Multi-Agent Systems: Path Planning, Communication and Task Allocation

ROB 5994: Robotics Directed Research Report
Amitabha Deb, Srijan Pal
Advisor - Prof. Maria Gini
University of Minnesota, Twin Cities

1 Introduction

Many of the problems in the world cannot be solved by a single agent or a centralized system. Often, collaboration among various entities is required to accomplish a goal. To design an efficient multi-agent system there are several aspects that need to be addressed such as communication, information sharing, task allocation, and achieving individual goals.

In the field of robotics, multi-agent problems have constraints on time as well as resources. Several techniques of path planning, task allocation, and effective communication are applied to optimize the system. Our focus was to perform an in-depth study on these aspects and determine such techniques' advantages, limitations, and potential extensions.

Researchers in this field are actively trying to address the challenges such as uncertainty, dynamic obstacles, and complex interactions among the entities. Most of these challenges come under NP-hard problems with no efficient algorithms currently available to solve these.

2 Literature Survey

Multi-agent physical systems comprise a number of autonomous agents with specific goals. These agents can be both homogeneous and heterogeneous based on their capabilities and contribute towards a common goal. The agents usually have their own sensors, processors and actuators which help them communicate, make decisions and complete their tasks.

2.1 Path Planning

Path planning, also known as motion planning, is necessary for autonomous robots to let the robot find the shortest obstacle-free path from a starting location to a specified goal. The path is a set of states (position or orientation) or waypoints in space generated from computational algorithms, also known as path planning algorithms. The primary objective of these algorithms is to find a sequence of valid waypoints or orientations for the robot to traverse without hitting any obstacle.

There are various path-planning algorithms, the selection of an appropriate algorithm primarily depends on the specific factors outlined in the problem statement, such as whether the obstacle is static or dynamic, the availability of a map of the robot's surroundings, the desired time complexity, and other relevant considerations. Some of the most commonly used path-planning algorithms include Dijsktra's algorithm[1], A*[2], D*[3], D* Lite[4], RRT[5], and RRT*[6] algorithm. A*, D*, D* Lite and Dijsktra's algorithms are tree-based search algorithms which mean the space around the robot is represented as a graph, and the shortest-path graph is generated to guarantee the shortest path. On the other hand, RRT, and RRT* algorithms are sampling-based algorithms that are probabilistic complete, meaning that they create possible optimal paths by randomly adding sample points to a tree until the best solution is found within a specified time.

Dijkstra's algorithm[1], a tree-based search, is for finding the shortest paths between nodes in a weighted graph. It creates a list of "visited" and "unvisited" nodes. The starting node is assigned a zero distance and all other nodes' distances are set to infinity. Then, each neighbor is visited and its distance from the current node is calculated. The value is updated if the distance is less than the previously defined distance. Once all neighboring values are updated, the algorithm moves the current node of the "visited" set and repeats the process of the next neighboring node with the shortest distance. The algorithm continues until all nodes have been moved from "unvisited" to "visited".

Compared to Dijkstra's algorithm, the A*[2] only finds the shortest path to the goal, not the

shortest path from the start to all possible goals. It can be seen as an extension of Dijkstra's algorithm as A* achieves better performance by using heuristics to guide its search. A* is one of the best and most popular techniques used in path-finding and graph traversals. The algorithm initially creates an open list containing the starting node and an empty closed list. It selects the lowest cost node from the open list and moves it to the closed list. It then examines all its neighbors and their costs, taking into account the estimated distance from each neighbor to the goal. If a neighbor is not already on the open list, it is added, and its cost is calculated. If a neighbor is already on the open list, its cost is updated if the newly calculated cost is lower. The algorithm continues this process until it reaches the goal or the open list becomes empty. Once the goal is reached, the algorithm traces back through the closed list to generate the shortest path.

D* Lite[4] is an incremental heuristic search algorithm which is an improvement over the A* algorithm. The basic idea of D* Lite is to perform a series of local search operations on a graph representing the environment. The algorithm uses a set of nodes and edges to represent the graph, where each node corresponds to a location and each edge corresponds to a path between two nodes. Each node is associated with a cost value that represents the estimated cost of reaching that node, and a heuristic value that represents the estimated cost of reaching the goal from that node. The edges are associated with a cost value that represents the actual cost of traversing that edge. D* Lite works by maintaining a priority queue of nodes to be explored, with the node with the lowest estimated total cost (i.e., the sum of the cost value and heuristic value) being explored first. When a node is expanded, its neighbors are updated based on the cost of the edge connecting them and the cost of the node itself. One of the key features of D* Lite is that it can handle changes in the environment or the goal position by re-planning only the affected parts of the graph, rather than starting the search from scratch. This is achieved by maintaining a set of "dirty" nodes that have been affected by changes in the environment and updating their cost and heuristic values accordingly.

Rapidly exploring Random Trees (RRT)[5] is another popular path-planning algorithm that both creates a graph and finds a path, but the path will not necessarily be optimal. RRT*[6] is an optimized modified version of the RRT algorithm that aims to achieve the shortest path, whether by distance or other metrics. Both algorithms are based on building a tree structure and exploring the space by randomly sampling points and incrementally connecting them to the tree. The main difference between RRT and RRT* is the way they handle the tree structure. In RRT, the tree is built by connecting each new randomly sampled point to the nearest point in the tree, resulting in a tree that may have sub-optimal paths due to obstacles or uneven terrain. In contrast, RRT* is an extension of RRT that considers the cost of each path and tries to optimize the tree structure by rewiring the tree whenever a shorter path is found.

The RRT* algorithm[6] starts from the initial configuration by randomly sampling the space. It then calculates the nearest point in the tree to the sampled point and extends the tree from the nearest point to the sampled point, if the extension is feasible. The cost of the path from the root to all the points in the tree that were affected by the new extension is updated and finally, the tree is rewired by checking if there is a shorter path from the root to any other point in the tree that can be obtained by going through the new point and as a result bother the path and the cost are updated. If the algorithm reaches the goal then the path is returned from the initial configuration to the goal or else again a point is randomly sampled from the space and again the rewiring is performed. RRT* is generally considered to be more effective than RRT in finding optimal paths. However, it also requires more computation and can be slower in some cases.

When choosing between algorithms to solve a problem, it's important to compare their feasibility in the given scenario. While A* is less computationally expensive than Dijkstra's algorithm, it's limited to static obstacles. The dynamic character in the obstacles can be implemented in A* considering the dynamic obstacles to be static in every single frame of time and generating the path repeatedly for every time frame. But to generate a path from the starting position to the finish repeatedly will ultimately increase the time complexity of the algorithm.

On the other hand, RRT and RRT* are capable of handling dynamic obstacles, as their tree structures are reconfigured with each iteration after random sampling. RRT* is particularly useful for gener-

ating the optimal path in the presence of both dynamic and static obstacles. Therefore, we choose to implement the RRT* algorithm for our multi-agent task allocation problem. There are other more modified versions of RRT* like RRT* FND which takes lesser time to compute the optimal path from start to finish considering dynamic obstacles by storing the information of the explored space. But we will be sticking mainly to RRT* for our work and try to implement these more modified versions in the near future.

2.2 Communication

Adopting multi-agent systems in Robotics is challenging due to the fact that the agents need to interact with the dynamic environment as well as with each other. Such a dynamic environment makes it essential for the agents to share information to succeed. In contrast to software-based multi-agent systems, these systems mostly focus on sharing low-level information on the state of the world such as sensor data or videos[7].

The paper 'Communication Efficiency in Multi-Agent Systems'[7] also identified the shortcomings of ACLs such as KQML and FIPA-ACL in handling large low-level data and proposes an efficient and portable ACL. They introduced additional lines or backchannels in a single-line architecture to facilitate the exchange of low-level data. The establishment of backchannels, their frequency, and the content of the messages are predefined at the ACL level.

In the work 'ROS: an open-source Robot Operating System [8], they present the implementation of a structured communication layer that operates on top of a host operating system for a heterogeneous computing cluster. A ROS-based system has a number of heterogeneous hosts which are connected through a peer-to-peer topology at run-time. This peer-to-peer connection uses a name server to facilitate processes to find each other, which is called the ROS master.

A large ROS system dependent on a single master can face several challenges. Firstly, the system on which the master is running might face performance issues due to the lack of storage. Secondly, the traffic on a single network might be too high. And finally, there might be scalability issues such as name resolution conflicts. Next, we explore whether a multi-master ROS system might be able to address such limitations.

In the report 'Multi-Master ROS Systems'[link], they discuss the concepts, limitations, and possible solutions for systems built with multiple ROS networks. Each of the ROS networks has a master of its own, which acts as a facilitator of communication within the local network. Finally, the ROS masters communicate with themselves and share global information through a common network.

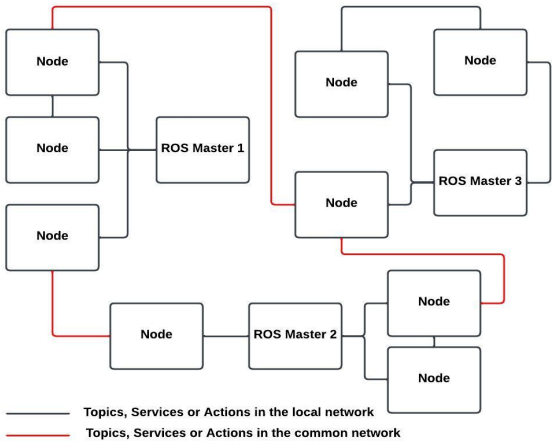

Figure 1: Sketch of a possible multi-master system.

ROS provides a solution for such a network using the *multimaster_fkie* that consists mainly of two nodes: the *master_discovery* and the *master_sync nodes*. The former is responsible to emit messages to make the other ROS masters aware of its presence and detect other ROS masters. It further checks the local roscore for changes and notifies all the other ROS masters in case of any. The latter uses the information sent and received through *master_discovery* to register the remote topics to local roscores respectively. It can be also used to configure which hosts, topics, and services need to be synced rather than syncing everything.

Although the initial testing in the report [link] points to good performance only in wired connections, still this technique has some advantages such as simplicity to use, control of the information being shared, and periodic synchronization of a global and local network.

2.3 Task Allocation

Task allocation refers to the distribution of tasks and responsibilities to the agents in a system. Tasks can be allotted in both a centralised and distributed manner. In a centralised architecture, a central system typically allocates the task based on the central computer's knowledge of the agents' availability

and cost. While in a decentralised system, tasks are allotted based on the local information of the agent. The centralised approach can be more optimised but it has only one point of failure.

In the paper 'A Taxonomy for Task Allocation Problems with Temporal and Ordering Constraints'[9], the author categorises the task allocation problems based on ordering constraints, precedence constraints and problem complexity. The temporal constraints deal with factors such as the duration of a task, deadlines, etc, the precedence constraints deal with the dependencies in order and the problem complexity deals with how hard the problem is. The authors introduced a new Taxonomy based on Gerkey and Matari's work[10].

The temporal and precedence-constrained task allocation problems needed an action-based method to allocate tasks in a multi-agent environment. The Temporal and Precedence constrained Sequential Single-Item auction (TePSSI)[11] algorithm uses the Prioritized Iterated Auction (pIA)[12], to hierarchically decompose the precedence constraints. Tasks are allocated based on their criticality which results in tasks with precedence constraints on a longer chain of tasks being allocated earlier. The tasks are then auctioned off using TeSSI[13] algorithm sequentially in a single item auction[14] from the subset of tasks with high precedence. During the bidding phase, a temporal check is done using the Simple Temporal Network(STN) maintained by individual robots.

2.4 Robot Operating System(ROS)

Robot Operating System is an open-source, meta-operating system that provides libraries and tools to develop software for robots[link]. ROS offers a distributed and flexible framework that facilitates coordination and communication among various components of a robot.

ROS has a master node that acts as a centralized mechanism to establish connections between the nodes. ROS nodes are executables that use ROS to communicate with other nodes. The primary functions provided by ROS master are node registration, topic registration, connection management, and others. After the connection is established the nodes communicate directly using a publisher-subscriber messaging model. The nodes can share messages on several topics by publishing them to specific topics. Other nodes can subscribe to these topics to receive the data and process them.

One of the advantages of ROS is it uses a distributed framework of processes. This allows the processes to be loosely coupled so that the processes can run independently and easily reused. Another advantage of ROS is the huge repository of existing functionality along with vast community support, which allows code to be implemented quickly and reliably. Moreover, ROS is language dependent, appropriate for large-scale deployment, and easily tested with the help of 'rostest'[link].

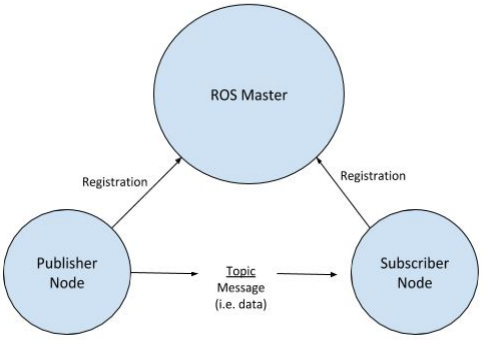

Figure 2: ROS topic communication. [link]

2.5 LiDAR and SLAM

The primary sensor that we are using in our experiments on Turtlebots is LiDAR, which stands for Light Detection and Ranging. LiDAR is an active remote-sensing system, which means that the system itself generates light - to measure obstacle distance around itself. In LiDAR, light is emitted from a rapidly firing laser, which travels and reflects off of obstacles. The reflected light energy then returns to the LiDAR sensor where it is recorded. A LiDAR system measures the time it takes for emitted light to travel to the obstacle and back. That time is used to calculate the distance travelled. Distance travelled is the key component of a liDAR system that identifies the location of all the obstacles with respect to the orientation of the TurtleBot. As the TurtleBots are considered to be placed on a 2-dimensional floor we are using a 2-dimensional LiDAR for our work.

SLAM stands for Simultaneous Localization and Mapping. It is a technique used in robotics to build a map of a robot's surroundings while simultaneously localizing the robot within the map. The robot uses LiDAR data to estimate its own position and to create a map of the environment, which it can then use to navigate autonomously. The process involves comparing the current LiDAR scan

with previous scans and the map, to identify new obstacles or changes in the environment, and to update the map accordingly. As the robot moves through the environment and collects more data, SLAM algorithms combine the new information with the existing map and robot position estimate to create a more accurate map and localization estimate. This is what makes mobile mapping possible and allows map construction of large areas in much shorter spaces of time as areas can be measured using mobile robots, drones, or other autonomous vehicles.

One of the most popular applications of ROS is SLAM which uses LiDAR sensor data to generate a map of the unknown environment while simultaneously localizing the robot in the map. There is a separate SLAM node that brings up the RVIZ window where the map of the robot surrounding can be visualized using LiDAR data from the robot. If there are multiple robots present in the same environment we can even merge the maps generated from the multiple robots while simultaneously localizing all the robots in the map using the multi-robot-map-merge package in ROS.

3 Experiments and Results

All the experiments we have performed are using ROS Melodic and TurtleBot3. We have referred to ROS tutorials, Gazebo tutorials, TurteBot3 tutorials, and several GitHub repositories for our work.

3.1 Getting Familiar with ROS and Gazebo

We followed the ROS WIki[ROS Wiki] to install ROS melodic along with the Gazebo simulator for Ubuntu 18.04. The Wiki helped us understand ROS architecture and its features. We then followed the tutorials to set up our workspace and test out the basic publisher and subscriber nodes.

Next, we followed the TurtleBot3 Gazebo simulation [link] documentation to get the robot working in a simulated environment. We familiarized ourselves with TurtleBot3 topics such as 'odom', 'cmd_vel', 'scan' and the message types 'Twist', 'Laserscan', and 'Odometry".

Finally, we were able to successfully control single and multiple TurtleBots by sending them Twist messages in both real and simulated environments. The multiple turtle bots were assigned a namespace of their own to avoid any conflict in respective commands.

3.2 Mapping

Mapping of the environment is one of the most essential parts of a multi-agent system. Maps are generated using the information gathered from the SLAM combined with the LiDAR sensor. The maps generated are visualized in Rviz, which can be stored as an image using map_server. We started experimenting with generating maps in Gazebo simulation using a single TurtleBot which was controlled using the teleop_twist_keyborad command manually. We successfully extended the idea to generate a single merged map from the information gathered by multiple robots present simultaneously in a single environment using multirobot_map_merge.

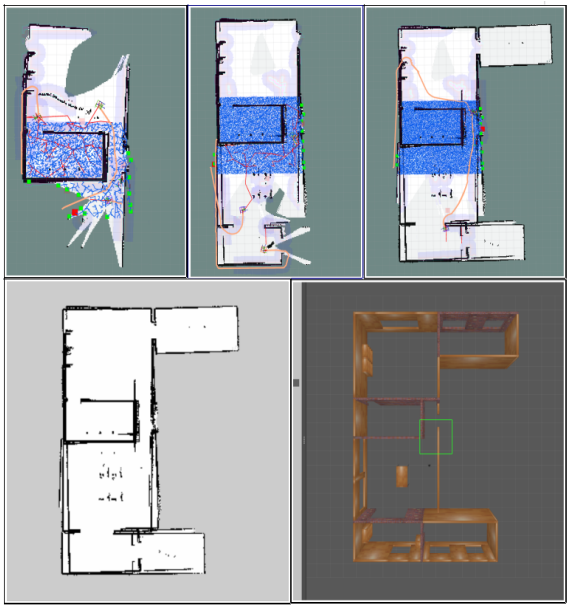

Figure 3: Multi-robot Autonomous Exploration using RRT*

Going forward we were able to autonomously explore a given region in space and generate a map of the surroundings using RRT* algorithm with both a single robot and also with a multi-robot system. Figure 3 shows how three robots are exploring a given environment in a Gazebo simulation and gradually generating the map from top left to top right. On the bottom left, we have the final map image of the environment autonomously generated by the robots and saved using the map_server. It can be clearly seen that the map is almost accurate with the original top view of the environment which is shown on the bottom right of the figure.

3.3 Path Planning

3.3.1 RRT/RRT*

We have implemented the RRT and RRT* algorithms for a single robot. We have seen that the RRT and RRT* algorithms are much faster than the search based algorithms beacuse of the random selection of points. For RRT: time used = 5.094030141830444, Total distance = 44.73909546141162 RRT*: Time used = 64.99294900894165, Total distance = 17.98421254891342

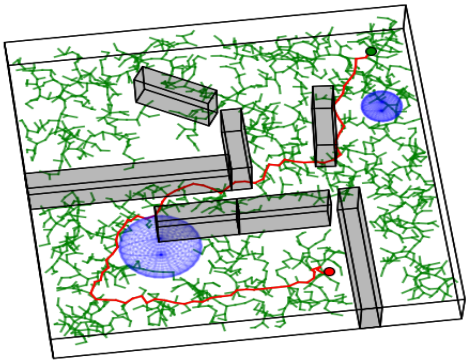

Figure 4: RRT Implementation

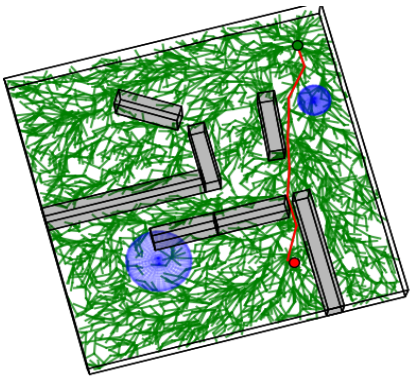

Figure 5: RRT* Implementation

We see that the RRT* gets a more optimal path in terms of distance while the RRT algorithm optimises the time. The distance is optimised in RRT every time a better path is found in the subsequent itteration.

3.3.2 D* Lite

The version of D* Lite is basically running the A* search in reverse starting from the goal and attempting to work back to the start. The solver then gives out the current solution and waits for some kind of change in the weights or obstacles that it is presented with.

While implementing the algorithm we notice that it is extremely dynamic. Initially, it computes a path from the start to the goal and incrementally repairs the path keeping its modifications local around the robot pose. As opposed to repeated A* search, the D* Lite algorithm avoids re-planning from scratch and is much more effective in a dynamic environment. Further, we have noticed that the re-planning time is very small.

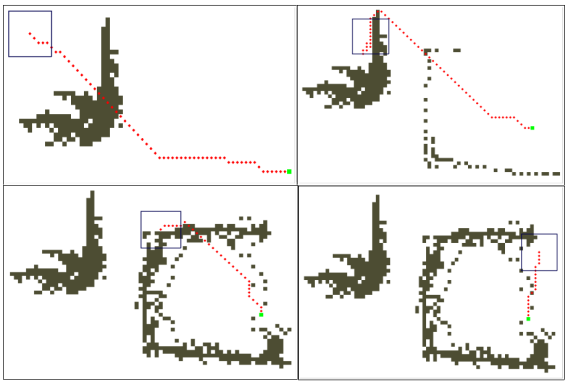

Figure 6: D-star-Lite Implementation

3.4 Communication

For a multi-agent system, knowing the current locations of the other robots in the environment is advantageous in many ways. The robot can decide on this information to subscribe to specific topics of any particular robot based on its distance. We have generated three nodes on three TurtleBots that publish and subscribe to the same topic called 'Location'. The location is derived from the 'Odom' topic, which can be further fine-tuned using liDAR and map data. The location data is a custom message which has the x and y coordinates as well as the TurtleBot id. This data is used to calculate the absolute distance of the TurtleBot with respect to every other TurtleBots.

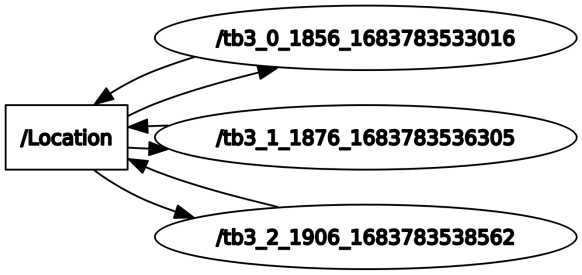

Figure 7: rqt_graph of the architecture

4 Conclusion and Future Work

Multi-robot systems have great potential to assist humans in numerous sectors such as warehouses, rescue operations, etc. We have seen that there exist great algorithms such as RRT* for dynamic path planning for multiple robots, TePSSI for allocating the tasks considering the temporal and precedence constraints, and ROS framework for sharing information within the system. These systems when combined together provide a great backbone for realizing a robust multi-agent system. But there are numerous limitations and challenges such as collision avoidance, scheduling, etc which need to be identified and addressed in the future.

During the course of our study, we have developed a great understanding and hands-on experience with the ROS framework. We have learnt about the various aspects of Multi-agent physical systems and their advantages and limitations.

This work can be further extended to combining all the aspects and implementing an ecosystem. This can have practical use cases such as rescue operations during a fire where a heterogeneous multi-agent system can be used and tasks can be defined as rescuing humans, rescuing objects, and extinguishing the fire. These tasks can be further broken down and defined at a smaller level.

References

[1] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[2] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[3] Anthony Stentz. The d* algorithm for real-time planning of optimal traverses. 04 2011.

[4] Sven Koenig and Maxim Likhachev. D*lite. pages 476–483, 01 2002.

[5] Steven M. LaValle. Rapidly-exploring random trees : a new tool for path planning. *The annual research report*, 1998.

[6] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning, 2011.

[7] M. Berna-Koes, I. Nourbakhsh, and K. Sycara. Communication efficiency in multi-agent systems. In *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA '04. 2004*, volume 3, pages 2129–2134 Vol.3, 2004.

[8] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[9] Ernesto Nunes, Marie Manner, Hakim Mitiche, and Maria Gini. A taxonomy for task allocation problems with temporal and ordering constraints. *Robotics and Autonomous Systems*, 90:55–70, 2017. Special Issue on New Research Frontiers for Intelligent Autonomous Systems.

[10] Brian P. Gerkey and Maja J. Matarić. A formal analysis and taxonomy of task allocation in multi-robot systems. *The International Journal of Robotics Research*, 23(9):939–954, 2004.

[11] Ernesto Nunes, Mitchell McIntire, and Maria Gini. Decentralized multi-robot allocation of tasks with temporal and precedence constraints. *Advanced Robotics*, 31(22):1193–1207, 2017.

[12] Mitchell McIntire, Ernesto Nunes, and Maria Gini. Iterated multi-robot auctions for precedence-constrained task scheduling. In *Proceedings of the 2016 International Conference on Autonomous Agents amp; Multiagent Systems*, AAMAS '16, page 1078–1086, Richland, SC, 2016. International Foundation for Autonomous Agents and Multiagent Systems.

[13] Ernesto Nunes and Maria Gini. Multi-robot auctions for allocation of tasks with temporal constraints. *Proceedings of the AAAI Conference on Artificial Intelligence*, 29(1), Feb. 2015.

[14] Michail Lagoudakis, Evangelos Markakis, David Kempe, Pinar Keskinocak, Anton Kleywegt, Sven Koenig, Craig Tovey, Adam Meyerson, and Sonal Jain. Auction-based multi-robot routing. pages 343–350, 06 2005.

Path Planning Resources - MIT Resource, GitHub
ROS Wiki - ROS Wiki
Multi-Master ROS Systems - Technical Report
Lidar - LiDAR Reference
RRT for SLAM Application - RRT and SLAM
Multi-Robot SLAM - Multi-Robot SLAM
D-star Implementation - GitHub
Turtlebot3 eManual - eManual

5 Acknowledgement

We want to extend our sincere thanks to Ebasa Temesgen (PhD CS Student at the University of Minnesota). He mentored us throughout the entire semester and helped us with tutorials, ideas, debugging, etc.