Comparison of Path Planning Algorithms

Bireshwar Das
RBCCPS
IISC
bireshwardas@iisc.ac.in

Jacob James K
CSA
IISC
jacobk@iisc.ac.in

Prashil Wankhede Rajesh
EE
IISC
prashilw@iisc.ac.in

Tailor Yash Snehal
RBCCPS
IISC
tailorsnehal@iisc.ac.in

Abstract—**Path-planning is an important primitive for autonomous mobile robots that lets robots find the shortest – or otherwise optimal – path between two points. optimal paths could be paths that minimize the amount of turning, the amount of braking or whatever a specific application requires. Algorithms to find a shortest path are important not only in robotics, but also in network routing, video games and gene sequencing.Path-planning requires a map of the environment and the robot to be aware of its location with respect to the map. The goals of this project are:**

- **introduce suitable map representations**
- **explain basic path-planning algorithms: Dijkstra, A*, RRT and RRT***
- **Comparisons between these 4 algorithms.**
- **Demonstration of these algorithms using simulations and in 3D environment using PyBullet.**

I. Introduction

The problem to find a “shortest” path from one vertex to another through a connected graph is of interest in multiple domains, most prominently in the internet, where it is used to find an optimal route for a data packet. The term “shortest” refers here to the minimum cumulative edge cost, which could be physical distance (in a robotic application), delay (in a networking application) or any other metric that is important for a specific application. One of the earliest and simplest algorithms is Dijkstra’s algorithm. Starting from the initial vertex where the path should start, the algorithm marks all direct neighbors of the initial vertex with the cost to get there. Reaching a destination via the shortest route is a daily activity we all do. A-star (also referred to as A*) is one of the most successful search algorithms to find the shortest path between nodes or graphs. It is an informed search algorithm, as it uses information about path cost and also uses heuristics to find the solution. A* achieve optimality and completeness, two valuable property of search algorithms. A* and D* become computationally expensive when the search space is large.
A more recent development known as Rapidly-Exploring Random Trees (RRT) addresses this problem by using a randomized approach that aims at quickly exploring a large area of the search space with iterative refinement. For this, the algorithm selects a random point in the environment and connects it to the initial vertex. Subsequent random points are then connected to the closest vertex in the emerging graph. The graph is then connected to the goal node, whenever a point in the tree comes close enough given some threshold. Although generally a coverage algorithm (see also below), RRT can be used for path-planning by maintaining the cost-to-start on each added point, and biasing the selection of points to occasionally falling close to the goal. RRT can also be used to take into account non-holonomic contraints of a specific platform when generating the next random way-point. One way to implement this is to calculate the next random point by applying random values to the robot’s actuators and use the forward kinematics to calculate the next point. Most recently, variations of RRT have been proposed that will eventually find an optimal solution. Nevertheless, although RRT quickly finds some solution, smooth paths usually require additional search algorithms that start from an initial estimate provided by RRT. RRT* is an optimized version of RRT. RRT* does the rewiring of the tree. After a vertex has been connected to the cheapest neighbor, the neighbors are again examined. Neighbors are checked if being rewired to the newly added vertex will make their cost decrease. If the cost does indeed decrease, the neighbor is rewired to the newly added vertex. This feature makes the path more smooth.

II. Path Planning Algorithms

Search Based Path Planning

Dijkstra algorithm: A very common algorithm for calculating the shortest distance between two nodes is Dijkstra algorithm. Using the Dijkstra algorithm, it is possible to determine the shortest distance (Lowest Cost) between a start node and any other node in the map or maze. The idea of the algorithm is to continuously calculate the shortest distance beginning from a starting point and to exclude longer distances when making an update. It consists of following steps:

1) Initialization of all nodes with distance ”infinite”; initialization of the starting node with 0
2) Marking of the distance of the starting node as permanent, all other distances as temporarily.
3) Setting of starting node as active.
4) Calculation of the temporary distances of all neighbour nodes of the active node by summing up its distance with the weights of the edges.
5) If such a calculated distance of a node is smaller as the current one, update the distance and set the current node as antecessor. This step is also called update and is Dijkstra’s central idea.

6) Setting of the node with the minimal temporary distance as active. Mark its distance as permanent.
7) Repeating of steps 4 to 7 until there aren't any nodes left with a permanent distance, which neighbours still have temporary distances.

[4]

DIJKSTRA

```
Function Dijkstra(Graph, source):
for each vertex v in Graph:
    do dist[v] = infinity
        previous[v] = undefined
        dist[source] = 0
        Q = the set of all nodes in Graph
        while Q is not empty:
            do u = node in Q with smallest dist[ ]
                remove u from Q
                for each neighbor v of u:
                    do alt = dist[u] + dist-between(u, v)
                        if alt < dist[v]
                            then dist[v] = alt
                                previous[v] = u
                                return previous[ ]
```

Disdvantages of Dijkstra:
We see there is no heuristics involved in Dijkstra algorithm as it is a brute force search so finding path through this algorithm is more computationally involved as it will search through all the maze in all possible directions. There by resulting more time in computation and many times resulting in more distant roads (totally in different direction) when cost is also dependent on traffic as well. So that inspires in finding shortest distance in the direction (Heuristic) of the destination node from the starting node.

***A* algorithm**:* The uninformed search such as Dijkstra has no information about the environment and is brute force search. This uninformed search is a very lengthy process because there is no information to move on except what is the goal node and what is not a goal node. This lengthy process also needs a large memory to operate and find a solution. An informed search is a better solution to a problem and an even better solution is a heuristic solution. The heuristic function is a method where each node of a branching step and evaluated based on the information known at that node in the path. A heuristic is a rule that helps to find a solution. In this case, the heuristic is an estimate of the distance from the current position on the map to the goal. Algorithms such as A* and D* are examples of heuristic searches.

Components of the A* Algorithm: There are two main components that make up A* algorithm:

- **OPEN list:** It keeps track of the nodes that need to be explored and the list begins with the start node. OPEN consists of nodes that have been visited but not expanded (meaning that successors have not been explored yet). This is the list of pending tasks.
- **CLOSE list:** It keeps track of the nodes that have been explored and the list begins empty. The CLOSE list also consists of all the nodes that have been explored fully and determined to have too high of a cost to move to or a dead end from the goal. The CLOSE list starts empty of nodes.

The decision to move from one node to another is dependent on the f(n) value. The f(n) value for a node 'n' is determined by using the equation:

$$f(n) = g(n) + h(n)$$

g(n) is the cost of the path from the start node to n.
h(n) is the heuristic function that estimates the cost from n to goal, and lastly
f(n) is the sum of g(n) and h(n).
This is an estimate for the complete cost of travelling from the starting point, via node 'n', onwards to the goal. The lower f is, the better we think this path is likely to be. At each stage, the A* algorithm chooses which node to explore by selecting the one with the lowest f(n) value. The nodes with the lowest f(n) values make up the path taken from start to finish.

Dijkstra's Algorithm is a special case of A Search Algorithm, where h = 0 for all nodes.*

Choosing a heuristic for A*:
A heuristic that estimates a value that is guaranteed to be less than or equal to the actual h value it is estimating is called an admissible heuristic. For A* to be optimal the heuristic it uses must be admissible, meaning that if it never overestimates the actual cost to get to the goal, A* is guaranteed to return a least-cost path from start to goal. [8] For choosing a heuristic we can do things:

- Either calculate the exact value of h (which is certainly time consuming).

OR

- Approximate the value of h using some heuristics (less time consuming).

A) Exact Heuristics

We can find exact values of h, but that is generally very time consuming. Below are some of the methods to calculate the exact value of h.

1) Pre-compute the distance between each pair of cells before running the A* Search Algorithm.
2) If there are no blocked cells/obstacles then we can just find the exact value of h without any pre-computation using the distance formula/Euclidean Distance.

B) Approximation Heuristics
There are generally three approximation heuristics to calculate h

- **Manhattan Distance:** It is nothing but the sum of absolute values of differences in the goal node's x and y coordinates and the current node's x and y coordinates respectively.
  When to use this heuristic? When we are allowed to move only in four directions only (right, left, top, bottom)

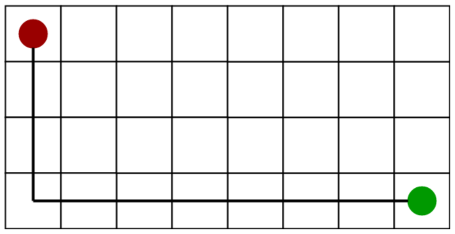


Fig. 1. Manhattan Distance

- **Euclidean Distance:** As it is clear from its name, it is nothing but the distance between the current node and the goal node calculated using the distance formula.
  When to use this heuristic? – When we are allowed to move in any directions.

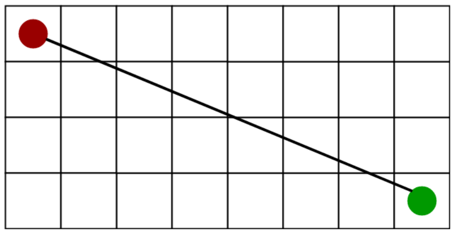


Fig. 2. Euclidean Distance

A*

```
1  Make an OPEN list containing only the starting
   node and set its value f(n_start)=h(n_start).
2  Make an empty CLOSE list.

3  while Consider the node with the lowest f(n) value
   in the OPEN list.
4      do Consider the node with the lowest f(n)
         value in the OPEN list.
5          if this node is our goal node
6            then
7                We are done and reached the goal node.
8            else
9                Put the current node in the CLOSE list
                 and look at all of its neighbours.
```

```
1  for each neighbour of the current node
2      do if neighbour has lower g(n) value than
                the current and is in the CLOSE list
3          then
4              Replace the neighbour with the new,
                 lower g(n) value. Current node is now the
                 neighbour's parent.

5          else  if current g(n) value is lower and this
                 neighbour is in the OPEN list
6                then
7                    Replace the neighbour
                       with the new , lower g(n) value.
                       Change the neighbour's parent
                       to our current node.
8                else
9                    Add it to the open list and
                     set its g(n) value.
```

Disdvantages of A:*
A* can only work if the entire environment is known.It works only on static environments meaning that the obstacle positions are fixed. One of the disadvantages of A* is inability to react to unexpected added or moving obstacles in the testing area. [5] A* becomes computationally inefficient for large environments. It requires more time when the grid size is very large or is expanding. The sampling based methods overcomes these issues.

Random Sampling Based Path Planning

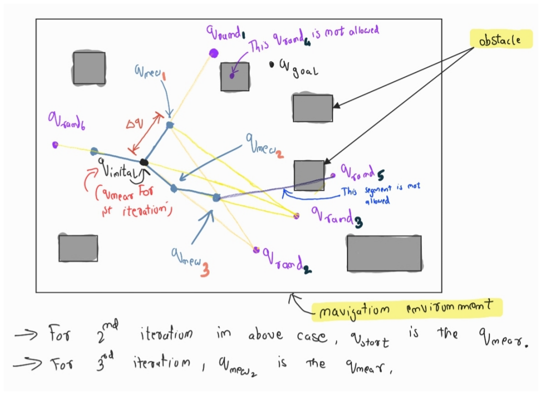
RRT algorithm: RRT(Rapidly Exploring Random Tree) [2] was developed by Steven M. LaValle and James J. Kuffner Jr. RRT is a sampling based algorithm for obtaining a solution path from initial point(root) to targated point when the number of iteration tend to infinity, if the path exits.

RRT plans a navigation path in a Q configuration space, which is the set of points/positions of a navigation environment. Q is divided in two subsets, $Qfree$, representing the navigable regions of the navigation environment, ie., the regions without obstacles and collision risks, and $Qobs$, the spatial representation of the obstacles.

The root node of the tree i.e, $qinit$ belong to $Qfree$ is the starting point of the path to be planned. The algorithm works by expanding a G tree randomly from the root node until one of its branches reaches the final point $qgoal$ which also belong to $Qfree$ of the navigation environment,or until a maximum number of iterations k is reached.

RRT [2]

```
1   G.init(qinit)
2   for k = 1 to K
3       do
4           qrand = RAND_CONF()
5           if IsInObstacle(qrand) == True
6             then
7                 continue
8           qnear = NEAREST_VERTEX(qrand, G)
9           qnew = NEW_CONF(qnear, qrand, $\Delta$q)
10          G.add_vertex(qnew)
11          G.add_edge(qnear, qnew)
12          if qnew in qgoal
13            then return G
14  return G
```



q_{rand_1}
This q_{rand_4} is not allowed
obstacle
q_{goal}
q_{new_1}
q_{rand_6}
Δq
q_{new_2}
$q_{initial}$
(q_{near} for 1st iteration)
q_{rand_5}
This segment is not allowed
q_{rand_3}
q_{new_3}
q_{rand_2}
navigation environment
→ For 2nd iteration in above case, q_{start} is the q_{near}.
→ For 3rd iteration, q_{new_2} is the q_{near},

Fig. 3. RRT Algorithm

Advantage

- Even if any $qrand$ is in a direction opposite to $qgoal$, it will not affect the path which is on its way to the qgoal.
- For large size environment RRT outputs the path much quicker than A* or Dijkstra.

Disadvantage

- As the G tree grows, the computational cost of finding the $qnear$ grows.
- It doesn't have the ability to "look-ahead" for unidentified obstacles, therefore this algorithm is best suited for robots that are in a fixed environment like a factory.

RRT*: RRT* [6] follows a similar set of procedures as in RRT by maintaining a tree of nodes and computing a path from the source to the destination. However, an extra procedure called REWIRE is used in RRT*, which reconstructs the tree so that the distance to a particular node in the tree from the starting point is as minimized as possible. The procedure REWIRE is called after a new vertex qnew is added to the tree

REWIRE(rrt_tree, q_new)

```
1   d = distance of q_new from start
2   Neighbours = nodes within a radius r of
    q_new other than its parent

3   for neighbour in Neighbours
4       do
5           new = distance(qnew) + EUCLID(qnew,neighbour)
6           if distance(neighbour) $>$ new
7             then
8                 distance(neighbour) = new
9                 parent(neighbour) = q_new
10  return rrt_tree
```

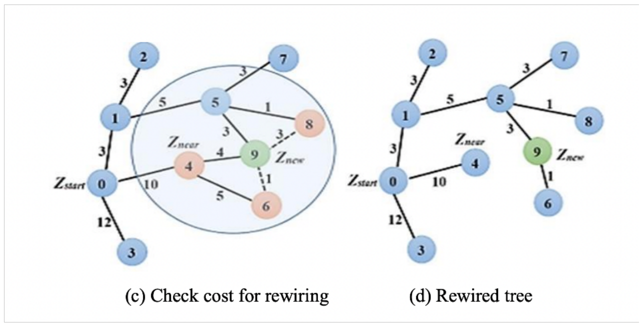


Fig. 4. Rewiring Procedure RRT* [7]

The usage of rewire makes the graph narrower e each time the rewire procedure is called. The optimality of RRT* depends on the number of times the rewire procedure is called which depends on the parameter r of the algorithm. If r is too small, we will not be getting substantial benefits in using RRT*.

Differential Drive Kinematics

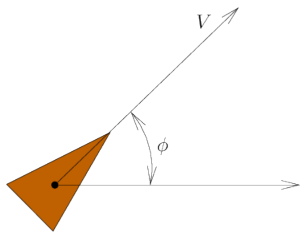


Fig. 5. UnicycleModel

A Differential Drive Robot [8] [9] consists of two wheels mounted on a common axis and each wheel can independently being driven either forward or backward. The Differential Drive Robot can be approximated as a unicycle model whose kinematic equations can be derived as

$$\dot{x} = V cos(\phi)$$

$$\dot{y} = V sin(\phi)$$

where ϕ is the yaw

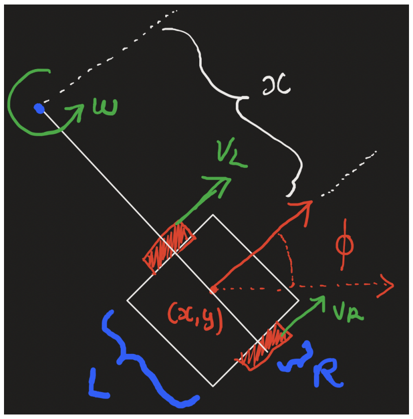


Fig. 6. Differential Drive Robot

The velocity at the centre is equal to the average velocity of both the wheels.

$$V = \frac{V_l + V_r}{2}$$

$$\omega\left(x - \frac{L}{2}\right) = V_l \tag{1}$$

$$\omega\left(x + \frac{L}{2}\right) = V_r \tag{2}$$

From equations 1 and 2

$$x = \frac{V_l + V_r}{2 * \omega} = \frac{V}{\omega}$$

Substituting in (1) and (2)

$$V_l = V - \frac{\omega L}{2}$$

$$V_r = V + \frac{\omega L}{2}$$

$$\begin{bmatrix} V_l \\ V_r \end{bmatrix} = \begin{bmatrix} 1 & -\frac{L}{2} \\ 1 & \frac{L}{2} \end{bmatrix} \begin{bmatrix} V \\ \omega \end{bmatrix}$$

The control equation would be

$$\begin{bmatrix} \omega_l \\ \omega_r \end{bmatrix} = \frac{1}{R} \begin{bmatrix} 1 & -\frac{L}{2} \\ 1 & \frac{L}{2} \end{bmatrix} \begin{bmatrix} V \\ \omega \end{bmatrix}$$

where R is the radius of the wheels and L is. the distance of seperation.

$$\omega = \frac{R(V_r - V_l)}{L}$$

Experimental SetUp

We simulated the four path planning Algorithms on a 2D plane with random obstacles. The program written for simulation were in Python using the python modules Matplotlib (Search Based Planning) and Pygame(Random Sampling Based Planning)

A 3D simulation of Dijkstra's Algorithm was done on the Bullet Physics Engine using the PyBullet [10] API. The simulation was done on a Differential drive robot having two wheels whose control equations have been specified in the previous section. We have not employed any control algorithms for determining the linear velocity and yaw rate of the robot as the environment is more or less static and we have used constant values for both linear and angular velocity.

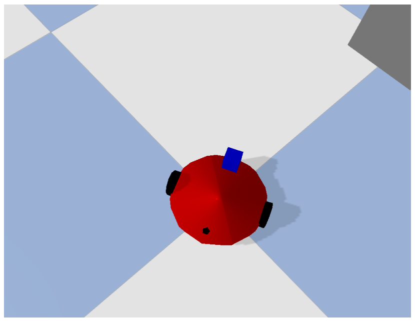


Fig. 7. Unicycle Robot

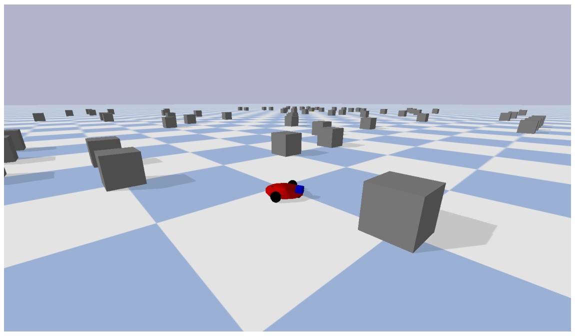


Fig. 8. Robot Navigation in Physics Engine

The source code of our implementations is available on https://github.com/JACOBIN-SCTCS/CP214_Project

Results

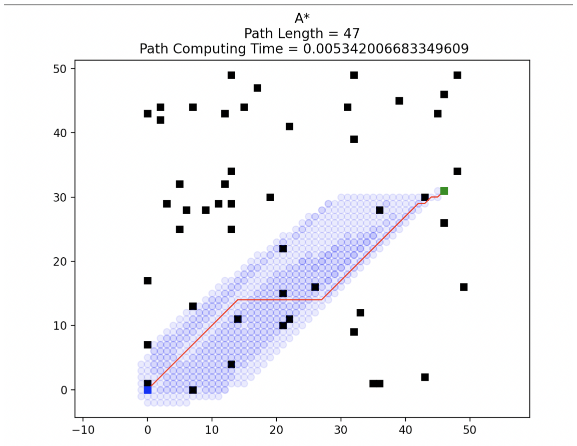


Fig. 9. Simulation result of Astar on 50 x 50 grid

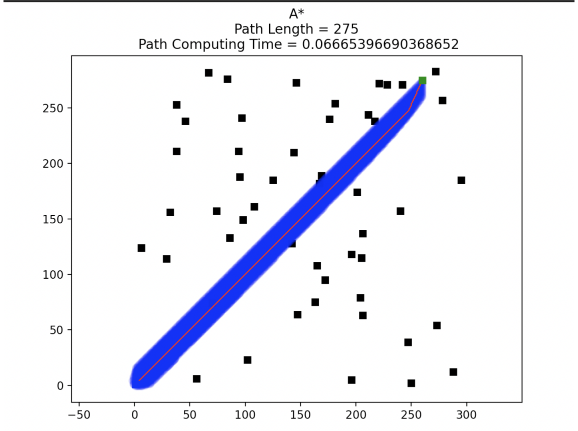


Fig. 10. Simulation result of A* on 300x300 grid

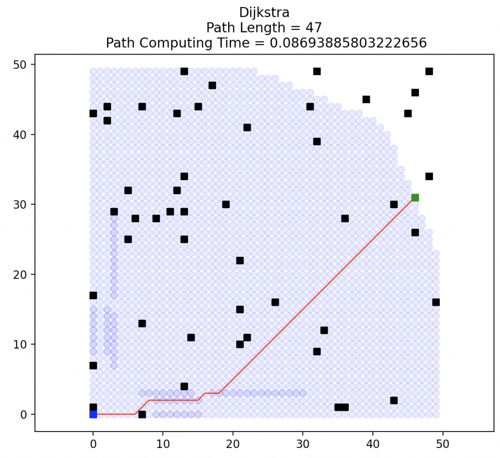


Fig. 11. Simulation result of Dijkstra on 50 x 50 grid

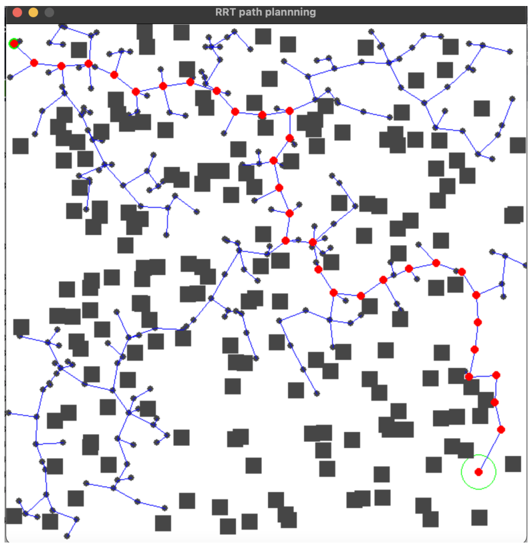


Fig. 12. Simulation result of RRT on 650 x 650 grid

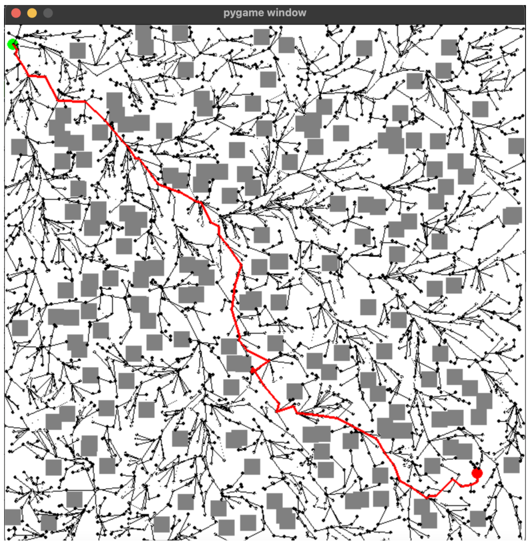


Fig. 13. Simulation result of RRT* on 650 x 650 grid

Parameters	RRT	RRT*	Dijkstra	A*
grid size	650x650	650x650	50x50	50x50
Start node	(10,25)	(10,25)	(0,0)	(0,0)
Goal node	(590,560)	(590,560)	(46,31)	(46,31)
Obstacle count	200	200	20	20

Metric	RRT	RRT*
Simulation Time(in sec)	4.6292	32.9125
Distance	1107.8410	1000.2230

Metric	Dijkstra	A*(50x50)	A*(300x300)
Simulation Time(in sec)	0.0869388	0.005342	0.0666
Distance	47	47	275

Observations

1) The path length found out by applying A* and Dijkstra algorithm is equal when applied on same environment. It signifies that both of them give out the shortest(optimal) cost path in the maze.
2) when applied on the exact same environment,paths found by Dijkstra and A* may be different but the total cost of traversal is same in both cases as both are supposed to find out the least cost path and the anomaly is only due to the directionality feature (Heuristics) involved in A*.
3) In Dijkstra, we see that the algorithm searches whole environment(it is brute force search) and in A* ,the algorithm searches the environment only in a particular direction which is directed towards the goal(the expanding blue region in A* is directed only towards the goal), So the path computing time is smaller in case of A* than Dijkstra algorithm.
4) Both Dijkstra and A* are computationally expensive in case of large environments as it searches through all points of space (in a particular direction or through the whole space) so more time consuming in case of

large environment. This is verified in results section where A* algorithm when run on 50x50 grid takes 0.005342 seconds and when run on 300 x 300 grid takes 0.0666 seconds. Thus it can be observed that A* is approximately 12 times slower when run on 300x300 sized grid. Thus, as grid size increases A* takes longer time for path computing.

5) RRT path planning algorithm is a random search process so in large environments path finding through the RRT algorithm is computationally more efficient and fast.
6) RRT algorithm is random sampling based, it doesn't provide shortest or any optimal path, rather we don't get unique path for multiple simulations in the same environment.
7) Due to RRT algorithm's random sampling of points, A* and Dijkstra algorithm is faster (most of the time) in small environments but RRT is much faster in larger environment involving various obstacles.
8) RRT* algorithm only differs from RRT algorithm in terms of the process of rewiring.
9) RRT* algorithm generate smoother paths. The smoothness of the path found by RRT* depends on the number of times the rewiring process is called.
   If rewiring is called lesser number of times then RRT* behaves more like RRT.
10) RRT* is slower than RRT due to the fact that RRT* optimizes (in terms of sharpness of the path) the path obtained by the RRT algorithm by calling rewiring.
11) Due to rewiring process RRT* requires more memory that the RRT algorithm.

Conclusions

For small environments search based algorithm are more efficient than sampling based algorithms. Out of Dijkstra and A*, Dijkstra is computationally expensive than A* because of brute force search, so A* is preferred over Dijkstra for small environments. For large environment random sampling based algorithms are more efficient than search based algorithms .So for large environments RRT RRT* algorithms have clear advantage over A* Dijkstra algorithms in terms of path computation time. Path obtained by RRT is quite zigzag, to get a smoother path RRT* can be used but RRT* is computationally expensive. To get a smoother path with less computation we can use rewiring for limited number of times.

References

[1] https://www.geeksforgeeks.org/a-search-algorithm/

[2] LaValle, S. M. 1998b. Rapidly-exploring random trees: A new tool for path planning. Report No. TR 98-11, Computer Science Department, Iowa State University.

[3] Véras, Luiz Gustavo D. O. et al. "Systematic Literature Review of Sampling Process in Rapidly-Exploring Random Trees." IEEE Access 7 (2019): 50933-50953.

[4] http://www.gitta.info/Accessibiliti/en/html/DijkstralearningObject1.html

[5] AN ANALYSIS OF PATH PLANNING ALGORITHMS FOCUSING ON A* AND D*-Megan Reeves http://rave.ohiolink.edu/etdc/view?acc_num=dayton1557245975528397

[6] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. International Journal of Robotics Research, 30(7):846–894, 2011.

[7] http://paper.ijcsns.org/07_book/201610/20161004.pdf

[8] https://www.youtube.com/watch?v=aE7RQNhwnPQ

[9] http://faculty.salina.k-state.edu/tim/robotics_sg/Control/kinematics/unicycle.html

[10] Erwin Coumans and Yunfei Bai. PyBullet, a Python module for physics simulation for games, robotics and machine learning. http://pybullet.org. 2016–2021

[11] https://github.com/JACOBIN-SCTCS/CP214 Project