Simulation and Implementation of Temporal Logic-based Motion Planning for Autonomous Vehicles

August, 2014

HAUKUR INGI HEIDARSSON

Master’s Degree Project
Stockholm, Sweden 2014

XR-EE-RT 2014:016
Simulation and Implementation of Temporal Logic-based
Motion Planning for Autonomous Vehicles
— Diploma Thesis —

Haukur Ingi Heidarsson

Automatic Control Laboratory
School of Electrical Engineering, KTH Royal Institute of Technology, Sweden

Supervisors
Meng Guo
Dr. D. V. Dimarogonas
KTH Stockholm

Examiner
Dr. D. V. Dimarogonas
KTH Stockholm

Stockholm, August, 2014

1haukurh@kth.se
Abstract

This thesis focuses on temporal logic-based motion planning for autonomous vehicles. Specifically planning based on Linear Temporal Logic statements. This type of motion planning allows for automatic generation of correct by construction controllers that implement missions defined in a language that is both quite expressive and easy to understand.

The main contribution of this thesis is to provide a framework of reusable Python modules that implement algorithms for the production of plans based on Linear Temporal Logic and efficient online revising of existing plans when the environment changes.

The general case of offline planning is presented as well as several options for online planning based on partially known and dynamic environments. The different online planning methods are simulated and their feasibility in different scenarios is discussed. Finally, implementation concerns and future directions are discussed.
# Table of Contents

## List of Figures

1. Introduction
   1.1 Motivation .................................................. 1
   1.2 Previous Work ............................................... 1
   1.3 Main Contribution .......................................... 2
   1.4 Thesis structure ............................................ 2

2. Background
   2.1 Linear Temporal Logic ....................................... 3
      2.1.1 Syntax .................................................. 3
      2.1.2 Semantics .............................................. 3
      2.1.3 Examples ............................................... 4
      2.1.4 Limitations ............................................ 4
      2.1.5 Model Checking ........................................ 5
   2.2 Graphs ....................................................... 5
      2.2.1 Finite Automata ........................................ 5
      2.2.2 Search Algorithms ..................................... 6

3. Nominal Case
   3.1 Buchi ......................................................... 9
      3.1.1 Buchi construction ..................................... 10
   3.2 Finite Transition System (TS) ............................... 10
   3.3 Product Transition System ................................. 13
   3.4 Planning ..................................................... 16
   3.5 Optimal Planning ........................................... 16

4. Online Planning ................................................ 21
   4.1 Dynamic environments ....................................... 21
      4.1.1 Changes caused by the agent ......................... 21
   4.2 Superposition of Buchi states ............................. 23
   4.3 Product Graph Revision .................................... 23
Table of Contents

4.4 Alternative: Product-less planning ............................................ 24
4.5 Dealing with insufficient information ......................................... 26
4.6 Dynamic Search ........................................................................ 29
   4.6.1 Shortest Path Trees ............................................................... 29
   4.6.2 Dynamic Dijkstra ................................................................. 30

5 Implementation ........................................................................... 35
   5.1 Graph representation ............................................................... 35
   5.2 Buchi generation .................................................................... 36
      5.2.1 Guard parser .................................................................... 37
   5.3 Product generation ................................................................. 38
   5.4 Planning ................................................................................. 40
      5.4.1 Revision ........................................................................... 40
      5.4.2 Agent Model ...................................................................... 41
      5.4.3 Simulations ...................................................................... 42
      5.4.4 Performance ..................................................................... 42
      5.4.5 Example usage: Basic ......................................................... 43
      5.4.6 Example usage: Agent ......................................................... 45

6 Discussion .................................................................................... 53
   6.1 Multi-Agent ............................................................................. 53
      6.1.1 Collective .......................................................................... 53
      6.1.2 Individuals ....................................................................... 53
   6.2 Future directions ..................................................................... 54
      6.2.1 Better partitioning .............................................................. 54

References ...................................................................................... 57
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Graph types (from left to right): undirected, undirected with edge weights, directed and directed with edge weights.</td>
<td>6</td>
</tr>
<tr>
<td>3.1</td>
<td>An accepting run is composed of a prefix and a suffix.</td>
<td>10</td>
</tr>
<tr>
<td>3.2</td>
<td>Examples of the different kinds of transition systems that could be used for planning.</td>
<td>12</td>
</tr>
<tr>
<td>3.3</td>
<td>An example of the product of a Buchi and a transition system. The <em>sleep</em> and <em>wake</em> propositions have been abbreviated as <em>s</em> and <em>w</em>.</td>
<td>15</td>
</tr>
<tr>
<td>3.4</td>
<td>An example of the kinds of plans generated by a regular DFS based planner and Dijkstra based one.</td>
<td>20</td>
</tr>
<tr>
<td>4.1</td>
<td>The average planning times for two versions of the optimal Dijkstra planner. One uses ahead-of-time product generation and the other the iterative post algorithm in [10]. The benchmark was done by randomly generating a maze problem as described in chapter 5.4.4. Ten mazes were generated for each size on the x-axis and the average planning time computed for each size. Note that the problem in this example favors the iterative algorithm since the planner is unlikely to explore the whole product.</td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>Exploration</td>
<td>28</td>
</tr>
<tr>
<td>4.3</td>
<td>Examples of valid safety specifications</td>
<td>28</td>
</tr>
<tr>
<td>5.1</td>
<td>Expression tree for ( \neg(a \lor b) \lor a )</td>
<td>37</td>
</tr>
<tr>
<td>5.2</td>
<td>Agent model</td>
<td>41</td>
</tr>
<tr>
<td>5.3</td>
<td>A run with the revising NDFS planner.</td>
<td>49</td>
</tr>
<tr>
<td>5.4</td>
<td>A run with the revising Dijkstra planner.</td>
<td>50</td>
</tr>
<tr>
<td>5.5</td>
<td>The relative planning time required by the three types of planners. All of them use the iterative method of product generation. Like in figure 4.1 this benchmark was done by randomly generating a set of mazes of each size on the x-axis and computing the average planning time for each planner.</td>
<td>51</td>
</tr>
</tbody>
</table>
The goal of this thesis is to explore the field of Temporal-logic based motion planning and to create a framework for generating such plans.

1.1 Motivation

We would like to be able to create controllers that fulfill complex specifications for large and complicated systems. The naive approach for creating a completely custom controller for each combination of system dynamics and control specification is both cumbersome and error-prone. A better approach is to use symbolic planning wherein one writes a high level specification for the controller in terms of predicates over the environment or state-space and generates it algorithmically in a way that guarantees that the specifications are fulfilled. Linear Temporal Logic (LTL) serves as an adequate specification language for a variety of specifications and there exists theory that allows us to apply it to automatic controller generation. In short, the benefits of using LTL formalism over the naive approach are:

1. Controller specifications can be written in a relatively easy to understand language.
2. Specifications can be changed without requiring a massive effort.
3. The surface area for modeling and control errors is reduced to a well defined interface.

1.2 Previous Work

Linear Temporal Logic was first introduced by Amier Pnueli in his 1977 paper titled ”The Temporal Logic of Programs” wherein he considers it for the pur-
pose of automatic program verification. Since then LTL has been used for model checking [1] but more recently it has been applied to the automatic generation of hybrid controllers for dynamical systems [6]. An excellent introduction to model-checking based approaches for planning and control can be found in "Symbolic Planning and Control of Robot Motion" by Belta and Bichi [2]. Therein, they provide a high-level overview of symbolic planning using LTL as well as different specification languages.

The methods presented in [6] and [2] are what I refer to in this paper as the nominal approach to motion planning. Others, including Meng Guo in [5], have considered cases where the environment over which the predicates are evaluated is not perfectly known ahead of time so that, since the planning operation is an expensive one, the agent (robot) must be able to "patch" the plan in a way that is cheaper than re-planning from scratch.

1.3 Main Contribution

The main contribution in this thesis and the project in general has been to write a framework of Python modules that

1. Implement automatic plan generation for specifications written as Linear Temporal Logic formulas.
2. Consider plan optimality by using shortest-path algorithms.
3. Offer an agent based planning model that supports environment updates and efficient re-planning.

1.4 Thesis structure

This thesis is divided into the following chapters

<table>
<thead>
<tr>
<th>Background</th>
<th>Linear Temporal Logic is introduced and defined and necessary graph concepts are introduced.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal Case</td>
<td>The nominal case of symbolic planning using LTL is introduced as well as optimality considerations.</td>
</tr>
<tr>
<td>Online Planning</td>
<td>The nominal case is extended to include dynamic environments.</td>
</tr>
<tr>
<td>Implementation</td>
<td>The framework implementation is discussed.</td>
</tr>
<tr>
<td>Discussion</td>
<td>Multi-agent planning and future directions.</td>
</tr>
</tbody>
</table>
2 Background

2.1 Linear Temporal Logic

Linear Temporal Logic (LTL) is an extension of propositional logic that adds temporal operators. That is operators on propositional logic expressions that introduce temporal ordering to their operands.

We use LTL formulas to specify motion plans because it translates fairly well into natural language. The goal is that non-experts should be able to come up with plans in LTL that can then be used to generate correct by construction controllers.

2.1.1 Syntax

The basic syntax of LTL is composed of the boolean “true” literal, the logical and ($\land$) and negation ($\neg$) operators, labels that can either be true or false ($= \neg$true), as well as two new temporal operators: next ($\sigma$) and until (U). Other common operators that can be composed from these basic ones are or ($\lor$), implication ($\rightarrow$), equivalence ($\leftrightarrow$) and xor ($\oplus$). The details of how these composite operators are defined can be found on page 232 in [1]. There are two additional temporal operators whose definitions I repeat here. These operators are the eventually ($\Diamond := \text{true}U\phi$) and always ($\Box := \neg\Diamond\neg\phi$).

2.1.2 Semantics

An LTL formula, $\phi$, specifies a language over the alphabet, $2^{AP}$, of the labels (atomic propositions) used in the formula. A language in this sense is the set of (infinite) words, $\sigma$, that satisfy the formula. Furthermore; a word in this context is a sequence of letters which are members of $2^{AP}$.
2 Background

An infinite word, $\sigma$, over a particular alphabet, $2^{AP}$, satisfies ($\models$) an LTL formula, $\phi$, if it satisfies the following recursive criteria [1].

$$\sigma \models \text{true}$$

$$\sigma \models a \text{ iff } a \text{ is in the first letter of } \sigma$$

$$\sigma \models \phi_1 \land \phi_2 \text{ iff } \sigma \models \phi_1 \text{ and } \sigma \models \phi_2$$

$$\sigma \models \neg \phi \text{ iff } \sigma \not\models \phi$$

$$\sigma \models \Diamond \phi \text{ iff the word starting after the current letter in } \sigma, \sigma[1 \ldots] \text{ satisfies } \phi$$

$$\sigma \models \phi_1 U \phi_2 \text{ iff there exists a letter number } j \text{ in } \sigma \text{ such that letters } 1 \text{ to } j - 1 \text{ satisfy } \phi_1 \text{ and the letters after and including } j \text{ satisfy } \phi_2$$

$$\sigma \models \Diamond \phi \text{ iff there exists some letter in } \sigma \text{ that satisfies } \phi$$

$$\sigma \models \Box \phi \text{ iff every letter in } \sigma \text{ satisfies } \phi$$

2.1.3 Examples

One of the most important factors that make LTL an interesting language in which to write specification is its relatively human readable syntax. Following are a few examples of missions stated in plain English and how they translate into LTL statements.

- Eventually reach the goal state
- Eventually reach the goal state and stay there
- Return to the goal state infinitely often
- "turn around” and then "bright eyes”
- Never enter the danger zone

<table>
<thead>
<tr>
<th>Event</th>
<th>LTL Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eventually reach the goal state</td>
<td>$\Diamond \text{goal}$</td>
</tr>
<tr>
<td>Eventually reach the goal state and stay there</td>
<td>$\Diamond \Box \text{goal}$</td>
</tr>
<tr>
<td>Return to the goal state infinitely often</td>
<td>$\Box \Diamond \text{goal}$</td>
</tr>
<tr>
<td>&quot;turn around” and then &quot;bright eyes”</td>
<td>&quot;turn around” $\rightarrow \Diamond &quot;bright eyes”$</td>
</tr>
<tr>
<td>Never enter the danger zone</td>
<td>$\Box ! \text{danger zone}$</td>
</tr>
</tbody>
</table>

The ease of use is especially apparent in that when LTL formula are read, they usually produce sentences that are quite understandable to the layman. For example: $\Diamond \text{goal}$ simply reads "eventually goal” and "turn around” $\rightarrow \Diamond "bright eyes”$. While it is possible for LTL statements to grow complicated it is usually through conjunction of simpler parts that are still understandable.

2.1.4 Limitations

While LTL can express ordering of events, the concept of the next operator ($\Diamond$) does not make sense for continuous time since for every two different points in continuous time there always exists a point between them and therefore there is no well defined "next” point in time. Therefore the next operator is not included.
2.2 Graphs

in LTL for continuous systems. For discrete time systems however every point in
time does indeed have one and only one predecessor so the next operator makes
sense.\[1\]

2.1.5 Model Checking

When Linear Temporal Logic was first proposed by Amir Pnueli, his motivation
was to use LTL to reason about the execution of computer programs. Since then
LTL has been applied to formal model checking and it is from that field that we
get the necessary theory to apply it to finite automata. In model checking one
has a system model and would like to verify that its behavior complies to some
specification. Some such specifications can be expressed in LTL. Given a finite
transition system model (FTS) and an LTL specification, formal verification the-
ory gives us the tools to check if the FTS complies with the specification. There
exist efficient implementations of model checking algorithms that can handle
systems with hundreds of thousands of states \[1\] and check them against speci-
fications written in LTL.

2.2 Graphs

Most of the theory presented in this thesis is based on the concept of graphs.
A graph is composed of a set of nodes and edges between them. A graph is
undirected if for every edge from node $a$ to node $b$ there exists an opposite
edge going from $b$ to $a$. Graphs that are not undirected are directed. In this
thesis we only work with directed graphs.

When using graphs to model physical systems the nodes stand for discrete points
or regions in the state-space and the edges are transitions between them.

For convenience we define the pre set of a graph node $a$ as the set of nodes that
have an outgoing edge that ends in $a$. Similarly we define the post set of $a$ to be
the set of nodes to which there is an edge that originates in $a$.

2.2.1 Finite Automata

In this thesis I work with a rather specific kind of graph, namely Finite Au-
atomata. An automaton is a graph whose nodes model states in a state machine
and the edges model transitions between those states. The transitions between
states can be guarded by guard functions that usually depend on some external
input.
2 Background

Figure 2.1: Graph types (from left to right): undirected, undirected with edge weights, directed and directed with edge weights.

If there is a possibility for two or more edges from a state to be valid (guard functions evaluate to true) at the same time the automaton is non-deterministic but otherwise it is deterministic.

A run in an automaton is the sequence of states corresponding to some sequence of inputs.

2.2.2 Search Algorithms

To find an accepting path in the product automaton one utilizes graph search algorithms. There are two basic types of graph search algorithms: ones that are based on depth first search and ones that are based on breadth first search. **Definition 2.1.** Breadth First search consider every neighbor in the post set of a node before proceeding to expand more distant nodes.
Algorithm 1: breadth-first-search
\textbf{Data}: Graph $G$, a root node $r$ and a set of target nodes $T$
\textbf{Result}: The path from the root to the target set $P$
\begin{algorithmic}
\STATE $Q \leftarrow$ empty list;
\STATE append $x$ to $Q$;
\WHILE{$Q$ is not empty}
\STATE $q \leftarrow$ pop-head($Q$);
\FOR{$n \in \text{post}(q)$}
\IF{$n \not\in Q$}
\STATE $n.prev \leftarrow x$;
\STATE append $n$ to $Q$;
\ENDIF
\ENDFOR
\IF{$x \in T$}
\STATE \textbf{return} path-to($x$)
\ENDIF
\ENDWHILE
\end{algorithmic}

Algorithm 2: path-to
\textbf{Data}: Node $n$
\textbf{Result}: Path $P$
\begin{algorithmic}
\STATE $x \leftarrow n$;
\STATE $P \leftarrow [n]$;
\WHILE{$x.prev$}
\STATE $x \leftarrow x.prev$;
\STATE prepend $x$ to $P$;
\ENDWHILE
\STATE \textbf{return} $P$
\end{algorithmic}

Definition 2.2. Depth First moves on to expand nodes in the post set of previous node, $n$, without first expanding all of $n$'s siblings.

Algorithm 3: depth-first-search
\textbf{Data}: Graph $G$, a root node $r$ and a set of target nodes $T$
\textbf{Result}: The path from the root to the target set $P$
\begin{algorithmic}
\STATE $n \leftarrow$ pick one from post($c$);
\end{algorithmic}
Given a state space and an alphabet of atomic propositions on that space we can construct LTL formulas that specify trajectories in that state space that we want an agent (system) to follow.

**Definition 3.1.** An atomic proposition is a boolean variable.

The state space on which the atomic propositions are defined can be whatever is relevant for the mission specification. Examples of state spaces that might be of interest include:

- Internal states of a chemical processing plant.
- The discrete states of a hybrid automaton.
- A robot’s location in 2D or 3D space.

To generate a trajectory through the state space that satisfies an LTL formula we need to

1. Construct a Buchi corresponding to the LTL formula.
2. Partition the continuous state space into a finite transition system (TS).
3. Create a product transition system of the Buchi and the TS.
4. Search for an accepting run in the product.

Each of these steps is covered in the subsequent sections.

### 3.1 Buchi

Since there is not a way to use LTL formulas directly to constrain paths in a finite transition system we first need to translate them into something that we can use. That something is a Buchi automaton. What makes Buchi automata interesting for us is the fact that there are ways to construct a buchi
automaton, $A_φ$, for a given LTL formula $φ$ such that the set of words in the language specified by $φ$ is the same one as the one specified by the language of infinite words produced by $A_φ$.

$$\text{Words}(φ) = L_ω(A_φ)$$

**Definition 3.2.** A Buchi automaton is usually written as a tuple $A_φ = (Q, 2^{AP}, δ, Q_0, F)$. Its components are:

- $Q$ The set of states (vertices)
- $2^{AP}$ The set of atomic propositions in use
- $δ$ A guard function $δ : Q \times 2^{AP} \rightarrow 2^Q$ that maps a current state and a set of atomic proposition values to a set of available next states
- $Q_0$ A set of initial states
- $F$ A set of accepting states

If a Buchi can be constructed for a given LTL formula, there exists an accepting run through the Buchi that satisfies the LTL formula. The accepting run is composed of a prefix from an initial state, $q_i$, to one of the accepting states, $q_a$, and a suffix loop that begins and ends at $s_a$ and repeated ad infinitum.

![Figure 3.1: An accepting run is composed of a prefix and a suffix.](image)

### 3.1.1 Buchi construction

There are algorithms for constructing a Buchi automaton for a given LTL formula $φ$ in time and space that is exponential in the length $φ$ [11]. The different algorithms for constructing a Buchi are out of scope for this thesis but I utilize a program written by Denis Oddoux and Paul Gastin called *LTL2BA*. This program transforms an LTL formula into a Buchi automaton represented in a syntax understood by the *Spin* model checker that I then parse (see chapter 5).

### 3.2 Finite Transition System (TS)

In order to construct a plan the state space on which the atomic propositions are defined needs to be discretized. How this partitioning is done should be informed by the controllability of the system because we also need to compute reachability relations between partitions. Another thing to keep in mind during
partitioning is to avoid creating an unnecessary number of partitions since that will result in a larger TS and therefore a larger search space for planning.

In this thesis, the main focus is on environment based discretization (as opposed to the internal states of an agent) where the state space is discretized into (usually polygonal) partitions and represented as a finite transition system. The discretization is informed by what kind of low-level controllers are available and the set of atomic propositions of interest.

**Definition 3.3.** In our framework, the state-space is discretized into a finite **transition system** where the vertices are regions in the state-space and the edges are transition relations between region. Our transition systems have the following attributes

- $\Pi$ The set of states (vertices)
- $\rightarrow_c$ The set of state transitions (edges)
- $\Pi_0$ The set of initial positions
- $AP$ The set of atomic propositions
- $L_c$ A labeling function $L_c : \Pi \rightarrow 2^{AP}$ that associates each state with values for the atomic propositions
- $W$ A function $\rightarrow_c \rightarrow \mathbb{R}$ that returns the weight of an edge.

We will refer to the transition system as $T_c = (\Pi, \rightarrow_c, \Pi_0, AP, L_c, W)$.

In the typical robotic navigation scenario we have a two-dimensional environment that is partitioned into polygons with edges between polygons that share a facet through which the low level controller can drive the robot. Typical atomic propositions could be “obstacle in region” or “region of interest”.

Examples of Finite Transition systems that one would use to plan a robot’s motion could include

1. A two dimensional map of the environment that is partitioned into areas that the robot can or cannot traverse such as the maze in figure 3.2a.

2. A road network where the states are intersections and the transitions are the roads between them such as the one around T-Centralen in Stockholm 3.2b.

3. Any hybrid automaton for which one has a controller such as the one in 3.2c.
(a) A maze where red squares are obstacles around T-Centralen

(b) The road network

(c) A hybrid automaton.

Figure 3.2: Examples of the different kinds of transition systems that could be used for planning.
3.3 Product Transition System

Now that we have a Buchi automaton that encodes the LTL specification that we want to implement and a finite transition system (TS) abstraction of the continuous state space we somehow need to apply the Buchi to the TS. The traditional way to do this in the literature on LTL-based planning is to compute a synchronized product of the Buchi and the TS. In the next chapter on online planning (chapter 4) we will mention an alternative method and discuss situations where that method might be more efficient.

Before proceeding further, we need to define the product graph and present some notation.

**Definition 3.4.** A Synchronized Product \( Ap \) is a product of a transition system, \( T_c \) and a Buchi automaton, \( A_\phi \). Its attributes are

- \( Q' \) The set of states \( q' = (\pi, q) \) where \( \pi \in \Pi \) and \( q \in Q \)
- \( 2AP \) The set of atomic propositions in use
- \( \delta' \) A state transition map \( \delta' : Q' \to Q' \) that maps \( (\pi_i, q_m) \) to \( (\pi_j, q_n) \) iff \( \pi_j \in \text{Post}(\pi_i) \) and \( q_n \in \delta(q_m, L_c(\pi_i)) \)
- \( Q'_0 \) A set of initial states \( Q'_0 = \{ (\pi, q) | \pi \in \Pi_0, q \in Q_0 \} \)
- \( F'_0 \) A set of accepting states \( F'_0 = \{ (\pi, q) | \pi \in \Pi, q \in F \} \)

Now we can present an algorithm that takes a Buchi automaton and a TS and constructs a synchronized product.

**Algorithm 4: build-product**

**Data:**
- Transition system \((\Pi, \to_c, \Pi_0, AP, L_c)\)
- Buchi automaton \((Q, 2AP, \delta, Q_0, F)\)

**Result:** The product automaton \((Q', 2AP, \delta', Q'_0, F')\)

```
1 foreach \((\pi_i, q_m)\) \in \Pi \times Q \do
  2 | if \(q_m \in Q_0\) and \(\pi_i \in \Pi_0\) then
  3 | (\pi_i, q_m) \in Q'_0
  4 | end
  5 | if \(q_m \in F\) then
  6 | (\pi_i, q_m) \in F'_0
  7 | end
  8 | foreach \((\pi_j, q_n)\) \in \Pi \times Q \do
  9 | if \(\pi_j \in \text{Post}(\pi_i)\) and \(q_n \in \delta(q_m, L_c(\pi_i))\) then
 10 | (\pi_j, q_n) \in \delta'((\pi_i, q_m))
 11 | end
 12 | end
end
```

An example of a Buchi, a TS and the corresponding synchronized product is
presented in figure 3.3. In the example we have a transition system with two states labeled ”wake” and ”sleep”. We also have a Buchi generated by LTL2BA for the LTL formula that encodes the following rules.

- Never sleep and wake at the same time: $\square \neg (\text{sleep} \land \text{wake})$
- Always either sleep or wake: $\square (\text{sleep} | | \text{wake})$
- Eventually wake up after sleeping: $(\text{sleep} \rightarrow \Diamond \text{wake})$
- Eventually sleep after waking: $(\text{wake} \rightarrow \Diamond \text{sleep})$

Together, these rules give us the LTL formula

$$\square \neg (\text{sleep} \land \text{wake}) \land \square (\text{sleep} | | \text{wake}) \land (\text{sleep} \rightarrow \Diamond \text{wake}) \land (\text{wake} \rightarrow \Diamond \text{sleep})$$

The synchronized product that we now have computed encodes both the LTL formula through the Buchi and the possible system transitions through the TS. Now that we have all this information we can search for an accepting run in the product that projects to an accepting run in the Buchi.
Figure 3.3: An example of the product of a Buchi and a transition system. The sleep and wake propositions have been abbreviated as s and w.
3 Nominal Case

3.4 Planning

Given an accepting run in a product graph

\[(\pi_h, q_i) \ldots ((\pi_j, q_a) \ldots (\pi_j, q_a))^{\omega}\]

we can compute a path to follow in the transition system that satisfies the LTL specification by simply projecting the product states into their corresponding TS states

\[\pi_h \ldots (\pi_j \ldots \pi_j)^{\omega}\]

What now remains is to compute the accepting run but that is essentially a two-step graph search problem. The steps are:

1. Find a path from an initial state to an accepting state.
2. Find a path from the accepting state back to itself.

These steps can be performed by any number of graph search algorithms.

3.5 Optimal Planning

There exist efficient algorithms from the model checking community that can find accepting runs in a product transition system and we can indeed use those. The downside to using these algorithms is that they search for any accepting run no matter how long or convoluted while we would often like to find a run that is optimal or at least good in the sense that it projects to a path in the TS that it has a low cost. An example of plans generated by a regular DFS-based planner and an optimal one can be found in figure 3.4.

The edges in the transition system are assumed to have costs associated with them. This cost is modeled with the weight attribute. Since edges in the Buchi do not represent any real world changes, we let product edges inherit the weight of their corresponding transition system edges.

To find an optimal solution to a planning problem we must compute the shortest path from every initial state in the product graph to every accepting state (prefix) and then again from every accepting state to itself (suffix). Once we have those we must decide what it means for a combination to be optimal since the shortest prefix will not necessarily end in an accepting state that has the shortest suffix. To this end we define a weighting parameter, \(\beta\), for the length of the suffix.
Then the selection of the optimal plan is the combination of prefix and suffix that minimizes.

$$|P_p| + \beta |P_s|$$  \hspace{1cm} (3.1)

For applications that have some sort of steady state movement e.g. specifications that do not contain $\Diamond \square p$ where $p \in 2^A$, the suffix is typically far more important since it will be repeated ad infinitum while the prefix will only be executed once and thus it is reasonable to select $\beta \gg 1$. Even for specifications of the type $\Diamond \square p$, it is OK to pick a high $\beta$ since the lightest suffix will usually be zero.

To find the shortest prefix is to find the shortest path in the product graph between an initial state and an accepting state. To do this end we must choose from the available selection of shortest path algorithms. The classic choice for a shortest path algorithm is Dijkstra’s algorithm. Another option would be to use a more efficient algorithm such as A* but those gain their efficiency by placing constraints that the product graph does not fulfill. In particular, the product graph does not have the Euclidean distance property that A* requires.

Dijkstra’s algorithm works by computing the distance of every node from a chosen source node. It does this by visiting every node in a breadth first fashion, starting with the source node and always visiting the closest nodes first. The traditional version of the algorithm will not terminate until all nodes in the graph have been visited but since we typically only care about the distance to a small subset of the graph we make a slight modification to the algorithm to terminate when every node in a specified target set has been visited (and thus has a known distance). There is no point in continuing once we have the shortest paths to the nodes that we are interested in. There are many cases where this optimization will save quite a bit of effort. The modified version of
Dijkstra’s algorithm is listed as algorithm 5.

Algorithm 5: targeted-dijkstra

**Data:**
- Product automaton \((Q', 2^{AP}, \delta', Q'_0, F')\)
- A source state \(q_0\)
- A set of targets \(\text{targets}\)

**Result:** A map of distance from a particular node to the source, \(\text{dist} : Q \rightarrow \mathbb{R}\), and a map from a node, \(p\) to the neighbor through which the shortest path lies, \(\text{prev} : Q \rightarrow Q\).

1. \(\text{tovisit} \leftarrow \emptyset\)
2. \(\text{visited} \leftarrow \emptyset\)
3. **foreach** \(q \in Q'\) **do**
   4. \(\text{dist}(q) \leftarrow \infty\)
   5. \(\text{prev}(q) \leftarrow \emptyset\)
4. **end**
6. \(\text{dist}(q_0) \leftarrow 0\)
7. \(\text{tovisit} \leftarrow \text{tovisit} \cup \{q_0\}\)
8. **while** \(\text{tovisit} \neq \emptyset \land \text{targets} \neq \emptyset\) **do**
   9. \(c \leftarrow n \in \text{tovisit} \text{ that minimizes } \text{dist}(n)\)
10. \(\text{tovisit} \leftarrow \text{tovisit} \setminus \{c\}\)
11. \(\text{visited} \leftarrow \text{visited} \cup \{c\}\)
12. **foreach** \(p \in \text{Post}(c)\) **do**
   13. **if** \(\text{dist}(c) + \text{weight}(c,p) < \text{dist}(p)\) **then**
   14. \(\text{dist}(p) \leftarrow \text{dist}(c) + \text{weight}(c,p)\)
   15. \(\text{prev}(p) \leftarrow c\)
   16. **if** \(p \notin \text{visited}\) **then**
   17. \(\text{tovisit} \leftarrow \text{tovisit} \cup \{p\}\)
   18. **end**
19. **end**
20. **end**
21. **if** \(c \in \text{targets}\) **then**
22. \(\text{targets} \leftarrow \text{targets} \setminus \{c\}\)
23. **end**
24. **end**
25. **end**

Finding the shortest suffix calls for a little extra complication. Since Dijkstra’s algorithm defines the distance from the source node to itself to be zero, we can not simply have the target set be the accepting state. Instead we have the target set be the \(\text{Pre}\) set of the accepting state (excluding the accepting state itself). Once we have the \(\text{dist}\) and \(\text{prev}\) functions we can compute the paths from the accepting state to each state in its \(\text{Pre}\) set as well as their lengths. Then we simply append the last remaining edge to the accepting state and its weight and pick the shortest path.
The implementation of the algorithms presented in this chapter is discussed in chapter 5.

**Algorithm 6: shortest-suffix**

**Data:**
- Product automaton \((Q', 2^{AP}, \delta', Q'_0, F')\)
- An accepting state \(q_a\)

**Result:** The shortest path, \(P_a\) from \(q_a\) to \(q_a\)

```plaintext
paths ← ∅

if edge from \(q_a\) to \(q_a\) exists then
  if weight(\(q_a, q_a\)) = 0 then
    return [\(q_a, q_a\)]
  else
    paths ← paths ∪ \{[\(q_a, q_a\)]\}
  end
end

targets ← \{p|p ∈ Pre(\(q_a\)) ∧ p ≠ q_a\}
dist, prev ← targeted-dijkstra(product, ts, q_0, targets)

foreach p ∈ targets do
  path ← compute-path(prev, p)
  append [\(p, q_a\)] to path
  paths ← paths ∪ path
end

return path from paths that minimizes path-length(ts, path)
```
Figure 3.4: An example of the kinds of plans generated by a regular DFS based planner and Dijkstra based one.
Online Planning

When the planning environment is not static it may become necessary to reevaluate an existing plan given new information. Given infinite time and memory a dynamic environment is just a series of static ones where an agent simply re-computes it’s plan from scratch every time there is a change. A more interesting scenario is when the agent does not have infinite time but must be able to come up with a new plan in a reasonably expedient fashion. We can identify some qualities that we would like an online planning system to have:

1. Be able to come up with a plan quickly
2. Utilize whatever information is still valid from previous planning episodes
3. Given more information, refine the plan to become more optimal

4.1 Dynamic environments

Another reason why an agent might not know it’s environment ahead of time is that the environment might change with time. Furthermore, changes to the environment can be divided into two classes:

1. Changes caused by the agent
2. Changes caused by something else - i.e. the environment itself.

Changes caused by the environment are either hard or impossible for the agent to work around so the only way to handle them is to simply change the environment’s transition system and have the agent replan as necessary.

4.1.1 Changes caused by the agent

It is quite natural to assume that an agent might change the environment somehow during the course of its mission. An example could be a robot tasked with
picking up items and bringing them to a drop-off location. An LTL specification for this mission is

\( \Box \neg item \land \Box (item \rightarrow \Diamond base) \).

Say that once an item has been picked up it is no longer at its original location so that the atomic proposition \( item \) is no longer true at that point in the environment. Now the question is: how do we represent this change? The options are to

1. Change the environment so that the atomic proposition changes to reflect the change in the real world.

2. Add a one-way transition from the state containing the now-removed item to another copy of the transition system where the proposition is false.

3. Change the meaning of the atomic propositions so that instead of the atomic proposition \( item \) meaning that an item is present at some state we change it’s meaning to: a specific item is present at a state. For this to work we change the specification to

\( \Diamond \neg item \land (item \rightarrow \Diamond base) \)

This way the agent only needs to pick the item up once and we do not need to reflect that change in the transition system.

The first method is simple but it has the downside that the agent will have to replan which can potentially take a long time.

The second method is powerful in the sense that the initial plan can take the change to the environment into account in the initial plan but it has the downside that the size of the transition system is potentially doubled for every change so that the size of the transition system is in fact exponential in the number of dynamic propositions. Handling changes by expanding the environment transition system is obviously a powerful approach but it’s feasibility is limited to small systems and small number of dynamic propositions.

The third approach is quite simple but it does not work in cases where the propositions do not represent discrete and a priori known entities. Also, in the example of picking up things and dropping them off, the specification gets longer with the number of things that need to be picked up:

\( \Diamond \neg item1 \land (item1 \rightarrow \Diamond base) \land \Diamond \neg item2 \land (item2 \rightarrow \Diamond base) \)

Unfortunately, the Buchi size and the upper bound on the search time is exponential in the length of the LTL specification so this approach is likely to break down as the number of discretized dynamic propositions grows.

The most general method and therefore the one that we choose to go with in the implementation is the first one, to change the transition system online and have
the agent replan as necessary. This motivates us to come up with ways in which to efficiently detect when re-planning is necessary and to replan as efficiently as possible by using whatever information we can that is still valid from previous planning episodes. How efficient we go about these things depends on the desired trade-off between speed and optimality.

### 4.2 Superposition of Buchi states

Since the Buchi is a non-deterministic FSA, one must consider the possibility that for every transition in the TS, there might be several equally valid transitions in the Buchi. This must be taken into account if the environment is not perfectly known a priori and revisions to the plan must be made. When re-planning becomes necessary, the planner should consider all the combinations of the valid Buchi states and the current transition system state.

**Algorithm 7:** The book-keeping necessary to keep track of the current Buchi states

**Data:**
- The current set of valid Buchi states $Q$
- The label of the next TS state $L(p)$

**Result:** $Q'$, The set of Buchi states after the current transition

```plaintext
1 $Q' \leftarrow \emptyset$
2 foreach $q \in Q$ do
3    $Q' \leftarrow Q' \cup \{q' | q' \in \delta(q, L(p))\}$
4 end
```

If the superposition of Buchi states is not taken into account, the planner might not be able to find a solution even if one exists or it might find a suboptimal one.

### 4.3 Product Graph Revision

Computing the product graph for LTL planning is exponential in the length of the LTL specification which is unfortunate if it needs to be recomputed often. In the case of a partially unknown environment, a cheaper way to keep the product up to date is to only revise the parts of it that have changed.

**Definition 4.1.** Changes to the transition system come in three varieties which we define as the following sets.
4 Online Planning

\[ A \] The set of edges added to the transition system
\[ R \] The set of edges removed from the transition system
\[ C = (\pi, L'_c(\pi)) \] The set of transition system nodes with updated labels.
\[ C_W = (\pi_a, \pi_b, W) \] The set of edges that are not in \( A \) but have updated weights.

Given a set of environment changes a few observations can be made.

1. If there are any removed edges on the planned path, one can be quite sure that the plan is no longer valid.
2. If there are any changed labels on the planned path, one can no longer be sure that the plan is valid.
3. If there are no label changes or removed edges but there are some added edges or updated weights, one can no longer be sure that the current plan is optimal.

Perhaps the most valuable consequence of these observations is that in the case of no label changes or removed edges on the planned path one could save time and postpone revising the product graph until future changes make it necessary. If there are any changes on the planned path, revision becomes necessary and for that one can use the product revision algorithm from [5] (listed here as algorithm 8).

Once the product has been revised there are two possibilities:

1. Search for a new accepting path in the revised product.
2. Search for a bridge from the current superposition of product nodes to the undamaged part of the previous plan (assuming that it exists).

Which method is better depends on the relative value one places on optimality vs. time and how much of the previous plan was invalidated by the revision.

To find a bridge from the current superposition of product nodes one searches for a path from them to any of the product nodes in the tail end of the previously planned path that is still intact (algorithm 9). The choice of the search algorithm depends on the nature of the transition system and the expected nature of revisions. If no bridge is found there might still be accepting paths in the product, just not ones that include the tail end of the invalidated plan.

4.4 Alternative: Product-less planning

An alternative to revising the product as new information arrives is to simply avoid keeping the product around in the first place. When searching through the product graph one only needs to have the \textit{pre} and \textit{post} sets of it’s nodes. Armed
Algorithm 8: revise-product

Data:
Old product automaton \((Q', 2^{AP}, \delta', Q'_0, F')\)
Revised transition system \((\Pi, \rightarrow_c, \Pi_0, AP, L_c)\)
Buchi automaton \((Q, 2^{AP}, \delta, Q_0, F)\)
Transition system changes \(A, R,\) and \(C\)

Result: The revised product automaton \((Q', 2^{AP}, \delta', Q'_0, F')\)

1. foreach \((\pi_f, q_f, \pi_t, q_t)\) \(\in R\) do
2. \(\delta'(\pi_f, q_f) \leftarrow \delta'(\pi_f, q_f) \setminus (\pi_t, q_t)\)
3. end
4. foreach \((\pi_f, \pi_t)\) \(\in A\) do
5. foreach \((q_f, q_t)\) \(\in \{(q_f, q_t) | q_f \in Q \land q_t \in \delta(q_f, L_c(\pi_t))\}\) do
6. \(\delta'(\pi_f, q_f) \leftarrow \delta'(\pi_f, q_f) \cup (\pi_t, q_t)\)
7. end
8. end
9. foreach \((\pi_t, L'_c(\pi_t))\) \(\in C\) do
10. foreach \(q_t \in \{q(\pi_t, q) \in Q\}\) do
11. if \{\(\pi_f | \pi_f \in Pre(\pi_t) \land q_t \in \delta(q_f, L'_c(\pi_t))\}\} = \emptyset then
12. \(\delta'(\pi_f, q_f) \leftarrow \delta'(\pi_f, q_f) \setminus (\pi_t, q_t)\)
13. else
14. \(\delta'(\pi_f, q_f) \leftarrow \delta'(\pi_f, q_f) \cup (\pi_t, q_t)\)
15. end
16. end
17. end

with this insight, one can simply compute the pre and post sets of product nodes as they are needed. The following algorithm \([10]\) shows how to compute the post set without a pre-built product graph.

Algorithm 10: product-post

Data:
Transition system \((\Pi, \rightarrow_c, \Pi_0, AP, L_c)\)
Buchi automaton \((Q, 2^{AP}, \delta, Q_0, F)\)
Transition system state \(\pi\)
Buchi state \(q\)

Result: The post states of the product state \((\pi, q)\)

1. return \(\{(\pi_n, q_n) | \pi_n \in Post(\pi) \land q_n \in \delta(q, L_c(\pi_n))\}\)
Algorithm 9: bridge-path

Data:
- Product automaton $(Q', 2^{AP}, \delta', Q'_0, F')$
- Transition system $(\Pi, \rightarrow, \Pi_0, AP, L_c)$
- A superposition of product nodes $Q_s \in Q'$
- A valid remainder of a path $P_v$
- A graph search algorithm $\text{search} : Q' \times P_v \rightarrow P$

Result: A valid product path $P$

1. foreach $(\pi, q) \in Q_s$ do
2.   bridge $\leftarrow \text{search}((\pi, q), P_v)$
3.   if bridge exists then
4.     $P \leftarrow$ prepend bridge to $P_v$
5.     return $P$
6.   end
7. end

Figure 4.1 plots the planning time of two versions of the same planning algorithm differing only in that one uses algorithm 10 to compute post and the other generates the product ahead of time. The transition system and formula used in the comparision are the ones described in chapter 5.4.4. It is important to note that there are cases when generating the product ahead-of-time might be preferable such as when the planner might need to traverse most of the product so the just-in-time method might not end up saving any effort.

4.5 Dealing with insufficient information

In a scenario wherein an agent enters an unknown environment it might not have enough information to come up with a valid plan. In that case the logical next step would be to explore the environment. The problem here is that “explore the environment” is not a specification that is easily expressed in LTL. The solution here is to have a higher level planner that can generate trajectories suitable for exploration. Even though we cannot use the usual LTL motion planning algorithms to compute trajectories for exploration, we can still use LTL to express safety criteria that we can then check generated trajectories against. This way an agent could explore an environment until its main mission becomes feasible.

Not all LTL specifications are valid safety specifications in this context. A valid
4.5 Dealing with insufficient information

Figure 4.1: The average planning times for two versions of the optimal Dijkstra planner. One uses ahead-of-time product generation and the other the iterative post algorithm in [10]. The benchmark was done by randomly generating a maze problem as described in chapter 5.4.4. Ten mazes were generated for each size on the x-axis and the average planning time computed for each size. Note that the problem in this example favors the iterative algorithm since the planner is unlikely to explore the whole product.

A safety specification is one that produces a Buchi where all the states are accepting states. Examples of valid safety specifications are

An example scenario for this kind of behavior would be a robot tasked with retrieving items from the environment and bringing them to a drop-off location while avoiding obstacles. In this case the mission specification would be

\[ \Box \Diamond item \land \Box (item - > \Diamond base) \land \Box \neg obstacle \]

For the situation wherein the robot cannot fulfill this specification because it cannot reach or doesn’t know of an item to pick up or the base we want it to explore its environment while still fulfilling the safety part of its specification in the hope that it will obtain information that enables it to compute a valid
Figure 4.2: Exploration

(a) $\Box \neg fire$  
(b) $drink \rightarrow \neg Xdrive$

(c) $\Box \neg fire \land (drink \rightarrow \neg Xdrive)$

Figure 4.3: Examples of valid safety specifications

plan. To this end we just use the safety part of the specification, $\Box \neg obstacle$, and generate a random walk through the environment that satisfies $\Box \neg obstacle$. The random walk need not be all that random but rather try to reach parts of the environment that have not been previously explored.
4.6 Dynamic Search

In section 4.3, we discussed how one could react to changes in the transition system. I mentioned that there is a trade-off between the optimality of the plan and the computational burden of re-planning. Using the previously discussed algorithms, it can be very expensive to maintain optimality in dynamic environments since that would require re-planning every time when there is a change. A way to mitigate that cost is to use dynamic graph search algorithms that can keep track of search state between executions and use it to recompute the shortest paths more cheaply than restarting from scratch.

The dynamic graph search algorithm that we will be using is based on Dijkstra’s algorithm and is called Dynamic Dijkstra. The algorithm is from a paper \([3]\) by Edward P. F. Chan and Yaya Yang and the notation used here is based on theirs.

4.6.1 Shortest Path Trees

The core concept in Dynamic Dijkstra is the shortest path tree (SPT). A SPT is a tree structure rooted in the search source node where the leaves are the target nodes and the shortest path from the source, \(s\), node to any of the target nodes, \(t\), is the tree branch from \(s\) to \(t\). To encode the SPT in a directed graph we define two additional node properties:

- The shortest path parent of \(v\): \(spp(v)\). The shortest path parent is the next node on the shortest path between \(v\) and the source node.
- The shortest path children of \(v\): \(spc(v)\). A node can have any number of shortest path children.

Using these two properties we can easily trace the shortest path from any tree node to the source by simply following the parent nodes. In addition to these two properties we use the notation

- \(T_s\) for a SPT rooted in \(s\)
- \(d_s(v)\) for the distance from \(s\) to the tree node \(v\)
- \(\text{tail}(e)\) for the source node of the edge \(e\)
- \(\text{head}(e)\) for the destination node of the edge \(e\)
- \(w(e)\) for the weight of the edge \(e\)

If \(T_s\) is a SPT after some graph change, then \(T'_s\) is updated SPT.

A variable \(x\) that has a hat like \(\hat{x}\) belongs to a SPT that has been affected by some change.
4 Online Planning

4.6.2 Dynamic Dijkstra

Dynamic Dijkstra is divided into two algorithms:

1. `dynamic-dijkstra-inc` deals with the case where a tree edge has gained weight.

2. `dynamic-dijkstra-dec` deals with the case where any edge has lost weight.

The keen will note the following

- There is no explicit handling of weight increases in edges that are not in the SPT. Since we already know that those are not on the shortest path, we can rest assured that their weight gain will not affect the shortest path.

- There is no explicit handling of removed edges. Like in the case of weight increases of non-SPT edges, we do not care about removed edges that were not part of the SPT but we do need to consider those that were part of the SPT. We can handle removed SPT edges as if they had gained infinite weight.

- Edge additions are not explicitly considered but we can handle them as if they had previously existed with infinite weight and apply `dynamic-dijkstra-dec`.

- We do not need to handle weight decreases and edge additions in parts of the graph that are not reachable from the source so it might be a reasonable optimization to check maintain some reachability information if the transition is sparse and/or disconnected.

To handle all of the cases discussed in section 4.3 we must follow the following procedure

1. Run `dynamic-dijkstra-inc` with the tree-edge weight increases and removals.

2. Apply the non-tree-edge weight increases and removals to the graph.

3. Run `dynamic-dijkstra-dec` with edge weight decreases and additions.

`dynamic-dijkstra-inc` and `dynamic-dijkstra-dec` are listed as algorithms 11 and 12 respectively.
Algorithm 11: dynamic-dijkstra-inc from [3]

Data:
Directed graph $G$ 
Affected Shortest Path Tree rooted in $s \hat{T}_s$
Set of edges that have gained weight $e^+$
The weight that edge $e_i \in e^-$ has gained $\tau_i$

Result: The SPT $\hat{T}_s$ updated to account for $e^+$

1. $\epsilon \leftarrow \emptyset$
2. $\epsilon^+ \leftarrow \emptyset$
3. $\epsilon_i \leftarrow w(e_i) + \tau_i$
4. $t \leftarrow tail(e_i)$
5. $h \leftarrow head(e_i)$
6. if $t = spp(h)$ then
   7. $\hat{spc}(t) \leftarrow \hat{spc}(t) - \{h\}$
   8. $spp(h) \leftarrow \emptyset$
   9. $\epsilon \leftarrow \epsilon \cup \{e_i\}$
end
10. $\bar{N} \leftarrow$ find-locally-affected-nodes($\hat{T}_s, \epsilon$)
11. $Q \leftarrow$ new priority queue
12. $\bar{N} \leftarrow$ find-locally-affected-nodes($\hat{T}_s, \epsilon$)
13. $Q \leftarrow$ new priority queue
14. $\epsilon \leftarrow \emptyset$
15. $\epsilon^+ \leftarrow \emptyset$
16. $\epsilon_i \leftarrow w(e_i) + \tau_i$
17. $t \leftarrow tail(e_i)$
18. $h \leftarrow head(e_i)$
19. if $t = spp(h)$ then
   20. $\hat{spc}(t) \leftarrow \hat{spc}(t) - \{h\}$
   21. $spp(h) \leftarrow \emptyset$
   22. $\epsilon \leftarrow \epsilon \cup \{e_i\}$
end
23. $\epsilon \leftarrow \emptyset$
24. $\epsilon^+ \leftarrow \emptyset$
25. $\epsilon_i \leftarrow w(e_i) + \tau_i$
26. $t \leftarrow tail(e_i)$
27. $h \leftarrow head(e_i)$
28. $\hat{spc}(t) \leftarrow \hat{spc}(t) - \{h\}$
29. $spp(h) \leftarrow \emptyset$
30. $\epsilon \leftarrow \epsilon \cup \{e_i\}$
end
31. $\bar{N} \leftarrow$ find-locally-affected-nodes($\hat{T}_s, \epsilon$)
32. $Q \leftarrow$ new priority queue
33. $\epsilon \leftarrow \emptyset$
34. $\epsilon^+ \leftarrow \emptyset$
35. $\epsilon_i \leftarrow w(e_i) + \tau_i$
36. $t \leftarrow tail(e_i)$
37. $h \leftarrow head(e_i)$
38. $\hat{spc}(t) \leftarrow \hat{spc}(t) - \{h\}$
39. $spp(h) \leftarrow \emptyset$
40. $\epsilon \leftarrow \epsilon \cup \{e_i\}$
end
41. $\epsilon \leftarrow \emptyset$
42. $\epsilon^+ \leftarrow \emptyset$
43. $\epsilon_i \leftarrow w(e_i) + \tau_i$
44. $t \leftarrow tail(e_i)$
45. $h \leftarrow head(e_i)$
46. $\hat{spc}(t) \leftarrow \hat{spc}(t) - \{h\}$
47. $spp(h) \leftarrow \emptyset$
48. $\epsilon \leftarrow \epsilon \cup \{e_i\}$
end
49. $\epsilon \leftarrow \emptyset$
50. $\epsilon^+ \leftarrow \emptyset$
51. $\epsilon_i \leftarrow w(e_i) + \tau_i$
52. $t \leftarrow tail(e_i)$
53. $h \leftarrow head(e_i)$
54. $\hat{spc}(t) \leftarrow \hat{spc}(t) - \{h\}$
55. $spp(h) \leftarrow \emptyset$
56. $\epsilon \leftarrow \epsilon \cup \{e_i\}$
end
57. $\epsilon \leftarrow \emptyset$
58. $\epsilon^+ \leftarrow \emptyset$
59. $\epsilon_i \leftarrow w(e_i) + \tau_i$
60. $t \leftarrow tail(e_i)$
61. $h \leftarrow head(e_i)$
62. $\hat{spc}(t) \leftarrow \hat{spc}(t) - \{h\}$
63. $spp(h) \leftarrow \emptyset$
64. $\epsilon \leftarrow \epsilon \cup \{e_i\}$
end
65. $\epsilon \leftarrow \emptyset$
66. $\epsilon^+ \leftarrow \emptyset$
67. $\epsilon_i \leftarrow w(e_i) + \tau_i$
68. $t \leftarrow tail(e_i)$
69. $h \leftarrow head(e_i)$
70. $\hat{spc}(t) \leftarrow \hat{spc}(t) - \{h\}$
71. $spp(h) \leftarrow \emptyset$
72. $\epsilon \leftarrow \epsilon \cup \{e_i\}$
end
73. $\epsilon \leftarrow \emptyset$
74. $\epsilon^+ \leftarrow \emptyset$
75. $\epsilon_i \leftarrow w(e_i) + \tau_i$
76. $t \leftarrow tail(e_i)$
77. $h \leftarrow head(e_i)$
78. $\hat{spc}(t) \leftarrow \hat{spc}(t) - \{h\}$
79. $spp(h) \leftarrow \emptyset$
80. $\epsilon \leftarrow \epsilon \cup \{e_i\}$
end
81. $\epsilon \leftarrow \emptyset$
82. $\epsilon^+ \leftarrow \emptyset$
83. $\epsilon_i \leftarrow w(e_i) + \tau_i$
84. $t \leftarrow tail(e_i)$
85. $h \leftarrow head(e_i)$
86. $\hat{spc}(t) \leftarrow \hat{spc}(t) - \{h\}$
87. $spp(h) \leftarrow \emptyset$
88. $\epsilon \leftarrow \epsilon \cup \{e_i\}$
end
89. $\epsilon \leftarrow \emptyset$
90. $\epsilon^+ \leftarrow \emptyset$
91. $\epsilon_i \leftarrow w(e_i) + \tau_i$
92. $t \leftarrow tail(e_i)$
93. $h \leftarrow head(e_i)$
94. $\hat{spc}(t) \leftarrow \hat{spc}(t) - \{h\}$
95. $spp(h) \leftarrow \emptyset$
96. $\epsilon \leftarrow \epsilon \cup \{e_i\}$
end
Algorithm 12: dynamic-dijkstra-dec from [3]

Data:
Directed graph $G$
Affected Shortest Path Tree rooted in $s$, $\tilde{T}_s$
Set of edges that have lost weight $\epsilon^-$
The weight that edge $e_i \in \epsilon^-$ has lost $\tau_i$

Result: The SPT $\tilde{T}_s$ updated to account for $\epsilon^-$

1. **foreach** $e_i \in \epsilon^-$ **do**
2. \hspace{1em} $Q \leftarrow \text{new priority queue}$
3. \hspace{1em} $w(e_i)' \leftarrow w(e_i) + \tau_i$
4. \hspace{1em} $t \leftarrow \text{tail}(e_i)$
5. \hspace{1em} $h \leftarrow \text{head}(e_i)$
6. \hspace{1em} **if** $\hat{d}_s(t) + w(e_i)' < \hat{d}_s(h)$ **then**
7. \hspace{2em} $\hat{d}_s(h) \leftarrow \hat{d}_s(t) + w(e_i)'$
8. \hspace{2em} Enqueue $e_i$ in $Q$ with key $\hat{d}_s(h)$
9. \hspace{1em} **end**
10. **while** $Q$ is not empty **do**
11. \hspace{1em} $e \leftarrow \text{pop smallest value from } Q$
12. \hspace{1em} $y \leftarrow \text{head}(e)$
13. \hspace{1em} $x \leftarrow \text{tail}(e)$
14. \hspace{1em} $d \leftarrow \hat{d}_s(y)$
15. \hspace{1em} $\hat{spc}(x) \leftarrow \hat{spc}(x) \cup \{y\}$
16. \hspace{1em} $p \leftarrow \hat{spp}(y)$
17. \hspace{1em} $\hat{spc}(p) \leftarrow \hat{spc}(p) - \{y\}$
18. \hspace{1em} $\hat{spp}(y) \leftarrow x$
19. \hspace{1em} **foreach** $e \in \text{outgoing edges of } y$ **do**
20. \hspace{2em} $q \leftarrow \text{head}(e)$
21. \hspace{2em} **if** $\hat{d}_s(y) + w(e)' < \hat{d}_s(q)$ **then**
22. \hspace{3em} $\hat{d}_s(q) \leftarrow \hat{d}_s(y) + w(e)'$
23. \hspace{3em} Enqueue $e$ in $Q$ with key $\hat{d}_s(q)$
24. \hspace{2em} **end**
25. **end**
26. **end**
27. **end**
Algorithm 13: find-locally-affected-nodes from [3]

Data:
Directed graph $G$
Affected Shortest Path Tree rooted in $s$ $T_s$
A set of edges $\epsilon$

Result: The set of nodes, $\bar{N}$ that are affected by changes in the edges in $\epsilon$

1. $\bar{N} \leftarrow \emptyset$
2. foreach $e_i \in \epsilon$ do
   3. $a \leftarrow \text{head}(e_i)$
   4. $c \leftarrow \{(a, b) | b \in \text{spc}(a)\}$
   5. $\bar{N} \leftarrow \bar{N} \cup \{a\} \cup \text{find-locally-affected-nodes}(T_s, c)$
3. end
One of the original goals of this Master Thesis was to construct a generic framework as MATLAB function scripts. Since work on the thesis started it was decided to use Python instead of MATLAB for the implementation because of more flexible and generic syntax and the rich ecosystem of open source libraries.

The framework is constructed as a set of python functions and classes that are packaged together into the `ltlplanner` submodule. The most important submodules are listed in table 5.1.

<table>
<thead>
<tr>
<th>Submodule</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>ltlplanner.ts</td>
<td>functions to create and manipulate transition systems.</td>
</tr>
<tr>
<td>ltlplanner.buchi</td>
<td>functions to create Buchi automata from LTL formulas and to manipulate them afterwards.</td>
</tr>
<tr>
<td>ltlplanner.product</td>
<td>functions to create product automata both ahead of time and on-demand as discussed in [4].</td>
</tr>
<tr>
<td>ltlplanner.algorithms</td>
<td>submodules for graph search and manipulation.</td>
</tr>
<tr>
<td>ltlplanner.simulation</td>
<td>classes and functions to simulate one or more agents operating in a two-dimensional environment.</td>
</tr>
</tbody>
</table>

Table 5.1: ltlplanner submodules.

### 5.1 Graph representation

Since we are going to be primarily working graphs, a good graph representation is desirable. It is quite simple to create custom Graph, Edge and Node classes with a nice API but there is no need to do so since there is already a selection of excellent python packages for working with graphs that have a lot of functionality out of the box. The package that is used in ltlplanner is called networkx. It
includes a wide range of well tested algorithms for graph search that are useful for testing our specialized implementations against as well as utility functions for things such as generating Graphviz visualizations.

## 5.2 Buchi generation

The `ltlplanner.buchi` module creates a Buchi automaton by calling the LTL2BA program [4] and parsing the output. The said output is a never claim in the Promela language [4]. An example output for the formula □◊a ∧ □◊b is

```plaintext
never { /* []<>a & []<>b */
T0_init:
  if
    :: (1) -> goto T0_init
    :: (a) -> goto T1_S1
    :: (a & b) -> goto accept_S1
  fi;
T1_S1:
  if
    :: (1) -> goto T1_S1
    :: (b) -> goto accept_S1
  fi;
accept_S1:
  if
    :: (1) -> goto T0_init
    :: (a) -> goto T1_S1
    :: (a & b) -> goto accept_S1
  fi;
}
```

This never claim encodes a Buchi buchi automaton which we can parse out. The `ltlplanner.promela` module contains a recursive decent parser for never claims in the Promela language and the `ltlplanner.boolean_formulas` module contains another recursive decent parser for the boolean guard expressions.

The Promela never claim format is quite straight forward. It lists buchi states and their outgoing edges (with guard expressions). Its grammar is as follows.

```
<state> ::= <state-name> ":" <newline> <tab>
"if" <newline>
<edge-list>
"fi," <newline>
<state-name> ::= [a-zA-Z][a-zA-Z0-9]*_[a-zA-Z][a-zA-Z0-9]*
<edge-list> ::= 
```
5.2 Buchi generation

<edge> ::= <tab> ":: " <boolean-expression> " -> goto " <state-name> <newline>
<boolean-expression> ::= <or-expr>
<or-expr> ::=<and-expr> "||" <or-expr>
<and-expr> ::=<not-expr> "&&" <and-expr>
<not-expr> ::= "!" <par-expr>
<par-expr> ::= "(" <or-expr> ")"

5.2.1 Guard parser

The guard parser produces boolean expression trees on a negative normal form (NNF). An expression (tree) on NNF is one that does not contain any negation except directly applied to atomic propositions. An example of an expression tree for the formula \(\neg(a \lor b) \lor a\) is given in figure 5.1.

![Expression tree examples]

(a) On NNF  
(b) Not on NNF

Figure 5.1: Expression tree for \(\neg(a \lor b) \lor a\)

To evaluate a guard against a label one simply evaluates it against the root node in the expression tree. The and and OR nodes behave as expected by recursively
evaluating the label against their child nodes. Eventually the leaves, which are always either propositions or negated propositions, are evaluated and they either return true or false depending on whether the proposition in question is present in the label or not.

The reason why we are interested in expression trees on NNF is that we want to be able to define a distance between a label and a guard and for that to work we can not apply negation to anything other than atomic propositions.

The way we define this boolean distance is as a recursive function of an expression tree.

\[
\begin{align*}
\text{distance}(\text{OR}) &= \min\left(\text{distance}(\text{left}), \text{distance}(\text{right})\right) \\
\text{distance}(\text{AND}) &= \text{distance}(\text{left}) + \text{distance}(\text{right}) \\
\text{distance}(\text{Symbol}) &= \begin{cases} 
0 & \text{if given label contains the symbol} \\
1 & \text{otherwise}
\end{cases}
\end{align*}
\]

In the implementation, the expression trees are defined in an object oriented fashion so that there are different classes for AND, OR, and Symbol expressions and each of those classes defines a distance method that is appropriate for its respective expression type. For example, the implementation for the symbol class is

```python
class SymbolExpression(Expression):
    ...
    def distance(self, label):
        if self.symbol in label:
            return 0
        else:
            return 1
```

### 5.3 Product generation

As previously discussed, there are two ways in which one can generate a product automaton. Those are

1. Generate it ahead of time.
2. Calculate \textit{Pre} and \textit{Post} sets for product nodes on-the-fly.

Which method is better depends on the specific usage pattern for the product. If the product is likely to remain static and a large part of its nodes are likely to be examined (e.g. in a ”flat” environment where most transitions cost about
the same), then it makes sense to memorize the product by keeping a copy in memory. If however the product is likely to change due to changes in the transition system or if most of its nodes are not likely to be visited by a planner due to low connectedness or costly transitions, it makes more sense to compute just the nodes that are needed (\textit{Pre} and \textit{Post} sets) as they are needed.

```python
def create_product(ts, buchi):
    prod = DiGraph()
    prod.ts = ts
    prod.buchi = buchi
    prod.type = "tight"
    prod.initial = set()
    prod.accept = set()
    for (p,q) in cartesian_product(ts.nodes_iter(), buchi.nodes_iter()):
        pnode = str((p,q))
        prod.add_node(pnode, ts_node=p, buchi_node=q)
        if q in buchi.initial and p in ts.initial:
            prod.initial.add(pnode)
        if q in buchi.accept:
            prod.accept.add(pnode)
    for (p,q) in cartesian_product(ts.nodes_iter(), buchi.nodes_iter()):
        fnode = str((p, q))
        for (pn,qn) in cartesian_product(ts.nodes_iter(), buchi.nodes_iter()):
            if not (ts.has_edge(p,pn) and buchi.has_edge(q,qn)):
                continue
            tnode = str((pn,qn))
            label = ts.node[p]["label"]
            guard = buchi.edge[q][qn]["guard"]
            if guard.check(label):
                prod.add_edge(fnode, tnode, weight=ts.edge[p][pn]["weight"])
    return prod
```

```python
def product_predecessors_iter(ts, buchi, pt, qt):
    label = ts.node[pt]["label"]
    qfs = (q for q in buchi.predecessors_iter(qt)
           if buchi.edge[q][qt]["guard"].check(label))
    return cartesian_product(ts.predecessors_iter(pt), qfs)
```

```python
def product_successors_iter(ts, buchi, pf, qf):
    for pt in ts.successors_iter(pf):
        label = ts.node[pt]["label"]
```
for qt in buchi.successors_iter(qf):
    if buchi.edge[qf][qt]["guard"].check(label):
        yield (pt, qt)

5.4 Planning

Various types of planning are implemented in the agent model in the `ltlplanner.simulation`. In some implementations the general purpose algorithms from `ltlplanner.algorithms` which is the preferred way but in some search is integrated into the agent.

There are two main types of search algorithms used

- NDFS and MDFS from [5]
- Dijkstra’s algorithm

The implementation for NDFS and MDFS is quite straight-forward but the implementation for Dijkstra is split up into one part that generates one prefix path at a time and another that

1. Iterates over every viable prefix
2. Calculates the shortest suffix for each accepting state for which there exists a prefix
3. Selects the optimal prefix-suffix combination based on the relative weight that the user gives to the prefix and suffix costs

It is up to the agent implementation to decide which algorithms to use for the prefix and suffix planning but the choice will be heavily affected by the relative value that is placed on optimality vs. speed, prefix vs. suffix length and whether one expects much change in the environment.

5.4.1 Revision

For revision there are two approaches implemented

- The `update` and `revise` functions from [5] are implemented in the `ltlplanner.revision` module.
- Keeping track of all potential Buchi position and re-planning from all of them is implemented in `IterAgent` and `OptimalIterAgent` in the `ltlplanner.simulation` module.

The latter lets us keep the plan optimal through revision but the first one is faster.
There is a third approach that is to use the Dynamic Dijkstra algorithms for the prefix part of the planning but the agent implementation for that is experimental.

### 5.4.2 Agent Model

![Agent model diagram]

To orchestrate planning, execution and revision we add an agent model that keeps track of the robot’s state from the time when it has received a mission specification to when it either turned off or receives a new specification. The state machine structure for the agent model is implemented in the `AgentNG` class and the different types of agents (Dijkstra, NDFS etc.) inherit the common from it.

A set of monitor classes can be attached to an agent to control its execution and to collect runtime information for visualization and performance analysis. The monitors work by listening to events emitted from state changes in the agent.
at which point they can inspect it’s state and/or issue commands. Examples of monitors that were implemented for the purposes of simulation and analysis include

- The StopIterationMonitor, which keeps track of the number of suffix iterations that the agent has completed and can optionally stop it when it has completed a specific number of iterations (otherwise it would run forever).

- The TransitionSystemMonitor, which keeps track of the agent’s internal representation as it evolves throughout the course of it’s execution so that one can analyse it afterwards.

- The GraphicMonitor, which leverages the TransitionSystemMonitor to render visualizations of the agent’s execution.

- The PlanningPerformanceMonitor, which listens to the pre- and post-planning events to measure the time that agent spends planning.

5.4.3 Simulations

Following are a few simulations that serve to illustrate the characteristics of the different planning algorithms.

There were two different revising agent’s implemented, one using the NDFS algorithm and the other using Dijkstra. As discussed earlier, the NDFS approach is faster but it tends to produce convoluted plans. In the simulated scenario the transition cost is uniform across the grid, there are three propositions, trap (green), goal1 (red) and goal2 (blue) and the specification is

\[ \Box \Diamond \text{goal1} \land \Box \Diamond \text{goal2} \land \Box \neg \text{trap}. \]

The agent’s position is indicated as a yellow circle, the prefix is dotted blue and the suffix is dotted yellow. The agent’s “sensors” are configured to have a range of one square. The execution of the NDFS-based agent is visualized in figure 5.3. The corresponding execution of the Dijkstra-based agent is visualized in figure 5.4.

5.4.4 Performance

The graph in figure 5.5 visualizes the relative planning times for the three different agent types (Dijkstra, greedy Dijkstra and NDFS) for a simple patrolling and obstacle avoidance specification in a maze environment similar to the one in figure 3.2a. The specification used is

\[ \Box \Diamond \text{goal1} \land \Box \Diamond \text{goal2} \land \Box \neg \text{wall}. \]
The relative performance of the planning algorithms was as expected with NDFS being the fastest and the optimal Dijkstra planner requiring the most planning time.

The problem dimension varied in the benchmarks was the size of the transition system. It would have been equally valid to vary the complexity of the LTL specification or the distribution of propositions in the transition system. This type of benchmark requires a way to generate transition systems and the method used here was to generate a maze using the `maze` function in the `ts` module.

### 5.4.5 Example usage: Basic

The following listing contains a complete usage example for the library functions.

```python
import ts
import buchi
import product
import algorithms.planning

# Configure the transition system
transition_system = ts.create_rectworld(6, 7)
transition_system.node["r3c3"]["label"]["label"]["goal1"]
transition_system.node["r1c4"]["label"]["label"]["goal2"]
traps = ["r1c1", "r2c1", "r3c2", "r4c3", "r3c4", "r2c4"]
for node in traps:
    transition_system.node[node]["label"]["label"]["trap"]
initial_position = "r0c0"
transition_system.initial.add(initial_position)

# Configure the Buchi (mission)
mission = buchi.buchi_from_ltl("<> goal1 && <> goal2 && ![trap]")

# Calculate the initial product states
product_positions = product.potential_product_positions(
    initial_position,
    mission.initial)

# Initialize the planner and plan
planner = algorithms.planning.DijkstraIterPlanner(
    transition_system,
    mission,
    beta=1)
plan_cost, prefix, suffix = planner.plan_optimal(product_positions)
```
The first step is to configure the transition system. In this case we use an automatically generated 2D map 6 rows high and 7 columns wide.

```
transition_system = ts.create_rectworld(6, 7)
```

The `ts.create_rectworld` function automatically labels each state with a row and column number and we use these names to refer to them when assigning atomic propositions. One example of such usage is when we assign the transition system state `r3c3` the atomic proposition `goal1`.

```
transition_system.node["r3c3"]['label'].add("goal1")
```

Aside from assigning atomic propositions to states we also need to define an initial transition system state. That is done by adding it to the set of initial states like so.

```
initial_position = "r0c0"
transition_system.initial.add(initial_position)
```

Now the transition system is ready and we can move on to the Buchi. Constructing a Buchi graph for a given LTL expression is accomplished via the `buchi.buchi_from_ltl` function. In this case we use the same expression as section 5.4.3.

```
mission = buchi.buchi_from_ltl("[<> goal1 && [<> goal2 && []!trap"]
```

We need to give the planning algorithm a set of product positions to start searching from. In this case we only have one product position but we still use the `product.potential_product_positions` to turn it into a set of product position just like we would have if there were multiple possible product states.

```
product_positions = product.potential_product_positions(
    initial_position,
    mission.initial)
```

In this case we are using Dijkstra’s algorithm for planning and so we use the `algorithms.planning.DijkstraIterPlanner` class. It is initialized with the transition system and the Buchi but it also takes the optional `beta` parameter (default value is 1), as defined in equation [3.1] to control the relative penalties on the prefix and suffix costs used to select the optimal plan.

```
planner = algorithms.planning.DijkstraIterPlanner(
    transition_system,
    mission,
    beta=1)
```

Finally, the `planner.plan_optimal` does the actual planning and returns the plan cost, prefix and suffix.

```
plan_cost, prefix, suffix = planner.plan_optimal(product_positions)
```
5.4 Planning

5.4.6 Example usage: Agent

The following listing contains a complete usage example for the agent-based planning model including example use of the agent monitor classes as well as plotting and movie making functionality from the `simulation.presentation` module.

```python
import ts
import buchi
import simulation.agents
import simulation.monitors
import simulation.presentation
import storage

# Configure the transition system
transition_system = ts.create_rectworld(6, 7)
transition_system.node["r3c3"]['label'].add("goal1")
transition_system.node["r1c4"]['label'].add("goal2")
traps = ['r1c1', 'r2c1', 'r3c2', 'r4c3', 'r3c4', 'r2c4']
for node in traps:
    transition_system.node[node]['label'].add("trap")
initial_position = "r0c0"
transition_system.initial.add(initial_position)

# Configure the Buchi (mission)
mission = buchi.buchi_from_ltl('[<> goal1 && <>goal2 && !trap]')

# Initialize the agent and monitor objects
agent = simulation.agents.DijkstraAgentNG(
    transition_system,
    initial_position,
    mission, mission.initial)
perfmon = simulation.monitors.PlanningPerformanceMonitor()
perfmon.attach(agent)
stopper = simulation.monitors.SuffixIterationMonitor(limit=2)
stopper.attach(agent)
recorder = simulation.monitors.TransitionSystemMonitor()
recorder.attach(agent)
planmon = simulation.monitors.PlanMonitor()
planmon.attach(agent)

# Start execution
agent.start()
```
5 Implementation

# Configure formatting options for the output graph
prefix_color = "blue"
prefix_style = "--"
suffix_color = "yellow"
suffix_style = "-"
output_format = "pdf"

# Get an available filename prefix
file_prefix = storage.get_unused_filename_prefix("map")

# Loop over the agent’s execution record
last_step = agent.execution_step
for step in range(0, last_step):
    filename = file_prefix + str(step) + "." + output_format
    agent_state = recorder.history[step]
    prefix, suffix = planmon.plan_for_step(step)
    chart = simulation.presentation.RectworldMap(agent_state.ts)
    chart.draw_agent(agent_state.ts_position)
    chart.draw_node_labels()
    chart.draw_path(suffix,
                     path_color=suffix_color,
                     path_style=suffix_style)
    chart.draw_path(prefix,
                     path_color=prefix_color,
                     path_style=prefix_style)
    chart.draw_propositions()
    chart.draw_proposition_legend()
    chart.draw_path_legend(prefix_color,
                            prefix_style,
                            suffix_color,
                            suffix_style)
    with open(filename, "w") as f:
        chart.save(output_file=f,
                   output_format=output_format)

# Create a movie of the agent’s execution history
simulation.presentation.create_movie(file_prefix,
First we set up the transition system and Buchi graphs as already described in section 5.4.6 so we will not go over that part again here.

When we have the transition system and Buchi, we proceed to initialize the agent object. In this case we are again using Dijkstra’s algorithm for planning so we select `simulation.agents.DijkstraAgentNG` and pass it the transition system, Buchi and initial states.

```python
agent = simulation.agents.DijkstraAgentNG(
    transition_system,
    initial_position,
    mission, mission.initial)
```

Next we initialize a series of monitors and attach them to the agent. The monitors serve to collect information on the agent’s execution (e.g. `simulation.monitors.PlanningPerformanceMonitor`) and stop it when it has reached some given milestone (e.g. `simulation.monitors.SuffixIterationMonitor`).

```python
perfmon = simulation.monitors.PlanningPerformanceMonitor()
perfmon.attach(agent)
...
```

Now that everything is set up, we can start the agent’s execution by calling it’s `start` method.

```python
agent.start()
```

Now the agent will have entered it’s execution loop from which it will not break unless it encounter’s an error or is instructed to by a monitor.

Once the agent has finished execution we can proceed to extract information out of the monitors that we attached. Since we will be drawing figures for each execution step we need to define some formatting options

```python
prefix_color = "blue"
...
output_format = "pdf"
```

and since we will be writing the figures to disk we ask the `storage` module for an unused path prefix using the `storage.get_unused_filename_prefix` function.

```python
file_prefix = storage.get_unused_filename_prefix("map")
```

Now we are ready loop over the agent’s execution steps. We do that by checking it’s execution step counter

```python
last_step = agent.execution_step
```
and looping from from 0 to \texttt{last\_step}.

In each iteration of the loop we extract the recorded agent’s state from our 
\texttt{simulation.monitors}\texttt{.TransitionSystemMonitor} instance

\begin{verbatim}
agent_state = recorder.history[step]
\end{verbatim}

and the plans the agent had at that step from our \texttt{simulation.monitors}\texttt{.PlanMonitor} instance

\begin{verbatim}
prefix, suffix = planmon.plan_for_step(step)
\end{verbatim}

Now we use the \texttt{simulation.presentation.RectworldMap} class to draw the 
agent’s (possibly incomplete or incorrect) model of the transition system, it’s 
current position and it’s current plan.

\begin{verbatim}
chart = simulation.presentation.RectworldMap(agent_state.ts)
chart.draw_background()
...
chart.draw_path_legend(
    prefix_color,
    prefix_style,
    suffix_color,
    suffix_style)
\end{verbatim}

When the chart is ready we save it via it’s \texttt{save} function.

\begin{verbatim}
with open(filename, "w") as f:
    chart.save(
        output_file=f,
        output_format=output_format)
\end{verbatim}

After the loop, when we have saved a figure for every execution step we combine 
them into a movie with

\begin{verbatim}
simulation.presentation.create_movie(
    file_prefix, 
    image_format=output_format)
\end{verbatim}

The end result is a set of images and a movie as listed in figure \ref{fig:movie}.
Figure 5.3: A run with the revising NDFS planner.
Figure 5.4: A run with the revising Dijkstra planner.
Figure 5.5: The relative planning time required by the three types of planners. All of them use the iterative method of product generation. Like in figure 4.1, this benchmark was done by randomly generating a set of mazes of each size on the x-axis and computing the average planning time for each planner.
In this chapter I will briefly discuss the multi-agent planning scenario and reflect on future directions.

6.1 Multi-Agent

There is a class of planning scenarios that we have not considered yet and that is one involving multiple separate robots or *agents* that cooperate to fulfill a mission specification or at least don’t crash into each other.

For multi-agent planning we have two approaches.

1. All the agents are modeled together as a whole.
2. Every agent is modeled individually and if a part of another agent’s state is relevant to it’s mission (e.g. collision avoidance), that is modeled as changes in the environment.

6.1.1 Collective

An advantage of modeling multiple agents as a collective is that they can fulfill specifications that a single agent can not (e.g. be in two places at once). The downsides are that such planning is by necessity more complicated (exponential with each added agent) and it requires synchronization if transition times are not deterministic and known a priori.

6.1.2 Individuals

The greatest advantage of modeling each agent individually and the others as part of the environment is that such planning is much simpler and therefore faster.
and more amenable to finding optimal solutions. If the different agents do not need to cooperate in order to fulfill their specifications and their specifications do not include propositions that depend on each other’s states, then there is no reason not to model them as individuals.

Even though each agent is modeled as an individual does not mean that it does not benefit from communicating with others. By communicating an agent can acquire information about the environment that can enable it to come up with realistic and optimal plans.

6.2 Future directions

In this thesis I covered symbolic planning using LTL. While LTL-based planning certainly works there are some challenges that remain. As we are interested in finding optimal or at least good plans we are especially affected the problem of having a large product graph through which search for accepting runs. Unless the transition system has a very ”hilly” topology meaning that some regions will probably not be entered because of the high cost of their incoming edges, Dijkstra’s algorithm will likely visit a large portion of the product graph making the size of the product graph the most significant contributing factor to the computational cost.

6.2.1 Better partitioning

We do not have much control over the Buchi for a given LTL formula which leaves us with the transition system and its partitioning scheme as our primary means of controlling the product size. The most straight forward partitioning approach of partitioning the environment on a grid as well as many of the automated polyhedral partitioning schemes produce a finite transition system that may be finer grained than necessary in some or all areas. A better approach would be to follow the following steps

1. Start with a polyhedral or grid partitioning scheme.
2. Merge nodes that have the same set of atomic propositions and are mutually reachable (they are neighbors and are in each others pre and post sets).
3. Place the merged nodes in a separate graph and keep the original partitioning scheme intact.
4. Create edges between the merged nodes by computing the shortest path between their composite nodes in the original partitioning scheme and saving those paths as properties of the edges.
6.2 Future directions

When this procedure is done we will have two transition systems, one based on the original partitioning scheme and another smaller one with only a few nodes representing the distinct sets of fulfilled atomic propositions. Now we can use the smaller graph for planning and expand the full run by looking up the shortest paths that the edges in that graph represent. For many scenarios, this should produce a product graph that is much smaller and therefore much quicker to search through.

There is (at least) one downside to this method and that is that there might be edges with non-zero weights inside the merged nodes that constitute hidden costs. Here there is a trade-off to be made: one can either choose only to merge nodes that have edges in each direction that have zero weight or one could use some heuristic or rule to choose which nodes to merge based on what internal edge weight is acceptable.

Implementing this partitioning procedure and observing its effectiveness for different kinds of scenarios would be an interesting exercise.
References


