Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's Thesis Nr. 5

Mixed-Integer Linear Programming Applied to Temporal Planning of Concurrent Actions

Florentin Mehlbeer

Course of Study:	Informatik
Examiner:	Prof. Dr. rer. nat. Marc Toussaint
Supervisor:	Ngo Anh Vien, Ph.D.
Commenced:	May 12, 2014
Completed:	November 11, 2014
CR-Classification:	I.2.8

Kurzfassung

Auf Grund ihrer vielfältigen Anwendungsmöglichkeiten gewinnen autonome Systeme zunehmend an Bedeutung. Entsprechend besteht großes Interesse an effizienten Verfahren für das automatische Planen. Das Planen mit STRIPS-ähnlichen Operatoren ist ein kombinatorisches Problem. Die gemischt-ganzzahlige Optimierung kann effektiv zur Lösung von Problemen dieser Art eingesetzt werden. In dieser Arbeit werden allgemeine, vom konkreten Anwendungsfall unabhängige Formulierungen gemischt-ganzzahliger linearer Programme für das temporale Planen nebenläufiger Aktionen und eine Verallgemeinerung des in Graphplan verwendeten Planungsgraphen vorgestellt. Gemeinsam werden sie zur Bestimmung von Plänen minimaler Dauer angewendet. Dabei darf die Dauer von Aktionen reellwertig sein. Ein Vergleich mit Temporal Graphplan basierend auf experimentell ermittelten Perfomanzdaten liefert erfolgversprechende Resultate.

Abstract

Due to the wide range of applications, the relevance of autonomous systems is growing. Hence, there is great interest in efficient automated planning methods. Planning of STRIPS-like operators is a combinatorial problem. Mixed-Integer Programming is a powerful tool for modeling and solving problems of this type. In this thesis domain-independent Mixed-Integer Linear Programming formulations for temporal planning of concurrent actions and a generalization of Graphplan's planning graph are presented. In combination they are applied to compute plans, that are optimal with respect to their duration. The method can handle actions of real-valued duration. A comparison with Temporal Graphplan based on experimental performance data yields promising results.

Contents

1 Introduction 9

2 Graphplan 11
- 2.1 The Planning Scenario . . . 11
- 2.2 The Planning Graph . . . 14
- 2.3 The Algorithm . . . 17

3 Planning via Mixed-Integer Linear Programming 23
- 3.1 Mixed-Integer Linear Programs . . . 23
- 3.2 Branch-and-Bound . . . 24
- 3.3 Solving Planning Problems via Mixed-Integer Linear Programming . . . 29
- 3.4 A Formulation for Non-temporal Planning of Concurrent Actions . . . 30

4 Temporal Planning via Mixed-Integer Linear Programming 35
- 4.1 The Temporal Planning Scenario . . . 35
- 4.2 The Temporal Planning Graph . . . 38
- 4.3 Temporal SATPLAN-based Formulations . . . 40
- 4.4 Temporal Level Off . . . 51

5 A State-change Approach to Temporal Planning 55
- 5.1 The Five Types of State-change Variables . . . 55
- 5.2 Including State-change Variables in the Temporal Planning Graph . . . 56
- 5.3 The State-change Formulation . . . 56

6 Real-valued Durations and Dynamic Graph Expansion 63

7 Comparison 67
- 7.1 Implementation . . . 67
- 7.2 Empirical Comparison . . . 68

8 Summary 73

A Notation 75

B Temporal Planning Domains 77

List of Figures

2.1 Exemplary planning graph 16

3.1 Exemplary Branch-and-Bound tree 25

4.1 Exemplary temporal planning graph 41
4.2 Temporal planning graph for the counter example 53

List of Tables

7.1 Experimental results 70

List of Listings

2.1 The airplane domain 13

4.1 Exemplary TSTRIPS-operator 35
4.2 Affecting actions 36
4.3 Domain of the counter example 52

B.1 The temporal box assembly domain 78
B.2 The temporal blocks world domain 79

B.3 The temporal logistics domain 80

List of Algorithms

2.1 Graph expansion 18
2.2 Solution extraction 20

3.1 Branch-and-Bound 26

4.1 Temporal graph expansion 42

5.1 Temporal Graph Expansion with State-change Variables 57

6.1 Dynamic graph expansion 65

1 Introduction

Planning is an important skill. In this world every higher animal has to plan activities at some point in its live. Lions have to plan how to catch their prey and monkeys how to reach the branch of a tree. But in particular human beings have to plan frequently. Whether it is simple tasks like preparing your next meal or difficult ones such as planning a large construction project or playing a game of chess against a strong opponent, it is required in many activities. For this reason scientists realized in the last century that a good understanding of planning is key to engineering powerful autonomous systems. In particular in the last few decades the relevance of autonomous systems grew rapidly. The field of application is almost unlimited: automated production, autonomous driving or spaceflight are just a few application examples. Consequently the need for sophisticated planning methods was and is huge. Considerable progress has already been made and different approaches to planning have been developed.
Regarding planning in STRIPS-domains Blum and Furst published a seminal paper on their method *Graphplan* in 1995 [BF95]. It had immense influence on subsequent research in this area. One reason for this is that Graphplan represents a given problem in a very handy planning graph as an intermediate step. The planning graph encodes the knowledge about the problem and can be analyzed in order to compute a plan. Compared to former methods the idea to incorporate the planning graph yielded remarkable speedup. As Graphplan is related to other planning methods such as planning as satisfiability (also called SATPLAN) [KS92] and (Mixed-)Integer Programming applied to planning [VBS99] the planning graph can be incorporated in these approaches as well. While Graphplan incorporates the parallel execution of actions it is limited to the case in which all actions have unit duration. Therefore Smith and Weld [SW99] generalized it to the temporal case in which the durations of actions can vary. Analogously SATPLAN was generalized to the temporal case [ML06]. Regarding mathematical programming Dimopoulos and Gerevini developed a Mixed-Integer Linear Program (MILP) for approximate temporal planning [DG02]. That is their method can deal with temporal problems but it does not guarantee that an optimal plan is found with respect to its duration. In this thesis we apply Mixed-Integer Linear Programming to compute optimal plans for temporal planning problems. That is the generated plans are of minimal duration. The durations of actions may even be real-valued. For this purpose we introduce the reader to Graphplan and the Branch-and-Bound method which is the most common solution method applied to MILPs. After that we describe the temporal planning scenario we consider, generalize the planning graph and present three different formulations. Finally we compare the formulations presented, to each other and Temporal Graphplan experimentally. The author's contribution

are in particular the generalization of the planning graph in this form and its algorithmic construction, the three MILP-formulations, and the experimental evaluation.

Outline

This thesis is structured in the following way:

Chapter 2 – Graphplan: In this chapter the planning algorithm Graphplan is introduced. In particular the STRIPS-like domain Graphplan can deal with, the planning graph it uses, the two phases graph expansion and solution extraction as well as a termination test are described.

Chapter 3 – Planning via Mixed-Integer Linear Programming: In this chapter the basics of planning via Mixed-Integer Linear Programming are presented. First a definition for Mixed-Integer Linear Programs (MILPs) is given and the Branch-and-Bound procedure applied to solve them is explained. After that we show how non-temporal planning problems can be solved applying these techniques.

Chapter 4 – Temporal Planning via Mixed-Integer Linear Programming: Here we explain how Mixed-Integer Linear Programming can be applied to solve temporal planning problems. First, we introduce the temporal planning scenario assuming integer durations for actions. After that, we explain the structure of the generalized temporal planning graph and how it is constructed. Finally, we introduce two SATPLAN-based formulations that model the temporality of operators.

Chapter 5 – A State-change Approach to Temporal Planning: In this part of the thesis we introduce a second type of MILP-formulation called the state-change formulation. Moreover we explain how the temporal planning graph has to be modified to be able to derive efficient formulations of this kind.

Chapter 6 – Real-valued Durations and Dynamic Graph Expansion: In this chapter it is explained how we can plan with real-valued durations and further increase the efficiency of our method.

Chapter 7 – Comparison: In this chapter we compare the MILP-formulations introduced in this thesis with each other experimentally. Furthermore we compare the implementation of our approach to an implementation of the Temporal Graphplan algorithm.

Chapter 8 – Summary: In this part we summarize the content of this thesis and give an overview on possible improvements and potential future work.

2 Graphplan

Graphplan is a planning method which had immense influence on subsequent research in particular in the fields of planning and scheduling [BF95]. Graphplan builds a *planning graph* and determines plans by analyzing it. The main reason for its success was the introduction of this very handy planning graph that encodes a given problem in a practicable way. For instance common algorithms can be applied on it in order to extract information from it and/or modify it. Additionally the size of the graph is small compared to graphs modeling the state-space of a problem directly. It can be constructed in polynomial time and hence requires only polynomial space. Further neat properties of Graphplan are its speed compared to previously published methods and the ability to terminate on unsolvable problems. For the latter a termination test exists. Moreover it allows to transform planning problems into constraint satisfaction problems and mixed integer programs as well as factor graphs. Thus it connects different approaches to planning problems.
In the following sections we explain the basics of Graphplan. That is we first take a look at the planning scenario we will be dealing with. Following this, we will see how the planning graph looks like, how it is constructed and exploited to compute plans. This chapter's content is based on [BF95] but at some points our version differs slightly. Algorithm 2.2 is a result of own work formalizing the explanations in the paper mentioned above.

2.1 The Planning Scenario

In the scenario we are going to consider a problem is given by its domain, that is a set of predicates $\mathcal{P}$ and a set of STRIPS-operators Op, together with a set of objects $\mathcal{C}$ as well as an initial condition $\mathcal{I}$ and a problem goal $\mathcal{G}$. By grounding/instantiating the predicates, that is replacing the variables of predicates with objects, we get a set of predicate instances $\hat{\mathcal{P}}$. By further distinguishing the values *true* and *false* of the predicate instances we get a set of literals $\mathcal{L}$. Instead of the term literals we will refer to *propositions* from now on as common in the planning community. As opposed to the common case we allow both negative and positive literals/propositions as just described. Furthermore we introduce the notion of abstract literals/propositions which are parametrized propositions. We denote the set of abstract propositions by $\mathcal{L}_\theta$. The difference between predicates and abstract propositions is that the latter correspond to a value *true* or *false*. The initial condition and the problem goal which we are going to call *goal* from now on are sets of propositions, that is $\mathcal{I} \subseteq \mathcal{L}$ and $\mathcal{G} \subseteq \mathcal{L}$. Initial

condition and goal represent conjunctions of propositions that initially hold or have to old in the end to succeed respectively. Each operator has a parametrized set of preconditions pre_{op} and effects eff_{op} which represent conjunctions of propositions. We always assume that the set of operators contains one so called *no-op* per abstract proposition. A no-op has exactly one precondition and one effect which equal the proposition it corresponds to. Applying a no-op means that the truth value of a predicate instance is simply propagated forward. We get the set of actions, that is instantiated operators, $\mathcal{A}$ by replacing the parameters of STRIPS-operators with objects. For now we assume that all operators and corresponding actions have the same unit duration (duration $= 1$ without loss of generality).
An exemplary *airplane domain* is illustrated in listing 2.1. It was derived from a test domain considered by Smith *et al.* [SW99]. The airplane domain models a logistics problem which is a typical field of application for planners such as Graphplan. The predicates are given by `packet(c1)`, `at(c1, c2)` etc. where

$$\texttt{packet(x)} = \begin{cases} 1 & \text{if x is a packet} \\ 0 & \text{else} \end{cases}$$

and

$$\texttt{in(x, y)} = \begin{cases} 1 & \text{if packet x is in airplane y} \\ 0 & \text{else} \end{cases}$$

etc.. An example for a predicate instance of `in(x, y)` is $\hat{\texttt{in}}$`(P, F)` where `P` is a packet and `F` is an airplane. Usually there are many possible predicate instances per predicate. By relating the truth values *true* and *false* to the mentioned predicate instance we get two propositions, namely the positive proposition $+$`in(P, F)` and the negative proposition $-$`in(P, F)`. If we do not provide the sign of a proposition $l \in \mathcal{L}$ or an abstract proposition $l_\theta \in \mathcal{L}_\theta$ it can denote either a positive or a negative proposition. Moreover we denote the negation of a proposition l by $\neg l$. Furthermore we have got three operators in the domain. One for loading a packet, one for unloading a packet and one for flying from one airport to another. In order to get a concrete problem for this domain we can introduce some airports, airplanes and packets as objects. Which type(s) an object has is defined by the defining propositions in the initial condition. For instance if we want object `F` to be an airplane we add the defining proposition $+$`airplane(F)` to the initial condition. Moreover we have to place each object at an initial location which can be achieved by adding the respective $+$`at`- and $+$in-propositions to the initial condition etc.. Similarly we can define the goal of the problem by appending the desired propositions. In the logistics domain the goal is to transport the packets by airplane from certain starting locations to their destinations. This can be achieved by sequencing instantiated `load`-, `unload`- and `fly`-operators. An action can be started at a time t if and only if all of its preconditions hold at this time. Moreover starting an action guarantees that all of its effects hold at time $t + 1$. We will denote an action a's precondition and effect by pre_a and eff_a. By this means a sequence of states at times $1, 2$, etc. given by the propositions holding at the respective times is generated.

Listing 2.1 The airplane domain.

```
domain airplanes {

   predicates { packet(c1), airplane(c1), airport(c1),
                at(c1, c2), in(c1, c2), equal(c1, c2) }

   operators {

      load(c1, c2, c3)
         precondition { +packet(c1), +airplane(c2), +airport(c3),
                        +at(c1, c3), +at(c2, c3) }
         effect       { -at(c1, c3), +in(c1, c2) }

      unload(c1, c2, c3)
         precondition { +packet(c1), +airplane(c2), +airport(c3),
                        +in(c1, c2), +at(c2, c3) }
         effect       { -in(c1, c2), +at(c1, c3) }

      fly(c1, c2, c3)
         precondition { +airplane(c1), +airport(c2), +airport(c3),
                        -equal(c2, c3), +at(c1, c2) }
         effect       { -at(c1, c2), +at(c1, c3) }

   }

}
```

If we reach a state in which all goal propositions are present we have achieved the goal. If an action a has a proposition l as an effect we say that a *adds* l and *deletes* $\neg l$. We remark at this point that adding l implies deleting $\neg l$ and vice versa. Regarding predicate instances we say that a adds $\hat{p}$ if a has $+l(\hat{p})$ among its effects and that it deletes $\hat{p}$ if $-l(\hat{p})$ is an effect of a. Thereby $+l(\hat{p})$ and $-l(\hat{p})$ denote the positive and negative propositions corresponding to $\hat{p}$. As opposed to classical planning scenarios it is allowed Graphplan computes parallel plans that is at each time step may start several actions. Consequently the succeeding state is given by the union of the effects of all the actions started at time t. Allowing parallel execution of actions generally reduces the length of plans: For example we could fly packet `P1` from airport `A1` to airport `B1` in airplane `F1` and packet `P2` from airport `A2` to airport `B2` in airplane `F2` at once rather than in two subsequent steps. However, it is not possible to execute all actions in parallel. For example it would not be possible to load packet `P1` into both `A1` and `A2` since the packet can only be in one place/airplane at a time. Hence we require a notion of *independent* and *interfering* actions that cannot be applied in parallel. There exist different definitions of independence. However, we apply the following definition.

Definition 2.1.1 (Interference and Independence)
Two actions a_1 and a_2 are interfering if and only if either one of them deletes at least one precondition or effect of the other and $a_1 \neq a_2$ or they have preconditions which are negations of each other. They are independent in the opposite case.

If two actions are interfering we also say that they are eternally mutually exclusive (*emutex*). Generally a set of actions $A = \{a_1, \ldots, a_n\}$ is emutex if at least one pair of actions (a_i, a_j), $i \neq j$ is interfering and independent otherwise. As with actions propositions can also be eternally mutex.

Definition 2.1.2 (Eternal Proposition-Proposition Mutual Exclusiveness)
Two propositions l_1 and l_2 are interfering if and only if $l_1 = \neg l_2$ holds.

2.2 The Planning Graph

The planning graph that Graphplan generates is organized in levels. A leveled graph is a special kind of graph whose nodes can be partitioned into disjoint subsets $S^1, \ldots, S^n$ such that edges from nodes in S^i are connected to nodes in S^{i-1} and S^{i+1} only. In the planning graph the levels alternate between proposition levels and action levels containing proposition and action nodes respectively. That is $S^1 = \mathcal{L}^1$, $S^2 = \mathcal{A}^1$, $S^3 = \mathcal{L}^2$, $S^4 = \mathcal{A}^2$, etc. where $\mathcal{L}^t$ denotes a proposition level and $\mathcal{A}^t$ an action level. A proposition level $\mathcal{L}^t$ contains all propositions that could potentially hold at time t. This means in particular that a proposition l and its negation $\neg l$ can be present in the same level. To be able to refer to propositions which are present in different proposition levels $\mathcal{L}^t$ we introduce the time-indexed notation l^t. Moreover to denote the set of all time-indexed propositions or nodes in the planning graph we introduce the notation $\boldsymbol{\mathcal{L}}$. A special proposition level is the first one. It represents the initial condition and contains one node per proposition in the initial condition. Similarly to the proposition levels an action level $\mathcal{A}^t$ consists of all actions that could potentially be executed in period t if we consider only mutexes between pairs of preconditions. Analogously to time-indexed propositions we introduce the notation a^t for time-indexed actions that can be executed in period t and $\boldsymbol{\mathcal{A}}$ for the set of time-indexed actions. There are two basic types of edges connecting preconditions, actions and effects.

- Precondition edges: There is a precondition edge between a proposition node $l^t \in \mathcal{L}^t$ and an action node $a^t \in \mathcal{A}^t$ if and only if l is a precondition of a.
- Effect edges: There is an effect edge between action node $a^t \in \mathcal{A}^t$ and a proposition node $l^{t+1} \in \mathcal{L}^{t+1}$ if and only if l is an effect of a.

In order to guarantee Graphplan's correctness we also have to encode eternal mutexes in the planning graph. However, besides eternal mutexes there exists also the notion of conditional mutexes (*cmutexes*). Cmutexes describe the fact that two actions/propositions can be mutex for some time but the exclusivity ends at some point in time.

Definition 2.2.1 (Conditional Action-Action Mutual Exclusiveness)
Two actions $a_1^t \in \mathcal{A}^t$ and $a_2^t \in \mathcal{A}^t$ are conditionally mutual exclusive if and only if they have competing needs, that is there exists a precondition $l_1^t \in \mathcal{L}^t$ of a_1^t and a precondition $l_2^t \in \mathcal{L}^t$ of a_2^t that are eternally or conditionally mutually exclusive.

This means that conditional mutexes between actions result from emutexes or cmutexes between two of its preconditions. In turn cmutexes and emutexes between actions can cause cmutexes between two of their effects.

Definition 2.2.2 (Conditional Proposition-Proposition Mutual Exclusiveness)
Two propositions $l_1^t \in \mathcal{L}^t$ and $l_2^t \in \mathcal{L}^t$ are conditionally mutual exclusive if they are competing effects, that is all actions that have l_1^t as one of their effects are eternally or conditionally mutual exclusive with all actions having l_2^t as one of their effects.

Although cmutexes are not necessary to ensure the correctness of Graphplan they further reduce the number of propositions and actions in the planning graph because they allow to exclude propositions and actions that cannot be present/started in the respective graph level. Furthermore they can speed up the backward-chaining search significantly as we will explain later on in this chapter. In the future we will not distinguish between cmutexes and emutexes any longer because their effect on the planning problem is equivalent. In the graph mutexes are represented by edges. Therefore we introduce two new types of edges, namely mutex-edges between two propositions and two actions.

- Proposition-proposition edges: There is a mutex-edge between two propositions $l_1^t \in \mathcal{L}^t$ and $l_2^t \in \mathcal{L}^t$ if and only if l_1^t and l_2^t are mutually exclusive.
- Action-action edges: There is a mutex-edge between two actions $a_1^t \in \mathcal{A}^t$ and $a_2^t \in \mathcal{A}^t$ if and only if they are mutually exclusive.

Figure 2.1 illustrates an example for a planning graph. In the example we have one packet `P`, one airplane `F` and two airports `A` and `B`. Initially the packet and the airplane are at airport `A`. The goal is to bring the packet to airport `B`. Circle shaped nodes represent propositions and rectangular ones represent actions. However, only a subset of the possible propositions and actions are shown for reasons of clarity. Straight lines represent precondition and effect relations. For example the action `fly(F, A, B)` has (among others that are not illustrated) the precondition `+at(F, A)`. Furthermore it has the effects `-at(F, A)` and `+at(F, B)`. Dashed lines represent mutexes. Once again only a subset of the actually present mutexes is illustrated in the figure. For instance there is an emutex between `+at(F, A)` and `-at(F, A)`. Moreover we can see that there exists a cmutex between `+in(P, F)` and `+at(F, B)` in the second proposition level. They are cmutex in this proposition level since it takes two time steps (`load(P, F, A)`, `fly(F, A, B)`) to load the packet and fly to `B`. The goal proposition `+at(P, B)` appears for the first time in the fourth proposition level. Finally let us fix some important properties of the planning graph. Firstly it is easy to recognize that if a proposition is present at some level it will also be present at all subsequent levels due to the no-ops. The same holds for

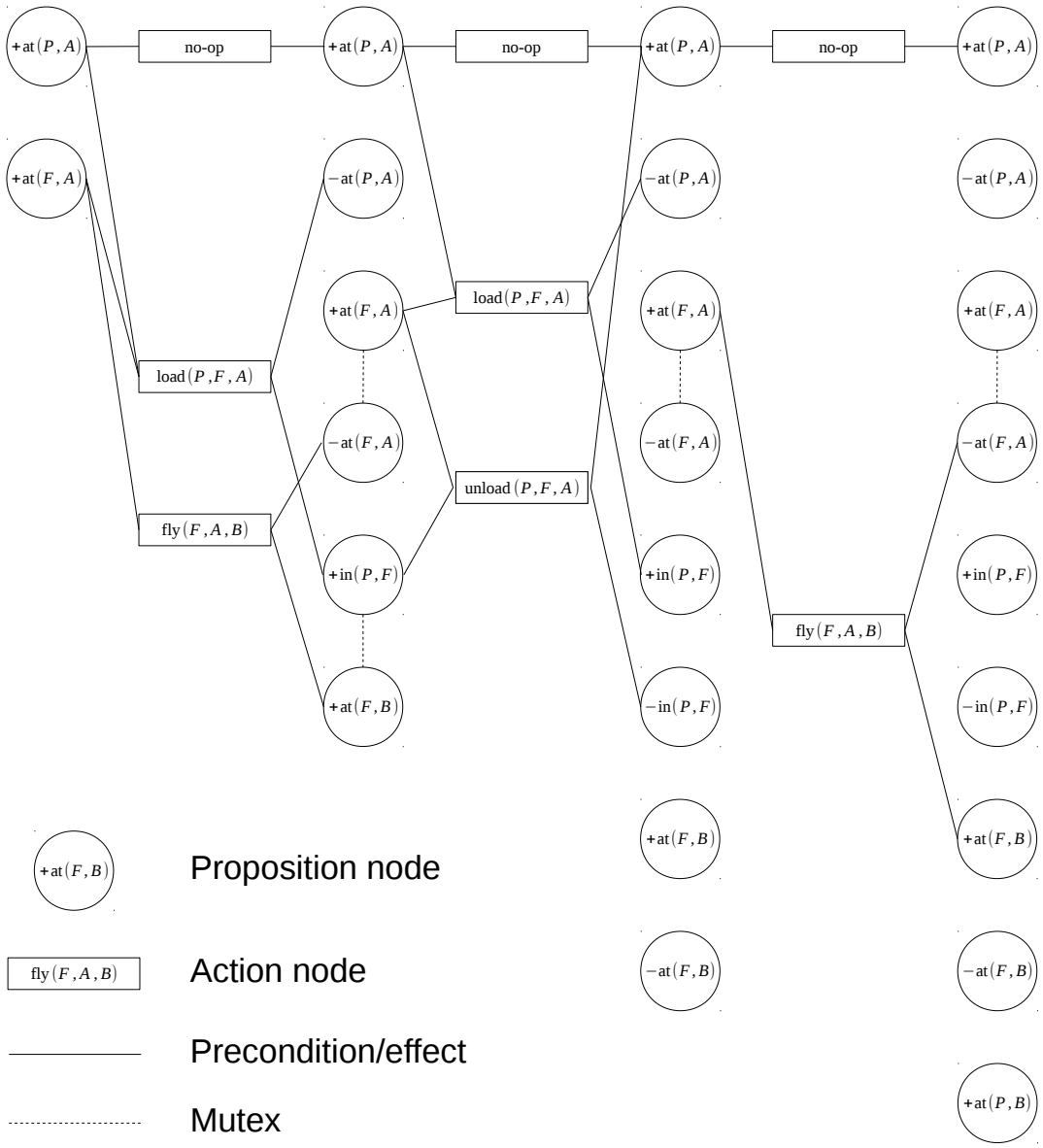

Figure 2.1: Exemplary planning graph.

actions. Regarding mutexes the planning graph behaves differently. If two actions/propositions are not mutex at some level they will not be mutex in any subsequent level. This is also because of the no-ops.

2.3 The Algorithm

The aim of Graphplan is to find a sequence of independent action sets $A^1 \subseteq \mathcal{A}^t, \ldots, A^T \subseteq \mathcal{A}^T$ that achieves the given goal in minimal time T. In other words the plan has to transform the initial condition into the goal by applying actions. In order to determine such a valid plan Graphplan alternates between two phases starting with a planning graph that contains only one proposition level representing the initial condition. In the graph expansion phase the planning graph is extended by one action level and one proposition level. In the subsequent solution extraction phase a backward-chaining search is performed on it. If a valid plan is found the procedure terminates otherwise the graph is expanded once again and another solution extraction is performed. This is repeated iteratively until either a plan is found or Graphplan proves the problem to be unsolvable. Instead of the term iteration we will also use the term *stage* to denote a pair of graph expansion and solution extraction executed in the same iteration. Thereby Graphplan follows a principle which is very common for planning algorithms: In the expansion phase we compute and optimistic guess on the plan duration T by incorporating only mutexes between pairs of propositions or actions and during solution extraction it is verified or falsified.
In the following subsections the method is presented in detail. First we take a look at how the planning graph is constructed. Afterwards it is shown how a plan can be extracted from it. Finally we describe how Graphplan identifies unsolvable problems.

2.3.1 Graph Expansion

Initially the graph contains one proposition level containing the propositions in the initial condition. Now every operator defined in the domain including the no-ops is instantiated in all possible ways. Then Graphplan checks for each action whether (1) all its preconditions are present in the initial condition and (2) no pair of preconditions is mutex in the first proposition level. If both conditions are satisfied a node representing the action is added to the new action level, the node is linked to all preconditions, nodes for its effects are added to the new propositions level if not already present, and the action node is connected to its effects. Finally the mutex relations are added. Therefore it is checked for each pair of actions in the newly added action level whether they have competing needs or are interfering. Afterwards for each pair of propositions in the newly added proposition level it is checked if they are competing effects or not. In the second graph expansion the recently added proposition level is taken as the precondition level and so on. A pseudo code for graph expansion is given in

algorithm 2.1. PG denotes the planning graph and coveredActions is a function returning the set of actions that can be executed given the propositions $\mathcal{L}^t$ as preconditions and the mutexes among them. Blum et al. demonstrate that the number of nodes in a proposition level is in

Algorithm 2.1 Graph Expansion

function EXPAND(PG, Op, $\mathcal{C}$, t)
  $\mathcal{A}^t \leftarrow$ coveredActions(Op, $\mathcal{C}$, $\mathcal{L}^t$)
  Connect preconditions in $\mathcal{L}^t$ with actions in $\mathcal{A}^t$ via an edge
  Add mutex-edges between actions in $\mathcal{A}^t$
  $\mathcal{L}^{t+1} = \mathcal{L}^t \cup \bigcup_{a \in \mathcal{A}^t} \text{eff}_a$
  Connect actions in $\mathcal{A}^t$ with effects in $\mathcal{L}^{t+1}$ via an edge
  Add mutex-edges between propositions in $\mathcal{L}^{t+1}$
  return PG
end function

$\mathcal{O}(|\mathcal{I}| + |\text{Op}|sn^k)$ and the number of actions in an action level is in $\mathcal{O}(|\text{Op}|n^k)$ where $|\mathcal{I}|$ is the number propositions/literals in the initial condition, |Op| the number of operators, s and k the size of the longest effect-list and the maximum number of parameters of operators in Op and n the number of objects [BF95]. Consequently the size of the planning graph is polynomial in $|\mathcal{I}|$, |Op|, s and n. Furthermore they argue that the time complexity of one graph expansion is polynomial in the number of nodes in the current graph level because one graph expansion can be broken down into the computation of covered actions and the computation of both action-action and proposition-proposition mutexes.

2.3.2 Solution Extraction

The solution extraction function is called after each graph expansion step. It starts in the most recently added proposition level $T + 1$. At first it checks whether all goal propositions are present in the last proposition level and if any two of them are mutex. If either a goal proposition is missing or at least one pair of goal propositions is mutex the function returns failure. Otherwise it tries to find a non-mutex set of actions which has the goal propositions as effects. For this purpose bundles of actions A_T are computed by recursively adding single actions to it at a time that are not mutex with any of the actions already in the bundle. If no such action set exists the algorithm returns failure otherwise these actions' preconditions form a set of sub-goals at the penultimate proposition level T. This means that if the sub-goals can be achieved at the penultimate proposition level we know that the complete problem can be solved by appending the current temporary action bundle A_T. Having computed the subgoals of an action bundle for proposition level T the procedure is started over recursively for the precedent graph level. That is Graphplan recursively tries to find action sets A_{T-1} now that have the sub-goals as effects and form another set of sub-goals for proposition level $T - 2$ etc.. If all recursive calls are successful, that is the recursion continues until the first proposition

level, the initial condition, is reached and the current sub-goals are contained in the initial condition, we are finished. Algorithm 2.2 illustrates the whole procedure. In order to start the solution extraction we hand over the current planning graph PG, the goal $\mathcal{G}$ and the index T of the most recently added action level. PG contains the initial condition $\mathcal{I}$ since $\mathcal{L}^1 = \mathcal{I}$. Moreover mutex$(A, a)$ is a function determining if a is mutex with any of the actions in bundle A. In section 2.2 it was mentioned that cmutexes can speed up the solution extraction process. Having the algorithm in front of us this becomes clear quickly. The reason is just that a cmutex between actions or their preconditions will reduce the size of the recursion tree because they allow to detect dead ends at many points at which the recursion would proceed if we had not computed them. This is due to the mutex check performed in the function *PickAction*. In other words cmutexes encode knowledge about infeasible plans computed during graph expansion. The information is propagated through the planning graph via competing needs and effects.

2.3.3 Terminating on Unsolvable Problems

As already mentioned Graphplan also terminates on unsolvable problems. Termination can be guaranteed by applying a termination test after each unsuccessful solution extraction. Before we come to the test itself we have to clarify what a leveled off planning graph and nogoods are.

Level Off

As already explained a proposition which is present at some planning graph level is also going to be present at all subsequent levels due to the no-ops. Additionally actions cannot create new objects and the maximum number of propositions is finite. Thus the graph will reach a level after that proposition levels are not going to change anymore. Furthermore if there is no mutex between two propositions at some level there will not be a mutex between them at any subsequent level. This is also a consequence of the no-ops' presence. Since there cannot exist less than zero mutexes they will also reach a level after which they will not change anymore.

Definition 2.3.1 (Level Off)
We say that a graph has leveled off at level n if and only if all proposition levels $n+1$, $n+2$, etc. contain the same propositions and mutexes as level n.

It is easy to see that in case the proposition levels n and $n+1$ are equal with respect to both propositions and mutexes proposition level $n+2$ will also contain the same propositions and mutexes as level $n+1$ and consequently level n. That is from n on all proposition levels will be equal regarding these two properties. Thus we can test for a level off by comparing the last two planning graph levels after each graph expansion.

Algorithm 2.2 Solution Extraction

```
function SolutionExtraction(G, PG, T)
    if G ⊆ [L^{T+1}] and [∀l1, l2 ∈ L^{T+1} ∩ G : not mutex(l1, l2)] then
        return BackwardChaining(G, PG, T + 1)
    else
        return failure                                              // failure
    end if
end function
function BackwardChaining(G, PG, t)
    if t = 1 then
        if G ⊆ L^1 then                                             // L^1 = I
            return empty list                                       // success
        else
            return failure                                          // failure
        end if
    else
        return PickAction(∅, A^{t−1}, G, ∅, PG, t − 1)
    end if
end function
function PickAction(A, B, G, P, PG, t)
    B' ← B
    while B' ≠ ∅ do
        a ← action b ∈ B' with eff_b ∩ G ≠ ∅
        B' ← B' \ {a}
        if not mutex(A, a) then                                     // Mutex check
            A' ← A ∪ {a}, G' ← G \ eff_a, P' ← P ∪ pre_a
            Plan ← empty list
            if G' = ∅ then
                Plan ← BackwardChaining(P', PG, t)
            else
                Plan ← PickAction(A', B', G', P', PG, t)
            end if
            if Plan ≠ failure then
                Plan.append(A')
                return Plan                                         // success
            end if
        end if
    end while
    return failure                                                  // failure
end function
```

Nogoods

Nogoods are proposition sets that are proven to be unachievable at certain proposition levels. In the standard algorithm 2.2 nogoods are ignored. However, they can speed up the solution extraction phase significantly similarly as mutexes. To store nogoods an empty set of nogoods is initialized for each planning graph level when it is added. And in case the backward recursion detects a failure at some graph level the currently considered set of subgoals is stored in the respective nogood set (*memoization*). The next time the solution extraction algorithm ends up at the same level with the same set of subgoals it can directly return failure after comparison with the memoized nogoods. Apart from the speedup nogoods provide they are also required for the termination test.

The Termination Test

If the planning graph has not leveled off yet Graphplan checks for it after each unsuccessful solution extraction. In case no level off was detected the algorithm proceeds with another expansion step. In the case that a level off is detected it can happen that a goal proposition is missing or two present goal propositions are mutex. If this is the case Graphplan can obviously return that the problem is unsolvable. However, this criterion is not sufficient. To illustrate this let us consider a problem from the blocks world (see also listing B.2) mentioned in the Graphplan paper [BF95]. If we have three blocks `A`, `B` and `C` and the goal is $\{$`On(A, B)`$,$`On(B, C)`$,$`On(C, A)`$\}$ it is possible to make every pair of these propositions true in one proposition level. That is at some proposition level there will not be a mutex between any pair of propositions in the goal. But it is impossible that all three propositions hold at the same time. Consequently we need a different criterion.

Theorem 2.3.2 (Termination)
Let S_i^u be the set of nogoods for proposition level i after stage u and n be the level at which the planning graph leveled off. Moreover assume that Graphplan passed an unsuccessful stage $t > n$. Then if

$$|S_n^t| = |S_n^{t-1}|$$

the problem is unsolvable.

A proof for this theorem is given in [BF95]. It yields a simple termination test. All Graphplan has to do is check for each pair of subsequent leveled off proposition levels whether or not the number of nogoods changes any longer.

3 Planning via Mixed-Integer Linear Programming

In the last chapter we already mentioned that Graphplan is strongly related to planning as satisfaction, planning as inference and Mixed-Integer Programming which are in turn related to each other. In this thesis we are going to deal with the latter approach. Hence we define Mixed-Integer Programs (*MILPs*) in this chapter and show how they can be solved generally. Moreover we explain how Mixed-Integer Programming is related to non-temporal planning of concurrent actions and how we can exploit the knowledge encoded in the planning graph that was introduced in the last chapter.
The content of this chapter is based on fundamental literature [Kru06], [CSCT07], [BGR12] and a diploma thesis [Ber06] regarding the general description of MILPs and the part about Branch-and-Bound. The content of the last two chapters describing how Mixed-Integer Linear Programming can be applied to planning of concurrent actions is based on work by Vossen *et al.* [VBS99].

3.1 Mixed-Integer Linear Programs

Mixed-Integer Linear Programs have a linear objective function. We will always consider the case in which the aim is to minimize the objective value. Moreover as the name implies MILPs are a mixture of Linear Programs (*LPs*) and Integer Linear Programs (*ILPs*). That is the set of variables can be partitioned into two subsets where the first subset contains variables with integer-valued domain and the variables in the second subset have a real-valued domain.

Definition 3.1.1 (Mixed-Integer Linear Program)
Let $\mathbb{R}_- = \mathbb{R} \cup -\infty$, $\mathbb{R}_+ = \mathbb{R} \cup +\infty$, $k, m, n \in \mathbb{N}$, $\mathbf{x} \in \mathbb{R}^k$, $\mathbf{y} \in \mathbb{Z}^m$, $\boldsymbol{\kappa} \in \mathbb{R}^{k+m}$, $C \in \mathbb{R}^{n\times(k+m)}$, $\mathbf{b} \in \mathbb{R}^n$, $\mathbf{c} \in \mathbb{R}_-^{k+m}$, $\mathbf{d} \in \mathbb{R}_+^{k+m}$ *and* $\mathbf{u} = (\mathbf{x} \quad \mathbf{y})^T$ *then*

$$\begin{aligned} &\min && \boldsymbol{\kappa}^T\mathbf{u} \\ &\textit{such that} && C\mathbf{u} \leq \mathbf{b} \\ & && \mathbf{c} \leq \mathbf{u} \leq \mathbf{d} \end{aligned}$$

is a Mixed-Integer Linear Program.

In the following we will denote real-valued variables with the letter x and integer-valued ones with y if not explicitly defined differently. Depending on the ratio of the number of integer variables to the number of real-valued variables MILPs can be as easy to solve as LPs or as difficult as an ILP (NP-hard). Usually their difficulty lies somewhere in between. MILP-formulations for planning problems are often designed in way that the integrality of the integer-valued variables implies the integrality of the real-valued variables.

3.2 Branch-and-Bound

Branch-and-Bound is a general purpose solution method Mixed-Integer Programs. If the programs are linear as in our case Branch-and-Bound solves a series of LPs in order to determine the solution of the MILP. To compute solutions for the LPs any appropriate LP-solver such as a penalty method [SC97] or the simplex method [NM65] can be applied. However, the latter is the most frequently used algorithm. The Branch-and-Bound method starts by deriving the LP-relaxation of the MILP given as an input and attaching it to a node. Afterwards an LP-solver is applied to solve the relaxed problem. If no solution is found Branch-and-Bound can return that the given MILP is unsolvable because not even the relaxed problem could be solved. Otherwise there are two cases. Either the LP-solution satisfies the removed integrality restrictions or not. In the first case we say that an integer-feasible solution was found. Finding an integer-feasible solution means that we also found a solution to the original MILP. In this case the algorithm terminates and outputs the solution which is guaranteed to be the best possible one. In the second case we choose an integer variable $c_i \leq y_i \leq d_i$ and a branching value $\overline{y}_i \in \{c_i, \ldots, d_i - 1\}$. Given $\overline{y}_i$ we get two new LPs by inheriting the original problem's constraints and adding either of the two constraints $y_i \leq \overline{y}_i$ and $y_i \geq \overline{y}_i + 1$. Once again each of the two nodes is attached to a node and each node is linked to its parent which is the root in this special case. Now one of the new LPs is chosen and solved applying the LP-solver. Depending on the result there are three cases.

- The solution is LP-feasible but not integer-feasible, that is it satisfies the LP-relaxation's constraints but not the integrality constraints of the original MILP.
- The solution is LP-infeasible, that is it does not even satisfy the LP-relaxation's constraints.
- The solution is integer-feasible, that is it is LP-feasible and additionally satisfies the integrality constraints.

In the first case we choose a branching-variable and -value once again and create two new LPs. After that we can either decide to solve one of the LPs attached to the two new child nodes or continue with the root's second child. In the third case it does not make sense to branch further on this node since any of its children will either yield an integer-infeasible solution or a worse objective value. If the only goal is to find any feasible solution we are finished at this point. Otherwise we may continue the search for better solutions at the root's second child node as in

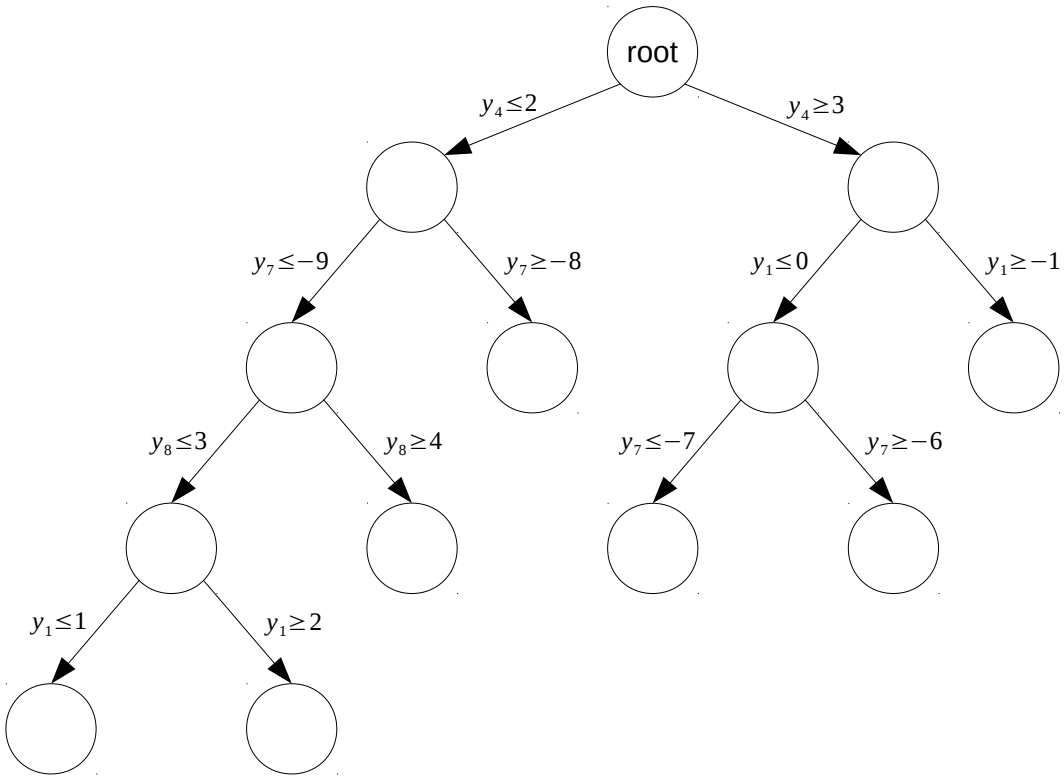

Figure 3.1: Exemplary tree of LPs produced by the Branch-and-Bound method.

the first case. In the second case branching on this node would also be worthless since none of its children is going to give a feasible solution. So in this case we are forced to continue the search at the root's second child node which is the only remaining *active node*. Active nodes are nodes which have not been visited yet. All others are *inactive nodes*. In the second case the node is declared to be an *infeasible fathomed node* additionally to making it inactive. In the first and the third case the node can be declared *feasible fathomed* if its objective value is greater than the objective value of the current *incumbent node*. The incumbent is the node with the currently lowest objective value. Feasible and infeasible nodes share the property that it is not necessary to branch further on them. Repeating the described process yields a tree of LPs as illustrated in figure 3.1. Thereby the Branch-and-Bound method produces both a lower and an upper bound on the best possible objective value. Initially the lower bound is $-\infty$ and the upper bound $+\infty$. Later on the lower bound is given as the minimum of the value $+\infty$, the objective values of the incumbent and all nodes with at least one active child. The upper bound takes on the value $+\infty$ until a first feasible solution is found. After that the upper bound

is given as the objective value of the current incumbent. If Branch-and-Bound reaches a state where the lower bound equals the upper bound it can terminate. This condition is equivalent with the condition that the number of active nodes equals zero. In the special case that the bounds equal $+\infty$ the problem is unsolvable. Algorithm 3.1 formalizes the Branch-and-Bound method. It is based on the description by Cole-Smith *et al.* [CSCT07].

Algorithm 3.1 Branch-and-Bound

0. Set the lower bound $\lambda_- = -\infty$, the upper bound $\lambda_+ = +\infty$ and the number of active nodes $k = 1$. Go to step 1.
1. If $k = 0$ or $\lambda_- = \lambda_+$ then stop and either output *No solution* ($\lambda_+ = +\infty$) or the solution ($\lambda_+ < +\infty$) else go to step 2.
2. Choose an active node, solve the corresponding LP and declare it as inactive. If the solution is LP-infeasible go to step 3. Else if the solution is integer-feasible go to step 4. Else if the node is LP-feasible, not integer-feasible and has a greater objective value than the incumbent go to step 5. Else go to step 6.
3. Declare the node as infeasible fathomed. Update the lower bound. Set $k = k - 1$. Go to step 1.
4. Declare the node as feasible fathomed. Update the lower bound. If there is no incumbent yet or the objective value is lower than the incumbent's set $\lambda_+ = \lambda_0$ where λ_0 is the current node's objective value and declare the current node as the incumbent. Set $k = k - 1$. Go to step 1.
5. Declare the node as feasible fathomed. Update the lower bound. Set $k = k - 1$. Go to step 1.
6. Select a fractional variable y_i and a branching value $\overline{y}_i$. Create two new child nodes. As constraints for the child nodes take the constraints of the current node and add the constraint $y_i \leq \overline{y}_i$ for the left child and the constraint $y_i \geq \overline{y}_i + 1$ for the right child. Update the lower bound. Set $k = k + 1$. Go to step 1.

3.2.1 Components of Branch-and-Bound

In each iteration of the Branch-and-Bound procedure three decisions have to be taken. (1) The decision which node in the tree to consider next, (2) which variable to branch on and (3) which value to branch on. Since there is no method for either decision that can predict exactly which decision is the best one there exist a lot of general but also problem dependent heuristics. The decision rule of choice can significantly influence the performance of Branch-and-Bound.

Other interesting points in the branch and bound procedure are preprocessing and presolve as well as dynamically introducing constraints (cutting planes in the case of Linear Programming) and applying heuristic search. In the following we give a brief introduction into all of these topics.

Choosing the Next Node

There exist several rules for taking the next node to explore depending on the aim of the user. The most well known techniques are Depth-First Search (*DFS*) and Breadth-First Search (*BFS*). DFS always chooses the node deepest in the tree. It tends to yield integer-feasible solutions quickly but performs poor if the goal is to find an optimal solution. BFS always chooses the node which was added to the tree first among the unexplored ones. Usually it takes a lot of time until BFS finds the first feasible solution but if it finds one this solution tends to have a good objective value. Moreover BFS usually yields good lower bounds quickly. Another natural choice is to always pick the node with the smallest lower bound in the tree. This approach tends to produce big trees but tends to terminate quickly after the first integer-feasible solution was found.
Mixtures of these strategies are possible. For instance initially a DFS can be performed in order to find a first feasible solution quickly. Afterwards a BFS can be applied to try to find better ones.

Choosing the Branching-variable and -value

Given the branching-variable y_i the standard strategy for choosing the branching-value is $\overline{y}_i = \lfloor y_i \rfloor$. Regarding the selection of the branching-variable there exist many strategies once again. The most simple branching-variable selection strategy is to randomly choose an integer variable with a fractional solution with respect to the LP-solution of the current node. Further strategies are to pick the variable whose fractional value is closest to $.5$ or to perform so called *Strong Branching*. That is to solve the LP-relaxation for each possible branching-variable and -value and then to choose the combination yielding the best objective value. Since this method can be very time consuming usually only a subset of the fractional integer variables is considered. Moreover pseudo cost approaches exist that record the influence on the objective value of branching on certain variables and compute a prediction on it. Then it is branched on the variable promising the smallest objective value.

Preprocessing and Presolve

Preprocessing techniques are applied once before the initial LP is solved. They can reduce the size of a MILP significantly. For instance preprocessing removes redundant constraints and

substitutes variables where possible. But preprocessing also includes more elaborate operations such as clique detection. As an example consider the following set of exemplary constraints

$$\begin{aligned} y_1 + y_2 &\leq 1 \\ y_2 + y_3 &\leq 1 \\ y_1 + y_3 &\leq 1 \end{aligned} \tag{3.1}$$

where $y_1, y_2, y_3 \in \{0, 1\}$. In this case clique detection would find the clique $\{y_1, y_2, y_3\}$ and the three constraints could be replaced by one stronger constraint

$$y_1 + y_2 + y_3 \leq 1\ . \tag{3.2}$$

Stronger in this case means that the first formulation 3.1 allows the fractional solution $y_1 = y_2 = y_3 = \frac{1}{2}$. But this fractional solution would not satisfy the constraint in the second formulation 3.2. However, for integer values in $\{0, 1\}$ which we need to find in the end both formulations are equivalent. In other words strong formulations have the property that the derived LP-relaxations approximate the constraints of the original MILP well. The strength of formulations is an important property of good formulations since they guide the search for good feasible solutions over the course of Branch-and-Bound and allow to find integer-feasible solutions quickly. Additionally the second formulation reduces both the number of constraints and elements in the constraint matrix A from six to three. This in turn tends to reduce the complexity of Simplex iterations speeding up the search.
Beyond preprocessing presolve can be applied at each node in the Branch-and-Bound tree to reduce the complexity of LPs. This makes sense since each branching step further restricts the LP-feasible region. For instance consider the following constraint set

$$\begin{aligned} y_1 + y_2 &\geq 3 \\ -1 \leq y_1 &\leq 1 \\ 1 \leq y_2 &\leq 5 \end{aligned}$$

with two integer variables y_1, y_2. This extract of a potentially bigger problem has many solutions such as $y_1 = 1$ and $y_2 = 2$, $y_1 = -1$ and $y_2 = 4$ or $y_1 = 1$ and $y_2 = 3$. If we now branch on y_2 and choose the branching value two then one of the two LPs will contain the extract

$$\begin{aligned} y_1 + y_2 &\geq 3 \\ -1 \leq y_1 &\leq 1 \\ 1 \leq y_2 &\leq 2\ . \end{aligned}$$

Now the only possible solution is $y_1 = 1$ and $y_2 = 2$. Presolve can recognize such behavior and replace all values of y_1 by one and y_2 by two and adapt the constraint matrix A respectively.
Although preprocessing and presolve are very powerful the operations performed can also be computationally difficult. Hence MILP-formulations should be designed in a way so that as few preprocessing steps are necessary as possible.

Cutting Planes

A Branch-and-Bound algorithm that dynamically introduces constraints is commonly called Branch-and-Cut method. Dynamically adding constraints, that is reducing the volume of the polyhedron containing all LP-feasible solutions by introducing a new cutting plane, tightens the MILP-formulation without adding new nodes to the tree. Therefore they can significantly improve the performance of Branch-and-Bound methods. Consider the following example

$$3y_1 + 2y_2 + 2y_3 \leq 5$$

where $y_1, y_2, y_3 \in \{0, 1\}$. Since evaluating the left hand side of this constraint for $y_1 = y_2 = y_3 = 1$ would yield a value of 7 we can add the constraint

$$y_1 + y_2 + y_3 \leq 1$$

that prohibits the case of all three variables taking on value one. However, adding constraints tends to increase the computational effort required in each simplex iteration. Thus this technique has to be applied carefully.

3.2.2 Heuristics

There are two types of heuristics: Start heuristics and improvement heuristics. Heuristics of the first kind are designed to find feasible solutions quickly. Heuristics of the second kind can be applied to find better solutions given an integer feasible solution. They can be applied at each node and build a separate temporary tree that results from heuristic node selection- and branching-strategies. Start heuristics often apply elaborate rounding techniques and perform a DFS-like node selection. Examples are *Relaxation Enforced Neighborhood Search* (RENS) and the *Feasibility Pump* [Ber06]. Improvement heuristics exploit the knowledge about known integer-feasible solutions in order to produce a new solution with a better objective value. Examples are *Local Branching* and *Relaxation Induced Neighborhood Search* (RINS) [DRL05]. RINS, for example, fixes each variable which has the same value in the current node and the incumbent. Then a sub-MILP is solved for the remaining variables. For a good overview on heuristics we refer to the diploma thesis by Timo Berthold [Ber06].

3.3 Solving Planning Problems via Mixed-Integer Linear Programming

As with Graphplan solving planning problems via MILP is performed iteratively. Each iteration consists of three phases. In the first phase the planning graph is extended exactly as in Graphplan. In the second phase a MILP-formulation is derived from the planning graph. Finally

a Branch-and-Bound method is applied to the MILP in the third phase. If a feasible solution is found the procedure terminates. Otherwise the algorithm continues by performing another iteration.

3.4 A Formulation for Non-temporal Planning of Concurrent Actions

Before we introduce novel MILP-formulations for temporal planning we present a formulation for non-temporal planning of concurrent actions based on existing work [VBS99]. However, the formulation presented in the following differs slightly from the one by Vossen et al. because we allow both positive and negative propositions rather than positive ones only. The formulation which we call a *SATPLAN-based formulation* directly model states by a set of propositions holding at a certain time. Its name stems from the closely related planning as satisfiability method SATPLAN. The constraints of the SATPLAN-based formulation can be derived from the boolean formulas that SATPLAN incorporates. In the next section we describe how these formulas can be translated into ILP-constraints. Afterwards we present the MILP-formulation for planning. The MILP-formulation consists of different types of constraints whose derivation we will illustrate. The final form of a constraint type is always surrounded by a box.

3.4.1 Transforming Boolean Formulas to Constraints

The constraints of the SATPLAN-based formulation can be translated from boolean formulas in the following way. First each formula is transformed into the Conjunctive Normal Form (CNF). For positive and negative literals $y_i^+, y_i^- \in \{0, 1\}$ each clause

$$\bigvee_{i=1}^{M} y_i^+ \vee \bigvee_{i=1}^{N} y_i^- \tag{3.3}$$

in the boolean formula can be transformed into an equivalent constraint

$$\sum_{i=1}^{M} y_i^+ + \sum_{i=1}^{N} 1 - y_i^- \geq 1\ . \tag{3.4}$$

The boolean formula 3.3 is satisfied if at least one positive variable takes on the value one or at least one negative variables takes on the value 0. In this case the constraint 3.4 is satisfied as well. Otherwise both are not satisfied. It remains to answer the question why this principle can also be applied to generate MILP-formulations rather than ILP-formulations although the former can contain real-valued variables. The reason for this is that the constraints of the MILP-formulations for planning presented in the following are designed in a way so that the integrality of the relevant real-valued variables is implied by the integrality of the integer-valued ones. Relevant means in this case that the computed plan depends on their value.

3.4.2 The SATPLAN-based Formulation

We come to the description of the SATPLAN-based formulation now. In order to express it comprehensively we introduce a couple of auxiliary sets in the following. Let T be the number of action levels in the planning graph, $l^t \in \boldsymbol{\mathcal{L}}$ be the instance of proposition l in proposition level $t \in \{1, \ldots, T+1\}$ and $a^t \in \boldsymbol{\mathcal{A}}$ be the instance of action a in action level t for $t \in \{1, \ldots, T\}$ as already defined in the last chapter. For $1 \leq t \leq T+1$ we define the following auxiliary sets.

- $\text{pre}_a^t \subseteq \mathcal{L}^t$: Represents propositions in level $\mathcal{L}^t$ that a^t has got as preconditions.
- $\text{add}_l^t \subseteq \mathcal{A}^t$: Represents actions in level $\mathcal{A}^t$ that add l^t.
- $\text{mutex}_a^t \subseteq \mathcal{A}^t$: Represents actions in level $\mathcal{A}^t$ that are exclusive with action a^t.

We remark at this point that these sets contain the no-ops. Furthermore we remark that they are represented as sub-graphs in the previously computed planning graph and hence can be derived from it efficiently. pre_l^t can be constructed by iterating over all precondition-edges that connect l^t with actions from action level t in the planning graph. The sets pre_l^{T+1} are empty. add_l^t can be determined by iterating over all effect-edges that connect the node corresponding to l^t with actions from action level $t+1$. The set add_l^1 is empty as well. Finally mutex_a^t can be computed by investigating all mutex-edges that connect a^t with actions from the same action level.

The Variables

There are two kinds of variables in the SATPLAN-based formulation. The variables of the first kind are the proposition variables. For each proposition l^t we have one variable

$$x_{l,t} = \begin{cases} 1 & \text{if } l \text{ holds in proposition level } t \\ 0 & \text{otherwise} \end{cases}$$

in the MILP-formulation. Furthermore we have one action variable for each action a^t.

$$y_{a,t} = \begin{cases} 1 & \text{if } a \text{ is executed in action level } t \\ 0 & \text{otherwise.} \end{cases}$$

The action variables are binary and the proposition variables are real-valued.

Initial Condition and Goal Constraints

The initial- and goal-state constraints ensure that propositions specified in the initial condition and goal have to hold at the beginning or end of plan execution respectively:

$$\boxed{\begin{aligned} x_l^1 &= 1 \text{ if } l \in \mathcal{I} \\ x_l^{T+1} &= 1 \text{ if } l \in \mathcal{G} \,. \end{aligned}}$$

Precondition Constraints

The precondition constraints ensure that started actions imply their preconditions. For all a^t and $l^t \in \text{pre}_a^t$ they are given by

$$y_a^t \leq x_l^t \,.$$

The precondition constraints can be represented in a stronger way by aggregating all preconditions in a single sum. For every a^t the precondition constraints can be expressed equivalently as

$$\boxed{|\text{pre}_a^t| y_a^t \leq \sum_{l^t \in \text{pre}_a^t} x_l^t \,.}$$

Backward-Chaining Constraints

Backward-chaining constraints express that propositions which hold have to be added by some action. For all l^t they are given by

$$\boxed{x_l^{t+1} \leq \sum_{a \in \text{add}_l^t} y_a^t \,.}$$

3.4.3 Proposition Exclusiveness Constraints

The proposition exclusiveness constraints ensure that a proposition and its negation cannot hold at the same time. Hence we add the constraint

$$\boxed{x_l^t + x_{\neg l}^t \leq 1}$$

for every predicate instance corresponding to l^t.

Action Exclusiveness Constraints

The action exclusiveness constraints express that two mutually exclusive actions cannot be executed in parallel. For each a^t and $b^t \in \text{mutex}_a^t$ we have

$$y_a^t + y_b^t \leq 1 \ .$$

As the precondition constraints this type can also be strengthened by aggregating all mutually exclusive actions in a single sum. This means that we can express mutual exclusiveness of actions by adding a constraint

$$\boxed{|\text{mutex}_a^t| y_a^t + \sum_{b^t \in \text{mutex}_a^t} y_b^t \leq |\text{mutex}_a^t|}$$

for every a^t.

Objective Function

In principle there are many possible choices for reasonable objective functions. Vossen et al. suggest

$$\sum_{a^t \in \boldsymbol{\mathcal{A}}} y_a^t \tag{3.5}$$

[VBS99]. This means that the number of actions in the plan is minimized. We remark that the objective function of choice can have significant influence on the computation time required to solve a planning problem since it can serve as a guide suggesting good branching variables to the Branch-and-Bound procedure.

The Principle

The goal constraints ensure that all variables corresponding to goal propositions take on the value one in the final proposition level. The backward-chaining constraints ensure that at least one action that has the respective goal proposition as an effect is started. An activated action in turn requires all its preconditions to hold. By this means the truth values of propositions are propagated backwards until the first proposition level. Consequently a valid solution exists if and only if the preconditions of the actions in the first action level are among the initial conditions. Thereby the exclusiveness constraints enforce that no mutually exclusive action bundles are executed and that it is impossible that propositions that are negations of each other hold in the same proposition level. Moreover we can see that the integrality of relevant proposition variables is indeed guaranteed by the precondition constraints.

4 Temporal Planning via Mixed-Integer Linear Programming

In the following we are going to deal with the more general temporal case of planning of concurrent actions. As opposed to the scenario described in section 2.1 we are going to allow from now on that actions can have different durations. The relevance of temporal planning problems becomes clear by considering the airplane domain specified in listing 2.1 once again. In this domain the only means of transportation is the airplane. But in real-world logistics domains we normally have various additional means of transportation such as trains and trucks. However, different vehicles require different amounts of time to get to their destination. For instance transporting goods in an airplane is typically faster than transporting them in a truck. Therefore we introduce the temporal planning scenario in this chapter. That is, in particular we will explain the transition model that is applied and generalize the notion of mutual exclusiveness. After that, we introduce two generalizations of the SATPLAN-based formulations for temporal planning. Additionally we present a test to determine a level off of the temporal planning graph in the last section.

4.1 The Temporal Planning Scenario

To be able to express that actions can have different durations we have to adapt the STRIPS-operators. Therefore we add a field *duration* to them. We call this version of STRIPS-operators a *Temporal STRIPS-operator* (TSTRIPS-operator). An example is given in listing 4.1. In this case the duration of each instance of the *fly*-operator is three. In this chapter we assume that the durations are integers. In chapter 6 we will explain how we can implement real-valued durations. Furthermore we assume that all operators have at least one effect and there is no operator having both an abstract proposition and its negation as effects. Moreover it is

Listing 4.1 Exemplary TSTRIPS-operator.

```
fly(c1, c2, c3)
   precondition { +airplane(c1), +airport(c2), +airport(c3),
                  -equal(c2, c3), +at(c1, c2) }
   effect       { -at(c1, c2), +at(c1, c3) }
   duration 3
```

Listing 4.2 Actions affecting proposition l and the corresponding predicate instance.

```
preAdd(c1)
   precondition { +l(c1) }
   effect       { +l(c1) }
   duration 3

preDel(c1)
   precondition { +l(c1) }
   effect       { -l(c1) }
   duraction 1

del(c1)
   precondition { }
   effect       { -l(c1) }
   duraction 1

add(c1)
   precondition { }
   effect       { +l(c1) }
   duration 5
```

important to specify a detailed transition model for temporal actions since they can overlap in arbitrary ways now. In the following we are going to denote starting times of actions by the Greek letter $\alpha \in \mathbb{N}$, time points when they stop by $\beta \in \mathbb{N}$ and their duration by $\delta = \beta - \alpha$. Now the transition model is based on the following two definitions.

Definition 4.1.1 (Execution Time)
The execution time of an action is defined as the interval $I^{exec} = \{\alpha + 1, \ldots, \beta - 1\}$.

Definition 4.1.2 (Affected Propositions and Predicate Instances)
A proposition l is affected by an action a if and only if a either adds or deletes the proposition. A predicate instance $\hat{p}$ is affected by an action a if and only if $+l(\hat{p})$ or $-l(\hat{p})$ are affected.

In order to specify that a starting/stopping time, duration or execution time corresponds to a certain action a we write α_a, β_a, δ_a and I_a^{exec}. From this definition we can conclude that if an action affects a proposition l it also affects its negation $\neg l$. In order to make things clear we emphasize that all actions illustrated in figure 4.2 affect proposition l and its negation. Finally we can define the temporal transition model which is applied in this thesis.

Definition 4.1.3 (Temporal Transition Model)
The transition model complies with the following rules.

1. *A predicate instance that is affected by an action is undefined during execution time. That is neither $+l(\hat{p})$ nor $-l(\hat{p})$ may hold.*
2. *A precondition of an action a which is affected by a must hold when the action is started, that is at $t = \alpha_a$.*

3. *A precondition of an action a which is not affected by a must hold at the start and throughout execution, that is during $\{\alpha_a\} \cup I_a^{exec} = \{\alpha_a, \ldots, \beta_a - 1\}$.*

4. *An effect of an action a is guaranteed to hold when the action terminates, that is at $t = \beta_a$.*

Moreover we have do discuss the role of mutexes in the temporal planning scenario. Whereas there were only two types of mutexes, that is proposition-proposition and action-action mutexes in the non-temporal case, we require an additional third type, namely proposition-action mutexes, now. Proposition-action mutexes are motivated by the first rule of the transition model because affected propositions may not hold while an action is executed, that is during I^{exec}.

Definition 4.1.4 (Temporal Proposition-Action Emutex)
A proposition l and an action a are temporally emutex if and only if a affects l (and consequently $\neg l$) or $\neg l$ is a precondition of a.

Proposition-proposition emutexes are defined as in the non-temporal case.

Definition 4.1.5 (Temporal Proposition-Proposition Emutex)
Two propositions $l_1, l_2 \in \mathcal{L}$ are temporally emutex if and only if l_1 is the negation of l_2.

Furthermore we adopt the definition of interference.

Definition 4.1.6 (Temporal Action-Action Emutex)
Two different actions are temporally emutex if and only if at least one of them deletes the precondition or effect of the other or they have temporally emutex preconditions.

Both the transition model and the temporal emutexes (temutexes) are defined as in the paper on Temporal Graphplan by Smith *et al.* [SW99]. From the transition model we can conclude another type of emutex between two actions.

Corollary 4.1.7 (Effective Temporal Action-Action Emutex)
Two different actions a and a' are effectively temporally emutex if one of the following conditions holds.

1. *Both a and a' add l, $I_a^{exec} \cap I_{a'}^{exec} \neq \emptyset$ and $\beta_a \neq \beta_{a'}$.*

2. *a has precondition l, a does not add l but a' does or vice versa. Furthermore $I_a^{exec} \cap I_{a'}^{exec} \neq \emptyset$.*

Proof 4.1.8 (Effective Temporal Action-Action Emutex)

1. *Since $\beta_a \neq \beta_{a'}$ holds, one of the two actions has to terminate first. Without loss of generality let a be this action. As $I_a^{exec} \cap I_{a'}^{exec} \neq \emptyset$ holds, we can conclude $\beta_a \in I_{a'}^{exec}$. According to the fourth rule of the transition model l has to hold at $t = \beta_a$ because a adds it. Furthermore, according to the first rule l is undefined during the execution time of a'. Hence there is a conflict.*

2. *a has precondition l but does not add it. Thus the precondition must hold during a's execution time according to the third rule of the transition model. As a' adds l, it may not hold during the execution time of a'. Since $I_a^{exec} \cap I_{a'}^{exec} \neq \emptyset$ there is a conflict. The same holds if we switch the roles of a and a'.* □

We will use the abbreviation etemutex for effective mutexes. This knowledge will enable us to add further mutex-edges to the planning graph. In turn, the information about the mutexes can be exploited when we derive MILP-formulations based on the planning graph. The aim of temporal planning is the same as in the non-temporal case, that is we try to find actions with attached start times that transform the initial condition into the goal. However, the plans are generally less symmetric than in the non-temporal case because actions can overlap in arbitrary ways. This also implies that actions cannot be aggregated to bundles any longer.

4.2 The Temporal Planning Graph

In order to derive efficient temporal MILP-formulations we will have to build a generalized planning graph. Therefore we have to discuss a couple of issues. Firstly since the operators' durations can be arbitrary integers now actions can overlap in arbitrary ways. That is the graph will lose its leveled structure. Furthermore temporal planning is a problem that is continuous in time rather than discrete as in the non-temporal case. Hence there is an uncountable number of time points where we could start actions. Thus the answer arises whether we can find a countable set of time points at which we take decisions without losing optimality. Furthermore the generalized planning graph has to incorporate the novel transition model 4.1.3 as well as the mutexes 4.1.4, 4.1.5 and 4.1.6.

4.2.1 Proposition Levels as Samples on the Time Line

So far we thought of planning as a discrete problem in principle: We start in some state $\mathcal{L}^1$ given by the set of propositions in the initial condition. Then we transition from one state $\mathcal{L}^t$ to another $\mathcal{L}^{t+1}$ by applying actions until we reach the goal state. Thereby the value of a proposition l at time step t denoted by l^t changes from one state to the other. However, in the temporal case the problem becomes continuous in time and we can interpret the value of a proposition l^t as a function of time. Still, for our type of operators and integer-valued durations the continuous problem can be reduced to a discrete one. In order to explain how to do this we require the following definition by Mausam *et al.* [MW06].

Definition 4.2.1 (Decision Epochs and Pivots)
Firstly a time point when a new action may be started is called a decision epoch. Secondly a time point is called a pivot if it is either zero or a time when an action might terminate.

In the same paper one can find a sketch of a proof showing that for TSTRIPS-operators as defined in this thesis we only have to consider pivots as decision epochs. That this holds indeed becomes plausible if one considers the following argument. There is two reasons why an action cannot be executed. Either its precondition does not hold or it is mutex with another action that is currently executed. Since effects do not hold until an action has terminated in our model and mutexes can end only when actions terminate it does not make sense to consider time points when no action could terminate.
Since we have to decide at each pivot whether or not to start an action and this decision depends on the current value l^t of the propositions we will have to add a proposition level corresponding to each pivot. That is in a sense the proposition levels are samples of the proposition value functions l^t in time. It remains the question if we can find a structure describing the set of pivots. And indeed it is easy to recognize that it suffices to consider the time point at $t = 0$ and all integral multiples of the greatest common divisor of the TSTRIPS-operators' durations since they cover the pivots.

4.2.2 The Structure of the Planning Graph

Bearing in mind this observation the non-temporal planning graph can be generalized straightforwardly. Once again we have proposition levels $\mathcal{L}^1, \ldots, \mathcal{L}^{T+1}$ containing a proposition node for each proposition that could hold. Every proposition level corresponds to a time point. The first proposition level $\mathcal{L}^1$ containing the initial condition should correspond to time point $t = 0$, $\mathcal{L}^2$ to time point $t =$ GCD, $\mathcal{L}^3$ to time point $t = 2$GCD etc., where GCD denotes the greatest common divisor. Furthermore we have action nodes for actions which could be executed if we consider only the temutexes defined in defintions 4.1.4, 4.1.5 and 4.1.6. As the graph loses its leveled structure actions cannot be assigned to action levels any longer. However, actions can still be assigned to action sets $\mathcal{A}^t$ according to the proposition level in which they are started. Now proposition and action nodes are connected via the following edges.

- Precondition edges: There is a precondition edge between a proposition node $l^t \in \mathcal{L}^t$ and an action node $a^t \in \mathcal{A}^t$ if and only if l is a precondition of a.
- Effect edges: There is an effect edge from action node $a^s \in \mathcal{A}^s$ to a proposition node $l^t \in \mathcal{L}^t$ if and only if l is an effect of a and $t = s + \delta_a$.

Furthermore there are three types of mutex-edges implementing the transition model 4.1.3. We remark at this point that we only incorporate temutexes in the graph and no temporal cmutexes. Hence our method could be improved at this point.

- Proposition-proposition mutex-edges: There is a mutex-edge between two propositions $l_1^t \in \mathcal{L}^t$ and $l_2^t \in \mathcal{L}^t$ if and only if l_1^t and l_2^t are temutex.
- Proposition-action mutex-edges: There is a mutex-edge between a proposition $l^t \in \mathcal{L}^t$ and an action $a^s \in \mathcal{A}^s$ if and only if they are temutex.

- Action-action mutex-edges: There is a mutex-edge between two actions $a_1^t \in \mathcal{A}^t$ and $a_2^t \in \mathcal{A}^t$ if and only if they are temutex or etemutex.

We remark at this point that temporally shifted instances of the same action cannot overlap in the non-temporal case as they all have duration one. However, in the temporal case the execution times of actions of duration greater than one do overlap. Thus there are also mutex-edges between temporally shifted actions in the graph which is ensured by the second condition for action-action mutex-edges. We denote the execution time of a time-indexed action by $I_a^{t,\text{exec}}$. Figure 4.1 illustrates an exemplary temporal planning graph for a temporal version of the airplane domain from listing 2.1 assuming duration three for the `fly`-operator and duration one for the `load`- and `unload`-operators. Precondition and effect edges are denoted by straight lines whereas mutexes are represented as dashed lines. Only a subset of propositions, actions and edges is shown. For instance there is a proposition-action mutex between the proposition `+at(F, A)` and the action `fly(F, A, B)` because this operator forces `+at(F, A)` to be undefined during its execution time. Furthermore there is an action-action mutex between the actions `load(P, F, A)` and `fly(F, A, B)` because the latter deletes the precondition `+at(F, A)` of the former.

4.2.3 Temporal Graph Expansion

An algorithmic description of the temporal graph expansion procedure is given in algorithm 4.1. coveredActions is a function returning the set of actions that can be executed given the propositions $\mathcal{L}^t$ as preconditions and the temutexes among them. Moreover $\mathcal{B}$ is a global set of all actions that were returned by the function coveredActions at some proposition level but could not be added to the graph yet because the planning graph was not expanded to the point in time when they terminate.

4.3 Temporal SATPLAN-based Formulations

According to the transition model from definition 4.1.3 the value of predicate instances can be undefined in the temporal planning scenario. Hence we must incorporate this notion in the temporal MILP-formulations. In the following we are going to present two possibilities how to achieve this. The first formulation uses one binary variable per proposition. That is in case both the positive and the negative proposition are present at a time this formulation contains two binary variables per predicate instance. For the second formulation we use a tri-state approach, that is we incorporate a single three-valued variable per predicate instance. We remark that the information about whether or not a predicate instance has to be undefined is encoded by the mutex-edges between actions and propositions in the planning graph. It is important to note that the following constraints are designed in a way so that it is not required

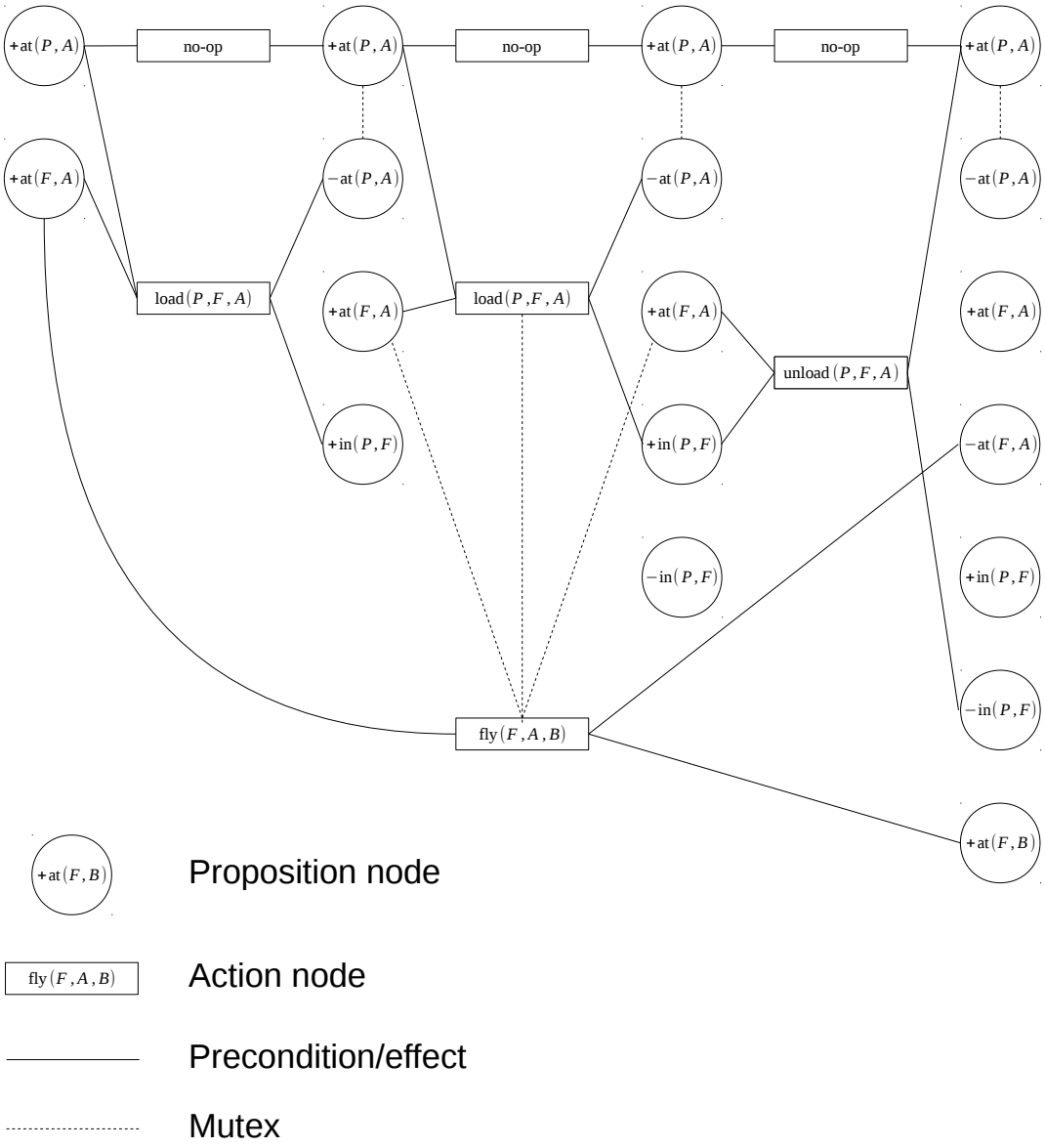

Figure 4.1: Exemplary temporal planning graph for a temporal version of the airplane domain. The `fly`-operator has duration three, the others duration one.

Algorithm 4.1 Temporal Graph Expansion

function EXPAND(PG, $\mathcal{B}$, Op, $\mathcal{C}$, t)
  $\mathcal{B}^{\text{new}}$ = coveredActions(Op, $\mathcal{C}$, $\mathcal{L}^t$)
  $\mathcal{B} = \mathcal{B} \cup \mathcal{B}^{\text{new}}$
  $\mathcal{B}^{\text{add}} = \{b \in \mathcal{B} | b$ terminates at $(t+1)$ * GCD$\}$
  $\mathcal{B} = \mathcal{B} \setminus \mathcal{B}^{\text{add}}$
  $\mathcal{L}^{t+1} \leftarrow \mathcal{L}^t$, attach time point $(t+1)$ * GCD to $\mathcal{L}^{t+1}$
  for all $b \in \mathcal{B}^{\text{add}}$ **do**
    $t_0 \leftarrow$ time index of the proposition level in which b is started
    Add b to $\mathcal{A}^{t_0}$
    Connect b to its preconditions in $\mathcal{L}^{t_0}$ via an edge
    Add mutex-edges between b and propositions in PG
    Add mutex-edges between b and actions in PG
    $\mathcal{L}^{t+1} \leftarrow \mathcal{L}^{t+1} \cup \text{eff}_b$
    Connect b with its effects in $\mathcal{L}^{t+1}$ via an edge
    Add mutex-edges between effects of b and propositions in PG
  end for
  return PG, $\mathcal{B}$
end function

to incorporate no-ops any longer which reduces the size of the action set $\boldsymbol{\mathcal{A}}$ and consequently the amount of integer variables substantially.

4.3.1 The Binary SATPLAN-based Formulation

In the following let T be the number of action levels in the planning graph, $l^t \in \mathcal{L}^t$ be the instance of proposition l in proposition level $t \in \{1, \ldots, T+1\}$ and $a^t \in \mathcal{A}^t$ be the instance of action a that can be started in proposition level t where $t \in \{1, \ldots, T\}$. Before we come to the actual formulation we introduce a few auxiliary sets.

- $\text{pre}_a^t \subseteq \boldsymbol{\mathcal{L}}^t$: Represents propositions that a^t has as a precondition. We distinguish between self-affected preconditions $\text{aff}_a^t \subseteq \text{pre}_a^t$ and unaffected preconditions $\text{unaff}_a^t \subseteq \text{pre}_a^t$.
- $\text{add}_l^t \subseteq \boldsymbol{\mathcal{A}}$: Represents actions a^s that add l^t, that is $t = s + \delta_a$.
- $\text{del}_l^t \subseteq \boldsymbol{\mathcal{A}}$: Represents actions a^s that delete l^t, that is $t = s + \delta_a$.
- $\text{trans}_l^t \subseteq \boldsymbol{\mathcal{A}}$: Represents actions a^s that are temutex with l^t.
- $\text{trans}_a^t \subseteq \boldsymbol{\mathcal{L}}$: Represents propositions l^s that temutex with a^t.
- $\text{mutex}_a^t \subseteq \boldsymbol{\mathcal{A}}$: Represents actions that are temutex or etemutex with action a^t.

We remark that these sets can be extracted efficiently from the planning graph by iterating over the outgoing edges of the respective proposition l^t regarding add_l^t, del_l^t and trans_l^t or action a^t regarding pre_a^t, trans_a^t and mutex_a^t.

The Variables

The proposition variables are defined as in the non-temporal case. For every l^t there is a variable

$$x_{l,t} = \begin{cases} 1 & \text{if } l \text{ holds in proposition level } t \\ 0 & \text{otherwise.} \end{cases}$$

The value of a predicate instance $\hat{p}$ corresponding to a literal l takes on the value *true* if and only if $x_{+l(\hat{p}),t} = 1$ holds. Furthermore $\hat{p}$ takes on the value *false* if and only if $x_{-l(\hat{p}),t} = 1$ holds. Now the notion of undefinedness is modeled by the case in which $x_{+l(\hat{p}),t} = x_{-l(\hat{p}),t} = 0$, that is $\hat{p}$ is neither *true* nor *false*.
The action variables are defined slightly different than in the non-temporal case:

$$y_{a,t} = \begin{cases} 1 & \text{if } a \text{ is executed is started at proposition level } t \\ 0 & \text{otherwise.} \end{cases}$$

Since actions can be active across proposition levels now they are defined with respect to the level in which they are started.

Initial Condition and Goal constraints

The initial and goal constraints are given just as in the non-temporal case:

$$\boxed{\begin{aligned} x_l^1 &= 1 \text{ if } l \in \mathcal{I} \\ x_l^{T+1} &= 1 \text{ if } l \in \mathcal{G} \; . \end{aligned}}$$

Precondition Constraints

According to the transition model from definition 4.1.3 the precondition constraints differ depending on whether a precondition is affected or not by an action. For all $a^t \in \boldsymbol{\mathcal{A}}$ and their affected preconditions $l^t \in \text{aff}_a^t$ we have a constraint

$$y_a^t \leq x_l^t \; .$$

Moreover there are constraints

$$y_a^t \leq x_l^s$$

for every $a^t \in \boldsymbol{\mathcal{A}}$ and propositions $l^s \in \text{unaff}_a^t$. The latter constraints can be aggregated in a sum over the execution time of a^t

$$|\text{unaff}_a^t| y_a^t \leq x_l^t + \sum_{s \in I_a^{t,\text{exec}}} x_l^s$$

in order to strengthen the formulation. Finally we can even aggregate the constraints for affected and unaffected preconditions in a single sum:

$$\boxed{|\text{pre}_a^t| y_a^t \leq \sum_{l^s \in \text{pre}_a^t} x_l^s \ .}$$

The constraints of this type can be expressed independently of the sets of affected and unaffected propositions since $\text{aff}_a^t \uplus \text{unaff}_a^t = \text{pre}_a^t$.

Backward-chaining Constraints

If l^t shall hold we require that it has already held in the last proposition level or an action $a^s \in \text{add}_l^t$ is started that adds l^t:

$$x_l^t \leq x_l^t + \sum_{a^s \in \text{add}_l^t} y_a^s \ .$$

Analogously we can derive constraints expressing that an action has to delete l if it does not hold in level t. Furthermore, in order to model that actions deleting a proposition l^t are prevented to be started if l holds in level t we add constraints

$$y_a^s \leq (1 - x_l^t) \ .$$

for every l^t and $y_a^s \in \text{del}_l^t$ which can be equivalently but stronger expressed by a single constraint

$$\sum_{a^s \in \text{del}_l^t} y_a^s \leq |\text{del}_l^t|(1 - x_l^t)$$

for each l^t. Analogously we require constraints expressing that no action adding l may be executed if it does hold in level t. Putting all these constraints together backward-chaining is modeled by the constraints

$$
\begin{aligned}
x_l^t &\leq x_l^{t-1} + \sum_{a^s \in \text{add}_l^t} y_a^s \\
1 - x_l^t &\leq 1 - x_l^{t-1} + \sum_{a^s \in \text{del}_l^t} y_a^s + \sum_{a^{t-1} \in \text{trans}_l^t} y_a^{t-1} \\
|\text{del}_l^t|(1 - x_l^t) &\geq \sum_{a^s \in \text{del}_l^t} y_a^s \\
|\text{add}_l^t| x_l^t &\geq \sum_{a^s \in \text{add}_l^t} y_a^s
\end{aligned}
$$

for every l^t. The second and the fourth constraint type correspond to the case in which l does not hold in level t. As we can see the right hand side of the second constraint incorporates a second sum as opposed to the first constraint type corresponding to the case in which l does hold. This sum is necessary because a proposition's value can toggle from *true* to *false* if there is an action started at proposition level $t-1$ affecting l, that is if there is a proposition-action mutex between the affecting action and the proposition.

Proposition Exclusiveness Constraints

Proposition exclusiveness constraints guarantee that if a proposition l^t holds its negation takes on the value *false* and vice versa:

$$x_l^t + x_{\neg l}^t \leq 1 \ .$$

However, since actions can be active across proposition levels we have to ensure that all affecting actions $a^s \in \text{trans}_l^t$ are inactive:

$$x_l^t + y_a^s \leq 1 \ .$$

Once again we can represent these constraints in a single one consisting of two parts, namely one part implementing proposition-proposition mutexes and one part for proposition-action mutexes. That is for all $l^t \in \boldsymbol{\mathcal{L}}$ we have one constraint

$$
(1 + |\text{trans}_l^t|)x_l^t + \overbrace{x_{\neg l}^t}^{\text{proposition-proposition}} + \overbrace{\sum_{a^s \in \text{trans}_l^t} y_a^s}^{\text{proposition-action}} \leq 1 + |\text{trans}_l^t| \ .
$$

Action Exclusiveness Constraints

As the proposition exclusiveness constraints, the action exclusiveness constraints can also be split up into two parts. The first one models proposition-action mutexes, that is for every $a^t \in \boldsymbol{\mathcal{A}}$ and every proposition $l^s \in \text{trans}_a^t$ there are constraints

$$y_a^t + x_l^s \leq 1 \ .$$

The second part models action-action mutexes. For every $a^t \in \boldsymbol{\mathcal{A}}$ and all $b^s \in \text{mutex}_a^t$ there is a constraint

$$y_a^t + y_b^s \leq 1 \ .$$

In order to strengthen the formulation we aggregate these constraints in one constraint per $a^t \in \boldsymbol{\mathcal{A}}$:

$$(|\text{mutex}_a^t| + |\text{trans}_a^t|)y_a^t + \overbrace{\sum_{b^s \in \text{mutex}_a^t} y_b^s}^{\text{action-action}} + \overbrace{\sum_{l^s \in \text{trans}_a^t} x_l^s}^{\text{proposition-action}} \leq |\text{mutex}_a^t| + |\text{trans}_a^t| \ .$$

Objective Function

We can use a straightforward generalization of the cost function applied in the non-temporal case (compare equation 3.5) as the objective:

$$\sum_{a^t \in \boldsymbol{\mathcal{A}}} \delta_a y_a^t \ . \tag{4.1}$$

An alternative is to reward goal propositions that are true early:

$$-\sum_{t=2}^{T+1} \gamma^{t-2} \sum_{l \in \mathcal{G}} x_l^t \ , \tag{4.2}$$

where $0 < \gamma < 1$ is the discount factor.

4.3.2 The Tri-state SATPLAN-based Formulation

In the tri-state formulation we are guaranteed to have only one variable with domain $\{-1, 0, +1\}$ per predicate instance representing both the positive and negative version of a proposition. However, as the planning graph contains proposition nodes rather than nodes representing predicate instances we have to clarify how to extract predicate instances and the corresponding auxiliary sets from it. Therefore we introduce the following definition.

Definition 4.3.1 (Induced Predicate Instance Sets)
Let L be a set of propositions. Then we get a corresponding set of predicate instances as

$$\hat{P} = \{\hat{p} \in \hat{\mathcal{P}} | + l(\hat{p}) \in L \vee -l(\hat{p}) \in L\} .$$

Consequently we introduce the notation $\hat{\mathcal{P}}^t$ for predicate instances corresponding to a proposition level $\mathcal{L}^t$ and $\hat{\boldsymbol{\mathcal{P}}}$ for the set of all time-indexed predicate instances. Furthermore since relations between actions and propositions are encoded as edges between action and proposition nodes rather than predicate instance nodes we have to distinguish between positive and negative propositions regarding these relations. Hence we introduce the notation

$$L^+ = \{\hat{p} \in \hat{\mathcal{P}} | + l(\hat{p}) \in L\}$$
$$L^- = \{\hat{p} \in \hat{\mathcal{P}} | - l(\hat{p}) \in L\} .$$

for an arbitrary set of propositions L. This notation can be applied to express for instance precondition relations and proposition-actions mutexes. In the following let T be the number of action levels in the planning graph, $l^t \in \mathcal{L}^t$ and $p^t \in \mathcal{P}^t$ be the time-indexed proposition l and predicate instance $\hat{p}$ in level $t \in \{1, \ldots, T+1\}$. Moreover let $a^t \in \mathcal{A}^t$ be the instance of action a that can be started in proposition level $t \in \{1, \ldots, T\}$. We come to the definition of the auxiliary sets now.

- $\text{pre}_a^{t,\pm} \subseteq \hat{\boldsymbol{\mathcal{P}}}^t$: Represents predicate instances corresponding to positive/negative propositions that a^t has as a precondition. We distinguish between self-affected preconditions $\text{aff}_a^{t,\pm} \subseteq \text{pre}_a^{t,\pm}$ and unaffected preconditions $\text{unaff}_a^{t,\pm} \subseteq \text{pre}_a^{t,\pm}$.
- $\text{add}_{\hat{p}}^t \subseteq \boldsymbol{\mathcal{A}}$: Represents actions a^s that add $\hat{p}^t$, that is $t = s + \delta_a$ (compare chapter 2).
- $\text{del}_{\hat{p}}^t \subseteq \boldsymbol{\mathcal{A}}$: Represents actions a^s that delete $\hat{p}^t$, that is $t = s + \delta_a$ (compare chapter 2).
- $\text{trans}_{\hat{p}}^{t,\pm} \subseteq \boldsymbol{\mathcal{A}}$: Represents actions a^s that are temutex with $+l(\hat{p}^t)$ or $-l(\hat{p}^t)$.
- $\text{trans}_a^{t,\pm} \subseteq \hat{\boldsymbol{\mathcal{P}}}$: Represents predicate instances $\hat{p}^s$ for which $+l(\hat{p}^s)/-l(\hat{p}^s)$ are temutex with a^t.
- $\text{mutex}_a^t \subseteq \boldsymbol{\mathcal{A}}$: Represents actions that are temutex or etemutex with action a^t.

These auxiliary sets can be determined once again by iterating over the outgoing edges of the respective proposition and action nodes. Finally we remark that the constraints of the tri-state formulation cannot be expressed directly as boolean formulas any longer in general since the proposition variables can take on three different values.

The Variables

There is a predicate instance variable for each predicate instance $\hat{p}^t$ defined as

$$x_{\hat{p}}^t = \begin{cases} 1 & \text{if } +l(\hat{p}) \text{ holds in proposition level } t \\ 0 & \text{if neither } +l(\hat{p}) \text{ nor } -l(\hat{p}) \text{ holds in proposition level } t \\ -1 & \text{if } -l(\hat{p}) \text{ holds in proposition level } t\ . \end{cases}$$

$x_{\hat{p}}^t = 1$ and $x_{\hat{p}}^t = 0$ correspond to the cases of $\hat{p}$ being *true* or *false*. The case in which $\hat{p}$ is undefined is modeled by $x_{\hat{p}}^t = 0$. Furthermore we have one action variable for each action a^t

$$y_a^t = \begin{cases} 1 & \text{if } a \text{ is started at proposition level } t \\ 0 & \text{otherwise} \end{cases}$$

as in the binary formulation.

Initial Condition and Goal Constraints

The initial condition and goal constraints are given by

$$\boxed{\begin{aligned} x_{\hat{p}}^1 &= \begin{cases} +1 & \text{if } +l(\hat{p}) \in \mathcal{I} \\ -1 & \text{if } -l(\hat{p}) \in \mathcal{I} \end{cases} \\ x_{\hat{p}}^{T+1} &= \begin{cases} +1 & \text{if } +l(\hat{p}) \in \mathcal{G} \\ -1 & \text{if } -l(\hat{p}) \in \mathcal{G}\ . \end{cases} \end{aligned}}$$

These constraints are well-defined because for each predicate instance either the corresponding positive or negative proposition can be in the initial condition and goal respectively but not both.

Precondition Constraints

In the tri-state formulation we have to take into account the sign of the predicate instance variables. Hence we get constraints

$$2y_a^t \leq +x_{\hat{p}}^t + 1$$

for every a^t and $\hat{p}^t \in \text{pre}_a^{t,+}$ as well as constraints

$$2y_a^t \leq -x_{\hat{p}}^t + 1$$

for every a^t and $\hat{p}^t \in \text{pre}_a^{t,-}$. The precondition relations for unaffected predicate instances in $\text{unaff}_a^{t,\pm}$ can be aggregated in sums

$$2|\text{unaff}_a^{t,\pm}|y_a^t \leq \pm x_{\hat{p}}^t \pm \sum_{s \in I_a^{t,\text{exec}}} x_{\hat{p}}^s + |\text{unaff}_a^{t,\pm}|$$

in order to get a stronger version. Since all preconditions of an action have to hold to be able to execute it we can express the precondition relations by a single constraint per action a^t containing both affected and unaffected preconditions:

$$\boxed{2(|\text{pre}_a^{t,+}| + |\text{pre}_a^{t,-}|)y_a^t \leq \sum_{\hat{p}^s \in \text{pre}_a^{t,+}} x_{\hat{p}}^s - \sum_{\hat{p}^s \in \text{pre}_a^{t,-}} x_{\hat{p}}^s + |\text{pre}_a^{t,+}| + |\text{pre}_a^{t,-}| \,.}$$

Backward-chaining Constraints

In order to model the backward-chaining we require two constraints

$$x_{\hat{p}}^t \leq x_{\hat{p}}^{t-1} + 2 \sum_{a^s \in \text{add}_{\hat{p}}^t} y_a^s + \sum_{a^{t-1} \in \text{trans}_{\hat{p}}^{t,+}} y_a^{t-1}$$

$$-x_{\hat{p}}^t \leq -x_{\hat{p}}^{t-1} + 2 \sum_{a^s \in \text{del}_{\hat{p}}^t} y_a^s + \sum_{a^{t-1} \in \text{trans}_{\hat{p}}^{t,-}} y_a^{t-1}$$

per predicate instance $\hat{p}^t$ which are already in their strongest form. The first inequality ensures that if a predicate instance is *true* the predicate instance has already been *true* in the last level $t-1$ or there has to be an action that adds it independently of its value in the previous level or in the case in which $\hat{p}^t$ is undefined an action $a^{t-1} \in \text{trans}_{\hat{p}}^{t,+}$ must have been started that affects it. The second inequality implements the same semantics for the case that $\hat{p}^t$ is *false*. Moreover we have two types of constraints

$$\begin{aligned} -x_{\hat{p}}^t + 1 &\geq 2y_a^s \quad \text{for every } a^s \in \text{del}_{\hat{p}}^t \\ x_{\hat{p}}^t + 1 &\geq 2y_a^s \quad \text{for every } a^s \in \text{add}_{\hat{p}}^t \,. \end{aligned} \tag{4.3}$$

The constraints of the first type in equation 4.3 express that if $\hat{p}^t$ is *true* or undefined then no action deleting it may be active. The constraints of the second kind implement the same for the case in which $\hat{p}^t$ is *false*. Notice that it is not possible to aggregate these constraints in a sum as in the binary case. Putting all constraints together the backward-chaining is modeled by the constraints

$$\boxed{\begin{aligned} x_{\hat{p}}^t &\leq x_{\hat{p}}^{t-1} + 2 \sum_{a^s \in \text{add}_{\hat{p}}^t} y_a^s + \sum_{a^{t-1} \in \text{trans}_{\hat{p}}^{t,+}} y_a^{t-1} \\ -x_{\hat{p}}^t &\leq -x_{\hat{p}}^{t-1} + 2 \sum_{a^s \in \text{del}_{\hat{p}}^t} y_a^s + \sum_{a^{t-1} \in \text{trans}_{\hat{p}}^{t,-}} y_a^{t-1} \\ -x_{\hat{p}}^t + 1 &\geq 2y_a^s \quad \text{for every } a^s \in \text{del}_{\hat{p}}^t \\ x_{\hat{p}}^t + 1 &\geq 2y_a^s \quad \text{for every } a^s \in \text{add}_{\hat{p}}^t \end{aligned}}$$

for every $\hat{p}^t$.

Proposition Exclusiveness Constraints

Proposition exclusiveness constraints ensure that if a predicate instance $\hat{p}^t$ is either *true* or *false* all affecting actions are inactive. This can be expressed by constraints

$$\begin{aligned} x_{\hat{p}}^t + y_a^s &\leq 1 \\ -x_{\hat{p}}^t + y_a^s &\leq 1 \end{aligned}$$

for every $\hat{p}^t$ and $a^s \in \text{trans}_{\hat{p}}^{t,+}$ or $a^s \in \text{trans}_{\hat{p}}^{t,+}$ respectively. We can express these constraints in a stronger way by expressing the proposition-action mutexes as sums

$$\boxed{\begin{aligned} |\text{trans}_{\hat{p}}^{t,+}| x_{\hat{p}}^t + \sum_{\text{trans}_{\hat{p}}^t} y_a^s &\leq |\text{trans}_{\hat{p}}^{t,+}| \\ -|\text{trans}_{\hat{p}}^{t,-}| x_{\hat{p}}^t + \sum_{\text{trans}_{\hat{p}}^t} y_a^s &\leq |\text{trans}_{\hat{p}}^{t,-}| \end{aligned}}$$

for every $\hat{p}^t$. As opposed to the binary case these constraints only consist of the transition part since the proposition-proposition mutexes are already given by the fact that each predicate instance variable $\hat{p}^t$ can take on either of the values value -1, 0 and $+1$ but not two at once.

Action Exclusiveness Constraints

The action exclusiveness constraints express that during an action a^t's execution time (1) all affected predicate instances must be undefined and (2) all other actions that are mutex with a^t have to be inactive. The first part can be expressed by the two constraint types

$$\begin{aligned} y_a^t + x_{\hat{p}}^s &\leq 1 \\ y_a^t - x_{\hat{p}}^s &\leq 1 \end{aligned}$$

for every a^t and $\hat{p}^s \in \text{trans}_a^{t,\pm}$. Notice that these constraints cannot be expressed using sums over the predicate instance variables. The reason for this is that predicate instance variables can take on positive and negative values and hence a sum over them can yield the result zero although not all of them take on the value zero. The second part can be expressed as in the binary case. For every a^t we add a constraint

$$|\text{mutex}_a^t| y_a^t + \overbrace{\sum_{b^s \in \text{mutex}_a^t} y_b^s}^{\text{action-action}} \leq |\text{mutex}_a^t| \,,$$

which is already in its strongest form. Putting all equations together action exclusiveness can be expressed by the constraints

$$\begin{aligned} y_a^t + x_{\hat{p}}^s &\leq 1 \quad \text{for every } \hat{p}^s \in \text{trans}_a^{t,+} \\ y_a^t - x_{\hat{p}}^s &\leq 1 \quad \text{for every } \hat{p}^s \in \text{trans}_a^{t,-} \\ |\text{mutex}_a^t| y_a^t + \overbrace{\sum_{b^s \in \text{mutex}_a^t} y_b^s}^{\text{action-action}} &\leq |\text{mutex}_a^t| \ . \end{aligned}$$

Objective Function

The cost objective function from equation 4.1 does not have to be modified for the tri-state formulation because the action variables are defined equally. However, since the tri-state formulation incorporates predicate instance variables rather than proposition variables the rewarding objective function from equation 4.2 has to be adapted:

$$\sum_{t=2}^{T+1} \gamma^{t-2} \left(\sum_{l \in \mathcal{G}^-} x_l^t - \sum_{l \in \mathcal{G}^+} x_l^t \right) ,$$

where $0 < \gamma < 1$ is the discount factor.

4.4 Temporal Level Off

As in the non-temporal case the set of propositions in a proposition level and the set of mutexes between pairs of them will reach a fixed point. That is from some level on neither will change any longer. Since an action is added to the planning graph in level t if and only if all its preconditions are present in $\mathcal{L}^t$ and no pair of them is temutex, we can conclude that in a leveled off graph the set of actions which can be started $\mathcal{A}^t$ will also not change any longer. This property of the planning graph allows us to stop expanding it in case a level off was detected. In the non-temporal case we can identify a level off by finding two subsequent proposition levels that have both the same propositions and mutexes. But this criterion does not suffice for the temporal case as the counter example illustrated in listing 4.3 proves. The temporal planning graph that is produced for this problem is illustrated in figure 4.2. The actions are not shown for reasons of clarity. Beyond the proposition levels the figure also indicates all mutexes between pairs of propositions depicted by dashed lines. We can see that neither the set of propositions nor the set of binary mutexes changes from level three to four. That is the old criterion would detect a level off at this point. However, since operators op1 and op2 have duration two the set of binary mutexes changes from level four to five hence the planning graph has actually not leveled off. Instead we introduce the following criterion for level offs in temporal planning graphs.

Listing 4.3 Counter example illustrating that the criterion for level offs has to be modified for temporal planning.

```
domain counterExampleDomain {

   predicates { p(c1), q(c1), r(c1) }

   operators {

      op1(c1, c2, c3)
         precondition { }
         effect       { +p(c1), -v(c1) }
         duration 2

      op2(c1, c2, c3)
         precondition { }
         effect       { +q(c1), +v(c1) }
         duration 2

      op3(c1, c2, c3)
         precondition { }
         effect       { +r(c1) }
         duration 1

   }

}

problem counterExampleProblem {

   Domain = counterExampleDomain

   Objects { A }

   Initial condition { }

   Goal { p(A), q(A), r(A) }

}
```

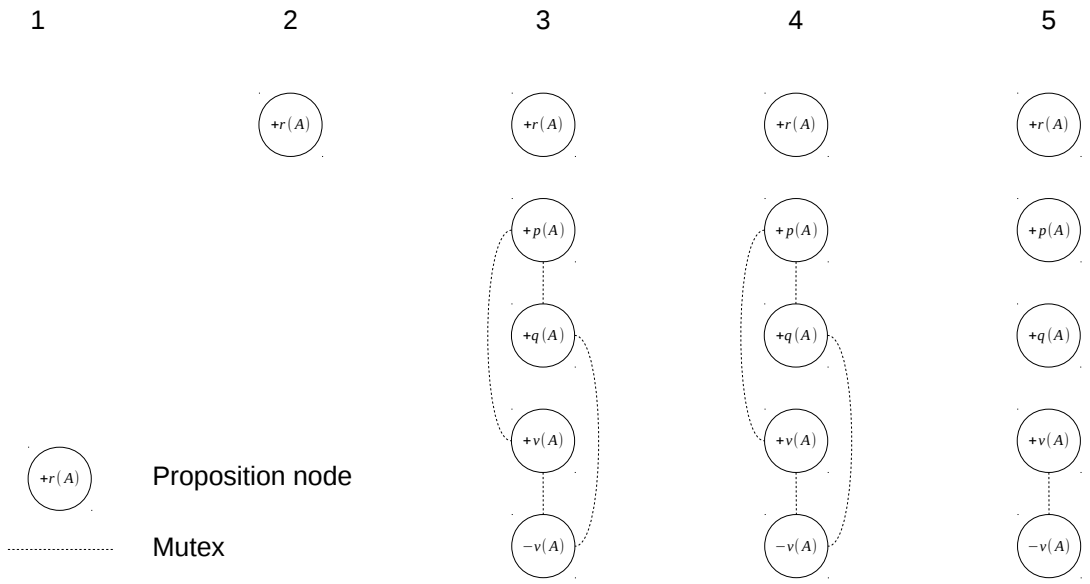

Figure 4.2: Temporal planning graph for the counter example from listing 4.3.

Lemma 4.4.1 (Temporal Level Off)
Let $\delta_{max} = \max_{op}\{\delta_{op}\}$. The temporal planning graph is leveled off in proposition level n if and only if for $\delta_{max} + 1$ subsequent proposition levels $n, n+1, \ldots, n+\delta_{max}$ neither the set of propositions nor the set of binary mutexes changes.

Proof 4.4.2 (Temporal Level Off)
***A)** If the graph is leveled off the condition above will apply at some level n.*
If the graph has leveled off at n then the proposition levels and the corresponding sets of mutexes do not change from level n on. Thus they will in particular not change in the proposition levels $n, \ldots, n + \delta_{max}$.
***B)** If the condition given above holds then the graph is leveled off at n.*
We remark that whether or not an action is added to the planning graph in a certain level $\mathcal{L}^t$ depends only on the propositions in this level and the binary mutexes among them (function coveredActions in algorithm 4.1). Now, if both the propositions and the set of binary mutexes do not change in subsequent levels $n, \ldots, n + \delta_{max}$, we know that no action was added starting in proposition level n that appended a new proposition or forced a binary mutex to end. Furthermore, since proposition levels n and $n+1$ contain the same propositions and have the same set of binary mutexes, it holds $\mathcal{A}^n = \mathcal{A}^{n+1}$. Hence, no action which adds propositions or terminates mutexes in level $n + \delta_{max} + 1$ and starts in proposition level $\mathcal{L}^{n+1}$ can be added to the graph. □

5 A State-change Approach to Temporal Planning

In this chapter we are going to take a different approach to modeling temporal planning problems as MILPs than in the last chapter. Instead of directly modeling states and actions we rather model state-changes. Vossen *et al.* already used this approach to model non-temporal planning problems [VBS99]. In the following section we are going to introduce five types of state-change variables (*SCVs*) representing different types of state-changes. Afterwards we will show how the temporal planning graph has to be extended to be able to derive efficient state-change MILP-formulations and how it is constructed. Finally we introduce the MILP-formulation itself.

5.1 The Five Types of State-change Variables

Each type of SCV models a different state-change according to the transition model from definition 4.1.3. For every SCV-type there are actions that trigger it. The first type are *maintain-variables* $v_{l,\text{maintain}}$. They model a transition in which the value of a proposition is simply propagated from a proposition level to the level succeeding it. They are triggered by no-ops. The second type are *pre-variables* $v_{l,\text{pre}}$. Similarly to the maintain-variables they model the propagation of a propositions value. However, they can propagate them over multiple time steps. SCVs of this type are triggered by actions which have a proposition as a precondition but do not affect it. Thirdly we have *add-variables* $v_{l,\text{add}}$. They model a state-change in which a proposition is added independently of whether it held before or not. Add-variables correspond to actions adding a proposition that they do not have as a precondition. *pre-add*-variables $v_{l,\text{pre-add}}$ also model state-changes in which propositions are added but differ from add-variables since they imply that the proposition held when the state-change was initiated. Hence they are triggered by actions having a proposition as both a precondition and an effect. Similarly there are further *pre-del*-variables that model state-changes in which propositions are deleted. If the type of an SCV is irrelevant we also write v_l. The set of SCVs is denoted by $\mathcal{V}$. Moreover in order to denote which levels a maintain-variable corresponds to we use the time-indexed notation $v^{s,t}_{l,\text{maintain}}$ meaning that proposition l is maintained from level s to t, where $t = s + 1$ for maintain-variables. For the other state-change variables $v^{s,t}_{l,\text{type}}$, where

type $\in \{\text{pre}, \text{add}, \text{pre-add}, \text{pre-del}\}$ t can take on an arbitrary values $s < t \leq T + 1$. The set of all time-indexed SCVs is denoted by $\boldsymbol{\mathcal{V}}$. Finally we have to define mutexes among SCVs.

Definition 5.1.1 (Temporal SCV-SCV Emutex)
Two SCVs v_l and $w_{l'}$ are temporally emutex if and only if the actions triggering v_l are temutex or etemutex with the actions triggering $w_{l'}$. Additionally we declare maintain-variables $v^{s,s+1}_{l,maintain}$ artificially as mutex with all pre-variables $v^{q,r}_{l,pre}$ if $\{s, s+1\} \cap \{q, \ldots, r\} \neq \emptyset$.

Adding the artificial mutexes will allow us to formulate stronger MILP-formulations later on. For the regular case we give the following examples: For example $v^{s,t}_{l,\text{pre-add}}$ and $w^{s,t}_{l,\text{add}}$ are mutex since the actions triggering these state-changes both affect l and their execution times overlap. Another example for two mutex SCVs are $v^{s,t}_{l,\text{pre-add}}$ and $w^{s+1,t+2}_{\neg l,\text{pre-del}}$. However, $v^{s,t}_{l,\text{pre-add}}$ and $w^{s-1,t}_{l',\text{pre-del}}$ for instance are not emutex if $l' \neq l$ and $l' \neq \neg l$.

5.2 Including State-change Variables in the Temporal Planning Graph

As a starting point we take the temporal planning graph described in the last chapter. For every possible SCV $v^{s,t}_{l,\text{type}}$, where $1 \leq s < t \leq T + 1$ and $l \in \mathcal{L}$ we add a corresponding node to the temporal planning graph if there is at least one action a^s with duration $\delta_a = t - s$ that can trigger it. Furthermore we introduce the following edges.

- Trigger-edges: There is a trigger-edge between an SCV $v^{s,t}_{l,\text{type}}$ and an action a^s if and only if a^s triggers a state-change of this type and terminates at time t.
- Mutex-edges: There is a mutex-edge between every two SCVs that are temutex according to definition 5.1.1.

The new graph expansion procedure is illustrated in algorithm 5.1. SCVsFromAction is a function that creates all SCVs triggered by an action.

5.3 The State-change Formulation

In the following let T be the number of action levels in the planning graph, $l^t \in \mathcal{L}^t$ be the instance of proposition l in proposition level $t \in \{1, \ldots, T + 1\}$ and $a^t \in \mathcal{A}^t$ be the instance of action a that can be started in proposition level t, where $t \in \{1, \ldots, T\}$. Moreover, let $v^{s,t}_{l,\text{type}} \in \boldsymbol{\mathcal{V}}$ be an SCV of the respective type, where type $\in \{\text{maintain}, \text{pre}, \text{add}, \text{pre-add}, \text{pre-del}\}$ and $1 \leq s < t \leq T + 1$. Once again we define a couple of auxiliary sets to make the formulation comprehensive.

Algorithm 5.1 Temporal Graph Expansion with State-change Variables

function EXPAND(PG, $\mathcal{B}$, Op, $\mathcal{C}$, t)
  $\mathcal{B}^{\text{new}} = \text{coveredActions}(Op, \mathcal{C}, \mathcal{L}^t)$
  $\mathcal{B} = \mathcal{B} \cup \mathcal{B}^{\text{new}}$
  $\mathcal{B}^{\text{add}} = \{b \in \mathcal{B} | b \text{ terminates at } (t+1) * \text{GCD}\}$
  $\mathcal{B} = \mathcal{B} \setminus \mathcal{B}^{\text{add}}$
  $\mathcal{L}^{t+1} \leftarrow \mathcal{L}^t$, attach time point $(t+1)$ * GCD to $\mathcal{L}^{t+1}$
  for all $b \in \mathcal{B}^{\text{add}}$ **do**
    $t_0 \leftarrow$ time index of the proposition level in which b is started
    Add b to $\mathcal{A}^{t_0}$
    Connect preconditions of b in $\mathcal{L}^{t_0}$ with b via an edge
    Add mutex-edges between b and propositions in PG
    Add mutex-edges between b and actions in PG
    $\mathcal{L}^{t+1} \leftarrow \mathcal{L}^{t+1} \cup \text{eff}_b$
    Connect b with its effects in $\mathcal{L}^{t+1}$ via an edge
    Add mutex-edges between effects of b and propositions in PG
    $\mathcal{V}^{\text{add}} \leftarrow \text{SCVsFromAction}(b)$
    for all $scv \in \mathcal{V}^{\text{add}}$ **do**
      if not exists scv **then**
        Add scv to PG
        Add mutex-edges between scv and other SCVs in PG
      end if
      Connect scv with b via a trigger-edge
    end for
  end for
  return PG, $\mathcal{B}$
end function

- $\text{pre}_l^{s,t} \subseteq \boldsymbol{\mathcal{A}}$: Represents actions a^s that terminate in proposition level t, have l as a precondition but neither add nor delete it.
- $\text{add}_l^{s,t} \subseteq \boldsymbol{\mathcal{A}}$: Represents actions a^s that terminate in proposition level t and have l as an effect but not as a precondition.
- $\text{pre-add}_l^{s,t} \subseteq \boldsymbol{\mathcal{A}}$: Represents actions a^s that terminate in proposition level t and have l both as a precondition and an effect.
- $\text{pre-del}_l^{s,t} \subseteq \boldsymbol{\mathcal{A}}$: Represents actions a^s that terminate in proposition level s and have l as a precondition and $\neg l$ as an effect.

- The following sets represent SCVs of different types corresponding to transitions starting at proposition level s

  $$\text{type}_l^{s,\star} = \{v_{l,\text{type}}^{s,r} \subseteq \boldsymbol{\mathcal{V}} | 2 \leq r \leq T+1\} ,$$

  where type $\in \{\text{pre}, \text{add}, \text{pre-add}, \text{pre-del}\}$.

- Similarly we introduce sets representing SCVs of different types corresponding to transitions ending at proposition level t

  $$\text{type}_l^{\star,t} = \{v_{l,\text{type}}^{q,t} \subseteq \boldsymbol{\mathcal{V}} | 1 \leq q \leq T\} ,$$

  where type $\in \{\text{pre}, \text{add}, \text{pre-add}, \text{pre-del}\}$.

- $\text{mutex}_{l,\text{type}}^{s,t} \subseteq \boldsymbol{\mathcal{V}}$: Represents SCVs that are mutex with $v_{l,\text{type}}^{s,t}$, where type $\in \{\text{pre}, \text{add}, \text{pre-add}, \text{pre-del}\}$.

- Finally let

  $$\text{pre}_{l,\times}^{t} = \{v_{l,\text{pre}}^{q,r} \in \boldsymbol{\mathcal{V}} | q < t \leq r\}$$

  denote the set of SCVs of type pre that cross proposition level t.

We remark that these sets can be efficiently extracted from the new planning graph. Moreover it is not necessary to incorporate no-ops because they are irrelevant for the plan we want to find. Hence it is unnecessary to introduce a set of actions triggering the maintain-variables.

The Variables

We introduce five types of real-valued variables modeling the different types of state-changes. We start with the maintain-variables:

$$x_{l,\text{maintain}}^{s,s+1} = \begin{cases} 1 & \text{if the value of } l \text{ is propagated from level } s \text{ to } t \\ 0 & \text{otherwise} \end{cases}$$

for every l^s. The other state-change variables are defined via the triggering actions. For every SCV $v_{l,\text{type}}^{s,t}$ where type $\in \{\text{pre}, \text{add}, \text{pre-add}, \text{pre-del}\}$ we incorporate a variable

$$x_{l,\text{type}}^{s,t} = \bigvee_{a^s \in \text{type}_l^{s,t}} y_a^s \,. \tag{5.1}$$

Moreover we have integer-valued action variables

$$y_a^t = \begin{cases} 1 & \text{if } a \text{ is started in proposition level } t \\ 0 & \text{otherwise} \end{cases}$$

for all a^t as common.

The Variable Definition Constraints

The definitions from equation 5.1 have to be included in the MILP-formulation. This can be done by replacing the logical or by a sum over the corresponding MILP-variables for all $v^{s,t}_{l,\text{type}}$ and $a^s \in \text{type}^{s,t}_l$

$$x^{s,t}_{l,\text{type}} \leq \sum_{a^s \in \text{type}^{s,t}_l} y^s_a$$

and adding constraints

$$x^{s,t}_{l,\text{type}} \geq y^s_a \text{ ,}$$

where $\text{type} \in \{\text{pre}, \text{add}, \text{pre-add}\}$. A special case are the pre-del variables whose definition can be expressed by a single equation, since all actions triggering a pre-del state-change are temutex with each other. Hence the definition constraints are given as

$$\boxed{\begin{aligned} x^{s,t}_{l,\text{type}} &\leq \sum_{a^s \in \text{type}^{s,t}_l} y^s_a \quad \text{for every } v^{s,t}_{l,\text{type}} \\ x^{s,t}_{l,\text{type}} &\geq y^s_a \quad \text{for every } v^{s,t}_{l,\text{type}} \text{ and } a^s \in \text{type}^{s,t}_l \\ x^{s,t}_{l,\text{pre-del}} &= \sum_{a^s \in \text{pre-del}^{s,t}_l} y^s_a \quad \text{for every } v^{s,t}_{l,\text{pre-del}} \text{ ,} \end{aligned}}$$

where $\text{type} \in \{\text{pre}, \text{add}, \text{pre-add}\}$. We remark that the variable definition constraints can be improved heuristically as often only one action exists that triggers a state-change. In this case the definition constraints for the state-change variables of a type other than pre-del can also be expressed by a single equality.

Initial Condition and Goal Constraints

To define the initial condition in the MILP we add an artificial layer zero-to-one of state-change variables and fix add-variables corresponding to the propositions in the initial condition $\mathcal{I}$ to the value one. Adding a constraint

$$\boxed{x^{0,1}_{l,\text{add}} = 1}$$

for every $l \in \mathcal{I}$ implements this behavior. To achieve that a goal proposition holds we require that at least one state-change happens that propagates it or adds it to the last proposition level $\boldsymbol{\mathcal{L}}^{T+1}$. This can be modeled by constraints

$$\boxed{x^{T,T+1}_{l,\text{maintain}} + \sum_{\text{type}} \sum_{v^{q,T+1}_{l,\text{type}} \in \text{type}^{\star,T+1}_l} x^{q,T+1}_{l,\text{type}} \geq 1}$$

for every $l \in \mathcal{G}$, where $\text{type} \in \{\text{pre}, \text{add}, \text{pre-add}\}$.

Exclusiveness Constraints

For all $v^{s,t}_{l,\text{type}_1}$ exclusiveness is expressed by the constraints

$$|\text{mutex}^{s,t}_{l,\text{type1}}|x^{s,t}_{l,\text{type1}} + \sum_{v^{q,r}_{l,\text{type2}} \in \text{mutex}^{s,t}_{l,\text{type2}}} x^{q,r}_{l,\text{type2}} \leq |\text{mutex}^{s,t}_{l,\text{type1}}| \,,$$

where type1 $\in \{\text{pre}, \text{add}, \text{pre-add}, \text{pre-del}\}$ and
type2 $\in \{\text{maintain}, \text{pre}, \text{add}, \text{pre-add}, \text{pre-del}\}$. Constraints for the case type1 $=$ maintain are not required. An explanation is given after the backward-chaining constraints were introduced.

Backward-chaining Constraints

In order to implement backward-chaining we add one constraint per proposition l^s expressing that if a state-change with respect to proposition l starts in proposition level s (maintain, pre, pre-add or pre-del) then another state-change adding l to level s (maintain, pre, add, pre-add, pre-del) must have happened before:

$$\begin{aligned} x^{s,s+1}_{l,\text{maintain}} + \sum_{\text{type1}} \sum_{v^{s,r}_{l,\text{type1}} \in \text{type1}^{s,\star}_l} x^{s,r}_{l,\text{type1}} \leq x^{s-1,s}_{l,\text{maintain}} &+ \sum_{v^{q,r}_{l,\text{pre}} \in \text{pre}^s_{l,\times}} x^{q,r}_{l,\text{pre}} \\ &+ \sum_{v^{q,s}_{\neg l,\text{pre-del}} \in \text{pre-del}^{q,s}_{\neg l}} x^{q,s}_{\neg l,\text{pre-del}} \\ &+ \sum_{\text{type2}} \sum_{v^{q,s}_{l,\text{type2}} \in \text{type2}^{q,s}_l} x^{q,s}_{l,\text{type2}} \,, \end{aligned}$$

where type1 $\in \{\text{pre}, \text{pre-add}, \text{pre-del}\}$ and type2 $\in \{\text{add}, \text{pre-add}\}$. The pre-variables $v^{q,r}_{l,\text{pre}}$ add l to every proposition level they cross because an active state-change of this type guarantees that l holds throughout execution, that is in the interval $\{q+1, \ldots, r\}$. Moreover, the we have to add the pre-del variables corresponding to l's negation to the right hand side since deleting l means adding $\neg l$. Notice that the correctness of the backward-chaining constraints is ensured by the exclusiveness constraints. They guarantee that only one variable on the left hand side can take on the value one. It remains to explain why no exclusiveness constraints for maintain variables are necessary. Assume that we have a plan in which two maintain-variables are active in parallel corresponding to a proposition l and its negation $\neg l$. Then the corresponding backward-chaining constraints ensure that a state-change has to happen that adds both of these propositions. Any state-changes other than maintains cannot achieve this since they would be exclusive due to the exclusiveness constraints. Hence the only possibility is to activate the maintain-transitions for the previous proposition level. This can go on backwards until the first artificial layer implementing the initial condition is reached. However, here either the

add-variable corresponding to l or the add-variable corresponding to its negation can be active but not both. Otherwise the initial condition would be ill-defined.

Objective function

Since proposition variables would have to be added in order to apply the reward objective on the state-change formulation we suggest to apply the cost objective from equation 3.5. The latter objective function can be adopted without any changes.

6 Real-valued Durations and Dynamic Graph Expansion

Up to now we assumed that actions have integer-valued durations. This assumption enables us to compute the greatest common divisor (GCD) of the TSTRIPS-operators' durations and to add proposition levels to the graph corresponding to times which are integral multiples of the GCD. Considering these proposition levels is correct because the pivots are integral multiples of the GCD. And the pivots are the decision epochs in turn (see definition 4.2.1). In the following we will show at which time points proposition levels must be sampled in the case of real-valued durations. Furthermore we will introduce dynamic sampling of proposition levels in order to further reduce the number of proposition levels.

6.0.1 Dealing with Real-valued Durations

In the case of real-valued durations there is no GCD and the pivots are not that regularly ordered. That is in this case we have to consider all time points

$$\{t \in \mathbb{R} | t = f(\boldsymbol{\gamma}) = \sum_{i=1}^{|\mathrm{Op}|} \gamma_i \delta_{\mathrm{op}_i}, \gamma_i \in \mathbb{N}_0\} \ , \tag{6.1}$$

where δ_{op_i} denotes the duration of TSTRIPS-operator i. For integer-valued durations holds

$$\begin{aligned}
&\{t \in \mathbb{R} | t = f(\boldsymbol{\gamma}) = \sum_{i=1}^{|\mathrm{Op}|} \gamma_i \delta_{\mathrm{op}_i}, \gamma_i \in \mathbb{N}_0\} \\
=&\{t \in \mathbb{R} | t = f(\boldsymbol{\gamma}) = \mathrm{GCD} \sum_{i=1}^{|\mathrm{Op}|} \gamma_i k_{\mathrm{op}_i}, \gamma_i \in \mathbb{N}_0, k_{\mathrm{op}_i} \in \mathbb{N}\} \\
\subseteq&\{t \in \mathbb{R} | t = f(\boldsymbol{\gamma}) = \mathrm{GCD} \cdot n, n \in \mathbb{N}_0\} \ .
\end{aligned}$$

The fact that the set of time points we considered so far is a superset of the time points we actually have to consider shows that we include more proposition levels than actually necessary. For instance if we have two operators with durations five and seven the GCD is 1 and starting at $t = 0$ we incorporate proposition levels at time points $t = 1, 2, 3, 4, 6, 8, 9, \ldots$ in the planning graph that are actually redundant because no action can terminate at these times. However using the time points depicted in equation 6.1 to sample proposition levels

instead of the integral multiples of the GCD introduces one issue: We have to distinguish the actual continuous duration δ_a of an action a from its discrete duration $\overline{\delta}_a$. With discrete duration we mean the number of periods between two proposition levels their execution time overlaps with. For example an action's execution time will overlap with the periods $0 \to 1$, $1 \to 2$, ..., $4 \to 5$ (five periods in total) if it is an instance of the operator of duration five and we use the integral multiples of the GCD to sample proposition levels. So their discrete and real durations are equal. But if we drop the unnecessary proposition levels at $t = 1, \ldots, 4$ its discrete duration will only be one. However, if the same action is started in the required proposition level corresponding to $t = 5$ then its discrete duration will be two (necessary time point $t = 7$ in between $t = 5$ and $t = 10$). Thus the two notions of duration will not be equivalent any longer. This affects the MILP-formulations we presented in the last two chapters since we supposed that

$$\delta_a = \underline{\delta}_a \text{ and } \underline{\delta}_a^1 = \underline{\delta}_a^2 = \ldots = \underline{\delta}_a^T \,,$$

where $\underline{\delta}_a^t$ denotes the discrete duration of action a if it is started at proposition level t. Similarly, we have to distinguish between an action's real $I_a^{\text{exec},t}$ and discrete execution time $\underline{I}_a^{\text{exec},t}$, real α_a and discrete starting time $\underline{\alpha}_a$, and real β_a and discrete stopping time $\underline{\beta}_a$ which become level dependent as well. However, we can solve this problem for both integer and real-valued operators by substituting the real values by the corresponding discrete level dependent ones in the MILP-formulations. Furthermore, the test for a level off changes slightly. Let $\delta_{\text{max}} = \max_{\text{op}} \delta_o p$ be the longest real duration and t be the real time attached to proposition level $\mathcal{L}^n$. To test whether a level off occurred in level n, we must check if all proposition levels whose attached real time is in the interval $[t, t + \delta_{\text{max}}]$ contain the same propositions and binary temutexes.

6.0.2 Adding Proposition Levels Dynamically

Moreover even if we drop the proposition levels corresponding to GCD multiples at which no action can possibly terminate we still add more levels than necessary because it can happen that certain actions cannot be executed since their precondition does not hold. For instance if an action of the operator of duration five mentioned in the last section cannot be started in the first proposition level the levels corresponding to $t = 5, 10, 15, \ldots$ do not have to be added. The same holds for real-valued durations. This leads to the idea of adding proposition levels dynamically to the planning graph that is to check for each time point in the sets given above if an action can terminate in fact. Since we want to add as few proposition levels as possible this requires to sort the time points in equation 6.1. For this purpose we use a heap that replaces the set $\mathcal{B}$ in algorithm 5.1. The heap will store all action instances not yet added to the planning graph but whose preconditions are satisfied. Every time a new proposition level is added new actions are added to the heap and sorted according to their stopping time. Then we extract all actions from the heap with the minimal stopping time among all actions in

it and add them to the planning graph. Algorithm 6.1 gives a formal description. $\mathcal{H}$ denotes the heap and $b.\beta$ denotes the real stopping time of action b.

Algorithm 6.1 Dynamic Graph Expansion

```
function EXPAND(PG, H, Op, C, t)
    B^new = coveredActions(Op, C, L^t)
    Add all b ∈ B^new to the heap H
    b ← H.extractMin(), t_0 ← b.β, t_1 ← t_0
    B^add ← {b}
    while t_1 = t_0 do
        b ← H.extractMin(), t_1 ← t_0
        B^add ← B^add ∪ {b}
    end while
    L^{t+1} ← L^t, attach time point b.δ to L^{t+1}
    for all b ∈ B^add do
        t_0 ← time index of the proposition level in which b is started
        Add b to A^{t_0}
        Connect preconditions of b in L^{t_0} with b via an edge
        Add mutex-edges between b and propositions in PG
        Add mutex-edges between b and actions in PG
        L^{t+1} ← L^{t+1} ∪ eff_b
        Connect b with its effects in L^{t+1} via an edge
        Add mutex-edges between effects of b and propositions in PG
        V^add ← SCVsFromAction(b)
        for all scv ∈ V^add do
            if not exists scv then
                Add scv to PG
                Add mutex edges between scv and other SCVs in PG
            end if
            Connect scv with b via a trigger-edge
        end for
    end for
    return PG, B
end function
```

7 Comparison

As already mentioned in the introduction planning via MILP is closely related to other planning techniques such as planning as satisfaction and planning as inference. Regarding the logical structure the SATPLAN-based formulations presented in chapter 4 are similar to the planning as satisfaction formulations presented in [ML06]. Broadly speaking we can generate a planning as satisfaction formulation by transforming each MILP-constraint into a logical disjunction. Connecting the individual disjunctions with conjunctions yields a big formula in conjunctive normal from which a plan can be extracted by determining a satisfying assignment. The search for such an assignment can be performed applying common SAT-solvers. Similarly we can transform our MILP-formulation into factor graphs which yields formulations for planning as inference. Once again this can be done by translating constraints into factors of the factor graph principally. If planning as inference is performed in NID-rules domains it is distinguished between *actions* and *rules*. Thereby the action defines a precondition and a number of possible effects (rules) of which one happens with a given probability. Although NID-rules domains are very different from the STRIPS-like domains considered in this thesis the logical structure of the non-temporal SATPLAN-based formulation presented in chapter 3 is very similar to the logical structure of the factor graphs that Hübner derived in the case of one deterministic NID-rule per action [Hü13]. In this special case it is not necessary to distinguish between rules and actions. Thus precondition-effect relations can be expressed by incorporating only nodes in the factor graph that represent the notion of actions as instantiated operators.
As mentioned before another algorithm that is closely related to our method is the Temporal Graphplan algorithm by Smith *et. al* [SW99] which is a generalization of Graphplan to the temporal case. In this chapter we will compare our method empirically to this algorithm. But beforehand we will describe roughly how we implemented our approach.

7.1 Implementation

A big part of this thesis was to implement the planning via MILP method presented in this thesis. Therefore the graph expansion procedure illustrated in algorithm 6.1 and the solution extraction which is performed by solving one of the MILPs presented in the previous chapters. The language of choice was C/C++.

7.1.1 Implementing the Graph Construction

As a framework for relational planning we used libPRADA [LT10]. libPRADA is mainly a framework for developing planning as inference methods for NID-rules domains. But as already explained we can use NID-rules to express STRIPS-operators by allowing only one rule that happens with probability one. In order to solve temporal planning problems we extended the NID-rules by attaching a duration to them. libPRADA allows to automatically instantiate a given set of operators. Furthermore by adapting libPRADA's state transition functionality we could implement the agglomerative computation of the proposition levels in a neat way. Based on these modifications we implemented the construction of the temporal planning graph as described in algorithm 6.1. Thereby our data structures for proposition, action and SCV-nodes wrap around the corresponding literals and rules of libPRADA.

7.1.2 Implementing the Solution Extraction

To implement the solution extraction we used the COIN Branch-and-Cut Solver (CBC) [FLH05]. CBC is an open source solver for mathematical programming problems coded in C++ and is a part of the COmputational INfrastructure for Operations Research project (COIN-OR or simply COIN). It offers established techniques for all components of the Branch-and-Bound method such as preprocessing, cut generation and heuristics as introduced in chapter 3. In our implementation solution extraction is started for the first time when a proposition level is reached in which all goal propositions are present and then after each graph expansion. Before the solution extraction is started we derive a MILP-formulation based on the planning graph as presented in chapters 4 and 5. This formulation is encoded in an MPS-file. The MPS-file contains a description of the constraint matrix and the right hand side as well as the limits of the incorporated variables (compare definition 3.1.1). After the optional preprocessing the Branch-and-Bound method is applied to try to determine a solution for the current MILP-formulation. If at least one solution is found it is written to an output file and decoded in order to determine the actions that form a plan for the given problem. Otherwise if the MILP-formulation is proven to be infeasible the method continues with another graph expansion.

7.2 Empirical Comparison

In this section the performance of the three MILP-formulations presented in this thesis are compared to each other and Temporal Graphplan (TGP) [SW99]. We run both procedures on problems from three different domains which are listed and described in detail in the appendix B. The first domain is a temporal logistics domain which is provided with the Common Lisp implementation of TGP [SW02]. In this domain packages have to be delivered from certain starting locations given in the initial condition to destinations given in the goal using various

vehicles. The second domain is a blocks world. Given an initial configuration of the blocks the goal is to build towers or exchange blocks. The third domain is the *Box Assembly Domain*. Given a number of materials, for example walls and screws, the goal is to assemble parts of a box.
In the experiments both methods start with an empty graph and solve all problems until a solution is found or the time limit of one hour is reached. Our planning via MILP method performs standard CBC-preprocessing before every solution extraction. For Branch-and-Bound we apply the default CBC-parameters. The selection strategy is a mixture of depth-first and breadth-first search. We perform strong branching for five possible node splits. We neither make use of heuristics nor cutting plane generators. As the objective we apply the cost function given in equation 4.1 and the optimization is stopped when the first feasible solution is found. Regarding TGP we deactivate conditional mutex reasoning since it is not implemented in our planning via MILP method but could be in the future. As TGP's backward-chaining search we apply the version that does subset memoization. The experiments are run on a notebook with a 2.3 GHz quad core processor where we use only one core for the experiments because both implementations are fully sequential. Furthermore the system has got 8 GB of main memory.

7.2.1 Results

Table 7.1 illustrates the results of the experiments performed. Each row corresponds to a certain problem and each column to a MILP-formulation or method. In the cells of the table we use the notation $t/c/N$ where t denotes the time required to solve the problem, c the cost of the solution that was found and N the number of LPs/nodes in the Branch-and-Bound tree solved. The times are rounded up and in seconds. If a star is placed in a cell, this means that the problem could not be solved within the time limit of one hour. We applied the methods to three problems from each domain. The problems for one domain are ordered by their difficulty.

7.2.2 Evaluation

We can see that the state-change formulation is fastest among the MILP-formulations in all domains. Moreover, the state-change formulation allowed to solve more problems within the time limit than the other two formulations. In all but one case the solution of the state-change formulation had equal cost compared to the other two formulations. However, the solution for the problem blocks2 found by applying the SATPLAN-based formulations is better. The reason for this could be that the Branch-and-Bound method applied to the state-change formulation terminated already in the root. Applied to the SATPLAN-based formulations, the tree is explored further until a feasible solution is found. The binary SATPLAN-based formulation is the second fastest one. It solves the problems blocks2 and log2 in significantly less time. Applied to the other problems the SATPLAN-based formulations perform very similarly with

Problem	Binary SATPLAN-based	Tri-state SATPLAN-based	State-change	TGP
box1	1/10/8	1/10/5	1/10/1	1/10
box2	⋆	⋆	1388/17/3431	⋆
box3	⋆	⋆	⋆	⋆
blocks1	2/6/1	3/6/1	1/6/1	1/6
blocks2	79/12/26	192/13/19	22/13/1	11/15
blocks3	103/12/47	114/12/42	22/12/5	1/12
log1	2/8/19	2/8/1	1/8/4	1/8
log2	249/25/439	1629/23/5074	218/25/30	1/25
log3	⋆	⋆	695/25/157	27/25

Table 7.1: Comparison of the three MILP-formulations and TGP. In the cells of the table we use the notation $t/c/N$ where t denotes the time required to solve the problem, c the cost of the solution that was found and N the number of LPs/nodes in the Branch-and-Bound tree solved. All times are rounded up and given in seconds. A star denotes a problem which could not be solved within the time limit of one hour.

respect to both the required time. In case of the problems blocks2 and blocks3 the tri-state formulation requires less nodes to be solved, but the total amount of time consumed is higher. The reason for this is that the tri-state formulation tends to have more constraints and non-zero elements in the constraint matrix, in particular before the preprocessing. This is caused by the backward-chaining constraints of the third and fourth kind (compare backward-chaining equations of the binary and the tri-state formulation) and the part of the action exclusiveness constraints modeling exclusiveness with predicate instance variables (compare action exclusiveness constraints of the binary and the tri-state formulation). For the tri-state formulation constraints per pair of predicate instance variable and action are necessary as opposed to the binary formulation where one constraint per proposition or action are necessary respectively. Comparing the number of explored nodes, the superiority of the state-change formulation becomes clear. The number of explored nodes is an indicator for the strength of a formulation. If few nodes have to be solved, this means that the LP-relaxation approximates the mixed-integer program effectively. The number of LPs solved is smaller by an order in case of the more difficult problems.
Comparing the state-change formulation to TGP, we can see that our planning method performs better in the box assembly domain, similarly in the blocks world domain and worse in the logistics domain with respect to the required time. However, our method solves more problems and the solution found for the problems blocks2 has less cost. Finally, we remark at this point that the performance of our method depends on the MILP-solver of choice. An empirical comparison of various commercial and non-commercial solvers revealed that the current version of the CBC-solver is 17.1 to 19.8 times slower than the best solver CPLEX depending on

the setup [Mit14]. In particular if we divide the times required to solve the problems by this factor we can conclude that our method performs similarly if not even better than this version of TGP not performing conditional mutex reasoning.

8 Summary

This thesis started with an introduction to Graphplan. We explained how Graphplan can determine plans, that are optimal with respect to their duration, in an iterative procedure. In particular, the planning graph was presented and we described how it can be used to encode knowledge about a planning problem effectively. Moreover, it was described how we can extract valid plans from the graph. After that, we gave an introduction to Mixed-Integer Linear Programming. That is, we defined the programs and described how they can be solved applying the Branch-and-Bound method. It turned out that Branch-and-Bound is based on several heuristic techniques. Besides, we explained the basic approach to solving planning problems via Mixed-Integer Programming and described a formulation for non-temporal planning. After that, the temporal planning scenario we considered was described and the required definitions were given. In particular, the transition model was introduced. In order to be able to derive efficient MILP-formulations a generalization of Graphplan's planning graph was presented, that contains one sample of propositions at each integral multiple of the operators' GCD. Next, we presented two SATPLAN-based MILP-formulations that can exploit the knowledge encoded in the temporal planning graph. The first formulation uses two binary variables per predicate instance to model the three values the predicate instance can take on, namely *true*, *undefined* and *false*. The tri-state formulation uses one three-valued variable per predicate instance to model these three cases. Furthermore, a test for a level off was presented. After that, we introduced a state-change formulation which is expressed in terms of five different state-change variables modeling all possible types of state-changes. Afterwards it was explained how to deal with real-valued durations. Thereby it turned out that considering all integral multiples of the GCD is not necessary. Moreover, it was explained that we can reduce the size of the planning graph by adding proposition levels dynamically. Finally, we compared the three MILP-formulations to each other. The results indicated that the state-change formulation performs best in all three domains we considered. Furthermore, a comparison of our method to Temporal Graphplan showed that the two algorithms perform similarly in total but differently depending on the domain of the problems. As our method depends heavily on the MILP-solver that is applied and significantly faster MILP-solvers exist we can conclude that our method performs similarly as TGP with deactivated conditional mutex reasoning, if not even better with respect to the time required to solve planning problems. Moreover, the fact that Mixed-Integer Programming allows to incorporate an optimization objective the solutions produced by our approach tend have less cost.

Outlook

Smith and Weld compared Temporal Graphplan with and without conditional mutex reasoning experimentally by applying their method to several temporal planning problems [SW99]. Thereby, they used a time limit of a hundred seconds. The results of the experiment show that Temporal Graphplan exceeded the time limit in many cases if applied without conditional mutex reasoning. However, considering conditional mutexes, many of these problems could be solved in a couple of seconds. This indicates the strength of conditional mutexes. Thus we suppose that incorporating them in our approach will improve its performance significantly as well. To achieve this only the planning graph has to be modified.
To improve the performance of the formulations for certain domains it is also possible to adapt them by incorporating domain-specific knowledge. Dimopoulos did exactly that for a non-temporal state-change formulation [Dim01]. Van den Briel *et al.* developed non-temporal MILP-formulations for multi-valued state variable descriptions [BVK05]. These descriptions can be derived automatically from propositional representations [Hel06]. The multi-valued state variables take on propositions as values and allow to particularly efficiently encode mutexes between actions. According to the authors their novel formulations outperform the one for propositional representations.

A Notation

The following list enumerates symbols frequently used throughout this thesis and their meaning.

$\mathcal{C}$	Set of objects
$\mathcal{P}$	Set of predicates
$\hat{\mathcal{P}}$	Set of predicate instances
$\mathcal{L}$	Set of propositions
$\mathcal{L}_\theta$	Set of abstract propositions
Op	Set of operators
$\mathcal{A}, \mathcal{B}$	Set of actions
$\mathcal{V}$	Set of state-change variables
$\mathcal{I}$	Initial condition
$\mathcal{G}$	Goal
p	Predicate
$\hat{p}$	Predicate instance
l	Proposition
l_θ	Abstract proposition
op	Operator
a, b	Action
A, B	Bundle of actions
v, w	State-change variable
t	Time or time index
T	Number of proposition levels minus one
PG	Planning graph
x	Real-valued variable
y	Integer-valued variable
c	Lower bound of a variable
d	Upper bound of a variable
α_a	Starting time of an action
β_a	Stopping time of an action
$\delta_{\text{op}/a}$	Duration of an operator/action
I_a^{exec}	Execution time of an action

We use bold fonts for vectors, subscripts to denote vector components and superscripts to

denote time-indexed mathematical objects. For example x_i^t stands for the value of the i-th vector component of $\mathbf{x}$ at time t.

B Temporal Planning Domains

In the following the domains of the problems used for the empirical comparison are listed and described. The first domain depicted in listing B.1 is the temporal box assembly domain. Here we are given a number of materials, that is a ground object and an equal number of screws and walls. The goal is to install the walls on the four sides of the ground object. The installation is executed by three independent robotic hands that can grab either of the materials. In order to install a wall a single screw is drilled through the wall and into the ground. Thereby one hand has to hold the ground object and one hand the wall. The third hand attaches the screw.
The second domain is a temporal blocks world domain illustrated in listing B.2. There are three types of blocks: light, normal and heavy blocks. For every block type there are three operators for moving a block (1) from the table onto another block, (2) from one block to another one and (3) from a block to the table. Light blocks take one time unit to move, normal blocks two and heavy blocks three. In the listing we use the abbreviation light/normal/heavy to denote the three versions of each operator. In the blocks world the goal is to build or take apart towers or to exchange blocks of towers. Thereby a block can be moved only if no other block is currently placed on top of it. In the last domain, the temporal logistics domain illustrated in listing B.3, the goal is to deliver packets from certain starting locations to their destinations. Starting locations and destinations of packets are always either offices, that is a location in the propositional formulation, or airports, that is a location and an airport in the propositional formulation. A problem can contain various cities and in each city there can be arbitrary many offices and airports. Packets can be transported by trucks or airplanes. Airplanes can only move from one airport to another. Trucks are more flexible since they can get to every location. However, driving between cities is much slower than taking a flight.

Listing B.1 The temporal box assembly domain.

```
domain temporalBoxAssembly {

   predicates { isMaterial(c1), isGround(c1), isWall(c1),
                isScrew(c1), isLocation(c1), inHand(c1),
                empty(c1), fixed(c1), installed(c1),
                in(c1, c2) }

   operators {

      grab(c1, c2)
         precondition { +isHand(c1), +isMaterial(c2), +empty(c1),
                        -inHand(c2), -fixed(c2) }
         effect       { -empty(c1), +inHand(c2), +in(c2, c1) }
         duration 2

      release(c1, c2)
         precondition { +isHand(c1), +isMaterial(c2), +in(c2, c1) }
         effect       { +empty(c1), -inHand(c2), -in(c2, c1) }
         duration 1

      action attachScrew(c1, c2, c3, c4, c5)
         precondition { +isHand(c1), +isScrew(c2), +isWall(c3),
                        +isLocation(c4), +isGround(c5), -fixed(c2),
                        +in(c2, c1), -fixed(c3), +inHand(c3),
                        -installed(c4), +inHand(c5) }
         effect       { +fixed(c2), +fixed(c3), +installed(c4),
                        +fixed(c5) }
         duration 1

   }

}
```

Listing B.2 The temporal blocks world domain.

```
domain temporalBlocksWorld {

   predicates { block(c1), table(c1),
                light(c1), normal(c1, c2), heavy(c1, c2),
                on(c1, c2), clear(c1), equal(c1, c2) }

   operators {

      moveFromTableLight/Normal/Heavy(c1, c2, c3)
         precondition { +light/normal/heavy(c1), +table(c2), +block(c3),
                        +on(c1, c2), +clear(c1), +clear(c3),
                        -equal(c1, c3) }
         effect       { +on(c1, c3), -on(c1, c2), -clear(c3) }
         duration 1/2/3

      moveLight/Normal/Heavy(c1, c2, c3)
         precondition { +light/normal/heavy(c1), +block(c2), +block(c3),
                        +on(c1, c2), +clear(c1), +clear(c3),
                        -equal(c1, c3), -equal(c2, c3) }
         effect       { +on(c1, c3), +clear(c2), -on(c1, c2),
                        -clear(c3) }
         duration 1/2/3

      putOnTableLight/Normal/Heavy(c1, c2, c3)
         precondition { +light/normal/heavy(c1), +block(c2), +table(c3),
                        +on(c1, c2), +clear(c1) }
         effect       { +on(c1, c3), +clear(c2), -on(c1, c2) }
         duration 1/2/3

   }

}
```

Listing B.3 The temporal logistics domain.

```
domain temporalLogistics {

   predicates { packet(c1), vehicle(c1), truck(c1),
                airplane(c1), location(c1), airport(c1),
                city(c1), locAt(c1 c2), at(c1, c2),
                in(c1, c2), equal(c1, c2) }

   operators {

      load(c1, c2, c3)
         precondition { +packet(c1), +vehicle(c2), +location(c3),
                        +at(c1, c3), +at(c2, c3) }
         effect       { -at(c1, c3), +in(c1, c2) }
         duration 1

      unload(c1, c2, c3)
         precondition { +packet(c1), +vehicle(c2), +location(c3),
                        +in(c1, c2), +at(c2, c3) }
         effect       { -in(c1, c2), +at(c1, c3) }
         duration 1

      drive(c1, c2, c3, c4)
         precondition { +truck(c1), +location(c2), +location(c3),
                        +city(c4), -equal(c2, c3), +locAt(c2, c4),
                        +locAt(c3, c4), +at(c1, c2) }
         effect       { -at(c1, c2), +at(c1, c3) }
         duration 2

      driveInterCity(c1, c2, c3)
         precondition { +truck(c1), +location(c2), +location(c3),
                        -equal(c2, c3), +at(c1, c2) }
         effect       { -at(c1, c2), +at(c1, c3) }
         duration 11

      fly(c1, c2, c3)
         precondition { +airplane(c1), +airport(c2), +airport(c3),
                        -equal(c2, c3), +at(c1, c2) }
         effect       { -at(c1, c2), +at(c1, c3) }
         duration 3

   }

}
```

Bibliography

[Ber06] T. Berthold. *Primal Heuristics for Mixed Integer Programs*. Master's thesis, Technische Universität Berlin, 2006. (Cited on pages 23 and 29)

[BF95] A. L. Blum, M. L. Furst. Fast Planning Through Planning Graph Analysis. *Artificial Intelligence*, 90(1):1636–1642, 1995. (Cited on pages 9, 11, 18 and 21)

[BGR12] B. Bixby, Z. Gu, E. Rothberg. Presolve for Linear and Mixed-Integer Programming, 2012. (Cited on page 23)

[BVK05] M. V. D. Briel, T. Vossen, S. Kambhampati. Reviving Integer Programming Approaches for AI Planning: A Branch-and-Cut Framework. In *ICAPS*, pp. 310–319. AAAI Press, 2005. (Cited on page 74)

[CSCT07] J. Cole-Smith, Z. Caner-Taskin. A Tutorial Guide to Mixed-Integer Programming Models and Solution Techniques. 2007. (Cited on pages 23 and 26)

[DG02] Y. Dimopoulos, A. Gerevini. Temporal Planning Through Mixed Integer Programming. In *Principles and Practice of Constraint Programming - CP 2002*, volume 2470, pp. 47–62. 2002. (Cited on page 9)

[Dim01] Y. Dimopoulos. Improved Integer Programming Models and Heuristic Search for AI Planning. In *ECP*, volume 6. 2001. (Cited on page 74)

[DRL05] E. Danna, E. Rothberg, C. LePape. Exploring Relaxation Induced Neighborhoods to Improve MIP Solutions. *Mathematical Programming*, 102:71 – 90, 2005. (Cited on page 29)

[FLH05] J. Forrest, R. Lougee-Heimer. CBC User Guide, 2005. URL `http://www.coin-or.org/Cbc/cbcuserguide.html`. (Cited on page 68)

[Hel06] M. Helmert. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26:191–246, 2006. (Cited on page 74)

[Hü13] M. Hübner. *Efficient Message Passing for Logical Factor Graphs and its Application to Relational Planning*. Master's thesis, Freie Universität Berlin, 2013. (Cited on page 67)

[Kru06] S. O. Krumke. Integer Programming - Polyhedra and Algorithms. Course notes, 2006. (Cited on page 23)

[KS92] H. Kautz, B. Selman. Planning as Satisfiability. In *IN ECAI-92*, pp. 359–363. Wiley, 1992. (Cited on page 9)

[LT10] T. Lang, M. Toussaint. Planning with Noisy Probabilistic Relational Rules. *Journal of Artificial Intelligence Research*, 39:1–49, 2010. (Cited on page 68)

[Mit14] H. Mittelmann. Mixed Integer Linear Programming Benchmark, 2014. URL http://plato.asu.edu/ftp/milpc.html. (Cited on page 71)

[ML06] A. D. Mali, Y. Liu. T-SATPLAN: A SAT-based Temporal Planner. *International Journal on Artificial Intelligence Tools*, 15, 2006. (Cited on pages 9 and 67)

[MW06] Mausam, D. S. Weld. Probabilistic Temporal Planning with Uncertain Durations. In *AAAI*. 2006. (Cited on page 38)

[NM65] J. A. Nelder, R. Mead. A Simplex Method for Function Minimization. *Computer Journal*, 7:308 – 313, 1965. (Cited on page 24)

[SC97] A. E. Smith, D. W. Coit. *Penalty Functions*. 1997. (Cited on page 24)

[SW99] D. E. Smith, D. S. Weld. Temporal Planning With Mutual Exclusion Reasoning. In *In Proceedings of IJCAI-99*, pp. 326–337. 1999. (Cited on pages 9, 12, 37, 67, 68 and 74)

[SW02] D. E. Smith, D. S. Weld. Common Lisp Implementation of Temporal Graphplan, 2002. URL http://www.cs.washington.edu/ai/tgp.html. (Cited on page 68)

[VBS99] T. Vossen, M. Ball, R. H. Smith. On the Use of Integer Programming Models in AI Planning. In *In Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pp. 304–309. Morgan Kaufmann, 1999. (Cited on pages 9, 23, 30, 33 and 55)

All links were last followed on November 10, 2014.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature