UNIVERSITÄT BASEL

A General LTL Framework for Describing Control Knowledge in Classical Planning

Master's Thesis

Faculty of Science, University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence
ai.cs.unibas.ch

Examiner: Prof. Dr. Malte Helmert
Supervisor: Gabriele Röger

Salomé Simon
salome.simon@unibas.ch

March 31st, 2014

Abstract

State-of-the-art planning systems use a variety of control knowledge in order to enhance the performance of heuristic search. Unfortunately most forms of control knowledge use a specific formalism which makes them hard to combine. There have been several approaches which describe control knowledge in Linear Temporal Logic (LTL). We build upon this work and propose a general framework for encoding control knowledge in LTL formulas. The framework includes a criterion that any LTL formula used in it must fulfill in order to preserve optimal plans when used for pruning the search space; this way the validity of new LTL formulas describing control knowledge can be checked. The framework is implemented on top of the Fast Downward planning system and is tested with a pruning technique called *Unnecessary Action Application*, which detects if a previously applied action achieved no useful progress.

Contents

1 Introduction **2**
1.1 Related Work 3

2 Background **5**
2.1 Planning Task Representation 5
2.2 A* search algorithm 6
2.3 Linear Temporal Logic 9
2.3.1 Evaluating LTL Formulas with progression 10

3 LTL in Classical Planning **13**
3.1 Adaption of LTL for Classical Planning 13
3.2 Evaluating LTL formulas during Search 15
3.3 Merging LTL Formulas from Nodes with Identical State 17
3.4 Integration in A* search 19

4 Unnecessary Action Applications in LTL **23**
4.1 Intended Effects 23
4.2 Unnecessary Action Applications 24
4.3 Adaption to LTL 27

5 Experimental Results **30**
5.1 Implementation 30
5.1.1 LTL framework 30
5.1.2 Unnecessary Action Application 31
5.2 Results 32
5.2.1 Memory Efficiency 33
5.2.2 Search Time 35
5.2.3 UAA with a Specifically Designed Domain 38

6 Conclusion **44**
6.1 Future work 44

Chapter 1

Introduction

Classical search tries to solve problems by creating graphs based on states which represent the current situation and actions which lead from one state to another; and trying to find a path from the initial state to a state which includes all goal constraints. Since these graphs usually grow very big, search algorithms can be enhanced with several methods in order to guide the search. The most known method is using heuristic functions which estimate the distance from a given state to the nearest goal state. Another method (which is often used in combination with heuristics) is using *control knowledge*. Control knowledge describes information gained by analyzing the problem domain and can constrain the search space. One example of control knowledge are *invariants*. Invariants describe statements which are true in any state reachable from the initial state of a planning task.

A promising way of describing such control knowledge is using *Linear Temporal Logic* (Pnueli, 1977). Linear Temporal Logic (LTL) allows us to express logical formulas where variable assignments change over time; such as "if action a_i is applied, apply action a_j next", or "eventually v must be true". Bacchus and Kabanza (2000) created a system called TLPlan, which uses domain dependent control knowledge encoded in first-order LTL to limit the search space in a forward-chaining planner. TALplanner (Kvarnström and Doherty, 2000) introduces a complex temporal logic called Temporal Action Logic (TAL) which encodes actions into logical formulas with a time component, and combines this with domain dependent control knowledge in a forward-chaining planner.

While both approaches use temporal logic to describe control knowledge, they use hand-coded domain dependent control knowledge and offer no criterion that LTL formulas describing control knowledge must fulfill in order to correctly guide the search. Furthermore, their application of temporal logic is built into a planner which was specifically designed for this use. The

framework we propose is usable in any forward search planner. It describes how LTL formulas can be integrated into the search and what criterion those formulas must fulfill. We will implement the framework in a heuristic forward search planner and test one concrete implementation of LTL-encoded domain independent control knowledge on the framework.

The thesis is organized in the following way: Chapter 2 gives some background about the planning formalism we use, the A* forward search algorithm and the LTL formalism. In Chapter 3 we explain how LTL can be translated into the planning setting and what requirements LTL formulas should fulfill in order to use them in optimal planning. In Chapter 4 we propose a pruning technique called *Unnecessary Action Application* and describe how it can be used in the LTL framework. In Chapter 5 we show some details on how the framework was implemented in Fast Downward, and present and analyze the results of testing the Unnecessary Action Application pruning technique in combination with heuristic search on several standard benchmarks for optimal planning. In Chapter 6 we conclude the thesis and present some ideas on how to extend and utilize the presented LTL framework in future work.

1.1 Related Work

TLPlan (Bacchus and Kabanza, 2000) is a knowledge-based forward-chaining planner which utilizes user-defined first order LTL formulas in order to prune paths which do not fulfill the given formulas. It uses a progression technique which incrementally checks if plan prefixes can satisfy the given formulas. TLPlan's search algorithm is only complete if there exists a plan which satisfies the user-defined LTL formulas, but it does not provide any guidance on how to build such formulas.

TALplanner (Kvarnström and Doherty, 2000) is based on TLPlan, but introduces a new temporal logic language called Temporal Action Logic (TAL). The language is split into a surface language, which is easier to read, and a base language. Control knowledge is user-defined in the surface language and automatically translated to the base language used by the planner. The surface language describes actions as STRIPS variables with time variables, where the preconditions of an action must be true at a time t_1, and the effects take place at a time t_2 (in between t_1 and t_2 the variables affected by the action are undefined). Furthermore TAL can define observations, which must be true at all times (for example invariants can be described this way) and goal control statements, which are temporal logic formulas and must be fulfilled by any optimal plan. With all these features, TAL is a very

expressive language which also allows to define concurrent actions. For classical nontemporal planning however, TAL provides more functionality than needed.

Baier and McIlraith (2006) proposed to convert LTL formulas into infinite state automata. Such automtata can be added to a planning task, where each state in the search space will be given an additional variable for each automaton, denoting in which state the automaton currently is. Since automata can have dead-end states (non-final states from which no final state can be reached), a state in the search space can be pruned if one of the automata variables denotes such a dead-end state. In order to transit from one automaton state to another, two approaches are suggested: either the actions in the planning task are modified in such a way that they also reflect the automaton transitions, or the automata variables are defined as derived predicates (axioms) and causal rules are defined which decide in which state the automaton currently is.

Wang et al. (2009) described how landmarks (Porteous et al., 2001) and their orderings can be described in LTL. Based on Baier and McIlraith (2006) they translated those formulas into automata and augmented the FF heuristic (Hoffmann and Nebel, 2001) by including the automata variables in the relaxed planning graph created by FF.

Chapter 2

Background

In this chapter we define the planning and Linear Temporal Logic formalism used in this thesis. Furthermore, we describe the A* search algorithm, which we will later use as an example on how to integrate our LTL framework into search algorithms.

2.1 Planning Task Representation

We consider planning tasks given in the STRIPS formalism (Fikes and Nilsson, 1972):

Definition 2.1 (STRIPS task)**.** A STRIPS planning task is a quadruple $\Pi = \langle V, A, I, G \rangle$, where

- V is a finite set of propositional variables.
- A is a finite set of actions $a = \langle \mathrm{pre}(a), \mathrm{add}(a), \mathrm{del}(a), \mathrm{cost}(a) \rangle$, where $\mathrm{pre}(a) \subseteq V$, $\mathrm{add}(a) \subseteq V$, $\mathrm{del}(a) \subseteq V$ and $\mathrm{cost}(a) \in \mathbb{N}_0$.
- $I \subseteq V$ is the initial state.
- $G \subseteq V$ is the set of goal variables.

A *state* s is defined as a set of variables $v \in V$, e.g. the initial state I. An action a is *applicable* to a state s if $\mathrm{pre}(a) \subseteq s$. When a is applied to s, the *successor state* is defined as follows: $\mathrm{succ}_a(s) := (s \setminus \mathrm{del}(a)) \cup \mathrm{add}(a)$. A sequence of actions $\langle a_0, a_1, \ldots a_n \rangle$ is applicable to s if a_0 is applicable to s, a_1 is applicable to $\mathrm{succ}_{a_0}(s)$ and so forth. When several actions are applied in a row we will use a short notation: $\mathrm{succ}_{\langle a_0,\ldots,a_n \rangle}(s) := \mathrm{succ}_{a_n}(...(\mathrm{succ}_{a_0}(s))...)$.

A sequence of actions $\rho = \langle a_0, \ldots, a_n \rangle$ applicable to a state s resulting in a state $\mathrm{succ}_\rho(s)$ is also called a *path* from s to $\mathrm{succ}_\rho(s)$. The *cost* of a

path is defined as the sum of the cost of each action application in the path (meaning that if an action is for example applied twice in a path, its costs also need to be counted twice).

A *goal state* is a state s which includes all goal variables: $G \subseteq s$. A *plan* π is a path from the initial state to a goal state. An *optimal plan* π_{opt} is a plan with the lowest possible cost, meaning that there exists no other plan $\pi^{'}$ with $\text{cost}(\pi^{'}) < \text{cost}(\pi_{\text{opt}})$.

2.2 A* search algorithm

The STRIPS formalism for describing planning tasks contains an implicit definition of a weighted directed graph, where nodes represent states and edges represent actions. An edge between node n_1 and n_2 exists, if the action a represented by the edge is applicable to the state s_1 represented by n_1, and the resulting state $s_2 = \text{succ}_a(s_1)$ is represented by n_2. This graph is called the *state space*. Based on this, many search algorithms build parts of the so-called *search space*, until they find a goal node (a node with a goal state). The search space is similar to the state space, but nodes are not only a representation of a state, they also hold other (partially path-dependent) information. It is build by *expanding* an already existing node (the first existing node is n_I with state I). Expanding a node n with associated state s means that for each action a which is applicable to s, a new node n' with state $\text{succ}_a(s)$ is created and an edge from n to n' with weight $\text{cost}(a)$ is added. This means the search space is actually a tree with n_I as root.

Search algorithms which explore the search space by expanding nodes mainly differ in how they choose which nodes to expand. They usually use a priority queue called *open list* in which they save nodes that are created but not yet expanded, and choose the node with highest priority for the next expansion. This is called *best-first search*. The A* algorithm (Hart et al., 1968) assigns each node n inserted in the open list a value $f = g(n) + h(n)$, where lower f values relate to higher priorities. The value $g(n)$ is the cost of the path from n_I to the current node, and $h(n)$ is the expected cost for reaching a cheapest reachable goal node from n. The value $g(n)$ of a child node n with parent n_p is the sum of $g(n_p)$ and $\text{cost}(a)$, where a is the action represented by the edge from n_p to n (for the initial node, $g(n_I) = 0$ is assigned). For calculating $h(n)$, a variety of *heuristics* exists.

A* can guarantee to find optimal plans if the used heuristic is *admissible*, meaning the heuristic never overestimates the costs for reaching the closest goal node. A* also uses a technique called duplicate elimination. This means, A* only keeps one node for each unique state in the search space, the node n

with lowest $g(n)$ (because it is desirable to reach a state with lowest possible cost).

A pseudo code of A* can be seen in Algorithm 1. The algorithm starts with creating a node n_I, assigning the initial state I to n_I, 0 as the g-cost $g(n_I)$ and the heuristic estimate for I as $h(n_I)$ (heuristic estimates are typically only state dependent). It then inserts n_I into the open list with priority $g(n_I) + h(n_I)$. The actual search is done in the while loop starting at line 8. As long as the open list is not empty, a node n with minimal f value is picked from the open list (line 9) and put in the closed list (line 10). If n is a goal node (line 11), the algorithm reconstructs the plan by backchaining through the parent nodes and returns the plan (line 12). If n is not a goal node, its children are generated: The algorithm loops over all actions a applicable to the state $s(n)$ associated with n (line 14) and creates children nodes with associated state $\text{succ}_a(s(n))$. Since A* uses duplicate elimination, not all children nodes are necessarily created. This is done in the following way:

For a successor state s_{succ}, a temporary g value for reaching s_{succ} over parent node n is calculated (line 16). Then, the algorithm checks if the state has been seen already. If not (line 17), a node n_{succ} is created (line 18), s_{succ} is associated with n_{succ} (line 19), $g(n_{\text{succ}})$ is set to g_{temp} (line 20), $h(n_{\text{succ}})$ is estimated (line 21), the parent pointer $\text{parent}(n_{\text{succ}})$ is set to n (line 22) and n_{succ} is inserted into the open list with priority $g(n_{\text{succ}}) + h(n_{\text{succ}})$ (line 23). If the state has been seen before in a node n', but the temporary g value is lower than $g(n')$ (line 33), then, instead of creating a new node, the algorithm simply updates the $g(n')$ value to g_{temp} and parent pointer $\text{parent}(n')$ to n (line 25 and 26). Since the heuristic estimate is typically only state dependent, it does not need to be recalculated. If n' is already in the closed list (line 27), it needs to be reopened by taking it out of the closed list (line 28) and inserting it again into the open list with the sum of $h(n')$ and the newly assigned $g(n')$ as priority (line 29). If n' is in the open list (line 30), the algorithm simply updates the priority of n' to the sum of $h(n')$ and the newly assigned $g(n')$ (line 31). If there exists a node n' with s_{succ} in the open and closed list with $g(n') \leq g_{\text{temp}}$, then the algorithm does nothing, since the existing node is equally good or better than the new found path to s_{succ}.

Finally, if the while loop should end without finding a solution, the algorithm will report that the problem is unsolvable (line 36).

Algorithm 1 A* search

1: $n_I \leftarrow$ new node
2: $s(n_I) \leftarrow I$
3: $g(n_I) \leftarrow 0$
4: $h(n_I) \leftarrow$ heuristic estimate for I
5: closed $\leftarrow$ []
6: open $\leftarrow$ new priority queue
7: insert n_I in open with priority $g(n_I) + h(n_I)$
8: **while** open not empty **do**
9:     $n \leftarrow$ remove minimum from open
10:     insert n in closed
11:     **if** n is a goal node **then**
12:         **return** reconstruct path from n
13:     **end if**
14:     **for all** actions a applicable to $s(n)$ **do**
15:         $s_{\text{succ}} \leftarrow \text{succ}_a(s)$
16:         $g_{\text{temp}} = g(n) + \text{cost}(a)$
17:         **if** there exists no n' with $s(n') = s_{\text{succ}}$ in open or closed **then**
18:             $n_{\text{succ}} \leftarrow$ new node
19:             $s(n_{\text{succ}}) \leftarrow s_{\text{succ}}$
20:             $g(n_{\text{succ}}) \leftarrow g_{\text{temp}}$
21:             $h(n_{\text{succ}}) \leftarrow$ estimate heuristic value of s_{succ}
22:             parent$(n_{\text{succ}}) \leftarrow n$
23:             insert n_{succ} in open with priority $g(n_{\text{succ}}) + h(n_{\text{succ}})$
24:         **else if** there exists n' with $s(n') = s_{\text{succ}}$ in open or closed and $g_{\text{temp}} < g(n')$ **then**
25:             $g(n') \leftarrow g_{\text{temp}}$
26:             parent$(n') \leftarrow n$
27:             **if** n' in closed **then**
28:                 remove n' from closed
29:                 insert n' in open with priority $g(n') + h(n')$
30:             **else**
31:                 change priority of n' to $g(n') + h(n')$
32:             **end if**
33:         **end if**
34:     **end for**
35: **end while**
36: **return** unsolvable

2.3 Linear Temporal Logic

The idea of creating a formalism for Temporal Logic was first proposed by Pnueli (1977) in order to provide a method for verifying the correctness of both serial and parallel programs. The basic idea is to express logical formulas where the variable assignment can change over time. Since change over time can be interpreted in several ways, there have been various approaches on how to formalize Temporal Logic (e.g. can there be concurrent actions, do actions have a duration and so on).

Since we will use Temporal Logic in classical, nontemporal planning, we chose to use Linear Temporal Logic (LTL). Linear Temporal Logic interprets temporal change as an (infinite) *sequence of worlds* $\boldsymbol{w} = \langle w_0, w_1, \dots \rangle$, where each world w_i is of a truth assignment for a set of propositional variables V ($w_i : V \to \{\mathrm{T}, \mathrm{F}\}$). As will be shown later, this setting can easily be adapted into the planning formalism. LTL exists both as Propositional LTL and First-Order LTL. We will use Propositional Linear Temporal Logic (PLTL) as described in Emerson (1990). PLTL extends Propositional Logic by four modal operators: $\Box$ (Always), $\Diamond$ (Eventually), $\bigcirc$ (Next) and $\mathcal{U}$ (Until). The following two definitions (based on Emerson (1990) but with different notation) describe syntax and semantics of PLTL:

Definition 2.2 (PLTL syntax)**.** The set of PLTL formulas over a set of propositional variables V is inductively defined as follows:

- $\top$ and $\bot$ are PLTL formulas.
- p is a PLTL Formula for $p \in V$.
- if φ and ψ are PLTL formulas, then $\neg\varphi$, $(\varphi \vee \psi)$,$(\varphi \wedge \psi)$, $\Box\varphi$, $\Diamond\varphi$, $\bigcirc\varphi$ and $(\varphi \mathcal{U} \psi)$ are PLTL formulas.

The order of precedence used in this thesis differs slightly from Emerson (1990): Unary operators bind the strongest ($\neg$, $\Box$, $\Diamond$, $\bigcirc$), followed by $\mathcal{U}$, followed by $\wedge$, and finally followed by $\vee$ as operator with weakest binding power. As an example, consider the following equivalence:

$$\Box a \vee \neg \bigcirc b \wedge c \mathcal{U} d \equiv (\Box a) \vee \Big(\big(\neg(\bigcirc b)\big) \wedge (c \mathcal{U} d) \Big)$$

Definition 2.3 (PLTL semantics)**.** Let φ be a PLTL formula over a set of propositional variables V, $\boldsymbol{w} = \langle w_0, w_1, \dots \rangle$ an infinite sequence of worlds where $w_i : V \to \{\mathrm{T}, \mathrm{F}\}$, and let $\boldsymbol{w^i} = \langle w_i, w_{i+1}, \dots \rangle$ denote an infinite subsequence of $\boldsymbol{w}$ for $i \geq 0$.

- if $\varphi = p$ where p is a propositional variable in V, then $\boldsymbol{w} \models p$ iff $w_0(p) = \mathrm{T}$
- if $\varphi = \neg\psi$, then $\boldsymbol{w} \models \varphi$ iff $\boldsymbol{w} \not\models \psi$
- if $\varphi = \psi_1 \vee \psi_2$, then $\boldsymbol{w} \models \varphi$ iff $\boldsymbol{w} \models \psi_1$ or $\boldsymbol{w} \models \psi_2$
- if $\varphi = \psi_1 \wedge \psi_2$, then $\boldsymbol{w} \models \varphi$ iff $\boldsymbol{w} \models \psi_1$ and $\boldsymbol{w} \models \psi_2$
- if $\varphi = \Box\psi$, then $\boldsymbol{w} \models \varphi$ iff for all $i \geq 0$: $\boldsymbol{w^i} \models \psi$
- if $\varphi = \Diamond\psi$, then $\boldsymbol{w} \models \varphi$ iff there exists a $i \geq 0$ where $\boldsymbol{w^i} \models \psi$
- if $\varphi = \bigcirc\psi$, then $\boldsymbol{w} \models \varphi$ iff $\boldsymbol{w^1} \models \psi$
- if $\varphi = \psi_1 \mathcal{U} \psi_2$, then $\boldsymbol{w} \models \varphi$ iff there exists a $i \geq 0$ where $\boldsymbol{w^i} \models \psi_2$ and for all $0 \leq j < i$, $\boldsymbol{w^j} \models \psi_1$
- if $\varphi = \top$ then $\boldsymbol{w} \models \varphi$
- if $\varphi = \bot$ then $\boldsymbol{w} \not\models \varphi$

For the remainder of this paper, we will refer to PLTL simply as LTL.

2.3.1 Evaluating LTL Formulas with progression

Sometimes we want to evaluate an LTL formula when we do not have the whole sequence of worlds $\boldsymbol{w} = \langle w_0, w_1, \dots \rangle$ given but only the beginning of that sequence $\boldsymbol{w[i]} = \langle w_0, w_1, \dots, w_i \rangle$. In these cases we can evaluate the formula *progressively* over $\boldsymbol{w[i]}$ (Bacchus and Kabanza, 2000). Progressing an LTL formula φ with a world w means that we build a new formula φ' which is satisfied by all sequences of worlds $\langle w_0, w_1, \dots \rangle$, where the same sequence of worlds preceded by w: $\langle w, w_0, w_1, \dots \rangle$ satisfies φ. In the above example, we could progress an LTL formula φ first with w_0, then progress the resulting formula with w_1 and so forth until w_i, and we would get a formula which is satisfied by all sequences of worlds $\langle w_{i+1}, \dots \rangle$ where $\langle w_0, w_1, \dots, w_i, w_{i+1}, \dots \rangle$ satisfies φ.

The use of progressing a formula φ over $\boldsymbol{w[i]}$ becomes apparent when the progressed formula can be proven to be unsatisfiable or a tautology. In these cases we know that any sequence of worlds starting with $\boldsymbol{w[i]}$ will never satisfy φ or respectively always satisfy φ.

Definition 2.4. progress(φ, w) must fulfill the following condition:

$$\langle w_1, w_2, \dots \rangle \models \text{progress}(\varphi, w_0) \text{ iff } \langle w_0, w_1, w_2, \dots \rangle \models \varphi\ .$$

In order to avoid a nested notation we will use a short notation for iterative progression:

$$\text{progress}_{\langle w_0, w_1, \ldots, w_i \rangle}(\varphi) := \text{progress}(\ldots \text{progress}(\varphi, w_0) \ldots, w_i)$$

Bacchus and Kabanza (2000) introduced progression rules for all First-Order LTL operators with bounded quantification and proved their correctness. The following definition includes all progression rules needed for Propositional LTL:

Definition 2.5 (Progression rules (Bacchus and Kabanza, 2000))**.** Let φ be an LTL formula over V which should be evaluated over a sequence of worlds starting with world w. Then progress(φ, w) is recursively defined as follows:

$\varphi = p \in V$	progress$(\varphi, w) := \top$ if $w \models p$, $\bot$ otherwise
$\varphi = f_1 \wedge f_2$	progress$(\varphi, w) :=$ progress$(f_1, w) \wedge$ progress(f_2, w)
$\varphi = f_1 \vee f_2$	progress$(\varphi, w) :=$ progress$(f_1, w) \vee$ progress(f_2, w)
$\varphi = \neg f$	progress$(\varphi, w) := \neg$progress(f, w)
$\varphi = \Box f$	progress$(\varphi, w) :=$ progress$(f, w) \wedge \varphi$
$\varphi = \Diamond f$	progress$(\varphi, w) :=$ progress$(f, w) \vee \varphi$
$\varphi = \bigcirc f$	progress$(\varphi, w) := f$
$\varphi = f_1 \mathcal{U} f_2$	progress$(\varphi, w) :=$ progress$(f_2, w) \vee ($progress$(f_1, w) \wedge \varphi)$

While progressing one can use the following equivalences to shorten the formula:

- $\varphi \wedge \bot \equiv \bot \wedge \varphi \equiv \bot$
- $\varphi \vee \top \equiv \top \vee \varphi \equiv \top$
- $\varphi \wedge \top \equiv \top \wedge \varphi \equiv \varphi$
- $\varphi \vee \bot \equiv \bot \vee \varphi \equiv \varphi$
- $\neg \top \equiv \bot$
- $\neg \bot \equiv \top$

Example 2.1. Consider a formula $\varphi = (a \mathcal{U} b) \vee (\bigcirc c \wedge \Box d)$, and a world $w : \{a \rightarrow \mathrm{F}, b \rightarrow \mathrm{F}, c \rightarrow \mathrm{T}, d \rightarrow \mathrm{T}\}$. We can calculate the progression of φ with w as follows:

$$
\begin{aligned}
\text{progress}(\varphi, w) &= \text{progress}(a\mathcal{U}b, w) \vee \text{progress}(\bigcirc c \wedge \Box d, w) \\
&= \Big(\text{progress}(a, w) \vee \big(\text{progress}(b, w) \wedge (a\mathcal{U}b)\big)\Big) \\
&\quad \vee \big(\text{progress}(\bigcirc c, w) \wedge \text{progress}(\Box d, w)\big) \\
&= \Big(\bot \vee \big(\bot \wedge (a\mathcal{U}b)\big)\Big) \vee \big(c \wedge (\text{progress}(d, w) \wedge \Box d)\big) \\
&= c \wedge (\top \wedge \Box d) \\
&= c \wedge \Box d
\end{aligned}
$$

As we will see in Chapter 3, the progression technique will be very important when using LTL in classical Planning, because during the search we will never have complete sequences of worlds.

Chapter 3

LTL in Classical Planning

3.1 Adaption of LTL for Classical Planning

Our goal is to describe control knowledge in LTL. Since we want to use our LTL framework in optimal planning, we are interested in control knowledge describing statements which are true in any optimal plan. Translated into the LTL setting this means that *LTL formulas representing control knowledge should be satisfied by any infinite sequence of worlds representing an optimal plan*. To define such a representation of an optimal plan, we first need to define what a world is in classical planning and then describe how plans can be expressed as infinite sequences of worlds.

In Chapter 2 we defined that in STRIPS states are represented as the set s of propositional variables which are true in this state. Since in LTL a world is defined as a truth assignment for a set of propositional variables, we can easily interpret a state as a world by assigning all propositional variables which are in the state set s the value T and all other propositional variables $V \setminus s$ the value F (where V is the set of all propositional variables that exist for the given planning task).

When trying to describe a plan as an infinite sequence of worlds, we have two problems: a plan is finite, and plans are described as a sequence of actions that are applied to a state, not a sequence of states. The latter problem is easily solved by replacing the sequence of actions with the corresponding sequence of states. Given a plan $\pi = \langle a_0, a_1, \dots, a_n \rangle$ applied to state I, the state sequence induced by this world is $\langle I, \text{succ}_{a_0}(I), \text{succ}_{\langle a_0, a_1 \rangle}(I), \dots, \text{succ}_{\pi}(I) \rangle$.

The most common solution to the former problem was proposed by Bacchus and Kabanza (2000). They argued that the goal state in a plan should be repeated indefinitely, since as soon as a goal state is reached, no more actions will be applied and thus the state will not change anymore. We will

use the same idea but extend it to not only plans but all paths that lead from any state to a goal state. We will use a short notation for the infinite sequence of worlds describing these paths:

Given a path $\rho = \langle a_0, a_1, \ldots, a_n \rangle$ which is applicable to a state s and results in a goal state $\mathrm{succ}_\rho(s)$, the following notation describes an infinite sequence of states (worlds), which starts with s, follows the path to $\mathrm{succ}_\rho(s)$ and repeats the last state indefinitely:

$$\boldsymbol{w^s_\rho} = \langle s, \mathrm{succ}_{a_0}(s), \mathrm{succ}_{\langle a_0,a_1\rangle}(s), \ldots, \mathrm{succ}_\rho(s), \mathrm{succ}_\rho(s), \ldots \rangle$$

We will call this notation the infinite extension of a path ρ.

With this notation, we can now define what an LTL formula must fulfill in order to represent control knowledge in optimal planning:

Definition 3.1 (LTL Formulas Describing Control Knowledge in Optimal Planning)**.** An LTL formula describing control knowledge for a STRIPS task $\Pi = \langle V, A, I, G \rangle$ must be satisfied by the infinite extension $\boldsymbol{w^I_{\pi_{\mathrm{opt}}}}$ of any optimal plan π_{opt} of the task Π.

Bauer and Haslum (2010) summarized some alternatives on how to deal with finite trace semantics for LTL and compared them to the infinite extension method, focusing on the question whether an LTL formula $\bigcirc\varphi$ should be satisfied at the end of a finite trace (i.e. if we progress a given formula over a finite trace and the resulting formula is $\bigcirc\varphi$). With the infinite extension method this would depend on whether φ is satisfied by the infinite extension of the last world in the finite trace.

One alternative proposed by Manna and Pnueli (1995) is to change the semantics of LTL to finite traces, i.e. $\Box\varphi$ is satisfied by a sequence of worlds $\langle w_0, \ldots, w_n \rangle$ iff for all $0 \leq i \leq n : \langle w_i, \ldots, w_n \rangle \models \varphi$ holds. The next operator is treated in a special way: a formula $\bigcirc\varphi$ is satisfied by a sequence of worlds ending with w_n only if there exists a next world w_i and if $\langle w_i, \ldots, w_n \rangle \models \varphi$. If no next world exists, $\bigcirc\varphi$ is not satisfied. They also introduce a "weak next" operator, which is the same as the normal next operator except that a formula with the weak next operator is also satisfied if no next world exists. Baier and McIlraith (2006) propose a similar finite trace semantics, but instead of the weak next operator they introduce a 0-ary FINAL operator which is only true in the last world of the finite trace. Bauer and Haslum (2010) showed that these two semantics are equivalent, since the weak next operator can be expressed by $\bigcirc\varphi \vee \mathrm{FINAL}$ and the FINAL operator by $\neg\bigcirc\mathrm{TRUE}$. They also showed that the satisfiability of those two semantics differs from the satisfiability of the infinite extension semantics, but only as soon as the weak next operator (or FINAL operator) is used.

Bauer and Haslum (2010) also mention two other semantics: LTL_3 and RV-LTL. The LTL_3 semantics adds *inconclusive* as a possible evaluation value besides $\top$ and $\bot$. So if for example a formula $\bigcirc\varphi$ should be evaluated over a finite trace where no next world exists, the evaluation would return "inconclusive". RV-LTL extends this idea and instead of "inconclusive" reports *possibly true* or *possibly false*, depending on whether the infinite extension semantics would evaluate the formula to $\top$ or $\bot$.

We decided to use the infinite extension semantics since we currently do not need added semantics such as weak next or inconclusive evaluation. It is however possible to change the framework to use one of the above mentioned semantics instead of infinite extension, should the need arise.

3.2 Evaluating LTL formulas during Search

As described in the previous section, we want to evaluate an LTL formula describing control knowledge over a sequence of states induced by a plan. We assume for the moment that control knowledge is given in the beginning of the search in form of a "global" LTL formula which must be satisfied by any optimal plan. We will show later how control knowledge can be added during the search.

While searching we cannot evaluate the given global LTL formula yet since we do not have a plan, but only nodes with states in a search tree. Since those node sequences can denote beginnings of plans, we can use the progression technique presented in Section 2.3.1 by progressing the given formula over the states sequence induced by the path from the initial node to each node. If this progressed formula should be equal to $\bot$ at any node, we know that the path to this node cannot be the beginning to an optimal plan (see Section 2.3.1) and we can prune the node from the search space. Furthermore, if the progressed formula is not equal to $\bot$, it will give us information on what requirements a path from the current node to the nearest goal node must fulfill in order to result in an optimal plan.

When progressing a given LTL formula, we will associate each node in the search tree with a separate LTL formula. We will use the notation φ_n to describe a formula which is associated to a node n. Given a global formula φ we associate the node n_I representing the initial state I with the formula $\varphi_{n_I} = \text{progress}(\varphi, I)$. Each successor node of the initial node is associated with the formula which results by progressing φ_{n_I} with the state of the successor node, and so forth. More formally:

Definition 3.2. For a node n with state s, parent node n_p and formula φ_{n_p} associated to n_p we assign the following formula to n:

$$\varphi_n = \text{progress}(\varphi_{n_p}, s)$$

Note that φ_n must be evaluated over a path starting with a successor state of s and not a path starting with s, since we progressed φ_n with s already,

Example 3.1. Consider a sequence of three nodes $\boldsymbol{n}[\mathbf{2}] = \langle n_0, n_1, n_2 \rangle$ with states $s(n_0) = \{a, b\}$, $s(n_1) = \{a, b, c\}$, $s(n_2) = \{a, d\}$ as shown in Figure 3.1, and a global LTL formula $\varphi = \Box a \wedge \Diamond c \wedge (\bigcirc e \vee (b\mathcal{U}d)$. We associate the following formulas with the nodes n_i:
n_0: $\varphi_{n_0} = \text{progress}(\varphi, s(n_0)) = \Box a \wedge \Diamond c \wedge (e \vee (b\mathcal{U}d))$
n_1: $\varphi_{n_1} = \text{progress}(\varphi_{n_0}, s(n_1)) = \Box a \wedge (b\mathcal{U}d)$
n_2: $\varphi_{n_2} = \text{progress}(\varphi_{n_1}, s(n_2)) = \Box a$

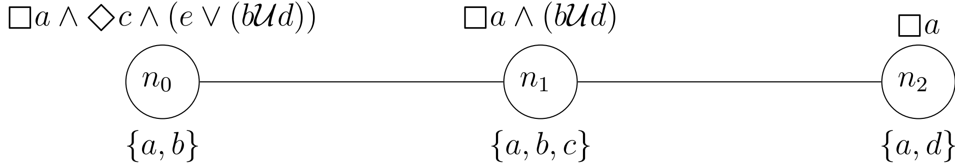


Figure 3.1: A Progression Example

Associating each node with its own LTL formula also gives us the possibility of adding control knowledge during the search. We will do this by progressing the formula from the parent node and adding new information to the progressed formula. Thus we must define a criterion for node-associated LTL formulas which ensures that if the node is part of an optimal plan, the continuation of the optimal plan satisfies the node-associated LTL formula.

Definition 3.3 (Optimal Plan Satisfiability Criterion for Node-associated LTL Formulas)**.** Consider a planning task $\Pi = \langle V, A, I, G \rangle$ with a node n in its search tree. The node n is associated with state s and an LTL formula φ_n, and was reached with cost $g(n)$. For all paths $\rho = \langle a_0, a_1, \ldots, a_n \rangle$ that are applicable to s and where $G \subseteq \text{succ}_\rho(s)$ and $g(n) + \text{cost}(\rho)$ equals the cost of any optimal plan of the task Π, we require

$$\boldsymbol{w}_{\boldsymbol{\rho}^+}^{\text{succ}_{\boldsymbol{\rho}[\mathbf{0}]}(\boldsymbol{s})} \models \varphi_n, \text{ where } \rho[0] = a_0 \text{ and } \rho^+ = \langle a_1, \ldots a_n \rangle$$

Theorem 3.1 (LTL-based node pruning)**.** Consider a node n_p with associated LTL formula φ_{n_p} which fulfills the Optimal Plan Satisfiability Criterion, and a node n with state s which is a successor of n_p. If the progression of φ_{n_p} with s results to $\bot$, the path to n cannot be the beginning of an optimal plan, and thus n can be pruned from the search space without loss of optimality.

Proof. Since φ_{n_p} fulfills the Optimal Plan Satisfiability Criterion, it is satisfied by any state sequence induced by a path that will result in an optimal plan when appended to n_p. Since the progression of φ_{n_p} with s is unsatisfiable, there can be no state sequence starting with s which satisfies φ_{n_p} (according to Definition 2.4), and thus any state sequence starting with s cannot describe a path that will result in an optimal plan when appended to n_p. But any path from n_p over n induces such a state sequence (since s is the state associated to n), thus no path over n can be the beginning of an optimal plan. $\square$

3.3 Merging LTL Formulas from Nodes with Identical State

In classical planning it often happens that several nodes share the same state. In these cases, it would be desirable to conjunct the LTL formulas associated to these nodes in order to gain more information for the individual nodes. This is however not always correct with our requirement to node-associated LTL formulas, as the following example will show.

Example 3.2. Consider two nodes n_1 and n_2 in the search tree of a planning task Π, which share the same state s and are reached with cost $g(n_1)$, respectively $g(n_2)$, as shown in Figure 3.2. The associated LTL formulas φ_{n_1} and φ_{n_2} both fulfill the Optimal Plan Satisfiability Criterion. Further assume that $g(n_1) < g(n_2)$ and that there exists a path $\rho = \langle a_0, a_1, \dots, a_n \rangle$ applicable to s, where $\mathrm{succ}_\rho(s)$ is a goal state and $g(n_1) + \mathrm{cost}(\rho) = c_{\mathrm{opt}}$ (c_{opt} being the optimal plan cost for Π). Note that $g(n_2) + \mathrm{cost}(\rho)$ can never be equal to the optimal plan cost, since $g(n_1) < g(n_2)$.

In Definition 3.3 we required that the LTL formula associated to a node n with state s needs to be satisfied by the infinite extension of all paths applicable to s which reach a goal state with overall optimal costs. Since there can be no such state sequence for n_2, φ_{n_2} might not be satisfied by any state sequence. The formula associated to n_1 however needs to be satisfied by $\boldsymbol{w}_{\boldsymbol{\rho}^+}^{\mathrm{succ}_{\rho[0]}(\boldsymbol{s})}$. If we assign $\varphi_{n_1} \wedge \varphi_{n_2}$ to n_1, we cannot guarantee that φ_{n_2} would be satisfied by $\boldsymbol{w}_{\boldsymbol{\rho}^+}^{\mathrm{succ}_{\rho[0]}(\boldsymbol{s})}$ and thus we cannot guarantee that the formula

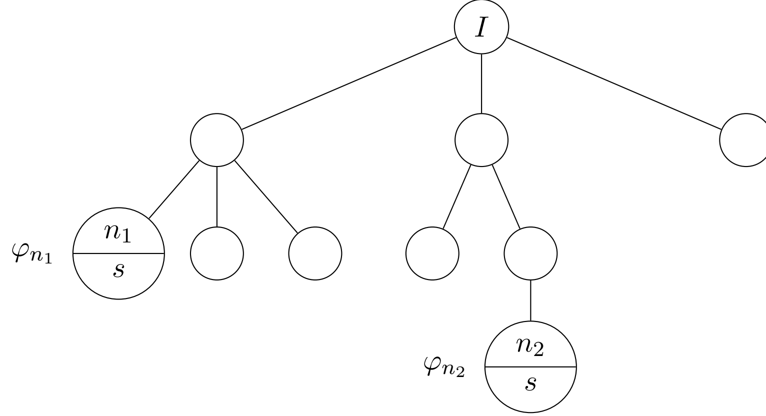


Figure 3.2: A search tree with two nodes with identical state

assigned to n_1 is satisfied by all state sequences which represent a path to a goal state with overall optimal plan costs. This would violate the requirement from Definition 3.3 and is therefore not a correct assignment.

The example above is based on different g-costs. When the g-costs are the same, it is indeed possible to conjunct the two LTL formulas, which we will show with the proof of the following theorem.

Theorem 3.2. Consider two nodes n_1 and n_2 in the search tree of a planning task Π, which share the same state s and are reached with cost $g(n_1)$, respectively $g(n_2)$. The associated formulas φ_{n_1} and φ_{n_2} both fulfill the Optimal Plan Satisfiability Criterion for their respective node. If $g(n_1) = g(n_2)$ then $\varphi_{n_1} \wedge \varphi_{n_2}$ fulfills the Optimal Plan Satisfiabilty Criterion for both n_1 and n_2.

Proof. Since the role of n_1 and n_2 is symmetric in the theorem, it is sufficient to show that $\varphi_{n_1} \wedge \varphi_{n_2}$ satisfies the Optimal Plan Satisfiability Criterion for n_1:

We need to show that $\boldsymbol{w}_{\boldsymbol{\rho}^+}^{\mathrm{succ}_{\boldsymbol{\rho}[0]}(\boldsymbol{s})} \models \varphi_{n_1} \wedge \varphi_{n_2}$ for all paths ρ applicable to s (the state of n_1), where $\mathrm{succ}_\rho(s)$ is a goal state and $g(n_1) + \mathrm{cost}(\rho) = c_{\mathrm{opt}}$ (c_{opt} being the optimal plan cost of Π). Consider an arbitrary such path ρ. It is sufficient to show separately that the infinite extension $\boldsymbol{w}_{\boldsymbol{\rho}^+}^{\mathrm{succ}_{\boldsymbol{\rho}[0]}(\boldsymbol{s})}$ (1) satisfies φ_{n_1} and (2) satisfies φ_{n_2}, because from this follows that $\boldsymbol{w}_{\boldsymbol{\rho}^+}^{\mathrm{succ}_{\boldsymbol{\rho}[0]}(\boldsymbol{s})} \models \varphi_{n_1} \wedge \varphi_{n_2}$.

1. This is already granted in the theorem since φ_{n_1} fulfills the Optimal Plan Satisfiability Criterion for n_1.

2. Since φ_{n_2} fulfills the Optimal Plan Satisfiability Criterion for n_2, the infinite extension of all ρ' satisfies φ_{n_2}, where ρ' is applicable in s (the state of n_2, $\text{succ}_{\rho'}(s)$ is a goal state and $g(n_2) + \text{cost}(\rho') = c_{\text{opt}}$. Since n_1 and n_2 have the same state and $g(n_1) = g(n_2)$, ρ is such a ρ' and thus its infinite extension satisfies φ_{n_2}

□

3.4 Integration in A* search

Since pruning based on LTL formulas that fulfill the Optimal Plan Satisfiabilty Criterion preserves optimality, it can be used in the A* algorithm.

If we look at Algorithm 1 we see two possibilities where to include our LTL evaluation. The first option is to evaluate the formula once the node is expanded. This will often result in substantially less LTL formula evaluations since there are usually many more nodes generated than actually expanded. The second option is to evaluate the formula as soon as a node is generated. While it will be more effort and take more memory to do this, we argue that it is in most cases more beneficial for several reasons:

- The evaluation of the LTL formula is time efficient in comparison to most heuristic estimates (as long as the formula is not too long).
- If a node which reaches a new state is pruned, the heuristic estimate for that state does not need to be computed.
- The open list is smaller if nodes are pruned (potentially saving memory and reducing time required for open list operations depending on the implementation).
- If we progress the formula only when the node is expanded, we cannot gain additional information from other nodes with identical state anymore. A* prunes nodes with states that are already reached with lower or equal cost, before putting the nodes into the open list, thus when expanding a node there is no other node with the same state in the search space.

If however the formulas are too long, memory usage and evaluation time can become a problem. In this case, it might be more suitable to evaluate the formula only when expanding the node.

Algorithm 2 A* search with LTL pruning

```
1: n_I ← new node
2: s(n_I) ← I
3: g(n_I) ← 0
4: h(n_I) ← estimate heuristic value of I
5: closed ← []
6: open ← new priority queue
7: φ_{n_I} ← progress(φ, I)
8: insert n_I in open with priority g(n_I) + h(n_I)
9: while open not empty do
10:   n ← remove minimum from open
11:   insert n in closed
12:   if n is a goal node then
13:     return reconstruct path from n
14:   end if
15:   for all actions a applicable to s(n) do
16:     s_succ ← succ_a(s)
17:     g_temp = g(n) + cost(a)
18:     φ_temp ← simplified version of progress(φ_n, s_succ)
19:     if φ_temp equals ⊥ then
20:       continue with next successor
21:     end if
22:     if new control knowledge φ_new can be deduced then
23:       φ_temp ← φ_temp ∧ φ_new
24:     end if
25:     if there exists no n′ with s(n′) = s_succ in open or closed then
26:       n_succ ← new node
27:       s(n_succ) ← s_succ
28:       g(n_succ) ← g_temp
29:       h(n_succ) ← estimate heuristic value of s_succ
30:       parent(n_succ) ← n
31:       φ_{n_succ} ← φ_temp
32:       insert n_succ in open with priority g(n_succ) + h(n_succ)
```

Algorithm 2 A* search with LTL pruning (continued)

```
33:       else if there exists n' with s(n') = s_succ in open or closed and
                  g_temp < g(n') then
34:          g(n') ← g_temp
35:          φ_n' ← φ_temp
36:          parent(n') ← n
37:          if n' in closed then
38:             remove n' from closed
39:             insert n' in open with priority g(n') + h(n')
40:          else
41:             change priority of n' to g(n') + h(n')
42:          end if
43:       else if there exists n' with s(n') = s_succ in open or closed and
                  g_temp = g(n') then
44:          φ_n' ← φ_n' ∧ φ_temp
45:       end if
46:    end for
47: end while
48: return unsolvable
```

A conceptual example of how to integrate LTL evaluation into A* can be seen in Algorithm 2. It is based on the same pseudocode as in Algorithm 1; added lines are colored red.

In line 7 the progression of a given global formula with the initial state is associated with n_I. When expanding a node n, for each successor state s_{succ} the progression of φ_n with s_{succ} is calculated, simplified and saved temporarily as φ_{temp} (line 18). If the simplified progression is equal to $\bot$, the node will be pruned by just continuing with the next successor (lines 19 and 20). If not, we can enhance φ_{temp} with additional control knowledge fulfilling the Optimal Plan Satisfiability Criterion, if such control knowledge exists (line 22 and 23).

If there exists no other node in the search space with s_{succ} as state and thus the node n_{succ} is created, φ_{temp} is assigned to n_{succ} (line 31). If there exists a node n' but g_{temp} (the costs of reaching s_{succ} over expanding n) is lower than $g(n')$, then φ_{temp} is assigned to n', since n' will represent the new path over n (line 35). Else, if g_{temp} is equal to $g(n')$, then the conjunction $\varphi_{n'} \wedge \varphi_{\text{temp}}$ will be assigned to n', which is valid according to Theorem 3.2 (line 43 and 44). If g_{temp} is larger than $g(n')$, we do not need to do anything, since we cannot gain any information from φ_{temp} in this case.

Algorithm 2 could be improved in two ways:

1. If the simplified progression of an LTL formula φ_{n_p} with state s is equal to $\perp$, we could check the open list for nodes n which have the same state s and are reached with higher or equal costs than reaching s over n_p. These states could also be pruned from the open list, because technically we could assign $\text{progress}(\varphi_{n_p}, s) = \perp$ respectively $(\text{progress}(\varphi_{n_p}, s) \wedge \varphi_n) = \perp$ to n according to Theorem 3.2 and thus show that the path to n cannot denote an optimal plan

2. For each pruned node we could save the information that a node with state s and g-cost $g(n)$ has been pruned. If A* would later on create a node n' with state s and $g(n') \geq g(n)$, this node could also be pruned (for the same reasons as above).

Chapter 4

Unnecessary Action Applications in LTL

This chapter presents a concrete example of control knowledge in form of a pruning technique that we call *Unnecessary Action Applications*. It then shows how this control knowledge can be translated into LTL formulas which can be used in optimal planning. Unnecessary Action Applications are based on *intended effects*, which were introduced by Karpas and Domshlak (2012).

4.1 Intended Effects

Karpas and Domshlak (2012) defined intended effects as a way of describing the potentially useful progress a path from the initial state to any state has made. Their definition is built on the terms *achiever* and *consumer*. Given a state s reached by a path $\rho = \langle a_0, a_1, \ldots, a_n \rangle$, an achiever of a variable $p \in s$ is the last action application a_i with $i \leq n$ that made p true. In an path $\rho' = \langle a_{n+1}, \ldots, a_m \rangle$ applied to s, the consumer of $p \in s$ is the first action application a_j with $j > n$ which has p in its preconditions. Given a node n with state s which is reached at some point in a plan π, they further define π_1 as the path that reached n from the initial state and π_2 as the path that will reach a goal state from n. An intended effect at node n is now the set of variables which is achieved by an action application in π_1 and will be consumed by an action application π_2.

They approximate intended effects by building a shortcut library for each node n, considering the entire path $\rho = \langle a_0, a_1, \ldots, a_n \rangle$ that reached that node from the initial state. First, they build a so-called *causal structure of* ρ. This is a graph where the action applications a in ρ are the nodes, and there is a directed arc from each action application i to an action application j,

where a_i is an achiever of a variable that is consumed by a_j. With this graph they can define *isolated chains*, that is a chain of action applications in the graph that does not have any outgoing arcs to an action application outside the isolated chain. As example, consider a path $\rho = \langle a_0, a_1, a_2, a_3 \rangle$ where the causal structure would have the following arcs: $a_0 \rightarrow a_1$ and $a_1 \rightarrow a_3$. The chain $a_0 \rightarrow a_1 \rightarrow a_3$ would be such a chain, but also $a_1 \rightarrow a_3$ and the single action applications a_3 and a_2. These isolated chains can be removed from ρ resulting in a path ρ' which is still applicable to the initial state.

For each of the paths ρ' found by analyzing the causal structure of ρ where $\text{cost}(\rho') < \text{cost}(\rho)$, a shortcut is saved denoting that a state $\text{succ}_{\rho'}(I)$ can be reached with cost $\text{cost}(\rho')$ (they showed that under certain assumptions a ρ' with equal costs as ρ can also be utilized, but we omit this for the sake of simplicity). For all sets of variables that are true in both the state s belonging to n and in a state of the shortcut library, it can be concluded that this cannot be the entire intended effect of ρ for any continuation of an optimal plan, since it can be accomplished with a shorter path. Thus, for all states in the shortcut library s', all sets of variables that are true in s, but not in s', are possible intended effects. If there exists no such set of variables for a particular s', (meaning that $s \subseteq s'$), then ρ cannot have any intended effects.

4.2 Unnecessary Action Applications

Intended effects are inspired by the idea that every action must be there for a reason. Based on this we define the notion of an *Unnecessary Action Application*. Intuitively an action application in a path is unnecessary if it achieves nothing that is useful for the continuation of the path. By achieving something useful, we mean that the action a_i is the achiever of at least one variable v which is consumed by a future action a_j, or the achiever of a goal variable.

Example 4.1. Consider the following planning task (see Figure 4.1):

- $V = \{i, v, x, g_1, g_2, g_3\}$
- $A = \{a_1, a_2, a_3, a_4\}$ with $a_1 = \langle \emptyset, \{x, g_1, g_2\}, \{i\}, 1 \rangle$; $a_2 = \langle \emptyset, \{v, x\}, \emptyset, 1 \rangle$; $a_3 = \langle \{v\}, \{g_3\}, \{g_1, g_2\}, 1 \rangle$ and $a_4 = \langle \{g_3\}, \{g_1, g_2\}, \emptyset, 1 \rangle$
- $I = \{i\}$
- $G = \{g_1, g_2, g_3\}$

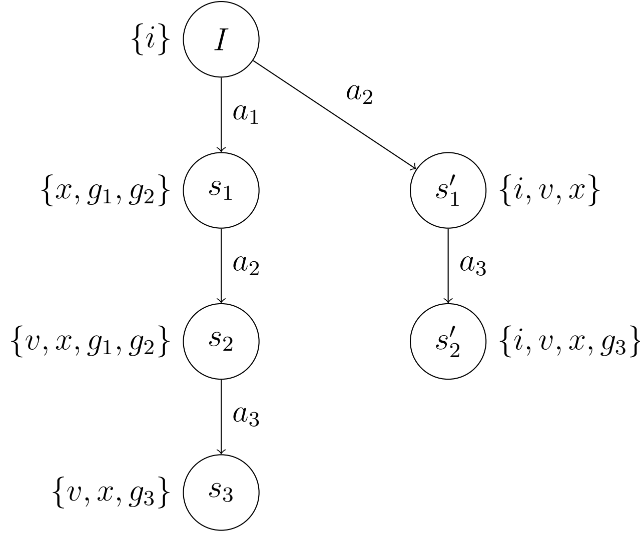


Figure 4.1: Illustration of Example 4.1

Since action a_1 adds two of the three goal variables it seems to be a good idea to start with this action, resulting in $s_1 = \{x, g_1, g_2\}$. Next we will execute a_2, because it s the only action (aside from a_1 which we just executed) that is applicable, resulting in $s_2 = \{v, x, g_1, g_2\}$. Then we can only proceed by executing a_3, resulting in $s_3 = \{v, x, g_3\}$. At this point we can see however that the execution of a_1 was unnecessary. Action a_1 added three variables: x, g_1 and g_2. Variable x was made true again by a_2, g_1 and g_2 were deleted by a_3, and none of the three variables was used as a precondition by a_2 or a_3. Thus, a_1 did not add any effects that were of use. If we would just execute $\langle a_2, a_3 \rangle$ we would arrive at a state $s'_2 = \{i, v, x, g_3\}$. While it is not the same state as s_3, we can reach any state from s'_2 that we could reach from s_3, since $s_3 \subset s'_2$.

Formally we say that in a plan $\pi = \langle a_0, \ldots, a_i, \ldots, a_n \rangle$ of planning task Π, the application of a_i at step i is unnecessary if for all of its add-effects $e \in \text{add}(a_i)$:

- e is falsified or made true by another action application a_j with $j > i$, before it is used as a precondition for any action application a_k with $i < k \leq j$,
- or (if no such action a_j exists) e is still true in the goal state but is not a goal variable and has never been used as a precondition for another action application a_k with $k > i$.

If the first case is true, then the intended effect technique by Karpas and Domshlak (2012) can detect a_i as an isolated chain when analyzing the node n reached by $\rho = \langle a_0, \ldots, a_j \rangle$, where j is the step where the last add-effect of a_i is falsified or made true again. The shortcut saved by removing a_i will be a state $s' \supseteq s$ (where s is the state associated to n), and thus it can be concluded that there exists no intended effect on path ρ.

In order to avoid the special treatment for effects that are never falsified and never added again, we change $\Pi = \langle V, A, I, G \rangle$ by adding an artificial goal action which has all goal variables in its precondition, deletes all variables and adds a new variable which will be the new goal. More formally, we define a new planning task $\Pi' = \langle V', A', I, G' \rangle$ with $V' = V \cup \{v_{\text{goal}}\}$, $A' = A \cup \{a_{\text{goal}} = \langle G, v_{\text{goal}}, V, 0 \rangle\}$ and $G' = \{v_{\text{goal}}\}$. This definition implies that any plan π' for such a Π' is equal to a plan $\pi = \langle a_0, \ldots, a_n \rangle$ for Π extended with a_{goal}, which also means that for any optimal π' of Π', the corresponding π of Π is also optimal, and vice versa. For the remainder of this chapter we will assume that planning task Π is represented by the corresponding task Π' with an artificial goal action.

Definition 4.1 (Unnecessary Action Applications)**.** Consider a planning task Π'. Let $\pi = \langle a_0, \ldots, a_i, \ldots, a_n = a_{\text{goal}} \rangle$ be a plan with only one occurrence of a_{goal}. An action $a_i \neq a_{\text{goal}}$ is *applied unnecessarily at step i* iff for all of its add-effects $e \in \text{add}(a_i)$, e has not been a precondition in an action application a_k with $i < k \leq j$, where j is the first step after i where the applied action a_j has e in its add- or delete-effects.

Note that a_{goal} cannot be unnecessary. The artificial goal action ensures that achievers of the original goal variables are necessary, and that for any effect e of an action $a \neq a_{\text{goal}}$, there will be a future action (a_{goal}) which will delete e (the only effect that is not deleted by a_{goal} is v_{goal}, but the only action having v_{goal} in its add-effects is a_{goal}).

Our goal is to describe control knowledge which is true for any optimal plan. The statement *any action application must be necessary* is not true for all optimal plans: Consider two plans $\pi = \langle a_0, \ldots a_{i-1}, a_i, a_{i+1}, \ldots, a_n \rangle$ with unnecessary action application a_i, and $\pi' = \langle a_0, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n \rangle$ (this is for example possible if a_i adds a single variable $v \neq v_{\text{goal}}$ which is never used as a precondition for any action application a_j with $j > i$. If π' is an optimal plan and $\text{cost}(a_i) = 0$, then π is also an optimal plan, even though it has an unnecessary action application. If we would however say that *any application of an action with cost > 0 must be necessary*, then this statement is true for any optimal plan:

Theorem 4.1. If in a plan $\pi = \langle a_0, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n \rangle$ the action application a_i is unnecessary and $\text{cost}(a_i) > 0$, then π cannot be an optimal plan.

Proof. We prove this theorem by showing that there exists a plan $\pi' = \langle a_0, \ldots a_{i-1}, a_{i+1}, \ldots, a_n \rangle$ with lower costs. For this we need to prove that (1) π' is applicable to the initial state I, (2) $G \subseteq \text{succ}_{\pi'}(I)$ and (3) $\text{cost}(\pi') < \text{cost}(\pi)$:

1. If π' was not applicable to the initial state, a_i must be an achiever of a variable v which is consumed by an action a_j with $j > i$. But a_i is an unnecessary action application and thus cannot be an achiever for any variable $v \in \text{pre}(a_j)$.

2. The only goal variable v_{goal} is added only by $a_n = a_{\text{goal}}$. Since π' has a_n, as last action (and π' is applicable to the initial state), $\text{succ}_{\pi'}(I)$ must have v_{goal} in its set of variables.

3. Plan π' consists of exactly the actions of π minus action a_i. Thus $\text{cost}(\pi') = \text{cost}(\pi) - \text{cost}(a_i)$. Since $\text{cost}(a_i) > 0$, we can conclude that $\text{cost}(\pi') < \text{cost}(\pi)$.

□

4.3 Adaption to LTL

In order to construct an LTL formula which forbids unnecessary action applications (for actions a with $\text{cost}(a) > 0$), we need to include actions into the definition of a world. Calvanese et al. (2002) did this by introducing a propositional variable for each action. To ensure that at any time exactly one action is applied, they constructed additional formulas expressing this constraint. We will use a similar approach, but we will denote the action that reached the current world separately from the variables which are true in the current world. Thus we do not need to add additional formulas to ensure exactly one action is applied at any time.

Definition 4.2 (Modified Definition of World)**.** Let V be the set of variables that are true in state s of a given search node n, and a be the action that reached this node. Then the world describing n is defined as a tuple $w_i = \langle a, V \rangle$. We will further on refer to the action belonging to world w_i as a_{w_i} and respectively the set of variables belonging to world w_i as V_{w_i}. For the initial state we define an empty action a_{init}, where $\text{pre}(a_{\text{init}}) = \emptyset$, $\text{add}(a_{\text{init}}) = \emptyset$ and $\text{del}(a_{\text{init}}) = \emptyset$.

With this modified definition of a world we need to define new semantics for $\boldsymbol{w} \models \varphi$ where φ is an propositional variable or an action. All other semantics rules remain unchanged.

Definition 4.3 (Changes in LTL Semantics with Modified Worlds)**.** Given an LTL formula φ and an infinite sequence of worlds $\boldsymbol{w} = \langle w_0, w_1, \dots \rangle$:

- if $\varphi = p$ where p is a propositional variable, then $\boldsymbol{w} \models p$ iff $V_{w_0}(p) = \mathrm{T}$
- if $\varphi = a$ where a is an action, then $\boldsymbol{w} \models p$ iff $a = a_{w_0}$

We want to construct an LTL formula which describes that in a plan $\pi = \langle a_0, \dots a_n = a_{\text{goal}} \rangle$ all action applications a_i at step i (with $0 \leq i < n$) must be necessary if $\text{cost}(a_i) > 0$. First we need to define what a *necessary* action application is. Above we defined that in π the action application $a_i \neq a_{\text{goal}}$ at step i is unnecessary if for all of its add-effects $e \in \text{add}(a)$, e is falsified or made true again by a future action application a_j with $j > i$ and for all a_k with $i < k \leq j$, e is not in the preconditions of a_k. Since for all add-effects e of any action application $a_i \neq a_{\text{goal}}$ at step i there always exists an action application a_j at step $j > i$ which deletes e (at the latest, this a_j is a_{goal}), an action application is only necessary if for at least one add-effect e an action application a_k at step k with $i < k \leq j$ exists which has e in its preconditions.

We can reformulate this the following way: An action application a_i is necessary at step i, if for at least one add-effect e there exists an action application a_k at step k with $k > i$ which has e in its preconditions, and until then no action application a_l at step l with $i < l < k$ has e in its add- or delete-effects. Since the definition of a necessary action application a_i at step i is only dependent on the path $\langle a_{i+1}, \dots a_n \rangle$ to a goal node, we can formulate this definition as a node-associated LTL formula:

Definition 4.4 (Necessary Action Applications in LTL)**.** Consider a node n in the search space of a planning task $\Pi' = \langle V, A, I, G \rangle$, where n has associated state s and was reached over a path $\rho_1 = \langle a_0, \dots a_i \rangle$. For any plan π of Π' resulting from extending ρ_1 with a path $\rho_2 = \langle a_{i+1}, a_{i+2}, \dots, a_n \rangle$, a_i is necessary in π if the following formula is satisfied by $\boldsymbol{w}_{\boldsymbol{\rho_2^+}}^{\text{succ}_{\boldsymbol{\rho_2[0](s)}}}$ (otherwise a_i is unnecessary):

$$\bigvee_{e \in \text{add}(a_i)} [(e \wedge \bigwedge_{a \in A \text{ with } e \in \text{add}(a)} \neg a) \mathcal{U} \bigvee_{a \in A \text{ with } e \in \text{pre}(a)} a]$$

Note that the LTL formula requires that at least one effect is actually used in a precondition, since $a \mathcal{U} b$ requires that b is eventually satisfied. The condition

that no action application can delete e until it is used as a precondition was simplified to the equivalent condition that e must stay true.

Theorem 4.2. In a planning task Π' with artificial goal action, consider a node n with state s and which was reached with action a. If $\mathrm{cost}(a) > 0$, adding the LTL formula from Definition 4.4 for action a (denoted by φ_a) to φ_n fulfills the Optimal Plan Satisfiability Criterion.

Proof. The Optimal Plan Satisfiability Criterion requires that for any path ρ which is applicable to s and where $\mathrm{succ}_\rho(s)$ is a goal state and $g(n) + \mathrm{cost}(\rho)$ is equal to the optimal plan cost, the sequence of worlds $\boldsymbol{w}_{\boldsymbol{\rho}^+}^{\mathrm{succ}_{\boldsymbol{\rho}[0]}(\boldsymbol{s})}$ must satisfy φ_a.

Let ρ' be an arbitrary such ρ. Assume that its infinite extension does not satisfy φ_a. Then a is unnecessary for the plan π resulting by appending ρ' to n. We showed in Theorem 4.1 that a plan with an unnecessary application of an action a_i with cost > 0 cannot be optimal. Since $\mathrm{cost}(a) > 0$ and a is unnecessary, π cannot be an optimal plan, and thus $g(n) + \mathrm{cost}(\rho')$ cannot be equal to the optimal plan costs. This is a contradiction to the definition of ρ', and from this follows that its infinite extension must satisfy φ_a. □

From Theorem 4.2 follows, that for each node n reached with an action a with $\mathrm{cost}(a) > 0$, we can add the LTL formula from Definition 4.4 for a to φ_n without loss of optimality.

Chapter 5

Experimental Results

5.1 Implementation

The described LTL framework and the Unnecessary Action Application pruning technique have been implemented on top of the Fast Downward Planning System (Helmert, 2006), a state-of-the-art heuristic forward search planner.

5.1.1 LTL framework

Fast Downward uses a finite domain representation (FDR) instead of STRIPS, which represents sets of mutually exclusive STRIPS variables by one variable with a finite domain (the domain is defined by the set of STRIPS variables mapped in this mutex group and an optional "none of those" value, which means that none of the STRIPS variables is true). Thus we could not directly use STRIPS variables in our LTL formulas. Instead an LTL formula describing a STRIPS variable holds a FDR variable-value pair, which represents the STRIPS variable.

The LTL formulas are created by a factory, which also takes care of saving the formulas. Formulas are created incrementally, meaning that first the atomic formulas are created and then the more complex formulas, which only hold pointers to their subformulas. Conjunctions and disjunctions can have n subformulas, where $n \geq 2$. Progression is done in a top-down approach, as described in Definition 2.4.

In order to limit memory usage, any particular formula is only saved once in the factory. The factory takes care of detecting if a formula exists already when we try to create it from the factory; and if yes simply returns the pointer to the already existing formula. This also extends to conjunctive and disjunctive formulas: When creating a conjunctive or disjunctive formula with n subformulas, the subformulas are ordered according to their hashvalue

and the framework can then detect if there exists such an conjunctive or disjunctive formula already. For example, if there is an LTL formula $a \wedge b \wedge c$ in the factory and we ask the factory to create a formula $b \wedge a \wedge c$, the factory would recognize them to be identical and return the pointer for $a \wedge b \wedge c$ without actually creating a new formula.

5.1.2 Unnecessary Action Application

In order to implement the Unnecessary Action Application pruning technique, we need to have access to the add-effects and preconditions of an action. However, in FDR actions are not defined in STRIPS semantics, they have so-called Prevail and PrePost tuples. A Prevail is a pair $\langle \text{variable}, \text{value} \rangle$ and is equivalent to a precondition for a mutex-group variable that is not changed by the action. A PrePost is a triple $\langle \text{variable}, \text{pre-value}, \text{post-value} \rangle$ which makes a variable change to a new value. It is thus precondition and effect in one (if the pre-value is -1, then the PrePost is a pure effect). A PrePost can optionally have a vector of Prevails which describes conditional effects, meaning that this particular variable change only takes effect if the vector of Prevails is true in the state where the action is applied to (else the action is still applicable, but the variable change in the PrePost with unsatisfied conditional effects will not be executed). We do however not yet support conditional effects.

Add-effects are recognized easily: all executed PrePosts which change a variable to a value other than "none of those", is an add-effect. Preconditions however can take several forms:

- A Prevail tuple describes a precondition of a STRIPS variable and can easily be recognized
- In a PrePost tuple $\langle \text{var}, \text{pre-val}, \text{post-val} \rangle$, var and pre-val denote a precondition if pre-val $\neq -1$. This can also easily be recognized.
- In a PrePost, an element of the vector of Prevails describing conditional effects usually describes a precondition for the conditional PrePost effect. Since we do not support conditional effects, we do not need to consider those preconditions.

In order to optimize our code, the implementation of the Unnecessary Action Applications formula differs from the formula defined in Definition 4.4. Instead of enumerating two sets of actions (those that have e in their add-effects and those that have e in their precondition), we introduced two new

abbreviations as pseudo LTL operators: Use and Add. Both operators can only be used in combination with a FDR pair $\langle$mutex group variable, value$\rangle$. The progression of the Use operator on a pair $\langle$var, val$\rangle$ returns $\top$ if an action is applied which has a Prevail $\langle$var, val$\rangle$ or a PrePost with var as variable and val as pre-value; $\bot$ otherwise. The progression of the Add operator is similar: If an action is applied which has a PrePost with var as variable and val as post-value, $\top$ is returned; $\bot$ otherwise. While semantically identical, this notation takes less space and the progression is easier to compute.

5.2 Results

We tested the Unnecessary Action Application pruning technique (UAA) on the benchmarks from the deterministic sequential optimization track of the International Planning Competition (IPC) 2011 [1]. We omitted the tidybot domain, because it has negative preconditions, and we assumed in UAA that all preconditions are positive. We tested UAA in combination with the *LM-Cut heuristic* $h^{\text{LM-Cut}}$ (Helmert and Domshlak, 2009) as a state-of-the-art heuristic, and with the *Max heuristic* $h^{\max}$ (Bonet and Geffner, 2001) as a less informed heuristic.

Unfortunately, UAA could not prune a single node on these benchmarks with either configuration. After analyzing the domains we realized that there simply do not exist any operators on these domains which could potentially be unnecessary. The reason for this is that if a STRIPS variable occurs as a delete effect in an action, it is also in the precondition of this action. For example consider the transport domain, where vehicles with limited capacity deliver packages to destinations. The domain has three types of actions: *drive*, *pick-up* and *drop*. Drive moves a vehicle from one location l_1 to another location l_2. The only delete effect is that the vehicle is not at location l_1 anymore, but obviously this is also a precondition of this action. Pick-up loads a package in a vehicle v and decreases the usable capacity of v. There are two delete effects: the package is not at l_1 anymore (but in the vehicle) and the old capacity for v is not true anymore. But again, both of these delete-effects are also in the precondition of the action. Drop is basically the inverse action of pick-up and also has no delete-effects which are not preconditions at the same time.

While the pruning technique did not actually prune anything, we can still draw conclusions on how efficient the LTL framework in itself is. For this,

[1] http://www.plg.inf.uc3m.es/ipc2011-deterministic/

Table 5.1: Memory usage with $h^{\text{LM-Cut}}$

Domain	no UAA	with UAA	Factor
barman-opt11-strips (4)	719880	2010888	2.79
elevators-opt11-strips (13)	243592	451704	1.85
floortile-opt11-strips (6)	283948	1011108	3.56
nomystery-opt11-strips (14)	122952	148088	1.20
openstacks-opt11-strips (17)	11502180	12734540	1.11
parcprinter-opt11-strips (13)	242188	424064	1.75
parking-opt11-strips (2)	57988	73888	1.27
pegsol-opt11-strips (17)	598792	775356	1.29
scanalyzer-opt11-strips (12)	416568	1200640	2.88
sokoban-opt11-strips (20)	444116	547364	1.23
transport-opt11-strips (6)	64368	81672	1.27
visitall-opt11-strips (11)	2134956	2291988	1.07
woodworking-opt11-strips (11)	407456	739132	1.81
Sum (146)	**17238984**	**22490432**	**1.30**

we ran the benchmarks for both heuristics without UAA and compared the results. We were mainly interested in how search time and memory usage increase when UAA is used.

5.2.1 Memory Efficiency

Tables 5.1 and 5.2 show a summary of the memory usage in the tested domains and configurations (the memory usage is calculated by adding the memory usage of each single problem in the domain that was solved by both configurations). As we can see, the difference between memory usage when using UAA and when not using it is very varied. For example, with $h^{\text{LM-Cut}}$, the floortile domain uses 3.56 times as much memory with UAA as without, while the visitall domain only uses 1.07 times as much memory with UAA as without. One reason for this is that floortile has a higher ratio of actions/problem than visitall (the ratio of actions/problem for problems that were solved by both configurations is 188 for floortile and 66.2 for visitall). More unique actions in a problem mean that there exists more unique LTL formulas describing the necessary action application criterion. However, it seems that the most prominent reason for high memory consumption is the number of evaluations. We see this best when comparing the absolute memory difference to the number of evaluations. The biggest absolute memory difference for the $h^{\text{LM-Cut}}$ configurations is in the barman domain. This is also by far

Table 5.2: Memory usage with h^{max}

Domain	no UAA	with UAA	Factor
barman-opt11-strips (4)	826596	2076376	2.51
elevators-opt11-strips (3)	60440	181520	3.00
nomystery-opt11-strips (8)	240592	897588	3.73
openstacks-opt11-strips (16)	9405504	10642760	1.13
parcprinter-opt11-strips (4)	27096	30292	1.12
pegsol-opt11-strips (17)	840044	1124756	1.34
scanalyzer-opt11-strips (3)	24608	102192	4.15
sokoban-opt11-strips (20)	1024964	1324108	1.29
transport-opt11-strips (6)	141664	365044	2.58
visitall-opt11-strips (16)	14852184	15191692	1.02
Sum (97)	**27443692**	**31936328**	**1.16**

the domain with the highest evaluations/problem ration (see Table 5.3).

This correlation seems to imply that there are many nodes which have a unique LTL formula. We believe this could be improved in our implementation: When building LTL formulas for a node in the Unnecessary Action Application pruning technique, we progress the parent formula and build the new UAA formula for the applied action. We then create a new conjunction with the two previous formulas as subformulas. This leads to a nested conjunction looking like $a \wedge (b \wedge (c \wedge (d \wedge e)))$. If there would be another formula $b \wedge (a \wedge (c \wedge (d \wedge e)))$, then our framework would not identify the two as being the same formula because of the nesting. If we would avoid nesting the formulas, the framework could identify the two formulas as being identical, and we would save memory.

This problem should also be considered for other implementations of LTL control knowledge, because when adding control knowledge during the search we will always end up with a conjunction: We defined that control knowledge can be added by progressing the parent formula and adding more control knowledge, which leads to a formula looking like

$$b \wedge \text{progress}(c \wedge \text{progress}(d, s_0), s_1)$$

This formula is equivalent to

$$b \wedge \text{progress}(c, s_1) \wedge \text{progress}(\text{progress}(d, s_0), s_1)$$

because the progression of $\text{progress}(a \wedge b, s)$ is equivalent to $\text{progress}(a, s) \wedge \text{progress}(b, s)$. If we nest these conjunctions, then the framework cannot recognize symmetries and creates more unique formulas than needed.

Table 5.3: Evaluations with $h^{\text{LM-Cut}}$

Domain	Evaluations
barman-opt11-strips (4)	1614166.25
elevators-opt11-strips (13)	84426.71
floortile-opt11-strips (6)	198754.83
nomystery-opt11-strips (14)	1728.80
openstacks-opt11-strips (13)	848018.63
parcprinter-opt11-strips (13)	1311.40
parking-opt11-strips (2)	95692.04
pegsol-opt11-strips (17)	82661.29
scanalyzer-opt11-strips (12)	3233.60
sokoban-opt11-strips (20)	27027.59
transport-opt11-strips (6)	12153.84
visitall-opt11-strips (10)	516.12
woodworking-opt11-strips (11)	16034.78

5.2.2 Search Time

Tables 5.4 and 5.5 show a summary of the average search time in the tested domains and configurations (the depicted number is the average search time on all problems of the domain where both configurations found a solution). Since no pruning was achieved with UAA, it is obvious that the search times with UAA are higher. The difference is however higher than expected. We saw two possible explanations as to why the search time increased so much:

- the progression is more expensive than expected
- when there is a big amount of nodes, the fetching and saving of the formulas takes long (they are stored in a hashtable)

We ran a profiler on an example problem in order to find out where the code spends so much time when using UAA. We chose problem 14 from the nomystery domain because in the $h^{\max}$ configurations this problem used a lot more memory and time with UAA than without. The profiler showed that with the $h^{\max}$ configuration with UAA, 81.95% of the search time was spent progressing the formula, while only 0.04% was spent with fetching and 0.08% with saving the formulas.

This shows that the progression of the formula is the bottleneck in our implementation. When progressing a formula, the framework first creates a tentative formula by simply applying the progression rules from Definition 2.5. In a second step, it simplifies the tentative formula with the equivalences listed in Section 2.3.1. It then deletes the tentative formula and saves

Table 5.4: Search Time with $h^{\text{LM-Cut}}$

Domain	no UAA	with UAA	Factor
barman-opt11-strips (4)	233.59	851.74	3.65
elevators-opt11-strips (13)	8.41	24.45	2.91
floortile-opt11-strips (6)	9.68	39.09	4.04
nomystery-opt11-strips (14)	0.98	1.04	1.06
openstacks-opt11-strips (13)	14.05	26.34	1.88
parcprinter-opt11-strips (13)	60.48	0.55	1.15
parking-opt11-strips (2)	135.22	138.12	1.02
pegsol-opt11-strips (17)	2.97	3.90	1.31
scanalyzer-opt11-strips (12)	5.05	5.83	1.16
sokoban-opt11-strips (20)	3.31	3.48	1.05
transport-opt11-strips (6)	5.14	5.73	1.11
visitall-opt11-strips (10)	0.30	0.32	1.06
woodworking-opt11-strips (11)	3.79	5.73	1.51
Geometric mean (141)	**5.56**	**8.59**	**1.54**

Table 5.5: Search Time with h^{max}

Domain	no UAA	with UAA	Factor
barman-opt11-strips (4)	60.48	1662.42	27.49
elevators-opt11-strips (3)	3.12	163.16	52.27
nomystery-opt11-strips (8)	0.61	2.25	3.69
openstacks-opt11-strips (13)	17.54	30.10	1.72
parcprinter-opt11-strips (4)	0.10	0.33	3.27
pegsol-opt11-strips (17)	3.00	4.44	1.48
scanalyzer-opt11-strips (3)	0.39	4.00	10.26
sokoban-opt11-strips (20)	2.33	2.70	1.16
transport-opt11-strips (6)	2.72	23.57	8.67
visitall-opt11-strips (9)	0.22	0.34	1.57
Geometric mean (87)	**1.78**	**8.64**	**4.85**

the simplified formula. The profiler shows that 28.61% of the time spent progressing the formula is used for deleting the tentative formula, and 28.19% for creating it. This could be avoided by simplifying the formula on the go when progressing it and thus avoiding to create a tentative formula which must again be deleted.

We argued in chapter 3 that the evaluation time of LTL formulas is small in comparison to most heuristics. As we see in Table 5.4 and 5.5 this is only partially true. Since h^{max} is a heuristic that is calculated very fast, the evaluation time of LTL formulas is very large in comparison. The $h^{LM\text{-}Cut}$ heuristic takes longer to calculate, thus most domains in the $h^{LM\text{-}Cut}$ configuration do not take much more time for searching with UAA than without. But there are some domains where the time increase is still very large. We assume that in those domains, the formulas are longer in average, resulting in longer progression time. We do not have a measurement for formula size implemented in the framework, but when simply printing out the formulas during the search we saw that formulas from the floortile domain are much longer than formulas in domains like nomystery and sokoban.

We can also try to estimate how long the UAA formulas can become. The formulas consist of the UAA formula for the newly applied action and the progressed formula for the previously applied actions, for example: $\varphi_{a_2} \wedge \text{progress}(\varphi_{a_1} \wedge \text{progress}(\varphi_{a_0}, s_0), s_1)$. The UAA formula of an action is a disjunction over Until formulas for each add-effect of the action. The (simplified) progression of an Until formula $a\mathcal{U}b$ is either $\top$ if b is true, $\bot$ if a and b is false, or the formula itself if a is true but not b. Since the UAA formula is a disjunction of these formulas, the progression of it will return true as soon as one add-effect has been used, otherwise the disjunction of the Until-formulas for all add-effects which are still true and not added again. The formula associated with a node consists now of a conjunction of the newest UAA formula and the progression of the old UAA formulas. This means the length of the formula depends on two factors:

- how many previous actions have not yet been detected necessary (increasing the size of the conjunction)
- how many add-effects these unclassified actions have that are still true and not added again (increasing the size of the disjunction for each action)

Thus, if a domain has many actions which have a lot of add-effects (it is hard to tell how long they can stay true and are not added again, so we just take the absolute number of add-effects as measurement), and a lot of

actions are independent of each other (meaning the add-effects of one action are not needed as precondition of another action), it seems likely that in this domain, the node-associated LTL formulas from UAA grow rather long. In the floortile domain, there exists two actions types which paint a tile in a certain color. These effects are never used as precondition, but only as a goal variable; nor are they falsified. Thus, all these paint actions can never be declared unnecessary or necessary during the search, meaning that those actions will pile up in the conjunction of the node-associated LTL formula. In the domain nomystery, which describes a transport problem with three action types load, unload and drive, the actions depend much more on each other, and thus an action is faster detected as being necessary or unnecessary.

5.2.3 UAA with a Specifically Designed Domain

In order to still be able to test the Unnecessary Action Application pruning technique, we designed a new domain which should be more suitable to demonstrate the pruning technique (meaning actions have delete effects which are not at the same time preconditions). The domain describes a minigame consisting of tiles in a grid that can be activated. Given an activated tile t_{ij} with two opposing neighbors, one of its neighbors can be activated, but the other (opposing) neighbor will be deactivated. As an example, Listing 5.1 shows the action which activates the left neighbor ("(right ?t1 ?t2)" denotes that t1 is right of t2), and Figure 5.1 shows an exemplary application of this action.

Listing 5.1: Definition for Action *activate-left*

```
(:action activate-left
 :parameters (?t1 ?t2 ?t3)
 :precondition (and (activated ?t2) (right ?t1 ?t2) (right ?t2
    ?t3))
 :effect (and (activated ?t3) (not(activated ?t1)) ) )
```

A total of 45 problems with increasing difficulty were generated, where some problems might be unsolvable.

The domain was again tested with a total of 4 configurations for optimal planning: $h^{\text{LM-Cut}}$ without UAA, $h^{\text{LM-Cut}}$ with UAA, $h^{\max}$ without UAA and $h^{\max}$ with UAA. Table 5.6 and 5.7 show the number of evaluations in each problem for $h^{\text{LM-Cut}}$ without and with UAA, respectively $h^{\max}$ without and with UAA. They also show how many nodes could be pruned in total with UAA.

Table 5.6: Evaluations and Pruning in Minigame Domain with $h^{\text{LM-Cut}}$

Problem	no UAA	with UAA	Factor	pruned with UAA
prob01.pddl	31	31	1.00	0
prob02.pddl	86	81	0.94	11
prob03.pddl	12	12	1.00	0
prob04.pddl	1670	1646	0.99	392
prob05.pddl (unsolvable)	18177	18075	0.99	15804
prob06.pddl	167	149	0.89	33
prob07.pddl	4289	4186	0.98	1385
prob08.pddl	954	929	0.97	196
prob09.pddl	994	979	0.98	231
prob10.pddl (unsolvable)	32354	32055	0.99	27996
prob11.pddl (unsolvable)	32354	32129	0.99	27767
prob12.pddl (unsolvable)	32354	32059	0.99	26538
prob13.pddl (unsolvable)	32354	32015	0.99	26966
prob14.pddl (unsolvable)	32354	32055	0.99	27996
prob15.pddl	499	483	0.97	86
prob16.pddl	3510	3432	0.98	595
prob17.pddl	39183	38238	0.98	9267
prob18.pddl	2554	2513	0.98	318
prob19.pddl	5028	4719	0.94	1127
prob20.pddl (unsolvable)	432723	431546	1.00	646980
prob21.pddl	144334	140009	0.97	45347
prob22.pddl	3644334	3514872	0.96	2045521
prob23.pddl	553546	536464	0.97	204018
prob24.pddl	491726	475355	0.97	195833
prob29.pddl	372750	351744	0.94	116694
prob32.pddl	862992	838451	0.97	193929
prob33.pddl	445058	434844	0.98	86145
prob34.pddl	6324423	6030014	0.95	1887638
prob35.pddl	2577325	2470621	0.96	719727

t_{00}	t_{10}	t_{20}	t_{30}
t_{01}	t_{11}	t_{21}	t_{31}
t_{02}	t_{12}	t_{22}	t_{32}
t_{03}	t_{13}	t_{23}	t_{33}

(a) initial state

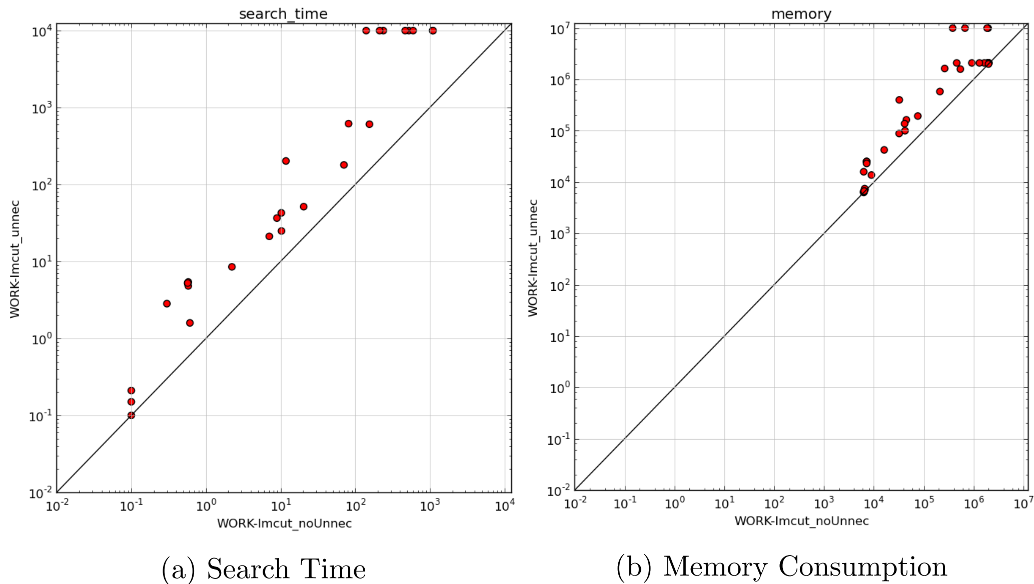
t_{00}	t_{10}	t_{20}	t_{30}
t_{01}	t_{11}	t_{21}	t_{31}
t_{02}	t_{12}	t_{22}	t_{32}
t_{03}	t_{13}	t_{23}	t_{33}

(b) after *activate-left(*t_{21}*,* t_{11}*,* t_{01}*)*

Figure 5.1: Exemplary Illustration of *activate-left*

Table 5.7: Evaluations and Pruning in Minigame Domain with h^{max}

Problem	no UAA	with UAA	Factor	pruned with UAA
prob01.pddl	141	132	0.94	14
prob02.pddl	342	332	0.97	40
prob03.pddl	23	22	0.96	1
prob04.pddl	9219	8874	0.96	3764
prob05.pddl (unsolvable)	18177	18078	0.99	8561
prob06.pddl	1958	1898	0.97	382
prob07.pddl	9858	9623	0.98	4356
prob08.pddl	11497	10945	0.95	2815
prob09.pddl	4833	4659	0.96	1818
prob10.pddl (unsolvable)	32354	32293	1.00	12387
prob11.pddl (unsolvable)	32354	32293	1.00	12421
prob12.pddl (unsolvable)	32354	32293	1.00	12305
prob13.pddl (unsolvable)	32354	32294	1.00	12140
prob14.pddl (unsolvable)	32354	32293	1.00	12387
prob15.pddl	13897	13449	0.97	3879
prob16.pddl	131556	123495	0.94	59328
prob18.pddl	122692	118721	0.97	27415
prob19.pddl	335708	315778	0.94	85731



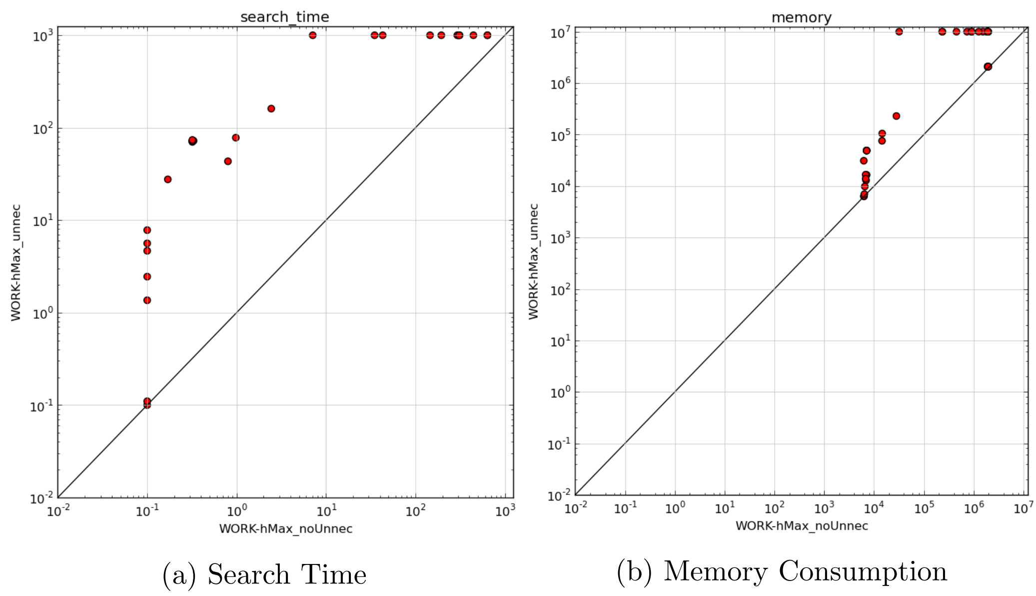
(a) Search Time

(b) Memory Consumption

Figure 5.2: Search Time and Memory Consumption with $h^{\text{LM-Cut}}$

The results show, that UAA can indeed help the search by pruning nodes. It is noteworthy however that the actual amount of evaluations saved is much smaller than the pure amount of pruned nodes. This is because evaluations are done only once for each state; and for many pruned nodes, the search later finds a cheaper path to the corresponding state without unnecessary action applications and thus evaluates that state anyway. It is also interesting, that while UAA could prune more nodes in the h^{max} configuration, the actual percentage-wise gain of avoided evaluations is no higher than in the $h^{\text{LM-Cut}}$ configuration.

Since the amount of saved evaluations is rather small, the Unnecessary Action Application pruning technique was not very helpful in this case either, as shown in Figure 5.2 and 5.3. This is because the actual time spent evaluating the LTL formulas is bigger than the time saved by avoiding the other evaluations of the node. However, with a few optimizations in our implementation these results should improve. The figures also show that UAA performs much worse with h^{max}. This is because the search generates more nodes with a less informed heuristic and we need to do an LTL evaluation for all these nodes which is computationally much more expensive than calculating the h^{max} heuristic.



(a) Search Time

(b) Memory Consumption

Figure 5.3: Search Time and Memory Consumption with h^{max}

Chapter 6

Conclusion

This master's thesis introduced a general framework which shows how the LTL formalism can be used for describing control knowledge, and how LTL formulas representing control knowledge can be used in heuristic forward search. Furthermore we introduced a criterion that all LTL formulas describing control knowledge in optimal planning should fulfill in order to preserve optimality if they are used for pruning. With the Unnecessary Action Application pruning technique we showed a concrete example how such control knowledge can be adapted to LTL and used in the search. While the pruning technique did not achieve positive results, its results still showed the advantages and disadvantages of our framework. We expect that with a few optimizations and with different control knowledge, our framework is capable of achieving much better results.

6.1 Future work

There are a couple of already discussed improvements which can be added to our current implementation of the LTL framework, such as more efficient saving and progressing of the LTL formulas, and adaptions to the LTL integration in the A* search. We believe these improvements can considerably increase the efficiency of our framework.

While the Unnecessary Action Application pruning technique seems to add no useful information in commonly used planning domains, the LTL framework in itself provides a number of opportunities. We only showed a pruning technique as application of the framework, but it would also be possible to develop heuristics which return an actual goal distance estimate based on this framework. There has been previous work showing how landmarks and their orderings can be defined through LTL formulas (Wang et al., 2009).

The integration into our LTL framework should be rather straightforward. It remains open however, how to build a heuristic estimate out of these LTL formulas. Wang et al. (2009) build finite state automata from their LTL formulas and used them to enhance the FF heuristic by adding those automata to the relaxed planning graph crated by FF. But since relaxed exploration ignores all delete effects this could even be counterproductive with a general LTL formula (i.e. if the formula describes $\neg e$ for a STRIPS variable e), and thus is not very suitable for general use.

While landmarks could also be implemented as a pure pruning-technique, its uses are doubtful. Most landmarks and their orderings are derived in such a way that the orderings are correct in any reachable state. An exception are "reasonable orderings", but they are rather a recommendation and if a reasonable ordering is not fulfilled in a path, this does not necessarily mean that the path cannot be the beginning of a plan. Thus pruning based on reasonable orderings would be incorrect.

The presented LTL framework is most efficient when the formulas used are short and no big variety of unique formulas exist. An interesting approach which would fulfill those criteria is to analyze a planning problem on whether it makes sense to falsify reached goal variables. If we take the logistics domain as example (where trucks and airplanes deliver packages to destinations), it cannot be optimal to take a package away from its goal location. In this case we could add an LTL formula $\Box g$ to any node reaching a goal variable g which should not be falsified anymore. The memory consumption would be minimal, since the LTL framework only needs to save one always-formula for each goal state that should not be falsified once achieved. The difficulty of this idea is to find a formal criterion which goal states should always stay true once achieved.

Finally, the LTL framework could also utilize a new feature of PDDL 3.0 called *State Trajectory Constraints* (Gerevini and Long, 2005). These hard constraints are encoded in modal-logic expressions (such as *always* φ and *sometime-after* φ ψ) and added to the problem description. We think that these constraints can easily be translated into our LTL framework and can then be used for pruning or for heuristic estimates.

Bibliography

Fahiem Bacchus and Froduald Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1):123–191, 2000.

Jorge A Baier and Sheila A McIlraith. Planning with first-order temporally extended goals using heuristic search. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI 2006)*, volume 21, pages 788–795, 2006.

Andreas Bauer and Patrik Haslum. LTL goal specifications revisited. In *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010)*, pages 881–886, 2010.

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

Diego Calvanese, Giuseppe De Giacomo, and Moshe Y Vardi. Reasoning about actions and planning in LTL action theories. In *Proceedings of the 8th International Conference on the Principles of Knowledge Representation and Reasoning (KR 2002)*, pages 593–602, 2002.

E. Allen Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 995–1072. 1990.

Richard E Fikes and Nils J Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3): 189–208, 1972.

Alfonso Gerevini and Derek Long. Plan constraints and preferences in PDDL3. Technical Report RT-2005-08-47, Dipartimento di Elettronica per l'Automazione, Università degli Studi di Brescia, 2005.

Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

Malte Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research (JAIR)*, 26:191–246, 2006.

Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS 2009)*, pages 162–169, 2009.

Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research (JAIR)*, 14:253–302, 2001.

Erez Karpas and Carmel Domshlak. Optimal search with inadmissible heuristics. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS 2012)*, pages 92–100, 2012.

Jonas Kvarnström and Patrick Doherty. TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence*, 30(1-4):119–169, 2000.

Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*, volume 2. Springer, 1995.

Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on the Foundations of Computer Science (FOCS 1977)*, pages 46–57, 1977.

Julie Porteous, Laura Sebastia, and Jrg Hoffmann. On the extraction, ordering, and usage of landmarks in planning. In *Proceedings of the 6th European Conference on Planning (ECP 2001)*, pages 37–48, 2001.

Letao Wang, Jorge Baier, and Sheila McIlraith. Viewing landmarks as temporally extended goals. In *Proceedings of the ICAPS Workshop on Heuristics for Domain Independent Planning (HDIP 2009)*, pages 49–56, 2009.