PATH PLANNING FOR TURTLEBOT USING MODIFIED A-STAR ALGORITHM WITH MONOCULAR SLAM

Kamakshi Jain , Abhinav Modi , Rohan Singh ,

***Abstract*—This project deals with path planning of a mobile robot based on a grid map generated using ORB SLAM. The main aim of the project is to implement modified A-star algorithm by implementing research papers ([1],[2],[13]) on RSR(Rectangualr Symmetry Reduction) and JPS(Jump Point Search). These modifications are focused primarily on reducing computational time, maintaining the optimality of the solution. Modified A-star is evaluated and compared against the traditional A-star algorithm.**

***Index Terms*—SLAM(Simultaneous Localization and Mapping), RSR(Rectangular Symmetry Reduction), JPS(Jump Point Search), ORB(Oriented fast and Rotated Brief), A-star, Turtlebot.**

I. Introduction

Path planning has different definitions in different fields. In robotics, path planning is concerned as to how to move a robot from one point to another point. It generally focuses on designing algorithms to generate motion by processing simple or complicated geometric models. A common task for a mobile robot(here Turtlebot) is to navigate in an environment depending upon the knowledge of the environment. Tasks for the mobile robot can vary from building a map of the environment to determining its precise location within a map. These tasks can be connected to SLAM (Simultaneous Localization And Mapping). SLAM techniques build a map of an unknown environment and localize the sensor in the map with a strong focus on real-time operation. In this project we implement ORB SLAM under the assumption that there exist no dynamic constraints of the robots movement and the path generated is the connection between the points of interest[1].

As the environment changes, the algorithm must be revised to ensure an optimistic collision-free path. A* algorithm is a heuristic function based algorithm that can be applied on grid/metric map for proper path planning. It calculates heuristic function's value at each node on the work area and involves the checking of too many adjacent nodes for finding the optimal solution with zero probability of collision[14]. However, it also suffers from limitations involving processing time and work speed. Various variants of the A* algorithm have been devised in this project such Rectangular Symmetry Reduction (RSR), Jump Point Search (JPS). These modifications are focused primarily on computational time and the path optimality and will be implemented on the SLAM developed environment.

A recent study has shown that the nurses spend about 30-35% of their time “fetching and gathering” general supplies for the hospital rooms. This makes them less available for the immediate help of the patients. A mobile robot can be used here to pre-fetch these supplies for the nurses and get them to the rooms where nurses can use them. For these problems the hospital environment can be mapped and a mobile robot can use it to navigate its way through the hospital. With this as our motivation we aim to use ORB-SLAM to map an environment using a single monocular camera so that the turtlebot can navigate its way to reach a desired location.

This project will aim at simulation of TurtleBot in Gazebo and implementing monocular SLAM using the ORB-SLAM2 libraries[6], Implementing A* Algorithm, Implementing Rectangular Symmetry Reduction (RSR), Implementing Jump Point Search (JPS), Doing a comparative study of modified A* with basic A* algorithm based on computation time and if time permits, testing of above algorithms on hardware, i.e., the Turtlebot.

II. Methodology

A. *ORB SLAM*

Vision based SLAM is a very popular tool these days for mapping. This is due to the fact that a lot of research has been put into development of cameras by many big research institutes in terms of hardware. We can obtain a lot of information form just a single image, in terms of quality as well as quantity. In this project, we are implementing monocular SLAM(using a single camera) on the turtlebot, based on the work in [3],[4]. The algorithm is named ORB-SLAM (ORB-SLAM2 for the latest version), and is available as an out-of-the-box SLAM library that can be used with ROS[6]. This SLAM algorithm uses ORB features which are *"Oriented FAST and Rotated BRIEF"* features. These features are an alternate to SIFT features which are slow when it comes to feature matching between two frames. An output of the features detected in the image as seen by the turtlebot is shown in the fig(1).

We are using this library to generate maps to test the path planning algorithms mentioned in the following sections.

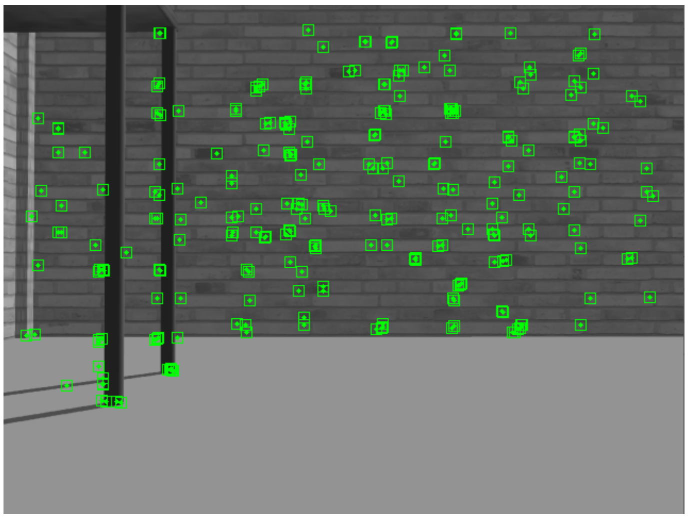

Fig. 1. ORB features detected in the image as seen by the onboard camera of the turtlebot

The ORB-SLAM system[12] has three main parallel threads:

1) Tracking to localize the camera with every frame by finding feature matches to the local map and minimizing the re-projection error applying motion-only BA(Bundle Adjustment)
2) Local mapping to manage the local map and optimize it, performing local BA
3) Loop closing to detect large loops and correct the accumulated drift by performing a pose-graph optimization. This thread launches a fourth thread to perform full BA after the pose-graph optimization, to compute the optimal structure and motion solution

Bundle Adjustment and Loop closure

The system performs BA[4] to optimize the camera pose in the tracking thread, to optimize a local window of keyframes and points in the local mapping thread and then loop closure to optimize all keyframes and points. This optimization uses Levenberg-Marquardt method to minimize the reprojection error which is the mean error in euclidean distance between the keypoints obtained using orb and the projection of 3D world points on the image plane using the pose(R,t). The optimized pose is obtained as:

$$\{\mathbf{R},\mathbf{t}\} = {}_{R,t}\sum_{i\in\mathcal{X}} \rho(\|x^i - \pi^i(\mathbf{RX^i}+\mathbf{t})\|^2_{\sum}) \tag{1}$$

where ρ is the robust Huber cost function and $\sum$ is the covariance matrix for teh scale of keypoint($\mathbf{x}$) and π is the projection function defined as:

$$\pi_m\left(\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}\right) = \begin{bmatrix} f_x\frac{X}{Z} + c_x \\ f_x\frac{Y}{Z} + c_y \end{bmatrix} \tag{2}$$

where (f_x, f_y) is the focal length and (c_x, c_y) is the principal point in the intrinsic camera parameters.

Loop closing is also performed once a loop is detected and validated. This is done using a full BA optimization after pose-graph to achieve the optimal solution. To make this step cost-efficient this optimization is performed in a seperate thread allowing the system to continue creating maps and detecting loops. Once the optimization is complete the outputs are then merged by aborting the BA and then restarting the full BA.

We are using the map for the Robotics Realization Lab, University of Maryland. This map will be used to generate a point cloud data which will be used to generate a grid for implementing A-star. A sample of the point cloud data generated using ORBSLAM2 is shown in fig(8).

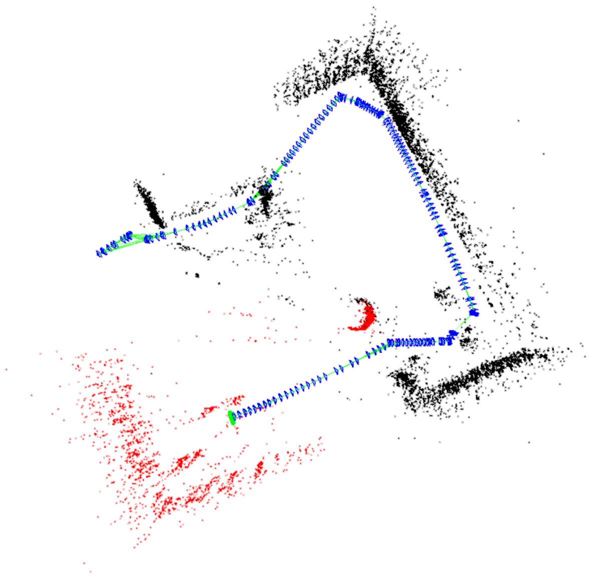

Fig. 2. Sample point cloud output generated for the RRL map

Some prerequisite assumptions taken into account for this implementation are listed below:

1) The Primary assumption for ORB-SLAM to work is that the image frame contains a minimum number of features.
2) The speed of the turtle-bot is under a threshold so that there is no image-blur which can cause loss of features.
3) The environment in consideration is not dynamic.
4) The dynamics of the motion are considered ideal and no-slip condition is assumed for wheels of the turtle-bot.
5) For Algorithm implementation, the maps will be generated beforehand by manually controlling the TurtleBot.

B. A Algorithm*

A* algorithm [1] uses a combination of heuristic searching and searching based on the shortest path. In this algorithm, each cell in the configuration space is evaluated by the value,

$$f(v) = h(v) + g(v),$$

where $h(v)$ is heuristic distance (Manhattan, Euclidean or Chebyshev) of the cell to the goal state and $g(v)$ is the length of the path from the initial state to the goal state through the selected sequence of cells.

The pseudo code for A* is shown in Algorithm (1)

Algorithm 1 Traditional A*

```
Require: $s$: start, $g$: goal, $checkGoal(n)$, $expand(n)$, $actions$
  if $goal(s) = true$ then
    return $makePath(s)$
  end if
  $open \leftarrow s$
  $closed \leftarrow \emptyset$
  while $open \neq \emptyset$ do
    $sort(open)$
    $n \leftarrow open.pop()$
    $childs \leftarrow expand(n, actions)$
    for all $child \in childs$ do
      if $goal(child) = true$ then
        return makePath(child)
      end if
      if $child \notin closed$ then
        $open \leftarrow child$
      end if
    end for
    $closed \leftarrow n$
  end while
```

The paper [2] lists the some basic modifications to the A* algorithm like basic Theta*[8], Phi*[8] and JPS[5],[11] (Jump Point Search) and implementation of RSR[1],[9],[10] (Rectangular Symmetry Reduction) to maps to reduce computation time for these modifications and improve their performance for real-time use. We will be using RSR and JPS in our project and a brief overview of these methods are given in the following sections.

C. *Rectangular Symmetry Reduction (RSR)*

RSR is a pre-processing step that speeds up the process of finding the optimal path applied to the map generated from our SLAM. This identifies and eliminates symmetries in the graph by only ever expanding nodes from the perimeter of each empty rectangle, and never from the interior.The symmetries in the graph are defined as the rectangles inside which only free space occurs. These are treated as one big rectangle, and path inside these rectangles is defined only by the set of the rectangles edges. If the initial, actual or goal state is inside the rectangle, cell representing this state is temporary added as a new cell. This temporary cell is connected with all peripheral cells of the correspondent rectangle. By this data reduction, the computational time of path planning algorithms can be reduced.

Step 1: Grid decomposition
In this step we decompose the map into a set of obstacle free rectangles. After decomposition, prune all nodes from the interior of each empty rectangle as shown in figure (3)

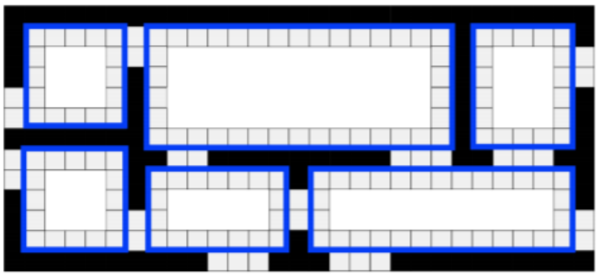

Fig. 3. Step 1 Grid decomposition

Step 2: Addition of Macro Edges
In this step we add a series of macro edges to connect each node on the perimeter of a rectangle with other nodes from the perimeter.

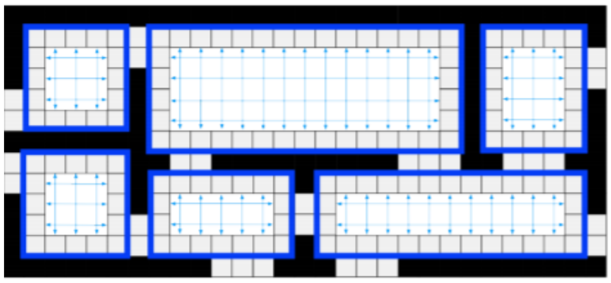

Fig. 4. Step 2 Addition of Macro Edges

Step 3: Online Insertion
This step is done when the start or goal is located in the interior of an empty rectangle. Here we use a temporary node to connect the temporary node, online, to the perimeter nodes.This procedure is called re-insertion procedure.

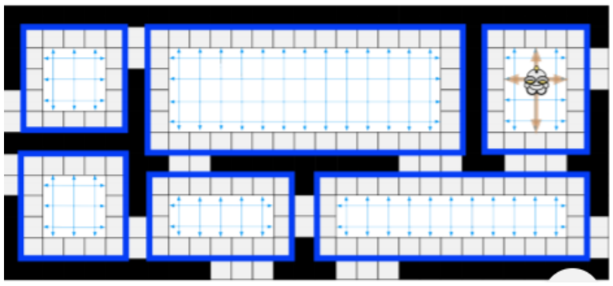

Fig. 5. Step 3 Online Insertion

Advantages: RSR has a low computational demands and possibility of combination with the family of star algorithms such as A*, Theta* etc while preserving optimality. It also has a small memory overhead in practice.

D. *Jump Point Search (JPS)*

Jump Point Search (JPS) [1],[2],[13] is an online symmetry(defined above) breaking algorithm which speeds up path-finding on uniform-cost grid maps by ”jumping over” many locations that would otherwise need to be explicitly considered. JPS requires no pre-processing and has no memory overheads. It can speed up A* search by over many orders of magnitude. A more detailed explanation of this process is given below. An example of path following with this method is given in Figure (6 and 7).

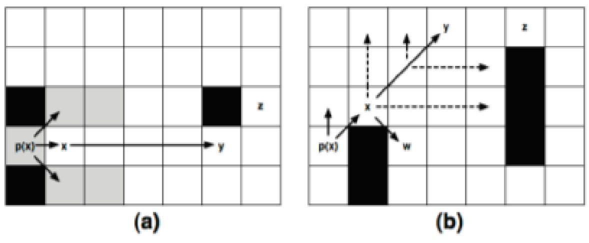

Fig. 6. a) Examples of straight and b) Examples of diagonal jump points. Dashed lines indicate a sequence of interim node. Strong lines indicate eventual successor nodes

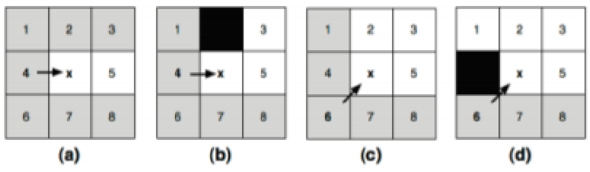

Fig. 7. Cases where a node x is reached from its parent p(x) by either a straight or diagonal move. When x is expanded we can prune from consideration all nodes marked grey. In (a) node 5 is a natural node. In (b) node 5 is natural whereas node 3 is forced.

The main idea behind JPS is trying to ignore symmetric paths when exploring nodes in A*.

*1) **Neighbour Pruning**:* First step of JPS involves pruning neighbours, i.e., ignoring neighbours of an open node that do not need to be evaluated to achieve optimality. These neighbours are not included in the ”open” list. Let the current node be x, parent be $p(x)$ and neighbour be n. The criteria for deciding which neighbours are ignored is based on comparing 2 paths, both going from the p(x) to the neighbour, one including the x and the other not. First we define natural neighbours,

Definition 1. *A node n ∈ neighbours(x) is natural if:*

1) *For straight moves:*

$$len(\langle p(x), ..., n\ x\rangle) > len(\langle p(x), x, n\rangle)$$

2) *For diagonal moves:*

$$len(\langle p(x), ..., n\ x\rangle) \geq len(\langle p(x), x, n\rangle)$$

Next we define forced neighbours,

Definition 2. *A node n ∈ neighbours(x) is* forced *if:*

1) *n is not a not a natural neighbour of x.*
2) $len(\langle p(x), ..., n\ x\rangle) > len(\langle p(x), x, n\rangle)$

All nodes that neither forced nor natural are pruned, or ignored. For a 8-connected grid, maximum numbers of neighbours left to evaluate can be 2.

Algorithm 2 Identify Successors

```
Require: $x$: current node, $s$: start node, $g$: goal node
  $successor(x) \leftarrow \emptyset$
  $neighbours(x) \leftarrow prune(x, neighbours(x))$
  for all $n \in neighbours(x)$ do
    $n \leftarrow jump(x, direction(x, n), s, g)$
    add $n$ to successors(x)
  end for
  return successors(x)
```

*2) **Jump Points**:* Next step is finding the jump point,i.e., finding the node that is to be explored next.

Definition 3. *Node y is the jump point from node x, heading in direction $\vec{d}$. Let $\vec{d_1}, \vec{d_2}$ be directional changes in $\vec{d}$,i.e., if $\vec{d}$ is diagonal, $\vec{d_1}$ & $\vec{d_2}$ are straight and vice versa. If y minimizes the value k such that $y = x + k\vec{d}$ and one of the following conditions hold [13]s:*

1) *Node y is the goal node.*
2) *Node y has at least one neighbour whose evaluation is forced according to Definition 1.*
3) *$\vec{d}$ is a diagonal move and there exists a node $z = y + k_i\vec{d_i}$ which lies k_i steps in direction $\vec{d_i}, i \in 1, 2$ such that z is a jump point from y by condition 1 or condition 2.*

Algorithm 3 Function *jump*

```
Require: $x$: initial node, $\vec{d}$: direction, $s$: start, $g$: goal
  $n \leftarrow step(x, \vec{d})$
  if $n$ is an obstacle or is outside the grid then
    return $null$
  end if
  if $n = g$ then
    return $n$
  end if
  if $\exists n' \in neighbours(n)$ s.t. $n'$ is forced then
    return $n$
  end if
  if $\vec{d}$ is diagonal then
    for all $i \in \{1, 2\}$ do
      if $jump(n, \vec{d_i}, s, g)$ is not $null$ then
        return $n$
      end if
    end for
  end if
  return $jump(n, \vec{d}, s, g)$
```

The pseudo code for the above two processes is given in Algorithms (2) and (3). The condition for choosing the next node from the "open" list remains same, only the nodes that are added in the lists change from traditional A*. This algorithm provides the optimal path, but is orders of magnitude faster than A*. For a detailed proof of optimality of the algorithm refer to [13].

E. *TurtleBot*

It is a low-cost, personal robot kit with open-source software. TurtleBot3 is made up of modular plates that users can customize the shape. TurtleBot3 consists of a base, two Dynamixel motors, a 1,800mAh battery pack, a 360 degree LIDAR, a camera(+ RealSense camera for Waffle kit, + Raspberry Pi Camera for Waffle Pi kit), an SBC(single board computer: Raspberry PI 3 and Intel Joule 570x) and a hardware mounting kit attaching everything together and adding future sensors. We will be only employing the camera sensor for our project (if we move onto hardware implementation).

For our simulation environment, we will use Gazebo with TurtleBot ROS implementation. We will be using RViz to get camera data in our simulation.

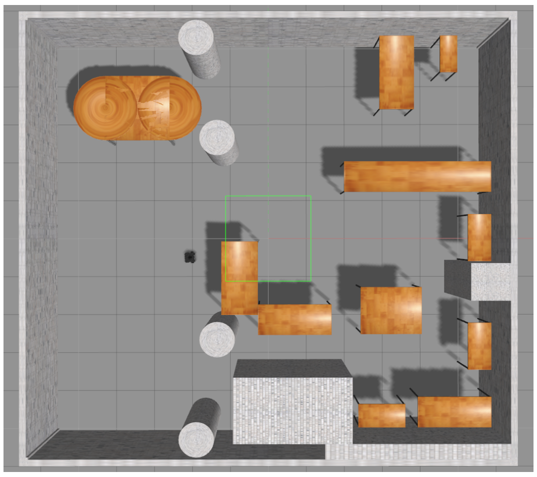
Fig. 8. Turtlebot Waffle in the simulation environment - RRL map

III. Results

We tested and compared the 2 algorithms against traditional A*, on a map of the Robotics Realization Lab, University of Maryland, generated by the running monocular SLAM on a Turtlebot in simulation. RSR was performed on a 4-connected graph whereas JPS on a 8-connected one. The term for comparison that was considered was the ratio of run-time of the algorithms with traditional A*. We calculated the ratio over different set of start and goal nodes, and our findings are reported below.

The node exploration for the applied algorithms are shown in Figures (10) and (11). The grid decomposition for RSR is shown in Figure(9)

Algorithm	Run-Time Ratio
A* + RSR	1.2 - 10.1
A* + JPS	2.7 - 12.0

TABLE I
Run-Time Ratios of RSR and JPS wrt Traditional A*

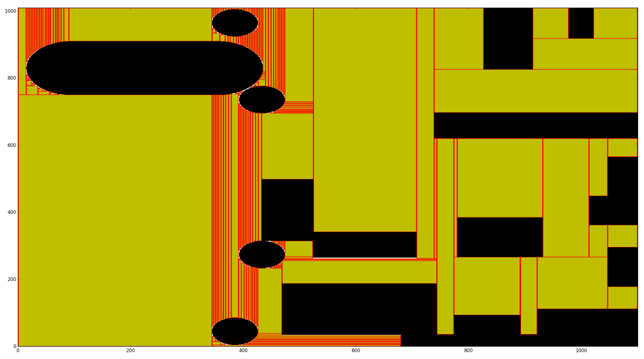
Fig. 9. Grid Decomposition of the RRL map for RSR Algorithm (Resolution = 1cm). Olive Green - Cells, Red - Cell Edges, Black - Obstacles

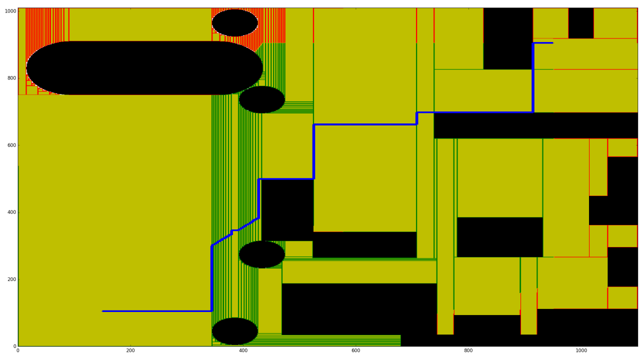
Fig. 10. Node Exploration using A* with RSR Algorithm (Resolution = 1cm). Light Green - Explored Nodes, Blue - Path. (Start at bottom left)

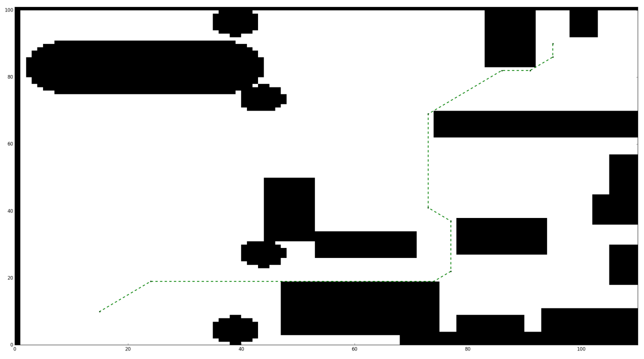
Fig. 11. Node Exploration using A* with JPS Algorithm (Resolution = 10cm). 35 nodes explored. Dotted line - Path, Turning points(/vertices) in path - Explored Nodes lying on path. (Start at bottom left)

IV. Previous Studies

Summarizing the results from our main paper [1], the paper evaluated RSR and JPS on a number of benchmark grid map sets taken from Nathan Sturtevant's freely available pathfinding library, Hierarchical Open Graph. One of the map sets is from the game Baldur's Gate II: Shadows of Amn. The other two map sets (Rooms and Adaptive Depth) are both synthetic, though the latter could be described as semi-realistic.

The paper reported that,

1) RSR can consistently speed up A* search by a factor of between 2-3 times(Baldur's Gate), 2-5 times (Adaptive Depth) and 5-9 times (Rooms).
2) JPS can speed up A* by a factor of between 3-15 times (Adaptive Depth), 2-30 times (Baldur's Gate) and 3-16 times (Rooms).

V. Conclusion

We have implemented python codes for A* and modified A* (RSR and JPS [1],[2],[13]) with SLAM simulation in Gazebo and can make the following concluding remarks based on our implementation. These include our observations and experiences in completing the project.

1) RSR speeds up A* by a considerable factor but requires knowledge of the map beforehand, hence cannot be used with SLAM.
2) JPS is a good strategy which can speed up A* on 8-connected graphs, and can also work with SLAM. Though, it's implementation requires careful programming as recursive function calls may slow the process. We observed that direct implementation of JPS following the pseudo code in [13] isn't as effective with high resolution maps. It is considerably better on lower resolution graphs. One improvement that can be implemented in future is Goal Bounding Search, which further speeds up JPS.
3) Monocular ORB-SLAM is a fast algorithm for SLAM. Though there is a slight inaccuracy in depth as the camera setup is monocular, it's still very accurate for the speed it operates at.

References

[1] Harabor, D., 2012. Fast Pathfinding via Symmetry Breaking. 2013-11-20]. http://aigamedev, com/open/tutorial/symmetry-in-path finding.

[2] Ducho, F., Babinec, A., Kajan, M., Beo, P., Florek, M., Fico, T. and Juriica, L., 2014. Path planning with modified a star algorithm for a mobile robot. Procedia Engineering, 96, pp.59-69.

[3] Mur-Artal, R. and Tards, J.D., 2017. Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras. IEEE Transactions on Robotics, 33(5), pp.1255-1262.

[4] Mur-Artal, R., Montiel, J.M.M. and Tardos, J.D., 2015. ORB-SLAM: a versatile and accurate monocular SLAM system. IEEE transactions on robotics, 31(5), pp.1147-1163.

[5] Harabor, D.D. and Grastien, A., 2011, August. Online graph pruning for pathfinding on grid maps. In Twenty-Fifth AAAI Conference on Artificial Intelligence.

[6] ORB-SLAM2, `https://github.com/raulmur/ORB_SLAM2`

[7] ORB-SLAM, `https://github.com/raulmur/ORB_SLAM`

[8] Yap, P.K.Y., Burch, N., Holte, R.C. and Schaeffer, J., 2011, October. Any-angle path planning for computer games. In Seventh Artificial Intelligence and Interactive Digital Entertainment Conference.

[9] Zhang, An, Chong Li, and Wenhao Bi. "Rectangle expansion A pathfinding for grid maps." Chinese Journal of Aeronautics 29, no. 5 (2016): 1385-1396.

[10] Harabor, D.D., Botea, A. and Kilby, P., 2011, December. Path symmetries in undirected uniform-cost grids. In Ninth Symposium of Abstraction, Reformulation, and Approximation.

[11] Harabor, D.D. and Grastien, A., 2012, July. The JPS Pathfinding System. In SOCS.

[12] Mur-Artal, R., Montiel, J.M.M. and Tardos, J.D., 2015. ORB-SLAM: a versatile and accurate monocular SLAM system. IEEE transactions on robotics, 31(5), pp.1147-1163.

[13] Harabor, D.D. and Grastien, A., 2011, August. Online graph pruning for pathfinding on grid maps. In Twenty-Fifth AAAI Conference on Artificial Intelligence.

[14] Akshay Kumar Guruji,Himansh Agarwal,D.K. Parsediya 2016 Time-efficient A* Algorithm for Robot Path Planning