Pathfinding in Uniform Grid using an Improved Jump Point Search Algorithm

Alimurtaza Merchant - 191IT205
Information Technology
National Institute of Technology Karnataka
Surathkal, India 575025
Email: alimurtaza.191it205@nitk.edu.in

Abhinav Bharali - 191IT201
Information Technology
National Institute of Technology Karnataka
Surathkal, India 575025
Email: abhinavbharali.191it201@nitk.edu.in

Naveen Shenoy - 191IT134
Information Technology
National Institute of Technology Karnataka
Surathkal, India 575025
Email: naveenshenoy.191it134@nitk.edu.in

***Abstract*—Pathfinding is extensively used in real world applications involving navigation systems, packet routing. Pathfinding in uniform grids finds extensive use in various video games. Thus, developing efficient and reliable methods to detect the shortest path between two points in uniform grids is of immense importance. In this paper, we analyze the Jump Point Search (JPS) algorithm and compare it with the existing widely used path finding algorithms in uniform grids such as Dijkstra's algorithm, A* Algorithm & BFS algorithm. We discover various ways in which the JPS algorithm can be improved so as to improve its running time complexity. These include offline pre-processing steps in which the jump points can be searched and stored before pathfinding from the source node is started. Another improvement includes bounded pruning or searching of jump points. Also, we show the various trade-offs of each of the above mentioned path finding algorithms and finally compare all the pre-existing algorithm with the improved JPS algorithm. Finally, we visualize these algorithms on a uniform grid so as to show the improvement in performance that is achieved due to the improved JPS algorithm.**

Index Terms - Pathfinding, JPS algorithm, Uniform grid, Shortest path

I. Introduction

The problem of grid-based pathfinding often appears in application such as computer games and robotics. Although, the encoding of grids is quite simple to understand and apply, the task of finding shortest paths between arbitrary start-target pairs can be of great difficulty. One of the reasons for the difficulty of pathfinding problem in grids is because of the existence of symmetries. When the individual steps of a path can be permuted so as to derive a new and equivalent path that has identical cost, we can say that such a path is asymmetric path. Classical algorithms such as A* waste more time looking at permutations of all shortest paths from the start node to each expanded node in the presence of symmetry.

In this paper, we have implemented the standard Jump Point Search (JPS) and also proposed an improved JPS in which the jump points are pre-processed before for a particular grid. The JPS is an effective algorithm to eliminate symmetries found in grid and optimise the speed of finding shortest paths in grids. The JPS algorithm in simple words is a modified version of the A* algorithm with two simple neighbour-pruning rules. Recursive application of these rules helps in improving the performance of grid based pathfinding. The ability to quickly scan many nodes from the underlying grid map in order to identify jump points is what mainly determines the efficiency of the JPS algorithm. This process helps the compiler to skip many unnecessary node expansions. However as this operation proceeds in a step-wise manner the same node can be scanned multiple times during a single search.

For the improved JPS algorithm, we have precomputed the jump points for all points in a grid in all directions. This process takes only once when the grid is initially setup. The computations of changing start and endpoints optimal path takes place quickly after this step. A web-based visualizer has also been made as a part of this paper. The web-based visualizer contains a uniform grid where the users have been allowed to select different starting and ending points according to their wish along with the ability to select various obstacles. This will help people to better understanding the working of the JPS along with various well known path finding algorithms like Dijkstra's algorithm, BFS and A* algorithm.

Section II. contains the Literature Survey of all the papers we have taken influence from to come up with this paper. Section III. contains the Problem Statement and lists all the objectives we had while coming up with this paper. Section IV. contains the detailed explanation of the Methodology followed by us along with pseudo-code for the various algorithms for better understanding of this paper. Section V. contains the Experimental Results and Analysis for the JPS algorithm and the improved JPS algorithm.

TABLE I
SUMMARY OF LITERATURE SURVEY

Authors	Methodology	Merits	Limitations	Additional Details
Daniel Harabor & Alban Grastien	Jump Point Search makes use of jump points or intermediate nodes in the uniform grid which are selectively used to expand the search path.	No memory overhead, finds optimal path and better performance than current state-of-the-art hierarchical algorithms	Not suitable for all grids	Significant improvement over pruning techniques like Swamp, HPA*
Daniel Harabor & Alban Grastien	Improved Jump Point Search algorithm by improving the pruning techniques through block-based symmetry breaking	Provides an improvement in performance over normal JPS by an order of magnitude	Offline pre-processing is time consuming	Compared with SUB and received competitive and at times better performance
X. Zheng, X. Tu & Q. Yang	Novel Safe-Distance JPS Proposed for efficient path planning of robot in uniform grid	Provides flexibility and shorter optimal path planning time	May not necessarily be faster than standard JPS	Primary focus on obstacle avoidance in path planning
L. Sauya	Scope for improvement in A* and JPS algorithm	Hierarchically annotated A* and rectangular symmetry reduction work better than JPS for certain types of grids	Faster algorithms like SRC spend way more memory than JPS	Combining JPS with RSR or SRC may give better performance in RTS games

Also contains the comparisons of these algorithms with standard path finding algorithms like A*, Dijkstra's and BFS. Section VI contains the conclusion derived from this paper. The references that have been used by us have also been listed.

II. LITERATURE SURVEY

As an improvement over the then state-of-the-art pathfinding algorithms which were hierarchical in nature, [1] proposed an algorithm which is fast, optimal and does not involve any memory overhead. This algorithm named Jump Point Search (JPS) makes use of jump points or intermediate nodes in the uniform grid which are selectively used to expand the search path. It was found that these jump points provide a speed up of about an order of magnitude over the A* algorithm.

In 2014, [2] proposed an improvement to their previously proposed JPS algorithm. This made use of offline and online techniques to improve the performance of the JPS algorithm. The first proposal was online symmetry breaking by considering blocks of nodes. This was implemented by storing the grid as a sequence of bits so many points in the grid can be processed at the same rather than doing it one by one. Later comparison was done with SUB which is a state of the art offline path finding algorithm.

In [3], a novel SD-JPS algorithm was implemented for the path planning of a robot in a uniform grid with obstacles. The idea is to obtain a safe domain matrix which is used to determine what the robot sees as a safe distance between itself and the obstacles. This provides the robot flexibility and helps to plan the optimal path in a shorter time in comparison to the A* algorithm. [4] attempts to discover the various improvements or shortcomings in the traditional A* algorithm and JPS algorithm which gives space for improvement. In the paper, the various approaches developed by Daniel Harabor are discovered in the grids in which JPS is not useful. These include Hierarchically annotated A* and Rectangular Symmetry Reduction.

Table 1 gives a brief summary of the methodology along with the advantages and disadvantages of the major previous works which were referred.

III. PROBLEM STATEMENT

To design and develop an improved Jump Point Search Algorithm to solve the problem of pathfinding in a uniform grid. To decrease the runtime of the JPS algorithm by improving its pruning techniques.

A. Objectives

- Implementing the standard JPS algorithm and detect scope for improvement
- Improving the JPS algorithm by applying pre-processing and improved pruning techniques.
- Analysing the performance of standard and improved JPS algorithm against other widely-used uniform grid shortest pathfinding algorithms such as BFS, A* Algorithm and Dijkstra's Algorithm.
- Visualising the performance of the improved JPS algorithm with the other well-known shortest path finding algorithms on a uniform grid.

IV. METHODOLOGY

The Input for path-finding algorithm is a 2 dimension undirected uniform-cost grid map which can be represented by a 2D array and a start node and a goal node. Each node has $\leq$ 8 neighbours and each node can be empty or there can be an obstacle such as wall on it or we can be say the node either traversable or not traversable. From a particular node one can travel in 8 directions. Out of which 4 moves

are straight (i.e horizontal or vertical) each of which have a cost of 1. And other 4 moves are diagonal each has a cost of $\sqrt{2}$. Move on a nodes having walls are disallowed. direction array contains the all the 8 direction in which each value contains two number representing the amount of movement in horizontal and vertical for each of possible directions.

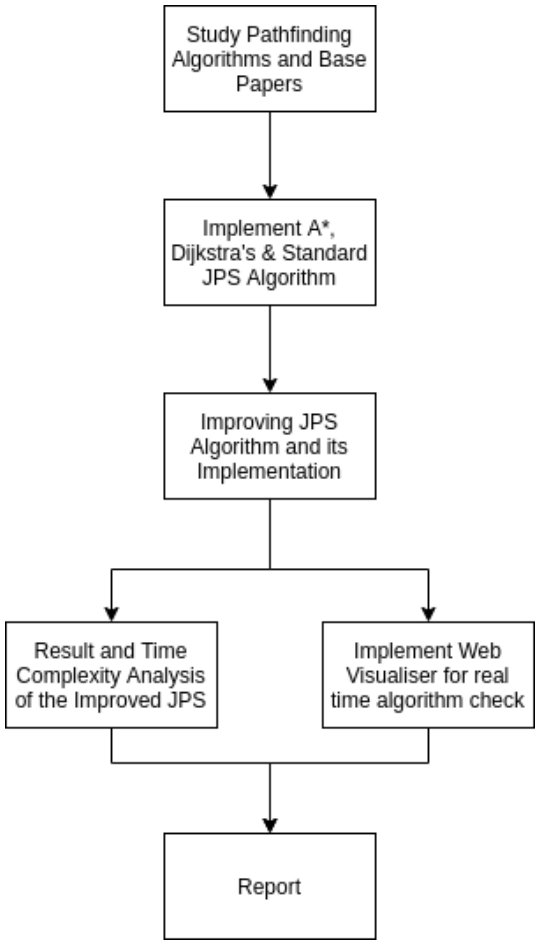

Fig. 1. Methodology

A. A* Algorithm

A* algorithm is a classical algorithm used for path finding in grid based maps. Unlike other traversal techniques it is a smart algorithm which separates it from conventional algorithm. A* uses a heuristic approach which allows it to take a shortcut and helps it to make judgements and solve the problem quickly. In the most conventional Dijkstra's algorithm, we consider the node and its shortest distance from start node and add it in a open list which gives the node having shortest distance from the start node. A* algorithm is also based on this concept, the only difference being that the instead of adding the distance of current node from start node, we consider the sum of the node's distance from start node and heuristic distance from current node to goal node. For calculating heuristic distance we can consider different methods such as euclidean distance, Manhattan distance etc.

Euclidean Distance: $h = \sqrt{(x1 - x2)^2 + (y1 - y2)^2}$
Manhattan Distance: $h = |x1 - x2| + |y1 - y2|$

In this paper we have considered the euclidean distance for heuristic distance since for the traversal from one node to other, we have also considered diagonal movement. Euclidean gives more accurate results while calculating diagonal distance. So the final distance that will be added for each node in the open list will be considering i as a node, h(i) as heuristic distance of node i to goal node and g(i) as the shortest possible distance of node i from start node

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

Algorithm 1 A*

Require: $grid[N*N], start, goal, priority\_queue, visited[N*N], distance[N*N], direction[8]$

$priority\_queue.push(0, start)$
$distance[start] \leftarrow 0$
while $priority\_queue.isEmpty() \neq true$ **do**
  $curr \leftarrow priority\_queue.pop()$
  $curr\_distance \leftarrow curr.first$
  $curr\_node \leftarrow curr.second$
  if $visited[curr] == true$ **then**
    $continue$
  end if
  $visited[curr] \leftarrow true$
  if $curr\_node == goal$ **then**
    $result \leftarrow curr\_distance$
    $break$
  end if
  for $dir \in direction$ **do**
    $adjacent \leftarrow curr\_node + dir$
    if $adjacent \neq boundary\, and\, grid[adjacent] \neq wall\, and\, visited[next] \neq true$ **then**
      if $distanc[adjacent] > curr\_distance + euclidean(start, adjacent)$ **then**
        $distanc[adjacent] \leftarrow curr\_distance + euclidean(start, adjacent)$
        $priority\_queue.push(distance[adjacent] + euclidean(adjacent, goal), adjacent)$
      end if
    end if
  end for
end while
return

B. Jump Point Search (JPS)

Jump Point Search is an improved A* algorithm, or we can say its a combination of A* with simple pruning technique. The pruning helps JPS to eliminate many path symmetries considering all 8 direction in a 2 dimensional grid based map.

Algorithm 2 jump

Require: $grid[N * N], start, goal, curr, dir, visited[N * N], distance[N * N]$

$next \leftarrow curr + dir$
if $next \neq boundary$ **then**
  return $NULL$
end if
if $next \neq wall\ and\ visited[next] \neq true$ **then**
  return $NULL$
end if
if $next == goal$ **then**
  return $next$
end if
if $forced\_neighbor(next) == true$ **then**
  return $next$
end if
if $dir \in Diagonal$ **then**
  $jumpPoint = jump(next, horizontal)$
  if $jumpPoint \neq NULL$ **then**
    return $next$
  end if
  $jumpPoint = jump(next, vertical)$
  if $jumpPoint \neq NULL$ **then**
    return $next$
  end if
end if
return $jump(next, dir)$

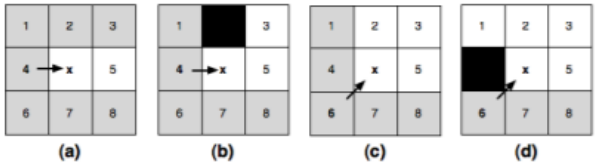

Fig. 2. Pruning Technique

The pruning technique involves certain rules which go as follows:-

Pruning Rules: Lets take a node p (parent node) and a direction d we find a jump point or look ahead node x in the direction d. And for considering node x as a jump point we need to check for any forced neighbor of x or to check a node n which is adjacent x which is not in direction d and there is no other shorter path present to reach n from p other than via x.

$$\pi =< p, x, n >$$

$$\pi' =< p, y, n >$$

$$\pi < \pi'$$

If above equation comes out to be true, then we can consider x as a jump point or else we can skip x, move in direction d and repeat this pruning process until we find any jump point, goal node or the boundary of the grid There are two kinds of moves involved in JPS, viz. straight and diagonal moves.

Straight Move: Consider Fig. 2 (a) and (b). Here p is node 4, and we have two consider two nodes as n {3,8}. Since in (a) path 4- 2- 3 has equal length to 4- x- 3 but in (b) there is an obstacle at 2 which makes the path 4- x- 3 only shortest possible path from 4 to 3. This makes 3 as a forced neighbor of x and x as jump point of 4.

Diagonal Move: Consider Fig. 2 (c),(d). Here p is 6 and for n we have take the nodes {2,5} for reaching 2 we have two paths in (c) of same length 6- 4- 2 and 6- x- 2 so we can skip x for the jump point but in the (d) 4 is blocked so this makes only one possible shortest path from 6 to 2 via x therefore here we have to consider as x as a jump point of 6. In every diagonal we also do straight pruning in corespondent horizontal and vertical direction.

The jump function is the recursive which keeps on moving in a given direction until one of the below condition satisfies:

- If the node has a forced neighbor of
- If the node is on the boundary of the grid
- If node having obstacle or the node has been already visited
- If node is the the goal node

Algorithm 3 JPS

Require: $grid[N*N], start, goal, priority\_queue, visited[N*N], distance[N*N], direction[8]$

if $priority\_queue.isEmpty() == true$ **then**
  return
end if
$curr \leftarrow priority\_queue.pop()$
$curr\_distance \leftarrow curr.first$
$curr\_node \leftarrow curr.second$
if $visited[curr] == true$ **then**
  return $JPS()$
end if
$visited[curr] \leftarrow true$
if $curr\_node == goal$ **then**
  $result \leftarrow curr\_distance$
  return
end if
for $dir \in direction$ **do**
  $jumpPoint \leftarrow jump(curr\_node, dir)$
  if $jumpPoint \neq NULL$ **then**
    if $distance[jumpPoint] > curr\_distance + euclidean(start, jumpPoint)$ **then**
      $distance[jumpPoint] \leftarrow curr\_distance + euclidean(start, jumpPoint)$
      $priority\_queue.push(distance[jumpPoint] + euclidean(jumpPoint, goal), jumpPoint)$
    end if
  end if
end for
return $JPS()$

Now we take the jump points of a node in all 8 directions in 2-dimensional grid and make them as a successors of the

current node. And check for all jump points if its last set distance from start is shorter than the distance from start through its parent node if later one comes shorter we update the distance and add the node in open list. If we fail to find any jump point of the node we add nothing. This process we start from considering the start node as our first node and keeps on repeating until we find goal node or the open list become empty.

This is how JPS reduces the time instead taken in exploring all immediate neighbors of p. JPS exploits this technique to recursively and immediately explore only jump points. When we find jump Point we add only those node to the open list and this has double benefits. One, it reduces the number of operation of insertion and removing on the open list and two, this pruning reduces the number of nodes present in the list at a given time which makes the operation of list faster.

C. Preprocessing

In this we proposed to separate the pruning technique from the the JPS algorithm and do it before running the JPS algorithm. Basically we will pre-compute the jump points for every traversable node on the map in all 8 direction and store them in a separate list and use directly the list for finding next set of jump point at the time of traversal instead of doing the whole pruning process again. This method has a trade off between time and space here we are improving the time taken by algorithm at an expense of space.

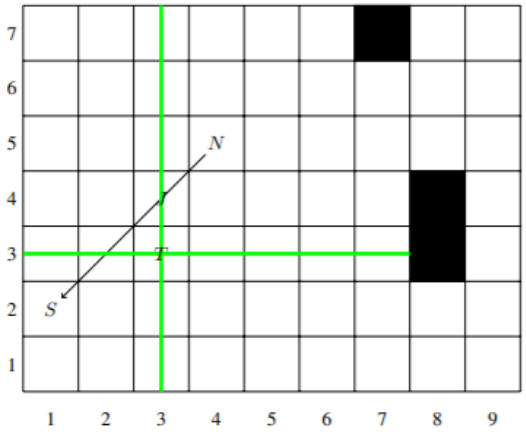

Fig. 3. Preprocess Pruning

Since preprocess is done only once over an instance of a grid, this means while prepossessing there are no fixed start and goal node present. So we have to handle a possible situation separately. Consider Fig. 3, where T is the target node. In normal JPS, J would be considered as a jump point because it is an intimidate turning point as it lies on the way to T. But in preprocessing, J will not be identified as a jump point since the goal node is unknown at that particular time. While doing traversal considering any jump point we need to check if any node in between leads us to the column or the row of the target node and consider it as a possible jump point.

Algorithm 4 Preprocessed_JPS

Require: $grid[N*N], start, goal, priority\_queue, visited[N*N], distance[N*N] jumpPoints[N*N*8]$

if $priority\_queue.isEmpty() == true$ **then**
  return
end if
$curr \leftarrow priority\_queue.pop()$
$curr\_distance \leftarrow curr.first$
$curr\_node \leftarrow curr.second$
if $visited[curr] == true$ **then**
  return $preprocessed\_JPS()$
end if
$visited[curr] \leftarrow true$
if $curr\_node == goal$ **then**
  $result \leftarrow curr\_distance$
  return
end if
$i \leftarrow 0$
while $i \neq 8$ **do**
  $jumpPoint \leftarrow jumpPoints[curr][i]$
  if $goal.row \in [curr\_node.row, jumpPoint.row]$ **then**
    $intermediate = find(curr\_node, jumpPoint)$
    $jumpPoint \leftarrow intermediate$
  end if
  if $goal.col \in [curr\_node.col, jumpPoint.col]$ **then**
    $intermediate = find(curr\_node, jumpPoint)$
    $jumpPoint \leftarrow intermediate$
  end if
  if $jumpPoint \neq NULL$ **then**
    if $distance[jumpPoint] > curr\_distance + euclidean(start, jumpPoint)$ **then**
      $distance[jumpPoint] \leftarrow curr\_distance + euclidean(start, jumpPoint)$
      $priority\_queue.push(distance[jumpPoint] + euclidean(jumpPoint, goal), jumpPoint)$
    end if
  end if
  $i \leftarrow i + 1$
end while
return $preprocessed\_JPS()$

D. Bounded Jump

In practice jump point search generates many states to be effective which partially depends on how open map is or the density of the obstacle present on the map. So the idea here is to put limit while pruning basically restricting the jumping in a certain limit. So instead of jumping all of the way to jump point. Bounded jump limits how far it can jump. Once we reach a limit the nodes are considered as a jump points and are placed into the open list. This reduces the generation of unnecessary nodes in opposite direction and can save time while pruning and makes the jump point search algorithm more effective.

Algorithm 5 bounded_jump

Require: $grid[N * N], start, goal, curr, dir, visited[N * N], distance[N * N], limit, flag$

if $limit == LIMIT$ **then**
  if $flag == true$ **then**
    return $curr$
  end if
end if
$next \leftarrow curr + dir$
if $next \neq boundary$ **then**
  return $NULL$
end if
if $next \neq wall\ and\ visited[next] \neq true$ **then**
  return $NULL$
end if
if $next == goal$ **then**
  return $next$
end if
if $forced\_neighbor(next) == true$ **then**
  return $next$
end if
if $dir \in Diagonal$ **then**
  $jumpPoint = bounded\_jump(next, horizontal, 0, false)$
  if $jumpPoint \neq NULL$ **then**
    return $next$
  end if
  $jumpPoint = bounded\_jump(next, vertical, 0, false)$
  if $jumpPoint \neq NULL$ **then**
    return $next$
  end if
end if
$limit \leftarrow limit + 1$
return $bounded\_jump(next, dir, limit, flag)$

E. Web Visualizer

A web-visualizer was made using HTML, CSS and Javascript to show the working of various optimal path finding algorithms like A*, Dijkstra and JPS. The web visualizer allows the users to select the start and ending points with start and endpoint buttons. Those particular grids are then marked by Javascript with special CSS classes such as end and start to identify them being unique points. The users can also set various points as walls using the same method that is used for starting and endpoints. An algorithm drop down menu option is present with various choices such as A*, Dijkstra and JPS so that people can click on the algorithm whose visualization they want to observe. On clicking a particular option, the javascript sets the algorithm selected to be that. All of these algorithms have been implemented in separate javascript files and functions and when an option is selected that particular function is called by the main function.

V. Experimental Results and Analysis

The problem of finding the shortest path in a graph is an optimization problem. To classify the problem in a class, it is re framed as a decision problem. Since a solution can be obtained in polynomial worst case complexity for shortest path finding in a bounded space, it is a class P problem. JPS algorithm is a greedy heuristic-based algorithm with a worst case polynomial runtime.

A. Time and Space Complexity Analysis

The worst case time complexity of JPS is similar to A* algorithm. A* is a heuristic algorithm means it makes some assumption about the possible result and proceed in that direction. There the exact time complexity analysis of A* is tricky. But we can say its worst time complexity will be when we explore all of the node in grid which will be similar to Dijkstra's algorithm that is O(NlogN) where N is the number of nodes in grid. Now looking at JPS, its basic idea is similar to A*, the only difference being it saves time by directly jumping to look ahead nodes instead of exploring all its immediate neighbors. Finding those look ahead nodes require us to do pruning for which the worse case time complexity would be O(N). When this occurs, the worst case scenario of pruning for operation of open list becomes O(1) and results are found in O(N) time. If there are lot of walls present, this results in identifying immediate neighbors only as a jump points. This will reduce the time taken in pruning but increase the operation time complexity of open list. But considering an average case scenario, we can say that the time complexity of JPS will be a factor times better than A*. This factor will depend on the grid and will be changing from one grid to another and can be observed in the results. Preprocessing separates the pruning from JPS algorithm which is a trade off between time and space since there will be extra O(N) space required while preprocessing the jump points in offline mode. Preprocessing will be much faster when there are multiple start goal pairs on a same grid instance.

B. Observations

The JPS algorithm was first compared with the other well-known pathfinding algorithms such as A* algorithm and Dijkstra's algorithm. Then, we compared the various implementations of the JPS algorithm along with the final improved version. A sample run of both the algorithms i.e, A* and Improved JPS were inspected using a visualizer.

From Table II, we observe that the runtime for the pathfinding increases with increasing size of the grid. The running time of the JPS algorithm is significantly faster than that of the most common optimal path finding Dijkstra's (almost 2x or 3x). The JPS algorithm can also be seen to slightly faster than the A* algorithm for path finding. The reason for the A* begin faster than Dijkstra's algorithm is that we also account for how close we are to the destination in A* algorithm where on the other hand, in Dijkstra's algorithm, we just expand from the cell closest to the source. In JPS algorithm, we further improve the performance by finding certain jump points which speed

TABLE II
RUNTIME COMPARISON OF DIFFERENT PATHFINDING ALGORITHMS

Size of the Grid (N*N)	Time taken for Dijkstra's(in ms)	Time taken for A*(in ms)	Time taken for JPS(in ms)
50	3795	1905	1573
100	15418	4830	4749
150	36001	19001	13999
200	62087	31038	24041
250	99101	47029	39000
300	107173	56883	43050
350	133467	59021	52280
400	174960	69792	66913
450	218737	87679	78692
500	272563	107115	98376
550	334385	114257	104623
600	397712	160540	145607

up the process of expansion.

In Table III, we compare the time taken by the various implementations of the JPS algorithm for different size of the input grid. It is observed that the best runtime is achieved by the final improved JPS algorithm which consists of pre-computed jump points and follows a bounded search strategy for detecting the jump points. The run time when either of the two optimizations i.e, pre-processing and bounded search is almost similar but better than the original JPS algorithm. For the above observation, the limit for bounded search is set at a constant value of 5. In general, it is seen that the bounded search optimization performs well in grids which are sparse in nature compared to grids with dense obstacle paths.

TABLE III
RUNTIME COMPARISON OF DIFFERENT IMPLEMENTATIONS OF THE IMPROVED JPS ALGORITHM

Size of the Grid (N*N)	Time taken By JPS(in ms)	Time taken by JPS with Bounded Search(in ms)	Time taken by Pre-processed JPS(in ms)	Time taken by Improved JPS(in ms)
50	1213	600	399	384
100	2085	1178	928	909
150	3737	2341	2037	2007
200	8244	5183	4679	4918
250	13894	10485	9144	7580
300	15996	15021	10611	9527
350	27651	23364	19810	18337
400	37950	31746	31948	30942
450	43143	41716	33998	32551
500	47514	43557	41771	39296
550	55383	51019	45152	46662
600	69634	63108	59365	56070

A visualizer was developed using HTML, CSS and Javascript to be able to inspect the working of the A* algorithm and the improved JPS algorithm implemented in this paper. The algorithms were run on a uniform grid. The orange points show the points popped from the priority queue and the jump points in the A* and JPS algorithm respectively. The blue and red cell represent the source and the destination respectively. Fig. 4 and Fig. 5 show that the JPS algorithm is more efficient than the A* algorithm. The number of jump points in the case of the improved algorithm is less than the elements that were popped from the priority queue in the A* algorithm which formed the path to the destination cell.

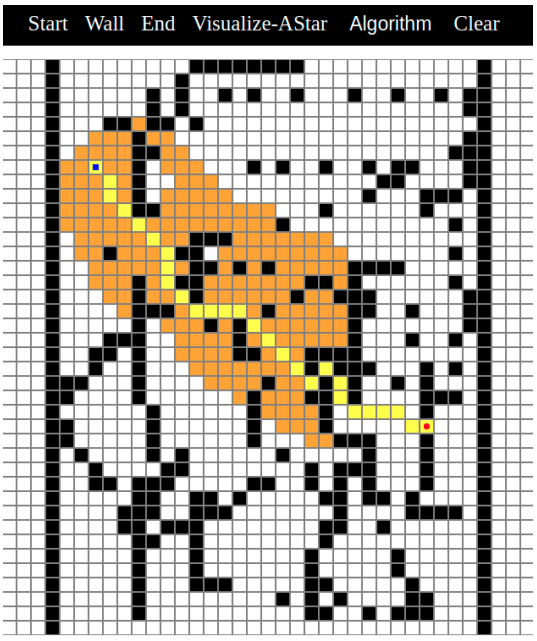

Fig. 4. A* Algorithm

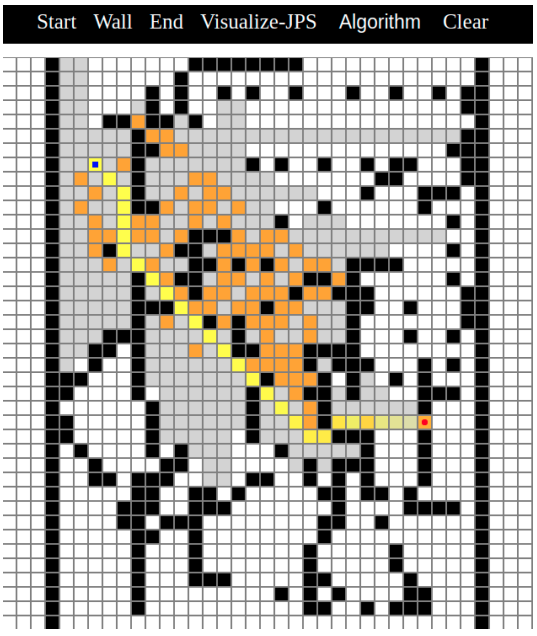

Fig. 5. Improved JPS Algorithm

VI. CONCLUSION

The Jump Point Search algorithm (JPS) along with various modifications namely preprocessed JPS , JPS with bounded search and JPS with both pre-processing and bounded search. A web based visualizer was also made to help in understanding of the working of the JPS algorithm and to visually see that

the algorithm works faster than other optimal path finding algorithms like A*, Dijkstra etc. To understand the runtime speed of the JPS along with various other algorithms we computed the runtime speed of JPS along with that of Dijkstra and A* and it was found that the JPS was considerably better than Dijkstra and was slightly quicker than the A* algorithm. Moreover, analysis was also done on JPS along with various improvements such as bounded JPS, pre-processed JPS and both bounded+pre-processed JPS. It was found that the bounded JPS worked better in grids which were sparse when compared with those with dense obstacles. Overall it was observed that the bounded + pre-processed JPS performed the task of path finding most efficiently while only pre-processed was just slightly worse off than the combined JPS. The bounded JPS was also slightly faster than the normal JPS.

BASE PAPER

The base paper referred for this research was [2]. The authors of the paper are Daniel Harabor and Alben Grastien. It was published in 2014 in the Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling. The paper suggested improvements in their 2011 published JPS algorithm [1]. This included improved pruning techniques and online symmetry breaking by considering blocks of nodes.

REFERENCES

[1] D. Harabor and A. Grastien, "Online graph pruning for pathfinding on grid maps," in *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, ser. AAAI'11. AAAI Press, 2011, p. 1114–1119.

[2] ——, "Improving jump point search," in *Proceedings of the Twenty-Fourth International Conferenc on International Conference on Automated Planning and Scheduling*, ser. ICAPS'14. AAAI Press, 2014, p. 128–135.

[3] X. Zheng, X. Tu, and Q. Yang, "Improved jps algorithm using new jump point for path planning of mobile robot," in *2019 IEEE International Conference on Mechatronics and Automation (ICMA)*, 2019, pp. 2463–2468.

[4] L. Suaya, "A* optimizations and improvements - research consisting on investigating and exposing improvements for a* algorithm for pathfinding (resulting on jps algorithm)," 2018. [Online]. Available: https://lucho1.github.io/JumpPointSearch/