# Course 6 - Path Planning and Navigation

## Introduction to Path Planning and Navigation

### Lesson Overview

Video 1.1

**Summary**

In this lesson, we'll focus on the decision-making aspects of mobile robotics, i.e. path planning and navigation. **Path planning** is a strategic solution to the problem of "How do I get there?", while **obstacle avoidance** is a series of tactical decisions the robot must make as it moves along its path.

**Lesson Roadmap**

* Classic Path Planning
* Sample-Based and Probabilistic Path Planning
* Lab: Path Planning
* Project: Home Service Robot

### Applications

Before we dive into the details - let’s look at where path planning can be applied!

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 us. 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.

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.

A diagram of a science experiment

AI-generated content may be incorrect.

**Lesson Outcomes:**

* Understand real-world applications of path planning.
* Select the appropriate path planning algorithm for a given application.
* Be able to successfully implement path planning algorithms in code.
* Interface a path planning package with ROS

## Classic Path Planning

### Introduction to Path Planning

Video 2.1

**Outcomes**

* Recognize different types of path planning algorithms
* Comprehend the inner workings of a collection of algorithms
* Evaluate the suitability of an algorithm for a particular application
* Be ready to implement search algorithms in the upcoming lab

### Examples of Path Planning

Video 2.2

**Terminology**

**Complete** - An algorithm is complete if it is able to find a path between the start and the goal when one exists.

**Optimal** - An algorithm is optimal if it is able to find the best solution.

A screenshot of a quiz

AI-generated content may be incorrect.

A yellow and blue rectangle with blue text

AI-generated content may be incorrect.

In the above example, the robot would end up traversing the outer wall of the obstacle endlessly. There exist variants to the bug algorithm that will remedy this error, but the bulk of path planning algorithms rely on other principles that you will be introduced to throughout this lesson. In studying new algorithms, we will revisit the notion of Completeness and Optimality in analyzing the applicability of an algorithm to a task.

### Approaches to Path Planning

In this lesson, you will be studying 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 Planning**

Discrete 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 image below displays one possible implementation of discrete path planning applied to a 2-dimensional workspace.

A square and a yellow diamond on a white square with blue dots

AI-generated content may be incorrect.

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.

**Sample-Based Planning**

Sample-based path planning 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 below displays a graph representation of a 2-dimensional workspace created using sample-based planning.

A yellow and blue lines and dots

AI-generated content may be incorrect.

**Probabilistic Path Planning**

The last type of path planning that you will learn about in this module is probabilistic path 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 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 below displays probabilistic path planning applied to an environment containing a hazard (the lake at the top right).

A diagram of numbers and symbols

AI-generated content may be incorrect.

**Multi-Lesson Map**

In this lesson, you will be learning several discrete path planning algorithms and in the next lesson, you will move on to study sample-based and probabilistic planning. After you’ve become a wizard of path planning, you can apply your knowledge in a path planning lab, where you will code search algorithms in C++.

### Discrete Planning

Video 2.4

### Continuous Representation

Video 2.5

**Summary**

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 CFree*CFree*​ and CObs*CObs*​.

### Minkowski Sum

The Minkowski sum 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 your robot, with the robot’s origin as the tip of the paintbrush. The painted area is *Cobs*​. The image below shows just this.

A triangle with a purple line

AI-generated content may be incorrect.

To create the configuration space, the Minkowski sum is calculated in such a way for every obstacle in the workspace. The image 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.

A diagram of different shapes

AI-generated content may be incorrect.

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.

If you are interested in understanding the Minkowski Sum in more detail, then you may find the following resource helpful:

* [**An interesting read on how collisions are detected in video games**](https://www.toptal.com/game/video-game-physics-part-ii-collision-detection-for-solid-objects).

### Quiz: Minkowski Sum

A white square and purple triangle

AI-generated content may be incorrect.

Which of the following images represents the Configuration Space for the robot (purple) and obstacle (white) presented above?

A group of white squares

AI-generated content may be incorrect.

A white background with black text

AI-generated content may be incorrect.

### Minkowski Sum C++

Now that you've learned the **Minkowski Sum**, you'll get a chance to code it in C++!

Example

A graph with a red and blue triangle

AI-generated content may be incorrect.

In this example, you can see two triangles - a blue and a red one. Let's suppose the robot is represented by a blue triangle and the obstacle is represented by a red triangle. Your task is to compute the configuration space **C** of robot **A** and obstacle **B** in C++.

* **Robot**: Blue triangle denoted by A
* **Obstacle**: Red triangle denoted by B

A screenshot of a computer program

AI-generated content may be incorrect.

#### main.cpp

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Print 2D vectors coordinate values

void print2DVector(vector<vector<int> > vec)

{

     // Sorting the vector for grading purpose

    sort(vec.begin(), vec.end());

    for (int i = 0; i < vec.size(); ++i) {

        for (int j = 0; j < vec[0].size(); ++j) {

                cout << vec[i][j] << "  ";

        }

        cout << endl;

    }

}

// \*\*\*TODO: Check for duplicate coordinates inside a 2D vector and delete them\*\*\* //

vector<vector<int> > delete\_duplicate(vector<vector<int> > C)

{

}

// \*\*\*TODO: Compute the Minkowski Sum of two vectors\*\*\*//

vector<vector<int> > minkowski\_sum(vector<vector<int> > A, vector<vector<int> > B)

{

    C = delete\_duplicate(C);

    return C;

}

int main()

{

    // \*\*\*TODO: Define the coordinates of triangle A and B using 2D vectors\*\*\* //

    // Compute the minkowski sum of triangle A and B

    vector<vector<int> > C;

    C = minkowski\_sum(A, B);

    // Print the resulting vector

    print2DVector(C);

    return 0;

}

#### solution.cpp

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Print 2D vectors coordinate values

void print2DVector(vector<vector<int> > vec)

{

    // Sorting the vector for grading purpose

    sort(vec.begin(), vec.end());

    for (int i = 0; i < vec.size(); ++i) {

        for (int j = 0; j < vec[0].size(); ++j) {

                cout << vec[i][j] << "  ";

        }

        cout << endl;

    }

}

// Check for duplicate coordinates inside a 2D vector and delete them

vector<vector<int> > delete\_duplicate(vector<vector<int> > C)

{

    // Sort the C vector

    sort(C.begin(), C.end());

    // Initialize a non duplicated vector

    vector<vector<int> > Cn;

    for (int i = 0; i < C.size() - 1; i++) {

        // Check if it's a duplicate coordinate

        if (C[i] != C[i + 1]) {

            Cn.push\_back(C[i]);

        }

    }

    Cn.push\_back(C[C.size() - 1]);

    return Cn;

}

// Compute the Minkowski Sum of two vectors

vector<vector<int> > minkowski\_sum(vector<vector<int> > A, vector<vector<int> > B)

{

    vector<vector<int> > C;

    for (int i = 0; i < A.size(); i++) {

        for (int j = 0; j < B.size(); j++) {

            // Compute the current sum

            vector<int> Ci = { A[i][0] + B[j][0], A[i][1] + B[j][1] };

            // Push it to the C vector

            C.push\_back(Ci);

        }

    }

    C = delete\_duplicate(C);

    return C;

}

int main()

{

    // Define the coordinates of triangle A and B using 2D vectors

    vector<vector<int> > A(3, vector<int>(2));

    A = {{ 1, 0 }, { 0, 1 }, { 0, -1 },};

    vector<vector<int> > B(3, vector<int>(2));

    B = {{ 0, 0 }, { 1, 1 }, { 1, -1 },};

    // Compute the minkowski sum of triangle A and B

    vector<vector<int> > C;

    C = minkowski\_sum(A, B);

    // Print the resulting vector

    print2DVector(C);

    return 0;

}

### Translation and Rotation

Video 2.9

### 3D Configuration Space

As you saw, 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 the robot at 0°, and the second at 18°.

A colorful triangle with white text

AI-generated content may be incorrect.

A colorful paper with a triangle and a triangle

AI-generated content may be incorrect.

A three-dimensional configuration space can be generated by stacking two-dimensional configuration spaces as layers - as seen in the image below.

A diagram of a diagram of a stack of paper

AI-generated content may be incorrect.

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 below.

A diagram of a spiral structure

AI-generated content may be incorrect.

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 following video from the Freie Universität Berlin is a wonderful visualization of a 3D configuration space. The video will display different types of motion, and describe how certain robot motions map into the 3D configuration space.

[**Configuration Space Visualization**](https://www.youtube.com/watch?v=SBFwgR4K1Gk) - This is a *must* watch!

A screenshot of a question

AI-generated content may be incorrect.

### Discretization

Video 2.11

### Roadmap

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.

A map of a subway system

AI-generated content may be incorrect.

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 that you will learn next 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.

The following image illustrates a visibility graph for a configuration space containing polygonal obstacles.

A yellow and blue triangle with blue lines

AI-generated content may be incorrect.

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.

Once the Visibility Graph is built, a search algorithm can be applied to find the shortest path from Start to Goal. The image below displays the shortest path in this visibility graph.

A blue and orange triangle with blue dots

AI-generated content may be incorrect.

**Quiz**

Although the algorithms used to search the roadmap have not yet been introduced - it is still worth analysing whether *any* algorithm would be able to find a path from start to goal, and whether the optimal path lies within the roadmap.

A white background with black text

AI-generated content may be incorrect.

Having completed the quiz, you should have by now seen the advantages of the Visibility Graph method. 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. You can see a Voronoi Diagram for our configuration space in the graphic below.

A blue and orange background with triangles

AI-generated content may be incorrect.

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.

**Quiz**

Once again, it is worth investigating - will the roadmap built by the voronoi diagram contain a path from start to goal, and will it contain the optimal path.

A screenshot of a computer

AI-generated content may be incorrect.

### Cell Decomposition

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**

* Approximate 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. You can see the result of exact cell decomposition of a configuration space in the image below.

A graphic of triangles and lines

AI-generated content may be incorrect.

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 the image below.

A red lines with dots

AI-generated content may be incorrect.

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, 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 2-dimensional implementation of this method is called quadtree decomposition. It can be seen in the graphic below.

A screenshot of a computer

AI-generated content may be incorrect.

**Algorithm**

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

A screenshot of a cell phone

AI-generated content may be incorrect.

A screenshot of a game

AI-generated content may be incorrect.

The three dimensional equivalent of quadtrees are octrees, depicted in the image 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.

A diagram of different types of cell

AI-generated content may be incorrect.

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.

**Quiz**

**A screenshot of a question

AI-generated content may be incorrect.**

### Potential Field

Onto the last discretization method that you will be learning in this lesson - potential field method. 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 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.



Where x**x** represents the robot’s current position, and **x***goal*​ the goal position. *ν* is a scaling factor.

A fragment of the attractive potential field is displayed in the image below.

A green and blue grid

AI-generated content may be incorrect.

**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.

A math problem with numbers

AI-generated content may be incorrect.

Where the function *ρ*(**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 *ν* is a scaling parameter.

An image of a repulsive potential field for an arbitrary configuration space is provided below.

A blue and green grid with two mountains

AI-generated content may be incorrect.

The value *ρ*0​ controls how far from an obstacle the potential field will be non-zero, and how steep the area surrounding an obstacle will be.

Past *ρ*0​, the potential field is zero. Within a *ρ*0​ distance of the obstacle, the potential field scales with proximity to the obstacle.

**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 below shows the summation of the functions, and the image immediately after displays the final function.

A diagram of a graph

AI-generated content may be incorrect.

A diagram of a graph

AI-generated content may be incorrect.

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 images below depict one such instance. Depending on where the robot commences, it may be led to the bottom of the smile.

The image below depicts the configuration space, and the following image displays the corresponding potential field.

A yellow smiley face with blue background

AI-generated content may be incorrect.

A grid of green and yellow hills

AI-generated content may be incorrect.

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.

### Discretization Wrap-Up

Video 2.18

### Graph Search

Video 2.19

**Uninformed vs Informed Search**

Uninformed search algorithms are not provided with any information about the whereabouts of the goal, and thus search blindly. The only difference between different uninformed algorithms is the order in which they expand nodes. Several different types of uninformed algorithms are listed below:

* Breadth-first Search
* Depth-first Search
* Uniform Cost Search

Informed searches, on the other hand, are provided with information pertaining to the location of the goal. As a result, these search algorithms are able to evaluate some nodes to be more promising than others. This makes their search more efficient. The informed algorithm that you will be learning in this lesson is,

* A\* Search

Several variations on the above searches exist, and will be briefly discussed.

### Terminology

You are already familiar with two terms that can be used to describe an algorithm - completeness and optimality. However, there are a few others that you should know before starting to learn individual graph search algorithms.

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?

Keep these concepts in mind as you learn about each search algorithm. Let’s dive into the algorithms!

### Breadth-First Search

Video 2.21

A diagram of a network

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Video 2.21.1

### Depth-First Search

Video 2.22

A diagram of a tree

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Video 2.22.1

A screenshot of a computer

AI-generated content may be incorrect.

### Uniform Cost Search

Video 2.23

A diagram of a network

AI-generated content may be incorrect.

A screenshot of a question

AI-generated content may be incorrect.

Video 2.23.1

A screenshot of a questionnaire

AI-generated content may be incorrect.

### A\* Search

Video 2.24

A screenshot of a quiz

AI-generated content may be incorrect.

As you saw in the video above, A\ \*search orders the frontier using a priority queue, ordered by f(n), 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, as you may have discovered in the quiz - 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*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 neighbouring points (x1,x2,x3*x*1​,*x*2​,*x*3​), the heuristic value for x1*x*1​ to x3*x*3​ must be less than the sum of the heuristic values for x1*x*1​ to x2*x*2​ and x2*x*2​ to x3*x*3​.
* The heuristic function must be admissible. This means that h(n) must always be less than or equal to the true cost of reaching the goal from every node. In other words, h(n) must never overestimate the true path cost.

To understand where the admissibility clause comes from, take a look at the image below. Suppose you 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, you have four nodes on the frontier, but Node N would be expanded first because its h(n) 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(n) 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.

A diagram of a mathematical equation

AI-generated content may be incorrect.

As you saw in the image above, 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 you saw in the video), 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 you would need to determine a scaling parameter to be able to sum the two in a useful manner.

If you are interested in learning more about heuristics, visit [**Amit’s Heuristics Guide**](http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html) on Stanford’s website.

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 - some accommodate the use of A\* search in dynamic environments, while others help A\* become more manageable in large environments.

**Additional Resources**

The following visualization is a great tool that allows you to draw your own obstacles, set your own rules, and perform search using different algorithms.

[**Path Finding Visualization**](https://qiao.github.io/PathFinding.js/visual/)

For more information on A\* variants, take a look at:  
[**MovingAI A\* Variants**](http://movingai.com/astar-var.html)  
[**Variants of A\* - Stanford**](http://theory.stanford.edu/~amitp/GameProgramming/Variations.html)

Take some time to investigate the efficiency of A\ \*over BFS in different scenarios! And if you're feeling extra adventurous, research some of the other algorithms that are provided in the simulation and compare their results to those of BFS & A\*.

### Overall Concerns

**Overall Concerns Regarding Search**

**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. As you can see in the image below, the volume swept out by a unidirectional search is noticeably greater than the volume swept out by a bidirectional search for the same problem.

A diagram of a search

AI-generated content may be incorrect.

**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.

### Graph-Search Wrap-Up

Video 2.26

### Discrete Planning Wrap-Up

Video 2.27

## Lab: Path Planning

### Introduction

Welcome to the path planning lab! Here, you will get a chance to code two of the path planning algorithms that you’ve learned earlier.

You will first code the **Breadth-first search** algorithm in C++. This algorithm is divided into different coding quizzes. You will challenge yourself and code each one of them to finally generate the shortest path of a robot moving from start to goal.

Then, you will move on and make the necessary changes to code the **A star** algorithm. After coding both the BFS and A\* algorithms, you will visually compare the expansion lists generated. After a close inspection, you will judge which of these algorithms is more efficient.

Later in this lab, you will apply the A\* algorithm to a real-world problem. The real-world problem is nothing but the map that you’ve generated using the occupancy grid mapping algorithm.

Alright now, let’s get started!

A screenshot of a computer

AI-generated content may be incorrect.

### Modeling the Problem

The purpose of this lab is to find the shortest path for a robot moving inside a 5x6 map from start to goal position using different path planning algorithms. The robot can only move in four directions: up, left, down, and right. We will first model this problem using classes in C++ and later solve it with the BFS and A\* algorithms.

**Given**

**Grid(5x6)**:

0 1 0 0 0 0  
0 1 0 0 0 0  
0 1 0 0 0 0  
0 1 0 0 0 0  
0 0 0 1 1 0

Where 1's represent obstacles and 0's represent free space.

**Robot Start position:** 0,0

**Robot Goal Position:** 4,5

**Direction of Movement:** Up(-1,0) - left(0,-1) - down(1,0) - right(0,1)

The *Direction of Movement* vector is a collection of four different 2D vectors each of which enables you to move between grid cells in the map.

**Movement Arrows:** Up(^) - left(<) - down(v) - right(>)

The *Movement Arrows* vector store the robot actions, this vector will be used later in this lab to visualize the robot orientation at each grid cell on the shortest path.

**Cost of Movement:** 1

The *Cost of Movement* value indicates how much it cost to move from one cell to another. Here, the cost is equal for all possible movements.

A screenshot of a computer

AI-generated content may be incorrect.

**Note**

Throughout this lab, you'll be working with 1D and 2D vectors in C++. Vectors allow you to easily manage and manipulate the data with pre-built functions. For example: the pop\_back function can be used to delete the last element in the vector.

If you are not already familiar with vectors, I recommend you read the following two sources before you start coding:

1. [**2D Vectors**](https://www.geeksforgeeks.org/2d-vector-in-cpp-with-user-defined-size/): Learn how to define and use 2D Vectors in C++.
2. [**Documentation**](http://www.cplusplus.com/reference/vector/vector/): Learn the vectors iterators and modifiers.

#### main.cpp

#include <iostream>

#include <string.h>

#include <vector>

#include <algorithm>

using namespace std;

/\* TODO: Define a Map class

   Inside the map class, define the mapWidth, mapHeight and grid as a 2D vector

\*/

/\* TODO: Define a Planner class

   Inside the Planner class, define the start, goal, cost, movements, and movements\_arrows

   Note: The goal should be defined it terms of the mapWidth and mapHeight

\*/

/\* TODO: Define a print2DVector function which will print 2D vectors of any data type

   Example

   Input:

   vector<vector<int> > a{{ 1, 0 },{ 0, 1 }};

   print2DVector(a);

   vector<vector<string> > b{{ "a", "b" },{ "c", "d" }};

   print2DVector(b);

   Output:

   1 0

   0 1

   a b

   c d

   Hint: You need to use templates

\*/

/\*############ Don't modify the main function############\*/

int main()

{

    // Instantiate map and planner objects

    Map map;

    Planner planner;

    // Print classes variables

    cout << "Map:" << endl;

    print2DVector(map.grid);

    cout << "Start: " << planner.start[0] << " , " << planner.start[1] << endl;

    cout << "Goal: " << planner.goal[0] << " , " << planner.goal[1] << endl;

    cout << "Cost: " << planner.cost << endl;

    cout << "Robot Movements: " << planner.movements\_arrows[0] << " , " << planner.movements\_arrows[1] << " , " << planner.movements\_arrows[2] << " , " << planner.movements\_arrows[3] << endl;

    cout << "Delta:" << endl;

    print2DVector(planner.movements);

    return 0;

}

#### solution.cpp

#include <iostream>

#include <string.h>

#include <vector>

#include <algorithm>

using namespace std;

// Map class

class Map {

public:

    const static int mapWidth = 6;

    const static int mapHeight = 5;

    vector<vector<int> > grid = {

        { 0, 1, 0, 0, 0, 0 },

        { 0, 1, 0, 0, 0, 0 },

        { 0, 1, 0, 0, 0, 0 },

        { 0, 1, 0, 0, 0, 0 },

        { 0, 0, 0, 1, 1, 0 }

    };

};

// Planner class

class Planner : Map {

public:

    int start[2] = { 0, 0 };

    int goal[2] = { mapHeight - 1, mapWidth - 1 };

    int cost = 1;

    string movements\_arrows[4] = { "^", "<", "v", ">" };

    vector<vector<int> > movements{

        { -1, 0 },

        { 0, -1 },

        { 1, 0 },

        { 0, 1 }

    };

};

// Template function to print 2D vectors of any type

template <typename T>

void print2DVector(T Vec)

{

    for (int i = 0; i < Vec.size(); ++i) {

        for (int j = 0; j < Vec[0].size(); ++j) {

            cout << Vec[i][j] << ' ';

        }

        cout << endl;

    }

}

int main()

{

    // Instantiate map and planner objects

    Map map;

    Planner planner;

    // Print classes variables

    cout << "Map:" << endl;

    print2DVector(map.grid);

    cout << "Start: " << planner.start[0] << " , " << planner.start[1] << endl;

    cout << "Goal: " << planner.goal[0] << " , " << planner.goal[1] << endl;

    cout << "Cost: " << planner.cost << endl;

    cout << "Robot Movements: " << planner.movements\_arrows[0] << " , " << planner.movements\_arrows[1] << " , " << planner.movements\_arrows[2] << " , " << planner.movements\_arrows[3] << endl;

    cout << "Delta:" << endl;

    print2DVector(planner.movements);

    return 0;

}

### BFS: Expansion List

**Implementing BFS for Pathfinding**

**Objective** You will implement a **Breadth-First Search (BFS)** algorithm to find the shortest path from a starting point to a goal on a grid map. You will write a search function that expands cells one by one based on their cost (g) steps until the goal is reached.

**How BFS Works in Pathfinding?**

**What is BFS?** BFS explores all possible paths systematically, layer by layer, starting from the initial cell. It guarantees the shortest path is found if all moves have the same cost.

**How Do We Represent Cells?** Each cell is represented as a **triplet**: [g, x, y], where:

g: The cost to reach this cell (number of steps).

x: Row index of the cell.

y: Column index of the cell.

**Steps to Implement BFS**

1. **Initialization**:
   * Start by marking the starting position as visited and adding it to a list of cells to explore, represented as a triplet [g=0, x=start\_x, y=start\_y].
   * Ensure visited cells are tracked to avoid reprocessing the same cell.
2. **Processing Cells**:
   * Continuously pick the cell with the lowest cost (g) from the list.
   * Check if this cell is the goal:
     + If yes, print its triplet [g, x, y] and stop.
   * If not, expand the cell by examining its neighbors (up, down, left, right).
3. **Neighbor Expansion**:
   * Add valid neighboring cells to the list if they meet the following criteria:
     + They are within the grid boundaries.
     + They are not obstacles.
     + They have not been visited yet.
   * Update the cost for each neighbor based on the steps taken.
4. **Handling Roadblocks**:
   * If the list of cells to explore becomes empty before reaching the goal, print a message indicating the goal is unreachable.
   * This ensures that the algorithm actively detects and handles situations where no path exists.
5. **Decision-Making**:
   * Always expand the cell with the lowest cost first.
   * If two cells have the same cost, choose either one to continue the search.

A screenshot of a grid

AI-generated content may be incorrect.

A black and white grid with a black rectangle

AI-generated content may be incorrect.

**Expansion Steps**

1. **Expansion #0**
   * **Open List**: [0 0 0]
   * **Cell Picked**: [0 0 0]
2. **Expansion #1**
   * **Open List**: [1 1 0]
   * **Cell Picked**: [1 1 0]
3. **Expansion #2**
   * **Open List**: [2 2 0]
   * **Cell Picked**: [2 2 0]
4. **Expansion #3**
   * **Open List**: [3 3 0]
   * **Cell Picked**: [3 3 0]
5. **Expansion #4**
   * **Open List**: [4 4 0]
   * **Cell Picked**: [4 4 0]
6. **Expansion #5**
   * **Open List**: [5 4 1]
   * **Cell Picked**: [5 4 1]
7. **Expansion #6**
   * **Open List**: [6 4 2]
   * **Cell Picked**: [6 4 2]
8. **Expansion #7**
   * **Open List**: [7 3 2]
   * **Cell Picked**: [7 3 2]
9. **Expansion #8**
   * **Open List**: [8 3 3], [8 2 2]
   * **Cell Picked**: [8 2 2]
10. **Expansion #9**
    * **Open List**: [9 2 3], [9 1 2], [8 3 3]
    * **Cell Picked**: [8 3 3]
11. **Expansion #10**
    * **Open List**: [9 3 4], [9 2 3], [9 1 2]
    * **Cell Picked**: [9 1 2]
12. **Expansion #11**
    * **Open List**: [10 1 3], [10 0 2], [9 3 4], [9 2 3]
    * **Cell Picked**: [9 2 3]
13. **Expansion #12**
    * **Open List**: [10 2 4], [10 1 3], [10 0 2], [9 3 4]
    * **Cell Picked**: [9 3 4]
14. **Expansion #13**
    * **Open List**: [10 3 5], [10 2 4], [10 1 3], [10 0 2]
    * **Cell Picked**: [10 0 2]
15. **Expansion #14**
    * **Open List**: [11 0 3], [10 3 5], [10 2 4], [10 1 3]
    * **Cell Picked**: [10 1 3]
16. **Expansion #15**
    * **Open List**: [11 1 4], [11 0 3], [10 3 5], [10 2 4]
    * **Cell Picked**: [10 2 4]
17. **Expansion #16**
    * **Open List**: [11 2 5], [11 1 4], [11 0 3], [10 3 5]
    * **Cell Picked**: [10 3 5]
18. **Expansion #17**
    * **Open List**: [11 4 5], [11 2 5], [11 1 4], [11 0 3]
    * **Cell Picked**: [11 0 3]
19. **Expansion #18**
    * **Open List**: [12 0 4], [11 4 5], [11 2 5], [11 1 4]
    * **Cell Picked**: [11 1 4]
20. **Expansion #19**
    * **Open List**: [12 1 5], [12 0 4], [11 4 5], [11 2 5]
    * **Cell Picked**: [11 2 5]
21. **Expansion #20**
    * **Open List**: [12 1 5], [12 0 4], [11 4 5]
    * **Cell Picked**: [11 4 5]

#### main.cpp

#include <iostream>

#include <string.h>

#include <vector>

#include <algorithm>

using namespace std;

// Map class

class Map {

public:

    const static int mapWidth = 6;

    const static int mapHeight = 5;

    vector<vector<int> > grid = {

        { 0, 1, 0, 0, 0, 0 },

        { 0, 1, 0, 0, 0, 0 },

        { 0, 1, 0, 0, 0, 0 },

        { 0, 1, 0, 0, 0, 0 },

        { 0, 0, 0, 1, 1, 0 }

    };

};

// Planner class

class Planner : Map {

public:

    int start[2] = { 0, 0 };

    int goal[2] = { mapHeight - 1, mapWidth - 1 };

    int cost = 1;

    string movements\_arrows[4] = { "^", "<", "v", ">" };

    vector<vector<int> > movements{

        { -1, 0 },

        { 0, -1 },

        { 1, 0 },

        { 0, 1 }

    };

};

// Template function to print 2D vectors of any type

template <typename T>

void print2DVector(T Vec)

{

    for (int i = 0; i < Vec.size(); ++i) {

        for (int j = 0; j < Vec[0].size(); ++j) {

            cout << Vec[i][j] << ' ';

        }

        cout << endl;

    }

}

/\*#### TODO: Code the search function which will generate the expansion list ####\*/

// You are only required to print the final triplet values

void search(Map map, Planner planner)

{

}

int main()

{

    // Instantiate map and planner objects

    Map map;

    Planner planner;

    // Search for the expansions

    search(map, planner);

    return 0;

}

#### solution.cpp

#include <iostream>

#include <string.h>

#include <vector>

#include <algorithm>

using namespace std;

// Map class

class Map {

public:

    const static int mapWidth = 6;

    const static int mapHeight = 5;

    vector<vector<int> > grid = {

        { 0, 1, 0, 0, 0, 0 },

        { 0, 1, 0, 0, 0, 0 },

        { 0, 1, 0, 0, 0, 0 },

        { 0, 1, 0, 0, 0, 0 },

        { 0, 0, 0, 1, 1, 0 }

    };

};

// Planner class

class Planner : Map {

public:

    int start[2] = { 0, 0 };

    int goal[2] = { mapHeight - 1, mapWidth - 1 };

    int cost = 1;

    string movements\_arrows[4] = { "^", "<", "v", ">" };

    vector<vector<int> > movements{

        { -1, 0 },

        { 0, -1 },

        { 1, 0 },

        { 0, 1 }

    };

};

// Template function to print 2D vectors of any type

template <typename T>

void print2DVector(T Vec)

{

    for (int i = 0; i < Vec.size(); ++i) {

        for (int j = 0; j < Vec[0].size(); ++j) {

            cout << Vec[i][j] << ' ';

        }

        cout << endl;

    }

}

// Search function which generates the expansions

void search(Map map, Planner planner)

{

    // Create a closed 2 array filled with 0s and first element 1

    vector<vector<int> > closed(map.mapHeight, vector<int>(map.mapWidth));

    closed[planner.start[0]][planner.start[1]] = 1;

    // Defined the triplet values

    int x = planner.start[0];

    int y = planner.start[1];

    int g = 0;

    // Store the expansions

    vector<vector<int> > open;

    open.push\_back({ g, x, y });

    // Flags

    bool found = false;

    bool resign = false;

    int x2;

    int y2;

    // While I am still searching for the goal and the problem is solvable

    while (!found && !resign) {

        // Resign if no values in the open list and you can't expand anymore

        if (open.size() == 0) {

            resign = true;

            cout << "Failed to reach a goal" << endl;

        }

        // Keep expanding

        else {

            // Remove triplets from the open list

            sort(open.begin(), open.end());

            reverse(open.begin(), open.end());

            vector<int> next;

            // Stored the poped value into next

            next = open.back();

            open.pop\_back();

            x = next[1];

            y = next[2];

            g = next[0];

            // Check if we reached the goal:

            if (x == planner.goal[0] && y == planner.goal[1]) {

                found = true;

                cout << "[" << g << ", " << x << ", " << y << "]" << endl;

            }

            //else expand new elements

            else {

                for (int i = 0; i < planner.movements.size(); i++) {

                    x2 = x + planner.movements[i][0];

                    y2 = y + planner.movements[i][1];

                    if (x2 >= 0 && x2 < map.grid.size() && y2 >= 0 && y2 < map.grid[0].size()) {

                        if (closed[x2][y2] == 0 and map.grid[x2][y2] == 0) {

                            int g2 = g + planner.cost;

                            open.push\_back({ g2, x2, y2 });

                            closed[x2][y2] = 1;

                        }

                    }

                }

            }

        }

    }

}

int main()

{

    // Instantiate map and planner objects

    Map map;

    Planner planner;

    // Search for the expansions

    search(map, planner);

    return 0;

}

### BFS: Expansion Vector

Now that you have expanded the cells until you've reached the goal, you are asked to print the order in which each cell was expanded. For that, you’ll need to modify the search function and create a 2D **expansion** vector that is equal in size to the map. Each cell in the expansion vector will store the order at which it was expanded. Some of the cells were never expanded, and should show a value of **-1**.

**Hint**

Take a look at the expansion list generated after running the code:

A number with numbers on it

AI-generated content may be incorrect.

You can see that we started with the first cell and ended at the goal cell which was expanded after **20** iterations. All the obstacles and some cells were never expanded and thus are showing a value of **-1**.

Now, go ahead and modify the search function to generate and print the expansion 2D vector.

#### main.cpp

#include <iostream>

#include <string.h>

#include <vector>

#include <algorithm>

using namespace std;

// Map class

class Map {

public:

    const static int mapWidth = 6;

    const static int mapHeight = 5;

    vector<vector<int> > grid = {

        { 0, 1, 0, 0, 0, 0 },

        { 0, 1, 0, 0, 0, 0 },

        { 0, 1, 0, 0, 0, 0 },

        { 0, 1, 0, 0, 0, 0 },

        { 0, 0, 0, 1, 1, 0 }

    };

};

// Planner class

class Planner : Map {

public:

    int start[2] = { 0, 0 };

    int goal[2] = { mapHeight - 1, mapWidth - 1 };

    int cost = 1;

    string movements\_arrows[4] = { "^", "<", "v", ">" };

    vector<vector<int> > movements{

        { -1, 0 },

        { 0, -1 },

        { 1, 0 },

        { 0, 1 }

    };

};

// Template function to print 2D vectors of any type

template <typename T>

void print2DVector(T Vec)

{

    for (int i = 0; i < Vec.size(); ++i) {

        for (int j = 0; j < Vec[0].size(); ++j) {

            cout << Vec[i][j] << ' ';

        }

        cout << endl;

    }

}

/\* #### TODO: Modify the search function to generate and print the expansion 2D vector #### \*/

void search(Map map, Planner planner)

{

    // Create a closed 2 array filled with 0s and first element 1

    vector<vector<int> > closed(map.mapHeight, vector<int>(map.mapWidth));

    closed[planner.start[0]][planner.start[1]] = 1;

    // Defined the triplet values

    int x = planner.start[0];

    int y = planner.start[1];

    int g = 0;

    // Store the expansions

    vector<vector<int> > open;

    open.push\_back({ g, x, y });

    // Flags

    bool found = false;

    bool resign = false;

    int x2;

    int y2;

    // While I am still searching for the goal and the problem is solvable

    while (!found && !resign) {

        // Resign if no values in the open list and you can't expand anymore

        if (open.size() == 0) {

            resign = true;

            cout << "Failed to reach a goal" << endl;

        }

        // Keep expanding

        else {

            // Remove triplets from the open list

            sort(open.begin(), open.end());

            reverse(open.begin(), open.end());

            vector<int> next;

            // Stored the poped value into next

            next = open.back();

            open.pop\_back();

            x = next[1];

            y = next[2];

            g = next[0];

            // Check if we reached the goal:

            if (x == planner.goal[0] && y == planner.goal[1]) {

                found = true;

                cout << "[" << g << ", " << x << ", " << y << "]" << endl;

            }

            //else expand new elements

            else {

                for (int i = 0; i < planner.movements.size(); i++) {

                    x2 = x + planner.movements[i][0];

                    y2 = y + planner.movements[i][1];

                    if (x2 >= 0 && x2 < map.grid.size() && y2 >= 0 && y2 < map.grid[0].size()) {

                        if (closed[x2][y2] == 0 and map.grid[x2][y2] == 0) {

                            int g2 = g + planner.cost;

                            open.push\_back({ g2, x2, y2 });

                            closed[x2][y2] = 1;

                        }

                    }

                }

            }

        }

    }

}

int main()

{

    // Instantiate map and planner objects

    Map map;

    Planner planner;

    // Search for the expansions

    search(map, planner);

    return 0;

}

#### solution.cpp

#include <iostream>

#include <string.h>

#include <vector>

#include <algorithm>

using namespace std;

// Map class

class Map {

public:

    const static int mapWidth = 6;

    const static int mapHeight = 5;

    vector<vector<int> > grid = {

        { 0, 1, 0, 0, 0, 0 },

        { 0, 1, 0, 0, 0, 0 },

        { 0, 1, 0, 0, 0, 0 },

        { 0, 1, 0, 0, 0, 0 },

        { 0, 0, 0, 1, 1, 0 }

    };

};

// Planner class

class Planner : Map {

public:

    int start[2] = { 0, 0 };

    int goal[2] = { mapHeight - 1, mapWidth - 1 };

    int cost = 1;

    string movements\_arrows[4] = { "^", "<", "v", ">" };

    vector<vector<int> > movements{

        { -1, 0 },

        { 0, -1 },

        { 1, 0 },

        { 0, 1 }

    };

};

// Template function to print 2D vectors of any type

template <typename T>

void print2DVector(T Vec)

{

    for (int i = 0; i < Vec.size(); ++i) {

        for (int j = 0; j < Vec[0].size(); ++j) {

            cout << Vec[i][j] << ' ';

        }

        cout << endl;

    }

}

// Search function will generate the expansions

void search(Map map, Planner planner)

{

    // Create a closed 2 array filled with 0s and first element 1

    vector<vector<int> > closed(map.mapHeight, vector<int>(map.mapWidth));

    closed[planner.start[0]][planner.start[1]] = 1;

    // Create expand array filled with -1

    vector<vector<int> > expand(map.mapHeight, vector<int>(map.mapWidth, -1));

    // Defined the triplet values

    int x = planner.start[0];

    int y = planner.start[1];

    int g = 0;

    // Store the expansions

    vector<vector<int> > open;

    open.push\_back({ g, x, y });

    // Flags and counters

    bool found = false;

    bool resign = false;

    int count = 0;

    int x2;

    int y2;

    // While I am still searching for the goal and the problem is solvable

    while (!found && !resign) {

        // Resign if no values in the open list and you can't expand anymore

        if (open.size() == 0) {

            resign = true;

            cout << "Failed to reach a goal" << endl;

        }

        // Keep expanding

        else {

            // Remove triplets from the open list

            sort(open.begin(), open.end());

            reverse(open.begin(), open.end());

            vector<int> next;

            // Stored the poped value into next

            next = open.back();

            open.pop\_back();

            x = next[1];

            y = next[2];

            g = next[0];

            // Fill the expand vectors with count

            expand[x][y] = count;

            count += 1;

            // Check if we reached the goal:

            if (x == planner.goal[0] && y == planner.goal[1]) {

                found = true;

                //cout << "[" << g << ", " << x << ", " << y << "]" << endl;

            }

            //else expand new elements

            else {

                for (int i = 0; i < planner.movements.size(); i++) {

                    x2 = x + planner.movements[i][0];

                    y2 = y + planner.movements[i][1];

                    if (x2 >= 0 && x2 < map.grid.size() && y2 >= 0 && y2 < map.grid[0].size()) {

                        if (closed[x2][y2] == 0 and map.grid[x2][y2] == 0) {

                            int g2 = g + planner.cost;

                            open.push\_back({ g2, x2, y2 });

                            closed[x2][y2] = 1;

                        }

                    }

                }

            }

        }

    }

    // Print the expansion List

    print2DVector(expand);

}

int main()

{

    // Instantiate map and planner objects

    Map map;

    Planner planner;

    // Search for the expansions

    search(map, planner);

    return 0;

}

### BFS: Shortest Path

The final step is to print the shortest path that the robot has to take in order to travel from start to goal. You will need to record each action that the robot should take(ex: turning left **<**) and store all the actions in a **policy** 2D vector.

**Hint**

Here’s the output policy vector generated after running the code:

A line of different symbols

AI-generated content may be incorrect.

You can see the different actions(**v** - **>** - **<** - **^**) that the robot has to take in order to reach the goal marked with the \*\*\*\*\*. Some of these cells will never be visited by the robot and are marked with an **-**.

Now, go ahead and modify the search function to generate the policy 2D Vector.

#### main.cpp

#include <iostream>

#include <string.h>

#include <vector>

#include <algorithm>

using namespace std;

// Map class

class Map {

public:

    const static int mapWidth = 6;

    const static int mapHeight = 5;

    vector<vector<int> > grid = {

        { 0, 1, 0, 0, 0, 0 },

        { 0, 1, 0, 0, 0, 0 },

        { 0, 1, 0, 0, 0, 0 },

        { 0, 1, 0, 0, 0, 0 },

        { 0, 0, 0, 1, 1, 0 }

    };

};

// Planner class

class Planner : Map {

public:

    int start[2] = { 0, 0 };

    int goal[2] = { mapHeight - 1, mapWidth - 1 };

    int cost = 1;

    string movements\_arrows[4] = { "^", "<", "v", ">" };

    vector<vector<int> > movements{

        { -1, 0 },

        { 0, -1 },

        { 1, 0 },

        { 0, 1 }

    };

};

// Template function to print 2D vectors of any type

template <typename T>

void print2DVector(T Vec)

{

    for (int i = 0; i < Vec.size(); ++i) {

        for (int j = 0; j < Vec[0].size(); ++j) {

            cout << Vec[i][j] << ' ';

        }

        cout << endl;

    }

}

/\* #### TODO: Modify the search function and generate the policy vector #### \*/

void search(Map map, Planner planner)

{

    // Create a closed 2 array filled with 0s and first element 1

    vector<vector<int> > closed(map.mapHeight, vector<int>(map.mapWidth));

    closed[planner.start[0]][planner.start[1]] = 1;

    // Create expand array filled with -1

    vector<vector<int> > expand(map.mapHeight, vector<int>(map.mapWidth, -1));

    // Defined the triplet values

    int x = planner.start[0];

    int y = planner.start[1];

    int g = 0;

    // Store the expansions

    vector<vector<int> > open;

    open.push\_back({ g, x, y });

    // Flags and counters

    bool found = false;

    bool resign = false;

    int count = 0;

    int x2;

    int y2;

    // While I am still searching for the goal and the problem is solvable

    while (!found && !resign) {

        // Resign if no values in the open list and you can't expand anymore

        if (open.size() == 0) {

            resign = true;

            cout << "Failed to reach a goal" << endl;

        }

        // Keep expanding

        else {

            // Remove triplets from the open list

            sort(open.begin(), open.end());

            reverse(open.begin(), open.end());

            vector<int> next;

            // Stored the poped value into next

            next = open.back();

            open.pop\_back();

            x = next[1];

            y = next[2];

            g = next[0];

            // Fill the expand vectors with count

            expand[x][y] = count;

            count += 1;

            // Check if we reached the goal:

            if (x == planner.goal[0] && y == planner.goal[1]) {

                found = true;

                //cout << "[" << g << ", " << x << ", " << y << "]" << endl;

            }

            //else expand new elements

            else {

                for (int i = 0; i < planner.movements.size(); i++) {

                    x2 = x + planner.movements[i][0];

                    y2 = y + planner.movements[i][1];

                    if (x2 >= 0 && x2 < map.grid.size() && y2 >= 0 && y2 < map.grid[0].size()) {

                        if (closed[x2][y2] == 0 and map.grid[x2][y2] == 0) {

                            int g2 = g + planner.cost;

                            open.push\_back({ g2, x2, y2 });

                            closed[x2][y2] = 1;

                        }

                    }

                }

            }

        }

    }

    // Print the expansion List

    print2DVector(expand);

}

int main()

{

    // Instantiate map and planner objects

    Map map;

    Planner planner;

    // Search for the expansions

    search(map, planner);

    return 0;

}

#### solution.cpp

#include <iostream>

#include <string.h>

#include <vector>

#include <algorithm>

using namespace std;

// Map class

class Map {

public:

    const static int mapWidth = 6;

    const static int mapHeight = 5;

    vector<vector<int> > grid = {

        { 0, 1, 0, 0, 0, 0 },

        { 0, 1, 0, 0, 0, 0 },

        { 0, 1, 0, 0, 0, 0 },

        { 0, 1, 0, 0, 0, 0 },

        { 0, 0, 0, 1, 1, 0 }

    };

};

// Planner class

class Planner : Map {

public:

    int start[2] = { 0, 0 };

    int goal[2] = { mapHeight - 1, mapWidth - 1 };

    int cost = 1;

    string movements\_arrows[4] = { "^", "<", "v", ">" };

    vector<vector<int> > movements{

        { -1, 0 },

        { 0, -1 },

        { 1, 0 },

        { 0, 1 }

    };

};

// Template function to print 2D vectors of any type

template <typename T>

void print2DVector(T Vec)

{

    for (int i = 0; i < Vec.size(); ++i) {

        for (int j = 0; j < Vec[0].size(); ++j) {

            cout << Vec[i][j] << ' ';

        }

        cout << endl;

    }

}

// Search function will generate the expansions

void search(Map map, Planner planner)

{

    // Create a closed 2 array filled with 0s and first element 1

    vector<vector<int> > closed(map.mapHeight, vector<int>(map.mapWidth));

    closed[planner.start[0]][planner.start[1]] = 1;

    // Create expand array filled with -1

    vector<vector<int> > expand(map.mapHeight, vector<int>(map.mapWidth, -1));

    // Create action array filled with -1

    vector<vector<int> > action(map.mapHeight, vector<int>(map.mapWidth, -1));

    // Defined the triplet values

    int x = planner.start[0];

    int y = planner.start[1];

    int g = 0;

    // Store the expansions

    vector<vector<int> > open;

    open.push\_back({ g, x, y });

    // Flags and counters

    bool found = false;

    bool resign = false;

    int count = 0;

    int x2;

    int y2;

    // While I am still searching for the goal and the problem is solvable

    while (!found && !resign) {

        // Resign if no values in the open list and you can't expand anymore

        if (open.size() == 0) {

            resign = true;

            cout << "Failed to reach a goal" << endl;

        }

        // Keep expanding

        else {

            // Remove triplets from the open list

            sort(open.begin(), open.end());

            reverse(open.begin(), open.end());

            vector<int> next;

            // Stored the poped value into next

            next = open.back();

            open.pop\_back();

            x = next[1];

            y = next[2];

            g = next[0];

            // Fill the expand vectors with count

            expand[x][y] = count;

            count += 1;

            // Check if we reached the goal:

            if (x == planner.goal[0] && y == planner.goal[1]) {

                found = true;

                //cout << "[" << g << ", " << x << ", " << y << "]" << endl;

            }

            //else expand new elements

            else {

                for (int i = 0; i < planner.movements.size(); i++) {

                    x2 = x + planner.movements[i][0];

                    y2 = y + planner.movements[i][1];

                    if (x2 >= 0 && x2 < map.grid.size() && y2 >= 0 && y2 < map.grid[0].size()) {

                        if (closed[x2][y2] == 0 and map.grid[x2][y2] == 0) {

                            int g2 = g + planner.cost;

                            open.push\_back({ g2, x2, y2 });

                            closed[x2][y2] = 1;

                            action[x2][y2] = i;

                        }

                    }

                }

            }

        }

    }

    // Print the expansion List

    //print2DVector(expand);

    // Find the path with robot orientation

    vector<vector<string> > policy(map.mapHeight, vector<string>(map.mapWidth, "-"));

    // Going backward

    x = planner.goal[0];

    y = planner.goal[1];

    policy[x][y] = '\*';

    while (x != planner.start[0] or y != planner.start[1]) {

        x2 = x - planner.movements[action[x][y]][0];

        y2 = y - planner.movements[action[x][y]][1];

        policy[x2][y2] = planner.movements\_arrows[action[x][y]];

        x = x2;

        y = y2;

    }

    // Print the path with arrows

    print2DVector(policy);

}

int main()

{

    // Instantiate map and planner objects

    Map map;

    Planner planner;

    // Search for the expansions

    search(map, planner);

    return 0;

}

### A\*: Shortest Path

A screenshot of a math problem

AI-generated content may be incorrect.

A screenshot of a cell phone

AI-generated content may be incorrect.

A screenshot of a cell phone number

AI-generated content may be incorrect.

Expansion #: 0  
Open List: [9 0 0 0 ]  
Cell Picked: [9 0 0 0]

Expansion #: 1  
Open List: [9 1 1 0 ]  
Cell Picked: [9 1 1 0]

Expansion #: 2  
Open List: [9 2 2 0 ]  
Cell Picked: [9 2 2 0]

Expansion #: 3  
Open List: [9 3 3 0 ]  
Cell Picked: [9 3 3 0]

Expansion #: 4  
Open List: [9 4 4 0 ]  
Cell Picked: [9 4 4 0]

Expansion #: 5  
Open List: [9 5 4 1 ]  
Cell Picked: [9 5 4 1]

Expansion #: 6  
Open List: [9 6 4 2 ]  
Cell Picked: [9 6 4 2]

Expansion #: 7  
Open List: [11 7 3 2 ]  
Cell Picked: [11 7 3 2]

Expansion #: 8  
Open List: [13 8 2 2 ], [11 8 3 3 ]  
Cell Picked: [11 8 3 3]

Expansion #: 9  
Open List: [13 9 2 3 ], [13 8 2 2 ], [11 9 3 4 ]  
Cell Picked: [11 9 3 4]

Expansion #: 10  
Open List: [13 10 2 4 ], [13 9 2 3 ], [13 8 2 2 ], [11 10 3 5 ]  
Cell Picked: [11 10 3 5]

Expansion #: 11  
Open List: [13 11 2 5 ], [13 10 2 4 ], [13 9 2 3 ], [13 8 2 2 ], [11 11 4 5 ]  
Cell Picked: [11 11 4 5]

#### main.cpp

#include <iostream>

#include <string.h>

#include <vector>

#include <algorithm>

using namespace std;

// TODO: Add a Manhattan-based heuristic vector to the Map class

class Map {

public:

    const static int mapWidth = 6;

    const static int mapHeight = 5;

    vector<vector<int> > grid = {

        { 0, 1, 0, 0, 0, 0 },

        { 0, 1, 0, 0, 0, 0 },

        { 0, 1, 0, 0, 0, 0 },

        { 0, 1, 0, 0, 0, 0 },

        { 0, 0, 0, 1, 1, 0 }

    };

};

// Planner class

class Planner : Map {

public:

    int start[2] = { 0, 0 };

    int goal[2] = { mapHeight - 1, mapWidth - 1 };

    int cost = 1;

    string movements\_arrows[4] = { "^", "<", "v", ">" };

    vector<vector<int> > movements{

        { -1, 0 },

        { 0, -1 },

        { 1, 0 },

        { 0, 1 }

    };

};

// Template function to print 2D vectors of any type

template <typename T>

void print2DVector(T Vec)

{

    for (int i = 0; i < Vec.size(); ++i) {

        for (int j = 0; j < Vec[0].size(); ++j) {

            cout << Vec[i][j] << ' ';

        }

        cout << endl;

    }

}

/\* #### TODO: Modify the search function and implement the A\* algorithm #### \*/

void search(Map map, Planner planner)

{

    // Create a closed 2 array filled with 0s and first element 1

    vector<vector<int> > closed(map.mapHeight, vector<int>(map.mapWidth));

    closed[planner.start[0]][planner.start[1]] = 1;

    // Create expand array filled with -1

    vector<vector<int> > expand(map.mapHeight, vector<int>(map.mapWidth, -1));

    // Create action array filled with -1

    vector<vector<int> > action(map.mapHeight, vector<int>(map.mapWidth, -1));

    // Defined the triplet values

    int x = planner.start[0];

    int y = planner.start[1];

    int g = 0;

    // Store the expansions

    vector<vector<int> > open;

    open.push\_back({ g, x, y });

    // Flags and counters

    bool found = false;

    bool resign = false;

    int count = 0;

    int x2;

    int y2;

    // While I am still searching for the goal and the problem is solvable

    while (!found && !resign) {

        // Resign if no values in the open list and you can't expand anymore

        if (open.size() == 0) {

            resign = true;

            cout << "Failed to reach a goal" << endl;

        }

        // Keep expanding

        else {

            // Remove triplets from the open list

            sort(open.begin(), open.end());

            reverse(open.begin(), open.end());

            vector<int> next;

            // Stored the poped value into next

            next = open.back();

            open.pop\_back();

            x = next[1];

            y = next[2];

            g = next[0];

            // Fill the expand vectors with count

            expand[x][y] = count;

            count += 1;

            // Check if we reached the goal:

            if (x == planner.goal[0] && y == planner.goal[1]) {

                found = true;

                //cout << "[" << g << ", " << x << ", " << y << "]" << endl;

            }

            //else expand new elements

            else {

                for (int i = 0; i < planner.movements.size(); i++) {

                    x2 = x + planner.movements[i][0];

                    y2 = y + planner.movements[i][1];

                    if (x2 >= 0 && x2 < map.grid.size() && y2 >= 0 && y2 < map.grid[0].size()) {

                        if (closed[x2][y2] == 0 and map.grid[x2][y2] == 0) {

                            int g2 = g + planner.cost;

                            open.push\_back({ g2, x2, y2 });

                            closed[x2][y2] = 1;

                            action[x2][y2] = i;

                        }

                    }

                }

            }

        }

    }

    // Print the expansion List

    //print2DVector(expand);

    // Find the path with robot orientation

    vector<vector<string> > policy(map.mapHeight, vector<string>(map.mapWidth, "-"));

    // Going backward

    x = planner.goal[0];

    y = planner.goal[1];

    policy[x][y] = '\*';

    while (x != planner.start[0] or y != planner.start[1]) {

        x2 = x - planner.movements[action[x][y]][0];

        y2 = y - planner.movements[action[x][y]][1];

        policy[x2][y2] = planner.movements\_arrows[action[x][y]];

        x = x2;

        y = y2;

    }

    // Print the path with arrows

    print2DVector(policy);

}

int main()

{

    // Instantiate map and planner objects

    Map map;

    Planner planner;

    // Search for the expansions

    search(map, planner);

    return 0;

}

#### solution.cpp

#include <iostream>

#include <string.h>

#include <vector>

#include <algorithm>

using namespace std;

// Map class

class Map {

public:

    const static int mapWidth = 6;

    const static int mapHeight = 5;

    vector<vector<int> > grid = {

        { 0, 1, 0, 0, 0, 0 },

        { 0, 1, 0, 0, 0, 0 },

        { 0, 1, 0, 0, 0, 0 },

        { 0, 1, 0, 0, 0, 0 },

        { 0, 0, 0, 1, 1, 0 }

    };

    vector<vector<int> > heuristic = {

        { 9, 8, 7, 6, 5, 4 },

        { 8, 7, 6, 5, 4, 3 },

        { 7, 6, 5, 4, 3, 2 },

        { 6, 5, 4, 3, 2, 1 },

        { 5, 4, 3, 2, 1, 0 }

    };

};

// Planner class

class Planner : Map {

public:

    int start[2] = { 0, 0 };

    int goal[2] = { mapHeight - 1, mapWidth - 1 };

    int cost = 1;

    string movements\_arrows[4] = { "^", "<", "v", ">" };

    vector<vector<int> > movements{

        { -1, 0 },

        { 0, -1 },

        { 1, 0 },

        { 0, 1 }

    };

};

// Template function to print 2D vectors of any type

template <typename T>

void print2DVector(T Vec)

{

    for (int i = 0; i < Vec.size(); ++i) {

        for (int j = 0; j < Vec[0].size(); ++j) {

            cout << Vec[i][j] << ' ';

        }

        cout << endl;

    }

}

// Search function will generate the expansions

void search(Map map, Planner planner)

{

    // Create a closed 2 array filled with 0s and first element 1

    vector<vector<int> > closed(map.mapHeight, vector<int>(map.mapWidth));

    closed[planner.start[0]][planner.start[1]] = 1;

    // Create expand array filled with -1

    vector<vector<int> > expand(map.mapHeight, vector<int>(map.mapWidth, -1));

    // Create action array filled with -1

    vector<vector<int> > action(map.mapHeight, vector<int>(map.mapWidth, -1));

    // Defined the quadruplet values

    int x = planner.start[0];

    int y = planner.start[1];

    int g = 0;

    int f = g + map.heuristic[x][y];

    // Store the expansions

    vector<vector<int> > open;

    open.push\_back({ f, g, x, y });

    // Flags and Counts

    bool found = false;

    bool resign = false;

    int count = 0;

    int x2;

    int y2;

    // While I am still searching for the goal and the problem is solvable

    while (!found && !resign) {

        // Resign if no values in the open list and you can't expand anymore

        if (open.size() == 0) {

            resign = true;

            cout << "Failed to reach a goal" << endl;

        }

        // Keep expanding

        else {

            // Remove quadruplets from the open list

            sort(open.begin(), open.end());

            reverse(open.begin(), open.end());

            vector<int> next;

            // Stored the poped value into next

            next = open.back();

            open.pop\_back();

            x = next[2];

            y = next[3];

            g = next[1];

            // Fill the expand vectors with count

            expand[x][y] = count;

            count += 1;

            // Check if we reached the goal:

            if (x == planner.goal[0] && y == planner.goal[1]) {

                found = true;

                //cout << "[" << g << ", " << x << ", " << y << "]" << endl;

            }

            //else expand new elements

            else {

                for (int i = 0; i < planner.movements.size(); i++) {

                    x2 = x + planner.movements[i][0];

                    y2 = y + planner.movements[i][1];

                    if (x2 >= 0 && x2 < map.grid.size() && y2 >= 0 && y2 < map.grid[0].size()) {

                        if (closed[x2][y2] == 0 and map.grid[x2][y2] == 0) {

                            int g2 = g + planner.cost;

                            f = g2 + map.heuristic[x2][y2];

                            open.push\_back({ f, g2, x2, y2 });

                            closed[x2][y2] = 1;

                            action[x2][y2] = i;

                        }

                    }

                }

            }

        }

    }

    // Print the expansion List

    print2DVector(expand);

    // Find the path with robot orientation

    vector<vector<string> > policy(map.mapHeight, vector<string>(map.mapWidth, "-"));

    // Going backward

    x = planner.goal[0];

    y = planner.goal[1];

    policy[x][y] = '\*';

    while (x != planner.start[0] or y != planner.start[1]) {

        x2 = x - planner.movements[action[x][y]][0];

        y2 = y - planner.movements[action[x][y]][1];

        policy[x2][y2] = planner.movements\_arrows[action[x][y]];

        x = x2;

        y = y2;

    }

    // Print the robot path

    cout << endl;

    print2DVector(policy);

}

int main()

{

    // Instantiate a planner and map objects

    Map map;

    Planner planner;

    search(map, planner);

    return 0;

}

### Comparison

A group of numbers and symbols

AI-generated content may be incorrect.

### A\*: Real-World Map

Now it’s time to apply the A\* algorithm that you’ve coded earlier to a real-world map.

A red and green graph

AI-generated content may be incorrect.

If you recall, this map is the one you generated with the occupancy grid mapping algorithm using both sonar and odometry data. Our aim is now to find the shortest path for the robot to cross from start **o** to goal \*\*\*\*\* position.

* Given

**Map(300x150)**: The map data stored in the map.txt file in form of log odds values. As a reminder, here's how you should interpret these numbers:

* A cell is considered **unknown** if its log odds value is **equal to 0**.
* A cell is considered as **occupied** if its log odds is **larger than 0**.
* A cell is considered as **free** if its log odds value is **less than 0**.

**Grid(300x150)**: The log odds values converted to 0’s and 1’s where **0** represents the **free** space and **1** represents the **occupied** or **unknown** space.

**Robot Start position:** 230,145

**Robot Goal Position:** 60,50

**Direction of Movement:** Up(-1,0) - left(0,-1) - down(1,0) - right(0,1)

**Movement Arrows:** Up(^) - left(<) - down(v) - right(>)

**Cost of Movement:** 1

**Heuristic Vector:** Manhattan

If you scroll down to the code, you will notice that I added three new functions to the Map class. I coded a GetMap function which reads the map.txt log odds values and assign them the map variable. You will code the MapToGrid function in order to convert the log odds values to 0’s and 1’s. These 0 and 1 values will be assigned to the grid variable. And finally, the GeneratedHeuristic function is another function that you have to code in order to generate a Manhattan-based heuristic vector by computing the Manhattan distance of each cell with respect to the goal position. As a reminder the Manhattan distance of each cell can be calculated as follow:

A math equations on a white background

AI-generated content may be incorrect.

A screenshot of a map

AI-generated content may be incorrect.

Uses file map.txt

#### main.cpp

#include <iostream>

#include <math.h>

#include <vector>

#include <algorithm>

#include <fstream>

using namespace std;

// Map class

class Map {

public:

    const static int mapHeight = /\* #### TODO: mapHeight #### \*/

    const static int mapWidth = /\* #### TODO: mapWidth #### \*/

    vector<vector<double> > map = GetMap();

    vector<vector<int> > grid = MaptoGrid();

    vector<vector<int> > heuristic = GenerateHeuristic();

private:

    // Read the file and get the map

    vector<vector<double> > GetMap()

    {

        vector<vector<double> > mymap(mapHeight, vector<double>(mapWidth));

        ifstream myReadFile;

        myReadFile.open("map.txt");

        while (!myReadFile.eof()) {

            for (int i = 0; i < mapHeight; i++) {

                for (int j = 0; j < mapWidth; j++) {

                    myReadFile >> mymap[i][j];

                }

            }

        }

        return mymap;

    }

    /\* #### TODO: Code the MaptoGrid function and convert the map to 1's and 0's #### \*/

    vector<vector<int> > MaptoGrid()

    {

        vector<vector<int> > grid(mapHeight, vector<int>(mapWidth));

        /\* Here's how you interpret the data stored inside map

           0:unkown

          <0:free

          >0:occupied

        \*/

        /\* You need to convert these data to 0's and 1's and assigned it to grid where:

           0: Free Space

           1: Occupied + Unkown Space

        \*/

        return grid;

    }

    /\* #### TODO: Generate a Manhttan Heuristic Vector #### \*/

    vector<vector<int> > GenerateHeuristic()

    {

        int goal[2] = { 60, 50 };

        vector<vector<int> > heuristic(mapHeight, vector<int>(mapWidth));

        // Generate a Manhattan heursitic vector

        return heuristic;

    }

};

// Planner class

class Planner : Map {

public:

    int start[2] = /\* #### TODO: start #### \*/

    int goal[2] = /\* #### TODO: goal #### \*/

    int cost = 1;

    string movements\_arrows[4] = { "^", "<", "v", ">" };

    vector<vector<int> > movements{

        { -1, 0 },

        { 0, -1 },

        { 1, 0 },

        { 0, 1 }

    };

    vector<vector<int> > path;

};

// Printing vectors of any type

template <typename T>

void print2DVector(T Vec)

{

    for (int i = 0; i < Vec.size(); ++i) {

        for (int j = 0; j < Vec[0].size(); ++j) {

            cout << Vec[i][j] << ' ';

        }

        cout << endl;

    }

}

Planner search(Map map, Planner planner)

{

    // Create a closed 2 array filled with 0s and first element 1

    vector<vector<int> > closed(map.mapHeight, vector<int>(map.mapWidth));

    closed[planner.start[0]][planner.start[1]] = 1;

    // Create expand array filled with -1

    vector<vector<int> > expand(map.mapHeight, vector<int>(map.mapWidth, -1));

    // Create action array filled with -1

    vector<vector<int> > action(map.mapHeight, vector<int>(map.mapWidth, -1));

    // Defined the quadruplet values

    int x = planner.start[0];

    int y = planner.start[1];

    int g = 0;

    int f = g + map.heuristic[x][y];

    // Store the expansions

    vector<vector<int> > open;

    open.push\_back({ f, g, x, y });

    // Flags and Counts

    bool found = false;

    bool resign = false;

    int count = 0;

    int x2;

    int y2;

    // While I am still searching for the goal and the problem is solvable

    while (!found && !resign) {

        // Resign if no values in the open list and you can't expand anymore

        if (open.size() == 0) {

            resign = true;

            cout << "Failed to reach a goal" << endl;

        }

        // Keep expanding

        else {

            // Remove quadruplets from the open list

            sort(open.begin(), open.end());

            reverse(open.begin(), open.end());

            vector<int> next;

            // Stored the poped value into next

            next = open.back();

            open.pop\_back();

            x = next[2];

            y = next[3];

            g = next[1];

            // Fill the expand vectors with count

            expand[x][y] = count;

            count += 1;

            // Check if we reached the goal:

            if (x == planner.goal[0] && y == planner.goal[1]) {

                found = true;

                //cout << "[" << g << ", " << x << ", " << y << "]" << endl;

            }

            //else expand new elements

            else {

                for (int i = 0; i < planner.movements.size(); i++) {

                    x2 = x + planner.movements[i][0];

                    y2 = y + planner.movements[i][1];

                    if (x2 >= 0 && x2 < map.grid.size() && y2 >= 0 && y2 < map.grid[0].size()) {

                        if (closed[x2][y2] == 0 and map.grid[x2][y2] == 0) {

                            int g2 = g + planner.cost;

                            f = g2 + map.heuristic[x2][y2];

                            open.push\_back({ f, g2, x2, y2 });

                            closed[x2][y2] = 1;

                            action[x2][y2] = i;

                        }

                    }

                }

            }

        }

    }

    // Print the expansion List

    //print2DVector(expand);

    // Find the path with robot orientation

    vector<vector<string> > policy(map.mapHeight, vector<string>(map.mapWidth, "-"));

    // Going backward

    x = planner.goal[0];

    y = planner.goal[1];

    policy[x][y] = '\*';

    while (x != planner.start[0] or y != planner.start[1]) {

        x2 = x - planner.movements[action[x][y]][0];

        y2 = y - planner.movements[action[x][y]][1];

        // Store the  Path in a vector

        planner.path.push\_back({ x2, y2 });

        policy[x2][y2] = planner.movements\_arrows[action[x][y]];

        x = x2;

        y = y2;

    }

    // Print the robot path

    //cout << endl;

    //print2DVector(policy);

    return planner;

}

int main()

{

    // Instantiate a planner and map objects

    Map map;

    Planner planner;

    // Generate the shortest Path using the Astar algorithm

    planner = search(map, planner);

    return 0;

}

#### solution.cpp

#include <iostream>

#include <math.h>

#include <vector>

#include <algorithm>

#include <fstream>

using namespace std;

// Map class

class Map {

public:

    const static int mapHeight = 300;

    const static int mapWidth = 150;

    vector<vector<double> > map = GetMap();

    vector<vector<int> > grid = MaptoGrid();

    vector<vector<int> > heuristic = GenerateHeuristic();

private:

    // Read the file and get the map

    vector<vector<double> > GetMap()

    {

        vector<vector<double> > mymap(mapHeight, vector<double>(mapWidth));

        ifstream myReadFile;

        myReadFile.open("map.txt");

        while (!myReadFile.eof()) {

            for (int i = 0; i < mapHeight; i++) {

                for (int j = 0; j < mapWidth; j++) {

                    myReadFile >> mymap[i][j];

                }

            }

        }

        return mymap;

    }

    //Convert the map to 1's and 0's

    vector<vector<int> > MaptoGrid()

    {

        vector<vector<int> > grid(mapHeight, vector<int>(mapWidth));

        for (int x = 0; x < mapHeight; x++) {

            for (int y = 0; y < mapWidth; y++) {

                if (map[x][y] == 0) //unkown state

                    grid[x][y] = 1;

                else if (map[x][y] > 0) //Occupied state

                    grid[x][y] = 1;

                else //Free state

                    grid[x][y] = 0;

            }

        }

        return grid;

    }

    // Generate a Manhattan Heuristic Vector

    vector<vector<int> > GenerateHeuristic()

    {

        vector<vector<int> > heuristic(mapHeight, vector<int>(mapWidth));

        int goal[2] = { 60, 50 };

        for (int i = 0; i < heuristic.size(); i++) {

            for (int j = 0; j < heuristic[0].size(); j++) {

                int xd = goal[0] - i;

                int yd = goal[1] - j;

                // Manhattan Distance

                   int d = abs(xd) + abs(yd);

                // Euclidian Distance

                // double d = sqrt(xd \* xd + yd \* yd);

                // Chebyshev distance

                // int d = max(abs(xd), abs(yd));

                heuristic[i][j] = d;

            }

        }

        return heuristic;

    }

};

// Planner class

class Planner : Map {

public:

    int start[2] = { 230, 145 };

    int goal[2] = { 60, 50 };

    int cost = 1;

    string movements\_arrows[4] = { "^", "<", "v", ">" };

    vector<vector<int> > movements{

        { -1, 0 },

        { 0, -1 },

        { 1, 0 },

        { 0, 1 }

    };

    vector<vector<int> > path;

};

// Printing vectors of any type

template <typename T>

void print2DVector(T Vec)

{

    for (int i = 0; i < Vec.size(); ++i) {

        for (int j = 0; j < Vec[0].size(); ++j) {

            cout << Vec[i][j] << ' ';

        }

        cout << endl;

    }

}

Planner search(Map map, Planner planner)

{

    // Create a closed 2 array filled with 0s and first element 1

    vector<vector<int> > closed(map.mapHeight, vector<int>(map.mapWidth));

    closed[planner.start[0]][planner.start[1]] = 1;

    // Create expand array filled with -1

    vector<vector<int> > expand(map.mapHeight, vector<int>(map.mapWidth, -1));

    // Create action array filled with -1

    vector<vector<int> > action(map.mapHeight, vector<int>(map.mapWidth, -1));

    // Defined the quadruplet values

    int x = planner.start[0];

    int y = planner.start[1];

    int g = 0;

    int f = g + map.heuristic[x][y];

    // Store the expansions

    vector<vector<int> > open;

    open.push\_back({ f, g, x, y });

    // Flags and Counts

    bool found = false;

    bool resign = false;

    int count = 0;

    int x2;

    int y2;

    // While I am still searching for the goal and the problem is solvable

    while (!found && !resign) {

        // Resign if no values in the open list and you can't expand anymore

        if (open.size() == 0) {

            resign = true;

            cout << "Failed to reach a goal" << endl;

        }

        // Keep expanding

        else {

            // Remove quadruplets from the open list

            sort(open.begin(), open.end());

            reverse(open.begin(), open.end());

            vector<int> next;

            // Stored the poped value into next

            next = open.back();

            open.pop\_back();

            x = next[2];

            y = next[3];

            g = next[1];

            // Fill the expand vectors with count

            expand[x][y] = count;

            count += 1;

            // Check if we reached the goal:

            if (x == planner.goal[0] && y == planner.goal[1]) {

                found = true;

                //cout << "[" << g << ", " << x << ", " << y << "]" << endl;

            }

            //else expand new elements

            else {

                for (int i = 0; i < planner.movements.size(); i++) {

                    x2 = x + planner.movements[i][0];

                    y2 = y + planner.movements[i][1];

                    if (x2 >= 0 && x2 < map.grid.size() && y2 >= 0 && y2 < map.grid[0].size()) {

                        if (closed[x2][y2] == 0 and map.grid[x2][y2] == 0) {

                            int g2 = g + planner.cost;

                            f = g2 + map.heuristic[x2][y2];

                            open.push\_back({ f, g2, x2, y2 });

                            closed[x2][y2] = 1;

                            action[x2][y2] = i;

                        }

                    }

                }

            }

        }

    }

    // Print the expansion List

    //print2DVector(expand);

    // Find the path with robot orientation

    vector<vector<string> > policy(map.mapHeight, vector<string>(map.mapWidth, "-"));

    // Going backward

    x = planner.goal[0];

    y = planner.goal[1];

    policy[x][y] = '\*';

    while (x != planner.start[0] or y != planner.start[1]) {

        x2 = x - planner.movements[action[x][y]][0];

        y2 = y - planner.movements[action[x][y]][1];

        // Store the  Path in a vector

        planner.path.push\_back({ x2, y2 });

        policy[x2][y2] = planner.movements\_arrows[action[x][y]];

        x = x2;

        y = y2;

    }

    // Print the robot path

    //cout << endl;

    print2DVector(policy);

    return planner;

}

int main()

{

    // Instantiate a planner and map objects

    Map map;

    Planner planner;

    // Generate the shortest Path using the Astar algorithm

    planner = search(map, planner);

    return 0;

}

### A\*: Visualization

Ew

**Path Planning**

So far, you’ve generated the shortest path using the A\* algorithm, but it was really hard to see it. Now, you'll edit the visualization function that you previously coded and modify it to plot the shortest path.

**Udacity Workspace**

For this quiz, you'll use the [**Udacity Workspace**](https://classroom.udacity.com/nanodegrees/nd209/parts/dad7b7cc-9cce-4be4-876e-30935216c8fa/modules/451b7eed-6813-422a-a4d0-ce5db5ee1bca/lessons/411e2410-8f65-4764-a02a-e219ac36c776/concepts/fc59506b-6059-45a2-9d4d-204f7343988a?contentVersion=1.0.0&contentLocale=en-us). Move to the next concept, enable the GPU, Go to Desktop, and follow these instructions. Remember to disable the GPU once you are done generating the image.

A screenshot of a computer

AI-generated content may be incorrect.

void visualization(Map map, Planner planner)

{

    //Graph Format

    plt::title("Path");

    plt::xlim(0, map.mapHeight);

    plt::ylim(0, map.mapWidth);

    // Draw every grid of the map:

    for (double x = 0; x < map.mapHeight; x++) {

        cout << "Remaining Rows= " << map.mapHeight - x << endl;

        for (double y = 0; y < map.mapWidth; y++) {

            if (map.map[x][y] == 0) { //Green unkown state

                plt::plot({ x }, { y }, "g.");

            }

            else if (map.map[x][y] > 0) { //Black occupied state

                plt::plot({ x }, { y }, "k.");

            }

            else { //Red free state

                plt::plot({ x }, { y }, "r.");

            }

        }

    }

    // TODO: Plot start and end states in blue colors using o and \* respectively

    // TODO: Plot the robot path in blue color using a .

    //Save the image and close the plot

    plt::save("./Images/Path.png");

    plt::clf();

}

Here are some helpful commands you can use to generate plots with the matplotlib library:

* *Set Title*: plt::title("Your Title");
* *Set Limits*: plt::xlim(x-axis lower limit, x-axis upper limit );
* *Plot Data*:plt::plot({ x-value }, { y-value }, "Color and Shape");
* *Save Plot*: plt::save("File name and directory");
* *Close Plot*: plt::clf();

Check out this [**link**](https://github.com/lava/matplotlib-cpp) for more information on the matplotlib C++ library. For information regarding the plot color and shape refer to the LineSpec and LineColor section of the [**MATLAB**](https://www.mathworks.com/help/matlab/ref/plot.html?requestedDomain=true) documentation.

A screenshot of a program

AI-generated content may be incorrect.

A red and green diagram

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

### Udacity Workspace

* To follow along with the lab's instructions, use local VM image (Ubuntu 16.04 LTS) running on your VMWare/VirtualBox.
* Once you log into the VM image, open a Terminal window.
* You're now ready to follow along in your development environment with this lab!

## Sample-Based and Probabilistic Path Planning

### Introduction to Sample-Based & Probabilistic Path Planning

Video 4.1

### Why Sample-Based Planning

**Why Sample-Based 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.

**Increased Dimensionality**

For a 2-dimensional 8-connected space, every node has 8 successors (8-connected means that from every cell you 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 3n−13*n*−1.

It is not uncommon for robots and robotic systems to have large numbers of dimensions. Recall the robotic arm that you 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.

A cartoon of a robot

AI-generated content may be incorrect.

**Constrained Dynamics**

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 you can see in the image below.

A video game with a car driving down the road

AI-generated content may be incorrect.

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!)*

A screenshot of a video game

AI-generated content may be incorrect.

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, you will learn two different path planning algorithms, and understand how to tune their parameters for varying applications.

### Weakening Requirements

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 who’s 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.

### Sample-Based 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.

A yellow and blue logo

AI-generated content may be incorrect.

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

* Probabilistic Roadmap Method
* Rapidly Exploring Random Tree Method

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

### Probabilistic Roadmap (PRM)

Video 4.5

**Algorithm**

The pseudocode for the PRM learning phase is provided below.

**A screenshot of a computer program

AI-generated content may be incorrect.**

**Setting Parameters**

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**](https://xlinux.nist.gov/dads/HTML/kdtree.html) 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.

**Probabilistically Complete**

As discussed before, sample-based path planning algorithms are probabilistically complete. Now that you have seen one such algorithm in action, you 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.

**Variants**

The algorithm that you 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.

* [**A Comparative Study of Probabilistic Roadmap Planners**](http://www.staff.science.uu.nl/~gerae101/pdf/compare.pdf)
* PRM is a Multi-Query Planner

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.

A screenshot of a white box

AI-generated content may be incorrect.

### Rapidly Exploring Random Tree Method (RRT)

A screenshot of a computer program

AI-generated content may be incorrect.**Setting Parameters**

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, you can sample uniformly - which would favour wide unexplored spaces, or you 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*δ*. As RRT starts to generate random configurations, a large proportion of these configurations will lie further than a distance δ*δ* from the closest configuration in the graph. In such a situation, a randomly generated node will dictate the direction of growth, while δ*δ* is the growth rate.

Choosing a small δ*δ* will result in a large density of nodes and small growth rate. On the other hand, choosing a large δ*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*δ* must be chosen carefully, with knowledge of the environment and requirements of the solution.

**Single-Query Planner**

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.

**RRT & Non-holonomic Systems**

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.

A screenshot of a question

AI-generated content may be incorrect.

### Path Smoothing

Video 4.7

A screenshot of a computer program

AI-generated content may be incorrect.

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.

### Overall Concerns

**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.

A white rectangular object with dots and lines

AI-generated content may be incorrect.

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.

**Conclusion**

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!

### Sample-Based Planning Wrap-Up

Video 4.9

**Extended Reading**

At this point, you have the knowledge to read through a paper on path planning. The following paper, [**Path Planning for Non-Circular Micro Aerial Vehicles in Constrained Environments**](https://www.cs.cmu.edu/~maxim/files/pathplanforMAV_icra13.pdf), addresses the problem of path planning for a quadrotor.

It is an enjoyable read that culminates the past two sections of path planning, as it references a number of planning methods that you have learned, and introduces a present-day application of path planning. Reading the paper will help you gain an appreciation of this branch of robotics, as well as help you gain confidence in the subject.

Some additional definitions that you may find helpful while reading the paper:

* **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.

### Introduction to Probabilistic Path Planning

Video 4.10

**Note**: At 1:07 the instructor means to say "*Similar to the approach we use in a Reinforcement Learning algorithm*".

### Markov Decision Process

**Recycling Robot Example**

Let's say we have a recycling robot, as an example. The robot’s goal is to drive around its environment and pick up as many cans as possible. It has a set of **states** that it could be in, and a set of **actions** that it could take. The robot receives a **reward** for picking up cans; however, it can also receive a negative reward (a penalty) if it runs out of battery and get stranded.

The robot has 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 (*St*​=*high*). In response, the agent decides to search for cans (*At*​=*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.

**MDP Definition**

A Markov Decision Process is defined by:

* A set of states: SS,
* Initial state: s0*s*0​,
* A set of actions: AA,
* The transition model: *T*(*s*,*a*,*s*′),
* A set of rewards: RR.

The transition model is the probability of reaching a state s′*s*′ from a state s*s* by executing action a*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.

**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.

A graph of a function

AI-generated content may be incorrect.

As you can see, the intended action of moving forward one cell is only executed with a probability of 0.8 (80%). 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.

A screenshot of a game

AI-generated content may be incorrect.

**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.

If we are to path plan using MDPs, we might be able to get a better result!

**Probabilistic Path Planning Solution**

In each state (cell), the robot will receive a certain reward, R(s)*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.

The image below displays the environment with appropriate rewards assigned.

A screenshot of a game

AI-generated content may be incorrect.

As you 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. Read on to see how that’s done in probabilistic path planning!

### Policies

Solution to a Markov Decision Process is called a policy, and is denoted with the letter *π*.

**Definition**

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 *π*∗, informs the robot of the\* best\* action to take from any state, to maximize the overall reward. We’ll study optimal policies in more detail below.

It is highly recommended that you are familiar with the basics of Reinforcement Learning (RL), ie. what a policy is, how state-value is calculated, and how the Bellman equations can be used to compute the optimal policy. Go through these RL concepts to solve gridworld problem.

(Optional) Readings:

* [**Wikipedia - Reinforcement Learning**](https://en.wikipedia.org/wiki/Reinforcement_learning)
* [**Reinforcement Learning 101 - solve the gridworld state-value function**](https://towardsdatascience.com/reinforcement-learning-rl-101-with-python-e1aa0d37d43b)

**Developing a Policy**

The image below 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*.

A diagram of a mountain range

AI-generated content may be incorrect.

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 Rewards**

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 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.

A black text on a white background

AI-generated content may be incorrect.

All of the expected rewards are calculated in this way, taking into account the transition model for this particular robot.

You may have noticed that a few expected rewards are missing in the image above. Can you calculate their values?

**Expected Reward Quiz**

A screenshot of a phone

AI-generated content may be incorrect.

Hopefully, after completing the quizzes, you are more comfortable with how the expected rewards are calculated. The image below has all of the expected rewards filled in.

A diagram of a mountain range

AI-generated content may be incorrect.

**Selecting a Policy**

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 the image 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

A screenshot of a math problem

AI-generated content may be incorrect.

A math equation with black text

AI-generated content may be incorrect.

**Determining the Optimal Policy**

Recall that the **optimal policy**, denoted π∗*π*∗, informs the robot of the *best* action to take from any state, to maximize the overall reward. That is,

A black text on a white background

AI-generated content may be incorrect.

In a state s*s*, the optimal policy *π*∗ will choose the action a*a* that maximizes the utility of s*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 below displays the set of all actions once more.

A diagram of a mountain range

AI-generated content may be incorrect.

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 *Uπ*(*s*)=*R*(*s*)+*Uπ*(*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.

Now it's your turn!

**Quiz**

Can you calculate what would the utility of the state to the right of the center mountain be, if the most rewarding action is chosen?

A diagram of a graph

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

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.

**Applying the Policy**

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 that shows 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.

A screenshot of a video game

AI-generated content may be incorrect.

A diagram of numbers and symbols

AI-generated content may be incorrect.

**Discounting**

One simplification that you may have noticed us make, is omit the discounting rate γ*γ*. In the above example, γ=1*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:

A mathematical equation with numbers and symbols

AI-generated content may be incorrect.

### 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



A screenshot of a computer

AI-generated content may be incorrect.

### Probabilistic Path Planning Wrap-Up

Video 4.15

## Home Service Robot

### Overview

Video 5.1

A screenshot of a computer

AI-generated content may be incorrect.

### Working Environment

In this project, Udacity provides you a in Classroom Workspace with ROS set up for you in the [**Project Workspace concept**](https://classroom.udacity.com/nanodegrees/nd209/parts/75c8f42b-c844-4f61-b3c6-521956c5cf70/modules/de8554d1-78db-4d1d-9a78-9b9a93d2879e/lessons/1ccf2893-a07b-41c5-b2ed-7cdc48bd26fc/concepts/acd9e789-8460-48a7-ace8-9f6b2aea25c9). If you are not familiar with the Workspace, please review the [**Workspace Introduction lessons**](https://classroom.udacity.com/nanodegrees/nd209/parts/0778207d-f34a-4178-8ccf-9e06b5bd2203/modules/48156d08-abb1-4c03-a18d-9db738a0b92b/lessons/e0c61e8d-7eac-4807-8737-d2bd321ae7a2/concepts/47784838-aea6-4834-9ebb-79fbb3e135af?contentVersion=2.0.0&contentLocale=en-us)!

A computer screen with a logo

AI-generated content may be incorrect.

After you enter the Workspace Desktop, please upgrade the system using the command

sudo apt-get **update** && apt-get upgrade

### Project Workspace: Home Service Robot

A screenshot of a computer program

AI-generated content may be incorrect.

### Shell Scripts

A shell script is a file containing a series of commands and could be executed. It is commonly used to set up environment, run a program, etc.

You already know how to build a roslaunch file. It is very convenient to launch multiple ROS nodes and set parameters from a single roslaunch command. However, when developing robotic software with different packages, it might get harder to track errors and bugs generated from different nodes.

That's when shell scripts come in handy! After you create a shell script file to launch one or many nodes each in separate terminals, you will have the power to track the output of different nodes and keep the convenience of running a single command to launch all nodes.

**Your launch.sh Script**

Let us start by creating this launch.sh script in the Udacity Workspace. Its goal is to launch Gazebo and Rviz in separate instances of terminals. Note that we are using xterm terminal in the script here.

A white background with red text

AI-generated content may be incorrect.

The launch.sh shell script launches three terminals and issues one or multiple commands in each terminal. Let’s break down this script to understand the meaning of each line.

**Code Breakdown**

*#!/bin/sh*

This statement is called a shebang. It must be included in every shell script you write since it specifies the full path of the UNIX interpreter to execute it.

*xterm -e " gazebo " &*

With the xterm -e statement, we launch a new instance of an xterminal. Inside this terminal, we launch gazebo using the command "gazebo". Then we add an ampersand & to indicate that another instance of an xterm terminal will be created in a separate statement.

*sleep 5*

We pause this script for 5 seconds using sleep.

*xterm -e " source /opt/ros/kinetic/setup.bash; roscore" &*

We launch a second instance of the xterm terminal. Inside this terminal, we source the ROS workspace and launch the ROS master node.

*sleep 5*

We pause this script for another 5 seconds.

*xterm -e " rosrun rviz rviz"*

We are launching a third instance of the xterm terminal, and running rviz.

Save your script file and give it execute pemission by chmod +x launch.sh. Then launch the shell script with ./launch.sh.

After launching this script, we’ll have three open xterm terminals, and we will be able to track any errors or bugs that occur. To recap, this script will open the first terminal and launch gazebo. Then it will pause for 5 seconds and open a second terminal to launch the ROS master. It will pause for another 5 seconds and, finally, open a third terminal to launch RVIZ.

Try to launch your script in the Workspace and verify its functions!

A screenshot of a computer script

AI-generated content may be incorrect.

### Simulation Set Up

**Catkin Workspace**

To program your home service robot, you will need to interface it with different ROS packages. Some of these packages are **official ROS packages** which offer great tools and others are **packages that you’ll create**. The goal of this section is to prepare and build your catkin workspace.

Here’s the list of the official ROS packages that you will need to grab, and other packages and directories that you’ll need to create at a later stage as you go through the project. Your catkin\_ws/src directory should look as follows:

**Official ROS packages**

Import these packages now and install them in the src directory of your catkin workspace. Be sure to clone the full GitHub directory and not just the package itself.

1. [**gmapping:**](http://wiki.ros.org/gmapping) With the **gmapping\_demo.launch** file, you 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:**](http://wiki.ros.org/turtlebot_teleop) With the **keyboard\_teleop.launch** file, you can manually control a robot using keyboard commands.
3. [**turtlebot\_rviz\_launchers:**](http://wiki.ros.org/turtlebot_rviz_launchers) With the **view\_navigation.launch** file, you can load a preconfigured rviz workspace. You’ll save a lot of time by launching this file, because it will automatically load the robot model, trajectories, and map for you.
4. [**turtlebot\_gazebo:**](http://wiki.ros.org/turtlebot_gazebo) With the **turtlebot\_world.launch** you can deploy a turtlebot in a gazebo environment by linking the world file to it.

**Your Packages and Directories**

You’ll install these packages and create the directories as you go through the project.

1. **map:** Inside this directory, you will store your gazebo world file and the map generated from SLAM.
2. **scripts:** Inside this directory, you’ll store your shell scripts.
3. **rvizConfig:** Inside this directory, you’ll store your customized rviz configuration files.
4. **pick\_objects:** You will write a node that commands your robot to drive to the pickup and drop off zones.
5. **add\_markers:** You will write a node that model the object with a marker in rviz.

Your package should look like this now:

A screenshot of a computer program

AI-generated content may be incorrect.

### SLAM Testing

The next task of this project is to autonomously map the environment you designed earlier with the Building Editor in Gazebo. But before you tackle autonomous mapping, it’s important to test if you are able to manually perform SLAM by teleoperating your robot. The goal of this step is to manually test SLAM.

Write a shell script test\_slam.sh that will deploy a turtlebot inside your 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 feel free to use your personalized robot to make your project stand out!

A screenshot of a computer program

AI-generated content may be incorrect.

**Run and Test**

Launch your test\_slam.sh file, search for the xterminal running the keyboard\_teleopnode, and start controlling your robot. You are not required to fully map your environment but just make sure everything is working fine. You might notice that the map is low quality, but don’t worry about that for now. If everything seems to be working fine, move on to the next concept!

A screenshot of a computer

AI-generated content may be incorrect.

### Localization and Navigation Testing

The next task of this project is to pick two different goals and test your robot's ability to reach them and orient itself with respect to them. We will 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 permits your robot to avoid any obstacle on its path by re-planning a new trajectory once your robot encounters them. You are familiar with this navigation stack from the localization project where you interfaced with it and sent a specific goal for your robot to reach while localizing itself with AMCL. If you are planning to modify the ROS navigation algorithm or you are curious to know how it's done, take a look at this official tutorial which teaches you how to write a global path planner as a plugin in ROS.

Video 5.7

A screenshot of a computer

AI-generated content may be incorrect.

### Navigation Goal Node

**Reaching Multiple Goals**

Earlier, you tested your robot capabilities in reaching multiple goals by manually commanding it to travel with the 2D NAV Goal arrow in rviz. Now, you will write a node that will communicate with the ROS navigation stack and autonomously send successive goals for your robot to reach. As mentioned earlier, the ROS navigation stack creates a path for your robot based on **Dijkstra's** algorithm, a variant of the **Uniform Cost Search** algorithm, while avoiding obstacles on its path.

There is an official ROS tutorial that teaches you how to send a single goal position and orientation to the navigation stack. You are already familiar with this code from the Localization project where you used it to send your robot to a pre-defined goal. Check out the [**tutorial**](http://wiki.ros.org/navigation/Tutorials/SendingSimpleGoals) and go through its documentation.

Here’s the C++ code of this node which sends a **single goal** for the robot to reach. I included some extra comments to help you understand it:

Video 5.8

#include <ros/ros.h>

#include <move\_base\_msgs/MoveBaseAction.h>

#include <actionlib/client/simple\_action\_client.h>

// Define a client for to send goal requests to the move\_base server through a 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, "simple\_navigation\_goals");

  //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");

  }

  move\_base\_msgs::MoveBaseGoal goal;

  // set up the frame parameters

  goal.target\_pose.header.frame\_id = "base\_link";

  goal.target\_pose.header.stamp = ros::Time::now();

  // Define a position and orientation for the robot to reach

  goal.target\_pose.pose.position.x = 1.0;

  goal.target\_pose.pose.orientation.w = 1.0;

   // Send the goal position and orientation for the robot to reach

  ROS\_INFO("Sending goal");

  ac.sendGoal(goal);

  // Wait an infinite time for the results

  ac.waitForResult();

  // Check if the robot reached its goal

  if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED)

    ROS\_INFO("Hooray, the base moved 1 meter forward");

  else

    ROS\_INFO("The base failed to move forward 1 meter for some reason");

  return 0;

}

A screenshot of a computer program

AI-generated content may be incorrect.

### Virtual Objects

**Modeling Virtual Objects**

The final task of this project is 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.

First, let’s see how markers can be drawn in rviz. Luckily, there’s an official ROS tutorial that teaches you how to do it. The tutorial is an excellent reference and includes a C++ node capable of drawing basic shapes like arrows, cubes, cylinders, and spheres in rviz. You will learn how to define a marker, scale it, define its position and orientation, and finally publish it to rviz. The node included in the tutorial will publish a different shape each second at the same position and orientation. Check out the [**tutorial**](http://wiki.ros.org/rviz/Tutorials/Markers%3A%20Basic%20Shapes) and go through the documentation to get started.

You will need to first run this node and visualize the markers in rviz. Then you’ll need to modify the code and publish a single shape example: a cube. Your code should follow this **algorithm**:

* Publish the marker at the pickup zone
* Pause 5 seconds
* Hide the marker
* Pause 5 seconds
* Publish the marker at the drop off zone

Later you will be able to combine this node with the pick\_objects node coded earlier to simulate the full home service robot.

Video 5.9

A screenshot of a computer

AI-generated content may be incorrect.

### Your Home Service Robot

Now it’s time to 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** node should be communicating. Or, more precisely, the **add\_markers** node should subscribe to your **odometry** to keep track of your robot pose.

Modify the **add\_markers** node as follows:

* Initially show the marker at the pickup zone
* Hide the marker once your robot reaches the pickup zone
* Wait 5 seconds to simulate a pickup
* Show the marker at the drop off zone once your robot reaches it

A screenshot of a computer

AI-generated content may be incorrect.

**Note**

There are many ways to solve this problem. To establish communications between the robot and the markers, one method already mentioned is to let your add\_markers node subscribe to your robot odometry and keep track of your robot pose.

Other solutions to this problem might be to use ROS [**parameters**](http://wiki.ros.org/ROS/Tutorials/UnderstandingServicesParams), subscribe to the AMCL pose, or even to publish a new variable that indicates whether or not your robot is at the pickup or drop off zone. Feel free to solve this problem in any way you wish.

Video 5.10

### Project Rubric

**Basic Requirements**

| **Criteria** | **Submission Requirements** |
| --- | --- |
| Did the student submit all required files? | Student submitted all required files:   * ROS Packages. * Shell scripts. |

**Simulation Setup**

| **Criteria** | **Submission Requirements** |
| --- | --- |
| Did the student set up the simulation environment properly? | Student's simulation world and robot could properly load in Gazebo. |

**Mapping**

| **Criteria** | **Submission Requirements** |
| --- | --- |
| Did the student's mapping function work properly? | The student must provide a functional test\_slam.sh script that:   * Launches the robot and SLAM node with necessary configurations. * Allows robot manual teleoperation to test SLAM and observe map generation in RViz. |
| Did the student create a map using SLAM? | Student created a functional map of the environment which would be used for localization and navigation tasks. |

**Localization and Navigation**

| **Criteria** | **Submission Requirements** |
| --- | --- |
| Was the student's navigation stack configured properly? | * The student's robot could navigate the environment after issuing a 2D Nav Goal command. * The student created a test\_navigation.sh script file to launch it for manual navigation test. |
| Did the student's goal node function properly? | The pick\_objects.sh script must:   * Send multiple goal coordinates to the robot. * Control the robot to:   + Travel to the pickup zone.   + Display a message (e.g., “Reached pickup zone”) upon arrival.   + Wait 5 seconds to simulate loading.   + Travel to the drop-off zone.   + Display a message (e.g., “Reached drop-off zone”) upon arrival. |

**Home Service Functions**

| **Criteria** | **Submission Requirements** |
| --- | --- |
| Did the student create virtual object with markers? | * The add\_marker.sh script must:   + Publish a marker in RViz at the pickup zone when the robot starts.   + Hide the marker when the robot reaches the pickup zone to simulate pickup.   + Re-publish the marker at the drop-off zone after the robot simulates delivery. |
| Does the student's robot perform home service tasks correctly? | * The home\_service.sh script must:   + Launch all required nodes to perform home service tasks.   + Simulate the following sequence:     - Start with the marker at the pickup zone.     - Hide the marker upon robot arrival at the pickup zone.     - Wait 5 seconds to simulate pickup.     - Move the robot to the drop-off zone and re-publish the marker at the drop-off location |
| Did the student include a write-up explaining the packages used to achieve home service functionalities? | The student should include a brief write-up explaining the packages used for this project, covering localization, mapping, and navigation. |

### Submit Project