Path Planning For Autonomous Vehicles

Jonathan Dorsey
https://github.com/JonnyD1117/python-path-planners

***Abstract*—This work introduces the problem of path planning for mobile autonomous vehicles, in static environments. It focuses on the practical context surrounding path planning such as occupancy grid maps, simultaneous localization and mapping (SLAM), and graph generation. After providing a broad overview of the general problem context, this paper surveys several of the most common and widely used planning algorithms. These algorithms include A*, Dijkstra's shortest path, Rapidly Exploring Random Trees, and Probabilistic Road Maps. Each of these algorithms has previously been the state-of-the-art, in path planning, for mobile robots over the years and each continues to provide the theoretical foundational that many modern path planners are often synthesized from. This paper concludes with a case study covering the realworld performance, capabilities, and limitations for each of the aforementioned algorithms under a scenario of path planning for autonomous vehicles in a known environment.**

***Index Terms*—A*, Dijkstra, PRM, RRT, path planning, mobile robots, autonomous vehicles**

I. Introduction

ROBOTICS is a broad, complex, and multifaceted field of study that integrates a wide range of technical disciplines. One of the most visible applications of robotics, in the public consciousness, is the emergence of autonomous vehicles and the advancements in technology and computing which has made it feasible. The task of automating the movement of a vehicle or robot in a safe, practical, and reliable manner is a complex endeavor that engages many tough societal and technical issues. One of the most fundamental technical challenges is the task of **path planning**. This problem is of primary importance as the creation of a feasible path through a known or unknown space is often the first of many stages in a guidance and navigation stack.

The broad objective of this paper is to survey a selection of the most popular path planners in use today, juxtapose their relative merits, and contextualize the process of constructing and implementing these algorithms in Python to test their performance. Additionally, this paper covers the post-processing of path planner waypoints into a usable continuous path for certain realworld applications.

II. The Path Planning Problem

The general path planning problem focuses on the task of generating a feasible and traversable sequence of waypoints (e.g. a path) between a starting position and a target destination. The creation of this path is possibly subject to finite compute resources, a cost-function, and/or a set of system constraints. Additionally, in its' most general form, path planning allows for the inclusion of dynamics obstacles, in the environment [6].

It should be immediately obvious that this definition encompasses a broad class of physical environments, algorithms, and path generation criteria. While the gold standard for path planning is real-time path generation in unknown dynamic environments, achieving this goal is frequently not desirable or realizable for a system to reach given the objectives, constraints, and limitations it must operate within. Depending on the intended function and computational resources available, planning assumptions often need to be relaxed or simplified to obtain a viable solution.

A. Path Planning in Mobile Robotics

It should be noted that path planning possesses many applications beyond those of autonomous vehicles and mobile robotics, such as in video games, and end effector manipulation tasks. While the general premise of generating a path between two configurations is the same, these applications frequently have extremely different limitations, constraints, and performance requirements even though the underlying theory transcends the specifics of the application.

As the intended application of this paper is to apply path planning techniques to real-world autonomous vehicles, it is important to understand the limitations, criteria, and specifications that this application usually demands. The following is a list of a few of the most common constraints.

1) Real-time performance
2) High bandwidth sensor inputs (e.g. Lidars & Cameras)
3) Obstacle avoidance
4) Limited onboard computation
5) Dynamic/Uncertain environments
6) Needs to both localize within and map its environment.

B. Offline Planning for Static Environments

By far, the most common simplification applied to path planning is to assume that the world the mobile robot is operating within is completely static. This implies that so long as the map the vehicle is using to plan is correct and a feasible path exists, planning can be done offline since the environment is assumed to be time invariant. This assumption possesses several attractive properties concerning both planning formulation as well as the planner implementation and performance.

From the perspective of formulating the planning problem, this assumption significantly reduces scope and complexity by removing all mapping and environmental uncertainty at runtime. By eliminating the need to handle dynamic obstacles or mapping uncertainty, this assumption lets roboticists get to the heart of the planning and to explore the theory of planning a path from point ”a” to point ”b”, without the messy engineering details that crop up in practice. As the objective of this paper is to present and compare a handful of fundamental path planning algorithms, the rest of this paper makes the assumption that path planning is occurring in a static environment.

III. Prerequisite Concepts

Like many other domains within the study of robotics, path planning involves the understanding and implementation of concepts borrowed from other disciplines. This section attempts to clearly and concisely present key concepts and details in hopes of anchoring the following discussion by first contextualizing some of the important tasks, principles, and assumptions that are fundamental in the design and implementation of the planners covered in this paper.

A. The Mapping Problem

One of the most fundamental requirements for any path planning system is to develop an understanding of where it is in the physical world. It would be impossible for a robot to navigate its environment without the ability to know where it currently is, the location of its final goal state, and the location of obstacles or constraints within the environment. This implies a need for the generation of an intermediate representation that a robot can utilize to localize and navigate through.

By definition, a physical environment is the ground truth representation which we would like the robot to understand; however, this is typically impossible for all but the simplest of environments. The best that can done is to embed as much information as possible into a simplified representation of environmental features. The task of transforming sensor data into a representation that is suitable for navigation is called **mapping**.

During the mapping process useful features, measurements, or landmarks are extracted from the environment, and structured into an often compressed and lossy representation of the ground truth. The specific data structures and algorithms used vary greatly depending on the application, the number and type of sensors available, as well as the type and speed of compute resources. These representations vary in structure, fidelity, and ease of use ranging from dense 2D/3D lidar point clouds to application specific feature extractors that can fuse several sensors together to extract the desired information.

One of the primary tradeoffs when generating a map is the balance between simplicity, size, and data quality. More often than not, maps are chosen to be overly simple as these can achieve better performance and faster update rates than a more complex mapping which might require more computation to process and query. Even so, the objective still remains to preserve as much feature quality as possible (e.g limit measurement/environmental uncertainty) while still completing the desired task(s) within an acceptable margin. As with many things in robotics, acceptable mapping performance often is a result of correct parameterization. As such the representation for a map (or the associated data structures) can have a tremendous impact on the fundamental design and limitations concerning the underlying mechanism or efficiency for a particular path planning algorithm.

It should be noted that in the domain of mobile robots the process of mapping is generally referred to as Simultaneous Localization and Mapping (SLAM). Unlike the mapping of a room by a static observer, SLAM algorithms need to account for the location and movement of the robot while it is generating a map from its sensor data. While considered by most to be a solved problem, improvements in this field are still evolving as sensor technology (e.g. 3D Lidar Sensors...etc) and onboard computation increases further unlocking the reality of higher fidelity maps that can be generated in close to real-time.

This brief overview has only provided the most superficial glimpse into the process of mapping for mobile robots to establish the dependency between map generation as a prerequisite to plath planning.

1) Dimensionality Reduction & Discretization: In the era of computers, map generation routinely tradeoffs quality and quantity of information for portability and efficiency in computing. This typically means that maps must be compact enough to fit into system memory or onboard storage as well as being efficient to query. To this end, even though robots exist in a three-dimensions of space, it is often sufficient to navigate the 3D world, by planning a path in a 2D projection. This technique is called **dimensionality reduction**.

Unfortunately, even a 2D continuous map is often computationally too untractable to resolve explicitly. The most common approach to this problem, not just in robotics but also in computational mathematics, is to discretize a continuous domain into discrete grids or cells. By doing so, the computer can efficiently store and access each cell of a map with finite resources. The side-effect of discretization is an unavoidable loss/degradation of the original information.

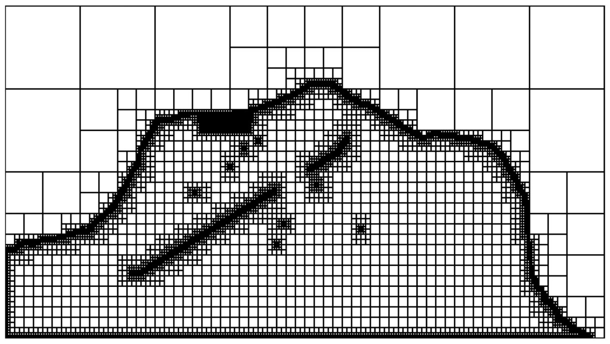
Fig. 1: Example of Quadtree Discretization

While undesirable, carefully parameterization and clever discretization methods aid in mitigating lossy behavior, or ensure that any degradation of quality is functionally insignificant to the problem at hand. There are several popular spatial discretization schemes, with the most common listed below.

- Uniform Grid
- Quad Tree (2D discretization)
- Oct Tree (3D voxelization)
- Delaunay Triangulation (2D or 3D Mesh generation)

Generally, the most commonly used method is a uniform discretization. This structure is intuitive, simple to understand, and fast to implement in code; however, uniform discretization is subject to the curse of dimensionality. Whenever an environment is large and requires a high resolution, uniform grids become slow to search and expensive to store. In these situations, a clever trick is to partition a space into an non-uniform grid structure using Quad or Oct trees. These data structures are built so that they will keep the grid as large as possible unless it is a region of the map that is determined to require higher resolution.

In the case of Quad and Oct trees, recursive discretization is only performed around the perimeter of obstacles. This allows for the majority of the grid to be constructed from coarse grid cells while grid cells surrounding obstacles are finely resolved. This variable resolution enables fast generation of paths through free space while simultaneously providing highly resolved grids around obstacles so that they can be circumnavigated efficiently.

2) SLAM: Frequently, there exists situations where functional maps do not exist for a given environment, and they must be either preprocessed or computed on the fly by the robot as it traverses the environment [10]. The latter operation is known as **Simultaneous Localization & Mapping** or SLAM for short. SLAM is an incredibly deep field of study with many different approaches from EKF SLAM, Graph SLAM, RGBD SLAM, and other vision based SLAM algorithms

For the purpose of this paper, it suffices to say that without a complete or partial mapping of the environment, the robot will first have to construct the map before its able to fully plan a path through the environment. While SLAM is an important area in mobile robotics, for the planners implemented in this paper, we assume that map has already been generated and has been provided to the path planner offline.

3) Occupancy Grid Maps: One of the most popular mapping representations is that of the **occupancy grid map** (OGM), also referred to as a probability grid map or PGM [10]. As the name implies, this map is composed of a grid; however the unique distinction of an OGM is that each cell is associated with a probability ranging from $[0, 1]$. This value indicates the probability that a grid cell is occupied. Zero indicates completely free and one indicates completely obstructed, with any value between indicating the likelihood of occupancy.

One of the many advantages an occupancy grid map possesses is how computers store and process them. If we assume a 2D grid map with a uniform discretization, the grid map effectively becomes a grayscale picture of size $m \times n$ pixels and a resolution of $2^d - 1$, where "d" is the bit depth. The similarity between occupancy grid maps and grayscale images

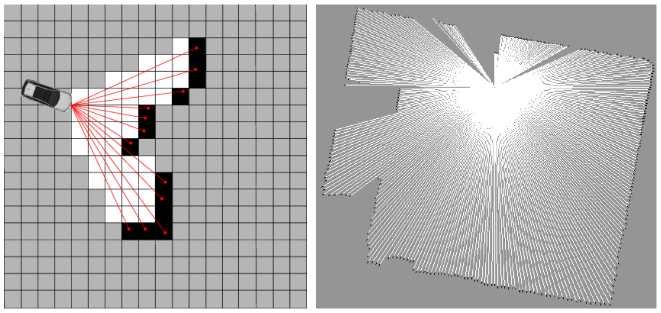
Fig. 2: Occupancy Grid Map

is so striking that most grid maps are actually stored as 8-bit grayscale images within the computer. This is very practical as many programming languages have libraries that can easily read in images and are optimized to perform operations over them rapidly. The remainder of this paper assumes that every map is encoded as a occupancy grid map.

B. Graph Generation

Graph-based planners have historically been the most popular and practical implementation for defining maps of non-trivial size. For the purposes of this paper, all of the planners that we will cover are graph-based and use some variant of a graph-like data structure (e.g. graphs or trees) [4].

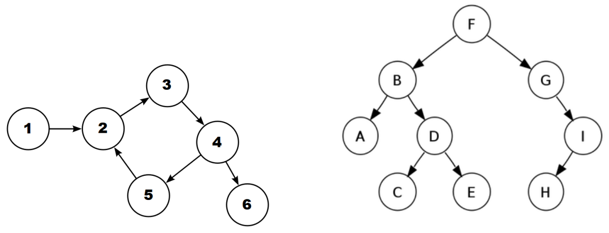

Fig. 3: Graph and Tree Data Structures

1) Grid World: One of the most common approaches to constructing a graph is the notion of directly converting each cell of an occupancy grid map into a graph node, and then applying rules for how to wire up edges between each of these nodes. When generated from a uniformly discretized grid map, this is called a **grid world**. The uniform node generation and rule-based edge connection means that this graph is intuitive and extremely simple to implement programmatically.

The two most popular connectivity rules are known as **connected-8** and **connected-4**. In connected-4, the rules that we imposes on transitioning to a new cell limit legal moves to up/down/left/right (no diagonals are permitted.) while connected-8 graphs are allowed to move one cell in any direction up/down/left/right and diagonals.

In addition to these connectivity rules, there are two different methods concerning how obstacles can be integrated into the final graph. These two methods are **infinite edge weights** and **edge removal**. Infinite edge weighting works by guaranteeing that the shortest (aka lowest cost) path through the graph will never be through an obstacle. Since transitioning to an

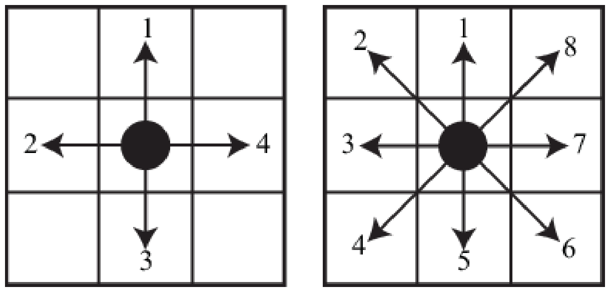

Fig. 4: Connected-4 & Connected-8

obstacle node will incur an infinite cost, this prevents obstacles from being included in the final path. On the other hand, the edge removal method simply removes all edges that would connect to an obstacle node, in the graph. This method works by literally making it impossible for the planner to traverse obstacles as they have all been removed from the graph. In the case of most algorithms used for static environments, this process of handling obstacle nodes is typically static and preprocessed before the robot begins navigating within the environment.

2) Random Sampling: One of the downsides of grid world search graphs is the node density. By definition, every pixel of the occupancy grid map could represent a free cell, and is subject to the curse of dimensionality as the size of the map increases. To address this scaling problem, a popular technique is to employ **random sampling**. This approach is based on the concept of Monte Carlo methods, where a uniformly sampled map is used to create nodes in the graph [9].

In principle, in the limit as the number of samples approaches infinity, we can guarantee the convergence of our path planner to the globally optimal path (if it exists). In practice, sampling-based methods can possess an order of magnitude fewer nodes while still providing enough spatial coverage at a high enough sample density to produce a feasible path through the environment that is trending towards the global optimal path. An excellent example of this is shown the in probabilistic road maps algorithm.

3) Binary Obstacle Inflation: One of the pitfalls of using occupancy grid maps as the basis for the search graph is that each cell is effectively a pixel in an image. Depending on the density of our sensor coverage, there is a non-zero chance that paths through the environment that are only a single pixel wide exist. This is a problem for our robot if it happens to be larger than the adjusted real world scale of the pixel. This is almost always the case.

To prevent situations like this from occurring and generating paths through infeasible regions of the map, we can apply a technique known as an binary obstacle inflation. This very simple technique artificially scales up every obstacle pixel in the scene take up more space than the actual sensor reading indicated. The result of this inflation is that sensor noise is usually absorbed into these inflated obstacles. Additionally, this technique provides a sort of poor-mans approach to prevent the phenomenon of ”wall-hugging”. In simple implementations of path planner, the planner does not have a concept of ”robot size”. This means that the planner will attempt to make the shortest path between two points, even if that path is infeasible for the robot to traverse. This is demonstrated as a sort of behavior where the path generated hugs walls in the environment to attempt reduce the distance (total cost) of the path. By applying an appropriately sized inflation layer, we can guarantee that the path between points, accounts for the physical footprint of the robot. While nifty, there are more sophisticated techniques based on applying cost gradients as a function of the distance away from an obstacle/wall. The cost gradient incentivizes the planner to prefer cells that are farther away from walls; however, this requires more postprocessing of a map before being fed into the planner.

IV. Path Planning

In the domain of robotics, there are hundreds if not thousands of different path planning implementations some of which present totally new formulations for planning but most present incremental improvements that decrease the computational resources, or optimize some custom metric. The goal of this paper is to present a handful of the most popular (and historically important) path planners that have made a significant impact in robotics and have become defacto standard benchmarks against which all other planning algorithms compare their performance.

This paper will cover A*, RRT, PRM, and Dijkstra's Shortest Path. Each of these algorithms is graph-based in so much as they utilize a graph or tree as the data structure from which the algorithm will attempt to generate the shortest path. While algorithms like A* and Dijkstra typically can be given a preexisting graph to traverse, algorithms like RRT and PRM implement a type of Monte Carlo random sampling to generate the graph/tree that they traverse, later after the path generation phase of the planner. This section will cover each of the aforementioned algorithms in broad strokes, describing the general construction and planning mechanism(s) at play.

A. Global vs Local Planners

Before diving into the details of each planner, it is important to clear up the role of these planners and how this role effects the type of planning being performed in the system. Planners can generally be categorized as either **global** and **local**.

A global planner provides an idealized ”global” path between the starting location and the target destination. Typically this plan is optimizing some metric(s) such as shortest distance, curvature constraints, and/or static object avoidance. In a perfect world, an autonomous robot/vehicle would only require a global planner and a means of localizing itself (from sensor measurements); however, dynamic/uncertain obstacles coupled with imperfect sensing, mapping, and localization require robots to be more robust to uncertainty. The planning algorithms shown in the paper are all examples of global planners, as we have already made the assumption that the environment is static.

To combat environmental uncertainty, roboticists developed the idea of local planners. Functionally, local planners use onboard sensor measurements to account for uncertainties in the global path in realtime. This is accomplished by planning deviations from the global path to maintain feasibility and to remain unobstructed. Additionally, local planners can include computation to smooth the global path and provide waypoints to the control system. However, these planners are more than just post-processing of the global path in so much as they need to be aware of the pose, dynamics, kinematics, constraints, and sensor inputs of the robot/vehicle.

B. Path vs Motion Planning

Frequently, the terminology of path planning is conflated with that of motion planning. While related, these concepts are very different and each solves a different problem. In path planning, the objective is to find the path through an environment that optimizes some metric and obeys specified constraints. Typically this metric is distance or cost and the constraint is to maintain a feasible and obstacle free path between the starting and ending points.

Motion planning, on the other hand, uses path planning as an input to determine how the path should be traversed through time. This means not only planning the position of the path (e.g path planning) but also planning the velocity, acceleration, and often the jerk of a robot as it follows the given path. Almost always, motion planning requires the ability to compute derivatives of the path position.

C. Planning Algorithms

The first category of planning algorithms are so-called **Graph-Search**. The primary mechanism these planners employ is to construct a graph from a map. Once this has been achieved, different algorithms, heuristics, and operations can be applied over the graph to endeavor to find the shortest path between two nodes that exist in that graph. Occupancy grid maps are ideal for the such path planners in that it is trivial to translate the grid map into a graph data structure by assuming that every pixel in the grid map is node in the graph, and then formulate the connectivity rules as desired (see **connected-4** or **connected-8** above).

1) Dijstra's Shortest Path: The predecessor of almost all graph-search planners is Dijkstra's shortest path. This algorithm was invented by Dutch mathematician Edsger Dijkstra in Norway in 1956 [2]. It implements a type of breadth-first search that will traverse an undirected graph, until it has found the shortest route between two nodes within the graph.

The implementation found in this paper for Dijkstra's algorithm uses an occupancy grid map to generate a dense graph. Cells in the map that have an occupancy value over a certain threshold are assumed to be occupied. The algorithm works by computing the cost of the current node, and then determining the total cost of each neighbor from the current node. The neighbor with the least cost is expanded in the same way until either the entire graph has been explored or the path from the start and end nodes has been determined. See algorithm for more details.

Algorithm 1 Dijkstra's Shortest Path

```
for node in Graph do
    node.score := $\infty$
    node.visted := false
end for
Solve Graph :
starNode.score := 0
while ( true ) do
    currentNode := nodeWithLowestScore(Graph)
    currentNode.visted := true
    for ( node in currentNode.neighbors ) do
        if ( node.visited == false ) then
            newScore := calculateScore(currentNode, node)
            if ( newScore < node.score ) then
                node.score := newScore
                node.routeToNode := currentNode
            end if
        end if
    end for
    Return Path :
    if ( currentNode == targetNode ) then
        return buildPath(targetNode)
    end if
    Raise Error if No Path is Found :
    if nodeWithLowestScore(graph).score == $\infty$ then
        return ERROR: No Path Found
    end if
end while
```

While this algorithm is simple and guaranteed to find the shortest path in the graph, assuming it exists, it cannot predict the efficiency of how well it is at finding the shortest path. This is predominately attributable to the fact that the algorithm is a generalization of a breadth-first search, blindly expanding every node closest to it without any concern to focus the direction of the search towards more likely paths (e.g. in the direction of the goal state).

2) A:* The A* algorithm was developed at the Stanford Research Institute in 1968, for the historically significant robot Shakey, being used in some of the earliest pioneering research into mobile robotics [3]. It is both the literal and conceptual successor to Dijkstra's algorithm. Both implement a graph-search; however, the primary improvement of A* is the addition of a search heuristic that improves the ability of the algorithm to focus on expanding nodes in the direction of the goal state.

Unlike Dijkstra's algorithm, the addition of a heuristic provides the planner with the ability to preferentially search in a given direction and will traverse nodes that are more likely to be approaching the goal node fom the starting node. So long as the heuristic is quick to compute and admissible (an exact or under estimate of the actual distance to the goal node), we can use a search heuristic to drastically improve the planners convergence rate.

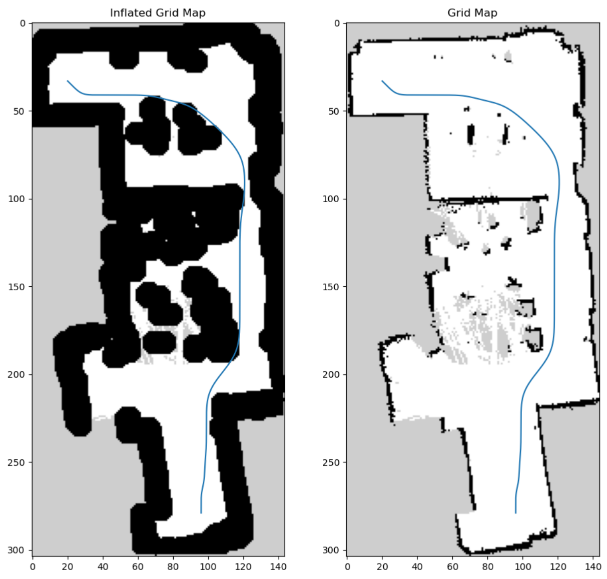

Fig. 5: Dijkstra Algorithm in SLAM generate Map

A* is extremely similar to Dijkstra's algorithm described above. The key difference is how A* computes the cost of getting to the current node, and then computes the cost of traversing to each neighboring node plus the addition of a heuristic cost from each neighbor to the goal state. The inclusion of a heuristic cost allows A* to preferentially choose to visit neighbors that are closer to the goal state by choosing the neighbor with the lowest combined current plus heuristic cost. See algorithm for more details.

A* has made a tremendous impact on the world of robotics and today is the defacto standard for simple path planning tasks. Unfortunately, both A* and Dijkstra suffer from the necessity to generate a graph of the environment. As previously mentioned, this usually involves the conversion of a grid map into graph. While not always an issue, the dimension of the grid map corresponds to the need to generate a larger search graph, which requires either algorithm to spend more time to compute the shortest path. Unlike Dijkstra's, the heuristic driven nature of A* also means that it is typically far more efficient of a search than Dijkstra's algorithm over the same search graph.

3) Probabilistic Road Maps: While graph-search planners have been popular and practical historically, methods like Dijkstra's and A* can only be as efficient as the size, resolution, and connectivity of their graph permits. Depending on how these graphs were generated, they often suffer from the inability to efficiently scale as the size of a given map increases, leading to unacceptable latency and computation times, known as the curse of dimensionality.

To address the problem of high graph densities, the algorithm Probabilistic Road Maps (PRM) was developed. PRM has two main stages; graph creation via random sampling, and path computation by directly applying Dijkstra's algorithm. The broad objective of PRM is to decrease the density of the search-graph by applying uniform sampling as the mechanism for generating graph nodes within the map [5]. The compromise this method makes is the reduction of graph generation and graph traversal times in exchange for a lower fidelity map that is no longer pixel accurate to the map.

Algorithm 2 A* Heuristic Search

```
1:  for ( node in Graph ) do
2:      node.gScore := $\infty$
3:      node.fScore := $\infty$
4:      node.visted := false
5:  end for
    Solve Graph :
6:  starNode.gScore := 0
7:  starNode.fScore := 0
8:  while ( true ) do
9:      currentNode := nodeWithLowestScore(Graph)
10:     currentNode.visted := true
11:     for ( node in currentNode.neighbors ) do
12:         if ( node.visited == false ) then
13:             newScore := calcScore(currentNode, node)
14:             if ( newScore $<$ node.g ) then
15:                 node.gScore := newScore
16:                 node.fScore := newScore $+$ calcHeuristic(nextNode, targetNode)
17:                 node.routeToNode := currentNode
18:             end if
19:         end if
20:     end for
        Return Path :
21:     if ( currentNode == targetNode ) then
22:         return buildPath(targetNode)
23:     end if
        Raise Error if No Path is Found :
24:     if nodeWithLowestScore(graph).score == $\infty$ then
25:         return ERROR: No Path Found
26:     end if
27: end while
```

PRM works by randomly selecting a number of points within the domain of the map. It will first check whether or not the randomly selected point is an obstacle. If the sample is contained within an obstacle, the algorithm will redraw random samples until it finds a point that is *free*. This point is then added to an ever growing graph, so long as the edge connecting this new node and the previous nodes in the graph do not intersect obstacles. Once this has been performed for a sufficient number of random samples, the graph or 'roadmap' now exists. The second phase of the algorithm merely selects a starting and target nodes, in the graph and applies a graph-search algorithm between those two endpoints.

This technique is not only faster for the graph generation portion of the algorithm, but since the graph it generates is typically over an order of magnitude less dense than an equivalent grid map based graph, finding the shortest path between two points on this "sampled" graph results in significantly faster graph traversal. It should be noted that

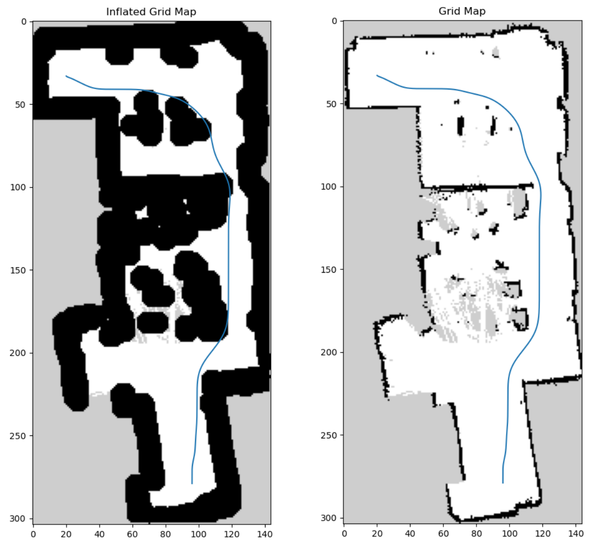

Fig. 6: A Star

unlike the other methods, PRM is only optimal within the limit as the number of samples becomes infinite. This means that unlike A* traversing a dense grid-based graph, PRM with a finite sample size is unlikely to find the optimally shortest path; however it does provide a locally optimal path across its sampled-graph. This behavior demonstrates how PRM needs to be tuned to its targeted application for effective results.

Algorithm 3 Roadmap Construction

Input: n: number of nodes in roadmap
k: number of closest neighbors

Output: A roadmap G = (V, E)

```
1: $V \leftarrow \emptyset$
2: $E \leftarrow \emptyset$
3: while $|V| < n$ do
4:     repeat
5:       q $\leftarrow$ random sample from Configuration Space $Q$
6:     until q is collision-free
7:     $V \leftarrow V \cup$ {q}
8: end while
9: for all q $\in$ V do
10:    $N_q \leftarrow$ the k nearest neighbors of q from V by dist
11:    for all $q' \in N_q$ do
12:        if (q, $q'$) $\notin$ E and $\Delta$(q, $q'$) $\neq 0$ then
13:            E $\leftarrow$ E $\cup$ {(q, q')}
14:        end if
15:    end for
16: end for
```

4) Rapidly Exploring Random Trees: Similar to PRM, Rapidly Exploring Random Trees (RRT), also attempts to leverage random sampling to improve the sample efficiently and reduce the graph generation time [7]. The primary difference between PRM and RRT is their data structures and how additional nodes are inserted into this data structure.

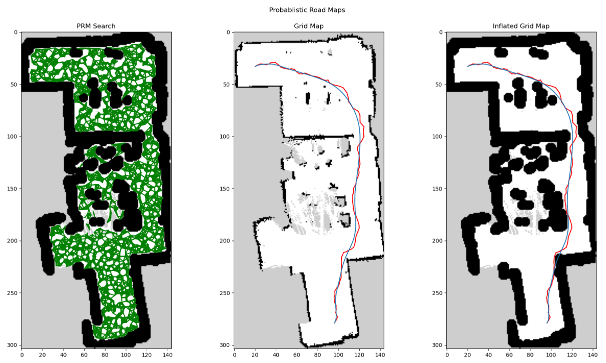

Fig. 7: Probabilistic Road Maps

While PRM utilizes a graph, the RRT class of algorithms use a tree structure. Tree structures possess several benefits but also impose several constraints on the representation of the map that these algorithms generate.

The most useful aspect of using a tree is the simplicity in determining the final path. Unlike all of the graph-based methods that require iteratively searching over a large portion of nodes in the graph to determine the shortest path, a tree structure is a very simple structure that lends itself to recursive solutions. Since unlike a node in a graph, a node in a tree can only possess one predecessor, one need only begin at the final target node and recursively make a list of each predecessor node, until the starting node is reached.

Like PRM, RRT will randomly sample its map, check that these random points are not obstacles in the map (resample if true) and then create a tree by tieing that random point into the nearest node of the tree, by taking a step of size towards the random point, from its closest node on tree. At this new location, it will create a new node that is parented to the nearest neighbor previously mentioned.

Algorithm 4 RRT Construction

Input: q_0:the configuration where the tree is rooted
n: the number of attempts to expand the tree

Output: A tree T = (V, E) rooted at q_0

```
1: $V \leftarrow \{q_0\}$
2: $E \leftarrow \emptyset$
3: for i = 1 to n do
4:     $q_{rand} \leftarrow$ a randomly chosen free configuration
5:     $q_{near} \leftarrow$ closest neighbor of q in T
6:     $q_{new} \leftarrow$ node a dist $\Delta S$ from $q_{near}$ to $q_{rand}$
7:     if $q_{new}$ is collision-free then
8:         V $\leftarrow$ V $\cup$ $\{q_{new}\}$
9:         E $\leftarrow$ E $\cup$ $\{(q_{near}, q_{new})\}$
10:        return $q_{new}$
11:    end if
12:    return 0
13: end for
```

V. Path Smoothing

It should be noted that all of the path planners covered above will generate a piece-wise linear path between the specified start and end points. With the exception of a path that only contains two points, the shortest path generated by these planners will always be discontinuous. The nature of this discontinuity is rooted in the discretization applied earlier during graph generation. Unfortunately, while discretized paths are computationally desirable, they often are incompatible with the robotics system that will use the path generated.

The most obvious reason for such an incompatibility is the requirement for continuous inputs into a controller, state estimator, or other component which are formulated under the assumption of continuous inputs. Secondly, even for discrete systems, it is often more desirable to generate waypoints at a much higher sampling frequency than those output directly by the path planner. This is especially true when refining the discretization of a map to a sufficient waypoint frequency would make global path planning through the environment untractable or slow to compute. This need leads to the need for the raw outputs of these planners to be post-processed via a path smoothing algorithm.

A. Bezier Path Smoothing

One of the most simple and straight forward methods for path smoothing is by using Bezier curves. While the derivation of Bezier curves is beyond the scope of this paper [1], sufficed to say that this technique is very effective for quickly computing a smooth continuous curve a discrete series of input points.

1) Bezier Curve Formulation: The formulation of a Bezier curve is the based on the three following equations.

$$\begin{pmatrix} n \\ i \end{pmatrix} = \frac{n!}{i!(n-1)!} \tag{1}$$

$$B_{i,n}(t) = \begin{pmatrix} n \\ i \end{pmatrix} t^i (1-t)^{n-i} \tag{2}$$

$$C(t) = \sum_{i=0}^{n} P_i B_{i,n}(t) \tag{3}$$

Where P_i is the point at index i the planners output list which is of length n and where $B_{i,n}$ is formula for a Bernstein polynomial. By applying these equations to each output of the planner a smooth curves is generate. As with the mathematical formulation of the Bezier curve, the implementing a Bezier curve in code is very direct and straightforward.

2) Limitation of Bezier Curves: Unless the length of the planner output is exceeding large, Bezier curves tend be a quick to compute path smoothing algorithm that is very effective for many applications including simple robotic environments or even video games. Unfortunately, the primary drawback of Bezier curves in robotics, is the inability to

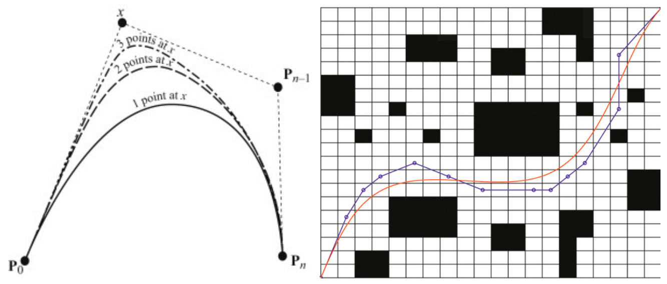

Fig. 8: Cubic Bezier & Bezier Path Smoothing

directly model system or environmental information into the path smoothing process. In the domain of robotics, this typically manifests as the inability to apply custom constraints or guarantee feasibility of the final smoothed path.

While guaranteeing feasibility or respecting applied constraints, might be of little meaningful importance in an application like video games, where the goal is to be good enough while using up the least compute resources, in robotics, obeying constraints and feasibility requirements is a very real necessity.

Even acknowledging this short-coming, this paper, has applied Bezier path smoothing to all of the path planner outputs as a means of quickly and consistently generating a smooth deterministic output path.

B. Optimization Based Path Smoothing

The other predominate class of path smoothing algorithms comes in the form of optimization based methods. Unlike Bezier smoothing, these methods work on the principle of minimizing some cost function associated with the points output by the planner. The strength of optimization-based methods involve the natural ability to define and apply constraints that the solution of the optimizer must obey [1]. For path smoothing, this translates into the ability for these methods to intuitively apply constraints such that the smoothed path will never intersect with static obstacles in the environment, or violate feasibility by generating a path whose curvature is impossible for the robot to realize.

Generally, this class of methods is far more computationally expensive but frequently the benefits that it brings justify the tradeoff for many robotic applications.

1) Timed-Elastic Band: One such method that has been specially adapted for the use with autonomous vehicles, is that of **Timed-Elastic Band** optimization. [8]

The advantage of this algorithm is that it incorporates a standard vehicle model and produces locally optimal paths. The downside of this method is that it is required to be run online, very similarly to the implementation of Model Predictive Control (MPC).

VI. Autonomous Vehicle Case Study

Since the intended application for the planners covered in this paper are for use with autonomous vehicles, it is only appropriate to investigate how these planners actually perform in practice and to compare their quantitative differences.

A. Scenario

The scenario we will use is taken from a concurrent research project within our lab to quantify the differences in performance between several control schemes that require an autonomous vehicle to navigate through a non-trivial environment. Since the objective of that project is to compare the relative performance between different controllers over the same path, this scenario only requires a static environment to ensure continuity between test runs using the different controllers and is the perfect candidate for testing the planners discussed above.

Unlike testing a control system, testing offline path planners can be done completely in simulation under the assumption that the input map is accurate. To ensure this, the map being fed into the planner in this implementation has been generated directly from the autonomous vehicle after SLAMing its environment. This will provide a one to one test of how these planners would have performed on the actual vehicle. This leads to a discussion about which metrics are appropriate markers for comparative performance between planners.

In this scenario, the objective of the planner is to provide the shortest feasible path to the controller as quickly as possible. The goal of the shortest path is to guarantee an efficient traversal of the environment and low compute times are a strong indictor of computational efficiency, both of which are frequently desirable in mobile robotic applications with limited onboard computational resources. While it is possible to select other path cost functions (such as energy minimization or handling characteristics ...etc), choosing the path cost function to be the total path distance is a simple, quick to compute, and intuitive performance metric that is directly measurable from the output of the planner. This being said, this does come at the cost of being generally less flexibility than other cost functions; however, given this scenario, the path distance is more than sufficient for providing relative performance across all planners solving for the same inputs.

B. Setup & Testing

1) Implementation: Each of the planning algorithms presented in this paper (A*, Dijkstra, RRT, & PRM) was written in the Python programming language following the corresponding algorithms definitions defined above. The code for each of these implementations as well as the planner that implements them are available at the following GitHub repository https://github.com/JonnyD1117/python-path-planners. While Python is not the most performant language, it is simple to develop in and is interoperable with the Robot Operating System 2 (ROS2) which was a requirement for this planner to integrate with the controller design project.

The only pre-processing applied to the map was a binary inflation layer to mitigate wall hugging and to attempt to keep the vehicle as centered as possible in corridors. Once the planner had computed its solution, it returns a list of waypoints that get post-processed via Bezier path smoothing. This final smoothed path is then sampled into a series of waypoints that get fed into the controller.

2) Testing: The testing methodology is very straightforward. As this problem only requires an offline planner, we were able to run the planner multiple times over the same map and endpoints to develop a statistical sample concerning the computation time, and the cost of the generated path. In this case, the cost of the path is the total distance of the generate path before applying path smoothing.

C. Results

The following table summarizes the performance characteristics over 30 trial runs per algorithm using the same starting and stopping locations.

Algorithm	**Compute Time [Seconds]**		**Path Cost [Meters]**	
	Mean	***Std. Dev***	***Mean***	**Std. Dev**
A*	0.173	.004	16.115	0
Dijkstra	7.036	.298	16.115	0
PRM	.480	0.018	16.816	0.171
RRT	8.385	6.984	396.233	14.045

TABLE I: Algorithm Performance Statistics

VII. Conclusions

From the results tabulated above, it's evident that there exists measurable performance differences between each of the planner covered in this paper. In particular, it should be noted that the A* and Dijkstra's algorithms produce deterministic output paths given the same input map. This is shown by the identical path cost between these two planners, both with zero variance between trials. However, an even more interesting result is the difference in computation time. While A* and Dijkstra's algorithms produce the same path the incorporation of a heuristic to A* significantly reduces the time it takes to compute a solution. This is one of many reasons why A* has long been the defacto standard.

There is one point of subtlety in the tabulated results for A* and Dijkstra that should be accounted for. While both algorithms will deterministically produce the same path (if given the same graph and start/stop locations), the compute time for each of these is not deterministic. The reason for this determinism in the path generation is purely based on algorithmic construction. Both algorithms do not incorporate any form of stochastic approximation or sampling that would provide any variance in the path being generated. The only algorithmic difference are related to the search efficiency each employs. Even so, both search algorithms are deterministic in nature. On the other hand, the compute time is stochastic due to the nature of process management on a modern computer operating system. The operating system of a modern computer is a complex and involved program that manages and maintains all of the programs that are currently running as processes on a CPU. This involves load balancing the CPU cores, managing calls for input(s)/output(s) ... etc. Even if the program being run is deterministic in nature, the process management being performed by the operating system is not. Additionally, CPU performance can also vary depending on operating conditions (e.g. CPU temperatures), which could cause the compute times to drift when running performance tests sequentially when developing statistical averages.

Moving on, PRM and A* take the least amount of time to compute a viable solution, when given the same input map. Unlike A*, PRM's use of random sampling means that the output path is not deterministic which explains the variance its is path cost. However, while PRM under performs when compared to A*, it is significantly more performant than the remaining algorithms. This could likely be due to the fact that PRM uses Dijkstra's algorithm to compute the shortest path once it has completed the graph generation process. Switching this algorithm to A*, might even conceivably produce better performance than pure A* as the typical PRM graph is far less dense than the grid map which A* searches.

By far the worst performing algorithm is RRT. In both compute time and path cost, the standard implementation of RRT performs terribly. This is partly due to the sample efficiency of tree generation, but is mostly due to the fact that RRT cannot rewire its tree structure to minimize the path, during tree generation. This results in extremely circuitous paths being locked into the tree structure. Even if new samples are generated that would (globally) reduce the path cost of the tree, RRT does not implement any mechanism for disconnecting certain branches and reconnecting them to others to improve the path cost. This means that RRT will only ever increase the path cost beyond the optimal value. These problems are addressed in more recent variants of RRT such as RRT* that optimizes its sampled tree as its being generated. This is also a technique borrowed by other algorithms like PRM* to accomplish the same behavior and trend towards the optimal path as the number of sampled points trends to infinity.

VIII. Future Recommendations

A. Model-based Path Planning/Smoothing

One of the drawbacks unique to the planners presented in this paper, is the inability to incorporate any form of model into the solution process. The graph-based methods previously discussed construct a graph-like data structure without concern for the system that will be using the generated path. While this methodology is simple, efficient, and useful in a broad range of planning applications, it is sometimes desirable to leverage information about the system during the solution process.

This can be done for a variety of reasons; however, the inclusion of a system model typically allows for more efficient implementations that target the model structure or permits the use of model driven constraints to guarantee certain criteria are never violated or that certain performance targets are always achievable. A possible avenue of future work would be to explore different model driven planning algorithms that use explicit vehicle model, to maintain feasibility or make guarantees about system performance.

B. Cost Map Planning

Throughout the implementation and testing of the planners in this paper, one of the most frustrating limitations was the absence of a global cost map. As implemented, grid maps are effectively a binary mapping of free or occupied cells. In theory, this information is sufficient to plan a path but in practice it is limiting, cumbersome, and does not permit more nuanced behavior. While a binary inflation layer can crudely be used to prevent wall hugging and other similar edge cases, a better solution would be the inclusion of a global cost map.

Such a cost map is effectively a grid map where the each cell is assigned a weighted between $[0, 1]$. These weights supplement the obstacle map and function as ”soft” constraints that incentivize (but do not force) the planner to avoid regions of the cost map that have been weighted higher. Since most path planners are attempting to minimize the cost of their generated path, including the cost map in these calculations will make a planner prefer regions of lower cost even if they are not physically the shortest path to the goal. This concept is particularly useful for avoiding wall hugging and guiding a vehicle through the center of doorways or corridors and future work would benefit from its flexibility by its inclusion.

C. Local Planning

The final recommendation for future work concerns the fundamental limitations by solely utilizing global planners. As alluded to throughout this paper, the assumption of global information restricts the ability for these planners to be used in dynamic or uncertain environments where the global information contained in the map might no longer be valid. By processing local sensor/vehicle data at runtime, local planning algorithms could be used to account for dynamic obstacles and mapping uncertainty making the planner significantly more robust and useful in a wider range of applications.

References

[1] Ravankar A et al. “Path Smoothing Techniques in Robot Navigation: State-of-the-Art Current and Future Challenges”. In: *Sensors (Basel)* (2018).

[2] Edsger W Dijkstra. “A note on two problems in connexion with graphs”. In: *Numerische mathematik* 1.1 (1959), pp. 269–271.

[3] Peter Hart, Nils Nilsson, and Bertram Raphael. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/tssc.1968.300136. URL: https://doi.org/10.1109/tssc.1968.300136.

[4] Dieter Jungnickel. *Graphs, Networks and Algorithms*. 3rd. Springer Publishing Company, Incorporated, 2007. ISBN: 3540727795.

[5] L.E. Kavraki et al. “Probabilistic roadmaps for path planning in high-dimensional configuration spaces”. In: *IEEE Transactions on Robotics and Automation* 12.4 (1996), pp. 566–580. DOI: 10.1109/70.508439.

[6] S. M. LaValle. *Planning Algorithms*. Available at http://planning.cs.uiuc.edu/. Cambridge, U.K.: Cambridge University Press, 2006.

[7] Steven M. LaValle. "Rapidly-exploring random trees : a new tool for path planning". In: *The annual research report* (1998).

[8] Christoph Rösmann, Frank Hoffmann, and Torsten Bertram. "Kinodynamic trajectory optimization and control for car-like robots". In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2017, pp. 5681–5686. DOI: 10.1109/IROS.2017.8206458.

[9] Gerardo Rubino and Bruno Tuffin. "An Introduction to Monte Carlo Methods and Rare Event Simulation". In: *2009 Sixth International Conference on the Quantitative Evaluation of Systems*. 2009, pp. 6–6. DOI: 10.1109/QEST.2009.13.

[10] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic robotics*. Cambridge, Mass.: MIT Press, 2005. URL: http://www.amazon.de/gp/product/0262201623/102-8479661-9831324?v=glance&n=283155&n=507846&s=books&v=glance.