Multi-Agent Path Planning II: Decentralized Path Planning

Navid, Dong-Ki, Bobby, Tom, Thomas, Junbin

I. Summary of motivation

The motivation for distributed (or decentralized) multi-agent path planing is borne out of necessity, whether its a shortcoming in computation capacity, communication latency and bandwidth, or available sensor information. In the use of multiple agents, great advantages in productivity or coverage are gained, but limitations in the factors above will be reached. For these reasons, distributed functionality is key to enabling the system to handle real-world systems with limitations. These factors inform the multi-agent team's selection of strategy. In handling task allocation, path planning, and deconfliction on a per-agent local basis, the problems are broken down into computationally feasible chunks and a certain level of robustness is maintained in regard to challenges such as communication throughout a system. For instance, if the communications are limited to agents which are near each other, these algorithms maintain their capabilities on a subset of agents and mesh with new agents as they come into range,

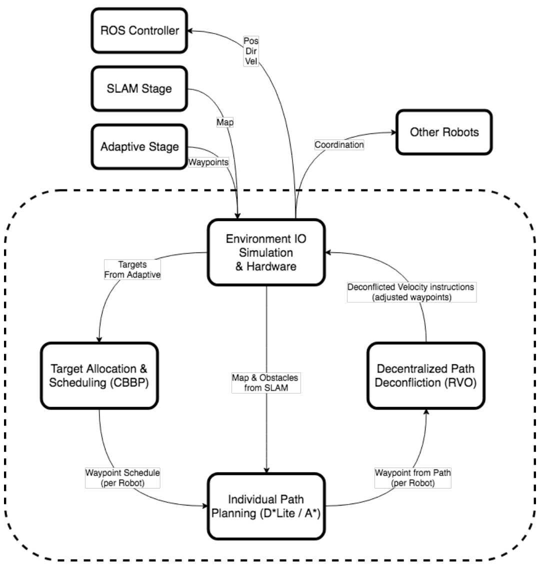


Fig. 1. MAII Architecture.

II. Problem statement

The multi-agent team framed its challenge as producing a pipeline which translated sets of target goals to allocated goals reachable by providing deconflicted optimal-path-based waypoints to system. In addition to this algorithmic breakdown, the problem included building up the ROS-based architecture around this and providing an interface to this functionality to the rest of the system. This ensured that our subsystem could successfully retrieve and deliver I/O from other subsystems, such as the SLAM subsystem, the adaptive subsystem, and the underactuated subsystem. The architecture considered is shown in the Figure.

III. Contribution of work

The contribution of technical work is as follows.

- Junbin and Thomas wrote the task allocation system.
- Navid adapted this to the ROS framework. Navid also integrated all the developed software into one within the ROS framework.
- Bobby implemented D*Lite in the ROS system.
- Tom implemented A* in the ROS system as alternatives in the path planning stage.
- Dong-Ki implemented the RVO module and simulation

Team members devised tests for their modules to evaluate effectiveness along the way. The team's primary contribution to the original algorithms considered is a combining of the path-planning (either A* or D*Lite) as a global planner and RVO functionality as a local planner. In this way, we could achieve near-optimal path planning and collision-free path planning at the same time. Another primary contribution is a comparison of D*Lite and A* for the path planning. An interesting aspect of the project has been to run the decentralized iterative target allocation algorithm (CBBA) repeatedly as the agents move in the state space. This may lead to an altered target allocation as the agents observe new targets, obstacles and neighboring agents. Furthermore, we have obtained some insight into what properties a reward function for CBBA should possess, and how we can we can use shortest path algorithms to find its values.

Individual Report

Thomas Leech

IV. Abstract

Grid search algorithms, such as BFS, A*, or D*, have been around for years but their adaptations to multi-agent applications have all been too slow for practical use. We look at a newer technique, Reciprocal Velocity Obstacles(RVO), in combination with traditional search algorithms to find an efficient way to generate conflict free paths.

V. Motivation

Many of today's multi-agent applications rely on a centralized approach because the central computer has a complete view of the agents and can make well informed decisions. This approach, however, is often not applicable in real world scenarios where communication is limit. The main case study for this class, the underwater exploration robots, are a great example of how a centralized approach will break down in a region where perfect communication is not possible. Another problem with centralized approaches is robustness to failing agents. If the central computer goes down, the remaining agents have no ability to self navigate. This is much more of a problem in applications where a leader is randomly chosen from a group of homogeneous agent. Both of these issues motivate the need for distributed algorithms that allow agents to act independently in cases of limited communication or agent failure.

VI. Problem Statement

Our focus was on the problem of multi-agent path planning. First, we take in a list of targets in the world for a set of agents to visit. The targets are distributed amongst the agents to maximize some measure of efficiency. We also deal with cases where certain agents may be uniquely capable of solving certain problems or reaching certain locations. We then generate paths for each agent individually without concern for other agents in the area. Especially in dense maps (those with many agents in a confined area), it is likely that these individually generated paths will intersect and therefore lead to a collision. Our goal is then to either generate new paths which are collision free or implement some other protocol so that we resolve these collisions and produce paths and trajectories for each agent that is collision safe.

VII. Background

A. Path Planning

As mentioned in the previous section, the first step of generating paths and trajectories is to have each agent independently solve for a path using standard path planning algorithms. The first algorithm we look at is A*. A* has been around for many years and implements a very simple concept. When looking to expand nodes in its search, it weights each possibility by the cost to reach that node as well as some heuristic value that gives an indication of how close that node is to the goal. Often this heuristic function will in fact be the Euclidean distance between the current node and the goal node. Choosing a heuristic function that underestimates the actual distance to the goal node will ensure that the search returns an optimal path. This is why Euclidean distance is often a good choice since it represents the fastest possible path to the goal, any actual path will be at least as costly as the euclidean distance.

A* is fantastic and widely used algorithm but for applications in robotics, it can often become expensive. Robotics applications often have changing maps and goals and each time one of those features change, A* must rerun from the beginning, learning nothing from its previous time. On large maps this extra computation can be substantial. D* is an algorithm that was designed to address this problem. More recently, D* lite has come to replace D* since it is simpler to implement and guarantees a result that is at least as good if not better than D*. D* lite achieves greater efficiency by saving previous costs associated with nodes and then when a change to the map is detected, it recomputes only the parts of the path that have been potentially altered by the change. This allows agents to repeatedly call D* lite as their knowledge of the world grows and not be required to repeat the same intensive search each time.

B. Reciprocal Velocity Obstacles

A fairly recent algorithm for solving the multi-agent path planning problem is Reciprocal Velocity Obstacles(RVO). RVO is an extension of Velocity Obstacles(VO). In the VO approach, first, an agent calculates the ”velocity obstacles” between itself and all the other agents in the vicinity. A velocity obstacle is the set of all velocities such that if both agents continue along that velocity they will at some point in the future collide. An example of a velocity obstacle is shown in Figure 2. The shaded gray area is the velocity obstacles and the arrows are the velocities of the agents.

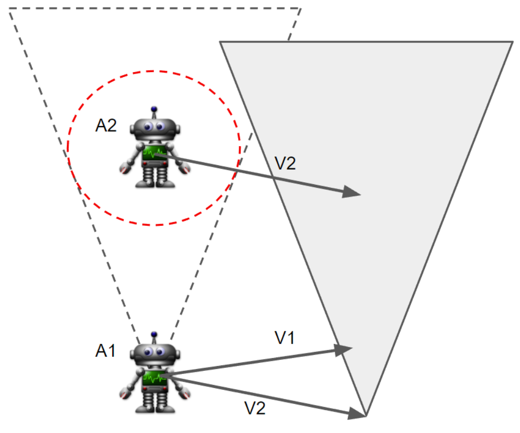


Fig. 2. Example velocity obstacle for two agents.

To create an algorithm for path planning out of the concept of velocity obstacles, we first find an optimal velocity and then adjust it to avoid any velocity obstacles. The first step of finding an optimal velocity is done by choosing the maximum velocity allowable by the physical constraints of the agent in the direction of the goal. The actual velocity chosen for the agent to follow is then found by minimizing the cost function:

$$penalty(v) = w_i \frac{1}{tc(v)} + ||v^{pref} - v||$$

Where the second term is the difference between the optimal and the chosen velocities and the first term is a large weight associated with choosing a velocity that falls within a velocity obstacle. We allow the algorithm to choose a velocity within a velocity obstacle to avoid deadlock in a situation where all possible velocities fall in a velocity obstacle.

The algorithm as describe so far achieves its purpose of having all agents reach their goals without colliding with one another. When two agents are approaching each other though, because the velocity obstacle only takes into consideration the current velocities of the agents, it can lead to oscillations where the velocity obstacles vary greatly from one timestep to another. To combat this, we use reciprocal velocity obstacles. RVO's are the same as VO's in every way except that when performing the translation of the apex of the RVO, we translate it by the average of both agents' velocities instead of just the second agent's. This results in much smoother trajectories and less oscillations.

VIII. Method

To solve our entire problem requires a combination of many different components. The first stage is to use our target allocation algorithm to divide possible targets amongst the available agents. To save on time, instead of redoing the target allocation every timestep, we do a single allocation to give each agent a goal. When an agent has reached its goal within some margin of error, it then requests a new target at which point allocation happens again. Putting the responsibility of managing the queues of possible targets onto the target allocation node frees that path planning components so that they can devote their time to the more computationally intensive path planning problem.

With our new goal point found, we now have to plan a path for our agent to reach that goal. In previous work, grid search algorithms like A* and D* lite have been used separately from algorithms like RVO. A* and D* lite both suffer from the problem of collision avoidance in multi-agent scenarios. Algorithms to deconflict paths generated by A* and D* lite are exponentially difficult with respect to the number of agents. This means the problem is nearly impossible to solve real time with more than a couple agents. RVO presents an attractive alternative because it does not scale with the number of agents and can easily be used as a real time algorithm. The problem attached to RVO is that it is essentially a gradient descent approach. Each timestep simple picking the velocity which most quickly approaches its goal. Because of this, RVO falls victim to all of the same problems faced by any gradient descent algorithm; namely local minima. If an agent running RVO approached a concave obstacle it would become stuck at the local minima since any direction it moved, at the next timestep, the preferred velocity would point it back toward the local minima.

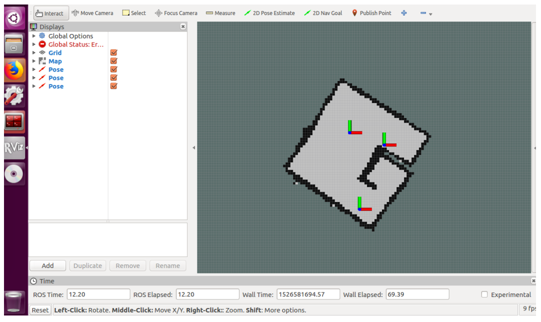
These two path planning algorithms are therefore used together so to compensate for each others weaknesses. Traditional path planning algorithms are good at finding an independent path without taking into account collisions. RVO is good at avoiding collisions given the area is made up of only convex objects. We can therefore use A* or D* lite to find an initial path over a coarse grid and without worrying about collisions and then use the waypoints generated to direct RVO. This effectively solves our problem of complexity with A* and D* lite because they no longer have to consider collisions. It also effectively solves the problem of local minima in RVO since the waypoints RVO will be navigating are guaranteed to be object free. The agent will move from waypoint to waypoint while using RVO to choose velocities and avoid collisions with other agents.

For this application we implemented both A* and D* lite so that we could have a comparison between the two and evaluate if and how much of an improvement D* lite offered over A*. As a member of the lecture team for the first portion of the assignment, I was not as expert on the implemented code as other members of the team so I fell into a position of implementation. I focused on building an A* implementation into our pipeline. As well as implementing A*, I generally helped the implementation where I could, discussing changes to the API, testing various aspects of the pipeline, and integrating pieces of code from the implementation team into ROS.

IX. Implementation

A. API

The inputs to our pipeline are straight forward: an occupancy grid represent the map of the world, our current location in the map, and a list of goal points to visit. The occupancy grid and current location were supplied by the slam team while the goal points were supplied by the adaptive sampling team. We also had a further interaction with the adaptive sampling team where our target allocation would request more points after one of the agents' queue of targets became empty. The simulation of our occupancy grid and localization of agents can be seen in Figure 3.

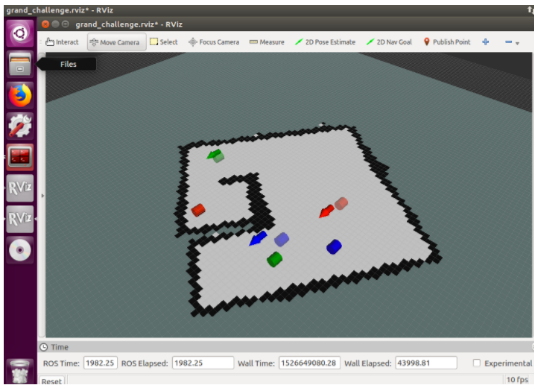


ROS Time: 12.20 ROS Elapsed: 12.20 Wall Time: 1526581694.57 Wall Elapsed: 69.39

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click:**: Zoom. **Shift**: More options.

Fig. 3. Example simulation.

We also found that rviz was a powerful tool to help in debugging since we can use it to visualize paths and desired velocities. Below in Figure 4 is an example of using rviz to display trajectories so we can debug why the agents were not moving as expected.



ROS Time: 1982.25 ROS Elapsed: 1982.25 Wall Time: 1526649080.28 Wall Elapsed: 43998.81

Fig. 4. Rviz simulation for debugging.

The outputs of our pipeline were more complicated and changed more frequently.

B. Implementing A*

For my implementation I decided to use the global planner included in ROS. This was and easy decision since it handled a lot of the tricky indexing and dealing with the map and would most likely be faster than any code I would write in that time. The actual implementation (as is often the case) was more difficult than expected. Firstly, the global planner package only supported c++. Secondly, the map it took was different from the occupancy grid which we were receiving from the SLAM team. The map transformation was done on the map topic callback along with the call to global planner. This setup is not ideal for ROS since it can cause the topic queue to fill up if the topic is being published faster than the code in the callback function can process the incoming data. To combat this potential problem, I set the queue size to one so that ROS knew to discard any data that came while the queue was still full. While this was certainly not an ideal scenario, the maps were small enough that it did not pose a problem

C. Underactuated

Another difference between grid search algorithms like A* and RVO is that grid search algorithms output a series of points which should be followed to reach the goal while RVO outputs a velocity to be realized. This is an important difference as we came to find out when we tried to instruct a turtlebot to move with a velocity perpendicular to its wheels. The turtlebot is an underactuated system meaning that it cannot instantaneously move with any desired velocity. It cannot physically move perpendicular to the direction its wheels are facing. We had made the assumption that because the input of the turtlebots low-level controller was a twist message, that it would handle the underactuated control for us. In reality, the turtlebot simply ignores all commands except for velocity in the x direction and angular velocity about the z axis.

One of our biggest mistakes as a team was not coordinating with the underactuated team early enough. By the time we realized our issue with giving linear velocity commands, the underactuated team had already set their API to handle an input of waypoints. To combat this problem by ourselves, we first attempted to implement our own simple proportional controller. While technically achieving the desired velocity eventually, the convergence was too slow to be useful and the robot often ended up well overshooting its targets. Our final solution was to multiply our velocities calculated by RVO by a small dt to generate a point. We then fed that point to the turtlebot's ”movebase” controller. The ”movebase” controller presented its own issues since it struggled with accepting new points before the current point had been reached. This resulted in suboptimal paths since the turtlebot would not be able to update its trajectory at the frequency intended by RVO. Even though the performance was not as good as expected, the proposed waypoints did produce valid paths and the agents reached their goals.

X. Evaluation

The lack of underactuated control made any type of end result evaluation difficult since even if RVO was operating correctly, we could not realize the trajectories fast enough to demonstrate it. One thing we were able to evaluate was the A* did not seem to be a bottle neck for the size of occupancy grid we were using. We were, however, operating in a fairly confined environment and it would be expected that A* would become more problematic as the size of our map grew. In those cases it is likely that D* lite would begin outperforming A*.

The target allocation was easier to evaluate since it was decoupled from the underactuated troubles. We found that our allocation scheme was able to successfully allocate targets to agents but that the allocations were suboptimal. This was because we were using Euclidean distance as a metric to evaluate which agents to assign. In an open world this would have worked well but in this case, it meant that often targets on the other side of a wall were allocated to an agent even though there was another agent on the correct side of the wall but further away in terms of Euclidean distance. The solution to this is to use the A* path length as the metric for assigning targets instead of Euclidean distance. We decided to forgo this method in pursuit of fixing our underactuated issues first.

XI. Discussion

The largest barrier to the full pipeline being finished was the problem with controlling an underactuated system. When proportional control failed, it became clear that a more powerful underactuated controller would be needed for the algorithms to be effective. Unfortunately our team did not have the time nor expertise to build a controller that would meet our needs. Our work-arounds were difficult to implement since there were many conversions from points to trajectories to points and back again. Our A*/D* lite algorithms produced a set of points which our RVO used to created trajectories. The RVO then converted the trajectories to a point which was given to ”movebase” which created a set of points. These points were then converted back into trajectories to be realized by the actual robot. All of these conversions compounded errors and each conversion to points shrank the resolution until the final waypoints were likely too close together to be accurately navigated by any controller.

In the future, besides building an underactuated controller, there were some other improvements that could be made. The first was previously discussed which was to base our target allocations off of an A* computed cost. In this case since the paths are not going to be recomputed, D* lite should not perform any better than A*. It would also be more effective to build a search algorithm from scratch that was not dependent on conversions from the occupancy grid to some other representation. This is largely unnecessary computation and was done primarily for ease of use and because our map was small enough that we were not running out of computational power.

Another consideration for our fairly constrained setup is that the collision avoidance algorithms for grid search, namely coordination diagrams, may be comparable in speed. The benefit of a coordination diagram is that it would simply produce points and cut back on the number of point-to-trajectory transformations needed. While this strategy is less interesting since it is know to not be applicable to larger settings, it may have actually performed better in this particular scenario.

XII. Conclusion

In conclusion, the combination of a grid search algorithm with RVO shows to be a promising pipeline if accompanied by a more substantial controller.

XIII. Self-evaluation

After being on the lecture team for the first part of the assignment, I focused mostly on the implementation. Being somewhat experienced in ROS but not the most expert on the team, I found myself focusing on odd jobs debugging mostly. This was not unlike my role in the first part of the assignment where I picked up the lecture subjects that didn’t fit in any one category and also helped write part of the code and pset. My main contribution to the grand challenge was my A* implementation. I also helped design the API and helped the task allocation team bring their software into ROS. I also spent a decent amount of time building the proportional controller that was ultimately too slow. I feel that I fit a niche in my team that was helpful despite not putting my name on many of the larger components. Overall, I think that we worked well as a team. We managed timed decently, communicated frequently, and many people had times when they stepped when the group needed it.