A Discrete Büchi Automata Distance for Formal Methods Based Control

Garrett Thomas

Master's Thesis in Automatic Control (30 Credits)
Master's Program in Applied and Computational Mathematics (120 credits)
Examiner: Professor Dimos Dimarogonas
Supervisor: Lars Lindemann

June 28th, 2017

Abstract

Model checking has proven to be a valuable design validation tool to ensure the correctness of hardware and software design. Recently, symbolic model checking using temporal logics has shown great promise in the field of control and task planning synthesis, as it allows for the formulation of complex tasks and provides an automatic and exhaustive search of all possible paths. The major drawback of this method is what is known as the state-space explosion problem. Even with relatively small environments and formulas, the resulting product automaton can become too large to search. Various methods exist which attempt to mitigate the problem, however these methods have limited applicability in the control and task planning synthesis context. We propose a greedy algorithm to address this problem. Our algorithm makes use of a novel distance measure in the Büchi automaton corresponding to the specified formula. At each step the optimal path is found which decrease this distance, which together produces an approximation of the globally optimal path. The performance of this algorithm is then analysed on various control planning synthesis examples from the literature and compared to the current accepted algorithm.

Sammanfattning

Acknowledgements

Contents

List of Figures

Chapter 1

Introduction

1.1 Problem

The use of formal methods, specifically *model checking*, in control planning synthesis is a new and exciting research area. Formal methods are originally mathematical techniques to specifiy and verify the design of software and hardware [7]. *formal verification* methods ensure that there are no bugs present in the software or hardware design which will cause unexpected behaviors. This is needed because other debugging techniques such as simulation and testing only ensure that there are no problems with a given the input. Bugs that go undetected can have disastrous effects, such as the explosion of the Ariane 5 rocket in 1996, which was caused by an exception being thrown when a 64-bit floating point number was converted to a 16-bit signed integer [6].

Model checking is an approach to formal verification which decides if a model of the program satisfies some behavior. These behaviors can be given as a temporal logic formula, commonly linear temporal logic (LTL) or computation tree logic (CTL). Since its advent, model checking under temporal logics has proved to be a very useful tool in software and hardware development. There are commercial and open source model checkers available [15], [5] and many companies have their own in house model checking programs.

Two main benefits of model checking are that 1) it is completely autonomous; after the program is modelled and the behavior is formalized into temporal logic, the model checker is self sufficient and no longer requires user interaction and 2) in the situation when the model fails to satisfy the given formula, a counter example is given (*why* the model does not satisfy the specification).

It is obvious to see how the counter example gives valuable information

for debugging purposes. This counter example however has also given rise to the new and exciting field of model planning as a tool for control planning synthesis. In this field of motion and task planning, the model of the program is replaced by a model of the robot's environment and the formula is the *negation* of the desired robot motion and tasks. The realization that the double negative of a *counter example* of the *negation* of the desired motion was in fact the desired motion is the basis of this field. Earlier works in this field have even used the same programs that were designed for classic model checking [10].

Model checking however is not perfect. It suffers from what is known as the *state-space explosion problem*. This is a phenomenon in which the number of states can quickly grow to an unmanageable number rendering the model checking uncomputable. Neither classical model checking or the application of model checking in motion planning is safe from the state explosion problem, however techniques have been developed to address this problem in classical model checking. These techniques include partial order reduction and abstraction, among others. Partial order reduction consists of trying to reduce the number of independent interleavings of concurrent processes, while abstraction attempts to simplify the model. These techniques have proved useful for classical model checking, however they unfortunately have limited applicability in model checking for robot motion.

To address the state space explosion problem, we are presenting a method that does not reduce the number of states. It does however speed up the following search through the states. Instead of an algorithm that computes the globally optimal path, we instead propose an algorithm that takes the optimal path at each step according to the greedy paradigm. The performance of our proposed algorithm is then analysed in the context of robot motion planning and compared to the accepted algorithm from the literature. The algorithm is also applicable to classical model checking, however this is out of the scope of this work.

1.2 Outline

Chapter 2 We will first present all the theoretical results necessary for understanding the technique of model checking for motion planning. This will be self contained and does not require any prior knowledge in the field.

Chapter 3 We will review the current algorithm that is accepted and widely used and also our algorithm. We will review key differences in the

procedures.

Chapter 4 We will test both algorithms on various formulas common in the field of motion planning for robot motion. The results are compared.

Chapter 5 We will move on to more complex formulas including mixtures of motion planning and task planning. The results of the two algorithms are compared.

Chapter 6 We draw conclusions and give recommendations of possible future work.

Chapter 2

Theoretical Background

In this chapter we provide the theoretical background that is needed to understand control planning synthesis and linear temporal logic.

2.1 Abstraction of the Workspace

In [3], Belta et al describe robot path planning as consisting of three parts: the specification level, execution level, and the implementation level. The first level, the specification level, involves creating a graph that takes into account the robot motion, workspace, and desired behavior. Next is the execution level, which involves finding a discrete path through the graph that satisfies the desired behavior. Lastly, in the implementation level, controllers are constructed such that the continuous trajectory satisfies the discrete path found in the previous step.

We assume that we have one robot which is located in a given bounded workspace denote $W_0 \subset \mathbb{R}^n$. To represent our workspace (which is a subspace of $\mathbb{R}^n$) in a finite graph we must partition it into a finite number of equivalence classes. A partition map is formally defined in definition 1. Any partition can be used as long as it satisfies the bisimulation property [4], which will be defined later once more notation has been introduced. We denote $\Pi = \{\pi_1, \pi_2, \ldots, \pi_w\}$ to be the set of equivalence classes the workspace has been partitioned into, and thus $\cup_{i=1}^{w} \pi_i = W_0$ and $\pi_i \cap \pi_j = \varnothing$, $\forall i, j = 1, 2, \ldots, w$ and $i \neq j$. We will henceforth refer to equivalence class π_i as region π_i for $i = 0, 1, \ldots, w$.

Definition 1. *A partition map, $T : W_0 \to \Pi$ sends each state $x \in W_0$ to the finite set of equivalence classes $\Pi = \pi_i$, $i = 1, 2, \ldots, w$. $T^{-1}(\pi_i)$ is then all the states $x \in W_0$ that are in the equivalence class π_i [10].*

We now introduce atomic propositions, which will be the building blocks of our task specification. Atomic propositions are boolean variables, and are used to express properties the robot and the workspace. We define the following set of atomic propositions $AP_r = \{\alpha_{r,i}\}$, $i = 1, 2, \ldots, w$ where

$$\alpha_{r,i} = \begin{cases} \top & \text{if the robot is in region } \pi_i \\ \bot & \text{else} \end{cases}$$

which represent the robot's location [13]. Note: $\top$ is the true logical predicate and $\bot$ is the false logical predicate. Other things we can express are potential tasks, denote $AP_p = \{\alpha_{p,i}\}$, $i = 1, 2, \ldots, m$. These can be statements such as "pick up the ball in region π_1" or "the robot beeps". The set of all propositions is defined as $AP = AP_r \cup AP_p$.

We now formally define a *Labelling Function* and a *Transition*, which will immediately be used in the definition *Bisimulations* and of a *Finite-State Transition System (FTS)*.

Definition 2. *The continuous labelling function $L_c : W_0 \to 2^{AP}$ maps a point $x \in W_0$ to the set of atomic propositions satisfied by x [13].*

Note: 2^{AP} is the powerset of AP, i.e. the set of all subsets of AP include the null set and AP. For example, by definition, $\alpha_{r,i} \in L_c(\pi_i)$. We also include a definition of the discrete counterpart.

Definition 3. *The labelling function $L_d : \Pi \to 2^{AP}$ maps a region $\pi_i \in \Pi$ to the set of atomic propositions satisfied by π_i.*

To define a graph that represents our environment, we must also consider the dynamics of the robot. The dynamics are relevant because they define the relationship between the various regions. The relationship we refer to is known as a transition. We define a transition between two points in W_0 as follows

Definition 4. *There is a continuous transition, $\to_c \subset W_0 \times W_0$ from x to x', denoted $x \to_c x'$ if it is possible to construct a trajectory $x(t)$ for $0 \leq t \leq T$ with $x(0) = x$ and $x(T) = x'$ and we have $x(t) \in (T^{-1}(T(x)) \cup T^{-1}(T(x')))$ [9]*

We then say that there is a transition between two regions if from any point in the first region there is a transition to a point in the second region. More formally

Definition 5. *There is a discrete transition, $\to_d \subset \Pi \times \Pi$, from π_i to π_j, denoted $\pi_i \to_d \pi_j$ if for every x in π_i i.e. $T(x) = \pi_i$ there exists x' such that $T(x') = \pi_j$ and $x \to_c x'$*

Note: $\pi_i \rightarrow_d \pi_i, \ \forall 1, 2, \ldots w$ We can now define bisimulations

Definition 6. *A partition $T : W_0 \rightarrow \Pi$ is called a bisimulation [9] if the following properties hold for all $x, y \in W_0$*

1. *(Observation Preserving): If $T(x) = T(y)$, then $L_c(x) = L_c(y)$.*
2. *(Reachability Preserving): If $T(x) = T(y)$, then if $x \rightarrow_c x'$ then $y \rightarrow_c y'$ for some y' with $T(x') = T(y')$*

The Observation Preserving requirement makes sure we do not allow the situation where part of π_i fulfils $\alpha \in AP$ while part of π_i does not, and the Reachability Preserving requirement ensures that for every point in region π_i, there exists a trajectory to some point x', such that $T(x') = \pi_j$ if $\pi_i \rightarrow_d \pi_j$. These two requirements together guarantee that the discrete path we compute is feasible at the continuous level.

We can now define an FTS, which is how we will represent our workspace and robot motion.

Definition 7. *An FTS, $\mathcal{T}$, is defined by a tuple*

$$\mathcal{T} = (\Pi, \rightarrow_d, \Pi_0, AP, L_d)$$

where Π is the set of states, $\rightarrow_d \subseteq \Pi \times \Pi$ is the transitions relation where $(\pi_i, \pi_j) \in \rightarrow_d$ iff there is a transition from π_i to π_j as defined in definition 5. In adherence to common notation, we will write $\pi_i \rightarrow_d \pi_j$. $\Pi_0 \subseteq \Pi$ is the initial state(s), $AP = AP_r \cup AP_p$ is the set of atomic propositions, and $L_d : \Pi \rightarrow 2^{AP}$ is the labelling function defined in definition 3.

An FTS can also have *weights* associated with each transition (the *cost* of the transition) which is known as a weighted FTS (WFTS). We will use only WFTS in this thesis.

Definition 8. *A WTFS, $\mathcal{T}_w$ is a tuple*

$$\mathcal{T}_w = (\Pi, \rightarrow_d, \Pi_0, AP, L_d, W_d) \tag{2.1}$$

where Π, $\rightarrow_d$, Π_0, AP, and L_d are defined as in definition 7 and $W_d : \Pi \times \Pi \rightarrow \mathbb{R}^+$ is the weight function i.e. the cost of a transition in $\rightarrow_d$.

Note: There are two common ways to assign the weights. The first being setting every weight to one (simply count the number of transitions taken) and the second assigning the distance from the centres of two adjacent cells to be the weight of the transition between them.

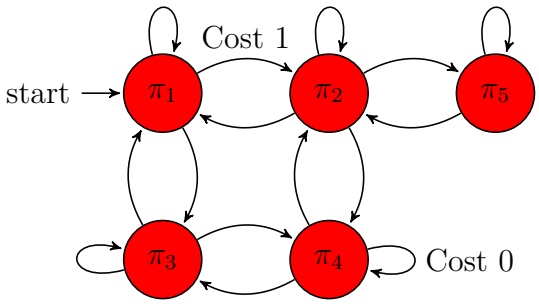

Figure 2.1: Simple Finite Transition System

For the simple FTS, the set of states Π is $\{\pi_1, \pi_2, \pi_3, \pi_4, \pi_5\}$, we have the following transitions

$$\begin{array}{lll}
\pi_1 \rightarrow_d \pi_1, & \pi_1 \rightarrow_d \pi_2, & \pi_1 \rightarrow_d \pi_3 \\
\pi_3 \rightarrow_d \pi_1, & \pi_3 \rightarrow_d \pi_3, & \pi_3 \rightarrow_d \pi_4 \\
\pi_4 \rightarrow_d \pi_2, & \pi_4 \rightarrow_d \pi_3, & \pi_4 \rightarrow_d \pi_4 \\
\pi_2 \rightarrow_d \pi_1, & \pi_2 \rightarrow_d \pi_2, & \pi_2 \rightarrow_d \pi_4 \\
\pi_2 \rightarrow_d \pi_5, & \pi_5 \rightarrow_d \pi_2, & \pi_5 \rightarrow_d \pi_5
\end{array}$$

the initial state Π_0 is π_1, the set of atomic propositions is AP= $\{\pi_1, \pi_2, \pi_3, \pi_4, \pi_5\}$, and the labelling function is defined as $L_d(\pi_i) = \pi_i$ for $i = 1, 2, \ldots, 5$. We will treat this as a WFTS. The weights are approximated as the distance between the centers of two states in which a transition exists between. All the states are squares, thus this distance is the same for all transitions. We therefore let the weights be 1 for every transition and 0 to stay in the same state i.e. $W_D(\pi_i, \pi_j) = 1$ and $W_d(\pi_i, \pi_i) = 0$ for $i = 1, 2, \ldots, 5$ and for $i \neq j$.

We use the FTS which represents our workspace to search for paths that are doable for our robot. When we search for a path, from one state we will only consider states which have a transition from our current state, because these are the only states the robot can move to. In this thesis, we will deal with infinite paths. An infinite path is an infinite sequence of states $\tau = \pi_0\pi_1 \ldots$ such that $\pi_0 \in \Pi_0$ and $\pi_i \in \Pi$ with $\pi_i \rightarrow_d \pi_{i+1}$, $\forall i > 0$. The trace of a path is the sequence of sets of atomic propositions that are true in the states along a path i.e. $\texttt{trace}(\tau) = L_d(\pi_0)L_d(\pi_1)\ldots$.

2.2 Linear Temporal Logic (LTL)

To define tasks for our robot we must choose a high level language. Temporal logics are especially suited for defining robot tasks because of their ability

to express not only fomulas constructed of atomic propositions and standard boolean connectives, but also temporal specifications e.g. α is true at some point of time. The particular temporal logic we will be using is known as linear temporal logic (LTL) [6]. LTL formulas are defined over a set of atomic propositions AP according to the following grammar:

$$\varphi ::= \top | \alpha | \neg\varphi_1 | \varphi_1 \vee \varphi_2 | \mathbf{X}\varphi_1 | \varphi_1 \mathcal{U} \varphi_2$$

where $\top$ is the predicate true, $\alpha \in AP$ is an atomic proposition, φ_1 and φ_2 are LTL formulas, $\neg$ and $\vee$ denote the standard Boolean connectives negation and disjunction respectively, X being the "Next" operator. $\mathcal{U}$ is the temporal operator "Until", with $\varphi_1 \mathcal{U} \varphi_2$ meaning φ_1 is true until φ_2 becomes true. Given these operators, we can define the following additional prepositional operators:

$$\begin{aligned} \text{Conjunction: } & \varphi_1 \wedge \varphi_2 = \neg(\neg\varphi_1 \vee \neg\varphi_2) \\ \text{Implication: } & \varphi_1 \Rightarrow \varphi_2 = \neg\varphi_1 \vee \varphi_2 \\ \text{Equivalence: } & \varphi_1 \Leftrightarrow \varphi_2 = (\varphi_1 \Rightarrow \varphi_2) \wedge (\varphi_2 \Rightarrow \varphi_1) \end{aligned}$$

We note quickly that we have the false predicate, $\bot = \neg\top$. We are also able to derive the following additional temporal operators:

$$\begin{aligned} \text{Eventuality: } & \diamond \varphi_1 = \top \mathcal{U} \varphi_1 \\ \text{Always: } & \Box \varphi_1 = \neg \diamond \neg \varphi_1 \end{aligned}$$

There is a growing interest in path and task planning in robots using temporal logic specifications given the easy extension from natural language to temporal logic [16]. We now give examples to illustrate this point and to introduce us to LTL formulas. There are some common tasks converted to LTL formulas given in [9]

1. **Reachability while avoiding regions**: "Go to region π_{n+1} while avoiding regions $\pi_1, \pi_2, \ldots, \pi_n$"
   $\neg(\pi_1 \vee \pi_2 \ldots \pi_n) \mathcal{U} \pi_{n+1}$

2. **Sequencing**: "Visit regions π_1, π_2, π_3 in that order"
   $\diamond(\pi_1 \wedge \diamond(\pi_2 \wedge \diamond\pi_3))$

3. **Coverage**: "Visit regions $\pi_1, \pi_2, \ldots, \pi_n$ in any order"
   $\diamond\pi_1 \wedge \diamond\pi_2 \wedge \cdots \wedge \diamond\pi_n$

4. **Recurrence (Liveness)**: "Visit regions $\pi_1, \ldots, \pi_n$ in any order over and over again"
   $\square(\diamond\pi_1 \wedge \diamond\pi_2 \wedge \cdots \wedge \diamond\pi_n)$

Of course more complicated tasks are also expressible in LTL, and atomic propositions need not only refer to the location of the robot. Here is an example given in [13]: "Pick up the red ball, drop it to one of the baskets and then stay in room one"
$\diamond(rball \wedge \diamond basket) \wedge \diamond\square r1$

We now look at what it means to satisfy an LTL formula. We will talk about *words* satisfying LTL formulas, in our case *infinite words*. An infinite word over the alphabet 2^{AP} is an infinite sequence $\sigma \in (2^{AP})^\omega$. The ω superscript means an infinte repetition; that is, $\sigma = S_0 S_1 S_2 \ldots$, where $S_k \in 2^{AP}$ for $k = 1, 2, \ldots$ and S_k is the set of atomic propositions that are true at time step k [13]. An infinite word σ satisfies an LTL formula φ based on the LTL semantics.

Definition 9. *The semantics of LTL are defined as follows:*

$$\begin{aligned}
(\sigma, k) \vDash \alpha \quad & \textit{iff} \quad \alpha \in S_k \\
(\sigma, k) \vDash \neg\varphi \quad & \textit{iff} \quad (\sigma, k) \nvDash \varphi \\
(\sigma, k) \vDash \boldsymbol{X}\varphi \quad & \textit{iff} \quad (\sigma, k+1) \vDash \varphi \\
(\sigma, k) \vDash \varphi_1 \vee \varphi_2 \quad & \textit{iff} \quad (\sigma, k) \vDash \varphi_1 \ \textit{or} \ (\sigma, k) \vDash \varphi_2 \\
(\sigma, k) \vDash \varphi_1 \mathcal{U} \varphi_2 \quad & \textit{iff} \quad \exists k' \in [k, +\inf], \ (\sigma, k') \vDash \varphi_2 \ \textit{and} \\
& \qquad \forall k'' \in (k, k'), \ (\sigma, k'') \vDash \varphi_1
\end{aligned}$$

Where (σ, k) refers to σ at time step k. An infinite word σ is said to satisfy formula φ if $(\sigma, 0) \vDash \varphi$. For the ease of reading we will refer to $(\sigma, 0)$ as σ.

There is a connection between these infinite words and the FTS described earlier that is crucial in motion planning technique. Given an infinite path τ of an FTS, we have that the trace of the path, $\texttt{trace}(\tau)$, is an infinite word over the alphabet 2^{AP}. Given the LTL semantics, we now have the ability to verify if a path satisfies an LTL formula! We will say an infinite path τ *satisfies* φ if its trace satisfies φ, i.e. $\tau \vDash \varphi$ if $trace(\tau) \vDash \varphi$. A path satisfying φ is called a *plan* for φ. We will use 'plan' and 'accepting path' interchangeably.

2.3 Büchi Automata

We can now tell if a path of an FTS satisfies a given LTL formula, however we are interested in *generating* paths that satisfy a given formula, which requires

more work! To do this we are going to need a finite representation of a given LTL formula that we can search. This representation is a Nondeterministic Büchi automaton (NBA).

Definition 10. *An NBA $\mathcal{A}_\varphi$ is defined by a tuple:*

$$\mathcal{A}_\varphi = (\mathcal{Q}, 2^{AP}, \delta, \mathcal{Q}_0, \mathcal{F})$$

where $\mathcal{Q}$ is a finite set of states, $\mathcal{Q}_0 \subseteq \mathcal{Q}$ is the set of initial states, 2^{AP} is the alphabet, $\delta : \mathcal{Q} \times 2^{AP} \to 2^{\mathcal{Q}}$ is a transition relation, and $\mathcal{F} \subseteq \mathcal{Q}$ is the set of accepting states

An infinite run of an NBA is an infinite sequence of states, $r = q_0 q_1 \ldots$, that starts from an initial state i.e. $q_0 \in Q_0$ and $q_{k+1} \in \delta(q_k, S)$ for some $S \in 2^{AP}$, for $k = 0, 1, \ldots$. The requirements for a run r to be accepting is $\texttt{Inf}(r) \cap \mathcal{F} \neq \varnothing$, where $\texttt{Inf}(r)$ is the set of states that appear in r infinitely often [13].

Now to tie together the concept of words and runs on an NBA. An infinite word $\sigma = S_0 S_1 \ldots$ corresponds to $r_\sigma = q_0 q_1 \ldots$ if $q_0 \in Q_0$ and $q_{i+1} \in \delta(q_i, S_i)$

It has been shown that given an LTL formula φ over AP, there exists a NBA over 2^{AP} corresponding to φ, denoted A_φ [2]. When we say an NBA corresponds to an LTL formula, we mean that the set of words that corresponds to accepting runs of the NBA is the same as the set of words accepted by the LTL formula.

2.4 Product Automata

These two structures are then combined to create the product automaton. The product automata is also a Büchi automata and is defined as follows:

Definition 11. *The weighted product Büchi automaton is defined by $\mathcal{A}_p = \mathcal{T}_w \otimes \mathcal{A}_\varphi = (Q', \delta', Q'_0, \mathcal{F}', W_p)$, where $Q' = \Pi \times Q = \{\langle \pi, q \rangle \in Q' | \forall \pi \in \Pi,\ \forall q \in Q\}$; $\delta' : Q' \to 2^{Q'}$. $\langle \pi_j, q_n \rangle \in \delta'(\langle \pi_i, q_m \rangle)$ iff $(\pi_i, \pi_j) \in \to_c$ and $q_n \in \delta(q_m, L_d(\pi_j))$; $Q'_0 = \{\langle \pi, q \rangle | \pi \in \Pi_0,\ q_0 \in Q_0\}$, the set of initial states: $\mathcal{F}' = \{\langle \pi, q \rangle | \pi \in \Pi, q \in \mathcal{F}\}$, the set of accepting states; $W_p : Q' \times Q' \to \mathbb{R}^+$ is the weight function: $W_p(\langle \pi_i, q_m \rangle, \langle \pi_j, q_n \rangle) = W_d(\pi_i, \pi_j)$, where $\langle \pi_j, q_n \rangle \in \delta'(\langle \pi_i, q_m \rangle)$*

Given a state q' $= \langle \pi, q \rangle \in Q'$, its projection on Π is denoted by $q'|_\Pi = \pi$ and its projection on Q is denoted by $q'|_Q = q$. Given an infinite run $R = q'_0 q'_1 q'_2 \ldots$ of $\mathcal{A}_p$, its projection on Π is denoted by $R|_\Pi = q'_0|_\Pi q'_1|_\Pi q'_2|_\Pi \ldots$ and its projection on Q is denoted by $R|_Q = q'_0|_Q q'_1|_Q q'_2|_Q \ldots$ [13].

Note: Given that $\mathcal{A}_p$ is a Büchi automaton, the requirements of an accepting run are the same as before i.e. $\texttt{Inf}(R) \cap \mathcal{F}' \neq \varnothing$.

Our problem is now to find an accepting run of $\mathcal{A}_p$. This can be a difficult task, given that an accepting run is a infinite sequence of states, and there are infinitely many possibilities. We also want to have some sort of measure of optimality, making the problem harder. To accomplish this, we are going to restrict our search to plans with a finite representation. This limits the plans that we can calculate, however it is much easier to deal with paths that admit a finite representation. Specifically, we are going to be looking for paths in the prefix-suffix structure i.e.

$$\tau = \langle \tau_{pre}, \tau_{suf} \rangle = \tau_{pre}[\tau_{suf}]^{\omega}$$

The prefix, τ_{pre}, is the path from an initial node to an accepting node. The suffix, τ_{suf}, is going to be a path from the same accepting node back to itself. So the full path is going to be the prefix and then the suffix repeated infinitely many times (which is the meaning of the ω superscript). Thus, the accepting node appears infinitely many times in τ which makes τ accepting. Plans of this form are preferred because, while they are still infinite plans, they have a finite representation which is easier to deal with.

2.5 Cost of a Run

As we said before, we want to have a way to measure the optimality of a run. We introduce the concept of the *cost* of a run to satisfy this requirement. We are focusing on the accepting runs of $\mathcal{A}_p$ with the prefix-suffix structure

$$\begin{aligned} R = \langle R_{pre}, R_{suf} \rangle &= q'_0 q'_1 \dots q'_f [q'_{f+1} \dots q'_n q'_f]^{\omega} \\ &= \langle \pi_0, q_0 \rangle \dots \langle \pi_f, q_f \rangle [\langle \pi_{f+1}, q_{f+1} \rangle \dots \langle \pi_n, q_n \rangle \langle \pi_f, q_f \rangle]^{\omega} \end{aligned}$$

where $q'_0 \in \mathcal{Q}'_0$, $q'_f \in \mathcal{F}'$ and $q_f \in \mathcal{F}$.

As we can see, our path is a sequence of states, $q'_0, q'_1, \dots, q'_n$ in $\mathcal{A}_p$, where $q'_{i+1} \in \delta'(q'_i)$ for all $i = 0, 1, \dots, n-1$. Each of these transitions has a weight or cost associated with it, given by $W_p(q'_i, q'_{i+1}) = W_d(q'_i|_{\Pi}, q'_{i+1}|_{\Pi})$. We simply define the cost of our path as the sum of the cost of the transitions in the path, with the cost of the suffix being weighted.

$$\begin{aligned} \texttt{Cost}(R, \mathcal{A}_p) &= \sum_{i=0}^{f-1} W_p(q_i, q_{i+1}) + \gamma \sum_{i=f}^{n-1} W_p(q_i, q_{i+1}) \\ &= \sum_{i=0}^{f-1} W_d(\pi_i, \pi_{i+1}) + \gamma \sum_{i=f}^{n-1} W_d(\pi_i, \pi_{i+1}) \end{aligned}$$

where $\gamma \geq 0$ is the relative weighting of the transient response (prefix) cost and steady response (suffix) cost [13]. We will be using $\gamma = 1$, meaning we give the same weight to transitions in the prefix as in the suffix. In [9] they say that they search for the path with the least amount of transitions and say this is the optimal path. This is an example converting a FTS to a WFTS by setting the weight of every transition to one.

We will denote the accepting run with prefix-suffix structure that minimizes the total cost as R_{opt}, with the corresponding plan $\tau_{opt} = R_{opt}|_{\Pi}$. We note however that this plan may not actually be the true optimal plan with prefix-suffix structure. In [20] we see that simplifications in the translation from LTL formulas to NBA can result in a loss of optimality. These NBA that do not have the optimality property are referred to as not *tight* NBA. This will come up again when we analyse the paths our algorithm generates.

Chapter 3

Search Algorithms

The task is now to compute a path that satisfies our LTL formula. The current accepted algorithm does an exhaustive search of the product automaton to find the optimal path (again this may not actually be the optimal path [20]. This however is a computationally intensive task. We present an approximation algorithm that gives a *good* path, but not necessarily the optimal path. This can be attractive if the cost of the path is not of dire importance. We first present the current standard algorithm and then our algorithm.

3.1 Accepted Algorithm

The search algorithm used in many recent works on the specific type of control planning synthesis comes from this prefix-suffix structure. The basic idea is to find a path from an initial node, q_0 to an accepting node, q_f, and then find a path from the q_f back to itself. The first part from q_0 to q_f is the prefix and the second part q_f back to q_f is the suffix. Then the resulting path, τ, will be the prefix, followed by the suffix repeated infinitely many times. This path is thus accepting because the suffix finds the path from an initial state back to itself, and thus contains the initial state, and is repeated infinitely many times $q_f \in \texttt{Inf}(\tau) \Rightarrow \texttt{Inf}(\tau) \cap \mathcal{F} \neq \varnothing$. This algorithm, or simple variations of it, are used in many works on motion planning synthesis [9], add more, so we will refer to it as the *accepted* algorithm.

Procedure 1, modified from [13], gives the pseudocode for computing R_{opt}.

Meng Guo has created a public github repository, P-MAS-TG (Planner for Multiple Agent System with Temporal Goals) [14]. The function `dijkstra_plan_networkX` in `P_MAS_TG`
`discrete_plan.py` is approximately equvialent to Algorithm 1. The work of

Procedure 1 OptRun()

Input: Input $\mathcal{A}_p$

Output: R_{opt}

1: From the initial state $q'_0 \in \mathcal{Q}'_0$, find the optimal path to each $q'_f \in \mathcal{F}$.
2: For each accepting state $q'_f \in \mathcal{F}'$, calculate the optimal path back to q'_f.
3: Find $q'_{f,opt}$ that minimizes the total cost
4: Optimal accepting run R_{opt}, prefix: shortest path from q'_0 to q'_{f*}; suffix: the shortest cycle from q'_{f*} and back to itself.

finding the optimal path from q'_f back to q'_f and from q'_0 to all q'_f is done by `dijkstra_predecessor_and_distance` from the NetworkX python package [19]. `dijkstra_predecessor_and_distance`$(\mathcal{A}_p, q_0)$ returns two dictionaries; one containing a list of all the nodes q_0 is a predecessor of and one containing the distances to each of these nodes. When we provide computational examples for the accepted algorithm, we will be using this repository.

The worst case computational complexity of this algorithm $\mathcal{O}(|\delta'| \cdot \log|\mathcal{Q}'| \cdot (1 + |\mathcal{F}'|))$ because the worst case complexity for a Dijkstra search is $\mathcal{O}(|\delta'| \cdot \log|\mathcal{Q}'|)$ and Algorithm 1 does $(1 + |\mathcal{F}'|)$ Dijkstra searches (one for the initial node and one for each accepting node).

3.2 Our Algorithm

As we can see, the current algorithm has to do a lot of work. First it has to do Dijkstra's search for each initial state, and then one for each accepting state (the number of accepting states is at least the size of the FTS). The state space that is being searched can also become very big, which is known as the state explosion problem [6]. The size of the product automaton, $|\mathcal{A}_p|$ is the size of the Büchi automaton corresponding to the LTL formula times the size of the FTS i.e. $|\Pi| \cdot |\mathcal{Q}|$. The size of the Büchi automaton corresponding to the LTL formula can be exponential in the size of the formula [12]. We can imagine how much searching is needed if we have and FTS and Büchi that are both fairly large. To solve this problem, we suggest a greedy algorithm that sacrifices optimality but preforms faster than the current accepted algorithm. A greedy algorithm is an algorithm that chooses the locally optimal path at each at each stage in an attempt to approximate the globally optimal path [8].

The idea stems from the fact that $q' = \langle \pi, q \rangle \in \mathcal{Q}'$ is an accepting state of $\mathcal{A}_p$ if and only if $q \in Q$ is an accepting state of $\mathcal{A}_\varphi$. Thus finding an accepting state in the product automaton is essentially finding an accepting state of the

LTL Büchi automaton. We therefore suggest assigning a distance measure in the LTL Büchi automaton that carries over to the product automaton. To do this, we first define a Büchi automaton that includes information on the distance to an accepting state.

Definition 12. *An NBA with distance, NBAD, is defined by a six-tuple:*

$$\mathcal{A}_{\varphi,d} = (\mathcal{Q}, 2^{AP}, \delta, \mathcal{Q}_0, \mathcal{F}, d)$$

where $\mathcal{Q}, 2^{AP}, \delta, \mathcal{Q}_0, \mathcal{F}$ *are defined as in definition 10 and* $d : \mathcal{Q} \to \mathbb{Z}$ *is defined as*

$$d(q_n) = \min_i \{i \mid q_i \in \mathcal{F} \text{ and } q_{k+1} \in \delta(q_k, S_k) \text{ for some } S_k \in 2^{AP},\ k = n, \ldots, i-1\}$$

which is the number of transitions in the shortest path from q_n *to an accepting state.*

Then we also have a product automaton with distance, $\mathcal{A}_{p,d} = \mathcal{T} \otimes \mathcal{A}_\varphi = (Q', \delta', Q'_0, \mathcal{F}', W_p, d_p)$, defined similarly, with $d_p(q') = d(q'|\mathcal{Q})$. We will refer to q' as being on level n if $d_p(q') = n$.

The idea of our algorithm is to start from $q'_0 \in \mathcal{Q}'$, say $d_p(q'_0) = n$ and then in adherence to the greedy paradigm finds the optimal path to q_i where $d_p(q'_0) = n - 1$. To do this we use a Dijkstra search to find the closest node that is on next smallest level, $n-1$. Then we will do another Dijkstra search on the next level down to find the closest node that has a transition down, and so on. This ensures that we will approach the accepting states i.e. those states on level 0.

Once we reach an accepting state, we must use a Dijkstra search of the whole product automaton. This is because the idea of decreasing levels cannot be used to find a specific accepting state, simply *an* accepting state. Therefore we need to use a Dijkstra search which will search through all the nodes until it finds the accepting state we are looking for. As opposed to the accepted algorithm, we only find the path from one accepting node back to itself. We are assuming that the closest accepting node will be a good node to use in terms of the cost of the prefix and the suffix. We will see that this assumption saves time and usually results in a path with the same or similar cost to optimal path.

Algorithm 2 is equivalent to the function `greedy_plan` which is provided in the appendix. This code was based on `dijkstra_plan_networkX` from [14] and still shares some of the structure. Finding the closest node on the level below the current level, i.e. q'_n s.t. $d_p(q'_n)$ == LEVEL $-$ 1 is done using the function `adapted_dijkstra_multisource` which is also included in the

Procedure 2 GreedyRun()

Input: Input $\mathcal{A}_{p,d}$
Output: R_g

1: Level = $d_p(q'_0 \in \mathcal{Q}'_0)$
2: **while** Level > 0 **do**
3: find optimal path down to q'_n s.t. $d_p(q'_n)$ == LEVEL – 1
4: LEVEL = LEVEL - 1
5: Find optimal path from q'_n back to itself
6: Accepting run R_g: the optimal paths calculated in the while loop concatenated together; suffix: optimal path from q'_n back to itself.

appendix. This code was based on the function `_dijkstra_multisource` in [19]. When we provide computation runs of the greedy algorithm in the following text, we will be referring to runs done with this algorithm. All computations were done on a 2.5 GHz MacBook Pro and used Python 2.7.5.

As we can see, assuming that we reach an accepting state and that there exists a path from this accepting state back to itself, we will do $n+1$ searches. This still may seem like a lot, however the searches are done on much smaller graphs than the accepted algorithm. The first n searches only look at graphs with $|\Pi|$ nodes i.e. the number of states $\mathcal{T}$, the FTS, has. These smaller graphs have a number of edges less than or equal to $|\rightarrow_d|$ i.e. the number of edges $\mathcal{T}$ has. This is because $\langle \pi_j, q_n \rangle \in \delta'(\langle \pi_i, q_m \rangle)$ if and only if $(\pi_i, \pi_j) \in \rightarrow_d$ and $q_n \in \delta(q_m, L_d(\pi_i))$, which implies the number of edges on one level is less than or equal to $|\rightarrow_d|$. From the accepting state we find, we must do one search to find the optimal path from this state back to itself. In the worst case scenario, this search has to look through the entire product automaton. Thus, resulting in a complexity of $\mathcal{O}(|\delta'| \cdot \log|\mathcal{Q}'|)$ as before. Therefore our worst case complexity will be $\mathcal{O}(|\rightarrow_d| \cdot \log|\mathcal{T}| \cdot n) + \mathcal{O}(|\delta'| \cdot \log|\mathcal{Q}'|) = \mathcal{O}(|\rightarrow_d| \cdot \log|\mathcal{T}| \cdot n + |\delta'| \cdot \log|\mathcal{Q}'|)$ where n is the level of the initial node. This complexity is applicable if the greedy algorithm finds an accepting node, the accepting node has a path back to itself, and there are no transfers on the same level of the Büchi automaton i.e. if there is a transfer from q_i to q_{i+1}, then $d(q_i) \neq d(q_{i+1})$.

We now analyse how this algorithm performs under certain LTL formulas.

Chapter 4

Algorithm Performance with Common Formulas

In this chapter we show the respective performances of the current accepted algorithm and the greedy algorithm. We do so by theoretically analyzing the general forms of four types of formulas, then computing an accepting path using both formulas and considering and cost of the path and the time taken to calculate it.

4.1 Reachability while avoiding regions

Reachability while avoiding regions is a property in which we wish to not cross over certain regions, say $\pi_1, \pi_2, \ldots, \pi_n$, until we get to a specified region, say π_{n+1}. After reaching π_{n+1} we are free to do what we want. This behaviour is expressed by the general formula $\neg(\pi_1 \vee \pi_2 \vee \ldots \pi_n)\boldsymbol{\mathcal{U}}\pi_{n+1}$.

The Büchi automaton corresponding to this formula is given in figure 4.1. As we can see $d_p(q_1) = 1$ and $d_p(q_2) = 0$. We look at the specific formula $\neg\pi_4\boldsymbol{\mathcal{U}}\pi_5$. The product automaton of this Büchi automaton combined with the FTS from figure 2.1 is shown in figure 4.2

Note: in figure 4.2 all nodes have a self loop, which are not included for the sake of the reader.

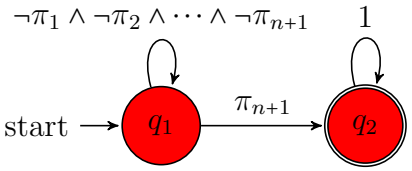

Figure 4.1: Büchi automaton corresponding to $\neg(\pi_1 \vee \pi_2 \vee \cdots \vee \pi_n)\boldsymbol{\mathcal{U}}\pi_{n+1}$

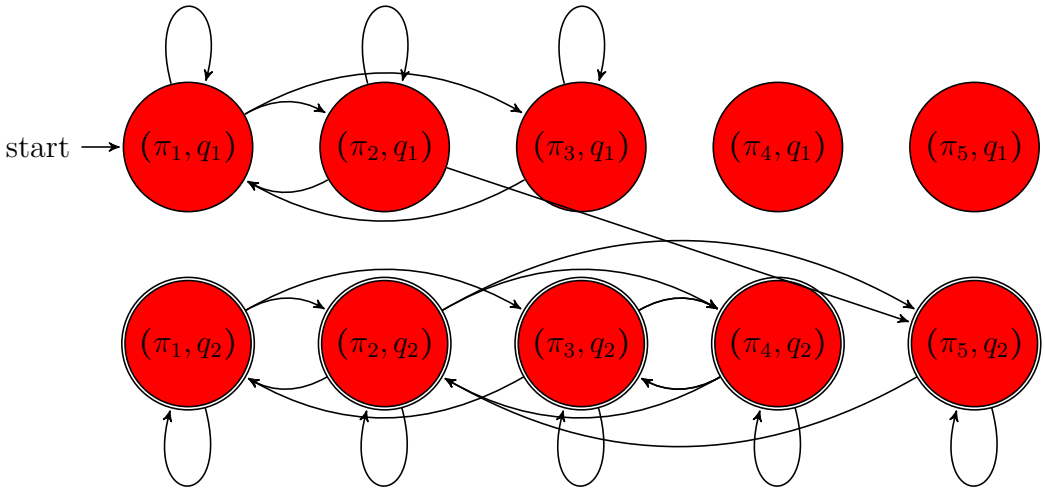

Figure 4.2: Product Automaton for $\neg\pi_4 \mathcal{U} \pi_5$ with Simple FTS

To find an accepting path, the accepted algorithm starts at the initial node, (π_1, q_1), and does a first Dijkstra search to find the optimal path to all the accepting states, (π_i, q_2), $\forall i = 1, 2, \ldots, 5$. Then the optimal path back from each of these accepting nodes is computed. The greedy algorithm does $n + 1$ Dijkstra searches, where n is the maximum level of the initial state in the Büchi automaton. As we can see in figure 4.1, n is 1 for all formulas of this form. Therefore our algorithm does one Dijkstra search starting from (π_1, q_1) which ends at (π_5, q_2). The greedy algorithm has a slightly faster runtime because it does not have to do a Dijkstra search for every accepting node; the greedy algorithm does only one.

Because node q_2 in automaton 4.2 has a self loop and every region in the simple FTS has a self loop, every accepting state in the product automaton will have a self loop. This implies that the accepting node that creates the optimal prefix-suffix plan, i.e. q'_{f*} in Procedure 1, is the accepting node closest to the initial node. This is the accepting node that the greedy algorithm finds, which in turn implies that both algorithms calculate the same plan.

We now define a workspace to use with our problem. Our workspace is a grid, 25 units across and 25 units up, a total of 625 equally sized squares. We say our robot can can move horizontally and vertically, however it cannot move diagonally. Additionally we say that the unit cost of going from any adjacent to another region is 1. The initial position is located at $(0, 0)$, region π_1 is located at (2,24), region π_2 is located at (12,12), and region π_3 is located

at (20,15) as seen in figure 4.1.

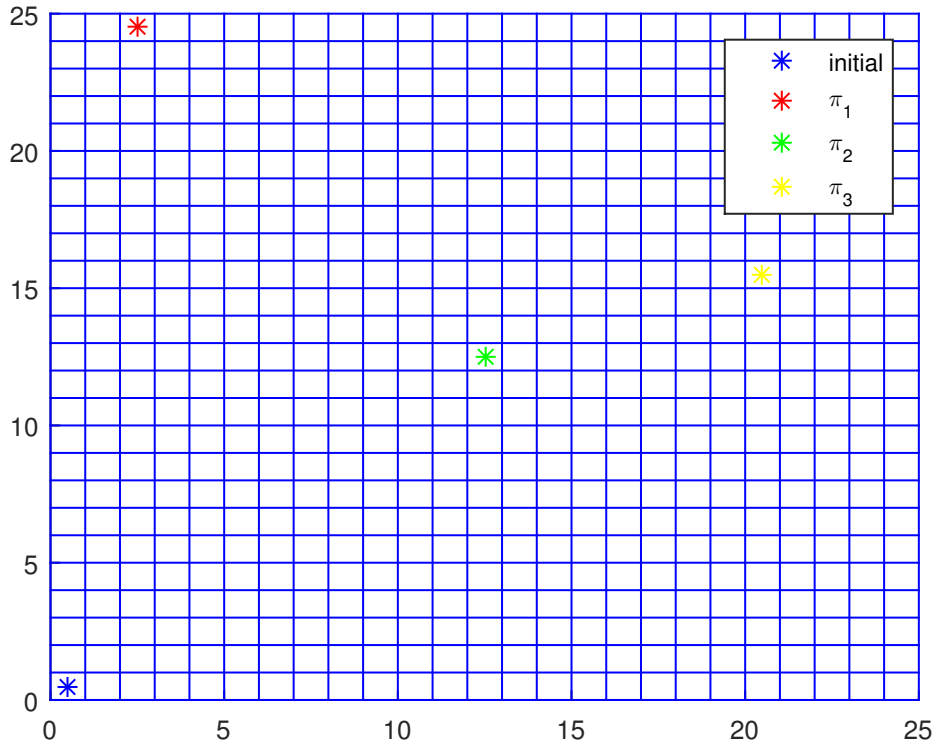

Figure 4.3: Workspace 1

The output from the accepted algorithm is

```
Accepted Algorithm
==========

Dijkstra_plan_networkX done within 0.02s: precost 37.00, sufcost 0.00
------------------------------
the prefix of plan **states**:
[((0, 0, 1), 'None'), ((1, 0, 1), 'None'), ((2, 0, 1), 'None'), ((3,
    0, 1), 'None'), ((3, 1, 1), 'None'), ((4, 1, 1), 'None'), ((4, 2,
    1), 'None'), ((5, 2, 1), 'None'), ((6, 2, 1), 'None'), ((6, 3, 1),
     'None'), ((6, 4, 1), 'None'), ((7, 4, 1), 'None'), ((7, 5, 1), '
    None'), ((8, 5, 1), 'None'), ((9, 5, 1), 'None'), ((9, 6, 1), '
    None'), ((9, 7, 1), 'None'), ((9, 8, 1), 'None'), ((10, 8, 1), '
    None'), ((10, 9, 1), 'None'), ((10, 10, 1), 'None'), ((11, 10, 1),
     'None'), ((12, 10, 1), 'None'), ((13, 10, 1), 'None'), ((14, 10,
    1), 'None'), ((14, 11, 1), 'None'), ((15, 11, 1), 'None'), ((16,
```

```
    11, 1), 'None'), ((17, 11, 1), 'None'), ((18, 11, 1), 'None'),
    ((19, 11, 1), 'None'), ((19, 12, 1), 'None'), ((20, 12, 1), 'None
    '), ((20, 13, 1), 'None'), ((20, 14, 1), 'None'), ((20, 15, 1), '
    None'), ((20, 16, 1), 'None')]
the suffix of plan **states**:
[((20, 17, 1), 'None'), ((20, 17, 1), 'None')]
------------------------------
the prefix of plan **actions**:
[(0, 0, 1), (1, 0, 1), (2, 0, 1), (3, 0, 1), (3, 1, 1), (4, 1, 1), (4,
     2, 1), (5, 2, 1), (6, 2, 1), (6, 3, 1), (6, 4, 1), (7, 4, 1), (7,
     5, 1), (8, 5, 1), (9, 5, 1), (9, 6, 1), (9, 7, 1), (9, 8, 1),
    (10, 8, 1), (10, 9, 1), (10, 10, 1), (11, 10, 1), (12, 10, 1),
    (13, 10, 1), (14, 10, 1), (14, 11, 1), (15, 11, 1), (16, 11, 1),
    (17, 11, 1), (18, 11, 1), (19, 11, 1), (19, 12, 1), (20, 12, 1),
    (20, 13, 1), (20, 14, 1), (20, 15, 1), (20, 16, 1), (20, 17, 1)]
the suffix of plan **actions**:
['None', 'None']
full construction and synthesis done within 0.11s
```

At the top of the output the time taken to calculate the path, the cost of the prefix and the cost of the suffix is given. The states can be thought of as the result of the labelling function, and actions can be thought of the labels of the transition of the Büchi automaton. The `full construction and synthesis` time is larger than the first time given because it includes the time taken to initialize and construct the graph. This almost always takes the majority of time. For the rest of this report, the calculated paths will appear in the appendix.

The output of our algorithm is as follows

```
Our Algorithm
=============
new_algorithm_plan done within 0.01s: precost 37.00, sufcost 0.00
...
full construction and synthesis done within 0.32s
```

As we can see, our algorithm computed the same path in about the same time.

4.2 Sequencing

This behaviour is ideal for our algorithm. Sequencing is the behaviour of visiting regions $\pi_1, \pi_2, \ldots, \pi_n$ in that order. A formula in this subset of formulas is $\diamond(\pi_3 \wedge \diamond\pi_5)$ and is shown in figure 4.4. We note that this automaton is only applicable because of the partition we defined earlier in Definition 1 which makes it impossible for π_i and π_j to be true at the same time if $i \neq j$. The LTL2BA tool [1] that is used the github repository [14] actually generates

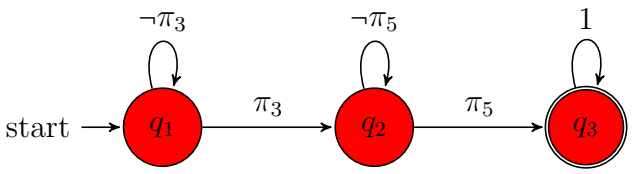

Figure 4.4: Büchi Automaton Corresponding to $\diamond(\pi_3 \wedge \diamond\pi_5)$

an automaton with connecting every node. For example, there is an edge from q_1 to q_3 labelled $\pi_3\&\&\pi_5$. This transition is impossible so we take it out before calculating the distances. Our algorithm would not work very well if every state in the Büchi automaton had a distance of 1 and the accepting states had a distance of 0.

We show why sequencing formulas are ideal for our algorithm in an example using the formula $\diamond(\pi_3 \wedge \diamond\pi_5)$ the simple FTS as before. The product automaton is shown in figure 4.5 In both algorithms the search starts at (π_1, q_1). In the first step our algorithm looks at the nodes connected to (π_1, q_1) i.e. (π_2, q_1) and (π_3, q_2). Our algorithm notices that we are on level 2, and (π_3, q_2) is on level 1. Then it finishes this Dijkstra search and starts another Dijkstra search beginning at (π_3, q_2). It will then do one step and search (π_1, q_2) and (π_4, q_2). Then it will do another step and search (π_2, q_2). Then in the third step it searches (π_5, q_3). It notices (π_5, q_3) is an accepting state and finishes the search with the plan being R_{new} = (π_1, q_1), (π_3, q_2), (π_4, q_2),(π_2, q_2), (π_5, q_3) (It may turn out that the plan returned is actually R_{new} = (π_1, q_1), (π_3, q_2), (π_4, q_2),(π_2, q_2), (π_5, q_3), but these paths have an equivalent cost and are found out the same time).

The accepted algorithm will also start at (π_1, q_1) and search (π_2, q_1) and (π_3, q_2) on the first step. In the next step it will search (π_4, q_1), (π_5, q_1), (π_1, q_2),(π_4, q_2). Then it searches (π_2, q_2) and then in the next step (π_5, q_3). It continues because it has to find the shortest path to *all* accepting nodes. So the next step it searches (π_2, q_3), then (π_1, q_3) and (π_4, q_3) and finally (π_3, q_3).

The advantage that our algorithm has over the accepted algorithm is that the accepted algorithm will search through more extraneous nodes, for example (π_5, q_1), and that it does not have to find the shortest path to all the accepting nodes.

We run both algorithms with the formula of $\diamond(\pi_1 \wedge \diamond(\pi_2 \wedge \diamond\pi_3))$. The output from the accepted algorithm is

```
Accepted Algorithm
==========

Dijkstra_plan_networkX done within 0.04s: precost 62.00, sufcost 0.00
...
```

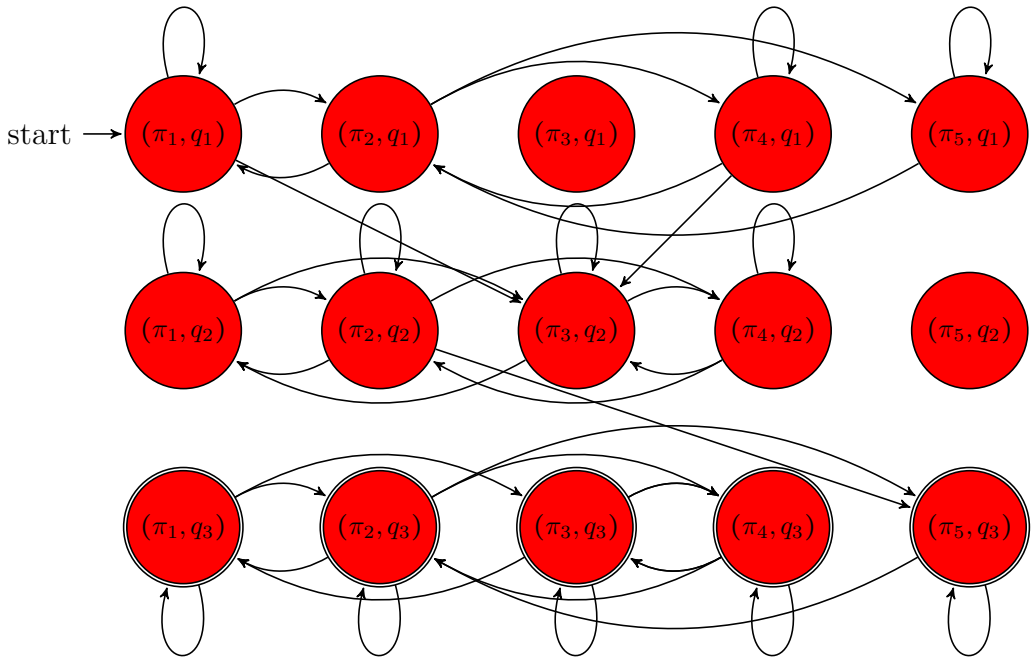

Figure 4.5: Product Automaton for $\diamond(\pi_4 \wedge \diamond\pi_5)$ with Simple FTS

```
full construction and synthesis done within 0.19s
```

Our algorithm computed the same path, with an output of

```
Our Algorithm
=====================
new_algorithm_plan done within 0.02s: precost 62.00,
    sufcost 0.00
...
full construction and synthesis done within 0.17s
```

as we can see, the plan synthesis part took half as long; 0.02 seconds compared to 0.04 seconds. We take a look at what causes the increased time.

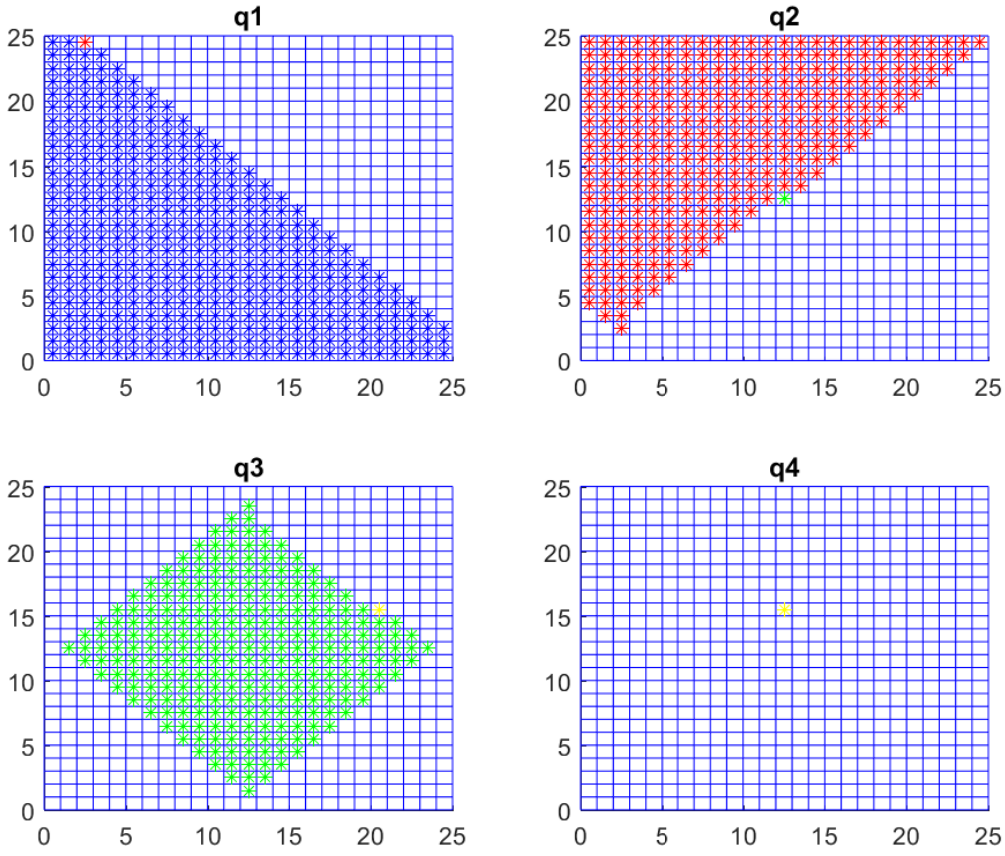

Figure 4.6: Nodes searched with our Algorithm

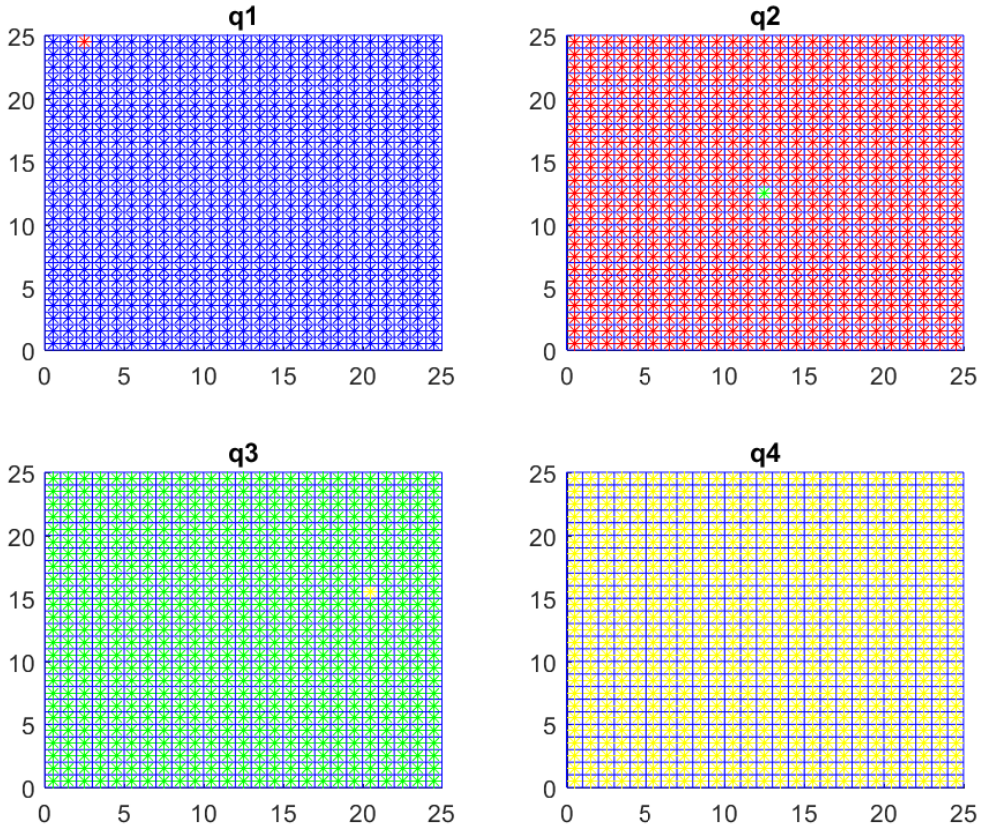

Figure 4.7: Nodes searched with the accepted Algorithm

Figure 4.2 shows the states searched with our algorithm and figure 4.2 shows the states visited with the accepted algorithm. Each figure shows a representation of the product automaton. The graph can be thought of as the discretization of the workspace, and there are four corresponding to the four states in the Büchi automaton. Any square filled in with blue, red green or yellow has been visited, all others have not. As we can see, our algorithm searches a significantly smaller number of nodes, while the accepted algorithm searches every node.

The FTS has 625 and the Büchi automaton has four states, which implies the product automaton has 2500 states. The accepted algorithm computes the shortest paths from the initial node to each of the 2499 other nodes, and then for each of the 625 accepting nodes computes the shortest path back to

itself. The algorithm then chooses which combination out of the 625 choices makes the shortest overall run.

The level of the initial node is three, so our algorithm does three Dijkstra searches; one for each level and one for the accepted node back to itself. To find an accepting node, the the first search searches through 326 nodes, the second 266 nodes, and the third 587 nodes. The last search simply finds the path from the accepting node back to itself, which is a self loop. Thus our algorithm searches 1179 nodes compared to 2500 nodes.

4.3 Coverage

A coverage formula represents the statement visit $\pi_1, \pi_2, \ldots, \pi_n$ in any order, and is of the form $\varphi = \diamond\pi_1 \wedge \diamond\pi_2 \wedge \cdots \wedge \diamond\pi_n$. We show the Büchi automaton corresponding to the formula $\diamond\pi_1 \wedge \diamond\pi_2 \wedge \diamond\pi_3$ in figure 4.8

So, we can see that to get to the accepting node, we have to choose which node to go to first, and which node to go to second (the third node we then have to visit is already decided). So, there are 6 possible paths to take from the initial node, q_1 to accepting state q_8. This is true in the product automaton too, if we only consider the option of taking the optimal path between nodes. The order that our algorithm will pick is it will pick first pick π_i which is the closest to it. From then, it will pick the next closest π_j out of the two that have not been visited yet.

When we use our algorithm on a coverage formula, we may not get the optimal path. We will however get an accepting path, with a bound on the cost of our path in terms of the cost of the optimal path.

We first show that this path corresponds to the one generated by the nearest neighbour approach to the travelling salesperson problem. Next we provide the bound on the cost of our path based on the worst case ratio of the nearest neighbour tour to the optimal tour given by Rosendrantz, Stearns, and Lewis [18].

4.3.1 Travelling Salesperson Problem

The travelling salesperson problem is stated in layman's terms as finding the shortest path for a salesperson to take such that he passes through a given set of cities and then returns back home at the end. More formally, it can be stated as finding the minimum Hamiltonian circuit with the lowest sum of distances between the nodes (cities). This problem has been studied extensively and "give quote about importance". This problem is NP-hard, and thus many algorithms and heuristics exist for finding an approximate

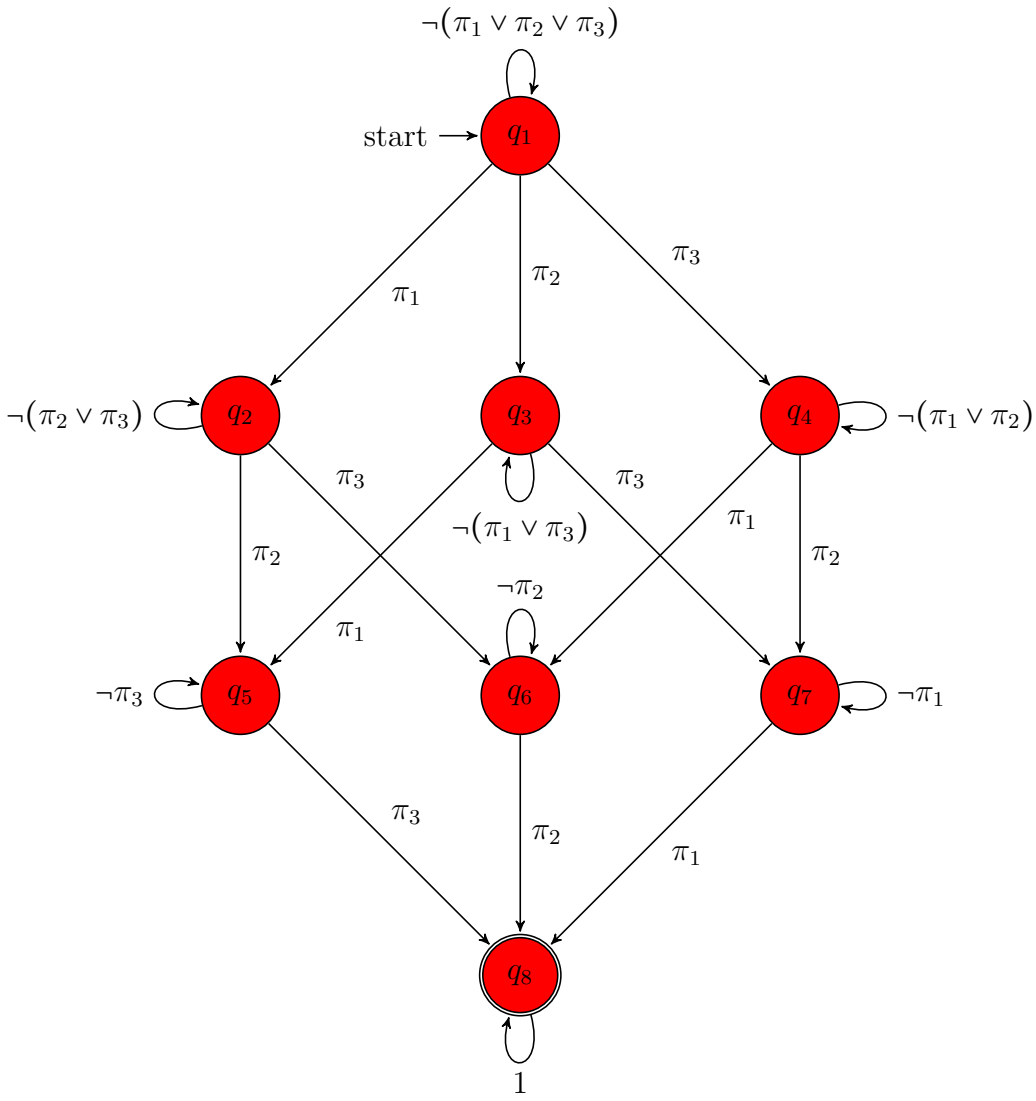

Figure 4.8: Büchi Automaton Corresponding to $\diamond\pi_1 \wedge \diamond\pi_2 \wedge \diamond\pi_3$

solution. One very simple algorithm to do this is called the nearest neighbour algorithm. It says from the starting city, pick the closest city to be the next stop. From there, pick the next closest city not including the starting city, and so on. If there is a tie in the next closest neighbour, we assume that the next node can be decided arbitrarily. This is exactly what our algorithm does in this situation, the first Dijkstra search finds the closest node, and the we start another search.

To formulate our problem as a travelling salesman problem we use the idea of a dummy node from Lenstra and Rinnooy Kan's computer wiring example in [17]. In their example, they are designing a computer interface at the Institute for Nuclear Physical Research in Amsterdam. An interface is made up of several modules, with multiple pins on each module. A given subset of pins has to be interconnnected by wires, and at most two wires can be connected to any pin. For obvious regions, it is desirable to minimize the amount of wire used. They show that this is actually a travelling salesperson problem in disguise. The only difference between this problem and a travelling salesman problem is that in the travelling salesman problem, the salesman must return home at the end. This is not true in this problem. It is also not true in our problem, we only need to pass through π_1, π_2 and π_3, there is no need to return to the starting state after we do this. To formulate this seemingly unrelated problem into a travelling salesperson problem, they set P to be the set of pins to be interconnected, c_{ij} to be the distance between pin i and pin j. They then introduce a dummy node $*$ that is a distance 0 from all the other nodes i.e. $c_{i*} = c_{*i} = 0$ for all i. Then the corresponding problem is solving the travelling salesperson problem on the set of nodes $N = P \cup \{*\}$.

For our problem, we set $c_{ij} = d(\pi_i, \pi_j)$, for $i, j = 0, 1, 2, 3$ where the initial state is from now on known as π_0, to be the shortest path our robot can take from π_i to π_j, insuring that the triangle inequality is satisfied for all i and j. We must preserve the the triangle inequality for a proof of a worst case scenario bound we will provide later on. We use this same idea as above of adding a dummy node, however to preserve the triangle inequality we cannot have the dummy node be distance 0 from the other nodes. Indeed, if $c_{i*} = c_{*i} = 0$ the triangle inequality would be violated because $c_{i*} + c_{*j} = 0 \geq c_{ij}$ which would make the cost from getting to any point 0, thus rendering the problem extremely trivial.

We can represent the relationship between the regions in our graph with the following *complete* subgraph, shown in figure 4.9. A complete graph is an undirected graph in which every pair of vertices is connected by an edge.

For the distances, we use the so called *Manhattan distance*,

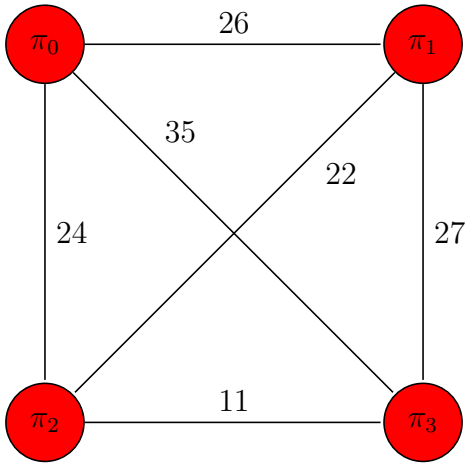

Figure 4.9: Complete Graph between Regions of Interest

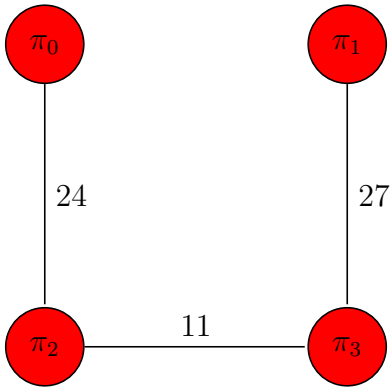

Figure 4.10: Nearest Neighbour Path

i.e. $d((x_1, y_1), (x_2, y_2)) = |x_1 - x_2| + |y_1 - y_2|$ because our robot can only move horizontally and vertically, not diagonally. Given the weights between the vertices, we easily see that the path that our algorithm, and the nearest neighbour, will take is shown in figure 4.10. The cost of this path is 62.

This is not the optimal path though, which is shown in figure 4.11 and has a cost of 59.

Because we have to make sure that the dummy node does not change the order that our algorithm and the nearest neighbour algorithm takes we have to set the distance the dummy node is away from every other node to be $max_{i,j}c_{ij}$ where c_{ij} is the distance between the nodes in the complete subgraph in figure 4.9. In our case, this is 35, the path between π_0 and π_3. This insures that the path taken is the same as the accepted neighbour because the dummy node will be the last node to be visited. This is because

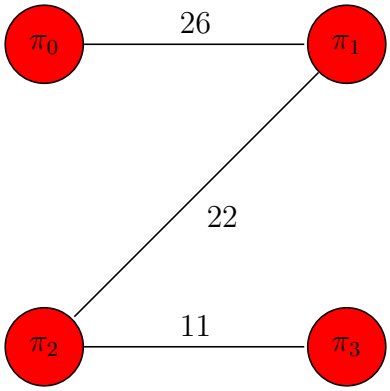

Figure 4.11: Optimal Path

in the nearest neighbour algorithm, ties are broken arbitrarily. Thus, the only time where it is a possibility that the nearest neighbour algorithm goes to the dummy node i.e. when the next nodes are $\max_{i,j} c_{ij}$ from the current node is when and if we are faced with the only choice being take the maximum path $\max_{i,j} c_{i,j}$ to π_j or to go to the dummy node, and we can choose to go to π_j because the ties can be broken arbitrarily. In any other case, the nearest neighbour path will choose a to go to a node where the cost is $c_{i',j'} < c_{i,j}$.

We show the new subgraph in figure 4.12

The path that the nearest neighbour algorithm takes in this situation, the complete Hamiltonian circuit, is given in figure 4.13, which gives a total cost of 132.

We note however that this is not the optimal solution. This optimal solution is shown in figure 4.14 and has a cost of 129.

4.3.2 Cost Bound

It has been shown [18] that for an n-node travelling salesperson problem which satisfies the triangle inequality i.e. $d(i,j) = d(j,k) \geq d(i,k)$ for all i, j, and k where $d(i,j)$ is the nonnegative distance between nodes i and j,

$$\text{NEARNEIBR} \leq (\frac{1}{2}\lceil \log(n) \rceil + \frac{1}{2})\text{OPTIMAL}$$

where NEARNEIBR is the cost of the path generated by the nearest neighbour algorithm and OPTIMAL is the cost of the optimal path.

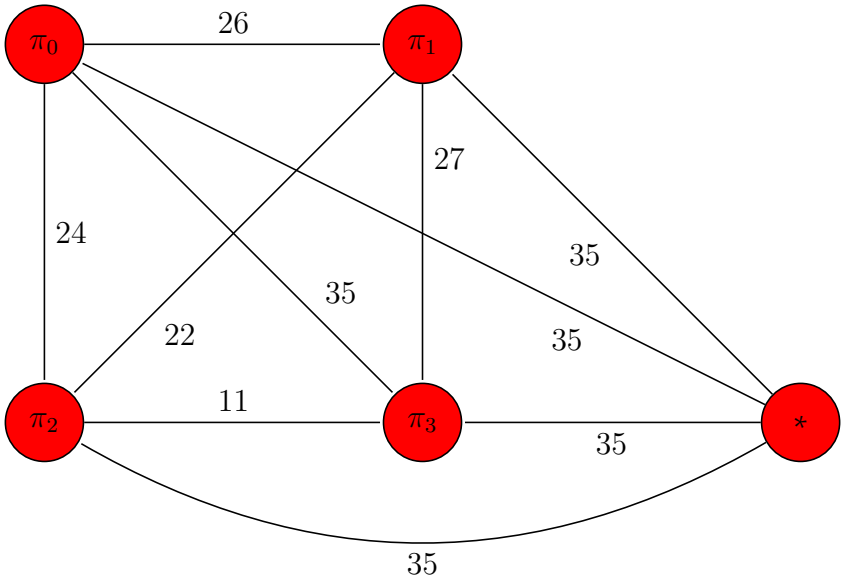

Figure 4.12: Complete Subgraph with Dummy Node

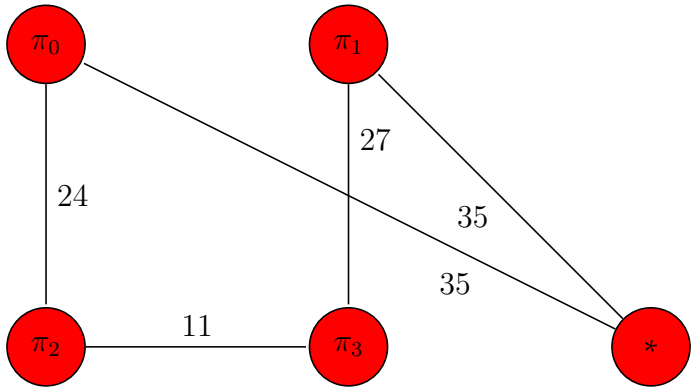

Figure 4.13: Nearest Neighbour Path with Dummy Node

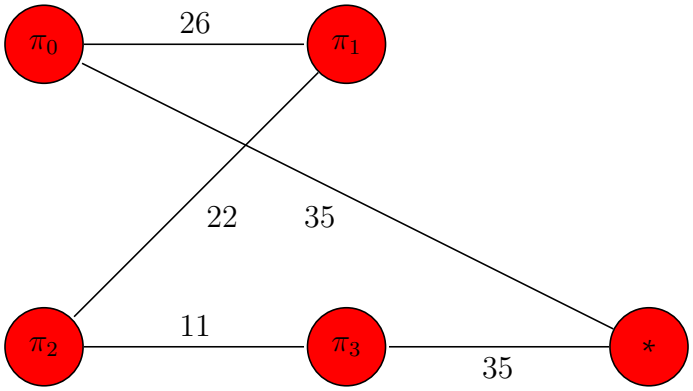

Figure 4.14: Optimal Path with Dummy Node

Our values do indeed satisfy this inequality

$$\begin{aligned} \text{NEARNEIBR} &\leq (\frac{1}{2}\lceil \log(n) \rceil + \frac{1}{2})\text{OPTIMAL} \\ 132 &\leq (\frac{1}{2}\lceil \log(5) \rceil + \frac{1}{2})129 \\ 132 &\leq 258 \end{aligned}$$

We also see that it is very conservative worst case bound and we will likely do much better.

We provide a proof of

$$\frac{\text{NEARNEIBR}}{\text{OPTIMAL}} \leq \frac{1}{2}\lceil \log(n) \rceil + \frac{1}{2} \tag{4.1}$$

which can be found in [18]. Proof: We begin by proving

$$\text{OPTIMAL} \geq 2 \sum_{i=k+1}^{\min(2k,n)} l_i \tag{4.2}$$

for all k, $0 \leq k \leq n$. Let l_i be the length of the i^{th} largest edge in the tour obtained by the nearest neighbour algorithm. For each i, $0 \leq i \leq n$, let a_i be the node *onto which* the i^{th} largest edge is added to (that would be the edge with length l_i). Let H be the complete subgraph defined on the set of nodes $\{a_i | 1 \leq i \leq min(2k, n)\}$.

Now, let T be the tour in H which visits the nodes of H in the same order as these nodes are visited in an optimal tour of the original graph. Let LENGTH be the length of T. We have

$$\text{OPTIMAL} \geq \text{LENGTH} \tag{4.3}$$

This is because the tour with cost OPTIMAL passes through all the nodes that the tour with cost LENGTH passes through, and more. Thus if H has an edge (b, c), then the OPTIMAL tour will either have the edge (b, c) or take a less direct route through some of its extra nodes. So the triangle inequality implies (4.3).

Let (a_i, a_j) be an edge of T. If the nearest neighbour method adds point a_i before a_j, we have $d(a_i, a_j) \geq l_i$, where $d(a_i, a_j)$ is the distance between nodes a_i and a_j. We also see that if a_j is added first we have $d(a_i, a_j) \geq l_j$. This is because, say we added a_i first, we know there is a point l_i away from a_i that the nearest neighbour method makes the path to. This can be a_j, because we know a_j has not been added yet or another node. If it is another node $d(a_i, a_j) \geq l_i$ because the nearest neighbour finds the closest node that has not yet been visited, or $d(a_i, a_j) = l_i$ if a_j is added next.

Since one has to be added before the other, we have

$$d(a_i, a_j) \geq \min(l_i, l_j) \tag{4.4}$$

Summing (4.4) over the edges of T, we get

$$\text{LENGTH} \geq \sum_{(a_i, a_j) \text{ in } T} \min(l_i, l_j) \tag{4.5}$$

If we let α_i be the number of edges (a_i, a_j) in T for which l_i is selected as $\min(l_i, l_j)$ we obtain

$$\sum_{(a_i, a_j) \text{ in } T} \min(l_i, l_j) = \sum_{a_i \text{ in } H} a_i l_i \tag{4.6}$$

Because a_i is the endpoint of two edges in T, $\alpha \leq 2$.

Because T has $\min(2k, n)$ edges (one for each node),

$$\sum_{a_i \text{ in } H} \alpha_i = \min(2k, n) \tag{4.7}$$

To get a lower bound on (4.6) we assume that $\alpha_i = 2$ for $k + 1 \leq i \leq \min(2k, n)$ and is zero of $i \leq k$. Thus,

$$\sum_{a_i \text{ in } H} a_i l_i \geq 2 \sum_{i=k+1}^{\min(2k,n)} l_i \tag{4.8}$$

Combining (4.3), (4.5), (4.6), and (4.8), we get

$$\text{OPTIMAL} \geq \text{LENGTH} \geq \sum_{(a_i,a_j) \text{ in } T} \min(l_i, l_j) = \sum_{a_i \text{ in } H} a_i l_i \geq 2 \sum_{i=k+1}^{\min(2k,n)} l_i$$

thus proving (4.2).

We now sum (4.2) for all values of k for all values of k equal to powers of two less than or equal to n i.e. $k = 2^j \leq n$ for $j = 0, 1, \ldots \lceil \log(n) \rceil - 1$. We then get

$$\sum_{j=0}^{\lceil \log(n) \rceil - 1} \text{OPTIMAL} \geq \sum_{j=0}^{\lceil \log(n) \rceil - 1} (2 \cdot \sum_{i=2^j+1}^{\min(2^{j+1},n)} l_i)$$

We have

$$\begin{aligned}\sum_{j=0}^{\lceil \log(n) \rceil - 1} \text{OPTIMAL} &\geq 2 \cdot \sum_{i=2}^{2} l_i + 2 \cdot \sum_{i=3}^{4} l_i + 2 \cdot \sum_{i=5}^{8} + \sum_{j=3}^{\lceil log(n) \rceil - 1} (2 \cdot \sum_{i=2^j+1}^{\min(2^{j+1},n)} l_i) \\ &\geq 2l_2 + 2l_3 + 2l_4 \cdots + 2l_8 + \sum_{j=3}^{\lceil log(n) \rceil - 1} (2 \cdot \sum_{i=2^j+1}^{\min(2^{j+1},n)} l_i)\end{aligned}$$

Therefore we can write

$$\lceil \log(n) \rceil \cdot \text{OPTIMAL} \geq 2 \sum_{i=2}^{n} l_i \tag{4.9}$$

Now OPTIMAL must be longer than twice any edge in the graph because it contains two paths between any given pair of points and these paths are , by the triangle inequality, longer than the distance of the edge connecting the points directly, i.e. $\text{OPTIMAL} \geq 2l_i$ for $i = 1, 2, \ldots, n$. Specifically,

$$\text{OPTIMAL} \geq 2ł_1 \tag{4.10}$$

Summing (4.9) and (4.10) we get

$$(\log(n) + 1) \cdot \text{OPTIMAL} \geq 2 \sum_{i=1}^{n} l_i$$

By definition, $\sum_{i=1}^{n} l_i = \text{NEARNEIBR}$, thus we have

$$\text{NEARNEIBR} \leq (\frac{1}{2} \lceil \log(n) \rceil + \frac{1}{2}) \text{OPTIMAL}$$

□

We have thus shown that when formulating and solving our problem as a travelling salesman problem with a dummy node, we get the same solution as the nearest neighbour search algorithm. This search algorithm then has a bound on the ratio of the resulting path to the optimal path i.e.

$$\frac{\text{NEARNEIBR}}{\text{OPTIMAL}} \leq (\frac{1}{2}\lceil \log(n) \rceil + \frac{1}{2})$$

We now must remove the dummy node and provide a bound for the true cost that we will get from our search.

NEARNEIBR and OPTIMAL as above are costs of Hamaltonian circuits. By definition every node in a Hamaltonian circuit is passed through exactly once. Therefore the dummy node will be passed through exactly once, and we have shown that it will be the last node passed through in the NEARNEIBR. In the NEARNEIBR path, because the dummy node is length $\max_{i,j} c_{i,j}$ it will never be the closest next node, unless we are given the choice to go from π_i to π_j for i and j being the maximum edge cost in the complete subgraph. In this case we can break the tie arbitrarily and choose to go to π_j instead of the dummy node. Thus the path found by the nearest neighbour search will be the path found by our our algorithm, and then going to the dummy node for a cost of $\max_{i,j} c_{i,j}$, then from there going to the initial node to for a cost of $\max_{i,j} c_{i,j}$. Therefore the cost of our algorithm, denote ALGOR is

$$\text{ALGOR} = \text{NEARNEIBR - } 2\max_{i,j} c_{i,j}$$

The path OPTIMAL, however is not guaranteed to have the dummy node be the last node visited. The cost of the path which is optimal and requires that the dummy node is the last node visited, is then greater than or equal to OPTIMAL. This is because of the freedom taken away by requiring the dummy node to be visited last, and less freedom in a minimization problem results in a larger value. Let ACCEPT be the cost of the accepted algorithm for path planning. ACCEPT $+2\max_{i,j} c_{i,j}$ is then equal to the cost of the optimal travelling salesman solution which requires that the dummy node is the last node visited. This is because we have already established that the accepted algorithm will find the optimal path. Therefore we have

$$\text{ACCEPT} + 2\max_{i,j} c_{i,j} \geq \text{OPTIMAL}$$

Plugging into the travelling salesman bound, we get

$$\begin{aligned}
\text{NEARNEIBR} &\leq (\frac{1}{2}\lceil\log(n)\rceil + \frac{1}{2})\text{OPTIMAL} \\
\text{ALG} + 2\max_{i,j} c_{i,j} &\leq (\frac{1}{2}\lceil\log(n)\rceil + \frac{1}{2})\text{OPTIMAL} \\
\text{ALG} + 2\max_{i,j} c_{i,j} &\leq (\frac{1}{2}\lceil\log(n)\rceil + \frac{1}{2})(\text{ACCEPT} + 2\max_{i,j} c_{i,j})
\end{aligned}$$

We can check with our previously calculated values for ALG and ACCEPT

$$\begin{aligned}
\text{ALG} + 2\max_{i,j} c_{i,j} &\leq (\frac{1}{2}\lceil\log(n)\rceil + \frac{1}{2})(\text{ACCEPT} + 2\max_{i,j} c_{i,j}) \\
62 + 2(35) &\leq (\frac{3}{2} + \frac{1}{2})(59 + 70) \\
132 &\leq 258
\end{aligned}$$

We can see that this is still a conservative bound, and emphasize that it is the worst case. Usually the algorithm will preform much better.

4.3.3 Simulation

The actual output from the accepted algorithm is

```
Accepted Algorithm
```

```
Dijkstra_plan_networkX done within 0.08s: precost 59.00, sufcost 0.00
...
full construction and synthesis done within 0.43s
```

and our algorithm is

```
Our Algorithm
```

```
new_algorithm_plan done within 0.02s: precost 62.00, sufcost 0.00
...
full construction and synthesis done within 0.38s
```

As we can see our algorithm calculates the path in 0.02 seconds while the accepted algorithm takes 0.08 seconds. We can break down the searches as we did before.

The accepted algorithm does one Dijkstra search of all 5000 states in the product automaton (625 states in the FTS and eight in the Büchi automaton). Even though there are eight states in the Büchi automaton, the initial node is still only on level three. Therefore we only do three searches to find

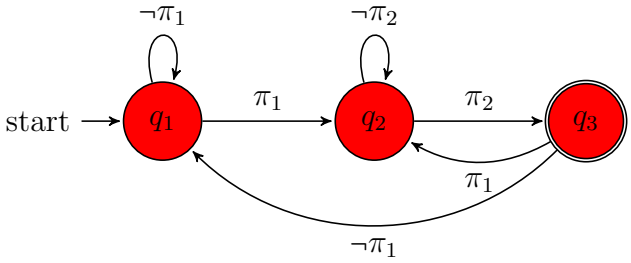

Figure 4.15: Büchi Automaton for $\square(\diamond\pi_1 \wedge \diamond\pi_2)$ 1

an accepting node. The first searches 326, the second 266, and the third 587. Thus our algorithm searches 1179 nodes compared to 5000 by the accepted algorithm and returns a path of cost 62 compared 59.

4.4 Recurrence (Liveness)

Recurrence is coverage over and over again, and can be expressed as $\square(\diamond\pi_1 \wedge \diamond\pi_2 \wedge \cdots \wedge \diamond\pi_n)$. This example is interesting for two reasons: it is prone to Büchi automata that are not tight [20], and an accepting path for it cannot stay in one state (in contrast to the other formulas, in which all accepting states have self loops). We first look at the tightness.

4.4.1 Tightness

A tight Büchi automaton is one that accepts the minimum prefix and the minimum suffix. We begin showing that the automaton produced for these formulas by [1] is not tight. We consider the formula $\square(\diamond\pi_1 \wedge \diamond\pi_2 \wedge \diamond\pi_3)$. The Büchi automaton corresponding to this formula, as calculated by [11] is given in figure 4.15

Note: Again, the actual automaton generated has much more edges. In this automation, $d(q_1) = 2$, $d(q_2) = 1$, and $d(q_3) = 0$. So, to get from $q'_{init} = \langle \pi_2, q_1 \rangle \in Q'_0$, we have to first get down to level 2. Given the Büchi automaton 4.15 the only way to do this is to go to region π_1. In this case the same statement holds for π_2. It follows that the prefix with the least cost is a concatenation of the shortest paths down from each each level (first to π_1, etc).

This path however is in general not truly optimal. It is because the Büchi automaton given in figure 4.15 not a tight Büchi automaton [20]. A Büchi automaton is tight if it accepts the shortest lasso (prefix and suffix). The loss of this optimality property is due to the fact that the algorithm in [11]

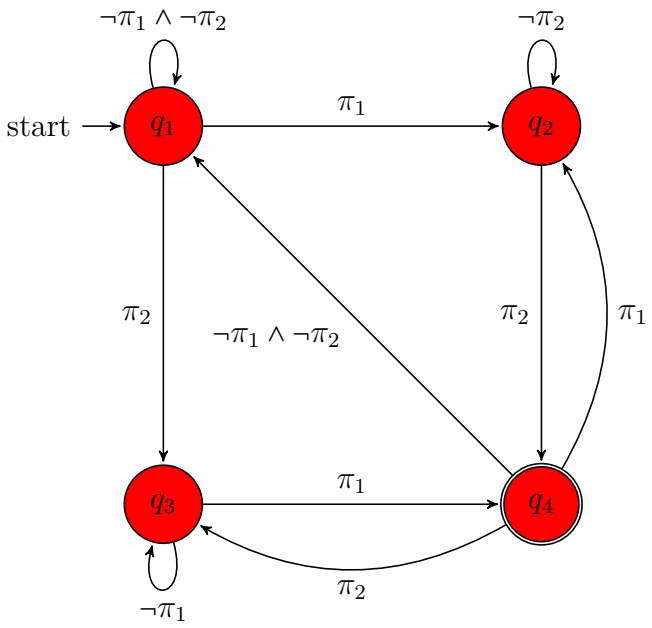

Figure 4.16: Büchi Automaton for $\square(\diamond\pi_1 \wedge \diamond\pi_2)$ 2

simplifies the Büchi automaton which is usually a good thing because it leads to a lower computational complexity in most applications. We take a look at a different automaton corresponding to the formula $\square(\diamond\pi_1 \wedge \diamond\pi_2)$, shown in figure 4.16.

In this automaton, $d(q_1) = 2$, $d(q_2) = d(q_3) = 1$, and $d(q_4) = 0$. So, we are starting at the same level i.e. 2, however this time we have two choices about what to do to get down to level 2; we can go to π_1 or π_2. Being able to choose is good in the sense that we can now find the optimal path, and bad in the sense that the extra state in the *Büchi* automaton increased the size of the product automaton by 33% (hence increasing the time it takes to search the automaton). This very well illustrates the trade off between the search time and optimally/cost of the resulting run. We propose that this is a good way to think about our algorithm. There is a trade off that sometimes it will not find the optimal run, even if this is possible, though it will be faster.

4.4.2 Suffix

The second aspect of this problem that we wish to look at is fact that it does not have a trivial suffix. In the other examples we have looked at, the suffix of the calculated path (with our algorithm and the accepted algorithm) was a single state; that is, the formula could be satisfied by staying in one state

indefinitely. In this example, π_1, π_2, and π_3 must all be visited infinitely often, and thus these states must be in the suffix.

The applicability of our algorithm to find the suffix has to be considered. For the total run, R, to be accepting, $\texttt{Inf}(R) \cap \mathcal{F}$ must not be empty. We are specifically looking for runs of the form

$$R = \langle R_{pre}, R_{suf} \rangle = q_0 q_1 \dots q_f [q_f q_{f+1} \dots q_n]^\omega$$

where $q_f \in \mathcal{F}$. Thus when calculating to the suffix we must find the path back to the *same* accepting state. We cannot not just look for any accepting state as we do in the prefix calculation. Our algorithm in general only looks for an accepting state, not a specific accepting state. Thus we have to do a Dijkstra search and if there is a path back to the accepting state it will find it. There is a benefit in this though because our algorithm only has to find the shortest path back to one accepting state, not all of them.

4.4.3 Simulation

We have shown previously that given the Büchi automaton by [1], prefix with the least cost is a concatenation of the shortest paths down from each each level (first to π_1, etc). Our algorithm goes to π_1 where it can get down a level and then starts a new Dijkstra search. Our algorithm does a Dijkstra search at each level so it will return this path as the prefix. The accepted algorithm will also return this prefix.

The algorithms produced the same path. The accepted algorithm did this in

```
Accepted Algorithm
==========

Dijkstra_plan_networkX done within 16.17s: precost 62.00, sufcost
    60.00
------------------------------
...
full construction and synthesis done within 16.35s
```

while our algorithm did it in

```
Our Algorithm
==========

new_algorithm_plan done within 0.04s: precost 62.00, sufcost 60.00
------------------------------
...
full construction and synthesis done within 0.21s
```

As we can see, this is the greatest difference in times out of all the examples so far. But why? This is the first example when the suffix is not trivial.

In our algorithm, again we search 326+266+587 = 1179 nodes to find the first accepting node. We then do a Dijkstra search to find the shortest path back to this accepting node. This search searches 1879 nodes, resulting in a total search of 3058 nodes. The accepted algorithm does a search of 1879 to find the accepting nodes, and then a search from every accepting node back to itself. These searches are not trivial any more, so each of these searches look through either 1879 or 1880 nodes depending on the accepting node. Seeing as there are 625 accepting nodes, this results in searching 1174996 nodes. This is where the difference comes from.

Chapter 5

More Complex Formulas

The formulas in the previous section are common formulas, however they are fairly simple and only cover a small subset of the infinite amount of possible formulas that can be formed by temporal logics. The benefit of using temporal logics is that a wide variety of behaviours can be expressed, including propositions about the robot *and* about the workspace. Up to now, we have not looked at any formulas that include atomic propositions about potential tasks. We will show through examples that the same ideas presented in the previous chapter still hold true for this complex tasks, and show the speed up we get by using our algorithm compared to the accepted algorithm.

5.1 Example 1

We look at the example from [13] which says "eventually pick up the red ball. Once it is done, move to one basket and drop it. At last come back to room one and stay there". This task can be written as the LTL formula $\varphi = \diamond(\text{pickrball} \wedge \diamond\text{droprball}) \wedge \diamond \square r1$. The Büchi automaton corresponding to this formula as translated by [11] is shown in figure 5.1, with pick being short for pickrball and drop being short for droprball.

As we can see, there are many edges in this automaton and edges that have && in the label. These paths can only be taken if we satisfy both of the propositions at the same time. However, because in our example the propositions do not overlap (the ball is not in the same room as the basket, and the neither the ball or basket is located in room 1) these edge are impossible to take. Therefore we remove these edges from the automaton. We then have a much simpler automaton that is shown in figure 5.2

In this automaton, we can see that $d(q_1) = 3$, $d(q_2) = 2$, $d(q_3) = 3$, $d(q_4) =$

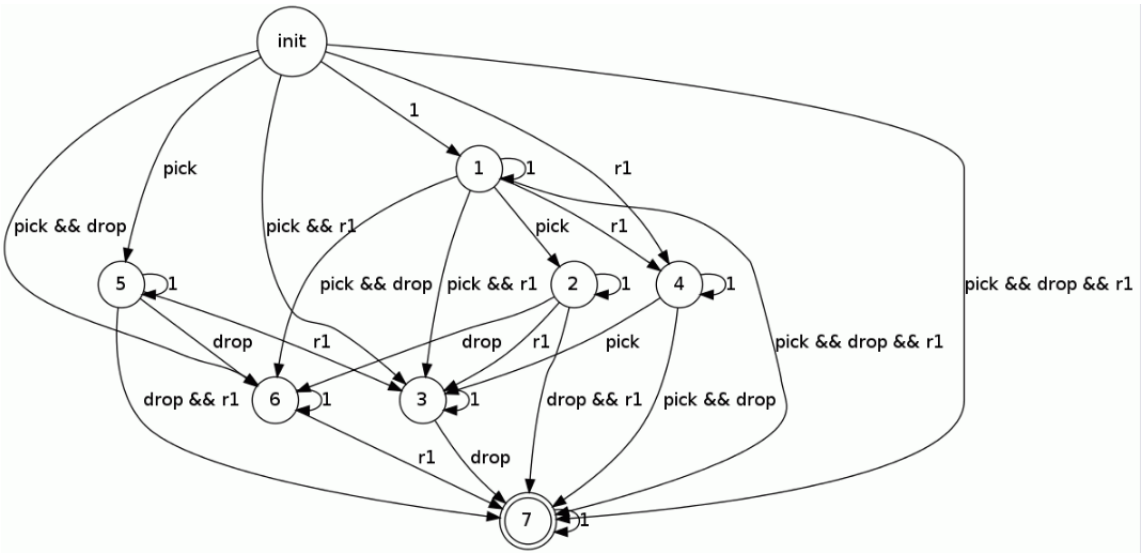

Figure 5.1: Büchi Automaton Corresponding to $\varphi = \diamond(\text{pickrball} \wedge \diamond\text{droprball}) \wedge \diamond\,\square\, r1$

1, $d(q_5) = 2$, $d(q_6) = 0$. For the first time, we have a node that connects to the initial node which is on the same level as the initial node. Examining our algorithm, we see that we will not start a new Dijkstra search until we find a node which is a level bellow our current level. Therefore we will not start a new search until we find a node in the product automaton with projection onto q_2 or q_5.

We can also see that from the illustration of the workspace, that the ball (rball) is not located next to the initial node, so the first proposition must be ¬pickrball. Examining the automaton in figure 5.2 we see we are guaranteed to take a path through nodes with projection q_3 and that we will never go to a node with the projection q_2. Therefore we are in the same situation as for sequencing i.e. there is only one sequence of actions that will satisfy the formula, implying that our algorithm will find the same path as the accepted algorithm, just faster.

We see that this is true in the output of the algorithms. Both give the same sequence of states and actions

The accepted algorithm gives

```
Accepted Algorithm
```

```
Dijkstra_plan_networkX done within 0.05s: precost 66.00, sufcost 0.00
```

while our algorithm gives

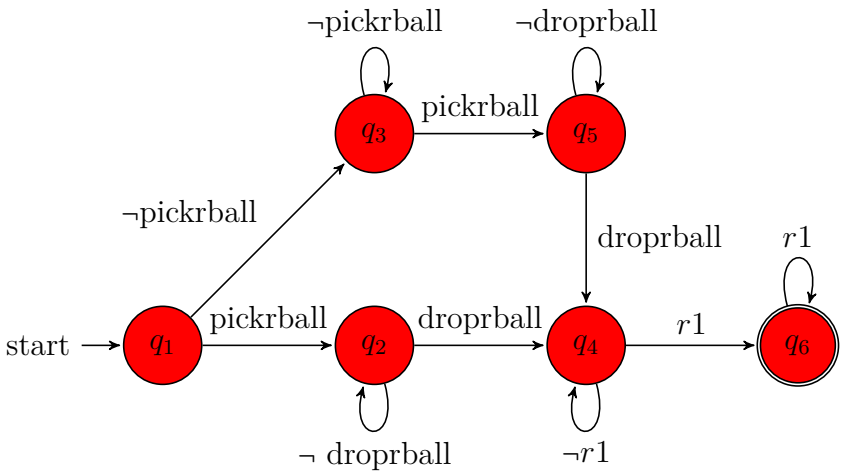

Figure 5.2: Simplified Büchi Automaton for $\varphi = \diamond(\text{pickrball} \wedge \diamond\text{droprball}) \wedge \diamond\,\square\, r1$ 1

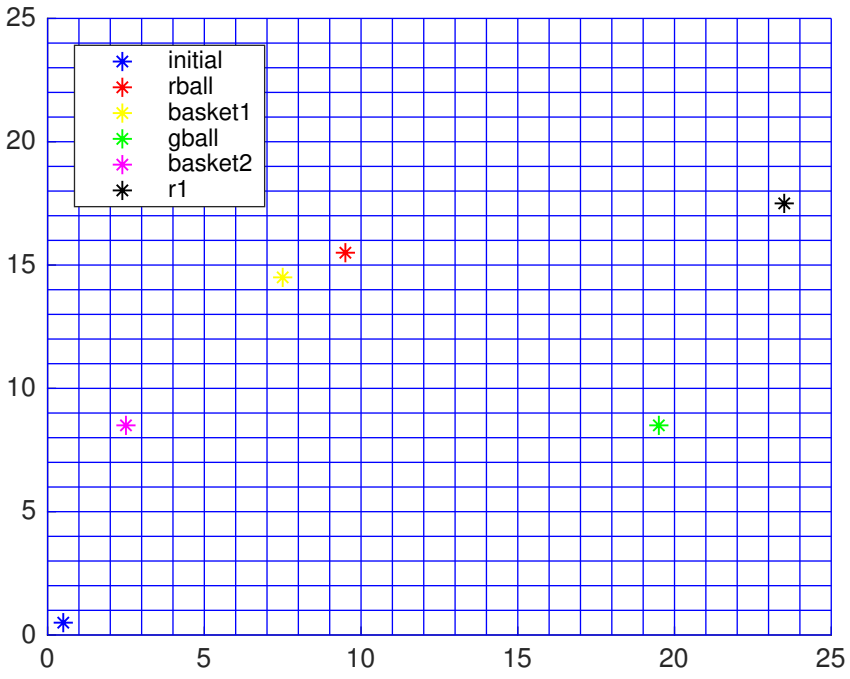

Figure 5.3: Workspace 2

Our Algorithm

```
new_algorithm_plan done within 0.03s: precost 66.00, sufcost 0.00
```

We note here pickrball and droprball are potential tasks i.e. they belong in AP_p. They are incoded in the action model in the P_MAS_TG framework. pickrball can only be done if rball is true, and this is only true in the region corresponding to (9,15). droprball can only be done if basket1 is true, and this is only true in the region corresponding to (7,14) (see figure 5.1). We give both of these actions an arbitrary cost of 10. The way P_MAS_TG treats actions gives increases the size of the product automaton by three fold. This is because when a predicate is incoded as an action, each state has corresponding states for doing that action in this state. So instead of having a product automaton of size |FTS| × |Büchi| we have a size |FTS| × |Büchi| × |possible actions|. The possible actions in this case are { "none", "pickrball", "droprball" }. We said that pickrball was only possible when rball is true and droprball is only possible when basket1 is true. This statement is still valid, the resulting contradictory nodes simply have no edges leading to them so they cannot be reached.

5.2 Example 1 Overlapping Regions

We now look at the same example, except now we have a different workspace. We choose this example to show what happens if the regions of interest are overlapping. Because of the way the code is structured, we are not able to make take transitions with two or more propositions at one time. We show the example if rball is in the same area as the basket. If this is the case, then pickrball and dropbasket could theoretically be done simultaneously. However, this is not possible because of the code. The only possible transitions that we can take that include && have to be a region and a potential task. We leave the transitions satisfying this requirement and remove all others so the algorithm can have a better distances to follow. The automaton is now

In this automaton, we now have $d(q_1) = 3$, $d(q_2) = 2$, $d(q_3) = 3$, $d(q_4) = 1$, $d(q_5) = 2$ and $d(q_6) = 0$. We see again that rball and basket are not one step away from the initial node, implying that we cannot take the first step pickrball. This means that we can never go to q_2, and that there is only one path through the automaton to take. Therefore our algorithm is guaranteed to compute the same path as the accepted algorithm and it will do so in less time.

We look at the results and see that our algorithm indeed calculates the same path in a shorter amount of time than the accepted algorithm.

```
Accepted Algorithm
```

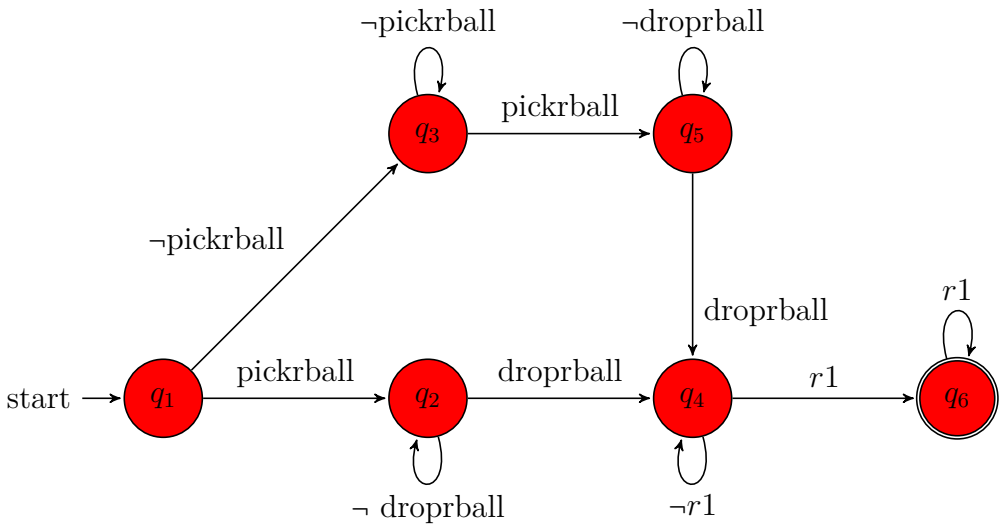

Figure 5.4: Simplified Büchi Automaton Corresponding to $\varphi = \diamond(\text{pickrball} \wedge \diamond\text{droprball}) \wedge \diamond \square r1$ 2

```
Dijkstra_plan_networkX done within 0.05s: precost 60.00, sufcost 0.00
...
full construction and synthesis done within 1.11s
    .
.
.
```

```
new_algorithm_plan done within 0.02s: precost 60.00, sufcost 0.00

full construction and synthesis done within 1.09s
```

5.3 Example 2

We now look at the example taken from [13] in which the robot has to pick up and deliver two different balls (rball and gball) to two different baskets, and the robot cannot carry two balls at once. After this is done the robot is to go to r1 and stay there. This task is formalized as $\varphi = \diamond(\text{pickrball} \wedge \diamond(\text{droprball})) \wedge \diamond(\text{pickgball} \wedge \diamond(\text{dropgball})) \wedge \square(\text{pickrball} \rightarrow \mathbf{X}(\neg\text{pickgball}\,\mathcal{U}\,\text{droprball})) \wedge \square(\text{pickgball} \rightarrow \mathbf{X}(\neg\text{pickrball}\,\mathcal{U}\,\text{dropgball}))\,\&\&\,\diamond\square r1$. This formula formalizes the basket corresponding to rball is in region r2 and the basket corresponding to gball is in r4. The Büchi automaton corresponding to this formula is much to large to show. It has 75 states and 797 edges. If the reader is

interested, the automaton can be found using the online tool [1] with the input F(pickrball && F(droprball)) && F(pickgball && F(dropgball)) && G(pickrball -> X(! pickgball U droprball)) && G(pickgball -> X(! pickrball U dropgball)) && F(G(r1)). It is too large for the tool to give a visual representation but it will provide a list of states and edges.

To analyse the performance of our algorithm on this problem, we are going to break up this problem into the choices that the robot has. The robot has to pick up one of the balls, return it to the corresponding basket, then pick up the second ball and return it to its corresponding basket. Assuming that everything else is done in the optimal way, the only choice that must be made is which ball to pick up first. This is the output, accepted algorithm first:

```
Accepted Algorithm
```

```
Dijkstra_plan_networkX done within 0.63s: precost 118.00, sufcost 0.00
...
full construction and synthesis done within 55.25s
```

and our algorithm

```
new_algorithm_plan done within 0.28s: precost 130.00, sufcost 0.00
...
full construction and synthesis done within 57.55s
```

as we can see, our algorithm picks the closest ball (rball) first even though it is not optimal overall. The added cost is relatively small though, only 12. However, it is possible that the difference could be much larger. In an analysis of speed, our algorithm does the search faster, In about half the time. In either case, the actual search takes around or less than a hundredth of the total time.

5.4 Example 2 Modified

We now look at a modified version of example two. Say the robot has to pick up and deliver two different balls (rball and gball) to two different baskets, and the robot cannot carry two balls at once and that is it. There is no need to go to r1, or do anything after the balls are returned to their respective baskets. This task is formalized as $\varphi = \diamond(\text{pickrball} \wedge \diamond(\text{droprball})) \wedge \diamond(\text{pickgball} \wedge \diamond(\text{dropgball})) \wedge \square(\text{pickrball} \rightarrow \mathbf{X}(\neg\text{pickgball}\,\mathcal{U}\,\text{droprball})) \wedge \square(\text{pickgball} \rightarrow \mathbf{X}(\neg\text{pickrball}\,\mathcal{U}\,\text{dropgball}))$. This version seems easier than original example. The Büchi automaton for this formula would seem to agree; it is smaller, with 38 nodes and 308 edges. However, here is the out put from both algorithms:

```
Accepted Algorithm
```

```
Dijkstra_plan_networkX done within 926.40s: precost 101.00, sufcost
    0.00
...
full construction and synthesis done within 950.95s
```

and from our algorithm

```
Our Algorithm
```

```
new_algorithm_plan done within 0.30s: precost 104.00, sufcost 0.00
...
full construction and synthesis done within 21.60s
```

This is by far the largest difference in computation times that we have seen thus far! We will show how this difference is caused by the searches from accepting nodes back to themselves.

As before, we have |product automaton| = |FTS|×|Büchi|×|possible actions|. The FTS still has 625 nodes, this buchi automaton has 38 nodes, and there are 5 possible actions ("none", "pickrball", "droprball", "pickgball", "dropgball"). This makes for a product automaton of 118750. There are 4 accepting states in the Büchi automaton. This gives us 12500 accepting nodes in the product automaton. The accepted algorithm calculates a path back from each accepting node back to itself. This calculation is relatively quick if the accepting node can transition back to itself; however, the cost is much longer if it doesn't. It turns out in this example only 629 of the accepting nodes have a self loop, while 11871 do not. It may seem like the accepting nodes should all be able to transfer back to themselves, but they cannot because of the accepting nodes in the Büchi automaton.

Each accepting node in the Büchi automaton has a self loop however, the labels of the self loop can make many of these transitions impossible. The four self loop labels for the four accepting nodes of the Büchi automaton are (droprball && dropgball), (!pickrball && dropgball), (!pickrball && !pickgball), and (droprball && !pickgball). Thus the only possible self loop is (!pickrball && !pickgball). All other accepting states must look for another path back to themselves, which results in the incredible increase in time. This is likely the largest danger and downside of using the accepted algorithm. The number of accepting states grows with the size of the FTS, the Büchi automaton, and the number of possible actions. Then we have to do a search for each of these accepting nodes.

5.5 Other Examples

As we have seen, due to the complexity of the Büchi automata it can be very hard to analyse the performance of our algorithm compared to the accepted algorithm with respect to cost for more complex formulas. We therefore provide the results of runs for various formulas in table 5.1. We use formulas from the table in [21] to try to show a comprehensive experimentation. The formulas are run on the workspace in figure 5.1.

Formula	Accepted Cost prefix, suffix	Accepted Time	Our Cost prefix, suffix	Our Time
'(!r223 U r445) || (!r268 U r435)'	27, 0	0.04	27, 0	0.01
'!r62 U(!r266 U r422)'	38, 0	0.05	38, 0	0.02
'[]<> r0 -> []<> r317'	1, 0	5.06	1, 0	0.00
'[]<> r0 <-> []<> r317'	1, 0	10.70	1, 0	0.00
'!(<><> r498 <-> r541)'	42, 0	0.03	42, 0	0.02
'!([]<> r3 -> []<>r591)'	3, 0	5.06	3, 0	0.00
'!([]<> r3 <-> []<>r591)'	3, 0	10.31	39, 0	0.01
'!r532 R (!r432 || r321)'	0, 0	4.97	0, 0	0.01
'<> r114 && [](r114 -> <> r12) && ((X r114 U X r12) || !X(r114 U r12))'	24	0.08	24	0.01
'<> pickrball && [](pickrball -> <> droprball) && ((X pickrball U X droprball) || !X(pickrball U droprball))'	47, 0	28.87	47, 0	0.03
' <> r124 && <> !r124'	28, 0	0.05	28, 0	0.01

Table 5.1: Comparison of Accepted Algorithm with Our Algorithm on Various Examples

As we can see, our algorithm always produces a path in a shorter time than the accepted algorithm. However, it can happen that our algorithm produces a plan that is much worse than the accepted algorithm e.g. '!([]<> r3 <-> []<>r591)'.

Chapter 6

Conclusion and Future Work

In this report we have presented a new algorithm for computing a path that satisfies a given LTL formula. Like the accepted algorithm we also represent the LTL formula as a Büchi automaton, and the robot's motion as a finite transition system. The product of these two graphs is then searched to find an accepting path, which is a path that passes through an accepting node an infinite number of times.

The novel idea of our algorithm is the injection of distance information into the Büchi automaton. Then when we search the product automaton, we are able to follow a path of decreasing distance to the accepting nodes. We have seen in the report that our algorithm produces a path faster than the accepted algorithm, although in general it does not produce the optimal path. For two common formula types (reachability while avoiding regions and sequencing) that our algorithm will produce the optimal path in a shorter time than the accepted algorithm. We have also shown that for other more complex formulas, such as example 1 in chapter Complex Formulas our algorithm is guaranteed to give the optimal path in a shorter amount of time than the accepted algorithm. The problem with this is that this required a thorough analysis of the problem and the Büchi automaton. This seems like it could be more trouble than it is worth in a practical setting, because one of the main advantages of this method of automated path generation is that one does *not* need to analyse and reason about the task. Also the problem we did this with was less complex than it could have been. Future work should focus on analysing more formulas in a generic form to provide a general rule on how our algorithm performs on formulas of a certain type. It is too much work to analyse each formula individually.

For coverage formulas we have derived a bound on the cost of the path computed by our algorithm and the optimal path, however it can be a very loose bound in general. Future work could be done of tightening this bound,

and also on finding bounds for formulas of other types. Even though our bound is loose, it still makes our algorithm more attractive. If the is no guarantee that our algorithm is not going to produce a path that is incredibly long compared to the accepted algorithm then it can make people nervous to use it. We note in the section of Recurrence formulas that the accepted algorithm is also not guaranteed to produce the optimal path when the Büchi automaton is not tight. This means that the optimal path is not necessarily needed and provides a reason to use our algorithm.

Besides the cost of the calculated run, we also looked at the time it takes both algorithms to calculate the runs. Our algorithm was faster in every example. We are not aware of any circumstances in which the accepted algorithm computes a path fast, and it is likely that it will not. However, the bottleneck both these algorithms is usually not the search, it is the construction of the product automata. In some situations, the search time is around a hundredth of the time to construct the product automaton; this is true even for the accepted algorithm. It seems that future work should then instead focus on reducing the size of the product automaton, or the time it takes to construct it.

It is true that the construction of the product automaton usually takes up the majority of the running time of the algorithm, except in the situation with non-trival suffixes. We saw in the modified example 2 in the complex formulas chapter that sometimes an incredibly large amount of time can be spent calculating the paths from the accepting nodes back to themselves. Our algorithm is safe from this as it does not calculate the shortest path from *all* accepting nodes back to themselves.

Overall our algorithm produces an accepting path faster than the accepted algorithm, however it is not guaranteed to be optimal. The accepted algorithm is also not guaranteed to be optimal unless the Büchi automaton is tight, but our algorithm can never produce a cheaper cost than the accepted algorithm. There are certain types of formulas that our algorithm will perform better on than the accepted, however besides these formulas it can be hard to analyse the performance of our algorithm. When deciding which algorithm to use, one should weigh the importance of optimally and speed. Our algorithm is faster, however given that the overwhelming majority of time is spent constructing the product automaton, our increased speed in the search may not be very attractive or helpful.

Chapter 7

Appendix

```
from collections import deque
from heapq import heappush, heappop
from itertools import count
import networkx as nx
from networkx.utils import generate_unique_node
import warnings as _warnings
from networkx import dijkstra_predecessor_and_distance

def adapted_dijkstra_multisource(G, source, cutoff=None, target=None):
    """Uses Dijkstra's algorithm to find shortest weighted paths
    Parameters
    ----------
    G : NetworkX graph
    sources : non-empty iterable of nodes
        Starting nodes for paths. If this is just an iterable
            containing
        a single node, then all paths computed by this function will
        start from that node. If there are two or more nodes in this
        iterable, the computed paths may begin from any one of the
            start
        nodes.
    target : node label, optional
        Ending node for path. Search is halted when target is found.
    cutoff : integer or float, optional
        Depth to stop the search. Only return paths with length <=
            cutoff.
    Returns
    -------
    dist : dictionary
        A mapping from node to shortest distance to that node from one
        of the source nodes.
    next_node : tuple
        The first node, n, the search finds that is one level below
```

```
        the current node
    i.e. d_p(n) = lev - 1
paths: dictionary
    dict to store the path list from source to each node, keyed by
        node.
Notes
-----
The optional predecessor and path dictionaries can be accessed by
the caller through the original pred and paths objects passed
as arguments. No need to explicitly return pred or paths.
"""

paths = {source: [source]}

# define weight function
weight = lambda u, v, data: data.get('weight', 1)

# succ = successors
G_succ = G.succ if G.is_directed() else G.adj

# rename functions
push = heappush
pop = heappop

dist = {}  # dictionary of final distances
seen = {}
# fringe is heapq with 3-tuples (distance,c,node)
# use the count c to avoid comparing nodes (may not be able to)
c = count()
fringe = []
next_node = []
# current level of starting node
cur_level = G.node[source]['dist']
if cur_level == 0:
    cur_level = 1
#for source in sources:
seen[source] = 0
push(fringe, (0, next(c), source))
while fringe:
    (d, _, v) = pop(fringe)
    if v in dist:
        continue  # already searched this node.
    dist[v] = d
    if G.node[v]['dist'] < cur_level:
        if cur_level == 1:
            if v in G.predecessors(v):
                print 'self loop'
                next_node = v
                break
```

```
                loop_pre, loop_dist =
                    dijkstra_predecessor_and_distance(G, v)
                if v in loop_dist.keys():

                    if loop_dist[v] != 0:
                        print 'not self loop'
                        next_node = v
                        break
            else:
                next_node = v
                break

        for u, e in G_succ[v].items():
            cost = weight(v, u, e)
            if cost is None:
                continue
            vu_dist = dist[v] + cost
            if cutoff is not None:
                if vu_dist > cutoff:
                    continue
            if u in dist:
                if vu_dist < dist[u]:
                    raise ValueError('Contradictory paths found:',
                                     'negative weights?')
            elif u not in seen or vu_dist < seen[u]:
                seen[u] = vu_dist
                push(fringe, (vu_dist, next(c), u))
                if paths is not None:
                    paths[u] = paths[v] + [u]

    return dist, next_node, paths

def greedy_plan(product, beta=10):
        # requires a full construct of product automaton
        # used for 'stadic' and 'ready'
        start = time.time()
        runs = {}
        loop = {}
        cycle = {}
        line = {}
        #
        #
        # Find the shortest path
        # to each accepting state
        #
        for prod_init in product.graph['initial']:
                # Find prefix
                lev = product.node[prod_init]['dist']
                start_node = prod_init
```

```
        precost = 0
        prefix = [start_node]
        if lev == 0:
                lev = 1
        while lev > 0:

                di,targ,path = ga.adapted_dijkstra_multisource
                    (product,start_node)

                if targ == []:
                        print '=====================',
                        print 'No accepting run found in
                            planning!'
                        return None, None

                lev = product.node[targ]['dist']

                start_node = targ

                print start_node
                precost = precost+di[targ]
                if lev == 0:
                        prefix.extend(path[targ][1:])

                else:
                        prefix.extend(path[targ][1:-1])

prod_target = targ



if prod_target in product.predecessors(prod_target):

        loop[prod_target] = (product.edge[prod_target][
            prod_target]["weight"], [prod_target, prod_target
            ])


else:

        loop_pre, loop_dist =
            dijkstra_predecessor_and_distance(product,
            prod_target)

        for target_pred in product.predecessors_iter(
            prod_target):

                if target_pred in loop_dist:
                        cycle[target_pred] = product.edge[
```

```
                                target_pred][prod_target]["weight"
                                ] + loop_dist[target_pred]

                if cycle:
                        opti_pred = min(cycle, key = cycle.get)
                        suffix = compute_path_from_pre(loop_pre,
                            opti_pred)
                        loop[prod_target] = (cycle[opti_pred], suffix)


        for target in loop.iterkeys():
                if target == targ:
                        line[target] = precost+beta*loop[target][0]
                if line:
                        runs[(prod_init, targ)] = (prefix, precost,
                            loop[targ][1], loop[targ][0])

        if runs:
                prefix, precost, suffix, sufcost = min(runs.values(),
                    key = lambda p: p[1] + beta*p[3])
                run = ProdAut_Run(product, prefix, precost, suffix,
                    sufcost, precost+beta*sufcost)
                print '=========================='
                print 'new algorithm plan done within %.2fs: precost
                    %.2f, sufcost %.2f' %(time.time()-start, round(
                    precost), round(sufcost))
                return run, time.time()-start

        print '=========================='
        print 'No accepting run found in optimal planning!'
        return None, None
```

Sequencing Path

```
-----------------------------
the prefix of plan **states**:
[((0, 0, 1), 'None'), ((1, 0, 1), 'None'), ((1, 1, 1), 'None'), ((1,
    2, 1), 'None'), ((2, 2, 1), 'None'), ((3, 2, 1), 'None'), ((3, 3,
    1), 'None'), ((4, 3, 1), 'None'), ((5, 3, 1), 'None'), ((5, 4, 1),
     'None'), ((5, 5, 1), 'None'), ((5, 6, 1), 'None'), ((6, 6, 1), '
    None'), ((7, 6, 1), 'None'), ((7, 7, 1), 'None'), ((8, 7, 1), '
    None'), ((9, 7, 1), 'None'), ((9, 8, 1), 'None'), ((9, 9, 1), '
    None'), ((9, 10, 1), 'None'), ((10, 10, 1), 'None'), ((11, 10, 1),
     'None'), ((11, 11, 1), 'None'), ((11, 12, 1), 'None'), ((12, 12,
    1), 'None'), ((12, 13, 1), 'None'), ((12, 14, 1), 'None'), ((12,
    15, 1), 'None'), ((13, 15, 1), 'None'), ((14, 15, 1), 'None'),
    ((15, 15, 1), 'None'), ((16, 15, 1), 'None'), ((17, 15, 1), 'None
    '), ((18, 15, 1), 'None'), ((19, 15, 1), 'None'), ((20, 15, 1), '
    None'), ((19, 15, 1), 'None'), ((19, 16, 1), 'None'), ((18, 16, 1)
```

```
    , 'None'), ((18, 17, 1), 'None'), ((17, 17, 1), 'None'), ((17, 18,
     1), 'None'), ((16, 18, 1), 'None'), ((15, 18, 1), 'None'), ((15,
    19, 1), 'None'), ((14, 19, 1), 'None'), ((14, 20, 1), 'None'),
    ((14, 21, 1), 'None'), ((13, 21, 1), 'None'), ((13, 22, 1), 'None
    '), ((12, 22, 1), 'None'), ((11, 22, 1), 'None'), ((11, 23, 1), '
    None'), ((11, 24, 1), 'None'), ((10, 24, 1), 'None'), ((9, 24, 1),
     'None'), ((8, 24, 1), 'None'), ((7, 24, 1), 'None'), ((6, 24, 1),
     'None'), ((5, 24, 1), 'None'), ((4, 24, 1), 'None'), ((3, 24, 1),
     'None'), ((2, 24, 1), 'None'), ((2, 24, 1), 'None')]
the suffix of plan **states**:
[((2, 24, 1), 'None'), ((2, 24, 1), 'None')]
------------------------------
the prefix of plan **actions**:
[(0, 0, 1), (1, 0, 1), (1, 1, 1), (1, 2, 1), (2, 2, 1), (3, 2, 1), (3,
     3, 1), (4, 3, 1), (5, 3, 1), (5, 4, 1), (5, 5, 1), (5, 6, 1), (6,
     6, 1), (7, 6, 1), (7, 7, 1), (8, 7, 1), (9, 7, 1), (9, 8, 1), (9,
     9, 1), (9, 10, 1), (10, 10, 1), (11, 10, 1), (11, 11, 1), (11,
    12, 1), (12, 12, 1), (12, 13, 1), (12, 14, 1), (12, 15, 1), (13,
    15, 1), (14, 15, 1), (15, 15, 1), (16, 15, 1), (17, 15, 1), (18,
    15, 1), (19, 15, 1), (20, 15, 1), (19, 15, 1), (19, 16, 1), (18,
    16, 1), (18, 17, 1), (17, 17, 1), (17, 18, 1), (16, 18, 1), (15,
    18, 1), (15, 19, 1), (14, 19, 1), (14, 20, 1), (14, 21, 1), (13,
    21, 1), (13, 22, 1), (12, 22, 1), (11, 22, 1), (11, 23, 1), (11,
    24, 1), (10, 24, 1), (9, 24, 1), (8, 24, 1), (7, 24, 1), (6, 24,
    1), (5, 24, 1), (4, 24, 1), (3, 24, 1), (2, 24, 1), 'None', 'None
    ']
the suffix of plan **actions**:
['None', 'None']
```

Coverage Path Accepted Algorithm

```
Accepted Algorithm
==========

Dijkstra_plan_networkX done within 0.08s: precost 59.00, sufcost 0.00
------------------------------
the prefix of plan **states**:
[((0, 0, 1), 'None'), ((1, 0, 1), 'None'), ((2, 0, 1), 'None'), ((2,
    1, 1), 'None'), ((2, 2, 1), 'None'), ((2, 3, 1), 'None'), ((2, 4,
    1), 'None'), ((2, 5, 1), 'None'), ((2, 6, 1), 'None'), ((2, 7, 1),
     'None'), ((2, 8, 1), 'None'), ((2, 9, 1), 'None'), ((2, 10, 1), '
    None'), ((2, 11, 1), 'None'), ((2, 12, 1), 'None'), ((2, 13, 1), '
    None'), ((2, 14, 1), 'None'), ((2, 15, 1), 'None'), ((2, 16, 1), '
    None'), ((2, 17, 1), 'None'), ((2, 18, 1), 'None'), ((2, 19, 1), '
    None'), ((2, 20, 1), 'None'), ((2, 21, 1), 'None'), ((2, 22, 1), '
    None'), ((2, 23, 1), 'None'), ((2, 24, 1), 'None'), ((2, 23, 1), '
    None'), ((2, 22, 1), 'None'), ((2, 21, 1), 'None'), ((3, 21, 1), '
    None'), ((4, 21, 1), 'None'), ((5, 21, 1), 'None'), ((5, 20, 1), '
    None'), ((5, 19, 1), 'None'), ((6, 19, 1), 'None'), ((7, 19, 1), '
    None'), ((8, 19, 1), 'None'), ((9, 19, 1), 'None'), ((9, 18, 1), '
```

```
    None'), ((10, 18, 1), 'None'), ((11, 18, 1), 'None'), ((12, 18, 1)
    , 'None'), ((12, 17, 1), 'None'), ((12, 16, 1), 'None'), ((12, 15,
     1), 'None'), ((12, 14, 1), 'None'), ((12, 13, 1), 'None'), ((12,
    12, 1), 'None'), ((13, 12, 1), 'None'), ((14, 12, 1), 'None'),
    ((14, 13, 1), 'None'), ((15, 13, 1), 'None'), ((16, 13, 1), 'None
    '), ((16, 14, 1), 'None'), ((17, 14, 1), 'None'), ((18, 14, 1), '
    None'), ((18, 15, 1), 'None'), ((19, 15, 1), 'None'), ((20, 15, 1)
    , 'None'), ((20, 15, 1), 'None')]
the suffix of plan **states**:
[((20, 15, 1), 'None'), ((20, 15, 1), 'None')]
------------------------------
the prefix of plan **actions**:
[(0, 0, 1), (1, 0, 1), (2, 0, 1), (2, 1, 1), (2, 2, 1), (2, 3, 1), (2,
     4, 1), (2, 5, 1), (2, 6, 1), (2, 7, 1), (2, 8, 1), (2, 9, 1), (2,
     10, 1), (2, 11, 1), (2, 12, 1), (2, 13, 1), (2, 14, 1), (2, 15,
    1), (2, 16, 1), (2, 17, 1), (2, 18, 1), (2, 19, 1), (2, 20, 1),
    (2, 21, 1), (2, 22, 1), (2, 23, 1), (2, 24, 1), (2, 23, 1), (2,
    22, 1), (2, 21, 1), (3, 21, 1), (4, 21, 1), (5, 21, 1), (5, 20, 1)
    , (5, 19, 1), (6, 19, 1), (7, 19, 1), (8, 19, 1), (9, 19, 1), (9,
    18, 1), (10, 18, 1), (11, 18, 1), (12, 18, 1), (12, 17, 1), (12,
    16, 1), (12, 15, 1), (12, 14, 1), (12, 13, 1), (12, 12, 1), (13,
    12, 1), (14, 12, 1), (14, 13, 1), (15, 13, 1), (16, 13, 1), (16,
    14, 1), (17, 14, 1), (18, 14, 1), (18, 15, 1), (19, 15, 1), (20,
    15, 1), 'None', 'None']
the suffix of plan **actions**:
['None', 'None']
full construction and synthesis done within 0.43s
```

Coverage Path Our Algorithm

```
Our Algorithm
==========
new_algorithm_plan done within 0.02s: precost 62.00, sufcost 0.00
------------------------------
the prefix of plan **states**:
[((0, 0, 1), 'None'), ((1, 0, 1), 'None'), ((2, 0, 1), 'None'), ((3,
    0, 1), 'None'), ((3, 1, 1), 'None'), ((4, 1, 1), 'None'), ((5, 1,
    1), 'None'), ((6, 1, 1), 'None'), ((6, 2, 1), 'None'), ((6, 3, 1),
     'None'), ((6, 4, 1), 'None'), ((6, 5, 1), 'None'), ((7, 5, 1), '
    None'), ((8, 5, 1), 'None'), ((8, 6, 1), 'None'), ((9, 6, 1), '
    None'), ((10, 6, 1), 'None'), ((10, 7, 1), 'None'), ((10, 8, 1), '
    None'), ((10, 9, 1), 'None'), ((11, 9, 1), 'None'), ((12, 9, 1), '
    None'), ((12, 10, 1), 'None'), ((12, 11, 1), 'None'), ((12, 12, 1)
    , 'None'), ((12, 13, 1), 'None'), ((13, 13, 1), 'None'), ((13, 14,
     1), 'None'), ((14, 14, 1), 'None'), ((15, 14, 1), 'None'), ((16,
    14, 1), 'None'), ((17, 14, 1), 'None'), ((17, 15, 1), 'None'),
    ((18, 15, 1), 'None'), ((19, 15, 1), 'None'), ((20, 15, 1), 'None
    '), ((20, 16, 1), 'None'), ((19, 16, 1), 'None'), ((19, 17, 1), '
    None'), ((18, 17, 1), 'None'), ((18, 18, 1), 'None'), ((17, 18, 1)
```

```
    , 'None'), ((17, 19, 1), 'None'), ((17, 20, 1), 'None'), ((16, 20,
     1), 'None'), ((15, 20, 1), 'None'), ((15, 21, 1), 'None'), ((14,
    21, 1), 'None'), ((14, 22, 1), 'None'), ((13, 22, 1), 'None'),
    ((13, 23, 1), 'None'), ((13, 24, 1), 'None'), ((12, 24, 1), 'None
    '), ((11, 24, 1), 'None'), ((10, 24, 1), 'None'), ((9, 24, 1), '
    None'), ((8, 24, 1), 'None'), ((7, 24, 1), 'None'), ((6, 24, 1), '
    None'), ((5, 24, 1), 'None'), ((4, 24, 1), 'None'), ((3, 24, 1), '
    None')]
the suffix of plan **states**:
[((2, 24, 1), 'None'), ((2, 24, 1), 'None')]
----------------------------
the prefix of plan **actions**:
[(0, 0, 1), (1, 0, 1), (2, 0, 1), (3, 0, 1), (3, 1, 1), (4, 1, 1), (5,
     1, 1), (6, 1, 1), (6, 2, 1), (6, 3, 1), (6, 4, 1), (6, 5, 1), (7,
     5, 1), (8, 5, 1), (8, 6, 1), (9, 6, 1), (10, 6, 1), (10, 7, 1),
    (10, 8, 1), (10, 9, 1), (11, 9, 1), (12, 9, 1), (12, 10, 1), (12,
    11, 1), (12, 12, 1), (12, 13, 1), (13, 13, 1), (13, 14, 1), (14,
    14, 1), (15, 14, 1), (16, 14, 1), (17, 14, 1), (17, 15, 1), (18,
    15, 1), (19, 15, 1), (20, 15, 1), (20, 16, 1), (19, 16, 1), (19,
    17, 1), (18, 17, 1), (18, 18, 1), (17, 18, 1), (17, 19, 1), (17,
    20, 1), (16, 20, 1), (15, 20, 1), (15, 21, 1), (14, 21, 1), (14,
    22, 1), (13, 22, 1), (13, 23, 1), (13, 24, 1), (12, 24, 1), (11,
    24, 1), (10, 24, 1), (9, 24, 1), (8, 24, 1), (7, 24, 1), (6, 24,
    1), (5, 24, 1), (4, 24, 1), (3, 24, 1), (2, 24, 1)]
the suffix of plan **actions**:
['None', 'None']
full construction and synthesis done within 0.38s
```

Recurrence (Liveliness) Path

```
the prefix of plan **states**:
[((0, 0, 1), 'None'), ((1, 0, 1), 'None'), ((2, 0, 1), 'None'), ((3,
    0, 1), 'None'), ((3, 1, 1), 'None'), ((4, 1, 1), 'None'), ((5, 1,
    1), 'None'), ((6, 1, 1), 'None'), ((6, 2, 1), 'None'), ((6, 3, 1),
     'None'), ((6, 4, 1), 'None'), ((6, 5, 1), 'None'), ((7, 5, 1), '
    None'), ((8, 5, 1), 'None'), ((8, 6, 1), 'None'), ((9, 6, 1), '
    None'), ((10, 6, 1), 'None'), ((10, 7, 1), 'None'), ((10, 8, 1), '
    None'), ((10, 9, 1), 'None'), ((11, 9, 1), 'None'), ((12, 9, 1), '
    None'), ((12, 10, 1), 'None'), ((12, 11, 1), 'None'), ((12, 12, 1)
    , 'None'), ((12, 13, 1), 'None'), ((12, 14, 1), 'None'), ((13, 14,
     1), 'None'), ((13, 15, 1), 'None'), ((14, 15, 1), 'None'), ((15,
    15, 1), 'None'), ((16, 15, 1), 'None'), ((17, 15, 1), 'None'),
    ((18, 15, 1), 'None'), ((19, 15, 1), 'None'), ((20, 15, 1), 'None
    '), ((19, 15, 1), 'None'), ((19, 16, 1), 'None'), ((18, 16, 1), '
    None'), ((18, 17, 1), 'None'), ((17, 17, 1), 'None'), ((16, 17, 1)
    , 'None'), ((16, 18, 1), 'None'), ((15, 18, 1), 'None'), ((15, 19,
     1), 'None'), ((14, 19, 1), 'None'), ((14, 20, 1), 'None'), ((14,
    21, 1), 'None'), ((13, 21, 1), 'None'), ((13, 22, 1), 'None'),
    ((12, 22, 1), 'None'), ((11, 22, 1), 'None'), ((11, 23, 1), 'None
```

```
    '), ((11, 24, 1), 'None'), ((10, 24, 1), 'None'), ((9, 24, 1), '
    None'), ((8, 24, 1), 'None'), ((7, 24, 1), 'None'), ((6, 24, 1), '
    None'), ((5, 24, 1), 'None'), ((4, 24, 1), 'None'), ((3, 24, 1), '
    None')]
the suffix of plan **states**:
[((2, 24, 1), 'None'), ((2, 23, 1), 'None'), ((2, 22, 1), 'None'),
    ((2, 21, 1), 'None'), ((3, 21, 1), 'None'), ((3, 20, 1), 'None'),
    ((4, 20, 1), 'None'), ((5, 20, 1), 'None'), ((6, 20, 1), 'None'),
    ((6, 19, 1), 'None'), ((6, 18, 1), 'None'), ((6, 17, 1), 'None'),
    ((7, 17, 1), 'None'), ((7, 16, 1), 'None'), ((8, 16, 1), 'None'),
    ((9, 16, 1), 'None'), ((10, 16, 1), 'None'), ((10, 15, 1), 'None')
    , ((10, 14, 1), 'None'), ((10, 13, 1), 'None'), ((11, 13, 1), '
    None'), ((11, 12, 1), 'None'), ((12, 12, 1), 'None'), ((12, 13, 1)
    , 'None'), ((12, 14, 1), 'None'), ((13, 14, 1), 'None'), ((13, 15,
     1), 'None'), ((14, 15, 1), 'None'), ((15, 15, 1), 'None'), ((16,
    15, 1), 'None'), ((17, 15, 1), 'None'), ((18, 15, 1), 'None'),
    ((19, 15, 1), 'None'), ((20, 15, 1), 'None'), ((19, 15, 1), 'None
    '), ((19, 16, 1), 'None'), ((18, 16, 1), 'None'), ((18, 17, 1), '
    None'), ((17, 17, 1), 'None'), ((16, 17, 1), 'None'), ((16, 18, 1)
    , 'None'), ((15, 18, 1), 'None'), ((15, 19, 1), 'None'), ((14, 19,
     1), 'None'), ((14, 20, 1), 'None'), ((14, 21, 1), 'None'), ((13,
    21, 1), 'None'), ((13, 22, 1), 'None'), ((12, 22, 1), 'None'),
    ((11, 22, 1), 'None'), ((11, 23, 1), 'None'), ((11, 24, 1), 'None
    '), ((10, 24, 1), 'None'), ((9, 24, 1), 'None'), ((8, 24, 1), '
    None'), ((7, 24, 1), 'None'), ((6, 24, 1), 'None'), ((5, 24, 1), '
    None'), ((4, 24, 1), 'None'), ((3, 24, 1), 'None'), ((2, 24, 1), '
    None')]
------------------------------
the prefix of plan **actions**:
[(0, 0, 1), (1, 0, 1), (2, 0, 1), (3, 0, 1), (3, 1, 1), (4, 1, 1), (5,
     1, 1), (6, 1, 1), (6, 2, 1), (6, 3, 1), (6, 4, 1), (6, 5, 1), (7,
     5, 1), (8, 5, 1), (8, 6, 1), (9, 6, 1), (10, 6, 1), (10, 7, 1),
    (10, 8, 1), (10, 9, 1), (11, 9, 1), (12, 9, 1), (12, 10, 1), (12,
    11, 1), (12, 12, 1), (12, 13, 1), (12, 14, 1), (13, 14, 1), (13,
    15, 1), (14, 15, 1), (15, 15, 1), (16, 15, 1), (17, 15, 1), (18,
    15, 1), (19, 15, 1), (20, 15, 1), (19, 15, 1), (19, 16, 1), (18,
    16, 1), (18, 17, 1), (17, 17, 1), (16, 17, 1), (16, 18, 1), (15,
    18, 1), (15, 19, 1), (14, 19, 1), (14, 20, 1), (14, 21, 1), (13,
    21, 1), (13, 22, 1), (12, 22, 1), (11, 22, 1), (11, 23, 1), (11,
    24, 1), (10, 24, 1), (9, 24, 1), (8, 24, 1), (7, 24, 1), (6, 24,
    1), (5, 24, 1), (4, 24, 1), (3, 24, 1), (2, 24, 1)]
the suffix of plan **actions**:
[(2, 23, 1), (2, 22, 1), (2, 21, 1), (3, 21, 1), (3, 20, 1), (4, 20,
    1), (5, 20, 1), (6, 20, 1), (6, 19, 1), (6, 18, 1), (6, 17, 1),
    (7, 17, 1), (7, 16, 1), (8, 16, 1), (9, 16, 1), (10, 16, 1), (10,
    15, 1), (10, 14, 1), (10, 13, 1), (11, 13, 1), (11, 12, 1), (12,
    12, 1), (12, 13, 1), (12, 14, 1), (13, 14, 1), (13, 15, 1), (14,
    15, 1), (15, 15, 1), (16, 15, 1), (17, 15, 1), (18, 15, 1), (19,
    15, 1), (20, 15, 1), (19, 15, 1), (19, 16, 1), (18, 16, 1), (18,
```

```
    17, 1), (17, 17, 1), (16, 17, 1), (16, 18, 1), (15, 18, 1), (15,
    19, 1), (14, 19, 1), (14, 20, 1), (14, 21, 1), (13, 21, 1), (13,
    22, 1), (12, 22, 1), (11, 22, 1), (11, 23, 1), (11, 24, 1), (10,
    24, 1), (9, 24, 1), (8, 24, 1), (7, 24, 1), (6, 24, 1), (5, 24, 1)
    , (4, 24, 1), (3, 24, 1), (2, 24, 1), 'None']
```

Example 1

```
------------------------------
the prefix of plan **states**:
[((0, 0, 1), 'None'), ((0, 1, 1), 'None'), ((0, 2, 1), 'None'), ((1,
    2, 1), 'None'), ((1, 3, 1), 'None'), ((2, 3, 1), 'None'), ((2, 4,
    1), 'None'), ((3, 4, 1), 'None'), ((4, 4, 1), 'None'), ((4, 5, 1),
     'None'), ((5, 5, 1), 'None'), ((5, 6, 1), 'None'), ((5, 7, 1), '
    None'), ((6, 7, 1), 'None'), ((6, 8, 1), 'None'), ((7, 8, 1), '
    None'), ((8, 8, 1), 'None'), ((8, 9, 1), 'None'), ((9, 9, 1), '
    None'), ((9, 10, 1), 'None'), ((9, 11, 1), 'None'), ((9, 12, 1), '
    None'), ((9, 13, 1), 'None'), ((9, 14, 1), 'None'), ((9, 15, 1), '
    None'), ((9, 15, 1), 'pickrball'), ((9, 14, 1), 'None'), ((8, 14,
    1), 'None'), ((7, 14, 1), 'None'), ((7, 14, 1), 'droprball'), ((7,
     15, 1), 'None'), ((7, 16, 1), 'None'), ((7, 17, 1), 'None'), ((8,
     17, 1), 'None'), ((9, 17, 1), 'None'), ((10, 17, 1), 'None'),
    ((11, 17, 1), 'None'), ((12, 17, 1), 'None'), ((13, 17, 1), 'None
    '), ((14, 17, 1), 'None'), ((15, 17, 1), 'None'), ((16, 17, 1), '
    None'), ((17, 17, 1), 'None'), ((18, 17, 1), 'None'), ((19, 17, 1)
    , 'None'), ((20, 17, 1), 'None'), ((21, 17, 1), 'None'), ((22, 17,
     1), 'None'), ((23, 17, 1), 'None'), ((23, 17, 1), 'None')]
the suffix of plan **states**:
[((23, 17, 1), 'None'), ((23, 17, 1), 'None')]
------------------------------
the prefix of plan **actions**:
[(0, 0, 1), (0, 1, 1), (0, 2, 1), (1, 2, 1), (1, 3, 1), (2, 3, 1), (2,
     4, 1), (3, 4, 1), (4, 4, 1), (4, 5, 1), (5, 5, 1), (5, 6, 1), (5,
     7, 1), (6, 7, 1), (6, 8, 1), (7, 8, 1), (8, 8, 1), (8, 9, 1), (9,
     9, 1), (9, 10, 1), (9, 11, 1), (9, 12, 1), (9, 13, 1), (9, 14, 1)
    , (9, 15, 1), 'pickrball', (9, 14, 1), (8, 14, 1), (7, 14, 1), '
    droprball', (7, 15, 1), (7, 16, 1), (7, 17, 1), (8, 17, 1), (9,
    17, 1), (10, 17, 1), (11, 17, 1), (12, 17, 1), (13, 17, 1), (14,
    17, 1), (15, 17, 1), (16, 17, 1), (17, 17, 1), (18, 17, 1), (19,
    17, 1), (20, 17, 1), (21, 17, 1), (22, 17, 1), (23, 17, 1), 'None
    ', 'None']
the suffix of plan **actions**:
['None', 'None']
```

Example 1 Overlapping

```
------------------------------
the prefix of plan **states**:
[((0, 0, 1), 'None'), ((0, 1, 1), 'None'), ((0, 2, 1), 'None'), ((1,
    2, 1), 'None'), ((1, 3, 1), 'None'), ((2, 3, 1), 'None'), ((2, 4,
```

```
    1), 'None'), ((3, 4, 1), 'None'), ((4, 4, 1), 'None'), ((4, 5, 1),
     'None'), ((5, 5, 1), 'None'), ((5, 6, 1), 'None'), ((5, 7, 1), '
    None'), ((6, 7, 1), 'None'), ((6, 8, 1), 'None'), ((7, 8, 1), '
    None'), ((8, 8, 1), 'None'), ((8, 9, 1), 'None'), ((9, 9, 1), '
    None'), ((9, 10, 1), 'None'), ((9, 11, 1), 'None'), ((9, 12, 1), '
    None'), ((9, 13, 1), 'None'), ((9, 14, 1), 'None'), ((9, 15, 1), '
    None'), ((9, 15, 1), 'pickrball'), ((9, 15, 1), 'droprball'),
    ((10, 15, 1), 'None'), ((10, 16, 1), 'None'), ((11, 16, 1), 'None
    '), ((12, 16, 1), 'None'), ((12, 17, 1), 'None'), ((13, 17, 1), '
    None'), ((14, 17, 1), 'None'), ((15, 17, 1), 'None'), ((16, 17, 1)
    , 'None'), ((17, 17, 1), 'None'), ((18, 17, 1), 'None'), ((19, 17,
     1), 'None'), ((20, 17, 1), 'None'), ((21, 17, 1), 'None'), ((22,
    17, 1), 'None'), ((23, 17, 1), 'None'), ((23, 17, 1), 'None')]
the suffix of plan **states**:
[((23, 17, 1), 'None'), ((23, 17, 1), 'None')]
------------------------------
the prefix of plan **actions**:
[(0, 0, 1), (0, 1, 1), (0, 2, 1), (1, 2, 1), (1, 3, 1), (2, 3, 1), (2,
     4, 1), (3, 4, 1), (4, 4, 1), (4, 5, 1), (5, 5, 1), (5, 6, 1), (5,
     7, 1), (6, 7, 1), (6, 8, 1), (7, 8, 1), (8, 8, 1), (8, 9, 1), (9,
     9, 1), (9, 10, 1), (9, 11, 1), (9, 12, 1), (9, 13, 1), (9, 14, 1)
    , (9, 15, 1), 'pickrball', 'droprball', (10, 15, 1), (10, 16, 1),
    (11, 16, 1), (12, 16, 1), (12, 17, 1), (13, 17, 1), (14, 17, 1),
    (15, 17, 1), (16, 17, 1), (17, 17, 1), (18, 17, 1), (19, 17, 1),
    (20, 17, 1), (21, 17, 1), (22, 17, 1), (23, 17, 1), 'None', 'None
    ']
the suffix of plan **actions**:
['None', 'None']
```

Example 2 Accepted Algorithm Path

```
------------------------------
the prefix of plan **states**:
[((0, 0, 1), 'None'), ((1, 0, 1), 'None'), ((1, 1, 1), 'None'), ((2,
    1, 1), 'None'), ((2, 2, 1), 'None'), ((3, 2, 1), 'None'), ((3, 3,
    1), 'None'), ((4, 3, 1), 'None'), ((5, 3, 1), 'None'), ((6, 3, 1),
     'None'), ((7, 3, 1), 'None'), ((8, 3, 1), 'None'), ((8, 4, 1), '
    None'), ((9, 4, 1), 'None'), ((10, 4, 1), 'None'), ((11, 4, 1), '
    None'), ((12, 4, 1), 'None'), ((13, 4, 1), 'None'), ((14, 4, 1), '
    None'), ((15, 4, 1), 'None'), ((16, 4, 1), 'None'), ((16, 5, 1), '
    None'), ((16, 6, 1), 'None'), ((17, 6, 1), 'None'), ((18, 6, 1), '
    None'), ((19, 6, 1), 'None'), ((19, 7, 1), 'None'), ((19, 8, 1), '
    None'), ((19, 8, 1), 'pickgball'), ((19, 9, 1), 'None'), ((18, 9,
    1), 'None'), ((18, 10, 1), 'None'), ((17, 10, 1), 'None'), ((16,
    10, 1), 'None'), ((15, 10, 1), 'None'), ((14, 10, 1), 'None'),
    ((13, 10, 1), 'None'), ((12, 10, 1), 'None'), ((11, 10, 1), 'None
    '), ((10, 10, 1), 'None'), ((9, 10, 1), 'None'), ((8, 10, 1), '
    None'), ((7, 10, 1), 'None'), ((6, 10, 1), 'None'), ((5, 10, 1), '
    None'), ((4, 10, 1), 'None'), ((3, 10, 1), 'None'), ((2, 10, 1), '
```

```
    None'), ((2, 10, 1), 'dropgball'), ((2, 11, 1), 'None'), ((2, 12,
    1), 'None'), ((2, 13, 1), 'None'), ((2, 14, 1), 'None'), ((3, 14,
    1), 'None'), ((4, 14, 1), 'None'), ((5, 14, 1), 'None'), ((5, 15,
    1), 'None'), ((6, 15, 1), 'None'), ((7, 15, 1), 'None'), ((8, 15,
    1), 'None'), ((9, 15, 1), 'None'), ((9, 15, 1), 'pickrball'), ((8,
     15, 1), 'None'), ((7, 15, 1), 'None'), ((7, 14, 1), 'None'), ((7,
     14, 1), 'droprball'), ((8, 14, 1), 'None'), ((8, 15, 1), 'None'),
     ((9, 15, 1), 'None'), ((10, 15, 1), 'None'), ((11, 15, 1), 'None
    '), ((12, 15, 1), 'None'), ((12, 16, 1), 'None'), ((13, 16, 1), '
    None'), ((14, 16, 1), 'None'), ((15, 16, 1), 'None'), ((16, 16, 1)
    , 'None'), ((17, 16, 1), 'None'), ((18, 16, 1), 'None'), ((19, 16,
     1), 'None'), ((20, 16, 1), 'None'), ((21, 16, 1), 'None'), ((22,
    16, 1), 'None'), ((22, 16, 1), 'None')]
the suffix of plan **states**:
[((22, 16, 1), 'None'), ((22, 16, 1), 'None')]
------------------------------
the prefix of plan **actions**:
[(0, 0, 1), (1, 0, 1), (1, 1, 1), (2, 1, 1), (2, 2, 1), (3, 2, 1), (3,
     3, 1), (4, 3, 1), (5, 3, 1), (6, 3, 1), (7, 3, 1), (8, 3, 1), (8,
     4, 1), (9, 4, 1), (10, 4, 1), (11, 4, 1), (12, 4, 1), (13, 4, 1),
     (14, 4, 1), (15, 4, 1), (16, 4, 1), (16, 5, 1), (16, 6, 1), (17,
    6, 1), (18, 6, 1), (19, 6, 1), (19, 7, 1), (19, 8, 1), 'pickgball
    ', (19, 9, 1), (18, 9, 1), (18, 10, 1), (17, 10, 1), (16, 10, 1),
    (15, 10, 1), (14, 10, 1), (13, 10, 1), (12, 10, 1), (11, 10, 1),
    (10, 10, 1), (9, 10, 1), (8, 10, 1), (7, 10, 1), (6, 10, 1), (5,
    10, 1), (4, 10, 1), (3, 10, 1), (2, 10, 1), 'dropgball', (2, 11,
    1), (2, 12, 1), (2, 13, 1), (2, 14, 1), (3, 14, 1), (4, 14, 1),
    (5, 14, 1), (5, 15, 1), (6, 15, 1), (7, 15, 1), (8, 15, 1), (9,
    15, 1), 'pickrball', (8, 15, 1), (7, 15, 1), (7, 14, 1), '
    droprball', (8, 14, 1), (8, 15, 1), (9, 15, 1), (10, 15, 1), (11,
    15, 1), (12, 15, 1), (12, 16, 1), (13, 16, 1), (14, 16, 1), (15,
    16, 1), (16, 16, 1), (17, 16, 1), (18, 16, 1), (19, 16, 1), (20,
    16, 1), (21, 16, 1), (22, 16, 1), 'None', 'None']
the suffix of plan **actions**:
['None', 'None']
```

Example 2 Our Algorithm Path

```
------------------------------
the prefix of plan **states**:
[((0, 0, 1), 'None'), ((0, 1, 1), 'None'), ((0, 2, 1), 'None'), ((1,
    2, 1), 'None'), ((2, 2, 1), 'None'), ((3, 2, 1), 'None'), ((4, 2,
    1), 'None'), ((4, 3, 1), 'None'), ((5, 3, 1), 'None'), ((5, 4, 1),
     'None'), ((6, 4, 1), 'None'), ((7, 4, 1), 'None'), ((7, 5, 1), '
    None'), ((8, 5, 1), 'None'), ((8, 6, 1), 'None'), ((8, 7, 1), '
    None'), ((8, 8, 1), 'None'), ((8, 9, 1), 'None'), ((9, 9, 1), '
    None'), ((9, 10, 1), 'None'), ((9, 11, 1), 'None'), ((9, 12, 1), '
    None'), ((9, 13, 1), 'None'), ((9, 14, 1), 'None'), ((9, 15, 1), '
    None'), ((9, 15, 1), 'pickrball'), ((9, 14, 1), 'None'), ((8, 14,
```

```
    1), 'None'), ((7, 14, 1), 'None'), ((7, 14, 1), 'droprball'), ((8,
     14, 1), 'None'), ((8, 13, 1), 'None'), ((8, 12, 1), 'None'), ((9,
     12, 1), 'None'), ((10, 12, 1), 'None'), ((11, 12, 1), 'None'),
    ((11, 11, 1), 'None'), ((11, 10, 1), 'None'), ((12, 10, 1), 'None
    '), ((12, 9, 1), 'None'), ((13, 9, 1), 'None'), ((14, 9, 1), 'None
    '), ((15, 9, 1), 'None'), ((15, 8, 1), 'None'), ((16, 8, 1), 'None
    '), ((17, 8, 1), 'None'), ((18, 8, 1), 'None'), ((19, 8, 1), 'None
    '), ((19, 8, 1), 'pickgball'), ((18, 8, 1), 'None'), ((18, 9, 1),
    'None'), ((18, 10, 1), 'None'), ((17, 10, 1), 'None'), ((16, 10,
    1), 'None'), ((15, 10, 1), 'None'), ((14, 10, 1), 'None'), ((13,
    10, 1), 'None'), ((12, 10, 1), 'None'), ((11, 10, 1), 'None'),
    ((10, 10, 1), 'None'), ((9, 10, 1), 'None'), ((8, 10, 1), 'None'),
     ((7, 10, 1), 'None'), ((6, 10, 1), 'None'), ((5, 10, 1), 'None'),
     ((4, 10, 1), 'None'), ((3, 10, 1), 'None'), ((2, 10, 1), 'None'),
     ((2, 10, 1), 'dropgball'), ((2, 11, 1), 'None'), ((2, 12, 1), '
    None'), ((2, 13, 1), 'None'), ((3, 13, 1), 'None'), ((3, 14, 1), '
    None'), ((4, 14, 1), 'None'), ((4, 15, 1), 'None'), ((4, 16, 1), '
    None'), ((5, 16, 1), 'None'), ((6, 16, 1), 'None'), ((7, 16, 1), '
    None'), ((8, 16, 1), 'None'), ((9, 16, 1), 'None'), ((10, 16, 1),
    'None'), ((11, 16, 1), 'None'), ((12, 16, 1), 'None'), ((13, 16,
    1), 'None'), ((14, 16, 1), 'None'), ((15, 16, 1), 'None'), ((16,
    16, 1), 'None'), ((17, 16, 1), 'None'), ((18, 16, 1), 'None'),
    ((19, 16, 1), 'None'), ((20, 16, 1), 'None'), ((21, 16, 1), 'None
    '), ((22, 16, 1), 'None'), ((22, 16, 1), 'None')]
the suffix of plan **states**:
[((22, 16, 1), 'None'), ((22, 16, 1), 'None')]
------------------------------
the prefix of plan **actions**:
[(0, 0, 1), (0, 1, 1), (0, 2, 1), (1, 2, 1), (2, 2, 1), (3, 2, 1), (4,
     2, 1), (4, 3, 1), (5, 3, 1), (5, 4, 1), (6, 4, 1), (7, 4, 1), (7,
     5, 1), (8, 5, 1), (8, 6, 1), (8, 7, 1), (8, 8, 1), (8, 9, 1), (9,
     9, 1), (9, 10, 1), (9, 11, 1), (9, 12, 1), (9, 13, 1), (9, 14, 1)
    , (9, 15, 1), 'pickrball', (9, 14, 1), (8, 14, 1), (7, 14, 1), '
    droprball', (8, 14, 1), (8, 13, 1), (8, 12, 1), (9, 12, 1), (10,
    12, 1), (11, 12, 1), (11, 11, 1), (11, 10, 1), (12, 10, 1), (12,
    9, 1), (13, 9, 1), (14, 9, 1), (15, 9, 1), (15, 8, 1), (16, 8, 1),
     (17, 8, 1), (18, 8, 1), (19, 8, 1), 'pickgball', (18, 8, 1), (18,
     9, 1), (18, 10, 1), (17, 10, 1), (16, 10, 1), (15, 10, 1), (14,
    10, 1), (13, 10, 1), (12, 10, 1), (11, 10, 1), (10, 10, 1), (9,
    10, 1), (8, 10, 1), (7, 10, 1), (6, 10, 1), (5, 10, 1), (4, 10, 1)
    , (3, 10, 1), (2, 10, 1), 'dropgball', (2, 11, 1), (2, 12, 1), (2,
     13, 1), (3, 13, 1), (3, 14, 1), (4, 14, 1), (4, 15, 1), (4, 16,
    1), (5, 16, 1), (6, 16, 1), (7, 16, 1), (8, 16, 1), (9, 16, 1),
    (10, 16, 1), (11, 16, 1), (12, 16, 1), (13, 16, 1), (14, 16, 1),
    (15, 16, 1), (16, 16, 1), (17, 16, 1), (18, 16, 1), (19, 16, 1),
    (20, 16, 1), (21, 16, 1), (22, 16, 1), 'None', 'None']
the suffix of plan **actions**:
['None', 'None']
```

Example 2 Modified Accepted Algorithm Path

```
------------------------------
the prefix of plan **states**:
[((0, 0, 1), 'None'), ((1, 0, 1), 'None'), ((1, 1, 1), 'None'), ((1,
    2, 1), 'None'), ((2, 2, 1), 'None'), ((2, 3, 1), 'None'), ((3, 3,
    1), 'None'), ((4, 3, 1), 'None'), ((5, 3, 1), 'None'), ((6, 3, 1),
     'None'), ((7, 3, 1), 'None'), ((7, 4, 1), 'None'), ((8, 4, 1), '
    None'), ((9, 4, 1), 'None'), ((10, 4, 1), 'None'), ((11, 4, 1), '
    None'), ((12, 4, 1), 'None'), ((12, 5, 1), 'None'), ((12, 6, 1), '
    None'), ((13, 6, 1), 'None'), ((13, 7, 1), 'None'), ((13, 8, 1), '
    None'), ((14, 8, 1), 'None'), ((15, 8, 1), 'None'), ((16, 8, 1), '
    None'), ((17, 8, 1), 'None'), ((18, 8, 1), 'None'), ((19, 8, 1), '
    None'), ((19, 8, 1), 'pickgball'), ((18, 8, 1), 'None'), ((17, 8,
    1), 'None'), ((17, 9, 1), 'None'), ((17, 10, 1), 'None'), ((16,
    10, 1), 'None'), ((15, 10, 1), 'None'), ((14, 10, 1), 'None'),
    ((13, 10, 1), 'None'), ((12, 10, 1), 'None'), ((11, 10, 1), 'None
    '), ((10, 10, 1), 'None'), ((9, 10, 1), 'None'), ((8, 10, 1), '
    None'), ((7, 10, 1), 'None'), ((6, 10, 1), 'None'), ((5, 10, 1), '
    None'), ((4, 10, 1), 'None'), ((3, 10, 1), 'None'), ((2, 10, 1), '
    None'), ((2, 10, 1), 'dropgball'), ((2, 11, 1), 'None'), ((2, 12,
    1), 'None'), ((2, 13, 1), 'None'), ((3, 13, 1), 'None'), ((4, 13,
    1), 'None'), ((5, 13, 1), 'None'), ((5, 14, 1), 'None'), ((6, 14,
    1), 'None'), ((6, 15, 1), 'None'), ((7, 15, 1), 'None'), ((8, 15,
    1), 'None'), ((9, 15, 1), 'None'), ((9, 15, 1), 'pickrball'), ((9,
     14, 1), 'None'), ((8, 14, 1), 'None'), ((7, 14, 1), 'None'), ((7,
     14, 1), 'droprball'), ((7, 14, 1), 'None')]
the suffix of plan **states**:
[((7, 14, 1), 'None'), ((7, 14, 1), 'None')]
------------------------------
the prefix of plan **actions**:
[(0, 0, 1), (1, 0, 1), (1, 1, 1), (1, 2, 1), (2, 2, 1), (2, 3, 1), (3,
     3, 1), (4, 3, 1), (5, 3, 1), (6, 3, 1), (7, 3, 1), (7, 4, 1), (8,
     4, 1), (9, 4, 1), (10, 4, 1), (11, 4, 1), (12, 4, 1), (12, 5, 1),
     (12, 6, 1), (13, 6, 1), (13, 7, 1), (13, 8, 1), (14, 8, 1), (15,
    8, 1), (16, 8, 1), (17, 8, 1), (18, 8, 1), (19, 8, 1), 'pickgball
    ', (18, 8, 1), (17, 8, 1), (17, 9, 1), (17, 10, 1), (16, 10, 1),
    (15, 10, 1), (14, 10, 1), (13, 10, 1), (12, 10, 1), (11, 10, 1),
    (10, 10, 1), (9, 10, 1), (8, 10, 1), (7, 10, 1), (6, 10, 1), (5,
    10, 1), (4, 10, 1), (3, 10, 1), (2, 10, 1), 'dropgball', (2, 11,
    1), (2, 12, 1), (2, 13, 1), (3, 13, 1), (4, 13, 1), (5, 13, 1),
    (5, 14, 1), (6, 14, 1), (6, 15, 1), (7, 15, 1), (8, 15, 1), (9,
    15, 1), 'pickrball', (9, 14, 1), (8, 14, 1), (7, 14, 1), '
    droprball', 'None', 'None']
the suffix of plan **actions**:
['None', 'None']
```

Example 2 Modified Our Algorithm Path

```
------------------------------
```

```
the prefix of plan **states**:
[((0, 0, 1), 'None'), ((1, 0, 1), 'None'), ((1, 1, 1), 'None'), ((2,
    1, 1), 'None'), ((2, 2, 1), 'None'), ((2, 3, 1), 'None'), ((3, 3,
    1), 'None'), ((4, 3, 1), 'None'), ((5, 3, 1), 'None'), ((5, 4, 1),
     'None'), ((6, 4, 1), 'None'), ((6, 5, 1), 'None'), ((7, 5, 1), '
    None'), ((7, 6, 1), 'None'), ((7, 7, 1), 'None'), ((8, 7, 1), '
    None'), ((9, 7, 1), 'None'), ((9, 8, 1), 'None'), ((9, 9, 1), '
    None'), ((9, 10, 1), 'None'), ((9, 11, 1), 'None'), ((9, 12, 1), '
    None'), ((9, 13, 1), 'None'), ((9, 14, 1), 'None'), ((9, 15, 1), '
    None'), ((9, 15, 1), 'pickrball'), ((9, 14, 1), 'None'), ((8, 14,
    1), 'None'), ((7, 14, 1), 'None'), ((7, 14, 1), 'droprball'), ((7,
     13, 1), 'None'), ((8, 13, 1), 'None'), ((8, 12, 1), 'None'), ((8,
     11, 1), 'None'), ((9, 11, 1), 'None'), ((9, 10, 1), 'None'), ((9,
     9, 1), 'None'), ((9, 8, 1), 'None'), ((10, 8, 1), 'None'), ((11,
    8, 1), 'None'), ((12, 8, 1), 'None'), ((13, 8, 1), 'None'), ((14,
    8, 1), 'None'), ((15, 8, 1), 'None'), ((16, 8, 1), 'None'), ((17,
    8, 1), 'None'), ((18, 8, 1), 'None'), ((19, 8, 1), 'None'), ((19,
    8, 1), 'pickgball'), ((18, 8, 1), 'None'), ((17, 8, 1), 'None'),
    ((17, 9, 1), 'None'), ((16, 9, 1), 'None'), ((16, 10, 1), 'None'),
     ((15, 10, 1), 'None'), ((14, 10, 1), 'None'), ((13, 10, 1), 'None
    '), ((12, 10, 1), 'None'), ((11, 10, 1), 'None'), ((10, 10, 1), '
    None'), ((9, 10, 1), 'None'), ((8, 10, 1), 'None'), ((7, 10, 1), '
    None'), ((6, 10, 1), 'None'), ((5, 10, 1), 'None'), ((4, 10, 1), '
    None'), ((3, 10, 1), 'None'), ((2, 10, 1), 'None'), ((2, 10, 1), '
    dropgball'), ((2, 10, 1), 'None')]
the suffix of plan **states**:
[((2, 10, 1), 'None'), ((2, 10, 1), 'None')]
------------------------------
the prefix of plan **actions**:
[(0, 0, 1), (1, 0, 1), (1, 1, 1), (2, 1, 1), (2, 2, 1), (2, 3, 1), (3,
     3, 1), (4, 3, 1), (5, 3, 1), (5, 4, 1), (6, 4, 1), (6, 5, 1), (7,
     5, 1), (7, 6, 1), (7, 7, 1), (8, 7, 1), (9, 7, 1), (9, 8, 1), (9,
     9, 1), (9, 10, 1), (9, 11, 1), (9, 12, 1), (9, 13, 1), (9, 14, 1)
    , (9, 15, 1), 'pickrball', (9, 14, 1), (8, 14, 1), (7, 14, 1), '
    droprball', (7, 13, 1), (8, 13, 1), (8, 12, 1), (8, 11, 1), (9,
    11, 1), (9, 10, 1), (9, 9, 1), (9, 8, 1), (10, 8, 1), (11, 8, 1),
    (12, 8, 1), (13, 8, 1), (14, 8, 1), (15, 8, 1), (16, 8, 1), (17,
    8, 1), (18, 8, 1), (19, 8, 1), 'pickgball', (18, 8, 1), (17, 8, 1)
    , (17, 9, 1), (16, 9, 1), (16, 10, 1), (15, 10, 1), (14, 10, 1),
    (13, 10, 1), (12, 10, 1), (11, 10, 1), (10, 10, 1), (9, 10, 1),
    (8, 10, 1), (7, 10, 1), (6, 10, 1), (5, 10, 1), (4, 10, 1), (3,
    10, 1), (2, 10, 1), 'dropgball', 'None', 'None']
the suffix of plan **actions**:
['None', 'None']
```

Bibliography

[1] Ltl 2 ba : fast translation from ltl formulae to büchi automata. `http://www.lsv.fr/~gastin/ltl2ba/index.php`. Accessed: 2017-05-03.

[2] Christel Baier, Joost-Pieter Katoen, and Kim Guldstrand Larsen. *Principles of model checking*. MIT press, 2008.

[3] Calin Belta, Antonio Bicchi, Magnus Egerstedt, Emilio Frazzoli, Eric Klavins, and George J Pappas. Symbolic planning and control of robot motion [grand challenges of robotics]. *IEEE Robotics & Automation Magazine*, 14(1):61–70, 2007.

[4] Calin Belta and LCGJM Habets. Constructing decidable hybrid systems with velocity bounds. In *Decision and Control, 2004. CDC. 43rd IEEE Conference on*, volume 1, pages 467–472. IEEE, 2004.

[5] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *International Conference on Computer Aided Verification*, pages 359–364. Springer, 2002.

[6] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[7] Edmund M Clarke and Jeannette M Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996.

[8] Thomas H.. Cormen, Charles Eric Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*, volume 6. MIT press Cambridge, 2001.

[9] Georgios E Fainekos, Antoine Girard, Hadas Kress-Gazit, and George J Pappas. Temporal logic motion planning for dynamic robots. *Automatica*, 45(2):343–352, 2009.

[10] Georgios E Fainekos, Hadas Kress-Gazit, and George J Pappas. Temporal logic motion planning for mobile robots. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pages 2020–2025. IEEE, 2005.

[11] Paul Gastin and Denis Oddoux. Fast ltl to büchi automata translation. In *International Conference on Computer Aided Verification*, pages 53–65. Springer, 2001.

[12] Dimitra Giannakopoulou and Flavio Lerda. From states to transitions: Improving translation of ltl formulae to büchi automata. *Formal Techniques for Networked and Distributed Sytems—FORTE 2002*, pages 308–326, 2002.

[13] Meng Guo. *Hybrid control of multi-robot systems under complex temporal tasks*. PhD thesis, KTH Royal Institute of Technology, 2015.

[14] Meng Guo. P-mas-tg. `https://github.com/MengGuo/P_MAS_TG`, 2015.

[15] Gerard Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, 2003.

[16] Hadas Kress-Gazit, Georgios E Fainekos, and George J Pappas. From structured english to robot motion. In *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*, pages 2717–2722. IEEE, 2007.

[17] Jan K Lenstra and AHG Rinnooy Kan. Some simple applications of the travelling salesman problem. *Journal of the Operational Research Society*, 26(4):717–733, 1975.

[18] Daniel J Rosenkrantz, Richard Edwin Stearns, and Philip M Lewis. Approximate algorithms for the traveling salesperson problem. In *Switching and Automata Theory, 1974., IEEE Conference Record of 15th Annual Symposium on*, pages 33–42. IEEE, 1974.

[19] Daniel A Schult and P Swart. Exploring network structure, dynamics, and function using networkx. In *Proceedings of the 7th Python in Science Conferences (SciPy 2008)*, volume 2008, pages 11–16, 2008.

[20] Viktor Schuppan and Armin Biere. Shortest counterexamples for symbolic model checking of ltl with past. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 493–509. Springer, 2005.

[21] Fabio Somenzi and Roderick Bloem. Efficient büchi automata from ltl formulae. In *International Conference on Computer Aided Verification*, pages 248–263. Springer, 2000.