to achieve value x'' from value x'. If the value of the given variable is equal to the goal value we return 0. Otherwise we look for the set of transitions with minimal cost. Note that we also take care of solving any other preconditions which are not part of the variable x' and x'' belong to. In order to do this, we first calculate the state which results from achieving x from x'. This is the value of x_i' which is written as $s(x|x')$ and obtained by replacing all values for the affected variables by applying o to the current state with the values in o_{effect}.

5.3 Landmarks

The last advancement we will discuss, which is also most relevant to our work, is landmarks[35]. Landmarks are states in a plan which must necessarily be visited before we can reach the goal state. Furthermore, given an ordering between these landmarks, we can use this information to guide planner towards its final goal and hopefully come up with better solutions which require less time to compute.

5.3.1 Finding landmarks

Unfortunately finding all landmarks for planning problem is a PSPACE-hard problem[22], so we necessarily have to consider an incomplete subset of all landmarks. The most trivial landmarks are the original goals, but if we note that all actions which can achieve any of these goals have a similar precondition, we can mark this precondition as a landmark as well. Unfortunately this will give us a very small set of landmarks for most domains, so additional work is necessary to unearth more landmarks.

Except finding landmarks it is also beneficial to find ordering between landmarks. For example, the notion of reasonable orders[24] states that a pair of goals A and B which, if ordered so that B is achieved before A, it is not possible to reach a state in which A and B are

both true, from a state in which just A is true, without having to temporarily destroy A, in that case it is reasonable to achieve B before A to avoid unnecessary effort. In the literature the following orderings have been defined:

- Natural ordering: $A \rightarrow B$, iff in each operator sequence where B is true, at time i, A is true at some time $j < i$.
- Necessary ordering: $A \rightarrow_n B$, iff in each operator sequence where B is true, at time i, A is true at time $i-1$.
- Greedy-necessary: $A \rightarrow_{gn} B$, iff every operator sequence where B is first added at time i, A is true at time $i-1$.
- Reasonable ordering: $A \rightarrow_{ro} B$, iff starting from any state where B was achieved before A: B must be true at some point later than the achievement of A; and one must delete B on the way to A.
- Obedient reasonable ordering: $A \rightarrow_o B$, iff given a set of reasonable ordering constraints O, if a planner commits to obey all the ordering constraitns in O, $A \rightarrow_{ro} B$ arises because of this.

In order to find landmarks and their orders, methods which utilize the RPG[22] (LM^{RPG}) and DTGs[37][36] have been devised.

Intuitively when we use RPGs to find landmarks, what we can do is construct the RPG and leave out any operator which would achieve some literal l. When the RPG levels off, the last layer of facts is an over-approximation of the set of facts that can be achieved before l in the planning task. Any operator that is applicable to this last layer and achieves l is possibly the first achiever of l. When we take a disjunctive set of literals from the first achievers' shared preconditions, such that a set contains one precondition fact from each first achiever, these sets from disjunctive landmarks.

The actual method is more sophisticated than described above and we refer to the respective paper for a detailed description. We will provide more detail about the approach adopted in LAMA since our solution is based on their method of deriving landmarks. The method of finding landmarks is quite similar to LM^{RPG}, but differs in a number of fundamental ways. In contrast to the previous method it only derives sound orderings and it takes into account disjunctive landmarks. Like LM^{RPG}, it creates disjunctive sets of facts from the preconditions of first achievers of a landmark B such that a set contains one precondition fact from each first achiever of B. In order for these preconditions to be considered landmarks it is required that they stem from the same predicate symbol. Each set A found this way is then recorded as a disjunctive landmark and ordered greedy-necessarily before B. If B is a disjunctive landmark, then the first achievers of B are all operators which achieve one if the facts in B. Additional landmarks can be derived from doing graph analysis on the DTGs. If every path from the value of a variable in the initial state s_0 to the value in the goal state s_g needs to visit a certain node, we mark this node as a landmark. The way to test this is by removing a single node from the DTG and test if there is still a path between the

two nodes. If not this node can also be added as a landmark which can be naturally ordered after the initial value.

5.3.2 Search with landmarks

After finding landmarks, there is still a question of how to utilize these as good as possible. One possible way is by constructing a landmark graph based on the ordering between landmarks, where the vertices are the landmarks and the edges are the orderings. Because the landmark generation process used for LM^{RPG} does not guarantee to produce sound orderings between landmarks cycles in this graph may occur and need to be broken. Once this is done, we can decompose the planning task into smaller subtasks which need to be accomplished in succession. The landmarks are presented to the planner as a disjunctive goal and upon achieving one of the landmarks, this is removed from the landmark graph and the process continues until the landmark graph is empty at which point the planner is asked to construct a plan to the actual goal from its current state. This process is called LM^{local}[22] and results show a speedup on most domains but at the expense of plan quality. Unfortunately due to the fact that the landmark generation process can create unsound orderings the search process sometimes fails to find solution where it previously was able to solve problems without the landmark heuristic.

A better approach was introduced by the planner LAMA[37], which is based on the Fast-Downward code base and also uses some of its heuristics. The way it utilizes landmarks is by simply counting how many more landmarks need to be achieved before the goal is satisfied. It also accounts for landmarks which are required multiple times during search. The heuristic employed to estimate the number of landmarks which still needs to be achieved is defined as: $l = n - m + k$, where n is the total number of landmarks, m is the number of landmarks that are accepted, and k is the number of accepted landmarks that are required again. A landmark B is accepted in a state s if it is true in that state and all landmarks ordered before B are accepted in the predecessor state from which s was generated. An accepted landmark is required again if it is not true in s and it is the greedy-necessary predecessor of some landmark which is not accepted. Given this heuristic it can be simply integrated with other heuristics we have described before, although it would not benefit admissible-heuristics directly as adding the landmark heuristic would make it non-admissible. Nonetheless, experiments have shown that integrating this information with other heuristics yields better results[36]. LAMA was actually the winner during the ICAPS '08 competition for the "Sequential satisficing track" and "Learning tracks domains".

5.4 STeLLa

Along with VHPOP another planner called STeLLa[39] entered the IPC-3 competition. This planner tries to gain better planning speedups by introducing a general technique of decomposing a planning problem $\langle s_0, s_g, O\rangle$ into several subproblems of the form $P_i = \langle IS_{i-1}, IG_i, O\rangle$, where IG_i is an intermediate goal and IS_{i-1} is the state reached by solving the previous subproblem. The first subproblem $IS_0 = s_0$ and the last subproblem $IG_n = s_g$ These subproblems are solved by any planner to obtain a plan $P_i = \langle o_{i1}, o_{i2}, \ldots, o_{ij}\rangle$ and the

final plan can be found by concatenating the plans P_i: $P = P_1 \circ P_2 \circ \cdots \circ P_n$. The idea is that solving these subproblems in sequence should be easier than solving the problem as a whole. Previous work has identified two ways of splitting a problem up into subproblems, for example SGPlan[7] tries to solve every goal atom independently and later 'glue' them together, this work was later extended in TSGP[9]. The decomposition technique in STeLLa, however, does not split the planning problem 'vertically' but rather 'horizontally'. Which means that instead of asking the planner to solve the goal state, the planner must solve a sequence subproblems and use their solution to solve the next subproblem until the final goal state is reached.

One of the nice features of STeLLa is that it provides a general framework for problem decomposition which can be applied to virtually any planner. In their experiments STeLLa has been tested with FF, LPG, and VHPOP as the base planner. In the case of VHPOP experimental results show considerable speedups in search and rendered previously unsolved problems solvable in the same time span.

5.4.1 Constructing the subproblems

In order to construct the set of subproblems STeLLa uses the landmarks generation process from LM^{RPG}[22]. Given a landmark generation graph $G = \{V, E\}$, where every vertex $v \in V$ represents a landmark and every edge $e \in E$ is labeled with the type of ordering between its two vertexes: $e = \{from, to, edgetype\}$, a subproblem is generated such that the following two properties hold:

- Consistency property: All literals in an IG must be consistent with each other $\forall l, l' \in IG : \neg inconsistent(l, l')$.
- Ordering property: A literal l belongs to an IG if and only if all of its predecessor nodes in the $G(V, E)$ have been included in a previous IG before l: $\forall_l \in IG_i : \forall_{l'}\ inV : l' \leq l \rightarrow l' \in IG_j \wedge j < i$.

Two literals are inconsistent if they cannot simultaneously coexist in the same correct planning state. STeLLa uses the inconsistent function provided by the TIM API[13] to approximate this relationship. Based on these relationships and the ordering between the landmarks the following 'active interference' rules are defined by STeLLa to compute the subproblems. Given three consecutive IGs: IG_{i-1}, IG_i, IG_{i+1}, and let l and l' be two landmarks the belong to G:

1. If l belongs to IG_{i-1}, l will be propagated to IG_i, until a successor literal in G is visited through a necessary order.
2. If two inconsistent landmarks l and l' belong to IG_i and l' is a propagated literal, l is delayed to IG_{i+1}.
3. If two inconsistent landmarks l and l' belong to IG_i, and there is a literal l_p in the previous IG such that $l_p \rightarrow_n l'$, then l is delayed to IG_{i+1}.

4. If two landmarks l and l' belong to IG_i, and there is a landmark l'' so that $l'' \rightarrow_n l'$ and l'' is inconsistent with l, then l is delayed because the plan should achieve l' first in order not to delete l'' with l.

Every subproblem is now constructed as follows:

1. First, an approximation to IG_i is computed with all the landmarks l that have a predecessor literal $l' \in IG_{i-1} : IG_i = \{l \in G / \exists_{l'} \in IG_{i-1} : l' \leq l\}$.
2. Secondly, this first approximation is refined in three stages:
   a) Delay landmarks l that have a predecessor literal l' in G such that l' has not been visited (Ordering property).
   b) Propagate the corresponding literals from IG_{i-1} to IG_i.
   c) Check remaining interference rules between the literals in IG_i.

A special case is applied to literals from s_g. Once a literal $g \in s_g$ has been included in an IG, g will be propagated unless an inconsistent landmark is added to the same IG. In this case g will be delayed to the last IG.

Chapter 6

Integrating landmarks in VHPOP

Having discussed the most recent advances in both partial order planning and state-space planning, we will now proceed to discuss our contribution to partial order planning which follows in the footsteps of RePOP and VHPOP. As we have seen, both successfully integrated techniques adopted from advances in state-space planning to gain a more competitive advantage. The most recent advance in planning is the successful adaptation of landmarks in planning to gain better performance both speed and quality wise, as exemplified by LAMA. Earlier work on integrating landmarks in FF, in which the planner would only see the landmarks closest to him effectively splitting the planning problem in vertical slices to work though also showed good results speed wise but took a bigger hit on the quality of the produced plans.

In this work we will adopt both approaches in the context of partial order planning and see if in doing so we can raise the competitiveness level of the latest partial order planner, VHPOP. The reason why we choose VHPOP is for a couple of reasons, first of all it is the last partial order planner to have competed in a international planning competition, in 2003, so we have a good benchmarks set the planner will work on which will serve as a good reference point. Secondly, the program is designed to handle multiple flaw and plan selection heuristics which made it easily adaptable to our needs.

In the following sections we first describe our approach to integrate landmarks into VHPOP by adopting the previously described 'FF approach', which means that landmarks will not solely function as part of the heuristic function, but rather serve to slice the planning problem of in vertical slices which need to be solved in succession. Unlike LM^{local} which is quite straightforward we will see that guiding the planner with this approach needs a little more work when applied to partial order planning. Next, we adopt the 'LAMA approach' by using the landmarks solely as an heuristic guidance on top of VHPOP's ordinary heuristic and report on the found results. Finally we discuss some other techniques we found during our research and which can help to prune the search space of VHPOP using landmarks and information derived from the SAS^+ representation and we discuss a novel way to do flaw selection based on the ADCG heuristic[27].

6.1 FF approach

In this section we describe the method employed to split up the planning problem into subproblems, using landmarks. These are also used in the next section in which we present our novel flaw selection strategy. One of the earliest papers on landmarks detailed a way to split planning problems using landmarks (LM^{local}[22]), by successively planning to the "nearest" landmark until all are visited. We propose a different approach. First we discuss the process we used to derive landmarks and the modifications we make in comparison with earlier approaches. Next we explain how the *landmark generation graph* is used to split the problem up into consecutive planning problems and finally we report on the heuristics used and results obtained.

6.1.1 Deriving landmarks

Whereas LM^{local} successively plans to the "nearest" landmark, in VHPOP we do not have the same option as we lack an explicit state definition. Furthermore, landmark orderings derived using the LM^{RPG} algorithm are not sound which can distort the planning process and can cause the planning process to fail on a task, even though the underlying planning process is complete.

For this reason we use the landmark generation process derived by Richter, Helmert and Wesphal[36] which produces sound orderings, leading to shorter plans and an improved success rate, compared to LM^{RGP}, when applied to the same planners. The process we use to generate the landmark layers is quite similar to the approach adopted by STeLLa, but does very in some significant ways. First of all, we also consider disjunctive landmarks and employ a different technique to handle inconsistencies within landmark layers. Where STeLLa uses a preprocessing algorithm to make sure every subproblem does not contain inconsistent landmarks we allow for inconsistent landmarks and let the planner decide which landmark it wishes to utilize. Furthermore, where STeLLa only considers a general framework which can be applied to any planner we focus solely on partial order planners and in effect derive more techniques based on landmarks than discussed in the work on STeLLa.

6.1.2 Determining the ordering of landmarks

As discussed before, because VHPOP lacks an explicit state representation it is not clear how we adopt the LM^{local} approach by planning to the "nearest" landmark. For example, consider the driverlog domain and suppose that the landmark closest to a goal is $(at\ package1\ s1)$ and the goal is $(at\ package1\ s2)$. From a planning perspective it is unclear how this problem can be solved, lacking information regarding the location of the trucks and potential drivers. All we know about the other variables is their goal value, an interesting approach — which is not pursued here — might be to inverse the preconditions and effects of actions. This approach would mimic the LM^{local} approach closer than the approach we pursued, we leave this for future work.

In our approach we insist that every landmark layer is a fully described state, which means that for every variable a value is defined. When a landmark generation graph is constructed we notice that several types of orderings can occur between landmarks, recall:

- Natual ordering: $A \rightarrow B$, iff in each operator sequence where B is true, at time i, A is true at some time $j < i$.
- Necessary ordering: $A \rightarrow_n B$, iff in each operator sequence where B is true, at time i, A is true at time $i-1$.
- Greedy-necessary ordering: $A \rightarrow_{gn} B$, iff every operator sequence where B is first added at time i, A is true at time $i-1$.
- Reasonable ordering: $A \rightarrow_{ro} B$, iff starting from any state where B was achieved before A: B must be true at some point later than the achievement of A; and one must delete B on the way to A.
- Obedient reasonable ordering: $A \rightarrow_o B$, iff given a set of reasonable ordering constraints O, if a planner commits to obey all the ordering constraitns in O, $A \rightarrow_{ro} B$ arises because of this.

Given a landmark generation graph $G = \{L, E\}$, where every vertex $l \in L$ represents a landmark and every edge $e \in E$ is labelled with the type of ordering between its two vertexes: $e = \{from, to, edgetype\}$. The landmark generation graph can be split up into separate landmark layers by iteratively grouping all landmarks that have no incoming edges. We start by labelling all landmarks as active. The first set of landmarks is the initial state. After every iteration we label all discovered landmarks as inactive and repeat the procedure, until all landmarks have been marked as inactive. For every landmark we denote the iteration number at which it was made inactive, which we refer to as the *layer number*, this procedure ensures that, if landmark l_1 is ordered before l_2 in the landmark generation graph, then the layer number of $l_1 < l_2$.

This process does not guarantee a correct ordering of the values each variable will take, it will only create a correct ordering for the explicit orderings which are defined in the landmark generation graph. Unfortunately, finding all landmarks and their ordering is a PSPACE-complete problem[22], so we cannot hope to ever find all correct orderings. Note that this means that we can have multiple values for the same variable in a single layer. How we deal with this issue will be explained in the next section where we explain the process of creating the actual landmark layers. Relating this work back to STeLLa, this process guarantees the ordering property.

6.1.3 Creating the landmark layers

Now we propose our stratification technique. Given a set of pairs of landmarks and their respective *layer number* $H = \langle l, n \rangle : l \in L$; and $n \in \mathbb{N}$, we divide these into consecutive subsets $X_1, X_2, \ldots, X_n$, such that every subset defines a state. The number of subsets is equal to $\max n \in N$. For the remainder of the discussion we will add a notion of directionality: because we are doing a goal-directed search, we say that the first subset is the goal set and the last is the initial state.

Definition 6.1.1 *A landmark $l \in L$ defines a variable $v \in V$ if*

- *l is not disjunctive and the value of* $l \in D_v$.
- *If l in respect of v is inconsistent, i.e.* $\exists_{a \in l} \exists_{b \in l} a \in D_v \wedge b \in D_v \wedge a \neq b$.

Definition 6.1.2 *A set of landmarks L defines a state if* $\forall_{v \in V} \exists_{l \in L} | l$ *defines v.*

Definition 6.1.3 *The* minimal landmark layer *at layer j is defined as the subset* X_j. *The value of every state variable* $v \in V$ *is defined as:*

$$X_j^v = \min_{i \geq j} \langle l, i \rangle \in H | l \text{ defines } v$$

This definition ignores all landmarks that do not define a variable, but it gives us a stratification of the planning problem. However, the choices represented by disjunctive landmarks are an essential part of planning, especially as more resources are available to accomplish a task. For example, given a landmark like $(at\ rover1\ s1) \vee (at\ rover2\ s1)$ it is unclear if this landmark defines the location of *rover1* and / or *rover2*. Either one or both could be true, but this will only become clear during the planning process and we cannot determine this in advance. Therefore we continue to traverse the next layers searching for the set of earliest landmarks which defines all variables. If no variable has been defined in any landmark we default to the value given in the initial state which defines a variable by definition. For example if we look at figure 6.1 which is derived from driverslog problem file pfile01 we see there is an ambiguity over which driver gets to drive the truck to its goal location. Note that none of these landmarks define the variable for either driver, forcing us to include the location of the drivers from the initial state in every landmark layer.

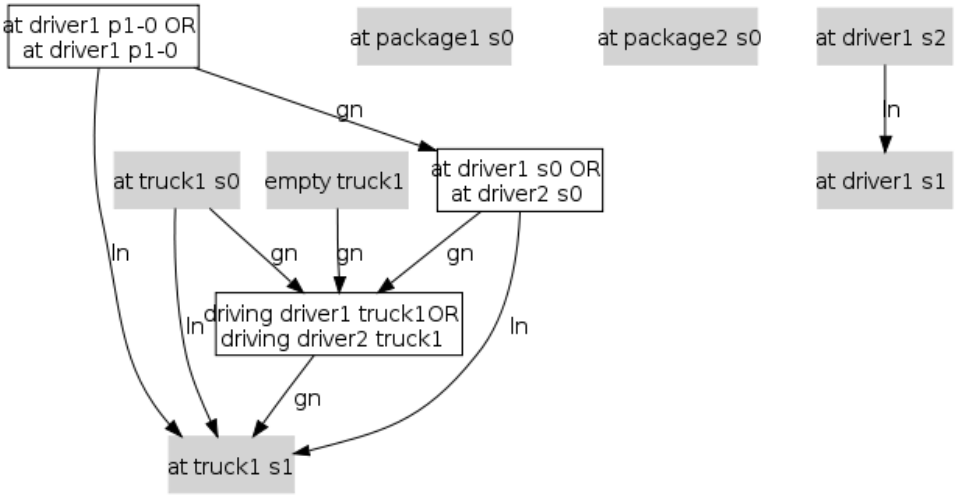

Figure 6.1: Landmark generation graph for pfile01 - Driverlog (atoms from the initial and goal state have been grayed out).

At a bare minimum we could opt to ignore landmarks which are ambiguous in which case we end up with the *minimal landmark layer* set $X_1, X_2, \ldots, X_n$. The problem with this

approach, however, is that as more objects are included in a planning problem we see that this ambiguity is an essential part of planning. Very rarely do we see non-disjunct or even conjunct set of landmarks which can be usefully utilized to split up the planning problem into several layers. Therefore we account for the ordering information derived from the landmark generation graph by adding a special *landmark action* to the planning problem, which is defined as an action which contains all landmarks which are part of a landmark layer but do not define a variable and are ordered before the minimal landmark layer as defined above.

Definition 6.1.4 *Given a* minimal landmark layer X_j *and the set of* layer numbers, *the* landmark action *at layer j is defined as the subset Y_j. The set of landmarks for a state variable $v \in V$, for which the* minimal landmark layer *was found at* layer number *i is defined as:*

$$Y_j^v = \bigcup_{i>x\geq j} \langle l, i\rangle \in H | l\neg \textit{ defines } v$$

We can now define the successive *landmark layers* as $Z_1, Z_2, \ldots, Z_n$, where $Z_i = \langle X_i, Y_i\rangle$.

6.1.4 Using landmark layers in planning

Given that we have stratified the planning problems into landmark layers, we will now describe the modifications which had to be made to the original planner to make effective use of these landmark layers. In this section we will describe the several aspect which needed to be modified. We will start off with the plan selection heuristic and later discuss a novel way of doing flaw selection.

Plan selection heuristic Given a planning problem $P : \langle A, L, O, B\rangle$, we define the first subproblem as: $P_1 = \langle \{a_\infty, X_1, Y_1\}, \emptyset, \{X_1 \prec Y_1, X_1 \prec a_\infty, Y_1 \prec a_\infty\}, \emptyset\rangle$. In the previous section we have discussed the *landmark action* which serves as an extra 'state' from which VHPOP can use atoms from, provided they do not interfere with the *minimal landmark layer*. The reason why we can do this is because we know from the *landmark generation graph* that all the values of the landmarks in the *landmark action* will occur at some point after the values of the landmarks in the minimal landmark layer for every variable respectively. Thus the planner can safely assume that any value from the *landmark action* is already satisfied, given that causal links it supports does not interfere with causal links supported from the *minimal landmark layer* because of the ordering constraints. There is a catch however, because all landmarks added to the *landmark action* are disjunctive landmarks, there is no guarantee that all these landmarks are true, i.e. we cannot treat these landmarks as conjunctive landmarks but rather force the planner to make a choice which of atoms in the disjunctive landmarks are true. This highlights why we include the *minimal landmark layer* which the planner can fall back on in case the value of a variable cannot be derived from any of the disjunctive landmarks.

To estimate the number of steps that must still be completed from the current *landmark layer* to the initial state, we apply the FF heuristic h_{ff}. Since h_{ff} uses the RPG to derive its

heuristic it can account for the possibly inconsistent goal state as it simply accumulates all positive effects. Other heuristics like h_{cg} from Fast-Downward cannot be applied directly because these heuristics assume that goal states do not contain variables to which multiple values are assigned. Although variations on these heuristics can be constructed, e.g. by considering the set of conflicting values independently and taking the maximum. This line of research is not pursued in this research and is left for possible future work.

To solve the planning problem, P, we use VHPOP, but with a couple of changes. Most importantly, whenever a causal link is created from either X_1 or Y_1, we remove all atoms from the corresponding action that refer to the same state variable but assign it a different value. This allows the planner to choose which values amongst disjunctive landmarks should be assigned. To guide the planner, the FF heuristic is recalculated every time values are removed from any action. Note that in our implementation we do not consider the different orderings within the *landmark action*, it could very well be that there is a sequence of ordered landmarks — assigning a different value to the same variable — but when one landmark is chosen all other options are no longer considered. This is also an option to extent upon in future work.

Once the subproblem is solved, we move on to the next subproblem. To do this, we first remove the *landmark action* and *minimal landmark layer* and all causal links it supports. Next we insert the next set $\{X_i, Y_i\}$, with the appropriate orderings $\{X_i \prec Y_i, X_1 \prec a_\infty, Y_1 \prec a_\infty\}$ as before. After calculating the FF heuristic the process is repeated until a solution is found for the last subproblem which is the initial state. This solution is the final plan.

Flaw selection heuristic As we have seen before, partial order planners have never been known for very 'smart' flaw selection strategies, earlier work focused a lot on the order of *types* of flaws, e.g. solve unsafe links before treating open conditions, etc. Pollack, Joslin, and Paolucci[34] came up with a slightly more sophisticated flaw select strategy which takes the actual estimated cost into consideration and argued that choosing to select the flaw with the minimal repair value $\{o,n,s\}LC$ yield better results than previous developed strategies in a selected set of benchmarks. In this section we will present a new approach to flaw selection which will not only look at the type and cost of a given flaw as previous methods have, but rather analysis how a particular flaw is related to the rest of the problem and from this analysis select the most constrained flaw for valuation first.

To do this we make use of the previous discussed additive-disjunctive casual graph heuristic (ADCG). The idea is that instead of only looking at the cost and type of a given flaw we place the flaw in a broader context and check how the variable related to the flaw is constrained in the planning problem. We do this by checking the links in the Causal Graph to estimate how constrained a variable is, based on the number of incoming edges. In the original Causal Graph heuristic cycles needed to be broken in order to determine the order in which the variables should be evaluated. This in turn gave a particular ordering on the variables, from least to most constrained. In the literature a common technique in, for example, CSP, is to assign the most constrained variables first so in case a dead end is detected it can be pruned as quick as possible. We want to apply the same principle to the flaw selection strategy in partial order planners.

While the following method might not be applicable in most planners it fits very well

with planners which use a stratifying approach[8, 39], because the subproblems between the landmark layers are relatively small in scope compared to the whole problem. We observe that, in order to solve these subproblems, only a small subset of all operators needs to be considered. For example if we have a landmark layer with all trucks occupied by a driver and packages loaded in various trucks, we only need to consider drive actions to get the trucks to the drop-off points for the packages and do not need to consider the actions for getting the drivers and packages into the trucks. In other words, if we place this discussion in the context of the Causal Graph heuristic, instead of breaking cycles by checking the number of incoming edges we can remove cycles first by checking which actions are relevant.

In order to estimate the operators that will be relevant for solving a subproblem between two landmark layers we make use of the ADCG heuristic. There is a slight modification made to the actual heuristic, because we can have a situation where a variable starts with multiple values assigned to it. For simplicity we assume that all possible start values are true, so the pathfinding algorithm is modified to be able to cope with this. While searching for the shortest path per goal atom we keep track of the operators applied in the process and store all applied operators per subgoal. We store the set of all operators which correspond to the shortest path according to the ADCG heuristic. Next we generalize the operators found by this process by allowing the arguments of the operators to be replaced with any object which shares the same DTG structure. That is to say, any object that shares the same values and transitions can replace an existing argument value.

Definition 6.1.5 *A variable $v \in V$ is the defining value for a DTG d, if*

- $|d| > 2$.
- *For every node, v is one of its arguments.*
- *For every transition, v is one of its parameters.*
- *There is no other $w \in V | w \neq v$ for which the above hold.*

Definition 6.1.6 *A DTG d_1 is isometric with DTG d_2, if all of the following conditions hold:*

- $|d_1| = |d_2|$,
- *Every node in d_1 has an equivalent node in d_2 for which the proposition is the same and the terms are all equal except for the defining variable for both DTGs, and vice versa.*
- *Every transition in d_1 has an equivalent transition in d_2 for which the preconditions are the same, equal except for the defining variable for both DTGs and for which the* from *and* to *nodes are isometric, as defined above, and vice versa.*

For example, if a solution is found which has at least one drive action we replace that one action with all drive actions for all trucks and drivers which effectively share the same range of states they can be in. The reason for doing this step is because in the real plan we might use any combination of values for the variables as arguments and we cannot derive

this information directly from the heuristic. This is especially relevant when using the ADCG heuristic, because it assumes goal independence and will not see the advantage of using multiple resources at the same time, though this might be vital in solving the problem at hand. So we make a very optimistic assumption by assuming that all variables that share the same range of states and transitions could be used.

Once we have this extended set of operators, we construct a Causal Graph and will only consider this identified subset of all possible operators. From this point we use the normal cycle breaking algorithm of Fast-Downward[17] to find the final variable ordering that we use for our flaw selection algorithm. Given a set of flaws to choose from, we only consider the subset that is — according to the presented flaw selection strategy — most constrained. To distinguish between the most constrained flaws, we use the flaw selection strategies of VHPOP.

6.1.5 Example

As an example of how all of the above comes together we will take the 3rd problem from the Driverlog domain (pfile03) and see how this problem is decomposed into landmark layers and how the flaw selection strategy works out on every layer.

Landmark layers First, we detail the landmark layers as they are constructed by the landmark generation process. The landmark generation graph is depicted in Figure 6.2.

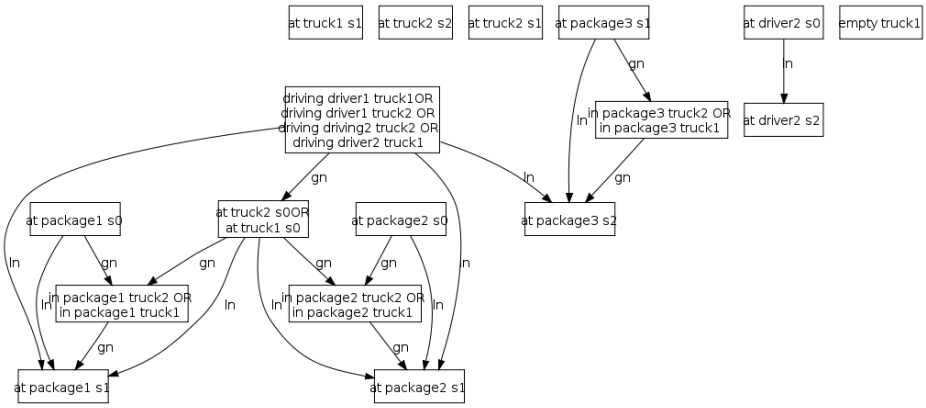

Figure 6.2: Landmark generation graph for pfile03 - Driverlog.

Following the landmark layer generation process as outlined above will cut the problem up into 5 layers as depicted in Table 6.1.

The last layer represents the initial state and the first layer the goal state. The layers show a nice progression towards the goal state. First the drivers board the trucks, next the packages are loaded and finally the packages are delivered, the trucks are at the right place

Layer	driver1	driver2	truck1	truck2	package1	package2	package3	package4
0th Layer	-	at s2	at s1	at s2	at s1	at s1	at s2	-
1st Layer	↑	↑	↑	↑	in truck1 ∨ in truck2		↑	↑
2nd Layer	↑	↑	at s0		↑	↑	in truck1 ∨ in truck2	↑
3rd Layer	driving truck1 ∨ driving truck2		↑	↑	↑	↑	↑	↑
4th Layer	at s1	at s0	at s1	at s2	at s0	at s0	at s1	at s1

Table 6.1: Stratification of pfile03 for Driverlog

Layer transition	Necessary operators	Ordering (least to most constraint)
0th to 1st Layer	Unload, Drive, Disembark	Drivers → Trucks → Packages
1st to 2nd Layer	Load	Driver1 → Packages → Truck → Driver2
2nd to 3rd Layer	Load, Drive	Drivers → Truck → Package
3rd to 4rd Layer	Load, Drive	Packages → Trucks → Drivers

Table 6.2: Variable ordering of pfile03 for Driverlog

and so is the driver. One of the challenges in Driverlog from the planner's perspective is to recognise that a truck will not move until there is a driver in it and packages cannot be delivered without a truck able to pick them up and deliver them. So, when we look at the goals, we see that there is an order in which the problem must be solved: first we need to deliver the packages, then we need to drop the trucks at the right location and last we need to get the drivers to where they belong.

Flaw selection strategy Interestingly, when we examine the orderings constraints derived from our flaw selection heuristic we see an interesting pattern when the planner traverses through the landmark layers. We will not show the actual search progress, but rather comment on the actions which the ADCG heuristic deems necessary to solve the next landmark layer upon solving the current one. The results are depicted in Table 6.2.

As we can see from this table, our flaw selection strategy actually arrives at the same ordering as we would hope for, to get from the goal layer to the first landmark layer, by first solving the most constrained variables (the packages), then place the trucks at the right locations and finally get the drivers to where they need to be. When we look at the next layer where a subset of the packages need to be loaded, we see that the packages variables are now less constrained than the variables for trucks, as they will remain stationary. The next layer is concerned with getting the first subset of packages into the trucks and, due to the drive actions, the priority switches back again. Finally, when we get to the initial state, we see that the constraints ordering is completely opposite from the previous layer, as the

drivers are now the most constrained. This is because we need to get the drivers into the trucks and the trucks at the right locations to pick up the first subset of packages.

Although this is a particularly good example, where this flaw selection strategy works out very well, this need not necessarily always be the case. However, for the examples we have tested, we get consistently good results.

6.1.6 Search process

Having discussed the various heuristics we use in our planner and the method of stratifying the landmark generation graph into *landmark layers*, we will now explain the search process. We did not change the VHPOP searching algorithm, safe for the heuristics and flaw selection strategy as detailed above. The major change is that we now apply VHPOP to smaller subproblems and force it to solve the problem by tackling a sequence of subproblems.

The general outline of how to solve a problem is defined as: given a planning problem $P : \langle A, L, O, B \rangle$, we define the first subproblem as: $P_1 = \langle \{a_\infty, X_1, Y_1\}, \emptyset, \{X_1 \prec Y_1, X_1 \prec a_\infty, Y_1 \prec a_\infty\}, \emptyset \rangle$. To estimate the number of steps that must still be completed from the current *landmark layer* to the initial state, we apply the FF heuristic h_{ff}. Since h_{ff} uses the RPG to derive its heuristic it can account for the possibly inconsistent goal state as it simply accumulates all positive effects. To solve this problem we use VHPOP, but with a couple of changes. Most importantly, whenever a causal link is created from either X_1 or Y_1, we remove all atoms from the corresponding action that refer to the same state variable but assign it a different value. This allows the planner to choose which values amongst disjunctive landmarks should be assigned. To guide the planner, the FF heuristic is recalculated every time values are removed from any action.

Once the subproblem is solved, we move on to the next subproblem. To do this, we first remove the *landmark action* and *minimal landmark layer* and all causal links it supports. Next we insert the next set $\{X_i, Y_i\}$, with the appropriate orderings $\{X_i \prec Y_i, X_1 \prec a_\infty, Y_1 \prec a_\infty\}$ as before. After calculating the FF heuristic the process is repeated until a solution is found for the last subproblem which is the initial state. This solution is the final plan.

One thing we tried to do early on in this project is to allow causal links to persist over layers if the causal link in question was 'deeper linked' than the next *landmark layer*. For example, if a causal link is made between an atom from from the initial layer to the goal layer, we would not break this link when advancing through layers and only remove those causal links which were actually linked to the next *landmark layer*. However, we soon found out that this policy severely drags the planner down as it is forced to stick to a decision early on in the planning process which may prove to be erroneous, but the planner might only find out a number of landmark layers later at which time it will need to spend a severe amount of time to backtracking to resolve this mistake. To avoid this situation we remove all causal links made to any of the actions from the *landmark layer*.

Additionally, when we find a solution for a *landmark layer* which is not the initial state, we allow for unsafe links to persist if such a flaw can be solved by both demoting and promoting the threatening action. This adheres to the least commitment principle because we do not force the planner to make an arbitrary decision but rather leave the question open

until we are forced to resolve it.

6.2 LAMA approach

A more recently adopted approach has been the direction LAMA has taken, not using landmarks directly in the search process as LM^{local} did, but incorporating them into the heuristic function. LAMA takes the approach of counting the number of landmarks that need to be accepted, plus the ones that are required again. In this section we will discuss how we have adapted this method and applied it to guide the heuristic in VHPOP.

The easiest way to integrate counting from LAMA is by counting the number of landmarks that need to be achieved in total and decrease this number every time we make a causal link that supports a previously unsupported landmark. So the heuristic function becomes: $l = n - m$, where n is the total number of landmarks and m is the distinct set of landmarks which have an achiever. This way of incorporating landmarks in the heuristic function, is simplistic as it takes neither the ordering nor the need to reachieve landmarks into account.

LAMA defines a landmark that must be reachieved as: "An accepted landmark is *required again* if it is not true in s and it is the greedy-necessary predecessor of some landmark which is not accepted". In order to calculate this part of the heuristic, LAMA needs to keep track of how landmarks have been achieved, thereby keeping the ordering of the landmarks as part of the heuristic. The latter aspect is harder to build in VHPOP since we are not required to specify any particular ordering between different steps in the plan. However, it is possible to check whether the ordering is satisfied between landmarks by checking if it is possible to impose a certain ordering between steps in a partial plan, or to wait until an ordering constraint is imposed before checking this constraint. In our implementation we decided to go for the latter option and only consider a landmark achieved is all parents are also accepted and explicitly ordered before this landmark. Since we plan for the goal backwards to the initial state, so we consider the landmarks which make up the goal accepted and from there work our way backwards.

To check if a particular landmark needs to be reachieved, we do the following: for every landmark which has been achieved we check if it has any (greedy-)necessary children it depends on. If this is the case we add all these dependencies to a list and, after establishing which landmarks have been achieved, we run a post process that checks whether, for every (greedy-)necessary dependency, there is at least one that is satisfied or whether all of them are satisfied for the greedy-necessary and necessary orderings respectively. so the heuristic function now becomes: $l = n - m + k$, where k is the number of landmarks that need to be reachieved. This function is quite similar to the heuristic used by LAMA and, as with LAMA, we will use this approach on top of the existing heuristic function utilized by VHPOP.

6.3 Additional techniques

In addition to considering how to use landmarks to increase the performance of VHPOP we also experimented with other techniques. In this section we give a short overview of these techniques and how we applied them to VHPOP.

6.3.1 Pruning by looking at (greedy-)necessary landmarks

When looking closely at the definition of (greedy-)necessary landmarks we notice that it defines a set of criteria which must be met before a landmark can be achieved. In other words it limits the number of operators which can be applied to support the given landmark. If we know that every operator sequence where B is (for the first time) added at time i, A is true at time $i-1$, then we can prune all operators which can achieve B which have a different value for the variable associated with A than A. e.g. if we have a necessary landmark $(at\ truck1\ s1) \rightarrow_n (driving\ driver1\ truck1)$, than we can prune any operator which has $(driving\ driver1\ truck1)$ as an effect but does not has $(at\ truck1\ s1)$ as its precondition. Assuming the ordering between landmarks is sound, this can be an effective way of pruning.

Unfortunately, necessary landmark orderings are very rare, greedy-necessary landmarks are more common especially in domains like rovers. However, pruning actions based on greedy-necessary orderings can lead to an incomplete planner as we might prune away necessary actions. In our experiments it turned out that this option sometimes caused slight improvements in planning, but in the majority of the cases it caused the planner to fail to find a solution where it previously did, so we disabled this option from our planner.

6.3.2 Solve earliest flaws first

When we look for open conditions to solve, it might be worthwhile to only consider the earliest set of flaws, i.e. the set of open conditions, per variable, which are all ordered as early as possible according to the landmark generation graph. For example, if we are looking for a way to drop a package at some location, but we have not yet worked out how the package will be picked up, it might be worthwhile to establish this first, before heading to the drop-off location. We found that this improved the planning process in all cases we tested it on.

6.3.3 Prune unsafe states as quick as possible

We noticed that the original planner sometimes allows for the generation of plans in which an unsafe link is unsolvable. Although this unsafe link will eventually be resolved, it might take several refinement steps before the planner realizes it cannot resolve the threat. So whenever we encounter an unsafe link, we first test whether it is directly solvable by either demoting or promoting. Separation is not an option in our case, since we only use grounded actions in our current planner.

6.3.4 Goal hiding

When using landmarks to split the problem into subproblems we experimented with not showing all the goals to the planner *a priori* but rather only letting it solve the goals for which there is no landmark in the landmark action related to the same variable. In doing so, a goal atom g that is a value of state variable v is visible on landmark layer x if $g \notin Y_x^v$. Thus, we force the planner to only work on goals for which the value in the current landmark layer can be directly established because the value is defined in that layer, or there are no disjunctive landmarks between the current layer and the layer where a value is defined. Once a goal has been unveiled to the planner it remains visible from that point on.

We explore this idea because disjunctive landmarks that do not define a variable are usually used to accomplish something else, other than the actual goal values for the variables involved. For example a landmark like *at truck1 s1* $\vee$ *at truck2 s1* is not included to get either truck to the goal state but rather to accomplish another landmark first. Thus, we first focus on the variables for which we have a value defined in the current landmark layer, or know that no intermediate steps are defined in subsequent landmark layers, and experiments have shown that this vastly decreases the search space that we need to investigate.

Chapter 7

Results

Now that we have discussed all our methods for integrating state of the art planning technology in a partial order planning context we will now evaluate these strategies. In order to do so we have taken all the IPC-3 STRIPS domains: Depots, DriverLog, ZenoTravel, Satellite, Rovers, and FreeCell and tested our configurations against the original VHPOP. We have chosen to take the latest possible version of VHPOP, version 3.0 (Beta), released on October 7, 2005. This is the same version we used as our framework. One thing we want to stress is that our current code is not optimized for speed, but rather tests whether the number of visited and generated plans are decreased due to better heuristic guidance. A lot of prototyping went on during the construction of above methods and no time was left to do any optimization of the code. For example our ADCG heuristic uses a dijkstra algorithm where an A* algorithm would be much faster, we calculate the number of landmarks which are achieved or need to be reachieved every time a plan is constructed instead of caching this information and only update it when necessary.

All experiments were run on a Intel Pentium 4 processor running on 3.40GHz with 1GB of memory and we cut of the search after 10 minutes of searching.

7.1 Original VHPOP

We start of by listing the results of the unmodified VHPOP planner with the above configuration. We use the same settings as were used during the IPC-3 competition. The results are listed in tables 7.1, 7.2, 7.3, 7.4, 7.5, and 7.6. We note that VHPOP is quite strong on most domains except the Depots and FreeCell domains where only a couple of solutions could be found by VHPOP.

Problem	Plans visited	Plans generated	Dead ends	#Steps	Time (ms)
pfile01	206	667	24	10	20
pfile02	69336	194537	14105	15	11189
pfile03	-	-	-	-	-
pfile04	-	-	-	-	-
pfile05	-	-	-	-	-
pfile06	-	-	-	-	-
pfile07	66662	283183	12230	25	29094
pfile08	-	-	-	-	-
pfile09	-	-	-	-	-
pfile10	81382	641195	14814	26	40742
pfile11	-	-	-	-	-
pfile12	-	-	-	-	-
pfile13	283927	957550	70261	25	61032
pfile14	-	-	-	-	-
pfile15	-	-	-	-	-
pfile16	704852	5443651	143872	28	467973
pfile17	-	-	-	-	-
pfile18	-	-	-	-	-
pfile19	-	-	-	-	-
pfile20	-	-	-	-	-

Table 7.1: Depots results - VHPOP

7.2 FF approach

We first turn to the FF approach and discuss the results below. We should note however that the current implementation still has some bugs, sometimes it fails to start planning and simply announces that no solution was found. We have yet to track down the source of this bug, to indicate when this happens we denote a * in the result table.

7.2.1 Depots domain

Unfortunately we were unable to solve any of the depots problems.

7.2.2 Driverlog domain

The results of the driverlog domain are listed in table 7.7. When we compare this table to the original VHPOP we might get a little discouraged. As the problems become more difficult the search space considered by the planner blows up. Part of this can be attributed

Problem	Plans visited	Plans generated	Dead ends	#Steps	Time (ms)
pfile01	16	57	0	8	0
pfile02	10950	20533	2122	21	616
pfile03	65	184	65	13	4
pfile04	29432	65673	5562	16	2112
pfile05	541	1547	81	19	40
pfile06	63	253	5	11	8
pfile07	79	357	5	15	16
pfile08	3699	10015	660	25	360
pfile09	241	836	34	27	40
pfile10	142	611	15	18	32
pfile11	12600	47941	2577	23	1312
pfile12	-	-	-	-	-
pfile13	8682	33141	1900	29	1484
pfile14	107311	401125	22731	41	22413
pfile15	3971	27063	784	44	1344
pfile16	-	-	-	-	-
pfile17	-	-	-	-	-
pfile18	-	-	-	-	-
pfile19	-	-	-	-	-
pfile20	-	-	-	-	-

Table 7.2: DriverLog results - VHPOP

to the fact that the broken causal links upon achieving every landmark layer are broken and need to be achieved again, but this is not the main reason why we see these results. When we analyzed the behaviour of our planner and how it traversed through the landmark layers we see a critical flaw occurring at every single instance where more drivers and trucks are available to deliver the packages; Recall that we force the planner to make a decision when faced with a disjunctive landmark, for example when faced with a choice of a number of trucks to pick up a particular package, it has very little information to go on to determine how hard it will be to get a particular truck at that location the same goes for getting a driver into a truck. From the planner's point of view it can make use of any available option from the landmarks which are not in the closed list and is unable to discriminate between them as the rest of the planning problem will only become apparent when advancing to the next layer.

In short the planner is forced at higher layers (i.e. closer to the goal) to make a decision regarding the distribution of resources and how to make use of them, while this might ultimately prove to be a very bad distribution but the planner has no way of knowing this in advance. So while it might decide that driver $d3$ will be driving $truck1$, it might be that in order to get to this state we need to execute a lot of actions while in the initial state there might be a truck which shares $d3$'s location and might be a more suitable choice. A couple of solutions are available to this problem, we come back to them in the conclusions.

Problem	Plans visited	Plans generated	Dead ends	#Steps	Time (ms)
pfile01	6	73	0	1	4
pfile02	1847	15101	407	6	248
pfile03	30	281	2	6	20
pfile04	3149	16581	664	8	412
pfile05	1041	8725	195	12	252
pfile06	1490	16678	282	12	448
pfile07	23513	207784	4512	16	6960
pfile08	7549	52456	1454	13	2160
pfile09	2644	30422	492	23	1664
pfile10	4151	59890	832	24	2748
pfile11	16505	256396	3461	16	8672
pfile12	8951	134851	1797	23	6800
pfile13	33793	350983	6687	27	22969
pfile14	60904	1566157	12091	35	107567
pfile15	223588	5305991	36994	41	558375
pfile16	-	-	-	-	-
pfile17	-	-	-	-	-
pfile18	-	-	-	-	-
pfile19	-	-	-	-	-
pfile20	-	-	-	-	-

Table 7.3: ZenoTravel results - VHPOP

In table 7.8 we see the results when we disable the ADCG heuristic to compute the flaw orderings. We can clearly see that using the ADCG heuristic benefits the overall planning process.

7.2.3 Zeno Travel and Satellite domains

Results are listed in table 7.9 and 7.11 respectively. We see that there is a major hit on the performance compared to the original. Not only do we solve less problems, but the problems we do solve tend to explore a larger search space and give worse plans quality wise. When analyzing the behaviour of the planner we noticed some abnormalities, for the satellite domain we took a look at the 2nd problem file and discovered that the FF heuristic used to calculate the heuristic from the initial state to the landmark layers would sometimes give an inconsistent heuristic value. In this case layer 1 got an heuristic value of four, while layer 2 got an heuristic value of six. This caused the planner to favour plans which were in the 1st layer, rather than plans which already advanced to the second layer.

The other problem we found is that the ADCG algorithm we used to determine the order in which open condition flaws were solved did not always produce better results with these

Problem	Plans visited	Plans generated	Dead ends	#Steps	Time (ms)
pfile01	34	101	3	9	0
pfile02	62	204	7	13	4
pfile03	46	159	3	11	4
pfile04	218	695	36	22	16
pfile05	82	361	9	16	12
pfile06	783	2530	135	21	48
pfile07	165	697	22	23	28
pfile08	166	854	22	26	40
pfile09	1443	6593	276	33	248
pfile10	762	4214	99	32	176
pfile11	129	985	9	33	76
pfile12	1324	8948	215	43	512
pfile13	-	-	-	-	-
pfile14	1126	8303	180	45	480
pfile15	1595	12797	249	54	1168
pfile16	1134	9472	173	47	796
pfile17	-	-	-	-	-
pfile18	582	4716	63	35	240
pfile19	-	-	-	-	-
pfile20	-	-	-	-	-

Table 7.4: Satellite results - VHPOP

domains. The results are listed in tables 7.10 and 7.12. Although it is a very small set of problem we can compare it with it seems that problems from the satellite domain are better suited for this heuristic. Additional research will have to show in which cases the new flaw selection strategy is most useful.

7.2.4 Rovers domain

The rovers domain is quite a nice domain in the sense that a lot of landmarks can easily be found. It is not surprising than, that this approach is able to solve the most problems in this domain as listed in table 7.13. However, again we see that the investigated search space becomes larger with the more difficult problems. Again, when we remove the variable ordering restriction we see better results in some problems and worse in other as listed in table 7.14.

Problem	Plans visited	Plans generated	Dead ends	#Steps	Time (ms)
pfile01	474	678	77	10	12
pfile02	77	113	1	8	4
pfile03	177	241	30	12	4
pfile04	72	107	1	8	8
pfile05	1960	2645	308	23	88
pfile06	807128	1052507	186998	38	84849
pfile07	2315	4420	252	18	164
pfile08	7653	13416	1043	26	724
pfile09	9931	14805	1848	34	880
pfile10	114100	213345	22175	35	15649
pfile11	51148	71259	11642	33	4780
pfile12	5024	10810	720	22	460
pfile13	-	-	-	-	-
pfile14	8492	13842	1479	30	980
pfile15	-	-	-	-	-
pfile16	-	-	-	-	-
pfile17	18509	32703	3399	51	4672
pfile18	-	-	-	-	-
pfile19	-	-	-	-	-
pfile20	-	-	-	-	-

Table 7.5: Rovers results - VHPOP

7.2.5 FreeCell domain

Unfortunately we were unable to produce any results on the free cell domain due to the aforementioned bug.

7.3 LAMA approach

Having discussed the FF approach we now turn to the LAMA approach which uses the landmark information as part of the heuristic.

7.3.1 Depots domain

The depots domain is a hard one to tackle for VHPOP and while the FF approach did not produce any results we see that the LAMA approach does better. In these results we see that it visits and generates significantly less plans for all problems and runs into far less dead ends. Surprisingly we also see that the plan quality never degrates and in some cases even

Problem	Plans visited	Plans generated	Dead ends	#Steps	Time (ms)
pfile01	7216	52842	1526	11	2716
pfile02	-	-	-	-	-
pfile03	-	-	-	-	-
pfile04	-	-	-	-	-
pfile05	-	-	-	-	-
pfile06	-	-	-	-	-
pfile07	-	-	-	-	-
pfile08	-	-	-	-	-
pfile09	-	-	-	-	-
pfile10	-	-	-	-	-
pfile11	-	-	-	-	-
pfile12	-	-	-	-	-
pfile13	-	-	-	-	-
pfile14	-	-	-	-	-
pfile15	-	-	-	-	-
pfile16	-	-	-	-	-
pfile17	-	-	-	-	-
pfile18	-	-	-	-	-
pfile19	-	-	-	-	-
pfile20	-	-	-	-	-

Table 7.6: FreeCell results - VHPOP

gets better than the original. While we were not able to solve pfile16 we were able to tackle pfile17. When we inspect the reason for the failure to solve pfile16, we see that our program is only able to inspect 40% of the plans the original VHPOP inspected before running out of memory.

7.3.2 Driverlog domain

The results for the driverlog domain are depicted in table 7.16. When we compare these results we notice that our approach is better in nine problems and the original one in four (excluding pfile14) so our approach does a little better although the original VHPOP was able to solve pfile14 while we were not. We note that the plan quality does in general remains the same, in some cases the plan quality degrades quite badly (e.g. pfile10).

7.3.3 Zeno Travel domain

The results for the zeno travel domain are listed in table 7.17. Unfortunately we were not able to solve all problems the original VHPOP was able to handle. While advantage is

Problem	Plans visited	Plans generated	Dead ends	#Steps	Time (ms)
pfile01	66	195	2	8	8
pfile02	281130	534563	3505	23	51483
pfile03	94	307	0	16	32
pfile04	109958	226141	2147	19	11381
pfile05	5181	16487	0	21	1650
pfile06	48	213	0	12	24
pfile07	-	-	-	-	-
pfile08	-	-	-	-	-
pfile09	-	-	-	-	-
pfile10	-	-	-	-	-
pfile11	-	-	-	-	-
pfile12	-	-	-	-	-
pfile13	-	-	-	-	-
pfile14	-	-	-	-	-
pfile15	-	-	-	-	-
pfile16	-	-	-	-	-
pfile17	-	-	-	-	-
pfile18	-	-	-	-	-
pfile19	-	-	-	-	-
pfile20	-	-	-	-	-

Table 7.7: Driverlog results - FF approach

Problem	Plans visited	Plans generated	Dead ends	#Steps	Time (ms)
pfile01	259	690	0	8	40
pfile02	-	-	-	-	-
pfile03	477	1319	3	16	100
pfile04	388307	755132	7230	20	65176
pfile05	10134	29476	40	21	2944
pfile06	49	213	0	12	20

Table 7.8: Driverlog results (variable ordering disabled) - FF approach

Problem	Plans visited	Plans generated	Dead ends	#Steps	Time (ms)
pfile01	6	61	0	2	8
pfile02	19723	133146	760	10	2572
pfile03	2034	15650	8	11	556
pfile04	4582	31240	107	14	992
pfile05	-	-	-	-	-
pfile06	-	-	-	-	-
pfile07	-	-	-	-	-
pfile08	-	-	-	-	-
pfile09	-	-	-	-	-
pfile10	-	-	-	-	-
pfile11	-	-	-	-	-
pfile12	-	-	-	-	-
pfile13	-	-	-	-	-
pfile14	-	-	-	-	-
pfile15	-	-	-	-	-
pfile16	-	-	-	-	-
pfile17	-	-	-	-	-
pfile18	-	-	-	-	-
pfile19	-	-	-	-	-
pfile20	-	-	-	-	-

Table 7.9: ZenoTravel results - FF approach

Problem	Plans visited	Plans generated	Dead ends	#Steps	Time (ms)
pfile01	6	61	0	2	12
pfile02	15572	133142	210	10	3280
pfile03	1354	12952	0	11	576
pfile04	6735	46665	057	15	2368

Table 7.10: ZenoTravel results (variable ordering disabled) - FF approach

gained in the few cases, when we compare the number of plans generated at the expense of plan quality, that were solved. Overall we would have to conclude that this approach does not benefit VHPOP in the Zeno Travel domain.

7.3.4 Satellite domain

The results for the satellite domain are depicted in table 7.18. These results are more encouraging than the Zeno Travel domain, interestingly we were able to solve a problem which the original VHPOP could not solve while on the other hand we were not able to solve two problems the original one did. Overall the results favour the LAMA approach as it gener-

Problem	Plans visited	Plans generated	Dead ends	#Steps	Time (ms)
pfile01	*	*	*	*	*
pfile02	-	-	-	-	-
pfile03	1454	4164	72	17	236
pfile04	-	-	-	-	-
pfile05	4313	19488	0	22	864
pfile06	-	-	-	-	-
pfile07	-	-	-	-	-
pfile08	-	-	-	-	-
pfile09	-	-	-	-	-
pfile10	-	-	-	-	-
pfile11	-	-	-	-	-
pfile12	-	-	-	-	-
pfile13	-	-	-	-	-
pfile14	5407	45869	2	50	9360
pfile15	-	-	-	-	-
pfile16	16453	115749	35	53	11529
pfile17	17887	121224	132	47	10649
pfile18	-	-	-	-	-
pfile19	-	-	-	-	-
pfile20	-	-	-	-	-

Table 7.11: Satellite results - FF approach

Problem	Plans visited	Plans generated	Dead ends	#Steps	Time (ms)
pfile03	1707	4159	180	17	228
pfile05	4247	19432	0	22	1056
pfile14	-	-	-	-	-
pfile16	13783	111106	113	52	9832
pfile17	19227	127870	132	47	13481

Table 7.12: Satellite results (variable ordering disabled) - FF approach

Problem	Plans visited	Plans generated	Dead ends	#Steps	Time (ms)
pfile01	214	336	0	11	12
pfile02	72	152	0	9	8
pfile03	632	829	10	16	56
pfile04	416	582	2	14	28
pfile05	385	674	0	27	56
pfile06	76986	136955	322	41	19273
pfile07	68728	88376	1468	30	6784
pfile08	-	-	-	-	-
pfile09	-	-	-	-	-
pfile10	-	-	-	-	-
pfile11	-	-	-	-	-
pfile12	3876	6809	39	24	1472
pfile13	-	-	-	-	-
pfile14	-	-	-	-	-
pfile15	-	-	-	-	-
pfile16	-	-	-	-	-
pfile17	-	-	-	-	-
pfile18	-	-	-	-	-
pfile19	-	-	-	-	-
pfile20	-	-	-	-	-

Table 7.13: Rovers results - FF approach

Problem	Plans visited	Plans generated	Dead ends	#Steps	Time (ms)
pfile01	155	235	0	11	8
pfile02	72	151	0	9	8
pfile03	356	485	4	16	36
pfile04	275	389	2	12	24
pfile05	422	719	0	23	56
pfile06	64693	102840	283	41	11485
pfile07	1066296	1481439	6038	30	104598
pfile12	87305	162150	1203	24	30654

Table 7.14: Rovers results (variable ordering disabled) - FF approach

Problem	Plans visited	Plans generated	Dead ends	#Steps	Time (ms)
pfile01	74	268	4	10	16
pfile02	17933	54949	1391	15	7324
pfile03	-	-	-	-	-
pfile04	-	-	-	-	-
pfile05	-	-	-	-	-
pfile06	-	-	-	-	-
pfile07	19385	87625	644	21	10857
pfile08	-	-	-	-	-
pfile09	-	-	-	-	-
pfile10	12554	93609	688	24	11109
pfile11	-	-	-	-	-
pfile12	-	-	-	-	-
pfile13	25651	108797	2477	25	16961
pfile14	-	-	-	-	-
pfile15	-	-	-	-	-
pfile16	-	-	-	-	-
pfile17	40833	222170	2883	27	57907
pfile18	-	-	-	-	-
pfile19	-	-	-	-	-
pfile20	-	-	-	-	-

Table 7.15: Depots results - LAMA approach

ates less plans to find a solution in eleven of the fourteen problems both planners solve and without sacrificing plan quality for most of these.

7.3.5 Rovers domain

The results for the rovers domain are listed in table 7.19. From the rovers domain we are able to extract a lot of landmarks, while in the FF approach we already saw some good progress with the LAMA approach we are able to generate results for all problem files in this domain. Although the quality of the generated plans do take a hit, the performance comparing the inspected search space is a lot better than the original VHPOP implementation for every single problem.

7.3.6 FreeCell domain

The results for the free cell domain are listed in table 7.20. Although we cannot say to much based on a single result, it seems that greedily trying to satisfy landmarks does not work in our favour in this particular domain.

Problem	Plans visited	Plans generated	Dead ends	#Steps	Time (ms)
pfile01	25	74	0	7	0
pfile02	9816	20091	190	21	1052
pfile03	47	148	0	13	8
pfile04	15433	37805	661	20	1908
pfile05	559	1527	4	18	84
pfile06	392	1310	2	11	60
pfile07	58	252	0	15	16
pfile08	2025	5934	44	25	492
pfile09	149	612	0	27	48
pfile10	755	2997	2	29	200
pfile11	41884	181885	999	23	8908
pfile12	-	-	-	-	-
pfile13	3635	17044	90	29	1500
pfile14	-	-	-	-	-
pfile15	3127	26423	9	47	3192
pfile16	-	-	-	-	-
pfile17	-	-	-	-	-
pfile18	-	-	-	-	-
pfile19	-	-	-	-	-
pfile20	-	-	-	-	-

Table 7.16: DriverLog results -LAMA approach

Problem	Plans visited	Plans generated	Dead ends	#Steps	Time (ms)
pfile01	6	73	0	1	8
pfile02	1598	14453	8	6	328
pfile03	28	279	1	6	20
pfile04	6429	32965	591	9	1080
pfile05	800	6912	0	12	260
pfile06	813	9694	9	15	456
pfile07	-	-	-	-	-
pfile08	-	-	-	-	-
pfile09	6501	80226	311	24	6060
pfile10	3424	48773	227	27	3416
pfile11	-	-	-	-	-
pfile12	-	-	-	-	-
pfile13	-	-	-	-	-
pfile14	50945	1598721	973	35	103762
pfile15	-	-	-	-	-
pfile16	-	-	-	-	-
pfile17	-	-	-	-	-
pfile18	-	-	-	-	-
pfile19	-	-	-	-	-
pfile20	-	-	-	-	-

Table 7.17: ZenoTravel results - LAMA approach

Problem	Plans visited	Plans generated	Dead ends	#Steps	Time (ms)
pfile01	31	98	0	9	0
pfile02	55	197	0	13	4
pfile03	43	156	0	11	8
pfile04	8190	2382	35	22	1016
pfile05	73	351	0	16	16
pfile06	753	2511	14	21	200
pfile07	580	2472	2	26	164
pfile08	131	736	0	26	100
pfile09	698	4032	1	31	552
pfile10	489	2988	0	34	328
pfile11	-	-	-	-	-
pfile12	1153	8886	10	43	1740
pfile13	1698	17022	2	60	9372
pfile14	1008	8297	1	45	780
pfile15	905	8893	6	51	1408
pfile16	997	9452	1	47	996
pfile17	-	-	-	-	-
pfile18	-	-	-	-	-
pfile19	-	-	-	-	-
pfile20	-	-	-	-	-

Table 7.18: Satellite results - LAMA approach

Problem	Plans visited	Plans generated	Dead ends	#Steps	Time (ms)
pfile01	105	179	1	12	4
pfile02	78	114	0	8	4
pfile03	100	156	0	14	8
pfile04	62	92	0	8	8
pfile05	394	537	7	25	44
pfile06	2957	4305	112	42	532
pfile07	216	337	0	18	28
pfile08	409	635	1	29	96
pfile09	3717	5619	50	38	772
pfile10	1793	2928	19	37	436
pfile11	3820	5639	112	37	888
pfile12	266	496	0	25	92
pfile13	9917	16883	141	52	3932
pfile14	2551	4333	46	35	664
pfile15	3474	6659	61	43	1136
pfile16	4870	4870	74	46	2160
pfile17	12875	21417	149	59	5016
pfile18	5592	12362	63	47	3872
pfile19	14682	32557	157	73	15545
pfile20	67907	144963	990	99	131108

Table 7.19: Rovers results - LAMA approach

Problem	Plans visited	Plans generated	Dead ends	#Steps	Time (ms)
pfile01	97700	569256	13379	12	55471
pfile02	-	-	-	-	-
pfile03	-	-	-	-	-
pfile04	-	-	-	-	-
pfile05	-	-	-	-	-
pfile06	-	-	-	-	-
pfile07	-	-	-	-	-
pfile08	-	-	-	-	-
pfile09	-	-	-	-	-
pfile10	-	-	-	-	-
pfile11	-	-	-	-	-
pfile12	-	-	-	-	-
pfile13	-	-	-	-	-
pfile14	-	-	-	-	-
pfile15	-	-	-	-	-
pfile16	-	-	-	-	-
pfile17	-	-	-	-	-
pfile18	-	-	-	-	-
pfile19	-	-	-	-	-
pfile20	-	-	-	-	-

Table 7.20: FreeCell results - LAMA approach

Chapter 8

Conclusion and future work

We set out to improve the competitiveness of partial order planning by modifying the latest version of VHPOP and include planning techniques developed over the last decade for state-space planning. As discussed before, one of the fundamental weaknesses of partial order planners is not having an explicit state representation which limits the informativeness of heuristics which have been developed until now. In this work we tried to develop better heuristics by exploiting landmark information. We have explored two different ways to exploited landmarks, firstly we tried to change the search process by splitting the problem into consecutive subproblems and secondly by integrating them into the heuristics. On top of that we developed a novel flaw selection strategy which exploits the CG and checks through the ADCG heuristic which actions are relevant and utilize this information to do more informed cycle breaking than the original CG heuristic. This allows us to do apply a more informed flaw selection strategy and gain better performances in most domains when applying the FF approach.

But whereas previous attempts (RePOP and VHPOP) were able to book significant improvements, our experimental results do not suffice to make the same claim. However, we were able to make improvements over VHPOP with the LAMA approach and while the FF approach did not yield better results we did show the effectiveness of our new flaw selection strategy. Furthermore, we do believe that this work can be extended and has a lot of scope to put partial order planning back on the map as we discuss in this section.

8.1 FF approach

We note that the results for our first approach, splitting up the problem, does not yield very good results. When compared with STeLLa, we notice that we achieve a worse performance even though we use a sound ordering between the landmarks where STeLLa does not[38]. Part of this can be attributed to the fact that, upon achieving each landmark layer, some causal links are broken and need to be achieved again, but this is not the main reason why we see these results. When we analyze the behavior of our planner and how it traverses through the landmark layers we see a critical flaw occurring in every single instance where more resources are available to accomplish a task, e.g. multiple drivers and trucks are

available to deliver the packages; Recall that we force the planner to make a decision when faced with a disjunctive landmark. So, for example, when faced with a number of trucks to pick up a particular package, the planner has very little information to use to determine how hard it will be to get a particular truck to that location; the same applies for getting a driver into a truck. From the planner's point of view, it can make use of any available option from the landmarks which are not in the closed list and it is unable to discriminate between them, as the rest of the planning problem will only become apparent when advancing to the next layer. The only guidance is provided by the change in the h_{ff} heuristic. In short the planner is forced at higher layers (i.e. closer to the goal) to make a decision regarding the distribution of resources and how to make use of them, when there is too little information to guide its search and poor choices can lead to very poor performance.

8.1.1 Lifted representation v.s. grounded actions

One of the lines of research we are pursuing is to allow disjunctive bindings of variables, allowing the planner to avoid making a premature decision. As we have seen with the rovers and the driverlog domains, any planning domain where multiple resources could be used to satisfy a property (e.g. multiple alternative trucks to deliver a package) our planner commits itself to a disribution of resources with little to no heuristic guidance. Only at a later landmark layer, when the planner has already committed to a distribution of the resources, will it be able to assess whether it has chosen a good distribution. Although most problems we have considered do not have any dead ends, the planner will go back and forth between the different landmark layers as it tries other distributions of the resources. In effect, the more resources there are, the larger the set of combinations and hence the harder it is for the planner to actually solve the problem.

Another problem we encountered was that the heuristic values between successive landmarks are not always monotonically decreasing which forces the planner, upon reaching a new landmark layer, to fall back immediately to a partial plan that tries to achieve the previous landmark layer, until all partial plans become big enough to dominate the worse heuristic value of the new landmark layer.

Possible ways to extend on this is either by changing the search algorithm and do not allow the planner to fall back on a previous landmark layer. This could be seen as an adaptation of enforced hill-climbing[21] but on a coarser level. However this approach will loose the completeness property as we cannot be sure that a partial plan found for a landmark layer can be refined to find a solution to the next. Opting for this solution would however solve another problem we have encountered with inconsistent heuristics from the landmark layers to the initial state. For some problems we found that the heuristic value could be higher for a landmark layer which is supposed to be closer to the initial state. When we encountered such a situation the planner would, upon reaching a landmark layer with an higher heuristic function than the previous landmark layer, immediately fall back to the previous layer until all partial plans become big enough to overcome the worse heuristic value.

One way to handle the decision making when encountering a disjunctive landmark is to allow the planner to make partial bindings to variables. So instead of deciding, for example,

which driver should drive which truck in the Driverlog domain, we allow the planner to produce a partial plan that contains the action *driving truck1 var*$_1$, where *var*$_1$ = *driver1* $\vee$ *driver2* $\vee \ldots$ *driver*$_n$. This will require us to use lifted actions. Both VHPOP and RePOP make use of grounded actions in order to be competitive with state-space planners but we believe that dealing with variable binding constraints can have many advantages in the work we pursue. With the use of disjunctive landmarks we can allow the planner to reason about 'move truck1 from s1 to s2 and use any of these drivers ...' or 'deliver package1 with any of these trucks ... driven by any of these drivers ...'. Not only is this form of planning more intuitive but also adheres to the least-commitment principle and — we believe — will revive the importance of partial order planning, as it will be able to cope with larger problem instances. A study of RealPlan[41] shows that most planners, paradoxically, have more trouble finding a solution when given more resources. This is partly because most state-space planners ground all actions prior to planning, which can take up quite some time, but also because they tend to explore all possible actions from the current state.

We hope that by using a lifted representation we can reason on a more abstract level about planning problems and delay making commitments about resource usage until we have more information available and are able to make a more informed decision than we are currently able to.

8.1.2 STeLLa

As we have seen, STeLLa uses a very similar approach in their first version of this planner. Judging from their paper[39], they were able to come up with better results than we could manage. The major difference is that they restrict themselves to non-disjunctive landmarks and use LM^{RPG} to derive their landmarks. Also they use additional interference to deal with inconsistent literals in a subproblem. Preliminary work to adopt the same type of interference in our work did not show any advantage, but it would definitely be interesting to see what would happen if we would only concentrate on non-disjunctive landmarks and check if the landmarks derived by LAMA show similar results.

8.2 LAMA approach

When using the landmarks solely for heuristic guidance we see that greedily trying to reach the landmarks, in the appropriate order, gives better results in most problems we have considered in this work (with the exception of ZenoTravel and FreeCell). However, the gains are not substantial enough to compete with the latest generation of state-space planners. It remains a fundamental difficulty in partial order planners to find good heuristics to guide search, since we lack an explicit state representation. That is why we believe that the best way to move the field of partial order planning forward is the approach outlined above and use the lifted representation to plan on a more abstract level and postpone decision-making about variable bindings as long as possible.

8.3 Closing remarks

Apart from adjusting above approaches one aspect we have not covered yet in this section is the usage of the original heuristics and flaw selection strategies. Although we devised a new flaw selection strategy which can be used in addition to the existing ones, we kept the original heuristics and flaw selections strategies in place. Although the authors have tried to optimize these heuristics for this particular benchmark set, it need not be the most advantageous setup for our techniques. So subsequent research in this area might help to get rid some of the weaknesses outlined above. We should, however, not expect the heuristics to solve all our problems as more fundamental changes need to be made to gain significant improvements[20].

We do not think that major progress is going to be made by solely using ground actions and foregoing lifted actions. While utilizing techniques developed for state-space planners has proven beneficial for partial order planning it has not been able to put partial order planning on an equal footing with state-space planning. The last planning competition where a partial order planner competed was in IPC-3 (VHPOP) and there it was outperformed by state-space planners like LPG. To gain speed RePOP and VHPOP resorted to using only grounded actions, and while Younes and Simmons[44] argue that using grounded actions reduces the amount of actions which need to be explored we think that the potential benefit of using a lifted representation has not been explored by either VHPOP nor RePOP. We think that using a lifted representation in combination with disjunctive variable bindings will allow partial order planners to gain an advantage in larger and resource rich domains and hopefully give partial order planning a competitive edge over current state-space planners.

Bibliography

[1] Fahiem Bacchus, Froduald Kabanza, and Universite De Sherbrooke. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116:2000, 1999.

[2] K. Bell, A. I. Coles, M. Fox, D. Long, and A. J. Smith. The application of planning to power substation voltage control. In *ICAPS Workshop on Scheduling and Planning Applications (SPARK)*, 2008.

[3] Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:1636–1642, 1995.

[4] Blai Bonet and Hctor Geffner. Heuristic search planner 2.0. *AI Magazine*, 22:77–80, 2001.

[5] Blai Bonet and Hctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129:5–33, 2001.

[6] Tom Bylander. The computational complexity of propositional strips planning. *Artificial Intelligence*, 69:165–204, 1994.

[7] Yixin Chen, Benjamin W. Wah, and Chih-Wei Hsu. Temporal planning using subgoal partitioning and resolution in sgplan. *J. Artif. Intell. Res. (JAIR)*, 26:323–369, 2006.

[8] Yixin Chen, You Xu, and Guohui Yao. Stratified planning. In Craig Boutilier, editor, *IJCAI*, pages 1665–1670, 2009.

[9] A. I. Coles, M. Fox, D. Long, and A. J. Smith. Planning with respect to an existing schedule of events. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 07)*, September 2007.

[10] Ken Currie, Austin Tate, and South Bridge. o-plan: the open planning architecture, 1990.

[11] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artif. Intell.*, 49(1-3):61–95, 1991.

[12] Richard Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. In *IJCAI*, pages 608–620, 1971.

[13] Maria Fox and Derek Long. The automatic inference of state invariants in tim. *J. Artif. Intell. Res. (JAIR)*, 9:367–421, 1998.

[14] Alfonso Gerevini, Alessandro Saetti, and Ivan Serina. Planning through stochastic local search and temporal action graphs in lpg. *J. Artif. Int. Res.*, 20(1):239–290, 2003.

[15] Alfonso Gerevini and Lenhart Schubert. Accelerating partial-order planners: Some techniques for effective search control and pruning. *J. Artif. Intell. Res. (JAIR)*, 5:95–137, 1996.

[16] Malik Ghallab, Ecole Nationale, Constructions Aeronautiques, Craig K. Isi, Scott Penberthy, David E. Smith, Ying Sun, and Daniel Weld. Pddl - the planning domain definition language. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[17] Malte Helmert. The fast downward planning system. *J. Artif. Intell. Res. (JAIR)*, 26:191–246, 2006.

[18] Malte Helmert. Concise finite-domain representations for pddl planning tasks. *Artif. Intell.*, 173(5-6):503–535, 2009.

[19] Malte Helmert, Robert Mattmller, and Albert ludwigs-universitt Freiburg. Accuracy of admissible heuristic functions in selected planning domains. In *In AAAI. (Extended abstract in the ICAPS07 workshops*, 2007.

[20] Malte Helmert and Gabriele Röger. How good is almost perfect? In *AAAI'08: Proceedings of the 23rd national conference on Artificial intelligence*, pages 944–949. AAAI Press, 2008.

[21] Jrg Hoffmann. Ff: The fast-forward planning system. *AI Magazine*, 22:57–62, 2001.

[22] Jrg Hoffmann, Julie Porteous, and Laura Sebastia. Ordered landmarks in planning. *J. Artif. Intell. Res. (JAIR)*, 22:215–278, 2004.

[23] Peter Jonsson and Christer Bckstrm. State-variable planning under structural restrictions: Algorithms and complexity. *Artificial Intelligence*, 100:125–176, 1998.

[24] Jana Koehler and Jrg Hoffmann. On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm, 2000.

[25] Hui Li and Brian Williams. Generative planning for hybrid systems based on flow-tubes. In *In proceedings of ICAPS-08*, 2008.

[26] Derek Long and Maria Fox. The 3rd international planning competition: Results and analysis. *J. Artif. Intell. Res. (JAIR)*, 20:1–59, 2003.

[27] Hector Geffner Malte Helmert. The causal graph heuristic is the additive heuristic plus context. *AAAI*, 2008.

[28] Smith D. Meuleau N., Plaunt C. Emergency landing planning for damaged aircraft. *ICAPS-08 Scheduling and Planning Applications Workshop (SPARK)*, 2008.

[29] Dana Nau, Yue Cao, Amnon Lotem, and Hector Munoz-Avila. Shop: Simple hierarchical ordered planner. Technical report, Department of Computer Science and Institute for Systems Research, University of Maryland, 1999.

[30] Dana Nau, Malik Ghallab, and Paolo Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[31] Xuanlong Nguyen and Subbarao Kambhampati. Reviving partial order planning, 2001.

[32] P. D. Pednault, Edwin. Adl: exploring the middle ground between strips and the situation calculus. In *Proceedings of the first international conference on Principles of knowledge representation and reasoning*, pages 324–332, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[33] J. Scott Penberthy and Daniel S. Weld. Ucpop: A sound, complete, partial order planner for adl, 1992.

[34] Martha E. Pollack, David Joslin, and Massimo Paolucci. Flaw selection strategies for partial-order planning. *J. Artif. Intell. Res. (JAIR)*, 6:223–262, 1997.

[35] Julie Porteous and Laura Sebastia. Extracting and ordering landmarks for planning. *J. Artif. Intell. Res. (JAIR)*, 22:2004, 2000.

[36] Silvia Richter, Malte Helmert, and Matthias Westphal. Landmarks revisited. In *AAAI*, pages 975–982. AAAI Press, 2008.

[37] Silvia Richter and Matthias Westphal. The lama planner. In *ICAPS-06*, 2009.

[38] Laura Sebastia, Eva Onaindia, and Eliseo Marzal. Stella: An optimal sequential and parallel planner. In Pavel Brazdil and Alípio Jorge, editors, *EPIA*, volume 2258 of *Lecture Notes in Computer Science*, pages 409–416. Springer, 2001.

[39] Laura Sebastia, Eva Onaindia, and Eliseo Marzal. Decomposition of planning problems. *AI Commun.*, 19(1):49–81, 2006.

[40] David E. Smith, Jeremy Frank, and Ari K. Jónsson. Bridging the gap between planning and scheduling. *Knowl. Eng. Rev.*, 15(1):47–83, 2000.

[41] Biplav Srivastava. Realplan: Decoupling causal and resource reasoning in planning. In *In AAAI/IAAI*, pages 812–818. AAAI/MIT Press, 2000.

[42] David E. Wilkins. *Practical planning: extending the classical AI planning paradigm*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[43] Hakan L. S. Younes and Reid G. Simmons. Vhpop: Versatile heuristic partial order planner. *J. Artif. Intell. Res. (JAIR)*, 20:405–430, 2003.

[44] Hakan L.S. Younes and Reid G. Simmons. On the role of ground actions in refinement planning, 2002.