Path Planning for Robotics: Localization, Mapping and Path Planning of a Home Service Robot in Simulation Environment

Smruti Panigrahi, Ph.D.
Udacity Robotics Nandegree

***Abstract*—In this paper, we explore various algorithms related to autonomous path planning. Various classic non-probabilistic path planning and probabilistic path planning algorithms are discussed in details. These algorithm's complexity, completeness and optimality are used to access the algorithm's robustness and efficiency. The role of the deep reinforcement learning (Deep RL) and latest robust algorithms that can be used to robustly perform path planning is discussed. An environment is created using Gazebo for the robot perform autonomous simultaneous planning, localization and mapping (SPLAM). The robot uses sensor information received from a camera or LiDAR, and robot odometry data to autonomously navigate itself in the environment. A wall follower algorithm is used in a closed environment and ROS packages navigation stack, AMCL, and gmapping are used for the robot's autonomous SPLAM. Various parameters used in order to tune the model for better accuracy and robustness is discussed.**

Keywords—Path Planning, Deep RL, Dijkstra, A*, SPLAM, gmapping, AMCL, Navigation Stack, Gazebo Simulation, ROS.

I. Introduction

Sitting in your home or office, some environment-specific examples come to mind right away - vacuum robots plan their paths around a house to ensure that every square inch of space gets cleaned. Self-driving cars are starting to appear around the world. These vehicles can accept a destination as an input from a human and plan an efficient path that avoids collisions and obeys all traffic regulations.

More peculiar applications of path planning in robotics include assistive robotics. Whether working with the disabled or elderly, robots are starting to appear in care homes and hospitals to assist humans with their everyday tasks. Such robots must be mindful of their surroundings when planning paths - some obstacles stay put over time, such as walls and large pieces of furniture, while others may move around from day to day. Path planning in dynamic environments is undoubtedly more difficult.

Another robotic application of path planning is the planning of paths by exploratory rovers, such as Curiosity on Mars. The rover must safely navigate the surface of Mars (which is between 55 and 400 million kilometers away!). Accurate problem-free planning that avoids risks is incredibly important.

As a roboticist or a robotics software engineer, building our own robots is a very valuable skill and offers a lot of experience in solving problems in the domain. Often, there are limitations around building our own robot for a specific task, especially related to available resources or costs. That's where simulation environments are quite beneficial. Not only there is freedom over what kind of robot one can build, but also get to experiment and test different algorithms and scenarios with relative ease and at a faster pace. For example, one can have a simulated drone to take in camera data for obstacle avoidance in a simulated city, without worrying about it crashing into a building!

Path planning is not limited to robotics applications, in fact it is widely used in several other disciplines. Computer graphics and animation use path planning to generate the motion of characters. While computational biology applies path planning to the folding of protein chains. With many different applications, there are naturally many different approaches. In the next few lessons you will gain the knowledge required to implement several different path planning algorithms.

II. Background

Just like localization and SLAM, there is not one correct way of accomplishing the task path planning for robotics. The path planning algorithm takes in inputs as provided environment geometry, the robot's geometry and the robot's start and goal poses, and uses this information to produce a path from start to the goal position as shown in Fig. 1 below.

Fig. 1: The inputs and outputs of a path planning algorithm.

In this paper, we will discuss the inner workings of different types of path planning algorithms [1] and evaluate the suitability of an algorithm for a particular application. We will also discuss the notion of **completeness** and **optimality** [2] in analyzing the applicability of an algorithm to a task. Other aspects one must consider while performing path planning are the time complexity, space complexity, and generality.

The **time complexity** of an algorithm assesses how long it takes an algorithm to generate a path, usually with respect to the number of nodes or dimensions present. It can also refer to the trade-off between quality of an algorithm (ex. completeness) vs its computation time.

The **space complexity** of an algorithm assesses how much memory is required to execute the search. Some algorithms must keep significant amounts of information in memory throughout their run-time, while others can get away with very little.

The **generality** of an algorithm considers the type of problems that the algorithm can solve - is it limited to very specific types of problems, or will the algorithm perform well in a broad range of problems?

A. Path Planning Example

Let's look at an application of path planning to better understand the problem at hand and to learn some relevant terminologies. An exploratory robot may find itself dropped off at a starting position and need to traverse the land, water or air to get to its goal position. In between its start and goal

locations, there will inevitably be some obstacles. Let's assume that our rover is on land performing a recovery operation after a natural disaster. The rover is dropped off at one position and needs to get to another position, to evaluate whether it is safe for humans to follow the path.

From the map, the robot knows that there are rubbles present along its path. Using GPS data and aerial photographs of the environment, the rover must plan a path through the rubbles to get to its destination. One option that the rover has is to follow the shortest path, the straight line between the start and the goal locations. However, due to large amounts of rubble present in its path, the rover might have to slow down considerably to safely navigate this path. If time is of the essence, this not likely the ideal path to take. Taking the direct route may not even be possible if there are large obstacles present in its way and the rover is unable to overcome. The rover will need a solution to this problem instead of just stopping dead in its track. One algorithm that the robot can apply would have the rover traverse any encountered obstacles clockwise until it reaches its intended path once again. This algorithm is often referred to as the "bug algorithm".

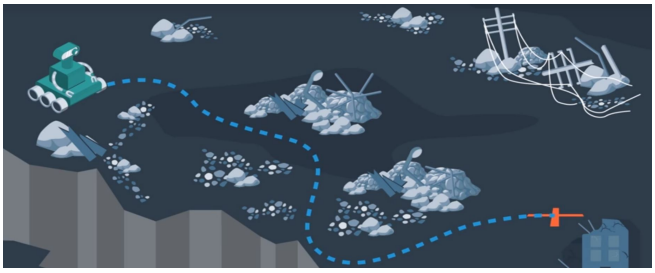

Fig. 2: A robot planning a path from its current location to its goal position.

An alternate route al together would be to go around as much rubble as possible. To accomplish this, the path planning algorithm would need a way to evaluate how long it takes to traverse different types of land and to take this information into account when planning a path. The resulting path may look like the one shown in Fig. 2. Although it is longer than the direct route, the rover will get to its goal location faster, because it can move quicker on flat lands. Algorithms that are more sophisticated may take into account the risk that the rover faces. Since the rovers are expensive tools, it would be wise to avoid unnecessary dangers during their operations. The robot's path planning algorithm may have it avoid unstable terrain, or moving too closely to a cliff.

Two methods of evaluating algorithms are to access whether they are **complete** and whether they are **optimal**. An algorithm is complete if it is able to find a path between a start and the goal locations when one exists. A complete is able to solve all solvable problems and return no solutions found to unsolvable problems. An algorithm is optimal if it is able to find the best solution. Best may mean different things. In the simplest case, best refers to the shortest path. But, as we saw in the rover example above, best may mean quickest or the path that minimizes the risk the most or a combination of factors.

B. Approaches to Path Planning

There are three different approaches to path planning. The first, called discrete (or combinatorial) path planning is the most straightforward of the three approaches. The other two approaches, called sample-based path planning and probabilistic path planning, will build on the foundation of discrete planning to develop more widely applicable path planning solutions.

Discrete (or **combinatorial**) **path planning** looks to explicitly discretize the robot's workspace into a connected graph, and apply a graph-search algorithm to calculate the best path. This procedure is very precise (in fact, the precision can be adjusted explicitly by changing how fine you choose to discretize the space) and very thorough, as it discretizes the complete workspace. As a result, discrete planning can be very computationally expensive - possibly prohibitively so for large path planning problems. The Fig. 3 (a) displays one possible implementation of discrete path planning applied to a two-dimensional workspace. Discrete path planning is elegant in its preciseness, but is best suited for low-dimensional problems. For high-dimensional problems, sample-based path planning is a more appropriate approach.

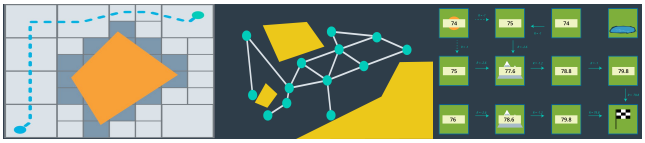

Fig. 3: (a) Discrete path planning. (b) Sample-based path planning. (c) Probabilistic path planning.

Sample-based path planning [3] probes the workspace to incrementally construct a graph. Instead of discretizing every segment of the workspace, sample-based planning takes a number of samples and uses them to build a discrete representation of the workspace. The resultant graph is not as precise as one created using discrete planning, but it is much quicker to construct because of the relatively small number of samples used. A path generated using sample-based planning may not be the best path, but in certain applications - it's better to generate a feasible path quickly than to wait hours or even days to generate the optimal path. The image in Fig. 3(b) displays a graph representation of a 2-dimensional workspace created using sample-based planning.

While the first two approaches looked at the path planning problem generically - with no understanding of who or what may be executing the actions - **probabilistic path planning** [4] takes into account the uncertainty of the robot's motion. While this may not provide significant benefits in some environments, it is especially helpful in highly-constrained environment or environments with sensitive or high-risk areas. The image in Fig. 3(c) displays probabilistic path planning applied to an environment containing a hazard (the lake at the top right).

III. Classic or Discrete Path Planning

Solving a classic path planning [1] problem through discrete planning, otherwise known as combinatorial planning, can be broken down into three distinct steps. The first is to develop a convenient continuous representation. This can be done by representing the problem space as the configuration space. The configuration space, also known as C space, is an alternate way of representing the problem space. The C space, takes into account the geometry of the robot and makes it easier to apply discrete search algorithms.

Next, the configuration space must be discretized into a representation that is more easily manipulated by algorithms. The discretized space is represented by a graph. Finally, a search algorithm can be applied to the graph to find the best path from the start node to the goal node (Fig. 4). For each of these steps, there are verity of method that can be applied to accomplish the desired outcomes, each with their own advantages and disadvantages.

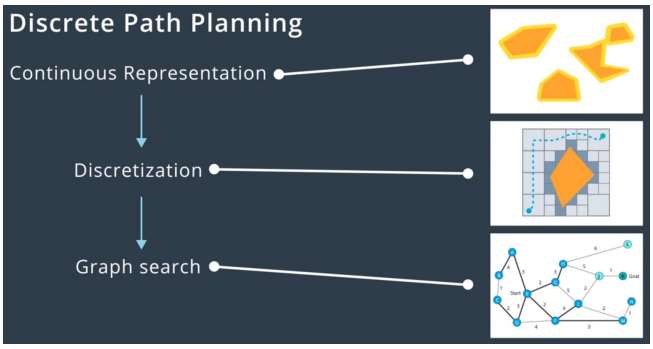

Fig. 4: Steps used in discrete path planning algorithm.

A. Continuous Representation

To account for the geometry of a robot and simplify the task of path planning, obstacles in the workspace can be inflated to create a new space called the configuration space (or C-space). With the obstacles inflated by the radius of the robot, the robot can then be treated as a point, making it easier for an algorithm to search for a path. The C-space is the set of *all* robot poses, and can be broken-down into C_{free} and C_{obs}. C_{free} represents the set of poses in the free space that do not collide with any obstacles. C_{obs} is the complement to C_{free}, representing the set of robot poses that are in collision with obstacles or walls as shown in Fig. 5.

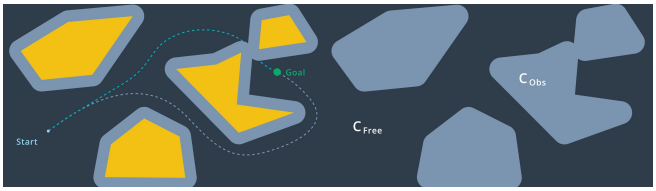

Fig. 5: (Left) The configuration space or C space where the obstacles are inflated by the robot dimension. (Right) The C space consisting of C_{free} and C_{obs}.

The Minkowski sum [5-7] is a mathematical property that can be used to compute the configuration space given an obstacle geometry and robot geometry. The intuition behind how the Minkowski sum is calculated can be understood by imagining to paint the outside of an obstacle using a paintbrush that is shaped like our robot, with the robot's origin as the tip of the paintbrush. The painted area is C_{obs}.

To create the configuration space, the Minkowski sum is calculated in such a way for every obstacle in the workspace. The Fig. 6 below shows three configuration spaces created from a single workspace with three different sized robots. As you can see, if the robot is just a dot, then the obstacles in the workspace are only inflated by a small amount to create the C-space. As the size of the robot increases, the obstacles are inflated more and more.

For convex polygons, computing the convolution is trivial and can be done in linear time - however for non-convex polygons (i.e. ones with gaps or holes present), the computation is much more expensive.

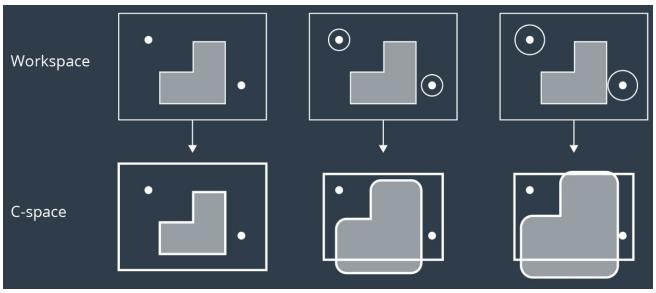

Fig. 6: Obstacle size increases in C space when the robot size increases in workspace.

The configuration space for a robot changes depending on its rotation. Allowing a robot to rotate adds a degree of freedom - so, sensibly, it complicates the configuration space as well. Luckily, this is actually very simple to handle. The dimension of the configuration space is equal to the number of degrees of freedom that the robot has.

While a 2D configuration space was able to represent the x- and y-translation of the robot, a third dimension is required to represent the rotation of the robot.

Let's look at a robot and its corresponding configuration space for two different rotations. The first will have a triangle-shaped robot at 0°, and the second at 18° as shown in Fig. 7.

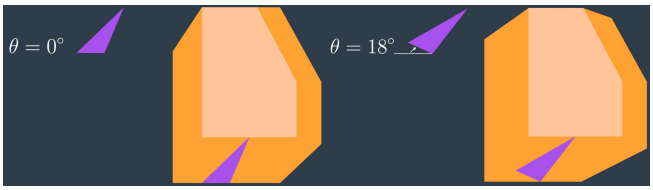

Fig. 7: Configuration space for two different rotations of a triangular robot.

A three-dimensional configuration space can be generated by stacking two-dimensional configuration spaces as layers - as seen in Fig. 8 (a).

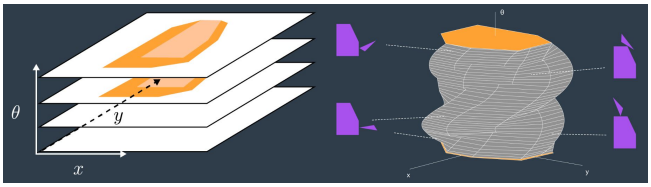

Fig. 8: (a) 3-D configuration space generated by stacking 2D C-spaces. (b) The complete 3D C-spaces generated through infinitesimally small rotations of the robot.

If we were to calculate the configuration spaces for infinitesimally small rotations of the robot, and stack them on top of each other - we would get something that looks like the image in Fig. 8 (b).

The image above displays the configuration space for a triangular robot that is able to translate in two dimensions as well as rotate about its z-axis. While this image looks complicated to construct, there are a few tricks that can be used to generate 3D configuration spaces and move about them. The video from the Freie Universität Berlin is a wonderful visualization of a 3D configuration space [8] that

displays different types of motion, and describe how certain robot motions map into the 3D configuration space.

B. Discretization

To be able to apply a search algorithm, the configuration space must be reduced to a finite size that an algorithm can traverse in a reasonable amount of time as it searches for a path from start to the goal. This reduction in size can be accomplished by discretization. Discretization is the process of breaking down a continuous entity, in this case a configuration space, into discrete segments. There are different methods that can be applied to discretize a continuous space. There are three different types of discretization; roadmap, cell decomposition, and gradient field or potential field. Each has its own advantages and disadvantages. Balancing trade-offs such as time and the level of details.

1) Roadmap Discretization

The first group of discretization approaches that you will learn is referred to by the name Roadmap. These methods represent the configuration space using a simple connected graph - similar to how a city can be represented by a metro map shown in Fig. 9.

Fig. 9: An example of a roadmap diagram.

Roadmap methods are typically implemented in two phases:

- The **construction phase** builds up a graph from a continuous representation of the space. This phase usually takes a significant amount of time and effort, but the resultant graph can be used for multiple queries with minimal modifications.
- The **query phase** evaluates the graph to find a path from a start location to a goal location. This is done with the help of a search algorithm.

In this Discretization section, we will only discuss and evaluate the construction phase of each Roadmap method. Whereas the query phase will be discussed in more detail in the Graph Search section, following Discretization.

The two roadmap methods are the Visibility Graph, and Voronoi Diagram methods.

- **Visibility Graph**

The Visibility Graph builds a roadmap by connecting the start node, all of the obstacles' vertices, and goal node to each other - except those that would result in collisions with obstacles. The Visibility Graph has its name for a reason - it connects every node to all other nodes that are 'visible' from its location.

Nodes: Start, Goal, and all obstacle vertices.

Edges: An edge between two nodes that does not intersect an obstacle, including obstacle edges.

Fig. 10 (a) illustrates a visibility graph for a configuration space containing polygonal obstacles. The motivation for building Visibility Graphs is that the shortest path from the start node to the goal node will be a piecewise linear path that bends only at the obstacles' vertices. This makes sense intuitively - the path would want to hug the obstacles' corners as tightly as possible, as not to add any additional length.

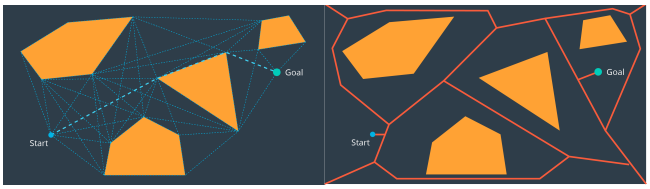

Fig. 10: (a) Visibility Graph with shortest path in the configuration space. (b) Voronoi Diagram maximizes clearance between obstacles in a configuration space.

Once the Visibility Graph is built, a search algorithm can be applied to find the shortest path from Start to Goal as shown in Fig. 13. The advantage of the Visibility Graph is that it is both complete (as it will always contain a path between the start and goal nodes) and optimal (The shortest path from start to goal would move as efficiently as possible - by hugging the corner). One disadvantage to the Visibility Graph is that it leaves no clearance for error. A robot traversing the optimal path would have to pass incredibly close to obstacles, increasing the risk of collision significantly. In certain applications, such as animation or path planning for video games, this is acceptable. However, the uncertainty of real-world robot localization makes this method impractical.

- **Voronoi Diagram**

Another discretization method that generates a roadmap is called the Voronoi Diagram. Unlike the visibility graph method which generates the shortest paths, Voronoi Diagrams maximize clearance between obstacles.

A Voronoi Diagram is a graph whose edges bisect the free space in between obstacles. Every edge lies equidistant from each obstacle around it - with the greatest amount of clearance possible. We can see a Voronoi Diagram for our configuration space in Fig. 10 (b).

Once a Voronoi Diagram is constructed for a workspace, it can be used for multiple queries. Start and goal nodes can be connected into the graph by constructing the paths from the nodes to the edge closest to each of them. Every edge will either be a straight line, if it lies between the edges of two obstacles, or it will be a quadratic, if it passes by the vertex of an obstacle.

2) Cell Decomposition Descretization

Another discretization method that can be used to convert a configuration space into a representation that can easily be explored by a search algorithm is cell decomposition. Cell decomposition divides the space into discrete cells, where each cell is a node and adjacent cells are connected with edges. There are two distinct types of cell decomposition:

- **Exact Cell Decomposition**

Exact cell decomposition divides the space into *non-overlapping* cells. This is commonly done by breaking up the space into triangles and trapezoids, which can be

accomplished by adding vertical line segments at every obstacle's vertex. We can see the result of exact cell decomposition of a configuration space in Fig. 11 (a).

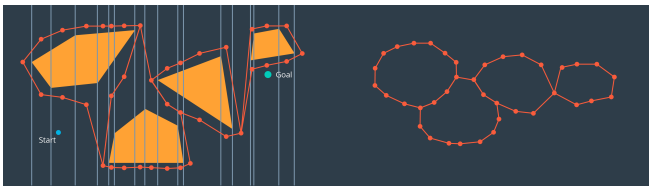

Fig. 11: (a) Exact cell decomposition of a configuration space. (b) Graph generated from exact cell decomposition.

Once a space has been decomposed, the resultant graph can be used to search for the shortest path from start to goal. The resultant graph can be seen in Fig. 11 (b) below.

Exact cell decomposition is elegant because of its precision and completeness. Every cell is either 'full', meaning it is completely occupied by an obstacle, or it is 'empty', meaning it is free. And the union of all cells exactly represents the configuration space. If a path exists from start to goal, the resultant graph *will* contain it.

To implement exact cell decomposition, the algorithm must order all obstacle vertices along the x-axis, and then for every vertex determine whether a new cell must be created or whether two cells should be merged together. Such an algorithm is called the Plane Sweep algorithm.

Exact cell decomposition results in cells of awkward shapes. Collections of uniquely-shaped trapezoids and triangles are more difficult to work with than a regular rectangular grid. This results in an added computational complexity, especially for environments with greater numbers of dimensions. It is also difficult to compute the decomposition when obstacles are not polygonal, but of an irregular shape.

For this reason, there is an alternate type of cell decomposition called approximate cell decomposition, that is much more practical in its implementation.

- **Approximate Cell Decomposition**

Approximate cell decomposition divides a configuration space into discrete cells of simple, regular shapes - such as rectangles and squares (or their multidimensional equivalents). Aside from simplifying the computation of the cells, this method also supports hierarchical decomposition of space (more on this below).

Simple Decomposition: A 2-dimensional configuration space can be decomposed into a grid of rectangular cells. Then, each cell could be marked full or empty, as before. A search algorithm can then look for a sequence of free cells to connect the start node to the goal node. Such a method is more efficient than exact cell decomposition, but it loses its completeness. It is possible that a particular configuration space contains a feasible path, but the resolution of the cells results in some of the cells encompassing the path to be marked 'full' due to the presence of obstacles. A cell will be marked 'full' whether 99% of the space is occupied by an obstacle or a mere 1%. Evidently, this is not practical.

Iterative Decomposition: An alternate method of partitioning a space into simple cells exists. Instead of immediately decomposing the space into small cells of equal size, the method recursively decomposes a space into four quadrants. Each quadrant is marked full, empty, or a new label called 'mixed' - used to represent cells that are somewhat occupied by an obstacle, but also contain some free space. If a sequence of free cells cannot be found from start to goal, then the mixed cells will be further decomposed into another four quadrants. Through this process, more free cells will emerge, eventually revealing a path if one exists. The 2D implementation of this method is called quadtree decomposition as seen in Fig. 12 (a).

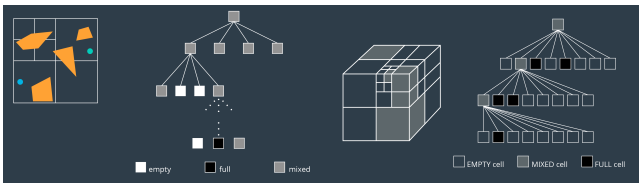

Fig. 12: (a) The 2D implementation framework (quadtree) of the iterative decomposition. (b) The 3D implementation framework (octtree) of the iterative decomposition.

The algorithm behind approximate cell decomposition is much simpler than the exact cell decomposition algorithm. The pseudocode for the algorithm is as below.

(1) Decompose the configuration space into four cells, label cells free, mixed, or full.

(2) Search for a sequence of free cells that connect the start node to the goal node.

(3) If such a sequence exists: Return path

(4) Else: Further decompose the mixed cells

The three dimensional equivalent of quadtrees are octrees, depicted in Fig. 12 (b) below. The method of discretizing a space with any number of dimensions follows the same procedure as the algorithm described above, but expanded to accommodate the additional dimensions.

Although exact cell decomposition is a more elegant method, it is much more computationally expensive than approximate cell decomposition for non-trivial environments. For this reason, approximate cell decomposition is commonly used in practice.

With enough computation, approximate cell decomposition approaches completeness. However, it is not optimal - the resultant path depends on how cells are decomposed. Approximate cell decomposition finds the obvious solution quickly. It is possible that the optimal path squeezes through a minuscule opening between obstacles, but the resultant path takes a much longer route through wide open spaces - one that the recursively-decomposing algorithms would find first.

Approximate cell decomposition is functional, but like all discrete/combinatorial path planning methods - it starts to be computationally intractable for use with high-dimensional environments.

3) Potential Field Descretization

Unlike the methods discussed thus far that discretize the continuous space into a connected graph, the potential field method performs a different type of discretization.

To accomplish its task, the potential field method generates two functions - one that attracts the robot to the goal and one that repels the robot away from obstacles. The two functions can be summed to create a discretized

representation. By applying an optimization algorithm such as gradient descent, a robot can move toward the goal configuration while steering around obstacles. Let's look at how each of these steps is implemented in more detail.

Attractive Potential Field: The attractive potential field is a function with the global minimum at the goal configuration. If a robot is placed at any point and is required to follow the direction of steepest descent, it will end up at the goal configuration. This function does not need to be complicated; a simple quadratic function can achieve all of the requirements discussed above.

$$f_{att}(\mathbf{x}) = v_{att}\,(\|\mathbf{x} - \mathbf{x}_{goal}\|)^2$$

Where $\mathbf{x}$ represents the robot's current position, and $\mathbf{x}_{goal}$ the goal position, and v is a scaling factor.

A fragment of the attractive potential field is shown in Fig. 13 (a) below.

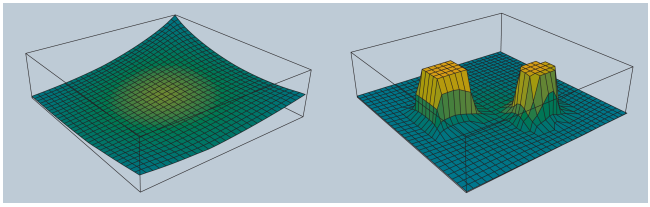

Fig. 13: (a) A fragment of the attractive potential field. (b) A repulsive potential field for an arbitrary configuration space.

Repulsive Potential Field: The repulsive potential field is a function that is equal to zero in free space, and grows to a large value near obstacles. One way to create such a potential field is with the function below.

$$f_{rep}(\mathbf{x}) = v_{rep}\,(\,1/\rho(\mathbf{x}) - 1/\rho_0\,)^2 \qquad if\ \rho \leq \rho 0$$

$$f_{rep}(\mathbf{x}) = 0 \qquad if\ \rho > \rho 0$$

Where the function $\rho(\mathbf{x})$ returns the distance from the robot to its nearest obstacle, ρ_0 is a scaling parameter that defines the reach of an obstacle's repulsiveness, and v is a scaling parameter. Fig. 13 (b) shows an image of a repulsive potential field for an arbitrary configuration space.

Potential Field Sum: The attractive and repulsive functions are summed to produce the potential field that is used to guide the robot from anywhere in the space to the goal. The image in Fig. 14 (a) shows the summation of the functions.

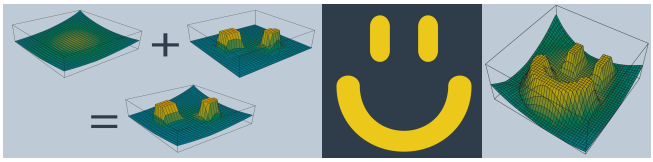

Fig. 14: (a) The attractive and repulsive functions are summed to produce the potential field that is used to guide the robot from anywhere in the space to the goal. (b) A configuration space (left) and the corresponding potential field (right).

Imagine placing a marble onto the surface of the function - from anywhere in the field it will roll in the direction of the goal without colliding with any of the obstacles (as long as ρ_0 is set appropriately)!

The gradient of the function dictates which direction the robot should move, and the speed can be set to be constant or scaled in relation to the distance between the robot and the goal.

Problems with the Potential Field Method: The potential field method is not without its faults - the method is neither complete nor optimal. In certain environments, the method will lead the robot to a local minimum, as opposed to the global minimum. The image in Fig. 14 (b) depict one such instance. Depending on where the robot commences, it may be led to the bottom of the smile. The image depicts the configuration space (left), and the corresponding potential field (right).

The problem of a robot becoming stuck in a local minimum can be resolved by adding random walks, and other strategies that are commonly applied to gradient descent, but ultimately the method is not complete.

The potential field method isn't optimal either, as it may not always find the shortest (or cheapest) path from start to goal. The shortest path may not follow the path of steepest descent. In addition, potential field does not take into consideration the cost of every step.

In continuous representation, we saw how to create a configuration space. In discretization, we have three different types of methods that are used to represent a configuration space with discrete segments. The first of these methods is the roadmap group of methods, where we modelled the configuration space a simple graph, by either connecting the vertices of the obstacles or building a Voronoi diagram. Next, we looked at cell decomposition which broke the space into a finite number of cells, each of which is assessed to be empty, full or mixed, and the empty cells were then linked together to create a graph. Lastly, gradient field or potential field is a method that models configuration space using a 3D function that has the goal as its global minimum and obstacles as tall structures. Most of these methods left us with a graph representation of the space as shown in Fig. 15. In the next section on graph search, we will focus on how to traverse a graph to find the best path for the robot.

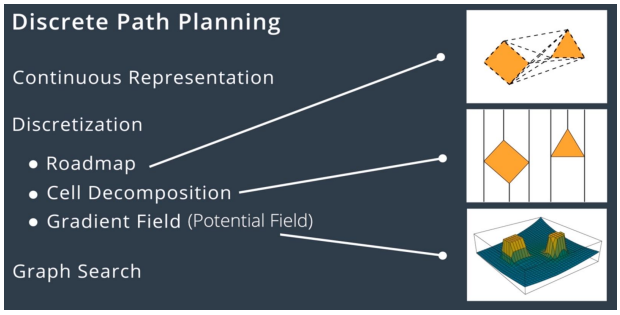

Fig. 15: Various discretization methods for discrete planning.

C. *Graph Search*

The last step of the discrete planning is the Graph Search. Graph search is used to search a finite sequence of discrete actions to connect the start state to a goal state. It does so by searching, visiting states sequentially, asking every state "are you the goal state?" As humans, we may pick out a solution to a search problem when the search problem is 2D and manageable in size. However, a computer must conduct the search manually; it goes through node-by-node and doesn't see the goal node until it is one node away (as shown in Fig. 16). As the size of the space grows and dimensions are added, the problem naturally becomes less trivial. It becomes

imperative to choose the appropriate algorithm for the task to achieve satisfactory results.

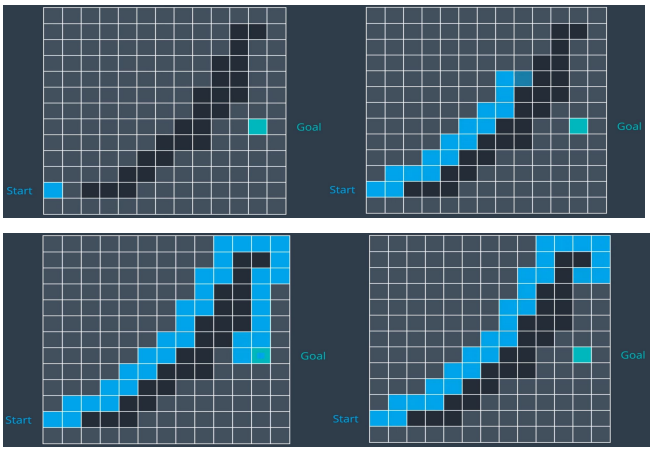

Fig. 16: The algorithm searching a graph node-by-node (top-left→top-right→bottom-right→bottom-left) to find the goal.

There are two different types of search algorithms; **informed** and **uninformed**. Uninformed algorithms search blindly. They are not given any contextual information about how close they are to the goal or how much of an effect every consequent action has. Informed searches on the other hand, can guide the search algorithm to make intelligent decisions; ideally getting them to the goal faster.

*1) **Breadth-First Search***

One of the simplest types of *uninformed* search is called breadth-first search (BFS). It has its name because the algorithm searches broadly before it searches deeply. Let's look at a search tree consisting of multiple interconnected nodes with the start node at the top of the tree. Breadth-first search traverses the tree exploring all of the nodes at one level at a time. It will traverse all of the nodes at the children level before it moves on to traverse the grandchildren nodes and so forth as depicted in Fig. 17 (left) below.

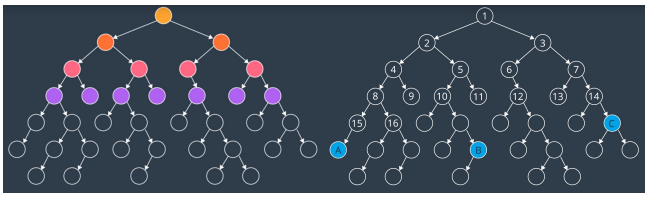

Fig. 17: (Left) BFS traversing one level at a time shown in different colors. (Right) The order at which the BFS search through the tree.

How the algorithm breaks ties changes from implementation to implementation. In a search tree as shown in Fig. 17, it is usually implied that the search moves from left to right at any particular level. After few steps on searching through this tree, the nodes would have been search in the order shown in Fig. 17 (right), starting with node-1 at the top of the tree. For example, the nodes labelled A, B and C shown in Fig. 17 (right) would be searched at steps 23, 27 and 22 respectively through a BFS algorithm.

Breadth-first search is an uninformed search algorithm. This means that it searches blindly without any knowledge of the space it's traversing or where the goal may be. For this reason, it is not the most efficient for its operation.

For a more complicated example shown in Fig. 18, is a discretized map of an environment. The robot starts at the middle of the open space denoted by "S", and would like to find a path to its goal location marked in green. Let's assume that the space is four-connected, meaning that the robot can move up-right-down-left (in this order) but not diagonally. Thus from its start location the robot has four options for where to explore next. The robot cannot explore all four directions at once. Therefore, we are going to add each of these options to something called the frontier.

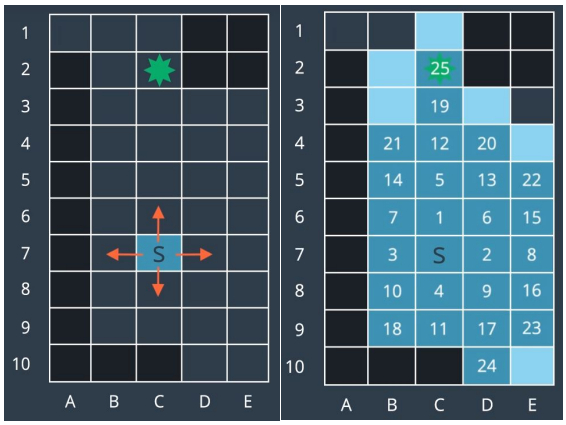

Fig. 18: (Left) A discretized map of an environment where the robot's space is four-connected. (Right) After all nodes are explored using BFS algorithm.

The frontier is the collection of all nodes that we have seen but not yet explored. When the time comes, each of these nodes will be removed from the frontier and explored. Before we add these nodes to the frontier, let's set a standard. In our example in Fig. 18, we will break ties in the following manner. When we have new nodes to add to the frontier, it will choose to add the top node first, then the one on the right, then the left, and if no other options are available then the node directly below.

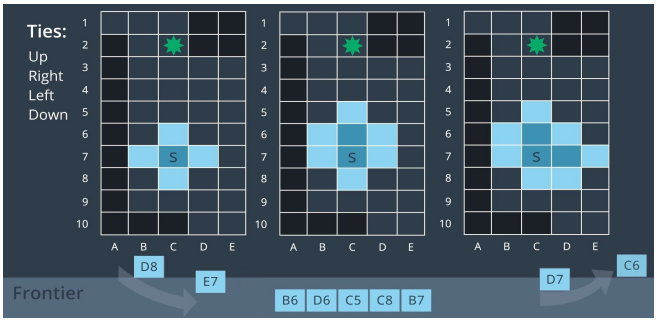

Fig. 19: The queue data structure of the frontier of a BFS algorithm with the first three consecutive search steps.

Now, we can add the four nodes (C6, D7, B7, C8) to the frontier seen in Fig. 19. For BFS, the data structure underlying the frontier is a *queue*. In a que, the first element to enter will be the first to exit. Now, we remove the first node (C6) from the frontier and explore that node, and mark this node dark blue as shown in Fig. 19 (middle). From this node, we can see three more unexplored nodes (C5, D6, B6) if we were to look up, right and left. We add these three nodes to our frontier. One might want to keep exploring upwards. However, as discussed, we always remove the first element from the frontier, then explore, and then add any new nodes to the frontier. So the next node to be explored is D7 as shown in Fig. 19 (right). Then D7 is removed from the

frontier, and E7 and D8 are added. Next, we explore B7, then C8, then C5 and so on. We will notice that the explored area radiates outward from the starting node. That is because; the BFS algorithm searches broadly, visiting the closest nodes first. For this reason, it takes the algorithm a long time to travel a certain distance as it is radiating in all directions. Eventually the algorithm will find the goal node. Now, we can see the order at which the nodes were explored in this BFS example shown in Fig. 18 (right) where the goal is reached at the 25th search step.

The BFS algorithm is *complete* because it will always find a solution, and it is *optimal* because it will always find the shortest solution (since it explores the shortest routes first), but it might take the algorithm a very long time to find the solution, making the algorithm *not efficient*.

2) Depth-First Search

Depth-first search (DFS) is another *uninformed* search algorithm. As the name suggests, DFS searches deep before it searches broadly. If we go back to our search tree example shown in Fig. 20, we can see that instead of commencing at the top node and searching level after level, DFS will explore the start node's first child, then that node's first child and so on, until it hits the lowest leaf in that branch as shown in Fig. 20 (left). Only then, the DFS will back up to a node which had more than one child and explore the node's second child. After few steps of searching through the tree, the nodes would have been searched in the order shown in Fig. 20 (right) starting with node-1 at the top of the tree. For example, the nodes labelled A, B and C shown in Fig. 20 (right) would be searched at steps 20, 18 and 32 respectively through a DFS algorithm.

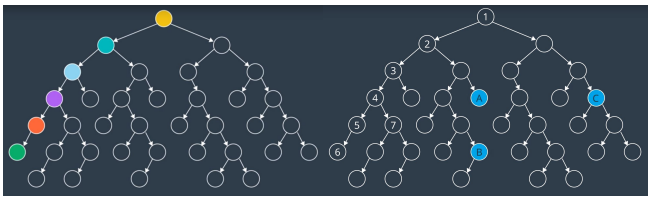

Fig. 20: (Left) DFS traversing all child, grandchild, great-grandchild etc. nodes with a left first strategy shown in different colors. (Right) The order at which the DBFS search through the tree.

Now going back to our discretized map of an environment, let's explore how the DFS algorithm would perform. Our start and goal nodes will remain the same, and so will our rules for breaking ties i.e. in the order up-right-left-down. However, our frontier will change. In BFS, we used the queue for our frontier, which supported expanding the oldest nodes first. In DFS, we wish to expand the newly visited nodes first. To accommodate this, the data structure underlying the frontier will be a *stack*. So, the four nodes visible from the start location will be added to the frontier stack. They are ordered in a way that would have the node above expanded before the right node, left node, and node below. After adding these nodes to the frontier, DFS would pop the top element off the stack and explore next. From C6, three more nodes (B6, D6, C5) are visible. So we add them to the top of the stack. This process continues; remove the front element of the frontier, explore it, and then add any new nodes to the frontier.

As we can see in Fig. 21, DFS is exploring deep in the upwards direction, simply because that's the way the ties are broken. The DFS algorithm continues searching and soon enough it finds the goal. Now, we can see the order at which the nodes were explored in this DFS example shown in Fig. 21 (right) where the goal is reached at the 5th search step, much better than the BSF search for this particular example. However, if the goal node was placed at E10, then it would take DFS 30 searches to find the goal node, where as it would have taken 29 steps for the BFS algorithm to locate the goal (Fig. 22), implying that neither DFS nor BFS is efficient.

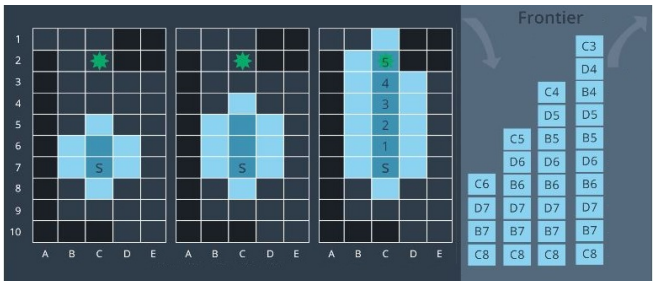

Fig. 21: The stack data structure of the frontier of a DFS algorithm with three search steps reaching goal at 5th step.

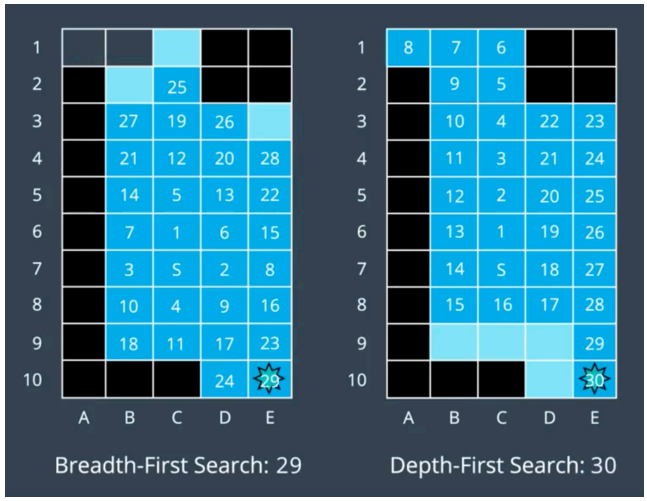

Fig. 22: Comparison between BFS and DFS algorithm performance for start location C7 and goal location E10.

DFS is *not complete* because in an infinitely deep space, the algorithm would end up continuing to explore one branch indefinitely. DFS is *not optimal*, because it may find a longer path down one branch before it is able to see a shorter path or another branch. DFS is *not efficient* either, as it might take the algorithm a very long time to find the solution.

3) Uniform-Cost Search

BFS is optimal because it explores the shallowest unexplored node with every step. However, BFS is limited to graphs where all step costs are equal. This next algorithm uniform cost search (UCS) builds upon breadth-first search to be able to search graphs with differing edge costs. Uniform cost search is also optimal because it expands nodes in order of increasing path costs. In certain environments, we can assign a cost to every edge. The cost may represent one of many things. For instance the time it takes a robot to move from one node to another. A robot may have to slow down, turn corners or to move across rough terrain. The associated delay can be represented with a higher cost to that edge. So far for both BFS and DFS, every edge in our search tree has had the same cost. So let's add some cost to this tree.

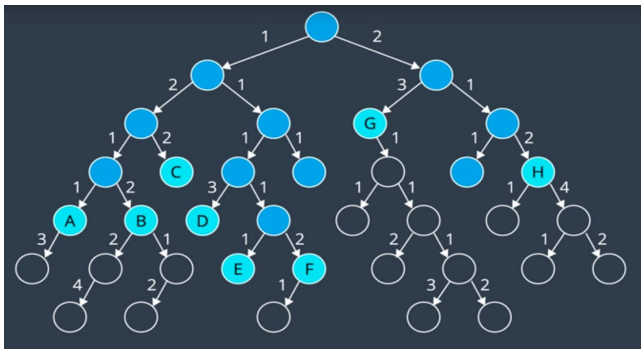

Fig. 23: An example of uniform cost search algorithm where each edge has pre-defined costs.

Uniform cost search explores nodes on the frontier starting with the note that has the lowest path cost. Path cost, refers to the sum of all edge costs leading from the start to that node. For example, the path cost of the right child is just 2, but the path cost of the left-child of the right child is 2+3=5. If we start at the top of the search tree and explore the nodes, they are explored in the following manner; from the start node we add two nodes to the frontier, one has a cost of 1 (left node) and the other a cost of 2 (right node). Next, we explore the node with lower path cost i.e. the left node. This adds two more nodes to the frontier, one with a path-cost of 2+1=3, and one with a path-cost of 1+1=2 as shown in Fig. 23. Now, we have explored all available nodes with a path-cost of 1. Next, the algorithm will explore the nodes with path-cost of 2 and explore them. We continue exploring the nodes on the frontier with the lowest path costs.

As shown in Fig. 22, the algorithm has explored the nodes with path costs of 1, 2, 3, and 4. Now if we were to expand the search to include all nodes with the path coast of 5, among the nodes A to H, the nodes B, D, and F will not be explored, since they all have a path cost of 6. These three nodes along with few other nodes will be explored in the following steps.

Let's explore a more complicated example, with a graph where each node is labeled with a letter A through N and each edge has a cost. We would like to apply a uniform cost search algorithm to find a path from the start node E to the goal node K. Recall that in the BFS the frontier was represented by a queue, and in the DFS the frontier was represented by a stack. Each accommodated to the corresponding algorithm's desired search order; first-in-first-out (FIFO) or last-in-first-out (LIFO). In uniform cost search we wish to explore nodes with the lowest path cost first. To accommodate this we can use a priority queue i.e. a queue that is organized by the path cost.

At the start, node E is connect to four nodes with their corresponding paths on the frontier. They are organized as shown in Fig. 24, with nodes G and F at the top of the queue, as they have the lowest path costs followed by nodes A and D. Next, we remove the top node from the frontier to explore it. As a result, two more nodes are added to the frontier. They assume their appropriate positions in the prioritized queue. Node H has a path cost of 2+3=5, and node I has a path cost of 2+5=7. These are the shortest path to these nodes that we are aware of at this time. Next, we explore node F. In this process node M is added to the queue and another interesting thing happens. The path cost of node I is reduced 6. This is because a shorter route has been found to this node. The algorithm continues searching, exploring the node on the frontier that has the lowest path costs, adding new nodes to the frontier, and updating the past cost for the node if a shorter path has been found. After some time, the goal is found with a path cost of 9, following the path E-F-I-J-K.

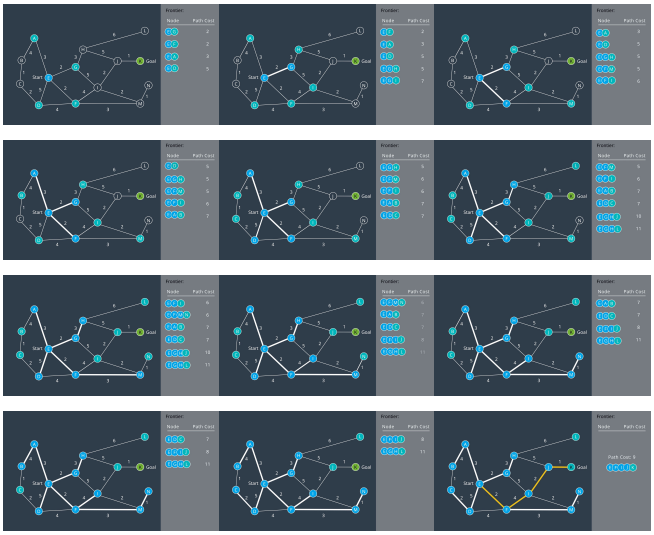

Fig. 24: Various search steps taken using the *uniform cost search algorithm* to go from start location E to end goal K.

The uniform cost search algorithm is *complete* (if every step cost is greater than some value ϵ, otherwise, it can get stuck in infinite loops) and *optimal* but still is *not efficient*, as it searches in all directions, since it has no information that can lead it to the direction of the goal quickly.

4) A Search*

The algorithms that we have discussed thus far have been *uninformed*. Their search was sprawled in all direction because they lacked any information regarding the whereabouts of the goal. However, the A* search algorithm is an *informed* search algorithm. This means that it takes into account about the goal's location as it goes about its search. It does so using something called a *heuristic function*. A heuristic function, Also, $h(p)$, represents the distance from a node to the goal. $h(p)$ is only an estimate of the distance, as the only way to know the true distance would be to traverse the graph. However, even an estimate is beneficial as it steers the search in the appropriate direction. A* uses more than just a heuristic function in its search strategy. It also takes into account the path costs, $g(p)$. A* chooses the path that minimizes the sum, denoted by $f(p)$, of the path cost and the heuristic function. Therefore,

$$f(p) = g(p) + h(p)$$

By doing so, it accomplishes two things at once. Minimizing, $g(p)$ favors shorter paths, and minimizing $h(p)$, favors paths in the direction of the goal. Therefore, A* searches for the shortest path in the direction of goal. Let's go back to our graph shown in Fig. 25 to see this in action. The objective we have is to find the shortest path from node E to node K. Let's try to search this graph once more, this time with the help of a heuristic to guide the algorithm to the goal. For a 2D graph as in Fig. 25, the value of heuristic will be the Euclidean distance from a node to the goal. As we see in the figure, nodes closest to the goal have a low heuristic value and nodes farther away from the goal have a higher heuristic value. The goal itself has a heuristic value of 0.

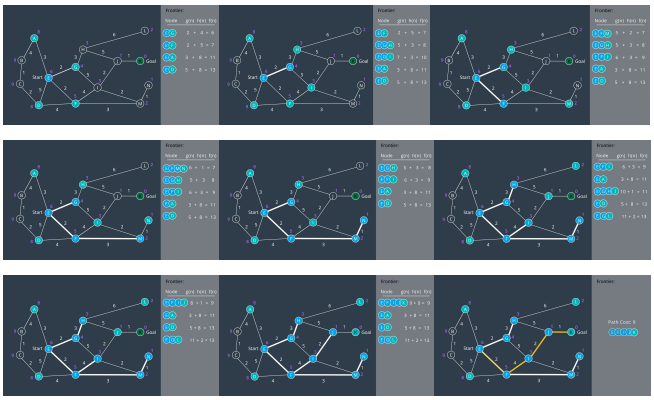

Fig. 25: Various search steps taken using the A* search algorithm to go from start location E to end goal K.

If we wanted calculate the $f(p)$ for a node, we would add the path cost $g(p)$ to the heuristic $h(p)$. For instance node F will have a value of $f(p)$=2+5=7, and the node D will have a value of $f(p)$=5+8=13. Now, let's go through the algorithm steps. Just like in uniform cost search, we will be using a priority queue for the frontier. For A* search we will order by $f(p)$. At the start we have four node and their corresponding paths on the frontier. The path to node G has the lowest $f(p)$, so we will explore it first. As new nodes are added to the frontier they are inserted into the appropriate locations in the priority queue based on $f(p)$. Once again if a shorter route is found to a node, the path and the value of the $f(p)$ for that node will be updated. The A* search continues. Unlike uniform cost search, we can see that the A* search is directed towards the goal. The nodes at the left hand side of the graph (nodes A, B, C, D) have not been explored. However, A* will still explore what it believe to be promising sections, like the dead-end at node N. Here is another instance where the nodes path and the $f(p)$ value are updated. As a result, node J gets bumped up to the top of the frontier and explored next, leading us to the goal. Once again, the shortest path have been found following the path E-F-I-J-K, with a total path cost of 9. Note that A* search took less steps to complete than uniform cost search algorithm as it used the heuristic values for each node to keep itself oriented towards the goal.

In summary, as we saw in Fig. 25, A* search orders the frontier using a priority queue, ordered by $f(p)$, the sum of the path cost and the heuristic function. This is very effective, as it requires the search to keep paths short, while moving towards the goal. However, A* search is not guaranteed to be optimal. Let's look at why this is so!

A* search will find the optimal path *if* the following conditions are met,

- Every edge must have a cost greater than some value, ϵ, otherwise, the search can get stuck in infinite loops and the search would not be complete.
- The heuristic function must be consistent. This means that it must obey the triangle inequality theorem. That is, for three neighboring points (x_1, x_2 , x_3), the heuristic value for x_1 to x_3 must be less than the sum of the heuristic values for x_1 to x_2 and x_2 to x_3.
- The heuristic function must be admissible. This means that $h(p)$ must always be less than or equal to the true cost of reaching the goal from every node. In other words, $h(p)$ must never overestimate the true path cost.

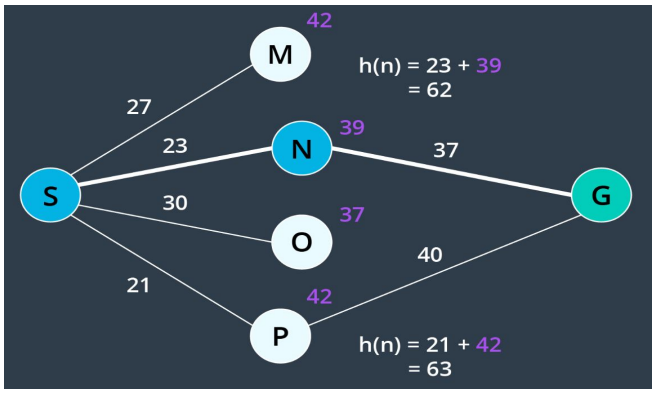

Fig. 26: Admissibility clause of the heuristic function.

To understand where the admissibility clause comes from, let's consider the graph in Fig. 26. Suppose we have two paths to a goal where one is optimal (the highlighted path), and one is not (the lower path). Both heuristics overestimate the path cost. From the start, we have four nodes on the frontier, but Node N would be expanded first because its $h(p)$ is the lowest (it is equal to 62). From there, the goal node is added to the frontier with a cost of 23 + 37 = 60. This node looks more promising than Node P, whose $h(p)$ is equal to 63. In such a case, A* finds a path to the goal which is not optimal. If the heuristics never overestimated the true cost, this situation would not occur because Node P would look more promising than Node N and be explored first.

Admissibility is a requirement for A* to be optimal. For this reason, common heuristics include the Euclidean distance from a node to the goal (as we saw in Fig. 25), or in some applications the Manhattan distance. When comparing two different types of values - for instance, if the path cost is measured in hours, but the heuristic function is estimating distance, then we would need to determine a scaling parameter to be able to sum the two in a useful manner. More about heuristics can be found here [9].

While A* is a much more efficient search in most situations, there will be environments where it will not outperform other search algorithms. This happens if the path to the goal happens to go in the opposite direction first. Variants of A* search exist [10], such as Dijkstra, Pure Heuristic Search, A*, weighted A*, A* with Differential Heuristics, Beam Search, Iterative Deepening, Dynamic Weighting, Bandwidth Search, Bidirectional Search, Dynamic A*, Jump Point Search, Theta*, - some accommodate the use of A* search in dynamic environments, while others help A* become more manageable in large environments. Some of them are described as below and visualization tool [11] can help improve the understanding of these algorithms.

Dijkstra: Dijkstra's algorithm searches uniformly without regard to the heuristic. Thus, when the search starts, we can see it moves outward from the start state in a uniform circle. Dijkstra's algorithm must expand almost every state in the entire map to solve this problem hence guaranteed to find the optimal path.

Bidirectional Search: One way to improve a search's efficiency is to conduct two searches simultaneously - one

rooted at the start node, and another at the goal node. Once the two searches meet, a path exists between the start node and the goal node. The advantage with this approach is that the number of nodes that need to be expanded as part of the search is decreased. The volume swept out by a unidirectional search is much greater than the volume swept out by a bidirectional search for the same problem.

Path Proximity to Obstacles: Another concern with the search of discretized spaces includes the proximity of the final path to obstacles or other hazards. When discretizing a space with methods such as cell decomposition, empty cells are not differentiated from one another. The optimal path will often lead the robot very close to obstacles. In certain scenarios this can be quite problematic, as it will increase the chance of collisions due to the uncertainty of robot localization. The optimal path may not be the best path. To avoid this, a map can be 'smoothed' prior to applying a search to it, marking cells near obstacles with a higher cost than free cells. Then the path found by A* search may pass by obstacles with some additional clearance.

Paths Aligned to Grid: Another concern with discretized spaces is that the resultant path will follow the discrete cells. When a robot goes to execute the path in the real world, it may seem funny to see a robot zig-zag its way across a room instead of driving down the room's diagonal. In such a scenario, a path that is optimal in the discretized space may be suboptimal in the real world. Some careful path smoothing, with attention paid to the location of obstacles, can fix this problem.

Beam Search: In the main A* loop, the OPEN set stores all the nodes that may need to be searched to find a path. The Beam Search is a variation of A* that places a limit on the size of the OPEN set. If the set becomes too large, the node with the worst chances of giving a good path is dropped. One drawback is that the set has to be kept sorted to accomplish this, which limits the kinds of data structures one could choose.

Weighted A*: Weighted A* focuses its search effort on states with low heuristic values, but does not completely ignore the g-costs as Pure Heuristic Search does. As a result, Weighted A* begins with a much "flatter" search than other algorithms. Once the search proceeds around the map, it quickly finds the goal. Weighted A* does not find optimal paths.

Dynamic Weighting: With dynamic weighting, we assume that at the beginning of our search, it's more important to get (anywhere) quickly; at the end of the search, it's more important to get to the goal.

$$f(p) = g(p) + w(p) * h(p)$$

There is a weight ($w >= 1$) associated with the heuristic. As we get closer to the goal, we decrease the weight; this decreases the importance of the heuristic, and increases the relative importance of the actual cost of the path.

Dynamic A* (D*): Another concern with discretized spaces is that the resultant path will follow the discrete cells. When a robot goes to execute the path in the real world, it may seem funny to see a robot zig-zag its way across a room instead of driving. There are variants of A* that allow for changes to the world after the initial path is computed. Dynamic A* or D* [12-14] is intended for use when we don't have complete information. If we don't have all the information, A* can make mistakes; D*'s contribution is that it can correct those mistakes without taking much time. LPA* is intended for use when the costs are changing. With A*, the path may be invalidated by changes to the map; LPA* can re-use previous A* computations to produce a new path.

However, both D* and LPA* require a lot of space—essentially while running A* we keep the internal information of A* (OPEN/CLOSED sets, path tree, g values), and then when the map changes, D* or LPA* will tell us if we need to adjust our path to take into account the map changes. D* and LPA* were designed for robotics, where there is only one robot—we don't need to reuse the memory for some other robot's path.

Theta*: A* would run faster and produce better paths if given a graph of key points (such as corners) instead of the grid. However if we don't want to precompute the graph of corners, we can use Theta*[15], a variant of A* that runs on square grids, to find paths that don't strictly follow the grid. When building *parent* pointers, Theta* will point directly to an ancestor if there's a line of sight to that node, and skips the nodes in between. Unlike path smoothing, which straightens out paths after they're found by A*, Theta* can analyze those paths as part of the A* process. This can lead to shorter paths than post processing a grid path into an any-angle path. Another algorithm called, Block A*[16, 17], can be much faster than Theta* by using a hierarchical approach.

IV. Probabilistic Path Planning

So far the discrete search-based algorithms performed quite well, especially the variants of A*, in a 2D world. However, in 3D world with robots, that have six degrees of freedom or more, the path planning problem becomes more challenging and complex, hence reducing the efficiency of the discrete path planning algorithm discussed thus far.

Performing a complete discretization of the entire space and applying graph search algorithm to the space may be too costly. To tackle the path-planning problem of large size and greater dimension, there exist alternate planning algorithms that fall under the umbrella of sample-based path planning. Instead of conducting complete discretization of the configuration space, these algorithms randomly sample the space hoping that the collection of samples adequately represent the configuration space.

Two methods we will explore here are:

- Sample-Based Path Planning
- Probabilistic Path Planning

So why exactly can't we use discrete planning for higher dimensional problems? Well, it's incredibly hard to discretize such a large space. The complexity of the path planning problem increases exponentially with the number of dimensions in the C-space.

For a 2-dimensional 8-connected space, every node has 8 successors (8-connected means that from every cell we can move laterally or diagonally). Imagine a 3-dimensional 8-connected space, how many successors would every node have? 26. As the dimension of the C-space grows, the number of successors that every cell has increases substantially. In fact, for an n-dimensional space, it is equal to 3^n-1.

It is not uncommon for robots and robotic systems to have large numbers of dimensions. Recall the robotic arm that we worked with in the pick-and-place project - that was a 6-DOF arm. If multiple 6-DOF arms work in a common space, the computation required to perform path planning to avoid collisions increases substantially. Then, think about the complexity of planning for humanoid robots such as the one depicted below. Such problems may take intolerably long to solve using the combinatorial approach.

Aside from robots with many degrees of freedom and multi-robot systems, another computational difficulty involves working with robots that have constrained dynamics. For instance, a car is limited in its motion - it can move forward and backward, and it can turn with a limited turning radius - as we can see in the image below.

However, the car is *not* able to move laterally - as depicted in the following image. *(As unfortunate as it is for those of us that struggle to parallel park!)*

In the case of the car, more complex motion dynamics must be considered when path planning - including the derivatives of the state variables such as velocity. For example, a car's safe turning radius is dependent on it's velocity.

Robotic systems can be classified into two different categories - holonomic and non-holonomic. **Holonomic systems** can be defined as systems where every constraint depends exclusively on the current pose and time, and not on any derivatives with respect to time. **Nonholonomic systems**, on the other hand, are dependent on derivatives. Path planning for nonholonomic systems is more difficult due to the added constraints.

In this section, we will discuss two different path planning algorithms, and understand how to tune their parameters for varying applications.

Combinatorial path planning algorithms are too inefficient to apply in high-dimensional environments, which means that some practical compromise is required to solve the problem! Instead of looking for a path planning algorithm that is both complete and optimal, what if the requirements of the algorithm were weakened?

Instead of aspiring to use an algorithm that is complete, the requirement can be weakened to use an algorithm that is probabilistically complete. A **probabilistically complete** algorithm is one whose probability of finding a path, if one exists, increases to 1 as time goes to infinity.

Similarly, the requirement of an optimal path can be weakened to that of a feasible path. A **feasible path** is one that obeys all environmental and robot constraints such as obstacles and motion constraints. For high-dimensional problems with long computational times, it may take unacceptably long to find the optimal path, whereas a feasible path can be found with relative ease. Finding a feasible path proves that a path from start to goal exists, and if needed, the path can be optimized locally to improve performance.

Sample-based planning is probabilistically complete and looks for a feasible path instead of the optimal path.

A. Sample-Based Path Planning

Sample-based path planning differs from combinatorial path planning in that it does not try to systematically discretize the entire configuration space. Instead, it samples the configuration space randomly (or semi-randomly) to build up a representation of the space. The resultant graph is not as precise as one created using combinatorial planning, but it is much quicker to construct because of the relatively small number of samples used.

Such a method is probabilistically complete because as time passes and the number of samples approaches infinity, the probability of finding a path, if one exists, approaches 1.

Such an approach is very effective in high-dimensional spaces, however it does have some downfalls. Sampling a space uniformly is not likely to reach small or narrow areas, such as the passage depicted in the image below. Since the passage is the only way to move from start to goal, it is critical that a sufficient number of samples occupy the passage, or the algorithm will return 'no solution found' to a problem that clearly has a solution.

Different sample-based planning approaches exist, each with their own benefits and downfalls. In the next few pages we will learn about,

- Probabilistic Roadmap Method (PRM)
- Rapidly Exploring Random Tree Method (RRT)

We will also learn about Path Smoothing - one improvement that can make resultant paths more efficient.

1) Probabilistic Roadmap Method

The pseudo-code for the Probabilistic Roadmap Method (PRM) learning phase is provided below.

```
Initialize an empty graph
For n iterations:
  Generate a random configuration.
  If the configuration is collision free:
    Add the configuration to the graph.
    Find the k-nearest neighbours of the configuration.
    For each of the k neighbours:
      Try to find a collision-free path between
        the neighbour and original configuration.
        If edge is collision-free:
          Add it to the graph.
```

After the learning phase, comes the query phase.

There are several parameters in the PRM algorithm that require tweaking to achieve success in a particular application. Firstly, the **number of iterations** can be adjusted - the parameter controls between how detailed the resultant graph is and how long the computation takes. For path planning problems in wide-open spaces, additional detail is unlikely to significantly improve the resultant path. However, the additional computation is required in complicated environments with narrow passages between obstacles. Beware, setting an insufficient number of

iterations can result in a 'path not found' if the samples do not adequately represent the space.

Another decision that a robotics engineer would need to make is **how to find neighbors** for a randomly generated configuration. One option is to look for the k-nearest neighbors to a node. To do so efficiently, a k-d tree can be utilized - to break up the space into 'bins' with nodes, and then search the bins for the nearest nodes. Another option is to search for any nodes within a certain distance of the goal. Ultimately, knowledge of the environment and the solution requirements will drive this decision-making process.

The choice for what type of **local planner** to use is another decision that needs to be made by the robotics engineer. The local planner demonstrated in the video is an example of a very simple planner. For most scenarios, a simple planner is preferred, as the process of checking an edge for collisions is repeated many times (k*n times, to be exact) and efficiency is key. However, more powerful planners may be required in certain problems. In such a case, the local planner could even be another PRM.

As discussed before, sample-based path planning algorithms are probabilistically complete. Now that we have seen one such algorithm in action, we can see why this is the case. As the number of iterations approaches infinity, the graph approaches completeness and the optimal path through the graph approaches the optimal path in reality.

The algorithm that we learned here is the vanilla version of PRM, but many other variations to it exist. The following link discusses several alternative strategies for implementing a PRM that may produce a more optimal path in a more efficient manner. More can be found in the comparative study of Probabilistic Roadmap Planners [18].

The Learning Phase takes significantly longer to implement than the Query Phase, which only has to connect the start and goal nodes, and then search for a path. However, the graph created by the Learning Phase can be reused for many subsequent queries. For this reason, PRM is called a **multi-query planner**.

This is very beneficial in static or mildly changing environments. However, some environments change so quickly that PRM's multi-query property cannot be exploited. In such situations, PRM's additional detail and computational slow nature is not appreciated. A quicker algorithm would be preferred - one that doesn't spend time going in *all* directions without influence by the start and goal.

*2) **Rapidly Exploring Random Tree Method***

The pseudo-code for the Rapidly Exploring Random Tree (RRT) learning phase is provided below.

```
Initialize two empty trees.
Add start node to tree #1.
Add goal node to tree #2.
For n iterations, or until an edge connects trees #1 & #2:
    Generate a random configuration (alternating trees).
    If the configuration is collision free:
        Find the closest neighbor on the tree to the configuration
        If the configuration is less than a distance δ away from the neighbor:
            Try to connect the two with a local planner.
        Else:
            Replace the randomly generated configuration
                with a new configuration that falls along the same path,
                but a distance δ from the neighbor.
            Try to connect the two with a local planner.
        If node is added successfully:
            Try to connect the new node to the closest neighbor.
```

Just like with PRM, there are a few parameters that can be tuned to make RRT more efficient for a given application.

The first of these parameters is the **sampling method** (ie. how a random configuration is generated). As discussed in the video, we can sample uniformly - which would favor wide unexplored spaces, or we can sample with a bias - which would cause the search to advance greedily in the direction of the goal. Greediness can be beneficial in simple planning problems, however in some environments it can cause the robot to get stuck in a local minima. It is common to utilize a uniform sampling method with a *small* hint of bias.

The next parameter that can be tuned is δ. As RRT starts to generate random configurations, a large proportion of these configurations will lie further than a distance δ from the closest configuration in the graph. In such a situation, a randomly generated node will dictate the direction of growth, while δ is the growth rate.

Choosing a small δ will result in a large density of nodes and small growth rate. On the other hand, choosing a large δ may result in lost detail, as well as an increasing number of nodes being unable to connect to the graph due to the greater chance of collisions with obstacles. δ must be chosen carefully, with knowledge of the environment and requirements of the solution.

Since the RRT method explores the graph starting with the start and goal nodes, the resultant graph cannot be applied to solve additional queries. RRT is a single-query planner.

RRT is, however, much quicker than PRM at solving a path planning problem. This is so *because* it takes into account the start and end nodes, and limits growth to the area surrounding the existing graph instead of reaching out into all distant corners, the way PRM does. RRT is more efficient than PRM at solving large path planning problems (ex. ones with hundreds of dimensions) in dynamic environments.

Generally speaking, RRT is able to solve problems with 7 dimensions in a matter of milliseconds, and may take several minutes to solve problems with over 20 dimensions. In comparison, such problems would be impossible to solve with the combinatorial path planning method.

While we will not go into significant detail on this topic, the RRT method supports planning for non-holonomic systems, while the PRM method does not. This is so because the RRT method can take into consideration the additional constraints (such as a car's turning radius at a particular

speed) when adding nodes to a graph, the same way it already takes into consideration how far away a new node is from an existing tree.

*3) **Path Smoothing***

The following algorithm provides a method for smoothing the path by shortcutting.

```
For n iterations:
    Select two nodes from the graph
    If the edge between the two nodes is shorter than the existing path between the nodes:
        Use local planner to see if edge is collision-free.
        If collision-free:
            Replace existing path with edge between the two nodes.
```

Keep in mind that the path's distance is not the only thing that can be optimized by the Path Shortcutter algorithm - it could optimize for path smoothness, expected energy use by the robot, safety, or any other measurable factor.

After the Path Shortcutting algorithm is applied, the result is a more optimized path. It may still not be *the optimal path*, but it should have at the very least moved towards a local minimum. There exist more complex, informed algorithms that can improve the performance of the Path Shortcutter. These are able to use information about the workspace to better guide the algorithm to a more optimal solution.

For large multi-dimensional problems, it is not uncommon for the time taken to optimize a path to exceed the time taken to search for a feasible solution in the first place.

Not Complete: Sample-based planning is not complete, it is probabilistically complete. In applications where decisions need to be made quickly, PRM & RRT may fail to find a path in difficult environments, such as the one shown below. To path plan in an environment such as the one presented above, alternate means of sampling can be introduced (such as Gaussian or Bridge sampling). Alternate methods bias their placement of samples to obstacle edges or vertices of the open space.

Not Optimal: Sample-based path planning isn't optimal either - while an algorithm such as A* will find the most optimal path within the graph, the graph is not a thorough representation of the space, and so the true optimal path is unlikely to be represented in the graph.

Overall, there is no silver bullet algorithm for sample-based path planning. The PRM & RRT algorithms perform acceptably in most environments, while others require customized solutions. An algorithm that sees a performance improvement in one application, is not guaranteed to perform better in others. Ultimately, sample-based path planning makes multi-dimensional path planning feasible! Some modified versions of sample-based planning are Anytime algorithm, and RRT*.

- **Anytime algorithm**: an anytime algorithm is an algorithm that will return a solution even if it's computation is halted before it finishes searching the entire space. The longer the algorithm plans, the more optimal the solution will be.
- **RRT***: RRT* is a variant of RRT that tries to smooth the tree branches at every step. It does so by looking to see whether a child node can be swapped with it's parent (or it's parent's parent, etc) to produce a more direct path. The result is a less zig-zaggy and more optimal path.

B. Probabilistic Path Planning

Let's reconsider the exploratory robot from Fig. 2, that was tasked with finding a path from its drop-off location to the goal location that would be safe for humans to follow. The terrain contains a lot of different hazards. The operator of the rover is willing to take whatever risk is necessary, but would naturally want to minimize it as much as possible. The algorithms that we have discussed thus far are unable to adequately model the risk.

For instance, a combinatorial path planning algorithm would have no difficulty finding the path to the goal location. The path may be the best by all other means, but due to the uncertainty of the rover motion there's a chance that the rover will meet its demise along this path. It is possible to inflate the size of the rover to ensure that there is enough room maneuver. But, as we have seem the algorithm will no longer be complete. Another idea is to give negative rewards to dangerous areas of the map so that the search algorithm is more likely to select alternate paths similar to what we did in reinforcement learning. This is a step in the right direction and would cause the rover to avoid dangerous areas, but it does not actually consider the uncertainty of the rover motion.

What we really like to model is the uncertainty. By considering a non-deterministic transition model. Since that path execution is uncertain, an algorithm that takes the uncertainty into account explicitly is more likely to produce realistic paths. So this leads us to the use the Markov Decision Process (MDP) for probabilistic path planning.

*1) **Markov Decision Process***

Recall the recycling robot example from the Deep Reinforcement Learning project [19]. The robot's goal was to drive around its environment and pick up as many cans as possible. It had a set of **states** that it could be in, and a set of **actions** that it could take. The robot would receive a **reward** for picking up cans, however, it would also receive a negative reward (a penalty) if it were to run out of battery and get stranded.

The robot had a non-deterministic **transition model** (sometimes called the *one-step dynamics*). This means that an action cannot guarantee to lead a robot from one state to another state. Instead, there is a probability associated with resulting in each state.

Say at an arbitrary time step t, the state of the robot's battery is high (S_t=*high*). In response, the agent decides to search for cans (A_t=*search*). In such a case, there is a 70% chance of the robot's battery charge remaining high and a 30% chance that it will drop to low.

Let's revisit the definition of an MDP before moving forward. A Markov Decision Process is defined by:

- A set of states: S,

- Initial state: s_0,
- A set of actions: A,
- The transition model: $T(s,a,s')$,
- A set of rewards: R.

The transition model is the probability of reaching a state s' from a state s by executing action a. It is often written as $T(s,a,s')$.

The Markov assumption states that the probability of transitioning from s to s' is only dependent on the present state, s, and not on the path taken to get to s.

One notable difference between MDPs in probabilistic path planning and MDPs in reinforcement learning, is that in path planning the robot is fully aware of all of the items listed above (state, actions, transition model, rewards). Whereas in RL, the robot was aware of its state and what actions it had available, but it was not aware of the rewards or the transition model.

*2) **Mobile Robot Example***

In our mobile robot example, movement actions are non-deterministic. Every action will have a probability less than 1 of being successfully executed. This can be due to a number of reasons such as wheel slip, internal errors, difficult terrain, etc. The image below showcases a possible transition model for our exploratory rover, for a scenario where it is trying to move forward one cell.

The intended action of moving forward one cell is only executed with a probability of 0.8 (80%) and with a probability of 0.1 (10%), the rover will move left, or right. Let's also say that bumping into a wall will cause the robot to remain in its present cell.

Let's provide the rover with a simple example of an environment for it to plan a path in. The environment shown below has the robot starting in the top left cell, and the robot's goal is in the bottom right cell. The mountains represent terrain that is more difficult to pass, while the pond is a hazard to the robot. Moving across the mountains will take the rover longer than moving on flat land, and moving into the pond may drown and short circuit the robot.

Combinatorial Path Planning Solution: If we were to apply A* search to this discretized 4-connected environment, the resultant path would have the robot move right 2 cells, then down 2 cells, and right once more to reach the goal (or R-R-D-R-D, which is an equally optimal path). This truly is the shortest path, however, it takes the robot right by a very dangerous area (the pond). There is a significant chance that the robot will end up in the pond, failing its mission.

Probabilistic Path Planning Solution: If we are to plan a path using MDPs, we might be able to get a better result! In each state (cell), the robot will receive a certain reward, $R(s)$. This reward could be positive or negative, but it cannot be infinite. It is common to provide the following rewards,

- small negative rewards to states that are not the goal state(s) - to represent the cost of time passing (a slow moving robot would incur a greater penalty than a speedy robot),
- large positive rewards for the goal state(s), and
- large negative rewards for hazardous states - in hopes of convincing the robot to avoid them.

These rewards will help guide the rover to a path that is efficient, but also safe - taking into account the uncertainty of the rover's motion. Fig. 28 displays the environment with appropriate rewards assigned.

As we can see, entering a state that is not the goal state has a reward of -1 if it is a flat-land tile, and -3 if it is a mountainous tile. The hazardous pond has a reward of -50, and the goal has a reward of 100.

With the robot's transition model identified and appropriate rewards assigned to all areas of the environment, we can now construct a **policy**. Let us discuss how the policy is implemented in probabilistic path planning!

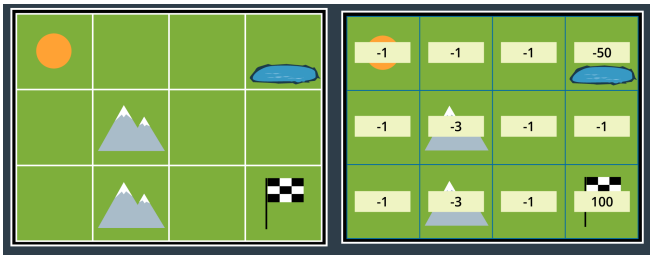

Fig. 28: An environment with appropriate rewards for a robot agent to arrive at its goal location.

Policies: Recall from the Reinforcement Learning Project [19] that a solution to a Markov Decision Process is called a policy, and is denoted with the letter π. A policy is a mapping from states to actions. For every state, a policy will inform the robot of which action it should take. An optimal policy, denoted $\pi*$, informs the robot of the *best* action to take from any state, to maximize the overall reward. More on this can be found in the Deep Reinforcement Learning writeup [19].

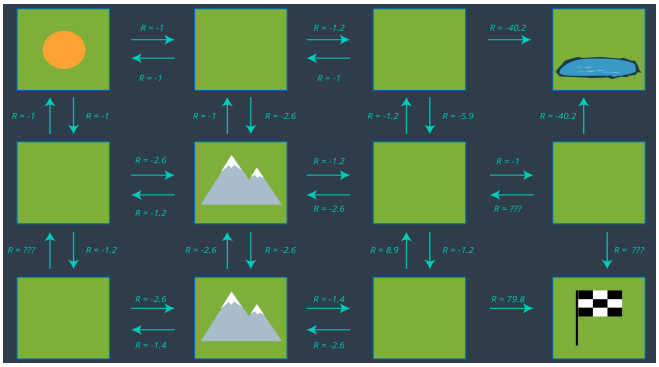

Fig. 29: The set of actions that the robot can take in its environment to arrive at its goal.

Fig. 29 displays the set of actions that the robot can take in its environment. Note that there are no arrows leading away from the pond, as the robot is considered DOA (dead on arrival) after entering the pond. As well, no arrows leave the goal as the path planning problem is complete once the robot reaches the goal - after all, this is an *episodic task*.

From this set of actions, a policy can be generated by selecting one action per state. Before we revisit the process of selecting the appropriate action for each policy, let's look at how some of the values above were calculated. After all, -5.9 seems like quite an odd number!

Calculating Expected Reward: Recall that the reward for entering an empty cell is -1, a mountainous cell -3, the pond -50, and the goal +100. These are the rewards defined according to the environment. However, if our robot wanted to move from one cell to another, it is not guaranteed to succeed. Therefore, we must calculate the expected reward, which takes into account not just the rewards set by the environment, but the robot's transition model too.

Let's look at the bottom mountain cell first. From here, it is intuitively obvious that moving right is the best action to take, so let's calculate that one. If the robot's movements were deterministic, the cost of this movement would be trivial (moving to an open cell has a reward of -1). However, since our movements are non-deterministic, we need to evaluate the *expected* reward of this movement. The robot has a probability of 0.8 of successfully moving to the open cell, a probability of 0.1 of moving to the cell above, and a probability of 0.1 of bumping into the wall and remaining in its present cell.

$$\textit{expected reward} = 0.8*(-1)+0.1*(-3)+0.1*(-3) = -1.4$$

All of the expected rewards are calculated in this way, taking into account the transition model for this particular robot. We may have noticed that a few expected rewards are missing in the image above. The Fig. ?? has all of the expected rewards filled in.

Now that we have an understanding of our expected rewards, we can select a policy and evaluate how efficient it is. Once again, a policy is just a mapping from states to actions. If we review the set of actions depicted in Fig. ?? above, and select just one action for each state - i.e. exactly one arrow leaving each cell (with the exception of the hazard and goal states) - then we have ourselves a policy.

However, we're not looking for *any* policy, we'd like to find the *optimal* policy. For this reason, we'll need to study the utility of each state to then determine the *best* action to take from each state. That's what the next concept is all about!

State Utility (State-Value): The utility of a state (otherwise known as the state-value) represents how attractive the state is with respect to the goal. Recall that for each state, the state-value function yields the expected return, if the agent (robot) starts in that state and then follows the policy for all time steps. In mathematical notation, this can be represented as so:

$$U^{\pi}(s) = E[\sum_{t=0}^{\infty} R(s_t) | \pi, s_0 = s]$$

The notation used in path planning differs slightly from what you saw in Reinforcement Learning. But the result is identical.

Here,

- $U^{\pi}(s)$ represents the utility of a state s,
- E represents the *expected* value, and
- $R(s)$ represents the reward for state s.

The utility of a state is the sum of the rewards that an agent would encounter if it started at that state and followed the policy to the goal.

We can break the equation down, to further understand it.

$$U^{\pi}(s) = E[\sum_{t=0}^{\infty} R(s_t) | \pi, s_0 = s]$$

Let's start by breaking up the summation and explicitly adding all states.

$$U^{\pi}(s) = E[R(s_0) + R(s_1) + R(s_2) + \ldots | \pi, s_0 = s]$$

Then, we can pull out the first term. The expected reward for the first state is independent of the policy. While the expected reward of all future states (those between the state and the goal) depend on the policy.

$$U^{\pi}(s) = E[R(s_0) | s_0 = s] + E[R(s_1) + R(s_2) + \ldots | \pi]$$

Re-arranging the equation results in the following. (Recall that the prime symbol, as on s', represents the next state - like s_2 would be to s_1).

$$U^{\pi}(s) = R(s) + E[\sum_{t=0}^{\infty} R(s_t) | \pi, s_0 = s']$$

Ultimately, the result is the following.

$$U^{\pi}(s) = R(s) + U^{\pi}(s')$$

As we see here, calculating the utility of a state is an iterative process. It involves all of the states that the agent would visit between the present state and the goal, as dictated by the policy. As well, it should be clear that the utility of a state depends on the policy. If we change the policy, the utility of each state will change, since the sequence of states that would be visited prior to the goal may change.

Determining Optimal Policy: Recall that the **optimal policy**, denoted $\pi*$, informs the robot of the *best* action to take from any state, to maximize the overall reward. That is,

$$\pi*(s) = argmax_a E[U^{\pi}(s)]$$

In a state s, the optimal policy $\pi*$ will choose the action a that maximizes the utility of s (which, due to its iterative nature, maximizes the utilities of all future states too). While the math may make it seem intimidating, it's as easy as looking at the set of actions and choosing the best action for every state. The image in Fig. ?? displays the set of all actions once more.

It may not be clear from the get-go which action is optimal for every state, especially for states far away from the goal, which have many paths available to them. It's often helpful to start at the goal and work your way backwards.

If you look at the two cells adjacent to the goal, their best action is trivial - go to the goal! Recall from your learning in RL that the goal state's utility is 0. This is because if the agent starts at the goal, the task is complete and no reward is received. Thus, the expected reward from either of the goal's adjacent cells is 79.8. Therefore, the state's utility is, 79.8 + 0 = 79.8 (based on $U^{\pi}(s) = R(s) + U^{\pi}(s')$).

If we look at the lower mountain cell, it is also easy to guess which action should be performed in this state. With an expected reward of -1.2, moving right is going to be much more rewarding than taking any indirect route (up or left). This state will have a utility of -1.2 + 79.8 = 78.6. The process of selecting each state's most rewarding action continues, until every state is mapped to an action. These mappings are precisely what make up the policy.

It is highly suggested that you pause this lesson here, and work out the optimal policy on your own using the action set seen above. Working through the example yourself will give you a better understanding of the challenges that are faced in

the process, and will help you remember this content more effectively. When you are done, you can compare your results with the images below.

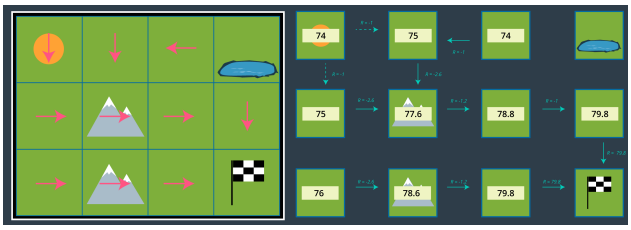

Fig. 30: The optimal set of actions the robot can take to arrive at its goal.

Once this process is complete, the agent (our robot) will be able to make the best path planning decision from every state, and successfully navigate the environment from any start position to the goal. The optimal policy for this environment and this robot is provided below.

The image below in Fig. 30 shows that the set of actions with just the optimal actions remaining. Note that from the top left cell, the agent could either go down or right, as both options have equal rewards.

Discounting: One simplification we made was we omitted the discounting rate γ. In the above example, γ=1 and all future actions were considered to be just as significant as the present action. This was done solely to simplify the example.

In reality, discounting is often applied in robotic path planning, since the future can be quite uncertain. The complete equation for the utility of a state is provided below:

$$U^{\pi}(s)=E[\sum_{t=0}^{\infty}\gamma^{t}R(s_t)|\pi,s_0=s]$$

More on the discounting can be found from in the Deep Reinforcement Learning report [19].

Value Iteration Algorithm: The process that we went through to determine the optimal policy for the mountainous environment was fairly straightforward, but it did take some intuition to identify which action was optimal for every state. In larger more complex environments, intuition may not be sufficient. In such environments, an algorithm should be applied to handle all computations and find the optimal solution to an MDP. One such algorithm is called the Value Iteration algorithm. *Iteration* is a key word here, and you'll see just why!

The Value Iteration algorithm will initialize all state utilities to some arbitrary value - say, zero. Then, it will iteratively calculate a more accurate state utility for each state, using $U(s)=R(s)+\gamma max_a \Sigma s' T(s,a,s')U(s')$. The algorithm is as follows:

$U'=0$

loop until *close-enough*(U,U')

  $U=U'$

  for s in S, do:

    $U(s)=R(s)+\gamma max_a \Sigma s' T(s,a,s')U(s')$

return U

With every iteration, the algorithm will have a more and more accurate estimate of each state's utility. The number of iterations of the algorithm is dictated by a function close-enough*close-enough* which detects convergence. One way to accomplish this is to evaluate the root mean square error,

$$RMS=(1/|S|)\ sqrt(\sum_s (U(s)-U'(s))^2)$$

Once this error is below a predetermined threshold, the result has converged sufficiently.

$$RMS(U,U') < \epsilon$$

This algorithm finds the optimal policy to the MDP, regardless of what U' is initialized to (although the efficiency of the algorithm will be affected by a poor U').

C. *Navigation Using Deep Reinforcement Learning*

In this sections, we take a look at navigation using Deep Reinforcement Learning. This is a new field and is an active area of research. New algorithms and improved approaches in Deep RL for navigation show promise for the future of AGI (Artificial General Intelligence), which motivates the research.

Autonomous navigation through complex environments has been demonstrated with various SLAM (Simultaneous Localization and Mapping) algorithms, where position inference is combined with mapping. Recent advances in deep neural networks and reinforcement learning (RL) have opened the door to solving navigation with deep RL where the basic idea is to use visual inputs and rewards to train a neural network to output correct actions for robotic navigation to a goal. In reality, there are a number of challenges in implementation.

An approach to the problem of navigation in complex environments using a deep RL solution is provided in the Deep Mind papers [20, 21]. The approach in [21] is tested in visually rich simulated 3-D game-like maze environments and provides an analysis of results that compare favorably to human behavior for the same mazes. The paper addresses the following challenges in navigation using RL:

- Rewards are often sparsely distributed in an environment
- Environments often include dynamic elements

The approach in the Mirowski paper addresses the problems of sparse rewards and dynamic elements by incorporating multiple objectives and using a number of Deep RL algorithmic improvements over the simple DQN algorithm implemented in the Deep RL project [19].

In addition to the primary objective of maximizing cumulative rewards, as usual, there are a couple of auxiliary objectives as well. These are to infer depth estimates from RGB observations and to detect loop closures in mapping. By training for these auxiliary objectives, the agent learns information that it also needs for object avoidance and path planning. As explained in the Deep Mind paper on auxiliary tasks in Deep RL [22], this is "similar to how a baby might learn to control their hands by moving them and observing the movements". For robust performance in deep RL for path planning, a new method called A3C can be used.

1) A3C Algorithm for Deep RL

For path planning, the deep RL algorithm can use an Asynchronous Advantage Actor-Critic (A3C) algorithm [23], instead of a DQN algorithm for improved results. In A3C, "Asynchronous" refers to the idea that multiple networks are

trying to solve the same problem in parallel. These "worker" agents can explore different parts of the environment simultaneously (Fig. 31). This speeds up the learning and results in more diverse training and eclipses DQN regarding efficiency. Actor-critic algorithms require both policy (π) and value function prediction(V) from the network. The A3C method is faster and more robust than DQN, and has essentially replaced DQN in many applications. A3C builds on actor-critic with the innovations of multiple "asynchronous" workers as well as an "advantage" feature. More details about the algorithm can be found in [23].

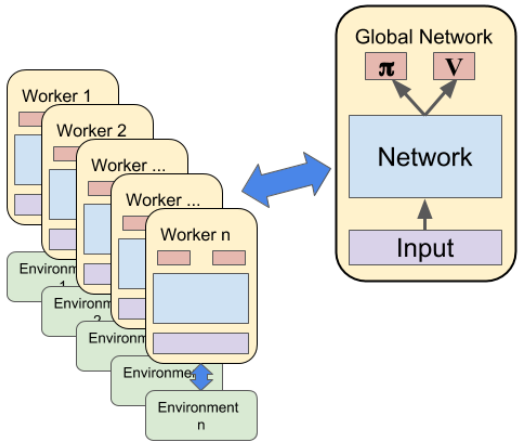

Fig. 31: The worker agents in A3C algorithm explore different parts of the environment simultaneously.

We've seen that the *value function* estimate is used as a scoring function by the "critic" to evaluate the "actor" *policy function*. As a further refinement, the *advantage function* can be estimated and used as the scoring function by the critic instead. The advantage function approximates the advantage of one action over others in a given state with the following equation:

$$A(s,a)=Q(s,a)-V(s)$$

A *Q-value* tells us how much reward we expect to get by taking action a in state s, whereas, the *advantage value*, $A(s,a)$ tells us how much **more** reward we can expect beyond the expected value of the state, $V(s)$. Put another way, it answers the question: "What are we gaining by taking this action instead of any action at random?"

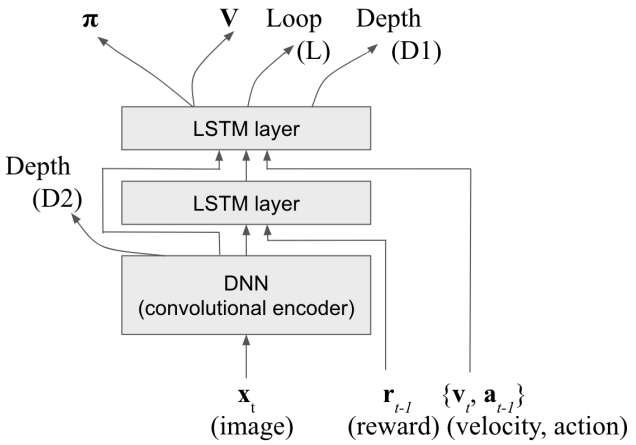

Fig. 32: Overview of the deep RL neural net architecture described in [21].

The network architecture also includes memory in the form of LSTM (Long Short Term Memory) layers, which add the capability of learning long-term dependencies. Fig. 32 gives an overview of the deep RL neural net architecture described in [21] and provides an analysis of navigation success using various permutations of the A3C algorithm, LSTM memory, and auxiliary objectives. Notice that the auxiliary depth predictions are provided at two locations, the convolutional (D1) and LSTM (D2) layers. These are then compared against human behavior as well. The conclusion is that the best combination for most of the mazes is to use the auxiliary objectives connected to the neural network, using the A3C algorithm and including LSTM memory. The results are very close to human behavior.

2) Challenges in Deep RL Navigation

The goal of autonomous navigation for robots using Deep RL technology is to provide an end-to-end, "pixels to actions", solution. As the cost of high performance compute power and data storage has gone down, approaches using Deep reinforcement learning have risen in popularity within the AI community. Autonomous navigation using Deep RL is now possible in some constrained environments.

Motivations: The motivations for pursuing this approach over more conventional methods include:

- The importance of end-to-end mobile autonomy as an aspect of a larger goal: to develop general-purpose AI systems that can interact and learn from the world.
- The implicit connection between representation and actions in Deep RL networks: actions are learned together with the representation (the network), ensuring that task-relevant features are present in the representation.
- The deeper understanding of surroundings that can be achieved with RL agents: an agent can perform experiments to better "understand" its complex and changing environment, which leads to more nuanced and human-like behavior by the robot.

Challenges: The field of Deep RL is still in its infancy and an area of active research. Applying it to robotic navigation is complex and full of challenges in the real world. Current challenges include:

- Rewards that the RL algorithm needs are often sparsely distributed and hard to define in environments. Auxiliary goals can be used to partially address this problem.
- Environments often include dynamic elements and constant change. This means the agents must have memory of some sort on different time scales to remember: the goal location, the developing map based on visual observations, and longer term boundaries and cues from the environment.
- Training a robot through trial and error in simulation cannot fully prepare it for a real world environment. While training in a simulator costs time and compute power, it doesn't involve the physical risk that occurs from failures with a real world mobile robot. Advances in photo-realistic generation of realistic simulation environments may hold promise in this area.

V. Autonomous Navigation Project Setup

In this project we will program a home service robot that will need to interface it with different ROS packages in order to simultaneously plan a path, localize, and map the environment autonomously. Some of these packages are official ROS packages, which offer great tools, and others are packages that we will create. The first goal is to prepare and build the catkin workspace.

For this project, we will not be using the probabilistic path planning and Deep RL based path planning discussed in Section IV. In this project we will be using Grid-based Fast SLAM and AMCL for mapping and localization, and the *uniform cost search* based *Dijkstra* algorithm for path planning.

The list of the official ROS packages and other custom packages and directories that we will need to create are discussed in this section. The following steps can be used for the Catkin Workspace setup for the project.

1. Update the system with *sudo apt-get update*
2. Install the ROS navigation system with *sudo apt-get install ros-kinetic-navigation*
3. Create a catkin workspace called *catkin_ws*
4. Install all the official ROS packages from GitHub
5. Install all the packages dependencies with *rosdep -i install <package name>* (Only a single version of each package must be present)
6. Build the catkin workspace with *catkin_make*

A. Use Official ROS Packages

Import the following ROS packages and install them in the *catkin_ws/src* directory. The full GitHub directory needs to be cloned and not just the package itself.

1. **gmapping** [25]: With the *gmapping_demo.launch* file, we can easily perform SLAM and build a map of the environment with a robot equipped with laser range finder sensors or RGB-D cameras.
2. **turtlebot_teleop** [26]: With the *keyboard_teleop.launch* file, we can manually control a robot using keyboard commands.
3. **turtlebot_rviz_launchers** [27]: With the *view_navigation.launch* file, we can load a preconfigured *rviz* workspace. We will save a lot of time by launching this file, because it will automatically load the robot model, trajectories, and perform mapping.
4. **turtlebot_gazebo** [28]: With the *turtlebot_world.launch* file we can deploy a turtlebot in a gazebo environment by linking the world file to it.

B. Setup Custom Packages and Directories

Inside the *catkin_ws/src* directory, we will create the following packages and directories in addition to the above official ROS packages.

1. **World:** Inside this directory, we will store the gazebo world file and the map generated from SLAM.
2. **ShellScripts:** Inside this directory, we will store the shell scripts.
3. **RvizConfig:** Inside this directory, we will store the customized *rviz* configuration files.
4. **wall_follower:** In this package we will store a node (*wall_follower.cpp*) that will autonomously drive the robot around to perform SLAM. Inside the *catkin_ws/src* directory create package with *catkin_create_pkg wall_follower*.
5. **pick_objects:** In this package we will write a node (*pick_objects.cpp*) that commands the robot to drive to the pickup and drop off zones. Inside the *catkin_ws/src* directory create package with *catkin_create_pkg pick_objects*.
6. **add_markers:** In this package we will write a node (*add_markers.cpp*) that model the object with a marker in *rviz*. Inside the *catkin_ws/src* directory create package *catkin_create_pkg add_markers*.

C. Create Gazebo World Using Building Editor

The steps below can be used to successfully design our environment with the Building Editor in Gazebo:

1. Open a terminal and launch *Gazebo*
2. Click edit and launch the *Building Editor*
3. Design a simple environment
4. Apply textures or colors
5. Save the *Building Editor* environment and go back to *Gazebo*
6. Save the Gazebo environment to the *World* directory of the *catkin_ws/src*.

For this project we have chosen the sketch in Fig. 33 as our world.

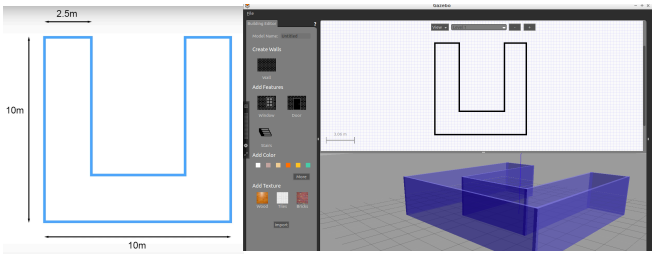

Fig. 33: An U-shaped layout (left) is used to create the Gazebo world (right).

D. The Package Tree

After all of the above steps have been performed the high level overview of the *catkin_ws/src* directory should look like:

```
├── # Official ROS packages
|
├── slam_gmapping          # gmapping_demo.launch file
|   ├── gmapping
|   ├── ...
├── turtlebot              # keyboard_teleop.launch file
```

```
|    ├── turtlebot_teleop
|    ├── ...
├── turtlebot_interactions  # view_navigation.launch file
|    ├── turtlebot_rviz_launchers
|    ├── ...
├── turtlebot_simulator     # turtlebot_world.launch file
|    ├── turtlebot_gazebo
|    ├── ...
├──  # Custom packages and direcotries
|
├── World                   # world files
|    ├── ...
├── ShellScripts            # shell scripts files
|    ├── ...
├── RvizConfig              # rviz configuration files
|    ├── ...
├── wall_follower           # wall_follower C++ node
|    ├── src/wall_follower.cpp
|    ├── ...
├── pick_objects            # pick_objects C++ node
|    ├── src/pick_objects.cpp
|    ├── ...
├── add_markers             # add_marker C++ node
|    ├── src/add_markers.cpp
|    ├── ...
└──
```

E. *Testing SLAM*

The next task is to autonomously map the environment we designed with the *Building Editor* in Gazebo. But before we tackle autonomous mapping, it's important to test if we are able to manually perform SLAM by teleoperating our robot. The goal of this step is to manually test SLAM. Write a shell script *test_slam.sh* that will deploy a turtlebot inside our environment, control it with keyboard commands, interface it with a SLAM package, and visualize the map in rviz. We will be using turtlebot for this project but other personalized robot models can be used as well. To *manually* test SLAM, create a *test_slam.sh* shell script that launches the following files:

1. The *turtlebot_world.launch* file to deploy a *turtlebot* in the gazebo environment.
2. The *gmapping_demo.launch* or run *slam_gmapping* to perform SLAM
3. The *view_navigation.launch* to observe the map in *rviz*
4. The *keyboard_teleop.launch* to manually control the robot with keyboard commands

```
#!/bin/sh
export TURTLEBOT_3D_SENSOR=kinect
xterm -e " roslaunch turtlebot_gazebo turtlebot_world.launch
          world_file:=/home/robond/catkin_ws/src/World/MyWorld.world" &
sleep 5
xterm -e " roslaunch turtlebot_navigation gmapping_demo.launch " &
sleep 5
xterm -e " roslaunch turtlebot_rviz_launchers view_navigation.launch " &
sleep 5
xterm -e " roslaunch turtlebot_teleop keyboard_teleop.launch "
```

Fig. 34: *test_slam.sh* shell script for testing SLAM.

Now we can launch our *test_slam.sh* file (Fig. 34), search for the xterminal running the *keyboard_teleop* node, and start controlling our robot. We are not required to fully map our environment in this step but only to make sure everything is working fine.

F. *Testing Wall Follower Autonomous SLAM*

Now that we have manually performed SLAM, it's time to automate the process and let our robot follow the walls and autonomously map the environment while avoiding obstacles i.e. simultaneous planning, localization and mapping (SPLAM). To do so, we'll get rid of the keyboard teleop node and instead interface the robot with a *wall_follower* node. A wall follower algorithm is a common algorithm that solves mazes. This algorithm is also known as the left-hand rule algorithm or the right-hand rule algorithm depending on which is our priority.

Here's the wall follower algorithm(the left-hand one) at a high level:

If left is free:

  Turn Left

Else if left is occupied and straight is free:

  Go Straight

Else if left and straight are occupied:

  Turn Right

Else if left/right/straight are occupied or you crashed:

  Turn 180 degrees

This algorithm has a lot of disadvantages because of the restricted space it can operate in. In other words, this algorithm will fail in open or infinitely large environments. Usually, the best algorithms for autonomous mapping are the ones that go in pursuit of undiscovered areas or unknown grid cells.

Let's follow the instructions below to autonomously map our environment:

1. Create a *wall_follower* package
2. Create a *wall_follower* C++ node
3. Edit the *wall_follower* C++ node name and change it to *wall_follower*
4. Edit the *wall_follower* C++ subscriber and publisher topics name
5. Write a *wall_follower.sh* shell script that launch the *turtlebot_world.launch*, *gmapping_demo.launch*, *view_navigation.launch*, and the *wall_follower* node

6. Edit the *CMakeLists.txt* file and add directories, executable, and target link libraries (Fig. 35)
7. Build the *catkin_ws* workspace with *catkin_make*
8. Run *wall_follower.sh* shell script (Fig. 36) to autonomously map the environment
9. Once we are satisfied with the map, kill the *wall_follower* terminal and save the map in both *pgm* and *yaml* formats in the World directory of the *catkin_ws/src*. The command *rosrun map_server map_saver -f myMap* will generate both *myMap.pgm* and *myMap.yaml* files.

```
cmake_minimum_required(VERSION 2.8.3)
project(wall_follower)

find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
)

catkin_package(
#  INCLUDE_DIRS include
#  LIBRARIES wall_follower
#  CATKIN_DEPENDS other_catkin_pkg
#  DEPENDS system_lib
)

include_directories(
 include
 ${catkin_INCLUDE_DIRS}
)

add_executable(wall_follower src/wall_follower.cpp)
target_link_libraries(wall_follower
  ${catkin_LIBRARIES}
)
```

Fig. 35: The CMakeList for the *wall_follower* package.

```
#!/bin/sh
export TURTLEBOT_3D_SENSOR=kinect
xterm -e " roslaunch turtlebot_gazebo turtlebot_world.launch
          world_file:=/home/robond/catkin_ws/src/World/MyWorld.world" &
sleep 5
xterm -e " roslaunch turtlebot_gazebo gmapping_demo.launch " &
sleep 5
xterm -e " roslaunch turtlebot_rviz_launchers view_navigation.launch " &
sleep 5
xterm -e " rosrun wall_follower wall_follower "
```

Fig. 36: *wall_follower.sh* shell script for testing autonomous planning and SLAM (or SPLAM) while crawling the wall.

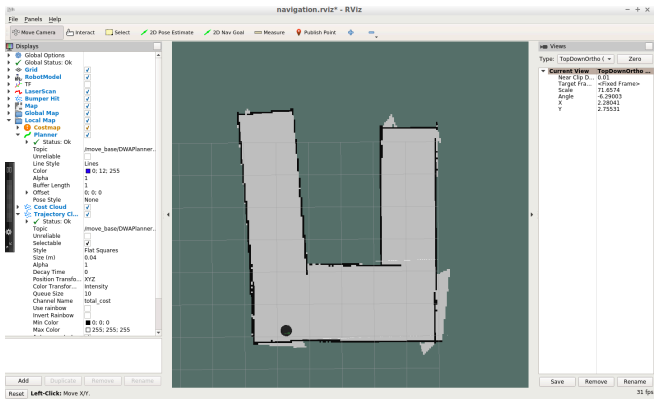

Fig. 37: Initial mapping formed without oprimization to the gmapping package.

The wall follower can only solve mazes with connected walls, where the robot is guaranteed to reach the exit of the maze after traversing close to walls. We will implement this basic algorithm in our environment to travel close to the walls and autonomously map it.

The *wall_follower.cpp* C++ node subscribes to the laser measurements of the robot and process them. To do this, the robot is placed in front of different blocks, then read back the measurements. This way the robot identifies its left side, right side, and forward side as well as distinguish between objects and free spaces. After processing the measurements, the robot is sent to the corresponding direction by publishing driving commands to actuate its wheels. This code implements the left-hand wall follower algorithm described earlier, but it's a bit more optimized to avoid obstacles.

Notice that the map shown in Fig. 37 is not 100% accurate, but still resembles the environment. That's because the gmapping parameter values used were the default values. In general, it's essential to tune them in order to get a 100% accurate map. These parameters are all listed under the gmapping documentation [25], We need to optimize these parameter values, in order to get better maps. Some of the optimizations can be performed in the file *slam_gmapping/gmapping/slam_gmapping_pr2.launch* using following steps. The resulting map is as shown in Fig. 38

1. Reducing the *angularUpdate* and *linearUpdate* values so the map gets updated for smaller ranges of movements.

   <param name="linearUpdate" value="0.2"/>

   <param name="angularUpdate" value="0.2"/>

2. Reducing the *x and y limits*, which represent the initial map size.

   <param name="xmin" value="-10.0"/>

   <param name="ymin" value="-10.0"/>

   <param name="xmax" value="10.0"/>

   <param name="ymax" value="10.0"/>

3. Increasing the *number of particles*.

   <param name="particles" value="100"/>

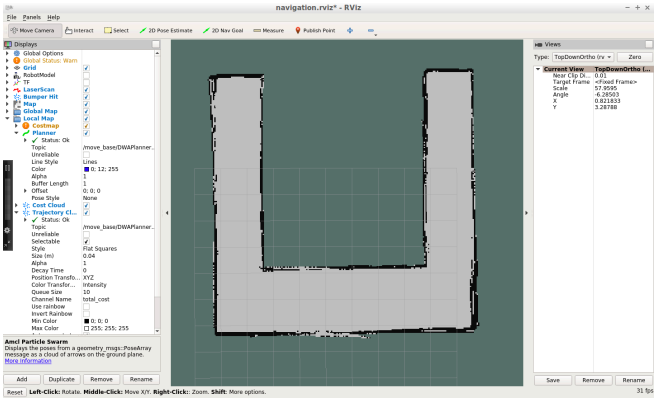

Fig. 38: Map of the environment created after oprimization to the gmapping parameters.

G. Testing Navigation

The next task is to pick two different goals and test our robot's ability to reach them and orient itself with respect to them. We'll refer to these goals as the pickup and drop off zones. This section is only for testing purposes to make sure our robot is able to reach these positions before autonomously commanding it to travel towards them.

We will be using the ROS Navigation stack, which is based on the **Dijkstra's**, a variant of the *Uniform Cost*

Search algorithm, to plan our robot trajectory from start to goal position. The ROS navigation stack [29] permits our robot to avoid any obstacle on its path by re-planning a new trajectory once our robot encounters them. We are familiar with this navigation stack from the localization project where we interfaced with it and sent a specific goal for our robot to reach while localizing itself with AMCL [30]. The ROS navigation algorithm can be modified and some instructions on how to write a global path planner as a plugin in ROS can be found in ROS wiki [31]. Write a *test_navigation.sh* shell script that launches the following files:

1. Add *turtlebot_world.launch* to deploy a *turtlebot* in the gazebo environment
2. Add *amcl_demo.launch* to localize the *turtlebot*
3. Add *view_navigation.launch* to observe the map in *rviz*

```
#!/bin/sh

xterm -e " roslaunch turtlebot_gazebo turtlebot_world.launch
          world_file:=/home/robond/catkin_ws/src/World/MyWorld.world" &
sleep 5
xterm -e " roslaunch turtlebot_gazebo amcl_demo.launch
          map_file:=/home/robond/catkin_ws/src/World/myMap.yaml
          3d_sensor:=kinect" &
sleep 5
xterm -e " roslaunch turtlebot_rviz_launchers view_navigation.launch "
```

Fig. 39: *test_navigation.sh* shell script for testing AMCL localization.

Once we launch all the nodes by launching *test_navigation.sh* (Fig. 39), we will initially see the particles around our robot, which means that AMCL recognizes the initial robot pose. Now, we can manually point out to two different goals (by using the *2D Nav Goal* in *rviz*), one at a time, and direct our robot to reach them and orient itself with respect to them.

H. Testing Autonomous Navigation

So far we tested the robot's capabilities in reaching multiple goals by manually commanding it to travel with the *2D NAV Goal* arrow in rviz. Now, we will write a node that will communicate with the ROS navigation stack and autonomously send successive goals for the robot to reach. As mentioned earlier, the ROS navigation stack creates a path for the robot based on Dijkstra's algorithm, a variant of the Uniform Cost Search algorithm, while avoiding obstacles on its path. The official ROS tutorial that describes how to send a single goal position and orientation to the navigation stack can be used as a reference [32]. This code was also used in the Localization project [30] where it was used to send the robot to a pre-defined goal. Let's follow the instructions below to autonomously command the robot to travel to both desired pickup and drop off zones:

1. Create a *pick_objects* package with *move_base_msgs*, *actionlib*, and *roscpp* dependencies
2. Create a *pick_objects.cpp* C++ node
3. Edit the C++ node and modify its node name in ros::init() to *pick_objects* as shown in Fig. 40
4. Then, edit the frame_id to *map* (Fig. 41), since our fixed frame is the *map* and not *base_link*
5. After that, we will need to modify the code and include a pick-up goal position and orientation for our robot to reach as shown in Fig. 41
6. Modify the C++ node and publish a second (drop-off) goal for the robot to reach as shown in Fig. 42
7. Display messages to track if robot successfully reached both zones
8. Pause 5 seconds after reaching the pick-up zone
9. Edit the *CMakeLists.txt* file and add directories, executable, and target link libraries as shown in Fig. 43.
10. Build the *catkin_ws* workspace with *catkin_make*
11. Create a *pick_objects.sh* script file (Fig. 44) that launches the *turtlebot*, *AMCL*, *rviz* and the *pick_objects* node.

```
#include <ros/ros.h>
#include <move_base_msgs/MoveBaseAction.h>
#include <actionlib/client/simple_action_client.h>

// Define a client to send goal requests to move_base server through SimpleActionClient
typedef actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction> MoveBaseClient;

int main(int argc, char** argv){
  // Initialize the simple_navigation_goals node
  ros::init(argc, argv, "pick_objects");

  //tell the action client that we want to spin a thread by default
  MoveBaseClient ac("move_base", true);

  // Wait 5 sec for move_base action server to come up
  while(!ac.waitForServer(ros::Duration(5.0))){
    ROS_INFO("Waiting for the move_base action server to come up");
  }
```

Fig. 40: Define node name in ros::init().

```
  move_base_msgs::MoveBaseGoal goal;

  // set up the frame parameters
  goal.target_pose.header.frame_id = "map";
  goal.target_pose.header.stamp = ros::Time::now();

  // Define a position and orientation for the robot to reach
  goal.target_pose.pose.position.x = 4.7;
  goal.target_pose.pose.position.y = 3.5;
  goal.target_pose.pose.orientation.w = 1.0;

   // Send the goal position and orientation for the robot to reach
  ROS_INFO("Sending pick-up goal...");
  ac.sendGoal(goal);
  ROS_INFO("Navigating towards pick-up location...");

  // Wait an infinite time for the results
  ac.waitForResult();

  // Check if the robot reached its goal
  if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED) {
    ROS_INFO("<<---Robot reached pick-up--->>");
    // Wait for 5 seconds
    ros::Duration(5.0).sleep();
```

Fig. 41: Define map as the fixed frame_id and set the goal position and orientation for pick-up location.

```
  // Check if the robot reached its goal
  if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED) {
    ROS_INFO("<<---Robot reached pick-up--->>");
    // Wait for 5 seconds
    ros::Duration(5.0).sleep();

    // Define a second goal
    goal.target_pose.pose.position.x = 2.8;
    goal.target_pose.pose.position.y = -0.1;
    goal.target_pose.pose.orientation.w = -1.0;

    ROS_INFO("Sending drop-off goal...");
    ac.sendGoal(goal);
    ROS_INFO("Navigating towards drop-off location...");

    // Wait an infinite time for the results
    ac.waitForResult();
    // Check if the robot reached its goal
    if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED)
      ROS_INFO("<<---Robot reached drop-off--->>");
    else
      ROS_INFO("Robot failed to reach drop-off goal.");

    ros::Duration(5.0).sleep();
  }
  else
    ROS_INFO("Robot failed to reach pick-up goal.");
  return 0;
}
```

Fig. 42: Set the drop-off goal position and orientation once the robot arrives at the pick-up location.

```
cmake_minimum_required(VERSION 2.8.3)
project(pick_objects)

find_package(catkin REQUIRED COMPONENTS
  actionlib
  move_base_msgs
  roscpp
)

catkin_package(
#  INCLUDE_DIRS include
#  LIBRARIES pick_objects
#  CATKIN_DEPENDS other_catkin_pkg
#  DEPENDS system_lib
)

include_directories(
# include
 ${catkin_INCLUDE_DIRS}
)

add_executable(pick_objects src/pick_objects.cpp)

target_link_libraries(pick_objects
  ${catkin_LIBRARIES}
)
```

Fig. 43: The CMakeList for the *pick_objects* package.

```
#!/bin/sh

xterm -e " roslaunch turtlebot_gazebo turtlebot_world.launch
           world_file:=/home/robond/catkin_ws/src/World/MyWorld.world" &
sleep 5
xterm -e " roslaunch turtlebot_gazebo amcl_demo.launch
           map_file:=/home/robond/catkin_ws/src/World/myMap.yaml
           3d_sensor:=kinect" &
sleep 5
xterm -e " roslaunch turtlebot_rviz_launchers view_navigation.launch " &
sleep 5
xterm -e " rosrun pick_objects pick_objects "
```

Fig. 44: *pick_objects.sh* shell script for reaching provided goal locations.

I. Modeling Virtual Objects

Now we need to model a virtual object with markers in rviz. The virtual object is the one being picked and delivered by the robot, thus it should first appear in its pickup zone, and then in its drop off zone once the robot reaches it.

The markers can be drawn on rviz window using the instructions in the ROS wiki [33] which includes a C++ node capable of drawing basic shapes like arrows, cubes, cylinders, and spheres in rviz. We can define a marker, scale it, define its position and orientation, and finally publish it to rviz. We will create the *object_markers* node and publish a single shape (a cube) following the steps below:

1. Publish the marker at the pickup zone
2. Pause 5 seconds
3. Hide the marker
4. Pause 5 seconds
5. Publish the marker at the drop off zone

We will combine this *object_markers* node later with the *pick_objects* node to simulate the full home service robot. The add_markers.cpp has been inspired by [34]. Let's follow the steps below to create a virtual object in rviz:

1. Create an *add_markers* package with *roscpp* and *visualization_msgs* dependencies
2. Create an *add_markers.cpp* C++ node
3. Edit the C++ node and modify its node name in ros::init() to *add_markers* as shown in Fig. 45
4. Then, edit the frame_id to *map* (Fig. 45), since our fixed frame is the *map*.
5. Modify the C++ code to publish a single shape as described earlier
6. Edit the *CMakeLists.txt* file and add the *executable*, and *libraries* (Fig. 46)
7. Build the *catkin_ws* workspace with *catkin_make*
8. Create an *add_markers.sh* shell script (Fig. 45) that launches the *turtlebot*, *AMCL*, *rviz*, and the *add_markers* node.
9. Launch the shell script and manually add a *Marker* in *rviz*

```
cmake_minimum_required(VERSION 2.8.3)
project(add_markers)

find_package(catkin REQUIRED COMPONENTS
  roscpp
  visualization_msgs
)

catkin_package(
#  INCLUDE_DIRS include
#  LIBRARIES add_markers
#  CATKIN_DEPENDS roscpp visualization_msgs
#  DEPENDS system_lib
)

include_directories(
# include
  ${catkin_INCLUDE_DIRS}
)

add_executable(add_markers src/add_markers.cpp)

target_link_libraries(add_markers
  ${catkin_LIBRARIES}
)
```

Fig. 44: The CMakeList for the *add_markers* package.

```
#!/bin/sh

xterm -e " roslaunch turtlebot_gazebo turtlebot_world.launch
           world_file:=/home/robond/catkin_ws/src/World/MyWorld.world" &
sleep 5
xterm -e " roslaunch turtlebot_gazebo amcl_demo.launch
           map_file:=/home/robond/catkin_ws/src/World/myMap.yaml
           3d_sensor:=kinect" &
sleep 5
xterm -e " roslaunch turtlebot_rviz_launchers view_navigation.launch " &
sleep 5
xterm -e " rosrun add_markers add_markers "
```

Fig. 45: The *add_markers.sh* shell script for reaching provided goal locations and showing the objects as markers.

J. Putting it all Together

Finally, we will simulate a full home service robot capable of navigating to pick up and deliver virtual objects. To do so, the *add_markers* and *pick_objects* nodes need to communicate. Essentially the *add_markers* node should subscribe to the robot *odometry* to keep track of the robot *pose*. Let's follow the steps below to successfully simulate a home service robot:

1. Edit the *add_markers* node and subscribe to odometry values
2. Modify the C++ node as described earlier

3. Build the *catkin_ws* workspace with *catkin_make*
4. Add markers to the *view_navigation.launch* file and save it as a new *rviz* configuration
5. Create a *home_service.sh* file (Fig. 46) that launches the *turtlebot*, *AMCL*, *rviz* config file, *pick_objects* and *add_markers* nodes

```
#!/bin/sh

xterm -e " roslaunch turtlebot_gazebo turtlebot_world.launch
           world_file:=/home/robond/catkin_ws/src/World/MyWorld.world" &
sleep 5
xterm -e " roslaunch turtlebot_gazebo amcl_demo.launch
           map_file:=/home/robond/catkin_ws/src/World/myMap.yaml
           3d_sensor:=kinect" &
sleep 5
xterm -e " rosrun rviz rviz -d /home/robond/catkin_ws/src/RvizConfig/home_service.rviz " &
sleep 5
xterm -e " rosrun add_markers add_markers " &
sleep 5
xterm -e " rosrun pick_objects pick_objects "
```

Fig. 46: *home_service.sh* shell script for a home service robot autonomously navigating to a pickup location, picking-up the marker object and navigating to the drop-off goal location.

Modify the *add_markers.cpp* node as follows in order to successfully navigate the robot toward the pick-up and drop-off locations as shown in Fig. 47:

1. Initially show the marker at the pick-up zone
2. Hide the marker once the robot reaches the pick-up zone
3. Wait 5 seconds to simulate a pick-up
4. Show the marker at the drop-off zone once the robot reaches it

```
  // Set the Marker pose and orientation
  marker.pose.position.x = PICK_POS_X;
  marker.pose.position.y = PICK_POS_Y;
  marker.pose.orientation.w = PICK_POS_W;

  marker.lifetime = ros::Duration();
  marker.action = visualization_msgs::Marker::ADD;

  if (_PICKING_UP)
  {
    marker.action = visualization_msgs::Marker::DELETE;
    ros::Duration(1.0).sleep();
    ROS_WARN_ONCE("<<---Robot picked-up object!--->>");
  }
  if (_DROPPING_OFF)
  {
    marker.pose.position.x = DROP_POS_X;
    marker.pose.position.y = DROP_POS_Y;
    marker.pose.orientation.w = DROP_POS_W;
    marker.action = visualization_msgs::Marker::ADD;
    ros::Duration(1.0).sleep();
    ROS_WARN_ONCE("<<---Robot dropped-off object!--->>");
  }

  marker_pub.publish(marker);
}
```

Fig. 47: The marker is either added or deleted depending on whether the robot is navigating to the pick-up or drop-off location.

We need to define a threshold for the position if the robot's odometry values are noisy. There are many ways to solve this problem. To establish communications between the robot and the markers, one method already discussed is to let our *add_markers* node subscribe to our robot odometry and keep track of our robot pose as shown in Fig. 48 and 49. Other solutions to this problem might be to use ROS parameters, subscribe to the AMCL pose, or even to publish a new variable that indicates whether or not the robot is at the pickup or drop off zone.

```
int main( int argc, char** argv )
{
  ros::init(argc, argv, "add_markers");
  ros::NodeHandle n;
  ros::Rate r(1);

  ros::Publisher marker_pub = n.advertise<visualization_msgs::Marker>("visualization_marker", 1)
  ros::Subscriber odom_sub = n.subscribe("/odom", 1000, callback_odom);

  visualization_msgs::Marker marker;

  // Set the frame ID and timestamp.  See the TF tutorials for information on these.
  marker.header.frame_id = "map";
  marker.header.stamp = ros::Time::now();
```

Fig. 48: Subscribing to the robot odometry.

```
void callback_odom(const nav_msgs::Odometry::ConstPtr &msg)
{
  float robot_pos_x = msg->pose.pose.position.x;
  float robot_pos_y = msg->pose.pose.position.y;

  float pickup_dist = 0.0f;
  float dropoff_dist = 0.0f;

  if(!_PICKING_UP && !_DROPPING_OFF)
  {
    pickup_dist = pow( pow((PICK_POS_X - robot_pos_x),2) + pow((PICK_POS_Y - robot_pos_y),2), 0.
    ROS_INFO("Distance to pickup: ([%f])", pickup_dist);
    if( abs(pickup_dist) <= POS_THRESH )
    {
      ROS_WARN_ONCE("<<--- Robot reached pickup goal! --->>");
      _PICKING_UP = true;
      marker.action = visualization_msgs::Marker::DELETE;
    }
  }
  if(!_DROPPING_OFF && _PICKING_UP)
  {
    dropoff_dist = pow( pow((DROP_POS_X - robot_pos_x),2) + pow((DROP_POS_Y - robot_pos_y),2), 0
    ROS_INFO("Distance to dropoff: ([%f])", dropoff_dist);
    if( abs(dropoff_dist) <= POS_THRESH )
    {
      ROS_WARN_ONCE("<<--- Robot reached dropoff goal! --->");
      _PICKING_UP = false;
      _DROPPING_OFF = true;
    }
  }
```

Fig. 49: Odometry callback function where the robot is compareto the pick-up and drop-off locations to know how far the goal position is from the robot.

VI. Results and Discussions

The physical robot model used here was the ROS official turtlebot. The robot in the gazebo world is shown in Fig. 50. The turtlebot is equipped with a laser range finder (LiDAR) which is used for the navigation of t robot in the gazebo environment.

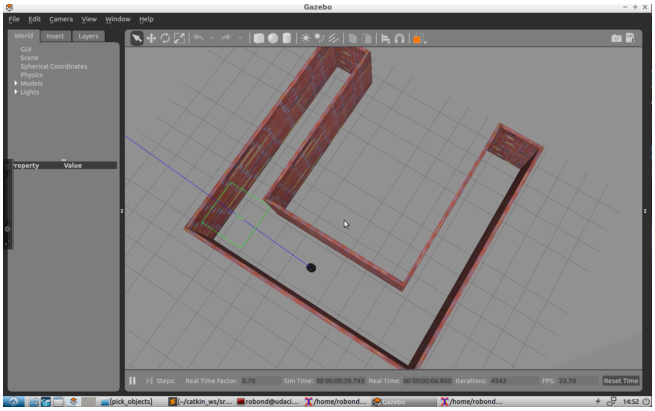

Fig. 50: The turtlebot in the Gazebo world.

Testing the navigation manually (using *test_navigation .sh*) in Rviz providing 2D Nav Goals in Rviz, we can provide mutiple goals one at a time and the robot reaches desired goals. Fig. 51 shows the robot localizing well inside the environment, using AMCL localization algorithm, while navigating around the virtual environment.

Then the autonomous navigation to pick-up and drop-off locations were performed using the pick_objects.sh shell script and some of the intermidiate results can be seen in Fig. 52 and Fig. 53.

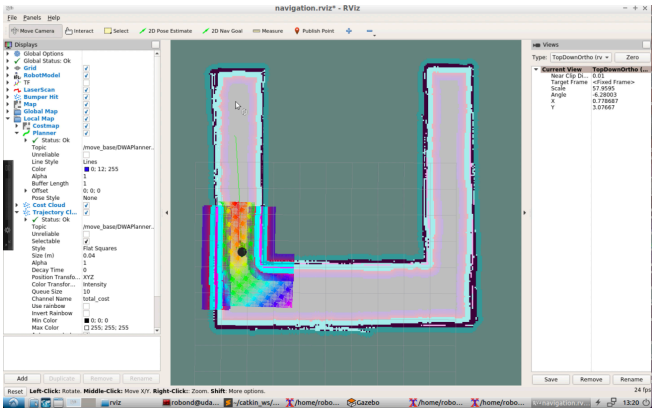

Fig. 51: Robot localization (using AMCL localization algorithm) while naviagting around the environment.

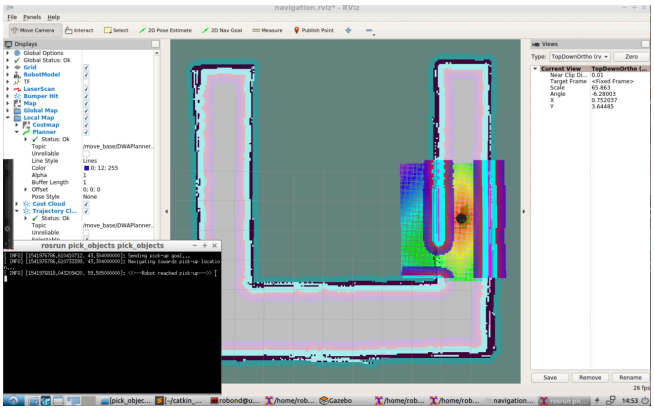

Fig. 52: The robot arrived at the pick-up location while autonomously navigating to the goal.

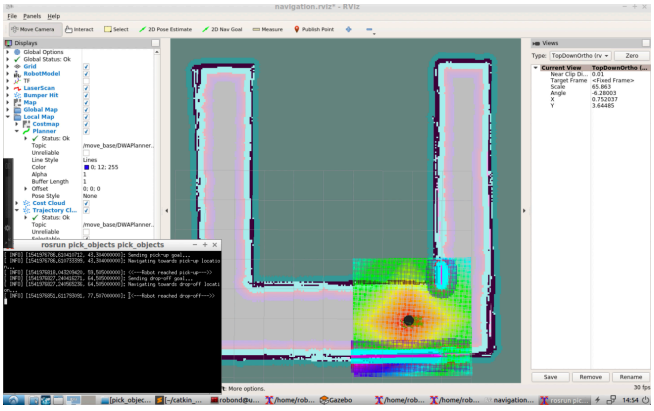

Fig. 53: The robot arrived at the drop-off location while autonomously navigating to the goal.

The ultimate goal was to create virtual objects or markers in rviz and place them at the pick-up and drop-off locations. The robot starts at the initial pose and the object at the pick-up location appears as shown in Fig. 54.

The virtual object appears at the pick-up location until the robot arrives at the pick-up goal as shown in Fig. 55, and the object disapers once the robot arrives at the pick-up goal position as shown in Fig. 56.

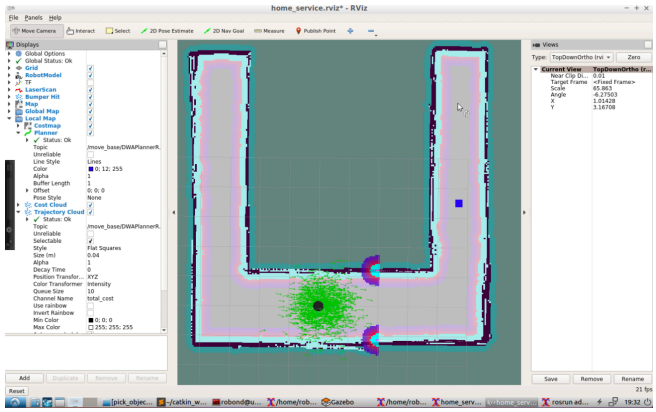

Fig. 54: The robot is in its initial pose and the pick-up location can be seen as the blue square object marker.

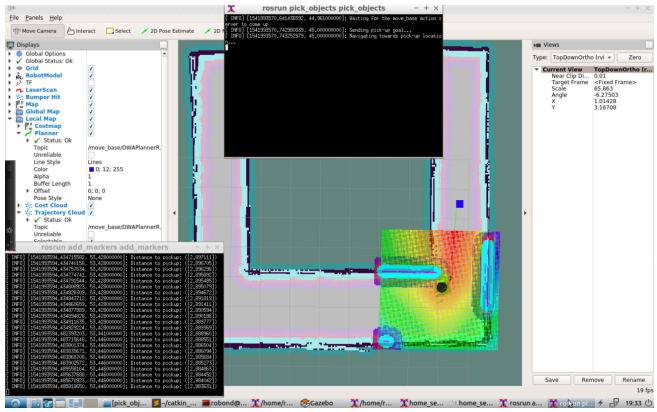

Fig. 55: The robot received the goal posiition and is navigating towards the marker.

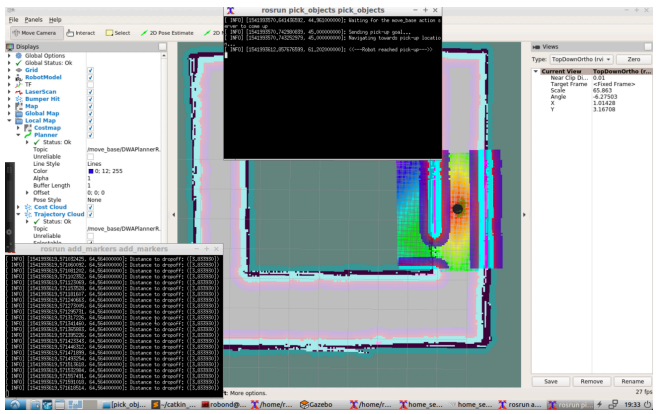

Fig. 56: The robot arrived at the pick-up location and the blue object disapeared to represent that the robot has pickedup the object.

Once the Robot has arrived at the pick-up location, the robot picks up the virtual object and must deliver it to the drop-off location. The robot then receives a drop-off goal location and navigates toward it as shown in Fig. 57. Once the robot has arrived near the goal position it must drop off thevirtual object it picked up previously. The blue marker in Fig. 58 represents the dropping-off of the virtual object at the drop-off location.

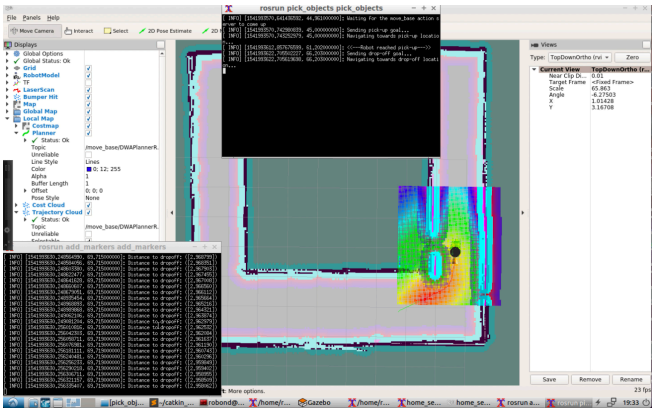

Fig. 57: The robot received the drop-off location and is navigating towards the drop-off goal.

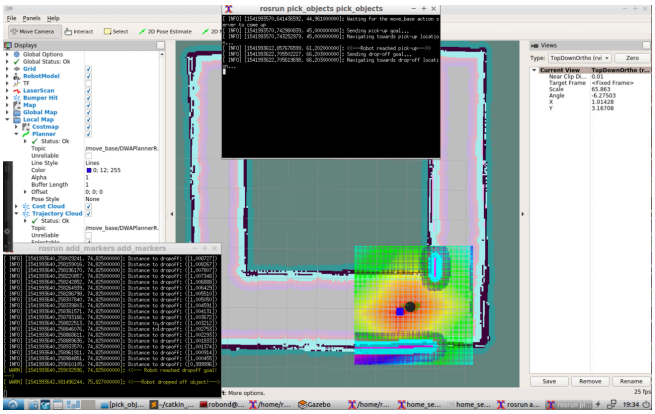

Fig. 58: The robot has arrived at the drop-off location and the blue object reappears representing the robot has dropped off the virtual blue object.

The robot was sucessful in autnomously mapping the environment (SPLAM), picking-up and dropping-off vistual objects while autonomously navigating towards the goal positions. The robot localized in the pripr-map using the AMCL localization algorithm and planned path using the Dijkstra algorithm.

VII. Conclusion

A custom gazebo environment was created in Gazebo using a floor plan. The ROS turtlebot robot model equipped with a LiDAR sensor was used for the project. Then the SLAM algorithm was performed using gmapping in order to map the unknown environment in the custom world while using wall follower algorithm to autonomously navigate in the closed environment, hence simultaneously planning, localizing, and mapping (SPLAM) in the environment. Various parameters were modified in order to create an accurate map of the environment.

A marker was created in rviz to represent a virtual object which the robot needed to naviagate to, “pick-up” the object and “deliver” to the the drop-off location. Using the custom packages for picking-up object and adding object markers were used in order to achive this goal. Ultimately, the robot was successful in autonomously planning a path to the multiple goal locatios and navigate while localizing itself in the map. The videos of the autonomous planning and navigation can be found in the GitHub videos folder [35].

Future work might include more complex environment for the robot to naviagte in using superior SLAM techniques such as graph-SLAM. Also, incorporating the entire package in a real robot to understand the real-world limitations and further optimization would be a natural next step.

Acknowledgment

The author would like to thank Udacity Inc. for providing fundamentals of the path planning and directions to create a SPLAM package.

References

[1] Kuffner, J.J. and LaValle, S.M., 2000. RRT-connect: An efficient approach to single-query path planning. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on* (Vol. 2, pp. 995-1001). IEEE.

[2] Stentz, A., 1994, May. Optimal and efficient path planning for partially-known environments. In *ICRA* (Vol. 94, pp. 3310-3317).

[3] Gammell, J.D., Srinivasa, S.S. and Barfoot, T.D., 2014. Informed RRT*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic. *arXiv preprint arXiv:1404.2334*.

[4] Kavraki, L., Svestka, P. and Overmars, M.H., 1994. *Probabilistic roadmaps for path planning in high-dimensional configuration spaces* (Vol. 1994). Unknown Publisher.

[5] Varadhan, G. and Manocha, D., 2004, October. Accurate Minkowski sum approximation of polyhedral models. In *Computer Graphics and Applications, 2004. PG 2004. Proceedings. 12th Pacific Conference on* (pp. 392-401). IEEE.

[6] https://www.toptal.com/game/video-game-physics-part-ii-collision-detection-for-solid-objects

[7] http://twistedoakstudios.com/blog/Post554_minkowski-sums-and-differences

[8] https://www.youtube.com/watch?v=SBFwgR4K1Gk

[9] http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

[10] http://theory.stanford.edu/~amitp/GameProgramming/Variations.html

[11] https://qiao.github.io/PathFinding.js/visual/

[12] http://www.frc.ri.cmu.edu/~axs/dynamic_plan.html

[13] Stentz, A., 1994, May. Optimal and efficient path planning for partially-known environments. In *ICRA* (Vol. 94, pp. 3310-3317).

[14] Stentz, A., 1995, August. The focussed D* algorithm for real-time replanning. In *IJCAI* (Vol. 95, pp. 1652-1659).

[15] http://aigamedev.com/open/tutorials/theta-star-any-angle-paths/

[16] Yap, P.K.Y., Burch, N., Holte, R.C. and Schaeffer, J., 2011, October. Any-Angle Path Planning for Computer Games. In *AIIDE*.

[17] Nash, A. and Koenig, S., 2013. Any-angle path planning. *AI Magazine*, *34*(4), pp.85-107.

[18] Geraerts, R. and Overmars, M.H., 2004. A comparative study of probabilistic roadmap planners. In *Algorithmic Foundations of Robotics V* (pp. 43-57). Springer, Berlin, Heidelberg.

[19] https://github.com/DrPanigrahi/RoboND-DeepRL-Project

[20] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G. and Petersen, S., 2015. Human-level control through deep reinforcement learning. *Nature*, *518*(7540), p.529.

[21] Mirowski, P., Pascanu, R., Viola, F., Soyer, H., Ballard, A.J., Banino, A., Denil, M., Goroshin, R., Sifre, L., Kavukcuoglu, K. and Kumaran, D., 2016. Learning to navigate in complex environments. *arXiv preprint arXiv:1611.03673*.

[22] Jaderberg, M., Mnih, V., Czarnecki, W.M., Schaul, T., Leibo, J.Z., Silver, D. and Kavukcuoglu, K., 2016. Reinforcement learning with unsupervised auxiliary tasks. *arXiv preprint arXiv:1611.05397*.

[23] Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D. and Kavukcuoglu, K., 2016, June. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning* (pp. 1928-1937).

[24] Arulkumaran, K., Deisenroth, M.P., Brundage, M. and Bharath, A.A., 2017. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866*.

[25] http://wiki.ros.org/gmapping

[26] http://wiki.ros.org/turtlebot_teleop

[27] http://wiki.ros.org/turtlebot_rviz_launchers

[28] http://wiki.ros.org/turtlebot_gazebo

[29] http://wiki.ros.org/navigation

[30] https://github.com/DrPanigrahi/RoboND-Localization-Project

[31] http://wiki.ros.org/navigation/Tutorials/Writing%20A%20Global%20Path%20Planner%20As%20Plugin%20in%20ROS

[32] http://wiki.ros.org/navigation/Tutorials/SendingSimpleGoals

[33] http://wiki.ros.org/rviz/Tutorials/Markers%3A%20Basic%20Shapes

[34] https://github.com/amiltonwong/RoboND-HomeServiceRobot-Project

[35] https://github.com/DrPanigrahi/RoboND-HomeServiceRobot-Project

[36] https://www.watermarquee.com