Planning with Preferences, Specification and Search

by

Christopher Schmidt

B.Sc., University of Wisconsin, 2002

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Summer 2016

Approval

Name:	**Christopher Schmidt**
Degree:	**Doctor of Philosophy (Computing Science)**
Title:	***Planning with Preferences, Specification and Search***
Examining Committee:	**Chair:** Dr. Janice Regan Senior Lecturer

Dr. James P. Delgrande
Senior Supervisor
Professor _____

Dr. Fred Popowich
Supervisor
Professor _____

Dr. Francis Jeffry Pelletier
Internal Examiner
Emeritus Professor
Department of Linguistics _____

Dr. Abhaya Nayak
External Examiner
Associate Professor
Department of Computing
Macquarie Univeristy _____

Date Defended: 12 August 2016

Abstract

Planning is one of the fundamental problems of artificial intelligence. A classic planning problem consists of an initial world state, a set of goal conditions, and a set of actions. The solution to the problem is a sequence of actions, a plan, that when applied to the initial state, leads to a final goal state that satisfies all goal conditions. In many cases, it makes sense to add preferences to a planning problem. A preference can be viewed as a 'soft' goal. Ideally, all preferences will be satisfied by a plan, but this is not necessary, and not necessarily possible. There are two basic components to planning with preferences, specifying the preferences and the search for a high quality plan. For the former, we present a rich language for specifying preferences for planning problems. For the latter, we use heuristic guided search, a successful approach to classic planning, for planning with preferences. Landmark based heuristics have been successful in classic planning, so we examine how they can be used for preferences. Finally, we present an adaptation to the greedy best-first search algorithm, Cascading Search, that diversifies the search space and examine how effectively it speeds the search process and ensure discovery of quality plans.

Keywords: Planning, Preferences, Landmarks, Heuristic, Search

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Automated planning, or simply planning, has been an important area of research in Artificial Intelligence for decades. In short, a planning problem includes a description of a starting situation, a goal, and a set of possible actions. To solve the problem, a planner must generate a sequence of actions, a plan, that when performed in the specified order from the starting situation leads to the goal being fulfilled. As the number of available actions grows, the number of possible plans grows exponentially, creating a vast number of possible plans. Research in planning focuses on finding efficient ways to find a high quality plan from within all of the possibilities.

Our work is on planning problems that involve preferences. In most planning, the goal conditions are hard constraints that must be satisfied, but an often more realistic approach to planning is to also consider preferences. These preference conditions don't have to be met for a plan to be successful, but set criteria a plan will ideally meet. In many problems, there is a large set of plans that satisfy the goal conditions. Preferences are a way to compare the quality of these plans. The aim of the planner is not to just satisfy the goal, but to satisfy the preferences to the highest degree possible.

One planning domain that lends itself to preferences is the planning of a vacation. The starting situation simply has the traveler at home. The goal could be as simple as creating a plan that visits a new city, visits museums while there, and then returns home. Preferences might include what cities are best to visit, how to travel, what sort of accommodations to stay in, what activities to partake in, etc. In this case the hard goal is simple, leading to a wide variety of plans. The preferences are used to differentiate the numerous plans.

Other problems where preferences are useful occur when goals may be too restrictive and prevent any solution from being found. A portion of goals that are not absolutely necessary can be converted to preferences. Ideally, a plan that satisfies all goals can be found, but otherwise the plans that come closest to completing all of the original goals can be found instead of failing to produce any plan.

1.1 Overview

At a high level, planning with preferences can be split into two pieces. First, a system for specifying a preferences planning problem must be chosen. The specification is more complex than for a standard planning problem because preferences require not just describing aspects of preferred plans, but a means for comparing plans. This can be as straightforward as assigning each preference a numeric value and scoring a plan by summing the value of all satisfied preferences. However, this may not accurately reflect some preferences. More complex approaches that can more accurately model a wide range of preferences are often more difficult to utilize.

Once a preferences planning problem is in hand, the second component needed is a program to efficiently search for desirable plans. It is straightforward to create one that simply checks every possible sequence of actions of increasing lengths that is guaranteed to find the optimal plan given enough computing resources. This is effective if the problem is easy enough, but for more difficult problems the best approach is to attempt to find a high quality plan that is not guaranteed to be optimal, but will probably be nearly as good as the optimal plan.

We address both of these areas of planning with preferences. We will set forth a novel system for specifying preferences for a planning problem. Then we take a successful strategy for standard planning, landmarks, and adapt it for planning with preferences. Finally, we will present a new search algorithm that is an effective adaptation of the basic greedy best-first search algorithm.

1.1.1 Specifying Preferences

A language for specifying preferences needs to accomplish two things. First, a user must use the language to describe characteristics that are relevant to the quality of the plan. If planning a vacation, one such characteristic might be whether an art museum was visited in the plan, whether travel was done by plane, or whether the destination is a city by the ocean. Once these relevant characteristics have been defined, most systems allow a user to then specify the relative importance of these characteristics. It may be preferable to travel by train, but it might be more important to stay in a city by the ocean.

Defining Characteristics

When choosing a language for adding preferences to a planning problem, there are two fundamental approaches. Either the language will only be concerned with the situation after the plan has executed or it will also be concerned with the changing situation throughout execution of the plan. Depending on the application, either choice can be appropriate. If planning a vacation, the current situation at any point during the plan is important to a

traveler. However, if planning how to schedule tasks among workers, the only concern might be with what tasks have been completed at the end of day.

Preference languages use well established logics for their purpose. The simplest option is propositional logic, wherein the world is described by a series of variables that can either be true or false. Formulas combine these variables with the standard operators, *and*, *or*, and *not*. First order logic can be used to specify more complex concepts in a compact manner. Temporal logic is an option to make characteristics concerning an entire plan easier to write. While temporal and first order logic can express many concepts more succinctly, everything that can be expressed in one logic can be expressed in the others as we are working with finite worlds.

Assigning Relative Value to Characteristics

Once a set of characteristics is in hand, their importance needs to be defined. There are preference systems that just assume all of the characteristics are of the same importance, but these are generally only the simplest approaches. Once the importance has been specified, plans can be compared. There are numerous strategies for doing so and numerous ways they can be classified [67].

A set of preferences can be applied by methods that assign each plan a numeric value or give a function that given two plans will determine if one is more preferred than the other. Assigning every characteristic a value and summing the values of all of them that a plan meets is a simple way to assign a value. Thus given two plans, their values can be compared to see which is preferable. Any two can be compared with the result being that either one is preferable or they are equally preferable. This numeric approach and other functions that can compare any two plans are useful because it makes it straightforward to compare plans and determine which is most preferable.

However, it is not always possible to compare two plans in a meaningful way given a set of preferences. For example, someone might prefer that if they are in Toronto on their vacation, that they visit the CN Tower and if they visit New York, that they visit the Empire State Building. But, they might have no preference as to whether they visit Toronto or New York. Assuming these are the only preferences, it is easy to compare plans that involve visiting Toronto with each other and plans that visit New York with each other, but how should plans that only visit Toronto compare to plans that only visit New York? In cases like this, it can make sense to use a system in which not all plans are comparable. This can more accurately model preferences, but can make it more difficult to determine the best plan.

1.1.2 Finding Preferred Plans

Methods for finding preferred plans are strategies and algorithms that have been used in classical planning that have been adapted for this area of research. The main issue specific to planning with preferences is how to balance a search toward the goals while simultaneously striving to satisfy the preferences as best possible.

One strategy involves adapting planning problems into different kinds of logical problems that have been well researched. Once the conversion is complete, all that is needed is to give the new problem to a program that solves that particular type of problem. Problem types that have been used include Answer Set Programming [70], Constraint Satisfaction [30], and satisfiability problems [63].

The method that our work utilizes is planning as search. The search can address the planning problem in different ways. In forward-chaining search, a planner starts at the initial state and builds a plan from start to finish by examining longer and longer sequences of actions. In backward-chaining search, a planner begins at the goal working from the end of the plan back towards the start. A third approach is partial order planning in which actions are found that lead to parts of the goal. When these actions do not conflict with one another, they may not be given a specific order in which to execute them, hence, partial order planning.

In recent years, forward-chaining search has been the most successful and it is this we focus on. Forward-chaining planners use heuristics to estimate how far a partial plan is from achieving the goal [13]. The planner then searches by expanding the most promising candidates. An example of a simple heuristic is to simply count how many parts of the goal have not yet been achieved. There have been an extensive number of heuristics varying widely in strategy that have been researched. The accuracy of a heuristic has a strong influence on the efficacy of a planner.

Some heuristics that have been effective involve landmarks. Landmarks in planning are analogous to the use of landmarks when someone is navigating [55]. When traveling, a person may know that they are approaching their destination when they see a particular statue. To a planner, the landmark will be a logical fact or formula that will signal that it is making progress towards the goal. Landmarks for a given planning problem are discovered in a preprocessing phase before the searching actually begins. We have adapted landmarks for use with preferences and developed two heuristics to guide a planner.

While conducting a forward-chaining heuristically guided search a planner either follows a search algorithm that guarantees finding the best plan first or an algorithm that attempts to find a high quality, if not best, plan quickly. In the former, a planner will explore a wide array of shorter partial plans before exploring longer plans. In the latter, a planner will aggressively follow the best heuristic values it has found and search a narrower range of plans to a greater length. Greedy best-first search is an algorithm which follows this aggressive

approach and has been successful at quickly finding plans. The disadvantage is that the plan found is likely to be less than optimal. We present an algorithm called Cascading Search that improves upon the standard greedy best-first search. The new algorithm widens the areas explored to a limited extent that increases the likelihood of finding a high quality plan while still finding plans quickly.

1.2 Contribution

Our contribution can be broken down into three areas, a preference specification language, an adaptation of landmark heuristics for planning with preferences, and a new algorithm that improve on the greedy best-first search algorithm.

Ranked Preference Specification

We present a system for specifying preferences for planning problems. It allows a rich preference specification that cannot be matched by other languages. Its advantages are that it allows for users to specify preferences so that all plans are directly comparable, but it also allows plans to be incomparable in carefully controlled situations. Two plans are incomparable if there is no means determine their relative quality. It uses a construct to compactly represent preferences that would otherwise be difficult or impossible to specify. It uses a simple, yet powerful, system for assigning relative priorities to the preferences and is flexible in that different strategies for interpreting a specification can be chosen.

Landmark Heuristics

The field of planning with preferences has received a minute amount of research relative to planning in general. Although heuristics to guide search have been heavily researched, there is only one brief series of papers examining their application to planning with preferences [3, 5, 4]. Landmark based heuristics have been well researched and and have shown success in classical planning [81]. We adapt landmarks and an available heuristic for preferences. We also present two new heuristics for use in planning with preferences. The effectiveness of these heuristics and how well they can be combined with a goal-oriented heuristic is examined. We show experimental results based on an available planner and how well these strategies succeed in finding high quality plans.

Cascading Search Algorithms

Finally, we present our Cascading Search adaptation of the greedy best-first search algorithm. This algorithm greatly increases the likelihood of finding a high quality, if not optimal, plan quickly. The algorithm is designed to diversify the number of quality alternatives that the search is examining without slowing the search by examining too many. The

success of search based planners is dependent on the accuracy of the heuristic used. Even with an accurate heuristic, mistakes are expected. This algorithm is designed to quickly correct for those mistakes. The algorithm utilizes a systematic approach and does not rely on randomness or additional computation. We show how effective the Cascading Search algorithm is on the planning search on preferences planning problems.

1.3 Organization

The paper will be organized as follows. The second chapter covers background and related work on planning in general and preferences in general. The third chapter will present our preference specification language along with related work on preference languages used in planning. The fourth chapter covers background and related work on strategies used in planning with preferences and landmarks before presenting our use of landmarks in planning with preferences. The fifth chapter presents the Cascading Search algorithm and related work on algorithms to create a diverse search tree. The sixth chapter explains our experiments and results. The final chapter is a conclusion and discussion of future avenues of research this work leads to.

Chapter 2

Background

This chapter consists of background and related work for planning in general and preferences in general. Further background and related work more specific to our area of research is contained within the relevant chapters.

2.1 Planning

Research in planning is diverse. As happens in these situations, there are numerous methods for defining a planning problem and associated concepts that have been used, each with its own subtle differences. We will use a single set of definitions and use it throughout this work. When necessary, we will adapt definitions from other work to fit the set that we have provided.

The field of planning covers a wide variety of problems. Except for the addition of preferences, our work falls completely within classical planning. Classical planning is sequential, is deterministic with full world knowledge, and uses actions with a unit-cost. Sequential planning allows only a single action to take place at a time, in contrast to concurrent planning where one or more actions can be taking place at any time. The exact state of the world and the exact effects of every action are known at all times. When using unit-cost actions, the length of a plan is the number of actions it contains. There are fields within planning that deal with changing these assumptions, but classical planning is a difficult problem that still receives a great deal of research.

Another fundamental split in planning is between domain independent and domain dependent planning. In the latter, planners are customized for a single domain. There are great advantages to this approach if you need a planner for a specific application. One domain dependent approach exploits the fact that most domains have sequences of actions that are completely useless, but a planner will explore. A planner can be customized so that these futile sequences can be avoided, greatly reducing the search space to explore. For example, if trying to plan a set of deliveries for a truck, there would never be a reason to

drive to a location and then drive to another location without performing a delivery action in between. Some domain-dependent planners can run on any domain, but allow for input of this useful knowledge, [31], while other planners, particularly if a planner is needed for a single application, will be purpose built for a single domain.

Our work is in domain independent planning. This is an inherently more difficult problem. In the "real world" there would be no reason to use a strictly domain independent planner if you had difficult problems in a specific application. However, domain independent planning presents more interesting obstacles in creating successful strategies to deal with any problem presented. Advances in domain-independent planning are applicable to domain dependent planning.

2.1.1 Preliminaries

A planning problem will first include a domain definition which involves two basic components, a set of *fluents*, F, and a set of *actions*, A. In some work, actions are called operators. Fluents describe the state of the world. Possible fluents in the vacation domain are $atCity(Toronto)$ and $attended(Toronto, Museum)$ which would be true if the traveler was present in Toronto and if the traveler had visited a museum in Toronto. A *state* describes one possible configuration of the world. We will use a version of the STRIPS planning formalism from Fikes and Nilsson [38]. A *fluent atom* is simply a single fluent such as the two mentioned above. A *fluent literal* is a possibly negated fluent atom.

Definition 1. *Given a set of fluent atoms, F, a state, s, is a possibly empty subset of F, $s \subseteq F$. The fluent atom v, is true in state s, $s \models v$ iff $v \in s$.*

As an example, consider a small set of fluents that just represents whether a traveler is in Vancouver or Toronto, and one possible state.

$$F = \{atCity(Vancouver), atCity(Toronto)\}\ s = \{atCity(Vancouver)\} \tag{2.1}$$

This represents the traveler being in Vancouver, $atCity(Vancouver) = true$, but not in Toronto, $atCity(Toronto) = false$. Now that the truth of atoms has been established, formulas can be built with the propositional operators.

The set A defines the actions that can be utilized in a plan. The two most common approaches are from STRIPS and ADL [77] from Pednault. There are many variations within these approaches. STRIPS actions are defined with a set of preconditions and a set of effects. The preconditions are generally positive fluent literals while the effects can be positive, *add-effects*, or negative, *delete-effects*. ADL is a more expressive option which also describes preconditions and effects, but preconditions are free to be propositional formulas and effects can be conditional. Conditional effects allow an action's effects to change based on the current state. Some work has kept the definition of actions more abstract by simply

stating there is a function over the set of possible states S and A, $S \times A \times S$ which directly defines when an action will cause a transition from one state to another.

Planners generally accept either STRIPS or ADL actions. However, if given ADL, most convert the ADL specification into a STRIPS format that is then used for the actual planning process. Unless noted otherwise, actions in this work will follow the STRIPS style.

Definition 2. *A STRIPS action a is a pair (P, E) where the set of preconditions, P, is a set of fluent atoms defining the conditions under which a is executable and the set of effects, E, is a set of fluent literals describing the effects of a.*

Example actions from a vacation domain could be an action to fly between cities, $fly(Vancouver, Toronto)$, and an action to attend a museum $attend(Vancouver, Museum)$. Preconditions define the required conditions for an action to be possible. The effects define how an action will change the fluents. The action $fly(Vancouver, Toronto)$ would require $atCity(Vancouver)$ to hold as a precondition as the action would not make sense if the traveler was not in Vancouver. The action would have one delete-effect, $\neg atCity(Vancouver)$, and one add-effect, $atCity(Toronto)$.

A STRIPS action is executable if its precondition literals hold and the resulting state is computed by taking the preceding state and adding/removing the atoms for the add/delete effects.

Definition 3. *Given a STRIPS action a, (P, E), and states s and s' the execution of a causes a transition from s to s', $s[a] = s'$ iff*

$$\begin{aligned} s &\models P \quad , \\ s' &\models v \quad \textit{iff} \quad v \in E, \textit{or} \\ &\qquad\quad s \models v \textit{ and } \neg v \notin E \end{aligned}$$

A complete planning problem includes a set of fluents, a set of actions, an initial state, and a goal.

Definition 4. *A planning problem $P = (F, A, s_0, g)$ consists of a set of fluents F, a set of STRIPS actions, A, an initial state s_0, and the goal g, a set of fluent atoms.*

Given a planning problem, a *plan* is a sequence of actions that is executable and leads from the initial state to a goal state. A *partial plan* is a sequence of actions that does not lead to a goal state.

Definition 5. *Given a planning problem, $P = (F, A, s_0, g)$, a plan $p = (a_1, a_2 \ldots, a_n)$ is a series of actions, that when iteratively applied to s_0 produces the series of states $(s_0, s_1, \ldots s_n)$, $s_0[a_0] = s_1$, $s_1[a_1] = s_2 \ldots s_{n-1}[a_{n-1}] = s_n$ every action is executable and the final state s_n satisfies the goal, g, $s_n \supseteq g$.*

Vacation Domain	
Cities	$Vancouver$, $Toronto$, $Orlando$, $NewYork$, $SanFrancisco$
Diversions	$Theater$, $Sports$, $ThemePark$, $Museum$
Areas	$Downtown$, $Waterfront$, $Airport$
Fluents	$atCity(city)$, $visited(city)$, $attended(diversion)$, $stayedAtHotel(city, area)$, $attended(city, diversion)$
Actions	$fly(city1, city2)$ $(city1 \neq city2)$
	Pre: $atCity(city1)$ Eff: $\neg atCity(city1)$, $atCity(city2)$, $visited(city2)$
	$stayAtHotel(city, area)$
	Pre: $atCity(city)$ Eff: $stayedAtHotel(city, area)$
	$attend(city, diversion)$
	Pre: $atCity(city)$ Eff: $attended(city, diversion)$

Problem	
Initial state	$\{atCity(Vancouver)\}$
Goal	$\{atCity(Vancouver), visited(San\_Francisco)\}$

Table 2.1: Vacation Domain: Pre - Preconditions, Eff - Effects

We also define the concept of a *history* which is a sequence of alternating states and actions. A history may correspond with a plan or a partial plan.

Definition 6. *Given a planning problem P,*

A history, H, of length n, over P is a sequence $(s_0, a_1, s_1, \ldots, s_{n-1}, a_n, s_n)$ such that $s_0[a_1] = s_1, \ldots, s_{n-1}[a_n] = s_n$

We present a simple vacation domain, Table 2.1, and planning problem that we will use for examples throughout this work. Instead of listing every possible STRIPS action, $fly(Vancouver, Toronto)$, $fly(Vancouver, NewYork)$, and so on we give a template. All of the STRIPS fly actions are produced by substituting in the available cities for fly and similarly for $stayAtHotel$ and $attend$.

A plan for this simple problem is

$$fly(Vancouver, SanFrancisco),\ fly(SanFrancisco, Vancouver) \tag{2.2}$$

The only change to the final state over the initial is the addition of $visited(SanFrancisco)$ and $visited(Vancouver)$ which will have become true.

2.1.2 Planning Strategies

There are numerous strategies that have been employed for solving planning problems. We will briefly explain the forward-chaining search strategy we employ along with a few that

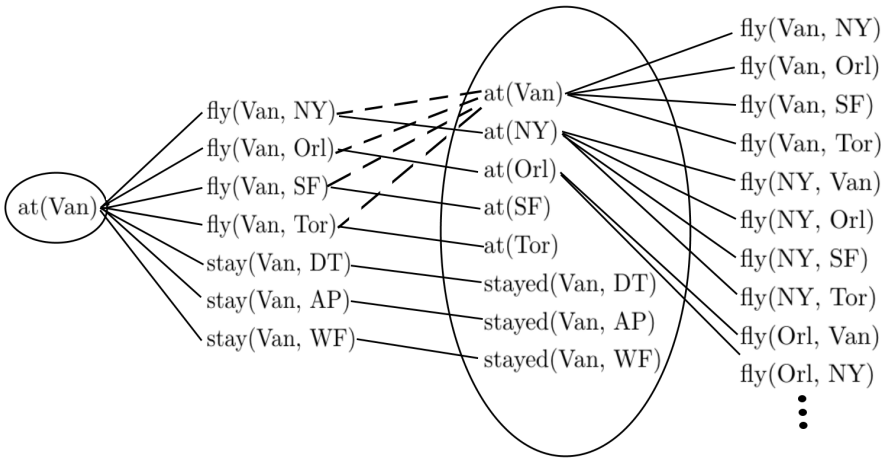

Figure 2.1: A partial planning graph for the vacation problem. The fluent layers are circled.

have received significant research. In chapter 4 we will discuss work that has adapted some of these strategies for planning with preferences.

Planning Graph Analysis

The *planning graph* is a structure that Blum and Furst [11] developed. It is an intuitive structure that lays out all of the fluents and actions and their relations to each other. Once the graph has been created, it is used to develop an effective planner, GraphPlan. A planning graph consists of layers that alternate between layers of fluents and layers of actions. The first layer corresponds to the initial state of the planning problem and contains a node for each fluent that is true. The next layer will contain an action node for every action that can be performed on the initial state. The next layer contains all fluents that might be true in the second state, then a layer for all possible actions from the second state. There are three kinds of edges in the graph. Every action is connected by precondition-edges to the appropriate fluents in the previous layer and add-edges and delete-edges to the appropriate fluents in the following layer.

A portion of the planning graph for the vacation problem is shown in Figure 2.1. We only include the fly and $stayAtHotel$ actions. The first layer contains only the single fluent included in the initial state, $atCity(Vancouver)$. The actions and fluents are abbreviated for space. For the second layer, we only include the possible fly and $stayAtHotel$ actions. The third layer includes all of the $atCity$ fluents as any city can be reached with a single fly action and the possible $stayedAtHotel$ fluents. Edges are added for the precondition and effects of each action. The dashed lines represent delete effects.

The planning graph is useful because it is easy to create in terms of computation and space and during construction, actions and fluents that are mutually exclusive can be

tracked. If a graph has t action layers, then it will contain any plan of length t or less as a subgraph. The GraphPlan algorithm first creates the planning graph and extends it until a fluent layer contains every goal fluent and none of them are mutually exclusive. It then works backwards from the goals, selecting a set of compatible actions in the final action layer that satisfy all or part of the goal. Then it repeats the process from the previous fluent layer using the unsatisfied preconditions of the selected actions as subgoals. If it hits a dead end it backtracks and changes the actions selected. Today, there is little work going into graph planning, but the planning graph has been adapted and utilized in developing heuristics.

Planning as State-Space Search

A basic technique for finding plans that has been in use since the early days of planning is a state-space search. The state-space is a tree where each node represents one possible assignment of fluent values, one possible state of the world. Directed edges, labeled with an action, lead from one node to another if that action would create a transition between the two states.

The two basic strategies used to explore the state-space are *forward-chaining* and *backward-chaining*, also called progression and regression planning. In forward-chaining search, the planner begins with a root node for the initial state. Then, it expands this node by applying every action whose preconditions are met by its state and adding the resulting nodes and the appropriate edges. The planner continues expanding nodes until one is found which represents a state that satisfies the goal conditions. The plan is read from the edges that lead from the initial node. When a node is expanded, there is usually a number of resulting nodes. Different strategies are available for choosing the next node to expand based on *heuristics* or other means. Heuristics are functions that can be applied to each state and estimate how close that state is to the goal. Heuristically guided forward-chaining search has been used to great success and is the strategy employed in our work. When discussing search, we use the terms node and state interchangeably as each node represents a single state.

One example of a heuristic is the goal-count heuristic. To evaluate a state it simply counts the number of the goal's atoms that are not satisfied in a state. This is unsophisticated and often inaccurate. For example, a planning problem's goal may be a single atom. In this case, every state that does not satisfy the goal is evaluated to 1. There's no differentiation between states far from the goal and a state that can reach the goal with only one additional action.

The success of a search guided by a heuristic depends on the ability of the heuristic to accurately evaluate a state. This has lead to a great deal of research into developing accurate heuristics. The FF heuristic is one such heuristic developed by Hoffman and Nebel [54]. The FF heuristic is based on the *relaxed planning graph*. The relaxed planning graph is

```
openList = ()
closedList = ()
add initialNode to openList
while openList is not empty
    currentNode = best node from openList
    if currentNode is not in closedList
        for each action executable from currentNode
            newNode = execute action on currentNode
            if newNode is a solution
                return plan(newNode)
            insert newNode in openList according to heur(newNode)
        add currentNode to closedList
return FAIL
```

Figure 2.2: Pseudocode for the greedy best-first search algorithm. heur() is the heuristic function that evaluates the quality of a node. plan(node) returns the series of actions that lead to that node

built in the same way as the standard planning graph, but using the delete-relaxation of the problem where actions delete-effects are ignored. This results in a planning graph where actions and fluents are never mutually exclusive. If the GraphPlan algorithm is used on a relaxed planning graph, a solution is found efficiently because when it chooses an action to satisfy a fluent, it never has to reverse that choice. Such reversals happen due to interference between actions. When actions never have delete effects, interference is impossible. Given a state, the FF heuristic returns the length of the solution found based on the relaxed planning graph. The FF heuristic came from work from Bonet and Geffner which also used the delete-relaxation of the planning problem to evaluate a heuristic [13]. FF used the relaxed planning graph to more efficiently and accurately estimate the distance to the goal.

Numerous approaches have been taken for developing heuristics. We will only mention a couple of the significant ones. Abstraction based heuristics create an abstracted version of the search space of the planning problem. The abstraction combines groups of closely related states into a single new state. It is built so that the path between two states in the abstracted space is never more than the path from any two states, one taken from each abstracted state. This abstracted search space is a fraction of the original and plans within it are easily found. The lengths of the abstracted plans are used as the heuristic estimate for states in the non-abstracted search space [52, 62, 36, 53]. Another heuristic strategy receiving attention is the critical path heuristic. It estimates the distance to the goal by looking at subsets of facts from the goal. The distance for the most difficult subset provides the heuristic estimate [47, 51].

Next, two forward search algorithms are presented. We begin with the *greedy best-first search* (GBFS) algorithm, Figure 2.2. An *open-list* maintains a list of all states that have been found but not expanded. The open-list is sorted based on the heuristic value of each state. The search repeatedly expands the best node in the open-list.

```
openList = ()
closedList = ()
add initialNode to openList
while openList is not empty
    currentNode = best node from openList
    if currentNode is a solution
        return currentNode
    if currentNode is not in closedList
        for each action executable from currentNode
            newNode = execute action on currentNode
            insert newNode in openList according to (heur(newNode) + g(newNode))
        add currentNode to closedList
return FAIL
```

Figure 2.3: Pseudocode for A* Search Algorithm. heur() is the heuristic function that evaluates the quality of a node. g() returns the depth of the node. plan(node) returns the series of actions that lead to that node

An alternative strategy is the *A** search shown in Figure 2.3 [46]. The open-list is sorted by the sum of the heuristic value and the depth of the node. This sum estimates the length of the shortest plan that passes through the node. A* returns a plan when a goal node is removed from the open-list. GBFS returned when it found a plan before adding the node to the open-list. The difference is because A* is attempting to find the shortest possible plan and must check if a shorter plan is in the open-list if one is found when expanding a node. If given an *admissible* heuristic, A* is guaranteed to return the shortest possible plan. A heuristic is admissible if it never overestimates the distance to the goal, otherwise it is *inadmissible*. An admissible heuristic ensures that when a goal node is removed from the open-list, no nodes left could possibly lead to a better plan.

The GBFS algorithm aggressively follows the heuristic values while A* factors in the depth of a node, distance from the initial node, to widen the search space. If a heuristic is accurate enough, GBFS may return an optimal plan, but the algorithm is used to narrow the exploration. In difficult planning problems A* usually fails because of memory or time limitations. GBFS is more likely to find a plan quickly. If an inadmissible heuristic is chosen, A* is still a valid choice. It just loses the guarantee of an optimal solution. Because of its consideration of the depth of a node, if it can return a plan, it is likely to be shorter than one found by GBFS.

The choice of algorithm is strongly influenced by the intent of the planner. The A* or similar variations are more prevalent in *optimal planning* where the sole aim is to find the best plan possible. *Satisficing planning* is used when planning problems are difficult enough that finding the best plan is unrealistic; just finding a plan is the intention and GBFS is likely a better choice.

Both of the algorithms are shown using a closed-list. The closed-list keeps track of all states that have been expanded. A state can appear many times in the search space. The closed-list prevents time from being wasted by expanding the same state multiple times.

This also prevents the search from getting stuck in cycles in the search space. The algorithms can also be used without a closed-list.

The two algorithms as presented immediately return a plan when one is found. Instead of returning the plan immediately the search can be allowed to continue. The plan is saved and nodes that are too deep to give a plan shorter than that found are ignored. In this way the search continues until a time limit is reached and the planner returns the best plan found. This is called an *anytime* search.

A backward-chaining search, starts with the goal conditions and attempts to work back to the initial state. A notable difference is that this strategy does not involve working with fully defined states with completely set fluent values as with forward-chaining. The goal node only includes the fluents necessary to satisfy the goal. The node is a partial definition of a state. That node then represents every possible goal state. An action is chosen whose effects include part of the goal. The action is applied in reverse to create another partially defined state that satisfies the action's preconditions. Those preconditions then become subgoals if they don't match the initial state. When possible these preconditions will lead to a variable in the partially defined state. From our example domain, the action $fly(X, Vancouver)$ satisfies the goal of $atCity(Vancouver)$ where X is any of the other cities. The search works in this way until a node is found that has no contradictions with the initial state. At this point a plan has been found.

Backward-chaining search has weaknesses that have lead to little research into it in the past decade. Some of these difficulties arise from the states in the search space only having partial assignments. For example, forward-chaining search can easily detect when different sequences of actions have led to the same state. This is difficult to determine when a state isn't completely defined. Also, the backward-chaining search may waste time exploring states that are unreachable from the initial state. By their nature, forward-chaining planners will never explore unreachable states. Recently, Alcázar et al. have done work on applying advances from the recent research in forward-chaining search to this approach with some success [2].

Problem Translations

If a planning problem can be translated into another type of problem, for which efficient techniques are available to solve, then those techniques can immediately be applied to solve the original planning problem. These transformations usually end with a formula or program written in a logical language. If a solution for that formula or program is found, a solution plan can be easily read from it. The first method we will discuss, planning as satisfiability, was developed by Kautz and Selman [63, 64]

A satisfiability problem is a propositional formula or formulas involving a large number of atoms. The goal is to either find an assignment of true or false to each atom such that the formula is satisfied or determine that no such assignment exists. When a planning

problem is converted to a satisfiability problem, if the SAT-solver, a program for solving these problems, returns an assignment that satisfies the formula, then a plan can be read from that assignment. When converting a planning problem to this format, a maximum plan length must be chosen. The answer will then specify if there is a plan of that length or less. So, to discover if a planning problem has a solution, the SAT-solver can be run multiple times with formulas for increasing possible plan lengths.

The formulas that are created are complex, but there are some basic ideas behind them. The SAT-solver has no concept of what the variables represent, so every aspect of the planning problem has to be explicitly encoded. A formula will have to encode every possible state and every possible plan up to the set maximum length. Therefore, the formula requires an atom for every action and every fluent for every state. These atoms must be combined in a way so that action preconditions and effects are enforced, that fluents unaffected by an action remain the same to the next state, and that only one action can take place at a time.

Many improvements have been made to the encoding of a planning problem into a formula, but one particularly interesting one, again by Kautz and Selman [64], uses the planning graph to improve the translation. The structure of the planning graph is used to detect mutually exclusive facts and actions. This knowledge is used to reduce the size of the final formula.

Do and Kambhampati [30] developed a method for translating a planning problem into a Constraint Satisfaction Problem. A CSP consists of a set of variables, each with a domain of possible values and a set of constraints that restricts the values that can be assigned to each variable. A problem that can easily be written as a CSP is the graph coloring problem. There is a variable for each node and the domain for each variable is the set of possible colors. The constraint is that no two adjacent nodes can have the same color. A search is then performed to find a set of variable assignments that satisfy the constraints.

A planning problem can be formulated as a CSP by having an action variable and fluent variables for each step of the plan. The domain of the action variables is the set of all possible actions. Each fluent will have a variable for every possible time. Each fluent variable's domain is the same as the original fluent's domain. Together these variables allow the assignment of actions and the appropriate fluents in response to those actions. Constraints are added to enforce preconditions and effects of actions, the initial state, and the goal. Once the CSP is solved, the plan can be read from the assignments to the action variables.

The final approach involving a fundamental translation of the planning problem is from Lifschitz [70]. In this approach, the problem is reformulated for Answer Set Programming (ASP) [29, 40]. ASP is a form of declarative programming designed to solve difficult search problems. The primary construct of an ASP problem is a rule of the form $head : body$ where *head* and *body* consist of lists of literals. If all literals in *body* evaluate to true, then at least

one literal in *head* must also be true. A set of these rules is an answer set program. Given a set of rules, a solution consists of a minimal set of literals that satisfies every rule, i.e. if you remove any literal, the set no longer satisfies the program.

There are clear parallels between these three methods. They have some strengths and weaknesses in common. The translation of a planning problem into these new logical formulations is not difficult, but if not done well can lead to a translated problem that is larger or more difficult to solve than it needs to be. Advances in these areas often involve efficient strategies to simplify the resulting translation. A weakness of these strategies is that the translation must be done with plans of a certain maximum length in mind. If no solution is possible, it means that no plan of that length or less is available, not that the planning problem is unsolvable. If an appropriate bound cannot easily be chosen, an iterative approach is necessary where the process is run for a given maximum length and if no plan is found, the process is repeated for a greater maximum length. The greater this limit on the length is, the more difficult it is for the appropriate solver to answer the translated problem. Therefore, it's inefficient to just use an overly large length limit at the start.

2.2 Preferences

The concept of preferences can be applied to a wide variety of fields and are in and of themselves a large field of research. As with planning, there is no possibility of giving a thorough background on the topic. We will explain fundamental concepts and present a few systems for specifying preferences that give an idea of the range of possibilities. In this section, we will focus solely on specifying preferences in general. In Chapter 3 we will present the available work on specifying preferences for planning in particular.

When preferences are needed, there are two elements at the start, a set of candidates and a user who has preferences over those candidates. For example, if a traveler has preferences over what city to visit, the candidates would be the cities the traveler could visit. A preference language is used to create a model of the user's preferences. That model is then applied to the candidates to find the optimal candidate, decide which of two candidates is better, or answer other questions about the quality of the candidates. For our discussion, we will assume candidates are described by a finite set of propositional variables.

A preference language first allows a user to specify the attributes that are relevant. The next step, though not necessary, allows the user to specify the relative value of these attributes. Some attributes may be desired and others unwanted. Among attributes that are desired, some may be more important than others. If it is not possible to satisfy every element of the preferences, it is helpful to know which elements should be ignored if a trade-off must be made. The final component is a method of interpreting preferences and their importance so that the best candidate can be selected.

A direct approach to specifying preferences is to apply a value to every possible candidate, Seattle is worth 5, New York is worth 10, etc., or order them in terms of quality, New York is better than Seattle is better than... This is generally untenable as many applications for preferences will have a vast number of candidates. A preference language facilitates a compact representation of a user's preferences. This representation is then used to evaluate or compare candidates.

The two most commonly used approaches are *ordinal* preferences and *cardinal* preferences. An ordinal approach uses the specified preferences to define a binary relation that specifies the relative value of two candidates. Given any two candidates, for every possible pair either one will be preferred, or they will be equally preferred. A cardinal approach uses a *utility function* to assign a value, utility, to each candidate. The value could be assigned from a set as broad as the set of real numbers or as restrictive of a set of two values (0,1). Comparing two candidates is then as simple as comparing the values assigned to them. If the values are numeric and meant to specify exactly how preferred a candidate is, that is a *quantitative* approach. First we will formally define a preference relation used in ordinal approaches and then define a utility function which cardinal approaches use to assign a value to a candidate.

Definition 7. *An Ordinal Preference Relation on a set of candidates S is a reflexive, transitive relation $\geq$*

- *$x \geq y$, x is at least as preferred as y.*
- *$x > y \leftrightarrow x \geq y$ and $y \ngeq x$, x is strictly preferred to y*
- *$x \sim y \leftrightarrow x \geq y$ and $y \geq x$, x is equally preferred to y*

An ordinal preference relation induces a *preorder* over the set of candidates. If $x \geq y$ and/or $y \geq x$ for every x and y then the relation induces a *total preorder* over the set of candidates. A partial preorder allows *incomparability* wherein there are pairs of candidates between which the user has no judgment. They are not equally preferred, as is possible with a total preorder, but no comparison can be made between them. Given the choice of cities (N)ew York, (S)an Francisco, and (T)oronto, a traveler could define the following ordinal preference relation.

$$\{(N \geq T), (S \geq T), (N \geq S), (S \geq N), (S \geq S), (N \geq N), (T \geq T)\} \tag{2.3}$$

This relation specifies that New York and San Francisco are equally preferred and that both are preferred to Toronto. This creates a total preorder over the cities. If a fourth city (O)rlando is included and the traveler adds only $(O \geq T)$ then it does not create a total preorder as Orlando can't be compared to New York or San Francisco.

Definition 8. *A utility function $u : S \to V$ assigns each candidate in S a value from a totally ordered set V*

Our traveler might define the utility function as $u(NewYork) = 5$, $u(SanFrancisco) = 5$, $u(Toronto) = 1$ to represent that New York and San Francisco are equally preferred and are both much more preferred than Toronto.

We need to look at how well these approaches function when used to ask different questions. How difficult is it to find the optimal candidate from all or part of the set of candidates, compare two candidates, or rank a set of candidates? More formally we will refer to a candidate x as optimal if it is *non-dominated*. A candidate is non-dominated if there are no candidates that are strictly preferred to it. It is possible for many candidates to fulfill this criteria.

A cardinal approach normally leads to computationally easier answers to these questions. If all candidates have been assigned a value, it takes a linear run through them to rank all candidates, that is, every candidate is examined once. Using a relation requires that every pair of candidates be compared to determine a ranking.

2.2.1 Propositional Formulas

We will look at defining the relevant characteristics through a set of propositional formulas, P. Each of the formulas represents something that is desired in a candidate. Then, the question is, how does one use this set to compare candidates? These simple approaches will remain relevant, even as the preference languages become more complex.

A cardinal approach is to simply count the number of formulas in P satisfied by a candidate. If every formula is satisfied, than a candidate is optimal. If only a single optimal candidate is desired, no more candidates need to be evaluated. However, if no candidates satisfy every formula, all candidates may need to be evaluated.

An ordinal approach is to use a superset based relation to compare two candidates. If we have candidates x and y with P_x and P_y representing the subsets of P satisfied by the candidates, then $x \geq y$ if and only if $P_x \supseteq P_y$. This approach leads to a great deal of incomparability as even if P_x is much larger than P_y, P_y only needs a single member not in P_x to not satisfy the relation.

Next, we move to approaches that assign relative importance to the formulas in P. The most intuitive is to add a numeric weight to each preference formula. In a cardinal approach, the obvious way to convert this information to a value is to sum the values. Other interpretations are to use the value of the most desired formula that is satisfied or the most desired formula that is unsatisfied.

The final approach based around basic preferences as propositional formulas is to assign priorities to the preferences. These priorities can be numeric values, but more generally, a priority relation can also be used that determines if one formula has a higher priority than

another. A priority relation follows the same structure as a preference relation as seen in Definition 7. A priority relation may induce a total preorder.

We will focus on an adaptation of this priority approach, a system for preferences centered around a *ranked knowledge base* (RKB) from Brewka [21]. Preference formulas are organized into a RKB which defines the priorities of satisfying the individual preferences. You can look at an RKB in two ways. The first is by attaching an integer rank to each of the preference formulas. A higher rank implies a higher priority. The second way is to consider an RKB a set of sets of formulas $(F_1, F_2, \ldots F_n)$ where F_n contains the highest priority formulas, $F_{(n-1)}$ lower priority formulas and so on down the line. The intuition is that one state is preferred to another state if it is preferred with respect to the preferences at a given rank and above regardless of which state better satisfies the preferences at the lower ranks.

After the structure of the preference language is laid out Brewka discusses how a specification in this language can be used to compare candidates. These strategies can be applied to any priority relation that induces a total preorder. He gives four basic strategies for interpreting the information in a RKB over candidates x_1 and x_2.

1. maxsat: x_1 is preferred to x_2 if the highest priority formula satisfied by x_1 has a higher priority than the highest priority formula satisfied by x_2
2. maxunsat: x_1 is preferred to x_2 if the highest priority formula not satisfied by x_1 is of a lower priority than the highest priority formula not satisfied by x_2
3. $\subseteq$(superset): x_1 is preferred to x_2 if at the highest rank where the two states differ in the preferences they satisfy, x_1 satisfies a superset of the preferences x_2 satisfies
4. #(cardinality): x_1 is preferred to x_2 if at the highest rank where they satisfy a different number of preferences, x_1 satisfies more

Of these strategies, all can be viewed as cardinal approaches except for the superset based strategy. These four strategies can give notably different orderings over a set of candidates, demonstrating the importance of the choice of strategy. For example, assume we have 4 preference formula A, B, C, and D organized into a RKB by assigning A and B the rank 2 and C and D the rank 1. Let's look at how four candidates represented by the formulas they satisfy, x_{AB}, x_{AC}, x_{BCD}, and x_C, where A and B hold in x_{AB}, are ordered by the four strategies.

1. maxsat: $\{x_{AB}, x_{AC}, x_{BCD}\} > x_C$
2. maxunsat: $x_{AB} > \{x_{AC}, x_{BCD}, x_C\}$
3. $\subseteq$(superset): $x_{AB} > \{x_{AC}, x_{BCD}\} > x_C$
4. #(cardinality): $x_{AB} > x_{BCD} > x_{AC} > x_C$

2.2.2 Conditional Preferences

Next, we look at languages that center around the idea of a conditional preference. The preferences we've looked at so far are formulas that are applied to every candidate. A conditional preference is a preference that is only applied if another criteria is satisfied. With vacation planning, an example would be, if visiting Seattle, traveling by train is preferred, but if visiting New York, traveling by air is preferred. If the appropriate city is visited, than the preference is relevant, but otherwise, the preference should be ignored. The basic form of a conditional preference is $\psi : \phi$ which states that if ψ is satisfied then it is preferred that ϕ also be satisfied.

An issue with these conditional preferences is that in an ordinal approach they tend to lead to a very weak order of candidates due to incomparability. Two candidates are only comparable if they satisfy the same set of conditions and therefore the same set of preferences can be applied. In a cardinal approach, values can be applied and used to calculate a value for each candidate. However, while this leads to easy comparisons among the candidates, it may lead to an inaccurate representation of the user's preferences. In the travel example from the previous paragraph, a preference has been expressed if traveling to Seattle or if traveling to New York, but no preference has been given to whether New York or Seattle is preferred. The utility function of a cardinal based approach leads to an implication of certainty of preference between candidates that is not necessarily intended by the user.

2.2.3 CP-nets and Extensions

We will now move onto graphical approaches to representing preferences that utilize conditional preferences. Graphs are a useful tool for representing users' preferences and relations between them. *CP-nets* are a widely used and adapted graphical means for representing preference that we will discuss along with some extensions.

CP-nets, conditional preference networks, are presented by Boutilier, Brafman, Domshlak, and Poole [17] and build upon conditional preferences. These preferences are then compiled into a graph that represents their relative importance. This graph can be used to efficiently find the optimal values for the system. It is simple and effective for a range of applications, but it is restricted to relatively basic preferences. In later work CP-nets are extended to represent more information about the relative importance of preferences in TCP-nets [20] and to incorporate a utility function in UCP-nets [16].

CP-nets center around the idea that a large number of preferences are conditional and of the form "All else being equal, I prefer this to that". These are *ceteris paribus*, all else being equal, statements and have been explored by others, [35]. Preferences are defined over a set of variables, V, each of which represents some feature or attribute that the preference will involve. These variable may be propositional or multi-valued. Each variable

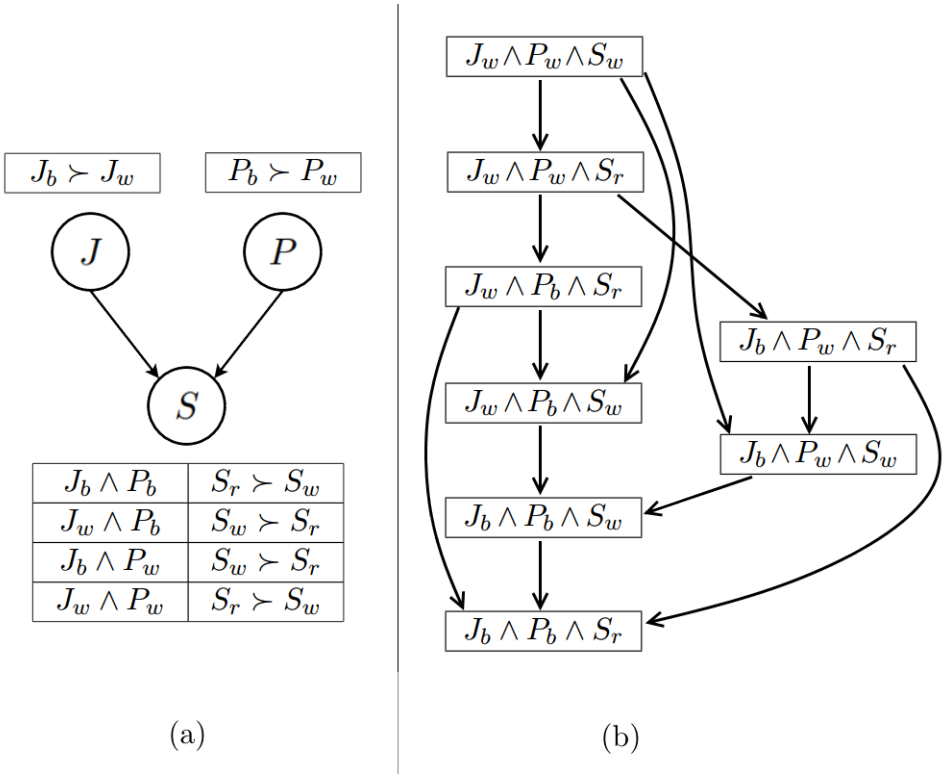

Figure 2.4: An example from [17] showing a CP-net (a) and the preference ordering that it induces (b).

is represented by a node in a CP-net. Directed edges are added to the graph that represent the dependence of preferences. If a preference over variable B varies depending on the value of variable A, then a directed edge is added from A to B. The last step requires defining the preference over the possible values of each of the variables in the CP-net. This is done with a *conditional preference table* (CPT). If a variable is not dependent on other variables i.e. it has no parent nodes, then its CPT contains one ordering over that variable's possible values. If it does have parent nodes, then the CPT will contain an ordering over the possible values for each complete setting of values over the parent variables.

We'll go through an example taken from [17] that involves preferences over what evening dress to wear. There are three variables, J, P, and S, which respectively indicate whether a black or white jacket is worn, whether black or white pants are worn, and whether a red or white shirt is worn. The user prefers black pants and jacket in all situations, but the shirt preference varies depending on which jacket and pants are chosen. Therefore we see edges from J and P to S representing S's dependence on the two variables. The graphical representation is seen in figure 2.4a. The $\succ$ symbol is used to define preferences in the CPTs. We see in the CPT for J that $J_b \succ J_w$ which indicates that the black jacket is preferred to the white and from the CPT for S that the red shirt is preferred if the pants and jacket are either both white ($J_w \wedge P_w$), or both black.

The CP-net can then be used to create a preference graph that represents which assignments to the variables are preferred to others. This is seen in figure 2.4b. An edge from one assignment to another indicates that the second assignment is more preferred. The preference relation in the graph is transitive, so this can be generalized to say that an assignment is preferred to another if there is a path from the latter to the former. This does not give a total preorder as can be seen with the settings $J_w \wedge P_b \wedge S_r$ and $J_b \wedge P_w \wedge S_r$ in the example. There is no preference induced between these settings by the CP-net. So long as the CP-net is acyclic finding the most preferred variable assignment is straightforward. It can be found by assigning the most preferred value to variables with no parents and then proceeding through the graph assigning variables the most preferred value based upon their parents' values which already have been chosen. If the CP-net contains cycles, there may or may not be a most preferred variable assignment.

In a sense, CP-nets implicitly provide information on which variable preferences are more important than others. The value of a parent node is more important than the value of its child when finding the most preferred variable assignment. However, a CP-net does not give the ability to specify relative importance of variables that are independent of each other. To address this issue, Brafman and Domshlak [20] define *tradeoff-enhanced CP-nets*, TCP-nets. TCP-nets have two new types of edges that can be added that both indicate relative importance between variables. The first type of edge leads from a more important variable to a less important one and this difference in importance is unconditional and represents that the value of some variable A is always more important than the value of another variable B. A second type of edge connects two nodes whose relative importance depends on the values of other variables. An edge of this type could indicate that if C is true, than A is more important than B, otherwise B is more important than A. Then Brafman and Chernyavsky apply the new structure to planning [19].

To demonstrate a TCP-net we'll use an example from [19] that involves a logistics domain. Imagine that p1, p2,...p5 are five packages that we wish to be delivered. Each is represented by a variable that is true if the package is delivered and false otherwise. Some packages are only important to deliver if others have also been delivered. Therefore, the relative importance of delivering a package may depend on other variables assignments. The TCP-net for this example is shown in Figure 2.5. The directed edge marked with a circle from p_1 to p_4 indicates that p_1 is more important in all situations. The undirected edge marked with a square and p_1 between p_2 and p_4 symbolizes that their relative importance depends on p_1. A similar edge between p_3 and p_4 notated with p_1 and p_2 means their relative importance depends on both of the variables' values. In these two cases tables use the $>$ symbol to denote which variable is more important. Once again $\succ$ is used to show preference.

Another extension of CP-nets is the UCP-net defined in [16]. While CP-nets represent ordinal preferences, UCP-nets use the same structure to represent a utility function for

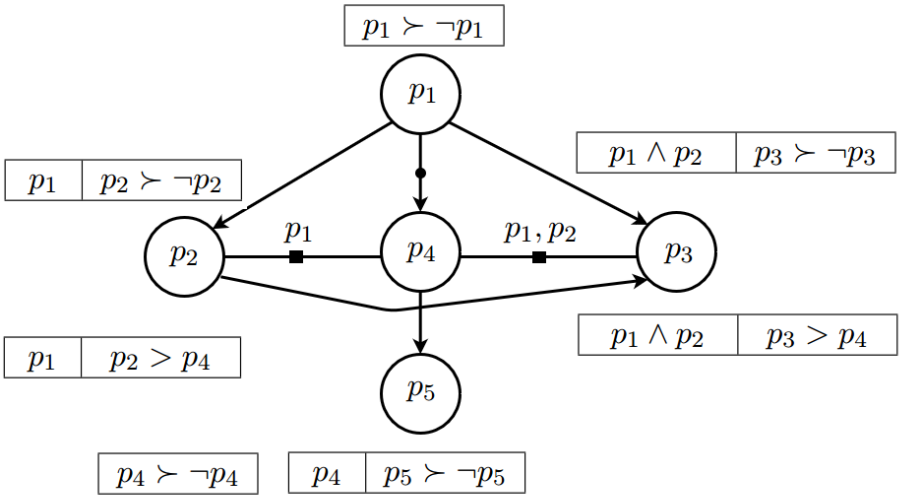

Figure 2.5: An example of a TCP-net from [19]

quantitative preferences. Once the network is created it is straightforward to find the value for a given candidate or to find the optimal candidate(s). The nodes in a UCP-net are the same as in a CP-net, but instead of defining an ordering of the values of a variable based on the parent variable values, a value is assigned to each possible assignment. Determining the value of a candidate simply requires a sweep through the network summing the appropriate values. Finding the optimal candidate can be done by sweeping through the UCP-net and determining the ideal value.

The attention CP-nets have received has resulted in other extensions. In [89] Wang et al. present WCP-nets. They add numerical weights that allow users to use qualitative weights to describe the relative preference between variable assignments. A value can range between mildly preferred to extremely to another value. These weights can also be used to specify which variables are more important than others. CI-nets, conditional importance networks, were proposed by Bouveret, Endriss and Lang [18] are closely related to TCP-nets and are used for specifying preferences over sets of goods.

Chapter 3

Ranked Preference Specification System

In this chapter, we present our system for specifying preferences for planning problems. We begin by presenting some basic definitions and cover the methods that others have used for specifying such preferences. We will only cover the specification here; the next chapter will contain related work on the actual planning process when preferences are involved. We will then present our system and discuss its characteristics. It was originally presented in [83].

3.1 Related Work

In the previous chapter, we introduced basic concepts and some important work related to preferences research in general. Here we will present the limited number of preference languages that have been created specifically for planning.

As we examine work in specifying preferences for planning, all of the languages allow for *temporally extended preferences* (TEPs), that is preferences that don't just concern the goal state, but the entire plan's history. When general preference systems have been applied to planning, they have been limited to goal preferences. This includes work with TCP-nets [19] and ranked knowledge bases [37]. We will also consider the categorizations from preferences in general already discussed. There are cardinal and ordinal approaches. The relevant criteria for preferences can be written in simple propositional logic or made more compact with first order or temporal logic.

The first distinction we make is between *goal preferences*, often called *simple preferences*, and temporally extended preferences. A goal preference is a modification of the classic concept of a goal. A goal preference only describes preferences over the possible goal states. On the other hand, temporally extended preferences describe preferences over entire histories that lead to a goal state. In the first case, it is only the final state that is considered. In the second case, the actions and intermediate states that lead to the goal state are also

involved. In a logistics domain where the goal is to deliver three packages, P1, P2, and P3, it may be more preferred to deliver only P1 than either P2 or P3. This is a goal preference; only the final state is considered. A preference that P1 be delivered by air versus ground is a temporally extended preference because the final state reached may be identical, but one route to that state is preferred to another.

Preference languages may involve simple propositional logic, first order logical, or temporal logic notations. In nearly all cases the planning domains are assumed to be finite, which means that first order notation isn't more expressive than propositional, but simply makes writing preferences more compact. A language is more useful when preferences are more easily specified. To demonstrate this, consider that in these systems a user could theoretically describe their preferences by enumerating every possible goal state or history. Looking at it in this way, the systems don't vary in their expressiveness, but such an enumeration is impossible in any reasonably complex domain. The ability to compactly specify preferences is important.

In this section, we will look at work that has been done in specifying preferences specifically for planning. Work has been done applying preferences to planning using general approaches such as we covered in section 2.2.

3.1.1 $\mathcal{PP}$

Son and Pontelli present a language for specifying preferences, $\mathcal{PP}$ [85]. The language uses a hierarchical structure. Similar structures will be seen in other preference systems, including our own. The lowest level involves defining basic properties of histories. These properties are combined into structures that specify which properties or combinations are preferred to others.

The simplest element of $\mathcal{PP}$ is the *basic desire formula* (BDF). These formulas describe properties of histories that can be preferred in and of themselves or combined through means to come. The simplest BDF is a propositional formula built from fluents. Such a BDF is satisfied if the formula is satisfied by the initial state. To include an action in a preference, $occ(a)$ is used, *occ* abbreviating occurs. It is satisfied if the action a takes place in the appropriate state.

To define properties throughout the entire history, elements from linear temporal logic are used to recursively build BDFs. These elements are $\mathbf{next}(\phi)$, $\mathbf{always}(\phi)$, $\mathbf{eventually}(\phi)$, and $\mathbf{until}(\phi_1, \phi_2)$. Their meanings are straightforward, $\mathbf{always}(\phi)$ is satisfied if ϕ is satisfied throughout the entire history, $\mathbf{until}(\phi_1, \phi_2)$ if ϕ_1 is satisfied until ϕ_2 becomes true etc. Finally, $\mathbf{goal}(\phi)$ is satisfied if ϕ is satisfied in the goal state.

The interpretation of a BDF given a set of histories is straightforward. Any history that satisfies the BDF is preferred to any history that doesn't. Two histories are said to be *indistinguishable* if they are equally preferred with respect to a BDF, i.e. both histories

satisfy or neither history satisfies the BDF. Here are a few possible BDFs from the vacation domain.

- Be in San Francisco after the first action has finished.

$$\mathbf{next}(atCity(SanFrancisco)) \tag{3.1}$$

- At some point, travel directly from San Francisco to Seattle

$$\mathbf{eventually}((atCity(SanFrancisco)) \wedge \mathbf{next}(atCity(Seattle))) \tag{3.2}$$

- Never fly from Vancouver to New York.

$$\mathbf{always}(\neg occ(fly(Vancouver, NewYork)) \tag{3.3}$$

- Visit San Francisco at some time and Seattle at some other time.

$$\mathbf{eventually}(atCity(SanFrancisco)) \wedge \mathbf{eventually}(atCity(Seattle)) \tag{3.4}$$

The next component of $\mathcal{PP}$ is the *atomic preference formula* (APF). Once BDFs that describe the relevant features of histories have been written, their relative importance is defined with an APF. APFs order a group of BDFs from most to least preferred.

Definition 9. *Given BDFs $\phi_1, \phi_2, \ldots \phi_n$ $(n \geq 1)$, then the formula $\phi_1 \lhd \phi_2 \lhd \cdots \lhd \phi_n$ is an APF that states that ϕ_1 is most preferred and ϕ_n is least preferred.*

An example:

- Visiting San Francisco is preferred to visiting New York is preferred to visiting Seattle

$$\begin{aligned} &\mathbf{eventually}(atCity(SanFrancisco)) \lhd \\ &\quad \mathbf{eventually}(atCity(NewYork)) \lhd \\ &\quad\quad \mathbf{eventually}(atCity(Seattle)) \end{aligned} \tag{3.5}$$

The intuition behind an atomic preference formula is that it represents a preference among related options as is seen with the example above. When comparing histories with respect to an APF, the most preferred BDF where the histories differ is considered. Two histories are equally preferred, indistinguishable, if they satisfy the same BDFs within the APF. This can be viewed as APFs having an implicit lowest possibility where none of the involved basic desires are satisfied. Two histories are always comparable with respect to an APF.

The final component of $\mathcal{PP}$ is the *general preference formula* (GPF), which is used to combine multiple APFs. The GPF can be used to define their relative importance or more complex relationships. First, a single APF can itself be a GPF. Then, there are four constructs to combine general preference formulas, Φ_1, Φ_2.

Definition 10. *A general preference formula is either a single atomic preference formula or created by combining GPFs, Φ_1, Φ_2, in the following ways.*

1. $\Phi_1 \,\&\, \Phi_2$*: A history is preferred if it is preferred by all GPFs in the conjunction. A history is indistinguishable if it is so for all GPFs in the conjunction.*

2. $\Phi_1 \,|\, \Phi_2$*: A history is preferred if it is preferred by at least one GPF and not less preferred by any GPF in the disjunction. A history is indistinguishable if it is so for all GPFs in the disjunction.*

3. $!\Phi$*: A history is preferred if it is less preferred by* Φ

4. $\Phi_1 \lhd \Phi_2 \lhd \cdots \lhd \Phi_n$*: A history is preferred if it is preferred with respect to Φ_i and indistinguishable with respect to all Φ_j, $(j < i \leq n)$.*

The atomic preference formulas do not allow for incomparability, but the general preferences do. The & and the | operator can both lead to incomparability. Two histories will be incomparable with respect to a conjunction if they don't have the same relative preference with respect to every GPF in the conjunction. With a disjunction the histories are incomparable if one of the GPFs favors one history and another favors the other history. Even if a conjunction only involves two GPFs, there is a high probability two histories will be incomparable. Let's look at the simplest possible conjunction and disjunction to illustrate this.

First consider the case of the conjunction, $\Phi_1 \,\&\, \Phi_2$ where both elements are basic desire formulas. We have two histories to compare, h_1 and h_2. Each history will satisfy both BDFs, one, or neither giving four possibilities for each history and sixteen possibilities total. There are two cases where one history will be preferred, when one satisfies both BDFs and the other satisfies neither. There are four cases where the two histories are indistinguishable, when the histories satisfy BDFs identically. In the other ten cases, the two histories are incomparable. In the case of the disjunction, $\Phi_1 \,|\, \Phi_2$, incomparability is much less likely, only occurring in the two cases where one history satisfies only one BDF and the other history satisfies only the other BDF. In the four cases where the histories satisfy the BDFs identically, the histories are indistinguishable. This leaves ten cases where one history is preferred. We will later discuss how the incomparability introduced by the & and the | operators can be difficult to utilize.

3.1.2 $\mathcal{LPP}$

Bienvenu, Fritz, and McIlraith present a language that later becomes known as $\mathcal{LPP}$ [9]. As implied by its name this is an evolution of the $\mathcal{PP}$ language discussed above. The concepts of basic desire formulas, atomic preference formulas, and general preference formulas are present with some alterations. An additional level of *aggregated preference formulas* (AgPFs) is added.

The most fundamental difference is that $\mathcal{LPP}$ uses a cardinal approach where $\mathcal{PP}$ uses an ordinal approach. When applied to a history $\mathcal{LPP}$ produces a value used to compare histories. An ordered set, V provides the possible values. The elements v_{min} and v_{max} represent the minimum and maximum values of the set. Lower values are more preferred.

Basic desire formulas remain as they were in $\mathcal{PP}$ with the addition of quantified variables over objects in the planning domain. This allows complex formulas such as the following to be expressed compactly.

- Visit every available city.

$$(\forall x) city(x) \supset \textbf{eventually}(atCity(x)) \tag{3.6}$$

- Never stay in a hotel near the airport.

$$\neg(\exists x\, city(x) \wedge \textbf{eventually}(occ(stayAtHotel(x, Airport)))) \tag{3.7}$$

The value of a history with respect to a given basic desire formula is either v_{min} if it is satisfied or v_{max} if it is not.

Once again atomic preference formulas are used to denote preference between a group of BDFs. Now each BDF in an atomic preference has a value from V attached to it with v_{min} necessarily attached to the most preferred BDF. This means that the relative preference across multiple APFs is clearly defined.

- Visiting San Francisco is preferred to visiting New York is preferred to visiting Seattle

$$\begin{aligned} &\textbf{eventually}(atCity(SanFrancisco))[0] \gg \\ &\textbf{eventually}(atCity(NewYork))[0.5] \gg \\ &\textbf{eventually}(atCity(Seattle))[0.8] \end{aligned} \tag{3.8}$$

The value of a history with respect to a given atomic preference formula is the value of the most preferred basic desire formula that the history satisfies. If none are satisfied, then the APF's value is v_{max}. So, once again there is an implicit final option of not satisfying any of the basic desires.

Now we return to general preference formulas. $\mathcal{LPP}$ also contains conjunction and disjunctions, but they are interpreted differently than in $\mathcal{PP}$. A new conditional form is added.

Definition 11. *A general preference formula in $\mathcal{LPP}$ consists of one of the following and is interpreted as stated.*

1. Φ*: An atomic preference formula evaluated as stated.*
2. $\Phi_1 \,\&\, \Phi_2$*: A history is assigned the maximum of the values from the constituent GPFs.*
3. $\Phi_1 \,|\, \Phi_2$*: A history is assigned the minimum of the values from the constituent GPFs.*
4. $\gamma : \Phi$*: (γ is a BDF) A history is assigned v_{min} if γ is not satisfied, otherwise, the value of Φ.*

The final component of $\mathcal{LPP}$ is the aggregated preference formula. AgPFs provide a few methods of combining GPFs that specify their relative priority and how to interpret them collectively. Different papers have offered different possibilities for AgPFs. Here we present the forms from [10]. An AgPF defines a preference relation used to compare two histories.

Definition 12. *An aggregated preference formula consists of one or more GPFs $(\Phi_1 \ldots \Phi_n)$, consists of one of the following and is interpreted as stated.*

1. Φ*: A general preference formula. Two histories' values are compared.*
2. **lex**$(\Phi_1 \ldots \Phi_n)$*: A history is preferred if it is preferred with respect to Φ_i and equivalent with respect to all Φ_j, $(j < i \leq n)$*
3. **leximin**$(\Phi_1 \ldots \Phi_n)$*: Starting with v_{min}, The number of GPFs that evaluate to that value are counted. If one history has a higher count, it is preferred. Then the next greater value from V is used for comparison.*
4. **sum**$(\Phi_1 \ldots \Phi_n)$*: The values of a history with respect to each of the GPFs is summed and that value is used for comparison.*

After defining these possibilities for an AgPF, the authors only use specifications based on a single general preference formula. While an ordinal approach can be used, they only use $\mathcal{LPP}$ with the cardinal approach. In all of these cases for an AgPF, a total preorder is induced over histories. Two histories are always comparable.

3.1.3 Query and Preference Languages Including Aggregates

Delgrande, Schaub, and Tompits [27] propose a preference language which directly defines when one history is preferred to a second by way of formulas simultaneously involving

elements of both histories. The same authors, [28] have proposed a language that can define aggregate values of histories. It makes use of the notation of first order logic to define temporally extended preferences. The language doesn't contain a structure to relate the importance of various preferences.

There are differences between the languages presented in [27, 28], but they are minor so we'll present them as one. They begin with a query language used to describe properties of histories. The language uses timestamps and timestamp variables to access states and actions throughout the entire history. The language also includes the typical constructs and connectives from first order logic. Instead of giving the rather lengthy list of specifics, that are very similar to those for our proposed language, we'll give a couple possible queries that demonstrate the basics.

For a given history, $(s_0, a_1, s_1, \ldots a_n, s_n)$, a *timestamp variable*, t, ranges over the time indexes of that history, $(0 \leq t \leq n)$.

- Visit San Francisco at some time.

$$\exists t \, atCity(SanFrancisco, t) \tag{3.9}$$

- Visit Toronto at some time and New York at a later time.

$$\exists t_1 \, \exists t_2 \; (t_1 < t_2) \wedge atCity\,(Toronto, t_1) \wedge \; atCity\,(NewYork, t_2) \tag{3.10}$$

To use queries to describe preferences, symbols are added to the query language that allow users to describe the properties of more and less preferred histories. If α, β, and γ are queries, a preference such as $(l : \alpha) \wedge (h : \beta)$ states that a history that satisfies the portion marked with the h is preferred to a history that satisfies the portion marked with l. The preference $(l : \gamma \wedge \alpha) \wedge (h : \gamma \wedge \beta)$ is a conditional preference that states that β is preferred to α when γ holds.

This preference language can deal with a wide array of complex preferences, but doesn't deal well with aggregate values. Aggregate values are values that involve multiple fluents throughout a history. For example, counting how many times a fluent takes on a certain value, summing the value of a fluent over all of the states of a history, and finding the maximum value a fluent takes on over a history are aggregate values. To define these values a system of macros is defined which recursively work over an entire history. We'll omit the details of their syntax, but give an example that illustrates the idea. In the vacation domain a user may be planning a road trip where they drive everywhere. In this case they may want to avoid overly long days of driving. To do this they can specify a preference to minimize the maximum distance traveled on any one day. Assume that there is a fluent $distanceTraveledToday$. A macro $maxDist$ can be defined recursively such that $maxDist(0)$ returns the value $distanceTraveledToday(0)$ at the initial state

and $maxDist(i)$ returns the maximum of $distanceTraveledToday(i)$ and $maxDist(i-1)$. Then the preference $h : maxDist(n) \leq l : maxDist(n)$, where n represents the timestamp for the final state in each history, states that histories with a lower maximum value of $distanceTraveledToday$ are preferred.

Preferences in this system aren't assigned differing levels of importance. Once again differing strategies on how to define preferred and optimal histories can be chosen. These include a cardinality based approach where an optimal history is decided based on the number of histories it is preferred to with respect to any of the preferences. There is also an ordinal approach based on supersets. A history is optimal if no other history is preferred to a superset of the histories the first is preferred to with respect to any of the preferences.

The authors do not present a concrete system for finding or producing preferred histories. The language they created describes preferences that are temporally extended. It also is able to handle more complex aggregate values that many of the other temporally extended preference systems cannot.

3.1.4 PDDL

PDDL (Planning Domain Definition Language) and its latest version, PDDL3, defined in [42, 43] is a language that has been used in the International Planning Competitions. It has become a de facto standard language for defining planning domains and problems. PDDL has features that support a number of types of non-classical planning including planning with preferences.

PDDL uses first order logic. Goal preferences are formulas that a plan will ideally satisfy. Each formula has a numeric penalty specified. A plan is evaluated by summing the penalties of all of the preferences that are not satisfied in the goal state. PDDL also supports temporally extended preferences using linear temporal logic similar to $\mathcal{PP}$ and $\mathcal{LPP}$. A plan is then compared to other plans by the value of this metric.

3.1.5 Discussion

To compare and contrast the various preference systems we have presented we will focus on the complexity of the preferences that can be created, how the systems are able to relate the relative importance of multiple preferences, and the flexibility of the systems.

Delgrande et al. [28, 27] present a system that focuses on the theoretical side of preferences. They build a first order query language upon which a preference language is constructed. The linear temporal logic constructs used in other systems can be implemented in a first order logic system such as this; the LTL operator definitions are often given by way of FOL. A unique attribute of the preference language is that the preference formulas simultaneously involve aspects of two histories. If the formula is satisfied when applied over the two histories, then one is preferred to the other.

A strength of the work is their methodology that involves defining aggregate values by means of a recursive macro system. The preferences that this allows are clearly useful and are otherwise difficult or impossible to represent. The language does lack a mechanism to rank the importance of preferences, but as with the RKB system, different methods of moving from a set of preferences to an ordering histories allow a certain amount of flexibility.

Next we have the language $\mathcal{PP}$ that Son and Pontelli proposed [85]. The BDFs making up the building blocks of the system are defined using linear temporal logic. Remember that BDFs are then organized into APFs which give them an order and that the APFs can be combined with a couple constructs to form GPFs. These preferences are strictly ordinal, there are no values numeric or otherwise that are involved. The structures allow users to relate the relative importance of various preferences.

Bienvenu et al. [9] made some additions and changes to create their language $\mathcal{LPP}$. At the BDF level the important distinction over $\mathcal{PP}$ is that first order notation has been included. APFs are the same except that values are attached to the BDFs that assign a value of how preferred each BDF is. GPFs include disjunction, conjunction, and conditional means to combine APFs. AgPFs provide possibilities for combing GPFs that can affect how they are interpreted. This language is rich in the preferences that it can represent and has multiple constructs that can be used to denote the importance of certain preferences over others.

Another issue worth mentioning depends on whether a preference specification in a given language necessarily induces a total preorder on possible histories. In a total preorder, given any two histories, either one of the histories is preferred to the other or they are equally preferred. In a partial preorder it is possible for histories to be incomparable, there is no indication which is preferred. Languages that force a total preorder cause a problem with conditional preferences. Assume a user's only preference is conditional and is, "If A is true, than B is preferred to C". This preference only states a desired ordering of histories where A holds. It does not imply any preference over two histories, on where A holds and one where it doesn't. In either a total or partial preorder, $H_{AB} > H_{AC}$ will hold (the subscripts indicate which of A, B, and C hold in that history. However, in a total preorder $H_{\_} > H_{AC}$ or $H_{AB} > H_{\_}$ must hold. These are preferences among histories not implied by the user's single preference statement. The incomparability a partial preorder allows is required to accurately represent conditional preferences without adding additional unintended preferences to the induced ordering.

3.2 Ranked Preference Specification System

In our system, preferences are defined by creating a *ranked preference specification* (RPS). The preference system has a hierarchical structure similar to $\mathcal{PP}$ and $\mathcal{LPP}$. At the lowest level we have *queries* which express properties of a history. The next level of *preference*

orderings, or simply orderings, express which queries are more desirable to satisfy than others. Along with orderings that are analogous to the atomic preference formulas of $\mathcal{PP}$ and $\mathcal{LPP}$, there are *unbounded orderings* which can represent more complex preferences. The top level is a single *ranked preference specification*, which places orderings into a ranked knowledge base which defines relative priorities of the orderings.

3.2.1 Queries

Queries are the building blocks of our preference specification system. Queries are simply formulas that when applied to a history evaluate to *true* or $false$. They are used to define the criteria that are relevant to a user's preferences. For queries, we use the language from [27], with some minor modifications.

Here we introduce the concept of an *action signature* using the notation from Delgrande et al. [27] that is an adaptation of [41]. The action signature defines the set of fluents that describe the world, the values they can be assigned, and the actions. Our examples will still be motivated by STRIPS planning, but basing the language on an action signature allows it to be compatible with more complex specifications.

Definition 13. *An action signature Σ is a quadruple (D, F, V, A), where D is a set of value names, F is a set of fluent names, $V : F \to 2^D \setminus \emptyset$ assigns a domain to each fluent and A is a set of actions.*

Σ is propositional iff $D = \{0, 1\}$ and is finite iff D, F, and A are finite. A fluent $f \in F$ is propositional iff $V(f) = \{0, 1\}$.

For a propositional action signature $\Sigma = (D, F, V, A)$, fluent $f \in F$ is said to be *true* at state s iff $s(f) = 1$, otherwise f is $false$ at s.

Our language is a first order sorted language. We need to define temporal preferences and will have timestamp variables that take on the values of the timepoints in a history as well as object variables that range over the objects in a domain. We will assume that there is a finite set of timepoints and objects and thus any formula involving quantified variables can be reduced to a propositional logic formula.

Definition 14. *Let $\Sigma = (D, F, V, A)$ be an action signature and $n \geq 0$ a natural number.*

We define the query language as follows:

1. *The alphabet consists of*
   (a) *a set $\mathcal{V}$ of variables,*
   (b) *the set of integers,*
   (c) *the arithmetic function symbols '+' and '-',*
   (d) *the arithmetic relation symbol '<',*

(e) the equality symbol '=',

(f) the set $A \cup F \cup D$ of action and fluent names and values,

(g) the sentential connectives '$\neg$' and '$\supset$',

(h) the quantifier symbol '$\exists$', and

(i) the parentheses '(' and ')'.

2. *A* time term *is an arithmetic term recursively built from $V \cup \{0, \ldots, n\}$, employing $+$, $-$, $\cdot$, and parentheses in the usual manner.*

   A time atom *is an arithmetic expression of form $(t_1 < t_2)$ or $(t_1 = t_2)$, where t_1, t_2 are time terms.*

3. *A* fluent value term *is either a member of D, an expression of the form $f(t)$, where $f \in F$ and t is a time term, or an arithmetic term recursively built from value terms employing $+$, $-$, $\cdot$, and parentheses in the usual manner.*

   A fluent value atom *is an expression of the form $(v_1 = v_2)$ or $(v_1 < v_2)$, where v_1 and v_2 are value terms.*

   An action atom *is an expression of form $a(t)$, where $a \in A$ and t is a time term.*

   The set of atoms *is made up of the set of time atoms, fluent value atoms, and action atoms. An atom containing no variables is* ground.

4. *A* literal *is an atom, or an atom preceded by the sign $\neg$.*

5. *A* formula *is a Boolean combination of atoms, along with quantifier expressions of the form $\exists i$, for $i \in \mathcal{V}$, formed in the usual recursive fashion.*

   A formula containing no free variables is closed.

6. *A* query *is a closed formula.*

Beyond the primitive operators above, we define the operators $\vee$ and $\wedge$, as well as the universal quantifier $\forall$ in the usual manner. Next is the definition for the semantics of the language, defining when a query is true with respect to a given history.

Definition 15. *Let $H = (s_0, a_1, s_1, \ldots, a_n, s_n)$ be a history over Σ of length n,*
We interpret the language recursively as follows:

1. *If Q is a ground time atom or a ground fluent value atom,then $H \models_\Sigma Q$ iff Q is true according to the rules of integer arithmetic.*

2. *If t is a ground time term t takes its value according to the rules of integer arithmetic.*

3. *If v is a ground fluent value term:*

(a) *if* $v = f(t)$ *where* $f \in F$ *and* t *is a time term, then* $v = s_t(f)$ *if* $0 \leq t \leq n$ *and is undefined otherwise.*

(b) *if* v *is an arithmetic term then* v *takes its value according to the rules of arithmetic.*

4. *If* $Q = a(t)$ *is a ground action atom, then* $H \models_\Sigma Q$ *iff* $a = a_t$*, if* $0 \leq t \leq n$ *and is undefined otherwise.*

5. *If* $Q = \neg\alpha$*, then* $H \models_\Sigma Q$ *iff* $H \not\models_\Sigma \alpha$*.*

6. *If* $Q = (\alpha \supset \beta)$*, then* $H \models_\Sigma Q$ *iff* $H \not\models_\Sigma \alpha$ *or* $H \models_\Sigma \beta$*.*

7. *If* $Q = \exists i\, \alpha$ *for a timestamp variable* α*, then* $H \models_\Sigma Q$ *iff, for some* $0 \leq m \leq n$*,* $H \models_\Sigma \alpha[i/m]$*.*

A fluent value term of the form $f(t)$ or action atom, $a(t)$, is considered undefined if the time term t is either negative or beyond the end of the history. If a query is written that depends on one of these cases, producing a value of *true* or $false$ could cause unexpected behavior. We avoid this by using an undefined value. Any arithmetic that involves an undefined value is itself undefined. Boolean operators on undefined operands can be understood by considering undefined to be the value 0.5 with *true* and $false$ taking their usual values of 1 and 0. Then $\alpha \vee \beta = max(\alpha, \beta)$, $\alpha \wedge \beta = min(\alpha, \beta)$, and $\neg\alpha = 1 - \alpha$.

Some examples of queries in the vacation domain:

- Visit San Francisco at some time.

$$\exists t\, atCity(SanFrancisco, t) \tag{3.11}$$

- Visit Seattle at some time and San Francisco at a later time.

$$\begin{gathered} \exists t_1\, \exists t_2\ (t_1 < t_2)\ \wedge \\ atCity\,(Seattle, t_1) \wedge\ atCity\,(SanFrancisco, t_2) \end{gathered} \tag{3.12}$$

- Never visit a city in the desert.

$$\forall x\, \forall t\, \neg(atCity(x, t) \wedge inDesert(x)) \tag{3.13}$$

- Visit a city with a baseball team.

$$\exists x\, \exists t\, atCity(x, t) \wedge hasBaseballTeam(x) \tag{3.14}$$

3.2.2 Preference Orderings

Now that queries can be specified, we need a structure to specify which queries are preferred to others. For this purpose we next define preference orderings. Preference orderings define an ordering of alternative queries, each alternative being more preferred than the next. There will be two kinds of orderings, basic and unbounded. We begin with the basic type.

Definition 16. *A basic preference ordering o is a formula of the form,* $\alpha_0 \gg \alpha_1 \gg \cdots \gg \alpha_n$ *where* $\alpha_i, 0 < i < n$ *are queries with* $n \geq 1$.

The lowest indexed query that a history satisfies will be used to compare it to other histories. If a history does not satisfy any query in a basic ordering, then it is not comparable to other histories with respect to that ordering.

Definition 17. *History* H_1 *is at least as preferred as* H_2 *with respect to the basic preference ordering o,* $\alpha_0 \gg \alpha_1 \gg \cdots \gg \alpha_n$*, written* $H_1 \geq_o H_2$*, iff there exists i and j,* $(i \leq j \leq n)$ *such that*

$$\begin{aligned} H_1 &\models \alpha_i \wedge \forall t(0 \leq t < i),\ H_1 \not\models \alpha_t, \\ H_2 &\models \alpha_j \wedge \forall t(0 \leq t < j),\ H_2 \not\models \alpha_t \end{aligned} \tag{3.15}$$

Here is an example that expresses a desire to visit New York most, followed by San Francisco, followed by Toronto. (Queries are given names representative of their meanings for the sake of brevity)

$$visitNewYork \gg visitSanFrancisco \gg visitToronto \gg true$$

The *true* query at the least preferred position means that the three cities mentioned are preferred to all other cities. The *true* and $false$ literals can be used as the least preferred query to signify whether or not all plans can be compared with respect to that ordering. If the formula *true* is the least preferred query then it will be satisfied by all plans and all plans will have a value for comparison. If $false$ is used instead, plans that fail to satisfy all other queries in the ordering will be incomparable with all other plans with respect to that ordering. The $false$ query can be omitted without changing the meaning of a query. A common situation where this is useful occurs with conditional preferences. If we prefer β to $\neg\beta$ only when α is satisfied, the ordering $\alpha \wedge \beta \gg \alpha \wedge \neg\beta$ is sufficient. Adding a query of *true* would add a preference of α over $\neg\alpha$.

A flexible aspect of the orderings is that it allows simple construction of systems where transitivity holds or doesn't hold. If a system is desired where $\alpha \gg \beta$ and $\beta \gg \gamma$, but $\alpha \not\gg \gamma$, it can be done by creating two separate orderings, $\alpha \gg \beta \gg false$ and $\beta \gg \gamma \gg false$. Otherwise, the single ordering $\alpha \gg \beta \gg \gamma \gg true$ can be used.

A limitation of basic preference orderings is seen if we consider a preference to maximize the number of states when a fluent is true. For example, if a user wanted to remain in San

Francisco as long as possible. Queries can be written that are satisfied if the fluent is true 0 times, 1 time, 2 times, etc. The problem is that a basic preference ordering such as this would be difficult to create and compute due to its size. Such a preference represents a theoretically unbounded preference ordering. To deal with this sort of preference we present a construct to formally represent them in a compact manner.

Definition 18. *An unbounded preference ordering, o, takes the form* $max/min\,(V, \sigma, q, opt)$ *where* V *is a set of variables,* σ *is a time or fluent value term possibly involving members of* V*,* q *is a query involving all variables in* V *and opt is an integer or fluent value. Define* Θ *as the set of all grounding substitutions for* V*. Given a history* H*,* $o(H)$ *will represent the sum:*

$$\sum_{\{\theta \in \Theta |\ H \models q(\theta)\}} \sigma(\theta) \tag{3.16}$$

The value $o(H)$ that an unbounded ordering is computed by finding all substitutions over all variables in V that cause q to be satisfied. The values of the term σ when these substitutions are applied to it are summed. The value *opt* defines an optimal value. A preference between two histories is determined by comparing their sum values. If both match or better than the optimal value they are considered equally preferred with respect to the unbounded ordering.

Definition 19. *Given a maximizing unbounded ordering o,* H_1 *is at least as preferred as* H_2*, written* $H_1 \geq_o H_2$*, iff*

$$o(H_1) > o(H_2),\ o(H_2) < opt$$

Interpretation of a minimizing order is similarly defined by reversing the inequalities.

Again thinking of the vacation domain, let's say we want to maximize the amount of time we spend in New York. This can be done with the following unbounded ordering.

$$max(\{t\}, 1, atCity(NewYork, t), 4) \tag{3.17}$$

Given a history, only timestamps that satisfy $atCity(NewYork, t)$ will be applied to the variable t. Since $\sigma = 1$ the preference will simply counts the number of times this occurs. The optimal value is 4, so if a history has four or more appropriate states, then it is optimal with respect to the ordering.

The inclusion of this structure will save space in defining preferences as well as minimizing necessary computation. To illustrate this, consider a user who wants to maximize the number of states where the fluent f is true. This could be done by creating a query that is satisfied for every possible number of occurrences of that fluent. The formula for f being true in exactly two states is as follows.

$$\exists t_1 \exists t_2 \forall t_3 \, (t_1 \neq t_2 \wedge t_2 \neq t_3 \wedge t_1 \neq t_3) \supset \\ (f(t_1) \wedge f(t_2) \wedge \neg f(t_3)) \tag{3.18}$$

Similar formulas can be written for f being satisfied in more states, but quickly become unwieldy. Although we are considering a very simple preference using a basic preference ordering to represent it is quite difficult. However, when expressed with the unbounded ordering structure you get a preference such as (3.17).

Not only is this simpler to define, it is much simpler computationally. If the ordering were defined by writing out each query in a basic ordering, then it becomes possible to have to check each and every formula to evaluate a history with respect to the ordering. In many examples, including this one, calculating the value of the unbounded preference will require drastically less computation than for a similar basic ordering. To evaluate this unbounded preference, one variable ranges over every timestamp. In formulas in an equivalent basic ordering, multiple variables may have to range over all timestamps.

Here is a more complex unbounded preference ordering that aims to maximize the maximum value of a non-propositional fluent f over a history. We have to be careful here so that only one substitution for the variable gives the maximum value. Therefore the equation specifies the earliest occurrence of this maximum value.

$$max(\{t\}, f(t), \quad \forall t_1 \, (t_1 \neq t \wedge f(t_1) < f(t)) \vee \\ (f(t_1) = f(t) \wedge t < t_1), \; 1000) \tag{3.19}$$

Now, a final example that involves more than one variable. If we are in a domain that involves loading trucks for delivery, a preference to minimize the unused space over all of the trucks and states may be desired. If the fluent $unusedSpace(x, t)$ defines the amount of unused space for truck x in state t we can use the following.

$$min(\{x, t\}, \qquad unusedSpace(x, t), \\ unusedSpace(x, t) > 0, 0) \tag{3.20}$$

One preference that is commonly assumed in planning with and without preferences is that all else being equal, a shorter plan is preferred. A preference for shorter plans can be defined with an unbounded ordering. Thus we have a natural way of defining this preference instead of assuming it.

$$min(\{t\}, 1, true, 1) \tag{3.21}$$

We will also be able to define the importance of this preference with respect to other preferences by using the ranking system described below instead of always using it as a tie-breaker in cases where the other preferences are satisfied equally.

Unbounded orderings have been defined to maintain the flexibility of the preference system. It will be applicable to different query languages as long as those languages support first order notation. However, if the query language doesn't, only supporting typical propositional notation, then these unbounded orderings won't be appropriate.

Preference orderings only define qualitative preferences between histories. We say that one history is preferred to another if it satisfies a more preferred query, or has a better value with an unbounded ordering, but we don't look to the size of the difference to provide any meaning.

3.2.3 Ranked Preference Specification

We've established a query language and a structure to define preference orderings among queries. We now want to be able to specify relative importance among orderings. One ordering may represent a preference over which city to visit while another represents a preference over method of travel. It is natural for a user to think that one preference's satisfaction is more important than another. Maybe they prefer travel by train to airplane, but it isn't as important as making it to a preferred city.

Given a group of preference orderings, importance between these orderings will be defined by attaching ranks in the manner ranks are attached to formulas in ranked knowledge bases (RKBs) as seen in [21]. Orderings will have a rank assigned to them represented by a non-negative integer, thus organizing them into a *ranked preference specification* (RPS). When looking at a specified set of preference orderings we will look at them as a group of sets where each set consists of all orderings given a specific rank.

Definition 20. *A ranked preference specification consists of a set of sets of orderings* $(O_0, O_1, \ldots O_n)$*. Ordering* o *is at least as important as* o' *iff* $o \in O_i$*,* $o' \in O_j$*, and* $i \leq j$*.*

If in our example there were three preference orderings that represented preference over which city to visit, C, how to travel, T, and what attractions to visit, A, and the city to visit and attractions visited are most important while method of travel is less so, an appropriate RPS would be:

$$O_0 = \{C, A\}, O_1 = \{T\}$$

Note that increasing ranks represent decreasing importance. As with the ordering indexes, these ranks represent purely ordinal difference in importance. An ordering in a lower rank is simply considered more important, but there is no meaning in the numeric values of the ranks beyond that.

Our system does not specifically define when one history is preferred to another with respect to a RPS. The strategy we use is described in the next section, but varying strategies that can be created given the information in a RPS. Depending on the strategy used, the meaning of the ranks could change, however, the intention in our definition is that if one history is preferred to another with respect to the orderings at a given rank, then it would be preferred irrespective of how the histories satisfy orderings in less important ranks.

Although the system is ordinal it is possible to create a RPS that embodies a meaning that is associated with using quantitative values. Let's assume a user has a choice between visiting San Francisco, New York, and Toronto and a choice between traveling by bus or train. If a user strongly prefers a trip to Montreal over Chicago or Detroit they might give Montreal a value of 8, Chicago, 4, and Detroit, 3. They might then give travel by train a 6 and bus a 4. Traveling to Montreal is clearly the most important and it might initially seem difficult to use our qualitative system to express it.

However, if instead of using a single preference ordering to represent the preference among cities, two orderings are used, it becomes possible. The following RPS is sufficient.

$$O_0 = \{visitMontreal \gg true\}$$
$$O_1 = \{visitChicago \gg visitDetroit \gg true,$$
$$rideTrain \gg rideBus \gg true\}$$

The preference for Montreal is clearly made stronger by moving it to a more important rank. If there are queries in an ordering that a user considers significantly more or less preferred than others, this can be represented by breaking an ordering into multiple pieces as was done with this example.

3.2.4 Defining Preferred Histories

We now have a system by which a user can describe their preferences. With the ranks of preference orderings and levels of satisfaction within the orderings there are numerous options on how to do this. Depending on the application or setup of the preferences, one method of comparison may be more appropriate than another. It may be more important to emphasize satisfaction of the entire set of preferences at a given rank or focus more on individual preferences. It might be more important to satisfy the most preferred queries in an ordering or to strongly avoid the least preferred queries. Some general strategies for doing this are discussed in [21]. They would need modification before being applied to RPSs, but the strategies are relevant. We present definitions for two possible approaches for comparing histories.

Definition 21. *Given histories H_1 and H_2 and RPS P, H_1 is at least as preferred as H_2, $H_1 \succeq_P H_2$, if for no i*

$$\left|\{o \mid o \in O_i, H_2 \geq_o H_1\}\right| > \left|\{o \mid o \in O_i, H_1 \geq_o H_2\}\right|$$

such that for all j, $0 \leq j < i$,

$$\left|\{o \,|\, o \in O_j, H_2 \geq_o H_1\}\right| = \left|\{o \,|\, o \in O_j, H_1 \geq_o H_2\}\right|$$

Define a strict preference, $\succ$, and equivalence, $\approx$, in the typical manner.

The first, Definition 21, uses a cardinality based approach. Given two histories, count how many times each history is preferred to another with respect to the orderings at a given rank. Whichever has more 'victories' at a given rank is preferred. If they tie, then the orderings at the next rank are examined. If the histories are tied in the end, then they are considered equally preferred. This approach induces a total preorder over all histories.

Definition 22. *Given histories* H_1 *and* H_2 *and RPS* P, H_1 *is at least as preferred as* H_2, $H_1 \succeq_P H_2$, *if either for some* i,

$$\exists o \in O_i (H_1 >_o H_2)$$

and for all j, $0 \leq j < i$,

$$\nexists o \in O_j (H_2 >_o H_1)$$

or *if for no* i,

$$\exists o \in O_i (H_2 >_o H_1)$$

and for some j,

$$\exists o \in O_j (H_1 \geq_o H_2)$$

Define a strict preference, $\succ$, and equivalence, $\approx$, in the typical manner.

An alternative approach, Definition 22, is to find the lowest rank where a history is strictly preferred to the other with respect to at least one ordering. If a history is strictly preferred to another with respect to a single ordering at that rank, then it is at least as preferred as that history. The relation has been defined so that if two histories are incomparable with respect to all orderings in an RPS, they will be incomparable with this relation. If two histories are comparable with even a single ordering, they will be comparable with this relation. If all orderings allow incomparability, this approach could lead to a great deal of incomparability between histories.

3.2.5 Discussion

Next, we discuss the advantages and disadvantages of the ranked preference specification system relative to the other planning specific preference languages. An RPS allows incomparability and allows it to be controlled. The unbounded orderings allow useful preferences that would otherwise be impractical to be used. The ranked knowledge base structure used to prioritize the orderings is simple but robust. However, using an ordinal approach to

interpreting an RPS can make it more difficult to compare plans or efficiently compare the quality of nodes during a planner's search.

Incomparability

As stated in Section 3.1.5, we believe that allowing incomparability between histories is important in properly representing preferences. Not only does a ranked preference specification allow incomparability, it does so in a more easily controlled manner. The $\mathcal{LPP}$ and PDDL languages don't allow any incomparability, inducing a total preorder on all histories. $\mathcal{PP}$ allows incomparability through the use of its version of conjunction and disjunction in general preference formulas, however, it is cumbersome. Let's look at specifying a single conditional preference. If in New York, it is preferable to attend the theater than to not attend. A history that includes travel to New York and attending the theater should be preferred to traveling to New York and not attending. To keep things clear we'll represent the relevant queries or BDF possibilities as $NYTheater$, $NYNoTheater$, and $NoNY$. An RPS with the following ordering would be sufficient.

$$NYTheater \gg NYNoTheater \gg false \tag{3.22}$$

It is possible to represent this with $\mathcal{PP}$, but it requires combining to APFs with conjunction in an unintuitive manner.

$$\begin{aligned}&(NYTheater \lhd NYNoTheater \lhd NoNY)\,\&\\&(NoNY \lhd NYTheater \lhd NYNoTheater)\end{aligned} \tag{3.23}$$

This is cumbersome, especially considering the simplicity of the preference being modeled. This will only get worse when involving more or more complex APFs. The orderings of an RPS allow incomparability and allow it to be controlled with greater ease. Additionally, the strategy chosen for interpreting the RPS as a whole can allow for incomparability even if none of the internal orderings do. By allowing incomparability in the orderings, the system allows for more precise control of it than the other available languages such as $\mathcal{PP}$ where its version of orderings must be combined to introduce incomparability.

Unbounded Orderings

The unbounded orderings create an easy way to specify preferences that would otherwise be much more difficult in writing and in computation. These orderings can be used to sum or find the maximum of numerically valued fluents. Even restricted to propositional fluents unbounded orderings can facilitate interesting and useful preferences that would otherwise be impractical if not impossible.

Within a propositional domain, an unbounded ordering can count the occurrences of a query in a history and either prefer to minimize or maximize that count. Someone may wish to attend as many events as possible on their vacation. It can be used to prefer to have a query satisfied as early or as late as possible within a history. The usual assumption that a shorter plan is preferred can be precisely specified and have its importance defined with respect to other preferences.

Flexibility

The final advantage of the RPS system is the flexibility with which multiple preferences can be combined and prioritized. The rankings allow a set of orderings to be clearly categorized based on their importance. When the available strategies for interpreting a given RPS are considered, the system allows a great deal of flexibility while containing only one complex construct, the unbounded ordering.

The combination of these factors is what makes the ranked preference specification system a useful, novel approach. Preferences can be specified that will induce a total preorder over histories with no incomparability or that allow carefully controlled incomparability. The unbounded orderings allow preferences to be specified that are at best impractical in the other languages that allow prioritization of the preferences. Finally, the system does these things while remaining easy to use.

Efficiency

The disadvantage of the ranked preference specification is its use of an ordinal approach. Using a cardinal approach that assigns a value to each plan allows the plans to be compared more efficiently as only their values need to be compared. If using a ranked preference specification with an ordinal interpretation it is possible that two plans must be compared with respect to every ordering in the specification. When considering the planning process, a similar difficulty will generally arise. With a cardinal approach, a heuristic would need to estimate the value of a plan a given search node would lead to. This value allows a simple comparison to an alternative node's value. With a ranked preference specification, a heuristic has to estimate how the orderings can be satisfied, but then also has to do a more complex comparison to decide which node is more promising. While the ordinal approach does allow preferences to be more accurately modeled, it does lead to difficulties in running an efficient planning search.

Chapter 4

Landmark Heuristics for Planning with Preferences

In this chapter we move to the actual planning process. We begin by covering the work that has been done creating planners that handle preferences. The next section gives a background on the use of landmarks in planning. Then we show how landmarks can be adapted for preferences and present heuristics that utilize landmarks for planning with preferences.

4.1 Related Work

In this section we cover the work that has been done in planning with preferences with regard to implementing and testing planners. We will break the work into two areas, planners that utilize search strategies similar to our work, and planners that use problem translation. First, a couple brief definitions for a preferences planning problem and an optimal plan.

Definition 23. *A preferences planning problem consists of a planning problem, P, and a preference specification, $Pref$, that defines an ordinal preference relation, $\geq_{Pref}$, over all plans for P.*

Definition 24. *Given a preferences planning problem $(P, Pref)$ a plan p is optimal if there does not exist plan p' such that $p' >_{Pref} p$.*

A preferences planning problem simply consists of a STRIPS planning problem and a preference specification. A plan is optimal if no other plan is preferred to it with respect to the preference specification.

4.1.1 Problem Translation Based Planners

As discussed in the second chapter, classic planning problems have been translated into a different type of problem and then solved in the new form. If a solution is found, a plan

is available and can be read from that solution. We mentioned work in translating a planning problems to answer set programs, constraint satisfaction problems, and satisfiability problems. Planning problems involving preferences have also been translated into all three.

The authors of the preference language $\mathcal{PP}$, described in Section 3.1.1, [85] show how to convert problems written with $\mathcal{PP}$ into answer set programs. However, they do not present experiments or results based on their strategy.

Constraint satisfaction programs were used for preferences that were specified with TCP-nets [19]. They build on the original work converting a planning problem to a CSP [30]. The preferences are defined by using a TCP to define the preferred values and relative importance of possible goal states. The TCP-net is used to direct the CSP solver to search for plans that optimally satisfy the given preferences. Otherwise, the solver continues looking for more preferred plans.

There also has been work with preferences in planning as a satisfiability problem [44, 45, 71]. The approach is similar to that just mentioned for CSPs. The preferences are encoded into the satisfiability formula such that a variable represents whether that preference has been satisfied. The solver is directed to first attempt to find a plan that satisfies every preference. If this fails it continues searching for plans that satisfy fewer satisfied preferences until a plan is found. The planner is tested on problems specified in PDDL with goal preferences.

4.1.2 Search Based Planners

We begin by mentioning work that doesn't focus on the details of the search, but makes use of an available search based planner. The ranked knowledge base preference language previously discussed [21] is used to specify preferences. First, the planner is called with the initial problem searching for any plan regardless of how it satisfies the preferences. Once it returns a plan, the planner is called again, but searches for a plan that is strictly better than the best solution found so far [37]. This generate and improve strategy is an iterative version of an anytime search.

The planner SGPlan is from Chen, Hsu, and Wah [91, 23]. The planner's strategy is to partition the original planning problem into a set of relatively independent subproblems. The subproblems can generally be solved more quickly than the entire problem. The subproblems are given to the Metric-FF planner, [56] and the returned solutions are combined to make up the final plan. To partition the original problem SGPlan analyzes the goals, preferences, and actions needed to achieve them. It ignores preferences that conflict with the goals and analyzes how closely related the remaining goals and preferences are. Using this information it separates portions of the goals and preferences that can be solved independently with a relatively small amount of interference with each other. These solutions are then combined through an algorithm that resolves any conflicts among them.

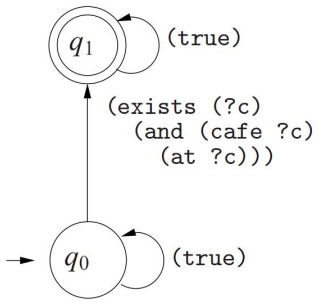

Figure 4.1: The automata for the BDF $(\exists c)\, cafe(c) \wedge$ **eventually**$(at(c))$

As to heuristic-based preference planning we begin with the planner, **PPLAN** from [9, 10]. **PPLAN** accepts temporally extended preferences in the $\mathcal{LPP}$ language, Section 3.1.2. Recall that given a plan, a preference specification applies a value to that plan. The heuristic is optimistic and assuming that any preference that may be satisfied in the future will be. The heuristic value assigned to a state is the best possible value that could be attained satisfying every preference that hasn't been contradicted to that point. As **PPLAN** expands states, a successor state can never have an improved heuristic value and is therefore admissible. Using a best-first search, the planner is guaranteed to find the optimal plan, if a plan is found. **PPLAN** uses a heuristic that only involves preferences and not the goal of a planning problem. It relies on the specified preferences leading it to a goal state.

In [3, 5, 4] Baier, Bacchus, and McIlraith continue the work in using heuristics to guide a search for optimal plans. They use two strategies to improve performance. First, they simplify the preferences reducing a preference specification to a single atomic preference formula. A general preference formula can be converted to an APF with only linear expansion for conditional and disjunctive GPFs, but exponential for conjunctions.

After this has been done each temporally extended BDF is used to create a nondeterministic finite state automata which has one state that represents satisfaction of the BDF and other states which represent progress toward satisfaction. Transitions between states are defined by appropriate first order formulas. In cases where the BDF has an existentially quantified variable, there will be a copy of the automata for each instantiation of the variable(s).

Figure 4.1 shows an example of an automata for a BDF that is satisfied if the user visits a cafe at any point, $(\exists c)\, cafe(c) \wedge$ **eventually**$(at(c))$. As soon as a user is present at a cafe the automata would switch from the initial state to the accepting state. Once automata have been created for each of the BDFs, new fluents are added to the domain to represent the current state of each automata throughout a history and other adjustments are made to the domain and planner so that these new fluents are appropriately updated. Therefore, each BDF can be seen as a preference for its respective automata to be in the accepting state when the plan is complete. In this way the temporally extended preferences

are essentially converted to goal preferences and heuristics can be applied to them on that basis.

Testing involved heuristics based on the altered domain that focus on both the goal and the preferences including variants of the FF heuristic and PPLAN's optimistic heuristic. The heuristic for PPLAN only focuses on preferences and is admissible. The new heuristics are not, but are generally able to outperform the admissible heuristic because the optimism of the admissible heuristic doesn't differentiate between partial plans that are close to and far from satisfying a preference. The more recent work, [4], uses preferences defined in PDDL as opposed to $\mathcal{LPP}$. The planner created also takes advantage of an anytime search strategy using an iterated approach. When a plan is found, the search will start again, but will prune the search space based on the best plan so far attained.

We also presented a planner that took preferences specified in our Ranked Preference Specification system [83]. The planner was tested with two heuristics, an admissible optimistic heuristic similar to that of PPLAN as well as an inadmissible greedy heuristic. The greedy heuristic directed the search towards states that were closer to satisfying the most preferred queries in a preference ordering. Testing showed that the greedy heuristic was more effective [83].

4.1.3 Partial Satisfaction Planning

Partial Satisfaction Planning or Oversubscription Planning [1] is an area of planning research related to planning with preferences. An oversubscription problem contains a set of goals each with a value assigned. The aim of an oversubscription planner is to maximize the value of achieved goals. The majority of work involves problems where actions have an associated cost. In this case, the aim is to maximize the goals achieved while minimizing the cost of the plan. These goals could be viewed as preferences. In this view, a planning problem has no goals in the classical planning sense. The majority of work in this area has followed a similar approach. In [87, 6, 8, 7, 84, 39, 76] planners use a relaxed planning graph based heuristic to guide their planner. Work from [95] uses a learning approach to identify promising features in the search space and calling a planner to search starting at those promising points. Abstraction heuristics have also been utilized [73].

4.2 Landmarks Background

In this section, we briefly move away from preference planning and explore the work done related to landmarks. Landmarks are a tool developed for classic planning that we will adapt for planning with preferences. Hoffman, Porteous, and Sebastis proposed the idea of landmarks and explored one method of using them to improve planning in [78, 55]. The idea is an extension of work by Cheng and Irani [58, 24]. More recently, the most notable

work with landmarks has been done by Richter, Westphal, and Helmart in [79] with the first two continuing the work in [81] where they present their LAMA planner.

The concept of a landmark in planning is intuitive and is analogous to the use of landmarks for navigation. While such landmarks are usually something like an unique building or something else specific to a small area, let's use an example that more closely mirrors the use of landmarks in planning; let's look at an example on a larger scale. If you are driving by car from Seattle to New York, there is a vast array of routes you can take. However, whether you take a direct route across the United States or a roundabout route through Canada or Mexico, there are two geographic features that cannot be avoided that give you an idea of how close to your destination you are, the Rocky Mountains, and the Appalachian Mountains. Every route, fairly early on, will encounter the Rocky Mountains and every route, fairly close to the destination, will encounter the Appalachian Mountains. These two mountain ranges act as landmarks. So, if you are looking at possible routes, you can discard any route that doesn't first cross the Rockies and then pass the Appalachians.

Landmarks in planning follow this approach. Given a planning problem, *landmarks* are logical formulas that must be true at some point in every plan. Since they must be satisfied before a plan can be completed they can be used to guide a planner towards histories that are likely to lead to a complete plan. Along with the landmarks themselves, it is often possible to find orderings between landmarks. An ordering states that one landmark will be passed before another in every plan and therefore the planner can ignore the second until the first has been satisfied.

Definition 25. *Given a planning problem, P, a logical formula l is a landmark if and only if l is satisfied in at least one state in every plan.*

It is evident from this definition that all logical facts and formulas that are satisfied in the initial state or the goal state are satisfied in every plan and are therefore landmarks. Due to this they may be referred to as *trivial landmarks*. We will refer to the trivial landmarks that are part of the goal as *goal landmarks*.

There has also been work focusing on *action landmarks* [12, 15, 51, 14, 48]. An action landmark is an action that must be executed in every plan. There is a close relation between fluent-based landmarks and action landmarks. If an action is a landmark then all of its preconditions and effects must be part of every plan and are landmarks. Similarly if a landmark can only be satisfied by a single action, then that action is an action landmark.

We will use a simple problem from the logistics domain that is illustrated in figure 4.2 as an example. There are five locations with A through D in one city and E in another. A truck can only travel within the city along the edges among A through D. A plane can fly between airports at C and E. The goal is to deliver the package at B to E. The package can be moved by loading it onto the truck or the plane. There are several landmarks in this planning problem. For example, in any plan the truck must move to location B so it

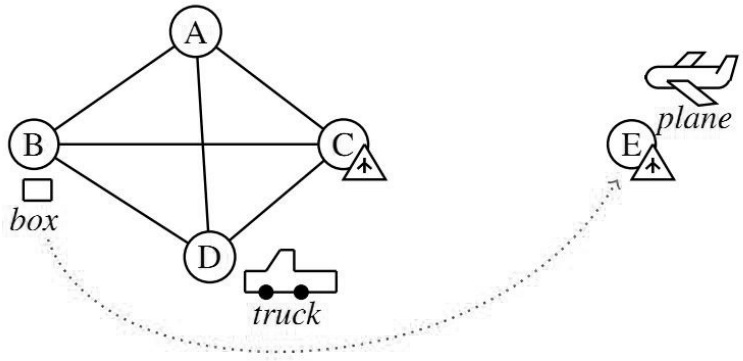

Figure 4.2: A planning problem from a logistics domain where a package needs to be taken from location B to location E.

is able to pick up the package. The package must then be driven to C before being loaded onto the plane and flown to E, giving us some simple landmarks. Therefore, we'll have such landmarks as $truckLocation(B)$, $packageLocation(onTruck)$, $truckLocation(C)$, and so forth. If looking for action landmarks, $loadPackageOntoTruck$ is one example. Its effect, $packageLocation(onTruck)$ is a clear landmark we have already discussed.

Different types of orderings between landmarks have been found to be useful. These orderings will state to varying degrees of specificity when one landmark must occur relative to another. We begin with three types of orderings that are guaranteed to hold in every plan.

Definition 26. *Given landmarks l_1 and l_2*

- *There is a* natural ordering *of l_1 before l_2, $l_1 \rightarrow l_2$, iff in all plans where l_2 is true in s_j, l_1 is true in s_i, $i < j$.*
- *There is a* necessary ordering *of l_1 before l_2, $l_1 \rightarrow_n l_2$, iff in each plan where l_2 becomes true in s_i, l_1 is true in s_{i-1}.*
- *There is a* greedy-necessary *ordering of l_1 before l_2, $l_1 \rightarrow_{gn} l_2$, iff in each plan where l_2 **first** becomes true in s_i, l_1 is true in s_{i-1}.*

Necessary orderings are the most specific, followed by greedy-necessary orderings and natural orderings. A necessary ordering implies a greedy-necessary ordering which in turn implies a natural ordering. A natural ordering simply states that one landmark must be satisfied at some point before another. In a necessary ordering the earlier landmark must always be satisfied immediately before the latter while in a greedy-necessary this is only true the first time the latter landmark becomes true.

Looking at our logistics problem we see examples of each type of ordering. The landmark $packageLocation(C)$ is naturally ordered before $packageLocation(E)$ because there is no possibility of the latter being true before the former. A greedy-necessary ordering is $planeLocation(C)$ before $packageLocation(onPlane)$ because in any plan the plane must be at C for the package to be loaded onto the plane. To extend this to a formula, it is simple to see that we could substitute $packageLocation(C) \wedge planeLocation(C)$ for $planeLocation(C)$ in that ordering. It is not a necessary ordering because an overly complicated plan could have the package unloaded and reloaded onto the plane at E. A necessary ordering would be that the truck and package must be in the same location immediately before $packageLocation(onTruck)$ can be satisfied.

These orderings are all strict; they will hold in all plans. Further orderings were proposed that do not strictly hold, but are usually useful [66, 55]. These are called *reasonable orderings*. These orderings try to avoid needless actions where a landmark might become true but must then be made false again to satisfy another landmark. Given landmarks l_1 and l_2, the course of events that should be avoided is l_2 followed by $\neg l_2$, l_1, and l_2. It has also been found to be useful to find further reasonable orderings based solely on the orderings already found.

Definition 27. *Given landmarks l_1 and l_2*

- *There is a* reasonable ordering *of l_1 before l_2, $l_1 \rightarrow_r l_2$, if in each plan where l_2 becomes true in s_i and l_1 first becomes true in a later state s_k, then in some state, s_j, $i < j < k$, l_2 must be false.*

- *A plan* obeys *a set of orderings, O, if for every ordering $l_1 \rightarrow l_2$ in O, regardless of type, if l_1 is first true in s_i then l_2 is not true in any state s_j, $j \leq i$.*

- *There is an* obedient-reasonable ordering *of l_1 before l_2 with respect to a set of orderings O, $l_1 \rightarrow_r^O l_2$, if in each plan that obeys O and l_2 becomes true in s_i and l_1 becomes true in s_k, $k > i$, then l_2 must not be true in s_j, $j \in \{i+1, \ldots k\}$ and then become true in state s_l, $l > k$.*

Continuing with the example, a reasonable ordering of $packageLocation(onTruck)$ before $truckLocation(C)$ is appropriate. The truck could drive to C before moving on to B to pick up the package but doing so wastes effort; it must drive back. It is important to note that there is a possibility that there may be no plans that obey a reasonable ordering so they are unsound, but have been found to be useful.

A set of landmarks and orderings can be viewed as a *landmark graph*, (L, O). An example is shown in Figure 4.3. Vertices represent landmarks, and edges represent known orderings. By their definition, a set of natural, necessary, and greedy-necessary orderings will never produce cycles. Cycles are possible with respect to reasonable and obedient-reasonable

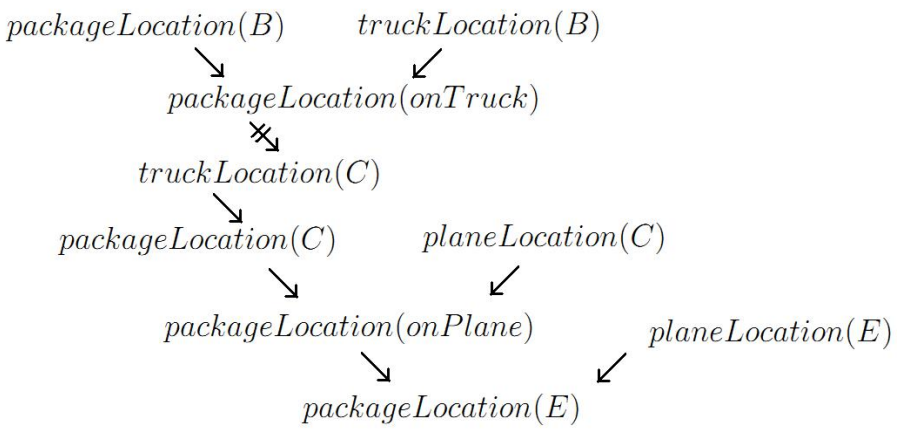

Figure 4.3: A landmark graph displaying landmarks and orderings from the example in figure 4.2. The plain arrows represent strict orderings while the arrow with the crosshatches represents a reasonable ordering.

orderings. If these orderings are used, the landmark graph needs to have the cycles broken before it is used for planning.

4.2.1 Discovering landmarks and orderings

Different methods for maximizing the number of useful landmarks found have been explored. Hoffman et al. proved that it is PSPACE complete to determine whether a given formula is a landmark. Therefore an exact method for discovering all landmarks is unrealistic. All of the methods discussed here only produce correct landmarks but none are able to find all of the landmarks for a problem.

Though landmarks are defined generally as a propositional formula, methods for landmark discovery focus on simple landmarks. Unless otherwise stated in this section a landmark is considered to be a fluent atom.

We will begin with the landmark discovery methods used in the LAMA planner [81]. This method looks at known landmarks that are not satisfied in the initial state and uses a backchaining method to find landmarks that must occur before them. The process starts with a goal landmark, l_1. It then looks at all actions that could possibly satisfy l_1. If all of these actions share a precondition x then x must be satisfied before l_1 can be satisfied. Therefore x is a new landmark and the ordering $x \rightarrow_{gn} l_1$ will hold. The procedure repeats until no more landmarks are found.

It is straightforward to find *disjunctive landmarks* with this method. Assume a landmark l_1 has two actions that can possibly achieve it, cause it to be satisfied. The first has preconditions x and y, the second has preconditions x and z. x is therefore a landmark but we also know that either y or z must be satisfied before l_1. Therefore $y \vee z$ is a landmark. The number of disjunctive landmarks grows exponentially as the number of possible achievers

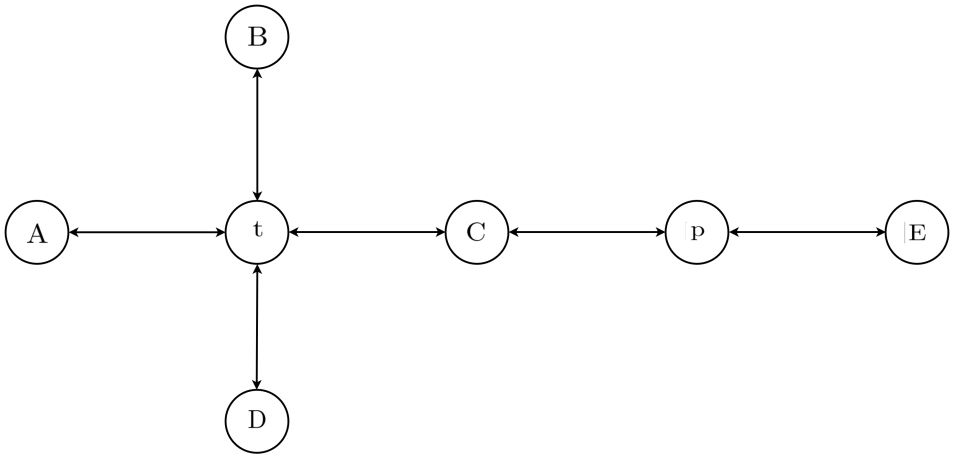

Figure 4.4: The domain transition graph for packageLocation from the logistics problem in figure 4.2. A,B,C,D,E symbolize the appropriate locations and t and p stand for $onTruck$ and $onPlane$. The edges denote which transitions are possible based on the possible actions.

grows. Therefore limiting disjunctive landmarks is necessary. First, a disjunctive landmark is only considered across preconditions that stem from the same fluent. Second, if there is a simple landmark x, then any disjunctive landmark involving x will be ignored.

A *restricted relaxed planning graph* is a variant of the relaxed planning graph previously discussed in Section 2.1.2 and is used to find *possible first achievers*. A first achiever of a landmark is an action that could cause that landmark to be satisfied for the first time. A restricted relaxed planning graph for landmark l_1 additionally removes any actions that would add l_1. Any action that achieves l_1 and could be executed based on the final level of facts in the restricted relaxed planning graph is a possible first achiever. As stated, any fact that is a precondition of all of these possible first achievers is a landmark that has a greedy-necessary ordering before l_1. The restricted relaxed planning graph may find more possible first achievers than actually exist in the original problem, but this will only reduce the number of landmarks found. All the landmarks that are found are correct, but some to be missed.

A second method used by the LAMA planner to discover landmarks makes use of a different type of graph that also represents how facts can be changed by the available actions. The planner reformulates a STRIPS planning problem into an equivalent problem that converts fluents into variables [59]. For example, in our logistics problem we wouldn't have several facts, $packageLocation(A)$, $packageLocation(B)$, $packageLocation(onTruck)$, etc., we'd have a variable $packageLocation$ that can be assigned the values A, B, $onTruck$, etc. A *domain transition graph*, DTG, is a graph with nodes for each possible value and directed edges for each possible transition. Any node that is on every path from the variable's assignment in the initial state to a variable assignment that is known to be a landmark must also be a landmark that is naturally ordered beforehand. Figure 4.4 shows the DTG

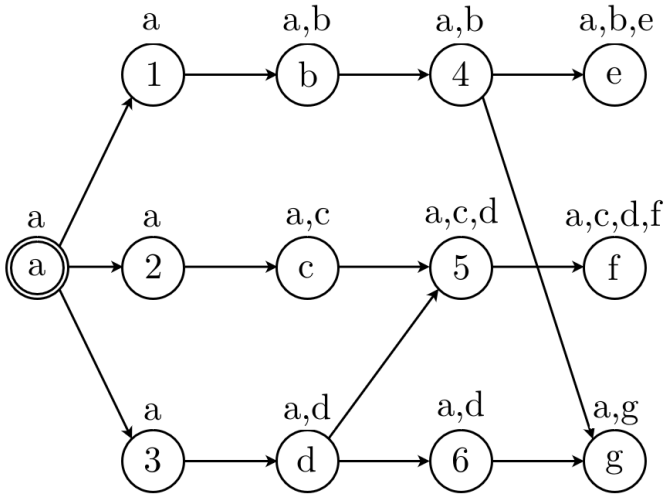

Figure 4.5: A relaxed planning graph for a hypothetical planning problem. Letters symbolize facts and numbers symbolize actions.

for *packageLocation* for the problem. The initial value is A and the goal value is E. Clearly the values C, $onTruck$, and $onPlane$ must be assigned to the variable to reach E supplying three new landmarks and orderings among them.

Given a landmark l_1, its restricted relaxed planning graph is used again. Any landmark l_2 that cannot be satisfied by the actions executable from the final layer of facts in this graph cannot be satisfied before l_1, and give us the natural ordering $l_1 \rightarrow l_2$.

The final process used by LAMA finds reasonable and obedient reasonable orderings. It does so by examining the known orderings and using chains of those orderings to imply reasonable orderings. It also checks whether pairs of landmarks are mutually exclusive and adds reasonable orderings to try and avoid a chain of events such as l_2, $\neg l_2$, l_1, and l_2 that signals wasted effort.

Another method for landmark discovery was proposed by Zhu and Givan [96] that involves propagation along a relaxed planning graph. They define a category of landmarks, *causal landmarks*, that they believe are the only truly important landmarks. A landmark is causal if it is the precondition for an action in every plan. These landmarks always signal necessary progress towards the goal. They consider non-causal landmarks to be "accidental" and possibly misleading to heuristics, as achieving them does not necessarily indicate progress towards the goal.

As a relaxed planning graph is built up level-by-level, fact labels are propagated forward based on the precondition logic used in the landmark discovery by LAMA. When the graph reaches the goals, the list of landmarks has been built up for the goals. See the labels in Figure 4.5 for an example of how this works. Every initial state fluent node is labeled with itself. An action node is labeled with the union of all its preconditions. If this action is

required, all of fluents represented by its labels are therefore landmarks. Fluents that are effects of actions are labeled with themselves and the intersection of the labels of all actions that can achieve the fluent. The graph is built until goal nodes are reached. The labels attached to the goal nodes are landmarks. Natural and greedy-necessary orderings can also be read from the order in which the labels accumulated.

Advancing this method is Richter et al. [65]. They develop a method for landmark discovery that uses an and/or graph to propagate information. The authors also show how a transformed version of the original problem in which a single fact represents sets of facts from the original problem. They limit the size of these sets to two. The landmark discovery procedure is then performed on the new problem with the compound facts. When one of these compound facts is found to be a landmark, the conjunction of the facts from the original problem is a landmark. Also, unlike the relaxed planning graph procedures used, this encoding of the problem includes some of the results of deletes that are ignored in the other methods. Therefore, more landmarks can be found. Even though their method found more landmarks, it has some serious trade offs. It takes more computation to find the landmarks than the set of methods from LAMA, and gains in planning efficiency generally did not make up for the lost time. We omit the details of this approach as it can't be applied to planning with preferences. The reasons for this will be discussed later in Section 4.3.

4.2.2 Use of landmarks during planning

Once a set of landmarks and orderings on them has been found, the question then becomes, how can they best be used to improve planning efficiency? The first method used follows the intuition used in navigation, driving to the nearest landmark before heading for the next one and so forth. A planner would be called repeatedly only being asked to plan to the earliest unsatisfied landmarks according to orderings. This method did produce plans more quickly, but the plan length tended to be longer than necessary [78, 55]. This is due to the planner missing chances to satisfy multiple landmarks at once by focusing on them individually. Vernhes et al. explored more sophisticated means of using the landmarks to split a problem into smaller problems [88].

Another strategy has been to enrich the planning problem provided to a planner by incorporating landmark information into it. In this way planners can use their own strategies and algorithms to take advantage of the landmark information. One approach involves encoding landmarks and their ordering into temporal logic formulas [90]. These formulas are then encoded as finite state automata whose states can be altered by the various actions. Goals added to the problem will lead the planner to the automata states that satisfy the temporal logic formulas. A second method adds a fluent for each landmark representing whether or not the landmark has been satisfied [32].

Landmark based heuristics

Landmarks came into fashion with their use in the LAMA planner. It used a heuristic that evaluates a state by counting how many landmarks are left that need to be satisfied in successor states before the goal can be satisfied. This heuristic is the basis for our work in adapting preferences.

Given a landmark graph, (L, O), the set of goal landmarks, G, and a history h of length n, $(s_0, a_1, s_1, \ldots, s_{n-1}, a_n, s_n)$, the sets *Accepted* and *Required* are defined over the states in h as follows.

$$Required(s) = (L \backslash Accepted(s)) \cup ReqAgain(s) \tag{4.1}$$

$$Accepted(s_0) = \{l \in L \mid s_0 \models l \text{ and } \nexists(l' \rightarrow_x l) \in O\} \tag{4.2}$$

$$Accepted(s_i)(i > 0) = Accepted(s_{i-1}) \cup \{l \in L \mid s_i \models l \text{ and } (\forall(l' \rightarrow_x l) \in O : l' \in Accepted(s_{i-1}))\} \tag{4.3}$$

$$ReqAgain(s_i) = \{l \in Accepted(s_i) \mid s_i \not\models l \text{ and } (l \in G \text{ or } (\exists(l' \rightarrow_{gn} l) \in O : l' \notin Accepted(s_i)))\} \tag{4.4}$$

The set $Required(s)$ is formed by combining all landmarks from L that haven't been accepted yet, with landmarks that have been accepted, but will need to be satisfied again in later states.

Computing the set of accepted landmarks for the initial state, $Accepted(s_0)$, is done by finding all landmarks that are satisfied by s_0, but discarding any that have an unsatisfied predecessor in the landmark graph. These are discarded because they may become unsatisfied before their predecessors can be satisfied. Once a landmark has been accepted, it is accepted in all succeeding states. For later states, $Accepted(s_i)$, is found by combining the accepted landmarks from its preceding states with landmarks that are now satisfied and don't have any predecessors in the landmark graph that have not been accepted. An accepted landmark is added to $ReqAgain(s_i)$ if it has been accepted, is not satisfied by the current state, and is either part of the goal or has a predecessor in a greedy-necessary ordering that is not satisfied by the current state.

Definition 28. *Given a state s, define the landmark-count heuristic h_{LM} to be the size of the set of landmarks that are required to be satisfied in successor states before the goals can be satisfied.*

$$h_{LM}(s) = |Required(s)| \tag{4.5}$$

To count how many landmarks still need to be satisfied involves not only counting the landmarks that have been satisfied by a given state, but also examining the orderings among

landmarks. The orderings can signal that a landmark that is currently satisfied will become unsatisfied and then need to be satisfied again before the goal is reached.

The landmark-count heuristic estimates the number of actions needed to reach the goal. It is inadmissible because it can overestimate this number. The heuristic assumes that it will take one action to satisfy each needed landmark, but it is possible that it will take less than one action per remaining landmark to reach the goal.

The heuristic was slightly modified for use with planning with action costs. To do this, each landmark is assigned a weight that is equal to the lowest-cost action that could cause the landmark to be satisfied.

The rest of the work with landmarks has centered around different admissible heuristics. An admissible landmark based heuristic was created by Karpas and Domshlak [60]. Instead of just counting how many landmarks remained that need to be satisfied, their heuristic estimates the number of actions needed to satisfy each landmark. Their initial solution was to calculate the cost of each landmark. Doing so optimally leads to the most efficient planning but is computationally expensive. A simple uniform cost partitioning is quick but less effective. They simplified this by focusing on action landmarks for which the cost is simply the cost of each action, however this can't completely solve the problem as there are still fact based landmarks whose cost must be found.

Work from Domshlak et al. incorporates landmarks into a problem before transforming the problem for an abstraction heuristic [33]. The landmarks add subgoals that the abstraction heuristics can pick up on to more accurately estimate the length of a path to the goal.

Landmarks have also been utilized in temporal planning with deadlines, goals need to be achieved at specific times in the plan. Actions have durations and multiple actions can occur simultaneously. Landmarks have been adapted for these problems by using them to create partial plans and integrating those partial plans [72, 61].

At this point there is no work involving the use of landmarks with preferences in planning. The closest work is with landmarks in oversubscription planning from Domshlak and Mirkis [34, 74]. Their approach recompiles the given problem into a classic planning problem incorporating landmarks that signal that the value of a state has improved. The process of compilation can be repeated to find landmarks that signal that a plan has been found that is better than a previous value. Thus an iterative approach is used to find a plan and then incrementally improve it. The approach is not applicable to the preferences planning problems we will be working with. The oversubscription problems have no classic goals that must be satisfied; they only involve values attached to individual facts. The problems also involve action costs, which our set of problems don't.

4.3 Adapting Landmarks for Preferences

Our goal is to adapt the work on landmarks and use it to guide a planner not just to a plan, but a plan that best satisfies the preferences defined in the problem. Landmarks inherently focus on guiding the planner towards satisfying the goal in the final state. As such, we will adapt landmarks for simple goal preferences only, atoms that are desired to be satisfied in the goal state.

We must define a new kind of landmark that factors in preferences. In the definition, preference formulas refer to the lowest level preference formulas in the specification. For example, the queries from an RPS specification or the basic desires from a $\mathcal{LPP}$ specification.

Definition 29. *Given a preference planning problem, $(P, Pref)$, a logical formula l is a preference-landmark (pref-landmark) if and only if l is satisfied in at least one state in every plan with a final state satisfying every preference formula.*

We will use the term *goal-preference landmarks* to describe the trivial set of landmarks based directly on the preferences. This definition of a preference-landmark may seem problematic as the definition rests on plans having final states that satisfy every preference. However, this isn't an issue. In the case that there is such a plan, a set of pref-landmarks will aid the planner in finding a plan that also satisfies every preference. In the case that there is no such plan, the definition actually allows any formula to be a pref-landmark. This is identical to the case for the standard definition of a landmark when there are no plans. In these cases, any landmark that is produced is technically a landmark as "every" plan satisfies it. In the case of standard landmarks, the planner simply searches and won't find a plan. In the case of pref-landmarks, even if there are no plans that satisfy every preference, the set of pref-landmarks should still guide a planner towards plans that satisfy as many of the preferences as possible.

The definitions for the different types of landmark orderings, Definition 26 and Definition 27, also apply to landmark-preferences.

We will use the landmark discovery methods from [81, 96] which don't need to be modified for the case of preference-landmarks. Landmarks for preferences that cannot be satisfied by a plan will be produced. Along with preferences that conflict with goals, preferences that conflict with other preferences can be expected. These situations will cause inaccuracies in heuristic values, but our belief is that the landmarks that lead to the goals and the non-conflicting preferences will successfully guide the planner.

We will not use discovery methods such as that in [65]. These methods attempt to find more conjunctive landmarks and do so by combining two or more elements of the goal and searching for landmarks that lead to that conjunction. When all of the landmarks are based on goals that must be satisfied, this is a good strategy that leads to a more informative set of landmarks. But in the case of preferences and goals that may have conflicts, this

leads to a greater number of inaccurate landmarks. Additionally, even with the planning problems without preferences, the algorithm to produce the landmarks took much longer to run. There was improved search performance, but it did not make up for the lost time.

4.3.1 Landmark Heuristics for Preferences

Next, we need to choose how to adapt the landmark heuristic, Definition 28, for use with preferences. The first heuristic that will be examined is the landmark heuristic itself without modification. The heuristic just needs to be supplied with a landmark graph based on a set pref-landmarks and orderings among them. Then we will define two new heuristics that take the landmark heuristic and use two methods to assign weights to the landmarks based on the preferences they are based on.

Most heuristics in classical planning estimate the distance from a state to the goal. A heuristic needs to be looked at differently when applying it to planning with preferences. With classic planning, with or without action costs, the aim is to find a plan. If multiple plans exist, the shortest plan is ideal and having a heuristic accurately estimate the distance is important. In planning with preferences, the primary concern is the satisfaction of preferences. Simply estimating the distance to the nearest goal state is unhelpful if the nearest goal state does not satisfy preferences. Instead of looking at the heuristic as providing an estimate of distance, it will provide a more abstract estimate of quality. The quality signals how likely the state is to lead to a goal state that optimally satisfies the set of preferences.

Basic landmark heuristic for preferences

The only modification that needs to be made to the definition for the landmark heuristic is in Equation 4.5 which defines the set $ReqAgain(s)$. In the definition a landmark will be required again if it is a member of $Accepted(s)$, is a goal landmark, and is currently unsatisfied. With preferences, it will also be required again if it is a goal landmark or a goal-preference landmark.

There are two ways that the basic landmark heuristic can be used, based on the set of landmarks it is given. It can be given a set of landmarks produced from the goals and preferences together or it can be given just the set of preferences. In this manner, the heuristic will either guide the planner to states that satisfy the goals and preferences, or just the preferences. When supplied with landmarks for both, the heuristic on its own can guide the planner towards high quality goal states. If only given preference-based landmarks, the planner will only be guided towards states that satisfy preferences. In this case, the planner will need a second heuristic that will guide it towards the goal.

Using the basic landmark heuristic on a set of pref-landmarks will be ideal when a goal state that satisfies all preferences exists. It will function like it does for a planning problem without preferences and guide the planner to the optimal plan. When there is no plan that

satisfies all preferences, the heuristic will still provide a guide towards plans that satisfy as many preferences as possible.

Weighted landmark heuristic for preferences

Our two variations on the landmark heuristic will weight the landmarks in two different ways. Unlike the base landmark heuristic, this heuristic will only evaluate a state based on the preferences. The two heuristics will be called the *max-pref-landmark heuristic*, h_{LM-max}, and the *shared-pref-landmark heuristic*, $h_{LM-share}$. The max-pref-landmark heuristic will weight a landmark according to the most important goal-preference landmark that it precedes, directly or indirectly, in the landmark graph. It will emphasize states that satisfy landmarks associated with the highest priority preferences. The shared-pref-landmark heuristic will use an algorithm that calculates the weight of each pref-landmark by assigning each goal-preference landmark a value and then propagating that value through all landmarks by having each landmark share half of its value with its predecessors in the landmark graph. This approach will also favor progress towards the most important landmarks, but less strongly so than the max-pref-landmark heuristic. It will also more strongly favor reaching landmarks closer to the goal-preference landmarks in the graph. The goal-preference landmarks will have the highest weights, so states that have satisfied preferences will be strongly favored.

To weight the landmarks, we need to have a value for each preference. We will be testing our heuristics with problems defined in PDDL. In these problems, each preference is assigned a positive numerical value, directly supplying values to use to calculate the landmarks' weights.

Each landmark, l, will be assigned a numeric weight, $w(l)$. Recall that the landmark graph is a directed graph. As such, a landmark, l' is a successor of l if there exists a path from l to l' in the directed edges. If the edge (l, l') is in the graph, l' is a *direct successor*. In these situations l is the predecessor or *direct predecessor* of l'. We define $Succ(l)$, $DirSucc(l)$, $Pred(l)$, and $DirPred(l)$ as the sets of successors, direct successors, predecessors, and direct predecessors of a landmark l.

We define $prefValue(l)$ as the value assigned by the preference specification if l is a goal-preference landmark and 0 if it is not.

For the max-pref-landmark heuristic h_{LM-max} we define the weight of each landmark as follows.

Definition 30.

$$w_{max}(l) = max(\{prefValue(l') : l' \in (Succ(l) \cup (l)\}) \tag{4.6}$$

Computing the value of w_{max} for a landmark graph is done by propagating the values from the goal-preference landmarks down through their predecessors. Then h_{LM-max} is calculated for a state by summing that weight for all landmarks that are still required.

Definition 31.

$$h_{LM-max}(s) = \sum_{l \in Required(s)} w_{max}(l) \tag{4.7}$$

The max-pref-landmark heuristic is not too far removed from the base landmark heuristic. If $prefValue()$ is 1 for all goal-preference landmarks, then the two heuristics will function identically. Otherwise, the planner will favor satisfying preferences with higher values as every landmark needed to achieve such preferences will have a higher weight. Even so, the landmarks remain balanced in the sense that their weight does not depend on the distance a landmark is from its goal-preference successors. If there are many landmarks ordered before a goal-preference landmark, satisfying one near the initial state is worth as much as satisfying the goal-preference landmark itself.

Next, we will define the weight function for $h_{LM-shared}$.

Definition 32.

$$w_{shared}(l) = prefValue(l) + \sum_{l' \in DirSucc(l)} w_{shared}(l')/|DirPred(l')| \quad DirSucc(l) \neq \emptyset \tag{4.8}$$

$$= prefValue(l) \quad DirSucc(l) = \emptyset \tag{4.9}$$

Similar to h_{LM-max}, $h_{LM-shared}$ is defined as follows.

Definition 33.

$$h_{LM-shared}(s) = \sum_{l \in Required(s)} w_{shared}(l) \tag{4.10}$$

To calculate the w_{shared} weights for a landmark graph, all landmarks that don't have any successors are assigned $prefValue(l)$ and the rest are marked as undefined. When every direct successor of a landmark has a value, its weight can be calculated. To calculate the weight for every landmark, the landmark graph is repeatedly scanned, calculating weights where possible until a weight has been assigned to every landmark.

The goal of w_{shared} is to assign each landmark the value of the preferences it leads to. If a landmark has two predecessors, then each of those predecessors gets half of the weight. This should more accurately define the relative value of the landmarks. If a landmark only leads to preferences with a low value, it will have a low value. If a landmark leads to a great number of preferences, than it's more valuable. This should lead the planner to favor states that are closer to satisfying more preferences or more valuable preferences. Also, if landmarks tend to have more than one direct predecessor, then the goal-preference landmarks and the landmarks closer to them will have higher values. The states that have

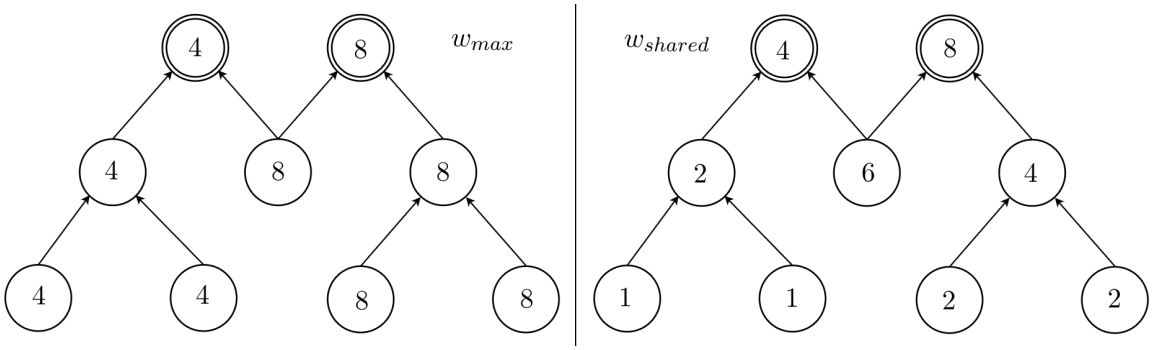

Figure 4.6: Two landmark graphs showing the weights calculated by w_{max} on the left and w_{shared} on the right. The double circles signify goal-preference landmarks.

made progress through the landmark graph towards satisfying a preference will be worth more than a state that has made progress further from the preference.

The two strategies for calculating weights are illustrated in Figure 4.6. The weighting from w_{max} heavily favors any landmark that leads to the higher valued preference. Satisfying the more valuable preference has a clear priority throughout the landmark graph. For w_{shared}, the weighting favors a more balanced approach. The landmarks for the more valuable preference are still a higher priority, but to a lesser degree. Additionally, this weighting clearly favors states that have completely satisfied one of the preferences.

4.4 Search Algorithm

With heuristics in hand, the precise search method needs to be chosen and adapted for planning with preferences. We will use a greedy best-first search, but the needs of planning with preferences necessitate some changes to the algorithm from Figure 2.2. There will likely be a large number of goal states with a wide range of quality. If the planner finds a plan that isn't definitively optimal, it should continue the search. Therefore we will use an anytime search that will continue searching for improved solutions until an optimal plan is found or the limit of time or memory is hit. Unlike most planners using an anytime approach [37, 4, 81], our anytime approach will not restart the search, but will simply continue the original search in the hope of finding an improved solution. This is also necessary because in classic planning a goal state will never be expanded. The quality of a plan is its length. But when preferences are involved, expanding a goal state is often desirable. There may be preferred goal states that are only reachable by expanding a goal state. If a goal state is found, its related plan needs to be saved, but the state still needs to be added to the open-list to be expanded just as every other node is.

We will use multiple heuristics as is done in LAMA which combines the FF [54] heuristic with the landmark heuristic. If one heuristic was doing a poor job in part of the search, the other heuristic is likely to fill in that gap. Each heuristic has its own open-list sorted by

```
      bestPlan = null
      bestPlanValue = null
      closedList = ()
      openLists = (openListH1, openListH2...)
      add initialNode to openListH1
      openList = openListH1

1     while openList is not empty and time limit not reached
2           currentNode = best node from openList
3           if currentNode is not in closedList
4                 for each action executable from currentNode
5                       newNode = execute action on currentNode
7                       if newNode is a solution
8                             if value(plan(newNode)) == 0
9                                   return plan(newNode) //plan is optimal
10                            if bestPlanValue == null or value(plan(newNode)) < bestPlanValue
11                                  bestPlan = plan(newNode)
12                      insert newNode in each openList according to associated heuristic
13          add currentNode to closedList
14          openList = next openList

15    if bestPlan ≠ null
16          return bestPlan
17    else
18          return FAIL
```

Figure 4.7: Pseudocode for the anytime multi-heuristic greedy best-first search algorithm.

the heuristic evaluation. When choosing a node to expand, the planner cycles through the open-lists. Regardless of which queue provided the node to expand, the successor nodes are added to every heuristic's queue. The merits of using multiple heuristics are examined by Helmert and Röger [82].

Using two heuristics or more should be especially effective in planning with preferences. The central difficulty of the area is finding a way to satisfy the goals while also satisfying preferences as best possible. Using more than one heuristic provides a natural way to do this. Simply supply the planner with one heuristic focused on satisfying the goals and another heuristic focused on satisfying preferences. Along with testing a single heuristic, we will test the effectiveness of two heuristics.

The greedy best-first search algorithm incorporating these methods is presented in Figure 4.7. There are a few changes from the basic algorithm presented in Figure 2.2. If a plan is found, it is returned if it is optimal; otherwise it is saved if its value is better than any plan found yet (lines 7-11). Also, nodes are inserted into every open-list (line 12) and the algorithm cycles through the available open-lists so that all are used (line 14). When removing nodes from the open-list ties are broken by taking the node nearest the initial node. If still tied, nodes are selected on a first-in first-out basis.

While the LAMA planner is the single strongest influence in the choices for our planner, there are strategies LAMA employed to maximize performance. These strategies are deferred heuristic evaluation and preferred operators. A node is not inserted into the open-

lists according to its own heuristic value, but according to its parent's heuristic value. This reduces the number of heuristic evaluations that take place since a node is only evaluated if it is chosen for expansion. This improves computation efficiency while sacrificing heuristic accuracy. Preferred operators involve marking an action as preferred if its execution leads to an improved heuristic value. When this happens, the action will then be prioritized over the other actions for a short period of time. While these strategies would probably lead to improved performance, we will leave them out as we wish to more closely focus on the accuracy of the different heuristics we will employ.

Chapter 5

Cascading Search Algorithms

A fundamental element of our strategy for searching for preferred plans is continuing the search, possibly finding multiple plans, until either an optimal plan is found or a time limit is reached. A downside of the greedy best-first search strategy is that the heuristic will usually guide the search down a particular path and examine a large number of possible plans that are similar. The greedy search is guided by the best heuristic values seen to a given point. The search will follow a direction that has steadily improving heuristic values, but then will only explore branches near the end of this path. All of the earlier unexpanded nodes have worse heuristic values and therefore won't be expanded. The search may get stuck in a local minimum, exploring an area of good heuristic values without finding different paths from the initial state that lead to the best values and the best plans.

When examining heuristic-guided search, the A* search algorithm is essentially the inverse of the greedy search algorithm. It expands the best node based on the distance from the initial state summed with the heuristic estimate. If the heuristic is estimating the distance to the goal and is admissible, A* will only return the shortest possible plan. This strategy leads to exploring more possible paths than the greedy best-first search. Therefore, this strategy is assured of finding the best plan, but is often extremely time and memory intensive, as it examines more states before reaching the goals. Even if a heuristic is inadmissible, A*'s behavior is likely to lead to a near-optimal plan. The ideal search strategy would be one that strikes a better balance between finding an optimal or near-optimal plan without expanding as many states as A*.

If a heuristic is extremely accurate, it will guide a greedy best-first search (GBFS) directly to the optimal plan. However, heuristics are often not this accurate and will lead to local minimums. Diversifying the paths being explored increases the likelihood of finding a quality plan. Another issue that a diverse search space can help counter is the presence of plateaus in the search space. A plateau is a large contiguous area of the search space filled with nodes that the heuristic can not differentiate. It can be difficult to break out of such areas.

Diversifying the search space is also useful in counteracting the problem of action execution order when a node is expanded, discussed by Dahlman and Howe [26]. When a node is expanded, the order that the executable actions are applied determines the order that the successor nodes are added to the open-list(s). If many of the new nodes have the same heuristic value, the order of the actions will determine the order in which they are expanded as the search continues. This can significantly effect a planner's likelihood of success. The order of the actions' execution can be randomized, but the random order could also cause problems. Diversifying the search space reduces the chance of this issue affecting the planner.

We present an algorithm that aims to diversify the search space of a GBFS search. Our algorithm focuses on finding new branches to explore by expanding the most promising states from throughout the entire depth of the search space. The approach does not involve randomness. We call our approach the Cascading Search algorithm.

5.1 Related Work

Planning with preferences has a particular need for exploring a diverse set of directions in the search space. However, this same diversification can be helpful in classical and other areas of planning as well. Most of the work involves satisficing planning. In satisficing planning, the expectation is not to find the optimal solution, but to just find a good solution. In difficult planning problems, finding the optimal solution is not practical, so the satisficing approach is taken. As such, a number of strategies to create an effectively diverse search space have been researched.

Our strategy uses a single search that attempts to explore diverse plans. Many planners use multiple searches to find diverse plans [81, 80, 37, 4]. After an initial search returns a plan, the planner initiates a new search. This search reduces the search space by looking for plans that are strictly better than the one already found. It may also use a different search algorithm or a different heuristic. Here, we will focus on work that diversifies the search space through more fundamental changes to the search algorithm instead of relying on restarting the search.

A number of these strategies involve an element of randomness. The simplest, from Valenzano et al. is the ϵ-GBFS algorithm [86]. The value ϵ is set, $0 < \epsilon \leq 1$ and determines the probability that the next node to expand is selected randomly. The node with the best heuristic value will be selected with probability $1 - \epsilon$. Otherwise, the node will be selected randomly from the entire list. The algorithm was chosen to see how much an element of randomness could improve planning performance compared to other deterministic enhancements such as preferred operators and multiple heuristics. The ϵ-GBFS produced a clear increase in the number of problems that could be solved.

Type-GBFS is another strategy with a random element that was presented by Xie et al, [93] and is based on *type systems* from [69]. A type system splits the nodes of an open-list into sets. When selecting a node to expand, the algorithm alternates between selecting a node in the standard best-first manner and using the type system. In the latter case, one of these sets is selected at random and a node is selected at random from that set. The types can be based on a single value such as the node's depth, g, or a heuristic. In this case, each possible value is its own set. Performance is best when the types were based on both g and a heuristic value h, giving a type for each possible combination, (g, h). This approach leads to nodes being expanded from every part of the search tree the planner has reached.

Another strategy to either diversify the search space, or break out of local minimums and plateaus involves numerous local searches within the global search space. Coles et al. develop an algorithm that utilizes repeated local searches designed to avoid local minimums [25]. If the search leads to a local minimum that cannot be escaped, the search is restarted. There is an element of randomness to the local search algorithm so the planner doesn't repeat itself. The planner also restarts if it hits a region of the search space where its progress is too slow.

Imai and Kishimoto follow a similar approach, but without restarts [57]. In their algorithm, each time a node is selected from the global open-list, a limited local GBFS is performed from that node. All nodes from the local search are added to the global open and closed lists. Then a new node is selected from the global open-list and a new local search is performed. Any node can be selected for this local search, but nodes with better heuristic values and larger g values are favored. Nakhost and Müller also use repeated local searches [75]. They looked at performing random walk searches from each node where a series of actions is selected randomly as well as improving performance by identifying the most helpful actions across these local searches and favoring them. Xie et al. combine local random walks with local searches to prevent GBFS from getting stuck [94, 92].

Rapidly exploring Random Trees, RRTs, is a method used to explore an array of diverse options in a path planning search space from Lavalle, [68]. They have been adapted for STRIPS planning by Burfoot et al. [22]. A random target node in the search space is chosen and a path to that node is searched for. The target node is chosen by selecting subsets of the goal. If the target is not quickly reached, a new target is chosen. If a node near enough to the goal is found, the planner attempts to find a plan from that node to the goal.

5.2 Cascading Search

The Cascading Search algorithm is a modified GBFS that attempts to find and follow multiple promising paths with an approach that focuses on every level of the search tree. Cascading Search requires an interval value that is an integer greater than or equal to one. A GBFS will run as normal, but after a number of expansions equal to this *cascade-interval*

```
    //openList member variables
    cascadeInterval =
    nodesRemovedSinceCascade = 0
    cascading = false
    currentG
    maxG


1   function openList.removeNode()
2       if cascading
3
4           //Loop to ensure that nodes are available at currentG
5           while currentG <= maxG and no nodes with currentG are available
6               currentG += 1
7
8           if currentG < maxG   //Normal cascading removal
9               returnNode = best node in openList with currentG
10              currentG += 1
11
12          else if currentG == maxG   //Final cascading removal
13              returnNode = best node in openList with currentG
14              cascading = false
15              nodesRemovedSinceCascade = 0
16
17          else   //Cascade ended and normal removal takes place
18              returnNode = best node in openList
19              cascading = false
20              nodesRemovedSinceCascade = 1
21      else
22          nodesRemovedSinceCascade += 1
23          if nodesRemovedSinceCascade == cascadeInterval
24              cascading = true
25              currentG = 1
26              maxG = openList.maximumG
27          returnNode = best node in openList
28      return returnNode
```

Figure 5.1: Pseudocode for removing nodes from an open-list in the Cascading Search Algorithm.

value have taken place, a *cascade* will occur. A cascade starts at the initial state and looks at each level of the search space. At each level, it will expand the unexplored node at that level with the best heuristic value. It expands the most promising node one action away from the initial state, then two actions away, etc. until the cascade hits the depth of the search space at the time the cascade began. The algorithm is shown in Figure 5.1. As the algorithm only requires changes to how the open-lists choose nodes to remove, that is what is shown. The algorithm in Figure 4.7 still holds.

There are two types of removal based on whether a cascade is taking place. When a cascade is not in progress, the most promising node is returned. The number of nodes removed since the last cascade is tracked. If it reaches the cascade interval, the member-level variables are set to begin a cascade on the next removal. The current depth of the

cascade is tracked by *currentG*. The last depth for the cascade is stored in *maxG*. During a cascade, the open-list must first make sure there is a node available at depth *currentG*, incrementing it if necessary (lines 5-6). The best node at depth *currentG* is removed. If *maxG* is reached, the best node at that depth is selected and the cascade ends (lines 12-15). It is unlikely, but possible that no nodes are available at depth *maxG*, so lines (17-20) end the cascade and perform a non-cascading removal.

The motivation for Cascading Search is as follows. Due to the ordering of the states in an open-list and the nature of the greedy best-first search, one node, or very few nodes, may be expanded at a certain depth. If the heuristic has not evaluated the states perfectly, it is likely that the search will explore a suboptimal path. Instead of leading to the optimal plan, it can lead to a local minimum where all of the successor nodes have an improved heuristic value, but never lead to a high quality plan, if any plan at all. Given the complexity of most planning problems, the search will rarely return to that shallower part of the search space before the time limit is reached.

The repeated cascades systematically search these promising but otherwise ignored branches of the search space. In the ideal case, the discovery of a better node at a shallow point in the tree will lead to a new best node at the next level, and at the level after that, and so on as the cascade advances to the edge of the explored search space. In this way the new promising branch will quickly be extended to the same depth as the overall search and be explored through the normal greedy search expansions.

5.2.1 Properties

With one exception, the Cascading Search algorithm inherits its properties from the GBFS algorithm it builds upon. We assume the algorithms return the first solution found instead of continuing to run using an anytime strategy. GBFS and Cascading Search are not *optimal*. If a solution is returned, there is no guarantee that the solution is the best possible. If supplied with an admissible heuristic, A* is optimal. If an infinite search space is allowed, GBFS is not *complete*, if a solution exists, it is not guaranteed to find it. Cascading Search is complete if given sufficient resources. GBFS will not explore every node at a given depth unless the heuristic leads it to. On the other hand, the cascades in Cascading Search ensure that given enough time, every depth of the search tree will be completely explored. Given a search tree of depth d and an average branching factor of b, the worst-case time and space complexity of the algorithms are $O(d^b)$. The actual performance of the algorithms is dependent on the accuracy of the heuristic guiding them.

5.2.2 Discussion

Figure 5.2 illustrates the motivation behind the Cascading Search algorithm. The search space can be pictured as a wedge with the node for the initial state at its tip. The nodes

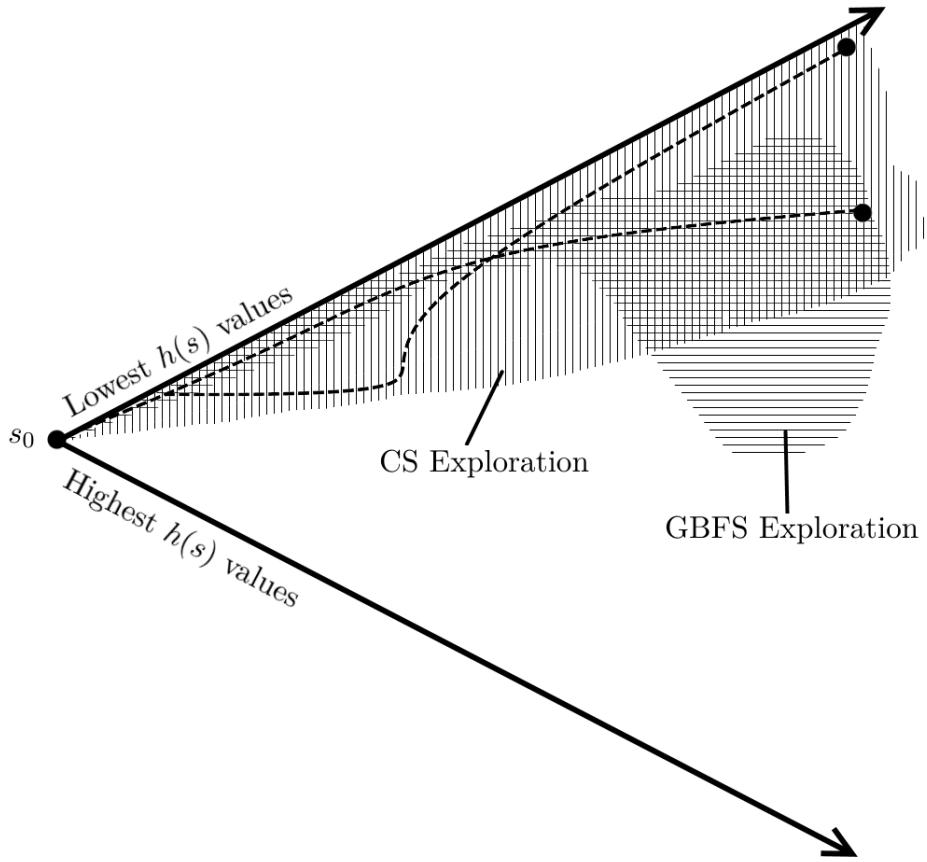

Figure 5.2: Diagram of how Cascading Search (CS) is intended to explore the search space versus a greedy best-first search (GBFS).

at each depth are sorted from best to worst heuristic value from top to bottom. A greedy best-first search begins exploring the search space near the top edge. If the heuristic is perfect, the GBFS algorithm will continue following the top edge and find the optimal plan. However, if the heuristic is inaccurate, the path through the search space to the best plans will veer away from the upper edge and the GBFS algorithm will probably miss it. In the figure, the path to the optimal plan does this. The GBFS search is led into part of the search space that won't allow it to find the optimal goal state. While the GBFS aspect of the cascading search algorithm continues, the cascades will cause the planner to sweep across the search space from top to bottom. In this way, the search will find the path to the best plans, even if the heuristic is inaccurate in portions of the search space. The cascade-interval controls how quickly the search will sweep across the search space. If it is large, the planner will spend most of its time in the standard GBFS search. If the interval is small, the planner will spend most of its time cascading. The more inaccurate the heuristic, the longer the cascades will take to find the best path. As such, the more accurate the heuristic, the higher the ideal cascade-interval should be.

Recall the other two variations on GBFS that don't involve restarts or local searches are ϵ-GBFS and Type-GBFS. The main difference between these algorithms and the Cascading Search algorithm is the use of randomness. The Cascading Search algorithm is completely deterministic. It systematically explores the entire search space. When ϵ-GBFS is making a random selection, it is completely random among all available nodes. The Type-GBFS algorithm chooses nodes randomly, but the type system ensures that those nodes are evenly spread throughout the search space. If the type system is based on a heuristic and g, (h, g), and there are ten thousand nodes of type $(10, 5)$ and only one of type $(20, 1)$, each type has the same probability of being chosen. Even if the search space is skewed towards a certain depth or heuristic value, nodes from all depths and heuristic values will be expanded. ϵ-GBFS is not likely to do so.

This means that the Cascading Search algorithm should be ideal when it is given heuristics that rarely make large mistakes when evaluating states. In the cases where it is inaccurate, the incorrectly valued node will not be far from its proper relative position. Even if the GBFS expansions miss the best node at a certain depth, it will be quickly caught up by the cascades sweeping across the search space. In these cases where the heuristic is accurate, the Cascading Search should find the best plans faster than ϵ-GBFS and Type-GBFS which both spend more time expanding nodes dispersed throughout the search space. However, if the heuristic does tend to make large mistakes, a Cascading Search may need to sweep across the majority of the search space before picking up the path to the optimal plan. Type-GBFS and ϵ-GBFS can catch the incorrectly evaluated nodes through their expansions from throughout the search space. If the heuristic is inaccurate enough, none of the algorithms are likely to find a solution.

Chapter 6

Experiments and Results

Three stages of experiments were performed. First, we compared the two landmark discovery methods that are appropriate for our use. The more promising method was used for the rest of the experiments. The second stage is to test the effectiveness of the landmark heuristics that have been developed in a greedy best-first search. The final stage is to examine how effectively the Cascading Search algorithm improves upon the GBFS algorithm, and compare that improvement to that provided by the Type-GBFS and ϵ-GBFS algorithms.

Our algorithms were implemented on top of the Fast Downward planner code base that is available online [50]. The LAMA planner among others that have competed in the International Planning Competitions is included.

We test our algorithms on planning problems specified in the PDDL language. Five sets of problems come from the Simple-Preferences track of the fifth International Planning Competition (IPC-5) and two sets of problems were created by us. The problem sets involved goal preferences only. We chose to use PDDL due to the availability of problem sets that have preferences, as well as domains that we could add preferences to. The Fast Downward planner we built upon uses the PDDL specification, but we had to add support for preferences. In a problem, each goal preference is assigned a positive rational number. The quality of a plan is assessed by summing the values of all preferences that haven't been satisfied. Therefore, if all preferences can be satisfied along with the goal, the optimal plan has a value of 0. We do not judge a plan's quality on its length. Satisfying the preferences is the primary motivation and it is on that basis only that plans will be compared.

Before the landmark discovery and planning search, the Fast Downward planner transforms the PDDL problem into the SAS+ specification [49], previously seen in Section 4.2.1. During this process the planner adds a new fluent to the planning problem to represent the satisfaction of preferences that are not single atoms or a conjunction of atoms.

Experiments were run on a computer with an Intel i7-3770 CPU and 16 GB of memory. The planner was given a search time limit of 10 minutes. The preprocessing steps and landmark discovery were not included in this time limit. A memory limit of 8 GB was

set. The value of 10 minutes was chosen based on the observation that for the majority of problems, if a plan is found, a plan is found in the first minute. The memory limit of 8GB allowed the search to continue for the full 10 minutes for nearly every problem.

Domains

The planner was run on seven problem sets from 6 different PDDL domains. Five of the problem sets are from IPC-5. These problem sets will contain "IPC" in their name.

Trucks The Trucks domain is a logistics domain. There are packages that need to be delivered, and trucks to deliver them. The packages must be delivered to a specific location with preferences over what time they are delivered. A set of fluents, $timeNow(t1)$, $timeNow(t2)$...is used to keep track of time. Each time a truck drives between locations, time advances one step. For example, $timeNow(t1)$ would become false and $timeNow(t2)$ would become true. One Trucks problem set is from the IPC-5 competition. All but one of the problems in that set have an optimal solution that satisfies all preferences. We have created a second problem set where it is never possible to satisfy all preferences. Either some of the preferences and goals are mutually exclusive, or some of the preferences are mutually exclusive with other preferences.

Openstacks The Openstacks domain is based on a problem from constraint satisfaction. A manufacturer has a list of orders for sets of products that it needs to fulfill. However, only one type of product can be manufactured at a time. Preferences are used to prioritize certain products or orders over others. The problems in this domain are designed so that it is very easy to find a plan, but difficult to find a good one.

Elevators The Elevators domain has been used in multiple International Planning Competitions, but did not involve preferences. We have added preferences. The problem involves a building with people on different floors wishing to take an elevator to their destination floor. Different elevators can stop at different sets of floors. The domain did involve action-costs, but they have been removed. Passengers have preferences over what floor they reach and which elevator they take. For all problems it is possible to find a plan that satisfies all preferences.

The problems in the following three domains have empty goal sets. Due to this, a plan with zero actions is considered a solution. However, the planner will not return a plan unless it has found a plan of higher quality than the empty plan with respect to the defined preferences.

Pathways The Pathways domain is based on molecular biology. It models chains of chemical reactions that cells utilize. Actions in the domain represent choosing pairs of molecules and creating a reaction between them to create an output. The output from one reaction may be needed as the input for a later reaction. Preferences specify the desired final outputs and the number of reactions that are performed to create those outputs.

TPP The Traveling Purchase Problem, TPP, domain is a variant of the traveling salesman problem. It involves markets that are selling goods and depots that need to be filled with those goods. Each market sells a limited supply of goods. Either one or two trucks are available to travel to the markets, purchase the goods, and bring them to the depots for storage. The preferences are over what goods have been bought from the market and how full the depots are. To satisfy the preferences, the truck(s) are required to make many trips between the markets and the depots.

Storage The Storage domain involves spatial reasoning. There are containers storing crates and hoists that are used to transfer the crates into depots. Hoists have movement limitations so crates need to be handled by multiple hoists. There are restrictions on where each crate can be stored. Preferences specify which areas of each depot should store which crates and which areas of each depot should be empty.

6.1 Landmark Discovery

First, the two landmark discovery methods, that from LAMA [81], and that from Zhu and Givan, ZG [96] will be compared. The two methods are described in Section 4.2.1. We will look at how many landmarks they produce and how effectively the landmark graphs produced guide the planner.

For problems with non-empty goals, both algorithms create a landmark graph based on the goals and preferences together. We will call these mixed landmark graphs. The algorithms will also produce a landmark graph based only on the preferences for every problem. The totals for each algorithm and for each set of problems are in Table 6.1. The number of problems that each method found more landmarks for is in parentheses.

The LAMA method discovered more landmarks for all problems when creating a mixed landmark graph and the majority of problems for pref-landmark graphs. This isn't surprising as the ZG method is designed to only discover causal landmarks.

The question is then, which sets of landmarks lead to a better heuristic evaluation? The base h_{LM} landmark-counting heuristic was used. The problem sets with goals and preferences use two open-lists when the landmark graphs are separate. Otherwise, a single heuristic and open-list is used.

Total Landmarks Discovered				
	LAMA:Mix	ZG:Mix	LAMA:Sep	ZG:Sep
IPC-Trucks	2914 (20)	2045 (0)	2047 (12)	1667 (7)
Trucks	2526 (20)	1305 (0)	2020 (20)	773 (0)
IPC-Openstacks	12766 (20)	6734 (0)	1442 (0)	6234 (20)
Elevators	2708 (16)	2128 (0)	1896 (16)	886 (0)
IPC-Pathways			3026 (20)	2689 (9)
IPC-Storage			199683 (8)	199876 (12)
IPC-TPP			3336 (0)	3411 (10)

Table 6.1: Total landmarks discovered over each problem set. The number in parentheses specifies how many problems each method found more landmarks in.

Problems Solved/Best Known Solution (Wins)				
	LAMA:Mix	ZG:Mix	LAMA:Sep	ZG:Sep
IPC-Trucks	20/6 (12)	20/5 (2)	20/8 (11)	20/6 (3)
Trucks	16/3 (13)	9/0 (1)	16/2 (11)	19/0 (9)
IPC-Openstacks	20/2 (5)	20/0 (15)	20/2 (14)	20/0 (3)
Elevators	8/8 (0)	12/12 (4)	0/0 (0)	0/0 (0)
IPC-Pathways			10/6 (1)	16/7 (10)
IPC-Storage			20/4 (3)	20/8 (10)
IPC-TPP			20/5 (2)	19/5 (2)
Totals	64/30 (30)	61/17 (22)	106/27 (42)	114/26 (37)

Table 6.2: Number of problems solved in each problem set and the number of solutions that match the best known. The number in parentheses is the number of problems with a better result than the alternative landmark discovery algorithm.

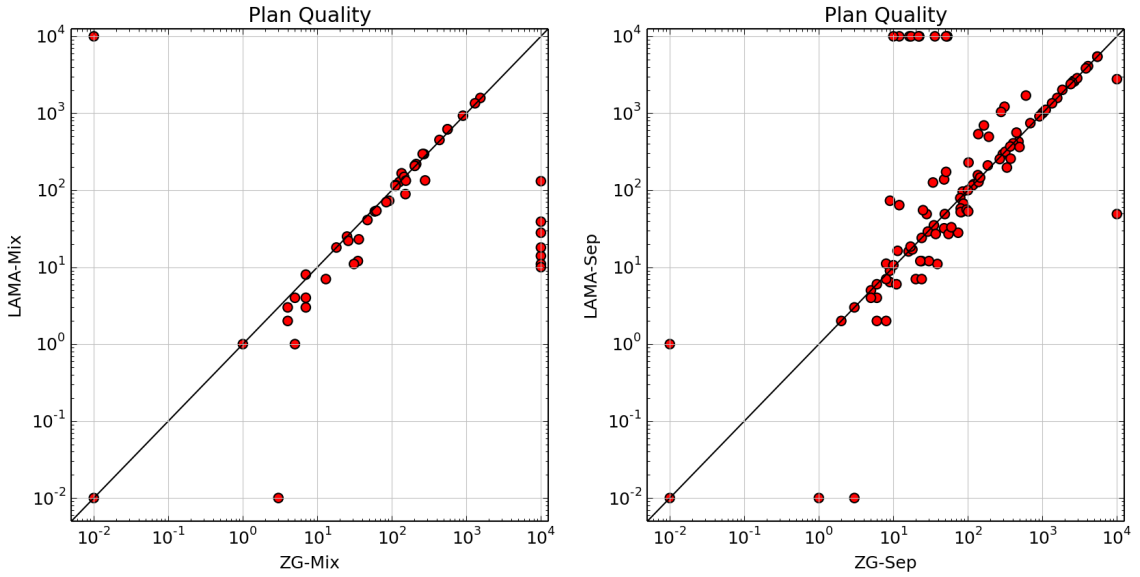

Figure 6.1: Scatter plots showing the quality of plans using h_{LM} heuristic and landmarks produced by the LAMA approach and Zhu and Givan's approach, ZG.

The plan quality results are summarized in Table 6.2. 'Mix' is used to signify that a mixed landmark graph was used and therefore is only applicable to problems with a non-empty goal. 'Sep' signifies that preferences and goals, if present, were kept in separate landmark graphs. The LAMA and ZG discovery approaches produce comparable results. The ZG approach solves more problems, but the LAMA approach has more wins and more often finds plans that match the quality of the best known solution. LAMA clearly performs better in both problem sets in the Trucks domain while ZG performs better in the IPC-Storage and IPC-Pathways domains. The IPC-Openstacks domain is interesting in that LAMA performs poorly when using the mixed planning graph even though it discovered a much larger number of landmarks. However, with separate planning graphs, the LAMA landmarks performed better although they were fewer in number.

To see how the quality of plans varied on a problem by problem level, the results are plotted on two graphs in Figure 6.1 A lower value is preferred, so points under the diagonal favor the LAMA method and points above the diagonal favor the ZG method. Points that are at the right edge or top edge signify that the ZG or LAMA method did not produce a solution. The plots show that for the majority of problems, the approaches produce plans of similar quality.

We don't present detailed results on time taken by the discovery methods. With the exception of 12 problems from the storage domain, landmark discovery took less than 2 seconds in all cases. The most complex problem in the storage domain had approximately 37000 landmarks for both methods. Both discovery methods took over an hour for this problem. There was no meaningful difference in time taken between LAMA and ZG. On the problems that took over 2 seconds, neither method was consistently faster and the largest single disparity was 11%.

Due to collecting more wins and matching the best known solution more often, the LAMA discovery method was chosen to be used for the remainder of the experiments.

6.2 Landmark Heuristics

Having chosen the landmark discovery algorithm, the effectiveness of the three landmark based heuristics, h_{LM}, h_{LM-max}, and $h_{LM-shared}$, were tested using greedy best-first search. For problems that include goals, h_{LM}, was run as the only heuristic on a mixed landmark graph as well as being run for two open-lists on separate goals and preferences landmark graphs. On these problems, $h_{LM-shared}$ and h_{LM-max}, used a mixed landmark graph. The goal landmarks have no weight unless they are involved in orderings with landmarks for the preferences. $h_{LM-shared}$ and h_{LM-max} were paired with h_{LM} running on the goals landmark graph.

As a baseline, problems with goals were run with h_{LM} only using the goal landmark graph. The planner will only be guided to satisfy goals. Therefore, it should produce plans,

Problems Solved						
	LM:Goals	LM:Mix	LM:Sep	LM-max	LM-shared	FF
IPC-Trucks (20)	20/5	20/6	20/8	20/9	20/7	18/7
Trucks (20)	20/0	16/3	16/2	20/1	18/0	20/1
IPC-Openstacks (20)	20/0	20/2	20/2	20/2	20/2	20/0
Elevators (16)	0/0	8/8	0/0	0/0	0/0	16/16
IPC-Pathways (30)			10/6	16/6	17/6	6/4
IPC-Storage (20)			20/4	20/7	20/5	20/4
IPC-TPP (20)			20/5	20/7	20/9	20/4
Totals: Goals	60/5	64/19	56/12	60/12	58/9	74/24
Totals: Empty Goal			60/15	56/20	57/20	46/12
Totals			106/27	116/32	115/29	120/36

Table 6.3: The number of problems solved and the number of the solution that match the quality of the best known solution The size of each problem set is listed in parentheses.

but not satisfy most preferences. The heuristics guided by preferences should consistently lead to superior plans.

All problems were also run with the FF heuristic. The FF heuristic has been adapted for preferences by supplying it with all goals and preferences together. The heuristic is effective on planning without preferences, outperforming the landmark-count heuristic [81]. Therefore, FF should fare well when it is possible to satisfy all preferences. In such cases the heuristic should guide the planner to an optimal goal state. If it is not possible to satisfy all preferences, it may not perform as well.

The overall results are summarized in two tables. Table 6.3 contains a summary of the planning problems solved and how often the best known solution was equalled for each heuristic. Totals are give over the sets of problems with goals, the sets of problems with only preferences, as well as the final total. There isn't a lot to differentiate between the heuristics based on this information. LM:Mix and FF have better numbers over problem sets with goals, but this is due to their success in the Elevators domain. With the exceptions of the Elevators and IPC-Pathways problem sets, all configurations were able to produce plans for the majority of problems.

Table 6.4 gives a second way of viewing the test results as a whole. It provides a pairwise comparison of each combination of heuristics and specifies how many times each heuristic was more successful on a given problem. Either it produced a better plan, or the second heuristic failed to produce a plan. For example, if the results for LM:Goals are compared to LM:Mix, LM:Goals returns a superior result 12 times versus 46 for LM:Mix. This gives a good sense of which heuristics led to higher quality plans.

The first thing the table makes clear is that while LM:Goals is producing plans, they are usually low-quality plans. This is to be expected, but still is a good sign that the other heuristics are guiding the planner well. The FF heuristic performed slightly better than

Heuristic v Heuristic Match-ups							
	LM:Goals	LM:Mix	LM:Sep	LM-max	LM-shared	FF	Total
LM:Goals		12-46	8-42	1-51	2-49	2-50	25-238
LM:Mix	46-12		19-32	23-31	25-31	25-30	138-136
LM:Sep	42-8	32-19		33-47	32-57	57-47	163-178
LM-max	51-1	31-23	47-33		38-37	77-33	244-127
LM-shared	49-2	31-25	57-32	37-38		67-35	241-132
FF	50-2	30-25	47-57	33-77	35-67		195-228

Table 6.4: Comparison of heuristics by how many problems they produced a better or worse solution for. For example, LM:Mix found 46 better solutions and 12 worse solutions than LM:Goals.

LM:Mix, but not as well as the other heuristics, excluding LM:Goals. The two weighted heuristics, LM-max and LM-shared, show the strongest performance overall. They clearly outperform all other heuristics. For a more detailed analysis, we break down the results by domain.

6.2.1 Trucks

There are two problem sets that were used from the Trucks domain. First, we consider IPC-Trucks. Of the twenty problems, it is possible to solve 19 while satisfying all preferences. It may be possible for the final problem, as the best known solution only leaves a single preference unsatisfied. The results for this problem set are in Table 6.5. Problems for which every heuristic led to a solution as good as the best known, have been removed.

Checking the baseline heuristic LM:Goals shows that as expected, most results were poor, but on Problems 7 and 9 it returned a higher-quality result than at least one other heuristic. This probably is a result of clusters of plans in the search space and LM:Goals happened upon such a cluster containing relatively good plans. LM:Mix was clearly outperformed by LM:Sep. The basic LM heuristic performed better when the pref-landmarks and goal landmarks are separated. Problems 11 through 20 had preferences which were all assigned the value 1, and thus the weighting led the LM-max heuristic to count landmarks in the same way as the basic landmark-counting heuristic. The landmark graph supplied to the LM-max heuristic does include the goal landmarks. Orderings must exist from this that enabled the LM-max heuristic to outperform LM:Sep on Problem 15. Over this set, LM-max performed best with its weighting of the landmarks keeping it in front of LM:Sep on four out of the five first problems. These problems have preferences with varying weights desiring packages be delivered earlier in the plan. The weighting for LM-max strongly emphasizing the more important preferences led to those preferences being better satisfied than the more balanced heuristics could manage.

Plan Quality: IPC-Trucks							
Problem	Best	LM:Goals	LM:Mix	LM:Sep	LM-max	LM-shared	FF
6	0	1	1	1	**0**	**0**	**0**
7	0	55	53	73	**3**	18	11
8	0	149	89	**49**	94	113	**49**
9	0	45	296	64	29	**24**	None
10	0	110	133	56	**20**	58	None
11	0	5	3	**0**	**0**	**0**	2
12	0	4	1	**0**	**0**	1	4
13	0	4	**0**	**0**	**0**	2	**0**
14	0	8	**2**	4	4	4	3
15	0	7	4	4	**1**	2	2
16	0	12	4	4	4	**1**	5
17	1	10	**4**	**4**	**4**	6	7
18	0	10	3	**2**	**2**	6	4
19	0	10	4	**2**	**2**	5	5
20	0	16	8	**6**	**6**	7	7

Table 6.5: Plan quality data for the IPC-Trucks problem set. Bold marks best value from these heuristics.

The FF heuristic's performance was good, but not dominant. With the exception of Problem 17, all preferences in all problems can safely be treated as goals. FF has outperformed the landmark-count heuristic on classical planning problems [81], so seeing the landmark based heuristics performing well in comparison is positive.

The second problem set in the Trucks domain was created with problems that all have conflicts between preferences or between preferences and goals. The results are in Table 6.6. The odd numbered problems all have interference only among preferences. The even numbered problems have incompatibility between preferences and goals. The effects of this are clear as every instance where a heuristic failed to lead to a plan had this interference with the goals.

The LM-max heuristic performed best among the landmark based heuristics with respect to the number of problems solved. However, it only produced the best plan twice. Excluding the problems where it didn't produce a plan, LM:Mix performed well. This suggests that the mixed landmark graph produced orderings between goal and preference landmarks, helping the planner satisfy both. The FF heuristic performed similarly well.

6.2.2 OpenStacks

As seen in Table 6.7, the LM:Sep configuration dominated. The reason for this can be seen in the number of landmarks discovered in this domain, see Table 6.1. The LAMA discovery method found nearly ten times the number of landmarks in the mixed landmark graphs as in the preferences landmarks graphs over the problem set. The result of this

Plan Quality: Trucks							
Problem	Best	LM:Goals	LM:Mix	LM:Sep	LM-max	LM-shared	FF
1	12	27	25	**12**	13	15	17
2	9	27	None	None	**9**	12	14
3	11	57	**11**	**11**	13	13	17
4	28	51	**28**	None	29	31	43
5	6	21	**7**	**7**	13	19	**7**
6	11	15	**11**	None	17	12	**11**
7	129	495	**134**	257	185	239	279
8	129	495	**131**	495	239	None	151
9	20	73	41	27	67	**21**	23
10	20	73	39	None	59	None	**21**
11	12	114	18	**17**	34	18	18
12	12	126	**18**	126	62	22	22
13	13	45	**23**	49	38	44	24
14	18	42	None	49	26	36	**25**
15	16	96	22	28	81	**21**	26
16	16	100	None	96	**22**	26	26
17	4	24	10	**7**	10	20	11
18	4	28	14	**7**	10	28	11
19	37	592	166	197	187	**86**	135
20	142	592	None	561	592	228	**161**

Table 6.6: Plan quality data for the Trucks problem set. Bold marks best value from these heuristics.

Plan Quality: IPC-Openstacks							
Problem	Best	LM:Goals	LM:Mix	LM:Sep	LM-max	LM-shared	FF
1	12	35	**12**	**12**	**12**	**12**	35
2	12	35	**12**	**12**	**12**	**12**	35
3	11	90	70	**32**	46	46	90
4	21	100	73	**33**	55	61	100
5	24	126	128	**67**	93	93	126
6	14	126	128	**52**	92	92	126
7	32	284	295	**210**	242	242	284
8	123	589	619	**428**	578	578	586
9	96	589	620	**364**	516	516	589
10	3	115	117	**52**	64	68	115
11	11	116	114	58	80	**38**	111
12	6	153	149	**53**	117	103	153
13	27	223	219	**157**	175	173	223
14	6	65	54	**27**	54	43	65
15	0	210	209	**146**	147	147	210
16	0	210	207	**128**	182	182	210
17	0	450	450	**316**	381	381	450
18	0	930	930	**744**	824	930	930
19	254	**1581**	**1581**	**1581**	**1581**	**1581**	**1581**
20	424	**1348**	**1348**	**1348**	**1348**	**1348**	**1348**

Table 6.7: Plan quality data for the IPC-Openstacks problem set. Bold marks best value from these heuristics.

is that the heuristics take more CPU time to process the larger landmark graphs and the planner doesn't make as many expansions. Over the entire problem set, LM:Sep expanded approximately 1.5 million nodes, while all other landmark heuristic configurations expanded approximately 1.2 million. The extra landmarks did not provide enough guidance to overcome the extra processing. The reason for this may be the nature of the problem's goals. Each goal requires an order to be shipped. It is trivial to ship an order, but the preferences require products to be manufactured and added to the order. The LM:Mix heuristic is likely getting misled into satisfying goals before preferences. When the landmark graphs are separate, the preferences landmark graph can guide the heuristic to satisfy preferences for an order. Then the heuristic guided by the goals can complete that order. Meanwhile, the FF heuristic ran much faster, expanding 15 millions nodes in total, but was likely also misled by the easy to achieve goals.

6.2.3 Elevators

This was the most difficult domain for the heuristics. Only LM:Mix and FF were able to solve any of the problems. LM:Mix solved 8 and FF solved all 16. All the solutions from both heuristics were optimal. The planner is able to solve problems in this domain that don't involve preferences, so it is difficult to say what is leading to the poor performance. Presumably, some aspect of the problems leads to the planner getting stuck in a local minimum. All preferences can be satisfied in every problem, so that does suit FF and LM:Mix which both make no differentiation between goals and preferences.

6.2.4 Pathways

The results for IPC-Pathways are presented in Table 6.8. The first four problems are removed because all configurations resulted in a plan of the best known quality. However, the other problems are removed because none of the configurations produced a solution. Note that this domain has problems with empty goals so LM:Goals and LM:Mix are not included. This is an extremely difficult domain for the heuristics. No plan was found for 10 of the problems. The LM-shared heuristic produces the most plans as well as the most highest-quality plans. Both LM-max and LM-shared outperform LM:Sep. On these problems with no goals, the only difference between these heuristics is the weighting of the landmarks. The weighting of LM-max and LM-shared leads to more focus on completing specific preferences and benefits in these problems. This domain produced fewer landmarks per problem than any other. This led to the landmark heuristics struggling. However, the FF heuristic failed to produce plans for the majority of problems as well.

Plan Quality: IPC-Pathways					
Problem	Best	LM:Sep	LM-max	LM-shared	FF
5	6	**6**	**6**	**6**	7.8
6	6.4	**6.4**	**6.4**	**6.4**	None
7	8	10.6	**9**	10.6	None
9	10	None	11.6	**11.5**	None
10	10	16.3	**12.4**	13	16.5
11	8	11.1	**11**	**11**	None
13	14.2	18.5	**16**	None	None
14	15.6	None	18.6	**16.2**	None
15	14.4	None	**17.7**	18.1	None
22	19	None	23.8	**20.8**	None
23	18	None	26.8	**25**	None
25	28.1	None	None	**35.5**	None
26	20.5	None	None	**30.5**	None
28	27	None	35.5	**34.4**	None

Table 6.8: Plan quality data for the IPC-Pathways problem set. Bold marks best value from these heuristics.

Plan Quality: IPC-Storage					
Problem	Best	LM:Sep	LM-max	LM-shared	FF
5	25	55	**25**	**25**	57
6	43	138	**43**	54	113
7	44	173	**44**	58	229
8	51	229	131	**63**	310
9	83	536	196	**131**	527
10	164	695	**431**	484	673
11	220	1042	**369**	1019	1003
12	310	1217	**593**	1157	1109
13	601	1699	1540	1540	1651
14	1233	2015	**1723**	1904	1851
15	1867	2602	**2389**	2554	2554
16	1988	2854	**2810**	2816	2816
17	2971	3834	**3632**	3860	**3632**
18	3192	4098	4143	**3934**	**3934**
19	4355	5457	5253	**5232**	**5232**
20	4574	5436	5436	**5191**	**5185**

Table 6.9: Plan quality data for the IPC-Storage problem set. Bold marks best value from these heuristics.

Plan Quality: IPC-TPP					
Problem	Best	LM:Sep	LM-max	LM-shared	FF
5	19	**79**	**79**	**79**	96
6	101	115	**101**	**101**	112
7	100	**100**	**100**	**100**	127
8	105	120	**105**	**105**	142
9	205	254	254	**205**	337
10	282	294	299	**286**	368
11	295	370	380	**303**	401
12	308	407	406	**308**	432
13	750	913	918	**757**	943
14	821	992	980	**846**	1006
15	837	1037	1053	**859**	1069
16	941	1108	1114	**1004**	1132
17	1821	2395	2393	**2328**	2411
18	1576	2511	2520	**2393**	2538
19	1829	2651	2647	**2587**	2665
20	2087	2776	2774	**2526**	2793

Table 6.10: Plan quality data for the IPC-Trucks problem set. Bold marks best value from these heuristics.

6.2.5 Storage

The results for IPC-Storage are displayed in Table 6.9, and show every configuration finding a plan for every problem. Other than that, the results are similar to the IPC-Pathways problem set in favoring the weighted landmark graphs of LM-max and LM-shared. This time LM-max produced more higher quality plans. Its weighting is more effective when given preferences with a large range of values. The FF heuristic finds plans for every problem and is competitive on many of the problems.

6.2.6 Traveling Purchase Problem

The results for the final problem set, IPC-TPP are in Table 6.10. The LM-shared heuristic led to the best plan for every problem. LM:Sep performs better than LM-max on many of the problems. LM-max is focusing on the most valuable preferences, but generally not succeeding. The more balanced weighting of LM-shared and the counting of LM:Sep allow the planner to find the preferences of all values that it can satisfy. The FF heuristic is the worst performer on every problem.

6.2.7 Conclusions

As is often the case with planning strategies and heuristics, different problem sets favored different configurations in our tests. Still, there are patterns. The weighted landmark

graphs used by LM-max and LM-shared led to good performance in all problem sets except Elevators.

With the exception of the Elevators problem set, the problem sets with goals were favored by a configuration running two heuristics. The interplay between a heuristic focused on goals and a heuristic focused on preferences led to the satisfaction of both. Having a single mixed landmark graph seems to muddle the heuristic values. The landmark heuristics performed well, generally better than FF, even on the problems where all preferences can be satisfied and shouldn't hinder FF's effectiveness.

The landmark heuristics show promise giving results far better than from the baseline LM:Goals heuristic and outperforming FF. The weighted heuristics successfully guided the planner to satisfying more or more valuable preferences than the basic landmark-counting heuristic.

6.3 Cascading Search

Testing of the Cascading Search algorithm began with running different cascade-interval values and choosing one that was promising. The value determines how often the search departs from a standard greedy best-first search and runs a cascade through the search tree. The most promising cascade-interval was then used and the Cascading Search algorithm's effectiveness compared to Type-GBFS and ϵ-GBFS. Finally we examined the specifics of why the Cascading Search leads to an improvement over GBFS and the two extensions of GBFS.

6.3.1 Cascade-Interval

To determine a good cascade-interval value the Cascading Search algorithm was run on all problems using the basic landmark-counting heuristic. Problems that had goals used a mixed landmark graph. The values 1, 10, 100, and 1000 were tested. A value of 1 has the algorithm only expanding one node between cascades, leading the algorithm to do practically nothing but sweep across the search space. A value of 1000 will be much closer to a standard GBFS algorithm. Preliminary testing showed that the optimal value would lie somewhere in this range and the full results borne this out. The majority of plans found in the previous tests were under 100 actions long. The longest, in the TPP problem set, were approximately 700 actions long. That gives an estimate of the depth of the search spaces that the cascade will be running through. If the GBFS is guided well by the heuristic, the cascades may delay the discovery of a plan beyond the time limit.

For a comparison, the wins and losses for each cascade-interval were totaled as was done in Table 6.4. The results are given in Table 6.11. All four values resulted in solutions for 119 of the problems, up from 114 with the GBFS search. The two best values are 1 and 10. This indicates that spending time using cascades to sweep across the search space is

Cascade Interval Match-ups					
	1	10	100	1000	Total
1		11-11	18-10	35-6	64-27
10	11-11		18-4	34-5	63-20
100	10-18	4-18		31-8	45-44
1000	6-35	5-34	8-31		19-100

Table 6.11: The performance of the four tested cascade-interval values versus each other.

GBFS Search Algorithms: Problems Solved				
	GBFS	ϵ-GBFS	Type-GBFS	CS
LM	114/34	114/36	107/44	**119/68**
FF	120/36	**121**/35	101/40	117/**47**
LM-max	116/32	**127**/38	114/35	**127/54**
Total	350/102	362/109	322/119	**363/169**

Table 6.12: The number of problems solved for each algorithm and heuristic pairing.

useful. For the rest of the tests, 10 was used as the cascade-interval. For optimal results the value could be tuned for any given domain/heuristic pair but we used the single value for consistency.

6.3.2 Comparison to Type-GBFS and ϵ-GBFS

A cascade-interval has been chosen, so next we compared Cascading Search to ϵ-GBFS and Type-GBFS. For ϵ-GBFS, we set $\epsilon = 0.2$. This was the ideal value found in the original work and indicates a one-in-five chance that the most promising node won't be expanded in favor of a random selection from the open-list. Type-GBFS will use the suggested type system (h, g). The three algorithms were run three times, each time with a single heuristic. We tested the base landmark-count heuristic, the FF heuristic, and the LM-max heuristic. The landmark-counting heuristic was given the mixed landmark graph if goals were present, the preference landmark graph if not. The three heuristics showed varying performance in the previous tests and should give an idea of how the search algorithms react to heuristics of varying accuracy.

GBFS Algorithm Match-ups by Heuristic			
	CS v ϵ-BGFS	CS v Type-GBFS	Type-GBFS v ϵ-GBFS
LM	65-14	46-21	71-22
FF	59-26	40-37	72-28
LM-max	68-23	69-18	48-46
Total	192-63	155-76	191-96

Table 6.13: The performance of the three algorithms versus each other for each heuristic.

The number of problems solved by each algorithm for each heuristic is given in Table 6.12. Cascading Search and ϵ-GBFS both improved upon basic GBFS by a small amount while Type-GBFS solved notably fewer. The number of solutions that match the best known was improved by all three algorithms over GBFS. Cascading Search increased the number by two thirds while Type-GBFS and ϵ-GBFS produced smaller improvements.

Table 6.13 shows the results when the algorithms are matched up with respect to each problem. Cascading Search beats ϵ-GBFS by similar margins for each heuristic, but there is more variation versus Type-GBFS. When compared respect to the LM-max heuristic Cascading Search has a much better ratio than with FF or LM. This supports the idea that Cascading Search will be more effective than Type-GBFS when the heuristic is more accurate as LM-max was the strongest heuristic in the previous tests.

The scatter plots in Figure 6.2 show how these algorithms improved plan quality over a GBFS search when using the landmark-counting heuristic. The plots are similar for the other two heuristics. The plots show that ϵ-GBFS does not improve plans by much. The random node selections are hitting on improved paths, but not to a meaningful degree. Both Cascading Search and Type-GBFS improved plan quality more significantly. The evenly dispersed strategy of the type-system is consistently finding improved paths, but Cascading Search finds better paths or the best path more often by selecting only nodes the heuristic marks as promising.

The most affected set of problems was Elevators. All of the solutions found by the different algorithms were optimal. The standard GBFS solved 8 of the 16 problems; ϵ-GBFS and Cascading Search each solved 12; Type-GBFS didn't solve any. The heuristic was accurate enough to solve 8 problems with the standard GBFS, but selecting nodes using the type-system took too much time away from the area the search needed to explore. The ϵ-GBFS picked up on better paths, but also follows the most promising node eighty percent of the time. Type-GBFS only follows the most promising node fifty percent of the time.

The IPC-Storage problem set demonstrates the opposite position. The landmark heuristics struggled to guide the planner and the time spent expanding nodes from throughout the search space was effective. Type-GBFS matched the best known plan 11 times compared to just 4 with Cascading Search. The heuristic seems to supply a low quality estimation of the states. The dispersed expansions of Type-GBFS put the planner on better paths through the search space. Cascading Search did not have the time to find these paths with its cascades.

The Cascading Search algorithm is intended to find high-quality plans quickly. As a final test, the A* algorithm was run using the landmark heuristic on the same landmark graphs. Instead of returning the first plan, A* will continue searching using the same anytime strategy as the Cascading Search. The landmark-counting heuristic is inadmissible, but it does estimate the distance to a high-quality goal state. It should work well with an A* search. In comparison, the LM-max or LM-shared heuristic would not work as well. The

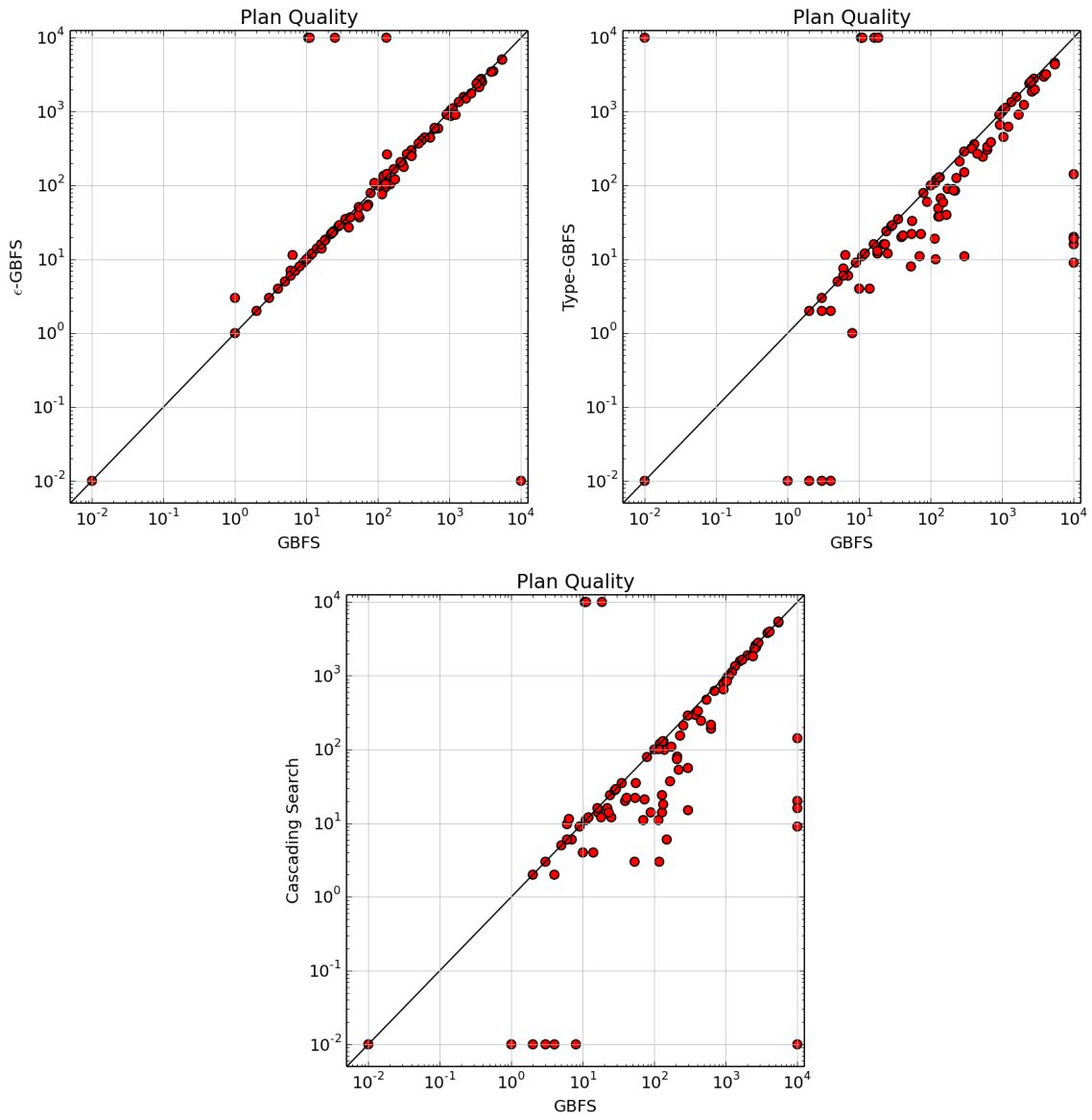

Figure 6.2: Scatter plots showing the quality of plans produced by ϵ-GBFS (Upper-Right), Type-GBFS (Upper-Left), and Cascading Search (Lower) versus GBFS.

weights those heuristics assign to the landmarks increases the heuristic values returned. Therefore, the depth value would have a smaller impact on the direction of the search. As the A* algorithm explores a wider area, it should not solve as many problems, but when it does find a plan, it should be of a high quality. If Cascading Search can return plans of a similar or higher quality, that would demonstrate the algorithm's effectiveness at finding high-quality plans.

When the tests were run, A* solved 17 fewer problems than the Cascading Search. That is as expected. There were 60 problems where one algorithm did better. Of those 60, the Cascading Search returned a higher-quality plan 55 times. This was a surprisingly good result for Cascading Search. The strategy of sweeping across the search space from the most heuristically promising nodes is picking up the best plans more often than A*'s strategy of building a wider search space and advancing it deeper into the search space.

6.3.3 Discussion

The results show the Cascading Search is a more effective alternative to the greedy best-first search algorithm. It leads to more solutions, more optimal solutions, and improved solutions in the majority of test cases. The ϵ-GBFS improved the performance of GBFS and Type-GBFS made a more notable improvement, but the Cascading Search had a more significant effect over the tests.

Cascading Search is effective because it methodically expands nodes at every depth of the search space. No matter the depth, if the heuristic chooses the wrong node, given time, the cascades will spread across the search space and expand the correct node. To see this in action we will look more closely at two of the problems. The first is Problem 7 from the IPC-Trucks problem set. To get data, the problem was run on the planner with a one minute time limit for both algorithms. The graph in Figure 6.3 has four plots, two for the Cascading Search and two for the standard GBFS search. The blue plot shows the number of nodes expanded by the GBFS at every depth. At the majority of depths up to depth 27, only one node has been expanded. Expanding that node led to an improved heuristic value that the GBFS followed deeper into the search space. Once the search had progressed to this point, it can't find more promising branches from nearer the initial state. The red plot shows the path that the best plan found took through the search space. For example, at depth 33, the node involved in the plan was the 12th node the search expanded at that depth. Once the search hit that region it explored a wide area expanding a large number of nodes, but the heuristic won't guide it back in a GBFS search. The black plot shows the number of nodes expanded by the Cascading Search algorithm. The cascades have expanded many nodes near the initial state exploring for more promising paths. The yellow plot shows the plan found. It very quickly veers up from the x-axis showing that the plan was found through the repeated cascades. The GBFS found multiple plans, but those plans are all similar. At most depths less than 30, the search only expanded one state forcing

every plan to have the same action at the same time. The best plan from GBFS was valued at 61, the best from Cascading Search at 3. The second example is the sixth problem from the IPC-Openstacks problem set and is shown in Figure 6.4. The shape of the search space is different. In this case the GBFS couldn't get past a certain depth because the heuristic wouldn't guide it that way. Even though the Cascading Search's cascades widen the search space, they led it to find a better path that it followed deeper into the search space. In this case Cascading Search found a plan of quality 35 and GBFS 137.

Exploring the branches is important because the heuristic might evaluate a node inaccurately. Even if it is accurate, it is easy for GBFS to go down the wrong path. When a node is expanded and leads to another node with an improved heuristic value, there are often many other nodes that were created with the same heuristic evaluation. If the next expansion again shows improvement, those nodes from the previous expansion will likely never be expanded even though they were just as promising according to the heuristic.

The primary distinction between Cascading Search and the two other improvements of GBFS is the use of randomness to diversify the search space. The Type-GBFS search widens the search space through its type system. It will expand many nodes that the heuristic evaluated as poor. An ϵ-GBFS search does the same thing in an even more unpredictable manner. The Cascading Search algorithm methodically widens the search space trusting that if the heuristic mis-values a node, it won't do so badly. By repeatedly expanding the most promising node at each depth, it can correct for mistakes. On the other hand, if the heuristic makes larger errors then the Type-GBFS will likely perform better.

On the Elevators problem set, both Cascading Search and Type-GBFS displayed the downside to exploration when the guiding heuristic was accurately guiding a GBFS search. The FF heuristic guiding GBFS solved all sixteen problems. Cascading Search solved twelve and Type-GBFS solved none. The exploration wasted effort. Type-GBFS spends more time expanding nodes that are not promising and therefore had a stronger effect.

A great deal of research in planning has been on developing more and more accurate heuristics. When the heuristic is accurate enough, a GBFS search is likely to quickly encounter a high quality goal state. However, even the most accurate heuristics make errors. The Cascading Search algorithm allows the search to explore the tree with a relatively aggressive manner, but with the ability to correct for mistakes.

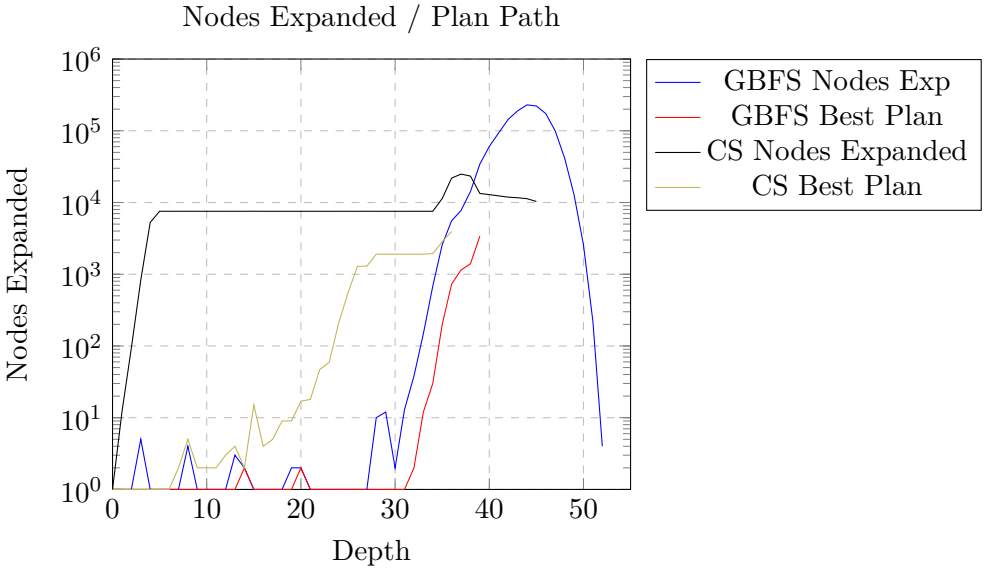

Figure 6.3: IPC-Trucks: Problem 7

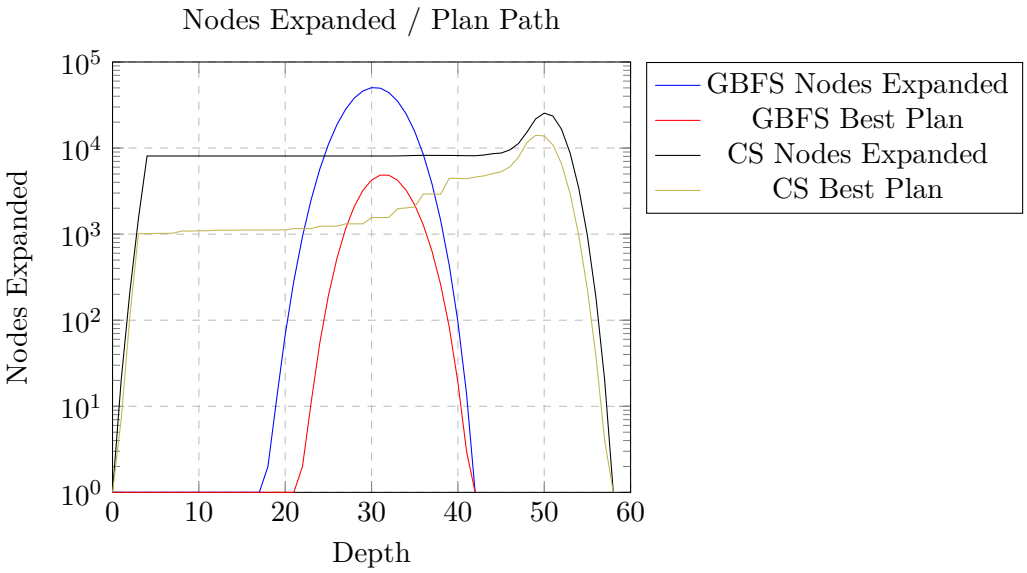

Figure 6.4: IPC-Openstacks: Problem 6

Chapter 7

Conclusion and Future Work

To conclude, we discuss the contributions of our work and possible directions for future work. We have presented a language for specifying preferences for planning domains. We have created adaptations of the landmark-counting heuristic for use in planning with preferences. Finally, we have presented Cascading Search, a novel search algorithms that adapts a greedy best-first search so that it systematically widens its exploration through the search space. Testing has demonstrated the effectiveness of the heuristics and the search algorithm.

7.1 Contribution

The Ranked Preference Specification (RPS) system is a powerful language for specifying preferences for planning problems. The preference orderings and ranks attached to them allow the desirability and importance of preferences to be specified with ease. Unbounded orderings allow more complex preferences that would otherwise be cumbersome, if not impossible, to be specified. They can compactly specify preferences such as the desire to maximize the number of times a fluent is true throughout a history. The RPS system allows for the fine control of incomparability. Many of the preference languages don't allow for incomparability and none allow it to be controlled as precisely as the RPS's approach. This is particularly important to properly model conditional preferences.

Although much research in planning has focused on developing heuristics, there has previously been little work crafting heuristics for planning with preferences. We adapted landmarks and the successful landmark-counting heuristic for use in planning with preferences. We proposed two heuristics that follow differing strategies using the value of preferences to weight the discovered landmarks. Testing has shown that the landmark-counting heuristic as well as our two adaptations are effective in guiding a planner, particularly when coupled with our new search algorithm.

We proposed a novel search algorithm, Cascading Search. It is an extension of the greedy best-first search algorithm. The Cascading Search algorithm balances following improving

heuristic values deep into the search space with cascades that systematically widen the entire search space. This approach allows Cascading Search to harness the power of heuristics by alternating between following the most promising states globally and expanding states at all depths that are promising but were passed over. In this way, it catches branches of the search space that were initially missed but may lead to high quality plans. Testing demonstrated the effectiveness of the algorithm. It leads to better plans than the basic greedy best-first algorithm and two variations. Cascading Search also found higher quality plans than the A* search, an algorithm that sacrifices speed for a high likelihood of finding a high-quality plan.

7.2 Future Work

Elements of the RPS system could be adapted and added to the PDDL language. This would make the RPS system more accessible for use. The preference orderings would integrate naturally into the language. The importance rankings assigned to the orderings as well as the varying strategies for interpreting a specification present more complications. The unbounded orderings would be difficult to integrate, but would add a great deal of functionality to the language.

We only used our heuristics to guide a planner on planning problems with goal preferences, but temporally extended preferences that consider the entire plan are more interesting. The strategy of transforming the problem by representing the temporal preferences with virtual automata, [3], may work well with the landmark-based heuristics. Alternatively, the automata could be used to discover landmarks that lead to the satisfaction of the preferences.

The Cascading Search algorithms can be applied to any area that utilizes a heuristically greedy best-first search. It should be effective in other areas of planning, as well as further afield.

In terms of the algorithms itself, the cascades of the algorithm widen the entire search space so that promising branches are not missed. However, the algorithm could be made more efficient if it could determine when it is no longer worthwhile to expand the remaining nodes at a specific depth. The search could track the heuristic quality of the nodes being expanded at each depth of the search space. As the search space widens, the heuristic value of each node expanded at a certain depth deteriorates and the probability that a high quality plan will include that node falls. If the algorithm could detect when the usefulness has fallen far enough, then it could stop expanding nodes at that depth, spending more time expanding more promising nodes.

Another possibility is to use Cascading Search as the local search algorithms with strategies that employ repeated local searches within the search space, [25, 57, 75]. As the Cas-

cading Search algorithm improves the performance of a global search, its usefulness would likely extend to the more focused local searches.

Bibliography

[1] Meysam Aghighi and Peter Jonsson. Oversubscription planning: Complexity and compilability. In Carla E. Brodley and Peter Stone, editors, *AAAI*, pages 2221–2227. AAAI Press, 2014.

[2] Vidal Alcázar, Daniel Borrajo, Susana Fernández, and Raquel Fuentetaja. Revisiting regression in planning. In Francesca Rossi, editor, *IJCAI*. IJCAI/AAAI, 2013.

[3] J. Baier, F. Bacchus, and S. McIlraith. A heuristic search approach to planning with temporally extended preferences. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages 1808–1815, Hyderabad, India, January 2007.

[4] Jorge A. Baier, Fahiem Bacchus, and Sheila A. McIlraith. A heuristic search approach to planning with temporally extended preferences. *Artificial Intelligence*, 173(5-6):593–618, 2009.

[5] Jorge A. Baier and Sheila McIlraith. On domain-independent heuristics for planning with qualitative preferences. In *Proceedings of the 7th IJCAI International Workshop on Nonmonotonic Reasoning, Action and Change (NRAC-07)*, Hyderabad, India, January 2007.

[6] J. Benton, Minh Binh Do, and Subbarao Kambhampati. Over-subscription planning with numeric goals. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI*, pages 1207–1213. Professional Book Center, 2005.

[7] J. Benton, Minh Binh Do, and Subbarao Kambhampati. Anytime heuristic search for partial satisfaction planning. *Artificial Intelligence*, 173(5–6):562–592, 2009.

[8] J. Benton, Menkes van den Briel, and Subbarao Kambhampati. A hybrid linear programming and relaxed plan heuristic for partial satisfaction planning problems. In Mark S. Boddy, Maria Fox, and Sylvie Thiébaux, editors, *ICAPS*, pages 34–41. AAAI, 2007.

[9] Meghyn Bienvenu, Christian Fritz, and Sheila A. McIlraith‘. Planning with qualitative temporal preferences. In *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR06)*, pages 134–144, Lake District, UK, June 2006. AAAI Press.

[10] Meghyn Bienvenu, Christian Fritz, and Sheila A. McIlraith. Specifying and computing preferred plans. *Artificial Intelligence*, 175(7–8):1308–1345, 2011.

[11] Avrim Blum and Merrick Furst. Fast planning through planning graph analysis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 95)*, pages 1636–1642, August 1995.

[12] Blai Bonet and Julio Castillo. A complete algorithm for generating landmarks. In Fahiem Bacchus, Carmel Domshlak, Stefan Edelkamp, and Malte Helmert, editors, *ICAPS*. AAAI, 2011.

[13] Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1–2):5–33, 2001.

[14] Blai Bonet and Malte Helmert. Strengthening landmark heuristics via hitting sets. In Helder Coelho, Rudi Studer, and Michael Wooldridge, editors, *ECAI*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 329–334. IOS Press, 2010.

[15] Blai Bonet and Menkes van den Briel. Flow-based heuristics for optimal planning: Landmarks and merges. In Steve A. Chien, Minh Binh Do, Alan Fern, and Wheeler Ruml, editors, *ICAPS*. AAAI, 2014.

[16] Craig Boutilier, Fahiem Bacchus, and Ronen I. Brafman. UCP-networks: A directed graphical representation of conditional utilities. *CoRR*, abs/1301.2259, 2013.

[17] Craig Boutilier, Ronen I. Brafman, Carmel Domshlak, Holger H. Hoos, and David Poole. Cp-nets: A tool for representing and reasoning with conditional ceteris paribus preference statements. *J. Artif. Intell. Res. (JAIR)*, 21:135–191, 2004.

[18] Sylvain Bouveret, Ulle Endriss, and Jérôme Lang. Conditional importance networks: A graphical language for representing ordinal, monotonic preferences over sets of goods. In Craig Boutilier, editor, *IJCAI*, pages 67–72, 2009.

[19] Ronen I. Brafman and Yuri Chernyavsky. Planning with goal preferences and constraints. In *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005)*, pages 182–191, Monterey, California, USA, 2005. AAAI.

[20] Ronen I. Brafman and Carmel Domshlak. Introducing variable importance tradeoffs into CP-nets. In Adnan Darwiche and Nir Friedman, editors, *UAI*, pages 69–76. Morgan Kaufmann, 2002.

[21] Gerhard Brewka. A rank based description language for qualitative preferences. In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence (ECAI04)*, pages 303–307. IOS Press, 2004.

[22] Daniel Burfoot, Joelle Pineau, and Gregory Dudek. RRT-plan: A randomized algorithm for STRIPS planning. In Derek Long, Stephen F. Smith, Daniel Borrajo, and Lee McCluskey, editors, *ICAPS*, pages 362–365. AAAI, 2006.

[23] 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.

[24] Jie Cheng and Keki B. Irani. Ordering problem subgoals. In *IJCAI*, pages 931–936, 1989.

[25] Andrew Coles, Maria Fox, and Amanda Smith. A new local-search algorithm for forward-chaining planning. In Mark S. Boddy, Maria Fox, and Sylvie Thiébaux, editors, *ICAPS*, pages 89–96. AAAI, 2007.

[26] E. Dahlman and A. E. Howe. A critical assessment of benchmark comparison in planning. 17, June 09 2011.

[27] James P. Delgrande, Torsten Schaub, and Hans Tompits. An extended query language for action languages (and its application to aggregates and preferences). In *Eleventh International Workshop on Non-Monotonic Reasoning NMR2006*, Lake District, UK, June 2006.

[28] James P. Delgrande, Torsten Schaub, and Hans Tompits. A general framework for expressing preferences in causal reasoning and planning. *J. Log. Comput.*, 17(5):871–907, 2007.

[29] Yannis Dimopoulos, Bernhard Nebel, and Jana Koehler. Encoding planning problems in non-monotonic logic programs. *Lecture Notes in Computer Science*, 1348:169–181, 1997.

[30] Minh Binh Do and Subbaro Kambhampati. Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP. *Artificial Intelligence*, 132(2):151–182, 2001.

[31] Patrick Doherty and Jonas Kvarnström. Talplanner: A temporal logic based forward chaining planner. *Ann. Math. Artif. Intell.*, 30(1-4):119–169, 2000.

[32] Carmel Domshlak, Michael Katz, and Sagi Lefler. When abstractions met landmarks. In Ronen I. Brafman, Hector Geffner, Jörg Hoffmann, and Henry A. Kautz, editors, *ICAPS*, pages 50–56. AAAI, 2010.

[33] Carmel Domshlak, Michael Katz, and Sagi Lefler. Landmark-enhanced abstraction heuristics. *Artificial Intelligence*, 189:48–68, 2012.

[34] Carmel Domshlak and Vitaly Mirkis. Deterministic oversubscription planning as heuristic search: Abstractions and reformulations. *J. Artif. Intell. Res. (JAIR)*, 52:97–169, 2015.

[35] Jon Doyle and Michael Wellman. Representing preferences as *ceteris paribus* comparatives. In *AAAI Spring Symposium on Decision-Theoretic Planning*, pages 69–75, 1994.

[36] Stefan Edelkamp. Planning with pattern databases. report00142, Institut für Informatik, Universität Freiburg, October 24 2000.

[37] Robert Feldmann, Gerhard Brewka, and Sandro Wenzel. Planning with prioritized goals. In *Proceedings, Tenth International Conference on Principles of Knowledge Representation and Reasoning*, pages 503–514, Lake District, UK, 2006. AAAI Press.

[38] Richard Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artif. Intell.*, 2(3/4):189–208, 1971.

[39] Angel García-Olaya, Tomás De La Rosa, and Daniel Borrajo. A distance measure between goals for oversubscription planning. In *roceedings of ICAPS'08 workshop on Oversubscribed Planning & Scheduling*, Sydney (Australia), 2008.

[40] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080, Cambridge, Massachusetts, 1988. The MIT Press.

[41] Michael Gelfond and Vladimir Lifschitz. Action languages. *Electron. Trans. Artif. Intell.*, 2:193–210, 1998.

[42] A. Gerevini and D. Long. Plan constraints and preferences in `PDDL3`. In *Proceedings of the ICAPS 2006 Workshop on Preferences and Soft Constraints in Planning*, 2006.

[43] Alfonso Gerevini, Patrik Haslum, Derek Long, Alessandro Saetti, and Yannis Dimopoulos. Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artificial Intelligence*, 173(5–6):619–668, 2009.

[44] Enrico Giunchiglia and Marco Maratea. Planning as satisfiability with preferences. In *AAAI*, pages 987–992. AAAI Press, 2007.

[45] Enrico Giunchiglia and Marco Maratea. Introducing preferences in planning as satisfiability. *J. Log. Comput*, 21(2):205–229, 2011.

[46] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. and Cybernetics*, SSC-4 (2):100–107, 1968.

[47] Patrik Haslum and Hector Geffner. Admissible heuristics for optimal planning. In Steve A. Chien, Subbarao Kambhampati, and Craig A. Knoblock, editors, *AIPS*, pages 140–149. AAAI, 2000.

[48] Patrik Haslum, John K. Slaney, and Sylvie Thiébaux. Minimal landmarks for optimal delete-free planning. In Lee McCluskey, Brian Williams, José Reinaldo Silva, and Blai Bonet, editors, *ICAPS*. AAAI, 2012.

[49] Malte Helmert. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence*, 173(5–6):503–535, 2009.

[50] Malte Helmert. The fast downward planning system. *CoRR*, abs/1109.6051, 2011.

[51] Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In Alfonso Gerevini, Adele E. Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *ICAPS*. AAAI, 2009.

[52] Malte Helmert, Patrik Haslum, and Jörg Hoffmann 0001. Flexible abstraction heuristics for optimal sequential planning. In Mark S. Boddy, Maria Fox, and Sylvie Thiébaux, editors, *ICAPS*, pages 176–183. AAAI, 2007.

[53] Malte Helmert, Patrik Haslum, and Jörg Hoffmann. Explicit-state abstraction: A new method for generating heuristic functions. In Dieter Fox and Carla P. Gomes, editors, *AAAI*, pages 1547–1550. AAAI Press, 2008.

[54] J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[55] J. Hoffmann, J. Porteous, and L. Sebastia. Ordered landmarks in planning. *Journal of Artificial Intelligence Research*, 22:215–278, 2004.

[56] Jörg Hoffmann. The metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *J. Artif. Intell. Res. (JAIR)*, 20:291–341, 2003.

[57] Tatsuya Imai and Akihiro Kishimoto. A novel technique for avoiding plateaus of greedy best-first search in satisficing planning. In Wolfram Burgard and Dan Roth, editors, *AAAI*. AAAI Press, 2011.

[58] K. B. Irani and J. Cheng. Subgoal ordering and goal augmentation for heuristic problem solving. In *Proc. of the Int. Joint Conf. on Artificial Intelligence*, pages 1018–1024, 1987.

[59] Peter Johnson and Christer Bäckström. State-variable planning under structural restrictions: Algorithms and complexity. *Artificial Intelligence*, 100(1–2):125–176, 1998.

[60] Erez Karpas and Carmel Domshlak. Cost-optimal planning with landmarks. In Craig Boutilier, editor, *IJCAI*, pages 1728–1733, 2009.

[61] Erez Karpas, David Wang, Brian C. Williams, and Patrik Haslum. Temporal landmarks: What must happen, and when. In Ronen I. Brafman, Carmel Domshlak, Patrik Haslum, and Shlomo Zilberstein, editors, *ICAPS*, pages 138–146. AAAI Press, 2015.

[62] Michael Katz and Carmel Domshlak. Implicit abstraction heuristics. *CoRR*, abs/1401.3853, 2014.

[63] Henry A. Kautz and Bart Selman. Planning as satisfiability. In *ECAI*, pages 359–363, 1992.

[64] Henry A. Kautz and Bart Selman. Unifying SAT-based and graph-based planning. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI'99)*, pages 318–325, Stockholm, Sweden, July 31-August 6 1999. Morgan Kaufmann.

[65] Emil Keyder, Silvia Richter, and Malte Helmert. Sound and complete landmarks for and/or graphs. In Helder Coelho, Rudi Studer, and Michael Wooldridge, editors, *ECAI*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 335–340. IOS Press, 2010.

[66] Jana Koehler and Jörg Hoffmann. On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm. *Journal of Artificial Intelligence Research*, abs/1106.0243, 2000.

[67] Jérôme Lang. Logical preference representation and combinatorial vote. *Ann. Math. Artif. Intell*, 42(1-3):37–71, 2004.

[68] Steven M. Lavalle. Rapidly-exploring random trees: A new tool for path planning, April 22 1998.

[69] Levi H. S. Lelis, Sandra Zilles, and Robert C. Holte. Stratified tree search: a novel suboptimal heuristic search algorithm. In Maria L. Gini, Onn Shehory, Takayuki Ito, and Catholijn M. Jonker, editors, *AAMAS*, pages 555–562. IFAAMAS, 2013.

[70] V. Lifschitz. Answer set planning. In Michael Gelfond, Nicola Leone, and Gerald Pfeifer, editors, *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-99)*, volume 1730 of *LNAI*, pages 373–374, Berlin, December 2–4 1999. Springer.

[71] Marco Maratea. Planning as satisfiability with IPC simple preferences and action costs. *AI Commun.*, 25(4):343–360, 2012.

[72] Eliseo Marzal, Laura Sebastia, and Eva Onaindia. On the use of temporal landmarks for planning with deadlines. In Steve A. Chien, Minh Binh Do, Alan Fern, and Wheeler Ruml, editors, *ICAPS*. AAAI, 2014.

[73] Vitaly Mirkis and Carmel Domshlak. Abstractions for oversubscription planning. In Daniel Borrajo, Subbarao Kambhampati, Angelo Oddi, and Simone Fratini, editors, *ICAPS*. AAAI, 2013.

[74] Vitaly Mirkis and Carmel Domshlak. Landmarks in oversubscription planning. In Torsten Schaub, Gerhard Friedrich, and Barry O'Sullivan, editors, *ECAI*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, pages 633–638. IOS Press, 2014.

[75] Hootan Nakhost and Martin Müller. Monte-carlo exploration for deterministic planning. In Craig Boutilier, editor, *IJCAI*, pages 1766–1771, 2009.

[76] Angel García Olaya, Tomás de la Rosa, and Daniel Borrajo. Using the relaxed plan heuristic to select goals in oversubscription planning problems. In José A. Lozano, José A. Gámez, and José A. Moreno, editors, *CAEPIA*, volume 7023 of *Lecture Notes in Computer Science*, pages 183–192. Springer, 2011.

[77] Edwin P. D. Pednault. Adl: Exploring the middle ground between strips and the situation calculus. In *KR*, pages 324–332, 1989.

[78] Julie Porteous, Laura Sebastia, and Jorg Hoffmann. On the extraction, ordering, and usage of landmarks in planning, May 08 2001.

[79] Silvia Richter, Malte Helmert, and Matthias Westphal. Landmarks revisited. In Dieter Fox and Carla P. Gomes, editors, *AAAI*, pages 975–982. AAAI Press, 2008.

[80] Silvia Richter, Jordan Tyler Thayer, and Wheeler Ruml. The joy of forgetting: Faster anytime search via restarting. In Ronen I. Brafman, Hector Geffner, Jörg Hoffmann, and Henry A. Kautz, editors, *ICAPS*, pages 137–144. AAAI, 2010.

[81] Silvia Richter and Matthias Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *J. Artif. Intell. Res. (JAIR)*, 39:127–177, 2010.

[82] Gabriele Röger and Malte Helmert. The more, the merrier: Combining heuristic estimators for satisficing planning. In Ronen I. Brafman, Hector Geffner, Jörg Hoffmann, and Henry A. Kautz, editors, *ICAPS*, pages 246–249. AAAI, 2010.

[83] Chris L. Schmidt and James P. Delgrande. Incorporating a qualitative ranked preference system into planning. In *Proceedings of the Twelfth International Workshop on Non-Monotonic Reasoning*, 2008.

[84] David E. Smith. Choosing objectives in over-subscription planning. In Shlomo Zilberstein, Jana Koehler, and Sven Koenig, editors, *ICAPS*, pages 393–401. AAAI, 2004.

[85] Tran Cao Son and Enrico Pontelli. Planning with preferences using logic programming. *TPLP*, 6(5):559–607, 2006.

[86] Richard Anthony Valenzano, Nathan R. Sturtevant, Jonathan Schaeffer, and Fan Xie 0001. A comparison of knowledge-based GBFS enhancements and knowledge-free exploration. In Steve A. Chien, Minh Binh Do, Alan Fern, and Wheeler Ruml, editors, *ICAPS*. AAAI, 2014.

[87] Menkes van den Briel, Romeo Sanchez Nigenda, Minh Binh Do, and Subbarao Kambhampati. Effective approaches for partial satisfaction (over-subscription) planning. In Deborah L. McGuinness and George Ferguson, editors, *AAAI*, pages 562–569. AAAI Press / The MIT Press, 2004.

[88] S. Vernhes, G. Infantes, and V. Vidal. Problem splitting using heuristic search in landmark orderings. pages 2401–2407, June 06 2013.

[89] Hongbing Wang, Jie Zhang 0002, Wenlong Sun, Hongye Song, Guibing Guo, and Xiang Zhou. WCP-nets: A weighted extension to CP-nets for web service selection. In Chengfei Liu, Heiko Ludwig, Farouk Toumani, and Qi Yu, editors, *ICSOC*, volume 7636 of *Lecture Notes in Computer Science*, pages 298–312. Springer, 2012.

[90] Letao Wang, Jorge Baier, and Sheila A. McIlraith. Viewing landmarks as temporally extended goals. In *Proceedings of the ICAPS09 Workshop on Heuristics for Domain Independent Planning*, 2009.

[91] Chih wei Hsu and Benjamin W. Wah. New features in sgplan for handling preferences and constraints in pddl3.0. In *In Proceedings of the Fifth International Planning Competition*, pages 39–42, 2006.

[92] Fan Xie, Martin Müller, and Robert Holte. Adding local exploration to greedy best-first search in satisficing planning. In Carla E. Brodley and Peter Stone, editors, *AAAI*, pages 2388–2394. AAAI Press, 2014.

[93] Fan Xie, Martin Müller, Robert Holte, and Tatsuya Imai. Type-based exploration with multiple search queues for satisficing planning. In Carla E. Brodley and Peter Stone, editors, *AAAI*, pages 2395–2402. AAAI Press, 2014.

[94] Fan Xie, Hootan Nakhost, and Martin Müller. Planning via random walk-driven local search. In Lee McCluskey, Brian Williams, José Reinaldo Silva, and Blai Bonet, editors, *ICAPS*. AAAI, 2012.

[95] Sung Wook Yoon, J. Benton, and Subbarao Kambhampati. An online learning method for improving over-subscription planning. In Jussi Rintanen, Bernhard Nebel, J. Christopher Beck, and Eric A. Hansen, editors, *ICAPS*, pages 404–411. AAAI, 2008.

[96] Lin Zhu and Robert Givan. Landmark extraction via planning graph propagation, May 09 2003.