ABSTRACT

Title of Thesis:	GRAPH-BASED METHODS FOR PATH PLANNING WITH DYNAMIC OBSTACLES USING LINEAR TEMPORAL LOGIC
	Wenqi Han Master of Science 2018
Thesis Directed By:	Professor Jeffrey W. Herrmann Department of Mechanical Engineering and Institute for Systems Research

Abstract: Autonomous vehicles are expected to play a key role in rescue and transportation. Planning an optimal path with the minimum computational effort for these vehicles in their missions improves their efficiency and adds safety for the vehicles and third parties on the ground. The objective of this thesis is to study the computational effort of four planning methods that implement linear temporal logic (LTL) to translate the high-level mission requirements and environmental specifications. The Potential Field Method and the Critical Path method required less

computational effort to find one of the shortest paths for the mission The Multigraph Network Planning method and the Critical Path method can find all the possible paths with predetermined path length. The Random Walk method required more computational effort and memory compared to the other three methods.

GRAPH-BASED METHODS FOR PATH PLANNING WITH DYNAMIC OBSTACLES USING LINEAR TEMPORAL LOGIC

by

Wenqi Han

Thesis submitted to the Faculty of the Graduate School of the

University of Maryland, College Park, in partial fulfillment

of the requirements for the degree of

Master of Science

2018

Acknowledgements

I sincerely thank my advisor Dr. Herrmann for giving me this opportunity to work on this interesting research problem. With his technical expertise, he inspired me to come up with different ideas for the research problem. With his insightful and motivating comments, he guided me to conduct the research step by step when I was lost in too many different ideas.

I would like to thank Dr. Mumu Xu and Dr. Cleaveland for teaching me model checking, working with me on UAV path planning projects, and agreeing to serve on my committee.

I also would like to thank Dr. MacCarthy for helping and encouraging me to take an extra step on learning programming skills. I acknowledge with a deep sense of gratitude and most sincere appreciation

Last but not the least, I would like to thank my parents for their support. They taught me the things that matter the most in life.

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

1.1 Motivation

Motion planning is an important problem in various areas such as robot navigation, driverless cars, robotic surgery, protein folding, and safety and accessibility in computer-aided architectural design. The research in this thesis is motivated by the problem of motion planning for unmanned aerial vehicles (UAV).

The idea of building flying machines was first conceived around 2,500 years ago in ancient Greece and China. In 425 BC, Archytas, known as Leonardo da Vinci of the Ancient World, built the first known autonomous mechanical bird, “the pigeon”, which is reported to be able to fly about 200 meters [1]. During the same era, the Chinese experimented different types of flying machines, such as hot air balloons, rockets, and kites. These machines were used both for entertainment and military. Historical records show that a “wooden hawk” was used for reconnaissance around 450 BC, and Ming Dynasty armies used a kite in the shape of a crow to bomb enemy positions [1]. Before the appearance of manned aviation in the late 1700s, these flying machines had already shown their potential in various areas. Around the time of First World War (1916), unmanned aircraft appeared.

In recent years, autonomous robots have replaced a lot of people to do the “dull, dirty, and dangerous” work over the years. While autonomous on-land vehicles (self-driving cars) have been used to improve our driving experience, unmanned aerial vehicles, have had many applications as well. Johns Hopkins researcher Timothy Amukele et al. [2] demonstrated that drones are safe for transportation of blood products

[2]. In Amukele’s experiment, his team not only showed that a drone could transfer blood samples in places like coastal Haiti after earthquake, where the land is rough, but waterways are clear, but also successfully tested that blood temperature would be kept in acceptable levels after 8 to 12 miles travelling at around 330 feet above the ground. Microdrones' demonstration with the German Lifeguard Association also showed that if a drone carries a self-inflating flotation device, it can help the swimmer float and give time to lifeguards to react [3]. Amukele’s study and Microdrones both showed that UAVs have the potential in helping save lives in emergency situations.

In terms of reacting to industrial accidents or other dangerous situations, UAVs plays a more important role nowadays, such as radiation detecting for toxic leaks and tracking hurricanes. Dronemakers FlyCam UAV partnered with US Nuclear Corp., a radiation detection company, demonstrated that “The UAVs can be used to detect radiation leaks in nuclear power plants or flown into plumes of smoke from a burning building to give first responders immediate data about what kinds of hazards might be present. It can also be used for to monitor public events, sea ports or geographic areas to detect possible dirty radiological bombs or the use of chemical and biological agents.” NASA’s RQ-4 Global Hawk is built for watching disasters unfold. In 2016, NASA used the Global Hawk to drop expendable sensors to record temperatures, pressure, relative humidity, and wind speed and direction, and transfer the information to scientists to track Hurricane Matthew [4].

UAVs with their ability to fly, has not only been used in those extreme environments, but also been used in our everyday life. It has taken roles such as inspection and monitoring in highly risky fields, surveying and mapping cities, condition survey in civil engineering field, and imaging for HD films, videos, and HR photos. Both in military and civil operations, more and more complicated tasked are excepted to be done by UAVs.

To accomplish the mission requirements, a UAV needs to maneuver from the initial position to the final targets, while avoiding both static and dynamic obstacles. Researchers have developed real-time obstacle avoidance approaches based on dynamical systems [5], potential fields [6], and receding horizon control [7]. However, these real-time obstacle avoidance techniques cannot guarantee perfect obstacle avoidance, and their computational effort may be excessive when there are many moving obstacles. In addition, when the mission requirements include a list of targets for the UAV to reach, real-time obstacle avoidance techniques do not provide the means to yield a shortest path for the UAV to accomplish its mission using the least time and energy. To solve this problem, temporal logic has been used to express complex UAV missions.

This thesis compares four different planning approaches that use linear temporal logic (LTL) to specify the mission requirements: The Multigraph Network Planning method, the Random Walk method, and the Potential Field method. It also presents the results of experiments conducted to compare the computational effort and solution quality of the planning approaches.

1.2 Problem Definitions

This thesis focuses on finding a shortest path for unmanned vehicles under high-level specifications in dynamic environment.

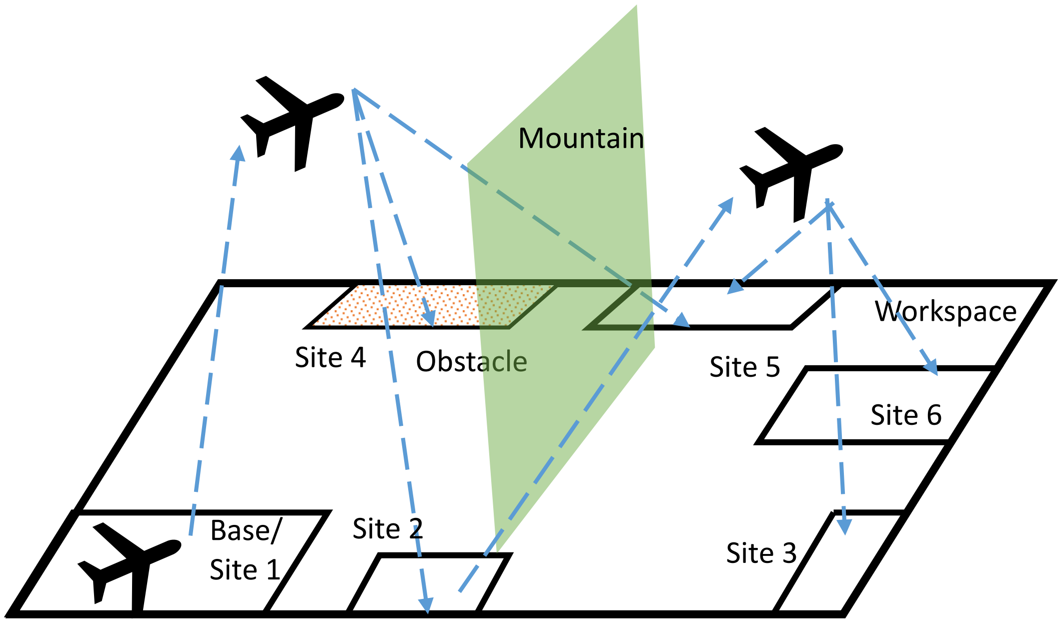


Figure 1-1: UAV fly in the workspace with obstacle area

Figure 1-1 shows an example of an UAV rescue mission. Site 1 is the base (starting point). Sites 2 – 6 for different locations where UAVs can land or possible mission area. Site 2 is a hospital. Site 3 is a food dropping point. Site 4 is an obstacle area where an earthquake took place and the weather can be very bad for flying according to the weather forecast. Also, there are patients waiting for pick-up in site 4. Site 5 and site 6 both provide supplies for UAVs. There is a high mountain in the middle green area. Because of this mountain, UAVs cannot fly between site 1 and site 6 directly. UAVs shall go through site 2 or site 5 to avoid the mountain. The same thing applies to transportation between site 3 and site 4. In robot path planning, mission requirements are usually very high-level task with some subtasks. For example, requirements can be that the UAV shall go to site 3 to drop food, or the UAV shall go to the hospital to pick up nurses and doctors and send them to the area where an earthquake took place. Radar and other real-time technologies can help vehicles find the shortest path between two points, but we also want to find the shortest path that fulfills all the high-level requirements. Researchers have created different formulations and computational approaches to mathematically represent these high-level requirements, such as motion sequencing. In this thesis we will focus on (LTL) and Potential Field. Meanwhile, we also want to find a way to avoid all the obstacles that may present in the workspace.

1.3 Organization

This section provides an outline of the thesis. Chapter 1 starts why the UAV motion planning problem should be solved and problem definitions, which goes into more detail about the problem scenario and relevant assumptions. Chapter 2 is the literature review of previous work in UAV motion planning with temporal logic and potential field. Chapter 3 describes the four planning algorithms and how they generate solutions. Chapter 4 uses Example 1 to show detailed steps of the implementation, results, and computational effort of each method. Four other examples are also included. Chapter 5 presents the analysis of 5 examples' results and the recommendations based on the evaluation. Chapter 6 completes the discussion by providing conclusions and possible future work.

Chapter 2 Related Work

This chapter reviews previous work in UAV motion planning using temporal logic and potential field - shortest path planning approaches, constraint reachability problem in path planning for nonlinear systems with temporal logic, and path planning with timing constraints in the mission requirements.

As the popularity of UAV increases, so do concerns about safety. The risk to people on the ground is related to the UAV path. Also, when it comes to rescuing after a disaster like an earthquake, time is very valuable. High-level motion planning for UAVs can help lower the risk for people on the path as well as the risk for UAVs by avoiding predictable weather changes and dangerous areas [1].

Numerous shortest path planning approaches have been proposed. Richards and How [8] used the mixed-integer linear program (MILP) to plan a trajectory with for discrete time steps multiple aircraft to reach multiple targets before a minimum required time and used a rectangular exclusion region around each aircraft to avoid the collision. Each aircraft was required to visit some points on the map, but the order of those visit was selected by the program for overall shortest flight time. This research did not consider the scenario when the obstacles like a bad weather area can move during the mission. Jun and Andrea [9] used sensors to build and adjust a probability map for obstacle avoiding. For path planning, Jun decomposed the region into uniform cells and then used the Bellman-Ford algorithm and the polygonal path to successfully find the shortest path for UAVs between two points. Jun also concluded that finding different paths for multiple UAV mission can decrease the overall risk. Another shortest path planning that solves dynamic obstacle problem is D* Lite. Koenig and Likhachev [10] introduced D* Lite method, which applies Lifelong Planning A* to vehicle navigation. This method works very well in terms of avoiding dynamic obstacles. However, these approaches did not consider the scenario when the mission might have multiple targets and include requirements like visiting one target before the others.

Constraint reachability problem in path planning for nonlinear systems with temporal logic has also be researched. There were different elements that constraints reachability, such as distance-constrained reachability problem [11]. In path planning, when there were too many obstacles in the workspace, not all the mission required sites could be reached. Wolff et al. [12] found a feasible trajectory for a single UAV using a coarse abstraction of the system and an automation representing the temporal logic specification, including sampling-based methods for motion planning, reachable set computations for linear systems, and graph search for finite discrete systems. Shaffer et al. [13] also implemented LTL for high-level mission planner to reactively fight wildfire, using TuLiP [14]. This research also detected the minimum number of UAVs to complete its mission requirements by searching for a feasible trajectory. It was pointed out that "on an Intel i5-6500 CPU @ 3.20 GHz processor, this total process, approximately 250 regions, took on the order of 8 hours for a system with 16 GB of RAM. Both papers did not look for an optimized solution. Also, computational time like 8 hours used by TuLiP to research for reachability may not be quick enough for large wildfire type of disaster.

Another important factor, timing constraints in the mission requirement, has also been addressed. Zhou et al. [15] used metric temporal logic (MTL) to encode the task specifications with timing constraints and mixed integer linear program solver for the optimization problem. Then, in [16], Zhou et al. used a more direct approach to access the time constraints – timed automata approach using metric interval temporal logic (MITL) using UPPAAL. On a computer with a 3.4GHz processor and 8GB memory, for workspace 16 and 64 locations and simple mission requirement, this approach ran very fast. Maity and Baras [17] extended LTL with bounded time to represent the bounded time high-level specification and generates a discrete path that met specifications with optimization. Sophisticated model checking tools such as SPIN Holzmann [18] was used to generate discrete robot paths.

Combining sensors and the use of LTL, Ulusoy and Belta [7] presented a controller that combines both offline high-level trajectory path plan and online receding horizon control for overall mission requirements which include temporal logic statement, prior known requests, dynamic requests that could be sensed only locally, and a servicing priority order over these dynamic requests. This controller had the advantage of computational efficiency. Ulusoy and Belta [7] also used LTL2BA tool (http://www.lsv.fr/~gastin/ltl2ba/index.php) [19] to obtain the Büchi automaton. However, the dynamic request might block the vehicle's progress and some low-priority dynamic request might not be serviced due to the sensing range as the vehicle moves towards a higher-priority dynamic request.

Artificial Potential Field (APF) is another method used in robot path planning. Khatib first introduced APF to robot real-time obstacle avoidance [20], and other improvements were made since then [21], [22]. Commonly, a particle representing the robot is under the influence of an APF, which is denoted by U. APF reflects the free space structure by identifying the local variation using potential function $U_{total} = U_{att} + U_{rep}$, which is the sum of attractive potential and repulsive potential. Attractive potential pulls the vehicle towards the target, while repulsive potential pushes the vehicle away from the obstacles [20] - [22].

Compared to AI methods like Evolutionary Algorithms (EAs), the APF method was more efficient in known environments, according to Montiel et al. [23]. However, the APF method frequently failed due to local minima, and computational effort increases quickly when introducing real-world landscapes with enormous complexity. In addition, traditional APF was a local planning method that did not perform very well in terms of finding a global optimal path, which was also due to the local minimum problem. Montiel et al. [23] presented a Parallel Evolutionary Artificial Potential Field (PEAPF) to overcome the local minimum problem, where they used parallel evolutionary computation for finding dynamically the optimal k_a and k_r values for the attractive and repulsive proportional gains in Eq. 3.4.1.2 and 3.4.1.3. Montiel proved that PEAPF method ensured path optimization even in complex real-world sceneries with dynamic obstacles. However, the examples in [23] only had one target on the path.

In conclusion, these existing approaches need improvement to solve high-level mission specifications and dynamic obstacles. Direct shortest path planning methods do not handle high-level mission requirements. Methods like D* Lite can avoid dynamic obstacles, but others cannot. Planning methods like MTL and LTL can express those high-level missions, but they tend to be very time consuming and grows exponentially as the mission becomes more complex. This is not good when we need to plan a path for emergency missions. To solve this problem, we need to find a method that can solve high-level mission requirements and dynamic obstacles, requiring less computational effort than existing methods.

Chapter 3 Planning Algorithms

This chapter provides detailed explanation about the workspace representation, state machine diagram, the extended linear temporal logic (LTL) for environmental specifications, four planning algorithms, and how they generate solutions: The Multigraph Network Planning method, the Random Walk method, the Potential Field method, and the Critical Path method.

3.1 Workspace Representation

The workspace, $\mathcal{W}$, represents the whole space that is involved in the mission. We can partition the continuous workspace $\mathcal{W}$ by dividing $\mathcal{W}$ into small cells.

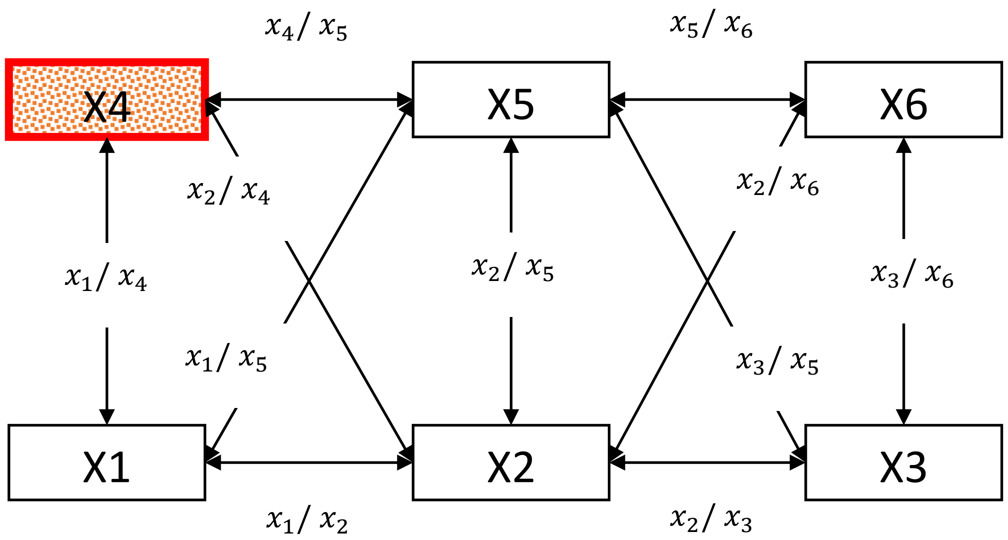


Figure 3-1: Workspace Example. A workspace with 6 stations that vehicles can visit. X4 may be blocked by moving obstacles during certain period. Vehicles can move according to the actions between stations (i.e.x_1, x_2).

Let $\boldsymbol{X} = \{\mathrm{X1}, X2, \dots, \mathrm{Xn}\}$ be a partition in the workspace. Figure 3-1 shows an example of the workspace. $\mathcal{A}$ denotes the set of actions the vehicle can take in the workspace. $\mathcal{A} = \{x_1, x_2, \dots, x_n\}$. Each action has its physical meaning, for example, x_1 means moving to the discrete cell space X1. The vehicle can be anywhere within the discrete cell space. We assume the vehicle always uses one step to perform one action.

3.2 Extended LTL for Environmental Specifications

Here we define a new operator to describe dynamic obstacles in the environment $\neg_{[t_1, t_2]}$. This is an addition to the extended operators in [17].

Definition 3.1.1: *The extension of the LTL grammar is* $\phi ::= \neg\phi_{[t_1,\ t_2]}$

$t_1, t_2 \in \mathcal{T}$, where $\mathcal{T}$ is the maximum time in which vehicles shall fly within their physical constraints. Note that $\neg\phi_{[t_0,\ \infty)}$ is equivalent to $\neg\phi$, where t_0 is starting time of σ.

Definition 3.1.2: *The semantics of* $\neg\phi_{[t_1,\ t_2]}$ *is defined as:*

$\sigma[1, 2, \dots] \vDash \neg p_{[t_1, t_2]}$

iff $\forall\, t,\ where\ t_1 \le t \le t_2, \sigma[t_1, t_1 + 1, \dots, t_2] \nvDash p,$

This is very convenient for path planning because the mission predetermines when an obstacle area may have obstacles in there.

3.3 Problem Formulation

To find the shortest path the fulfills the mission specifications under the environmental specifications for the dynamic obstacles, we assume that the information of workspace, the mission specifications for vehicles, and the environmental specification are all known. We also assume that it takes one step for a vehicle to move around its current location, as shown in Figure 3-2. We assume that each movement across cells the vehicle takes the same amount of time, which can be defined according to the various types and speed of the vehicles. The vehicle can be anywhere within the cell. The distance of the vehicle's one step is not considered when finding the shortest path, as long as the distance is reachable by that vehicle's velocity. Hence, in this thesis, the shortest path should contain the least number of steps in it. Note that researchers can also modify the algorithm to compare the distance or time of the vehicles' shortest path.

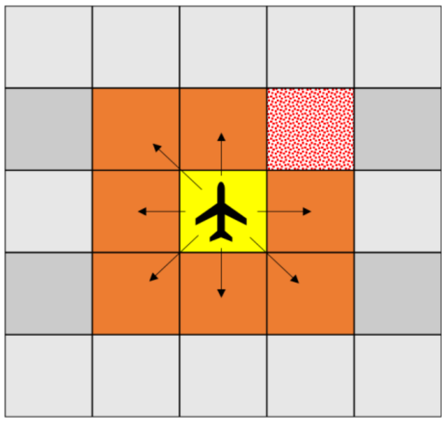


Figure 3-2: Vehicles moving rules. The cell with red large confetti pattern represents obstacle area.

While the control methodologies developed in this research can be extended to three dimensions, it is assumed that the vehicle is moving on the same altitude (layer). In addition, we do not consider bounded time requirements for the vehicles. Time is counted using the number of steps and is only considered for obstacle avoiding

3.4 State Machine Diagram

With each new action step, the vehicle will be in a set of states. For instance, a vehicle has only flown to X1 and then to X2 on its path, we define that when the vehicle was in X1, $x_1 =$ fly to X1, the vehicle was in state S_1. S_1 represents that the vehicle has only flown to X1. After the vehicle flew to X2, $x_2 =$ fly to X2, the vehicle was transferred to state S_2. S_2 represents that vehicle has been to X1 and X2 in the workspace. If the vehicle returns to X1, it achieves a new state, this state represents that vehicle has been to X1, X2, and X1 again in the workspace. Combining all the possible states creates a State Machine Diagram, $\mathcal{G}_{sm}$: S, S = $\{S_1, S_2, \dots, S_k\}$.

Let $\mathcal{A}_{s_k}$ $(\subseteq \mathcal{A})$ denote all the possible transition for a VEHICLE at state S_k to take in the state machine diagram. $\mathcal{A}_{s_k} = \{x_1, x_2, x_3, \dots, x_n\}$. $\mathcal{A}_{s_k}$ also follows the transition rules of $\mathcal{W}$.

LTL2BA tool (http://www.lsv.fr/~gastin/ltl2ba/index.php) [19] generates a Büchi Automaton $\mathcal{Ba}$ for the LTL rules specified. For all the four methods we researched, only the mission specifications were translated in to LTL and were used to create the Büchi Automaton. we reconstruct the State Machine Diagram based on Büchi Automaton. This thesis used Finite State Machine Designer (http://madebyevan.com/fsm/) [24] to draw the graph form state machine diagram. Note that $!x_n$ means any action other than move to cell Xn. One (1) is the power set of the possible actions set by the state that are allowed to be taken for any number of times. The rules are as follows:

- Create a dummy initial state "init" to represent the state when the vehicle is at rest. This is different from the "init" state in the Büchi Automaton
- Based on the purpose of finding the shortest path, we assume that for any edge in the state machine diagram, when an action x_k ($x_k \in \mathcal{A}_{s_k}$) is combined with ! x_n, ! x_n means do not do x_n and the end location of that edge shall be x_k.
- Eliminate the edges that require more than one end location in the physical workspace.
- When there is only ! x_n, it means that the vehicle can conduct any possible action from its current state other than cell Xn. This means the vehicle can go to any possible cell space that is reachable from its current cell space but cannot go to Xn in the next time unit.
- One (1) means any singular action. For example, if $\mathcal{A}_{s_k} = \{x_2, x_3\}, x_2$ && x_3 is not allowed. Only $\emptyset, x_2, and\ x_3$ is allowed

3.5 Overview – The Multigraph Network Planning

The overall approach of the Multigraph Network Planning is to use LTL to translate the high-level mission (user requirements) for the vehicles, to build a multigraph (network) for the workspace, $\mathcal{W}$, mission specifications, ϕ, and environmental specification, $\mathcal{E}$, and to find the shortest path that fulfills the requirements from the multigraph.

```
Algorithm 1: Extended LTL
Input: Workspace $\mathcal{W}$, global mission specification $\phi$, environmental
specifications $\mathcal{E}$ for the dynamic obstacles $\mathcal{E}$
Output: Shortest path $SP$ that fulfills the global mission $\phi$

1  Use $\mathcal{W}$ to construct a graph $\mathcal{G}$ to represent the workspace
2  Use $\mathcal{W}$ and $\mathcal{E}$ to construct Büchi automaton, $\mathcal{Ba}$
3  Translate $\mathcal{Ba}$ to a state machine diagram $\mathcal{G}_{sm}$
4  Construct a Multigraph $\mathcal{G}_{tot}$ which has a subgraph for each state in the $\mathcal{G}_{sm}$
5  Find the shortest path $SP$ according to $\mathcal{G}_{tot}$'s accepting state subgraph incoming
   edges and time count
```

3.5.1 Construct Multigraph

Based on the workspace and the reconstructed state machine diagram, we can construct graphs $\mathcal{G}$ and $\mathcal{G}_{sm}$. The next step is to construct subgraphs for each state (node) in $\mathcal{G}_{sm}$, based on $\mathcal{G}$. The goal is to include time step into these graphs. We define the network of these subgraphs as a multigraph $\mathcal{G}_{tot}$. The total number of steps t used in Algorithm 2 is the estimated total steps needed to complete all the mission requirements. In this thesis, for the first time we test Multigraph Network Planning implementation, we assumed that all the mission requirements could have been completed sometime before the vehicle moved the one more step than the total number of discrete cells in the workspace. For instance, in a workspace partitioned into 25 discrete cells, we assume the total number of step t is 26. Based on the complexity of mission requirements and the result of the first path planning result, researchers can adjust the value of t as needed.

```
Algorithm 2: Multigraph $\mathcal{G}_{tot}$

Input: Workspace graph $\mathcal{G}$, state machine graph $\mathcal{G}_{sm}$, global mission
specification $\phi$, environmental specifications $\mathcal{E}$ for the dynamic obstacles
1   $t$ = the number of discrete cells in $\mathcal{G}$ +1 // Assume the total number of steps is $t$
    //Create subgraph for each state:
2   Construct a set of vertices $V_1$ for $S_1$ in $\mathcal{G}_{sm}$ (appropriate $x_1, x_2, \ldots, x_n$ n at step 1,
    2, 3…, $t$)
3   Delete all vertices in $V_1$ that violate $\mathcal{E}$
4   Add edges according to the transition relationship in $\mathcal{G}$ in $V_1$
5   Construct a set of vertices $V_2$ for $S_2$ in $\mathcal{G}_{sm}$ (appropriate $x_1, x_2, \ldots, x_n$ n at step 1,
    2, 3…, $t$)
6   Delete all vertices in $V_2$ that violate $\mathcal{E}$
7   Add edges according to the transition relationship in $\mathcal{G}$ in $V_2$
8   For each edge $e$ in $\mathcal{G}_{sm}$ do
9           if $e$ has a beginning vertex at $S_1$ and ending vertex at $S_2$ do
10                  Add an edge from $V_1$ to $e$'s label in $V_2$
    Repeat line 4 - 10 to construct vertices and edges for all the states in $\mathcal{G}_{sm}$
```

For each vertex in $\mathcal{G}_{tot}$, it has a location ID, time, and state. In a multigraph, location ID can be repeated in each subgraph, but the combination of the location ID, time, and state, is unique. When construct vertices for one state (Algorithm 2, line 4), include every discrete cell, except the cells that are the ending vertices of that state's outgoing edges to other states. Edges within a state's subgraph follow the transition relationship in $\mathcal{G}$ (Algorithm 2, line 5). Except for the initial vertex, all the vertices must have at least one incoming edge in order to have outgoing edges. For edges between different states' subgraph, if $\mathcal{G}_{\mathrm{sm}}$ has such an edge that connects the two states, the vertices in the beginning state's subgraph can have an interstate edge to the ending state's vertices that have the same location ID as the edge's action (Algorithm 2, line 9-11). An edge can only connect vertices that have a time difference of 1. The beginning vertex's time must be smaller than the ending vertex's time (Algorithm 2, line 11). To make sure all the edges are created, building the state machine diagram requires that the state machine graph's vertices are create in such an order that a node shall not be create until all incoming edges connecting to this node and the nodes where these incoming edges come from have been created. Otherwise, some interstate edges may not be created.

3.5.2 Retrieve Shortest Path

According to the state machine diagram $\mathcal{G}_{\mathrm{sm}}$, we can easily determine the accepting state subgraph in the multigraph $\mathcal{G}_{tot}$. Each vertex in the accepting state's subgraph means all the mission specifications have been completed. Hence, the vertex with the shortest time in the accepting state's subgraph is the ending vertex in the shortest path we are looking for. To find the shortest path, we can backtrack vertices in the incoming edges of the ending vertex. By repeating the backtracking step, we can find the entire shortest path. One advantage of this planning method is that we can easily find all the possible paths within the preset total number of steps in the multigraph.

3.6 Overview – The Random Walk Method

The overall approach of the Random Walk method uses the idea of random walk to generate random paths. It uses the workspace information $\mathcal{W}$, mission specifications ϕ, and environment information $\mathcal{E}$ to make a state machine diagram $\mathcal{G}_{sm}$ as in Multigraph Network Planning. Among the valid paths found by the random path generator and verified in the workspace according to the state machine diagram $\mathcal{G}_{sm}$, we can choose the shortest path.

Algorithm 3: StateMachineChecker ($\mathcal{G}, \mathcal{G}_{sm}, \phi, \mathcal{E}, from$)

Input: Workspace graph $\mathcal{G}$ and state machine graph $\mathcal{G}_{sm}$, global mission specification ϕ, environmental specifications $\mathcal{E}$, initial location name *from*

Output: Shortest path that fulfills the global mission, SP

```
1  StateMachineChecker ($\mathcal{G}, \mathcal{G}_{sm}, \phi, \mathcal{E}, from$):
2          Assign a cell in $\mathcal{G}$ to be the initial location based on $from$
3          Add $from$ to the list of location names, onPath
4          Find the vertex $f$ in $\mathcal{G}$ with the location name $from$
5          CurrentState = the initial state $S_i$ in $\mathcal{G}_{sm}$
6          Call RandomWalker (Algorithm 5) for desired times to find the possible
           shortest path
7          Find the shortest path by running this algorithm multiple times
```

3.6.1 Random Walk

This RandomWalker Algorithm walks on the State Machine Diagram to ensure that the output path fulfills the global mission specification ϕ. This is because that each location to where the vehicle moves follow the State $\mathcal{G}_{sm}$ according to ϕ. The Algorithm

4 is the random action generator. The *listActions* is a list of actions the vehicle can undergo from its current location. The *listEdgeGsm* is a list of edges from the vehicle's current state in the state machine diagram. The action generated by Algorithm belongs to both the *listActions* and the *listEdgeGsm*.

```
Algorithm 4: randomAction ($\mathcal{G}, \mathcal{G}_{sm}, CurrentState, from$)
Input: Workspace graph $\mathcal{G}$ and state machine graph $\mathcal{G}_{sm}$, current state (CurrentState),
initial action from
Output: random action, $x_c$

randomAction ($\mathcal{G}, \mathcal{G}_{sm}, CurrentState, from$):
  1  listActions = from.getOutgoingEdgeLabel()
     // find all the possible actions the vehicle can take after action from
  2  listEdgeGsm = all the valid edges from CurrentState
  3  Randomly choose $x_c$ from listEdgeGsm's labels
  4  while ($x_c$ is not in listActions) do
  5        Randomly choose another $x_c$ from listEdgeGsm's labels
  6  Return $x_c$
```

We record the state S_k of the vehicle for each action it takes ($x_n \in \mathcal{A}_{S_k}$). When it reaches the accepting state, the program has found a path, where $path \models \phi$. For each x_n, the algorithm checks it against the environmental specifications $\mathcal{E}$. If x_n violates $\mathcal{E}$, we will take another random action where x_n ($\in \mathcal{A}_{S_k}$).

```
Algorithm 5: RandomWalker ($from, SP$):
Input: initial action from, previous Shortest path that fulfills the global mission, $SP$
Output: Shortest path that fulfills the global mission, $SP$
1    RandomWalker ($from, SP$):
```

```
2     Add $from$ to *onPath*
3     if $from$ violate $\mathcal{E}$
4             *Obs* = true // Obs is Boolean to record whether $from$ hit the
              obstacle
5     else
6             *Obs* = false
7     Remove $from$ to *onPath*
8     if current state is the same as accepting state in $\mathcal{G}_{sm}$ && *Obs* ==false do
9         if the verified path = null
10                *onPath* is the current shortest path, $SP$
11                *CurrentState* = the initial state $S_i$
12        else
13                Compare the length of *onPath* and $SP$
14                $SP$ = the path with shorter length
15                Reset *onPath* with only the starting cell X1
16                Return $SP$
17    else
18        randomAction $x_C$ ($\in \mathcal{A}_{S_i}$) (Algorithm 4)
          // $x_c$ is any random label of the edges from $from$ to $C$
19        Repeat line 3-7 to check $x_C$
20        If *Obs* ==true
21                *Obs* = false;
22                return RandomWalker(the last action $x_n$ on *onPath*, $SP$);
23        else
24                *Obs* ==false
25                If $x_C$ cause a change in state based on $\mathcal{G}_{sm}$ from $S_i$
27                        Set the current state to the new state, $S_j$
28                Add $C$ to *onPath*
29                Return RandomWalker($C, SP$)
```

3.6.2 Finding shortest path and Pros & Cons

With the random walk algorithm, the computer makes random choices resulting in verified paths with varying lengths. Each time we find a path that fulfills all the mission requirements under the environment specifications, we say that this path is our potential shortest path SP. We can run the random walk search algorithm many times and update SP with shorter verified *onPath*.

The biggest advantage of this approach is the minimum programming effort. It does not require a Multigraph for each state. Hence it does not require extensive programming time to identify a path that fulfills all the mission requirements under the given environment specifications.

The randomization means it is possible for the vehicle to have a minimal amount of random walk searchers; however, since the vehicle is programmed randomly, it may end up taking excessive steps to find its verified path. The worst case is that during one random walk search, the computer may never find a verified path before StackOverflow Error (for Java). Therefore, as $\mathcal{G}$ and ϕ get larger, it is more likely to take more random walk searches to find the shortest path.

3.7 Overview – The Potential Field Method

Comparing to the Random Walk method, the Potential Field method uses the potential field to guide the vehicle instead of total random walk. It uses the workspace information $\mathcal{W}$, mission specifications ϕ, and environment information $\mathcal{E}$ to make a state machine diagram $\mathcal{G}_{sm}$ as in Multigraph Network Planning. In addition, we add the potential field to the $\mathcal{G}_{sm}$. Like Random Walk method, we have a path generator to generate and verify paths in the workspace according to the state machine diagram $\mathcal{G}_{sm}$. When the generator chooses which action to take, the discrete cells with larger total force have higher priority than the cells with smaller total force. Among the verified paths, we can find the shortest path. Ge and Cui in [21] defined that the total force, $F_{total}(q)$, which is applied to the vehicle, is the sum of the attractive force F_{att} and the repulsive force F_{rep}.

3.7.1 Create Potential Field

The Potential Field method uses Algorithm 5 to find the potential shortest path, but in **line 18**, instead of Algorithm 4, Algorithm 6 is used to find *listG*, a list of vertices around the current location with Potential Field information. Based on [25] and [23], we need to define a coordinate system for cells in the workspace. In this thesis, we define the coordinate system as shown in Figure 3-3. The increment is based on the x-axes and y-axes. We always set the starting location as X10. 0 is the y-axis value and 1 is the x-axis value. Because we assumed that the vehicle can move to any cell along the blue arrows in one step, we do not consider the velocity of the vehicle or actual distance between the cells. If it takes one step to move to the next cell in the x-axis direction, that cell is X20.

If it takes one step to move to the next cell in the y-axis direction, that cell is X11. If it goes both on the x-axis and y-axis positive direction, that cell is X21.

Based on the attractive and repulsive force equation in [23], we can calculate

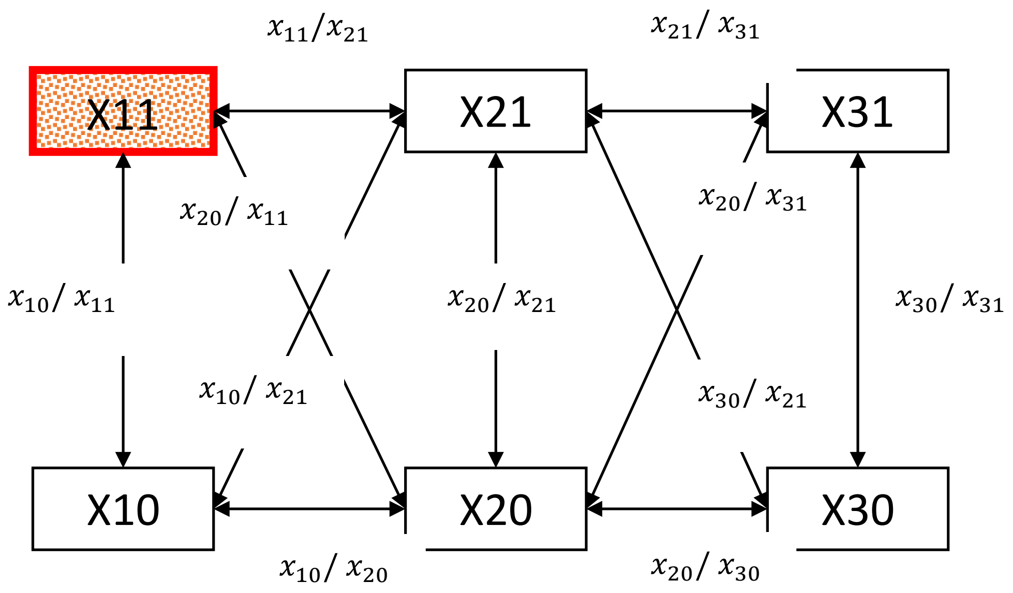


Figure 3-3: Workspace example (Example 1) for State Machine Diagram Potential Field Planning with coordinates. X10, where 0 is the y-axis value and 1 is the x-axis value.

potential field $U_{total}(q)$ for a vehicle *q*, in our coordinate system, *q = (x, y).*

$$U_{total}(q) = U_{att}(q) + U_{rep}(q) \qquad \text{(Eq. 3.4.1.1)}$$

$$U_{att}(q) = \frac{1}{2}k_a\left(q - q_f\right)^2 \qquad \text{(Eq. 3.4.1.2)}$$

$$U_{rep}(q) = \begin{cases} \frac{1}{2}k_r\left(\frac{1}{\rho} - \frac{1}{\rho_0}\right)^2 & if\ \rho \le \rho_0 \\ 0 & if\ \rho > \rho_0 \end{cases} \qquad \text{(Eq. 3.4.1.3)}$$

In Eq. 3.4.1.1, the potential field $U_{total}(q)$ comprises two terms, the attractive potential function $U_{att}(q)$ (Eq. 3.4.1.2), and the repulsive potential function $U_{rep}(q)$ (Eq.3.4.1.3). The start position q_0 is the vehicle's current position, q_f is the target position, and k_a and k_r are scalar variables. The obstacles have their position $O_1, \dots, O_n$, and the limited distance of influence of the potential field ρ_0. If the vehicle is further away than ρ_0, the obstacle does not influence the vehicle's current path plan. The value ρ is the shortest distance from the vehicle to the obstacle. Velocity is not considered in this thesis. In this *listG*, the next action x_c is to where the potential field has the total force $F_{total}(q)$.

As mentioned earlier, the total force, $F_{total}(q)$ is the sum of the attractive force F_{att} and the repulsive force F_{rep}:

$$F_{total}(q) = F_{att} + F_{rep} \qquad \text{(Eq. 3.4.1.4)}$$

$$F_{att} = -\nabla U_{att} = k_a(q_f - q) \qquad \text{(Eq. 3.4.1.5)}$$

$$F_{rep}(q) = -\nabla U_{att}$$

$$= \begin{cases} k_r\left(\frac{1}{\rho} - \frac{1}{\rho_0}\right) * \left(\frac{1}{\rho^2}\right) * \nabla\rho & if \rho \le \rho_0 \\ 0 & if\ \rho > \rho_0 \end{cases} \qquad \text{(Eq. 3.4.1.6)}$$

According to the Eq. 3.4.1.5, the attractive force F_{att} is a negative gradient function of the attractive field and converges to zero as the robot approaches the target. According to Khatib [20], $\nabla\rho = \frac{\partial\rho}{\partial q}$, denotes the partial derivative vector of the distance from the vehicle to the obstacle.

$$\frac{\partial\rho}{\partial q} = \left[\frac{\partial\rho}{\partial x} \ \ \frac{\partial\rho}{\partial y}\right]^T \qquad \text{(Eq. 3.4.1.6)}$$

3.7.2 Finding the shortest path

Since one mission may have multiple mission requirements, and there can be a sequence of requirements about which target to visit first, the targets in the potential field change according to the state machine diagram, current location of the vehicle, and the current state of the vehicle and the appearance of any dynamic obstacles. To find the shortest path, the goal is to transit from the initial state to the end state with the least amount of transition. Therefore, we need a potential field for each state. The targets of each state's potential field are the labels of that state's outgoing edges. Meanwhile, the environmental specifications apply to each state. Therefore, based on the current time of the path, the obstacles may create a different repulsive force on the vehicle.

For the discrete cells in the workspace that are far away from the targets or obstacles, they should not be set to normal. The affecting ranges of targets and obstacles depend on the problem. The value *normal* is a score set by the researcher depending on the size of the problem. It should be relatively higher than the attraction scores but lower than repulsive scores.

In addition, it will not be efficient to calculate potential field for the whole workspace for each step the vehicle takes, especially if the workspace has more than 9 discrete cells in the workspace. This is because the discrete cells do not have an impact on the vehicle's path plan if the discrete cells are beyond the reach of the vehicle from its current location. Therefore, in Algorithm 6, we only calculate potential field for the discrete cells that the vehicle can physically move to in the next step. In line 8, *Uatt* calculates the attractive force using Eq. 3.4.1.5, and in line 10 *Urep* calculates the repulsive force using Eq. 3.4.1.6.

Algorithm 6: PotentialField ($\mathcal{G}, \mathcal{G}_{sm}, currentloc, CurrentState, time, \mathcal{E}$)

Input: Workspace graph $\mathcal{G}$ and state machine graph, $\mathcal{G}_{sm}$, current location, *currentloc,* current state, *CurrentState*, current time, *time*, environmental specifications, $\mathcal{E}$

Output: *listG*, list of vertices around current location with Potential Field information

PotentialField ($\mathcal{G}, \mathcal{G}_{sm}, CurrentState, from$):

```
1   listG ← currentloc.getOutgoingEdge().getTo()  // for each outgoing edge of
    currentloc
    // Find all the possible physical cells where the vehicle can reach in one step
    from its current location
2   listEdgeGsm ← $\mathcal{G}_{sm}$.findVertexByState(CurrentState)
    // find all the possible edges from CurrentState in $\mathcal{G}_{sm}$
3   list ←  edge.getLable() // for each edge in listEdgeGsm  get its label
4   For each Vertex v in listG do
5       v.setScore(normal)      // Set scores to normal when movement to theta
                                cell does not lead to another state
6           For each edge e in listEdgeGsm
7               If e links two different state in $\mathcal{G}_{sm}$
8                   score ← Uatt(v, e.getLabel())  // Uatt is attractive potential
9                   If v.getScore()>score then v.setScore(score)
10      v.setScore(-1*v.getScore() + Urep(v))  // Urep is repulsive potential
11      For each vertex $x$ in listG
12          if $x \notin$ list then remove $x$
13  Return listG
```

Algorithm 7 calculates an attractive potential score att, which will be further processed in Algorithm 6 for the correct direction pointing to the target.

Algorithm 7: *Uatt(v, e.getLabel())*

Input: Vehicle current location's name *v*, target vertex (target action for the vehicle to take) *e.getLabel()*

Output: Negative attractive potential att

```
Uatt(v, e.getLabel()):
1   Obtain coordinates of the vehicle current location: x1, y1
2   Obtain coordinates of the vehicle's target: x2, y2
3   $k_a$ ← 1
4   att = 0.5*ka*($(x2 - x1)^2 + (y2 - y1)^2$)
5   Return $att$
```

Algorithm 8 calculates a repulsive potential force rep, which is used in Algorithm 7 for the total potential force.

Algorithm 8: *Urep (v, e.getLabel())*

Input: Vehicle current location's name *v*, vertex with obstacle area alert, *e.getLabel()*

Output: Repulsive force rep

```
Urep(v, e.getLabel()):
1   Obtain coordinates of the vehicle current location: x1, y1
2   $k_r$ ← 1, $\rho_0$ ← 2
3   for each obstacle area $O_n$ do (n =1, 2, 3…)
4           if the obstacle area is in alert do
5               Obtain coordinates of the obstacle area: x2, y2
6               $\rho$ ← sqrt ((($(x2 - x1)^2 + (y2 - y1)^2$)
7               If $\rho < \rho_0$ do
8                   $Frep_n = 0.5 * k_r * \left(\frac{1}{\rho} - \frac{1}{\rho_0}\right) * \left(\frac{1}{\rho}\right)^2$ (n =1, 2, 3…)
9               else
10                  $Frep_n = 0.0$ (n =1, 2, 3…)
11  $rep = \sum Frep_n$ (n =1, 2, 3…)
12  Return $rep$
```

3.8 Overview – Critical Path Method

Like the combination of the state machine diagram and the potential field path planning method, the Critical Path method combines the state machine diagram with D* Lite. Koenig and Likhachev [10] presented D* Lite method and demonstrated that this method can plan paths for robots in an unknown environment with dynamic obstacles. This Critical Path method uses Java code written by Beard [25] in DStarLiteJava (https://github.com/daniel-beard/DStarLiteJava) to obtain the shortest path between two cells in the workspace. The state machine diagram guides the vehicle to move in the way that fulfills the mission requirements and environmental specifications.

3.8.1 D* Lite

According to Koenig and Likhachev [10], D* Lite repeatedly calculates the shortest path from its current location to the target, based on the dynamic obstacle information. Therefore, based on the given dynamic obstacle information, when the vehicle reaches a new state in the state machine diagram, D* Lite can plan a shortest path to the next state based on the current time and dynamic obstacle information. D* Lite also allows us to set permanent obstacles in the workspace. The Critical Path method used this feature to set the boundary of the workspace and permanent obstacles that exist in the mission.

Based on Beard's code [26], we use the x and y axes coordinate system that is also used in the Potential Field method, where the workspace's most bottom left cell is X10. And we assume that the vehicle always starts from X10, as shown in Figure 3-4.

To set the boundary of the workspace, we assume that for each mission, there are permanent obstacles are the cells around the workspace in the coordinate system with

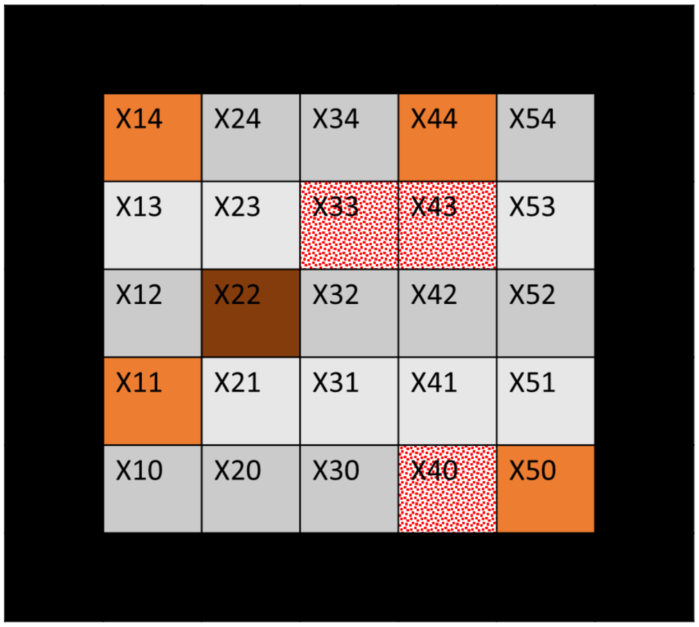


Figure 3-4: Coordinate system for the Critical Path System

the block color. Orange and brown colored cells are involved in the mission requirements. Brown colored cell means that the target needs to be visited in a certain order. Red large confetti pattern means these areas may have obstacles at certain time according to the environmental specifications.

An example of D* Lite implementation using the workspace of Example 2 (B) (Section 4.3) is shown in Figure 3-5 and Figure 3-6 for understanding the planning algorithm. The vehicle planned its path X6, X7, X8, X4, and X5, where X6 was the target cell for the state S3, and X5 was the target cell for the state S18 in Figure 4-13. However, when the vehicle reached X8, by checking the dynamic obstacle information from the environmental specifications, it realizes that during the next step, an obstacle may show up in X4. Hence, D* Lite planned a new path as shown in Figure 3-6.

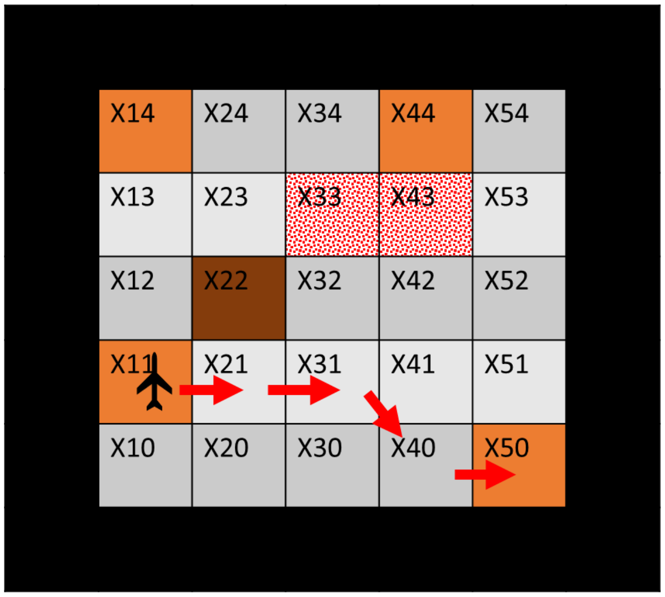


Figure 3-6: Updated path from S3 to S18 planned when the vehicle detected an obstacle in X4

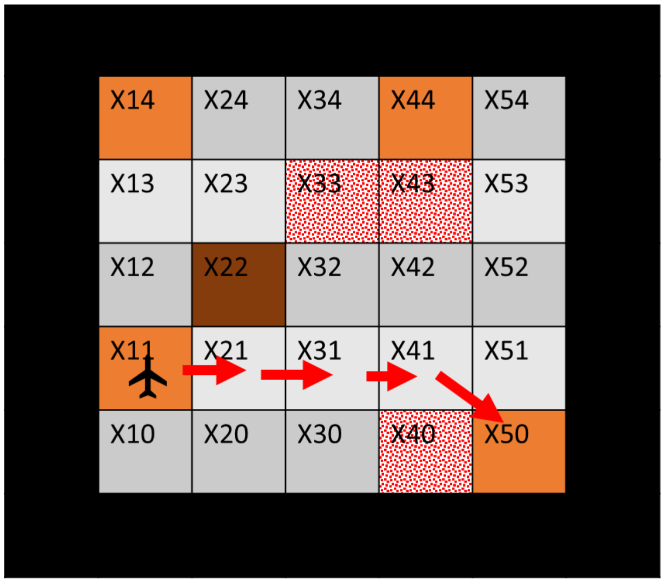


Figure 3-5: Original path from S3 to S18 (X6 to X5) planned when the vehicle arrived at X6

3.8.2 Find the Shortest Path on the State Machine Diagram

The use of the state machine diagram is to guide the vehicle to fulfill the mission requirements by reaching the accepting state. To find the shortest path from the initial state to the accepting state, we need to know the cost (path length) of each edge in the state machine diagram. However, unlike the traditional path planning problem, for example, the state S5 from Example 1 in Figure 3-7, its arrival time may vary depending on which state the vehicle comes from and where the vehicle comes from in the physical workspace. In Figure 3-7, we define that a state has its arrival time, t_{ijk} ..., where i, j, k, …, records the previous state the vehicle has been to.

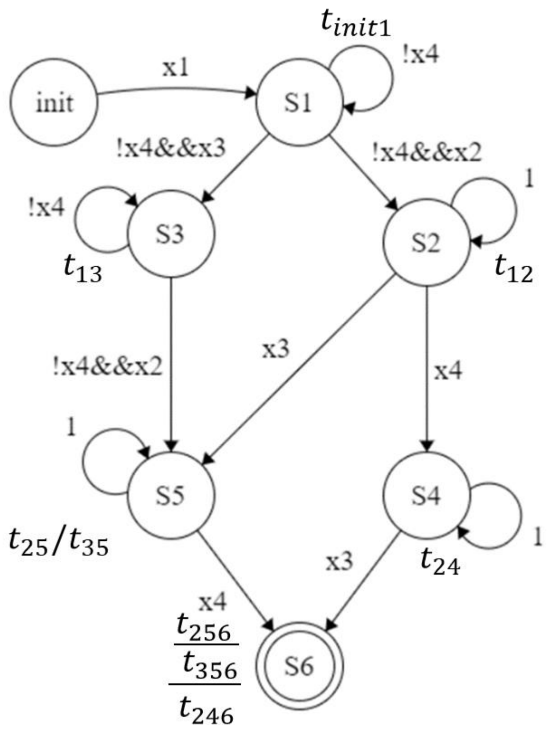


Figure 3-7: State machine diagram of Example 1

Unlike traditional path planning problem that requires path planning in the physical workspace, each edge in the state machine diagram have multiple cost. This is because there may be different ways to arrive at the state in the physical workspace and hence different ways to transit to the next state. For state S5, the vehicle may come from X3 or X2, resulting two arrival time t_{25} and t_{35}. Furthermore, because the arrival locations in S5 are different, when we calculate the arrival time for S6, when cannot pick the smaller arrival time in S5. Because in the physical workspace, the arrival time of X2 may be different from the arrival time of X3, and the paths from X2 to X4 and from X3 to X4 will be different. One thing to notice here is that maybe t_{25} is small than t_{35}, that does not conclude that t_{256} will also be small than t_{356}. Therefore, from S5 to S6, we will have two different arrival time t_{256} and t_{356}. To find the shortest path from the initial state to the state S6, we need to compare t_{256}, t_{356}, and t_{246}.

Algorithm 9 calculates all the states' arrival time and paths between the states. HashMap is used as the output for retrieving the path. Algorithm 9 requires that the state machine graph's nodes are created in such an order that a node shall not be created until all its predecessors have been created. Because when we use D* Lite to plan the path for an edge, the starting time of the path is required for dynamic obstacle avoidance. When planning a path for a state's outgoing edge, all of that state's incoming edges' information is required.

Algorithm 9: *CriticalPathPlan (v, e.getLabel())*

Input: Workspace graph $\mathcal{G}$ and state machine graph $\mathcal{G}_{sm}$, current time, *time*, environmental specifications $\mathcal{E}$

Output: Update HashMap *list*, which include the transition relationships between each pair of connecting state and the arrival time of the ending state;

Update HashMap *map*, which records the shortest paths in the physical workspace for each transition between connecting states in *list*

CriticalPathPlan (v, e.getLabel()):

```
1   for each node $v$ in $\mathcal{G}_{sm}$ do
2       if the node is the initial state in $\mathcal{G}_{sm}$ do
3           Record $v$'s arrival information to a HashMap *origins (*arrival
            information includes: physical starting cells name (x00, a dummy
            node), state name (init), and starting time as the value (n/a, because
            this is a dummy node) as the key in *origins*, and the arrival time to
            $v$ (0, dummy time) as the value to the key)
4           Add $\mathcal{G}_{sm}$ starting and ending node state name – init to S1, path
            starting and target cell name (X00 to X10), and total path cost (1)
            as a key in HashMap and vehicle path in the workspace (go from
            X00 to X10) as the key's value to *map*
5       else
6           for each incoming edge *eIn* of the node $v$, do
7               if $e$'s starting node *preV* has only one incoming edge do
8                   Record $v$'s arrival information to a HashMap *origins*
                    *(*physical starting cells name, state name, and starting time
                    as the value as the key in *origins*, and the arrival time
                    *arrivalTime (*cost of *eIn)* to $v$ as the value to the key)
9               else
10                  for each incoming edge *preE* of *preV* do
11                      if there is only 1 vehicle path in the workspace for *eIn*
12                          Record $v$'s arrival information to a HashMap
                            *origins. arrivalTime* = length of *preE* + length of
                            *eIn* - 1
13                      else
14                          For each vehicle path $p$ in the workspace for *eIn*
                            do
15                              if *eIn*'s starting location in workspace ==
                                $p$'s ending location in workspace, do
16                                  Record $v$'s arrival information to a
                                    HashMap *origins. arrivalTime* = cost
                                    of *preE* + cost of $p$ -1
17          For each arrival information key, $k$, in the *origins* do
18              Obtain coordinates of the vehicle starting location: *x1, y1*
19              For each outgoing edge *eOut do*
20                  Obtain coordinates of the vehicle's target: *x2, y2*
```

```
21    Add obstacles in the workspace based on the state machine diagram $\mathcal{G}_{sm}$ and the environmental specifications $\mathcal{E}$
22    Use D* Lite to plan a path from (x1, y1) to (x2, y2), path
23    Total path length for path from original starting point = origns.get(k)+(path.size()-1)
24    Record path in e
25    Add transition information to list ( $\mathcal{G}_{sm}$ starting and ending node state name – preV to v, and path starting and target cell name as the key in list, and path length calculated by D* Lite as the value of the key)
26    Add $\mathcal{G}_{sm}$ starting and ending node state name –preV to v, path starting and target cell name, and total path cost (1) to v as a key in HashMap and vehicle path in the workspace as the key's value to map
```

To find the shortest path, simple start from the smallest arrival time of the accepting state and retrieve the whole path based on the arrival time and physical locations of the vehicle in its previous states. One thing to notice here is that one state may have multiple incoming edges but the target location of some of these edges may be the same. When retrieving the whole path, we should not only check one state's direct incoming edges, but also which state this incoming edge is from.

Chapter 4 Implementation and Results

While the algorithms are applicable to myriad vehicle types, this research recorded the movements of unmanned aerial vehicles (UAV). This chapter shows 5 examples of UAV missions that implement the linear temporal logic (LTL) and four planning methods, the Multigraph Network Planning method, the Random Walk method, and the Potential Field method and the Critical Path method. The programs were run on an Intel CORE i5-3337U processor @ 1.8 GHz. The system had 256MB of RAM. In the experiment, most of the time the Random Walk method took much longer to find a path that had the same length as the other two method's results. Because more random walk searches would eventually yield a more optimized path, this thesis compares random walk searches up to 100,000 times for any complex examples using the Random Walk method and use that result to compare with other path planning methods.

4.1 Example 1

A UAV shall complete its mission requirements in the workspace shown in Figure 4-1**.** The UAV starts from Site X1. One environmental specification is that Site X4 may have several storms from step 4 to 5. Mission requirements included:

1. Go to X3 to drop food.
2. Go to X4 to pick up patients.
3. Before the UAV goes to X4, the UAV must go to X2 to pick up a nurse.

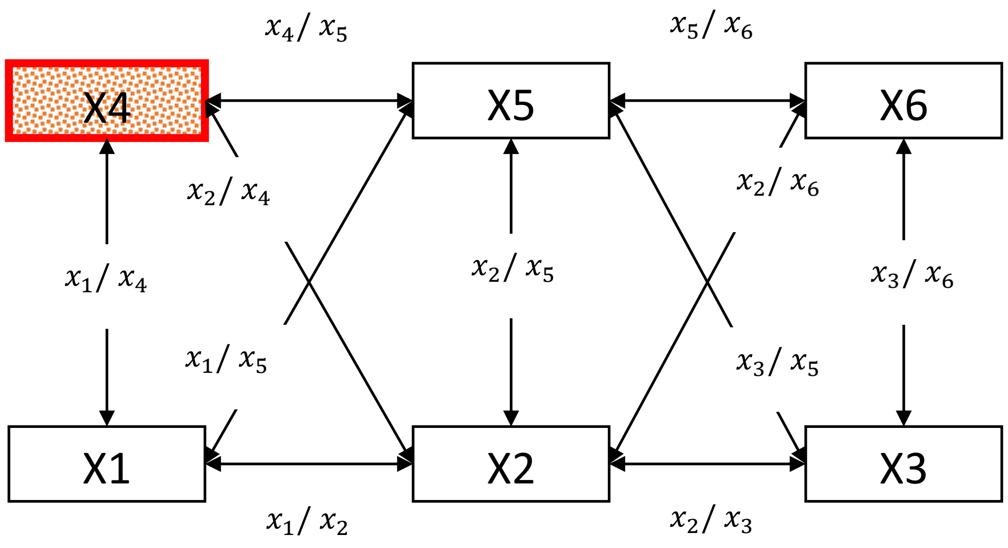


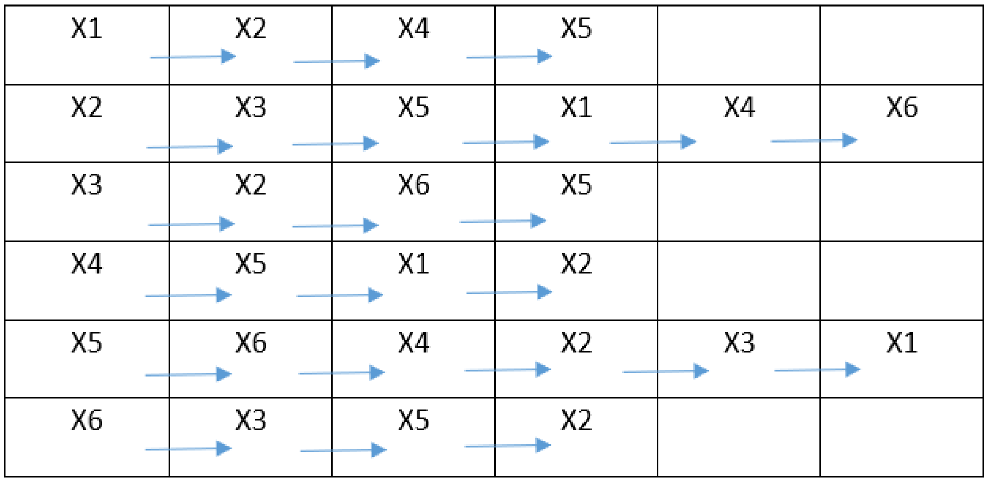
Figure 4-1: Example 1 workspace. A workspace with 6 stations that UAVs can visit. X4 may be blocked by moving obstacles during certain period. UAVs can fly according to the actions between stations (i.e. x1, x2, and x3).

4.1.1 Workspace Representation

As shown in Figure 4-1, the workspace had been partitioned into 6 rectangular, X1, X2, X3, X4, X5, and X6. The UAV could move forward, backward, diagonally, or stay at the same discrete cell. Figure 4-2 shows the transitional relationship in the form of a linked list.

The goal was to find the shortest path *SP* for a UAV so that it fulfilled the operation goals. In this example, the vehicle shall start from X1. Because it needed time to take off, we assumed that it took one step to finish taking off at X1. Then, the vehicle shall visit X2 before it can visit X4. And the vehicle shall visit X3 and X4 at least once on its path. In addition, X4 will not be available at step 4 to 5. LTL cannot directly express all the operation rules as stated in Section 1.1.3 due to the explicit time specification.

Hence, this specification was expressed using the extended LTL according to Maity and Baras [17].



X1	X2	X4	X5		
X2	X3	X5	X1	X4	X6
X3	X2	X6	X5		
X4	X5	X1	X2		
X5	X6	X4	X2	X3	X1
X6	X3	X5	X2		

Figure 4-2: Edge between each discrete cell representation by the linked list

Using the extended LTL to translate the mission requirements ϕ_1, there were three LTL rules, and one extended LTL rule for the dynamic obstacles.

- x2 B x4
- F x3
- F x4
- !x4$_{[4,5]}$ – Extended LTL

4.1.2 Linear Temporal Logic to State Machine Diagram

According to the physical constraints given in Figure 4-2, we could reconstruct the State Machine Diagram based on the original Büchi Automaton. LTL2BA tool developed by Gastin and Oddoux [19] (http://www.lsv.fr/~gastin/ltl2ba/index.php) created the Büchi Automaton for the first three LTL rules in Figure 4-3 and Figure 4-4.

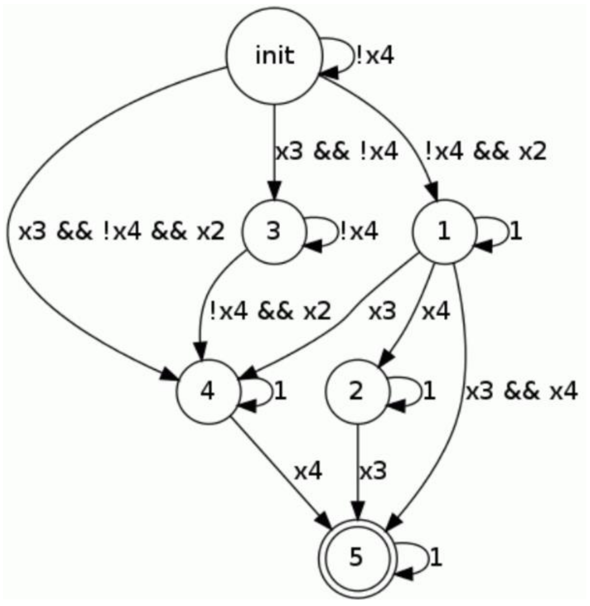


Figure 4-3: Example 1 Büchi Automaton

```
never { /* (F x3) && (F x4) && (x2 R !x4) */
T0_init :    /* init */
        if
        :: (!x4) -> goto T0_init
        :: (!x4 && x2) -> goto T0_S2
        :: (x3 && !x4) -> goto T1_S3
        :: (x3 && !x4 && x2) -> goto T1_S4
        fi;
T0_S2 :    /* 1 */
        if
        :: (1) -> goto T0_S2
        :: (x4) -> goto T0_S5
        :: (x3) -> goto T1_S4
        :: (x3 && x4) -> goto accept_all
        fi;
T0_S5 :    /* 2 */
        if
        :: (1) -> goto T0_S5
        :: (x3) -> goto accept_all
        fi;
T1_S3 :    /* 3 */
        if
        :: (!x4) -> goto T1_S3
        :: (!x4 && x2) -> goto T1_S4
        fi;
T1_S4 :    /* 4 */
        if
        :: (1) -> goto T1_S4
        :: (x4) -> goto accept_all
        fi;
accept_all :    /* 5 */
        skip
}
```

Figure 4-4: Büchi Automaton Transition Relationship

Note that !x4 meant any action other than move to the cell X4. One (1) was the power set of the possible actions set from the state are allowed to be taken for any number of times. The translation steps were as follows:

- Create a dummy initial state "init" to represent the state when the vehicle is at rest. This is different from the "init" state in the Büchi Automaton
- Based on the purpose of finding the shortest path, we assume that for any edge in the state machine diagram, when an action x_k ($x_k \in \mathcal{A}_{s_k}$) is combined with ! x_n, ! x_n means do not do x_n and the end location of that edge shall be x_k.
- Eliminate the edges that require more than one end location in the physical workspace.
- When there is only !x_4, it means the UAV can conduct any possible action from its current state other than x_4. This means the UAV can go to any possible cell space that is reachable from its current cell space but cannot go to X4 in the next step.
- One (1) means the all the singular action. For example, if $\mathcal{A}_s = \{x2, x3\}$, x2 && x3 is not allowed. Only $\emptyset, x2, and\ x3$ is allowed.

As shown in Figure 4-5 the Büchi automaton was translated into a state machine diagram.

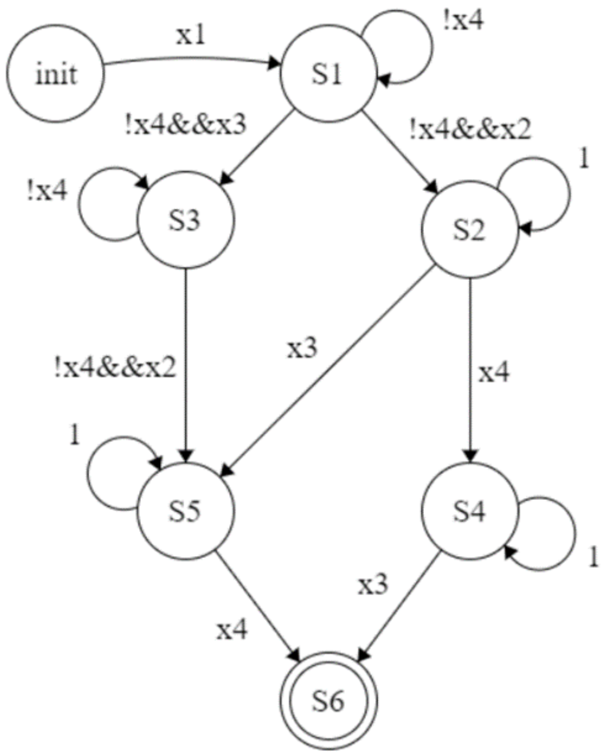


Figure 4-5: Example 1 State Machine Diagram

4.1.3 The Multigraph Network Planning Implementation

This section shows how example implements the Multigraph Network Planning algorithm. Figure 4-6 is a simplified Multigraph for this problem. Each node was named as “X#, #”. The first “X#” represents the physical discrete cell, and the second number is the time. All the nodes in the accepting state (S_6) in $\mathcal{G}_{sm}$ connects to the END node. To promote a clearer understanding of Multigraph and finding a potential shortest path, Figure 4-6 does not show all the nodes created in this approach for all the steps and locations.

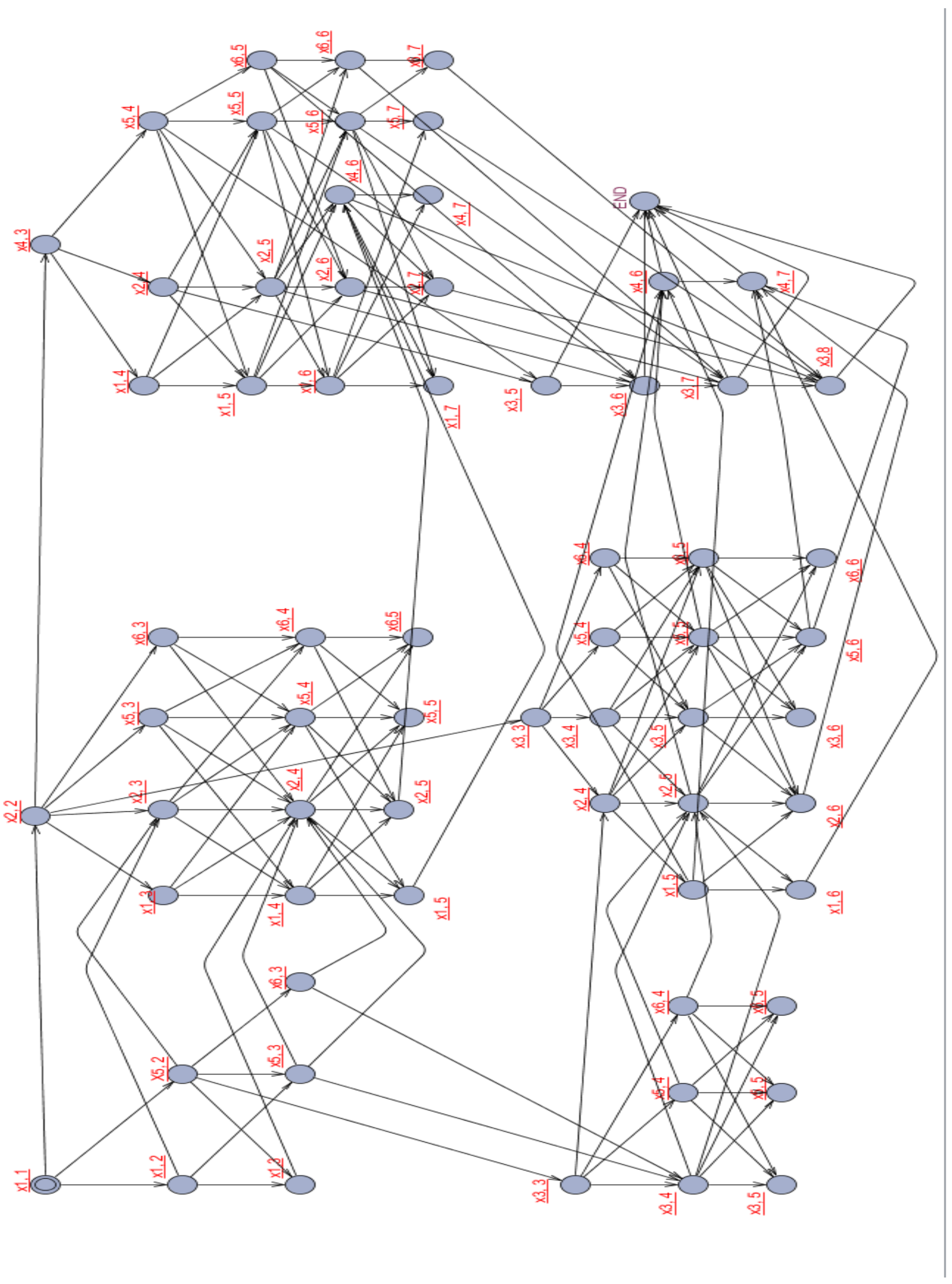


Figure 4-6 $\mathcal{G}_{tot}$, Graph combing time and state machine diagram

As shown in Figure 4-6, node X3, 5 connected to the END node and it had the smallest number of steps used. Therefore, the shortest path ended at cell X3, 5. Following the incoming edges, the program could retrieve the rest of the path. For example, from X3, 5, the previous step could be either X2, 4 or X5, 4. Randomly pick one of the cells could yield one of the shortest paths. Retrieving all the possible combinations could yield all the possible paths with the smallest number of steps.

One of the shortest paths found using Multigraph Network Planning for Example 1 is **X1, X2, X4, X2, X3**. The computational time was **16 ms**, and the program used 3,995,104 bytes of memory.

4.1.4 The Random Walk Method Implementation

4.1.4.1 Workspace Representation and State Machine Diagram

The workspace representation, the Büchi Automaton, and the state machine diagram are the same as in

4.1.4.2 Find the Shortest Path

Random Walk method does not generate the best result every time. Nevertheless, as the program conducted more random walk searches, it was more likely to generate a potential shortest path with short path length, comparing to Multigraph Network Planning's results. In this experiment, we tested the program with 50 random walk searches. Figure 4-7 shows the shortest path we found in this experiment is 5, which is the same as the Multigraph Network Planning method's result. The shortest path found was **X1, X2, X4, X5, X3**. The computational time used was **38 ms**, and the program consumed **20,659,896 bytes** of memory.

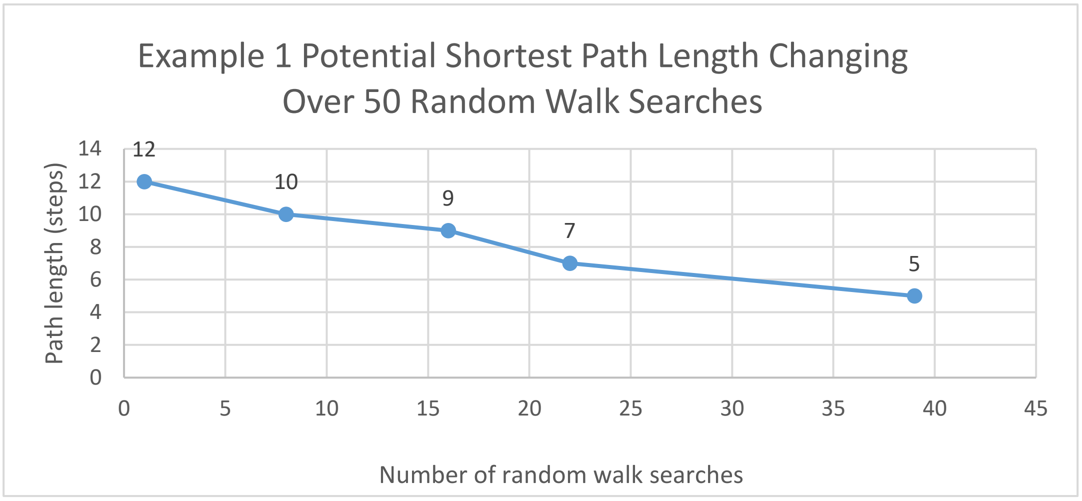


Figure 4-7: Example 1 Potential Shortest Path Length Changing Over 50 Random Walk Searches

4.1.5 The Potential Field Method Implementation

To build the potential field around the UAV, we assigned coordinates to each cell. As shown in Figure 4-8, The step distance was calculated based on the coordinates of the current UAV location and the targets. The state machine diagram was the same as Figure 4-5 in Section 4.1.2, except the labels changed according to the new names of each discrete cell in the workspace using the coordinate system. This applies to all the other examples as well.

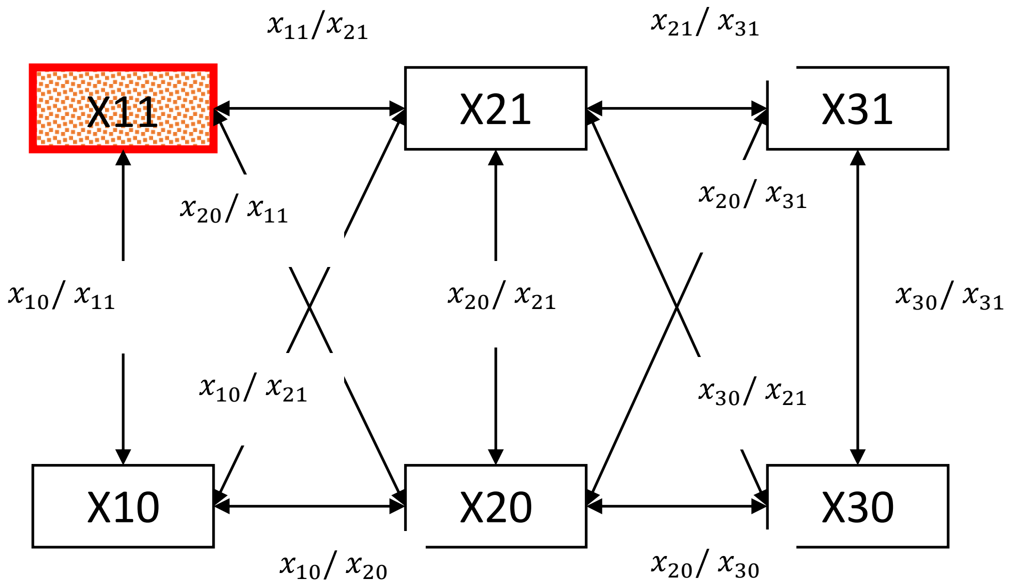


Figure 4-8: Example 1 workspace with coordinate system

The shortest path found was **X1, X2, X4, X2, X3 (X10, X20, X11, X20, X30)**. The computational time was **44 ms**, and the program used **20,667,808 bytes** of memory for 10 iterations of searches.

4.1.6 The Critical Path Method Implementation

This implementation method was explained in Section 3.8.2. The shortest path found by this method was **X1, X2, X3, X2, X4**. The computational time was **29 ms**, and the program used **19,328,560 bytes** of memory

4.2 Example 2 (A)

Example 2 has a workspace partitioned into 25 discrete cells (5 rows of 5 cells). In Example 2, the mission requirements included:

- The UAV shall visit X6, X12, and X24 at least once on its path.
- The UAV shall visit X5 before it goes to X12.

The environmental specification was that X4 have obstacles from step 4 to 5.

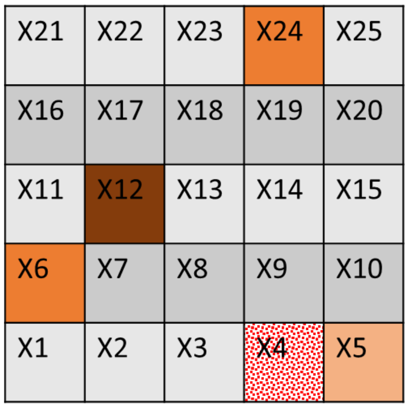


Figure 4-9 Example 2 Workspace. The transition relationship in Example 2 is that the UAV can move to up and down, left and right, and diagonally.

Hence, we can get mission requirements, ϕ_2 , and one extended LTL specifications for the dynamic obstacles, as below:

- !((!x5) U x12)
- Fx12 && Fx24 && Fx6
- !$x4_{[4,5]}$ – Extended LTL

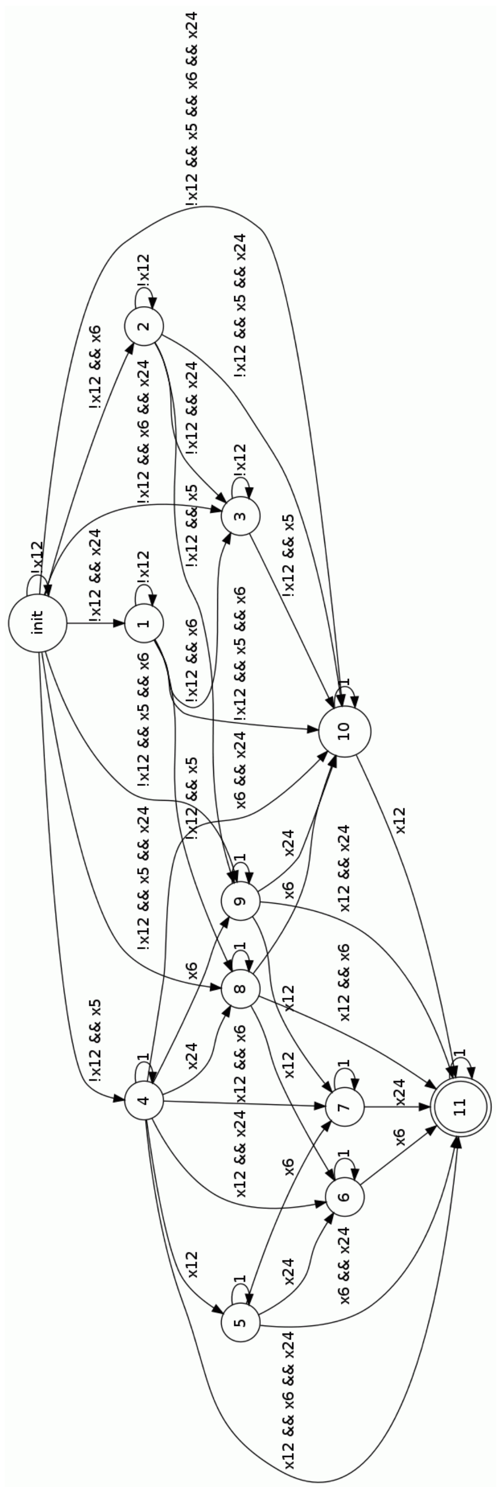


Figure 4-10 Example 2 (A) - Büchi Automaton. Refer to GIF file Figure 4-10 for a better view of this figure

LTL2BA tool (http://www.lsv.fr/~gastin/ltl2ba/index.php) [19] generated a Büchi Automaton for the first four LTL rules in Figure 4-10. Figure 4-11 is the State Machine Diagram drawn based on Figure 4-10 using [24]. Results of each method are shown in Table 4-1. The paths were shown using the workspace without coordinate system.

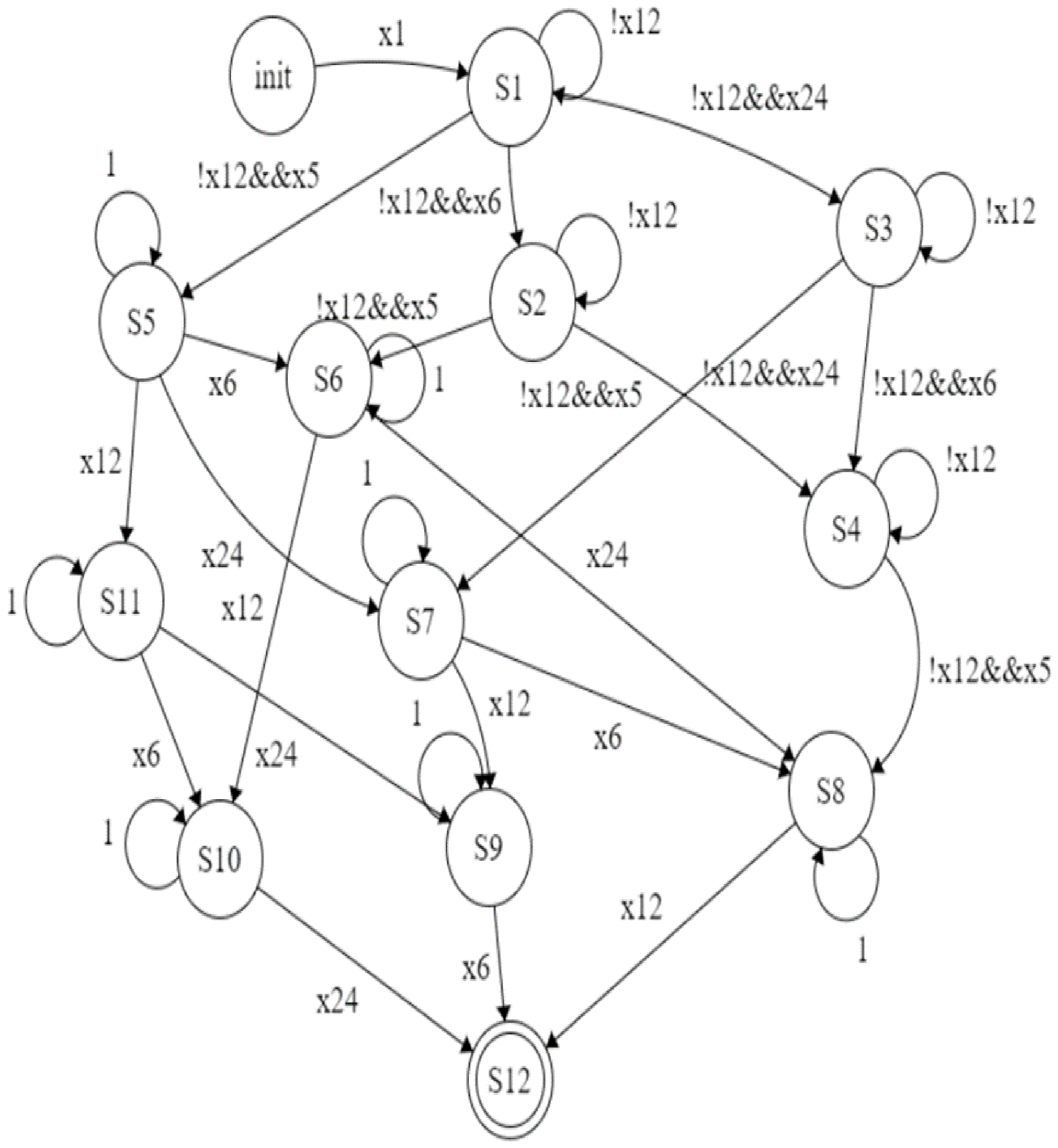


Figure 4-11 Example 2 (A) - State Machine Diagram

Table 4-1: Example 2 (A) Result sample

	Shortest Path	Path Length	Computational Effort
Multigraph Network Planning	X1, X6, X2, X8, X9, X5, X9, X8, X12, X18, X24	11	Total execution time: 979 ms Used 35,450,240 memory
Random Walk method	X1, X6, X1, X7, X8, X3, X4, X5, X4, X8, X12, X18, X24,	[205, 193, 58, 51, 38, 36, 20, 17, 14, 13]	Total execution time: 47,556 ms Used memory 51,541,792 after 100,000 random walk searches
Potential Field method	X1, X6, X2, X3, X9, X5, X9, X13, X12, X18, X24	11	Total execution time: 34 ms Used memory 20,659,592 bytes after 10 random walk searches
Critical Path method	X1, X6, X7, X8, X9, X5, X9, X8, X12, X18, X24	11	Total execution time: 34 ms Used memory 20,660,008 bytes

4.3 Example 2 (B)

Example 2 (B) had a workspace partitioned into 25 discrete cells (5 rows of 5 cells). In Example 2 (B), the mission requirements included:

- The UAV shall visit X6, X12, X21, and X24 at least once on its path.
- The UAV shall visit X5 before it is able to go to X12.

The environmental specification included:

- X18 and X19 will have obstacles from time step 6 to 10.
- X4 will have obstacles from time step 6 to 12.

Therefore, we could translate the mission requirements ϕ_4 and environmental specifications into extended LTL form as follows:

- F x6 && F x12 && F x21 && F x24
- x5 B x12
- $!x4_{[6,12]}$
- $!x18_{[6,10]}$ && $!x19_{[6,10]}$

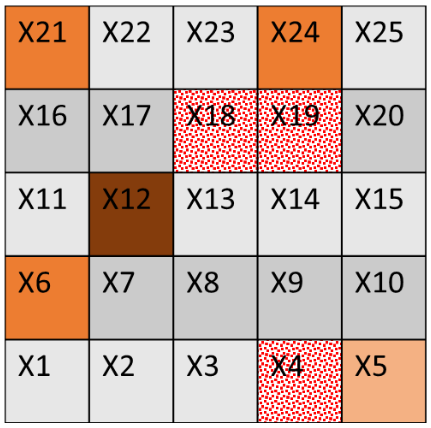


Figure 4-12:Example 2 (B) workspace

Figure 4-12 is a representation of the workspace for example 2 (B). Using mission requirements, ϕ_4 , LTL2BA tool (http://www.lsv.fr/~gastin/ltl2ba/index.php) [19] generated a Büchi Automaton for the first four LTL rules. Then, we construct the state machine diagram [24] accordingly. The state machine diagram and Büchi Automaton are shown in Figure 4-13 and Figure 4-14. The paths were shown using the workspace without coordinate system.

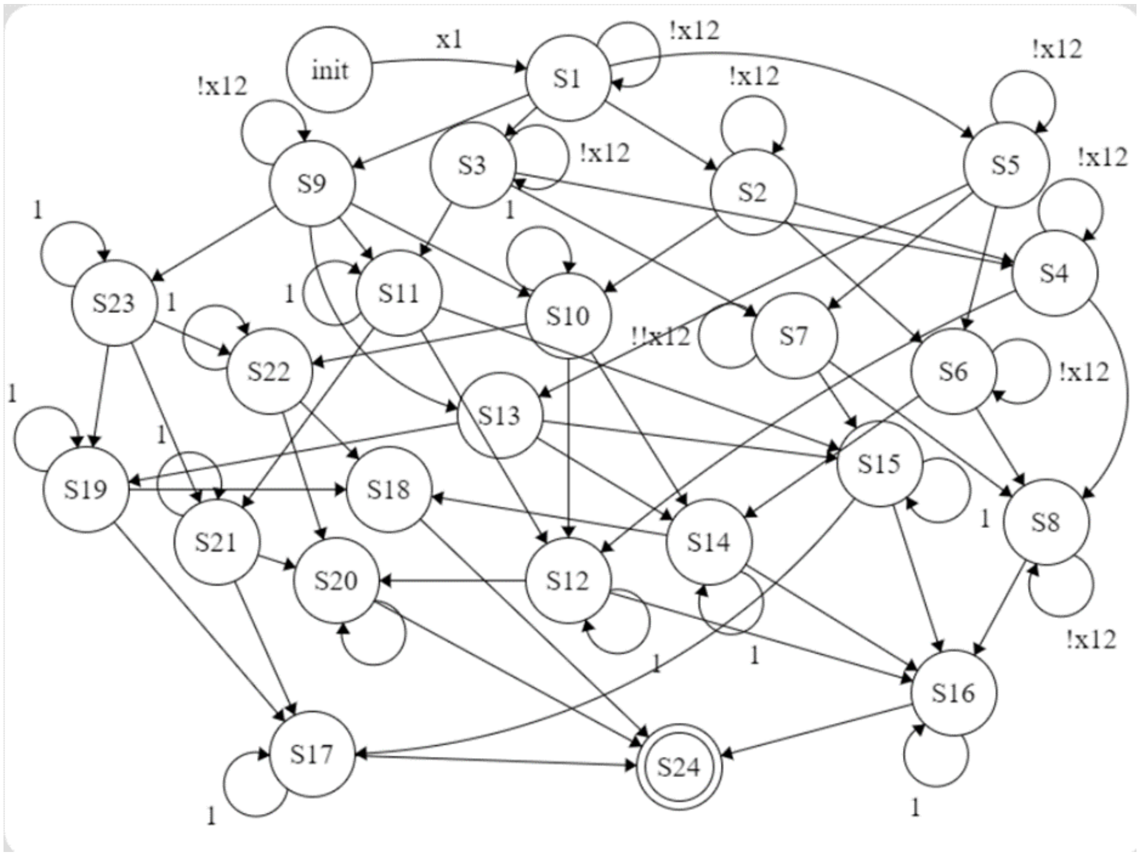


Figure 4-13: Example 2 (B) - State Machine Diagram. Labels of the edges are shown in Appendix A in text form.

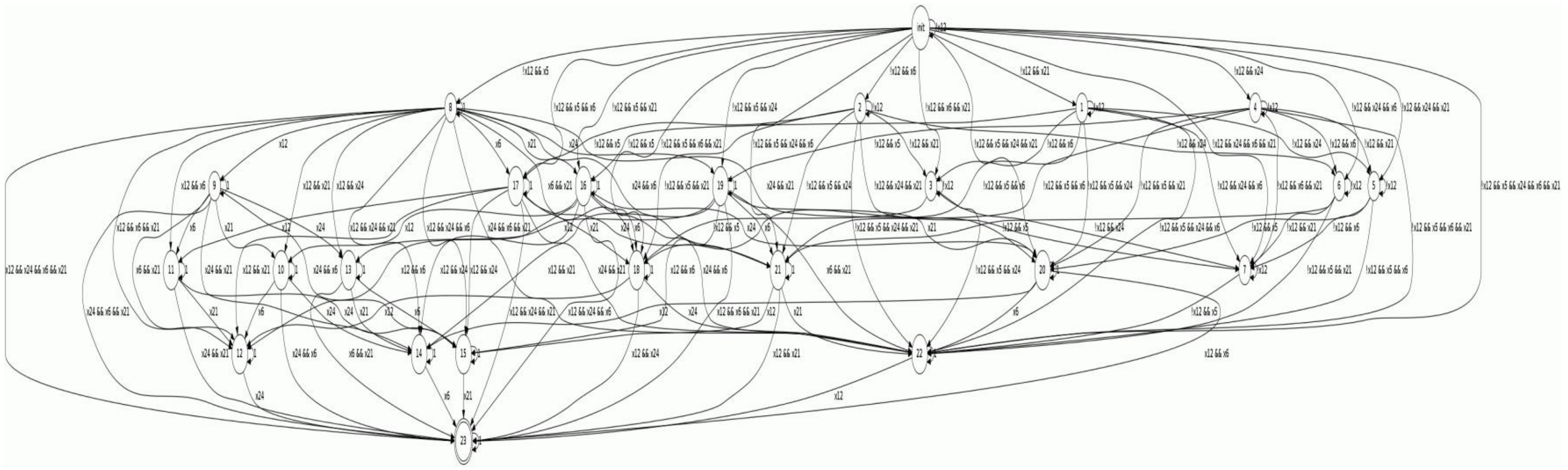


Figure 4-14: Example 2 (B) - Büchi Automaton. Refer to GIF file Figure 4-14 for a better view of this figure

Table 4-1 records all the results. The paths were shown using the workspace without coordinate system.

Table 4-2: Example 2 (B) Result sample

	Shortest Path	Path Length	Computational Effort
Multigraph Network Planning	X1, X6, X2, X3, X4, X5, X9, X13, X17, X21, X16, X12, X18, X24	14	Total execution time: 7,017 ms Used memory 45,266,224 bytes
Random Walk method	X1, X7, X6, X7, X13, X9, X5, X4, X3, X7, X12, X16, X12, X18, X24	[72, 70, 62, 59, 43, 34, 32, 31, 22, 21, 19, 17, 15]	Total execution time: 91,711 ms Used memory 120,292,640 bytes after 100,000 bytes random walk searches
Potential Field method	X1, X6, X11, X16, X21, X22, X23, X24, X18, X14, X10, X5, X9, X13, X12	[16, 15]	Total execution time: 44 ms Used 24,655,760 bytes memory for 10 random walk searches
Critical Path method	X1, X6, X7, X3, X4, X5, X9, X8, X12, X16, X21, X22, X23, X24	14	Total execution time: 99 ms Used 29,981,744 bytes memory

4.4 Example 3 (A)

Example 3 (A) had a workspace partitioned into 64 discrete cells (8 rows of 8 cells). In Example 3 (A), the mission requirements included:

- The UAV shall visit X14, X38, X51, and X62 at least once on its path.
- The UAV shall visit X51 before it is able to go to X32.

The environmental specification included:

- X4, X12, and X20 will have obstacles from time step 10 to 20.
- X34 will have obstacles from time step 6 to 12.
- X55 will have obstacles from time step 16 to 22.

Therefore, we could translate the mission requirements ϕ_3 and environmental specifications into extended LTL form as follows:

- F x14 && F x38 && F x51 && F x62
- x51 B x32
- $!x4_{[10,20]}$ && $!x12_{[10,20]}$ && $!x20_{[10,20]}$
- $!x34_{[6,12]}$
- $!x55_{[16,22]}$

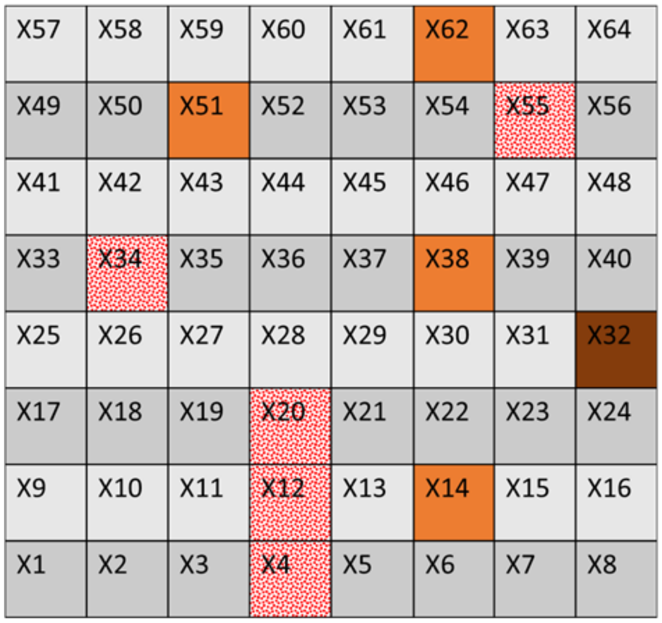


Figure 4-15: Example 3 (A) and (B) workspace representation

Using mission requirements, ϕ_3 , LTL2BA tool (http://www.lsv.fr/~gastin/ltl2ba/index.php) [19] generated a Büchi Automaton for the first four LTL rules in Figure 4-16. Then, we generated the state machine diagram accordingly, as shown in Figure 4-17 using [24]. Table 4-3 records all the results. The paths were shown using the workspace without coordinate system.

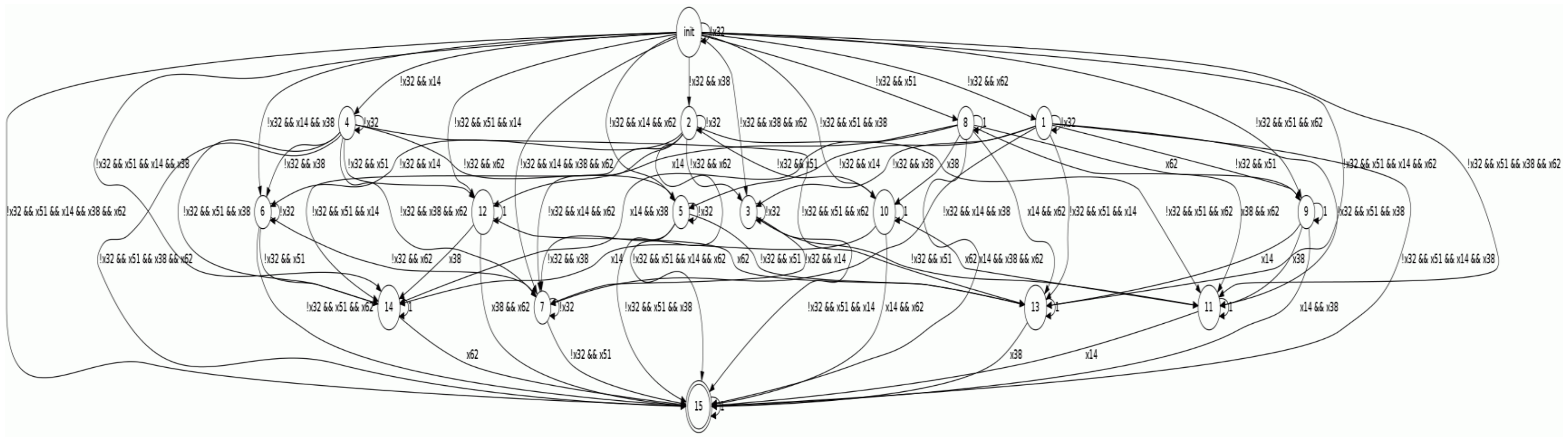


Figure 4-16: Example 3 (A) Büchi Automaton. Refer to GIF file Figure 4-16 for a better view of this figure

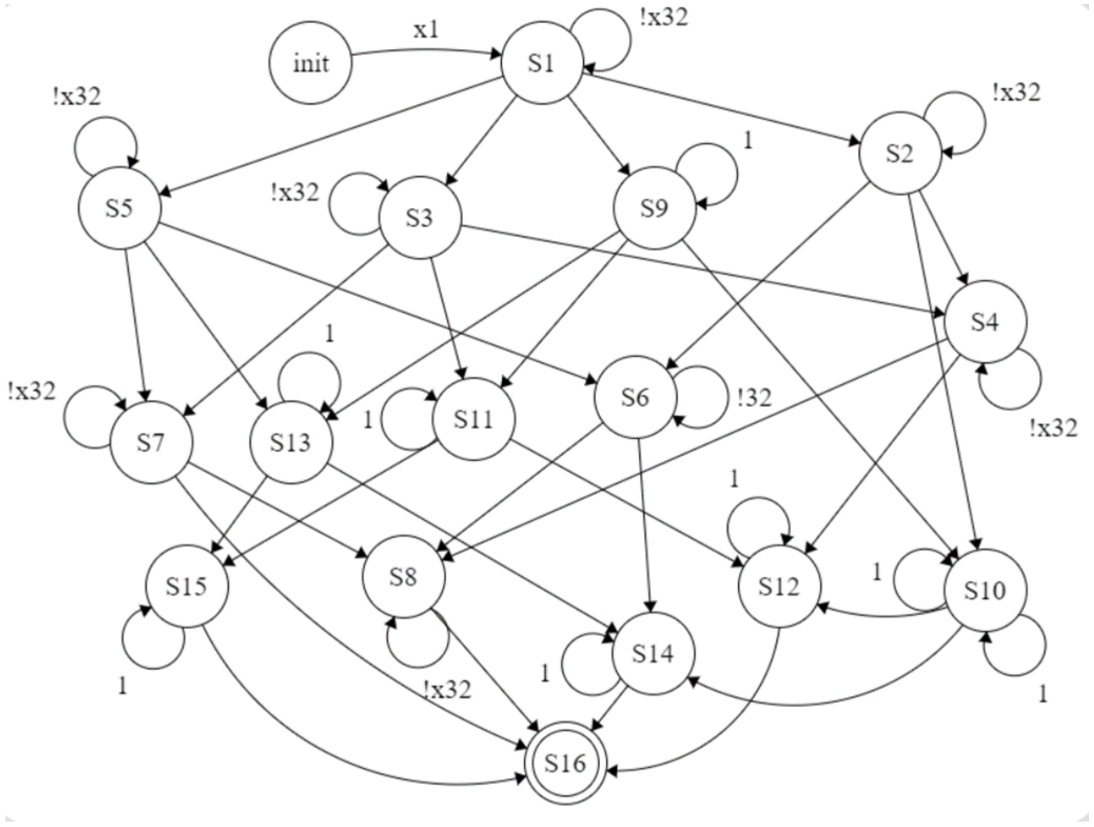


Figure 4-17: Example 3 (A) - State Machine Diagram. Labels of the edges are shown in Appendix B in text form.

Table 4-3: Example 3 (A) Result Sample

	Shortest Path	Path Length	Computational Effort
Multigraph Network Planning	X1, X2, X11, X4, X13, X14, X21, X29, X38, X45, X53, X62, X53, X44, X51	15	Total execution time: 106,586 ms Used memory 165,504,464 bytes

Random Walk method	X1, X2, X3, X12, X5, X14, X22, X29, X38, X46, X53, X62, X53, X60, X52, X51	[207, 157, 110, 104, 81, 64, 42, 36, 29, 28, 27, 16]	Total execution time: 376,385 ms Used memory 90,835,976 bytes after 100,000 iterations
Potential Field method	X1, X10, X11, X12, X13, X14, X22, X30, X38, X46, X54, X62, X53, X52, X51	[15]	Total execution time: 35 ms Used 23,323,800 bytes memory for 10 times of search
Critical Path method	X1, X2, X3, X12, X13, X14, X22, X30, X38, X46, X54, X62, X61, X52, X51	15	Total execution time:83 ms Used memory 25,986,432 bytes

4.5 Example 3 (B)

Example 3 (B) had a workspace partitioned into 64 discrete cells (8 rows of 8 cells) as Example 3 (A). In Example 3 (B), the mission requirements included:

- The UAV shall visit X14, X38, X51, and X62 at least once on its path.
- The UAV shall visit X32 before it is able to go to X51.

The environmental specifications are the same as Example 3 (A), which includes:

- X4, X12, and X20 will have obstacles from time step 10 to 20.
- X34 will have obstacles from time step 6 to 12.
- X55 will have obstacles from time step 16 to 22.

Therefore, we could translate the mission requirements ϕ_4 and environmental specifications into extended LTL form as following:

- F x14 && F x38 && F x51 && F x62
- X83 B X 51
- !x4$_{[10,20]}$ && !x12$_{[10,20]}$ && !x20$_{[10,20]}$
- !x34$_{[6,12]}$
- !x55$_{[16,22]}$

For evaluation purpose, Example 3 (B) had the same workspace as Example 3 (A). Also, the mission requirements of Example 3 (B) was the same as Example 2 (B) except with different names for the discrete cells and how they locate in the workspace. Therefore, we had the same the Büchi Automaton and the state machine diagram as Example 2 (B) except the name of the labels. Table 4-4 records all the results. The paths were shown using the workspace without coordinate system.

Table 4-4: Example 3 (B) Results

	Shortest Path	Path Length	Computational Effort
Multigraph Network Planning	X1, X2, X11, X4, X5, X14, X23, X32, X31, X38, X45, X53, X62, X53, X44, X51	16	Total execution time: 178,224 ms

			Used 255,219,664 bytes memory
Random Walk method	X1, X10, X2, X3, X12, X5, X13, X22, X31, X38, X31, X39, X47, X48, X40, X32, X31, X32, X39, X46, X53, X52, X61, X62	[245, 99, 80, 63, 49, 46, 41, 29, 26, 24]	Total execution time: 587,014 ms Used 73,334,496 bytes memory after 100,000 iterations
Potential Field method	X1, X10, X3, X4, X5, X14, X23, X32, X39, X38, X46, X54, X62, X61, X52, X51	[16]	Total execution time: 48 ms Used 24,654,768 bytes memory for 10 times of search
Critical Path method	X1, X2, X3, X12, X13, X14, X23, X32, X31, X38, X45, X44, X51, X52, X61, X62	16	Total execution time: 92 ms Used 33,984,648 bytes memory

Chapter 5 Evaluation

This chapter evaluates the four path planning methods based on their performance in Chapter 4. Multigraph Network Planning was used to find all the possible paths within the estimated number of steps. Therefore, if the Multigraph Network Planning method yielded a valid shortest path, we considered that path length to be the shortest path length we could find for that example. As mentioned in Chapter 4, although more random walk searches gave a more optimized path, we r ran random walk searches up to 100,000 times for any complex example in this thesis and used that result to compare with other path planning methods. Another reason for this is discussed in Section 5.2. For the Potential Field method, it was not very likely to happen that within the cells the vehicle could move from its current location because there were no more than two cells that had the same potential force. Hence, there were not as many different paths that could be generated by this method. Based on the experiment, we found that for all the examples except Example 2 (B), potential filed path searches that were run up to 10 times could find a path that had the same path length as the Multigraph Network Planning method's result.

5.1 Computational time, computer memory consumed, and the length of potential shortest path found by each approach

Based on the results in Section 4.1 – 4.4, we can analyze how the number of states in Büchi Automaton affects the computational time, computer memory consumed, and the length of potential shortest path found by each approach. The size of the workspace can increase the size of the multigraph and the number of moving direction choices in path planning. The number of states in the Büchi Automaton increases exponentially as the number of LTL requirements increases based on the mission requirements.

Table 5-1: Workspace and Büchi Automaton Comparison between Examples

	Number of discrete cells	Number of states in Büchi Automaton
Example 1	6	7
Example 2 (A)	25	13
Example 2 (B)	25	23
Example 3 (A)	64	17
Example 3 (B)	64	23

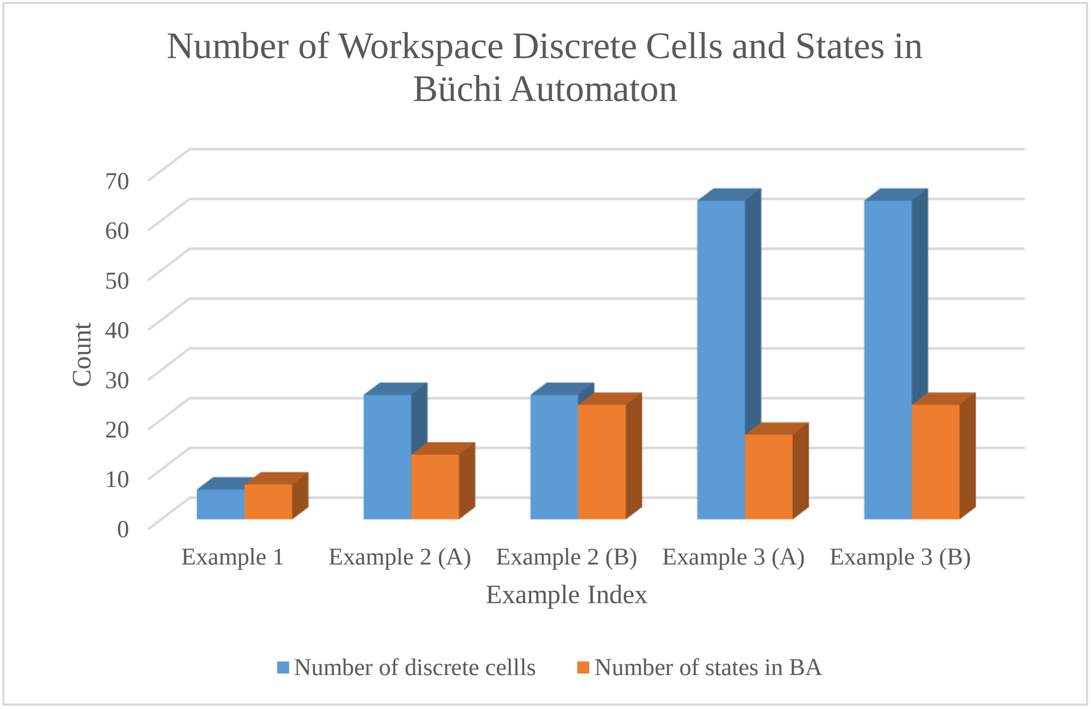


Figure 5-1: Number of Workspace Discrete Cells and States in Büchi Automaton

In Table 5-1 and Figure 5-1, Example 2 (A)/(B) and Example 3 (A)/(B) had the same number of discrete cells in their workspace – 25 and 64 respectively. Example 2 (B) has more states in the Büchi Automaton than Example 2 (A) has. Example 3 (B) has more states in the Büchi Automaton than Example 3 (A) has. Example 2 (B) and Example 3 (B) have the same number of states in the Büchi Automaton. Different mission requirements and obstacle areas in the workspace result in different shortest path planned for each example.

Table 5-2: Length of Potential Shortest Path Found by Each Approach (Steps)

	Multigraph Network Planning	Random Walk method (100,000 searches)	Potential Field method (10 searches)	Critical Path Method
Example 1	5	5	6	5
Example 2 (A)	11	13	11	11
Example 2 (B)	14	15	15	14
Example 3 (A)	15	16	15	15
Example 3 (B)	16	24	16	16

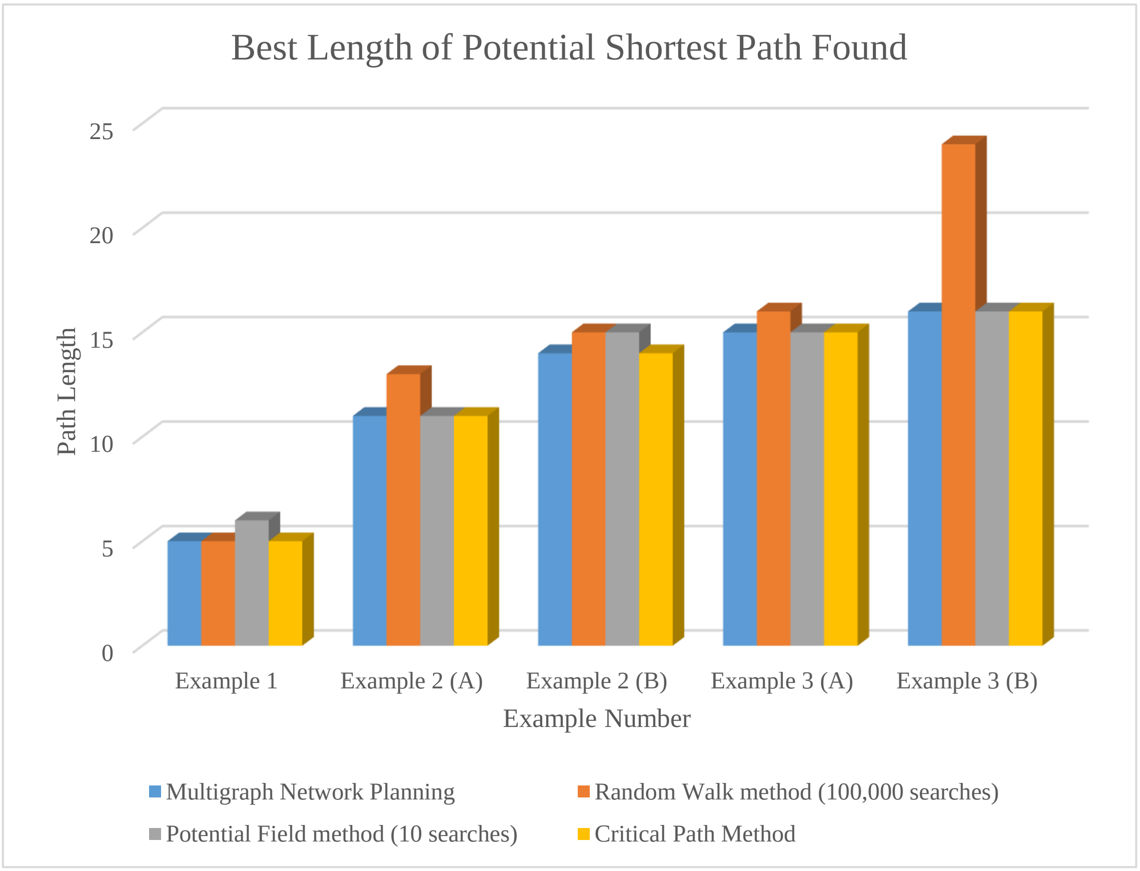


Figure 5-2: Length of Potential Shortest Path Found by Each Approach

Table 5-2: Length of Potential Shortest Path Found by Each Approach Table 5-2 and Figure 5-2 show that the Multigraph Network Planning method and the Critical Path method found the shortest path for all the examples. The Potential Field found the shortest path for all examples except Example 2 (B). This was caused by the physical location of each target. When a connecting state's target was very close to the vehicle's current location, that target location had very high attractive potential to get the vehicle move towards it first, even though moving toward that target might have cost more steps in the path due to dynamic obstacles. On the other hand, the Random Walk method's path length increased when the workspace size and number of states in the Büchi Automaton increased.

Table 5-3: Computational Time (ms)

	Multigraph Network Planning	Random Walk method (100,000 searches)	Potential Field method (10 searches)	Critical Path Method
Example 1	17	34	44	29
Example 2 (A)	979	47,556	34	59
Example 2 (B)	7,017	91,711	50	99
Example 3 (A)	106,586	376,385	35	83
Example 3 (B)	174,472	587,014	48	92

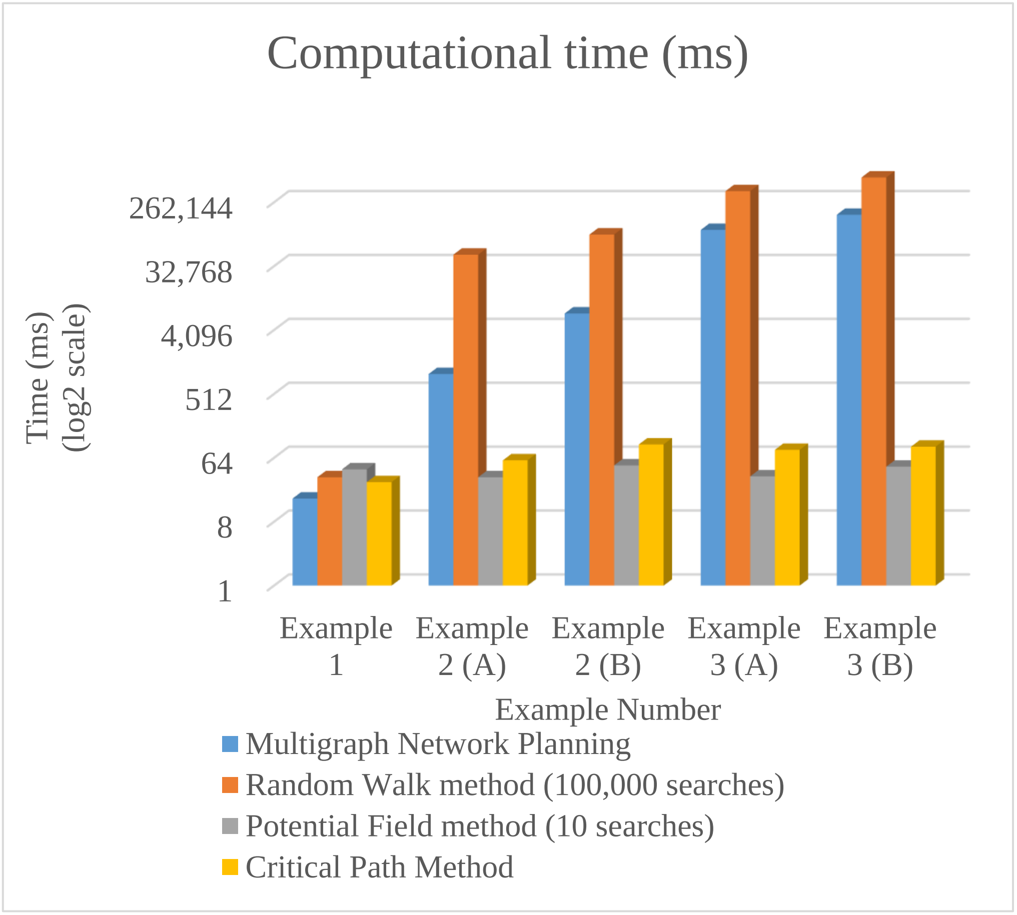


Figure 5-3:Computational Time (ms) of 5 Examples

As shown in Table 5-3 and Figure 5-3, when the workspace size was small and there were not many mission requirements, as in Example 1, all four methods had similar short computational time to find the shortest path(below 60 ms). The computational time of the Multigraph Network Planning method and the Random Walk method both increased much faster than the Potential Field method did. As the workspace size and number of states in the Büchi Automaton increased the Multigraph Network Planning method required less time than the Random Walk method did in all the examples. The Potential Field method had the overall best performance in computational time for all of the examples. There was a very small change in computational time when the mission requirements became more complex (i.e. from Example 1 to Example 3 (A)). The longest time finding the shortest path took by the Potential Field method was 50 ms for Example 3 (B), which was much shorter than the Multigraph Network Planning method or the Random Walk method did. The Critical Path method used the second shortest computational time to find shortest paths for all the examples. The computational time it took was steady and very close to the time used by the Potential Field method. The longest time it used was 99 ms for Example 2 (B). For the more complex missions among the five examples, Example 2 (B) and Example 3 (B) required the longer time than Example 2 (A) and Example 3 (A).

The workspace size varies in different examples. When the size of workspace increased from 25 to 64 discrete cells (from Example 2 (B) to Example 3 (B)) and the number of states in the Büchi Automaton remained at 23, the Multigraph Network Planning method's computational time increased 24.8 times, the Random Walk (100,000 searches) method's computational time increased 6.4 times, and the Potential Field (10 searches) method and the Critical method almost used the same amount of time. When the number of states in the Büchi Automaton increased from 13 to 23 (from Example 2 (A) to Example 2 (B)) and the workspace size remained at 25 discrete cells, the Multigraph Network Planning method's computational time increased 7.2 times, the Random Walk (100,000 searches) method's computational time increased 1.9 times, the Potential Field (10 searches) method's computational time increased 1.5 times, and the Critical Path method's computational time increased by 1.7 times.

The number of states in the Büchi Automaton also varies in different examples. When the number of states in the Büchi Automaton increased from 17 to 23 (from Example 3 (A) to Example 3 (B)) and the workspace size remained at 64 discrete cells, the Multigraph Network Planning method's computational time increased 1.6 times, the Random Walk (100,000 searches) method's computational time increased 1.6 times, the Potential Field (10 searches) method's computational time increased 1.4 times, and the Critical Path method's computational time increased by 1.1 times. The workspace size had a bigger influence on the computational time than the number of mission requirements for the Multigraph Network Planning method.

Table 5-4: Computer Memory Consumed (bytes)

	Multigraph Network Planning	Random Walk method (100,000 searches)	Potential Field method (10 searches)	Critical Path Method
Example 1	3,995,104	3,995,104	20,661,408	20,667,808
Example 2 (A)	35,450,240	35,450,240	51,541,792	20,659,592
Example 2 (B)	29,565,104	45,266,224	120,292,640	24,655,760
Example 3 (A)	165,504,464	165,504,464	90,835,976	23,323,800
Example 3 (B)	255,219,664	255,219,664	73,334,496	24,654,768

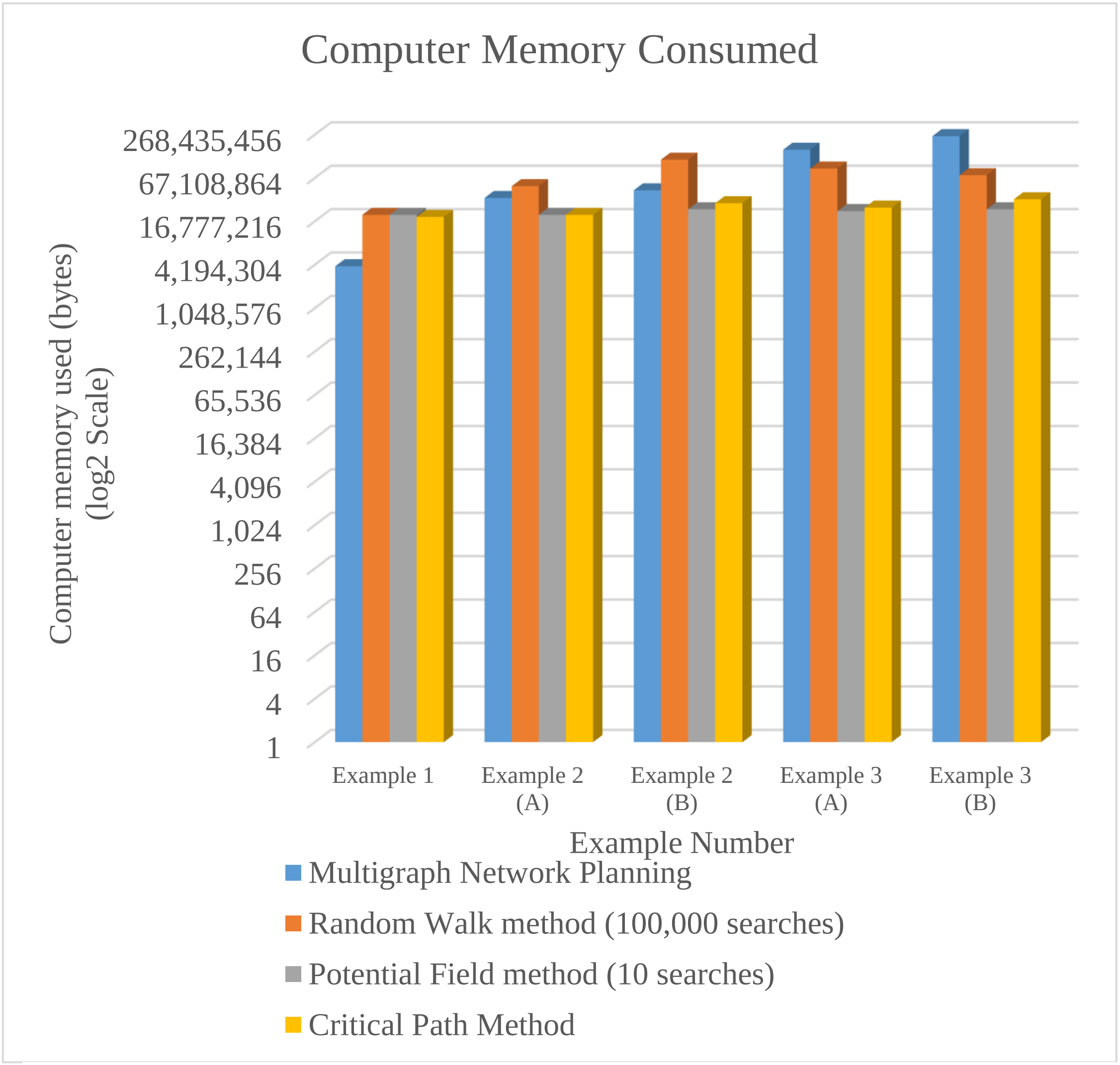


Figure 5-4: Computer Memory Consumed by 5 Examples

The computer memory consume by each method was calculated when running the java code on Eclipse Java Oxygen, using Java Runtime class's methods: Runtime.getRuntime().totalMemory()-Runtime.getRuntime().freeMemory(). Table 5-4 and Figure 5-4 show that when the workspace was small and there are not many mission requirements, as in Example 1, all four methods consumed computer memory, and the Multigraph Network Planning method performed much better than the two other methods. Except the Random Walk method, the computer memory consumed by the three other methods all increased as the workspace size and the number of states in the Büchi Automaton increased. The Multigraph Network Planning method required more computer memory than the Potential Field method and the Critical method did. For Although the Random Walk method's computer memory consumption was less than the Multigraph method, and the path it found was longer than the other methods. In addition, because the paths were randomly chosen, each time we ran the program could result in very different computer memory consumption depending on how fast we were able to find a shorter path. Example 2 (B) required more memory than any other examples using the Random Walk method. This may be caused by extreme long paths found by the program during the path planning. Regarding computer memory, the Potential Field method was the best method to find one valid path with the shortest path length. It required the lease computer memory. The Critical Path method requires a little more memory than the Potential Field method but was much better than the Multigraph Network Planning method and the Random Walk method.

Overall, we can see a general trend that as the workspace size and the number of states in the Büchi Automaton increases, the computational time, the computer memory consumed, and the length of the potential shortest path found by each approach all tend to increase. Although the Critical Path method performed not as well as the Potential Field method, it consistently performed in all aspects and was able to find all the shortest paths for all the example with the best quality of path length. Potential Field method also had steady performance. However, due to the algorithm it used to find paths, the physical location of the targets in the mission may influence the quality of the path length this method could planned. The Multigraph Network Planning method performed well for the small workspace and could find all the possible shortest paths. The Random Walk method required less programming effort, but it performed poorly compared with the other methods.

5.2 Random Walk Method Path Length Evaluation

The Random Walk method used extensive computational time and memory for all the examples, yet the paths generated were not optimized for most examples. Therefore, we investigated how the potential shortest path length changed as the computer conducted more random walk searches.

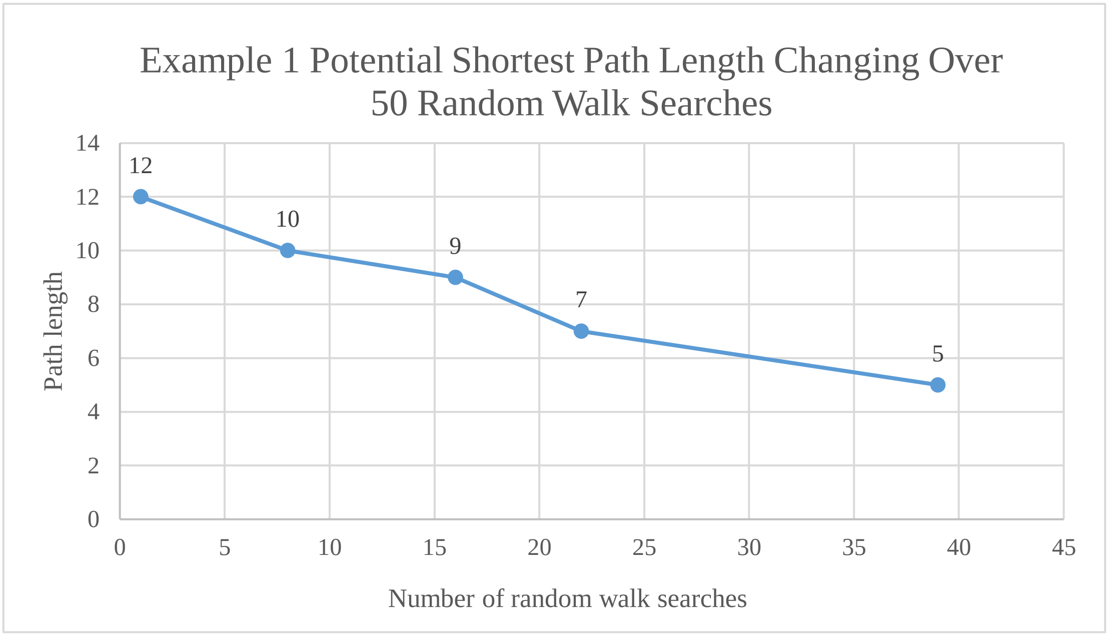


Figure 5-5: Example 1 potential shortest path length changing over 50 random walk searches

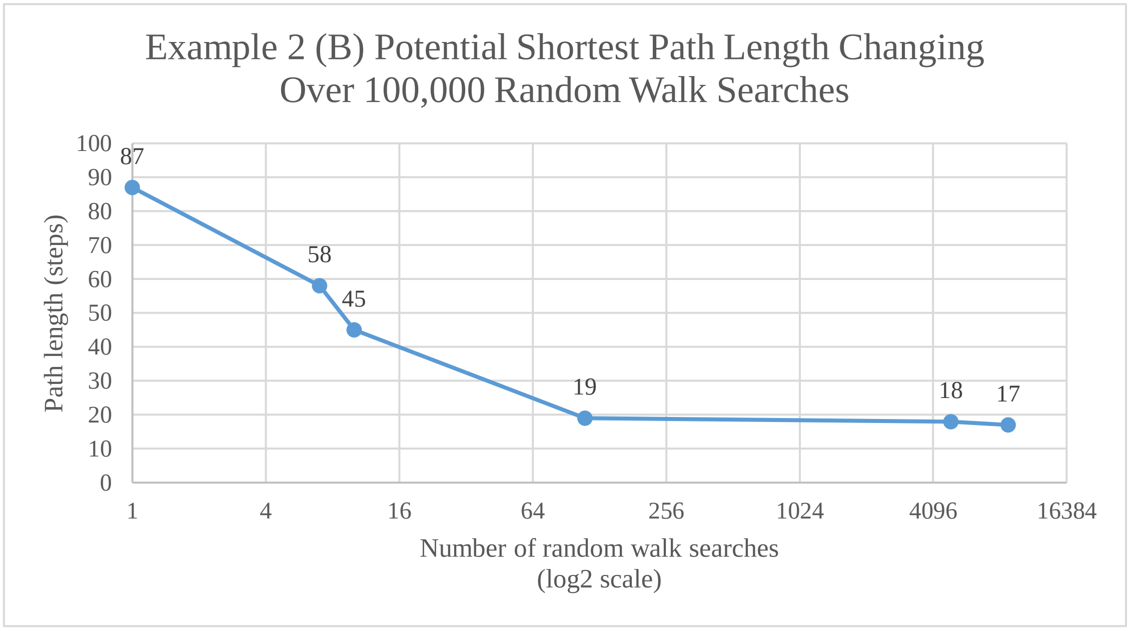


Figure 5-6: Example 2 (B) potential shortest path length changing over 100,000 random walk searches

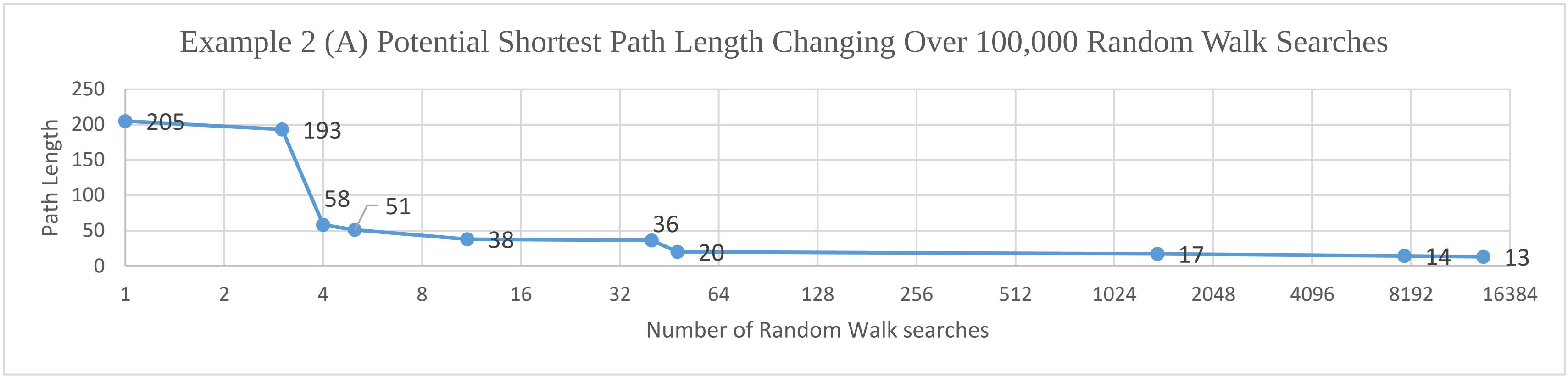


Figure 5-7: Example 2 (A) Potential Shortest Path Length Changing Over 100,000 Random Walk Searches

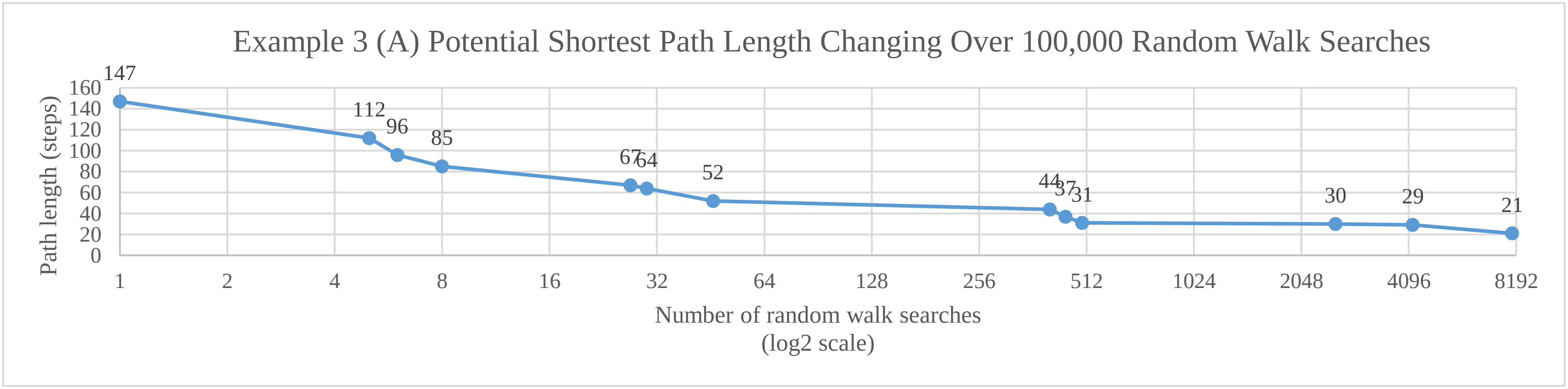


Figure 5-8: Example 3 (A) potential shortest path length changing over 100,000 random walk searches

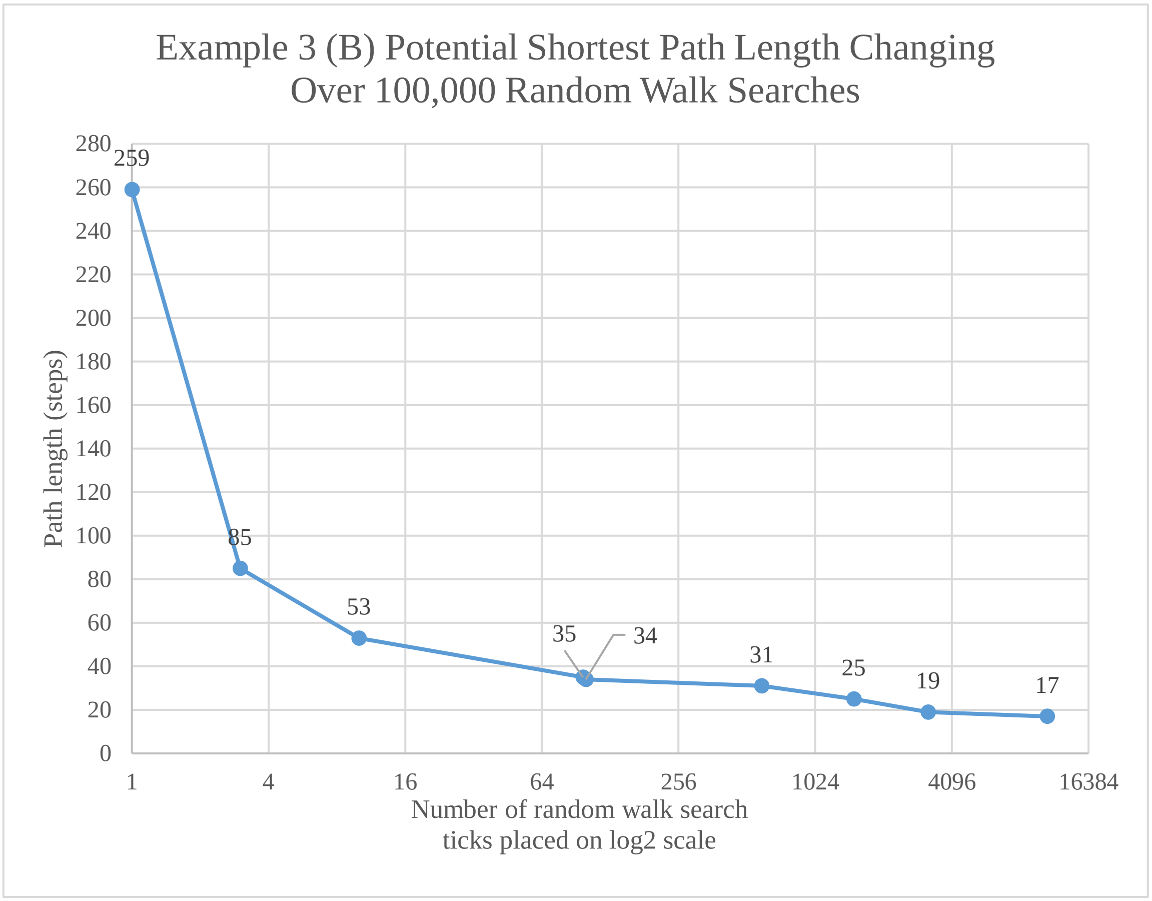


Figure 5-9: Example 3 (B) potential shortest path length changing over 100,000 random walk searches

Figure 5-5 shows that the path length yield by the Random Walk method decreased quickly during the first 50 random walk searches and then reached the optimized shortest path for a simple mission like Example 1. On the other hand, Figure 5-6 to Figure 5-9 all demonstrated that after the first 64 random walk searches, the path length decreased very slowly. For future research, the researcher can step up criteria on how to balance between the path length and the computational effort.

5.3 Improve Multigraph Network Planning method

One advantage of the Multigraph Network Planning method is that this method provides all possible paths that fulfill the mission requirements under the given environmental specifications. For future research, we may want multiple vehicles to operate on the same mission to increase the possibility of success. Therefore, it would be helpful to know all the possible paths with the shortest path length.

According to the results in Section 5.1, the total estimated steps for the Multigraph Network Planning method was critical for the method's performance. The reason for this is that the workspace size had the most influence on performance, and workspace size directly affected the estimated total steps we used in the method. To find at least one valid path, our estimated total steps were much bigger than the actual shortest path length. Therefore, we need to find a way to better estimate the total steps. The Potential Field method can be a good solution. This method aims to find one shortest path and had very good performance in terms of computational time. The Potential Field method could provide a good estimation of the total steps for creating the multigraph, and then using the Multigraph Network Planning method could help find all the valid shortest paths. Combining two methods, we have a two-step path planning method that worked more efficient for complex missions. Table 5-5 to Table 5-7 show that the performance of the Multigraph Network Planning method is better using the path length generated by the Potential Field method for complex examples like Example 2 (B), Example 3 (A), and 3 (B).

Table 5-5: Example 1 Multigraph Network Planning Method Performance with different estimated total steps

Method	With an estimate from workspace size	With an estimate from potential field planning
Estimated total steps	7	5
Computational Time (ms)	17	10
Computer Memory Consumed (bytes)	3,995,104	2,663,416

Table 5-6: Example 2 (A) Multigraph Network Planning Method Performance with different estimated total steps

Method	With an estimate from workspace size	With an estimate from potential field planning
Estimated total steps	26	11
Computational Time (ms)	1,073	366
Computer Memory Consumed (bytes)	35,450,240	14,584,984

Table 5-7: Example 2 (B) Multigraph Network Planning Method Performance with different estimated total steps

Method	With an estimate from workspace size	With an estimate from potential field planning
Estimated total steps	26	15
Computational Time (ms)	7,017	1,763
Computer Memory Consumed (bytes)	45,266,224	5,848,184

Table 5-8: Example 3 (A) Multigraph Network Planning Method Performance with different estimated total steps

Method	With an estimate from workspace size	With an estimate from potential field planning
Estimated total steps	65	15
Computational Time (ms)	106,586	5,499
Computer Memory Consumed (bytes)	165,504,464	30,105,440

Table 5-9: Example 3 (B) Multigraph Network Planning Method Performance with different estimated total steps

Method	With an estimate from workspace size	With an estimate from potential field planning
Estimated total steps	65	16
Computational Time (ms)	174,472	8,752
Computer Memory Consumed (bytes)	255,219,664	49,057,144

The results show that for all the five examples using the Potential Field method to estimate the total number of steps reduced the computational effort and memory requirements for complex missions. As shown in Figure 5-10, when the percentage of decreased estimation of total steps increased, the percentage of decreased computational effort increased. In addition, this result was also influenced by the missions' workspace size on complexity of the mission requirements. Example 2 (B) did not save much total estimated steps, but due to its complex mission, the computational time and memory were both reduced.

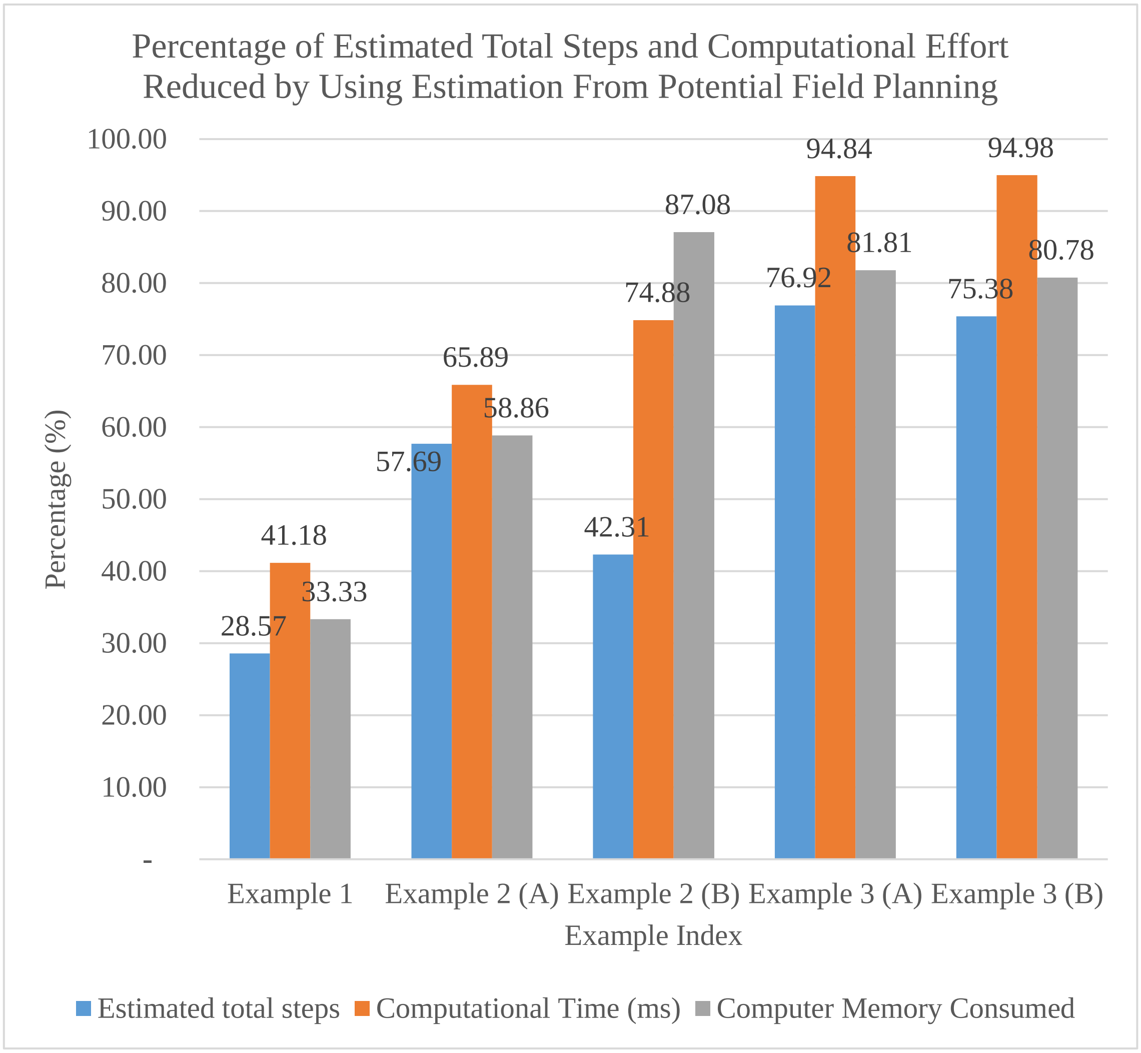


Figure 5-10: Percentage of Estimated Total Steps and Computational Effort Decreased by using Estimation from Potential Field Planning

For instance, in Example 2 (B), estimating with the Potential Field method saved 74.9% of the time used by the workspace size estimating method. When the number of states in the Büchi Automaton was 17 and the workspace size was at 64 discrete cells (Example 3 (A)), estimating with the Potential Field method saved 94.8% of the time used by the workspace size estimating method. When the workspace size increased from 25 to 64 discrete cells and the number of states in the Büchi Automaton remained the same (from Example 2 (B) to Example 3 (B)), estimating with the Potential Field method saved 95.0% of the time used by the workspace size estimating method.

Computed memory was save using the estimate from Potential Field method. The computer memory consumed by the Multigraph Network Planning method was reduced by 87.1% (Example 2 (B)), 81.8% (Example 3 (A)), and 80.8% (Example 3 (B)) using the estimation from the potential field planning method, comparing with the computer memory consumed by using the estimation from workspace size method.

For more complex problems, the computer memory size used by the Potential Field method had less influence on the overall computer memory consumed for path planning. The total computational time and memory used to run the Potential Field method and the Multigraph Network Planning method was evaluated. As shown in Figure 5-11, this two-step path planning saved much computational time and memory for the large workspace and complex mission examples. Other than Example 1, computational time was reduced for all the examples. Time was reduced up to 95% for Example 3 (A) and 3 (B). Computer memory consumed by Example 1 and 2 (B) was increased. When the size of workspace increased to 64 discrete cells, the two-step path planning method saved more than 50% of the computer memory. This demonstrates that the two-step planning method suits the missions involve large workspace.

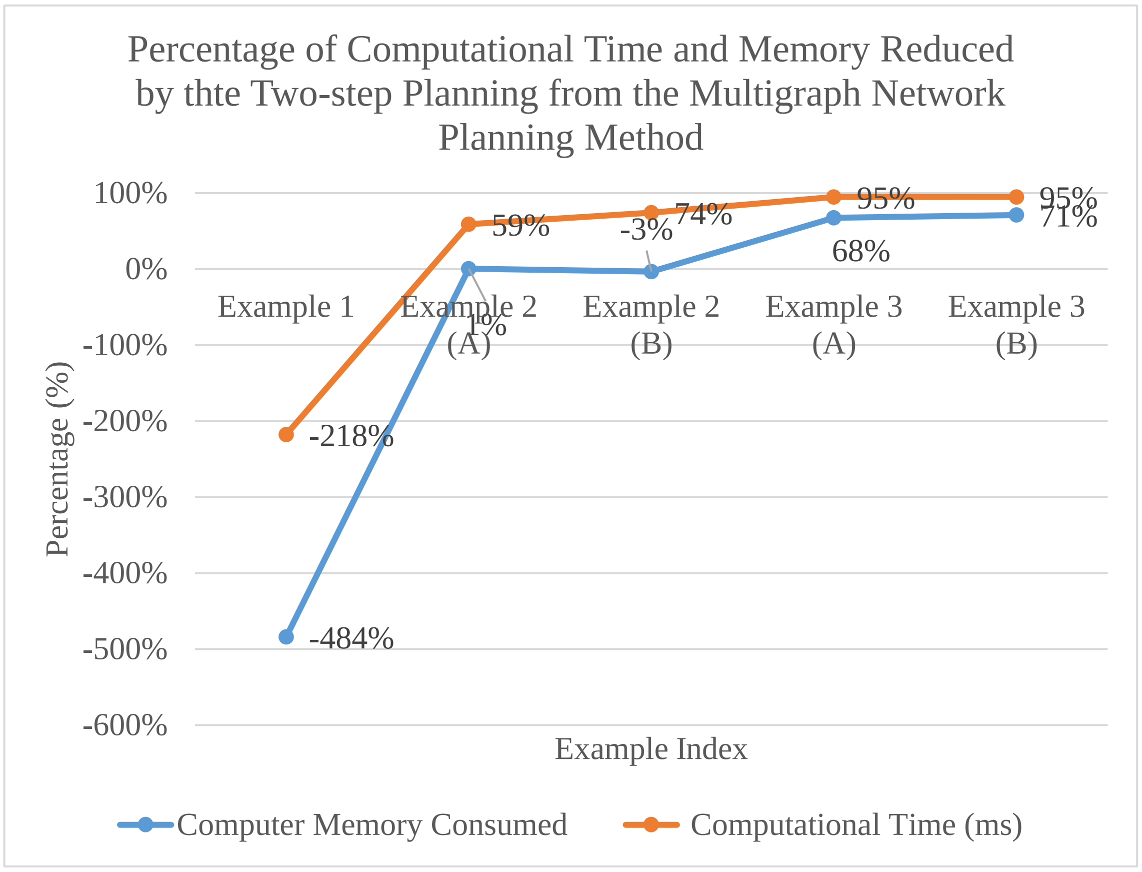


Figure 5-11: Percentage of Computational Time and Memory Reduced by thte Two-step Planning from the Multigraph Network Planning Method

According to Jun and Andrea [9], finding different paths for multiple UAV mission can decrease the overall risk. Based on the results, we can conclude that using the Potential Field method for estimating total steps in the Multigraph Network Planning method is a good way to improve the method.

Chapter 6 Conclusion and Future Work

This thesis compared four different path planning methods that use LTL for missions with a list of mission requirements and known environmental specifications. By using LTL, we were able to translate the high-level specifications and build a Büchi Automaton. Based on the Büchi Automaton we can build a state machine diagram for a single vehicle and then use it in the four path planning methods. All four methods were capable of finding paths that fulfill the mission requirements under the given environmental specifications.

The Potential Field method was the best method to find one of the shortest paths for the mission with minimum computational effort and memory requirement. Computational time required was very short and increased very slowly even with more complex missions. Computer memory it consumed increased the least among the four methods. It successfully found one shortest path for all the examples, except Example 2 (B) where the path found was one step longer than the shortest path's length. Another disadvantage of the method was that it was only able to find one shortest path.

The Critical Path method had the ability to find all shortest path and the computational effort and memory requirement was low and steady. One disadvantage of this method is that when we create the state machine diagram, we not only had to spend time to modify the Büchi Automaton's edges, but also need to make sure that the nodes in the state machine created in such an order that a node shall not be created until all the other nodes with outgoing edge connecting to this node have been created. The same problem applies to the Multigraph Network Planning method.

The Multigraph Network Planning method was also able to find all the possible paths with the shortest path length. The required computational time and memory grew as the workspace size and mission requirements' complexity increased. When a valid path for the mission existed, this method could always find it. The disadvantage of this method was that the required computational time and memory grew very fast especially when the workspace size increased. This is because before we planned the path for the first time, we did not know the least number of steps we would need. To find a valid path, we may have had to use a rather large estimated total steps for this method. This caused the quick growing in computational time and memory consumed. The Potential Field method was a very good solution to this problem. For simple missions, running the Potential Field method could add unnecessary computational time and memory. However, this extra step of total step estimation saved much computational time and memory for the complex missions like Example 3 (B). The next step for these the Multigraph Network Planning method will be finding a way to estimate the total steps needed for a mission so that we can plan paths for multiple vehicles.

The Random Walk method required the least programming effort. Nevertheless, this method did not suit big workspace or complex mission requirements because of the extensive computational time and memory required for this method to find the shortest path. The next step for this method is to study how to balance between the path length and the computational time and memory we are willing to spend for the number of the random walk researches we use to find a valid path.

In conclusion, the Potential Field method and the Critical Path method are good for quickly finding one shortest path for the mission. A trade-off analysis for these two methods needs to be conducted in terms of path length requirements and computational effort and memory requirement, and the number of paths required for the mission. If only a very short path is required for the mission and computational effort and memory is considered more important than the optimal path length, Potential Field method is better. If it’s more important to find the optimal shortest path and multiple paths for a mission, then the Critical Path method is better. The Multigraph Network Planning method is a good method to look for all possible paths with predetermined path length but requires high computational time and memory. The Random Walk method is not efficient in solving this shortest path planning with dynamic obstacle problem. All the methods need to be improved for mission requirements involving time constraints for the vehicles.

Other possible extensions to the work detailed in this thesis exist. An algorithm that can translate the Büchi Automaton directly to state machine diagram will help increase the path planning efficiency for all the path planning methods. It is also important to improve the path planning methods so that they can conduct multiple vehicle path planning. Continuous time path planning will provide more accurate path length and path plan for fulfilling the missions successfully.

Appendix A

Example 2 (B) state machine diagram in text form

```
never { /* !((!x50)Ux22) && Fx22 && Fx44 && Fx11 && Fx14 */
T0_S1 :    /* S1 */
           if
           :: (!x12) -> goto T0_S1
           :: (!x12 && x21) -> goto T0_S2
           :: (!x12 && x6) -> goto T0_S3
           :: (!x12 && x24) -> goto T0_S5
           :: (!x12 && x5) -> goto T0_S9
           fi;
T0_S2 :    /* 1 */
           if
           :: (!x12) -> goto T0_S2
           :: (!x12 && x6) -> goto T0_S4
           :: (!x12 && x24) -> goto T0_S6
           :: (!x12 && x5) -> goto T0_S10
           fi;
T0_S3 :    /* 2 */
           if
           :: (!x12) -> goto T0_S3
           :: (!x12 && x21) -> goto T0_S4
           :: (!x12 && x24) -> goto T0_S7
           :: (!x12 && x5) -> goto T0_S11
           fi;
T0_S4 :    /* 3 */
           if
           :: (!x12) -> goto T0_S4
           :: (!x12 && x24) -> goto T0_S8
           :: (!x12 && x5) -> goto T0_S12
           fi;
T0_S5 :    /* 4 */
           if
           :: (!x12) -> goto T0_S5
           :: (!x12 && x21) -> goto T0_S6
           :: (!x12 && x6) -> goto T0_S7
           :: (!x12 && x5) -> goto T0_S13
           fi;
T0_S6 :    /* 5 */
           if
           :: (!x12) -> goto T0_S6
           :: (!x12 && x6) -> goto T0_S8
```

```
          :: (!x12 && x5) -> goto T0_S14
          fi;
T0_S7 :   /* 6 */
          if
          :: (!x12) -> goto T0_S7
          :: (!x12 && x21) -> goto T0_S8
          :: (!x12 && x5) -> goto T0_S15
          fi;
T0_S8 :   /* 7 */
          if
          :: (!x12) -> goto T0_S8
          :: (!x12 && x5) -> goto T0_S16
          fi;
T0_S9 :   /* 8 */
          if
          :: (1) -> goto T0_S9
          :: (x21) -> goto T0_S10
          :: (x6) -> goto T0_S11
          :: (x24) -> goto T0_S13
          :: (x12) -> goto T1_S23
          fi;
T1_S23 :   /* 9 */
          if
          :: (1) -> goto T1_S23
          :: (x21) -> goto T1_S22
          :: (x6) -> goto T1_S21
          :: (x24) -> goto T2_S19
          fi;
T1_S22 :   /* 10 */
          if
          :: (1) -> goto T1_S22
          :: (x6) -> goto T1_S20
          :: (x24) -> goto T2_S18
          fi;
T1_S21 :   /* 11 */
          if
          :: (1) -> goto T1_S21
          :: (x21) -> goto T1_S20
          :: (x24) -> goto T3_S17
          fi;
T1_S20 :   /* 12 */
          if
          :: (1) -> goto T1_S20
          :: (x24) -> goto accept_all
          fi;
```

```
T2_S19 :    /* 13 */
        if
        :: (1) -> goto T2_S19
        :: (x21) -> goto T2_S18
        :: (x6) -> goto T3_S17
        fi;
T2_S18 :    /* 14 */
        if
        :: (1) -> goto T2_S18
        :: (x6) -> goto accept_all
        fi;
T3_S17 :    /* 15 */
        if
        :: (1) -> goto T3_S17
        :: (x21) -> goto accept_all
        fi;
T0_S10 :    /* 16 */
        if
        :: (1) -> goto T0_S10
        :: (x6) -> goto T0_S12
        :: (x24) -> goto T0_S14
        :: (x12) -> goto T1_S22
        fi;
T0_S11 :    /* 17 */
        if
        :: (1) -> goto T0_S11
        :: (x21) -> goto T0_S12
        :: (x24) -> goto T0_S15
        :: (x12) -> goto T1_S21
        fi;
T0_S12 :    /* 18 */
        if
        :: (1) -> goto T0_S12
        :: (x24) -> goto T0_S16
        :: (x12) -> goto T1_S20
        fi;
T0_S13 :    /* 19 */
        if
        :: (1) -> goto T0_S13
        :: (x21) -> goto T0_S14
        :: (x6) -> goto T0_S15
        :: (x12) -> goto T2_S19
        fi;
T0_S14 :    /* 20 */
        if
        :: (1) -> goto T0_S14
        :: (x6) -> goto T0_S16
        :: (x12) -> goto T2_S18
        fi;
T0_S15 :    /* 21 */
        if
        :: (1) -> goto T0_S15
        :: (x21) -> goto T0_S16
        :: (x12) -> goto T3_S17
        fi;
T0_S16 :    /* 22 */
        if
        :: (1) -> goto T0_S16
        :: (x12) -> goto accept_all
        fi;
accept_all :    /* 23 */
        skip
}
```

Appendix B

Example 3 (A) state machine diagram in text form

```
never { /* !((!x51)Ux32) && Fx14 && Fx38 && Fx51 && Fx62 */
T0_S1 :    /* S1 */
        if
        :: (!x32) -> goto T0_S1
        :: (!x32 && x62) -> goto T0_S2
        :: (!x32 && x38) -> goto T0_S3
        :: (!x32 && x14) -> goto T1_S5
        :: (!x32 && x51) -> goto T0_S9
        fi;
T0_S2 :    /* S2 */
        if
        :: (!x32) -> goto T0_S2
        :: (!x32 && x38) -> goto T0_S4
        :: (!x32 && x14) -> goto T1_S6
        :: (!x32 && x51) -> goto T0_S10
        fi;
T0_S3:    /* S3 */
        if
        :: (!x32) -> goto T0_S3
        :: (!x32 && x62) -> goto T0_S4
        :: (!x32 && x14) -> goto T2_S7
        :: (!x32 && x51) -> goto T0_S11
        fi;
T0_S4 :    /* S4 */
        if
        :: (!x32) -> goto T0_S4
        :: (!x32 && x14) -> goto T2_S8
        :: (!x32 && x51) -> goto T0_S12
        fi;
T1_S5 :    /* S5 */
        if
        :: (!x32) -> goto T1_S5
        :: (!x32 && x62) -> goto T1_S6
        :: (!x32 && x38) -> goto T2_S7
        :: (!x32 && x51) -> goto T1_S13
        fi;
T1_S6 :    /* S6 */
        if
        :: (!x32) -> goto T1_S6
        :: (!x32 && x38) -> goto T2_S8
        :: (!x32 && x51) -> goto T1_S14
```

```
            fi;
T2_S7 :    /* S7 */
            if
            :: (!x32) -> goto T2_S7
            :: (!x32 && x62) -> goto T2_S8
            :: (!x32 && x51) -> goto T3_S15
            fi;
T2_S8 :    /* S8 */
            if
            :: (!x32) -> goto T2_S14
            :: (!x32 && x51) -> goto S16
            fi;
T0_S9 :    /* S9 */
            if
            :: (1) -> goto T0_S9
            :: (x62) -> goto T0_S10
            :: (x38) -> goto T0_S11
            :: (x14) -> goto T1_S13
            fi;
T0_S10 :    /* S10 */
            if
            :: (1) -> goto T0_S10
            :: (x38) -> goto T0_S12
            :: (x14) -> goto T1_S14
            fi;
T0_S11 :    /* S11 */
            if
            :: (1) -> goto T0_S11
            :: (x62) -> goto T0_S12
            :: (x14) -> goto T3_S15
            fi;
T0_S12 :    /* S12 */
            if
            :: (1) -> goto T0_S12
            :: (x14) -> goto S16
            fi;
T1_S13 :    /* S13 */
            if
            :: (1) -> goto T1_S13
            :: (x62) -> goto T1_S14
            :: (x38) -> goto T3_S15
            fi;
T1_S14 :    /* S14 */
            if
            :: (1) -> goto T1_S14
```

```
          :: (x38) -> goto S16
          fi;
T3_S15 :    /* S15 */
          if
          :: (1) -> goto T3_S15
          :: (x62) -> goto S16
          fi;
S16 :     /* S16 */ (accept all)
          skip
}
```

Bibliography

[1] K. P. Valavanis and G. J. Vachtsevanos, “Handbook of unmanned aerial vehicles,” *Handb. Unmanned Aer. Veh.*, pp. 1–3022, 2015.

[2] T. Amukele, P. M. Ness, A. A. R. Tobian, J. Boyd, and J. Street, “Drone transportation of blood products,” *Transfusion*, vol. 57, no. 3, pp. 582–588, 2017.

[3] J. Reagan, “German Company Demonstrates _Lifeguard_ Drone - DRONELIFE.” DRONELIFE, 2018.

[4] Kelsey D. Atherton, “NASA Sent A Drone Out To Track Hurricane Matthew.” POPULAR SCIENCE, 2016.

[5] S. Mohammad Khansari-Zadeh and A. Billard, “A dynamical system approach to realtime obstacle avoidance,” *Auton. Robots*, vol. 32, no. 4, pp. 433–454, 2012.

[6] G. C. S. Cruz and P. M. M. Encarnação, “Obstacle Avoidance for Unmanned Aerial Vehicles,” *J. Intell. Robot. Syst.*, vol. 65, no. 1–4, pp. 203–217, 2012.

[7] A. Ulusoy and C. Belta, “Receding horizon temporal logic control in dynamic environments,” *Int. J. Rob. Res.*, vol. 33, no. 12, pp. 1593–1607, 2014.

[8] A. Richards and J. P. How, “Aircraft trajectory planning with collision avoidance using mixed integer linear programming,” *Am. Control Conf.*, vol. 3, no. 2, pp. 1936–1941 vol.3, 2002.

[9] M. Jun and R. D. Andrea, “Path Planning for Unmanned Aerial Vehicles in Uncertain and Adversarial Environment,” *Coop. Control Model. Appl. Algorithms*, pp. 95–111, 2003.

[10] S. Koenig and M. Likhachev, “D* Lite,” *Proc. Eighteenth Natl. Conf. Artif.*

Intell., pp. 476–483, 2002.

[11] R. Jin, “Distance-Constraint Reachability Computation in Uncertain Graphs,” *VLDB - 37th Int. Conf. Very Large Data Bases*, vol. 4, no. 9, pp. 551–562, 2011.

[12] E. M. Wolff, U. Topcu, and R. M. Murray, “Automaton-guided controller synthesis for nonlinear systems with temporal logic,” *Intell. Robot. Syst. (IROS), 2013 IEEE/RSJ Int. Conf.*, pp. 4332–4339, 2013.

[13] J. Shaffer, E. Carrillo, and H. Xu, “Receding Horizon Synthesis and Dynamic Allocation of UAVs to Fight Fires,” pp. 1–8.

[14] T. Wongpiromsarn, U. Topcu, N. Ozay, H. Xu, and R. M. Murray, “TuLiP: A Software Toolbox for Receding Horizon Temporal Logic Planning,” *Proc. 14th Int. Conf. Hybrid Syst. Comput. Control*, pp. 313–314, 2011.

[15] Y. Zhou, D. Maity, and J. S. Baras, “Optimal mission planner with timed temporal logic constraints,” *2015 Eur. Control Conf. ECC 2015*, pp. 759–764, 2015.

[16] Y. Zhou, D. Maity, and J. S. Baras, “Timed automata approach for motion planning using metric interval temporal logic,” *2016 Eur. Control Conf.*, pp. 690–695, 2016.

[17] D. Maity and J. S. Baras, “Motion planning in dynamic environments with bounded time temporal logic specifications,” *2015 23rd Mediterr. Conf. Control Autom.*, pp. 940–946, 2015.

[18] G. J. Holzmann, “The Model Checker,” *Ieee Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, 1997.

[19] P. Gastin and D. Oddoux, “Fast LTL to büchi automata translation,” *Lect. Notes*

Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics), vol. 2102, no. 1, pp. 53–65, 2001.

[20] O. Khatib, “Real time obstacle avoidance for manipulators and mobile robots,” *Int. J. Robot. Res.*, vol. 5, no. 1, pp. 90–98, 1986.

[21] S. S. Ge and Y. J. Cui, “New potential functions for mobile robot path planning,” *Robot. Autom. IEEE Trans.*, vol. 16, no. 5, pp. 615–620, 2000.

[22] S. L. Smith, T. Jana, C. Belta, and D. Rus, “Optimal path planning for surveillance with temporal-logic constraints,” vol. 30, no. 14, pp. 1695–1708, 2011.

[23] O. Montiel, R. Sepúlveda, and U. Orozco-Rosas, “Optimal Path Planning Generation for Mobile Robots using Parallel Evolutionary Artificial Potential Field,” *J. Intell. Robot. Syst.*, vol. 79, no. 2, pp. 237–257, 2015.

[24] E. Wallace, “Finite State Machine Designer,” *2010*. [Online]. Available: http://madebyevan.com/fsm/. [Accessed: 14-Feb-2018].

[25] W. Chen, X. Wu, and Y. Lu, “An improved path planning method based on artificial potential field for a mobile robot,” *Cybern. Inf. Technol.*, vol. 15, no. 2, pp. 181–191, 2015.

[26] Beard, Daniel, DStarLiteJava source code, [Source code]. 2012 https://github.com/daniel-beard/DStarLiteJava