Shadoks Approach for Lifelong Multi-Agent Path Finding

Aldo Gonzalez-Lorenzo[1], **Guilherme D. da Fonseca**[1]

[1]Aix Marseille Univ, CNRS, LIS, France
aldo.gonzalez-lorenzo@univ-amu.fr, guilherme.fonseca@lis-lab.fr

Abstract

The League of Robot Runners is a challenge about lifelong multi-agent path finding, where participants must submit algorithms to solve a set of ten instances. This technical note presents the algorithms that the *Shadoks* team used to find the best solution for six of the ten instances at the main round of 2023.

Introduction

The League of Robot Runners is a competition about solving ten instances of the lifelong multi-agent path finding problem (Ma et al. 2017). In 2023, our team, called *Shadoks*, won the "Line Honours" category for finding the best solution to six of the ten instances, and got second position in the other two categories.

Table 1 lists our best solution for each instance and the best solution known.

Instance	Algorithm	Our best solution	Best solution
I-01	**PlannerLazy**	**10385**	**10385**
I-02	PlannerLazy	49181	49186
I-03	**PlannerLong**	**3042**	**3042**
I-04	**PlannerLong**	**1741**	**1741**
I-05	PlannerLazy	7293	7432
I-06	**PlannerLazy**	**197275**	**197275**
I-07	PlannerComb	5809	5914
I-08	**PlannerSAT**	**6059**	**6059**
I-09	PlannerGame	19943	28954
I-10	**PlannerLazy**	**194677**	**194677**

Table 1: For each instance, we show the algorithm we used, the maximum number of finished tasks we obtained, and the maximum number of finished tasks found by any team in the competition. We highlight the instances for which we obtained the best solution.

While not working in the field of robotics, we have already won the CG:SHOP 2021 competition about multi-agent path finding (Crombez et al. 2022). Taking advantage of our previous work, specially our conflict-based optimization algorithm, we designed and tested a number of algorithms with several parameters and options. We note that the CG:SHOP competition is very different, since we have unlimited computing time and the agents can move in an unbounded map. We hope that our original ideas, albeit naive, may inspire researchers in the field.

Competition Environment

This section presents the rules and technicalities of the competition, see the website[1] for more details.

Each instance consists of a map and a number n of agents. The map is a bounded regular grid with obstacles. Each agent has an initial position, that is a location (a point of the grid with integer coordinates) and a direction (north, east, south or west), and an assigned task at some location. Time is discretized in timesteps. At each timestep, agents may perform the following *actions*: move forward, turn right, turn left or wait. However, agents are not allowed to move to an obstacle, occupy the same location (known as a *vertex collision*) nor exchange locations (*edge collision*). Once an agent finishes a task, that is, it reaches the location of its task, it is assigned a new task. The problem is to finish as many tasks as possible within an unknown number of timesteps.

The map of each instance is unknown although it belongs to a set of five possible maps that are known to all participants. The number of agents and their initial locations are unknown. The tasks are only known when they are assigned to an agent, that is in the first timestep and every time an agent finishes a task. The total number of timesteps is also unknown. Participants submit programs to implement a *planner* for solving the problem. A planner works in two stages:

1. In the preprocessing stage, the map and the number of agents is revealed to the planner. The planner has 30 minutes to analyze this input and precomputing information, such as distances between the locations.
2. In the planning stage, the initial locations of the agents and their tasks are revealed. The planner has one second to compute and submit the actions of all the robots. The number of timesteps is unknown, but bounded by 5000.

At the end of the execution, the participants only know how many tasks where finished in each instance. While the ten instances are not known, the participants are allowed to modify their code in order to infer some limited information

[1]https://www.leagueofrobotrunners.org

about each instance such as the map, the number of agents, and the total number of timesteps.

Algorithms

We present in this section three algorithms developed by our team. Our implementation is quite complex because we sought the best results instead of elegant algorithms, so we try to keep the presentation simple.

PlannerLong

This algorithm computes a *plan*, that is conflict-free path for each agent from its initial position to its task, which determines the actions to perform. At each timestep, we must compute a valid path for the agents that have a new task, which may need to change the paths that are already computed for other agents.

The difficulty of this approach is to find a valid set of paths for all agents because the map is bounded and has obstacles. This algorithm uses three parameters: α, β and μ. At the beginning of each timestep, we put into a queue the agents with a new task (or equivalently, agents with an empty path). Then, for each agent a in the queue, we search a path π to the task of a that minimizes the length of the path plus $\alpha \cdot \sum_{a' \in C}(q(a')+1)^\beta$ where C is the set of agents whose path has a conflict with π and $q(a')$ denotes the number of times an agent a' has been enqueued in this timestep. This is achieved with A* search. We assign this path π to a, and for each conflicting agent $a' \in C$, we remove its path and enqueue it.

The goal of this procedure is to balance the shortness of the paths with the exploration of more paths in order to find a valid plan. If we set $\alpha = 0$, every agent gets a shortest path to its task by removing all the necessary agents; thus, this is very unlikely to find an assignment of paths to all agents. On the other hand, by setting $\alpha \gg 0$, we disregard the length of the path and simply unplan as few agents as possible. The parameter β penalizes unplanning the same agent several times, forcing the algorithm to search for different paths. A large value of β will compute very different paths because we cannot unplan the same agents, whereas a value closer to 0 will take more time to explore similar paths.

However, there exist configurations in which this procedure runs into an infinite loop. To avoid such loop, if the agent a has been enqueued more than μ times, we take an agent $a' \in C$ (if there is any) and a position of its path that has a conflict with π, and forbid a' from being at that position at the same timestep.

If we find a valid assignment of paths before 1 second, we run the algorithm again with lower values of α and β. In practice, this means that we spend more time looking for a valid plan, but we find shorter paths. We keep the set of paths that minimizes $\sum |\pi(a)|^{1-d}$, where $|\pi(a)|$ denotes the length of the path of an agent a and d is the density of the instance, that is the number of agents divided by the number of non-obstacle locations of the map. The motivation of this formula is that, in a dense instance, long paths are likely to be replanned in a later timestep; therefore, optimizing their length is not important.

The A* search algorithm is guided by the exact distances. In the preprocessing stage, we precompute all pair-wise distances between locations, and for each pair of locations (p, p'), we save the directions from p that get closer to p'. If there are m non-obstacle locations in a map, these directions can be stored with $2 \cdot m^2$ bits, which fits in memory. Hence, the heuristic function h of the A* search algorithm uses these directions to estimate the distance to the task.

The best solution for instance `I-04` was found with a variant of this algorithm with two differences:

- We compute as many plans as possible in each timestep and we keep the plan that maximizes the function $\sum \max(0, 40 - |\pi(a)|)$. This means that we only minimize paths of length less than 40.
- At each planning, we reset 25% of the paths from the plan of the previous timestep.

PlannerLazy

The approach in this algorithm is to compute short paths for each agent towards its tasks instead of computing the full path. These paths have the following properties:

- There are no collisions between the paths.
- Their lengths are bounded by a parameter $\lambda \in \mathbb{Z}$.
- They may be empty.

At the beginning of the planning stage, we put all the agents in a queue and set their paths as empty. Then, each timestep is divided into two parts: updating paths (taking most of the time) and computing actions.

In the first part, we process the agents in the queue for as long as possible in the allowed time (0.91 seconds). For each agent, we compute a path towards its task avoiding the other paths using A* search. We stop this search if (1) we reach the task, (2) we find a path of length λ or (3) there are no nodes to explore because no path exists. We assign this path to the agent and move the agent to the back of the queue.

The choice of the parameter λ controls the time spent computing a path. A small value of λ allows to compute paths for many agents, while a large value allows agents to see far away and choose the best path towards their task, avoiding congestion in corridors.

In the second part, we extract the actions and prevent collisions. Note that vertex collisions may occur because the paths of the agents have different lengths. We identify the moving agents that produce a vertex collision, make them wait, reset their paths and put them in the front of the queue for the next timestep. Then, we submit the actions derived from the current paths.

A problem with this algorithm is that, since we plan agents with a bounded time horizon, it may produce congestion in narrow parts of the map. For this reason, we introduced a system of *barriers* for instances `I-06` and `I-10`. For each location of the map, we define a set of directions in which agents are allowed to move. These barriers are defined by hand and written into a text file that is read by the planner. They try to alternate directions of moving robots

between odd and even rows and columns while ensuring the existence of a path between any two locations. The barriers are not strictly enforced, but are used to guide the exploration of the A* search.

In addition, we compute a *congestion* map that counts the number of planned agents that pass by each location in the next 15 timesteps to detect congested parts of the map, such as corridors. The A* search algorithm of PlannerLazy uses this congestion map to break ties and try to avoid congestion.

PlannerSAT

This algorithm is designed for dense instances. Here, each timestep is independent. The objective is to find a next adjacent location for each agent, making as many agents as possible closer to their task, and compute the actions to move the agents to these locations.

At each timestep, we sort the agents by distance from their tasks. We call F the set of agents whose next location is closer to their respective tasks. For each agent $a \in F$, we define $B(a)$ as a bound of the number of actions required to reach its next location, which depends on its orientation. At the beginning, $F = \emptyset$. For $i = 1, \ldots, n$, we assign next locations for the agents such that:

1. Agents in F remain in F, even if they may have different locations.
2. For each agent $a \in F$, it can reach its next location within $B(a)$ steps.
3. The ith agent must get closer to its task.

We solve this problem using conflict optimization (Crombez et al. 2022, 2023). We initialize the queue with the ith agent, a_i, and add it to F with $B(a_i) = 3$. For each agent in the queue, we identify the locations that respect the first two conditions and choose the location that minimizes $\sum(q(a) + 1)^2$ for all agents a that have a vertex or edge collision with the current agent. We run this algorithm until we find a valid assignment of locations or until an agent is enqueued a number μ of times. If we find a solution, we add the agent to F and set $B(a_i)$ to the number of steps required to reach its next location.

The parameter μ then controls the time spent with each agent. A small value allows us to process many agents, while a large value will find more agents that can move towards their task.

To compute the actions to be submitted, we first make the agents that are not facing their next location turn. Then, we make the agents that cannot move because their next location is occupied by a turning agent wait and we back propagate from these waiting agents to identify all the agents that cannot move. The remaining agents move normally.

Discussion

The three algorithms presented in the previous section were designed for different types of instances depending on their density: PlannerLong for sparse instances, PlannerLazy for intermediate instances and PlannerSAT for very dense instances where it is inefficient to compute full or partial paths for all agents.

The choice of the parameters should be guided by running the algorithms on test instances. Our approach is to try to keep the planner as busy as possible during the time allowed for each timestep while comparing the results after a large enough number of timesteps. However, note that the actual results depend on the computational capabilities of the hardware used.

Conclusion

This technical note presents the three algorithms used to find the best solution for six of the ten instances of the 2023 main round of the League of Robot Runners competition. The description of the algorithms is simplified, since the actual implementation is convoluted given the need to find the best possible solution to win the competition and the limited allowed time.

We plan to further study every algorithm that we developed to better understand their efficiency on different types of instances and the impact of the different options and parameters and to compare them with state-of-the-art algorithms in lifelong multi-agent path planning.

We conclude this article with three characteristics of our methodology that could be helpful for future participants of the competition.

1. We tried very different algorithms, some of which where designed only for a single instance. Our work shows that it is inappropriate to solve all problems using a single algorithm.
2. It is helpful to have as much information as possible about the instance to solve, such as the map, the number of agents and the number of timesteps. This knowledge allows for running tests and finding the most efficient algorithm with the best choice of parameters for each instance.
3. The computation and solutions must be inspected in detail to detect pitfalls, since multi-agent path finding often produces errors such as infinite loops, unplanned agents and congestion.

References

Crombez, L.; da Fonseca, G. D.; Fontan, F.; Gerard, Y.; Gonzalez-Lorenzo, A.; Lafourcade, P.; Libralesso, L.; Momege, B.; Spalding-Jamieson, J.; Zhang, B.; and Zheng, D. W. 2023. Conflict Optimization for Binary CSP Applied to Minimum Partition into Plane Subgraphs and Graph Coloring. *ACM Journal of Experimental Algorithmics*, 28: 1–13.

Crombez, L.; da Fonseca, G. D.; Gerard, Y.; Gonzalez-Lorenzo, A.; Lafourcade, P.; and Libralesso, L. 2022. Shadoks Approach to Low-Makespan Coordinated Motion Planning. *ACM J. Exp. Algorithmics*, 27: 3.2:1–3.2:17.

Ma, H.; Li, J.; Kumar, T. K. S.; and Koenig, S. 2017. Lifelong Multi-Agent Path Finding for Online Pickup and Delivery Tasks. In Larson, K.; Winikoff, M.; Das, S.; and Durfee, E. H., eds., *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017, São Paulo, Brazil, May 8-12, 2017*, 837–845. ACM.