Optimal temporal planning using the plangraph framework

Tien Ba Dinh

A thesis submitted to the University of Huddersfield
in partial fulfilment of the requirements for
the degree of Doctor of Philosophy

The University of Huddersfield

May 2007

ABSTRACT

The past few years have seen a rapid development in AI Planning and Scheduling. Many algorithms and techniques have been studied and improved to deal with more complex and difficult planning domains.

One such innovation was ***Graphplan***, first developed by Blum and Furst in 1995 and soon became one of the best approaches for optimal classical planning systems. Planning systems that use Graphplan's plangraph framework can find optimal plans for temporal planning problems, in which actions have durations. However, these systems have had strict assumptions on the preconditions and effects of actions, for instance, effects happen only at the end of the execution. In addition, the algorithm used in the solution extraction phase of these plangraph-based systems does not take full advantage of the information provided by the expansion phase to prune irrelevant search branches early.

With the ambition to make temporal planning problems more realistic, the thesis proposes an extension to the Planning Domain Definition Language (PDDL) 2.1 level 3, to allow actions to have intermediate effects. Our optimal temporal planning system, CPPlanner, is introduced as the first Graphplan-based optimal planner to handle the richer temporal domains (i.e. actions can have intermediate effects). Futhermore, the planner applies "critical paths" as a backbone for the search in the solution extraction phase, so that irrelevant search branches are pruned early. This improves the performance even in more restricted temporal planning domains.

In our experimental evaluation, CPPlanner outperforms two leading plangraph-based optimal temporal planning systems, TGP and TPSYS, in almost all test cases. The state-of-the-art optimal planner CPT and latest temporal planning domains in the international planning competition in 2004 and 2006 are also used in the experimental evaluation.

ACKNOWLEDGEMENTS

First of all, I would like to thank my supervisors Prof. Barbara Smith and Prof. Lee McCluskey for all their help, advice, encouragement, and friendship throughout. With their inspiration, and help, the PhD research became easier and more attractive to me.

I would like to thank Dr. Ian Miguel for his help on ILOG Solver and AI Planning, Dr. Stephen Cresswell for his help in compiling TGP, and fixing the parser for TPSYS, and Dr. Diane Kitchin for her advice and enthusiasm.

I would like to thank all members of the ARTFORM research group, who have also helped me and inspired me.

Many thanks to the staff at the University of Huddersfield, especially the technical and the administrative staff, for their promptly support on any problems that I have encountered.

I would like to express my gratitude to my best friend, Dr. Damian Jenkinson, for his kindness and support, especially his offer on accommodation during the writing-up stage.

I am grateful to CoThu, ChuMao, beNa, Vi, Thuy and anhTuan, who are like my family members in England, for their help, consideration, and encouragement throughout. I will never forget the beautiful time in Stockport.

I would like to thank Prof. Duong Anh Duc, all of my friends, and staff members at the Faculty of Information Technology at the University of Natural Sciences, Ho Chi Minh City, Vietnam for their support and encouragement.

I wish to show my appreciation to my girlfriend Le Hong Van for her love, support and encouragement throughout.

Finally, and most importantly, my grandmother, parents, brother, and sisters have been both supportive and encouraging throughout the completion of this thesis, for which I am forever indebted.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

It is hard to establish when Artificial Intelligence Planning (AI Planning) was first introduced. It is known that AI Planning was already considered to be a subfield of Artificial Intelligence in the late 1960s. AI Planning applications can now be found in different fields, such as autonomous manufacturing, space exploration, and the game industry.

In our daily lives, planning problems are everywhere, e.g. going to school, going shopping etc. Any of us is able to find plans in these situations. However, we have little knowledge on how our brains work to find the solutions. The mechanism to find a good plan for a planning problem is still a challenge to scientists nowadays.

One of the very first planning problems considered in AI was the monkey-banana problem which was proposed by John McCarthy in 1963 [70]. The initial state tells the position of the monkey, the banana, and the chair. With actions such as walk, move the chair to underneath the hanging banana, climb etc., the solution is a sequence of actions for the monkey to get the banana (i.e. the goal state). In this example, John McCarthy introduced the idea of modelling planning domains into situation calculus by using axioms.

Planning systems and definitions were gradually proposed and developed. However, each research group developed their own planning system with their own specification language for describing a planning domain. This was the situation until 1998, in order to prepare for the first international planning competition, Drew McDermott and other members of the committee proposed the first Planning Domain Definition Language (PDDL) [72] to the planning community. Since then, planning systems are compared more easily using PDDL. PDDL has been developed and extended further to model real-world domains, and is still being developed.

In 1995, Blum and Furst first introduced Graphplan [13] to the planning community. The algorithm's innovation is in forming the union of all the reachable states of a current state together to reduce space complexity, and in finding a solution in a later stage by backtracking search. This introduction was a big step forward in dealing with planning problems. Years later, scientists are still working within the plangraph framework to handle more complex problems, e.g. in temporal planning or planning with resources. Some have extended the plangraph framework to find an optimal solution for temporal planning, while others have used the relaxed planning graph as a part of the heuristic function for state-space search.

However, planning is still a long way from being able to solve realistic problems efficiently. Planning problems are generally NP-complete [30], even the simple famous BlocksWorld problems.

At the start of the research described in this thesis, TGP [88] and TPSYS [38] were the two well-known plangraph-based optimal planning systems for temporal planning domains. These planning system use the plangraph framework to find a plan with minimal *makespan*. However, there are several limitations of these two systems, such as a strict assumption on conditions and effects while executing actions. Effects of actions are allowed only at the end of the action in TGP and only at the beginning or at the end in TPSYS. Also, the approaches used in the solution extraction phase does not take advantage of all the information available after the graph expansion phase. This motivated us to develop CPPlanner. CPPlanner extends PDDL2.1 to allow actions with intermediate effects and handles them using the plangraph framework. It also uses "critical paths" provided by the expansion phase to add some actions and propositions to the final plan before the actual backtracking search takes place. Other improvements are introduced to CPPlanner to speed up the search.

1.1 Thesis structure

The thesis is constructed as follows:

- Chapter 2: Introduction to Artificial Intelligence Planning. This chapter introduces AI Planning. It begins with the definition of what is a planning problem, and a solution to a planning problem. Different types of planning and approaches are introduced. Then, the Planning Domain Definition Language (PDDL) with its extensions is described briefly. The syntax and semantics of actions with intermediate effects are proposed in this chapter. This extension allows actions of the current PDDL 2.1 level 3 to have intermediate effects.

- Chapter 3: Temporal Planning Systems. This chapter gives an overview of well-known temporal planning systems. Each temporal planner is introduced briefly and its capabilities discussed. The chapter focuses on TGP [88] and TPSYS [38], which are the two main plangraph-based optimal temporal planning systems. These two planners will be considered and compared directly to our system in the experimental study.

- Chapter 4: CPPlanner - An Optimal Temporal Planning System using Critical Paths. The chapter describes the CPPlanner framework. It begins with the plangraph framework and extends it to deal with temporal domains. Extensions to handle intermediate effects are also illustrated. The details of the algorithms for both the graph expansion phase and solution extraction are described fully. The use of "critical paths", which is one of the main contributions, is analysed and shown in detail. A simple example is also included to illustrate the algorithm.

- Chapter 5: Improvements. This chapter contains the improvements to CPPlanner. It introduces the timebound in selecting next actions for the search in the solution extraction phase. In addition, conflict-directed backjumping is discussed and added to CPPlanner to help the planner jump right back to the origin of the conflict. These improvements are developed and described in detail.

- Chapter 6: Experiments and empirical analysis. This chapter illustrates the performance and comparison of CPPlanner to TGP and TPSYS on different temporal domains. The domains are mainly from the planning competitions and the TGP package. In addition, CPPlanner is compared to the state-of-the-art optimal temporal planning system, CPT in latest temporal planning domains in the last two international planning competitions, IPC2004 and IPC2006. This chapter also shows the comparison of CPPlanner with the earlier version of CPPlanner, called CPPlanner_Basic which does not include "critical paths" in the solution extraction phase and the improvements of chapter 5.

- Chapter 7: Conclusion and future work. This chapter contains the summary of contributions of this thesis. It also incudes the future directions for CPPlanner.

- Appendix A: Comparisons a same planner on different operating systems. The chapter shows the empirical study of the performance of a planning system on different platforms. It illustrates there is not much difference in running time if planning systems are compared under the same hardware settings but on different platforms.

- Appendix B. The chapter contains the details of test suites which were used in the empirical study.

1.2 Contributions

This section introduces and outlines the main contributions of this thesis:

- Introduction of intermediate effects to PDDL2.1 level 3. Because of the complexity of using PDDL2.1 level 3 to introduce actions with intermediate effects into temporal planning, the extension in syntax and semantics of PDDL2.1 is introduced. This introduction helps to move planning problems towards real-world problems.

- "Critical paths". This new idea improves the performance of plangraph-based temporal planning systems. The introduction of "critical paths" helps the solution extraction phase add actions and propositions before the actual search takes place. Note that with actions and propositions being added to the plan early, the search for solutions will be improved even when actions do not have intermediate effects. Time-bounds and conflict directed-backjumping are also introduced to the backtracking search of the solution extraction phase to reduce the search space.

- Development of CPPlanner. CPPlanner is developed with the ability to handle temporal planning domains with intermediate effects. The new improvements of the solution extraction phase are included and tested. The planner shows the efficiency of the development via experimental results.

- Empirical analysis of CPPlanner and the introduction of a planning domains with intermediate effects. It contributes to the planning community the comparison of CPPlanner and other optimal temporal planning systems, TGP and TPSYS, on many temporal planning problems. The comparison to CPT is also included. Especially, a new temporal planning domain with actions and intermediate effects are introduced. The empirical study of CPPlanner on this domain is also described.

Chapter 2

INTRODUCTION TO ARTIFICIAL INTELLIGENCE PLANNING

2.1 Introduction

Artificial Intelligence planning (AI planning) has been a sub-field of Artificial Intelligence since the 1960s [79, 70, 71, 44]. It is a well-known area in Artificial Intelligence that studies to build algorithms and techniques for planning. AI Planning is applied in many areas in the real world such as automated data-processing [43, 17], autonomous manufacturing [75] and [45], space exploration [74, 18, 86, 57], robotics [11, 10, 12, 9], game industry [93], and large-scale logistics problems [104]. Scientists have studied and developed efficient algorithms and models to deal with real world problems due to their size and complexity. An overview of progresses and algorithms in AI Planning can be found in [101, 87, 66].

This chapter will describe briefly the definition of planning problems, types of planning, approaches and techniques, and the description language used in the International Planning Competitions: PDDL.

Firstly, it is necessary to define what is a planning problem. Different researchers have different definitions of planning problems. However, below is the common definition of what is a planning problem.

Definition 2.1 *A planning problem is described as an initial state containing propositions, a goal state (i.e. a set of propositions called subgoals) to be achieved, and a collection of actions, in which each has a set of preconditions (which need to be true for the action to be executed) and a set of effects (which will be true when the action is applied).*

Definition 2.2 *A solution (or **a plan**) to a planning problem is a sequence of actions which* *will transform the initial state to a state that satisfies all the propositions/subgoals of the goal state.*

In everyday usage, planning is the process of finding a plan for a given initial state and goal state before actually executing any of them. For example, when we are cooking, we have a plan which shows a sequence of actions that we will follow. These actions have to be followed in a predefined order.

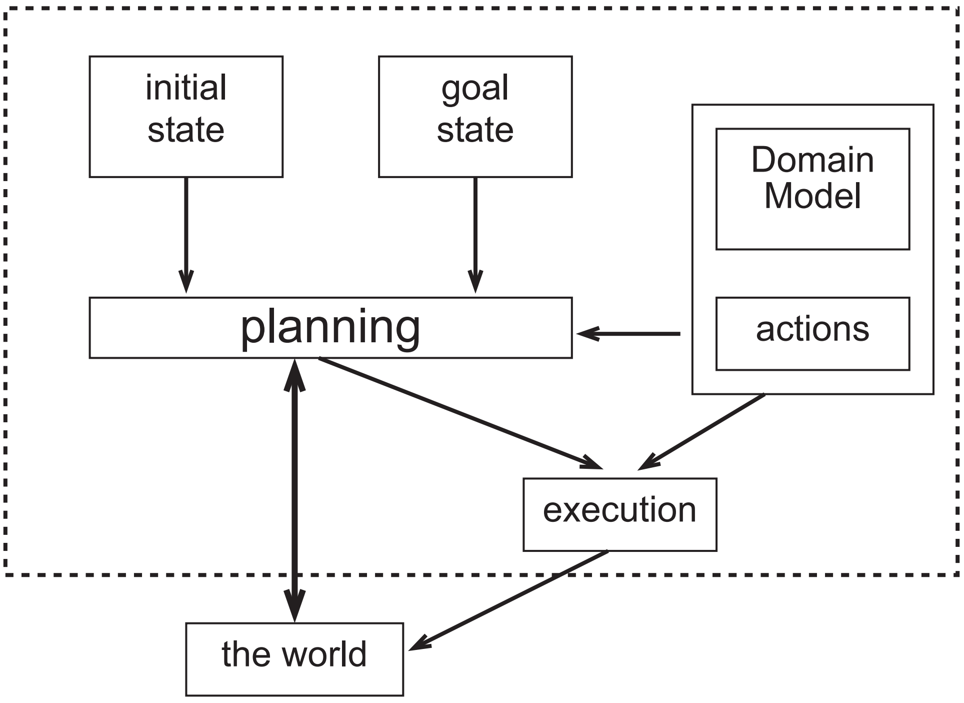

Figure 2.1: The abstract view of components in a planning system

Figure 2.1 shows the general view of components of a planning system. Given a initial state, a goal state, and a list of available actions, the planning system looks for a sequence of actions to achieve the goal state. When the plan is executed, the actions are applied to change from one state to another in the world to achieve the pre-defined goal state. The initial state, the goal state and actions are defined depending on the type of planning and the representation used. The details are described in the following sections.

With the hope of encouraging researchers to share their planning problems and algorithms, as well as to allow comparisons on performance of different planning systems, the Planning Domain Description Language (PDDL) was first proposed in 1998 by a committee led by McDewmott [72], and then extended by Fox and Long in 2002 [34], EdelKamp and Hoffmann in 2004 [27], and Gerevini and Long in 2006 [40]. The language has been used as a standard for modelling planning problems. The detail of the language is discussed later in this chapter.

AI Planning is categorized into different types, such as *classical planning*, *planning with time and resources*, *planning under uncertainty*, according to the expressivity of the representation of the planning problem (or the planning language required to describe the planning problem). The more expressive the planning language is, the bigger the search space to find a plan becomes. In the scope of this thesis, classical planning and its approaches are introduced and described briefly. The main focus is on temporal planning (i.e. planning with time).

However, there are other ways to classify a planning system. It can be classified as a *domain-independent* planner, or a *domain-dependent* planner, in which some hand-coded knowledge is provided for the domain.

Besides, planning systems might be classified according to the algorithms and techniques used, such as Graphplan-based planners (e.g. IPP [60], STAN [33], TGP [89]), SAT planners [56, 54], and HTN planners [103, 19, 76] etc.

The next sections introduce classical planning, and planning with time and resources. In each section, the representation and planning approaches, including algorithms and techniques, are also described.

2.2 Classical planning

2.2.1 Introduction

Early research in AI Planning was related to automated theorem proving [44]. In these systems, the initial state, goal state and actions are described in terms of axioms. Resolution theorem proving was used to produce a proof that a plan exists, and the actual plan found by applying answer extraction to the proof. However, those systems faced difficulties when it was required to specify axiomatically not only the changes that an action makes to a state but also the elements left unchanged. This encouraged the development of the classical formulation which introduced a simple solution to those types of problems.

Classical planning is a type of planning for restricted state-transition systems. These restricted systems have the following assumptions:

- The system has a finite set of states.
- The system is fully observable.
- The system is deterministic, i.e. a possible application of an action to a state only brings it to a single other state.
- The state remains unchanged until another application of an action.
- Actions have no duration.
- Goals are restricted and explicit. It means there are no constraints or conditions on the goal state.
- The solution is a sequence of linearly ordered finite actions.
- The planning is offline, i.e. there is no change while the system is planning.

2.2.2 Classical planning representations

In AI Planning, it is essential to have a description of the planning problem for planners. There are different types of representations [42] for classical planning: *set-theoretic representation*, *classical representation*, and *state-variable representation*. These representations have equivalent expressivity. It means that a planning problem can be represented equally well in any of those representations. In AI Planning, the classical representation is the most popular and often used by the community. The section below describes the classical representation.

2.2.2.1 Classical representation

In classical representation, states are represented as a set of logical atoms. Actions are represented as *operators* which change the truth values of atoms.

Firstly, the first-order language L has finite predicate symbols and constant symbols, but no function symbols. A *state* is a set of atoms of *L*. Because *L* is finite and has no function symbol, the set of all possible states *S* is finite.

Definition 2.3 *An operator is a triple* o = *(name(*o*), precond(*o*), effect(*o*)), in which:*

- *name(*o*) is the name of the operator. It is an expression in the form of* n$(x_1, ..., x_k)$ *where* n *is the* operator symbol*, and is unique; and every x_i is a variable symbol which can appear anywhere in* o.

- *precond(o) and effect(o) are preconditions and effects of the operator. These preconditions and effects are sets of literals (i.e. atoms and negations of atoms).*

- *The application of an instance of the operator to a state is described as follows:* $\gamma(s, a) = (s \setminus effect^{-}(a)) \cup effect^{+}(a)$.

Definition 2.4 *Let* L *be a first-order language, a* classical planning domain *is a restricted transition system* $\Sigma = (S, A, \gamma)$*, such that:*

- $S \subseteq 2^{\{allgroundatomsof\mathrm{L}\}}$
- *A* =$\{$*all ground instances of operators in* O$\}$
- γ*(s, a) = (s* $\setminus$ *effect*$^{-}$*(a))* $\cup$ *effect*$^{+}$*(a), if* a *is applicable to the state* s. *The new state constructed is in* S *as well.*

Definition 2.5 *A* classical planning problem *is triple* P = *(*Σ, s_0, g*), in which:*

- s_0 *is the initial state.*
- g *is the goal state - a set of literals which need to be achieved.*

2.2.3 Introduction to planning approaches

There are many different algorithms and techniques to deal with a wide variety of planning problems [87]. Some planning systems solve planning problems by searching through a graph representing the state space [77, 50, 6]. Each node in this space presents a state of the world. Arcs are state transitions or actions. Therefore, a plan is a path starting from the initial state, going through intermediate states, and ending at a goal state by applying a sequence of actions. This is called *state space planning*.

Other planning systems solve planning problems by searching through a space of plans [82, 8]. This approach is called *plan space planning*. Each node in this space is a partially specified plan. Arcs are plan refinement operations to achieve an open goal or to remove a possible inconsistency. In this approach, the planning system starts from an initial node which is an empty plan. The planner, then, is aiming at the final node containing a plan, which achieves the goal.

The planning graph approach is a synthesis of state space planning and plan space planning. It introduces a compact and powerful search space, which is called a *planning graph*. Starting from the initial state, also called the first level, the approach builds a next level containing all possible states reachable by applying actions to the current level. In the same level, if two or more states have parts in common, those parts are stored only once. Hence, the size of the graph is much smaller than the graph in the state space planning.

In addition, there are other approaches which apply Constraint Programming (CP), Satisfiability (SAT) techniques, or heuristics.

Those approaches can be applied to both classical planning and planning with time and resources. In the next sections, the classical representation is used in describing the approaches for classical planning. For planning with time and resources, the planning systems can use the same approaches but with extensions.

2.2.4 State-space planning

2.2.4.1 Introduction

As described earlier, in state-space planning, each node represents a state of the world. Arcs are state transitions. The solution plan is a path in the search space. In classical planning, classical state-space search algorithms are easy to understand. There are two approaches: *progressive* and *regressive* search: starting from the state representing the initial state or goal state respectively, the planning system searches through the state space to find the solution path leading to the goal state or initial state respectively. These two approaches are discussed in more detail in the following sections.

2.2.4.2 Progressive search

Progressive search, also known as *forward* search, is one of the simplest search algorithms in AI Planning. Starting from the initial state, the planner searches the state space to find a solution path leading to the goal state. Figure 2.2 is a general pseudo-code example of a progressive search algorithm in AI Planning:

In figure 2.2, the algorithm is a depth-first backtracking search. The algorithm might end up in an infinite search branch if there is a state s_i with $i < k$ in the sequence s_0, s_1, ..., s_k such that $s_i = s_k$. In this case the algorithm will have an infinite loop and repeatedly

```
01. Procedure ProgressiveSearch(O, $s_0$, $g$)
02.   $s \leftarrow s_0$
03.   $\pi \leftarrow \{\}$ //an empty plan
04.   Loop
05.     if $g$ is a subset of s then return $\pi$ as the solution plan and terminate
06.     // store the list of all possible actions
07.     possActions $\leftarrow$ a | a $\in$ O and precond(a) is satisfied in $s$
08.     If possActions=$\emptyset$ then return failure and backtrack
09.     // Try each action in the possible list
10.     For each $a_i$ in possActions do
11.       s $\leftarrow \gamma$ (s, $a_i$)
12.       $\pi \leftarrow \pi \bigoplus a_i$
13.       run the loop to continue
14.     od {end for}
15.   End {loop}
```

Figure 2.2: Pseudo-code for `ProgressiveSearch (O, $s_0$, $g$)` extracted from [42].

return to this state. In order for the algorithm to be complete, these infinite search branches must be pruned (i.e. the algorithm must check and return failure at any time it finds a state which is the same as a state earlier in the list π).

2.2.4.3 Regressive search

Regressive search, also known as *backward* search, is another approach in state-space planning. The solution plan is extracted from the state space by starting the search from the goal state, traversing through other states backwards, and ending at the initial state. The figure 2.3 shows the pseudo-code of the regressive search.

Like the progressive search, the regressive search also keeps a record of the list of state (..., s_k, ..., s_g) while searching. In order to avoid the infinite loop, the algorithm returns

```
01. Procedure RegressiveSearch(O, $s_0$, $g$)
02.   $s \leftarrow g$
03.   $\pi \leftarrow \{\}$ //an empty plan
04.   Loop
05.     if $s_0$ is a subset of s then return $\pi$ as the solution plan and terminate
06.     // store the list of all possible actions
07.     possActions $\leftarrow$ a | a $\in$ O and effect(a) $\cap\, s \neq \emptyset$
08.     If possActions=$\emptyset$ then return failure and backtrack
09.     // Try each action in the possible list
10.     For each $a_i$ in possActions do
11.       s $\leftarrow \gamma^{-1}$ (s, $a_i$)
12.       $\pi \leftarrow a_i \bigoplus \pi$
13.       run the loop to continue
14.     od {end for}
15.   End {loop}
```

Figure 2.3: Pseudo-code for `RegressiveSearch (O, `s_0`, `g`)` extracted from [42].

failure whenever it finds a state s_j, in which j > k and $s_j \subseteq s_k$.

2.2.4.4 STRIPS algorithm

In the previous two sections, the progressive and regressive search are introduced. However, the size of the search space is very big. The STRIPS algorithm [31], which was developed by Fikes and Nilsson in 1971 at Stanford university, is one of the first attempts to reduce the search space. The algorithm is very similar to regressive search, but is different in the following respects:

- In each recursive call to the STRIPS algorithm, only actions which have a part of their effects in common with the preconditions of the last chosen action are considered. This prunes the search space. However, it means that the search is incomplete.

```
01. **Procedure** STRIPS(O, $s$, $g$)
02.     $\pi \leftarrow \{\}$ // an empty plan
03.     **Loop**
04.         if $s$ satisfies g then return $\pi$
05.         A $\leftarrow \{a \mid a$ is a ground instance of an operator in O,
06.                 and a is relevant for $g\}$
07.         if A = $\emptyset$ then return failure.
08.         choose any action $a \in$ A nondeterministically.
09.         $\pi$' $\leftarrow$ STRIPS(O, $s$, precond($a$))
10.         if $\pi$' = failure then failure
11.         s $\leftarrow \gamma^{-1}$ ($s$, $\pi$')
12.         s $\leftarrow \gamma^{-1}$ ($s$, $a$)
13.         $\pi \leftarrow \pi.\pi$'.$a$
15.     **End** {loop}
```

Figure 2.4: Pseudo-code for STRIPS (O, s, g) extracted from [42].

- If at the current state, an action has all of its preconditions satisfied, STRIPS will execute the action and not backtrack over this commitment.

2.2.5 Plan-space planning

2.2.5.1 Introduction

This section introduces plan-space planning. In state-space planning, nodes represent states of the search space, arcs are transitions or actions between states, and a solution plan is a path of states from the initial state to the goal state. However, in plan-space planning, as described earlier, nodes are *partially specified plans*. Arcs are *plan refinement operations* which further complete a partial plan. Starting with an empty plan, the algorithm goes through the refinement operation to aim at the final node which contains a solution plan to achieve all the goals.

2.2.5.2 Search space

A solution plan is a sequence of actions which is organised in an order to achieve the goal. A partial plan can be considered as any subset of actions of this sequence.

A partial plan is gradually refined by adding actions, add ordering constraints for actions, adding causal links, and variable binding constraints. Ordering constraints tell the system the relationship between actions, i.e. which one needs to be done before the other. A causal link is added when a precondition of an action is supported by another action. Variable binding constraints make sure concerned objects of the relating actions are bound together.

A plan space is a directed graph in which vertices are partial plans and edges are refinement operations. The directed edge from vertex A to vertex B means a refinement that transform partial plan A to a successor partial plan B. The refinement operation can be one or more of the followings:

- Adding an action to A.
- Adding an ordering constraint to actions in A.
- Adding a causal link into A.
- Adding a varible binding constraint to A.

Planning for this approach is a search through this directed graph, starting from the initial partial plan to the solution plan. For each partial plan, there are subgoals which are unsupported preconditions. The refinement tries to add things into the partial plan while still keeps it satisfied.

```
01. Procedure PSP($\pi$)
02.     $flaws \leftarrow$ OpenGoals($\pi$) $\cup$ Threats($\pi$)
03.     if $flaws$ = $\emptyset$ then return($\pi$)
04.     select any $flaws$ $\phi \in flaws$
05.     $\psi \leftarrow$ Resolve($\phi$, $\pi$)
06.     if $\psi$ = $\emptyset$ then return(failure)
07.     choose a $\omega \in \psi$ nondeterministically
08.     $\pi$' $\leftarrow$ refine($\omega$, $\pi$)
09.     return(PSP($\pi$'))
```

Figure 2.5: Pseudo-code for plan-space planning PSP (π) extracted from [42].

2.2.5.3 Algorithms

A general algorithm for plan-space planning is described as in figure 2.5.

The algorithm starts with a partial plan, and tries to find its flaws, i.e. its *opengoals* and its *threats*. Then, the algorithm selects one of these flaws and tries to find ways to resolve it. Next, it will choose a resolver for the flaw and refine the partial plan.

The process of finding flaws consists of finding opengoals and threats. Opengoals are preconditions which are not supported by a causual link yet. Threats are actions which cause problems to the causal link of other two actions (i.e. a_k threatens the causal link $a_i \rightarrow a_j$). Finding threats can be done by checking all triple of actions (a_i, a_j, a_k) in the partial plan.

The process of resolving consists two parts: resolving opengoals and resolving threats. To resolve an opengoal is to find an action which can provide that proposition (i.e. opengoal). To resolve the threat a_k of the causal link $a_i \rightarrow a_j$ is to put a constraint forcing whether a_k happens before a_i or after a_j.

2.2.6 Planning graph

2.2.6.1 Introduction

In state-space planning, the plan is a sequence of actions, whereas in plan-space planning, planners synthesize a plan as partially ordered set of actions, i.e. any sequence that meets the constraints of the partial order is a valid plan. The planning graph approach applies both. The approach was first introduced by Blum and Furst in 1995 in the Graphplan planner [13]. The planning graph is *directed layered graph*, which is constructed based on the reachability analysis (*Note:* see the next section for details of constructing the graph).

The planning graph approach has been analyzed, extended and improved in many planning systems to speed up the search [22, 13, 38] or deal with more expressive planning domains. In addition, it has also been modified as a relaxed planning graph to build up heuristic functions [80, 23, 49, 14] to guide the search in bigger and more complicated domains.

2.2.6.2 Description

Starting from the initial state in the level 0, the planner will find all the possible actions including *no-op* actions (i.e. actions doing nothing) which have preconditions in level 0 to construct the level 1. This means that level 2 contains all the propositions which could be true as a result of applying all possible actions to the initial state. The process repeats to construct the whole planning graph. Thus, the planning graph is a directed layered graph which alternates between a level of propositions and a level of actions.

In the planning graph, each proposition level is a *union* of all possible states of that level. The figure 2.6 below illustrates an example of a planning graph.

2.2.6.3 Mutual exclusion

With the construction of the planning graph described in the above section, a certain proposition level contains all possible states which can be reached at this level. However, because

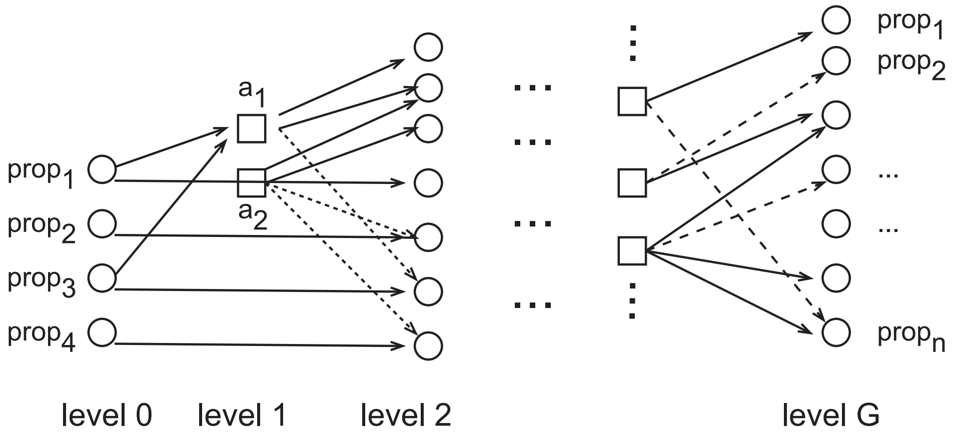

Figure 2.6: The overview of a planning graph

the algorithm stores states as a union of them, there might be two actions which cannot happen simultaneously due to a conflict in preconditions or effects. For example, an action may delete preconditions of another action. Hence, not all the propositions of the proposition level may be simultaneously true. Therefore, the *mutual exclusion relations*, also called as *mutex relations* or *mutexes*, are introduced to identify whether actions or propositions can appear simultaneously.

Definition 2.6 *Two actions a and b are* mutex *if:*

- [*Inference*] *either action deletes a precondition or an added effect of the other actions.*

- [*Competing needs*] *A precondition of a is logically inconsistent with a precondition of b in the previous level.*

Definition 2.7 *Two proposition p and q are mutex if:*

- *They are negations of each other, i.e. p = not q.*

- *All possible ways to create p are logically inconsistent with all possible ways to create q.*

2.2.6.4 Graphplan planner

The Graphplan planner [13] was introduce by Blum and Furst in 1995. It is an optimal planner which finds an optimal solution (i.e. using the minimal steps) for classical planning domains.

Graphplan contains two phases: the *graph expansion* and the *solution extraction*. In the graph expansion, from the initial state, the planner applies possible actions and their effects to advance the planning graph to the next level. The process is repeated until all of the propositions of the goal state appear and are pairwise non-mutex. At this time, the solution extraction is called to look for a solution in the planning graph. In the solution extraction, starting from the propositions of the goal state, the planner looks for all possible supporting actions and tries to add them into the plan. The preconditions of these selected actions are then added. If it leads to a dead-end in the search tree, the planner will backtrack and try other actions. The process continues to search backwards towards the initial state. If it reaches the initial state, a solution is found and is optimal. At this time, the algorithm is terminated. Otherwise, if all possible search branches have been tried, the graph expansion is called again to advance the planning graph to the next level. If the planning graph has levelled-off (i.e. the new level is the same as the previous one), there is no solution and the algorithm stops.

The extension of Graphplan to deal with temporal planning is discussed in detail in the next chapter. It describes the detail of the algorithms of TGP and TPSYS and the improvements over Graphplan. Chapter 4 then describes in detail CPPlanner, which uses on the planning graph to deal with temporal domains in which actions can have intermediate effects.

2.2.7 Constraint Programming in planning

2.2.7.1 Introduction

Constraint programming is a very powerful approach to find an optimal solution. Encoding the planning problems as CSPs [97, 22, 69, 68, 96] allows to use efficient built-in search algorithms and techniques in Constraint Programming Solvers. Graphplan has been adapted to use constraint programming [22, 68] and is a typical example to show that planning problems can be encoded to CSPs and solved using CSP algorithms.

2.2.7.2 Constraint Satisfaction Problem

A constraint satisfaction problem is defined as P = (X, D, C), in which:

- X = $\{x_1, x_2, ..., x_n\}$ is a finite set of variables.
- D = $\{D_1, D_2, ..., D_n\}$ is a set of finite domains for corresponding variables, $x_i \in D_i$.
- C = $\{c_1, c_2, ..., c_m\}$ is a finite set of constraints.

Definition 2.8 *A* solution *to a CSP is an assignment of (v_1, ..., v_n), in which $v_i \in D_i$, to variables (x_1, ..., x_n) which satisfy all of the constraints in C.*

For example, in Graphplan, the solution extraction process can be encoded as a dynamic constraint satisfaction problem [22]. In this encoding, *variables* are propositions which are looking for supporting actions. The *domain* for each variable consists of all possible supporting actions of that proposition. The CSP starts with variables of all propositions in the goal state. Each time new propositions appear which are preconditions of chosen actions, new variables are created and added to the CSP. The CSP solver tries to look for a possible assignment to variables. If there is a solution for the CSP problem, it is then mapped back to propositions/actions to have the solution plan for the problem.

Constraint programming is successfully used for resource allocation and scheduling problems. In AI Planning, applying constraint programming is still at the early stage. However, it is a promising approach for AI planning, especially for optimal planning systems. Besides the above example of encoding the solution extraction of Graphplan into CSP, there have been a few studies of applying constraint programming into AI Planning, such as CPlan developed by Peter van Beek [97], Encoding Temporal Planning as CSP by Mali [69], Generalizing GraphPlan by Formulating Planning as a CSP [68], and Utilizing Structured Representations and CSPs in Conformant Probabilistic Planning [52].

2.3 Planning with time and resources

In classical planning, actions are assumed to have no duration. The actions can take place instantaneously. However, this is unrealistic. In addition, resources are also considered in real world problems. For instance, in order to fly from city A to city B, it will take dur$_{AB}$ time and consume fuel$_{AB}$. With the ambition to deal with real world problems, temporal planning and planning with resources have been introduced.

With the introduction of time into planning, actions can take place concurrently as long as they are not in conflict with one another. Hence, the plan is now a sequence of actions attached with their starting time. With time and resources, the expressivity of the problem description increases, and so does the complexity. The search space becomes much bigger comparing to that of the classical planning. The next sections will show an overview of planning with time and resources.

2.3.1 Planning with time

2.3.1.1 Introduction

Unlike classical planning, in temporal planning, actions now have duration. This section will introduce the representation and techniques for temporal planning.

2.3.1.2 Temporal representation

According to the description of a temporal planning domain in the book [42], it has:

- *constant symbols* are objects in the planning domains, such as cars, places, people.
- *variable symbols* are either *object variables* that are typed variables or *temporal variables*.
- *relation symbols* are either *rigid relation symbols* representing relations that do not change during time, for example attach(crane, location), or *flexible relation symbols* representing relations of the constants that may or may not hold at some instant for a planning problem, for example at(car1, London).
- *constraints* are either *temporal constraints* which reside within point algebra calculus, e.g. $t_1 < t_2$, or *binding constraints* on object variables, which are expressions of the form x=y, x≠y and x $\in$ D, with D being a set of constant symbols.

The rigid relations and binding constraints are *object constraints*, which are time-invariant in this representation scheme.

The *temporally qualified expression* is an expression in the form: $\rho(\zeta_1,..., \zeta_k)@[t_s, t_e)$.

in which, ρ is a flexible relation, ζ_i is a constant or object variable, and t_s, t_e are temporal variables, such that: $\forall$ t $\in$ $[t_s, t_e)$, $\rho(\zeta_1,..., \zeta_k)$ holds at t.

A *temporal database* is defined as ϕ = (F, C), in which F is a finite set of temporally qualified expressions, and C is a finite set of temporal and object constraints.

A *temporal planning operator* is a tuple o = (name(o), precond(o), effects(o), const(o), in which:

- name(o) is in the form of o(x_1, ..., x_k, t_s, t_e) such that o is the operator symbol, x_1, ..., x_k are object variables which appear in o, and t_s, t_e are temporal variables in const(o).

- precond(o) and effects(o) are temporally qualified expressions.
- const(o) is a conjunction of temporal constraints and object constraints.

An *action* is an instance of the operator. The action a is applicable to ϕ=(F, C) iff precond(a) is supported by F within some consistent constraints.

2.3.1.3 Approaches in temporal planning systems

This section describes a brief introduction to different approaches and techniques in temporal planning, and illustrates temporal planning systems. Chapter 3 will discuss in detail these temporal planners.

Simple Temporal Constraint Problem (STP) and Temporal Constraint Network Problem (TCSP), which were introduced by Dechter *et al.* [21], have been widely used in several temporal planning systems [15, 51, 18], and many improvements have been introduced [16].

In temporal state-space planning, planning systems have applied heuristics to guide the search, such as TLplan [4], and later developments [5, 7, 3], TALplanner [24, 62, 61, 25], and the HS planner [41].

In plan-space planning, ZENO planner [83] dealt with rich representations with variable durations and linear constraints. It is one of the earliest planning systems which can deal with complex planning domains including deadline goals, metric conditions and effects, and continuous changes.

In the planning graph approach, TGP [89], TPSYS[38], and SAPA [23] deal with durative actions. TGP and the early version of TPSYS extend the Graphplan planner to find an optimal solution for temporal planning domains. SAPA applies a heuristic search based on the planning graph.

In HTN planning, several planning systems, such as O-Plan [19], and SHOP2 [78] use time windows and constraints in their representation and processes.

These approaches proved to successfully perform at the International Planning Competition 2002. However, they make restrictive assumptions on actions, such as that actions only have effects at the beginning or at the end of their execution.

2.3.2 Planning with resources

2.3.2.1 Introduction

Resources are an important part of making planning domains/problems more realistic. However, since our main topic is working on temporal planning and planning systems, planning with resources is not discussed in very detail. The section only gives an overview of planning with resources.

In planning problems, resources can be categorized into two types: *consumable* resources and *reusable* resources.

A consumable resource is consumed by an action during its execution. Fuel is a typical example of a consumable resource. A reusable resource is used by an action during its execution; the resource is then released and unchanged when the action finishes. For example, cars, locations are reusable resources.

Reusable resources are constraints on the number of actions which can perform in parallel, whereas consumable resources are constraints on whether the action can perform freely during the lifespan of the plan.

Consumable resources are quantitative resources. These are encoded as a quantity with a state. The value of this resource is often numeric. The consumption of this resource will reduced its value accordingly. In some planning domains, there are actions which allow to restore this resource. For example, an action drive(car, A, B) can consume an amount of fuel. Another action refuel(car) can restore the fuel level for the car.

Reusable resources are qualitative resources. They are represented by the states of objects. For example, when a robot arm is empty (or available), it means it is ready for actions which require this resource to be executed.

2.3.2.2 Approaches

There are several approaches to deal with planning domains with resources. In the work of Srivistava and Kambhampati [94, 95], the authors tried to separate resources from planning. Resources are then managed and attached into the skeletal plan which is constructed without resources earlier. The main difficulty of this idea is when the resource constraints are too tight. In this case, it is very difficult to manage them into the skeletal plan. Because of this difficulty, the attempt to separate resources from planning has been abandoned.

Many researchers have studied and implemented several planning systems to handle resources. Penberthy and Weld [83] introduced ZENO planner, Laborie and Ghallab with IxTeT [63], or Drabble and Tate with O-Plan [26], or Koehler's work [58, 59] with introduction of resources into the Graphplan framework. ZENO, IxTeT, and O-Plan are among earliest planners which can handle planning domains with resources. However, due to the limitation of the approaches they used at that time, their performance was not as good as the current planning systems with new approaches.

The common idea in those systems is to consider resources as constraints and use specialized solvers to deal with these constraints. Dealing with resources makes planning similar to scheduling. Systems, such as O-Plan and IxTeT, have used scheduling techniques to solve the planning problems. O-Plan with optimistic and pessimistic resource profile, and IxTeT identifying the minimum critical set of actions which have resource conflicts.

In ZENO, IxTeT, RealPlan and Koehler's work, they also consider time as a resource. ZENO and IxTeT used interval representation for actions and propositions, and applied constraint programming techniques to manage the relationships between intervals. In RealPlan and Koehler's work, time is considered as steps, in which each corresponds to a set of possible actions which can take place in parallel.

2.4 Planning Domain Definition Language (PDDL)

2.4.1 Introduction

The Planning Domain Definition Language (PDDL) was introduced in order to help the planning community to share planning models, problems and be able to compare their systems. It became the standard language for the first International Planning Competition in the community in 1998. Our planning system, CPPlanner, also uses PDDL as the language for planning domains and problems. This section describes an overview of the language and its development in the last few years.

Planning Domain Definition Language (PDDL) was first introduced by McDermott in 1998 with the purpose to make a standard planning language for the first International Planning Competition. Since then, it has been accepted and widely used by the planning community to share and exchange planning models. PDDL is an action-based planning definition language, which was mainly inspired from the ideas of STRIPS. In the first international planning competition in 1998, it was used as the standard definition language for all participating planners. Before the introduction of PDDL, in the planning community, each planning system used its own conventions on the input and output data. This caused difficulty in comparing and sharing planning models. PDDL provides the standards for the input and output data for planning systems. PDDL encourages scientists to develop and compare the performance of their systems.

With the ambition to deal with realistic planning problems, PDDL has been extended with more expressivity to describe more complex and realistic planning domains. In 2002, it was extended to PDDL 2.1 which is able to model temporal planning domains and domains with resources. In the international planning competition 2004, PDDL was extended to PDDL 2.2, which added derived predicates and timed initial literals. Recently, Gerevini and Long [40] extended it further by allowing to express strong and soft constraints on plan trajectories (i.e. constraints over possible actions in the plan and intermediate states reached by the plan). Strong constraints are constraints which must be satisfied, while soft constraints are ones attached with weights or values which are desired to be satisfied as "much" as possible. They also proposed strong and soft problem goals (i.e. goals that must be achieved in any valid plan, and goals that we desire to achieve, but that do not have to be necessarily achieved). The following sections will introduce PDDL and its development.

2.4.2 PDDL

PDDL uses Lisp-like syntax to describe planning problems. It was built based on the formalisms of existing planning systems at that time, such as STRIPS [31], ADL [81], UCPOP [82]. It is an action-centered language. PDDL separates planning domain descriptions from problem descriptions. The planning domain description shows the general domain behaviours via parameterised actions. The problem description contains initial state and goal state of the problem. A planning problem is a pair of a domain description and a problem description. Normally, one domain description can be paired with many problem descriptions to create different planning problems in the same domain.

In the domain description, actions are described at an abstract level. In addition to preconditions and effects, actions also have parameters which are assigned values when the actions are applied. The preconditions and effects (i.e. post-conditions) are logical propositions, objects and logical connectives.

Because PDDL is a general planning language, many planners just support a part of it. In a domain description, requirements are introduced so that planning systems know quickly whether they can handle it. Below are the most commonly-used requirements:

```
:strips
    Description consists STRIPS only.
:typing
    Domains uses types. It is used to declare
    parameter and object types.
:adl
```

```
    Some or all of the domain description
    uses ADL syntax, e.g. actions have quantified
    and conditional effects, disjunctions and
    quantifiers in preconditions and goals.
:equality
    Domain uses "=" predicate as equality
```

In the original PDDL, the semantics is not formally described for the syntax proposed. However, the language is widely accepted by the planning community. The language has the power to express types of objects in planning domains, constrain types of arguments of predicates, actions with negative preconditions or conditional effects etc. These expressive abilities were fully described and proposed as ADL. It also attempted to form a standard syntax for describing hierarchical domains which are used in HTN planners. However, this attempt of proposing a standard syntax for hierarchical domains was not successfully explored and was removed in PDDL 2.1. In addition, it also tried to propose a standard syntax for numeric-valued fluents. However, in the planning competition in 1998 and even 2000, this part of the language is not applied.

Figure 2.7 shows a simple example of a domain description. This domain description is extracted from the gripper domain of the planning competition in 1998.

```
(define (domain gripper-strips)
     (:predicates  (room ?r)
                   (ball ?b)
                   (gripper ?g)
                   (at-robby ?r)
                   (at ?b ?r)
                   (free ?g)
                   (carry ?o ?g))
                   ...
     )
)
```

Figure 2.7: A part of gripper domain extracted from planning competition 1998

```
(define (domain jug-pouring)
    (:requirements :typing :fluents)
    (:types jug)
    (:functors
        (amount ?j -jug)
        (capacity ?j -jug)
         -(fluent number))
    (:action empty
     :parameters (?jug1 ?jug2 - jug)
     :precondition (fluent-test
         (>= (-(capacity ?jug2) (amount ?jug2)) (amount ?jug1)))
     :effect (and (change (amount ?jug1) 0)
                  (change (amount ?jug2)
                  (+ (amount ?jug1)(amount ?jug2))))) )
```

Figure 2.8: Jug-pouring domain description, extracted from the AI Magazine article [73]

The example in the figure 2.8, shows how a planning system can tell from the domain's requirements whether or not it can handle the domain. For example, in the requirements of the domain, the planner is required to handle fluents.

In PDDL, numbers are supported, and numeric quantities can be assigned and updated. Figure 2.8 is also an example of numeric fluents used in PDDL. In this example, the water is poured from jug1 to jug2 with the condition that jug2 is big enough to hold the water from jug1. The effect of this action updates the quantities in each jug by a discrete quantity. PDDL is also tweaked to handle resource consumption without using numeric fluents. Figure 2.9 shows that the fuel is consumed by the car when it moves from one location to another.

```
(define (domain vehicle)
    (:requirements :strips :typing)
    (:types vehicle location fuel-level)
    (:predicates
        (at ?v - vehicle ?p - location)
        (fuel ?v - vehicle ?f - fuel-level)
        (accessible ?v - vehicle ?p1 ?p2 - location)
        (next ?f1 ?f2 - fuel-level))
    (:action drive
     :parameters (?v - vehicle ?from ?to - location
                  ?fbefore ?fafter - fuel-level)
     :precondition (and (at ?v ?from) (accessible ?v ?from ?to)
                        (fuel ?v ?fbefore) (next ?fbefore ?fafter))
     :effect (and (not (at ?v ?from))
                  (at ?v ?to)
                  (not (fuel ?v ?fbefore))
                  (fuel ?v ?fafter)))
)
```

Figure 2.9: Vehicle domain description, extracted from Fox and Long’s paper [37]

Because of the complexity of PDDL, the first International Planning Competition in 1998, there were only 5 participants, in which three planners only supported STRIPS, and the other two supported STRIPS/ADL. To encourage more participants, in the second competition in 2000, the PDDL was restricted. In particular, negative preconditions were removed; parsing of ADL actions was simplified to avoid unnecessary "nesting"; and objects were explicitly listed in the problems.

In short, the PDDL is the first planning language which is recognised and widely used by the community. It helps the community to share planning problems, resources and compare planning systems' performance. However, the semantics need to be formally defined. In addition, in order to encourage more people using PDDL as a standard language for planning domains and problems, more structure, guidelines and supporting tools need to be developed and introduced.

2.4.3 PDDL 2.1

PDDL 2.1 [37] was based on the original PDDL with the extension to allow numerical variables, and concurrent execution of durational actions. This extension was introduced in the international planning competition in 2002. PDDL 2.1 has 5 levels, in which level 1 is ADL planning, level 2 adds numerical variables, level 3 introduces discretised durative actions, and level 4 extends level 3 to allow continuous durative actions under certain restrict assumptions. Finally, level 5, which is also known as PDDL+, removes the assumptions and introduces processes and events allowing to model complex discrete and continuous real-time systems. The higher the level is, the more expressive language is. However, because planning systems could not handle the complexities of level 4 and 5, only the first 3 levels were used in the planning competition in 2002. The syntax and semantics of the first 3 levels are formally defined in [37]. Level 4 and 5 were defined in [35].

CPPlanner, the system described in this thesis, deals with PDDL 2.1 level 3 with some extensions to allow intermediate effects of actions.

```
(define (domain jug-pouring)
     (:requirements :typing :fluents)
     (:types jug)
     (:functions
          (amount ?j -jug)
          (capacity ?j -jug)
     (:action pour
      :parameters (?jug1 ?jug2 - jug)
      :precondition (>= (- (capacity ?jug2) (amount ?jug2))
                        (amount ?jug1)))
      :effect (and (assign (amount ?jug1) 0)
                   (change (amount ?jug2)
                   (increase (amount ?jug1)(amount ?jug2)))))
)
```

Figure 2.10: Jug-pouring domain description, extracted from PDDL 2.1 document [37]

In PDDL 2.1, levels 1 and 2 have some minor modifications to the original PDDL in order to simplify the parsing and the language. For example, instead of using *change* to assign a numeric value to an object, PDDL 2.1 uses *assign* as direct assignment, and *increase* and *decrease* as relative assignments, which makes the language clearer. The declaration of functions is modified to allow only numeric-valued functions. PDDL2.1 has functions in types of $objects \rightarrow R$, whereas in original PDDL, functions are in types of $objects \rightarrow object$, allowing $object$ to be extended by the application of functions to other objects. With the consideration that numbers do not exist as independent objects in the world but as attributes of objects, numeric expressions are only allowed as arguments to predicates or values to action parameters.

Figure 2.10 illustrates PDDL 2.1 functions in assign change and increase.

Previously, a plan was evaluated based on the number of steps or actions. In PDDL 2.1, plan metrics were first introduced for evaluation purposes. One planning problem may have entirely different optimal plans with different plan metrics. An example of a plan metric in PDDL 2.1:

```
(:metric minimize (+ (* 4 (fuel-consumption car))
                        (fuel-consumption van)))
```

Durative actions are the main introduction of PDDL 2.1 level 3 for the planning competition in 2002. Figure 2.11 illustrates a durative action - the *offload* which has duration of 10 (time units).

```
(:durative-action offload
 :parameters (?t - truck) (?l - location)
             (?b - box) (?c - crane)
 :duration 10
 :condition (and (at start (at ?t ?l))
                 (at start (empty ?c))
                 (at end (in ?b ?t))
                 (over all (at ?t ?l))
 :effect (and (at start (holding ?c ?b))
              (at start (not (in ?b ?t)))
              (at start (holding ?c ?b))
              (at end (not (empty ?c)))
```

Figure 2.11: Offloading a box from a truck

In level 3 of PDDL 2.1, in the description of preconditions and effects, it is explicitly defined whether the corresponding proposition must hold at the *start*, at *end* of the execution, or *over all* (i.e. throughout) the execution. Note that the *over all* annotation is only used for preconditions not effects. With the introduction of time, actions can be performed concurrently. Time is considered as point-based rather than interval-based. In this case, at an interesting timepoint, for example at the start or end of an action, logical states change instantaneously.

Durative actions with discrete effects in PDDL 2.1 can be used to model continuous changes in planning problems.

```
(:durative-action Fly
 :parameters (?a - airplane)
             (?x - location)
             (?y - location)
 :duration (= ?duration (travel-time ?x ?y))
 :condition (and (at start (at ?a ?x))
                 (at start (>= (fuel-level ?a)
                               (*(travel-time ?x ?y)
                                 (consumption-rate ?a))))
 :effect (and (at end (at ?a ?y))
              (at end (decrease (fuel-level ?a)
                      (* (travel-time ?x ?y)
                         (consumption-rate ?a))))
              (at start (not (at ?a ?x)))
```

Figure 2.12: Flying from one place to another

For example, in the domain shown in figure 2.12, the fuel-level changes continuously, but it is modelled by a discrete change at the end of the action of flying. In the example, when the numeric value of the fuel-level of the airplane is updated, no other concurrent actions are allowed to interfere with the value of the fuel-level.

In level 3 of PDDL2.1, numeric variables are modelled in a discretized way. There are some limitations in this approach, such as the unavailability of information about the values of numeric variables during the action execution, and actions are assumed to have effects at the end of their execution without further intervention by the planner. For example, the action of opening a tap to fill a bath should be explicitly terminated after a certain period of time. Otherwise, the bath will be full and overflow. Or, in another example, a truck is called to rescue a trailer out of mud. When moving to the accident place, the truck consumes fuel and partially empties the tank. This may cause the problem that the truck is not heavy enough to rescue the trailer out of mud.

PDDL+ (or level 5) models continuous time. It introduces *processes* which once initiated, run over time. The processes maintain logical aspects of a state while changing numeric values of the state over time. Processes are different from actions in that they do not cause state transitions. *Events* are also introduced in PDDL+ which have numeric pre- and post-conditions. Events are instantaneous state transition functions. They are not used to develop a plan as actions. Numeric post-conditions of processes are always time-dependent, whereas those of events are not. In PDDL+, a three-part structure, called *start-process-stop*, is introduced to model actions, processes and events. The *start* and *stop* can be the application of actions or events or it can be the point at which effects of active processes cause numeric values to reach critical thresholds (e.g. the bath is full, or water is boiling). Figure 2.13 shows a part of the bath-filling domain which contains an action, process and event.

```
(:action turn-on
 :parameters (?tap ?b)
 :precondition (and (off ?tap ?b)
                    (plug-in ?b))
 :effect (and (not (off ?tap ?b))
              (on ?tap ?b)
              (increase (flow ?b) (flow-rate ?tap)))
 :process bath-filling
 :parameters (?b)
 :precondition (<= (level ?b) (capacity ?b))
               (> (flow ?b) 0)
 :effect (increase (level ?b) (* #t (flow ?b)))

(:event flood
 :parameters (?b)
 :precondition  (and ($>=$ (level ?b) (capacity ?b))
                (> (flow ?b) 0)
                (dry-floor ?b))
 :effect (and (wet-floor ?b)
              (not (dry-floor ?b)))
```

Figure 2.13: A part of bath-filling domain, extracted from PDDL+ [35]

In the bath-filling example, the action turn-on will open the tap and the water will flow at the flow-rate. The process bath-filling will increase the level of water in the bath up to the capacity of the bath, while water is still running out of the tap. The event flood is triggered when the level of water in the bath reaches its capacity.

In short, the PDDL2.1 has 5 levels in which only the first 3 levels were used in the 2002 planning competition due to its complexity. The introduction of durative actions, processes and events for continuous behaviours makes planning problems more realistic. The development of planning systems are still far behind in order to handle the complexity of planning domains and problems.

2.4.4 PDDL 2.2

Because the complexity of the first 3 levels of PDDL2.1 is still a challenge to the planning community, PDDL2.2 just extended a bit further for the planning competition in 2004 based on the first 3 levels of PDDL2.1. PDDL2.2 is also divided into 3 levels, which are ADL, numeric, and durational planning. The main extension of PDDL2.2 over PDDL2.1 is the introduction of *derived predicates* and *timed initial literals*.

Derived predicates are predicates which are not affected by the execution of actions. In fact, the truth value of a predicate is evaluated based on a logical statement of the form **if** formula(x) **then** predicate(x). Below is an example of the derived predicate which is extracted from the planning competition 2004 webpage [28] to illustrate the rule of the ”above” predicate in *Blocksworld* problem:

```
if on(x,y) OR (exists z: on(x,z) AND above(z,y)) then above(x,y)
```

Figure 2.14: An example of a derived predicate

In fact, in the original PDDL, derived predicates already existed in the form of *axioms*, but were not used in the first planning competition.

Another extension of PDDL2.2 over PDDL2.1 is *timed initial literals*. They are one of the simple ways to describe exogenous events (i.e. facts that are unaffected by actions of the plan, but will become *true* or *false* at time points which are known by planning systems in advance). An example of a timed initial literal is illustrated below:

```
(:init (at 10 (store-open)) (at 18 (not (store-open))))
```

Exogenous events are very common in the real world. With timed initial literals, planning problems are more realistic.

PDDL2.2, which was used in the planning competition 2004, does not extend much on PDDL2.1. However, the derived predicates help planning systems to be able to express updates on the transitive closure of relations as in figure 2.14. The derived predicates help to identify the infeasibility when compiling them. The introduction of timed initial literals moves the planning community one more step to approach real world problems.

2.4.5 PDDL 3

PDDL3 [40] is the latest development of PDDL. With the ambition to introduce constraints into planning problems, Gerevini and Long extended the PDDL to PDDL3 which allows strong and soft constraints on plan trajectories, as well as strong and soft problem goals to be expressed in the problem.

Recently, planning systems have been compared mainly based on the CPU running time. In PDDL2.1 and PDDL2.2, the quality of plans is also considered. Metrics, such as number of actions in the plan, parallel steps, or more complex computations based on makespan and numerical quantities, are introduced to find a high quality plan. With the trend of setting up criteria or metrics for plan quality, PDDL3 introduced strong and soft constraints into goals and on plan trajectories.

For example, in the Blocksworld domain, possible constraints are that a “fragile” block cannot have any block on the top of it, or a tower of blocks always contains blocks of the same colour.

In PDDL3, trajectory constraints are constraints between actions in the plan, or intermediate states reached by the plan. Goals or trajectory constraints can be either strong or soft. Strong constraints must be obtained or satisfied in the plan, while soft constraints are optional goals and constraints which are desired to be satisfied “as much as possible”.

In PDDL3, in order to compare plan qualities among planners, soft constraints and goals have a weight or preference attached. The weight is a numerical value representing the cost of its violation in a plan. For example, *I prefer the plan which has all cars in their original place, rather than the plan with shorter makespan.* The constraint that each car should be in its original place at the end of the plan is a soft constraint with an attached weight. In this case, if at the end of the planning, any car which is not at the original place will be penalised heavily.

To describe the trajectory constraints , a new flag : $constraints$ is introduced to the syntax of PDDL. Modal operators, which are $sometime, always, at-most-one$, and $atend$, are also added to the syntax. Other operators, such as $sometime - before, sometime - after, always - within$, are introduced in order to avoid some nesting.

Soft constraints have two parts: the identification and how they affect the plan quality if they are not satisfied. The syntax for addressing the preference is described as follows:

```
    (preference [name] <goal-description>)
```

In this case, the goal-description can be extended to include preference expressions. However, those expressions are only conjunctions and universal quantifiers, not nested preferences in connectives. Below are examples extracted from [40]:

```
(preference VisitParis
    (forall (?x - tourist) (sometime (at ?x Paris))))
```

if at least one tourist fails to visit Paris, the above preference will return 1 for (is-violated VisitParis).

Using the same syntax for plan metrics from PDDL2.1, for example, PDDL3.0 might have the metric as below:

```
(:metric minimize (+ (* 10 (fuel-used)) (is-violated VisitParis)))
```

In this example, the objective of *fuel-used* is considered 10 times more important than the violation of visiting Paris.

In conclusion, PDDL3.0 introduced strong and soft constraints to planning domains in order to give planning systems other criteria in evaluating plans. With weights/preferences attached to soft constraints, planning systems are looking for solutions to optimize the plan metric. The new introduction makes planning problems closer to what is happening in the real world. However, the complexity of current PDDL and its extensions are too high for planning systems to deal with.

2.5 An extension of PDDL2.1 level 3 to handle intermediate effects

2.5.1 Introduction

As discussed in the section of PDDL2.1 above, in 2002, Fox and Long introduced time into classical planning. However, the notion of durative action is still quite limited. In the description of PDDL2.1 level 3, since PDDL2.1 is only able to handle actions with start and end conditions and effects, durative actions can be viewed as a combination of start action, process, and end action. Thus, durative actions are not expressive enough to illustrate conditions which hold for a particular period of time, and effects which can take place at anytime during the execution of the action.

In [36], Fox and Long have discussed and argued that PDDL2.1 is able to model durative actions with intermediate effects. It can be done by breaking up those actions into series of smaller actions, in which effects only happen at the beginning or at the end of the execution, and conditions are at the beginning, at the end, or hold over the entire action. Also, in this approach, two special types of actions, called *tightclip* and *looseclip*, were introduced to make sure the series of smaller actions attach to each other to construct the whole durative action. However, this approach makes the planning domain much more complex and cumbersome. Planning systems have to perform extra search and use more memory to store smaller actions.

The idea of supporting intermediate effects was first introduced in IxTeT [64] in 1995. However, at that time, the planning community did not have a common language to describe planning domains and planning problems yet. In this chapter, based on the analysis of the limitation of durative actions in PDDL2.1 from David Smith [91], an extension to PDDL2.1 level 3 is described in the following section to handle temporal planning domains which have actions with intermediate effects.

2.5.2 Syntax and semantics

To illustrate the extension of PDDL2.1 level 3, an example of turning a spacecraft into a particular target is considered. This example was extracted from [91].

In this example, in order to turn the spacecraft to a target, thrusters in the reaction control system (RCS) are fired to provide angular velocity. Then, the thrusters are switched off while the spacecraft coasts until it nearly reaches the target. The thrusters are fired again to stop the rotation. Firing the thrusters consumes resources (i.e. propellant) and changes the status of thrusters to in-use. In addition, during the time of firing the thrusters, it causes some vibration to the spacecraft. Due to those vibrations, certain operations cannot be performed. The duration of firing thrusters are very short comparing to that of the spacecraft coasting.

To model this domain intuitively, the domain definition language must support actions with intermediate effects. In this case, the whole process of turning the spacecraft must be modelled as one action.

In order to introduce intermediate effects into PDDL2.1 level 3, in addition to *at start*, *at end*, *over all*, we add two more temporal notations of the language: (***at*** (timepoint or time_expression) (something)) and (***during*** [start_time end_time] (something)). This proposal is similar to the one proposed by David Smith [91].

With these new notations, intermediate effects can be modelled as follow:

- (at (- endtime RCS−duration) (decrease (propellant) (/ propellant−require 2)))

- (during [(- endtime RCS−duration) endtime] (controller−in−use))

In the above example, in the first case, it tells that at the time (endtime - RCS-duration), the resource is decreased by propellant−require/2 to fire the thruster in order to stop the rotation of the spacecraft. In the second one, it tells that during the time the thruster is being fired from (endtime − RCS−duration) to the endtime of the action execution, the controller is in use.

With the extension of the two new temporal notations, PDDL2.1 level 3 can model temporal domain in which actions can have intermediate effects easier and more intuitively.

In the example above, the intermediate effect (***during*** [...]) can be modelled by 2 ***at*** effects, one at the start of the interval to tell that the controller is in use, and one at the end to release the controller. However, if it is modelled by 2 ***at*** effects, another independent action can intercept and release the controller before the end of the interval. Thus, it depends on the planning domain to use those temporal notations correctly.

Figure 2.15 shows the action turning the spacecraft with the new extension.

In our planning domains with intermediate effects, we assume that there is no independent action which intercepts during the time of the intermediate effect. Therefore, we use two ***at*** effects instead of *during*.

CPPlanner is the first plangraph-based optimal planning system, which can deal with this extension, as shown later.

```
(:durative-action turn
 :parameters (?cur ?dest - target)
 :duration (= ?duration (/ (angle ?cur ?dest) (turning-rate)))
 :condition (and (at start (pointing ?cur))
                 (at start (>= propellant
                            (/ propellant-require 2)))
                 (at start (not (controller-in-use)))
                 (at (- endtime RCS-duration)
                     (>= propellant (/ propellant$-$require 2)))
                 (at (- endtime RCS-duration)
                     (not controller-in-use))
 :effect (and (at start (not (pointing ?cur)))
              (at start (decrease propellant
                            (/ propellant-require 2)))
              (during [starttime (+ starttime RCS-duration)]
                      (controller-in-use))
              (during [starttime (+ starttime RCS-duration)]
                      (vibration))
              (at (- endtime RCS-duration)
                  (decrease propellant (/ propellant-require 2)))
              (during [(- endtime RCS-duration) endtime]
                      (controller-in-use))
              (during [(- endtime RCS-duration) endtime]
                        (vibration))
              (at end (pointing ?dest))
```

Figure 2.15: Turning the spacecraft from the current target to a new target

2.6 Summary

The chapter gives a brief introduction to AI Planning, particularly classical planning and planning with time and resources. Different search algorithms and approaches, such as progressive, regressive search, planning graph, and constraint programming, are being discussed.

In AI Planning, planning under uncertainty makes planning more realistic. However, in the scope of this thesis which concentrates on temporal planning, planning under uncertainty is not discussed.

The Planning Domain Definition Language (PDDL) and its developments are described. Through international planning competitions, PDDL is widely accepted by the planning community as a standard language for domain description. With PDDL, planning community is able to share planning resources (i.e. domains and problems), and compare the performance of planners.

Finally, the extension of PDDL2.1 level 3 to handle planning domains, in which actions have intermediate effects, is discussed. The syntax and semantics of the extension are described. This extension is used in a couple of planning domains in the experimental chapter 6 to show the ability of CPPlanner.

The next chapter discusses in detail different temporal planning systems and their approaches.

Chapter 3

TEMPORAL PLANNING SYSTEMS

3.1 Introduction

Temporal Planning differs from classical planning in that it allows actions to have durations. The introduction of time into actions makes planning domains more realistic. In temporal planning, durations can be static, dynamic, or uncertain: it often depends on the context. For example, the action $Fly(A, B)$, i.e. flying from the location A to the location B, can be dynamically calculated depending on the distance between A and B and the type of airplane in use. In addition, in temporal planning, duration introduces more complexity to the planning domains. For instance, when are the conditions of an action needed? How long are they needed for? When do the effects of an actions take place? In PDDL 2.1 (see Chapter 2), it is assumed that actions have at-start, at-end or overall conditions; at-start or at-end effects.

In the scope of this chapter, actions are considered in form of discretised durative actions. All conditions and effects must be temporally annotated. These temporal annotations will tell explicitly whether a condition must hold at the start of the interval (i.e. the start time when the action applies), at the end of the interval or over the interval from the start to the end (i.e. invariant); or whether an effect is immediate (i.e. it happens at the start time), or delayed (i.e. it happens in the middle of the interval or at the end of the interval). In this thesis, the view of time is point-based rather than interval-based. It means that activities are separated by *timepoints* at which state-changing activities happen. These changes happen instantaneously. Since time is discrete, between any two timepoints, there are only a finite number of happenings (i.e. activities).

There are several approaches to deal with temporal planning domains. The section below reviews different temporal planners with the approach they use.

3.2 Temporal Planning Systems

3.2.1 Graphplan-based temporal planners

3.2.1.1 Introduction

A brief discussion of the Graphplan planner was given in Chapter 2. It is described more fully here because although it is a classical planning system, it has been used as the basis for several temporal planners.

The Graphplan planner [13] was first introduced in 1995 by Blum and Furst. Since then, there have been many variants. The main idea of Graphplan-based planning system is to construct a compact search tree containing all possible states, and to find a solution in that tree. The algorithm consists of two distinct phases: the *graph expansion* and the *solution extraction*. A planning graph is a directed graph containing alternating proposition and action layers. Proposition layers contain proposition nodes which can be true up to that point. Action layers contain action nodes which can happen at that point. Mutexes represent the conflict relationships between nodes. The planning graph is constructed by the graph expansion phase. Starting from the initial state, the planner applies all possible actions to create a new proposition layer with all effects from possible actions. Once the planning graph is constructed to the proposition layer which contains all subgoals which are non-mutex, the solution extraction is called to look for a solution. The solution extraction starts from subgoals and does a backward chaining search to look for a valid plan. If a solution is found, the algorithm stops. Otherwise, the graph expansion phase is called again to advance the planning graph to a next layer.

The most important point of the Graphplan-based approach is that it is sound, complete and produces parallel optimal solutions. Variants of this approach follow that framework, but try to improve the performance of both phases, resulting in a time-space tradeoff.

3.2.1.2 Temporal Graphplan (TGP)

TGP [88] was built by Smith and Weld in 1999. Instead of constructing explicitly a full planning graph with alternating layers, it builds a compact graph which implies the full planning graph. The idea of a compact planning graph is the result of the following observations about the original Graphplan planner:

- Once propositions or actions appear in a layer of the graph, they will appear in all layers after that.
- If a mutex M between propositions P and Q appears at the layer L_k, M appears at all previous layers which have both P and Q. This also applies to actions.
- If a proposition P is not achieved at the layer L_k, it is not achieved in any previous layers.

TGP also introduces types of mutexes: *cmutexes* (or conditional mutexes) are those which disappear eventually; and *emutexes* (or eternal mutexes) are those which do not expire once they have appeared. It introduces a new type of mutex which is an action-proposition mutex: the proposition (i.e. it is an invariant) cannot be true while the action is executing. An action A is cmutex with a proposition P when either P is cmutex with any precondition of A or A is cmutex with all actions which have P as an effect.

TGP runs a mutex reasoning while expanding the planning graph. Figure 3.1 shows the causation diagram structure of the expansion phase of the TGP. For example, adding a new proposition node into the graph might cause new actions which have this proposition as a condition to be added to the graph. Starting from the initial state, the graph expansion phase chooses an action which has the earliest ending time among possible ones to apply. The planner then moves to the next possible timepoint. The graph will be expanded until all

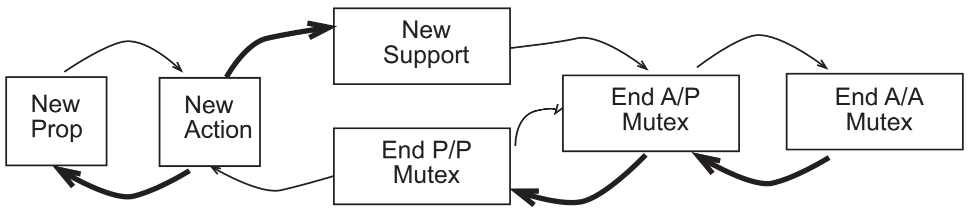

Figure 3.1: The causation diagram of the TGP graph expansion phase. Dark lines mean the effects happen later in time (i.e. after action execution). The figure is extracted from TGP paper [88]

subgoals appear and are pairwise non-mutex. At this time, the solution extraction is called to look for a solution. The solution extraction in TGP uses basic backtracking search. It searches backwards from the subgoals towards the initial state. If a solution is found, the algorithm stops and prints out the result. This is also the parallel optimal solution. Otherwise, the graph expansion is called again to advance the graph to the next possible timepoint.

With the compact planning graph and mutex reasoning, TGP saves space in storing the graph in the memory and the performance for the graph expansion, but increases the complexity of the solution extraction.

In TGP, there are some strict assumptions on the temporal planning domains which it can deal with. Actions only have conditions at the starting time and these conditions must hold during the execution. Actions only have ending effects. The assumptions make it easier in calculating the next possible timepoint and the starting time of next actions while it advances the graph.

In TGP, time is associated with the action layers. From a certain proposition layer, the graph is advanced by choosing an action which ends earliest among other possible actions. This causes the problem to the algorithm in the case that the difference of duration between actions is big. For example, if there is an action A that takes 100 units of time and another action B which only has 1 unit time then the planning graph is advanced gradually for every 1 time unit.

TGP extends the original Graphplan approach to find an optimal solution for temporal planning domains. It showed the best performance comparing to other temporal planners at that time. The compact planning graph reduces the memory consumption while expanding the graph. However, strict assumptions on the temporal planning domains make problems less realistic. The solution extraction uses the basic backtracking search. In addition, the performance of the system becomes worse if the ratio of the greatest common divisor (GCD) of the action durations is small relative to the longest action. In this case, if the final plan must contain the longer action, the planning system wastes a lot of time in choosing the shorter action each time it advances the graph.

3.2.1.3 Temporal Planning System (TPSYS)

TPSYS [38] was developed by Garrido, Fox, and Long. It is an optimal temporal planning system which can handle temporal planning domains with Level 3 of PDDL2.1.

TPSYS applies the idea of Graphplan planner and extends it to handle temporal planning domains. It is quite similar to the approach of TGP. However, there are some differences in the two systems.

Firstly, unlike TGP in which the planning graph is compact, TPSYS builds a whole multi-level planning graph. This leads to the fact that there is much more memory consumption in storing and maintaining the graph. However, it makes the solution extraction easier while looking for a solution. In TGP, the solution extraction search may traverse cycles in the graph, but not in TPSYS.

Secondly, TPSYS analyses and calculates the static mutexes in the preprocessing stage. This helps to speed up other stages later. In TPSYS, the mutexes are introduced in each level of the planning graph as they are in Graphplan planner. With the temporal extension, TPSYS has mutexes for the *end*-part, which are Action$_{end}$-Action$_{end}$, Prop$_{end}$-Action$_{end}$, Prop$_{end}$-Prop$_{end}$, and *start*-part, which are Action$_{start}$-Action$_{start}$, Action$_{end}$-Action$_{start}$, Prop$_{start}$-Action$_{start}$, Prop$_{end}$-Prop$_{start}$, and Prop$_{start}$-Prop$_{start}$.

Thirdly, because TPSYS handles temporal planning problems at the level 3 of PDDL2.1, an action can have effects at the beginning, unlike in TGP. There are cases in which an action executes and has beginning effects. With these effects, the final goals are satisfied before the action actually ends. At the level with these effects, if the goals all appear and are pairwise non-mutex, it calls the solution extraction. However the goals are still conditional because the action has not finished yet. It is impossible to find a solution at this level. Therefore, the authors introduced some propagation about the validity of the propositions in the graph to make sure that propositions are not conditional before doing the solution extraction.

Finally, the solution backtracking search of TPSYS is different from that of TGP or Graphplan planner. It also searches backwards, but differently in moving towards the initial state. In TPSYS, once it selects a proposition, using the backtracking search, it will try to find a path backwards starting from that proposition (i.e. originally it is a subgoal) to the initial state before moving to the next subgoal. It means the system tries to find a full support from the initial state to each subgoal. However, with this approach, the authors could not apply some time bounds to speed up the performance of the backtracking. The introduction of time bounds are discussed in more details in the extension and improvement of CPPlanner in a later chapter.

Like TGP, TPSYS also meets difficulties when the planning problems have actions with big differences in duration. Because the nature of the expansion is to move step by step in time, the planning graph in this case is built very slowly.

To handle the large-scaled problems, TPSYS was developed further [39] with the introduction of heuristic guides. However, with this extension, TPSYS is no longer an optimal temporal planning system.

3.2.1.4 Linear-Programming GraphPlan (LPGP)

LPGP [67] was built by Fox and Long in 2002. It is another Graphplan-based planning system. Like TGP and TPSYS, it extends the Graphplan planner to handle temporal planning problems. However, the extension is very different by attaching time into proposition level, not action.

The graph expansion runs like breadth-first search. From the initial state, it will apply all of the possible actions to have a new action level and new proposition level which is comprised from the effects of the actions. With this approach, it avoids the poor performance when actions have big differences in durations. However, this will sacrifice the optimality of the solution.

In LPGP, each action is divided into a start action, an end action, and invariant-checking actions. In the graph construction, instead of using no-op actions, the invariant-checking actions are used to propagate the propositions between levels. The duration of a level is not fixed. It depends on the difference in time between the two consecutive timepoints of activities (i.e. happenings). Figure 3.2 illustrates the overview of levels in planning graph of LPGP with time attached to proposition level and actions are divided into start actions, end actions, and invariant-checking actions.

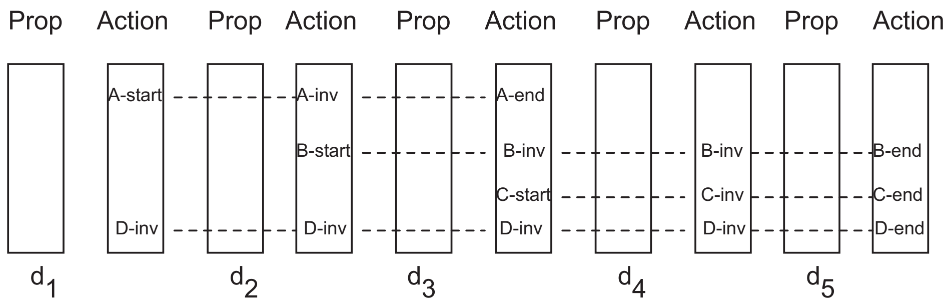

Figure 3.2: The overview of the planning graph of LPGP with duration attached to proposition level and actions are divided into smaller actions. Figure is extracted from the LPGP paper [67]

In the solution extraction phase, if an end action is selected to support a proposition, a temporal constraint is introduced to make sure that the total duration between the corresponding start action and this end action must equal to the duration of the real action. The invariant-checking actions also need to be performed correctly with time. In some cases, if a start action could satisfy a goal (i.e. its effects are in the goals), the corresponding end action must be already in the plan. The mutex relations are used to keep the validity of the plans. In addition, a small non-zero interval is introduced to avoid the interference between an end of an action and a start of the next action.

Like other Graphplan-based planning systems, LPGP also has two phases, the graph expansion and the solution extraction. Instead of attaching time to the action, the introduction of time to proposition levels helps to find plans more quickly. However, this approach will return the plan with fewer number of actions but maybe longer in time (i.e. makespan). The empirical study in [67] shows the better speed performance comparing to TP4, but worse plan quality (i.e. might have fewer number of actions but longer makespan).

3.2.2 Partial Order Planners

3.2.2.1 Introduction

Partial Order Planners (POP) are also known as *Least Commitment Planners*. This approach was introduced in the chapter 2. In that chapter, it is described in general and applied to classical planning. This section describes temporal planning systems which use this approach to deal with temporal planning domains.

The planners work by adding actions that will be needed into a plan which are constrained in the least possible way. For example, if it does not matter what order two actions are executed in, no order is put on them. After the partially ordered plan is built up, flaws are recognised, e.g. if one action prevents another action from being executed or if conditions (or subgoals) are not yet present in the plan. These are then dealt with by adding more actions, ordering the actions (promotion and demotion) or constraining variables to equal or not equal other variables. In 1994, an introduction paper [102] on these planners was written by Weld. POP planners, like Graphplan planners, can produce plans with some concurrency. It means any actions in the partially ordered plan which are not ordered and do not conflict can take place in parallel.

3.2.2.2 parcPLAN

parcPLAN [29] is a domain-dependent planning system which was developed by El-Kholy and Richards in 1996. It used the Constraint Programming approach to deal with planning problems. By the nature of its searching for a plan, it can be classified into Partial Order Planners. parcPLAN can reason on time and resources to search for a plan.

In parcPLAN, a simple temporal problem is constructed by imposing temporal constraints on endpoints of request intervals. The planner uses a simple temporal network to reason on time. Time is discrete and timepoints are represented as variables which take natural values. Propagation on timepoint variables is performed by a path consistency algorithm, which computes the minimal network. Boolean variables are used for each pair of request-intervals to show whether these intervals overlap. This will help to check the global resource constraint. For example, at a certain time, if the 2 request intervals overlap each other, the boolean variable of those 2 intervals is true. The resource consumption at this time will be calculated based on the two requests.

The search of parcPLAN is using meta-variable labelling (i.e. labelling the Boolean variables). In this search, the global resource constraint is checked. The planner first assumes that it has unlimited resources and tries to assign parallel requests. Then, the planner checks to see if the resource constraints are satisfied. If they are, a feasible solution is found. Otherwise, it tries to move a request away from others. The process is repeated until a solution found or all possible assignments are tried.

With this approach, parcPLAN runs well and returns solutions quickly if the planning problems have plenty of resources. However, if the resources are restricted, the planner performs poorly.

3.2.2.3 IxTeT

IxTeT [63] is a planner that could be classified in a number of categories. It produces a partial ordered plan through least commitment planning, but relies heavily on heuristics to guide its search and also allows for macro-operator actions which also puts it in the category of hierarchical task network planners. Rather than representing the world as a set of true literals as in STRIPS, it uses a set of multi-valued state attributes and a set of resource attributes. Each attribute is a mapping from a finite domain of objects to a value. Time is modelled by the predicates *hold* and *event* to represent invariants and change of attributes respectively:

- an assertion $hold$(att(x_1,...):v,(t_1,t_2)) asserts the persistence of the value of state attribute att(x_1,...) to v for each t: t_1 < t < t_2 .
- $event$(att(x_1,...):(v_1,v_2),t) states that an instantaneous change of value of att(x1,...) from v_1 to v_2 occurred at time t.

The predicates *use*, *consume*, and *produce* refer to using reusable resources and consuming and producing consumable resources respectively. These work in a similar way to hold and event. IxTeT uses timepoints, that are seen as *symbolic variables* that represent temporal constraints. Actions are replaced by tasks. Each which may include sub-tasks , a set of events that describe changes brought about to the world (the equivalent of effects with time points), some preconditions and invariants, the resource usage and a set of temporal and instantiation constraints that can bind the task to different time points in the plan. The initial state is a special task that contains the initial attributes as a set of explained events, a set of extraneous events (also as a set of explained effects), the initial resource level and the goals that must be achieved as a set of assertions.

The search of the partial plan starts with this initial state being a partial plan with flaws. Branches represent some tasks or constraints inserted on the current plan to solve a flaw. There are three types of flaws:

- Unexplained propositions are temporal propositions ($hold$ or $event$) have not been satisfied. These can be resolved by adding a causal-link (an assertion that protects the state attribute value from the establisher to the subgoal). The establisher could already be in the plan (simple establishment) or may need to be added (task insertion).

- Threats are events or assertion that threaten an assertion already in place. It can be resolved by ordering it by promotion or demotion (i.e. adding a temporal constraint) or by adding a variable constraint (setting two variables to equal or not equal one another).

- Resource conflicts are when tasks compete for the same resource or there is an insufficient amount. Again, these can be resolved by inequality constraints or addition of tasks to produce the resource.

There are two choice points (or branching places) in the search space; one is the selection of the flaw. The other is the selection of the resolver. This is done by a least commitment strategy, i.e. trying to keep options open (and so prune the search space to a minimum at that point). Flaws are selected opportunistically, i.e. the flaw is selected that maximises the easiness to make a choice between its resolvers. The number of flaws may be big, but any flaw which is resolved will only create new flaws at the same or a lower level of abstraction (i.e. in the sub tasks of a task). This hierarchy can be automatically generated from the domain and is not fixed but is dynamically ordered whilst planning. The search is controlled by a “near-admissible” heuristic.

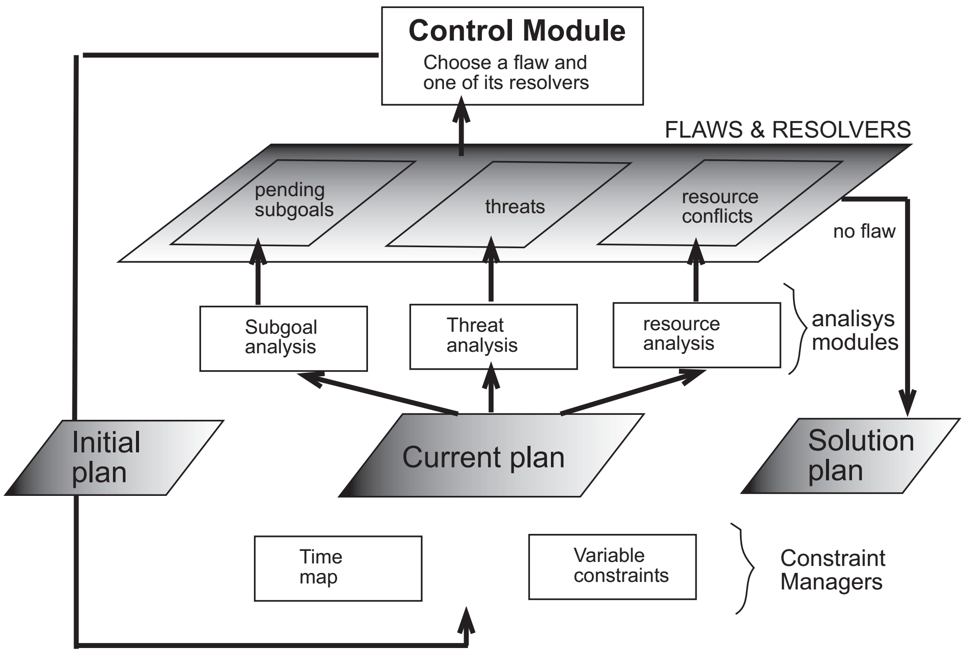

Figure 3.3: The IxTeT Planner Architecture, from [63]

IxTeT is not used as an optimal planner, but it can be made optimal by using the task deadlines. As the model of time is discrete, these deadlines can be decreased by unit time until no plan is found. The last valid plan found is optimal.

3.2.2.4 Zeno

ZENO [84, 83] is another least commitment planner that can deal with temporal planning domains including goals with deadlines. The actions have conditions and effects, but also have two compulsory labels, one for the start time and one for the end time. They can also have constraints between these labels. The schemas make planning domains that Zeno can handle are more expressive than that of PDDL2.1, as the schema specifies exactly when conditions, constraints and effects are true and even allow time intervals.

Because conditions and effects are attached with time, a planning problem can be encoded as a partial plan with a single dummy step. For example:

Action **Dummy**

at-time: $[t_0, t_1]$

condition: at (t_1, Peter, London) $\wedge$ at (t_1, Mary, London)

constraints: $t_0 < t_1 \leq t_0 + 2.5$

effect:

at (t_0, Peter, Manchester) $\wedge$ at (t_0, Mary, Liverpool)

$\wedge$ at (t_0, Linda, Liverpool) $\wedge$ fuel(t_0, car)=200

Figure 3.4: Dummy action

Initial condition, external events and domain axioms are the effect of this dummy action. Deadline goals and final goals are conditions, and the time of the dummy action is the desired time span for the plan to complete.

The algorithm that Zeno employs is a least commitment strategy with regressive search. It first checks to see if the plan is consistent (i.e. no constraints are broken). If not it fails and will have to backtrack. But, if the plan is consistent, it checks to see if there are any outstanding goals. If not then it returns the plan, but if so, it then chooses a goal. If this goal is complex it will reduce it to primitive goals and starts again. If it is primitive and metric it will add it to the constraints and start again. Else, it will add an supporting action and resolve the threats on it and restart the loop. There are three non-deterministic decisions to backtrack. The first one is to decompose goals into simpler formulae. This happens if the goal is a disjunction. The goal is replaced by one of its disjuncts. If the goal was a conjunction, then all the literals are added to the unresolved goal list. Selecting a time can be a problem as it is not discrete. Zeno does one of two things, it either splits the time interval in two and explores each one separately, or it marks that time interval as indivisible. This avoids infinite branching as it will only split time intervals to a preset depth. The second decision to be made is to decide which supporting action to use. Codesignation (e.g. x = y or x $\neq$ y) and primitive metric constraints (e.g. v1 $\leq$ v2) are posted to the constraint reasoning system which determines if it is consistent. The third decision is where constraints are introduced to prevent interference between actions and goals.

Zeno relies on constraint satisfaction for all temporal and metric reasoning. For codesignation a simple algorithm is used to maintain equivalence classes of all logical variables, then determines whether the noncodesignations are inconsistent with the classification. A Simplex algorithm handles the metric reasoning, whilst temporal relations are solved with Warshalls transitive closure algorithm. The Simplex algorithm assigns the greatest possible values to variables whilst not breaking constraints.

Zeno is a sound and complete planning system. However, because the time reasoning takes a lot of time in querying and updating the temporal cache, it is quite slow in performance. The system could not generate plans with more than a dozen actions. Since then, there has been no further development on the system.

3.2.2.5 Constraint Programming Temporal planner (CPT)

CPT [98, 99] was developed by Vidal and Geffner in 2004 to participate in the International Planning Competition 2004 (IPC2004). It is a least commitment planner based on Constraint Programming to find an optimal solution.

CPT uses a lower bound on the makespan to prune irrelevant partial order plans if they exceed the lower bound. The heuristic function used to calculate the lower bound is nearly the same the one using in [48, 49]. In partial order causal link planning, planners perform by picking up flaws (i.e. open preconditions or threats), and then try to find supports or solutions to those flaws. CPT adapted the branching to temporal planning by introducing temporal variable *T(a)* which is the starting time of an action *a* of the partial order plan. The temporal constraints were introduced, such as:

- If an action *a* has a precondition *p* which needs support, and *a'* is a supporting action, the casual link *a'[p]a* is created and a temporal constraint $T(a') + dur(a') \leq T(a)$.

- If an action a in the partial plan deletes a proposition p in the causal link $a_1[p]a_2$, a temporal constraint *T(a) + dur(a)* $\leq$ *T(a_1)* or *T(a_2) + dur(a_2)* $\leq$ *T(a)*.

The temporal constraints are then formed a Simple Temporal Problem (STP) [20] which is checked by using the *bounds consistency* introduced by Lhomme in 1993 [65].

Unlike other constraint-based partial order planners which only reason on actions in the current partial plan, CPT can reason on actions which are not in the partial plan yet. CPT introduced variables for all actions in the planning domains, *S(p,a)* for undetermined supporting action for precondition p of a, and its undetermined starting time *T(p, a)*.

The formulation of CPT has four parts, which are preprocessing, variables, constraints, and branching. In the preprocessing, CPT applies the heuristic function introduced by Haslum and Geffner [48, 49] to compute the heuristic value for each action a. In CPT, all branching decisions are binary and each branching node is a "partial plan". The novel idea of CPT which made CPT better than other Constraint-based planning systems is the ability to reason on action a, which is not yet in the plan.

CPT is developed on Choco Contraint Programming library. It is an optimal temporal planning system. The strict assumptions on actions in CPT are very similar to those of TGP (i.e. actions are not allowed to overlap each other if they are conflict), which is stricter than that of PDDL 2.1. CPT performed very well in the optimal track of the IPC2004. It was ranked 2nd in the competition. However, because SATPLAN [55], the 1st ranked planner, was only able to deal with classical planning domains, CPT was considered as the best optimal planning system for temporal planning domain.

In 2006, CPT was updated and participated in the 5th Planning Competition (called CPT2 [100, 2]). In this competition, it won the distinguished prize for brilliant performance in the optimal track over other competitors.

CPT will also be used for comparison in the experiment chapter to show the competency of our planning system.

3.2.3 Heuristic Planners

3.2.3.1 Introduction

Heuristic planners try to efficiently search the state space by looking at the most promising branches first by weighting search branches. They use heuristic functions to guide the search to the solution. Planners use admissible search algorithms, such as A* to branch the search tree. Calculating good heuristic functions can be computationally expensive. There is often a tradeoff between the complexity of the heuristic function with the accuracy of the guess.

3.2.3.2 Sapa

Sapa [23], which was developed by Do and Kambhampati, is a planner that applies a heuristic approach for temporal and resource planning. *Sapa* search is performed through a set of time-stamped states, represented by a tuple S = (P,M,Π,Q,t) in which:

- P is a set of predicates which are true at the time t and when they are last achieved.
- M is the set of values of all functions representing the metric resources.
- Π the set of invariants which must remain true for a period of time.
- Q is a set of updates in which each is scheduled to take place at a certain time in the future.
- t is the timestamp.

These states do not only describe the state of the world now, but the state of the planners search as well. All goal have deadlines. A goal is in a state if it is present in P and was achieved before its deadline, or if there is an event in Q that will achieve the goal before the deadline. The branching of the space comes from actions that can be applied to a state.

An action can be applied if all preconditions are satisfied by P and M, the effects do not interfere with anything in Π or Q and no event interferes with invariants of the action. Besides normal actions, *Sapa* introduces advance-time actions which are used to advance the time of the state S to the time point of the earliest event in the queue Q. *Sapa* uses a simple A* to guide the search.

Sapa applied the idea of Graphplan planner to build a relaxed planning graph from the initial state to the goal state to construct the heuristic function. The relaxed graph is built by just ignoring delete effects and resource constraints. With the relaxed temporal planning graph, the planner can prune irrelevant states which do not lead to the solution, use the time in which the subgoals appear as a lower bound for the search, and estimate the distance from the current state to the goal state.

Sapa is a domain-independent planner, which can handle durative actions, metric resource constraints, and deadline goals. It is designed to find a multi-objective solution.

Sapa can handle both time and resources. With the heuristic approach, the planner can return feasible solutions in an acceptable running time. It is suitable to deal with large-sized planning problems, which optimal solutions could not be found in a limited of time.

3.2.3.3 TP4

TP4 [49] was built in 2001 by Haslum and Geffner. It is a development of HSP planning approach. It searches for plans backwards from the goal state. It can deal with both time and resource, including consumable resource and renewable resource. In classical planning, plan tails (i.e. partial plans that achieve the goal if the conditions of the partial plan are satisfied) are built from the goal state. When such a tail is concatenated with a plan head, the end result is a valid plan. A plan tail can be summarised by a set of literals and it is this set of literals that forms a state in the search space. However, in temporal planning, a set of literals is insufficient to summarize the plan tail as it holds no information about the actions in the potential plan that have started before this time point but not finished. Thus, a state is now defined as a pair of sets (E,F) in which E is a set of atoms and F is the set of actions attached with time increments $(a_1,\delta_1),,(a_n,\delta_n)$. A plan P achieves $s = (E, F)$ at time t if P makes all the literals in E true at t and schedules the actions ai at the time $t - \delta_i$. The initial sate is (G_P,$\emptyset$) where G_P is the goal set of literals and the final state are all (E,$\emptyset$) where E is a subset of the initial facts.

The branching is done by selecting an action that has effects in E. In TP4, the heuristic function estimates how close the current state to the initial state. TP4 uses IDA* search algorithm. Like many other temporal planning systems, TP4 also tries to shift actions as late as possible without changing the plan structure.

TP4 also deals with resources by including the current value of each resource in the search state. Only renewable resources affect the branching rule although the heuristic only remains informative with unary capacity resources. With consumable resources the heuristic remains admissible but becomes less useful as they do allow for over consumption. However, if resources are limited to be monotonically decreasing (i.e. consumed but not produced) then some states can be pruned if actions consume more resource than the remaining availability. A new heuristic is presented that minimises the amount of resource consumed. By integrating the resource and temporal heuristic, some combination of optimising the duration and consumption can be achieved. TP4 also has the power to have no-ops that consume resources. These are called maintenance actions and could be used to model continuous effects.

TP4 uses IDA* and some other enhancements to guide the heuristic search. It is an optimal planning system. It can deal with time and resource to find a minimal makespan. In the empirical study [49], it shows that TP4 produces the solution similar to that of TGP, but slower.

In 2004, TP4 was re-implemented to be a more flexible experimental platform for variations of the basic planning algorithms [46]. It was also added some other enhancements to improve the performance of the search. The main difference of the new version and the old version of TP4 is described as follows:

Firstly, the new version of TP4 extended the resource finding procedure of the old version to correctly identify resources in some planning domains in the International Planning Competition 2004 (IPC2004) (e.g. the *umts* domain). However, the resource representation which is used in TP4 is less expressive than that of PDDL 2.1.

Secondly, TP4 are improved with some new tricks in the search. The first improvement is the "irrelevant detection". The new TP4 uses the standard reverse unreachability to detect and remove irrelevant propositions and actions. With this detection and removal, the performance is improved (e.g. in the *airport* domain). The second improvement is the introduction of two-stage optimization because the algorithm IDA* used by TP4 performs badly if the GCD (greatest common divisor) of actions' durations is small. In the 2-stage optimization, action durations are first rounded up to the nearest integer to improve the GCD. Then, TP4 will solve the problem with the new action durations. The result of this (i.e. makespan) is used as an upper bound for a branch-and-bound search to find the optimal solution with the real action durations. However, in the IPC2004, the new 2-stage optimization only helped to improve the performance in the *satellite* domain (see [47]).

In the new version of TP4, it is still an optimal planning system which can deal with time and resources. The resource finding procedure was extended to handle resource better in IPC2004 domains. The new search improvements were introduced but the performance was not improve much comparing to the old version.

3.3 Summary

The chapter shows an overview of different planners which deal with temporal planning domains. Originally, AI planning only deals with instantaneous actions. In 2002, temporal planning was first introduced in PDDL and officially used in the third International Planning Competition. Temporal planning is a big step towards the ambition of dealing with real-world problems. The chapter describes and groups different planners according to their approach, including plangraph, partial order, constraint programming, heuristics.

It focuses on the two plangraph-based planning systems, TGP and TPSYS, which use the same approach as our planning system. In general, all planning systems described in this chapter consider time as an integrated part rather than separating it as a sub-problem to deal with later. TGP, TPSYS, parcPLAN, CPT, and TP4 can give optimal solutions to temporal planning domains, while the other planners, LPGP, IxTeT, ZENO, and Sapa only give feasible solutions. IxTeT is not specifically designed to find an optimal solution, but it can modified slightly to become an optimal planner. The next chapter will describe our plangraph-based planning system, called CPPlanner, to deal with temporal domains.

Chapter 4

CPPLANNER - AN OPTIMAL TEMPORAL PLANNING SYSTEM USING CRITICAL PATHS

4.1 Introduction

Recently, the plangraph framework [13] has been applied and developed in many planning systems. It has been extended and modified to deal with more complex planning domains. This chapter continues in this vein, introducing a further extension of the plangraph framework to deal with more complex temporal planning domains. A new planning system, called CPPlanner, is described in detail. With the new extension, CPPlanner can find an optimal solution in terms of time (*makespan*) for temporal planning domains in which actions can have effects at any time during their execution. The main contribution of CPPlanner is the usage of a *critical path* in the planning graph as a backbone for the backtracking search while looking for an optimal solution.

The next section will introduce the representation of an action with the capability of handling actions with effects occurring during their execution (intermediate effects). The mutex relations between proposition-proposition, action-proposition, and action-action are also presented in detail to show the constraints between nodes in the planning graph. Finally, the graph expansion and the solution extraction algorithms are described and discussed thoroughly.

In addition, the operation of the algorithms is demonstrated via a small worked example prior to the detailed empirical study in the next chapter.

4.1.1 Towards the new extension of the plangraph framework

The previous chapters introduced and discussed the two current plangraph-based temporal planning systems, TGP and TPSYS. They extend the plangraph framework to find optimal solutions for temporal planning domains. However, in TGP, it is assumed that an action's conditions must hold at the beginning and until the action finishes. Effects of actions are undefined during the actions' execution and are only guaranteed to hold at the end of the execution. In TPSYS, the assumption is less strict, in that it allows actions to have beginning effects (i.e. effects that hold from the start of the action's execution). As in plangraph-based planning systems, the mutex relations are used to indicate whether actions and propositions can happen simultaneously. Because of the assumptions about the conditions and effects in both TGP and TPSYS, mutex relations are still relatively simple.

With the ambition to deal with real world problems, CPPlanner has been extended to have a broader assumption on effects of an action. An action can have effects at any time during its execution. With this assumption, two actions now can overlap *partially* if two actions have no mutex relation and a intermediate effect of one action is one of the conditions of another action. The graph expansion is then extended to handle such actions. In addition, due to the partial overlap of actions which depends on the starting time of actions, the mutex relation checking in the solution extraction is more complicated than that of TGP or TPSYS. The details of these extensions and techniques are discussed in the following sections.

4.2 Action representation

In CPPlanner, the action representation has been designed to deal with temporal planning domains in which actions can have effects at any time during their execution. Therefore, the representation of an action must show that the starting time of each effect happens while the action is executing. The action representation of CPPlanner is mainly influenced by the PDDL+ [32] and the representation of actions in the Sapa planner [23].

The action representation for CPPlanner is described as follows:

An action A is presented as $\{$Dur$_A$, Cond$_A$, Eff$_A\}$ in which:

- Dur$_A$: the duration of the action A. (Dur$_A > 0$).
- Cond$_A$ = $\{\langle$cond$_{A_1}$, type$_{A_1}\rangle$, ..., $\langle$cond$_{A_k}$, type$_{A_k}\rangle\}$.
- Eff$_A$ = $\{\langle$eff$_{A_1}$, $\delta_{A_1}\rangle$, ..., $\langle$eff$_{A_h}$, $\delta_{A_h}\rangle\}$ with $\forall i \in [1, h] : 0 \leq \delta_{A_i} \leq Dur_A$.

In this representation, each action A has a duration Dur$_A$, a starting time Start$_A$, a list of conditions Cond$_A$, and a list of effects Eff$_A$. The duration Dur$_A$ can be statically defined in the domain or dynamically calculated at the run time of the action. For example, the action *Fly* can be calculated based on the locations of the take-off airport, the landing airport, and the speed of the aircraft. For actions, it is assumed that the conditions are required at the beginning and need to be held throughout the execution of an action. Each Cond$_A$ is a list of propositions $\{\langle$cond$_{A_1}$, type$_{A_1}\rangle$, ..., $\langle$cond$_{A_k}$, type$_{A_k}\rangle\}$ which are required for the action to execute. Each condition cond$_{A_i}$ attaches with a type type$_{A_1}$ which is *at_start*, *at_end*, or *over_all* to describe that the condition needs to be held at the start, at the end or during the execution of the action.

Since CPPlanner has richer action representation in which actions can have intermediate effects, the list of effects Eff$_A$ for each action is a conjunction of tuples $\langle$eff$_{A_j}$, $\delta_{A_j}\rangle$ (i.e. $\langle$ proposition, difference-in-time $\rangle$). Each tuple represents that the effect eff$_{A_j}$ will happen at the time (Start$_A$ + δ_{A_j}) onwards.

With the representation shown above, an action A can have intermediate effects starting at any time during its execution. For example, in figure 4.1, the action A starts at time 10 with Cond$_A$ = $\{\langle$cond$_1$, $at\_start\rangle$, $\langle$cond$_2$, $at\_start\rangle\}$, and the Eff$_A$ = $\{\langle$eff$_1$, 5$\rangle$, $\langle$eff$_2$, 20 $\rangle$, $\langle$eff$_3$, 20 $\rangle\}$. In this example, eff$_1$ is an intermediate effect with δ_{A_1} = 5 (i.e. after 5 units of time from the starting of the action, the effect eff$_1$ takes place).

Allowing intermediate effects for an action leads to more complexity in expanding the planning graph and checking mutex relations in the solution extraction phase becomes more

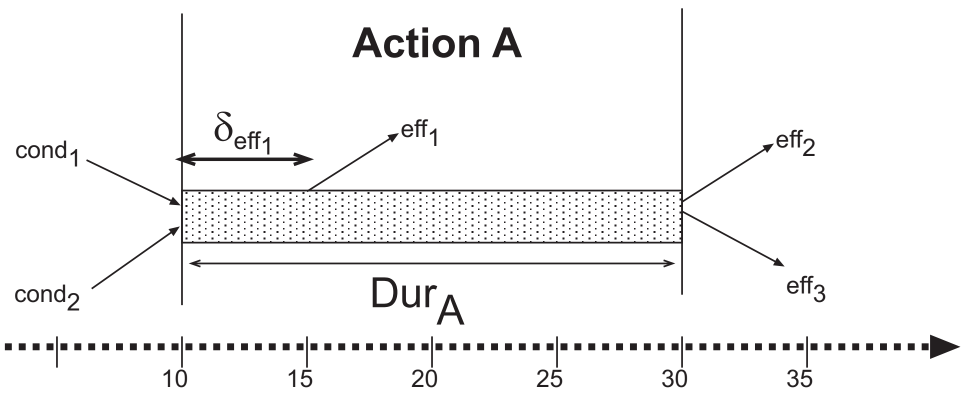

Figure 4.1: Action representation

difficult. In the graph expansion phase, because actions can have intermediate effects, two actions can overlap partially with each other. The calculation for the starting time of the next possible action is more complicated. The problem will be discussed in more detail in the next sections.

4.3 The planning graph

In the plangraph approach, the planning graph is a multiple-level graph which alternates between a level of propositions and a level of actions.

The graph is constructed from the initial state and advanced level by level from the left. The first level of the graph consists of propositions in the initial state. The next level is the level of actions which can apply, given these propositions. These actions are also called *possible actions*. Then, the effects of these actions form a new level of the graph. The propositions in the previous level, which do not appear again in the list of the effects, are also added to this level via no-op actions.

In temporal planning, because each action has its duration, each action or proposition has an attached *timestamp* showing the starting time of that action or proposition. Thus, in temporal planning, each node in the planning graph contains a timestamp showing the

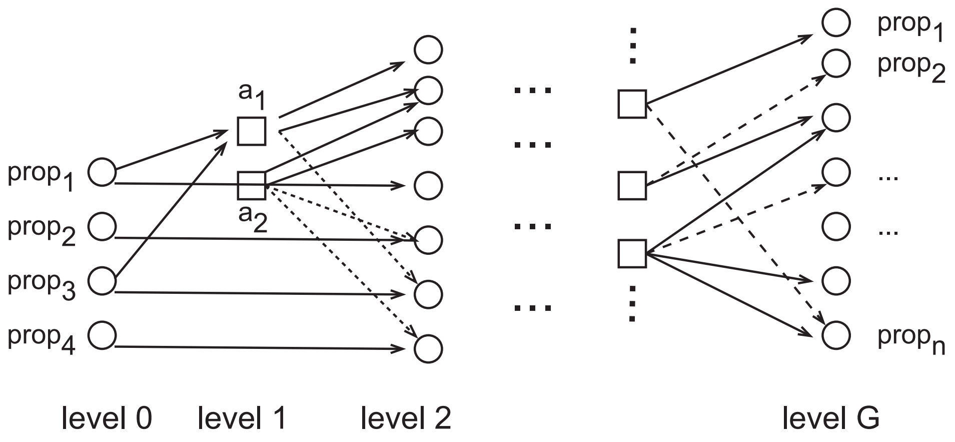

Figure 4.2: The multiple-level planning graph of the plangraph approach

earliest possible time it takes place.

Note that in plangraph approach, once a proposition appears in the graph, it will appear again in all subsequent levels of propositions. As in TGP, in order to save memory consumption, we also use the *bi-level compact graph* to represent the full planning graph. The bi-level graph is a graph containing a level of actions and a level of propositions. There are directed edges connecting between nodes in the two levels showing the links of conditions and effects for actions. Also, there are edges showing the mutex relations among them in the graph.

In TPSYS, because the planner uses the usual multiple-level planning graph, the expansion phase is simpler and more straightforward. In each level of the graph, all the nodes of the previous level appear again. This causes a problem with the memory consumption when the size of planning domain is big. In TGP, because of the strict assumption on conditions and effects, effects only become true at the end of the execution of actions. New actions, which need these effects as conditions, will use the current examining time *t* as the starting time. However, in CPPlanner, because an action may have intermediate effects, a new action might use one of these intermediate effects as a condition. Instead of using the

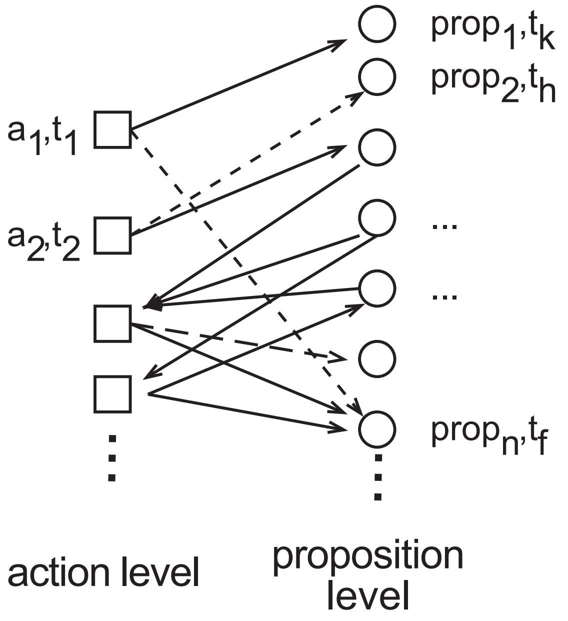

Figure 4.3: The Bi-level planning graph

current examining time *t* as starting time, it has to check all the timestamps of the conditions to find out the starting time for the action. This problem will be discussed in detail in section 4.4.

4.4 Graph Expansion

4.4.1 The concept

In the plangraph approach, the algorithm consists of two phases: the *graph expansion* and the *solution extraction*. The graph expansion builds up the planning graph, and the solution extraction tries to find an actual solution in the graph.

In the original plangraph approach, starting from the propositions in the initial state (called level 0), the planner applies all possible actions to these propositions. The possible actions are stored in the next level (i.e. level 1). The effects (=propositions) of these actions

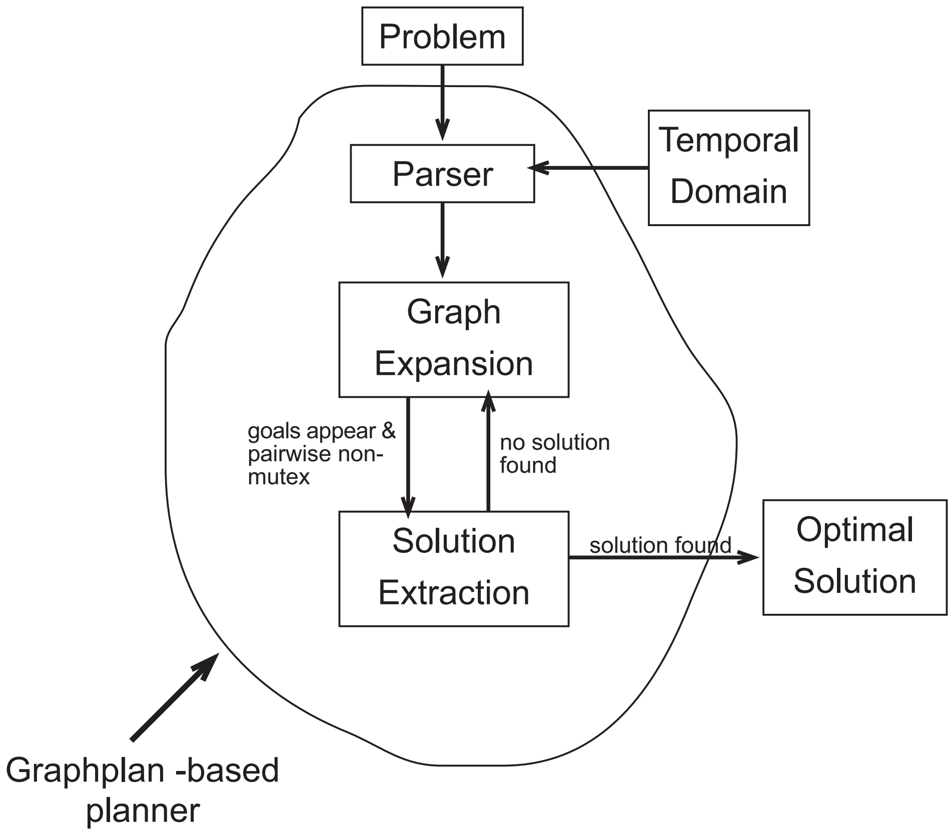

Figure 4.4: The model of a plangraph-based planning system

form a new level of propositions (level 3). Other propositions in the previous level (=level 0) which do not appear in this level are also being added (via no-op actions). This process is repeated until all the propositions of the goal state appear in the graph. At this time, the solution extraction is called to perform a backward-chaining search to look for an actual solution. If the solution is found, it is the optimal one. Otherwise, the graph expansion is called again to advance the graph to the next level, and the process repeats. The planning graph of the plangraph approach is shown in 4.2.

In the Graphplan planner, because it is dealing with the classical planning domains in which actions have no duration, the planner can apply all possible actions at each proposition level to create a new level in the graph. However, CPPlanner is dealing with temporal planning domains. The planner will advance the graph step by step in time. In order to do this, from a certain time with propositions given, the planner will choose an action from possible ones which finishes earliest (note that although this is not specially mentioned in the papers of TGP and TPSYS, it is believed that this also applies in those systems).

In temporal planning, the planning graph is constructed based on the idea of advancing from the intial state step by step in time until all the *subgoals* (propositions in the goal state) appear. However, in TGP and TPSYS, because actions do not have intermediate effects, they cannot overlap partially. The calculation for the new actions is straightforward based on the current examining time t (i.e. the ending time of the previous chosen action). However, in CPPlanner, actions can overlap partially. For example, while an action A is executing, another action B, which uses the intermediate effects of actions A as conditions, starts. The calculation for the starting time of action B needs to be based on the timestamps of the intermediate effects rather than the current examining time t. The details of the calculation are shown in the algorithm description section below.

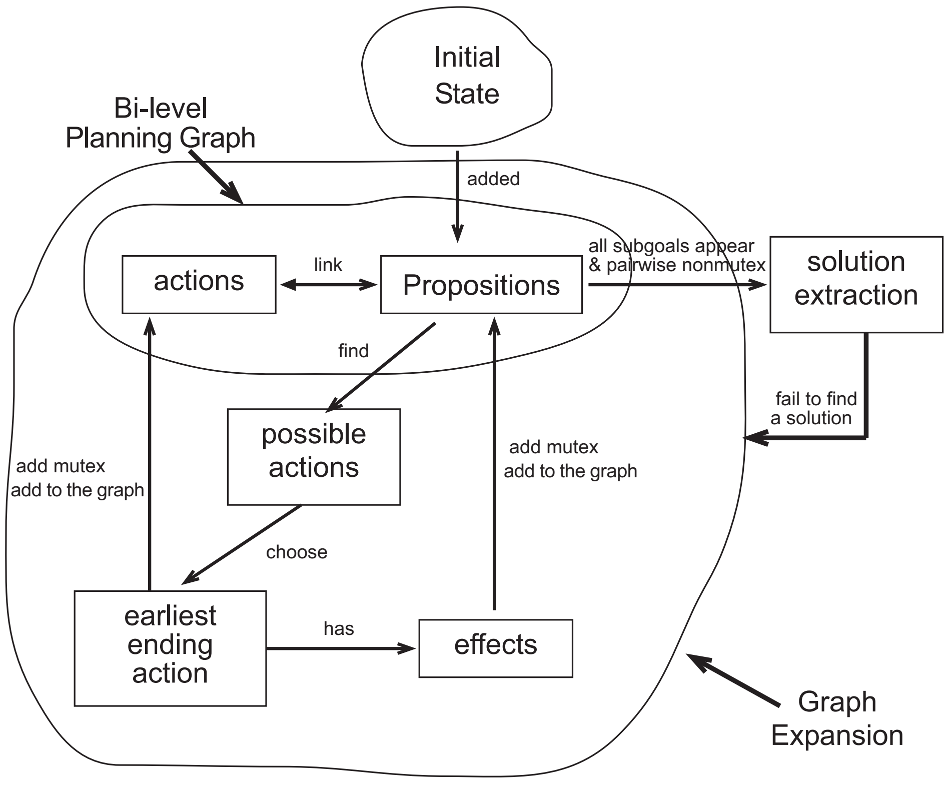

Figure 4.5: The Graph Expansion Diagram

4.4.2 Algorithm description

In CPPlanner, as in TGP and TPSYS, at the start, the planning graph has all propositions of the initial state with *timestamps* 0. The graph is advanced in time by choosing an action which can use current propositions as conditions and has the earliest ending time. If there are several actions which have the same ending time, all of them will be added. Other possible actions which can use these current propositions as conditions, but have longer ending time are added to the list of temporary actions *TmpActions* [1] with their timestamps. This temporary list *TmpActions* is maintained in order to calculate the next possible timepoint and avoid the redundancy in calculating possible actions. Also, the new possible actions are only calculated based on the updated effects of the new added actions. With this temporary list *TmpActions*, the next possible action is the next one in the list (Note: It is possible to have more than one action which end at the next possible timepoint. In this case, all of them are added).

After applying this earliest ending action, all of its effects with timestamps, which are calculated based on the starting time of the action, are added to the queue list *PropsQueue* to wait to be added to the graph. The *PropsQueue* is constructed and maintained in order to calculate the starting time for next actions. This is the main difference with TGP and TPSYS, and more complicated than the calculation for the starting time of new actions in those systems.

The propositions in *PropsQueue* are sorted in chronological order. Each of them will be chosen and added to the planning graph gradually. At the time of adding a proposition, say $prop_x$, from *PropsQueue* into the graph, the planner also checks and adds any new action which uses this proposition $prop_x$ and other propositions in the *Props* (i.e. propositions in the planning graph) as conditions with its timestamp to the temporary list *TmpActions*. The timestamp of a new action, which is added to the temporary list, is the *biggest timestamp*

[1] TGP has an equivalent to the *TmpAction* list. This is not evident in the papers describing the system, but can be seen in the code.

of all supporting propositions of that action (Note: sometimes if the new added proposition *prop*$_x$ is an intermediate effect of an earlier added action, it might have a smaller timestamp comparing to other supporting propositions).

When adding a proposition from *PropsQueue* in to *Props* (i.e. the planning graph), if the proposition appears again, the planner just needs to store the new timestamp for this proposition in the planning graph as well as the old timestamp. This queue of propositions maintains all possible new propositions which are waiting to be added to the graph. In this queue, it is possible that the same proposition appears many times with different timestamps. However, once it is added to the graph, only the *latest timestamp* is stored.

The process is repeated until all the propositions in the goal state (subgoals) appear and are pairwise non-mutex. At this time, solution extraction (see 4.6) is tried to look for a solution. If a solution is found, it is also the optimal solution for the problem and the algorithm is terminated. Otherwise, the process will be performed again to move to the next possible timepoint.

Figure 4.6 presents the details of the graph expansion algorithm.

```
01. Procedure GraphExpansion()
02.   //All initial props with timestamp 0 in the Queue
03.   PropsQueue = $\{< p_1, 0 >, < p_2, 0 >, ..., < p_n, 0 >\}$
04.   // Props - the propositions added to the graph
05.   Props = $\{\}$
06.   // Actions - the actions added to the graph
07.   Actions = $\{\}$
08.   // The examining timepoint starts at 0
09.   t = 0
10.   Loop
11.     Call ApplyingPossibleActions(Props, PropsQueue, TmpActions, TmpProps)
12.     // Move to the next possible timepoint where at least one action completes
13.     t = ending timestamp of the next action in the TmpActions
14.     // add the completed actions to the Graph
15.     Actions $\leftarrow$ Actions $\cup$ $\{$new completed actions after moving to the next timepoint$\}$
16.     Remove these actions in TmpActions
17.     PropsQueue $\leftarrow$ PropsQueue $+$ new props in TmpProps from completed actions at t
18.     Remove these propositions in TmpProps
19.     If goals $\subseteq$ $\{$Props$\cup$PropsQueue$\}$ & pairwise nonmutex
20.         then call IncludingAllRemainingActionsProps() and do solution extraction.
21.     If (solution extraction succeeds)
22.         then terminate the algorithm.
23.   End $\{$loop$\}$
24. End Procedure
```

Figure 4.6: Pseudo-code for GraphExpansion().

```
01. Procedure ApplyingPossibleActions(Props, PropsQueue, TmpActions, TmpProps)
02.   while $PropsQueue \neq \emptyset$
03.     CurProp = the first prop in the PropsQueue
04.     Remove it from PropsQueue
05.     Create its mutex relations with Props and Actions
06.     PossibleActions = {all actions having CurProp as one
07.       of their conditions, and the others from Props}
08.     $TmpActions \leftarrow TmpActions \cup PossibleActions$
09.     $TmpProps \leftarrow TmpProps \cup$ {effects of PossibleActions with timestamps}
10.     // Add CurProp to the graph.
11.     // If CurProp is already in the graph, compare the timestamps and keep the bigger.
12.     $Props \leftarrow Props \cup CurProp$;
13.   end {while}
```

Figure 4.7: Pseudo-code for `ApplyingPossibleActions()`.

In TGP, it is simple that the starting time of a new applied action is the current examining time, because of the strict assumptions on conditions and effects of an action. However, because CPPlanner allows actions to have intermediate effects, they can overlap partially. The starting time of the possible action is not simply the current examining time *t*, but is calculated based on the timestamp of the new added proposition from the *PropsQueue* and the other supporting propositions. In the case that if the new added proposition is an intermediate effect of the current action, the new possible action will start before the current action finishes.

For example, an action *X* may take some intermediate effects of action *Y* as its conditions and start while action *Y* is still under way. In this case, the starting time of *X* is the time when all conditions are true. This time is the biggest timestamp among supporting propositions, including some intermediate effects of *Y*. Sometimes, if the new chosen proposition of the *PropsQueue* to be added to the graph is already in the graph, only the timestamps will be compared, and the bigger one will be kept as the timestamp for that proposition. In some cases, if the new chosen proposition has a smaller timestamp (i.e. that proposition has been generated by another action which issued a bigger timestamp earlier), the graph remains the same.

4.5 Mutex relations

In the original Graphplan, the mutex relations are introduced to show the mutual exclusion between actions or propositions. Since the expansion of the planning graph applies all possible actions to given propositions, actions which conflict with one another are still applied to advance the graph to the next level. Thus, the mutex relations are introduced in order that, when the solution extraction phase is looking for a solution, it will not let mutually exclusive actions or propositions happen simultaneously. In temporal planning, actions have duration. Hence, the mutex relations are temporal constraints. They depend on the time in which an action takes place or a proposition becomes true.

As in TGP and TPSYS, CPPlanner has two kinds of mutex relations: *static mutex* and *dynamic mutex* (Note that TGP and TPSYS use different terms, but they have nearly the same meaning). The static ones will show that actions or propositions are mutex with each other regardless of the time. If an action A is *smutex* with an action B, this mutex relation is always there whenever these actions take place. However, the dynamic mutex relations are temporal constraints (or logical functions based on time) to show whether actions or propositions conflict with each other or not, depending on the time that action takes places or the proposition becomes true. In the solution extraction phase, because the timestamps are changed when being added to the *plan*, these constraints are checked again with these actual new timestamps.

The mutex relations of CPPlanner are described as follows:

- **Proposition - proposition:** propositions p and q are mutex if (1) they are negations of each other or (2) all actions supporting q are logically inconsistent with p and vice versa.

In (1), the mutex is static and remains the same, regardless of the time. This type of mutex is calculated once in the expansion phase, and can be used again in the solution extraction when looking for a plan.

In (2), the mutex is dynamic; whether p and q are mutex depends on the starting times of the supporting actions. For example, in the spacecraft domain, the vibration effect only appears when the thrusters are in use. This vibration lasts for a small amount of time during the whole turning process of the spacecraft. Due to the vibration, some operations cannot be performed (e.g. using a camera) because of the mutex relation of the propositions of those propositions with the vibration. The mutex relation needs to be re-checked once the actual timestamps for actions and propositions are attached in the solution extraction phase.

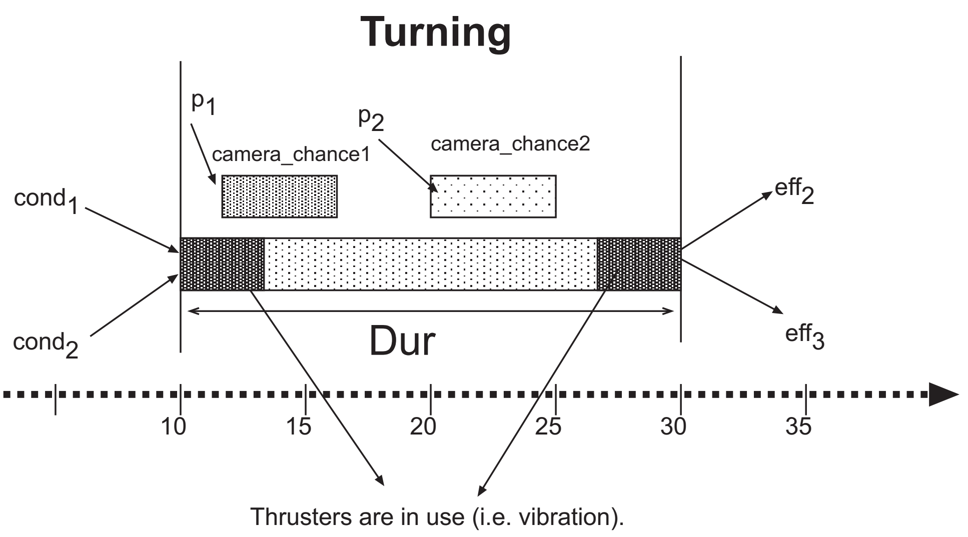

Figure 4.8: Action **turning** of the spacecraft domain. The vibration effects prevent action **camera_chance1** but not **camera_chance2.**

Figure 4.8 illustrates the action **turning** of the spacecraft domain. In this description, due to the vibration effect when the thrusters are in use, there is a mutex conflict between the condition p_1 and the vibration. Thus, action **camera_chance1** could not perform. However, action **camera_chance2** can perform because there is no conflict between p_2 and the vibration effect.

- **Action - proposition:** action *A* and proposition *p* are mutex if one of these holds
  - (q $\in$ Cond$_A$) $\wedge$ (p and q are logically inconsistent).

    If one of the conditions of the action *A* is mutex with the proposition *p*, the action *A* will be mutex with the proposition. In this case, it depends on the mutex relation between *p* and *q*. If the relation p-q is static, the mutex A-p is also static. Otherwise, it is dynamic.
  - (q $\in$ Eff$_A$) $\wedge$ (p and q are logically inconsistent) $\wedge$ (p is true at the time the effect q also becomes true).

    This is a dynamic mutex relation. It depends on the starting time of the action *A* to define the starting time that the effect *q* becomes true.
- **Action - action:** action *A*, *B* are mutex if one of these holds
  - (p and q are logically inconsistent) $\wedge$ (p $\in$ Cond$_A$) $\wedge$ (q $\in$ Cond$_B$)
  - (p and q are logically inconsistent) $\wedge$ (p $\in$ Cond$_A$) $\wedge$ (q $\in$ Eff$_B$) $\wedge$ (p is true at the time the effect q becomes true).
  - (p and q are logically inconsistent) $\wedge$ (p $\in$ Eff$_A$)$\wedge$ (q $\in$ Eff$_B$) $\wedge$ (p and q are true at the certain timepoint *t*).

4.6 Solution Extraction

4.6.1 The concept

When the graph expansion phase has built the planning graph to a time t_G in which all subgoals appear and are pairwise non-mutex, the solution extraction phase is called to look for a solution (or extract a *plan*). In a plangraph-based planning system, the solution extraction uses exhaustive backtracking search to find a solution in the planning graph. All the subgoals are first added to the plan. The planner builds and maintains a list of propositions which needs supporting, called *Goals*. The list *Goals* starts with all the subgoals. The planner will find an action which has at least one of the propositions in the list *Goals* as its effect. If the action does not conflict with other added actions and propositions in the plan, it will be added to the plan. The effects of the new added action will be removed from the list *Goals*, and conditions of the actions will be added to the list and the plan. At this time, the planner looks again to find a new supporting action for at least one of the propositions in the new list. This process is repeated moving backward in the planning graph, until the initial state is a subset of the propositions in the plan. If the process fails to find a new supporting action, it will backtrack by removing the last actions added with its conditions from the plan and trying another supporting action. If a solution is found (i.e. the initial state is a subset of the list of propositions of the plan), it is also the optimal solution and the algorithm is terminated. However, if the extraction phase cannot find a solution at this time t_G, the graph expansion phase is called again to advance the graph to the next timepoint.

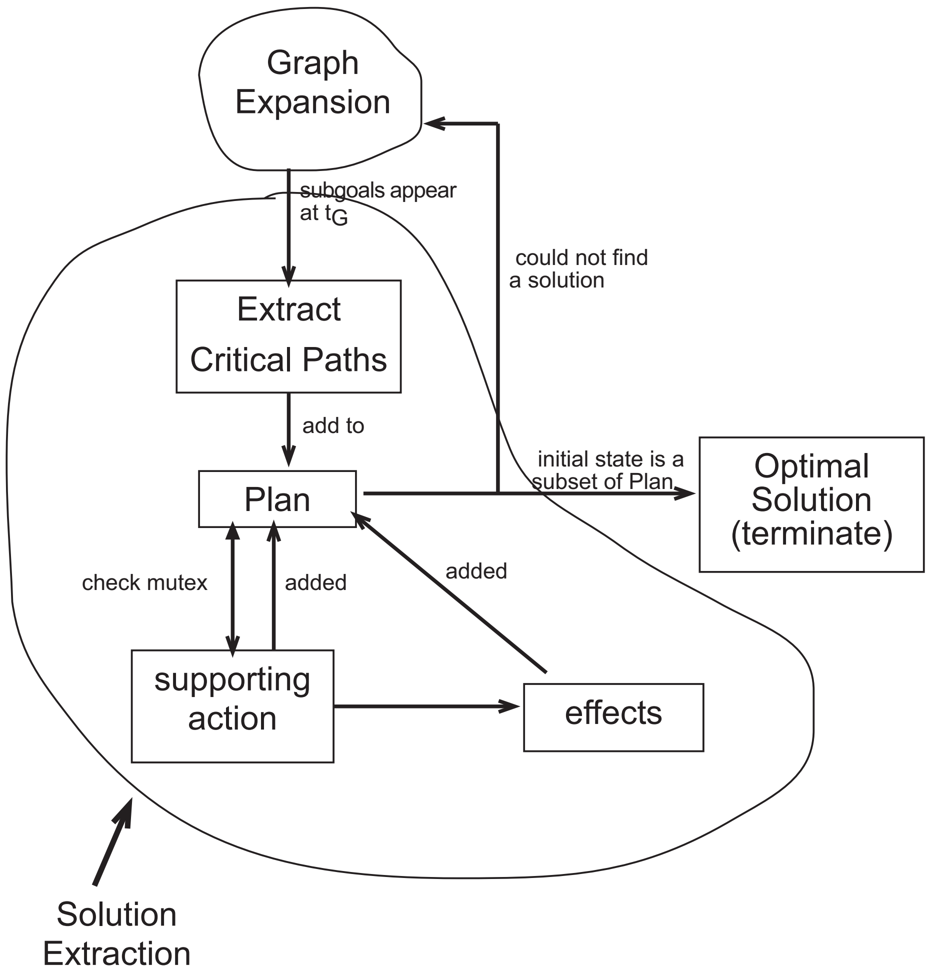

Figure 4.9: The diagram of the solution extraction

Normally, in these plangraph-based planning systems, the graph expansion phase only takes a small proportion of the whole running time to build the planning graph. Thus, the performance of these systems mainly depends on the speed of the solution extraction phase. In TGP and TPSYS, the search for solution is basic backtracking. They select a supporting action for current propositions in the goal state and check to see if it is mutex with other chosen actions and propositions already in the plan. If it is not mutex, the action will be added to the plan, and its conditions are also added to the plan and to the *Goals*. The search process continues until the propositions of the initial state are in the plan.

In CPPlanner, in order to improve the performance of solution extraction, the idea of *critical paths* is introduced, which will add actions and propositions to the plan earlier. A critical path is a proposition-action path starting from propositions in the initial state and ending at a subgoal with timestamp t_G in the goal state. There must be at least one such path, because the graph expansion phase has advanced to the time t_G.

The propositions and actions along the selected critical path are added to the plan at the start of solution extraction to form a *backbone* for the plan. This helps to prune irrelevant branches, since propositions and actions that are mutex with the propositions and actions of the critical path cannot be added to the plan. There can be more than one critical path, however, if two (or more) actions both have ending effects at time t_G. In such a case, if the solution fails to find a solution, the planner chooses another critical path to try. Note that if two actions have ending effects at t_G, both may need to be part of any plan, *or* they may be independent action in the sense that a plan containing either action might exist. To be safe, two separate critical paths are constructed in such cases.

In addition, instead of using only the basic backtracking search, each time CPPlanner chooses an action to the plan, the action is attached with a timestamp smaller than or equal to that of the earlier chosen action. This process helps CPPlanner to build up a timebound for timestamps of later actions added to the plan. It avoids the redundancy for the backtracking search.

```
01. Procedure FindProp(tmpAction, curPath, listOfCandidatePaths)
02.   // decisiveProps are propositions effecting the timestamp of the action
03.   For each tmpProp in tmpAction.decisiveProps do
04.     FindAllPossiblePaths(tmpProp, curPath, listOfCandidatePaths)
05.   Od {end for}
```

Figure 4.10: Pseudo-code to get a decisive proposition for an action.

4.6.2 Critical Path extraction

When the solution extraction is called, it means that all the subgoals appear and are pair wise non-mutex at the time t_G. In order to advance to this timepoint t_G, there is at least an action which ends at t_G. These actions make the difference to the planning graph comparing to the last time. Hence, if there is a solution at this stage, the plan (i.e. the solution) must contain at least one of these actions. In addition, because the solution (i.e. plan) is a subset of the whole planning graph, if the solution exists at this stage, one of the proposition-action paths which leads to the subgoal with timestamp t_G must be a part of the plan. However, because there might be a case that there are more than one subgoal with timestamp t_G or more than one such proposition-action paths. CPPlanner will trace back to find all those possible paths and consider them as critical path candidates. Then, the planner will select them one by one as a actual critical path as act as the backbone for the solution extraction search.

Figure 4.12 shows the pseudo-code of the extraction for the critical path candidates.

```
Procedure FindAllPossiblePaths(curProp, curPath, listOfCandidatePaths)
  // add curProp to the curPath
   curPath = curProp $\bigoplus$ curPath
  If curProp is in the initial state then
    // A new candidate found and added to the list
    listOfCandidatePaths = listOfCandidatePaths + curPath
  Fi {end if }
  else begin
    // decisiveActions are actions effecting the timestamp of the curProp
    For each tmpAction in curProp.decisiveActions do
      curPath = tmpAction $\bigoplus$ curPath
      FindProp(tmpAction, curPath, listOfCandidatePaths)
    Od {end for}
  end else
```

Figure 4.11: Pseudo-code to get all possible paths for a certain subgoal.

```
Procedure CPCANDExtraction(PropsGoal)
  listOfCandidatePaths = $\emptyset$
  For each curProp in PropsGoal do
    If curProp.timestamp is $t_G$ then
      curPath = {}
      FindAllPossiblePaths(curProp, curPath, listOfCandidatePaths)
    Fi {end if}
  Od {end for}
  return listOfCandidatePaths
```

Figure 4.12: Pseudo-code for `CriticalPath candidates extraction` ().

In figure 4.10 and 4.11, the planner tries to extract all possible critical candidate paths if there are more than one. In some cases, an action in the critical path candidates might have more than one decisive propositions (i.e. propositions decide the timestamp of this action). In those cases, because the planner is not sure which one is the main proposition supporting this action, each of decisive propositions will create a new critical path candidate. However, other supporting propositions which are not the decisive ones are not considered to be traced at this stage. Those propositions will be considered while the planner is doing the solution extraction backtracking search. It is because that if the planner traces based on those propositions, supporting actions of those propositions will be attached with timestamps which might not be the actual timestamps for them. If the actions are selected and fixed with those timestamps, it will prevent other actions and propositions being added to the plan because of the mutex conflicts. This will cause the incompleteness for the solution extraction search.

The critical path candidates are then used one by one as the backbone for the solution extraction search. The chosen actions and propositions of the critical path will prune some search branches earlier by preventing new actions and propositions being added to the plan because of the mutex conflicts. The details of the solution extraction are described in the section below.

4.6.3 The algorithm description

The solution extraction attempts to extract a *plan* in the current planning graph for the timepoint t_G. At first, the solution extraction calls *CPCANDExtraction()* to retrieve all critical path candidates. The solution extraction will choose one of them to act as the backbone for the backtracking search. If it fails, the planner will try another one from the candidates. If the planner cannot find any solution after trying all critical paths, it will call the graph expansion again to advance the graph to the next possible timepoint.

In figure 4.13, in order to illustrate the critical path in the graph, the full level planning graph is used instead of the the bi-level graph. In the graph, the subgoals are prop$_1$, prop$_h$, prop$_k$, and prop$_n$. However, only prop$_h$, and prop$_n$ have timestamps t$_G$. The planner will extract critical paths leading to these two propositions. In figure 4.13, the critical paths are prop$_1$-a$_1$-...-a$_q$-prop$_h$, and prop$_4$-a$_3$-...-a$_f$-prop$_n$. These paths will be added to the plan and act as backbones for the backtracking search.

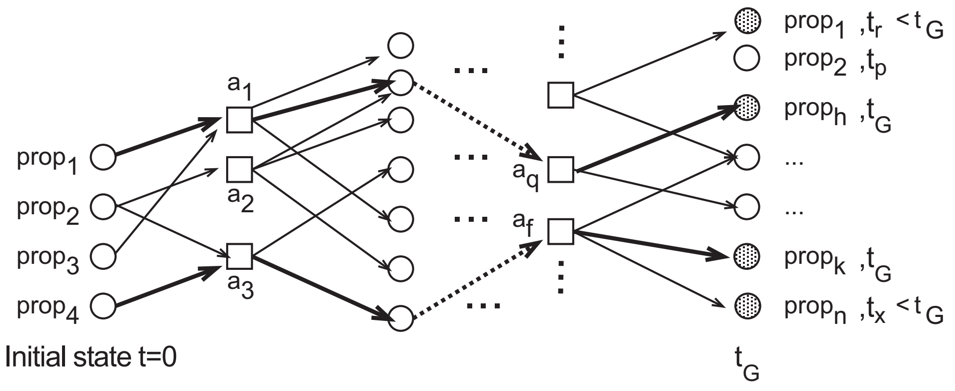

Figure 4.13: The planning graph after the expansion phase. Note that for simplicity the full level graph is used for the illustration purpose

When a candidate is selected to act as the backbone, all the actions and propositions along this path will be added to the plan before the planner actually starts the backtracking search. With the chosen actions and propositions, other actions or propositions will be eliminated in the search later on if they are mutex with those chosen. If the solution extraction fails to find a plan based on the selected critical path, another is chosen to try. Figure 4.14 presents the details of the solution extraction algorithm.

```
01. Procedure SolutionExtraction()
02.   // Attach the goal propositions with timestamp $t_G$
03.   Goals = $\{< p_1, t_G >, < p_2, t_G >, ..., < p_n, t_G >\}$
04.   if Goals is a subset of initial state then stop the algorithm and print the solution.
05.   Set $t$ = $t_G$
06.   Candidates = CPCANDExtraction(PropsGoal)
07.   while Candidates $\neq \emptyset$
08.     CriticalPath $\leftarrow$ get one from Candidates
09.     Delete it from Candidates
10.     ActionPlan = all actions of the critical path with their timestamps
11.     Remove propositions supported by the critical path from Goals
12.     Add all conditions of actions in the critical path to Goals with their timestamps
        (including the non-decisive propositions)
13.     // Note: excluding the conditions that are effects of other actions in the critical path
14.     while can choose (NextAction = one of the actions which supports p
            such that p$\in$Goals, and doesn't mutex with ActionPlan and Goals).
15.         Slide NextAction as late as possible, but its ending time not exceeding t.
16.         Add NextAction into the ActionPlan.
17.         Delete its effects in the Goals.
18.         Add its conditions to the Goals with timestamps.
19.         if the timestamp of any propositions $< 0$ then fail and try another NextAction
20.         if Goals is a subset of initial state then print the solution and terminate the algorithm.
21.         Update the time t = ending time of the chosen action
22.     end {while}
23.   end {while}
24.   if cannot find a solution then graph expansion phase is run again by the Loop
```

Figure 4.14: Pseudo-code for `SolutionExtraction ()`.

As shown in line 6 of the figure 4.14, the critical path candidates are retrieved before the search starts. These candidates are taken one by one to act as the backbone for the search. The actions and propositions in the critical path are added to the plan. At this stage, the proposition list of the plan has subgoals and propositions from the critical path. The planner will find a next action which supports one or more propositions in the list. If it does not conflict with the chosen ones, that action will be added to the plan. The effects of this action which are also in the proposition list are removed from the list (i.e they are supported by the newly chosen action). Conditions of this action are added to the list in order to find supporting actions. The process is repeated until the planner cannot find any action which can support the propositions in the list; or the list is a subset of the initial state. In the latter case, a solution is found and is optimal. In the former case, the search backtracks, and tries another critical path candidate. If the planner cannot find a solution based on any critical path, the graph expansion is called again to move to the next possible timepoint.

In this solution extraction, actions are chosen in reverse chronological order. When each action is chosen to add to the plan, it is attached with a real timestamp. The earlier in the process an action is chosen, the bigger its timestamp is. This helps to avoid redundant search. Actions and propositions of the critical paths are added to the plan early to help pruning irrelevant search branches. Therefore, the performance of the exhaustive backtracking search is significantly improved (see chapter 6 for details).

4.7 Summary

This chapter has presented the algorithm of CPPlanner. The algorithm has two phases: graph expansion and solution extraction. The graph expansion constructs the planning graph until all of the subgoals appear and are pairwise non-mutex. The solution extraction then tries to extract a plan from that planning graph. If a solution is found, it is the optimal one. Otherwise, the graph expansion is called again to advance the graph to the next possible timepoint. The pseudo-codes of the two phases are provided in detail.

With the critical path extracted from the planning graph before looking for a solution, the conditions and effects of the critical path are added early into the plan. The backtracking search will eliminate any mutex-conflict actions or propositions with the chosen ones. This helps to prune irrelevant branches early in the search tree. In addition, choosing actions in a chronological order helps to avoid the redundant search. Hence, the performance of the planning system is improved significantly.

The following chapter introduces some extensions and improvements on the solution extraction phase. Some CSP techniques are also applied to speed up the search. The empirical section after that will show the better performance of CPPlanner over TPG and TPSYS in most of testing domains.

Chapter 5

IMPROVEMENTS

5.1 Introduction

In all plangraph-based planning systems, the performance depends mainly on the solution extraction phase. Therefore, the improvement of the solution extraction phase will speed up the whole system significantly. In this chapter, some improvements to the backtracking process of the solution extraction phase are introduced to avoid redundant search and prune irrelevant branches.

5.2 Avoid redundant solution extraction calls

In the expansion phase of the plangraph-based planning systems, whenever the graph is expanded to the next timepoint, the system will check to see if all the subgoals (i.e. propositions) of the goal state appear and are pairwise non-mutex. If it happens, the solution extraction will be called to perform the backtracking search to find a solution from that graph. In each expansion step, a new possible action with the earliest ending time is chosen and applied from all possible actions. As mentioned in [13], the result (i.e. plan) is a subgraph of the plangraph. It is analysed and discussed in chapter 4 that the solution must contain at least one of the *critical paths* as the backbone (i.e. a part of the subgraph result). If the latest expansion step does not affect any subgoal (i.e. all the timestamps of the subgoal remain the same), the critical paths are the same as the previous expansion step. In this case, the planner does not need to run the solution extraction phase to look for the subgraph result.

Therefore, to apply this improvement, in the graph expansion phase, we only need to check whether the latest advance in timeline affects any timestamp of the subgoals of the goal state. If it does, the solution extract is called. Otherwise, the call is ignored and the graph expansion tries to advance the graph to the next possible timepoint.

5.3 Conflict Directed Backjumping

5.3.1 Motivation

In chapter 4, the solution extraction phase uses exhaustive backtracking search. The backtracking is very basic except the improvement in timebounds while looking for the next action. This section will review the inefficiency in the backtracking search when the planner fails to find out a supporting action and comes back to the previous chosen one to backtrack.

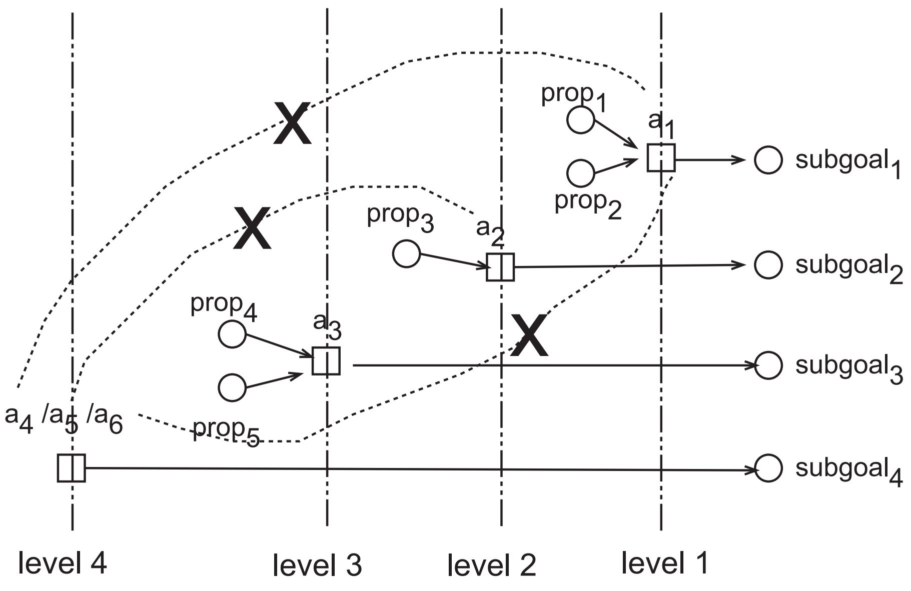

Figure 5.1: An example to illustrate the inefficiency in the backtracking of the solution extraction phase

We shall have a look at figure 5.1 and analyse to find out the inefficiency of the basic backtracking search of the algorithm introduced in chapter 4. First of all, we have 4 subgoals 1, 2, 3, and 4 of the goal state which need to be supported. Assuming that we will find supporting actions for them in that order, subgoal$_1$ needs to be checked first. Assuming that action a$_1$ is the only action which supports subgoal$_1$, a$_1$ is chosen for level *1* (Note that in solution extraction phase, the search is backward from the goal state. We name each action found in each level from *1* to *k* in which level *1* contains the first action which supports the first subgoal of the goal state). Proposition prop$_1$ and prop$_2$ which are conditions of action a$_1$ are added to the list of subgoals. However, because they have timestamps smaller than those of the original subgoals, they are checked later after the origial subgoals are checked. For the sake of simplicity, we assume that a$_2$ and a$_3$ are actions which support subgoal$_2$ and subgoal$_3$ respectively. At a result, other propositions prop$_3$, prop$_4$, and prop$_5$ are added to the subgoal list. We now have to look for actions which supports subgoal$_4$. In this example, we assume that subgoal$_4$ can be supported by any of the 3 actions a$_4$, a$_5$, and a$_6$. However, a$_4$ and a$_6$ are mutex with a$_1$, and a$_5$ is mutex with a$_2$. In this case, after trying all possible actions, due to mutex relation conflicts, the planner fails to find an action which supports subgoal$_4$. It has to backtrack and undoes choices. It comes back to the previous chosen one, which is subgoal$_3$ and choose a different search branch from there. Assume that there are a few actions which can support subgoal$_3$, it can choose the next one. However, whenever the planner chooses a supporting action for subgoal$_3$, and moves on to subgoal$_4$, it always fails to find a supporting action for subgoal$_4$. The reason is that the problem of choosing a supporting action for subgoal$_4$ is not at the subgoal$_3$, but at subgoal$_1$ and subgoal$_2$. The planner wasted a lot of time trying different actions at the level *3* where action a$_3$ is. Therefore, if at each level the planner stores information about the source of conflicts, it can easily jump back to the level where the problem is and try to choose another search branch. The next section will describe this method to avoid redundant search and jump back to the source of conflicts.

5.3.2 Improving the search with Conflict-Directed Backjumping

As introduced and analysed in [85] [53], the conflict-directed backjumping is an efficient method to avoid getting into irrelevant branches and jump back to the conflict source. In our extraction algorithm, because actions are chosen one by one supporting the current subgoal propositions, we can store the conflict list for the k^{th} action in the level *k*. In particular, when we are looking for the k^{th} action in the extraction phase, if action a_i is a supporting action for the current subgoal (i.e. proposition), but it is mutex with one of the earlier chosen action, says a_x. Assuming that a_x is in the level *h*. In this case, *h* is stored in the conflict list of level *k*. In the end, if the planner tries all possible actions and fails to choose one, it can rely on the conflict list to jump back to the closest earlier level to sort out the problem. If we have a look again at our example above, the level *4* will have a conflict list which contains *1* and *2*. The planner will be directed to jump back to level *2* because it is closer than level *1*. This improvement will help the planner to avoid getting stuck into irrelevant search branches.

5.4 Summary

This chapter analysed some inefficiencies of the basic backtracking described in chapter 4. The two improvements, avoiding redundant solution extraction calls and conflict-directed backjumping, are discussed and illustrated in detail. The improvements help CPPlanner avoid irrelevant search branches to speed up the performance. Chapter 6 shows the efficiency of applying those improvements in our planner via the empirical study.

Chapter 6

EXPERIMENTS AND EMPIRICAL ANALYSIS

6.1 Introduction

This chapter describes the experiments and empirical analysis of CPPlanner, our planning system. CPPlanner has been developed based on the algorithms and improvements described in chapter 4 and 5. This chapter also shows the comparison to an earlier version of CPPlanner, called CPPlanner_Basic, which has the same graph expansion phase but the solution extraction only uses the basic backtracking without "critical paths" and does not include the improvements in chapter 5. CPPlanner is compared directly to the two best optimal plangraph-based temporal planning systems, TGP and TPSYS, which were described in detail in chapter 3.

The next section describes the domains used in the comparisons. Then, CPPlanner is compared to CPPlanner_Basic, TGP and TPSYS. Because TGP cannot handle PDDL2.1 domain specifications, a planning domain defined within the TGP package was used in the comparisons. To compare with TPSYS, temporal domains from the 3rd International Planning Competition in 2002 were used. Result tables are also included to show the direct comparison. In each section, domain descriptions are given, and discussions and analysis are made on the results. Besides TGP and TPSYS, CPPlanner is tested on a temporal planning domain - the *Airport* domain extracted from the 4th Planning Competition in 2004, and the *Storage* domain extracted from the latest planning competition 2006 (Note that: because of the different syntax in the domain description, TGP and TPSYS could not deal with these domains). In these domains of IPC2004 and IPC2006, CPPlanner is compared to the state-of-the-art planning system, CPT [98] , which ranked 2nd in the optimal track of the competition 2004 (Note: the 1st ranking planner could not handle temporal domains), and won the distinguished prize in the competition 2006. Finally, CPPlanner was tested on the new domain, *Spacecraft*, in which actions can have intermediate effects.

The research focuses on finding an optimal solution for temporal planning domains. In the experimental evaluation, most of the test cases have been extracted from the temporal domain section of the international planning competitions, or from the package of TGP. In addition, temporal domain *Spacecraft*, in which actions can have intermediate effects, was created for the evaluation to show the ability to handle richer temporal domains of CPPlanner. Large test cases of the international planning competitions which are mainly built for heuristic planning systems are not included in this experimental evaluation. Since all the planning systems in the evaluation are optimal planners, the *makespans* of the result plan are the same. Hence, the main criteria used in comparison are the CPU time and the number of problems solved.

6.2 Temporal planning domains used in the experiments

This section illustrates all the temporal planning domains used in the experiments. Firstly, the temporal *logistic* domain provided in the package of TGP was used to have a clearer comparison with TGP.

Because our CPPlanner can handle the durative actions in PDDL2.1 level 3 specifications, most of the experimental domains are extracted from the Third International Planning Competition in 2002 (IPC2002). In that competition, when durative actions were first introduced to the planning community (see chapter 2 for details), most of the planning domains contained temporal definitions. In addition, the temporal planning domains in the planning competition 2004 (IPC2004) and the latest 2006 (IPC2006) have also been extracted for the experiments in this chapter. Finally, two planning domains which were specifically created to contain actions with intermediate effects were used to test the performance of CPPlanner.

6.2.1 Temporal domain logistics *extracted from the TGP package*

Since TGP was developed before the introduction of PDDL2.1, it cannot handle PDDL2.1 Level 3 specifications. In addition, TGP assumes that all the conditions of an action must hold before executing the action and remain undefined during the execution. However, the domain description that TGP can handle is quite similar to that of PDDL2.1 Level 3, except it only contains fixed times for durations, all conditions are at-start and over-all (i.e. required at start and holding throughout the execution), and effects happen only at the end of the execution. In the domain *logistics*, packets need to be delivered to different locations using trucks or airplanes. Below is the description of actions available in the domain *logistics* extracted from the TGP package. In this domain, durations of actions are static (i.e. the duration is a fixed number).

Actions:

- ***load***: at location z, load packet x to vehicle y. The duration is 1 time unit.
- ***unload***: at location z, unload packet x to vehicle y. The duration is 1 time unit.
- ***drive***: in city c, drive truck x from location y to location z. The duration is 2 time units.
- ***drive-inter-city***: drive truck x from location y to location z in a different city. The duration is 11 time units.
- ***fly***: airplane x flies from airport y to airport z. The duration is 3 time units.

In this domain, there are totally 6 planning problems, named Log_1 to Log_6. Table 6.1 shows the numbers of objects in each planning problem of the *Logistics* domain. The problem Log_1 is the biggest and most complex comparing to the others. Although Log_2 and Log_3 have fewer number of objects than those of Log_1. Log_4, Log_5, and Log_6 have much fewer number of objects and simpler initial and goal state.

Problem	**cities**	**offices**	**airports**	**trucks**	**airplanes**	**packets**
Log_1	3	3	3	3	1	3
Log_2	3	2	3	2	1	3
Log_3	3	2	3	1	1	2
Log_4	3	0	2	0	1	1
Log_5	2	0	2	1	0	1
Log_6	3	0	2	0	1	1

Table 6.1: The numbers of objects in each problem of the *Logistics* domain extracted from the TGP package.

6.2.2 Temporal domains extracted from IPC2002

6.2.2.1 Depots

This domain was constructed for IPC2002 by combining the two well-known domains, *logistics* and *blocks*. The domain describes trucks moving crates from one place to another. Crates are loaded into and unloaded from trucks by hoists. Durations of actions depend on the locations of objects or the weight of crates. Crates are then stacked onto pallets at the destination. The details of the domain are as follows:

Types: the domain uses the types: *place, locatable, object, depot, distributor, truck, hoist, surface, pallet,* and *crate*. A *pallet* and a *crate* are of type *surface*. A *truck*, a *hoist*, and a *surface* are of type *locatable*. A *depot* and a *distributor* are of type *place*. A *place* and a *locatable* are of type *object*.

Actions:

- ***Drive***: describes the driving of a truck from one place to another. The duration of the action depends on the speed of the truck and the distance between the two places.
- ***Lift***: describes that a hoist lifts up a crate from a surface. The duration of this action

is 1 time unit. The condition is the hoist and crate are in the same place; the hoist is available and nothing is on top of the crate.

- ***Drop***: describes a hoist dropping a crate, which it is holding, i.e. has lifted onto a surface. The duration of this action is 1 time unit.

- ***Load***: describes a hoist putting a crate into a truck. The duration of this action depends on the weight of the crate and the power of the hoist.

- ***UnLoad***: describes a hoist picking up a crate from a truck. The duration of this action depends on the weight of the crate and the power of the hoist.

Note that in the international planning competition 2002 in which temporal planning domains were first introduced to the competition, there was no special track for optimal planning systems. Since the domains and problems were designed for all participating planning systems including heuristic-approach planners, the size and the complexity of the domains and problems were far beyond the ability of optimal planners. Therefore, in this experimental chapter, only small size problems were extracted for the comparison.

In this *Depot* domain, *Depot_1* is only one problem extracted. The other problems in this domain are too big for the optimal planners to handle within the limited time (i.e. 30 minutes). In *Depot1*, there are 1 depot, 2 distributors, 2 trucks, 3 pallets, 2 crates, and 3 hoists.

6.2.2.2 DriverLog

This domain describes the driving of trucks around places to deliver packages. It is more complicated than the original logistics domain because drivers have to walk from one truck to another to drive it. The details of the domain are as follow:

Types: the domains uses the types: *object*, *location*, *locatable*, *driver*, *truck*, and *obj*. A *driver*, a *truck*, or an *obj* are of type *locatable*. A *location* and a *locatable* are of type *object*.

Actions:

- ***LOAD-TRUCK***: an object is loaded into a truck. This action takes 2 time units.
- ***UNLOAD-TRUCK***: an object is unloaded from a truck. This action takes 2 time units.
- ***BOARD-TRUCK***: a driver gets into a truck to be ready to drive. This action takes 1 time unit.
- ***DISEMBARK-TRUCK***: a driver gets out of a truck. This action takes 1 time unit.
- ***DRIVE-TRUCK***: a truck is driven from one location to another. A driver must be in the truck and there must be a route linking the two locations. The duration of this action depends on the two locations.
- ***WALK-TRUCK***: a driver walks from one place to another. It is required that there is a path linking the two places. The duration of the action depends on the two places.

In this *DriverLog* domain, there are two problems, which are *DriverLog1* and *DriverLog3*, extracted. Table 6.2 shows the number of objects in each problem.

Problem	**drivers**	**trucks**	**packages**	**locations**
DriverLog_1	2	2	2	5
DriverLog_3	2	2	4	6

Table 6.2: The numbers of objects in each problem of the *DriverLog* domain extracted from IPC2002.

6.2.2.3 Satellite

The satellite domain is first PDDL-based planning domain bringing AI Planning towards space related applications. The domain was inspired by the talk about "Ambitious Spacecraft" delivered by David Smith at AIPS2000 [90]. The domain involves planning observations from multiple satellites. Each satellite has different instruments.

Types: there are four different types in this domain. They are *satellite*, *direction*, *instrument*, and *mode*.

Actions:

- ***turn_to***: describes the turning of a satellite from one direction to another. The duration depends on the two directions.
- ***switch_on***: switch on an instrument on a satellite. The duration is 2 time units.
- ***switch_off***: switch off an instrument on a satellite. The duration is 1 time unit.
- ***calibrate***: at the end of the action, an instrument is calibrated and ready to take images. The duration depends on the instrument and the direction that satellite is aiming.
- ***take_image***: the action of using the instrument on the satellite to take an image. The duration is 7 time units.

In this *Satellite* domain, *Satellite_1* and *Satellite_3* were extracted. Table 6.3 shows the number of objects in each problem.

6.2.2.4 Rovers

This is another domain relating to space applications. It was inspired by planetary exploration. In the domain, rovers navigate and find rock and soil samples on the surface. Then, they communicate back to the lander.

Problem	**satellites**	**instruments**	**modes**	**directions**
Satellite_1	1	1	3	7
Satellite_3	2	4	3	8

Table 6.3: The numbers of objects in each problem of the *Satellite* domain extracted from IPC2002.

Types: there are 7 different types in this domain. They are *rover*, *waypoint*, *store*, *camera*, *mode*, *lander* and *objective*.
Actions:

- ***navigate***: move from a waypoint to another. Duration is 5 time units.
- ***recharge***: recharge the energy to the rover at the recharge-rate. Duration depends on how much energy it needs to be recharged.
- ***sample_soil***: get a full *store* of sample of soil at waypoint. Duration is 10 time units.
- ***sample_rock***: get a full *store* of sample of rock at waypoint. Duration is 8 time units.
- ***drop***: drop the sample from the *store*. *Store* becomes empty.
- ***calibrate***: calibrate the camera to be ready. Duration is 5 time units.
- ***take_image***: takes the image of an object at a waypoint. Duration is 7 time units.
- ***communicate_soil_data***: communicates to the lander about the soil data. Duration is 10 time units.
- ***communicate_rock_data***: communicates to the lander about the rock data. Duration is 10 time units.

- ***communicate_image_data***: communicates to the lander about the images. Duration is 15 time units.

In this *Rover* domain, there were 4 problems extracted, from *Rover_1* to *Rover_4*. Table 6.4 shows the number of objects in each problem.

Problem	**modes**	**rovers**	**stores**	**waypoints**	**cameras**	**objectives**
Rover_1	3	1	1	4	1	2
Rover_2	3	1	1	4	2	2
Rover_3	3	2	2	4	2	2
Rover_4	3	2	2	4	3	3

Table 6.4: The numbers of objects in each problem of the *Rover* domain extracted from IPC2002.

6.2.2.5 ZenoTravel

This is another transportation domain, which involves in transporting people around cities by airplanes. The airplanes have different modes to travel, which are fast or slow. The main introduction of this domain is the numeric track, in which the fuel levels are discrete. The faster movement of the airplane is, the more fuel consumption it will use.
Types: there are 4 different types in this domain. They are *aircraft*, *person*, *city*, and *fuel level*.
Actions:

- ***board***: at a certain city, a person gets on the aircraft. Duration is 20 time units.
- ***debark***: a person gets off an aircraft to the city. Duration is 30 time units.
- ***fly***: an aircraft flies from one city to another with slow speed. Duration is 180 time units.

- ***zoom***: an aircraft flies from one city to another with fast speed. Duration is 100 time units. Fuel is consumed double comparing to the action *fly*.

- ***refuel***: an aircraft refuels. Duration is 73 time units.

In this *ZenoTravel* domain, there were 4 problems extracted, from *Zeno_1* to *Zeno_4*. Table 6.5 shows the number of objects in each problem.

Problem	**aircrafts**	**people**	**cities**	**fuel_level**
Zeno_1	1	2	3	7
Zeno_2	1	3	3	7
Zeno_3	2	4	3	7
Zeno_4	2	5	3	7

Table 6.5: The numbers of objects in each problem of the *ZenoTravel* domain extracted from IPC2002.

6.2.3 Airport - *a temporal domains extracted from the IPC2004*

The *Airport* domain, created by Sebastian Trueg for IPC2004, describes the operations involving airplanes at an airport. The details of the airport domain are as follows:
Types:
The types of airplane are: *small*, *medium*, and *large*. This describes the types of an airplane. Direction: *north*, and *south*. This shows the direction that the airplane is moving. Segments: *seg_pp_0_60, seg_ppdoor_0_40, seg_tww1_0_200* etc. This describes the name of a certain segment of the airport. Airplane names: these were used to identify the airplanes.
Actions:

- ***Move***: an airplane moves from one segment to another. It depends on the two segments involved to know the duration and conditions and effects of that action.

- ***TakeOff***: an airplane takes off from a certain segment. The duration is 30 time units.
- ***Park***: an airplane parks at a certain segment. The duration is 40 time units.
- ***StartUp***: an airplane starts up and is in the status of moving. The duration depends on the engine of that airplane. It is calculated as (60 * engine) time units.

In this *Airport* domain, there were 7 problems extracted, from *Airport1* to *Airport7*. Table 6.6 shows the size and complexity of those problems.

Problem	**directions**	**airplane types**	**segments**	**airplanes**
Airport1	2	3	17	1
Airport2	2	3	17	1
Airport3	2	3	17	2
Airport4	2	3	40	1
Airport5	2	3	40	1
Airport6	2	3	40	2
Airport7	2	3	40	2

Table 6.6: The numbers of objects in each problem of the *Airport* domain extracted from IPC2004.

6.2.4 Storage - *A temporal planning domain extracted from IPC2006*

In the latest planning competition 2006, the new version of the planning domain description language - PDDL3 was introduced. In this competition, the constraints were introduced into the PDDL (see chapter 2 for details). There were 5 different domains for the benchmark, in which there were 3 new domains - *storage*, *truck* and *pathways*, and the other two domains were taken from the IPC2004. Since *pathways* domain does not contain temporal descriptions, and *truck* domain is too difficult for temporal optimal planners, only *storage* was extracted for the experiment of CPPlanner.

Storage is a planning domain which involves spatial reasoning. The domain is about moving crates from certain containers to depots by hoists. In each depot, a hoist can move according to the spatial map connecting different depots. The details of the *storage* domain are as follows:

Types: There are 4 different types of for domain. They are: *object*, *place*, *area*, *surface*. *Hoist, surface, place, area* are objects. Container and depot are of type *place*. *Storearea* and *transitarea* are of type *place*. *Area* and *crate* are of type *surface*.

Actions:

- ***lift***: a hoist lifts a crate from a surface. Duration is 2 time units.
- ***drop***: a hoist drops a crate to a surface. Duration is 2 time units.
- ***move***: a hoist goes from one storearea to another storearea. Duration is 1 time unit.
- ***go_out***: a hoist goes from a storearea to a transitarea. Duration is 1 time unit.
- ***go_in***: a hoist goes from a transitarea to a storearea. Duration is 1 time unit.

In this *Storage* domain, there were 15 problems extracted, from *Storage1* to *Storage15*. Table 6.7 shows the size and complexity of those problems.

Problem	**storeareas**	**hoists**	**crates**	**containers**	**depots**	**transitarea**
Storage1	2	1	1	1	1	1
Storage2	3	2	1	1	1	1
Storage3	4	3	1	1	1	1
Storage4	6	1	2	1	1	1
Storage5	6	2	2	1	1	1
Storage6	6	3	2	1	1	1
Storage7	9	1	3	1	1	1
Storage8	9	2	3	1	1	1
Storage9	9	3	3	1	1	1
Storage10	12	1	4	1	1	1
Storage11	12	2	4	1	1	1
Storage12	12	3	4	1	1	1
Storage13	15	1	5	2	2	2
Storage14	15	2	5	2	2	2
Storage15	15	3	5	2	2	2

Table 6.7: The numbers of objects in each problem of the *Storage* domain extracted from IPC2006.

6.2.5 Spacecraft - *A temporal domains with intermediate effects*

This section describes a temporal planning domain which has actions with intermediate effects. Instead of decomposing complex actions into many primitive actions with constraints to keep them together, actions with intermediate effects were introduced in this thesis (see chapter 2 for details). Actions with intermediate effects make planning domains look more natural and realistic. In this section, an example of temporal planning domain, called *spacecraft*, with intermediate effects are introduced. The domain describes a spacecraft must turn an instrument to a specific target with many smaller operations involved.

The domain *spacecraft* was created mainly for the experiments of CPPlanner. It is based on a *complex* action which has intermediate effects introduced by David Smith in [91]. The domain describes a spacecraft in process of taking pictures of particular objects. In this domain, the spacecraft must turn into a particular direction to point an instrument at a particular target. The process of turning the spacecraft consists of many actions. At first, thrusters in the reaction control system (RCS) are fired to provide the angular velocity. Then, those thrusters are switched off but the spacecraft still keeps turning until it is closely to the target direction. At this time, the thrusters are fired again in the opposite direction to stop the rotation. In the whole process, the thruster firing is very quick, but the turning of the spacecraft is slow which may take several minutes. During the time of firing the thrusters, it will cause some vibration to the spacecraft which prevents some operations from performing. The process of turning a spacecraft is modelled by an action with intermediate effects. After pointing to the right object, the camera is calibrated to be ready. It is also set to the right mode to take pictures. After the picture taken, it will be sent to the control center and also printed a copy. However, the printing process requires the spacecraft still (i.e. not vibrating). Below is the overview of the domain.

Types: There are 4 different types for this domain. They are: *object*, *camera*, *mode*, and *direction*.

Actions:

- ***turn***: turn the spacecraft to a particular heading. It consists of firing the thrusters, coasting of the spacecraft, and firing the thrusters again. During performing this action, there are two periods of time (at the start and at the end when the thruster are in use) in which the vibration appears. The vibration prevents other operations from performing (e.g. printing pictures)

- ***calibrate***: calibrate the camera to be ready and point to the right direction to take pictures.

- ***set_mode***: set the capture mode for the camera.

- ***take_picture***: take a picture of a particular object.

- ***communicate***: send the picture to the control center.

- ***print_picture***: print a picture. This action cannot perform if there is any vibration (i.e. thrusters are in use).

In this domain, 5 different problems are created for the experiment. Table 6.13 shows the complexity of each problem in this domain.

Problem	**object**	**camera**	**mode**	**heading**
Spacecraft1	2	1	3	6
Spacecraft2	2	2	3	6
Spacecraft3	3	2	3	7
Spacecraft4	3	3	3	9
Spacecraft5	5	5	3	14

Table 6.8: The numbers of objects in each problem of the *Spacecraft* domain

6.3 Comparisons of planning systems

The temporal planning systems which were used in the experiments are CPPlanner-Basic, CPPlanner, TGP, TPSYS, and CPT. The details of TGP, TPSYS, and CPT were described in chapter 3. CPPlanner-Basic is the first developed version of CPPlanner which does not have the improvements, such as conflict-directed backjumping. CPPlanner is our latest version which was developed based on the algorithm described in chapter 4 with all the improvements described in chapter 5.

The experiments were carried out on a Athlon AMD XP2000 machine running at 1.667GHz with 256MB RAM and 1.5GB virtual memory. CPPlanner_Basic, CPPlanner, and TGP were run under the Windows XP operating system, whereas TPSYS runs on MEPIS Linux on the same machine.

6.3.1 Logistics *domain from TGP package*

The domain used in this section is the logistics domain extracted from the TGP package which is available from the TGP website. According to the author of TGP, the version they provided on the website is improved by adding techniques described in [92]. Thus, it performs much better than the version which was described in their IJCAI-99 paper [88].

Because TGP was developed on Macintosh Common Lisp, there were some problems in re-compiling it and running it in Windows or Linux Common Lisp. In some source files, there are no proper line terminations (CR only) which causes the whole file to be read as one line only. In addition, because Common Lisp has an elaborate system for referencing files, it is not portable across dialects of Common Lisp (e.g. the function full-pathname appears to be unique in Macintosh Common Lisp).

By editing the TGP sources files [1], it was eventually possible to compile and run TGP in Windows Common Lisp. The experiment was carried out on six temporal problems, which are log_1 to log_6, provided in the TGP package downloaded from the TGP website

[1] Stephen Cresswell helped with modifying the TGP files

[92]. See table 6.9 for details.

Problem	**CPPlanner_Basic**	**CPPlanner**	**TGP**	**makespan**
Log_1	7.25	1.58	3.72	15
Log_2	4.31	0.73	1.46	15
Log_3	2.71	0.21	0.27	15
Log_4	1.23	0.05	0.03	8
Log_5	1.16	0.03	0.01	24
Log_6	1.03	0.01	0.01	5

Table 6.9: The comparison of different planning systems on the *Logistics* domain. Time is in seconds to find an optimal solution.

In this experiment, the problem Log_1 is the biggest with 3 cities, 3 offices, 3 airports, 3 trucks, an airplane, and 3 packets. Problems Log_2 and Log_3 are simplified from Log_1 by changing the goal. Log_4, Log_5, Log_6, and Log_7 have a much simpler goal.

The experiment showed that CPPlanner performed better than TGP on most of the tests. In Log_4 and Log_5, because the size of the problems are very small, and CPPlanner spent time checking whether there are actions with intermediate effects, the result showed it is slower than TGP in those small test cases. In other tests, Log_1 and Log_2, CPPlanner outperformed TGP.

There is not much difference in the running time of the expansion phase in TGP and CP-Planner because it is only a small proportion of the whole performance. The two systems applied a similar approach to expand the graph although CPPlanner spends little more time in calculating the time for the next applied action. However, in the extraction phase, the CPPlanner uses the critical paths to support the backtracking search, which consequently shows a big improvement. In particular, in the test Log$_1$ and Log$_2$, because of the critical paths, CPPlanner has chosen a small number of actions and propositions before the search actually starts. The large remainder of possible actions is therefore pruned (made into irrelevant search branches) by the detection of mutex conflicts, a process inherent in CPPlanner.

The earlier version of CPPlanner, called CPPlanner_Basic, ran quite slowly because of the basic backtracking used and the redundant search in the solution extraction. With the usage of the critical paths and improvements in chapter 5, for instance the use of time-bound and directed backjumping, the performance of CPPlanner is much better than that of CPPlanner_Basic..

6.3.2 Temporal planning domains from IPC2002

The international planning competition in 2002 did not divide participating planning systems into optimal and non-optimal groups, and the planning domains were mostly designed for non-optimal planners. In most of the competition's test cases, the size of problems are too big for optimal planners to deal with. Although optimal plangraph-based planning systems use the plangraph (which is a compact graph of all possible states), the graph is still too big to fit in the memory of current computing hardware. This is because the plangraph grows very quickly as all possibilities of actions are incorporated in the graph. Thus, CPPlanner and TPSYS could not handle planning problems with large size in the IPC2002 test sets. Big planning problems are designed for non-optimal planners which use heuristic approaches.

In our experiments, due to the size of the test cases, only a few small problems were extracted from each of the planning domains. Thus Table 6.10 shows only problems which can be solved by TPSYS within a 30-minute time allowance on the testing machine.

In the *DriverLog* domain, $DriverLog\_1$ has 2 drivers, 2 trucks, 2 packets, and 5 locations. $Driverlog\_2$ is bigger than $DriverLog\_1$ with 2 more packets, and 1 more location. In the *Rover* domain, $Rover\_1$ has a lander, 3 modes, 1 rover, 1 store, 4 waypoints, a camera, and 2 objectives. $Rover\_2$, $Rover\_3$, and $Rover\_4$ are bigger than $Rover\_1$, in which

Problem	**CPPlanner_Basic**	**CPPlanner**	**TPSYS**	**makespan**
Depot1	0.89	0.08	0.17	28
DriverLog1	1.2	0.14	0.16	92
DriverLog3	6.21	1.68	3.9	42
Rover1	2.42	0.22	0.47	57
Rover2	1.42	0.2	0.23	45
Rover3	2.82	0.24	0.59	64
Rover4	3.7	0.35	0.64	50
Zeno1	2.15	0.23	0.14	173
Zeno2	173.23	35.27	77.94	592
Zeno3	37.34	6.72	10.9	280
Satellite1	2.32	0.2	0.17	48
Satellite3	127.72	23.78	58.62	40

Table 6.10: Comparison to TPSYS on IPC2002 temporal planning domains. Time is in seconds to find an optimal solution.

$Rover\_2$ and $Rover\_3$ have 2 cameras; and $Rover\_4$ has 3 cameras and 3 objectives.

In the *Zeno* domain, $Zeno\_1$ has an airplane, 2 people, 3 cities, and 7 fuel levels. Comparing to $Zeno\_1$, $Zeno\_2$ has one more person, and $Zeno\_3$ has 1 more airplane and 2 more people. In the *satellite* domain, $satellite\_1$ has a satellite, an instrument, 3 modes, and 7 directions. $Satellite\_3$ has 1 more satellite, 3 more instruments, and 1 more directions.

In each planning domain, only a few small problems were solved optimally. In all those planning problems, CPPlanner was faster than TPSYS on the same hardware settings [2]. Table 6.10 shows the results on the DriverLog, Rover, Zeno, and Satellite domains.

[2] TPSYS runs on Linux and CPPlanner runs on Windows. However, both were developed based on C++

In the result, CPPlanner is about twice as fast as TPSYS for most of problems. In small test cases, e.g. *Zeno1*, *Rover_2*, *Satellite1*, because CPPlanner spent more time in checking whether actions have intermediate effects, the CPU time was longer than that of TPSYS. Again, the table demonstrates that the final version of CPPlanner is much faster than the basic version CPPlanner_Basic.

6.3.3 *The* Airport *temporal domain extracted from IPC2004*

Since other planning domains used in the international planning competition in 2004 have resource information, which CPPlanner does not support yet, the *Airport* temporal planning domain is the only one we extracted to use in our experiments. Because the *Airport* domain was described with PDDL2.2 specifications, TGP and TPSYS could not handle it. TGP was developed well before the introduction of PDDL2.2 and the domain specification that TGP supports is its own definition. However, although TPSYS can handle PDDL2.1 domains, these appear to be a little bit difference in the syntax of PDDL2.2 and PDDL2.1 (see chapter 2 for details), which raised a parsing error when trying to run TPSYS on this domain. Table 6.11 shows results on the *Airport* domain in comparison to CPT.

Although CPPlanner solves these problems optimally within a few seconds, the experiment showed that CPT (see chapter 3 for details) performed better than CPPlanner in all test cases. The good performance of CPT is mainly because of the ability to make inferences about actions which are not yet in the plan, the efficient calculation of the lower bound of the starting time of actions, and the propagation of Constraint Programming using the inferences and lower bounds. Thus, in CPT, all variables of the encoding problem are reasoned. The propagation helps to reduce the domain of each variable quickly and efficiently. Irrelevant branches can be pruned while the constraint solver is looking for a solution.

---

and tested on the same machine, i.e. using the same hardware settings. Runtimes are roughly comparable between the two systems. See the appendix A for more information on the difference when the same planning system runs on two different operating systems on the same machine

Problem	**CPPlanner**	**CPT**	**makespan**
Airport1	0.38	0.07	64
Airport2	0.57	0.09	185
Airport3	1.03	0.15	200
Airport4	0.72	0.18	127
Airport5	2.57	0.55	227
Airport6	5.35	0.59	232
Airport7	7.23	0.57	232

Table 6.11: Comparison to CPT on the *Airport* planning domain in the IPC2004. Time is in seconds to find an optimal solution.

Because CPT was developed using a totally different approach, CPPlanner was not expected to compete with CPT. In IPC2004, CPT was the second best planner in the optimal track. However, since SATPLAN'04, the best optimal planner in IPC2004, cannot handle temporal planning domains, CPT was considered to be the best planning system the optimal temporal planning.

CPPlanner would be competitive with CPT if the memoization (see chapter 7 for details) were applied. Also, the idea of using Constraint Programming in CPPlanner as described in the Further Developments (see also chapter 7) would be helpful to improve the performance of CPPlanner because the arc consistency checking and propagation might help to eliminate redundant branches early in the search. However, these improvements need to be fully analysed before they can be applied efficiently in CPPlanner.

6.3.4 The Storage *temporal domain extracted from the latest IPC2006*

There are 3 out of 5 planning domains which contain temporal parts in the planning competition IPC2006. Since one of them is too difficult for optimal planners, and another is the one from IPC2004, in this experiment, only the *storage* domain is extracted for comparison.

In IPC2006, the new version of CPT was introduced, named CPT2. CPT2 won the distinguished prize for the outstanding performance on temporal domains in IPC2006. Table 6.12 shows the comparison of CPPlanner and CPT2

Problem	**CPPlanner**	**CPT2**	**makespan**
Storage1	0.02	0.01	5
Storage2	0.03	0.01	5
Storage3	0.12	0.02	5
Storage4	0.57	0.05	12
Storage5	0.73	0.09	8
Storage6	0.98	0.11	8
Storage7	74.62	1.76	20
Storage8	136.13	0.87	12
Storage9	254.87	4.52	11
Storage10	-	-	-
Storage11	-	1252.21	17
Storage12	-	-	-
Storage13	-	-	-
Storage14	-	174.27	17
Storage15	-	64.82	13

Table 6.12: Comparison to CPT2 on the *Storage* planning domain in the latest IPC2006. Time is in seconds to find an optimal solution.

6.3.5 Spacecraft - A temporal domain with intermediate effects

Spacecraft is the first experiment of temporal domains with intermediate effects. Because other planners, such as TGP, TPSYS, and CPT could not handle planning domains with intermediate effects, only CPPlanner is tested for this domain. The table 6.13 shows the results on 5 different problems of this domains. In the 5 problems, the spacecraft5 is the most complicated one requiring to turn the camera of the spacecraft to 5 different objects.

Problem	**CPPlanner**
Spacecraft1	0.28
Spacecraft2	0.71
Spacecraft3	2.53
Spacecraft4	2.12
Spacecraft5	5.14

Table 6.13: Results for spacecraft domain in which actions has intermediate effects. Time is in seconds to find an optimal solution.

The experiment shows that CPPlanner is able to handle richer temporal planning domains in which actions may have intermediate effects. Instead of using many primitive actions to describe a complex action, the planning domain looks much simpler. In this case, the planning domain is more intuitive and smaller, and CPPlanner avoids redundant search branches by considering the whole complex action at one action.

6.4 Summary of the empirical study and discussion

The chapter presented empirical study of the development of CPPlanner and its comparison to the two best plangraph-based optimal temporal planning systems, TGP and TPSYS. In addition, it was also compared to the state-of-the-art planning system, CPT and the later version CPT2, on domain *Airport* of IPC2004, and domain *Storage* of IPC2006. The domains used in the experiments are taken from the TGP package, IPC2002, IPC2004, and the latest IPC2006.

Since the research focuses on improving plangraph approach to find optimal solutions for temporal planning domains, the main concern is the comparison with the two best plangraph-based optimal planners, TGP and TPSYS. In addition, the comparison to CPT and CPT2 is to show the competence of the plangraph approach to the state-of-the-art approach.

In the experiments, CPPlanner showed significant improvements in performance over TGP and TPSYS. In the *logistic* domain extracted from the TGP package, CPPlanner outperformed most of the test cases, except the small ones in which CPPlanner wasted time in checking whether actions have intermediate effects. With with improvements described in chapter 5, CPPlanner performs much better.

In the experimental evaluation with TPSYS, CPPlanner ran about twice as fast as TPSYS in most of test cases. Like the previous experiment, in small test cases, CPPlanner was a little bit slower because of spending time to check intermediate effects.

The experiment also shows that the critical paths and the improvements described in chapter 5, such as timebound, and directed backjumping, make CPPlanner several times faster than the basic version CPPlanner_Basic.

Finally, there were some experiments on CPPlanner against CPT and the later version CPT2. These planning systems use a totally different approach to find optimal solution (see chapter 3 for details). CPPlanner is not comparable to CPT/CPT2 in all test cases. However, CPPlanner still shows the competency of plangraph approach for optimal solutions.

It is worth noting that the problems solved in these experiments are small in comparison with most problems encountered in the real world. Although some small-scale planning problems do occur outside planning competition, for instance planning for photocopiers or elevator planning, optimal temporal planning remains very difficult for large problems.

Chapter 7

CONCLUSION AND FUTURE WORK

7.1 Introduction

This chapter revisits and expands on the main contributions, which were outlined in chapter 1, in the context of the previous chapters. Further developments for the CPPlanner are also discussed.

7.2 Summary of contributions

There follows the brief summary of the contributions of this thesis.

7.2.1 CPPlanner algorithm

The main contribution of the thesis is the introduction of a new and efficient plangraph-based algorithm to handle temporal planning domains to find an optimal solution. The novel ideas are the ”critical path” and the extension to handle intermediate effects.

7.2.1.1 “Critical paths”

The introduction of the “critical paths” into the plangraph is a novel contribution of the algorithm. While watching the plangraph expanding, it can be noticed that each time the plangraph is extended to the next timepoint, there is at least one “path” which starts from the proposition of the initial state and ends at a proposition at the current timepoint. One or more of these paths must be used in extending the graph to the next timepoint. Following this observation, the idea of a “critical paths” candidate was introduced. Actions and propositions of the “critical path” candidate are considered to be a part of the final plan. Thus, with this information from the expansion stage, actions and propositions of the selected critical path are added to the final plan before the backtracking search takes place. This helps to prune redundant search branches early and to reduce the search space efficiently. In addition to “critical paths”, other improvements, namely timebounds and conflict directed backjumping, are important contributions to the performance of the system. The “critical paths” and these improvements speed up the performance of CPPlanner even in temporal planning domains, in which actions have no intermediate effects. The experimental evaluation was illustrated in chapter 6, in which CPPlanner outperforms TGP and TPSYS in almost all test cases.

7.2.1.2 Extension to the algorithm to handle intermediate effects

In addition to “critical paths”, the algorithm has been extended to deal with intermediate effects. CPPlanner builds on the representation of time and duration used in TGP and extends the graph expansion and solution extraction phase to deal with intermediate effects. The graph expansion phase requires more complex calculations and comparisons to advance the graph to the next timepoint. Instead of using the current timepoint, the algorithm needs to to keep tracks of all new coming intermediate effects to find out which action will be applied next to extend the graph. In addition, the solution extraction phase requires complicated checkings (i.e. mutex relations need to be checked again) when actions with intermediate effects appear (see chapter 4 for details).

7.2.2 Extension to PDDL2.1 level 3

Another contribution of this thesis is the extension to the current PDDL2.1 level 3 to allow actions with intermediate effects. In the original PDDL 2.1, actions are assumed to have effects only at the beginning and at the end of the execution. This leads to the fact that if any other action needs to use the effect of the current action, it has to wait until the current action finishes. On the other hand, in our daily lives, there are many actions which have effects at any time during their executions (see chapter 2 for details). Within PDDL2.1 level 3, actions with intermediate effects can also be represented by splitting it into many primitive actions with constraints to join them together (see paper [36] in which Long and Fox tweaked the PDDL2.1 level 3 to handle intermediate effects). However, the decomposition makes the planning domain more cumbersome and complicated. It also increases the run time of planning systems because of scanning redundant search branches. This thesis enriched the specifications of PDDL 2.1 level 3 to be able to define actions with intermediate effects directly. Chapter 2 discussed details of this extension in terms of PDDL syntax and semantics.

7.2.3 CPPlanner - a new optimal planning system for temporal planning domains

At the start of this research, the plangraph framework was considered one of the most promising approaches by the planning community. After the introduction of Graphplan in 1995, there was a period of intense research activity, leading to different extensions and developments based on the plangraph framework. TGP, which uses the plangraph framework, was the state-of-the-art planning system for optimal temporal planning at the time. The plangraph framework therefore appeared to be a promising foundation for improvements to optimal planning. The development of CPPlanner followed this promising line of research. CPPlanner is a new plangraph-based optimal planning system to deal with temporal planning domains with intermediate effects. The extension to actions with intermediate effects allows CPPlanner to deal with richer planning domains than existing optimal temporal planners, while the improvements in search, such as the use of "critical paths", mean that it outperforms competing systems even in more restricted temporal planning domains. In the experimental evaluation, it performed better than TGP and TPSYS in almost all test cases. Although CPPlanner performs well in the experiments, it was not able to compete with the current best temporal planner, CPT. It is hoped that the performance would improve if the further developments in the next section were added.

7.3 Further developments

This section outlines ideas for further developments of CPPlanner.

7.3.1 Memoisation

In classical planning, using memoisation improves the backtracking search efficiency [53] because it avoids revisiting search branches in the solution extraction phase. However, in temporal planning, the memoisation is much more complex because of the introduction of time into each node of the plangraph. Further investigation needs to be carried out to find a compact structure to store propositions and their timestamps so that later backtracking search for a solution is still able to tell whether a search branch has already been checked. The tradeoff of memory consumption and speed needs to be investigated carefully.

7.3.2 Heuristic search

The planning community is trying to move towards solving real-world problems. However, in AI Planning, even the blocksworld problem is NP-complete. Therefore, in large problems, it is impossible to find optimal solutions within limited time and resources. One of the further developments to the current CPPlanner could be the introduction of heuristic search. However, the main framework of optimal backtracking search could be still kept. Whenever an input problem is over a size limit, defined by CPPlanner, the heuristic search would be triggered to handle it. In this case, the plangraph expansion phase could be relaxed to remove detailed information, to speed up the whole process.

There are a few existing planning systems [80, 50, 23, 33] which use the plangraph framework as the foundation for a heuristic approach. The main idea of this extension is to build up a relaxed plangraph and use it to provide a measurement for the heuristic function in the heuristic search.

In addition, another idea to relax the current backtracking search of CPPlanner to an heuristic approach is to go backwards beyond the timepoint 0. It is noted that when searching for a solution, if the planner goes back to timepoint 0 but has not found a solution yet, that search branch has reached a dead-end. However, the reason for the failure may be that the actions cannot fit into the current length because of the mutex relations. If the plan can be extended further, beyond the timepoint 0, there is a chance that the whole plan could be constructed and a solution found. In this case, the plan must be shifted to the right to give the solution. Of course, the solution found may not be optimal but it saves a lot of time comparing with expanding the graph to the next timepoint and do the solution extraction search again. This idea needs further investigation to identify when the planner should do a further search back beyond timepoint 0, and how far it should go.

These extensions could allow CPPlanner not only to be able to handle small planning problems to get optimal solutions but also allow it to deal with bigger and more complex problems to get feasible solutions in a limited time constraint.

7.3.3 Using current Constraint Programming Solvers

It is noted that the success of CPT was the use of constraint propagation within a Constraint Programming Solver, i.e. Choco [1]. Constraint Programming can be an efficient route to finding optimal solutions. In order to improve the performance of CPPlanner by using Constraint Programming, the following idea could be investigated.

Apply constraint programming for the solution extraction: In classical planning, GP-CSP [22] encoded the solution extraction phase as a CSP and used a CP Solver to deal with it. This approach can be investigated and extended to handle temporal information for each node of the graph. This idea can take advantage of current search strategies and constraint propagation of a CP Solver to speed up the search. The drawback is that it still relies on finding a solution by doing exhaustive search in each timepoint. However, if there is a good memoisation approach to avoid revisiting search branches, this approach might improve the whole performance dramatically.

Another idea which might be thought of is ***apply constraint programming for the whole planner***. However, finding a suitable CSP encoding of the graph expansion is very difficult. There was a proposal on how to encode the whole classical planning problem into a CSP in [68] and use the CP Solver to handle it, but this has not been taken further. The idea has potential but it has to be re-considered when introducing the temporal information into the problem.

7.4 ...and finally

In short, it is hoped that the contributions and experimental results of this thesis are of use for any further development in the field of Artificial Intelligence Planning. It is also hoped that CPPlanner can be developed further to handle bigger and more complex planning domains to move closer to real-world planning problems.

BIBLIOGRAPHY

[1] Choco constraint programming system. *Available at http://choco-solver.net/index.php?title=Main_Page*.

[2] CPT: Constraint Programming Temporal planner. *Available at http://www.cril.univ-artois.fr/ vidal/cpt.html*.

[3] F. Bacchus and M. Ady. Planning with Resources and Concurrency: A Forward Chaining Approach. In *Proceeding of International Joint Conference on Artificial Intelligence*, pages 417–424, 2001.

[4] F. Bacchus and F. Kabanza. Using Temporal Logic to Control Search in a Forward Chaining Planner. In M. Ghallab and A. Milani, editor, *New Directions in Planning*, pages 141–153. IOS Press, 1996.

[5] F. Bacchus and F. Kabanza. Planning for Temporally Extended Goals. *Annals of Mathematics and Artificial Intelligence*, 22:5–27, 1998.

[6] F. Bacchus and F. Kabanza. Using Temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 22:5–27, 2000.

[7] F. Bacchus and F. Kabanza. Using Temporal Logics to Express Search Control Knowledge for Planning. *Artificial Intelligence*, 116, 2000.

[8] A. Barrett and D. S. Weld. Partial order planning: Evaluating possible efficiency gains. *Artificial Intelligence*, 67(1):71–112, 1994.

[9] Eric Beaudry, Froduald Kabanza, and Franois Michaud. Planning for a Mobile Robot to Attend a Conference. In *Proceeding of the Canadian Conference on Artificial Intelligence*, pages 48–52, 2005.

[10] M. Beetz. Plan-based Control of Robotics Agents. *Lecture Notes om Artificial Intelligence*, 2554, 2002.

[11] M. Beetz and T. Belker. Environment and task adaptation for robotics aganets. In *Proceeding of the European Conference on Artificial Intelligence Robotics and Automation in Space*, pages 648–657. IOS Press, 2000.

[12] M. Beetz, J. Hertzberg, M. Ghallab, and M. Pollack. Advances in Plan-Based Control of Robotics Agents. *Lecture Notes om Artificial Intelligence*, 2266, 2002.

[13] A. L. Blum and M. L. Furst. Fast planning through Planning Graph Analysis. *Artificial Intelligence*, 90:281–300, 1997.

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

[15] Luis Castillo, Juan Fdez-Olivares, and Antonio Gonzalez. A temporal constraint network based temporal planner. In *Proceedings of the 21st workshop of the UK Planning and Scheduling Special Interest Group PLANSIG2002*, 2002.

[16] A. Cesta and A. Odi. Gaining efficiency and flexibility in the simple temporal problem. In *Proceeding of 3rd International Workshop on Temporal Representation and Reasoning*. IEEE-CS Press, 1996.

[17] S. Chien. Static and completion analysis for planning knowledge base development and verification. In *Proceeding of the Third International Conference on Artificial Intelligence Planning Systems*, pages 53–61, 1996.

[18] Steve Chien, Benjamin Smith, Gregg Rabideau, Nicola Muscettola, and Kanna Rajan. Automated Planning and Scheduling for Goal-Based Autonomous Spacecraft. *IEEE Intelligent Systems*, 13:5:50–55, 1998.

[19] K. Currie and A. Tate. O-PLAN: The open planning architecture. *Artificial Intelligence*, 51(1):49–86, 1991.

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

[21] R. Detchter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 91.

[22] M. B. Do and S. Kambhampati. Solving Planning Graph by compiling it into a CSP. In *Proceedings of the Fifth International Conferenceon Artificial Intelligence Planning and Scheduling*, 2000.

[23] M. B. Do and S. Kambhampati. Sapa: A Domain-Independent Heuristic Metric Temporal Planner. In *Proceedings of European Conference on Planning*, 2001.

[24] P. Doherty and J. Kvarnstrm. TALplanner: An Empirical Investigation of a Temporal Logic-based Forward Chaining Planner. In *Proceedings of the 6th International Workshop on the Temporal Representation and Reasoning*, 1999.

[25] P. Doherty and J. Kvarnstrm. TALplanner: A Temporal Logic Based Planner. *AI Magazine*, 2001.

[26] P. Drabble and A. Tate. The use of optimistic and pessimistic resource profiles to inform search in an activity based planner. In *Proceedings of 2nd International Conference AI Planning Systems*, 1994.

[27] S. Edelkamp and J. Hoffmann. The language for the 2004 international planning competition. *Technical Report, available at: http://ls5-www.cs.uni-dortmund.de/ edelkamp/ipc-4/pddl.html*, 2004.

[28] Stefan Edelkamp, Jorg Hoffmann, Michael Littman, Hakan Younes, et al. International Planning Competition 2004 - IPC4. Available at:http://andorfer.cs.uni-dortmund.de/ edelkamp/ipc-4/, 2004.

[29] Amin El-Kholy and Barry Richards. Temporal and resource reasoning in planning: The parcPLAN approach. In W. Wahlster, editor, *Proceedings of the 12th European Conference on Artificial Intelligence (ECAI-96)*, pages 614–618. Wiley & Sons, 1996.

[30] Kultuhan Erol, Dana Nau, and V.S. Subrahmania. Complexity, Decidability and Undecidability for Domain Independent Planning. *Artificial Intelligence*, 76:75–88, 1995.

[31] R. E. Fikes and N. J. Nilsson. STRIPS: a new approach to the application of theoremproving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.

[32] M. Fox and D.Long. PDDL+: An extension to PDDL2.1 for modelling planning domains with continuous time-dependent effects. In *Technical Report, Dept of Computer Science, University of Durham*, 2001.

[33] M. Fox and D. Long. Hybrid STAN: Identifying and Managing Combinatorial Optimisation Sub-problems in Planning. In *Proceedings of IJCAI*, 2001.

[34] M. Fox and D. Long. PDDL 2.1: An extension to PDDL for expressing temporal planning domains. *Technical Report, Department of Computer Science, University of Durham, UK*, 2001.

[35] M. Fox and D. Long. PDDL+ Level 5: An extension to PDDL2.1 for Modelling Planning Domains with Continous Time-dependent Effects. *Technical Report, available at: http://planning.cis.strath.ac.uk/publications/oldpapers/pddllevel5.ps.gz*, 2001.

[36] M. Fox, D. Long, and K. Halsey. An investigation into the expressive power of PDDL2.1. In *Proceedings of the Tenth European Conference on Artificial Intelligence*, 2004.

[37] Maria Fox and Derek Long. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

[38] A. Garrido, M. Fox, and D. Long. Temporal Planning with PDDL2.1. In *Proceeding of ECAI'02*, 2002.

[39] Antonio Garrido and Eva Onaindia. On the application of least-commitment and heuristic search in temporal planning. In *Proceedings of the International Joint Conference on Artificial Intellience*, pages 942–947, 2003.

[40] A. Gerevini and D. Long. Plan Constraints and Preferences in PDDL3. *Technical Report, RT 2005-08-47, Dept. of Electronics for Automation, University of Brescia, Italy. Available at: http://zeus.ing.unibs.it/ipc-5/pddl-ipc5.pdf*, 2006.

[41] A. Gerevini and L. Schubert. Efficient algorithms for handling qualitative reasoning about time. *Artificial Intelligence*, 74(1):207–248, 1995.

[42] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Pratice*. Elsevier, 2004.

[43] Keith Golden, Wanlin Pang, Ramakrishna Nemani, and Petr Votava. Automated Data Processing as an AI Planning Problem. In *NASA, avalaible at: http://ase.arc.nasa.gov/publications/pdf/0629.pdf*, 2003.

[44] C. Green. Theorem proving by resolution as a basis for questionanswering systems. *Machine Intelligence*, 4, 1969.

[45] S.K. Gupta, D.S. Nau, and W.C. Regli. IMACS: A case study in real-world planning. In *IEEE Expert and Intelligent Systems*, volume 3, pages 49–60, 1998.

[46] Patrik Haslum. Improving heuristics through search. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI-2004)*, pages 1031–1032, 2004.

[47] Patrik Haslum. TP'04 and HSP$^*{}_a$. *The 4th International Planning Competition Booklet. Available at http://ls5-web.cs.uni-dortmund.de/ edelkamp/ipc-4/*, pages 38–40, 2004.

[48] Patrik Haslum and Hector Geffner. Admissible heuristics for optimal planning. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling*, pages 70–82, 2000.

[49] Patrik Haslum and Hector Geffner. Heuristic planning with time and resources. In *Proceedings of the European Conference on Planning*, 2001.

[50] J. Hoffman. FF: The Fast Forward planning system. *AI Magazine*, 22(3):57–62, 2001.

[51] Luke Hunsberger. Algorithms for a temporal decoupling problem in multi-agent planning. In *The eighteenth national conference on Artificial intelligence*, pages 468–475. American Association for Artificial Intelligence, 2002.

[52] Nathanael Hyafil and Fahiem Bacchus. Utilizing structured representations and csp's in conformant probabilistic planning. In *Proceedings of the Tenth European Conference on Artificial Intelligence*, pages 1033–1034, 2004.

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

[54] Henry Kautz and Bart Selman. Unifying SAT-Based and Graph-Based Planning. In Jack Minker, editor, *Workshop on Logic-Based Artificial Intelligence, Washington, DC, June 14–16, 1999*. Computer Science Department, University of Maryland, 1999.

[55] Henry Kautz, Bart Selman, and Joerg Hoffmann. Satplan: Planning as satisfiability. http://www.cs.rochester.edu/u/kautz/satplan/index.htm, 2006.

[56] Henry A. Kautz and Bart Selman. Planning as Satisfiability. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, pages 359–363, 1992.

[57] Russell Knight, Gregg Rabideau, Steve Chien, Barbara Engelhardt, and Rob Sherwood. Casper: Space Exploration through Continuous Planning. *IEEE Intelligent Systems*, 16:5:70–55, 2001.

[58] J. Koehler. Planning under resource constraints. In *Proceedings of the 15th European Conference on AI*, 1998.

[59] J. Koehler. Metric planning using planning graphs - A first investigation. In *Technical report No. 127, Albert Ludwings University*, 1999.

[60] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos. Extending Planning Graphs to an ADL Subset. In *Proceedings of 5th the European Conference in Planning*, pages 273–285, 1997.

[61] J. Kvarnstrm and P. Doherty. TALplanner: A Temporal Logic Based Forward Chaining Planner. *Annals of Mathematics and Artificial Intelligence (AMAI)*, 30:119–169, 2001.

[62] J. Kvarnstrom, P. Doherty, and P. Haslum. Extending TALplanner with Concurrency and Resources. In *Proceedings of the 14th European Conference on Artificial Intelligence*, 2000.

[63] P. Laborie and M. Ghallab. Planning with sharable resource constraints. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1643–1649, 1995.

[64] P. Laborie and M. Ghallab. Planning with sharable resource constraints. In *Proceedings of the International Joint Conference on Artificial Intellience*, pages 1643–1649, 1995.

[65] Olivier Lhomme. Consistency techniques for numeric csps. In *Proceedings of the International Joint Conference on Artificial Intellience*, pages 232–238, 1993.

[66] D. Long and M. Fox. Progress in AI Planning Research and Applications. *Upgrade*, 5:10–25, 2002.

[67] D. Long and M. Fox. Exploiting a graphplan framework in temporal planning. In *Proceedings of The Thirteenth International Conference on Automated Planning and Scheduling*, pages 51–62, 2003.

[68] Adriana Lopez and Fahiem Bacchus. Generalizing graphplan by formulating planning as a csp. In *Proceedings of the International Joint Conference on Artificial Intellience*, pages 954–960, 2003.

[69] Amol Mali. Encoding temporal planning as CSP. In *Proceeding of the AIPS workshop on planning in temporal domains, Toulouse, France*, pages 18–25.

[70] J. McCarthy. Programs with common sense. In *Memo No 7, Stanford Artificial Intelligence Project*. Stanford University, 1963.

[71] J. McCarthy and P.J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4, 1969.

[72] D. McDermott and the AIPS-98 Planning Competition Committee. PDDLthe planning domain definition language. *Technical report, available at: http://www.cs.yale.edu/homes/dvm*, 1998.

[73] Drew McDermott. The 1998 AI Planning Systems Competition. *AI Magazine*, 21(2):35–55, 2000.

[74] N. Muscettola, B. Smith, S. Chien, C. Fry, K. Rajan, S. Mohan, G. Rabideau, and D. Yan. On-Board Planning for the New Millennium Deep Space One Spacecraft. In *Proceeding of the IEEE Aerospace Conference*, pages 303–318, 1997.

[75] D. Nau, S. Gupta, and W. Regli. Artificial intelligence planning versus manufacturing-operation planning: a case study. In *Proceeding of 14th International Joint Conference on Artificial Intelligence*, pages 1670–1676. Morgan Kaufmann, 1995.

[76] D. S. Nau, T. C. Au, O. Ilghami, U. Kuter, W. Murdock, D. Wu, and F. Yaman.

SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research*, 20(3):379–404, 2003.

[77] D. S. Nau, Y. Cao, A. Lotem, and H. Munoz-Avila. SHOP: Simple Hierarchical Ordered Planner. In *Proceedings of the International Joint Conference on Artificial Intellience*, pages 968–973, 1999.

[78] D. S. Nau, H. Munoz-Avila, Y. Cao, A. Lotem, , and S. Mitchell. Total ordering with partially ordered subtasks. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 2001.

[79] A. Newell and H. Simon. GPS, a program that simulates human thought. *Computers and Thought*, 1963.

[80] R. Nigenda, X. Nguyen, and S. Kambhampati. AltAlt: Combining the advantages of Graphplan and heuristic state search, 2000.

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

[82] J. Penberthy and D. S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of the International Conference on Knowledge Representation and Reasoning(KR)*, pages 103–114, 1992.

[83] J. Penberthy and D. S. Weld. Temporal planning with continuous change. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 1010–1015, 1994.

[84] J.S. Penberthy. Planning with Continuous Change. *PhD thesis*, 1993.

[85] P. Prosser. Domain filtering can degrade intelligent backtracking search. In *Proceedings of IJCAI*, pages 262–267, 1993.

[86] G. Rabideau, R. Knight, S. Chien, A. Fukunaga, and A. Govindjee. Iterative Repair Planning for Spacecraft Operations in the ASPEN System. In *Proceeding of the Fifth International Symposium on Artificial Intelligence Robotics and Automation in Space*, pages 99–106. ESA Publications Division, 1999.

[87] J. Rintanen and J. Hoffmann. An Overview of Recent Algorithms for AI Planning. *Knstliche Intelligenz*, 2/01:5–11, 2001.

[88] D. Smith and D. Weld. Temporal Planning with Mutual Exclusion Reasoning. In *Proceedings of IJCAI*, pages 326–337, 1999.

[89] D. Smith and D. Weld. Temporal Planning with Mutual Exclusion Reasoning. In *Proceedings of IJCAI*, pages 326–337, 1999.

[90] David Smith. Coping with Time and Continuous Quantities. *Talks at the AIPS 2000. Available at http://ic.arc.nasa.gov/people/de2smith/publications/AIPS-2000-talk.pdf*, 2000.

[91] David Smith. The case for durative actions: A commentary on PDDL2.1. *Journal of Artificial Intelligence Research*, 20:149–154, 2003.

[92] David Smith and Daniel Weld. Temporal Graphplan. *Homepage for Temporal Graphplan (TGP). Available at http://www.cs.washington.edu/ai/tgp.html.*

[93] S. Smith, D. Nau, and T. Throop. Total-order multi-agent task-network planning for contract bridge. In *Proceeding of the Fourteenth National Conference on Artificial Intelligence*, pages 108–113, 1996.

[94] B. Srivastava and S. Kambhampati. Scaling up planning by teasing out resource scheduling. In *Proceedings of 5th the European Conference in Planning*, 1999.

[95] B. Srivastava and S. Kambhampati. Realplan: decoupling causal and resource reasoning in planning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2000.

[96] Pavel Surynek and Roman Barták. Encoding htn planning as a dynamic csp. In *Proceeding of the Eleventh International Conference on Principles and Practice of Constraint Programming*, page 868, 2005.

[97] Peter vanBeek and Xinguang Chen. CPlan: A constraint programming approach to planning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 585–590, 1999.

[98] Vincent Vidal and Hector Geffner. Branching and pruning: An optimal temporal pocl planner based on constraint programming. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 570–577, 2004.

[99] Vincent Vidal and Hector Geffner. Branching and pruning: An optimal temporal pocl planner based on constraint programming. *Artificial Intelligence*, 170(3):298–335, 2006.

[100] Vincent Vidal and Sebastien Tabary. The New Version of CPT, an Optimal Temporal POCL Planner based on Constraint Programming. *The 5th International Planning Competition 2006. Available at http://zeus.ing.unibs.it/ipc-5/*, 2006.

[101] D. Weld. Recent Advances in AI Planning. *AI Magazine*, 20:2, 1999.

[102] Daniel S. Weld. An introduction to least commitment planning. *AI Magazine*, 15(4):27–61, 1994.

[103] D. Wilkins. Can AI planners solve practical problems? *Computational Intelligence*, 6(4):232–246, 1990.

[104] D. Wilkins and K. Myers. A multi-agent planning architecture. In *Proceeding of the Fourth International Conference on Artificial Intelligence Planning Systems*, pages 154–162, 1998.

Appendix A

COMPARISONS A SAME PLANNER ON DIFFERENT OPERATING SYSTEMS

In this appendix, CPT [2] is tested on two different operating systems, MS Windows XP and Linux MEMPIS, under the same hardware configuration to show the equivalent performance. CPT was developed and distributed on different operating systems, including Linux, MS Windows and Solaris. The test is carried out on two temporal planning domains, Airport and Satellite, which were extracted from the International Planning Competition 2004.

Problem	**MS Windows XP time(s)**	**Linux MEMPIS time(s)**	**makespan**
Airport1	0.06	0.06	64
Airport2	0.06	0.06	185
Airport3	0.12	0.11	200
Airport4	0.15	0.13	127
Airport5	0.23	0.21	227
Airport6	0.65	0.58	232
Airport7	0.65	0.56	232

Table A.1: Comparison of CPT on Windows XP and Linux MEMPIS on domain *Airport* extracted from IPC2004. Time is in seconds to find an optimal solution.

In the *Airport* domain, the running times of CPT on the two platforms are nearly the same. When the problems get bigger, the performance of CPT on MS Windows is slightly slower than that on Linux. I think the reason for the difference is that CPT was first and mainly developed on Linux. Then, it was ported to MS Windows. Some optimizations might not be considered thoroughly enough. When problems are getting bigger, the time difference shows more clearly.

Problem	**MS Windows XP time(s)**	**Linux MEMPIS time(s)**	**makespan**
Satellite1	0.07	0.11	135.486
Satellite2	1.36	1.26	156.13
Satellite3	0.24	0.26	65.198
Satellite4	3.44	3.36	122.24
Satellite5	1.17	1.14	105.26
Satellite6	3.87	3.59	64.82
Satellite7	2.66	3.02	60.202

Table A.2: Comparison of CPT on Windows XP and Linux MEMPIS on domain *Satellite* extracted from IPC2004. Time is in seconds to find an optimal solution.

In the *Satellite* domain, the difference in running time of CPT on the two platforms is also very small. In general, it runs a bit slower on MS Windows than on Linux (except *Satellite7*).

In the two temporal domains extracted from the IPC2004, CPT shows nearly the same performance on the two different operating systems, MS Windows XP and MEMPIS Linux. It is arguably inferred that planning systems which are developed on different platforms can be compared roughly if they are tested on the same hardware settings.

Appendix B

PLANNING DOMAINS AND PROBLEMS USED IN THE EXPERIMENT

B.1 Logistics domain from the TGP package

B.1.1 Logistics domain

Below is the detailed description of the ***logistics*** domain extracted from the TGP package [92].

```
(in-package :domains)
(define (domain logistics)
  (:requirements :strips :equality)
  (:predicates (packet ?x)(vehicle ?x)
               (truck ?x)(airplane ?x)
               (location ?x)(airport ?x)
               (city ?x)(loc-at ?x ?y)
               (at ?x ?y)(in ?x ?y))
  (:action load
    :parameters (?x ?y ?z)
    :precondition (and (packet ?x) (vehicle ?y) (location ?z)
                       (at ?x ?z) (at ?y ?z))
    :effect  (and (not (at ?x ?z)) (in ?x ?y))
    :duration 1
  )
```

```
(:action unload
  :parameters (?x ?y ?z)
  :precondition (and (packet ?x) (vehicle ?y) (location ?z)
                     (in ?x ?y) (at ?y ?z))
  :effect  (and (not (in ?x ?y)) (at ?x ?z))
  :duration 1
)

(:action drive
  :parameters (?x ?y ?z ?c)
  :precondition (and (truck ?x) (location ?y) (location ?z)
                     (city ?c)
                     (not (= ?y ?z)) (loc-at ?y ?c)
                          (loc-at ?z ?c)
                     (at ?x ?y))
  :effect  (and (not (at ?x ?y)) (at ?x ?z))
  :duration 2
)

(:action drive-inter-city
  :parameters (?x ?y ?z)
  :precondition (and (truck ?x) (location ?y) (location ?z)
                     (not (= ?y ?z)) (at ?x ?y))
  :effect  (and (not (at ?x ?y)) (at ?x ?z))
  :duration 11
)

(:action fly
    :parameters (?x ?y ?z)
    :precondition (and (airplane ?x) (airport ?y) (airport ?z)
                       (not (= ?y ?z)) (at ?x ?y))
    :effect  (and (not (at ?x ?y)) (at ?x ?z))
    :duration 3
  )
 )
```

B.2 Planning domains from IPC2002

Temporal planning domains, including ***Depot***, ***DriverLog***, ***Satellite***, ***Rovers***, ***ZenoTravel***, are extracted from the 3^{rd} Planning Competition in 2002. The detail of each domain is described as follow:

B.2.1 Depot

```
(define (domain Depot)
(:requirements :typing :durative-actions)
(:types place locatable - object
        depot distributor - place
        truck hoist surface - locatable
        pallet crate - surface)
(:predicates (at ?x - locatable ?y - place)
             (on ?x - crate ?y - surface)
             (in ?x - crate ?y - truck)
             (lifting ?x - hoist ?y - crate)
             (available ?x - hoist)
             (clear ?x - surface))

(:durative-action Drive
 :parameters (?x - truck ?y - place?z-place)
 :duration (= ?duration 10)
 :condition (and (at start (at ?x ?y)))
 :effect (and (at start (not (at ?x ?y)))
              (at end (at ?x ?z)))
 )
(:durative-action Lift
 :parameters (?x - hoist ?y - crate ?z - surface ?p - place)
 :duration (= ?duration 1)
 :condition (and (over all (at ?x ?p))
                 (at start (available ?x))
                 (at start (at ?y ?p)) (at start (on ?y ?z))
                 (at start (clear ?y)))
 :effect (and (at start (not (at ?y ?p)))
              (at start (lifting ?x ?y))
              (at start (not (clear ?y)))
              (at start (not (available ?x)))
              (at start (clear ?z))
              (at start (not (on ?y ?z))))
 )

(:durative-action Drop
 :parameters (?x - hoist ?y - crate ?z - surface ?p - place)
 :duration (= ?duration 1)
 :condition (and (over all (at ?x ?p)) (over all (at ?z ?p))
                 (over all (clear ?z)) (over all (lifting ?x ?y)))
 :effect (and (at end (available ?x))
              (at end (not (lifting ?x ?y)))
```

```
             (at end (at ?y ?p)) (at end (not (clear ?z)))
             (at end (clear ?y)) (at end (on ?y ?z)))
 )
(:durative-action Load
 :parameters (?x - hoist ?y - crate ?z - truck ?p - place)
 :duration (= ?duration 3)
 :condition (and (over all (at ?x ?p)) (over all (at ?z ?p))
                (over all (lifting ?x ?y)))
 :effect (and (at end (not (lifting ?x ?y)))
             (at end (in ?y ?z))
             (at end (available ?x)))
 )
(:durative-action Unload
 :parameters (?x - hoist ?y - crate ?z - truck ?p - place)
 :duration (= ?duration 4)
 :condition (and (over all (at ?x ?p))(over all (at ?z ?p))
                (at start (available ?x)) (at start (in ?y ?z)))
 :effect (and (at start (not (in ?y ?z)))
             (at start (not (available ?x)))
             (at start (lifting ?x ?y)))
 )
)
```

B.2.2 DriverLog

This is a logistics domain extracted from the IPC2002 Planning Competition:

```
(define (domain driverlog)
  (:requirements :typing :durative-actions)
  (:types location locatable - object
          driver truck obj - locatable)
  (:predicates
        (at ?obj - locatable ?loc - location)
        (in ?obj1 - obj ?obj - truck)
        (driving ?d - driver ?v - truck)
        (link ?x ?y - location) (path ?x ?y - location)
        (empty ?v - truck)
)

(:durative-action LOAD-TRUCK
  :parameters
   (?obj - obj
    ?truck - truck
    ?loc - location)
  :duration (= ?duration 2)
  :condition
   (and
   (over all (at ?truck ?loc)) (at start (at ?obj ?loc)))
  :effect
   (and (at start (not (at ?obj ?loc)))
        (at end (in ?obj ?truck))))

(:durative-action UNLOAD-TRUCK
  :parameters
   (?obj - obj
    ?truck - truck
    ?loc - location)
  :duration (= ?duration 2)
  :condition
   (and (over all (at ?truck ?loc))
        (at start (in ?obj ?truck)))
  :effect
   (and (at start (not (in ?obj ?truck)))
        (at end (at ?obj ?loc))))

(:durative-action BOARD-TRUCK
  :parameters
   (?driver - driver
    ?truck - truck
    ?loc - location)
  :duration (= ?duration 1)
  :condition
   (and (over all (at ?truck ?loc))
        (at start (at ?driver ?loc))
        (at start (empty ?truck)))
  :effect
   (and (at start (not (at ?driver ?loc)))
        (at end (driving ?driver ?truck))
        (at start (not (empty ?truck)))))

(:durative-action DISEMBARK-TRUCK
  :parameters
   (?driver - driver
    ?truck - truck
    ?loc - location)
  :duration (= ?duration 1)
  :condition
   (and (over all (at ?truck ?loc))
        (at start (driving ?driver ?truck)))
  :effect
   (and (at start (not (driving ?driver ?truck)))
    (at end (at ?driver ?loc)) (at end (empty ?truck))))

(:durative-action DRIVE-TRUCK
  :parameters
   (?truck - truck
    ?loc-from - location
    ?loc-to - location
    ?driver - driver)
  :duration (= ?duration 10)
  :condition
   (and (at start (at ?truck ?loc-from))
   (over all (driving ?driver ?truck))
   (at start (link ?loc-from ?loc-to)))
  :effect
   (and (at start (not (at ?truck ?loc-from)))
    (at end (at ?truck ?loc-to))))

(:durative-action WALK
  :parameters
   (?driver - driver
    ?loc-from - location
    ?loc-to - location)
  :duration (= ?duration 20)
  :condition
   (and (at start (at ?driver ?loc-from))
    (at start (path ?loc-from ?loc-to)))
  :effect
   (and (at start (not (at ?driver ?loc-from)))
    (at end (at ?driver ?loc-to))))
)
```

B.2.3 Satellite

The satellite domain extracted from the IPC2002:

```
(define (domain satellite)
 (:requirements :strips :equality :typing :durative-actions)
 (:types satellite direction instrument mode)
 (:predicates
           (on_board ?i - instrument ?s - satellite)
           (supports ?i - instrument ?m - mode)
           (pointing ?s - satellite ?d - direction)
           (power_avail ?s - satellite)
           (power_on ?i - instrument)
           (calibrated ?i - instrument)
           (have_image ?d - direction ?m - mode)
           (calibration_target ?i - instrument ?d - direction))

 (:durative-action turn_to
   :parameters (?s - satellite ?d_new - direction
                ?d_prev - direction)
   :duration (= ?duration 5)
```

```
  :condition (and (at start (pointing ?s ?d_prev))
                  (over all (not (= ?d_new ?d_prev))))
  :effect (and  (at end (pointing ?s ?d_new))
                (at start (not (pointing ?s ?d_prev))))
)
(:durative-action switch_on
  :parameters (?i - instrument ?s - satellite)
  :duration (= ?duration 2)
  :condition (and (over all (on_board ?i ?s))
                  (at start (power_avail ?s)))
  :effect (and (at end (power_on ?i))
               (at start (not (calibrated ?i)))
               (at start (not (power_avail ?s))))
)
(:durative-action switch_off
  :parameters (?i - instrument ?s - satellite)
  :duration (= ?duration 1)
  :condition (and (over all (on_board ?i ?s))
                     (at start (power_on ?i)))
  :effect (and (at start (not (power_on ?i)))
               (at end (power_avail ?s)))
)

(:durative-action calibrate
  :parameters (?s - satellite ?i - instrument ?d - direction)
  :duration (= ?duration 5)
  :condition (and (over all (on_board ?i ?s))
                  (over all (calibration_target ?i ?d))
```

```
                    (at start (pointing ?s ?d))
                    (over all (power_on ?i))
                    (at end (power_on ?i)))
   :effect (at end (calibrated ?i))
 )
 (:durative-action take_image
   :parameters (?s - satellite ?d - direction
                ?i - instrument ?m - mode)
   :duration (= ?duration 7)
   :condition (and (over all (calibrated ?i))
                    (over all (on_board ?i ?s))
                    (over all (supports ?i ?m) )
                    (over all (power_on ?i))
                    (over all (pointing ?s ?d))
                    (at end (power_on ?i)))
   :effect (at end (have_image ?d ?m))
 )
)
```

B.2.4 Rovers

Domain ***Rovers*** is extracted from IPC2002:

```
(define (domain Rover)
 (:requirements :typing :durative-actions)
 (:types rover waypoint store camera mode lander objective)

 (:predicates (at ?x - rover ?y - waypoint)
              (at_lander ?x - lander ?y - waypoint)
```

```
             (can_traverse ?r - rover ?x - waypoint ?y - waypoint)
             (equipped_for_soil_analysis ?r - rover)
             (equipped_for_rock_analysis ?r - rover)
             (equipped_for_imaging ?r - rover)
             (empty ?s - store)
             (have_rock_analysis ?r - rover ?w - waypoint)
             (have_soil_analysis ?r - rover ?w - waypoint)
             (full ?s - store)
             (calibrated ?c - camera ?r - rover)
             (supports ?c - camera ?m - mode)
             (available ?r - rover)
             (visible ?w - waypoint ?p - waypoint)
             (have_image ?r - rover ?o - objective ?m - mode)
             (communicated_soil_data ?w - waypoint)
             (communicated_rock_data ?w - waypoint)
             (communicated_image_data ?o - objective ?m - mode)
             (at_soil_sample ?w - waypoint)
             (at_rock_sample ?w - waypoint)
             (visible_from ?o - objective ?w - waypoint)
             (store_of ?s - store ?r - rover)
             (calibration_target ?i - camera ?o - objective)
             (on_board ?i - camera ?r - rover)
             (channel_free ?l - lander))

(:durative-action navigate
 :parameters (?x - rover ?y - waypoint ?z - waypoint)
 :duration (= ?duration 5)
 :condition (and (over all (can_traverse ?x ?y ?z))
                 (at start (available ?x))
                 (at start (at ?x ?y))
                 (over all (visible ?y ?z)))
 :effect (and (at start (not (at ?x ?y)))
              (at end (at ?x ?z)))
)
(:durative-action sample_soil
 :parameters (?x - rover ?s - store ?p - waypoint)
 :duration (= ?duration 10)
 :condition (and (over all (at ?x ?p)) (at start (at ?x ?p))
                 (at start (at_soil_sample ?p))
                 (at start (equipped_for_soil_analysis ?x))
                 (at start (store_of ?s ?x))
                 (at start (empty ?s)))
 :effect (and (at start (not (empty ?s)))
              (at end (full ?s))
              (at end (have_soil_analysis ?x ?p))
              (at end (not (at_soil_sample ?p))))
)
(:durative-action sample_rock
 :parameters (?x - rover ?s - store ?p - waypoint)
 :duration (= ?duration 8)
 :condition (and (over all (at ?x ?p))
                 (at start (at ?x ?p))
                 (at start (at_rock_sample ?p))
                 (at start (equipped_for_rock_analysis ?x))
                 (at start (store_of ?s ?x))
                 (at start (empty ?s)))
 :effect (and (at start (not (empty ?s)))
              (at end (full ?s))
              (at end (have_rock_analysis ?x ?p))
              (at end (not (at_rock_sample ?p))))
)
(:durative-action drop
 :parameters (?x - rover ?y - store)
 :duration (= ?duration 1)
 :condition (and (at start (store_of ?y ?x))
                 (at start (full ?y)))
 :effect (and (at end (not (full ?y)))
              (at end (empty ?y)))
)

(:durative-action calibrate
 :parameters (?r - rover ?i - camera
              ?t - objective ?w - waypoint)
 :duration (= ?duration 5)
 :condition (and (at start (equipped_for_imaging ?r))
                 (at start (calibration_target ?i ?t))
                 (over all (at ?r ?w))
                 (at start (visible_from ?t ?w))
                 (at start (on_board ?i ?r)))
 :effect (at end (calibrated ?i ?r))
)
(:durative-action take_image
 :parameters (?r - rover ?p - waypoint ?o - objective
              ?i - camera ?m - mode)
 :duration (= ?duration 7)
 :condition (and (over all (calibrated ?i ?r))
                 (at start (on_board ?i ?r))
                 (over all (equipped_for_imaging ?r))
                 (over all (supports ?i ?m) )
                 (over all (visible_from ?o ?p))
                 (over all (at ?r ?p)))
 :effect (and (at end (have_image ?r ?o ?m))
              (at end (not (calibrated ?i ?r))))
)
(:durative-action communicate_soil_data
 :parameters (?r - rover ?l - lander ?p - waypoint
              ?x - waypoint ?y - waypoint)
 :duration (= ?duration 10)
 :condition (and (over all (at ?r ?x))
                 (over all (at_lander ?l ?y))
                 (at start (have_soil_analysis ?r ?p))
                 (at start (visible ?x ?y))
                 (at start (available ?r))
                 (at start (channel_free ?l)))
 :effect (and (at start (not (available ?r)))
              (at start (not (channel_free ?l)))
              (at end (channel_free ?l))
              (at end (communicated_soil_data ?p))
              (at end (available ?r)))
)

(:durative-action communicate_rock_data
 :parameters (?r - rover ?l - lander ?p - waypoint
              ?x - waypoint ?y - waypoint)
 :duration (= ?duration 10)
 :condition (and (over all (at ?r ?x))
                 (over all (at_lander ?l ?y))
                 (at start (have_rock_analysis ?r ?p))
                 (at start (visible ?x ?y))
                 (at start (available ?r))
                 (at start (channel_free ?l)))
 :effect (and (at start (not (available ?r)))
              (at start (not (channel_free ?l)))
              (at end (channel_free ?l))
              (at end (communicated_rock_data ?p))
              (at end (available ?r)))
)
(:durative-action communicate_image_data
 :parameters (?r - rover ?l - lander ?o - objective ?m - mode
              ?x - waypoint ?y - waypoint)
 :duration (= ?duration 15)
 :condition (and (over all (at ?r ?x))
                 (over all (at_lander ?l ?y))
                 (at start (have_image ?r ?o ?m))
                 (at start (visible ?x ?y))
                 (at start (available ?r))
                 (at start (channel_free ?l)))
 :effect (and (at start (not (available ?r)))
              (at start (not (channel_free ?l)))
              (at end (channel_free ?l))
```

```
                (at end (communicated_image_data ?o ?m))
                (at end (available ?r)))
 )
)
```

B.2.5 ZenoTravel

Domain ***ZenoTravel*** is extracted from the IPC2002:

```
(define (domain zeno-travel)
 (:requirements :durative-actions :typing)
 (:types aircraft person city flevel - object)
 (:predicates (at ?x - person ?c - city)
             (at ?a - aircraft ?c - city)
             (in ?p - person ?a - aircraft)
             (fuel-level ?a - aircraft ?l - flevel)
             (next ?l1 ?l2 - flevel))

 (:durative-action board
  :parameters (?p - person ?a - aircraft ?c - city)
  :duration (= ?duration 20)
  :condition (and (at start (at ?p ?c))
                  (over all (at ?a ?c)))
  :effect (and (at start (not (at ?p ?c)))
              (at end (in ?p ?a))))

 (:durative-action debark
  :parameters (?p - person ?a - aircraft ?c - city)
  :duration (= ?duration 30)
  :condition (and (at start (in ?p ?a))
                  (over all (at ?a ?c)))
  :effect (and (at start (not (in ?p ?a)))
              (at end (at ?p ?c))))

 (:durative-action fly
  :parameters (?a - aircraft ?c1 ?c2 - city
              ?l1 ?l2 - flevel)
  :duration (= ?duration 180)
  :condition (and (at start (at ?a ?c1))
                  (at start (fuel-level ?a ?l1))
                  (at start (next ?l2 ?l1)))
  :effect (and (at start (not (at ?a ?c1)))
              (at end (at ?a ?c2))
              (at end (not (fuel-level ?a ?l1)))
              (at end (fuel-level ?a ?l2))))


 (:durative-action zoom
  :parameters (?a - aircraft ?c1 ?c2 - city
              ?l1 ?l2 ?l3 - flevel)
  :duration (= ?duration 100)
  :condition (and (at start (at ?a ?c1))
                  (at start (fuel-level ?a ?l1))
                  (at start (next ?l2 ?l1))
                  (at start (next ?l3 ?l2)))
  :effect (and (at start (not (at ?a ?c1)))
              (at end (at ?a ?c2))
              (at end (not (fuel-level ?a ?l1)))
```

```
                (at end (fuel-level ?a ?l3))))

  (:durative-action refuel
   :parameters (?a - aircraft ?c - city
                ?l - flevel ?l1 - flevel)
   :duration (= ?duration 73)
   :condition (and (at start (fuel-level ?a ?l))
                   (at start (next ?l ?l1))
                   (over all (at ?a ?c)))
   :effect (and (at end (fuel-level ?a ?l1))
                (at end (not (fuel-level ?a ?l)))))
)
```

B.3 Airport domain extracted from IPC2004

Domain ***airport*** is extracted from the International Planning Competition in 2004. Since the detail of the domain description is too big to fit into this appendix, only important parts are extracted and illustrated here:

```
(define (domain airport_fixed_structure)
 (:requirements :durative-actions :typing)
 (:types airplane segment direction airplanetype)
 (:constants
      north south - direction
      light medium heavy - airplanetype
      seg_pp_0_60 seg_ppdoor_0_40
      seg_tww1_0_200 seg_twe1_0_200
      seg_tww2_0_50 seg_tww3_0_50
      seg_tww4_0_50 seg_rww_0_50
```

```
     seg_rwtw1_0_10 seg_rw_0_400
     seg_rwe_0_50 seg_twe4_0_50
     seg_rwte1_0_10 seg_twe3_0_50
     seg_twe2_0_50 seg_rwte2_0_10
     seg_rwtw2_0_10 - segment
     airplane_CFBEG - airplane)
(:predicates
     ;; airport information
     (has-type ?a - airplane ?t - airplanetype)
     ;; plane a has type t
     (at-segment ?a - airplane ?s - segment)
     ;; planes are at segments,
     ;; ie at their end in driving direction
     (facing ?a - airplane ?d - direction)
     ;; planes face into their driving direction
     ;; how the planes affect the airport
     (occupied ?s - segment)
     ;; a plane is in here
     (not_occupied ?s - segment)
     (blocked ?s - segment ?a - airplane)
     ;; segment s is blocked if it is endangered by plane p
     (not_blocked ?s - segment ?a - airplane)
     ;; an airplane may lineup on segment s when facing d
     (is-start-runway ?s - segment ?d - direction)
     ;; airplane a is starting from runway s
     (airborne ?a - airplane ?s - segment)
     (is-moving ?a - airplane)
     (is-pushing ?a - airplane)
     (is-parked ?a - airplane ?s - segment))
(:functions
     ;; the length of a segment
     (length ?s - segment)
     ;; the number of engines of an airplane
     (engines ?a - airplane))

(:durative-action move_seg_pp_0_60_
                  seg_ppdoor_0_40_north_north_medium
 :parameters    (?a - airplane)
 :duration
       ;; length of the segment divided through
       ;; the speed of an airplane (fixed 30 m/s)
       (= ?duration 2)
 ...
)
(:durative-action move_seg_ppdoor_0_40_
                  seg_tww1_0_200_north_south_medium
 :parameters    (?a - airplane)
 :duration
     ;; length of the segment divided through
     ;; the speed of an airplane (fixed 30 m/s)
     (= ?duration 1)
 ...
)
(:durative-action move_seg_tww1_0_200_
                  seg_twe1_0_200_north_south_medium
 :parameters (?a - airplane)
 :duration
       ;; length of the segment divided through
       the speed of an airplane (fixed 30 m/s)
       (= ?duration 6)
 ...
)

(:durative-action move_seg_twe1_0_200_
                  seg_twe2_0_50_south_south_medium
 :parameters    (?a - airplane)
 :duration
   ;; length of the segment divided through
   ;; the speed of an airplane (fixed 30 m/s)
   (= ?duration 6)
   ...
)
(:durative-action move_seg_twe2_0_50_
                  seg_twe3_0_50_south_south_medium
 :parameters    (?a - airplane)
 :duration
   ;; length of the segment divided through
   ;; the speed of an airplane (fixed 30 m/s)
   (= ?duration 1)
 ...
)
(:durative-action move_seg_twe3_0_50_
                  seg_twe4_0_50_south_south_medium
 :parameters    (?a - airplane)
 :duration
   ;; length of the segment divided through
   ;; the speed of an airplane (fixed 30 m/s)
   (= ?duration 1)
 ...
)
(:durative-action move_seg_twe4_0_50_
                  seg_rwe_0_50_south_south_medium
 :parameters    (?a - airplane)
 :duration
   ;; length of the segment divided through
   ;; the speed of an airplane (fixed 30 m/s)
   (= ?duration 1)
 ...
)

(:durative-action move_seg_rwe_0_50_
                  seg_rw_0_400_south_south_medium
 :parameters    (?a - airplane)
 :duration
   ;; length of the segment divided through
   ;; the speed of an airplane (fixed 30 m/s)
   (= ?duration 1)
 ...
)
(:durative-action move_seg_rw_0_400_
                  seg_rww_0_50_south_south_medium
 :parameters    (?a - airplane)
 :duration
   ;; length of the segment divided through
   ;; the speed of an airplane (fixed 30 m/s)
   (= ?duration 13)
 ...
)
(:durative-action move_seg_rww_0_50_
                  seg_tww4_0_50_south_north_medium
 :parameters    (?a - airplane)
 :duration
   ;; length of the segment divided through
   ;; the speed of an airplane (fixed 30 m/s)
   (= ?duration 1)
 ...
)
(:durative-action move_seg_tww4_0_50_
                  seg_tww3_0_50_north_north_medium
 :parameters    (?a - airplane)
 :duration
   ;; length of the segment divided through
   ;; the speed of an airplane (fixed 30 m/s)
            (= ?duration 1)
 ...
)
(:durative-action move_seg_tww3_0_50_
                  seg_tww2_0_50_north_north_medium
 :parameters    (?a - airplane)
 :duration
```

```
    ;; length of the segment divided through
    ;; the speed of an airplane (fixed 30 m/s)
    (= ?duration 1)
  ...
)
(:durative-action move_seg_tww2_0_50_
                  seg_tww1_0_200_north_north_medium
  :parameters    (?a - airplane)
  :duration
    ;; length of the segment divided through
    ;; the speed of an airplane (fixed 30 m/s)
    (= ?duration 1)
  ...
)

(:durative-action move_seg_tww1_0_200_
                  seg_ppdoor_0_40_north_south_medium
  :parameters    (?a - airplane)
  :duration
    ;; length of the segment divided through
    ;; the speed of an airplane (fixed 30 m/s)
    (= ?duration 6)
  ...
)
(:durative-action move_seg_ppdoor_0_40_
                  seg_pp_0_60_south_south_medium
  :parameters    (?a - airplane)
  :duration
```

```
   ;; length of the segment divided through
   ;; the speed of an airplane (fixed 30 m/s)
   (= ?duration 1)
 ...
)

(:durative-action takeoff_seg_rww_0_50_north
 :parameters (?a - airplane)
 :duration  (= ?duration 30 )
 ...
)
(:durative-action takeoff_seg_rwe_0_50_south
 :parameters (?a - airplane)
 :duration  (= ?duration 30 )
 ...
)
(:durative-action park_seg_pp_0_60_north
 :parameters (?a - airplane)
 :duration (= ?duration 40)
 ...
)
(:durative-action park_seg_pp_0_60_south
 :parameters (?a - airplane)
 :duration (= ?duration 40)
 ...
)
(:durative-action startup_seg_pp_0_60_north_medium
 :parameters (?a - airplane)
 :duration  (= ?duration (* 60 (engines ?a) ) )
 ...
)
(:durative-action startup_seg_pp_0_60_south_medium
 :parameters (?a - airplane)
 :duration  (= ?duration (* 60 (engines ?a) ) )
 ...
)
(:durative-action startup_seg_ppdoor_0_40_north_medium
 :parameters (?a - airplane)
 :duration  (= ?duration (* 60 (engines ?a) ) )
 ...
)
(:durative-action startup_seg_ppdoor_0_40_south_medium
 :parameters (?a - airplane)
 :duration  (= ?duration (* 60 (engines ?a) ) )
 ...
)

(:durative-action startup_seg_tww1_0_200_north_medium
 :parameters (?a - airplane)
 :duration  (= ?duration (* 60 (engines ?a) ) )
 ...
)
(:durative-action startup_seg_tww1_0_200_south_medium
 :parameters (?a - airplane)
 :duration  (= ?duration (* 60 (engines ?a) ) )
 ...
```

```
)
(:durative-action startup_seg_twe1_0_200_north_medium
 :parameters (?a - airplane)
 :duration  (= ?duration (* 60 (engines ?a) ) )
 ...
)
(:durative-action startup_seg_twe1_0_200_south_medium
 :parameters (?a - airplane)
 :duration  (= ?duration (* 60 (engines ?a) ) )
 ...
)
(:durative-action startup_seg_tww2_0_50_north_medium
 :parameters (?a - airplane)
 :duration  (= ?duration (* 60 (engines ?a) ) )
 ...
)
(:durative-action startup_seg_tww2_0_50_south_medium
 :parameters (?a - airplane)
 :duration  (= ?duration (* 60 (engines ?a) ) )
 ...
)
(:durative-action startup_seg_tww3_0_50_north_medium
 :parameters (?a - airplane)
 :duration  (= ?duration (* 60 (engines ?a) ) )
 ...
)
(:durative-action startup_seg_tww3_0_50_south_medium
 :parameters (?a - airplane)
 :duration  (= ?duration (* 60 (engines ?a) ) )
 ...
)
(:durative-action startup_seg_tww4_0_50_north_medium
 :parameters (?a - airplane)
 :duration  (= ?duration (* 60 (engines ?a) ) )
 ...
)
(:durative-action startup_seg_tww4_0_50_south_medium
 :parameters (?a - airplane)
 :duration  (= ?duration (* 60 (engines ?a) ) )
 ...
)

(:durative-action startup_seg_twe4_0_50_north_medium
 :parameters (?a - airplane)
 :duration  (= ?duration (* 60 (engines ?a) ) )
 ...
)
(:durative-action startup_seg_twe4_0_50_south_medium
 :parameters (?a - airplane)
 :duration  (= ?duration (* 60 (engines ?a) ) )
 ...
)
(:durative-action startup_seg_twe3_0_50_north_medium
 :parameters (?a - airplane)
 :duration  (= ?duration (* 60 (engines ?a) ) )
 ...
```

```
  )
  (:durative-action startup_seg_twe3_0_50_south_medium
   :parameters (?a - airplane)
   :duration  (= ?duration (* 60 (engines ?a) ) )
   ...
  )
  (:durative-action startup_seg_twe2_0_50_north_medium
   :parameters (?a - airplane)
   :duration  (= ?duration (* 60 (engines ?a) ) )
   ...
  )
  (:durative-action startup_seg_twe2_0_50_south_medium
   :parameters (?a - airplane)
   :duration  (= ?duration (* 60 (engines ?a) ) )
   ...
  )
)
```

B.4 Domain storage extracted from IPC2006

The temporal part of domain ***storage*** is extracted from the International Planning Competition 2006:

```
(define (domain Storage-Time)
 (:requirements :typing :durative-actions)
 (:types hoist surface place area - object
         container depot - place
         storearea transitarea - area
         area crate - surface)
```

```
(:predicates (clear ?s - storearea)
             (in ?x - (either storearea crate) ?p - place)
             (available ?h - hoist)
             (lifting ?h - hoist ?c - crate)
             (at ?h - hoist ?a - area)
             (on ?c - crate ?s - storearea)
             (connected ?a1 ?a2 - area))

(:durative-action lift
 :parameters (?h - hoist ?c - crate
              ?a1 - storearea ?a2 - area ?p - place)
 :duration (= ?duration 2)
 :condition (and (at start (available ?h))
                 (at start (on ?c ?a1))
                 (over all (connected ?a1 ?a2))
                 (over all (at ?h ?a2))
                 (over all (in ?a1 ?p)))
 :effect (and (at start (not (in ?c ?p)))
              (at start (not (available ?h)))
              (at start (lifting ?h ?c))
              (at start (not (on ?c ?a1)))
              (at end (clear ?a1)))
)
(:durative-action drop
 :parameters (?h - hoist ?c - crate ?a1 - storearea
              ?a2 - area ?p - place)
 :duration (= ?duration 2)
 :condition (and (at start (clear ?a1))
                  (over all (lifting ?h ?c))
                  (over all (connected ?a1 ?a2))
                  (over all (at ?h ?a2))
                  (over all (in ?a1 ?p)))
 :effect (and (at start (not (clear ?a1)))
               (at end (not (lifting ?h ?c)))
               (at end (available ?h))
               (at end (on ?c ?a1)) (at end (in ?c ?p)))
)
(:durative-action move
 :parameters (?h - hoist ?from ?to - storearea)
 :duration (= ?duration 1)
 :condition (and (at start (at ?h ?from))
                  (at start (clear ?to))
                  (over all (connected ?from ?to)))
 :effect (and (at start (not (at ?h ?from)))
               (at start (not (clear ?to)))
               (at start (clear ?from))
               (at end (at ?h ?to)))
)

(:durative-action go-out
 :parameters (?h - hoist ?from - storearea ?to - transitarea)
 :duration (= ?duration 1)
 :condition (and (at start (at ?h ?from))
                  (over all (connected ?from ?to)))
 :effect (and (at start (not (at ?h ?from)))
               (at start (clear ?from)) (at end (at ?h ?to)))
)
(:durative-action go-in
 :parameters (?h - hoist ?from - transitarea ?to - storearea)
 :duration (= ?duration 1)
 :condition (and (at start (at ?h ?from)) (at start (clear ?to))
                  (over all (connected ?from ?to)))
 :effect (and (at start (not (at ?h ?from)))
               (at start (not (clear ?to)))
               (at end (at ?h ?to))))
)
```

B.5 *Domain* spacecraft *with intermediate effects*

The *spacecraft* domain contains an action *turn* which has intermediate effects. The details of this domain are as follows:

```
(:durative-action turn
 :parameters (?cur ?dest - object ?h - heading)
 :duration (= ?duration (+ (/ (angle ?cur ?dest) (turning-rate))) (h
 :condition (and (at start (pointing ?cur))
                 (at start (>= propellant
                            (/ propellant-require 2)))
                 (at start (not (controller-in-use)))
                 (at (- endtime RCS-duration)
                     (>= propellant (/ propellant$-$require 2)))
                 (at (- endtime RCS-duration)
                     (not controller-in-use)))
 :effect (and (at start (not (pointing ?cur)))
              (at start (decrease propellant
                            (/ propellant-require 2)))
              (at (starttime) (controller-in-use))
              (at (+ starttime RCS-duration) (not (controller-in-use
              (at (starttime) (vibration))
              (at (+ starttime RCS-duration) (not (vibration)))
              (at (- endtime RCS-duration)
                  (decrease propellant (/ propellant-require 2)))
              (at (- endtime RCS-duration) (controller-in-use))
              (at (endtime) (not (controller-in-use)))
              (at (- endtime RCS-duration) (vibration))
              (at (endtime) (not (vibration)))
              (at end (pointing ?dest ?h)))
  )
```