SC627 : Assignment 1

Adityaya Dhande **210070005**

I have implemented three planners, **RRT**, **RRT*** and **PRM**. This report contains explanation of my implementation of these planners, followed by some rough hyperparameter analysis. Following this is the summary of results and some conclusions drawn.

1 Code and approach explanation

1.1 Common functions and structures

1.1.1 Graph vertex

This struct is a template for the vertices of a graph which abstracts the different robot configurations. It contains the angles of the robot configuration which the node represents.

```
struct Node
{
    int id;
    vector<int> neighbours;
    vector<double> angles;
    Node(int nodeID) : id(nodeID) {}
};
```

1.1.2 Distance

This function calculates a difference vector as the difference in angles of two configurations, wraps each element of the difference vector to a range of $[0, \pi)$ and then returns the ℓ^2 norm of the difference vector.

```
double distance(Node node1, Node node2, int numOfDOFs){
  double d = 0.0;
  for(int i = 0; i < numOfDOFs; i++){
    double diff = fmod(node2.angles[i] - node1.angles[i] + M_PI, 2 * M_PI) - M_PI;
    d += diff * diff;
  }
  return sqrt(d);
}
```

1.1.3 Generating random sample

This function returns a `Node` object, containing `numOfDOFs` angles. Each of these angles are draw uniformly from $[0, 2\pi]$ except for the first angle, which is drawn from $[0, \pi]$ as the first link will go outside the boundary for angles greater than π.

```
Node generateRandomSample(int numOfDOFs, int dummy){
    Node node(-1);
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> dist(0, 2 * M_PI);
    for(int i = 0; i < numOfDOFs; i++){
        if( i != 0) node.angles.push_back(dist(gen));
        else node.angles.push_back(dist(gen) / 2.0);
    }
    return node;
}
```

1.1.4 Checking for obstacles between two configurations

Given two configurations q_1 and q_2, this function checks if $\bar{q} := \alpha q_1 + (1-\alpha)q_2$ is a valid configuration where α is drawn using a halton sequence of 2. The values of α used are the first `numChecks` number of elements of $\{\frac{1}{2}, \frac{1}{4}, \frac{3}{4}, \frac{1}{8}, \frac{5}{8}, \frac{3}{8}, \frac{7}{8}, \frac{1}{16}, \frac{9}{16} \ldots\}$

```
bool validEdge(Node node1, Node node2, int numOfDOFs, int x_size, int y_size, double *
    map, int numChecks){
    for (int i = 0; i < numChecks; i++) {
        double alpha = halton(i + 1, 2);
        double interpolatedConfig[numOfDOFs];
        for (int j = 0; j < numOfDOFs; j++)
            interpolatedConfig[j] = node1.angles[j] + alpha * (node2.angles[j] - node1.
    angles[j]);
        if (!IsValidArmConfiguration(interpolatedConfig, numOfDOFs, map, x_size, y_size
    ))
            return false;
    }
    return true;
}
```

1.1.5 K-nearest neighbours

This function takes a list of distances and returns the indices of the K smallest entries of the list. It is used to find the K closest nodes from a given node.

```
vector<int> KNN(const vector<double>& values, int k) {
    vector<int> indices(values.size());
    iota(indices.begin(), indices.end(), 0);
    partial_sort(indices.begin(), indices.begin() + k, indices.end(),
                 [&values](int i, int j) { return values[i] < values[j]; });
    return vector<int>(indices.begin() + 1, indices.begin() + k + 1);
}
```

1.1.6 Breadth first search

This function implements the breadth first search graph algorithm to find a path between two nodes. It is used in RRT and PRM to find the path between start and goal in the graph.

```
vector<int> findPath(const vector<Node>& nodes, int start, int goal) {
    vector<bool> visited(nodes.size(), false);
    vector<int> parent(nodes.size(), -1);
    int front = 0, rear = 1;
    vector<int> queue;
    queue.push_back(start);
    visited[start] = true;
    while (front < rear) {
        int current = queue[front];
        front++;
        for (int neighbor : nodes[current].neighbours) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                parent[neighbor] = current;
                queue.push_back(neighbor);
                rear++;

                if (neighbor == goal) {
                    vector<int> path;
                    while (neighbor != -1) {
                        path.push_back(neighbor);
                        neighbor = parent[neighbor];
                    }
                    reverse(path.begin(), path.end());
                    return path;
                }
            }
        }
    }
    return vector<int>();
}
```

1.2 Rapidly-exploring Random Tree (RRT)

The `createRRT` function takes a graph (possibly empty) and adds nodes, while also forming edges, till the graph increases to a desired size.

Algorithm 1 : `createRRT`

```
1: while Number of nodes < Desired number of nodes do
2:     Generate a random configuration $q_{rand}$
3:     Find the node $q_{near}$ in the graph, whose configuration is closest to $q_{rand}$
4:     Calculate the angles corresponding to a potential new configuration $q_{new}$, at a distance of
   STEP_SIZE from $q_{near}$, in the direction of $q_{rand}$
5:     if the path from $q_{near}$ to $q_{new}$ is obstacle free then
6:         Add $q_{new}$ to the graph and an edge between $q_{new}$, $q_{near}$
7:     end if
8: end while
```

In my implementation of **RRT**, first the `createRRT` function is called with the start node and the goal node is connected to all nodes, to which a straight obstacle free path exists, in the neighbourhood of the goal configuration. The neighbourhood consists of K-nearest neighbours of the goal configuration in the graph. Then the BFS graph search algorithm is used to find a path from the start node to the goal node. If no path is found, `createRRT` function is called recursively with the current graph to add more nodes, till a path to the goal is found.

1.3 RRT*

A different structure is used to store the Nodes in **RRT***, which stores the parent of each node and also the cost of each node. The cost of a node is defined as

$$\texttt{cost}(q) = \texttt{cost}(\texttt{parent}(q)) + \texttt{distance}(q, \texttt{parent}(q))$$

```
struct StarNode
{
    int id, parent;
    double cost;
    vector<double> angles;
    StarNode(int nodeID) : id(nodeID) {}
};
```

RRT* is similar to **RRT** with a few changes in how the graph is formed. The way the path is found in **RRT*** is different in my implementation. The path is formed in reverse, by starting from the goal node and moving to the parent node recursively till the start node is reached.

For the graph formation, a potential new node is calculated in the same way as in **RRT**. Instead of connecting this to the closest node in the graph, it is connected to the node with the least cost in its neighbourhood (if the straight line path between the two nodes is obstacle free). The neighbourhood is decided by a fixed radius in my implementation. After the edge is formed with a node in the neighbourhood, I check the remaining nodes in the neighbourhood and see if forming an edge with the new node reduces the cost of those nodes, and change the parent of those nodes accordingly.

1.4 Probabilistic Road Map (PRM)

The `createPRM` function takes a graph (possibly empty) and adds nodes, while also forming edges, till the graph increases to a desired size.

Algorithm 2 : `createPRM`

```
1: while Number of nodes < Desired number of nodes do
2:     Generate a random configuration $q_{rand}$
3:     Add $q_{rand}$ to the graph if it is a valid configuration.
4: end while
5: for node1 in graph do
6:     for node2 in neighbourhood(node1) do
7:         if the straight path between node1 and node2 is obstacle free then
8:             Add an edge between node1 and node2
9:         end if
10:    end for
11: end for
```

In my implementation of **PRM**, first a graph of a fixed size is formed, using `createPRM`, and the start and goal nodes are connected to this graph in a similar way as done with the goal node in **RRT**. Then BFS graph algorithm is used to find a path between the start and goal nodes. If a path is found then it is returned, and if no path is found, `createPRM` is called with the current graph to increase its size by adding more nodes and this process is repeated recursively till a path is found between the start and goal nodes.

1.5 Effect of Hyperparameters

The following are the hyperparameters encountered in the planners :

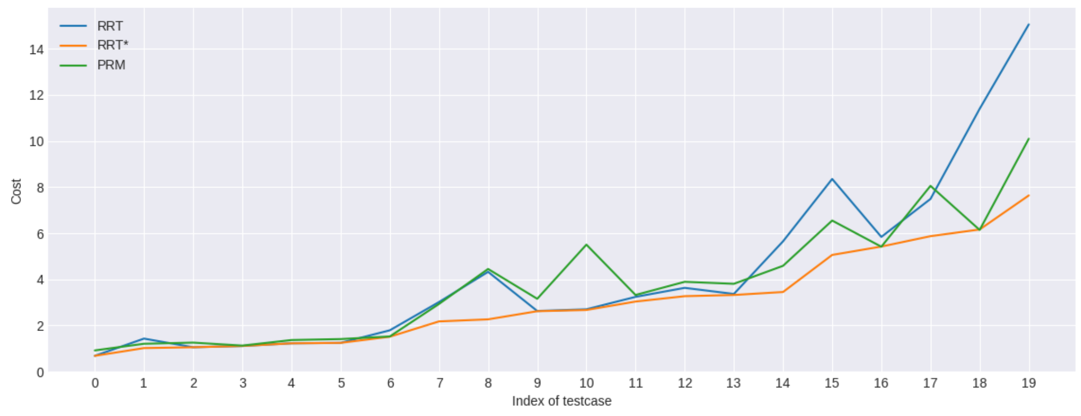
- **Incremental number of nodes** : From observation, increasing the nodes by n and then again by n takes more compute time than increasing nodes by $2n$ in one go, so a low value is not preferred. A high value also causes some unecessary computation when only a few additional nodes would suffice. A low value could also cause the cost to be high if a path is found.
- **Initial number of nodes** : A very high value increases time requirement for a lot of start and goal configurations. A low value will need more increments and causes increase in computation. A low value could also cause the cost to be high if a path is found.
- **Number of neighbours in neighbourhood** : A low value could cause the cost to be high if a path is found. Low value decreases chances of finding a path. High value increases computation.
- **STEP_SIZE in RRT and RRT*** : Small step size causes a denser tree formation which increases the computation where as large step size will not give a fine path and the path will have high cost.

- **Neighbourhood radius in RRT*** : Large neighbourhood radius increases computation and a small neighbourhood radius increases the cost of the path.
- **Number of points to check for obstacle free path** : From observation, this has not significantly affected planning time.

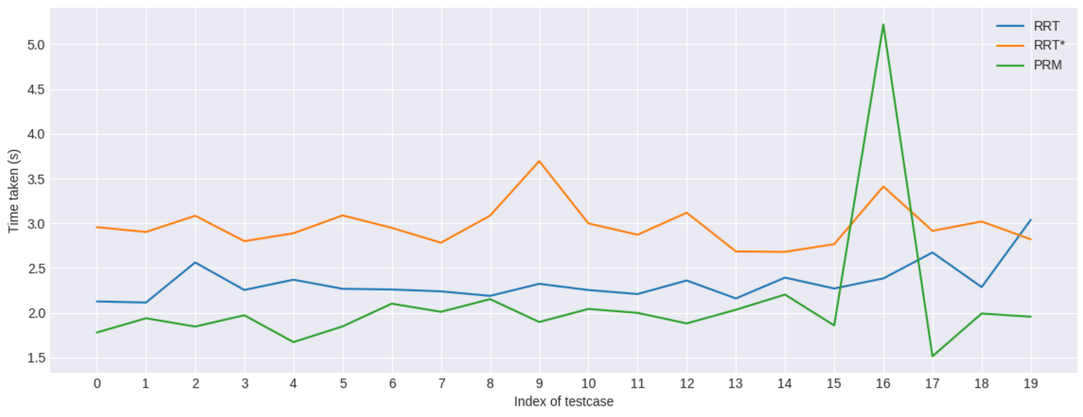
2 Results

Comparision of different planners

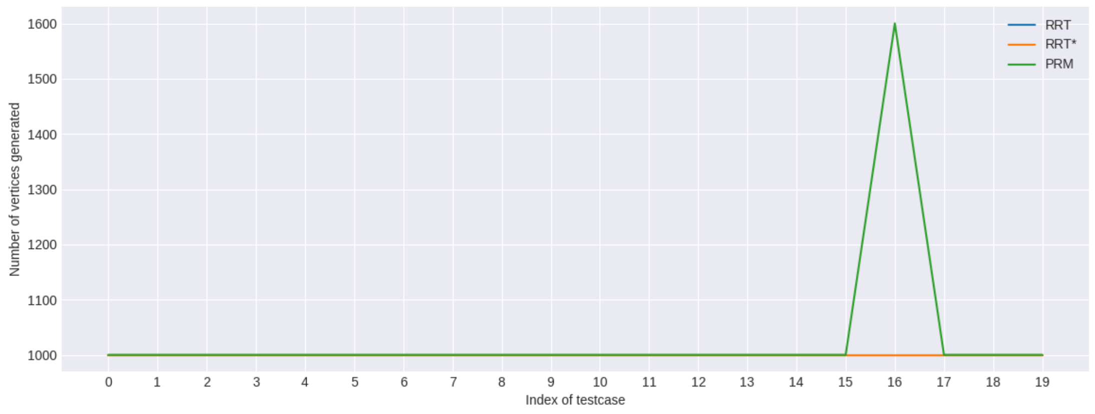
Field	**RRT**	**RRT***	**PRM**
Average planning time	2.337s	2.976s	**2.096s**
Success rate for < 5s	100%	100%	95%
Average path cost	4.263	**3.043**	3.838
Average number of nodes	1000	1000	1030
Max planning time	**3.039s**	3.695s	5.222s
Max path cost	15.056	**7.639**	10.099
Max number of nodes	1000	1000	1600



Costs of different planners



Time taken by different planners



Number of nodes generated by different planners

2.1 Summary and explanation of results

The **number of nodes** is the same for all the planners except for one test case where **PRM** generates a larger number of nodes. The variance in number of nodes for all the planners was more when I used different hyperparameters. For lower **initial number of nodes** and lower **incremental number of nodes** some testcases had lower than 1000 nodes generated but the time taken for many testcases exceeded 5 seconds.
PRM takes lesser **time** on average compared to **RRT** and **RRT*** because it is computationally cheaper. This is because for a given **number of initial nodes**, **PRM** first samples the nodes and then forms the edges. Where as in the other methods the edges are formed and only if a valid edge is found the sample is added to the graph. **RRT** still is faster than **RRT*** as after forming a node in a similar fashion to **RRT**, **RRT*** finds nodes in a **neighbourhood** and forms new edges that minimise the cost of all the edges in the neighbourhood.
The consistent trend in **cost** is that **RRT*** performs better than **RRT** and **PRM**. This is expected because whenever a new node is added, it forms an edge only with the node in its neighbourhood which will give it the lowest cost from the start node. **RRT** and **PRM** form edges with the closest nodes. Using a cost in **PRM** might perform as good as **RRT*** with the use of a weighted graph search algorithm like **Dijkstra's algorithm**.

2.2 Best planner for the environment

- I think **RRT*** planner is the most suitable for the environment as even though it takes more time than **RRT** and **PRM** it is consistent and the time taken is still reasonable. **PRM** has a high variance in the time taken and goes to high values some times. **RRT*** also generates fewer or same number of nodes as **RRT** and **PRM** and the most cost effective path amongst the three planners.
- The issues with my implementation of **RRT*** are on quite a few testcases it gave a path with the almost the same cost as **RRT** and **PRM** while taking relatively more time.
- Additional hyperparameter tuning will improve both the time and cost performance of my implementation of the **RRT*** planner. The **neighbourhood radius** and **STEP_SIZE** significantly affect the time and cost performance of the planner. They can be tuned for a specific environment to improve the performance of the planner.

3 Compiling the code

```
    g++ planner.cpp -o planner.out      # Compiles the planner.cpp
    g++ -O3 planner.cpp -o planner.out  # Compiles planner.cpp using optimised compiler
```

These commands can be used to compile the code.