Advanced Algorithm Implementation: Informed RRT* and D*

By

Ajith Kumar Jayamoorthy

Table of Contents

1. INFORMED RRT*

a. Introduction and Methodology

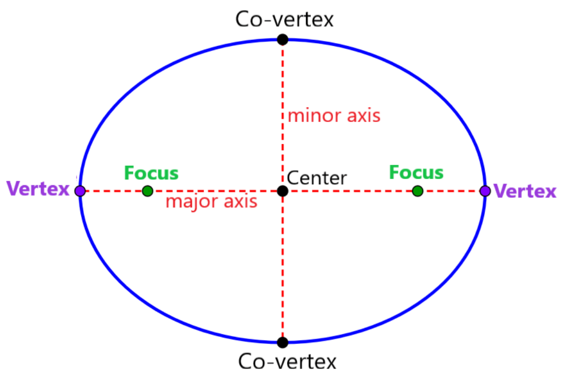
The basic idea behind the INFORMED RRT* is additional constraints in the sampling process. Initially, the informed RRT* starts with the same process as the RRT* algorithm. Like RRT* the new node is connected to the nearest node. Then the neighbors of the new node are considered and the cost-to-come from start to the new node is checked through all the neighbors. The neighbor that provides the least cost-to-come to the new node is chosen as the parent for the new node. Then the connections are rewired such that the cost-to-come for the neighbors are less than the cost-to-come through the new node, if not the new node is assigned as the parent for the neighbor node.

After the path is found, then the sampling process is changed. Instead of sampling randomly from the entire space, the sampling is restricted to an ellipse (or ellipsoid in 3D) with the focal points being the start and the end nodes. Based on this restricted sampling, a new point is obtained. This new point is then used to rewire and replan the entire path.

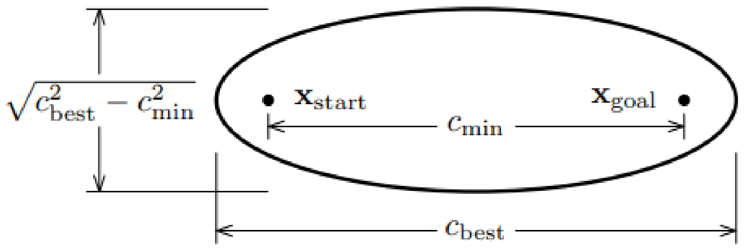
The following Pseudo code for Informed RRT* gives a brief flow of the process:

```
Algorithm 1: Informed RRT*($\mathbf{x}_{\text{start}}, \mathbf{x}_{\text{goal}}$)
$V \leftarrow \{\mathbf{x}_{\text{start}}\}$;
$E \leftarrow \emptyset$;
$X_{\text{soln}} \leftarrow \emptyset$;
$\mathcal{T} = (V, E)$;
for iteration $= 1 \ldots N$ do
    $c_{\text{best}} \leftarrow \min_{\mathbf{x}_{\text{soln}} \in X_{\text{soln}}} \{\texttt{Cost}(\mathbf{x}_{\text{soln}})\}$;
    $\mathbf{x}_{\text{rand}} \leftarrow \texttt{Sample}(\mathbf{x}_{\text{start}}, \mathbf{x}_{\text{goal}}, c_{\text{best}})$;
    $\mathbf{x}_{\text{nearest}} \leftarrow \texttt{Nearest}(\mathcal{T}, \mathbf{x}_{\text{rand}})$;
    $\mathbf{x}_{\text{new}} \leftarrow \texttt{Steer}(\mathbf{x}_{\text{nearest}}, \mathbf{x}_{\text{rand}})$;
    if $\texttt{CollisionFree}(\mathbf{x}_{\text{nearest}}, \mathbf{x}_{\text{new}})$ then
        $V \leftarrow \cup \{\mathbf{x}_{\text{new}}\}$;
        $X_{\text{near}} \leftarrow \texttt{Near}(\mathcal{T}, \mathbf{x}_{\text{new}}, r_{\text{RRT}^*})$;
        $\mathbf{x}_{\text{min}} \leftarrow \mathbf{x}_{\text{nearest}}$;
        $c_{\text{min}} \leftarrow \texttt{Cost}(\mathbf{x}_{\text{min}}) + c \cdot \texttt{Line}(\mathbf{x}_{\text{nearest}}, \mathbf{x}_{\text{new}})$;
        for $\forall \mathbf{x}_{\text{near}} \in X_{\text{near}}$ do
            $c_{\text{new}} \leftarrow \texttt{Cost}(\mathbf{x}_{\text{near}}) + c \cdot \texttt{Line}(\mathbf{x}_{\text{near}}, \mathbf{x}_{\text{new}})$;
            if $c_{\text{new}} < c_{\text{min}}$ then
                if $\texttt{CollisionFree}(\mathbf{x}_{\text{near}}, \mathbf{x}_{\text{new}})$ then
                    $\mathbf{x}_{\text{min}} \leftarrow \mathbf{x}_{\text{near}}$;
                    $c_{\text{min}} \leftarrow c_{\text{new}}$;
        $E \leftarrow E \cup \{(\mathbf{x}_{\text{min}}, \mathbf{x}_{\text{new}})\}$;
        for $\forall \mathbf{x}_{\text{near}} \in X_{\text{near}}$ do
            $c_{\text{near}} \leftarrow \texttt{Cost}(\mathbf{x}_{\text{near}})$;
            $c_{\text{new}} \leftarrow \texttt{Cost}(\mathbf{x}_{\text{new}}) + c \cdot \texttt{Line}(\mathbf{x}_{\text{new}}, \mathbf{x}_{\text{near}})$;
            if $c_{\text{new}} < c_{\text{near}}$ then
                if $\texttt{CollisionFree}(\mathbf{x}_{\text{new}}, \mathbf{x}_{\text{near}})$ then
                    $\mathbf{x}_{\text{parent}} \leftarrow \texttt{Parent}(\mathbf{x}_{\text{near}})$;
                    $E \leftarrow E \setminus \{(\mathbf{x}_{\text{parent}}, \mathbf{x}_{\text{near}})\}$;
                    $E \leftarrow E \cup \{(\mathbf{x}_{\text{new}}, \mathbf{x}_{\text{near}})\}$;
        if $\texttt{InGoalRegion}(\mathbf{x}_{\text{new}})$ then
            $X_{\text{soln}} \leftarrow X_{\text{soln}} \cup \{\mathbf{x}_{\text{new}}\}$;
return $\mathcal{T}$;
```

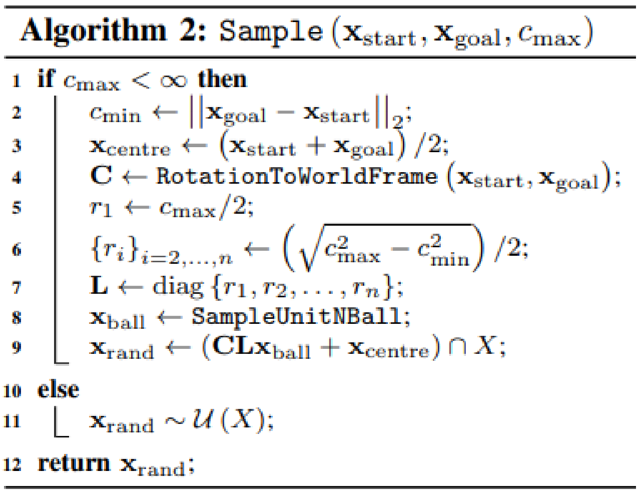
In the above pseudo code, the best path length calculated in each iteration is considered. This path length (**C$_{\text{best}}$**) is assigned as the length of the major axis. The distance between the two focal points is taken as '**C$_{\text{min}}$**'.



From the relation given below, using **C$_{\text{best}}$** and **C$_{\text{min}}$,** the minor axis is calculated.


$\sqrt{c_{\text{best}}^2 - c_{\text{min}}^2}$
$\mathbf{x}_{\text{start}}$
$\mathbf{x}_{\text{goal}}$
c_{min}
c_{best}

Then a point is sampled normally from a unit circle (or sphere in 3D) and then it is transformed into the frame of the ellipse using rotation matrix as shown below in lines 8 and 9:



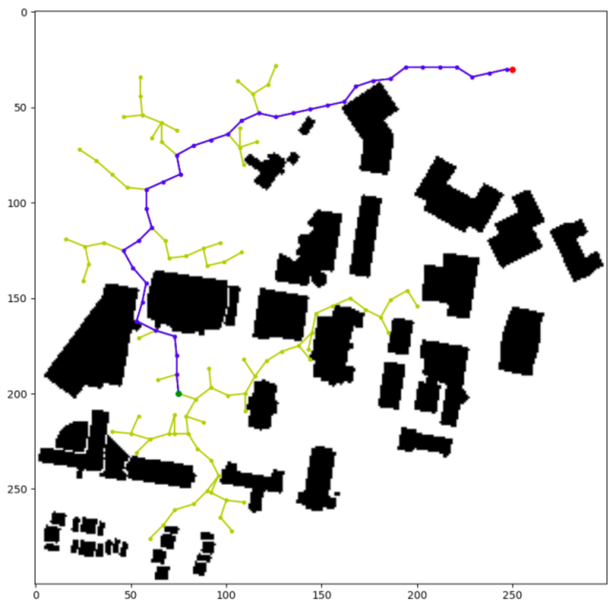
```
Algorithm 2: Sample ($\mathbf{x}_{\text{start}}, \mathbf{x}_{\text{goal}}, c_{\text{max}}$)
1  if $c_{\text{max}} < \infty$ then
2      $c_{\text{min}} \leftarrow \left\|\mathbf{x}_{\text{goal}} - \mathbf{x}_{\text{start}}\right\|_2$;
3      $\mathbf{x}_{\text{centre}} \leftarrow (\mathbf{x}_{\text{start}} + \mathbf{x}_{\text{goal}})/2$;
4      $\mathbf{C} \leftarrow$ RotationToWorldFrame ($\mathbf{x}_{\text{start}}, \mathbf{x}_{\text{goal}}$);
5      $r_1 \leftarrow c_{\text{max}}/2$;
6      $\{r_i\}_{i=2,\ldots,n} \leftarrow \left(\sqrt{c_{\text{max}}^2 - c_{\text{min}}^2}\right)/2$;
7      $\mathbf{L} \leftarrow \text{diag}\{r_1, r_2, \ldots, r_n\}$;
8      $\mathbf{x}_{\text{ball}} \leftarrow$ SampleUnitNBall;
9      $\mathbf{x}_{\text{rand}} \leftarrow (\mathbf{C}\mathbf{L}\mathbf{x}_{\text{ball}} + \mathbf{x}_{\text{centre}}) \cap X$;
10 else
11     $\mathbf{x}_{\text{rand}} \sim \mathcal{U}(X)$;
12 return $\mathbf{x}_{\text{rand}}$;
```

Using this new point, the path is replanned, and the new shortest path is obtained.

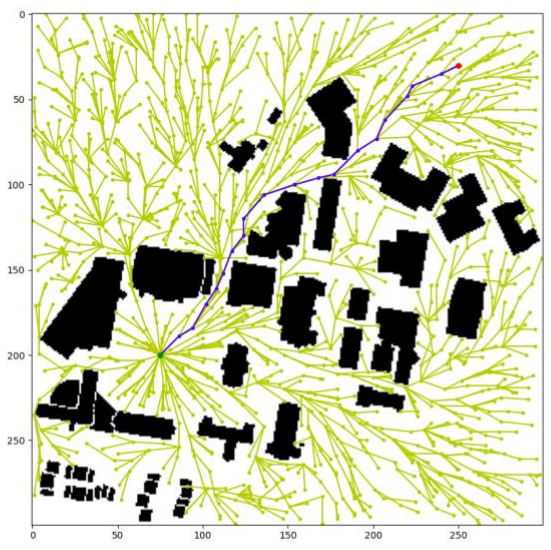
b. Results and Observations

The results shown below are for RRT, RRT* and informed-RRT*:

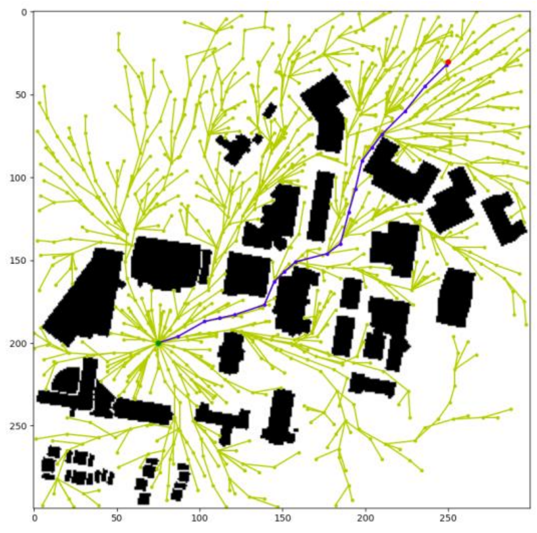
RRT



RRT*



Informed RRT*

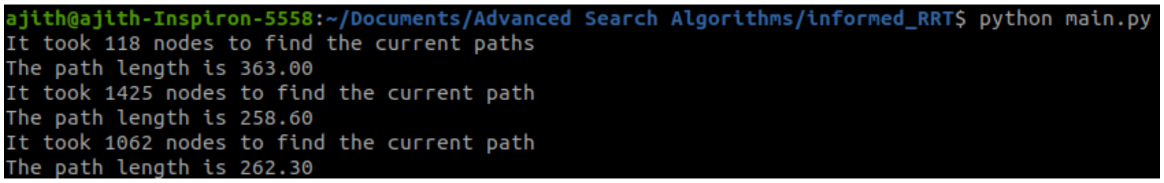


From the above plots it can be observed that the informed RRT* star has spread in a smaller region compared to the RRT* as well as both RRT* and informed RRT* have shorter path than the RRT algorithm.

Key difference between RRT* and informed RRT*:

RRT*	Informed-RRT*
New point is randomly sampled from the entire workspace.	New point is sampled from the restricted workspace (Ellipse/Ellipsoid region).
More number of nodes are explored because of large workspace, to find an optimum path	Less number of nodes are explored to find a equivalent optimum path because of confined sampling.
Path information such as path length is not utilized to help with the sampling process.	Path information such as path length is utilized to restrict the sampling process, as it is considered as major axis of ellipse/ellipsoid.

Advantages of using informed-RRT* over RRT*:



```
ajith@ajith-Inspiron-5558:~/Documents/Advanced Search Algorithms/informed_RRT$ python main.py
It took 118 nodes to find the current paths
The path length is 363.00
It took 1425 nodes to find the current path
The path length is 258.60
It took 1062 nodes to find the current path
The path length is 262.30
```

From the above output, it can be observed that informed-RRT* is able to obtain a significantly close optimum path like the RRT* but it utilizes a smaller number of nodes than the latter. Thus, informed-RRT* is both computationally as well as memory wise more efficient than the standard RRT* algorithm giving almost the same optimum path length.

From the above images of RRT* and informed-RRT*, it can be observed that the sampling region is restricted (within an ellipse) in the latter algorithm, thus reducing unnecessary sampling of nodes outside the required region, for a better path.

2. D*

a. Introduction and Methodology

In this section, the implementation of D* is discussed. Like Dijkstra or A*, D* algorithm also maintains an open list. Each node has a different state tag as follows:

a. NEW
b. OPEN
c. CLOSED
d. RAISE
e. LOWER

Based on each tag, the nodes are processed using the Process_State function. In Process_State function, we initialize the goal's heuristic as zero and then insert into the open list. The function is called until robot's state is removed from the open list. During each process state, the Prepare_Repair function is called. This function checks if the neighbours of the current state is an obstacle or not. If the one of the neighbours is an obstacle.

If one of the neighbours is observed to be an obstacle in the Prepare_Repair function, then the Modify_cost function is called. This function modified the neighbours cost according to the new obstacles. After the neighbours cost are modified, then the Repair-Replan function is called which takes care of repairing the cost of the neighbours by replanning the path.

The following pseudo codes represent the operations that are described above:

<u>D* Algorithm</u>

```
for each $X \in L$ do
    t(X) = NEW
end for
$h(G) = 0$
$INSERT(O, G, h(G))$
$X_c = S$
$P = INIT - PLAN(O, L, X_c, G)$
if P = NULL then
    Return (NULL)
end if
while $X_c \neq G$ do
    $PREPARE - REPAIR(O, L, X_c)$
    $P = REPAIR - REPLAN(O, L, X_c, G)$
    if P = NULL then
        Return (NULL)
    end if
    $X_c$ = the second element of $P$ {Move to the next state in $P$}.
end while
Return ($X_c$)
```

<u>Process_State()</u>

```
X = MIN-STATE( )
if  X= NULL then return –1
$k_{old}$ = GET-KMIN( );  DELETE(X);
if $k_{old}$< h(X) then
    for each neighbor Y of X:
            if t(Y) ≠ new and  h(Y) <= $k_{old}$ and h(X) > h(Y) +
c(Y,X) then
                b(X) = Y;  h(X) = h(Y)+c(Y,X);
if $k_{old=}$ h(X) then
    for each neighbor Y of X:
            if t(Y) = NEW or
            (b(Y) =X and h(Y) ≠ h(X)+c (X,Y) ) or
            (b(Y) ≠ X and h(Y) > h(X)+c (X,Y) ) then
            b(Y) = X ; INSERT(Y, h(X)+c(X,Y))
else
    for each neighbor Y of X:
            if t(Y) = NEW or
            (b(Y) =X and h(Y) ≠ h(X)+c (X,Y) ) then
                b(Y) = X ; INSERT(Y, h(X)+c(X,Y))
            else
                if  b(Y) ≠ X and h(Y) > h(X)+c (X,Y) then
                    INSERT(X, h(X))
                else
                    if  b(Y) ≠ X and h(X) > h(Y)+c (X,Y) and
                    t(Y) = CLOSED and h(Y) > $k_{old}$ then
                    INSERT(Y, h(Y))
Return GET-KMIN ( )
```

```
$MODIFY - COST(O, X, Y, cval)$
1: $c(X, Y) = cval$
2: if $t(X) = CLOSED$ then
3:     $INSERT(O, X, h(X))$
4: end if
5: Return $GET - KMIN(O)$
```

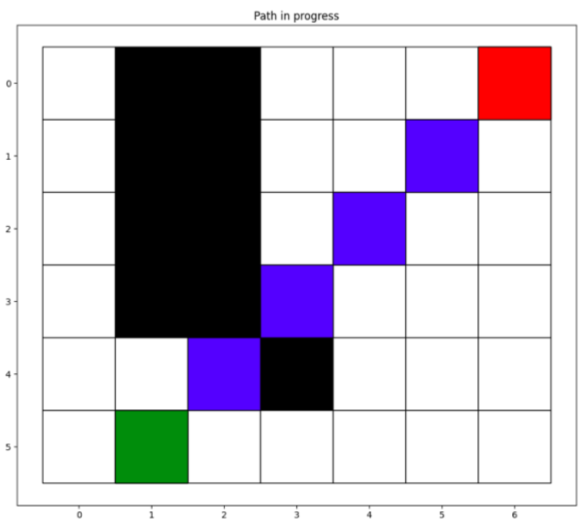
```
$REPAIR - REPLAN(O, L, X_c, G)$
1: repeat
2:     $k_{min} = PROCESS - STATE(O, L)$
3: until $(k_{min} \geq h(X_c))$ or $(k_{min} = -1)$
4: $P = GET - BACKPOINTER - LIST(L, X_c, G)$
5: Return $(P)$
```

```
$PREPARE - REPAIR(O, L, X_c)$
1: for each state $X \in L$ within sensor range of $X_c$ and $X_c$ do
2:     for each neighbor $Y$ of $X$ do
3:         if $r(Y, X) \neq c(Y, X)$ then
4:             $MODIFY - COST(O, Y, X, r(Y, X))$
5:         end if
6:     end for
7:     for each neighbor $Y$ of $X$ do
8:         if $r(X, Y) \neq c(X, Y)$ then
9:             $MODIFY - COST(O, X, Y, r(X, Y))$
10:        end if
11:    end for
12: end for
```

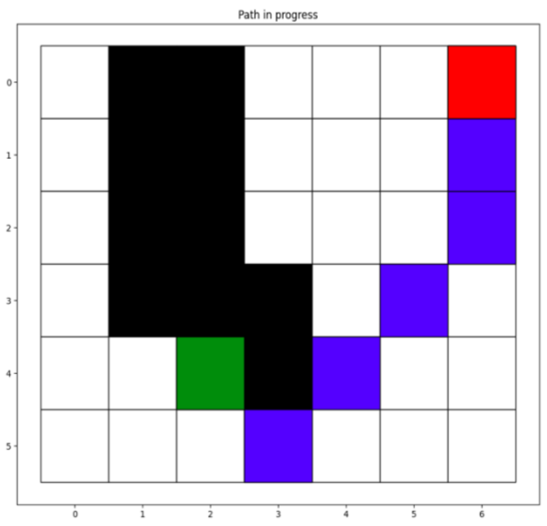
b. Results and Observations

SCENARIO -1

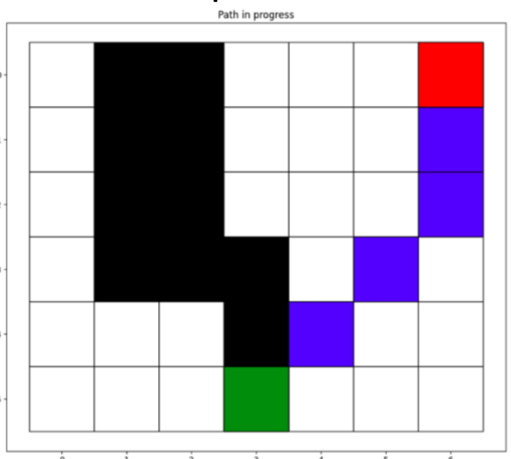
Step – 1



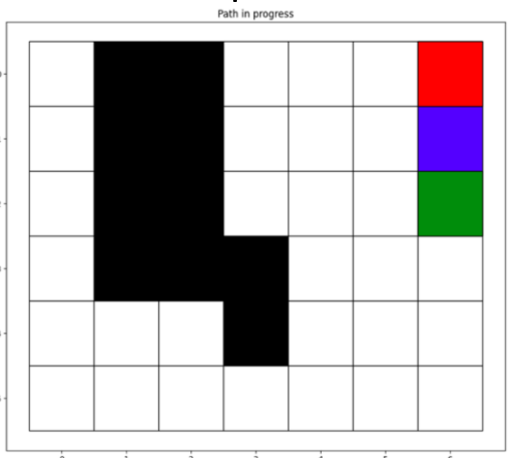
Step – 3



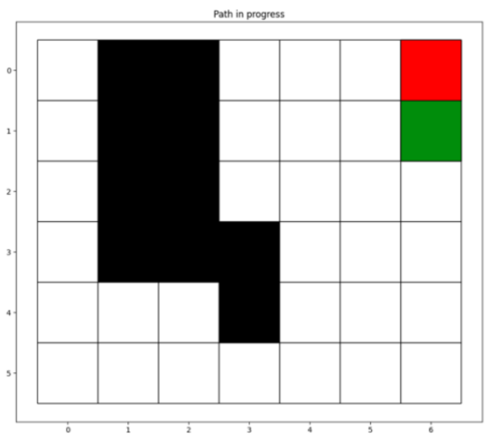
Step – 4



Step – 7

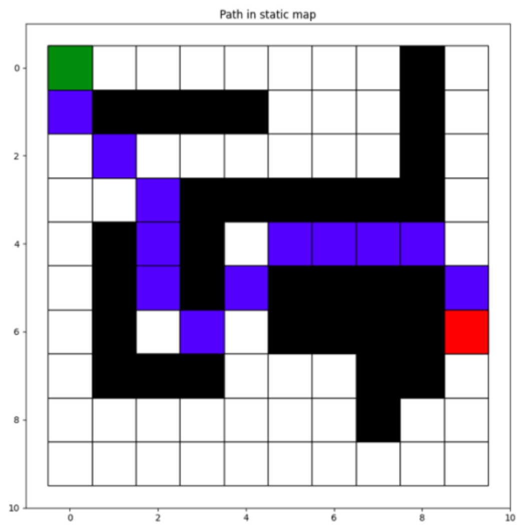


Step – 8

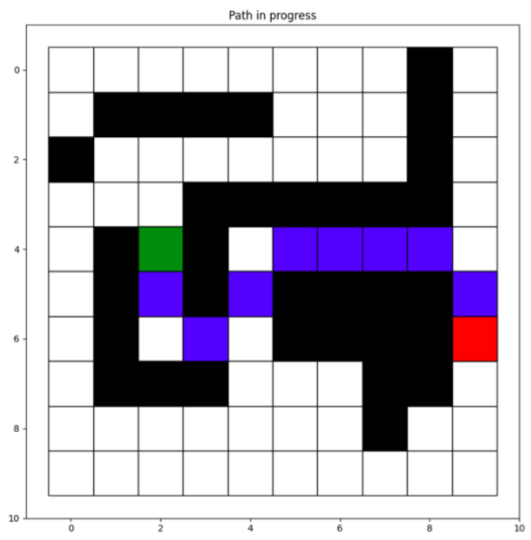


SCENARIO – 2

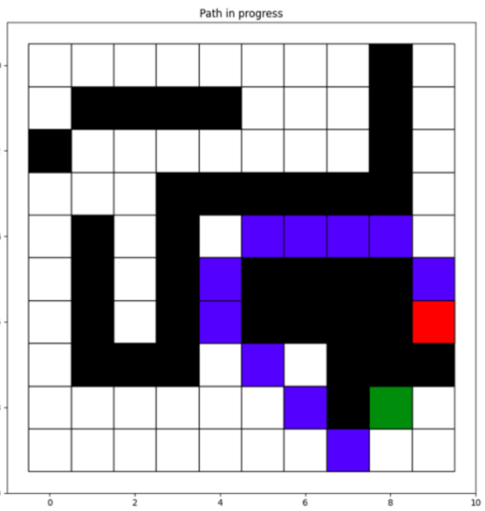
Step – 1



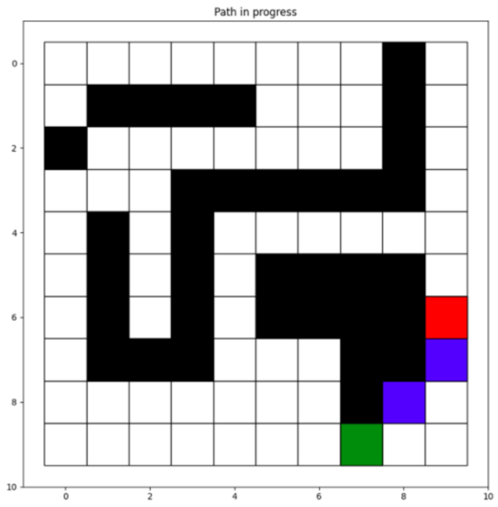
Step – 6



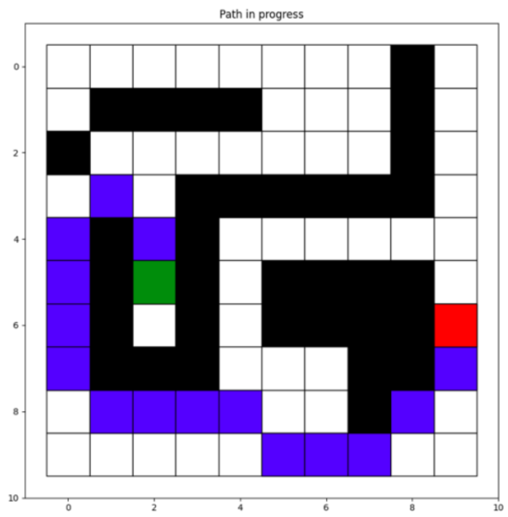
Step – 7



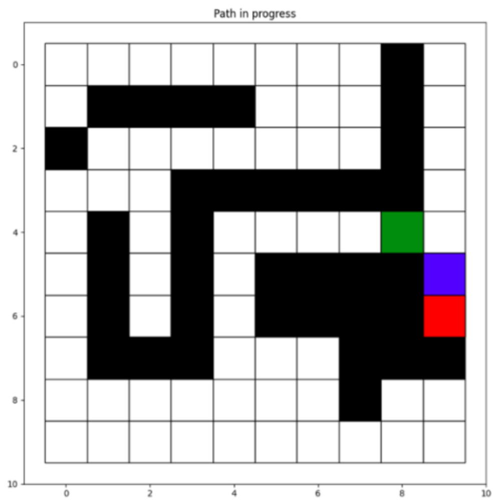
Step – 20



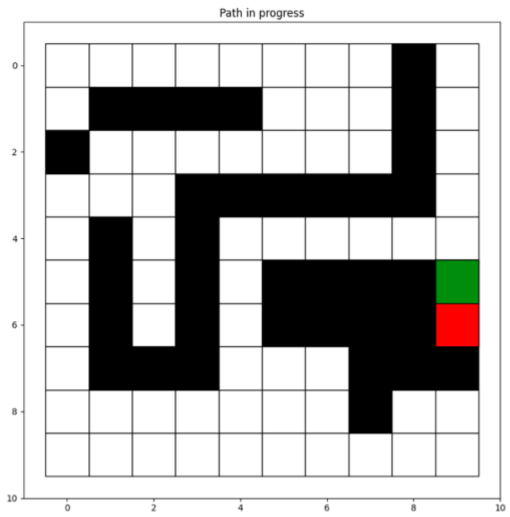
Step – 21



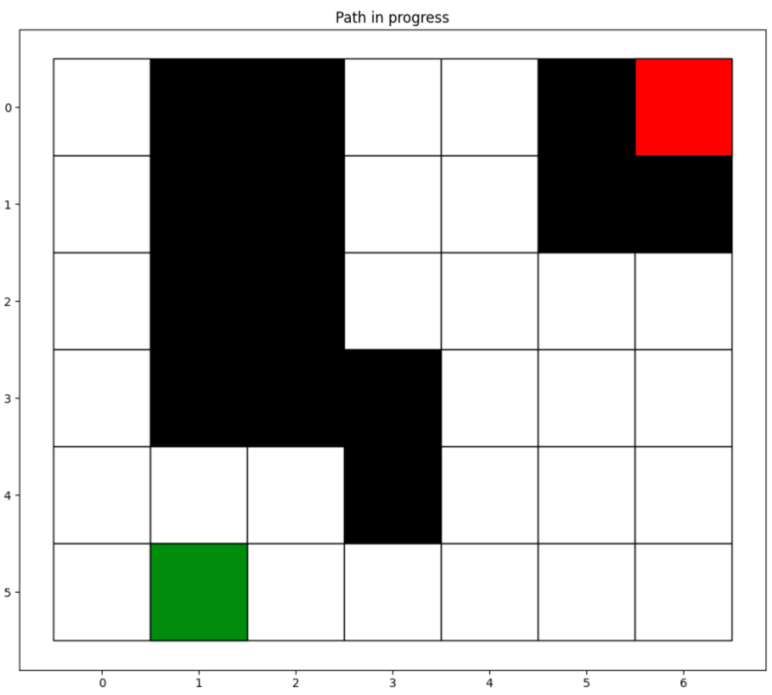
Step – 30



Step – 31

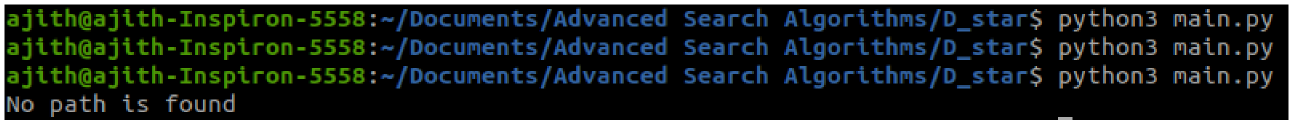


SCENARIO – 3



<u>Outputs</u>

For three different scenarios/maps, the following results were obtained:



```
ajith@ajith-Inspiron-5558:~/Documents/Advanced Search Algorithms/D_star$ python3 main.py
ajith@ajith-Inspiron-5558:~/Documents/Advanced Search Algorithms/D_star$ python3 main.py
ajith@ajith-Inspiron-5558:~/Documents/Advanced Search Algorithms/D_star$ python3 main.py
No path is found
```

<u>D* vs A* or Dijkstra</u>

1. A* and Dijkstra are used for static environment whereas D* is used for dynamic environment.
2. In case we use A* and Dijkstra in the dynamic environment, then the entire path must be replanned from the current position to the goal position. During this, the entire environment is again re-evaluated. Whereas in the case of the D* algorithm, the changes are made locally when the obstacle is faced and based on the change, a different path is followed which is optimum from the changed direction. Thus, it can be observed that A* and Dijkstra are computationally more expensive than D*. Therefore, D* is comparatively faster than A* and Dijkstra algorithm.