LONDON'S GLOBAL UNIVERSITY

COMP0037

Report

Path Planning in a Known World

Group AS

Student Name	Student number
Arundathi Shaji Shanthini	16018351
Dmitry Leyko	16021440
Tharmetharan Balendran	17011729

Department: Department of Electronics and Electrical Engineering

Submission Date: 25th of February 2020

Contents

1 Implement and Investigate Properties of Path Planning Algorithms

1.1 Path Planning Algorithms

```
1: function FORWARDSEARCH(x_I, X_G)
2:     Q.Insert(x_I) and mark x_I as alive
3:     while Q not empty do
4:         x←Q.GetFirst()
5:         if x ∈ X_G then
6:             return SUCCESS
7:         for all u∈U(x) do
8:             x'←f(x, u)
9:             if x' is unvisited then
10:                Mark x' as alive
11:                Q.Insert(x')
12:            else
13:                Resolve duplicate x'
14:        Mark x as dead
15:    return FAILURE
```

Figure 1.1: The general forward search algorithm pseudo-code.

All of the path planning algorithms have the same backbone and can be summed up into 15 lines of pseudo-code shown in Fig 1.1. This algorithm describes the process of finding a path from a cell (x_I) to a goal cell (X_G). This general algorithm is called the general forward search algorithm and is implemented in the **GeneralForwardSearchAlgorithm** class. All planners that are implemented inherit from this class through the **CellBasedForwardSearch** class. The search algorithm is similar for all the planners and has also been implemented in the same **GeneralForwardSearchAlgorithm** class. What differs between the planners is the type of queue that is used and the implementation of the **resolveDuplicate** function. These properties and functions are defined in the classes of each of the individual planners. It should also be noted that a class named **PlannerBase** is also present. This function consists of abstract functions and variable decelerations as well as functions required for visualisation of the planned path. The high-level working of the planners we implemented and their properties are described below.

1.1.1 FIFO - Breadth First Search

This algorithm works by selecting a particular cell and exploring its neighbours first. It would then pivot around those neighbours to explore each of their neighbours. Layer by layer it will explore all the cells on the graph network until it reaches the goal. The next cells to be considered are chosen by first considering the cell directly underneath and then considering the neighbouring cells going anti-clockwise. This way of allocating the next set of cells to be considered, is the same for all algorithms. The Breadth First Algorithm uses a queue to keep track of the order of the exploration in the network. If a particular cell shows itself twice to the algorithm, it will use the first instance of it ignoring the second. The first path that reaches the finishing cell is the final output path, no mater of the cost. The name First in First Out (FIFO) comes from the type of queue implemented in this search algorithm. The next cell to be considered at any point is the one that was added the first out of all the cells in the queue.
The Breadth First Search Algorithm utilizes a normal unsorted queue and was implemented using the python **deque** class. For the pushing and popping of cells, the in-built **append** and **popleft** functions were utilized. In the case a cell that was already visited is re-visited in the Breadth First Search, nothing is considered. Therefore the **resolveDuplicate** function is an empty function in this case.
The biggest advantage of FIFO is that it will never get stuck in a blind alley. If there is a solution it

will find it. If there are multiple solutions, it will find one with the least steps. The memory usage can be large as it stores all the cells that it visits. If the solution is far, it might take a lot of time to find, as well as that the FIFO algorithm does not guarantee optimality. In the worst case scenario with the FIFO algorithm will consider every possible cell and edge. Therefore the complexity can be expressed as $O(V + E)$ where V is the total number of vertices/cells and E is the total number of edges in the graph.

1.1.2 LIFO - Depth First Search

This algorithm is very similar to the Breadth First Search. Starting from the initial cell, the algorithm adds cells to the queue in the same fashion as previously described. Where Depth First differs is in the fact that instead of using a first in first out queue, a last in first out (LIFO) queue is utilized. As a result at any given moment the next cell to be considered is the last one that was added to the queue.
The depth first algorithm utilizes a normal python **list** class alongside the **pop** and **append** functions. Once again, in the event a cell is revisited the first instance is considered. This results in an empty implementation of **resolveDuplicate**.
Once again, the depth first algorithm does not grantee the optimal path. In fact, due to the nature of the LIFO queue, the computed paths tend to be chaotic with many zig-zag features. Similarly to the Breadth First Algorithm, in the worst case scenario the Depth first algorithm will consider each cell of the graph. This means that once again the complexity of the algorithm can be expressed as $O(V + E)$ where V is the total number of vertices/cells and E is the total number of edges in the graph.

1.1.3 Greedy Search

The Greedy Search Algorithm starts at the first cell and adds neighbouring cells to the queue. The type of queue used in this algorithm is a priority queue with the priority value being the euclidean distance between the cell and the goal. This means that the cells in the queue are sorted by their euclidean distance to the goal. As a result when a cell is popped from the queue it will be the cell with the lowest euclidean distance to the goal (out of the cells in the queue). As a result the algorithm will consider the cells closer to the goal first. This can drastically reduce the number of cells that are considered resulting in lower computational requirement.
The greedy algorithm was implemented using a priority queue from the **PriorityQueue** library. Along with the built-in **put**, **get** and **empty** functions it was relatively easy to implement the priority queue and thereby the greedy algorithm. Once again in the case a cell is revisited nothing is done, meaning that the **resolveDuplicate** function is empty.
The biggest advantage of the greedy algorithm is its easy implementation and its efficiency in simple cases. In maps with minimal/no obstructions, the greedy algorithm is able to find a path with significantly fewer cells visited compared to breadth first and depth first planners. However the optimality of the path is not guaranteed.

1.1.4 Dijkstra's

This algorithm works by first setting the values for distances to every cell to infinity. Once the starting cell is specified, its cost to go value is set to zero and it is set to have no parent cell. The algorithm then explores the neighbours of the starting cell and records the distance to each. Once this is done, the neighbouring cells are added to the queue which is once again a priority queue with the priority value being the cost to go to the cell. The next cell to be considered is the cell with the lowest cost to go (from the cells in the priority queue). The algorithm will continue until the queue is empty. The cost to go is calculated using the L-stage additive cost which considers the terrain cost as well as edge costs that are present.
Once again the same priority queue as implemented in the greedy algorithm was utilized with changes

to the priority value. In addition to that, the class **DijkstraPlanner** inherits from another class **DynamicPLanner**. This is due to the fact that Dijkstra and A* planners are both dynamic planners and not only use the same queue but only differ by the addition of heuristics. Therefore, the priority queue and functions associated with it are common to both and can therefore be shared. Additionally, the **resolveDuplicate** function is also shared amongst the two planners. In the case a cell is revisited, the existing cost to go and the new proposed cost to go are compared. If the new cost to go is smaller, then the cell's parent and cost to go are updated to reflect this.
The main advantage of utilizing Dijkstra's is the fact that the optimality of the path is guaranteed. In addition to this, Dijkstra computes the path to the goal from not only the start cell but all cells that have been visited. This means that if at a later date another path to the same goal is required, the algorithm doesn't have to restart from beginning. The main disadvantage is that the algorithm is computationally intense having a time complexity of $O(E\log(V)+E)$ where V are vertices/cells and E are edges.

1.1.5 A* Search

This algorithm works in a similar way to Dijkstra's however it additionally uses a heuristic to influence the way it selects which cells to explore and visit. Heuristics used in our assignments are: a non-negative constant value, the euclidean distance to the goal, the octile distance to the goal and the Manhattan distance to the goal. The algorithm is similar to Dijkstra and as previously discussed the two planners inherit from the same **DynamicPlanner** class. When adding a cell to the priority queue, the heuristic value is computed for that cell and it is summed to the L-stage additive cost. This means that the heuristic influences the position of the cell in the priority queue and therefore may change the order in which cells are considered. The heuristics bias the considered cells such that cells closer to the goal are considered first meaning that fewer cells my be visited in the process of finding the final path.
The euclidean distance is the straight line distance between two points expressed as shown in Eq. X.

$$\hat{G}_k = \sqrt{(x_k - x_G)^2 + (y_k - y_G)^2} \tag{1}$$

Depending on robot movement restrictions, the heuristic may be altered to take this into account. For example, the octile distance heuristic

$$\hat{G}_k = max(|x_k - x_G|, |y_k - y_G|) + (\sqrt{2} - 1)min(|x_k - x_G|, |y_k - y_G|) \tag{2}$$

$$\hat{G}_k = |x_k - x_G| + |y_k - y_G| \tag{3}$$

Given that the chosen heuristic is non-zero and never overestimates the cost to the goal, the heuristic is said to be admissible. In the case that a heuristic is admissible, it also follows that the computed path will be the optimal path. The most optimal heuristic will be the one that doesn't just approximate the distance to the goal but rather is exactly equal to the cost to the goal. In this case, the number of cells visited will be minimal and the optimal path will also be identified. However, this is non feasible as computing the actual cost to goal is computationally intense. In the case a heuristic is inadmissible, the optimality of the path is not guaranteed.
In addition to the type of heuristic used, the A* planner can be further customized and tuned by using a weighting parameter on the heuristic. This weighting parameter influences the number of cells searched by the planner and the optimality of the found path. By choosing a weighting factor greater than 1, the algorithm prioritizes cells closer to the goal. However this also means that the weighted heuristic may become inadmissible and therefore not necessarily derive the optimal path. On the other hand, a weighting factor lower than 1 explores more cells than the un-weighted heuristic but will return the same optimal path. This additional exploration of cells may only be useful, if paths from those cells to the goal are required at a later time.
As previously discussed, the implementation of the A* algorithm is very similar to the implementation of Dijkstra. The main difference is the addition of the **calc_heuristics** function that computes the heuristics.

The choice of heuristic is defined in the launch file and can be determined as a argument when launching the node using roslaunch.
The time complexity of the A* algorithm can be expressed as $O(E)$ where E is the total number of edges in the graph.

1.2 Algorithm Performance

1.2.1 Metrics to Quantify Performance

1.2.2 Comparing Performance of Algorithms

1.2.3 Planner Performance Prognosis

Describe the metrics used to compare performance
Descirbe how they were implemented
Compare and explain the difference in performance of the algorithms (See Teams for more details)

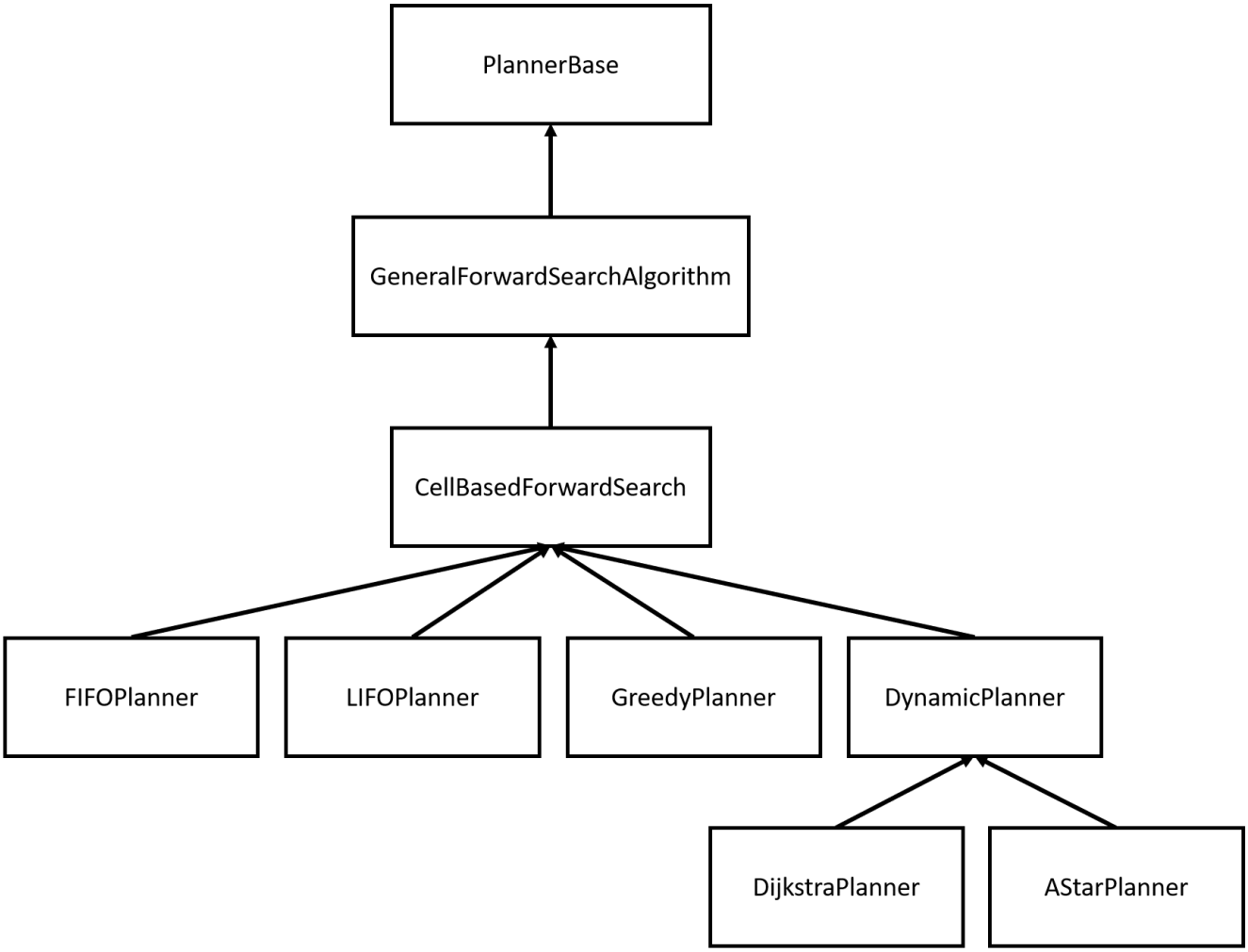
2 Implementation in ROS

controller - Improved on P - Considered PID - Considered reducing the number of waypoints to make controlling better - Sampling rate can be a factor

Appendices

A Class Inheritance

A.1 Planner Inheritance



A.2 Controller Inheritance