Implementation of Sparse Graphical Memory for Robust Planning in Robotics

Jie Wei Lek
CSD
SUTD
Singapore
1005007

Abstract—**Sparse Graphical Memory presents itself as a promising approach for robust planning not only in theory, but also in implementation in the field of robotics. It offers a middle ground between deep learning and classical planning techniques. By implementing SGM in robotics, we can see if this approach is feasible in nvaigation.**

I. INTRODUCTION

This document shall serve the purpose of detailing the author's simplified implementation of the Sparse Graphical Memory for Robust Planning in Robotics Operating System 2 (ROS2), and will be used for Singapore University of Technolgty and Design (SUTD) Mobile Robotics Individual Project.

II. RELATED WORK

This project takes inspiration from 'Sparse Graphical Memory for Robust Planning' by Scott Emmons, Ajay Jain, Michael Laskin, Thanard Kurutach, Pieter Abbeel and Deepak Pathak.

While our implementation primarily focuses on adapting SGM fundamentals within the context of ROS2 and Gazebo simulation, the concepts and insights provided by Emmons et al. serve as a guiding framework for my work.

III. MOTIVATIONS

My motivations for implementation was to create my own robotics project using skills that I have learnt. By doing so, I can gain better insight on how robotic systems work.

IV. CONTRIBUTIONS

All credits for the sparcity and long horizon goals algorithm go to Emmons et al. In this project, I mainly focused on the implementation of SGM in ROS2 from scratch.

V. INTRODUCTION TO SPARSE GRAPHICAL MEMORY (SGM)

Currently there are two conventional approaches to reninforcement learning agents. (1) Deep Learning — this method is good at learning from examples but does not scale with long-horizon tasks. (2) Classical Planning — this method plans ahead in long-horizon tasks but fails to presume state space is generalised without raw sensory input.

This is where SGM would come in as an useful alternative. SGM is able to store observations as nodes and learaned distances as edges and feasible transitions in a sparse memory. This approach would aggregate states according to two-way consistency objective, whiles adapting classic state aggregation criteria to goal-conditioned objective.

1) Two-Way Consistency — Sparsity: The formulas for two-way consistency can be denoted as:

$$\max(w) \left| Q^{hl}(s_1, a = w|g) - Q^{hl}(s_2, a = wg) \right| \leq \tau_a \quad (1)$$

$$\max(s_0) \left| Q^{hl}(s_0, a = w|g) - Q^{hl}(s_0, a = wg) \right| \leq \tau_a \quad (2)$$

This determines how nodes are merged together from a group of nodes to form these sparse nodes, which helps to minimise memory representation. They verify that both starting points and waypoints can be exchanged without altering the planner's behavior significantly, while making it more efficient.

2) Cleanup: Afterwards, lines between nodes that go through walls should be removed, since it is impossible for an entity to travel through the walls.

3) Route Planning: Finally, SGM would facilitate route planning by representing the environment as a sparse graph where nodes are represented by observations, and between them are edges that represent the feasible transitions between these states.

VI. PLANNED PROPOSAL SUMMARY

In this report, the author shall implement features that SGM uses and simulate it using a custom robot and world in ROS2 and Gazebo. The implementation shall be able to acheive the following.

1) Create a custom robot and world in Gazebo.
2) The world should also randomly create nodes and then perform sparsification and cleanup to find feasible edges
3) The robot should then be able to navigate among different nodes and their edges using a navigation algorithm.

VII. IMPLEMENTATION

You may view the project's implementation source code here. Detailed explanation on its installation and startup can also be found in there.

In essence, we can achieve the main two goals of SGM with:

1) Sparsity — Implementation in code, by merging points in grid with TWC.

2) Long Horizon Goals — Movement of robots through different waypoints in the map.

A. Overall Project Architecture

The project was created on Python 3.10.13, ROS2 Humble on WSL2 Ubuntu 22.04. The ROS2 workspace primary consists of 4 nodes:

1) Robot description — show robot's design
2) Robot navigation — contains core code for robot movement between ndoes
3) Robot interfaces — custom interface types to support robot movement
4) Robot bringup — launching robot in Gazebo world

B. Creating the Custom Robot

The construction of the robot was mainly done using URDF and XACRO, which is shown in RViz (Fig. 1.). The robot was built with collision and inertia properties for the links and joints for testing purposes. For its movement I provided the robot's wheels with the *diff_drive_controller* plugin. The robot was then launched in Gazebo aftewards, where a custom world was also created and launched (Fig. 2.).

I also created plugins for camera and laser scan.However, due to time constraints, they were not implemented further as additional features.
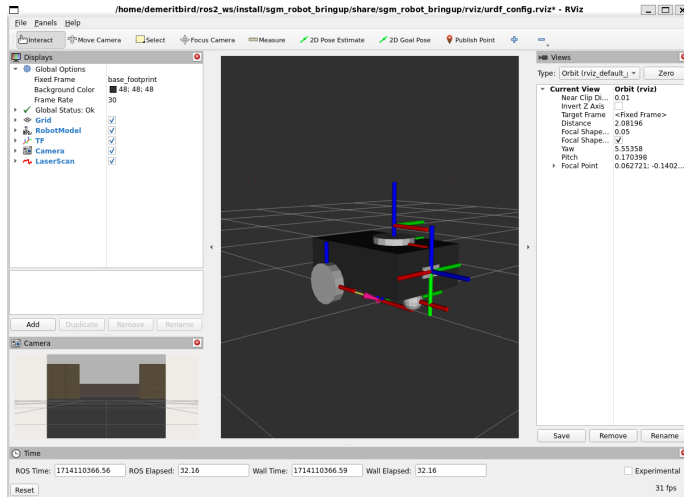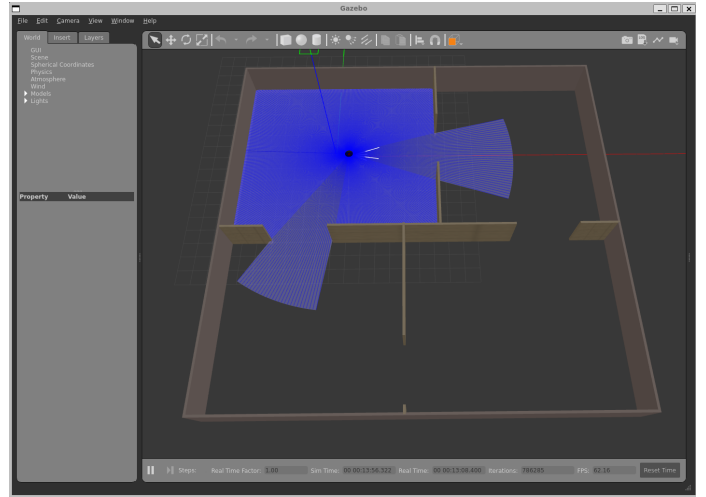
Fig. 2. Custom robot as seen in Gazebo

implementation. For the purposes of this project, I will only be using this navigation stack to get information about the map and not for its navigation features.
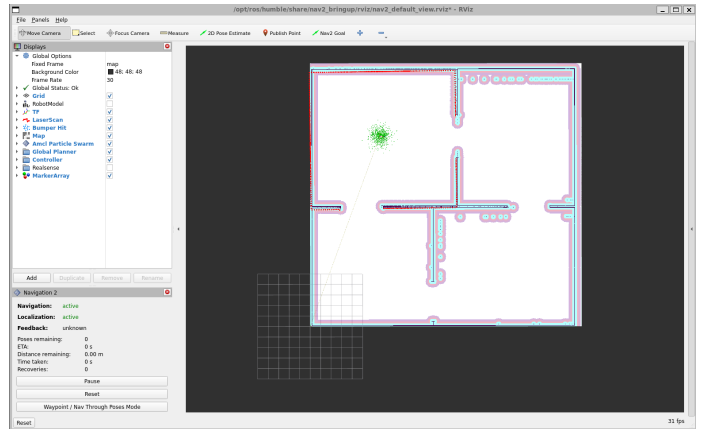
Fig. 1. Custom robot as seen in RViz

C. Integrating Robot and Map

With the robot and the map created, the robot needs to know where it is in our map. As such, I used SLAM toolkit and moved the robot around the map to allow the robot to understand the map information. In the map produced from SLAM toolkit, black pixels represent obstacles in the world (i.e. walls), and white spaces represent areas that robot can move around (Fig. 3.). Due to time constraints, I decided to lean in on ROS' turtlebot's Navigation2 Stack instead of creating from scratch to assist in publishing our custom world's map information for the robot to consume later in the

Fig. 3. Robot as understood by the Navigation2 Stack. Robot locationa and orientation can be seen after initialisation.

D. Generating and Plotting Points on World

With the map information published, I am able to use it to generate random points on the map and merge them together using two-way consistency sparsification (Fig. 4.). From there, I can link nodes that are close to one another using threshold and detect feasible edges that do pass through walls which will consequentially be used during the final navigation process (Fig. 5.).

The visualisation of the nodes is then transferred over to RViz through publishing to the MarkerArray *visualization_marker_array* topic (Fig. 6).

E. Robot Movement and Navigation

In the implemented code, there are two major nodes for robot movement and navigation, *marker_points_node* and *robot_controller_server*.
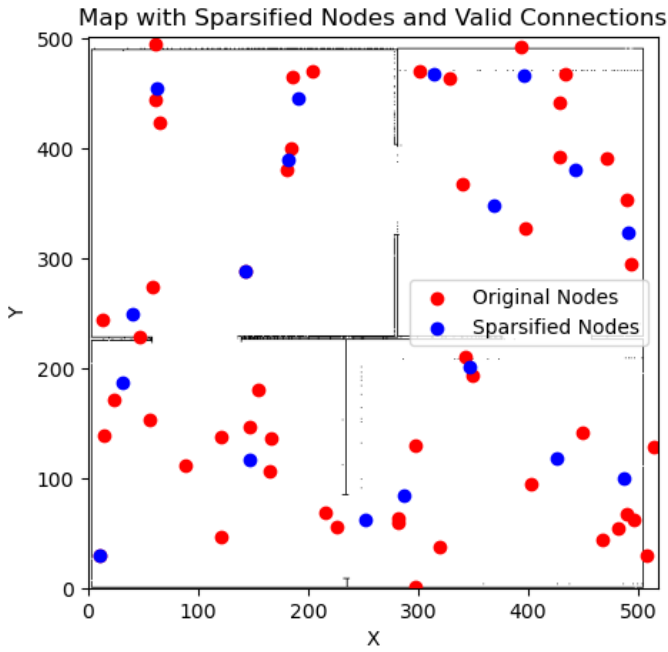
Fig. 4. Python simulation of how nodes would spawn on map (red), followed by sparsification of nodes (blue) using two-way consistency
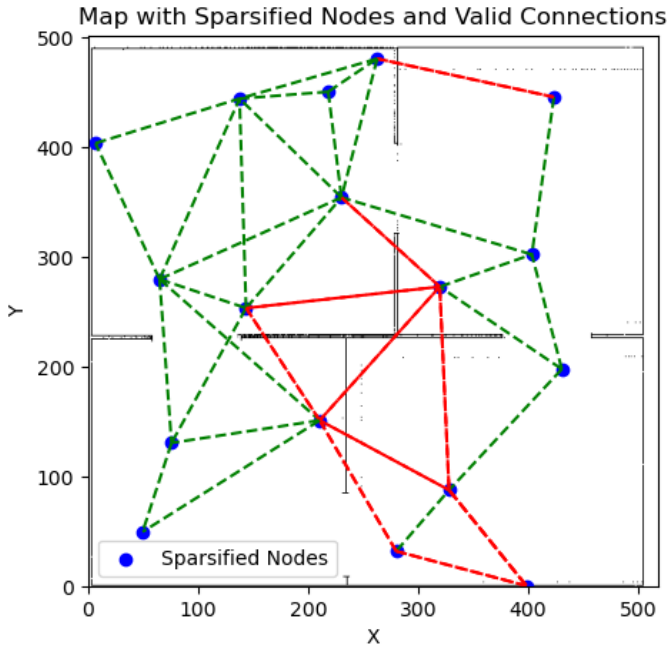

Fig. 5. Python simulation of another set of sparse nodes. Connected nodes via edges (green), edges that go through walls are non-feasible(red)

1) marker_points_node: This ROS node subscribes and collects information about the map, processes the node mentioned earlier to get the sparse nodes and feasible edges, before publishes this new sparse node information to the RViz Visualisation as well as *robot_controller_server*.
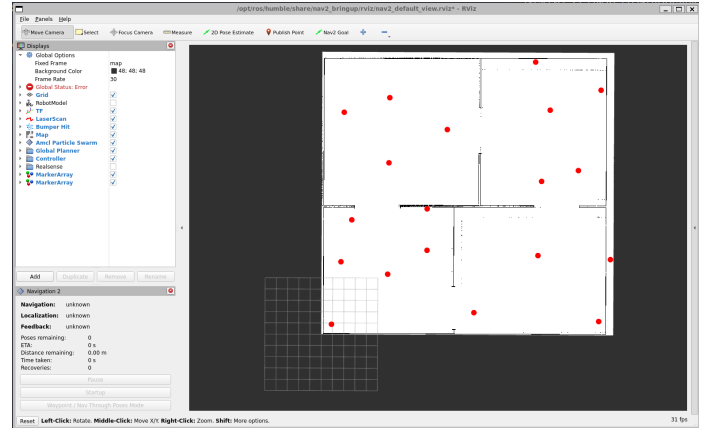

Fig. 6. Connected Nodes as seen in RViz.

2) robot_controller_server: This ROS node is a multi-threaded ROS Action Server that first localises the robot in the map and moves it to the nearest available node for initialisation. It listens for future goals of new target sparse nodes to travel to, calculates the fastest way to get there through edges, and translates this information into commands for the robot to perform. These commands come in the form of the robot first rotating anti-clockwise to face its destination sparse node, then moving a linear distance, and stopping at its different waypoints.

With these commands, the robot will then move from node to node to its destination node. In this implementation, since SGM is lenient on the navigation algorithm, I used the Breadth-First Search algorithm (BFS).

If the sparse node is unavailable to travel to (i.e. out-of-bounds), then the Server will reject this goal. The action server also comes with a feedback system to tell clients about which node location it will go to. The current goal policy for this server is to reject new goals if there is one in process, or if the robot is moving to its initial node location (Fig. 7).
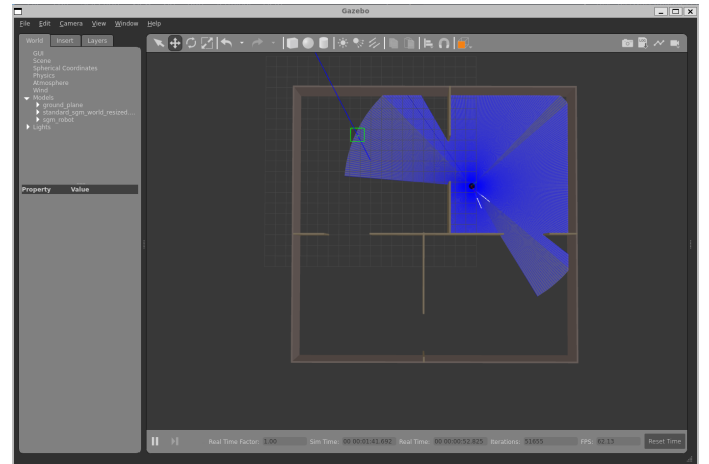

Fig. 7. Robot moving in the custom world

VIII. Evaluation and Limitations

During the development process, the robot demonstrated the ability to correctly and autonomously navigate from one node to another using the fastest efficient route available. It is able to detect nodes reliably and plan efficiently. With its goal policy, it can also handle new target nodes dynamically as well.

However, given factors such as lags in computer processing and possibly vehicle friction, deviations occur in the rotation and advancement of the robot which compounded if the entire journey was long.

Additionally, while the navigation system effectively utilized the SGM representation for path planning, certain features and optimizations of SGM were not fully implemented, given time constraints. This includes improvements towards the memory construction of the robot between different state transitions dynamically to provide increased robustness of the robot, or unimplemented plugins that could error handle much effectively.

IX. Conclusion

In conclusion, my implementation demonstrates the feasibility and effectiveness of SGM for robust planning in robotics within the ROS2 framework, even when only using its fundamental features. By integrating SGM principles into autonomous navigation, it is possible to improve future works in the field of robotics.

References

[1] Emmons, Scott and Jain, Ajay and Laskin, Michael and Kurutach, Thanard and Abbeel, Pieter and Pathak, Deepak, Neural Information Processing Systems, Sparse Graphical Memory for Robust Planning. 2020.