Thesis

Garrett Thomas

April 26, 2017

1 Abstraction of the Workspace

In [1], 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.

To create a graph that represents the robot motion we need to consider the workspace, denote $W_0 \subset \mathbb{R}^n$, our robot is in and the dynamics of the robot. First we partition our environment, W_0, into a finite number of equivalence classes. Various partitions can be used i.e. [2] IS THIS TOO SIMILAR TO FAINEKOS2005, however the partition must satisfy the bisimulation property, which will be defined later once more notation has been introduced. 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 $i \neq j$.

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

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. 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 ... 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 π_i. From meng*

We also include a definition of the discrete counterpart

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

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...

Next, we must define transitions to show the relationship between the regions. We define a transition as follows

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

Definition 5. *There is a discrete transition, $\rightarrow_D \subset \Pi \times \Pi$, from π_i to π_j, denoted $\pi_i \rightarrow_D \pi_j$ if there exists x and x' such that $T(x) = \pi_i$, $T(x') = \pi_j$ and $x \rightarrow_C x'$*

We can now define bisimulations

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

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

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

We can now definite 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 Π is the set of states, $\rightarrow_D \subseteq \Pi \times \Pi$ is the transitions relation where $(\pi_i, \pi_j) \in \rightarrow_C$ iff there is a transition from π_i to π_j as defined in definition* **??**. *In adherence to common notation, we will write $\pi_i \rightarrow_C \pi_j$. Note: $\pi_i \rightarrow_C \pi_i$, $\forall 1, 2, \ldots w$. $\Pi_0 \subseteq \Pi$ 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 1.*

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 Π, $\rightarrow_C$, Π_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 $\rightarrow_C$.*

When we use a WFTS, we will estimate the cost of a transition by the Euclidean distance between the centers of the two regions. Note: $W_C(\pi_i, \pi_i) = 0$.

When talking about FTS, it is helpful to use the notation $\text{Pre}(\pi_i) = \{\pi_j \in \Pi | \pi_j \rightarrow_D \pi_i\}$ to define the the predecessors of the state π_i and $\text{Post}(\pi_i) = \{\pi_j \in \Pi | \pi_j \rightarrow_D \pi_i\}$ to define the the successors of the state π_i. In this thesis, we will only be dealing with infinite paths. An infinite path of $\mathcal{T}$ 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. $\text{trace}(\tau) = L_D(\pi_1) L_D(\pi_2) \ldots$.

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. α is true at some point of time. The particular temporal logic we will be using is known as linear temporal logic (LTL) [3]. 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, φ_1 and φ_2 are LTL fomulas, $\neg$ and $\vee$ denote denote the standard Boolean connectives negation and disjunction respectively, X being the "Next" operator. U is the temporal operator "Until", with $\varphi_1 \mathcal{U} \varphi_2$ meaning φ_1 is true until φ_2 becomes true. Given these operators, we can define the following additional prepositional operators:

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

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

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

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. We now give examples to illustrate this point. 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". We now give examples of common tasks converted to LTL formulas [5]

1. **Reachability while avoiding regions**: "Go to region π_{n+1} while avoiding regions $\pi_1, \pi_2, \ldots, \pi_n$"
   $\neg(\pi_1 \vee \pi_2 \ldots \pi_n)\mathcal{U}\pi_{n+1}$
2. **Sequencing**: "Visit regions π_1, π_2, π_3 in that order"
   $\diamond(\pi_1 \wedge \diamond(\pi_2 \wedge \diamond\pi_3))$
3. **Coverage**: "Visit regions $\pi_1, \pi_2, \ldots, \pi_n$ in any order"
   $\diamond\pi_1 \wedge \diamond\pi_2 \wedge \cdots \wedge \diamond\pi_n$
4. **Recurrence (Liveness)**: "Visit regions $\pi_1, \ldots, \pi_n$ in any order over and over again"
   $\Box(\diamond\pi_1 \wedge \diamond\pi_2 \wedge \cdots \wedge \diamond\pi_n)$

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 [7]: "Pick up the red ball, drop it to one of the baskets and then stay in room one"
$\diamond(rball \wedge \diamond basket) \wedge \diamond\Box r1$

As it is possible to develop develop computational interfaces between natural language and temporal logics[8], LTL.

An infinite word over the alphabet 2^{AP} is an infinite sequence $\sigma \in (2^{AP})^{\omega}$, i.e. $\sigma = S_0 S_1 S_2 \dots$ where $S_k \in 2^{AP}$ for all $k = 1, 2, \dots$ where S_k is the set of atomic propositions that are true at the time stop k.

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

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

In the interest of the reader, we will denote $(\sigma, 0) \models \varphi$ by $\sigma \models \varphi$, which is what is used for all this thesis. The connection to an infinite path τ of an FTS is that the trace of the path, trace(τ), is a word over the alphabet 2^{AP}. Given the LTL semantics, we can now verify if a path satisfies an LTL formula! We will say an infinite path τ satisfies φ if its trace satisfies φ, i.e. $\tau \models \varphi$ if $trace(\tau) \models \varphi$. A path satisfying φ will be referred to as a plan for φ.

We now describe how to construct a path of an FTS that satisfies an LTL formula.

3 Büchi Automata

Given an LTL formula φ over AP, there exists a Nondeterministic Büchi automaton (NBA) over 2^{AP} corresponding to φ, denoted A_φ according to [1, Theorem 5.37] lars' paper

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

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

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

An infinite run of an NBA is an infinite sequence of states that starts from an initial state and follows the transition relation i.e. $r = q_0 q_1 \dots$ where $q_0 \in \mathcal{Q}_0$ and $q_{k+1} \in \delta(q_k, S)$ for some $S \in 2^{AP}$, $k = 0, 1, \dots$. An infinite run r is accepting if $\texttt{Inf}(r) \cap \mathcal{F} \neq \emptyset$, where $\texttt{Inf}(r)$ is the set of states that appear infinitely often in r. from meng As with the FTS, we denote the predecessors of $q_m \in Q$ $\texttt{Pre}(q_m)$ and the successors $\texttt{Post}(q_m)$

Definition 11. *Given an infinite word $\sigma = S_0 S_1 S_2 \dots$ over 2^{AP}, its resulting run in $\mathcal{A}_\sigma$ is denoted by$r_\sigma = q_0 q_1 q_2 \dots$, which satisfies: (i) $q_0 \in Q_0$; (ii) $q_{i+1} \in \delta(q_i, S_i)$, $\forall i = 1, 2, \dots$. Similar statement holds for a finite word $\bar{\sigma} = S_0 S_1 S_2 \dots S_{N+1}$.*

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 12. *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* $q_n \in \delta(q_m, L_c(pi))$; $Q'_0 = \{\langle \pi, q \rangle | \pi \in \Pi_0,\ q_0 \in Q_0\}$, *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* $\langle \pi_j, q_n \rangle \in \delta'(\langle \pi_i, q_m \rangle)$ *meng*

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

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

Lemma 1. *If there exists an infinite path* τ *of* $\mathcal{T}_c$ *such that* $\tau \models \varphi$, *then at least one accepting run of* $\mathcal{A}_p$ *exists.*

Proof. see the proof of Theorem from [11] meng

Lemma 2. *If* R *is an accepting run of* $\mathcal{A}_p$, *then* $R|_\Pi \models \varphi$

Proof. see proof in meng

Given 2, our problem is now to find an accepting run of $\mathcal{A}_p$. Given that an accepting run is a infinite sequence of states, and there are infinitely many possibilities, the process of finding one, nonetheless finding one that has measure of optimality is non-trivial, both theoretically and practically meng "both in theory and software implementation". Therefore we restrict our view of accepting runs to runs that satisfy the prefix-suffix structure i.e.

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

The prefix τ_{pre} is traversed only once and the suffix τ_{suf} is repeated infinitely often meng (which is the meaning of the ω superscript). Plans of this form are much easier to deal with because, while they are still infinite plans, they have a finite representation that we can exploit.

5 Cost of a Run

We are focusing on the accepting runs of $\mathcal{A}_p$ with the prefix-suffix structure

$$\begin{aligned} R &= \langle R_{pre}, R_{suf} \rangle = q_0 q_1 \dots q_f [q_f q_{f+1} \dots q_n]^\omega \\ &= \langle \pi_0, q_0 \rangle \dots \langle \pi_{f-1}, q_{f-1} \rangle [\langle \pi_f, q_f \rangle \langle \pi_f, q_f \rangle \dots \langle \pi_n, q_n \rangle]^\omega \end{aligned}$$

from meng where $q_0 = \langle \pi_0, q_0 \rangle \in \mathcal{Q}_0$ and $q_f = \langle \pi_f, q_f \rangle \in \mathcal{F}$. There is a finite set of transitions in our infinite path i.e.

$$\texttt{Edge}(R) = \{(q_i, q_{i+1}), i = 0, 1, \dots, (n-1)\} \cup \{(q_n, q_f)\}.$$

Each of these transitions has a cost, given by $W_p(q_i, q_{i+1})$ allowing us to define the total cost of R as

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

where $\gamma \geq 0$ is the relative weighting of the transient response (prefix) cost and steady response (suffix) cost. We then look for the accepting run with prefix-suffix structure that minimizes the total cost.

We will denote this accepting run 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 [11] we see that simplifications in the translation from LTL formulas to NBA can result in a loss of optimality. This will be important when we analyse the paths our algorithm generates.

6 Search Algorithm

The search algorithm used in many recent works on the specific type of control planning synthesis comes from this prefix-suffix structure. The basic idea is to find a path from an initial node, q_0 to an accepting node, q_f, and then find a path from the q_f back to itself. The first part from q_0 to q_f is the prefix and the second part q_f back to q_f is the suffix. Then the resulting path, τ, will be the prefix, followed by the suffix repeated infinitely many times. This path is thus accepting because the suffix finds the path from an initial state back to itself, and thus contains the initial state, and is repeated infinitely many times $q_f \in \texttt{Inf}\tau \Rightarrow \texttt{Inf}\tau \cap \mathcal{F} \neq \emptyset$.

Algorithm 1, from [7], 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: If Q'_0 or $\mathcal{F}'$ is empty, construct Q'_0 or $\mathcal{F}'$ first.
2: For each initial state $q'_0 \in S'$, call $\texttt{DijksTargets}(\mathcal{A}_p, q'_0, \mathcal{F}')$.
3: For each accepting state $q'_f \in \mathcal{F}'$, call $\texttt{DijksCycle}(\mathcal{A}_p, q'_f)$.
4: Find the pair of $(q'_{0,opt}, q'_{f,opt})$ that minimizes the total cost
5: 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.

$\texttt{DijksTargets}(\mathcal{A}_p, q'_0, \mathcal{F})$ computes the shortest paths in $\mathcal{A}_p$ from initial state $q'_0 \in \mathcal{Q}_0$ to every accepting node in $\mathcal{F}$ using Dijkstra's algorithm [4] and $\texttt{DijksCycle}(\mathcal{A}_p, q'_f)$ computes the shortest path in $\mathcal{A}_p$ from accepting state q'_f back to itself using Dijkstra's algorithm.

7 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 [3]. 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 13. *An NBA with distance, NBAD, is defined by a six-tuple:*

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

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

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

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 Dijkstra search to find the fastest way from the accepting state back to itself. We do not 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. An explanation of the the name will be given in the following discussion. Pseudocode is given in Algorithm 2

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.

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

8 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

Procedure 2 NearestNeighborRun()

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

PathPre = []
Level = $d_p(q'_0 \in \mathcal{Q}'_0)$
Found = false
while Found == false **do**
    find NextNode to add using Dijkstra's algorithm
    if d_p(NextNode) == Level - 1 **then**
        add path to NextNode onto Path
        if d_p(NextNode) == 0 **then**
            found = true
        else
            Level = Level - 1
PathSuf = shortest path from last node in path back to it self, found using Dijkstra's search
R_{nn}, prefix: PathPre; suffix: PathSuf.

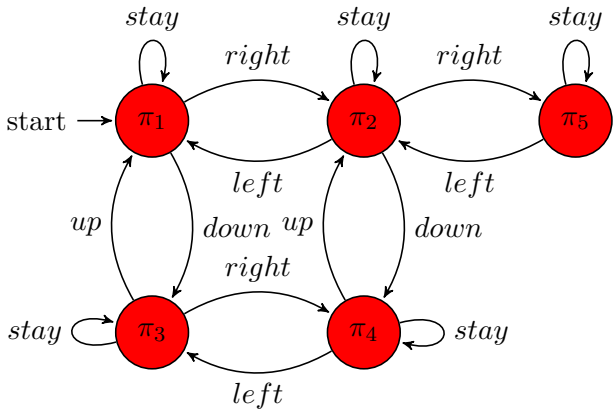

Figure 1: *FTS*

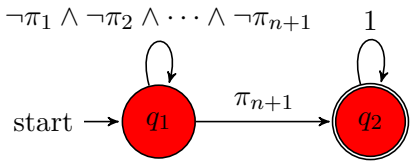

Figure 2: Büchi automaton corresponding to reachability while avoiding regions

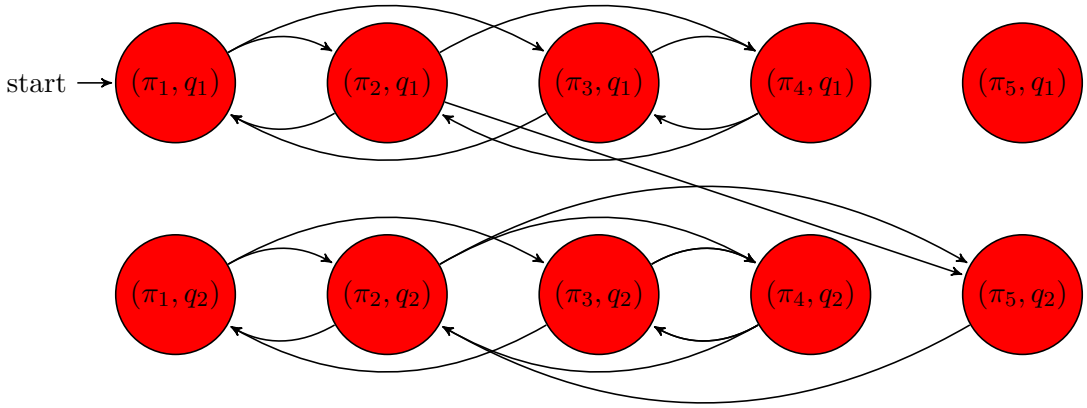

Figure 3: Product Automaton

8.1 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 π_{n+1}. After reaching π_{n+1} we are free to do what we want. This behaviour is expressed by the formula $\neg(\pi_1 \vee \pi_2 \vee \ldots \pi_n)\boldsymbol{\mathcal{U}}\pi_{n+1}$.

! include Wring LTL calculation

The Büchi automaton corresponding to this formula is given in figure 2

As we can see $d_p(q_1) = 1$ and $d_p(q_1) = 0$. 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 (π_1, q_1) which ends at (π_5, q_2). This is exactly what the accepted algorithm does, so we do not gain anything when using our algorithm on a formula of this type.

8.2 Sequencing

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. This behaviour is ideal for our algorithm.

We show why in an example using the formula $\diamond(\pi_2 \wedge \diamond\pi_5))$ the same FTS as before. The product automaton is shown in figure 5

Our algorithm finds (π_4, q_2), then starts another Dijkstra search and finds (π_5, q_3). Will search through extraneous nodes, for example (π_5, q_1). This may not seem like a lot in this example, but when we expand to larger problems the difference becomes significant. Check how

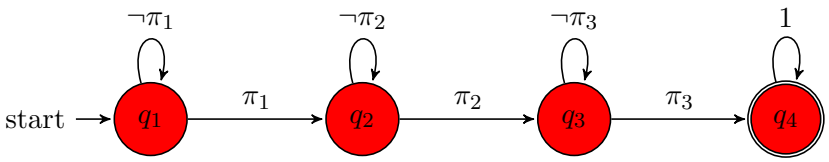

Figure 4: Example 1: BA

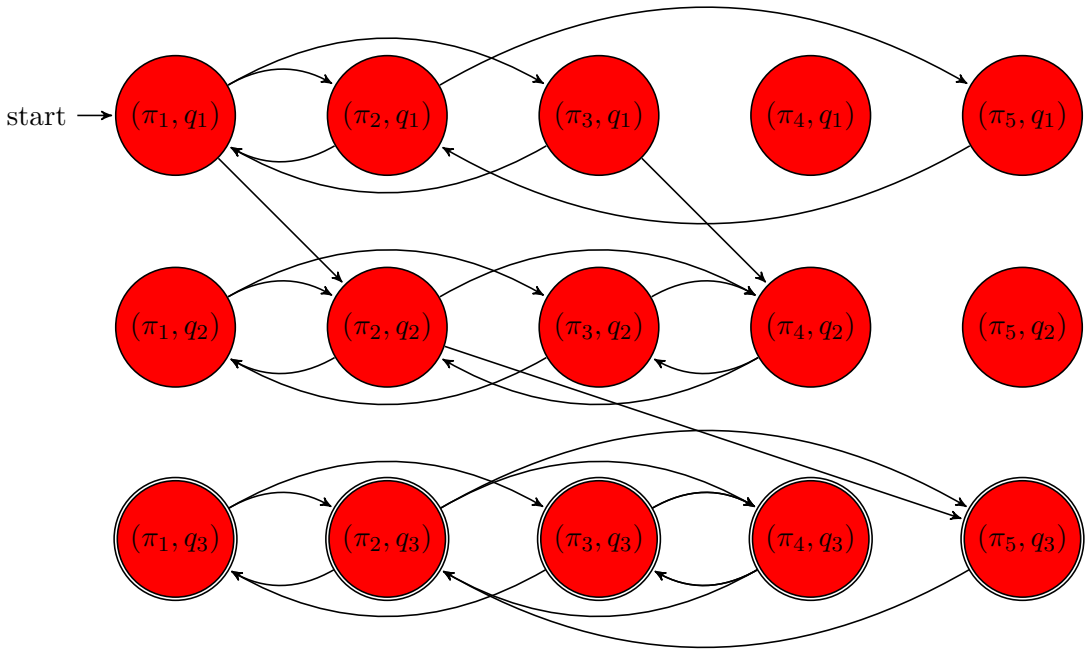

Figure 5: Product Automaton

many nodes are searched with both algorithms, and show time difference.

8.3 Coverage

When we use our algorithm on a coverage formula, we will likely 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 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 [10]. 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. It has been shown that for an n-node travelling salesperson problem,

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

where NEARNEIBR is the cost of the path generated by the nearest neighbour algorithm and OPTIMAL is the cost of the optimal path. We now need to formulate our problem as a travelling salesperson problem. 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 [9]. 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. To formulate this seemingly unrelated problem into a travelling salesperson problem, we set P to be the set of pins to be interconnected, c_{ij} to be the distance between pin i and pin j. They then introduce a dummy node $*$ that is a distance 0 from all the other nodes i.e. $c_{i*} = c_{*i} = 0$ for all i. Then the corresponding problem is solving the travelling salesperson problem on the set of nodes $N = P \cup \{*\}$. We use this same idea, however ...

We provide a proof that

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

We begin by proving

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

which can be found in [10]. Let l_i be the length of the i^{th} largest edge in the tour obtained by the nearest neighbour algorithm.

8.4 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,

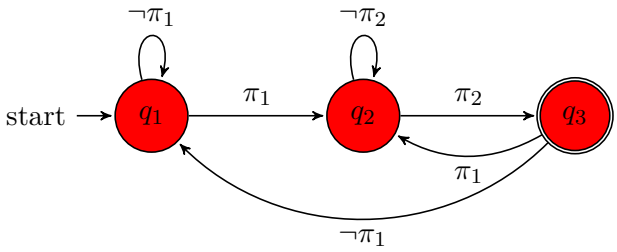

Figure 6: Example 1: BA

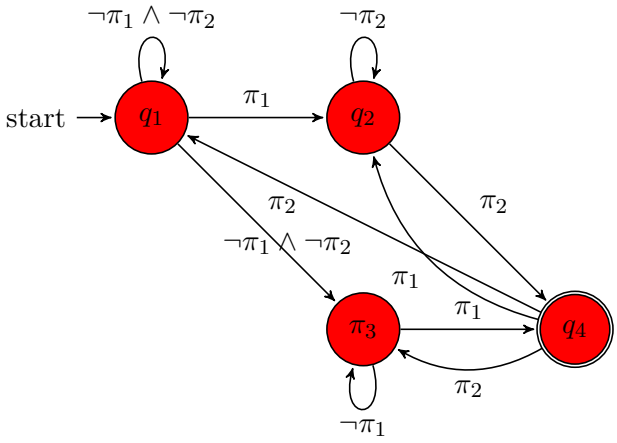

Figure 7: Product Automaton

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 [6] is

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 π_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 6 the only way to do this is to go to region π_1. Our algorithm does this, and then starts a new Dijkstra search. In this case the same statement holds for π_2. Therefore the optimal prefix is to concatenate the optimal paths down from each each level (first to π_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.

This path however is in general not truly optimal. It is because the Büchi automaton given in figure 6 not a tight Büchi automaton [11]. A Büchi automaton is tight if it accepts the shortest lasso (prefix and suffix) i think??? The loss of this optimality property is due to the fact that the algorithm in [6] 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 7.

In this automaton, $d(q_1) = 2$, $d(q_2) = d(q_3) = 1$, and $d(q_4) = 0$. So, we are starting at the same level i.e. 2, however this time we have two choices about what to do to get down to

level 2; we can go to π_1 or π_2. Being able to choose is good in the sense that we can now find the optimal path, and bad in the sense that the extra state in the *Büchi* automaton increased the size of the product automaton by 33% (hence increasing the time it takes to search the automaton). This very well illustrates the trade off between the search time and optimally/cost of the resulting run. We propose that this is a good way to think about our algorithm. There is a trade off that sometimes it will not find the optimal run, even if this is possible, though it will be faster.

The second aspect of this problem that we wish to look at is fact that it does not have a trivial suffix. In the other examples we have looked at, the suffix of the calculated path (with our algorithm and the accepted algorithm) was a single state; that is, the formula could be satisfied by staying in one state indefinitely. In this example, π_1, π_2, and π_3 must all be visited infinitely often, and thus these states must be in the suffix.

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

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

where $q_f \in \mathcal{F}$. Thus when calculating to the suffix we must find the path back to the *same* accepting state. We cannot not just look for any accepting state as we do in the prefix calculation. Our algorithm in general only looks for an accepting state, not a specific accepting state; however in certain circumstances it can find a specific accepting state. We illustrate this using the same examples above.

$\square(\diamond\pi_3 \wedge \pi_5)$ We notice how in figure 6 there is only one arrow to the accepting state, labelled π_5. This implies that the only way to get down to level 0 is to go to π_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 7 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.

References

[1] 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.

[2] 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.

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

[4] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[5] 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.

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.