Description of Ideas Used in League of Robot Runners 2023

Márton Ambrus-Dobai

Faculty of Informatics, ELTE Eötvös Loránd University
Pázmány Péter sétány 1/C, Budapest 1117, Hungary
n4yhn4@inf.elte.hu

Abstract

This article describes the ideas we used to create our solution for the 2023 League of Robot Runners competition. We aimed to create a simple, easy-to-understand, and implementable solution. We incorporated techniques inspired by real-world traffic and commonly used optimization. We begin with a baseline approach using Prioritized Planning and a reservation table. Acknowledging its limitations, we implemented a simpler algorithm and progressive improvements, including multi-threading, path reuse, a mechanism to move waiting agents, and special handling of maps. Our evaluation demonstrates significant performance gains regarding tasks completed and processing time with each improvement. Nearly two magnitudes better results in the final version compared to the first. Finally, we discuss potential areas for future research, including comparisons with established algorithms and exploration of alternative techniques.

Introduction

Multi-agent pathfinding (MAPF) is a category of problems in which we have agents in a shared environment, and we have to find paths to their goals so they do not collide with each other. MAPF is an increasingly important problem in various fields, including robotics, logistics, smart cities, and entertainment. In 2023, the first League of Robot Runners competition was held to boost the research of MAPF solvers. The competition was about a special version of the MAPF problem, where the robots have to complete tasks on a 2D grid map with a combination of forward-moving or turning actions. The addition of turning actions makes this problem more interesting compared to the regular MAPF problems (Stern et al. 2019). Also, the competition was about lifelong solving instead of a single shot with strict time constraints on the solver algorithms.

Some of the known MAPF solvers include Increasing Cost Tree Search (ICTS) (Sharon et al. 2013), Conflict-Based Search (CBS) (Sharon et al. 2015), Priority-Based Search (PBS) (Ma et al. 2019). These solvers work with two levels of search algorithms—the higher lever searches for the best configuration for the lover level. The lover level tries to find a solution for a given configuration if it exists. In ICTS, the high-level searches for the shortest path length for the agents that have a solution. The low-level algorithm searches on a subset of the k-agent search space, whereas the high-level search defines the agent's path length. In CBS and PBS, the low-level solver works with single-agent paths. In CBS, the high-level search defines a set of constraints, and the low-level tries to find paths with the added constraints. In PBS, the high-level search defines a partial ordering of the agents, and the low-level uses the given ordering to find paths with Prioritized Planning (PP) (Erdmann and Lozano-Perez 1987).

While these algorithms have proven to be effective, and some of them can find optimal solutions, they are complex, and their response time needs to be faster. So, we created a solver that is more straightforward and faster. First, we experimented with a PP-based solver. Then, we created a minimal solver, which we used for experiments like using map limitations, multi-threading, path caching, and tie-breaking rules.

Our results show that the tested techniques can improve a solver algorithm's speed, efficiency, and scalability.

Metodology

Prioritized Planning

Our goal for the 2023 League of Robot Runners competition was to experiment with how well a simple solution could perform. As the competition is about solving MAPF problems, we first experimented with one of the most evident algorithms, Prioritized Planning (PP) (Erdmann and Lozano-Perez 1987), with A* (Hart, Nilsson, and Raphael 1968) and a reservation table (Silver 2005).

This approach solves multiple single-agent pathfinding problems instead of the original, more complex problem. We plan paths for the agents one after another. As an agent finds a path, it reserves the locations at the timesteps in the reservation table. Then, the following agents will consider those reservations as dynamic obstacles on the map and have to plan around that.

As the organizers made the competition to challenge the solvers with many agents, our algorithm was slow, and most of the time, it could not find solutions. As the number of agents with plans grows, there are fewer and fewer open locations on the map, so the search becomes longer or runs out of options and leaves without a solution. A lack of solution

can happen as others have proven that we can only solve a subset of MAPF instances with PP (Ma et al. 2019).

Minimal Solver

We thought about how to make a solver that can respond quickly. The simplest, practical solution is to find the single-agent paths without considering the other agents. From that, we can calculate the first action required to reach the next step on the path and return that to the agents. This results in many conflicts, which we handled by creating a conflict resolution algorithm that checks for all agents to see if the action assigned to the agent would cause a collision with another agent. In that case, this agent waits in this timestep, and the other agent can complete their action. With this solution, every agent travels the shortest path to their target but sometimes should wait to avoid collisions.

Inspired by real-life traffic control, where we regularly use one-way roads to reduce accidents and improve traffic flow, we tried to put this technique into our solution. If we represent the 2D map with a graph, we turn the map grid cells into vertices and connect them with their neighbors with bidirectional edges. Sometimes, solvers use weighted edges to guide the pathfinding algorithm to prefer one path over another, but we experimented with removing edges. In theory, fewer edges should result in shorter planning time as there are fewer options to check, and some conflicts, like edge conflicts, can be eliminated.

To evaluate this, we used a simple algorithm to limit the possible directions as we wanted it to work on every map. To create a general algorithm, we do not base it on assumptions; we base it on the fact that the map is a 2D grid. The algorithm follows a few rules. Every even row is allowed to go east, every odd row is allowed to go west, every even column is allowed to go north, and every odd column is allowed to go south. This method is suitable because it is general, can be calculated only by a vertex's location, avoids edge conflicts, and supports fast movement of the agents as it removes options for turning but leaves all the options to move forward if the previous action was also a move forward.

Sadly, this approach created disjointed parts from the maps. To address this issue, we implemented Tarjan's algorithm (Tarjan 1972) to create a Strongly Connected Components (SCC) analysis on the limited graph. We wrote another algorithm that goes through the possible edges in a cycle, and if it encounters an edge that is not allowed but would connect to a separate component, it allows it. Then, the algorithm does the SCC analysis again. This cycle goes on until there is only one strongly connected component.

Improvements

While working on implementing the above algorithms, we got ideas to improve them, and in the rest of the competition, we experimented with some of the improvements.

Multi-threading One idea often used to speed up lengthy work is distributing the tasks into smaller ones and doing them simultaneously. The machine used in the competition to evaluate submissions has many cores, and this idea can significantly reduce processing time. As we calculate agents' paths independently of each other, we can execute these pathfinding tasks at the same time. They all use the same environment to get the map, the agent's location, and the task's location, but none modify this shared environment, so there will be no data race. They do not depend on each other, so there is no need for communication and synchronization between them. We used the thread pool implementation supplied by the Boost library, which the project already requires. A thread pool is an excellent abstraction for these kinds of tasks. It manages the optimal number of threads, usually equal to the number of cores. The user code gives them the tasks and waits for them to finish. It does not have to manage threads manually regarding lifetimes and task execution. With the thread pool, the planner code goes through the agents, creates a pathfinding task for each, and puts it in the pool. After that, it waits until the pool finishes all the tasks, and then the planning can continue with the conflict resolution.

Reuse paths Another idea was to store the calculated paths for the agents. Previously, the planner only used the first step of the path, and the paths were re-calculated in every timestep. As the shortest path is not changing, we can calculate, store, and reuse it. With this improvement, we only need to find new paths for agents who completed their tasks in the previous timestep. Others use the step in their saved path, which can result a significant time saving. In this situation, we sacrifice memory to speed up computation. We often use this technique if the environment has spare memory for the algorithm, and speed is more important than memory consumption.

Move waiting robots While analyzing some simulation data, we saw traffic jams at the crossroads where agents were waiting in long lines. We must introduce a tie-breaking mechanism for these situations, as the conflict resolver algorithm is not sophisticated enough. Some complex solutions include checking future conflicts or distances from the targets and acting based on that information. However, we experimented with another idea. An idea we got from the real world. In real life, people in traffic get angry and try to find another way if possible. To incorporate this idea, the agents measure how long they are waiting. If this waiting time exceeds a specific limit, the agents will move to any direction with free space and delete their previous plans. In the next step, they will plan a new path that may be more open than the last.

Special handling of maps In a last attempt to improve our scores, we tried to craft better limitations for the maps manually. On the warehouse maps, the agents have to travel around the grid part because the rule-based limitation resulted in one direction being allowed inside the grid. We ensured that the rows and columns have alternating directions in the grid parts of these maps. For the random map, we created limitations where there are no narrow sections with both directions allowed, and we tried to minimize the number of edges while keeping the straights as long as possible. This attempt resulted in long, rectangular loops.

Results

When the organizers published the problem instances of the competition, we ran the simulations on our machine to recreate the results. The results are not deterministic for a given instance because the conflict resolution algorithm uses random generation to decide which agent can move in an equal situation. So, with these results, we cannot compare configurations with little difference. However, it is good to spot more significant improvements. We included the average results in Table 1. The computer for these simulations has a 4-core 10th-generation Intel CPU with 16 GB of RAM.

The results of our simulations underscore the potential benefits of map limitation. In particular, the rule-based limitation led to improvements in both the number of tasks completed and the processing time. However, it's crucial to note that the quality of the limitation is a key factor, as demonstrated by the effectiveness of the special handcrafted limitation.

For the warehouse maps, it was a clear improvement. The agents found shorter paths in the grid. Thus, they accomplished more tasks. Unfortunately, on the random map, this limitation only improved the instance with the most agents, but on the other instances, it resulted in the same quality at best. One possibility is that there remain too few options and too long paths, which is worse for the random map where a small amount of chaos had good results (Move waiting robots). However, with many agents, more control can be beneficial. This result means that finding a limitation that improves search time and does not decrease solution quality is not easy. We should further research how to create valuable limitations.

With the multi-threaded version of the algorithm, we could use the given machine's computational power better. Thus, we were able to calculate the same results in less time. On our four-core machine, in some instances, we saw nearly four times faster responses. On the instance with 200 agents on the random map, the multi-threaded solver was slower as the problem was small enough that the thread usage overhead was more significant than the improvements.

Then, reusing the calculated paths also resulted in nearly the same quality but in less time. In the first steps, we have to find paths for all the agents, which takes a long time, but in subsequent steps, we have to work less. With the caching, the number of pathfindings in a timestep is bounded to the number of completed tasks, not bounded to the number of agents. In some instances, this solver completed fewer tasks. This can happen as our single-agent planner does not calculate the agent's orientation at the start of the search. When we replan in every timestep, it can find "shorter" paths, but now we keep the plans so the agents travel along their original paths.

Another notable improvement was introducing the tie-breaking method, which significantly improved the number of tasks finished but resulted in more pathfindings, as shown in Table 1. Unfortunately, this improvement is temporary. At first, the agents start to spread from the waiting lines and find other paths, but in the long term, this causes traffic jams with more agents in the middle of the jam and fewer agents waiting in the lines. So, this method improved our solution in the competition, but in a real lifelong problem, it is not a good option. On the other hand, it can solve problems where agents are waiting for each other or can be beneficial on maps where the shortest paths have shared sections, like on the random map, and this is why it helped on that map the most.

Summary

This work addressed the Multi-Agent Pathfinding (MAPF) problem by implementing a pathfinding algorithm with progressive improvements. We began with a simple approach using Prioritized Planning and a reservation table. However, this method proved to be slow for scenarios with many agents. To address this, we devised a simpler algorithm and a way to limit search space. We subsequently implemented several optimizations, including multi-threading, path reuse, a mechanism to move waiting agents, and special handling of maps. Our results demonstrate that these improvements significantly enhance the algorithm's performance regarding tasks completed and processing time.

While this work focused on improving a single approach, future efforts could involve comparing our method with established MAPF algorithms. Additionally, we have to explore alternative map limitations with new algorithms, which can ensure a reduction in the number of edges but keep the length of the shortest paths at a minimum. Other ideas may contribute to better limitation generation, which we must first research and understand. Then, we can incorporate more sophisticated conflict resolution techniques as the current simple algorithm stops agents randomly instead of deciding which is better to let go. Finally, combining other MAPF solvers with our ideas is also a promising avenue for further research. We can combine the map limitation with most MAPF solvers, giving them a processing speedup. Overall, this work contributes to developing efficient solutions for the MAPF problem.

References

Erdmann, M.; and Lozano-Perez, T. 1987. On multiple moving objects. *Algorithmica*, 2: 477–521.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2): 100–107.

Ma, H.; Harabor, D.; Stuckey, P. J.; Li, J.; and Koenig, S. 2019. Searching with consistent prioritization for multi-agent path finding. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, 7643–7650.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2015. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219: 40–66.

Sharon, G.; Stern, R.; Goldenberg, M.; and Felner, A. 2013. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial intelligence*, 195: 470–495.

Silver, D. 2005. Cooperative pathfinding. In *Proceedings of the aaai conference on artificial intelligence and interactive digital entertainment*, volume 1, 117–122.

	MR23-I-01	MR23-I-02	MR23-I-03	MR23-I-04	MR23-I-05	MR23-I-06	MR23-I-07	MR23-I-08	MR23-I-09	MR23-I-10
Map	Paris	Paris	Random	Random	Random	Sortation	Random	Random	Game	Warehouse
Agents	1500	3000	200	100	400	10000	600	800	6500	10000
Sim. time	1500	3500	500	500	1000	5000	1000	2000	5000	5000
	Tasks completed									
Simple	90.5	65.5	10.5	12.5	3.5	47	3	0	53.5	17
Limited	8246.5	8881.5	91	634	20.5	3337	3.5	1	725	7612
Multi	8450.2	10882	89.6	524	25	8275.75	3.75	1	1257.7	15620.5
Reuse	7818.2	11046	139.5	593.5	13	8887.5	3.5	1.25	4334.7	14268
Move	8814.7	16512	914.2	968.25	1099	10224.5	716	143.5	5789.2	21784.5
Special	8737	17907	968.6	954	1065	101135.5	702.6	297.6	5311.2	94477.2
	Average planning time (ms)									
Simple	931.98	1990.5	1.49	0.77	4.695	3215.18	7.14	8.75	9277.5	3032.08
Limited	423.675	950.21	1.37	0.59	5.19	2435.74	7.47	10.27	6132.9	1870.24
Multi	103.488	243.79	1.02	0.836	1.8	633.86	2.41	2.75	2062.3	460.54
Reuse	5.18	8.01	0.46	0.485	0.48	8.34	0.47	0.4375	62.82	10.12
Move	7.90	42.96	0.60	0.54	0.97	172.97	1.38	1.36	447.12	122.8
Special	8.69	41.89	0.65	0.63	1.05	55.45	1.10	1.4	458.81	56.35

Table 1: Our results for comparison purposes. In the evaluations, we based the configurations on the previous ones with one modification for each. The first configuration is the "Standard". Which is the base algorithm and the map without limitation, "Limited" is with the map limited by the rules, "Multi" is the multi-threading turned on, "Reuse" is path reusing turned on, "Move" is move waiting agents turned on, and "Special" is the special handling of maps turned on on top of the previous configuration.

Stern, R.; Sturtevant, N.; Felner, A.; Koenig, S.; Ma, H.; Walker, T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T.; et al. 2019. Multi-agent pathfinding: Definitions, variants, and benchmarks. In *Proceedings of the International Symposium on Combinatorial Search*, volume 10, 151–158.

Tarjan, R. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2): 146–160.