# Thesis

Garrett Thomas

May 8, 2017

# Contents

# List of Figures

## 0.1 Basics

### 0.1.1 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 (Büchi automata, which will be defined later) that represents the robot motion. Next is the execution level, which involves finding a path through the graph that satisfies a specification. Lastly, in the implementation level robot controllers are constructed that satisfy the path found in the previous step.

We assume that we have one robot which is located in a given workspace denote $W_0 \subset \mathbb{R}^n$, which is bounded. To create a graph that represents the robot motion we need to consider the workspace along with the dynamics of the robot. 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 satisfy the bisimulation property, which will be defined later once more notation has been introduced [4]. We denote the $\Pi = \pi_1, \pi_2, \ldots, \pi_w$ to be the set of equivalence classes the workspace has been partitioned into, and thus $\cup_{i=1}^{w} \pi_i = W_0$ and $\pi_i \cap \pi_j = \emptyset$, $\forall i, j = 1, 2, \ldots, w$ and $i \neq j$. We will henceforth refer to equivalence class $\pi_i$ as region $\pi_i$ 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 $\pi_i$ [7].*

We now introduce atomic propositions, which will be the building blocks for our task specification. Atomic propositions are boolean variables, and will be used to express properties about the state or the robot or the workspace. We define the following set of atomic propositions $AP_r = a_{r,i}$, $i = 1, 2, \ldots, w$ 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 [9]. Other things we want to be able to express are potential tasks, denote $AP_p = \alpha_{p,i}$, $i = 1, 2, \ldots, m$. These can be statements such as "there is a ball in region $\pi_I$" or "the robot beeps" We now define the set of all propositions $AP = AP_r \cup AP_p$.

**Definition 2.** *The labelling function $L_C : W_0 \to 2^{AP}$ maps a point $x \in W_0$ to the set of atomic propositions satisfied by $x$ [9].*

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 $\pi_i$.*

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, $a_{r,i} \in L_D(\pi_i)$.

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 $0 \le t \le T$ with $x(0) = x$ and $x(T) = x'$ and we have $x(t) \in (T^{-1}(T(x)) \cup T^{-1}(T(x')))$ [6]*

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 $\pi_i$ to $\pi_j$, denoted $\pi_i \to_D \pi_j$ if for every $x$ in $\pi_i$ i.e. $T(x) = \pi_i$ there exists $x'$ such that $T(x') = \pi_j$ and $x \to_C x'$*

We can now define bisimulations

**Definition 6.** *A partition $T : W_0 \to \Pi$ is called a bisimulation [6] 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 $x \to_C x'$ then $y \to_C y'$ for some $y'$ with $T(x') = T(y')$*

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

We can now define Finite-State Transition System (FTS), which is how we will represent our workspace.

**Definition 7.** *An FTS is a tuple* $\mathcal{T}_C = (\Pi, \rightarrow_D, \Pi_0, AP, L_C)$ *where* $\Pi$ *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* $\pi_i$ *to* $\pi_j$ *as defined in definition 6. In adherence to common notation, we will write* $\pi_i \rightarrow_D \pi_j$*. Note:* $\pi_i \rightarrow_D \pi_i, \ \forall 1, 2, \ldots w$*.* $\Pi_0 \subseteq \Pi$ *is the initial state(s),* $AP = AP_r \cup AP_p$ *is the set of atomic propositions, and* $L_C : \Pi \rightarrow 2^{AP}$ *is the labelling function defined in definition 3.*

In this thesis, we will also consider the weighted FTS (WFTS)

**Definition 8.** *A WTFS is a tuple* $\mathcal{T}_C = (\Pi, \rightarrow_C, \Pi_0, AP, L_C, W_C)$ *where* $\Pi$*,* $\rightarrow_C$*,* $\Pi_0$*,* $AP$*, and* $L_C$ *are defined as in definition 8 and* $W_C : \Pi \times \Pi \rightarrow \mathbb{R}^+$ *is the weight function i.e. the cost of a transition in* $\rightarrow_C$*.*

Note: Any FTS can be written can be converted to an WFTS with the weights of all transitions equalling one.

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. When talking about FTS, it can be helpful to use the notation $\texttt{Pre}(\pi_i) = \{\pi_j \in \Pi | \pi_j \rightarrow_D \pi_i\}$ to define the the predecessors of the state $\pi_i$ and $\texttt{Post}(\pi_i) = \{\pi_j \in \Pi | \pi_j \rightarrow_D \pi_i\}$ to define the the successors of the state $\pi_i$. In this thesis, we will deal with infinite paths. An infinite path is an infinite sequence of states $\tau = \pi_1 \pi_2 \ldots$ such that $\pi_i \in \Pi_0$ and $\pi_i \in \text{Post}(\pi_{i-1}) \ \forall i > 0$. The trace of a path is the sequence of sets of atomic propositions that are true in the states along a path i.e. $\text{trace}(\tau) = L_D(\pi_1) L_D(\pi_2) \ldots$.

### 0.1.2 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 (conjunction, disjunction, and negation), but also temporal specifications e.g. $\alpha$ is true at some point of time. The particular temporal logic we will be using is known as linear temporal logic (LTL) [5]. LTL formulas are defined over a set of atomic propositions $AP$ according to the following grammar:

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

where $\top$ is the predicate true, $\alpha \in AP$ is an atomic proposition, $\varphi_1$ and $\varphi_2$ are LTL fomulas, $\neg$ and $\vee$ denote the standard Boolean connectives negation and disjunction respectively, X being the "Next" operator. $\boldsymbol{\mathcal{U}}$ is the temporal operator "Until", with $\varphi_1 \mathcal{U} \varphi_2$ meaning $\varphi_1$ is true until $\varphi_2$ 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, $\perp = \neg\top$. We are also able to derive the following additional temporal operators:

$$\begin{aligned} \text{Eventuality: } & \diamond \varphi_1 = TU\varphi_1 \\ \text{Always: } & \square\varphi_1 = \neg \diamond \neg\varphi_1 \end{aligned}$$

There is a growing interest in path and mission planning in robots using temporal logic specifications given the easy extension from natural language to temporal logic [12]. We now give examples to illustrate this point and to introduce us to LTL formulas. First, the atomic operators generally capture properties of the robot or the environment i.e. "the robot is in region 1", "the ball is in region 2", "the robot is holding a ball". There are some common tasks converted to LTL formulas given in [6]

1. **Reachability while avoiding regions**: "Go to region $\pi_{n+1}$ 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 $\pi_1, \pi_2, \pi_3$ 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 [9]: "Pick up the red ball, drop it to one of the baskets and

then stay in room one"
$\diamond(rball \wedge \diamond basket) \wedge \diamond\Box r1$

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 $\omega$ superscript means an infinte repatition; 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 $k$ [9]. An infinite word $\sigma$ satisfies an LTL formula $\varphi$ based on the LTL semantics.

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

$$\begin{aligned}
(\sigma, k) &\models \alpha & &\textit{if} \quad \alpha \in S_k \\
(\sigma, k) &\models \neg\varphi & &\textit{if} \quad (\sigma, k) \not\models \varphi \\
(\sigma, k) &\models \boldsymbol{X}\varphi & &\textit{if} \quad (\sigma, k+1) \models \varphi \\
(\sigma, k) &\models \varphi_1 \vee \varphi_2 & &\textit{if} \quad (\sigma, k) \models \varphi_1 \textit{ or } (\sigma, k) \models \varphi_2 \\
(\sigma, k) &\models \varphi_1 \mathcal{U} \varphi_2 & &\textit{if} \quad \exists k' \in [k, +\inf],\ (\sigma, k') \models \varphi_2 \textit{ and} \\
& & &\phantom{\textit{if}} \quad \forall k'' \in (k, k'),\ (\sigma, k'') \models \varphi_1
\end{aligned}$$

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

There is a connection between these infinite words and the FTS described earlier that is crucial in motion planning technique. Given an infinite path $\tau$ 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 $\tau$ *satisfies* $\varphi$ if its trace satisfies $\varphi$, i.e. $\tau \models \varphi$ if $trace(\tau) \models \varphi$. A path satisfying $\varphi$ will be referred to as a *plan* for $\varphi$.

### 0.1.3 Büchi Automata

We now know 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 a significantly more amout of work! 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 five-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 or states, $r = q_0 q_1 \dots$, where 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 $k = 0, 1, \dots$. The requirements for a run $r$ to be accepting is $\mathtt{Inf}(r) \cap \mathcal{F} \neq \emptyset$, where $\mathtt{Inf}(r)$ is the set of states that appear in $r$ infinitely often [9].

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

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

### 0.1.4 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 $\mathcal{A}_p = \mathcal{T} \otimes \mathcal{A}_\varphi = (Q', \delta', Q'_0, \mathcal{F}', W_p)$, 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: $W_p(\langle \pi_i, q_m\rangle, \langle \pi_j, q_n\rangle) = W_c(\pi_i, \pi_j)$, 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 $\Pi$ is denoted by $q'|_\Pi = \pi$ and its projection on $Q$ is denoted by $q'|_Q = q$. Given an infinite run $R = q'_0 q'_1 q'_2 \dots$ of $\mathcal{A}_p$, its projection on $\Pi$ is denoted by $R|_\Pi = q'_0|_\Pi q'_1|_\Pi q'_2|_\Pi \dots$ and its projection on $Q$ is denoted by $R|_Q = q'_0|_Q q'_1|_Q q'_2|_Q \dots$ [9].

Given that $\mathcal{A}_p$ is a Büchi automaton, the requirements of an accepting run, r, is the same as before i.e. $\mathtt{Inf}(r) \cap \mathcal{F}' \neq \emptyset$

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 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, $\tau_{pre}$, is the path from an initial node to an accepting node. The suffix, $\tau_{suf}$, 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 $\omega$ superscript). Thus, the accepting node appears infinitely many times in $\tau$ which makes $\tau$ accepting. Plans of this form are much easier to deal with because, while they are still infinite plans, they have a finite representation which is easier to deal with.

### 0.1.5 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

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

where $q'_0 = \langle \pi_0, q_0 \rangle \in \mathcal{Q}'_0$ and $q'_f = \langle \pi_f, q_f \rangle \in \mathcal{F}'$.

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 $W_p(q'_i, q'_{i+1}) = W_c(q'_i|_\Pi, q'_{i+1}|_\Pi)$. 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.

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

where $\gamma \geq 0$ is the relative weighting of the transient response (prefix) cost and steady response (suffix) cost [9]. In [6] 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 [16] we see that simplifications in the translation from LTL formulas to NBA can result in a loss of optimality. This will come up again when we analyse the paths our algorithm generates.

## 0.2 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 [16]. 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.

### 0.2.1 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, $\tau$, 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 $q_f \in \texttt{Inf}\tau \Rightarrow \texttt{Inf}\tau \cap \mathcal{F} \neq \emptyset$. This algorithm, or simple variations of it, are used in many works on motion planning synthesis [6], add more, so we will refer to it as the *accepted* algorithm.

Algorithm 1, from [9], gives pseudocode of how to compute $R_{opt}$.

**Procedure 1** OptRun()

**Input:** Input $\mathcal{A}_p, S' = \mathcal{Q}'_0$ by default
**Output:** $R_{opt}$

1: For each accepting state $q'_f \in \mathcal{F}'$, calculate the optimal path back to $q'_f$.
2: For initial state $q'_0 \in S'$, find the optimal path to each $q'_f \in \mathcal{F}$.
3: Find the pair of $(q'_{0,opt}, q'_{f,opt})$ 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.

Meng Guo has created a public github repository, P-MAS-TG (Planner for Multiple Agent System with Temporal Goals) [10]. The function `dijkstra_plan_networkX` in `P_MAS_TG`
`discrete_plan.py` is approximately equvialent to Algorithm 1. The work of 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

[15]. `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 $\mathcal{O}(|\delta'| \cdot \log|\mathcal{Q}'| \cdot (|\mathcal{Q}'_0| + |\mathcal{F}'|))$ because the worst case complexity for a Dijkstra search is $\mathcal{O}(|\delta'| \cdot \log|\mathcal{Q}'|)$ and Algorithm 1 does $(|\mathcal{Q}'_0| + |\mathcal{F}'|)$ Dijkstra searches (one for each initial node and one for each accepting node).

### 0.2.2 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 [5]. 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. $|\Pi||\mathcal{Q}|$. The size of the Büchi automaton corresponding to the LTL formula is then usually exponential in the size of the formula. 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 an algorithm that sacrifices optimality but preforms much faster than the current accepted algorithm.

The idea stems from the fact that $q' = \langle \pi, q \rangle \in \mathcal{Q}'$ is an accepting state of $\mathcal{A}_p$ iff $q \in Q$ 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* $\mathcal{Q}, 2^A P, \delta, \mathcal{Q}_0, \mathcal{F}$ *are defined as in definition 10 and* $d : \mathcal{Q} \to \mathbb{Z}$ *is defined as*

$$d(q_n) = \min_x \{x | q_x \in \mathcal{F} \text{ and } q_k \in \delta(q_{k-1}, S_{k-1}) \text{ for some } S_k \in 2^{AP} \text{ and } k = 0, 1, \ldots, x-1\}$$

*which is the length of 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 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 use a either a Dijkstra search or a decreasing levels search to find the fastest way from the accepting state back to itself. Sometimes we have to use a Dijkstra search instead of use the idea of decreasing levels because, although it would be faster, in general this procedure cannot be use to find a specific accepting state. We will refer to the run generated by this algorithm as $R_{nn}$ in which $nn$ stands for nearest neighbour. We choose this name because in some situations this search is equivalent to the nearest neighbour search algorithm for the travelling salesperson problem [11]. Pseudocode is given in Algorithm 2

**Procedure 2** NearestNeighborRun()

**Input:** Input $\mathcal{A}_{p,d}, S' = \mathcal{Q}'_0$ by default
**Output:** $R_{nn}$

Level = $d_p(q'_0 \in \mathcal{Q}'_0)$, Prefix = $[q'_0]$
**while** Level $>$ 0 **do**
    find the closest node, NextNode, that is on level one less than Level
    add path to NextNode onto Path
    **if** $d_p$(NextNode) == 0 **then**
        break
    Level = Level - 1
Suffix = use Dijkstra search to find the optimal path from NexNode back to itself
$R_{nn}$, prefix + suffix.

Algorithm 2 is equivalent to the function `Garrett_search` which is provided in the appendix. This code was based on `dijkstra_plan_networkX` from [10] and still shares some of the structure. Finding the closest node on the level below the current, i.e. NextNode is done using the function `adapted_dijkstra_multisource` which is also included in the appendix. This code was based on the function `_dijkstra_multisource` in [15]. When we provide computation runs of our algorithm in the following text, we will be referring to runs done with this algorithm.

As we can see, assuming that we reach an accepting state in a strongly connected component, we will do $n+1$ searches. This still may seem like a lot, however the searches are done on much smaller graphs. The first $n$ searches only look at graphs with $|\Pi|$ nodes. These smaller graphs have a number of edges less than or equal to $|\delta|$ i.e. the number of edges $\mathcal{T}$ has. This is because $\langle \pi_j, q_n \rangle \in \delta'(\langle \pi_i, q_m \rangle)$ iff $(\pi_i, \pi_j) \in \rightarrow_c$ and $q_n \in \delta(q_m, L_c(pi))$, which implies the number of edges on one level is less than or equal to $|\delta|$. Therefore our worst case complexity will be $\mathcal{O}(|\delta| \cdot \log|\mathcal{T}| \cdot n) + \mathcal{O}(|\delta'| \cdot \log|\mathcal{Q}'|) = \mathcal{O}(|\delta| \cdot \log|\mathcal{T}| \cdot n + |\delta'| \cdot \log|\mathcal{Q}'|)$ where $n$ is the level of the initial node. This complexity applies in the situation that the accepting node we find has a path back to itself and that we do not have transfers on the same level of the Büchi automaton i.e. if there is a transfer from $q_i$ to $q_{i+1} \rightarrow d(q_i) \neq d(q_{i+1})$. However, when we get to examples in the complex formulas chapter this is not the case. If this distance requirement is not fulfilled, the worst case complexity of our algorithm is the same as the worst case complexity of the accepted algorithm i.e. $\mathcal{O}(|\delta'| \cdot \log|\mathcal{Q}'|)$.

We now analyse how this algorithm performs in when the LTL formula expresses certain behaviours.

## 0.3 Algorithm Performance with Specific Behaviours

To show how our algorithm differs with the current accepted algorithm, we illustrate examples using the FTS in figure 1

This FTS will be used in figures unless otherwise stated. We chose it to be very simple because the state explosion problem applies even to this report. If we chose a more complex FTS we would not be able to include illustrations of the product automaton because it gets very big very quickly. We will henceforth refer to this FTS as simple FTS. When providing computational results, we will use an FTS that is much larger, to bring out the difference between our algorithm and the accepted algorithm.

### 0.3.1 Reachability while avoiding regions

Reachability while avoiding regions is a property in which we wish to not cross over certain areas, say $\pi_1, \pi_2, \ldots, \pi_n$, until we get to a specified region, say $\pi_{n+1}$. After reaching $\pi_{n+1}$ we are free to do what we want. This behaviour is expressed by the 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 2

As we can see $d_p(q_1) = 1$ and $d_p(q_2) = 0$. For our example, we will look at the specific formula $\neg\pi_4\boldsymbol{\mathcal{U}}\pi_5$ The product automaton is shown in figure 3

Note: in figure 3 all nodes have a self loop, which are not included for the sake of the reader. Our algorithm does $n+1$ Dijkstra searches where $n$ is the maximum level of a state in the Büchi automaton. As we can see in 2, which is the Büchi automaton corresponding to the general from of reachability while avoiding regions, $n$ is 1 for all formulas of this form. Therefore our algorithm does one Dijkstra search starting from $(\pi_1, q_1)$ which ends at $(\pi_5, q_2)$. This is exactly what the accepted algorithm does, so we do not gain anything when using our algorithm to find an accepting node. However, we do gain a much faster run time because we do not have to do a Dijkstra search for every accepting node. We simply do one, and because there $q_2$ in automaton 3 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.

The search of a loop from every accepting node back to itself requires a lot of time. We can illustrate this in an example. We have a workspace partitioned into $25 \times 25 = 625$ equally sized squares

The output from the accepted algorithm is

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

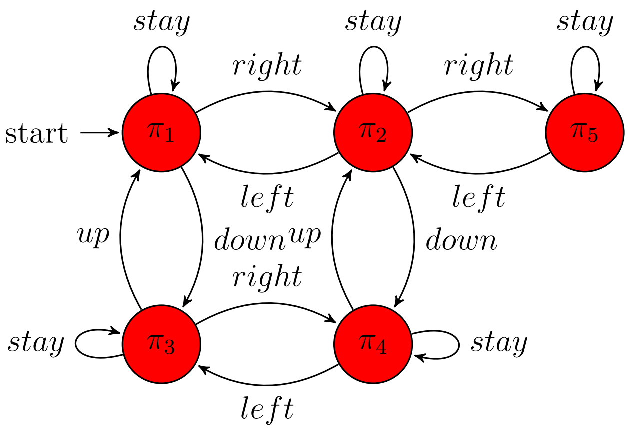



Figure 1: Simple Finite Transition System

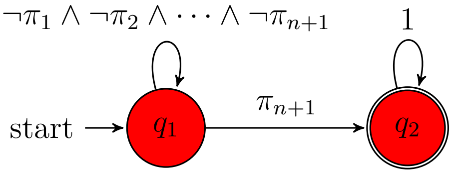



Figure 2: Büchi automaton corresponding to $\neg(\pi_1 \vee \pi_2 \vee \ldots \pi_n)\boldsymbol{\mathcal{U}}\pi_{n+1}$

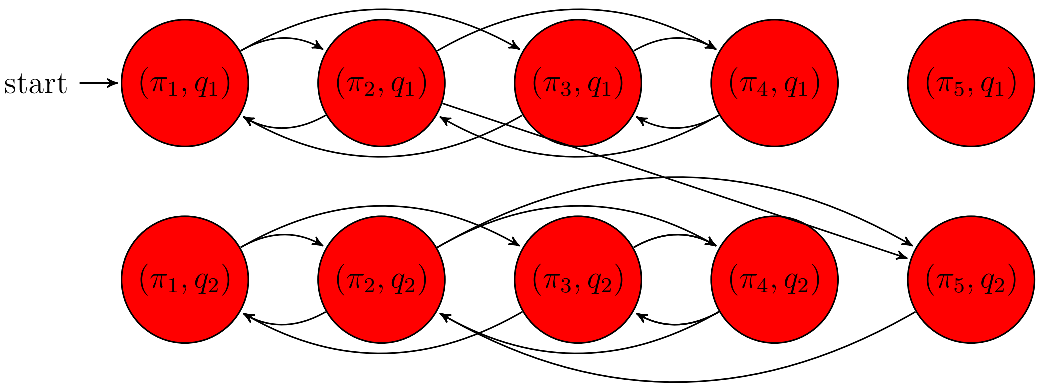



Figure 3: Product Automaton for $\neg\pi_4\boldsymbol{\mathcal{U}}\pi_5$ with Simple FTS

```
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'), ((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'), ((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'), ((20, 17, 1)
   , 'None'), ((20, 17, 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), (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),
   (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), 'None', 'None
   ']
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 time for `full construction and synthesis` is larger than the first time given because it includes the time taken to initialize and construct the graph, which takes the majority of time.

The output of our algorithm is as follows

```
==================
Dijkstra_plan_networkX done within 0.01s: 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)]
```

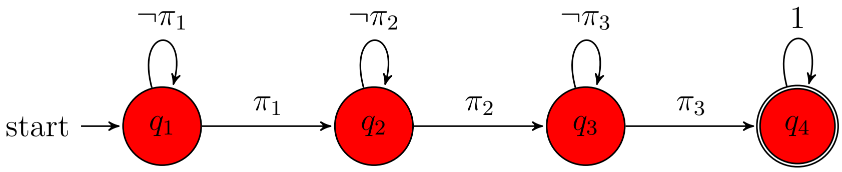



Figure 4: Büchi Automaton Corresponding to $\diamond(\pi_1 \wedge \diamond(\pi_2 \wedge \diamond\pi_3))$

```
the suffix of plan **actions**:
['None', 'None']
full construction and synthesis done within 0.32s
```

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

### 0.3.2 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 that describes this behaviour for $n = 3$ is $\diamond(\pi_1 \wedge \diamond(\pi_2 \wedge \diamond\pi_3))$ and is shown in figure 4. We note that this is automaton is only applicable if because of the partition we defined earlier in Definition 1 which makes it impossible for $\pi_i$ and $\pi_j$ to be true at the same time if $i \neq j$. The LTL2BA tool [1] that is used the github repository [10] actually generates an automaton with connecting every node. For example, there is an edge from $q_1$ to $q_4$ labelled $\pi_1 \&\& \pi_2 \&\& \pi_3$. 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 in an example using the formula $\diamond(\pi_2 \wedge \diamond\pi_5))$ the simple FTS as before. The product automaton is shown in figure 5

Our algorithm finds $(\pi_4, q_2)$, then starts another Dijkstra search and finds $(\pi_5, q_3)$. Will search through extraneous nodes, for example $(\pi_5, q_1)$. This may not seem like a lot in this example, but when we expand to larger problems the difference becomes significant.

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 regions. 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 $\pi_1$ is located at (2,24), region $\pi_2$ is located at (12,12), and region $\pi_3$ is located at (20,15) . See figure 0.3.2

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

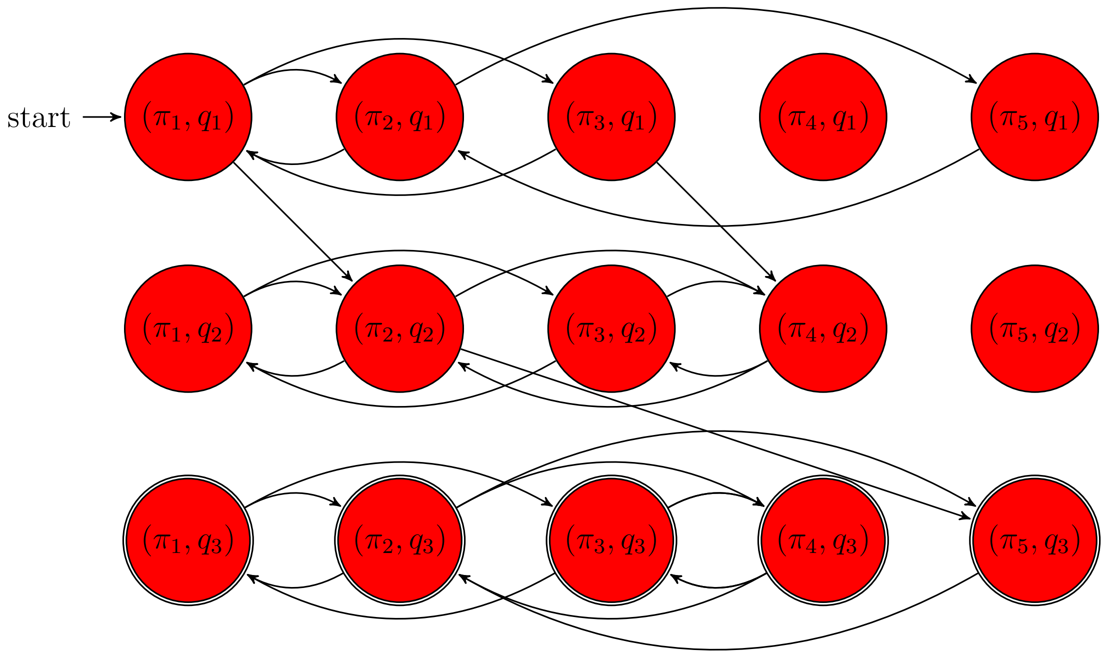
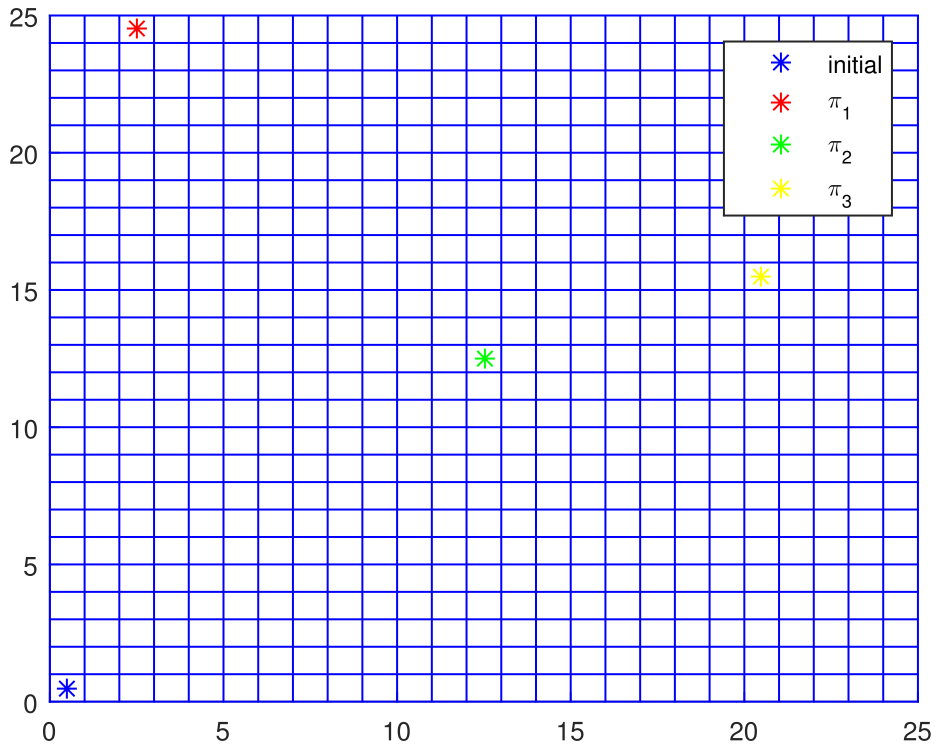



Figure 5: Product Automaton for $\diamond(\pi_2 \wedge \diamond\pi_5))$ with Simple FTS




```
Accepted Algorithm
===================
Dijkstra_plan_networkX done within 0.04s: precost
    62.00, sufcost 0.00
-------------------------------
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']
full construction and synthesis done within 0.19s
```

and the output from our algorithm is

```
==================
Dijkstra_plan_networkX done within 0.02s: precost
   62.00, sufcost 0.00
------------------------------
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')]
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)]
the suffix of plan **actions**:
['None', 'None']
full construction and synthesis done within 0.17s
```

as we can see, our algorithm returns the same path as the accepted algorithm, and 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.

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.

### 0.3.3 Coverage

A coverage formula represents the statement visit $\pi_1, \pi_2, \ldots, \pi_n$ in that order, and is of the form $\varphi = \diamond\pi_1 \wedge \diamond\pi_2 \wedge \cdots \wedge \pi_n$. We show the Büchi automaton corresponding to the formula $\diamond\pi_1 \wedge \diamond\pi_2 \wedge \pi_3$ in figure 6

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

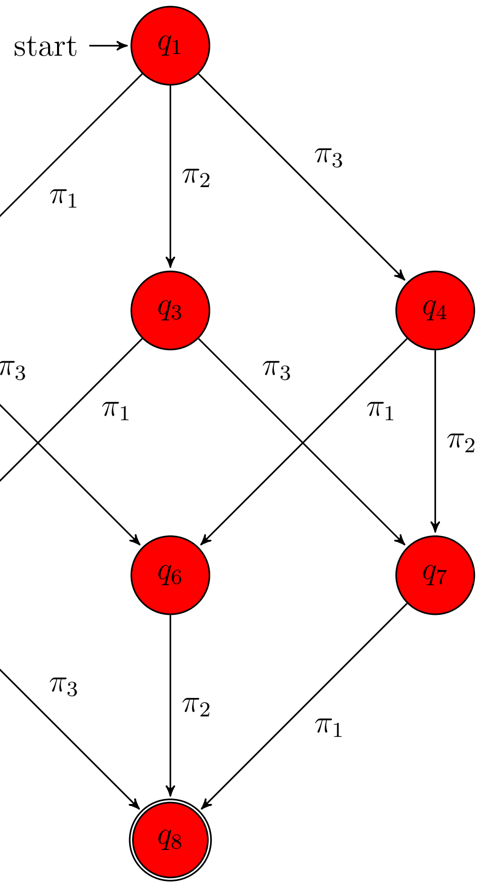



Figure 6: Büchi Automaton Corresponding to $\diamond\pi_1 \land \diamond\pi_2 \land \pi_3$

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 $\pi_i$ which is the closest to it. From then, it will pick the next closest $\pi_j$ 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, and we now show that this path corresponds to the one generated by the nearest neighbour approach to the travelling salesperson problem. We also provide a bound on the distance of our path based on the worst case ratio of the nearest neighbour tour to the optimal tour given by Rosendrantz, Stearns, and Lewis [14]. 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 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 [13]. 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 $\pi_1$, $\pi_2$ and $\pi_3$, 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

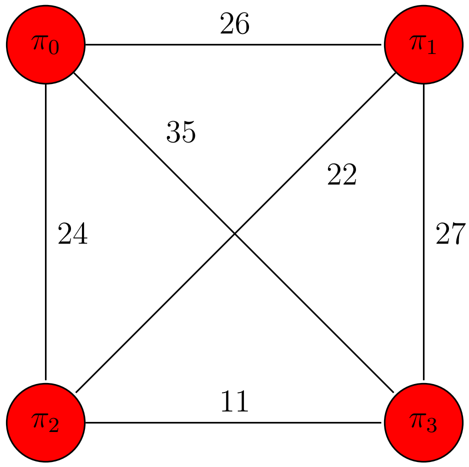



Figure 7: Complete Graph between Regions of Interest

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 where the initial state is from now on known as $\pi_0$, to be the shortest path our robot can take from $\pi_i$ to $\pi_j$, 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 7. 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*, 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 8. The cost of this path is 62.

This is not the optimal path though, which is shown in figure 9 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

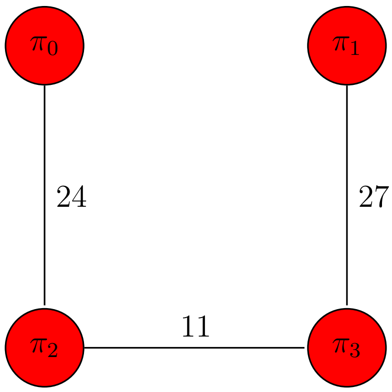



Figure 8: Nearest Neighbour Path

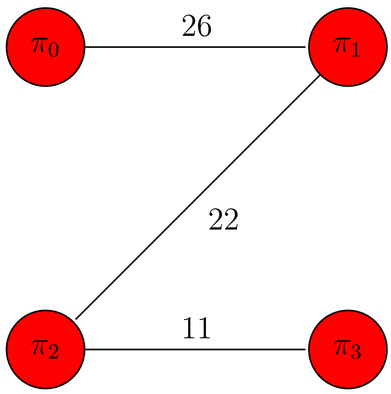



Figure 9: Optimal Path

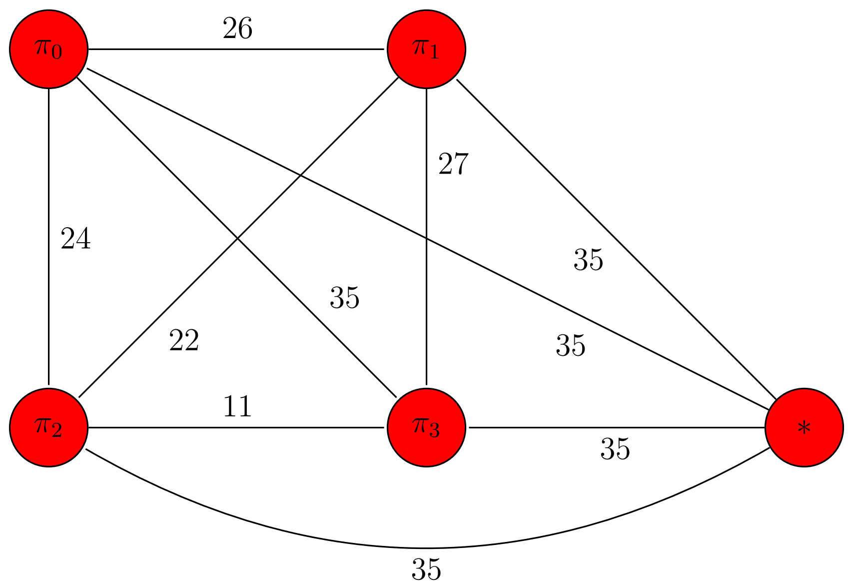



Figure 10: Complete Subgraph with Dummy Node

to be $max_{i,j}c_{ij}$ where $c_{ij}$ is the distance between the nodes in the complete subgraph in figure 7. In our case, this is 35, the path between $\pi_0$ and $\pi_3$. 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 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 $\pi_j$ or to go to the dummy node, and we can choose to go to $\pi_j$ 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 10

The path that the nearest neighbour algorithm takes in this situation, the complete Hamiltonian circuit, is given in figure 11, which gives a total cost of 132.

We note however that this is not the optimal solution. This optimal solution is shown in figure 12 and has a cost of 129.

It has been shown [14] 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 $j$,

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

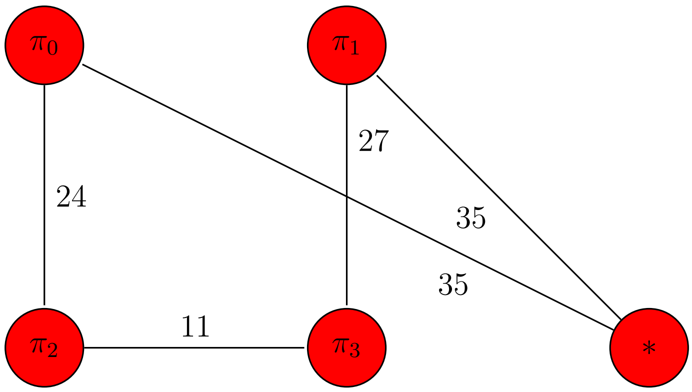



Figure 11: Nearest Neighbour Path with Dummy Node

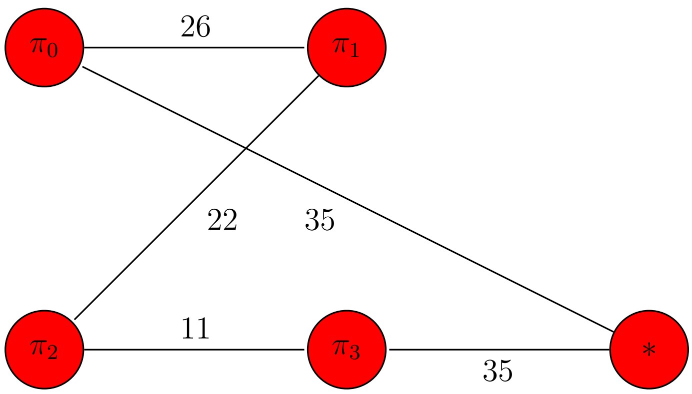



Figure 12: Optimal Path with Dummy Node

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

Our values do indeed satisfy this inequality

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

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

We provide a proof of

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

which can be found in [14]. Proof: We begin by proving

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

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

$$\text{OPTIMAL} \geq \text{LENGTH} \tag{3}$$

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 (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

$$d(a_i, a_j) \geq \min(l_i, l_j) \tag{4}$$

Summing (4) over the edges of $T$, we get

$$\text{LENGTH} \geq \sum_{(a_i,a_j) \text{ in } T} \min(l_i, l_j) \tag{5}$$

If we let $\alpha_i$ 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

$$\sum_{(a_i,a_j) \text{ in } T} \min(l_i, l_j) = \sum_{a_i \text{ in } H} a_i l_i \tag{6}$$

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),

$$\sum_{a_i \text{ in } H} \alpha_i = \min(2k, n) \tag{7}$$

To get a lower bound on (6) we assume that $\alpha_i = 2$ for $k + 1 \leq i \leq \min(2k, n)$ and is zero of $i \leq k$. Thus,

$$\sum_{a_i \text{ in } H} a_i l_i \geq 2 \sum_{i=k+1}^{\min(2k,n)} l_i \tag{8}$$

Combining (3), (5), (6), and (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 (2).

We now sum (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

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

Therefore we can write

$$\lceil\log(n)\rceil \cdot \text{OPTIMAL} \geq 2 \sum_{i=2}^{n} l_i \tag{9}$$

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. OPTIMAL $\geq 2l_i$ for $i = 1, 2, \ldots, n$. Specifically,

$$\text{OPTIMAL} \geq 2ł_1 \tag{10}$$

Summing (9) and (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 $\pi_i$ to $\pi_j$ 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 $\pi_j$ 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 AC-

CEPT

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

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.

The actual output from the accepted algorithm is

```
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
```

and our algorithm is

```
=====================
Dijkstra_plan_networkX 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
```

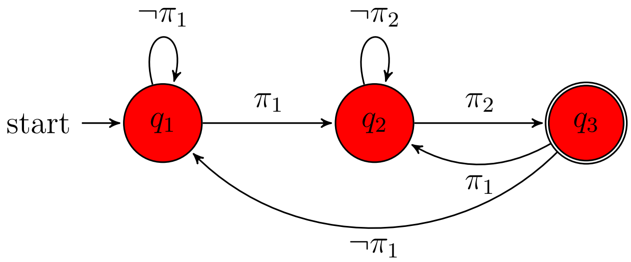



Figure 13: Büchi Automaton for $\square(\diamond\pi_1 \wedge \diamond\pi_2)$ 1

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 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.

### 0.3.4 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, and it an accepting path for it does 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.

To illustrate our point, 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 [8] is given in figure 13

Note: The actual automaton generated has much more edges. For example, there is an edge from $q_4$ to $q_2$ which is labelled $\pi_1\&\&\pi_2$. It is impossible for us to make this transition because $\pi_i$ for all $i$ is a region in our partition. This is because the requirements of our partition are chosen specially to guarantee that we are never in two regions at once. Thus they are excluded in the interest of the reader. 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 13 the only way to do this is to go to region $\pi_1$. Our algorithm does this, and then starts a new Dijkstra search.

In this case the same statement holds for $\pi_2$. Therefore the optimal prefix is to concatenate the optimal paths down from each each level (first to $\pi_1$, etc). 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 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']
```

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

```
Dijkstra_plan_networkX 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 anymore, so each of these searches 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.

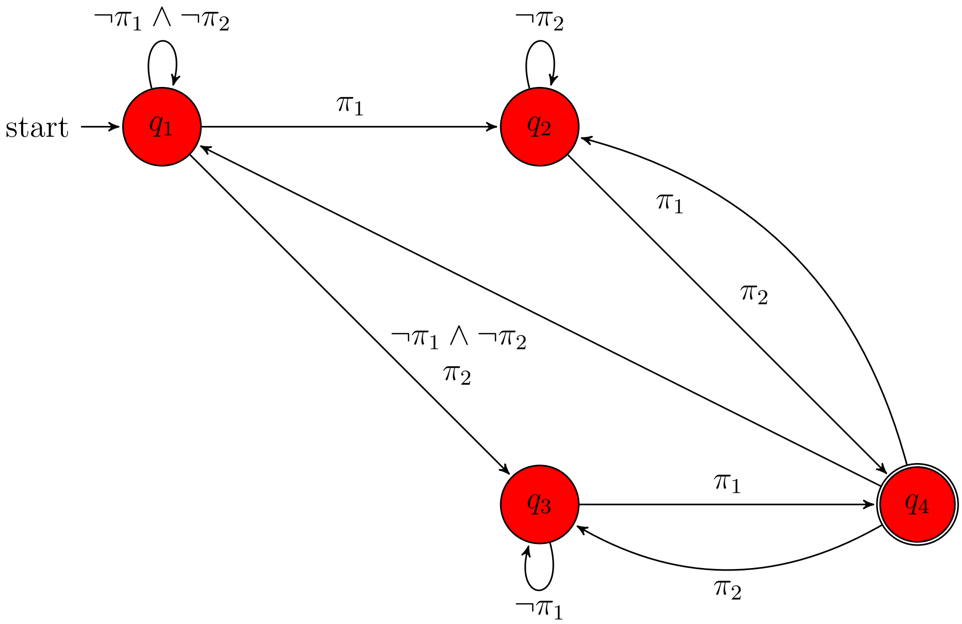



Figure 14: Büchi Automaton for $\square(\diamond\pi_1 \wedge \diamond\pi_2)$ 2

This path however is in general not truly optimal. It is because the Büchi automaton given in figure 13 not a tight Büchi automaton [16]. 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 [8] 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 14.

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 $\pi_1$ or $\pi_2$. 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.

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, $\pi_1$, $\pi_2$, and $\pi_3$ 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, $\mathtt{Inf}(R) \cap \mathcal{F}$ must not be empty. We are specifically looking for runs of the form

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

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; however in certain circumstances it can find a specific accepting state. We illustrate this using the same examples above.

$\Box(\Diamond\pi_3 \wedge \pi_5)$ We notice how in figure 13 there is only one arrow to the accepting state, labelled $\pi_5$. This implies that the only way to get down to level 0 is to go to $\pi_5$, and thus go to the accepting state $\langle \pi_5, q_3 \rangle$. There is no self loop on $q_3$, so we leave $q_3$ immediately. This implies that the only reachable accepting state is $\langle \pi_5, q_3 \rangle$. So because there is only one accepting state, our algorithm will find this state again, and thus is appropriate for finding the suffix.

In 14 on the other hand, there are two arrows going to the accepting state and there is no self loop. This implies that there are two reachable accepting states i.e. $\langle \pi_3, q_4 \rangle$ and $\langle \pi_5, q_4 \rangle$. This poses a problem to our algorithm that is only guaranteed to reach an accepting state. We thus propose using Dijkstra's search algorithm to find the path from the accepting node back to itself.

## 0.4 More Complex Formulas

The formulas in the previous section are common formulas, but are fairly simple. 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.

### 0.4.1 Example 1

We look at the example from [9] 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 $\varphi = \diamond(\text{rball} \wedge \diamond\text{basket}) \wedge \diamond\Box r1$. The Büchi automaton corresponding to this formula as translated by [8] is shown in figure 0.4.1

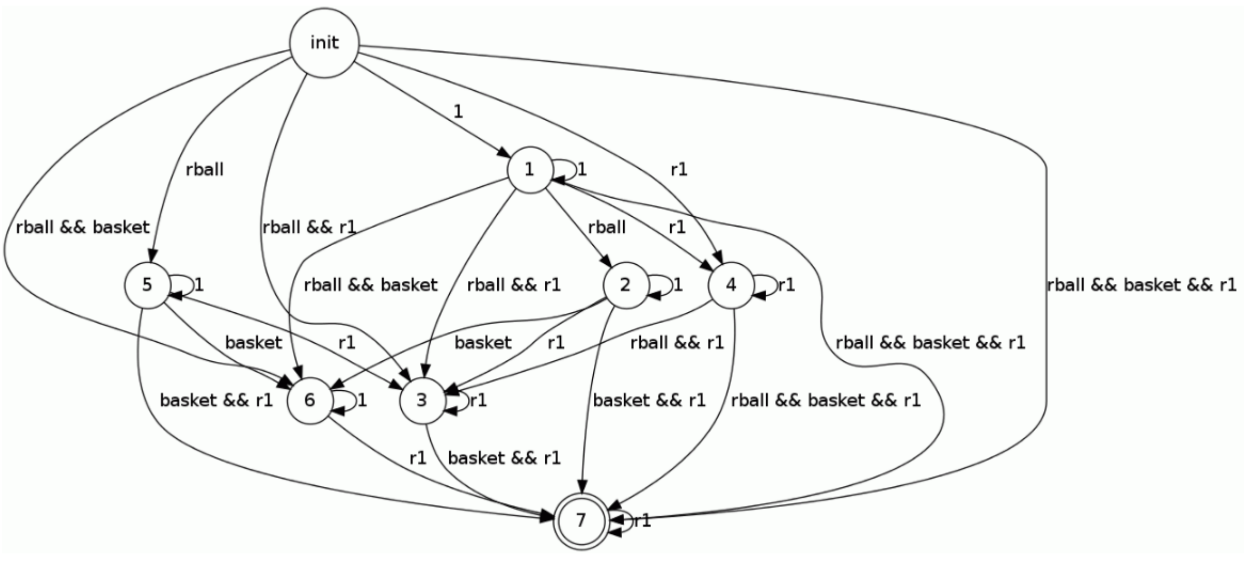



Figure 15: Büchi Automaton Corresponding to $\varphi = \diamond(\text{rball} \wedge \diamond\text{basket}) \wedge \diamond\Box r1$

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

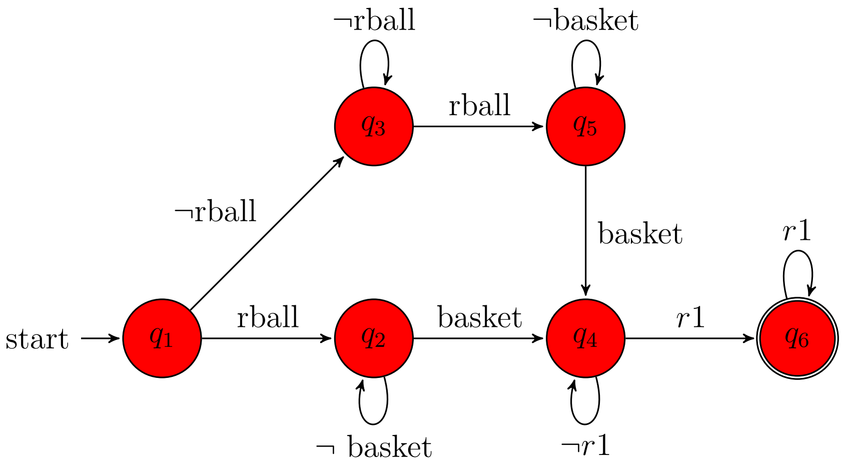



Figure 16: Simplified Büchi Automaton for $\varphi = \diamond(\text{rball} \wedge \diamond\text{basket}) \wedge \diamond\Box r1$ 1

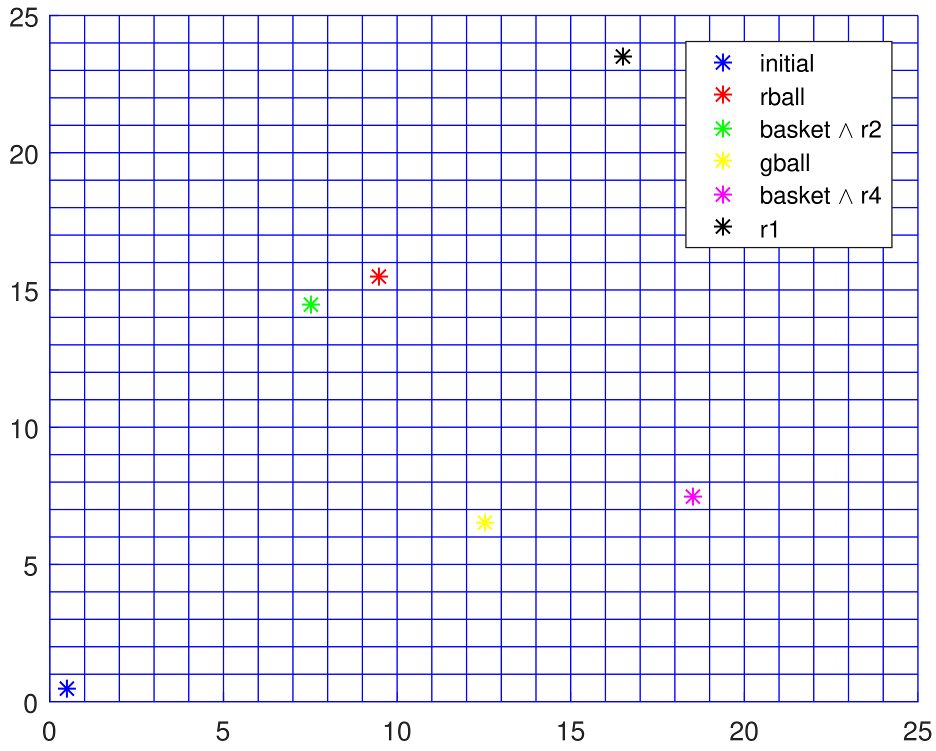
basket, and the neither the ball or basket is located in room 1) these edge are impossible to take. Therefore we remove these edges from the automaton (show the code for this). We then have a much simpler automaton that is shown in figure 16

In this automaton, we can see that $d(q_1) = 3$, $d(q_2) = 2$, $d(q_3) = 3$, $d(q_4) = 1$, $d(q_5) = 2$, $d(q_6) = 0$. 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 ¬rball. Examining the automaton in figure 16 that then 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 of $q_2$. 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 much faster.

The path that both algorithms calculate is

```
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, 16, 1), 'None'), ((9, 16, 1), '
pick'), ((8, 16, 1), 'None'), ((8, 15, 1), 'None'),
((8, 15, 1), 'drop'), ((8, 16, 1), 'None'), ((9, 16,
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')]
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), (9, 16, 1), 'pick', (8, 16, 1),
    (8, 15, 1), 'drop', (8, 16, 1), (9, 16, 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)]
the suffix of plan **actions**:
['None', 'None']
```

Our algorithm gives

```
Dijkstra_plan_networkX done within 0.02s: precost
    44.00, sufcost 0.00
```

while the accepted algorithm gives

```
Dijkstra_plan_networkX done within 0.10s: precost
    44.00, sufcost 0.00
```

### 0.4.2 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. We show multiple scenarios. First, if rball is with the basket. If this is the case, then rball and basket can be satisfied simultaneously. Therefore we have to admit paths with rball && basket into the automaton. The automaton is now

In this automaton, we now have $d(q_1) = 2$, $d(q_2) = 2$, $d(q_3) = 2$, $d(q_4) = 1$, $d(q_5) = 2$ 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 rball or rball && basket. This means that we cannot ever go to $q_2$.

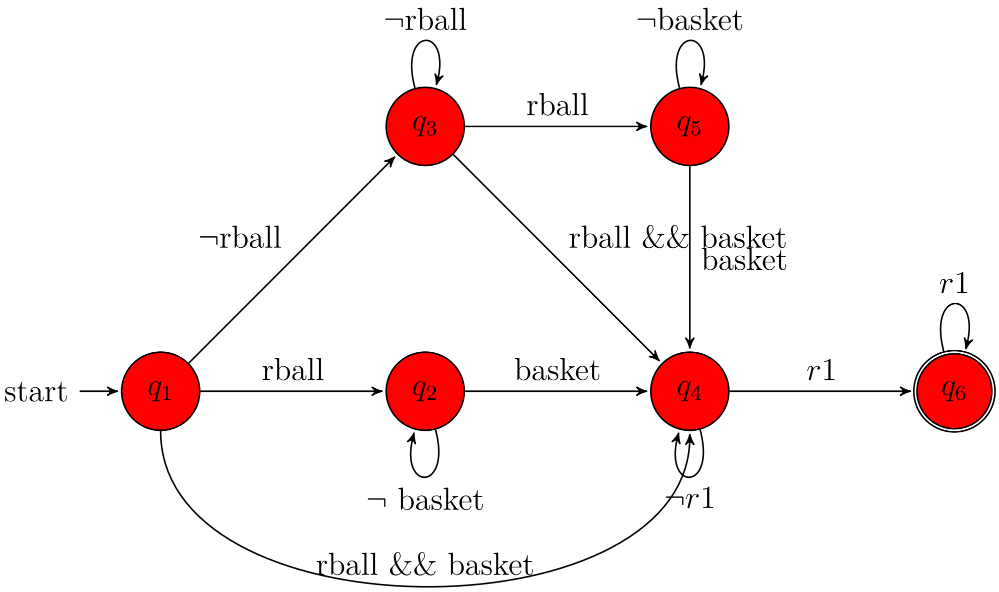



Figure 17: Simplified Büchi Automaton Corresponding to $\varphi = \diamond(\text{rball} \wedge \diamond\text{basket}) \wedge \diamond\Box r1$ 2

### 0.4.3 Example 2

We now look at the example taken from [9] 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 $\varphi = \diamond(\text{rball} \wedge \diamond(\text{basket} \wedge \text{r2})) \wedge \diamond(\text{gball} \wedge \diamond(\text{basket} \wedge \text{r4})) \wedge \Box(\text{rball} \rightarrow \mathbf{X}(\neg\text{gball}\boldsymbol{\mathcal{U}}\text{basket})) \wedge \Box(\text{gball} \rightarrow \mathbf{X}(\neg\text{rball}\boldsymbol{\mathcal{U}}\text{basket}))\&\& \diamond\Box r1$. 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 56 states and 673 edges. If the reader is interested, the automaton can be found using the online tool [1] with the input F(rball && F(basket && r2)) && F(gball && F(basket && r4)) && G(rball -> X(! gball U basket)) && G(gball -> X(! rball U basket)) && F(G(r1)).

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. Our algorithm chooses the ball that is closest

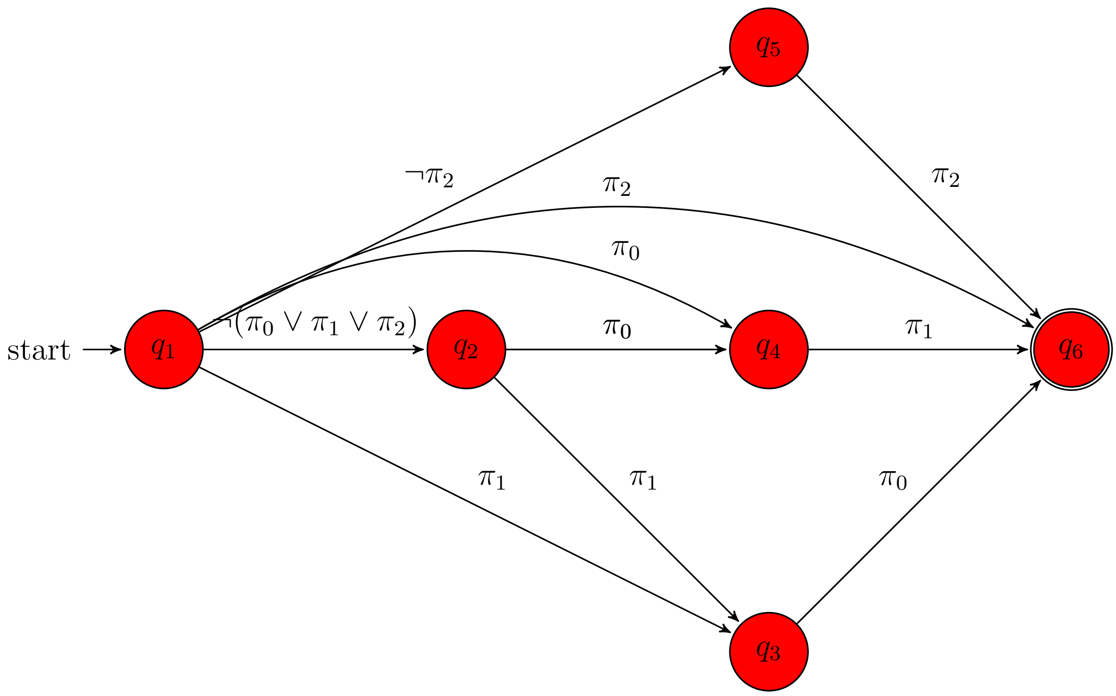



Figure 18: Simplified Büchi Automaton Corresponding to $(\diamond\pi_0 \wedge \diamond\pi_1) \vee \diamond\pi_2$

to us.

### 0.4.4 OR Operator

As we have seen in the LTL semantics, LTL formulas can contain an OR Boolean connective i.e. $\varphi = \varphi_1 \vee \varphi_2$. In all the other examples that we have seen, the formulas specify tasks and the algorithm has to *at most* choose the order of the tasks. The OR connective introduces the idea that the algorithm has to choose *which* tasks to do. Let us first look at the formula $(\diamond\pi_0 \wedge \diamond\pi_1) \vee \diamond\pi_2$. The Büchi automaton corresponding to this formula as calculated by [8] is shown in figure 18.

The distances corresponding to this automaton are $d(q_1) = 1$, $d(q_2) = 2$, $d(q_3) = 1$, $d(d_4) = 1$, $d(q_5) = 1$ and $d(q_6) = 0$. As one can see, starting with a distance of 1, the only lower level is 0, which is the accepting level. Therefore our algorithm will only do one Dijkstra search, which is the same as the accepted algorithm. Our algorithm therefore gives the optimal result.

## 0.5 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

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 = []

    # current level of starting node
    cur_level = G.node[source]['dist']

    #for source in sources:
    seen[source] = 0
    push(fringe, (0, next(c), source))
```

```python
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:
        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]
print next_node
print type(next_node)
return dist, next_node, paths
```

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

[6] 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.

[7] 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.

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

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

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

[11] AJ Hoffman, J Wolfe, RS Garfinkel, DS Johnson, CH Papadimitriou, PC Gilmore, EL Lawler, DB Shmoys, RM Karp, JM Steele, et al. *The traveling salesman problem: a guided tour of combinatorial optimization.* J. Wiley & Sons, 1986.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.