People’s democratic republic of Algeria

Ministry of Higher Education and Scientific Research

University of M’Hamed BOUGARA, Boumerdes

A Thesis Presented in Partial Fulfillment of The requirement for the Master’s degree

Major: Control engineering

Title:

FEEDBACK MOTION PLANNING FOR TETHERED MOBILE ROBOTS

by

- Billel HAMBLI
- Redha CHAREF

Supervisor:

- Dr. R GUERNANE

September, 2021

Abstract

In this report titled Feedback motion planning for tethered mobile robots, we employ many concepts in order to arrive at a methodology by which path planning of tethered robots can be achieved using feedback. These concepts include constructing a map of the encountered homotopic classes in our environment, building an augmented and tether aware virtual potential field that is responsible for both the advancement towards the goal and the retraction to the anchor of the tether, and the use of path shortening and length calculation algorithms. For the purpose of simplicity, our approach is only limited to the cases where there is no tether crossing while being wrapped around an obstacle. A discrete implementation of the suggested strategy using wavefront planner is presented as a proof of concept.

Acknowledgments

We would like to express our sincerest gratitude and most honest appreciation to our supervisor Dr. GUERNANE.R, for all the invaluable guidance he provided during this project and his many suggestions that helped refine the concepts that we tried to build. This appreciation also extends to his role in exposing us to the very interesting and fun field of motion planning. And for all of that, we are very grateful.

Another special appreciation goes to all the hardworking teachers and staff of our institute, and all the people who impacted us through this journey, without forgetting all the beautiful friendships we built while being students here and that will be maintained for the rest of our lives.

We would also like to thank and congratulate each other for delivering this project, and not giving up despite the many obstacles we encountered in our path and the instances where we had no clear plan on what to do.

Finally, our gratitude go to our parents and families for their consistent support and continuing caring and belief in us.

All these individuals have influenced us and this project and we express our dearest thanks and appreciation to them.

Table of contents

List of figures

Introduction

Human activity in many sectors is nowadays increasingly supported or substituted by robots, ranging from typical industrial to autonomous ones that are used for complex activities such as space exploration. Moreover, their remarkable versatility and flexibility allows them to be employed in a wide range of industries while handling a large number of tasks. When designing an automated system, one of the most important problems to consider is the motion planning of the robots, which means their ability to automatically determine a sequence of actions that are needed to transition from a start to a goal state. In advanced robotics applications, motion planning is certainly difficult, especially for robots with a high degree of autonomy or ones operating in hostile environments (space, underwater, nuclear, etc.). In such environments, wireless signal may not be strong enough for an operator to communicate to a robot. For instance, following the Fukushima incident, robots were placed in the reactor building due to the radioactive environment. In such situations, tethering the robot with power and communication lines is an effective solution. While tethering solves the problem of communication and power for mobile robots, it also creates numerous planning challenges. The cable is rigid and has a finite length, which limits the mobile robot's workspace around the fixed base point. Furthermore, due to obstacles, certain robot positions are only accessible through specific cable configurations (the homotopy class of the cable).

The branch of motion planning for tethered mobile robots is unfortunately still underdeveloped, due to the many complications of adding a tether. However, this is not a disparaging statement but rather an optimistic one, shedding the light on the many possibilities of applying typical motion planning techniques for solving this problem, and this is what motivated this work.

In this report, we consider the path planning problem for a tethered mobile robot. The work space of the tethered robot is limited by the cable of maximum length L, that connects the robot to a fixed base. Our algorithm design uses the feedback motion planning approach to guide the robot to the goal location or retract the tether and use a shorter path in case the length L is not sufficiently enough. Throughout this report we aim to achieve the following goals:

- Building a rigid base for tethered mobile feedback motion planning.
- Improving on the traditional distance calculation algorithms for the tether.
- Implementing this methodology for wavefront planners.
- Ensuring simplicity and consistency in the approach.

The following report will be structured as follows:

- In Chapter I, the concept of motion planning is introduced along with a classification of planners and methods that can be used to solve a motion planning problem. This is followed by a definition of the configuration space and the different methods for its construction. Homotopy and homotopic equivalence are then discussed, leading to our final section of this chapter which is tethered mobile robots and the main constraints imposed on the robot due to the inclusion of the tether.
- Chapter II serves as a display of the general strategy for applying feedback planning to tethered robots. The general procedure is presented first, then, the extended strategy for discrete environment is explained.
- Chapter III includes the used methods and are individually delved into one by one and explained in detail. These are algorithms used for the construction of the C-space, the recognition of the different homotopy classes, the preservation of homotopy, as well as the retraction and advancement and finally the path length and reduction algorithms.
- Chapter IV includes the results of our applied research, where the simulation tool that is PyGame is discussed first. A coding approach is then introduced to explain the many phases and modes for code execution. Some chosen results are then presented; these results cover the many cases that we try to tackle in our methodology. The simulation results are discussed and classified. This is followed by a review of the suggested Motion planning methodology and comparison of the simulated results to the expected results.

Chapter 1. Theoretical Background

This chapter will serve as an overview of the theory needed to examine this report. The reader will be first introduced motion planning, types of motion planning problems, types of planners, as well as the different methods for tackling a motion planning problem which will include Feedback motion planning. Then, one of the most important concepts of motion planning that is the Configuration space is defined, along with the approach to construct C-space obstacles, where a general idea about Convex Hulls and some of the most well-known algorithms employed to construct them are presented.

Half-way through this chapter, Tethered mobile robots will be introduced to the reader who'll be familiarized with the many advantages and cases for the tether use, in addition to the different constraints that will be imposed on the robot's motion and reachable space. Here, the effect of the initial configuration of the tether or alternatively its homotopy class on reachability is shown. Necessitating the discussion about homotopy and homotopic classes.

1.1 Introduction to motion planning

Motion planning can be defined as the computational problem of finding a robot's motion plan, or a consecutive set of actions to move from a start state to a goal state, while avoiding the obstacles in the environment and satisfying the constraints. Motion planning tackles problems in many areas, allowing many applications such as surgical planning, automated parking, robotics arms and other various applications.

Motion planning is often called The Piano's Mover Problem [1], which refers to a classic problem where a piano must be moved to a certain location in the room for the purpose of a furniture rearrangement. This problem demonstrates the many issues that can face the motion planning process from the collision prevention with furniture, to the prediction of the necessary steps to move the piano.

Robot motion planning usually ignores dynamics and other differential constraints and focuses primarily on the translations and rotations required to reach the goal state. However, recent work does consider other aspects, such as uncertainties, differential constraints, modeling errors, and optimality.

1.2 The ingredients of motion planning

A motion planning problem is often characterized by many ingredients, each of which presents a certain aspect of the problem [2]. These components are discussed below.

State. Planning problems involve a state space that captures all possible situations that could arise. The state could, for example, represent the position and orientation of a robot, the locations of tiles in a puzzle, or the position and velocity of a helicopter. the state space is usually represented implicitly by a planning algorithm. In most applications, the size of the state is too large to be explicitly represented.

Time. All planning problems involve a sequence of decisions that must be applied over time. Time might be explicitly modeled, as in a problem such as driving a car as quickly as possible through an obstacle course. Time may also be implicit, as in the case of solving the Piano Mover’s Problem, the particular speed of moving the piano is not specified in the plan.

Actions. A plan that handles the problem generates actions that manipulate the states. In the planning formulation, it must be specified how the state changes when actions are applied over time. For most motion planning problems, explicit reference to time is avoided by directly specifying a path through a continuous state space.

Initial and goal states. A planning problem usually involves starting in some initial state and trying to arrive at a specified goal state or any state in a set of goal states. The actions are selected accordingly to make this happen.

Feasibility or Optimality. a feasible plan that causes arrival at a goal state, regardless of its efficiency. An optimal plan optimizes performance of a feasible plan in some carefully specified manner, in addition to arriving in a goal state.

A plan. In general, a plan imposes a specific strategy or behavior on a decision maker. A plan may simply specify a sequence of actions to be taken; however, it could be more complicated.

1.3 Types of motion planning problems

Motion planning problems vary from one another depending on many factors that can include time, cost of operation, degree of precision and the desired outcome [3].

1.3.1 Path Planning vs. Motion Planning

Path planning is a purely geometric subset of motion planning that is aimed to find a non-collision path $q(s)$, where $s \in [0,1]$such that: $q(0)=q_{sta\,rt}$and $q(1)=q_{goal}$(assuming [0 ;1] is scalable). On the other hand, motion planning isn't necessarily geometric as the goal there is to transition from a start state to a goal state.

1.3.2 Online vs. Offline planning

Depending on the complexity of the problem, in terms of the number of obstacles or the unpredictability of their movement a corresponding planner is chosen accordingly. Offline planners are usually slower and used to process the available and non-changing data and come up with a plan to be executed with no further processing, whereas Online planners react to the environment and constantly come up with plans until the goal state is reached.

1.3.3 Optimal vs. Satisfiying

In some cases, finding a feasible plan is not enough, as on top of that the minimization of some cost parameter J might be desirable or even essential for the task at hand, such that:

$J=\int_0^T L(x(t),u(t))$, in most cases, finding the optimal motion plan might be computationally exhausting and time consuming, and in some situations a compromise is taken to obtain a satisfying solution.

1.3.4 Exact vs. Approximate

In a time interval [0;T], if the state x(T) is close enough to the goal state x_{goal}, x(T) may be accepted as an approximate solution if : $\|x(T)-x_{goal}\|<\epsilon$.

1.4 Classification of motion planners

A planner is a term that refers to an algorithm or a technique that is used to find a path or a set of transitional states between a start and a goal state. Planners can be classified into different categories, depending on the aspect of comparison. Consequently, different classifications naturally arise and are mentioned below.

1.4.1 Single vs. Multiple query planners

Single query planners are algorithms that are designed for an unchanging environment, with a built-in data set to represent C_{free}, which makes them highly efficient for their chosen environment. On the other hand, Multiple query planners tend to be general purpose planners that perform the processing for each problem from scratch and don't store data on the environment.

1.4.2 Completeness

Generally, completeness is the first measure to look into when designing or evaluating planners, which are then sorted in one the following categories.

a. Complete Planners

This type of planners is guaranteed to find a solution in a finite amount of time, if one exists, or report a failure in the case where no feasible plan is possible, at the condition that a complete representation of the C-space is provided.

b. Resolution Complete Planners

It is a weaker claim to completeness, that is present in discretely represented environments. These planners are guaranteed to find a solution in a finite amount of time, if one exists in a discretized representation of the problem.

c. Probabilistically Complete Planners

These planners' probability of finding a solution, if one exists, is 1 as $t \to \infty$. Sampling methods such PRM and RRT are a great demonstration for this concept.

1.4.3 Global vs. Local Planners

Motion planning algorithms can be classified to global and local planners. Global algorithms plan a motion from the starting to the goal configuration in a static environment, using a pre-planned map. A pre-planned map may be insufficient for a robot to be able to reach its goal in the case of dynamic environment with moving obstacles. This necessitate the use of a local planner.

1.4.4 Anytime Planners

These planners continue to search for better solutions after a first one is found, and can be stopped at any time after finding the first solution.

1.5 Path Planning Methods

1.5.1 Visibility Graphs

Here, we represent the complex high dimensional space C_{free}, by a one-dimensional roadmap *R* or *Visibility Graph* with the following properties:

- **Reachability:** From every point $q \in C_{free}$, a free path to a point $q' \in C_{free}$can be found trivially.
- **Connectivity:** For each connected component of C_{free}, there is one connected component of R.

A visibility graph is a network of intervisible locations in computational geometry and robot motion planning, typically for a set of points and obstacles in the Euclidean plane. Each vertex in the network is a point location, and each edge denotes a visible link between them. In other words, if the line segment linking two points does not pass through any obstacles, an edge is drawn between them in the graph.

For example: for a polygonal robot with polygonal obstacles, we construct an undirected visibility graph, where the weight associated with each edge is the Euclidean distance between the nodes, then use the A* algorithm to find the shortest path from the start point to the goal point.

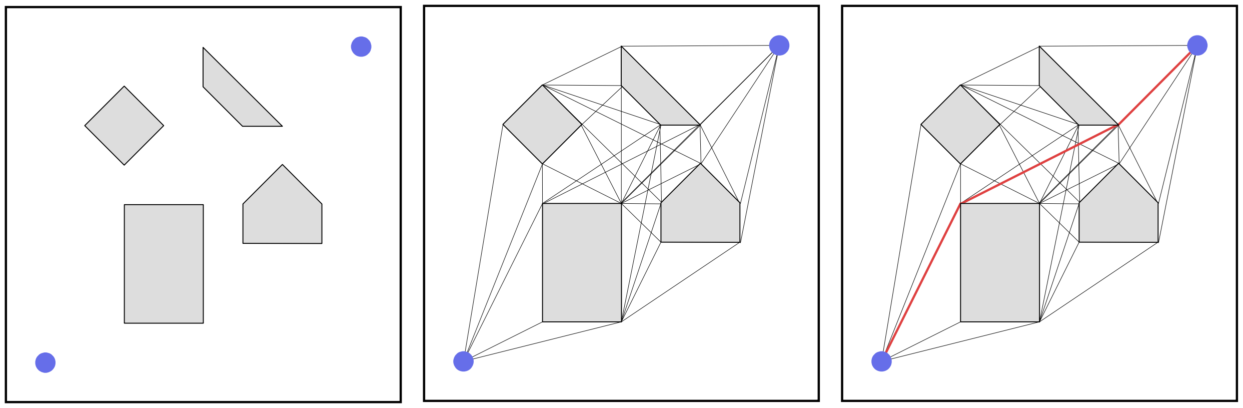


Figure 1.1: Given a set of polygonal obstacles, a visibility graph is constructed and then A* algorithm is used to find the shortest path

1.5.2 Grid-based methods

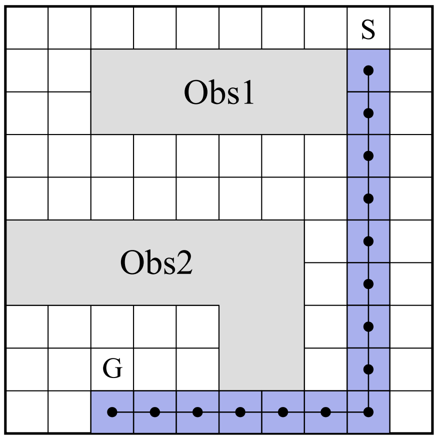


Figure 1.2: The shortest path is found in Cfree, and the obstacles are in gray

Grid-based approaches overlay a grid on the configuration space, and assume each configuration is identified with a grid point. At each grid point, the robot is allowed to move to adjacent grid points as long as the line between them is completely contained within (this is tested with collision detection). These methods are easy to implement and can return optimal solutions, but for a fixed resolution, the memory and time required to search the grid grows exponentially with the number of dimensions of the space.

For an n-dimensional configuration space and k-desired grid points along each dimension, the C-space is represented by k^n grid points. The A* can be used to navigate this C-space grid with the following modifications:

• Determining if the robot is constrained in axis aligned direction or can move in multiple dimensions simultaneously, in the last case the cost is the Euclidean distance.

• If only axis-aligned motions are used, the cost-to-go should be the Manhattan distance(the sum of the distances from only axis-aligned paths).

• Nodes are only added to the grid space if the path to each is collision-free.

1.5.3 Sampling-based methods

Roadmaps rely on an explicit representation of the free space, As a result, as the dimension of the configuration space grows these methods become impractical [3]. Sampling methods avoid the explicit construction of grids, and instead employ a variety of strategies for generating samples and for connecting the samples with paths to obtain solutions to path-planning problems. These types of algorithms rely on a random or deterministic function to choose a sample from the C-space, and a simple local planner to try to connect to, or move towards the new sample. These functions are used to build up a graph or a tree representing feasible motions of the robot. The two types of sampling-based planners are rapidly exploring random trees (RRTs) and probabilistic roadmaps (PRMs), where the former uses a tree representation and is particularly effective for single-query planning, while the latter is used for multiple-query planning.

1.5.4 Virtual Potential Fields methods

These methods are inspired by potential energy fields in nature, such as gravitational and magnetic fields. For example, a charged particle navigating through a magnetic field or a marble rolling down a hill. The basic idea is that the behavior of both the marble and particle will depend on the shape of the environment.

In the potential field approach, the robot in the configuration space is considered as a moving point subject to a potential field generated by the goal configuration and the obstacles in the C-space. The target configuration produces an attractive potential, while the obstacles generate a repulsive potential [4]. The sum of these two contributions is the total potential, which can be seen as an artificial force applied to the robot, aimed at approaching the goal and avoiding the obstacles. Thus, given any configuration during the robot motion, the next configuration can be determined by the direction of the artificial force to which the robot is subjected.

The major drawback of this method is that the robot can get stuck in local minima of the potential field, away from the goal, even when a feasible motion to the goal exists. This may occur, for example, when multiple obstacles surround the robot. Several solutions have been proposed to overcome this problem, namely, the RRP (Random Path Planners), a special planner used to avoid getting in a local minima by combining the concepts of artificial potential field with random search techniques.

a. Wave-Front Planner

The wavefront expansion algorithm is a specialized potential field path planner that avoids local minima by using a breadth-first search strategy. It revolves around the robot in a widening circle. The closest neighbors are analyzed initially, and then the circle's radius is increased to distant areas.

Before planning a path for the robot, the map is discretized into a grid. The vector data is transformed to a two-dimensional array and saved in memory. For each cell, the potential field path planning algorithm decides the robot's direction. This direction field is projected over the robotic map, which includes the robot and obstacles. The planner begins with a standard binary grid of zeros associated to free space and ones (or -1) related to obstacles. The start and goal locations are also known to the planner. A two is assigned to the goal pixel. All zero-valued pixels adjacent to the goal are labeled with a three in the first step. Following that, all zero-valued pixels adjacent to threes are given the number four. This approach essentially creates a wave front from the goal, with all pixels on the wave front having the same path length to the goal, measured with respect to the grid, at each iteration. When the wave front reaches the pixel containing the robot start location, the procedure ends.

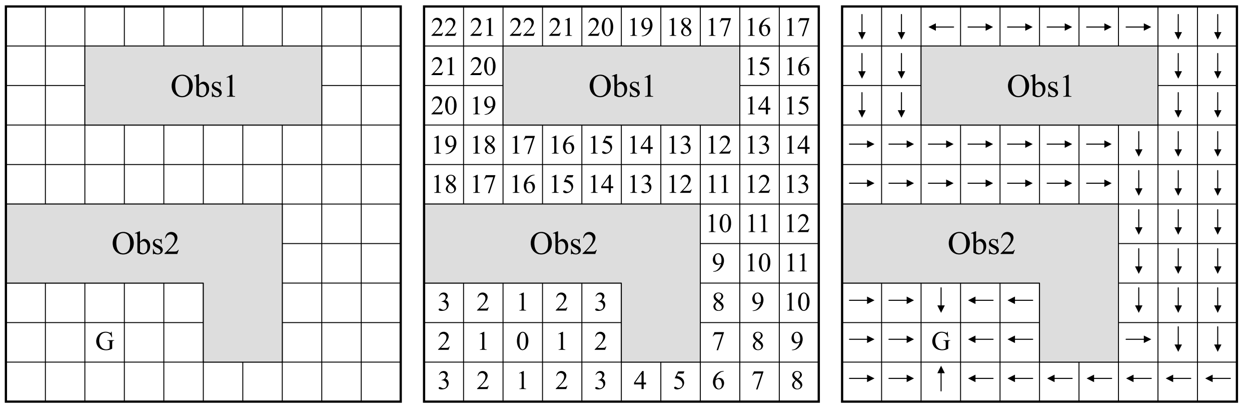


Figure 1.3: The steps of motion planning using a wavefront planner: 1. the C-space is discretized, 2. a wavefront propagation is initiated from the goal point, 3. the robot follows the wave down till reaching the cell with the value 0

The planner then calculates a path on the grid using gradient descent, starting from the beginning. The planner essentially calculates the path one pixel at a time. Assume that the start pixel's value is 22. Any surrounding pixel with a value of 21 is the next pixel in the path. There could be several options; simply select one of them. The following pixel has the value of 20.

A pseudo-code for the wave propagation algorithm is provided below, to further explain how this process works.

Wave Propagation Algorithm

```
1   Initialize explorationQueue as empty
2   StartNode = (Goalx,Goaly,1)  //assign goal node coordinates to starting node with flowValue 1
3   Append startNode to explorationQueue
5   While explorationQueue
6       MainNode = explorationQueue[0]
7       Pop the first element of explorationQueue
8       if flowValue of node north of MainNode = 0 and MainNode < top display border then
9           North = node north of mainNode
10          Increment flowValue of North by 1
11          Append North to explorationQueue
12      if flowValue of node south of mainNode = 0 and mainNode > bottom display border then
```

```
            South  =  node south of mainNode
            Increment flowValue of South by 1
            Append South to explorationQueue
        if flowValue of node east of mainNode = 0 and mainNode < left display border then
            East  =  node east of mainNode
            Increment flowValue of East by 1
            Append East to explorationQueue
        if flowValue of node west of mainNode = 0 and mainNode > right display border then
            West  =  node west of mainNode
            Increment flowValue of West by 1
            Append West to explorationQueue
    Return flowValue
```

something that we can add here that is not necessarily feedback motion planning tool but can help finding the shortest path a wave propagation is the Dijkstra algorithm.

- **Dijkstra algorithm**

Dijkstra's algorithm is a very simple algorithm for finding the shortest paths between between a start node S and a goal node G in a graph. This algorithm was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later [12].

Given a weighted graph as the one described by generated by a wave propagation, the operation of the Dijkstra can summarized in the following points:

1. From the current node, all neighboring unvisited nodes are found and stored in a list.
2. From this list, the node with the smallest assigned distance measure (in this case the measure is cost-to-go) is our new current node, and the previous node is marked as visited, and stored in the shortest path list.
3. If the current node is the goal point, it’s marked visited and added to the shortest path list. And the algorithm execution is terminated.

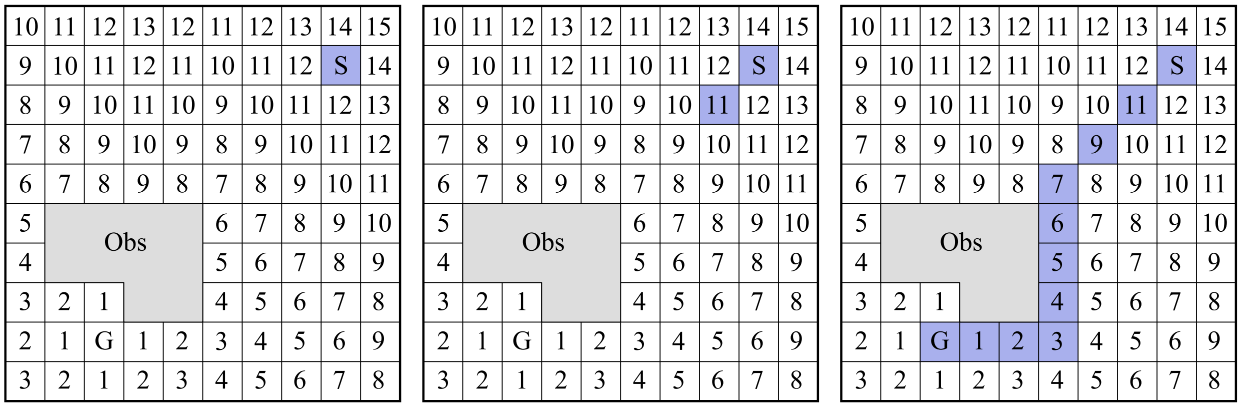


Figure 1.4: A visualization of the Dijkstra's Algorithm path searching in a discrete environment

1.6 The Configuration Space

1.6.1 Geometric Modeling

Formulating and solving motion planning problems requires defining and manipulating complicated geometric models of a system of bodies in space [2]. The most widely used approaches and techniques for geometric modeling are:

- **Boundary Representation**: In which entities are described by equations that roughly or exactly represent the object's surface.
- **Solid Representation**: In which entities are described by all points contained inside.

Let W denote our workspace that contains the robots and obstacles and can be described in a $2D$ world (in which $W=R^2$), or a $3D$ world (in which $W=R^3$). The world generally contains two kinds of entities:

- **Obstacles:** Portions of the world that are "permanently" occupied, for example, as in

the walls of a building.

- **Robots:** Bodies that are modeled geometrically and are controllable via a motion plan.

Let O denote the Obstacle region that is the set of all points W lying in one or more obstacles. Our goal is to represent O in the most expressive and computationally effective way.

1.6.2 Configuration space definition

Let ***c*** be a point corresponding to a unique configuration that describes the pose of the robot; the configuration space **C** is the set of points corresponding to all possible configurations.

For example, If the robot is a circle translating in a *2-D* plane (the workspace W), C is a plane, and a configuration can be represented using two parameters (x, y).

- **Free space**

The free space C_{free} is a set of all configurations where the robot avoids colliding with obstacles.

- **Obstacle space**

An obstacle in C-space is the set of points that are not legal configurations, for a certain obstacle in the workspace. The sum of these C-space obstacles is the Obstacle space C_{obs}.

For the planar circle-sized robot below, an obstacle in c-space is a set of (x, y)-positions of the robot that are not allowed (because the robot would be colliding with an obstacle if it tried to achieve that position).

1.6.3 Configuration space construction

For a planar robot capable of both x and y translation but not of rotation, the concept is easy; we pick a reference point on the robot and swipe it around the workspace obstacle while tracing out the reference point. That reference point will then represent our robot in C-space. Since swiping the robot around the workspace might not be efficient, Instead the robot (or the robot's reference point) is placed at each vertex of the obstacles and its new node coordinates are added to a set M_i corresponding to the current obstacle. Then each set M_i is used to construct the corresponding C-obstacle using either the Gift Wrapping Algorithm or the Chan's Algorithm that will be discussed later.

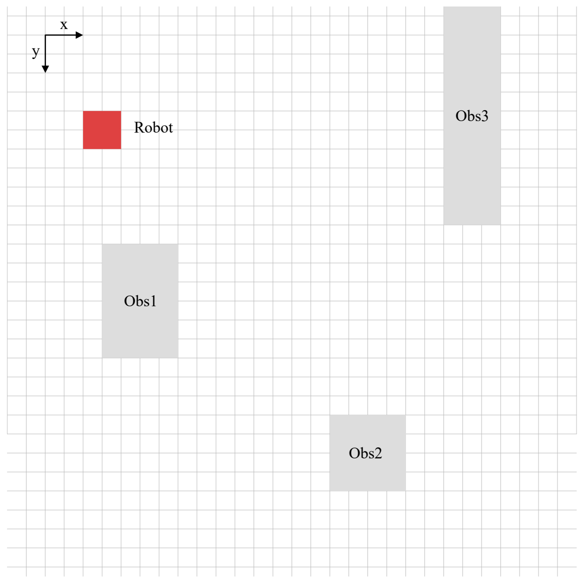


Figure 1.5: A workspace environment with a square robot

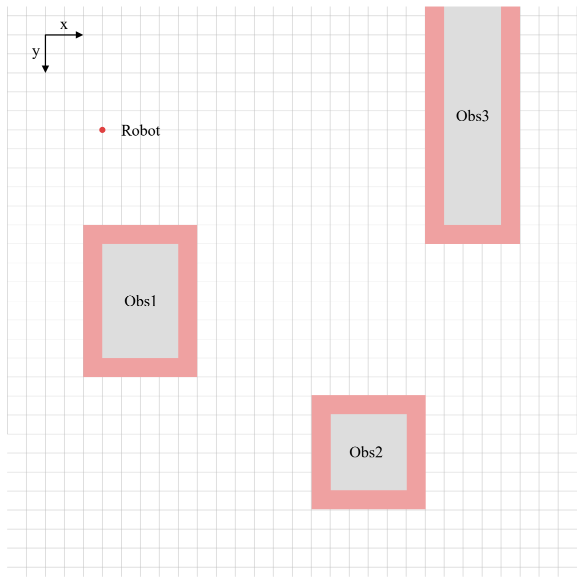


Figure 1.6: A configuration space where the C-space obstacles were constructed and the robot is a point

- **Convex Hulls**

In geometry, a subset of a Euclidean space, is convex if, given any two points in the subset, the subset contains the whole line segment that joins them [5].

X is a convex polygonal, if and only if, for every two points x_1 and x_2 of *X*, all the points on the line between x_1 and x_2 also belong to *X*, Implying $\lambda x_1 + (1-\lambda) x_2 \in X$ such that $\lambda \in [0;1]$.

A variety of mathematical approaches have been developed to construct a convex polygonal *X* given a set of points *M;* two of the most well known and used Algorithms are *Jarvis' March* and *Chan's Algorithm*.

- **Jarvis's march/ Gift-wrapping Algorithm**

In computational geometry, the gift wrapping algorithm is an algorithm for computing the convex hull of a given set of points. It is also known as Jarvis's March, after R. A. Jarvis, who published it in 1973; it has O(*nh*) time complexity, where *n* is the number of points and *h* is the number of vertices of the resulting convex hull, thus we can say that it is "Output Sensitive". Its real-life performance compared with other convex hull algorithms is favorable when n is small or h is expected to be very small with respect to n [6].

The idea of Jarvis's Algorithm is simple; Given a set of points *M,* we start from the leftmost point (or point with minimum x coordinate value) and we build a semi-infinite line that rotates counterclockwise around the leftmost point. The first point in *M* that this line intersects with is added to the list of vertices and becomes the new center of rotation for the semi infinite line starting from the intersection angle, this operation is repeated until the intersection point is the left most point in *M*. When the latter is reached, the list of vertices will contain all nodes of the convex hull ordered counterclockwise.

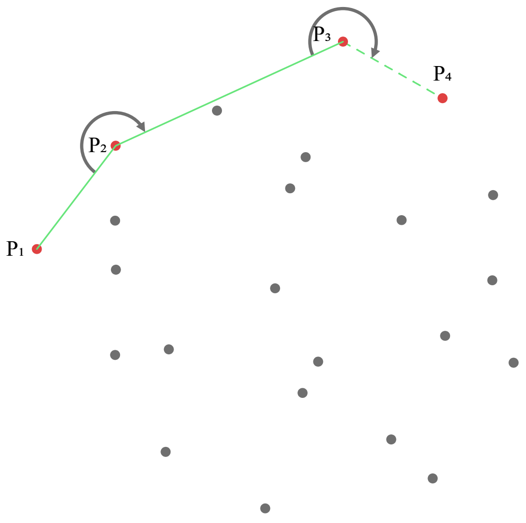


Figure 1.7: The rotation of the semi infinite line starting from p1 to p2 to p3 then p4

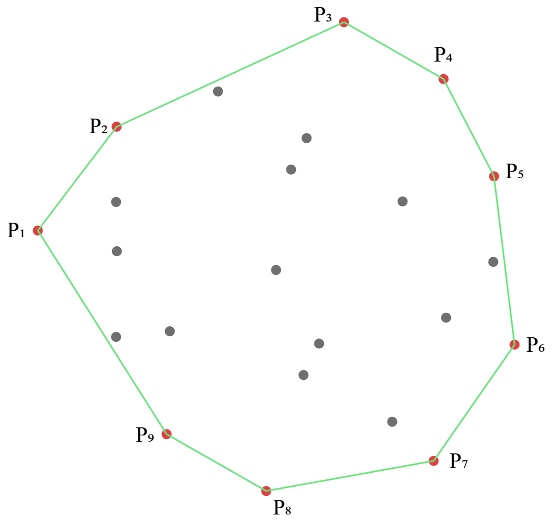


Figure 1.8: A convex hull has been constructed using the gift wrapping algorithm

A pseudo code [6] is also provided for this algorithm, down below.

Jarvis's March Algorithm

```
    // S is the set of points
    // P will be the set of points which form the convex hull. Final set size is i.
1   pointOnHull = leftmost point in S  // which is guaranteed to be part of the CH(S)
2   i := 0
3   Repeat
4       P[i] := pointOnHull
```

```
        endpoint := S[0]    // initial endpoint for a candidate edge on the hull
        for j from 0 to |S| do
            // endpoint == pointOnHull is a rare case and can happen only when j == 1 and a better endpoint has not yet been set for the loop
            if (endpoint == pointOnHull) or (S[j] is on left of line from P[i] to endpoint)
                endpoint := S[j]   // found greater left turn, update endpoint
        i := i + 1
        pointOnHull = endpoint
    until endpoint = P[0]     // wrapped around to first hull point
```

- **Chan's Algorithm**

In computational geometry, Chan's algorithm, named after Timothy M. Chan, is an optimal output-sensitive algorithm to compute the convex hull of a set P of n points, in 2 or 3 dimensional space. The algorithm takes $O(n \cdot \log(h))$ time, where *h* is the number of vertices of the output (the convex hull). In the planar case, the algorithm combines an $O(n \cdot \log(n))$ algorithm (Graham scan, for example) with Jarvis march $O(n \cdot h)$, in order to obtain an optimal $O(n \cdot \log(h))$ time [7].

1.7 Tethered mobile robots

Many practical scenarios necessitate the use of a tether by a robot. Attaching a tether to robots can be used to provide power (to save the mass and volume of the batteries), especially to robots which perform high power tasks. The tether can also be used for reliable high-speed communications (e.g., underground and underwater robots), or to track a robot's position, or as a safety harness (in case the robot needs to be dragged out) [8]. Tethering also helps in deploying robot in environments with limited accessibility. For example, Nassiraei et Al. designed a tethered sewer pipe inspection robot to work instead of a human operator to decrease the cost and to speed-up the inspection [9]. In some cases, the cable is not essential but it can boost the task's performance. For example, Cables have also been used for manipulation by helping robots collect or separate objects. These examples emphasize the need of efficient solutions for solving tethered robot motion planning problems.

1.7.1 Tether Constraints

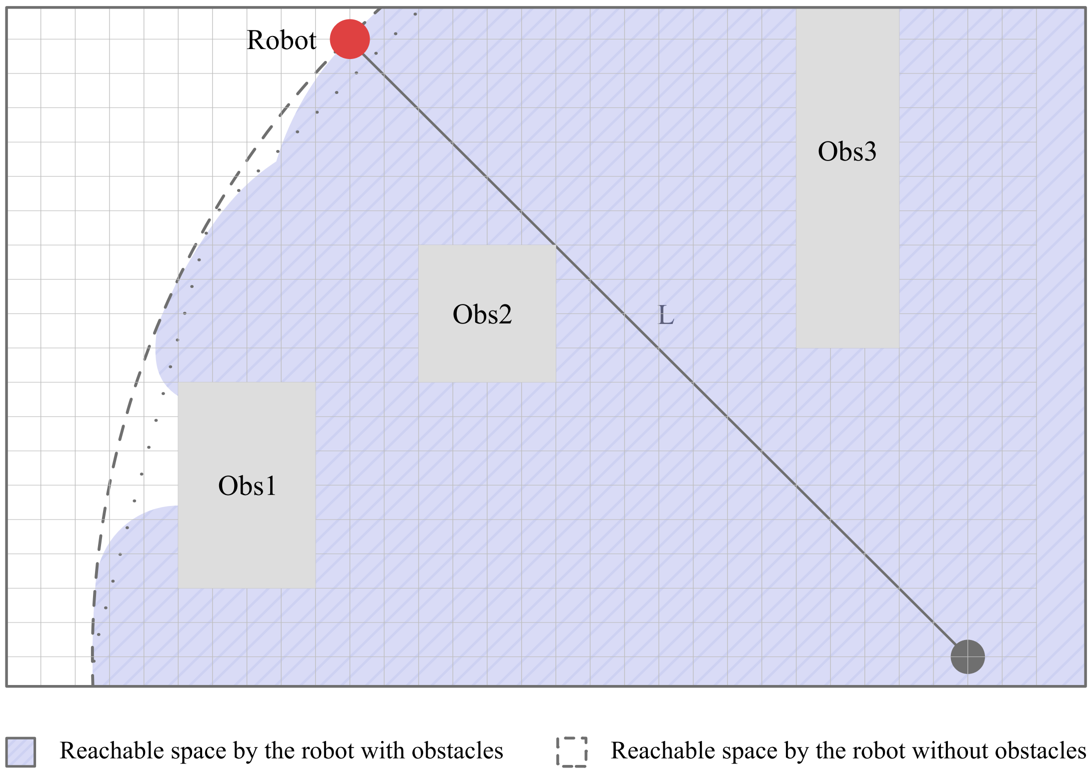

Robot
Obs3
Obs2
L
Obs1
Reachable space by the robot with obstacles
Reachable space by the robot without obstacles

Figure 1.9: The reachable space by robot that's tethered by a cable of length L

In motion planning for tethered robots, along with the usual constraints that are considered in any motion planning problem (e.g., collision avoidance, distance, etc.), two key additional constraints are imposed on the robot's mobility due to the presence of the cable:

1. The radius of the robot’s movement is limited by the cable’s length. Consider the example in the provided figure. The robot moves in a known, bounded workspace, $\mathcal{W}$ (a subset of $\mathbb{R}^2$, which is of interest). One end of a cable, whose length is L, is attached on the robot while the other end is anchored on a fixed point, base, at $\mathbf{q_b}$. In the absence of any obstacles, the reachable space of robot will be the intersection of $\mathcal{W}$ and a disk of radius L centered at the base, $\mathbf{q_b}$.

2. The presence of obstacles introduces geometric constraints as well as topological constraints. For instance, in Figure 1.9, the point can only be reached if the cable configuration lies in the appropriate homotopy class (the homotopy classes of C1 or C2, for example, but not the homotopy class of C3). In other words, if the robot takes the path on the left of the obstacle O3, it must retract its cable and return to its initial position to take one of the two other paths that gets it to the goal point q_g.

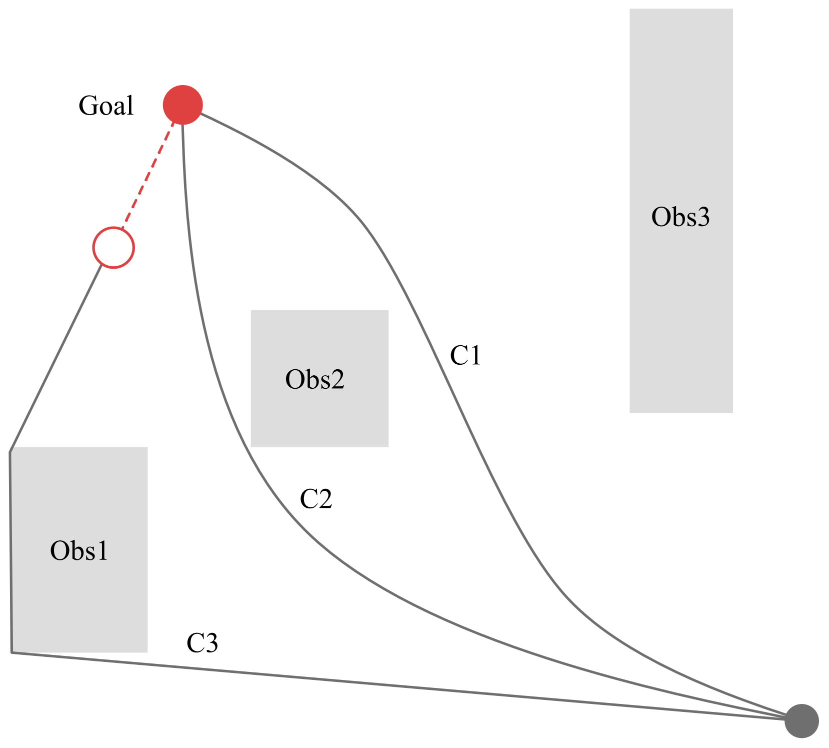


Figure 1.10: Because of the tether length constraint, the goal can be reached via some homotopy classes (C1 and C2), but not others (C3).

1.7.2 Homotopy of paths and cables

In the previous figure, the paths C1, C2 and C3 each belong to a distinct homotopy class. These classes of trajectories arise due to the presence of obstacles in an environment. Two trajectories with the same start and goal coordinates are said to be in the same homotopy class, if one can be smoothly deformed into the other without intersecting any obstacle in the environment, otherwise, they are in distinct homotopy classes. In many applications, it is important to distinguish between trajectories in different homotopy classes [10].

In the last decades, the classification of homotopy classes in 2-D spaces has been the subject of many research papers exploring the various methods to do so. These methods range from geometric to PRM-based and triangulation-based.

When planning motion for tethered robots, it is essential to employ the concept of homotopy provided an information about obstacles is given initially. By having an idea about the different classes in a certain environment, the motion planning for tethered robots becomes much easier as the robot gains awareness about the tether configuration. This ensures a safe passage to the goal respecting the initial configuration of the tether.

a. Defining homotopy

A homotopy between two continuous functions f and g from a topological space X to a topological space Y is formally defined as a continuous function $H: X \times [0,1] \rightarrow Y$ from the product of the space X with the unit interval [0, 1] to Y such that $H(x,0)=f(x)$ and $H(x,1)=g(x)$ for all $x \in X$.

b. Homotopy classes of curves

Two curves γ_1, γ_2: $[0, 1] \rightarrow (\mathcal{W} / \mathcal{O})$ connecting the same start and end points, are homotopic (or belong to the same homotopy class), if and only if one can be continuously deformed into the other without intersecting any obstacle. This is illustrated in Figure 1.10.

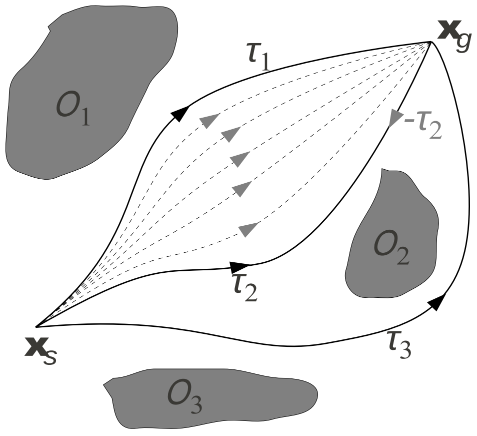


Figure 1.11: Path homotopy: paths γ1 and γ2 between qs and qg are homotopic while path γ3 belongs to a different homotopy class.

This concept of homotopy is very essential to the motion planning of tethered mobile robots, an an important basis for Curve Shortening algorithm and retraction that will be discussed in the following chapter.

Chapter 2. Methodology for motion planning

In the following chapter, The general strategy for designing a tether-aware motion planner is presented to the reader, which comprises of two parts that are reachability test and the navigation loop. After that, an extension of this general strategy can be found in section 2.3, where a discrete environment is chosen and consequently a wavefront planner is used to achieve feedback.

2.1 Introduction

Some things of high importance to note first before diving into the suggested methodology, are as follows:

- In almost all of the rest of the report, the C-space obstacles are the ones taken into consideration, except when smoothing the path, where the original workspace obstacles are the employed ones.
- This general strategy only applies for the case where there's no tether crossing when wrapping around the obstacles for the reduced or tightened version of the tether.

Now, the goals of our proposed approach can be stated as follows:

- A 2-dimensional virtual potential field (VPF) is to be built. This virtual potential field is the one responsible for both advancement towards the goal, as well as the retraction of the robot.
- A map of the different homotopy classes that are present while retracting is constructed, in order to to preserve homotopy.
- An efficient approach to calculate the shortest path taken is to be implemented.

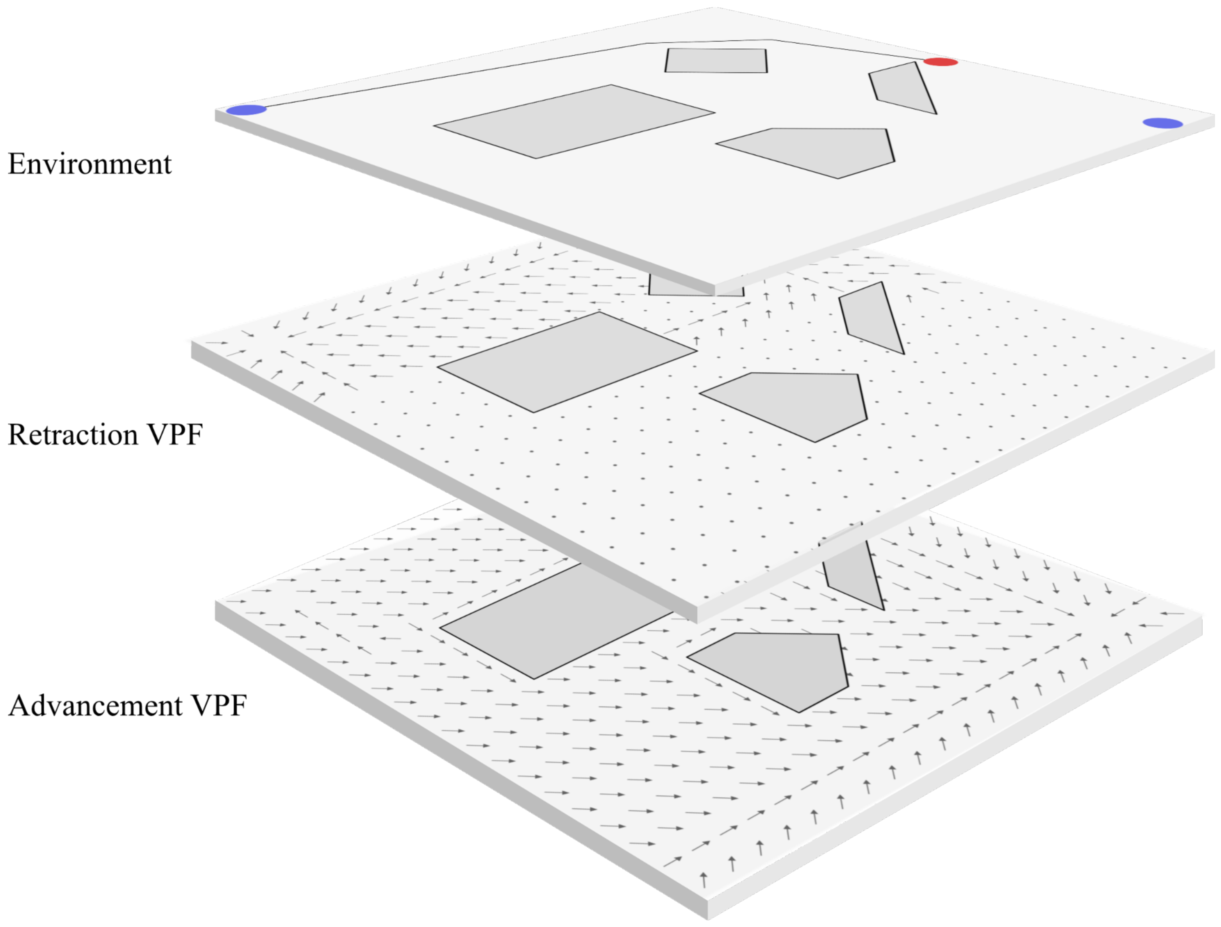


Figure 2.1: 2-D augmented virtual potential field for both retraction and advancement

2.2 General strategy for a tether aware motion planning using feedback

Given an initial configuration of our environment, which includes the anchor, start and goal point as well as the tether points, the general procedure for navigation to the goal, if possible, is presented below.

2.2.1 Reachability Test

The first step that is shared with other types of planners, is the verification that a solution is actually possible. This assessment is done by finding the shortest path from the anchor to the goal point. The distance of this path is then compared to the length of the tether *L*, which will decide the continuation of execution, if the shortest distance is less than *L*, or the termination of our process otherwise.

2.2.2 Navigation Loop

The next step is where the actual motion planning happens; here, a loop is continuously run until the goal state is reached. This loop contains different blocks that interact with each other, with each block having a distinct functionality and output.

a. Finding the Path to Goal

The shortest path from the current point of the robot to the goal « *PathToGoal* » is found and is verified if it belongs to one of the explored spaces (at the first time this is run, there are no explored spaces, so this verification step is skipped).

These explored spaces are spaces that were encountered by the robot before, and found not to have feasible paths after the calculation of the shortest distance possible when traveling in these spaces.

If *PathToGoal* doesn't belong to one of the explored spaces, we move to the next step of the loop (that is finding the shortest path). However, if it does belong to an already explored space, further processing is not needed for this current point; since the shortest path of this space has already been proven to be larger than *L*, this requires retraction and initiating the loop once again.

- ***R*****etraction**

To do such action, a retraction space must be created from the initial tether configuration; in this space, the retraction VPF is built and directed towards the anchor. Each time the robot is required to retract, it follows the retraction VPF by one step. This ensures that the backtracking of the robot respects the homotopy of the tether and prevents deviation to other homotopy classes. Something to note, is that the retraction space is built the first time a retraction is required and reused again whenever needed.

b. Calculating the shortest path distance

PathToGoal from the current point is then added to the tether, to form our complete path. This path is then shortened to find its length *l,* that is then compared to the tether length *L*. If *l* turns out to be less than *L*, a solution exists and we move to the next step; if not, the space in which the current path exists is ought to be found and added to the explored spaces.

c. Following the Advancement Field

Once the length *l* is enough to reach the goal, the robot follows the virtual potential field leading to the goal point that will be eventually reached.

In order to further demonstrate the suggested strategy, a flowchart summarizing the process described above is deployed in Figure 2.2. Preceding that, an example is shown in Figure 2.1 where the general strategy of the feedback motion planning in the presence of a tether is applied to reach the goal from the initial configuration presented in (a). In (b), A reachability test is performed and a solution exists, *PathToGoal* (red-dotted line) is then found and it doesn't belong to the explored spaces and $l > L$, hence a retraction is needed. As shown in (c), this retraction happens due to the VPF present in the retraction space, and leading towards the anchor. After few retractions, the robot reaches the configuration shown in (d); *PathToGoal* is found from this current point and so is the shortest distance *l* that turns out to be smaller than *L,* indicating that a solution is found. Due to this, the robot follows the advancement field as shown in (e) to finally reach the goal as displayed in (f).

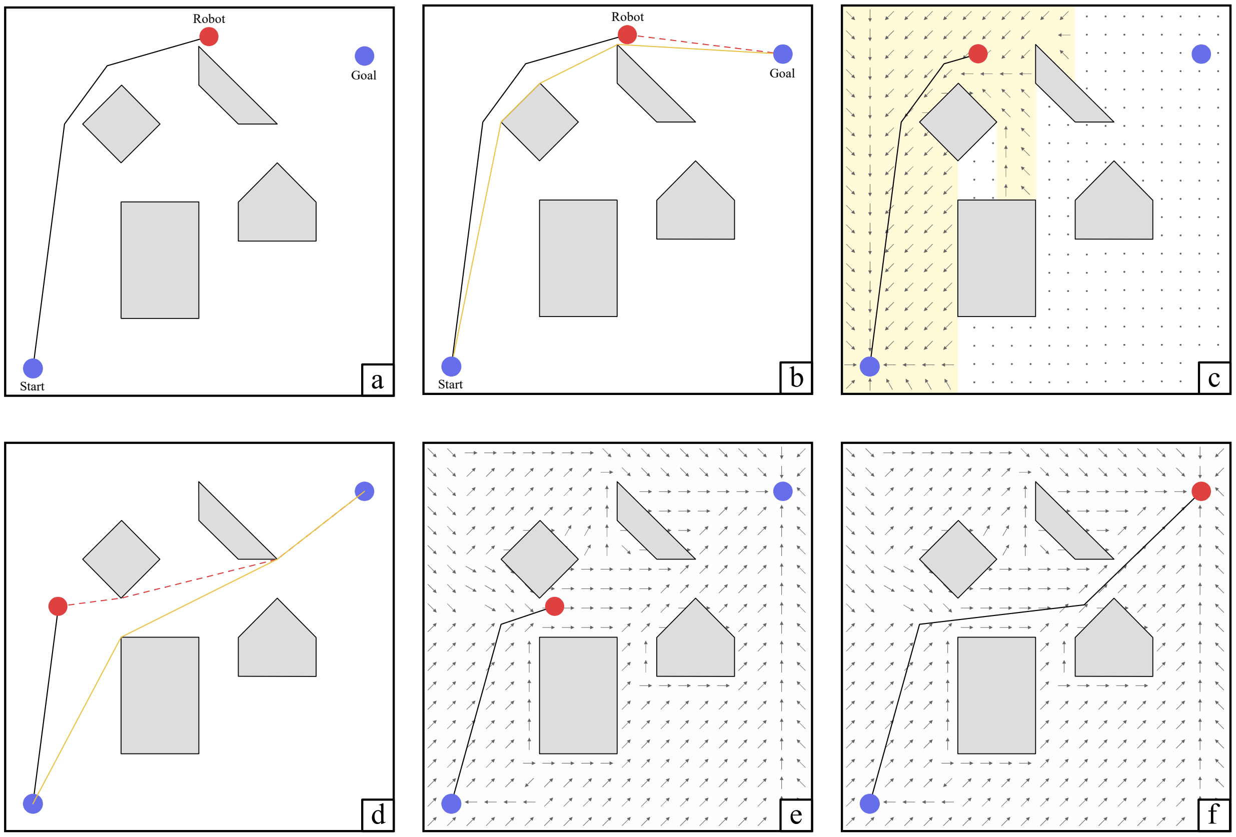


Figure 2.2: An example of the proposed general strategy of motion planning for tethered mobile robots

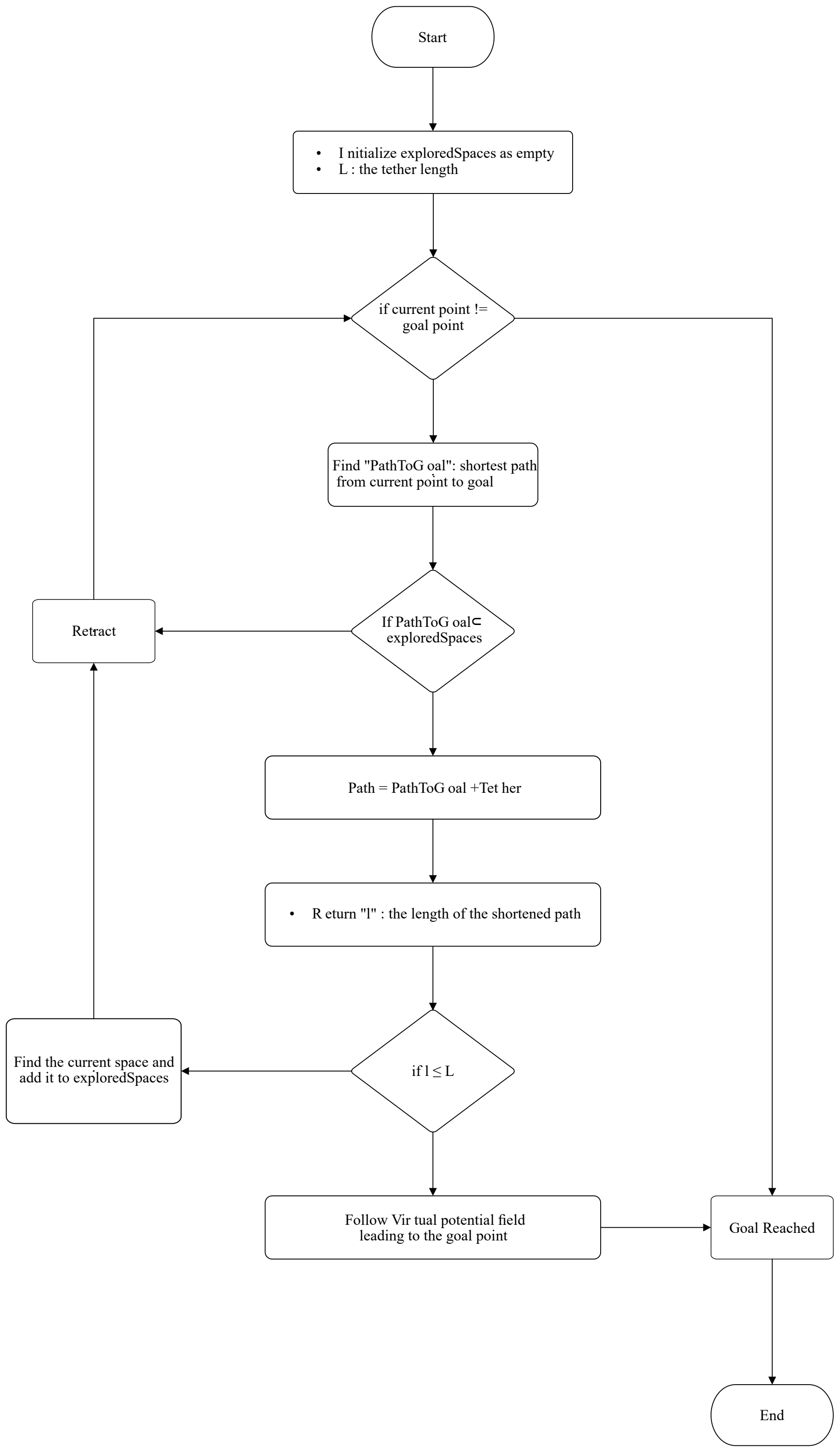

Start
• I nitialize exploredSpaces as empty
• L : the tether length
if current point != goal point
Find "PathToG oal": shortest path from current point to goal
If PathToG oal⊂ exploredSpaces
Retract
Path = PathToG oal +Tet her
• R eturn "l" : the length of the shortened path
if l ≤ L
Find the current space and add it to exploredSpaces
Follow Vir tual potential field leading to the goal point
Goal Reached
End

Figure 2.3: Flowchart for the general strategy for a tether aware motion planning using feedback

2.3 Motion planning procedure for tethered robots using wavefront planners

After introducing the general approach for tackling the problem of motion planning for tethered mobile robots, an extension of this strategy is then discussed in this section of the report.

The first step in tackling the problem of feedback motion planning is working on top of an environment that best helps with the handling of the many situations a robot can encounter, while providing a dependable and logically persistent coding approach. This is why a discrete representation is chosen. This representation provides consistency in approaching the many steps to design our algorithm and an aspect of reusability that will prove to be quite convenient throughout this report. In addition to that, discrete environments offer a basis for implementing feedback by the use of wavefront planners that were discussed in Chapter 1.

In this chapter, a discrete representation of the configuration space is assumed to be already available. The methods to obtain such environment will be covered extensively in chapter 3. With the discrete environment and the initial configuration at hand, the motion planning can start by implementing the general strategy using wavefront planners.

2.3.1 Reachability test in a discrete environment

As discussed in the general strategy, a reachability test is essential for determining the feasibility of a motion plan to the goal. In order to perform such assessment in a discrete environment, the advancement field must be built by wave propagation starting from the goal point as shown in Chapter 1; using this field, the shortest discrete path from the goal to the anchor can be found using the Dijkstra algorithm. This path is further shortened to obtain its minimum euclidean distance l using shortening algorithms that will be discussed in Chapter 3. This length l, when compared to the maximum tether length L, determines the existence of a solution and hence the execution of the rest of the process.

2.3.2 Navigation Loop:

This loop is run continuously until the robot reaches the goal cell, or until the current point becomes the goal point.

a. Finding the path to Goal

Since the advancement VPF has been created when performing the reachability test, finding the path to goal denoted *PathToGoal* is as simple as finding the shortest path from the current point to the goal using the Dijkstra algorithm. This obtained path is tested, to check if it belongs to one of the explored spaces. This checking process can done using the *BelongsTo* Algorithm whose pseudo-code is provided below.

This function takes as arguments *PathToGoal* and the list of explored spaces denoted exploredSpaces, and returns True if *PathToGoal* is a subset of one of the exploredSpaces or False otherwise.

BelongsTo Algorithm

```
For currentSpace in exploredSpaces
    For point in PathToGoal
        if point doesn't belong to currentSpace then
            Return False
        End if
    End for
Return True
```

After running this function, two outcomes are possible:

i. The function returns False, this will mean that the space has not been explored yet and we move to the next step of the loop, which is the calculation of the shortest path.

ii. The function returns True, this will mean that the space is already explored and there is no point in calculating the shortest distance as it has been already calculated and found to be not enough, hence, a retraction is needed.

- **Retraction is a discrete environment**

As discussed earlier, retraction necessitates the construction of a retraction space. To do so:

1. A discretized version of the initial tether configuration is obtained using the *DetectCell* algorithm that will be discussed in Chapter 3.

2. The space of retraction from the discretized tether cells using the *FindSpace* algorithm that is also explained in Chapter 3.
3. A wave propagation is performed from the anchor in the obtained space, as shown in Chapter 1.

To retract, the robot moves backward in this generated potential field by one step, every time it is required to.

b. Calculating the shortest path distance

The total path from the anchor to the goal point is found by adding the *PathToGoal* points that go from the current point to the goal point, to the shortest path from the anchor to the current point. This whole path is then smoothed using the shortening algorithms, to find the corresponding shortest distance *l.*

Once the distance is calculated, a crossroad is faced. If the distance of the path "*l*" is larger than "L", a retraction is the logical follow-up, this is accompanied by finding this path's space and adding it to the explored spaces using the *FindSpace* algorithm. If the latter condition is not satisfied, in other words "*l*" is smaller or equal to "L", It is concluded that a solution is found, and the robot can reach the goal from the current point.

c. Following the advancement VPF

Once a solution is confirmed to exist in the previous step, the robot is ought to follow the Advancement wave and reach the goal eventually, advancing one cell at a time. When the goal is reached, the execution of the algorithm is terminated.

Now that we have the general strategy for feedback motion planning for tethered robots using wavefront planner, the specific methods used in this implementation will be covered in detail in chapter 3.

Chapter 3. Methods used for implementing a tether-aware wavefront planner

In the process of designing a feedback motion planner for a tethered mobile robot, many problems have been faced, necessitating the design of some Algorithms to solve them. These algorithms were carefully devised with a sense of reliability and reusability throughout the procedure, with the goal of achieving the best results with the minimum of complexity, and hence resulting in the most concise code.

Something to note, is that these Algorithms were designed for 2-dimensional discrete environments, however extension to continuous or higher dimensional spaces may be possible, within the computational limits and the desired performance measures.

3.1 Setting the 2-D grid environment

The first step needed before the actual motion planning is the discretization of the environment, this is assuming that a set of polygonal obstacles are given initially.

The workspace W is discretized into a 2-D grid of "n" cells depending on the width and height of the environment, as well as the grid resolution "D"; This latter is an environment sensitive variable that is chosen as to accommodate the obstacle sizes, in order to avoid (if possible) the elimination of feasible paths. Each cell of our discretized environment is 8-connected, which entails that if all surrounding cells are not detected as obstacles, we can move North, South, West, East, North-West, North-East, South-West and South-East. In addition, each cell is associated with 4 n-sized arrays which are *FlowValueFromGoal* (used to create the advancement VPF), *FlowValueFromAnchor* (used to create the backtracking VPF), *TetherValue* and *BarrierValue*. To ease the storing of the data, an indexing is needed to transform x and y coordinates to a 1-Dimensional array, This mapping is one-to-one and hence can be represented by the following functions:

$$index = integer\left(\frac{displayWidth}{CellWidth}\right) \cdot integer\left(\left(\frac{y}{CellWidth}\right) - 1\right) + integer\left(\frac{x}{CellWidth}\right)$$ and

$$x = \left(i \% \left(\frac{displayWidth}{CellWidth}\right) \cdot CellWidth\right)$$ and $$y = integer\left(\frac{i}{\frac{displayWidth}{CellWidth} + 1}\right) \cdot CellWidth$$

3.1.1 Constructing the C-space obstacles

After Setting up the grid, the next logical step is to construct the C- space obstacles. For simplicity, a planar convex robot ***A*** capable of translation in both x and y directions is chosen and the workspace obstacles ***O*** are set to be convex polygons. The process contrives of picking a reference point on the robot (which in most cases can be the center of mass) and swiping this latter around each workspace obstacle while ensuring that the reference point and the obstacle edges remain in contact. The resulting set of points are used to construct the convex hull using the gift wrapping algorithm discussed in chapter 1. The resulting polygon is the equivalent C-space obstacle, the collection of these obstacles will form our C_{obs} space.

Since a discrete implementation is used to represent our space, an appropriate continuation is to discretize the C-space obstacles as well, while distinguishing them from the free C-space. A simple way to achieve that, is by assigning a value of -1 to cells that intersect with obstacles and 0 to the ones that don't. A zero valued array called flowValue of size 1*n is used to store this info about our space. The values of this array are then inherited by FlowValueFromGoal and FlowValueFromAnchor which are responsible for advancement and retraction respectively.

So how can obstacle Cells be found?

Given a polygonal C-space obstacle whose every edge is defined by a starting point pi and an ending point p_{i+1}, K evenly spaced points occurring in the straight line between p_i and p_{i+1} are calculated, and eventually their equivalent cell coordinates are determined which are then stored in a list. After removing duplicate cells from this list, all remaining

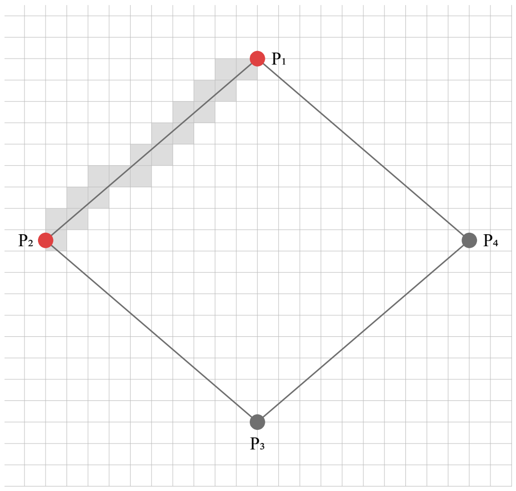


Figure 3.1: Cell Detection Algorithm demonstration

cells are assigned a flowValue of -1 and so are the cells inside them. The cell detection process is explained by the algorithm below, where K is 51.

Cell Detection Algorithm

```
1   Initialize crossingCells as empty
2   For i in range 0 to 51
3       u ← i / 50
4       x = x1 * u + x2 * (1 - u)
5       y = y1 * u + y2 * (1 - u)
6       point =(int(x / 20) * 20, int(y / 20) * 20)
7       Append point to crossingCells
8   End for
9   cellsWithNoDuplicates = [ ]
10  For cell in crossingCells
11      if cell is not in cellsWithNoDuplicates then
12          Append cell to cellsWithNoDuplicates list
13  End for
14  Return cellsWithNoDuplicates
```

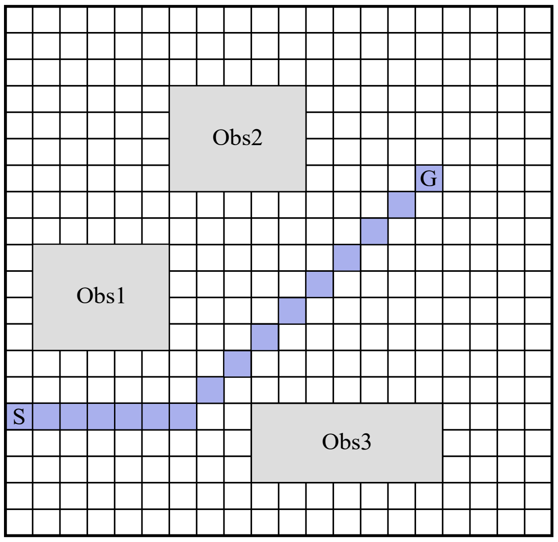
3.2 FindSpace Algorithm

The algorithm that'll be discussed here, is very essential to the retraction of the robot, as it is the one used for finding the explored spaces and the retraction space. A general understanding of this technique and the problems that it tries to solve will be presented first; specific use cases will be discussed later.

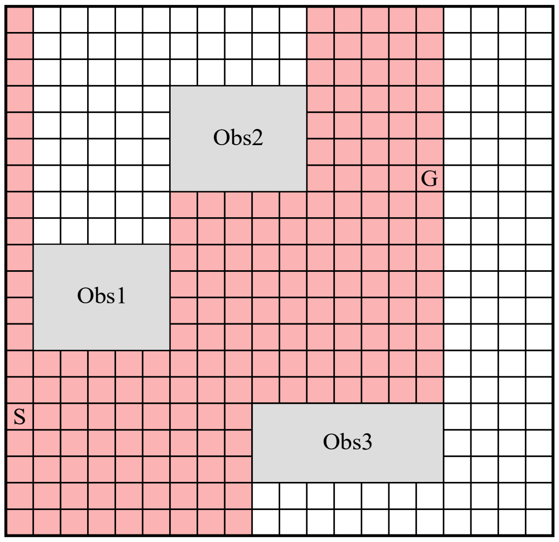
3.2.1 Defining the homotopy space

A homotopy space is the set of all paths that can be deformed into each other without cutting or pasting. These paths are then said to be homotopic. A mathematical description of this space S can be as the union of all the homotopic paths P_i or $S = \bigcup_{i=1}^{n} P_i$.

When trying to apply this concept to our discrete representation of the environment, an easy solution to find the space *S*, given a certain path P_i that is a list of grid cells, is to perform a vertical scanning for each cell of the path that stops when reaching the obstacles. A chosen example of this displayed in Figure 3.2.



The path is discretized using detectCell Al gorithm, and its cells are shown in blue.



The resulting tether space is displayed in pink. Th e vertical scanning was performed on each cell of the discrete path, and stops when reaching an obstacle.

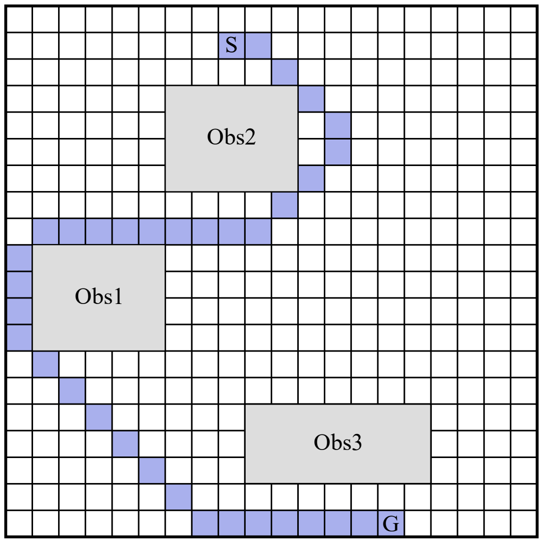
Figure 3.2: An example of using the vertical scanning to find the path Space

3.2.2 Obtaining a rigid definition of the homotopy space

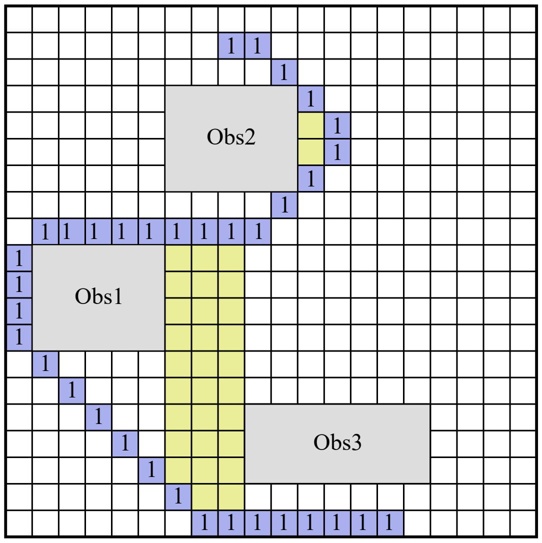
With the vertical scanning, some problems might arise that necessitate the introduction for two new variables that are *TetherValue* and *BarrierValue*, theses variables are arrays associated with each cell.

1. **TetherValue:**

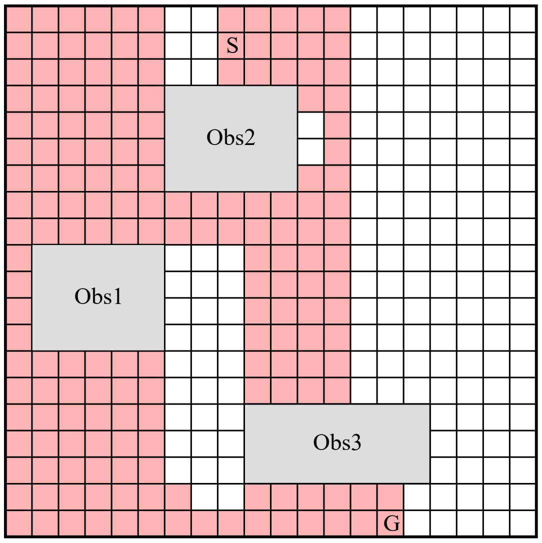
This variable is used to track tether cells or path cells, to facilitate the creation of virtual obstacles that preserve the homotopy of the space, where cells that the tether crosses have a *TetherValue* of 1. When performing the vertical scanning, if a tether cell is encountered, all the vertical cells in between will be considered as obstacles. This virtual obstacle creation helps preserve the homotopy class. An example for the use of *TetherValue* is presented in Figure 3.3.



The path is discretized using detectCell Al gorithm, and its cells are shown in blue.



The path cells are assigned a Tet herVal ue of 1, and virtual obstacles (in yellow) are created accordingly

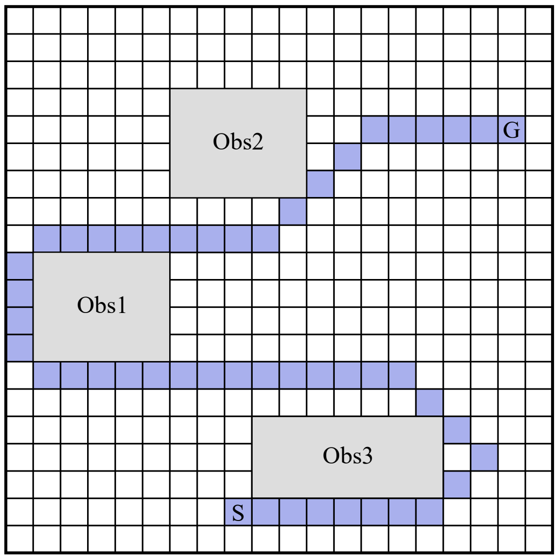


The Tet her space (in pink) is found by using the virtual scanning and stopping at the C-space and virtual obstacles

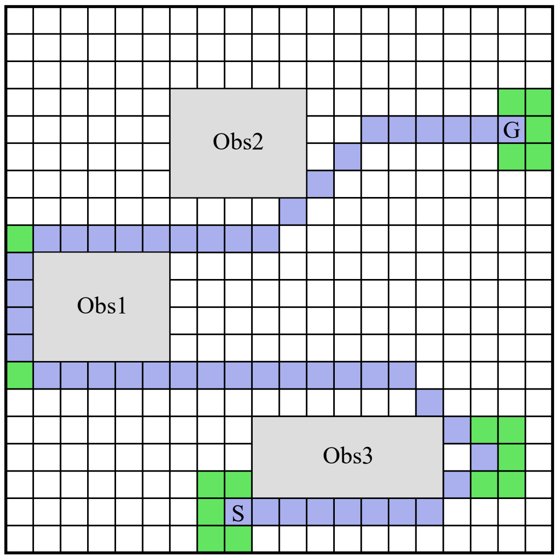
Figure 3.3: Tether space construction using TetherValue and creating artificial obstacles

2. **BarrierValue:**

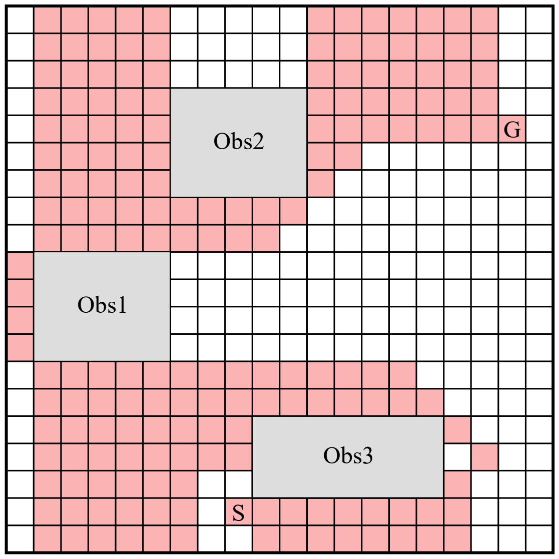
   This variable is used to prevent the space around minimas and maximas of the path to be interfered with by other cells that have smaller or larger x values. A minima here, can be defined as a protrusion in the path, that happens when a path surrounds or hugs an obstacle from the left side; A maxima is also a protrusion in the path, however it happens when a path surrounds or hugs an obstacle from the right side. The start and end point of the path are also included, depending on their relative position to the neighborhood cells. These maximas and minimas can be found whenever the path points change the direction of their x- values (from increasing to decreasing or the other way around). When these minimas and maximas are found, virtual obstacles or barriers are creating left or right to them respectively. Figure 3.4 shows the creation of a tether space for a path that surrounds Obstacle 1 from the left and Obstacle 3 from the right. In this case, barriers are built to the left and right of these protrusions and are treated as obstacles. Since the path goes right at the beginning, the start point is considered a minima and hence a barrier is constructed to its left; the goal point is a maxima here, and hence a barrier is created to its right.



The path is discretized using detectCell Al gorithm, and its cells are shown in blue.



A b arrier is created to left or right of the path minimas and maximas and is colored in green



The tether space is created and shown in pink

Figure 3.4: Tether space construction using BarrierValue

All these concepts can be summarized in the findSpace Algorithm.

FindSpace Algorithm

```
1    Input: tetherCells, flowValue, tether
2    Output: occupied space of the tether
3    Initialize verticalList as empty list
```

```
for cell in tetherCells
    temp ← [cell[0],cell[1]]
    upperCell ← [temp[0],temp[1]-20]
    upperCell2 ← [upperCell[0],upperCell[1]-20]
    While upperCell doesn't intersect with obstacles and barrier != 1
        If  tetherValue of upperCell2 != 1
                verticalList = []
                break
        temp ← upperCell
        upperCell ← upperCell2
        upperCell2 ← [upperCell[0],upperCell[1]-20]
        Append temp to verticalList
    End while
    // checking lower cells
    temp ← [cell[0],cell[1]]
    lowerCell ← [temp[0],temp[1]+20]
    lowerCell2 ← [lowerCell [0], lowerCell[1]+20]
    While lowerCell doesn't intersect with obstacles and barrier != 1
        If  tetherValue of lowerCell2 != 1
                verticalList = []
                break
        temp ← lowerCell
        lowerCell ← lowerCell2
        lowerCell2← [lowerCell[0], lowerCell[1]+20]
        Append temp to verticalList
    End while
Return verticalList
```

3.3 TightenPath algorithm

When given the initial configuration of the tether, the cable might be loose or not as tight enough around the obstacles; this loose cable when used with the shortening algorithm *CurveShorten,* might give a euclidean distance that is much larger than the actual distance, causing in many cases the elimination of some feasible paths and labeling them as not attainable by the limits of the tether length when they actually are. To prevent such problem, a suggested algorithm denoted *TightenPath* is used to tighten and remove unnecessary protrusions from the discrete version of the path. Figure 3.5 shows the tighten path algorithm in action, where the path in blue is tightened in a two step operation; these steps are vertical tightening and horizontal tightening. In vertical tightening as shown in (b), the function finds the minima points on the x values and checks if the there can be a non-intersecting line between the previous point and the next point of these minimas; if so the minima points will be shifted to be on the same level x-level as the previous and next points. This process is done continuously on each minima until an intersection happens. In horizontal tightening, the same process happens but with the y values in mind.

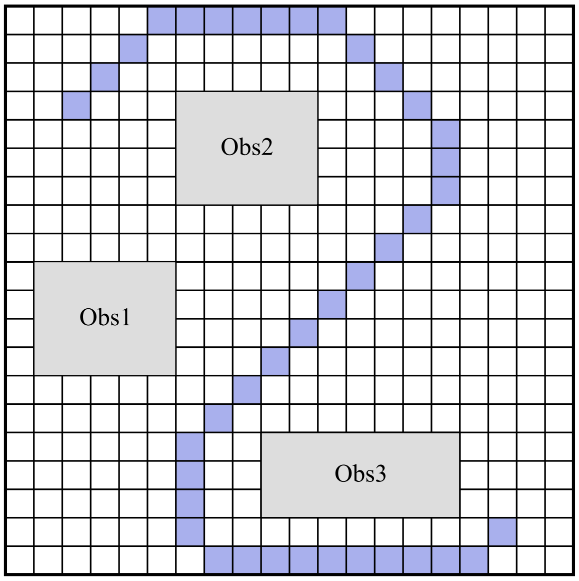
TightenPath's functionality in the discrete feedback motion planning is not limited to only tightening the path, as it can help remove the cases of tether crossing without being wrapped around an obstacle. *TightenPath* will always produce a path that has less or equal number of cells to the original, while keeping the same homotopy class.

In the discrete implementation, this function will be used with *CurveShorten* to produce improved distance calculation results. The algorithm of *TightenPath* can be understood by reading the pseudo code provided below.

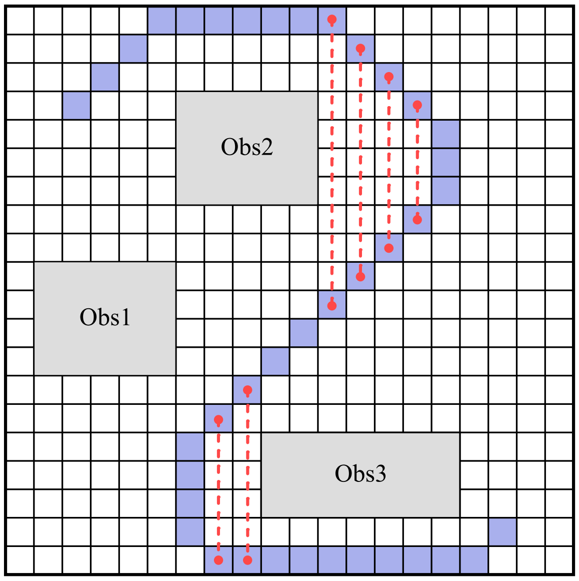
TightenPath Algorithm

```
Initialize start index and end index as empty and set counter to zero
//start the vertical scanning by varying x
For x in display width
    For point in path
        if point[0] = x then
            if end = [ ] then
```

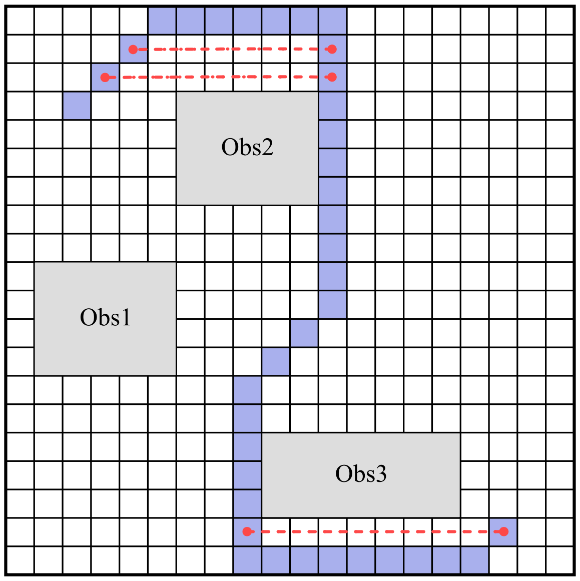
```
                start ← counter
            end ← counter + 1
        **else if** start **and** end **then**
            **if** there's no intersection with obstacles between previous and next point
                **For** j in range of start to end
                    path[j] ← path[start-1]
End for
//start the horizontal scanning by varying y
**For** y **in** display height
    **For** point **in** path
        **if** point[1] = y **then**
            **if** end = [ ] **then**
                start ← counter
            end ← counter + 1
        **else if** start **and** end **then**
            **if** there's no intersection with obstacles between previous point and next
                **For** j in range of start to end
                    path[j] ← path[start-1]
End for
PathWithNoDuplicates = [ ]
**For** point **in** path
    **if** point is not in PathWithNoDuplicates **then**
        Append point to PathWithNoDuplicates list
End for
**Return** PathWithNoDuplicates
```



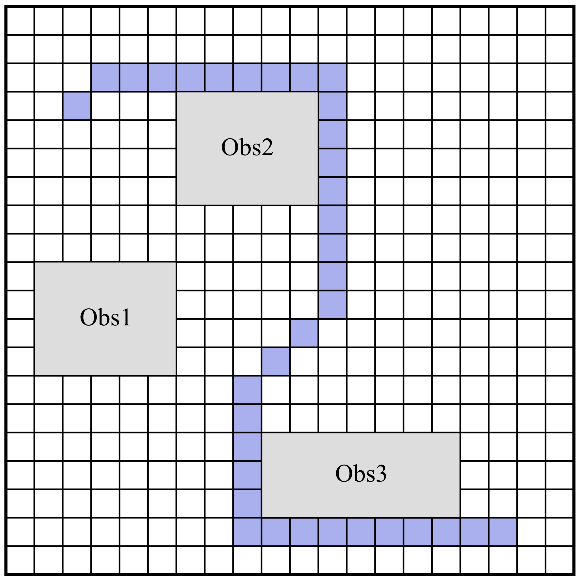
A l oose path that occupies 39 cells, is used as a testing example for the Tig htenPath algorithm



In Tig htenPath,Ver tical scanning is performed first, which will remove saddles on the right of Obstacle 2 and on the left of Obstacle 3.



The path has been tightened vertically, and horizontal scanning is now performed.



The tightened path is generated and it occupies 35 cells

Figure 3.5: Demonstration of the TightenPath algorithm and its effect on the output of CurveShorten

3.4 Curve Shorten Algorithm

This function returns the length of the shortened path while ensuring no intersection with the obstacles and keeping the same homotopy class [11]. Figure 3.6 shows the effect of using *CurveShorten,* where we continuously check if there's obstacle intersection with line starting from q_i to q_{i+1}, q_{i+2},…, q_j, q_{j+1},...etc. If the line q_iq_{j+1} intersects with an obstacle, the new euclidean distance becomes $d = d + \|q_iq_j\|$ and q_j becomes the new start node for the intersection check. This process is iterated until, the final point of the path is reached, where d gives the shortened path distance.

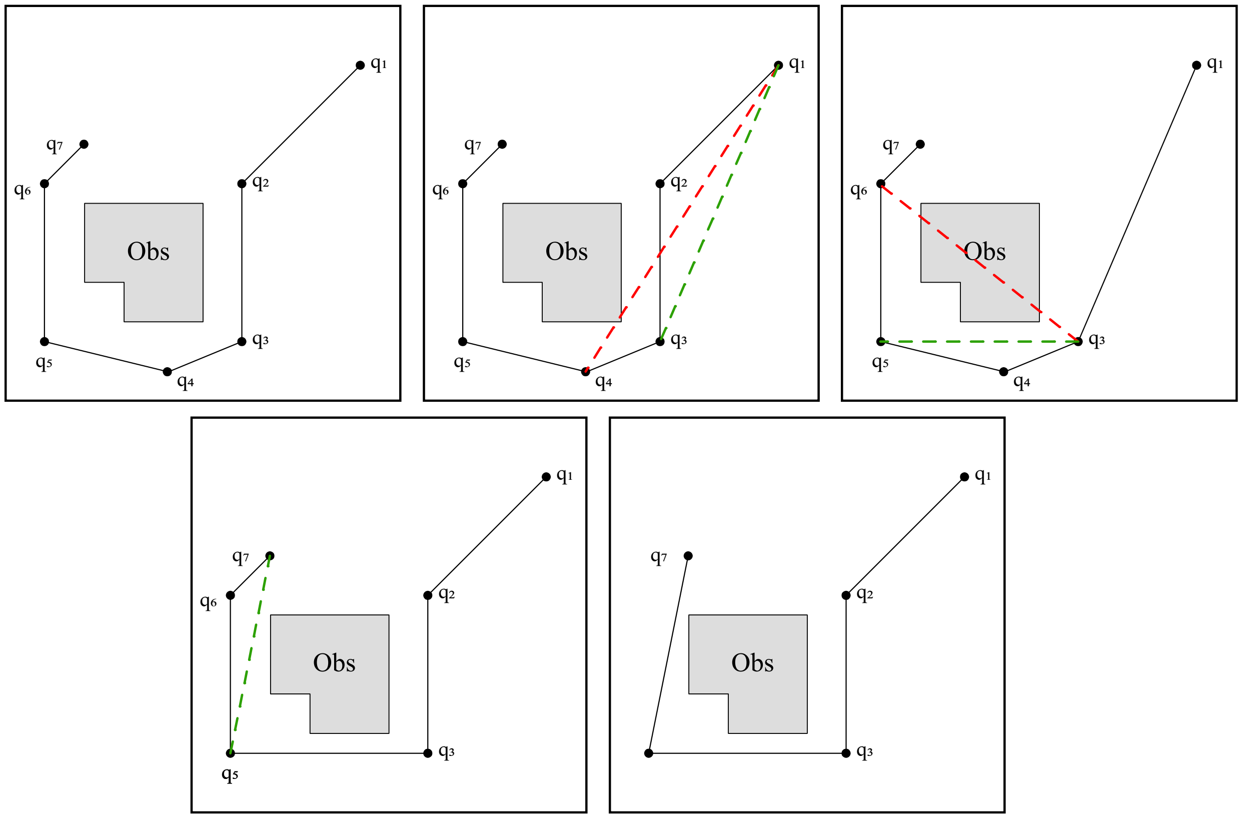


Figure 3.6: A demonstration of CurveShorten algorithm on a shortened discrete path

Below is a pseudo-code [11] for the *CurveShorten* function:

CurveShorten Algorithm

```
Initialize $l$, i and j as 0
While j ≤ Length of the path -1
    If j<n and no obstacle intersection with the line between path[i] and path[j+1]
        j = j+1
    Else
        $l = l + \|$path[i] to path[j]$\|$
        i = j
    End if
End While
Return $l$
```

Chapter 4. Simulation and Results

After introducing the suggested methodology of feedback motion planning while under tether constraints, an application of the proposed concept is then portrayed in this chapter by the use of the game design library in Python known as PyGame.

We start this chapter by providing a brief introduction to the PyGame library in Python, and its different use cases and capabilities. This is then followed a presentation of the abstract structure of the code and some specific design choices that were made. The core of this chapter follows next, where a batch of results for different configurations and environments were presented, and a commentary and analysis on these obtained results is then provided. This accompanied by some conclusions on the success of the used methods.

4.1 An overview of PyGame

Pygame is a set of Python modules designed for writing video games that adds functionality on top of the excellent SDL library. This allows the user to create fully featured games, multimedia programs and other non-gaming related simulation projects in the python language. Pygame is highly portable and runs on nearly every platform and operating system [13].

Figure 4.1: Pygame Library Logo

Some other reasons that make PyGame the perfect platform for testing our code include:

Efficient and concise code. The core of PyGame is kept simple, and extra things like GUI libraries, and effects are developed separately outside of Pygame, resulting in fewer lines of code.

Uses optimized C and assembly code for core functions. C code is often 10-20 times faster than python code, and assembly code can easily be 100x or more times faster than python code.

4.2 Structure of the code

In the programming approach, the motion planning process is compartmentalized by creating three modes of operation where each corresponds to a certain phase of our problem-solving process.

4.2.1 Obstacle creation mode

This mode is executed first, once and before all other modes. Here, a grid is set up and the obstacles are drawn from a pre-determined list of polygons that exist within the bounds of our environment. These obstacles are then used to construct our C-space obstacles as demonstrated in chapter 2. A discretized representation of our C-space is stored in the variable flowValue where 0 represents a free space cell and -1 represents an obstacle space cell in the configuration space.

4.2.2 Tether mode

This mode runs after draw mode execution is finished. Here, an option is presented to the user to either enter the coordinates of all the tether nodes in order or to use the mouse to create the tether nodes by consecutively clicking in the free space. A function is implemented to notify the user in real time if an edge between two consecutive nodes crosses an obstacle. By the end of this mode, the start, anchor and goal point as well as the tether would all be defined.

4.2.3 Path planning mode

This is where the motion planning happens. The logic discussed in the navigation process of chapter 2 is implemented here and the robot movements as well as the tether and shortened path are animated to provide a visual representation of the methodology discussed in this thesis.

4.3 Conducting the simulation

Since Pygame is used in simulation, an appropriate unit of measurement for distance would be pixels. An Environment size of 720 by 1280 pixels is chosen, accompanied with grid cells of 20 by 20 px which will result in 2304 cells. The boundary cells will be given flow values of -1. This grid resolution was chosen to accommodate our obstacles and prevent the omission of any feasible path. A set of pre-defined polygonal obstacles are then provided to be discretized and used in the construction of C-space obstacles. The initialization of the anchor, start, and goal point as well as the tether configuration is left to the user. Many trials were conducted using different environments to test the reliability of the suggested approach for tethered robots' motion planning.

A sample of these trials were chosen as to cover some interesting cases as well as demonstrate the viability of our approach; these trials are to be encountered in the following section of this chapter.

4.4 Simulation Results and discussion

For the sake of diversifying the cases for tethered mobile robots' feedback motion planning, different environments were chosen with different initial configurations. And some of the obtained results are shown below. Each time the distance is measured it's presented along with a statement on whether or not the goal is reachable from the current point.

4.4.1 Analysis of the first example

The first encountered simulation result is in the Figure 4.2, the anchor is presented as a black circle and the goal point is a red circle, while the black lines constitute the tether. Initially, the program outputs "The goal is reachable", which signifies that a reachability test has been assessed successfully, and that the goal is indeed reachable (from the anchor that is). The program then outputs "The shortest distance l = 522.84 px", followed by "A solution is found and the goal is reachable from the current point". It is worth mentioning that no retraction occurred during this first example. From these results, it's obvious that the navigation loop was initiated and that the shortest distance of the first discovered path belonging to the first homotopy class was found to be less than the maximum tether length L=700, hence, the robot moves towards the goal location following the advancement VPF; this advancement field is nothing but FlowValueFromGoal that was found by initiating a wave propagation from the goal when performing the first universal reachability test. This first example sheds a light on:

- the usefulness of using the wave propagation coupled with the distance calculation algorithm which are TightenPath and CurveShorten in determining the feasibility of our motion plan.
- The importance of the reachability test that can save a lot of the computational hassles.

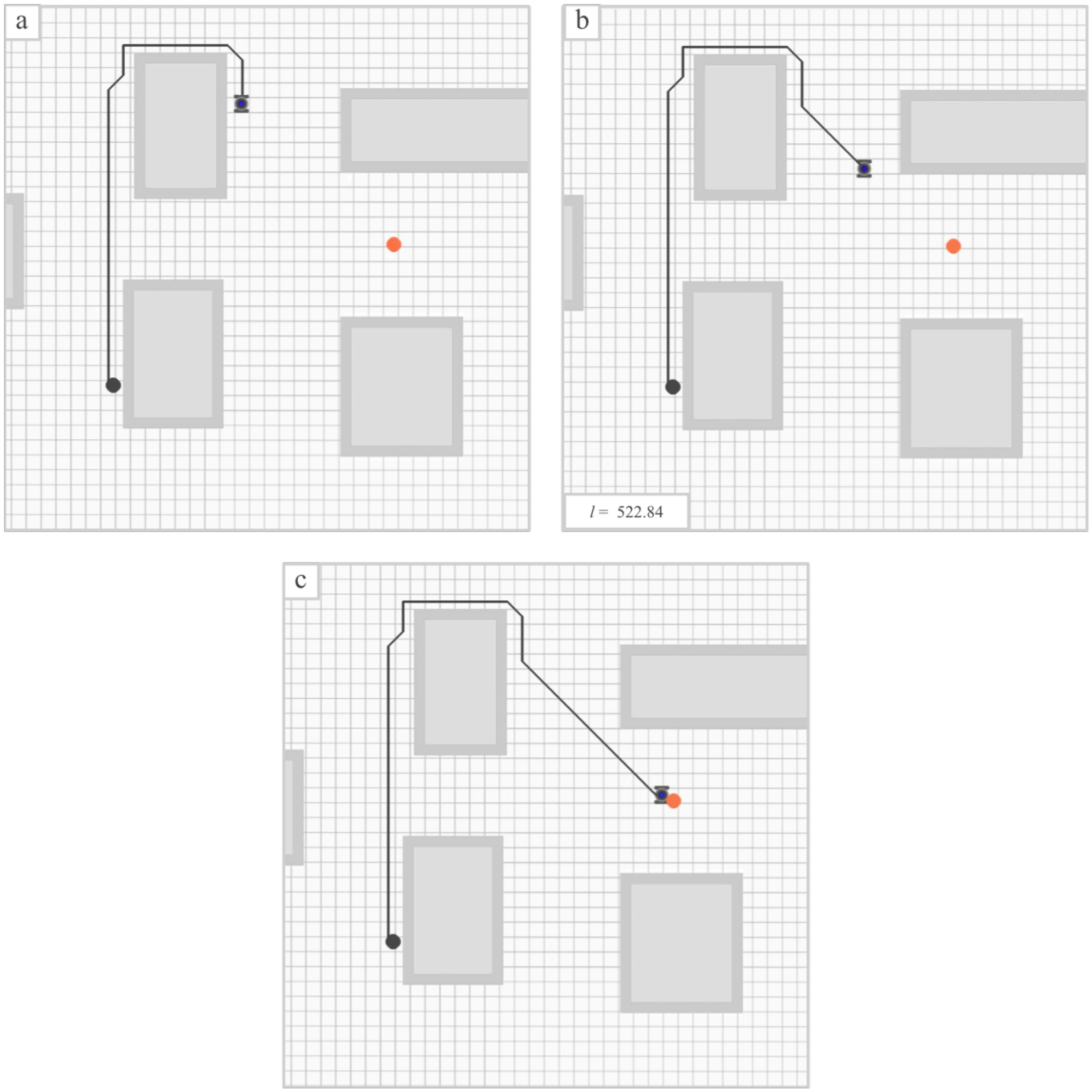


Figure 4.2: The first example where the robot advances towards the goal immediately

4.4.2 Analysis of the Second example

The second example is shown in figure 4.3. Here again the program outputs “The goal is reachable”, indicating the existence of a possible motion plan from the anchor to the goal. The first iteration of the navigation results in a distance of l = 861.29 px which is much larger than L, and the following message of “A solution couldn’t be found from the current point”. The robot subsequently retracts by multiple cells, until the configuration is that of the 3rd illustration on the figure, where it indicates that a new tether space was found. Going from the current point of the robot to the goal point was not possible since the distance of the shortest path preserving homotopy was larger than L. Thus, the robot retracts as shown in B and keeps retracting until it finds the second homotopy space which satisfies the distance with condition l = 521.36. This results in an advancement and eventually an arrival at the goal as shown in C and D. These results show:

- The success of findSpace algorithm in identifying the retraction space and the explored spaces.
- An emphasis of the usefulness of the shortening algorithms.

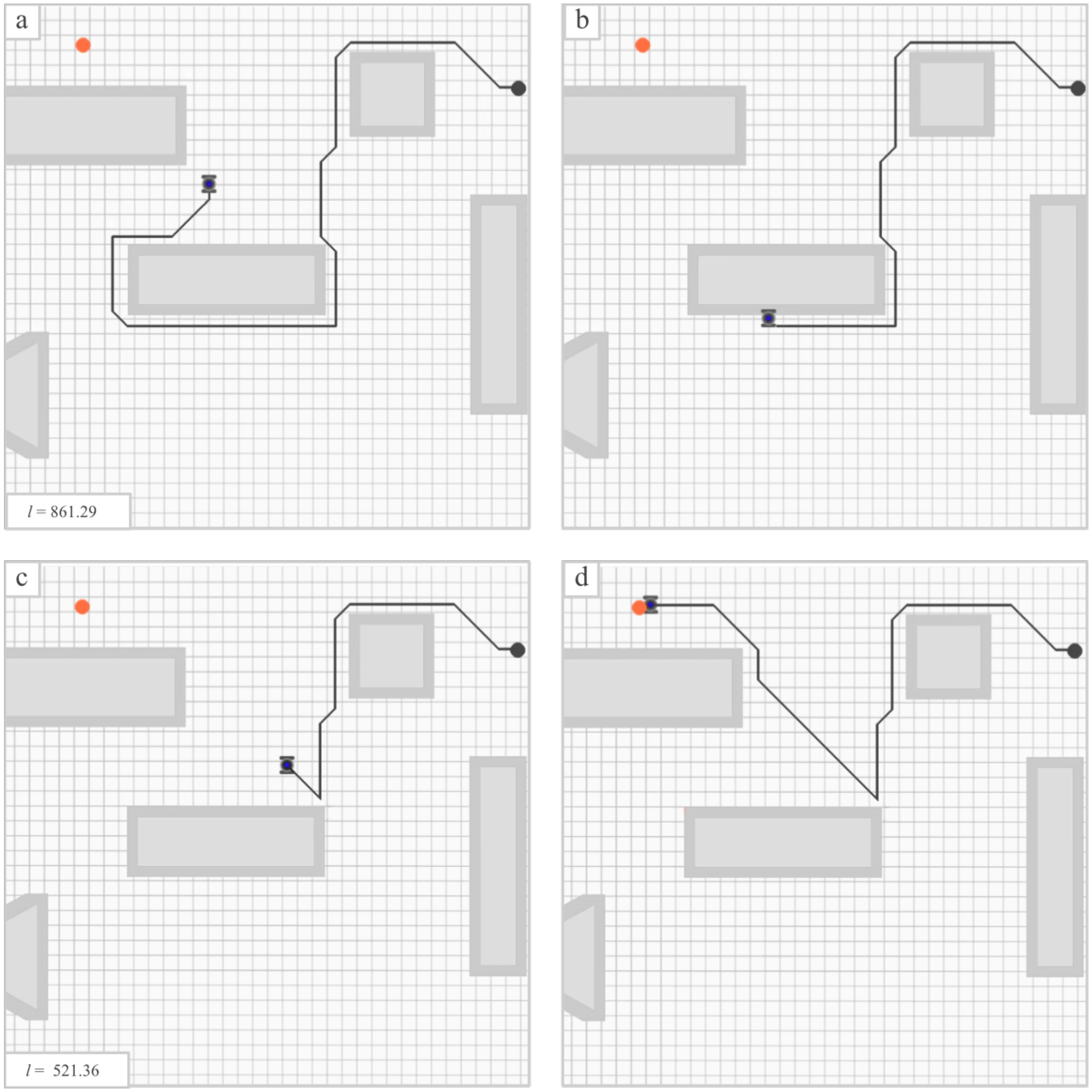


Figure 4.3: The second example where the robot retracts first then heads to the goal

4.4.3 Analysis of the third example

The third example is displayed in figure 4.4 and figure 4.5, the first figure shows the retraction step where the robot retracts twice discovering three spaces, the first space has a shortened distance of l = 857.33 px with is more than 700 px means it's not a feasible path. The second space is found after the first retraction, this space has a shortened distance of l = 749.71 px which is close to L, but not enough. Hence, retraction is required once again, which leads to the configuration shown in c, with a distance of l = 552.77 px. A solution is then found and advancement is to be performed. In the figure 3.5, the robot seems to follow the advancement VPF and heads towards the goal.

This example reassures once again, the viability of concepts and the general methodology presented in chapter 2. Starting from the definition of the tether space that helped ensure a safe retraction that respects the homotopy, to the discovery and storage of all the different spaces encountered while retracting to the efficient distance calculation algorithms and approach.

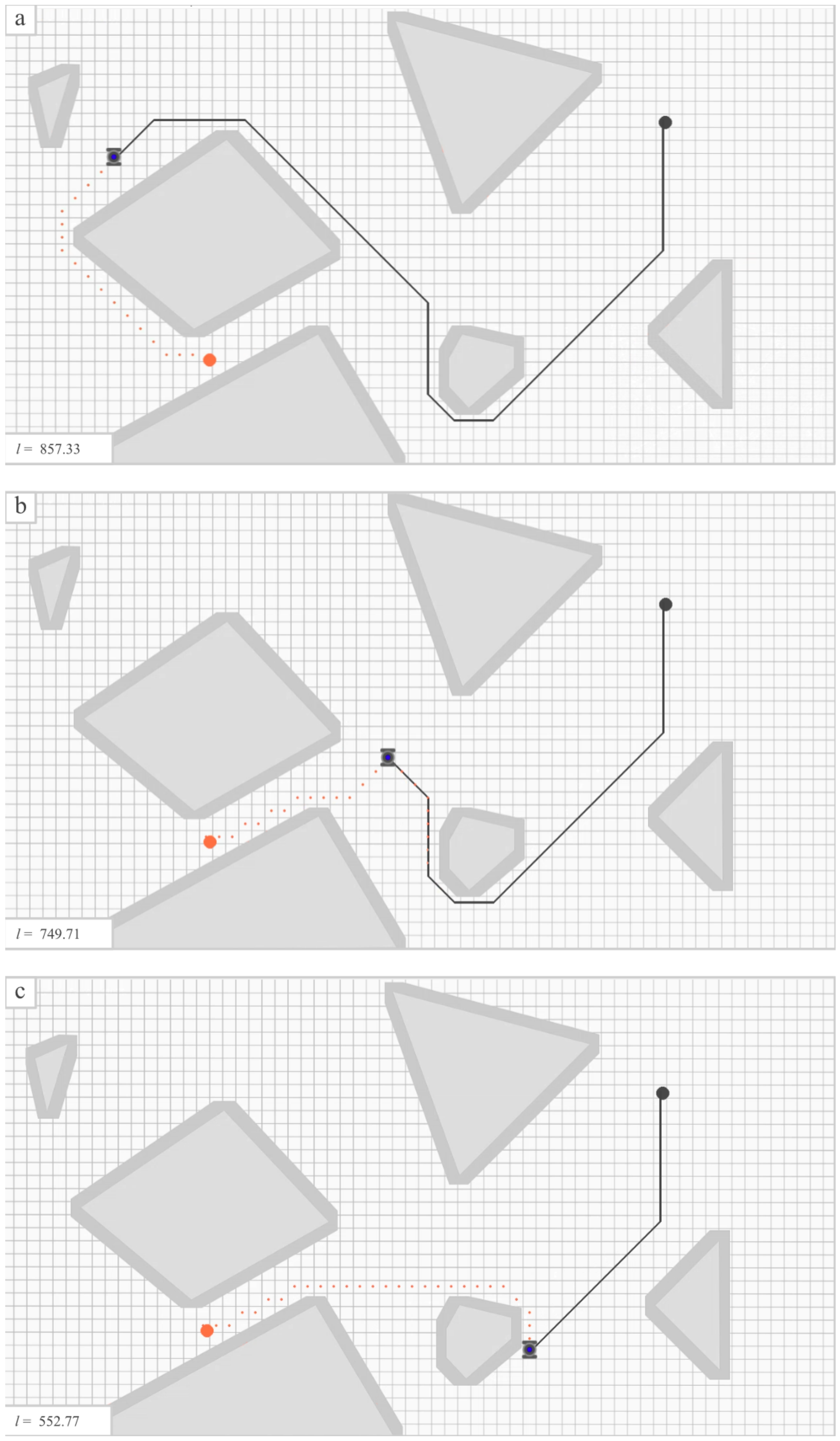


Figure 4.4: The first Part of the third example where the robot retracts and checks the distance twice

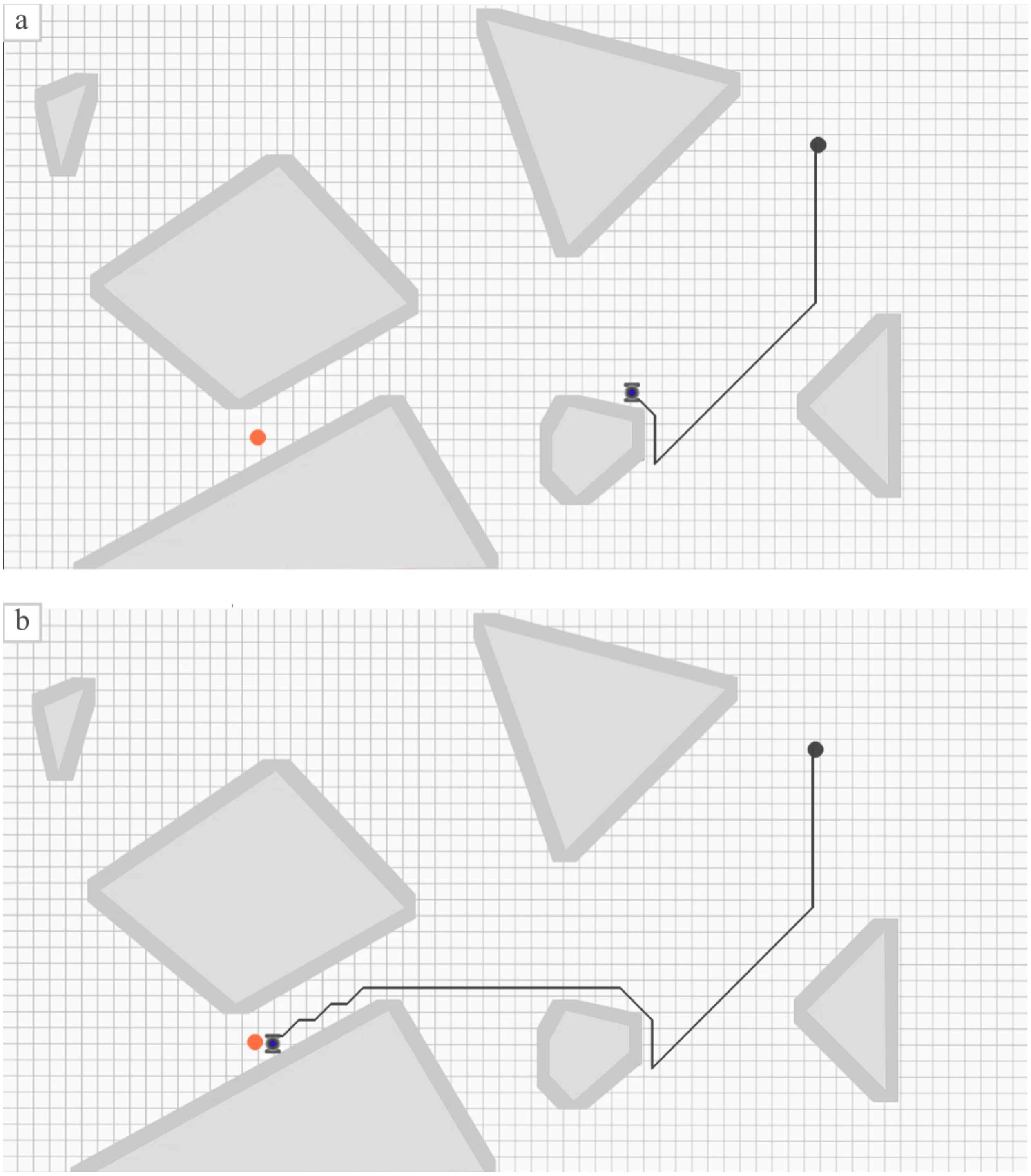


Figure 4.5: part 2 of the third example where the robot heads to the goal

Conclusion

In this work, we demonstrated a new approach for the problem of motion planning for tethered mobile robots. This approach consists of using a discrete representation of the environment that will be a basis for our motion planning. This research was conducted under the condition of absence of tether crossing when wrapping around an obstacle. The robot was assumed to be connected to a fixed base by a fixed-length cable. The objective is to find the shortest path from the initial robot-cable configuration to a final goal position. Two constraints imposed by the cable on the motions of a tethered robot were considered: limited radius of movement and the topological constraints. To resolve the mentioned constraints, a grid was overlaid in our environment and the wavefront planner was employed in it. After testing the reachability of the goal location via some homotopy class, the planner either designs a path to the goal using the flow field, or retracts back to search for a path in a different homotopy class in case the tether length is not enough. We illustrated the algorithm using simulations in multiple environments filled with obstacles (displayed in chapter 3), which proves the success of the general strategy that was suggested in chapter 2.

The results of the implementation of our suggested approach, were very promising as examples have shown the ability of the robot to retract to the correct homotopy class to find solution, further consolidating the usefulness of the used techniques and methods.

When reflecting on it, this general strategy can be adapted to fit other feedback motion planning methods, and some of the displayed techniques can be used in other motion planning areas. An important thing to note here as well, is the possibility of expanding this approach to include the cases where tether crossing happens when wrapping around obstacles, by creating an augmented 3d representation of the space whenever a wrapping happens.

Bibliography

[1] J.H. Davenport, “A ‘Piano-Movers’ Problem,” *SIGSAM Bull*, pp. 15–17, 1986.

[2] Steven M. LaValle, *Planning Algorithms*. New York (NY): Cambridge University Press, 2014.

[3] F. C. P. Kevin M. Lynch, *Modern Robotics: Mechanics, Planning, and Control*. 2017.

[4] F. G.-B. Giuseppe Carbone, *Motion and Operation Planning of Robotic Systems: Background and Practical Approaches*. 2015.

[5] Mark de Berg, Marc van Kreveld, and Mark Overmars, *Computational Geometry: Algorithms and Applications*. 1997.

[6] “Gift wrapping - Wikipedia.” [Online]. Available: https://en.wikipedia.org/wiki/Gift_wrapping

[7] “Chan’s algorithm - Wikipedia.” [Online]. Available: https://en.wikipedia.org/wiki/Chan%27s_algorithm

[8] Stephen Cass, “DARPA Unveils Atlas DRC Robot,” Jul. 2013, [Online]. Available: tp://spectrum.ieee.org/automaton/robotics/humanoids/darpa-unveils-atlas-drc-robot.

[9] R.J. Wood P. Chirarattananon and K.Y. Ma, “Adaptive control for takeoff, hovering, and landing of a robotic fly,” *Intell. Robots Syst. IROS*, pp. 3808–3815, Nov. 2013.

[10] S. Bhattacharya, M. Likhachev, and V. Kumar, “Topological Constraints in Search-based Robot Path Planning.”

[11] Soonkyum Kim, Subhrajit Bhattacharya, and Vijay Kumar, “Path Planning for a Tethered Mobile Robot,” pp. 4–5.

[12] É. T. Jon Kleinberg, *Algorithm Design*. 2005.

[13] “Pygame wiki.” [Online]. Available: https://www.pygame.org/wiki/about