Anytime Multi-Agent Path Finding via Large Neighborhood Search

Extended Abstract

Jiaoyang Li,[1] Zhe Chen,[2] Daniel Harabor,[2] Peter J. Stuckey,[2] Sven Koenig[1]
[1]University of Southern California, USA
[2]Monash University, Australia
{jiaoyanl,skoenig}@usc.edu,{zhe.chen,daniel.harabor,peter.stuckey}@monash.edu

ABSTRACT

Multi-Agent Path Finding (MAPF) is the challenging problem of computing collision-free paths for a cooperative team of moving agents. Algorithms for solving MAPF can be categorized on a spectrum. At one end are (bounded-sub)optimal algorithms that can find high-quality solutions for small problems. At the other end are unbounded-suboptimal algorithms (including prioritized and rule-based algorithms) that can solve very large practical problems but usually find low-quality solutions. In this paper, we consider a third approach that combines both advantages: anytime algorithms that quickly find an initial solution, including for large problems, and that subsequently improve the solution to near-optimal as time progresses. To improve the solution, we replan subsets of agents using Large Neighborhood Search, a popular meta-heuristic often applied in combinatorial optimization. Empirically, we compare our algorithm MAPF-LNS to the state-of-the-art anytime MAPF algorithm anytime BCBS and report significant gains in scalability, runtime to the first solution, and speed of improving solutions.

KEYWORDS

Multi-Agent Path Finding, Large Neighborhood Search, Anytime Algorithms

ACM Reference Format:
Jiaoyang Li,[1] Zhe Chen,[2] Daniel Harabor,[2] Peter J. Stuckey,[2] Sven Koenig[1]. 2021. Anytime Multi-Agent Path Finding via Large Neighborhood Search: Extended Abstract. In *Proc. of the 20th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2021), Online, May 3–7, 2021*, IFAAMAS, 3 pages.

1 INTRODUCTION

Multi-Agent Path Finding (MAPF) asks us to plan collision-free paths for multiple agents on a given graph in discrete timesteps, each from a given start vertex to a given target vertex, while minimizing their *flowtime*, i.e., the sum of their travel times. MAPF appears in a wide variety of application areas, including robotics and computer games. In practical settings, MAPF problems can involve hundreds of agents. Desirable solutions are those which can be computed quickly but are also of high quality. Unfortunately, solving MAPF optimally is intractable [9]. Thus, practitioners must select between high-quality solutions for small problems (using optimal or bounded-suboptimal algorithms) or low-quality solutions for large problems (using prioritized or rule-based algorithms).

Proc. of the 20th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2021), U. Endriss, A. Nowé, F. Dignum, A. Lomuscio (eds.), May 3–7, 2021, Online.

In this work, we consider an anytime approach to solving MAPF. First, we use an efficient MAPF solver to find an initial solution fast so that a solution is usually quickly available, even for extremely challenging problems. Next, if more time is available, we employ Large Neighborhood Search (LNS) [6] by repeatedly replanning subsets of agents to improve the solution quality. Empirically, we compare our algorithm MAPF-LNS to the state-of-the-art anytime algorithm anytime BCBS [1] and report significant gains in scalability, runtime to the first solution, and speed of improving solutions.

2 MAPF-LNS

To begin with, we call an efficient MAPF solver to find an initial solution P for the given MAPF instance. Any non-optimal MAPF algorithm can be used. Then, in each iteration, we select a *neighborhood*, i.e., a subset of agents A_s, using a *destroy* heuristic (introduced later), remove their paths P_s^- from P, and replan new paths for them by calling a modified MAPF solver. The modified MAPF solver returns a set of paths P_s^+, one for each agent in A_s, that do not collide with each other and with the paths in P. Most optimal, bounded-suboptimal, and prioritized MAPF algorithms can be easily adapted to yield such a modified MAPF solver by treating the paths in P as moving obstacles. We then compare the (old) set of paths P_s^- with the (new) set of paths P_s^+ and add the paths in the one with the smaller flowtime to P. This procedure is repeated until we time out.

Adaptive LNS [4] is a very strong variant of LNS. It makes use of multiple destroy heuristics by recording their relative success in improving solutions and choosing the next neighborhood to explore guided by the most promising destroy heuristic. Formally, we maintain a weight $w_i \geq 0$ for each destroy heuristic i, that represents the relative success of destroy heuristic i in improving solutions. Initially, all $w_i = 1$. Then, in each iteration, we select a destroy heuristic i with probability $w_i / \sum_j w_j$ to generate a neighborhood and replan the paths of the agents in it. After the new paths are found, we update w_i according to how much the new paths improved the solution quality. Assume that we select destroy heuristic i in the current iteration. If the new paths are accepted, i.e., the flowtime σ^+ of the paths in P_s^+ is smaller than the flowtime σ^- of the paths in P_s^-, then w_i is set to $\gamma \cdot (\sigma^- - \sigma^+) + (1-\gamma) \cdot w_i$; otherwise, w_i is set to $(1-\gamma) \cdot w_i$. Here, $\gamma \in [0, 1]$ is a user-specified reaction factor, which controls how quickly the weights react to the changes in the relative success in improving the current solution. We use $\gamma = 0.01$ in our experiments. The weights for the other destroy heuristics remain the same.

Critical to the success of LNS is to find good neighborhoods for exploration. For adaptive LNS to be most successful, the destroy

Table 1: Results of MAPF-LNS (denoted by LNS) and anytime BCBS (denoted by BCBS). Time to sol is the runtime to the first solution. Initial and final are the sum of delays of the initial and final solutions. Subopt is the suboptimality of MAPF-LNS.

	m	Success rate		Time to sol (s)		Iterations		Initial		Final		Sub
		BCBS	LNS	BCBS	LNS	BCBS	LNS	BCBS	LNS	BCBS	LNS	opt
warehouse	50	1.00	1.00	0.23	0.03	3	14k	35	16	6	6	≤1.001
	100	0.92	1.00	1.54	0.12	23	12k	238	85	92	34	≤1.004
	150	0.92	1.00	7.65	0.94	17	6k	680	261	589	123	≤1.009
	200	0.80	1.00	27.00	3.02	5	3k	1,216	526	1,200	267	≤1.016
	250	0.28	1.00	33.04	5.62	6	2k	1,761	909	1,735	489	≤1.024

	m	Success rate		Time to sol (s)		Iterations		Initial		Final		Sub
		BCBS	LNS	BCBS	LNS	BCBS	LNS	BCBS	LNS	BCBS	LNS	opt
den520d	100	0.92	1.00	0.69	0.26	2	6k	31	25	12	12	≤1.001
	200	0.68	1.00	4.35	0.88	5	4k	118	93	75	44	≤1.001
	300	0.44	1.00	12.39	1.83	3	3k	252	208	219	95	≤1.002
	400	0.08	1.00	20.98	3.16	1	2k	590	362	590	170	≤1.003
	500	0.00	1.00	-	3.58	-	2k	-	569	-	269	≤1.003

heuristics should be *orthogonal*, in the sense that they explore different kinds of neighborhoods. We thus define two destroy heuristics as follows. Due to the space limit, we provide only algorithm sketches.

The first destroy heuristic is based on agents. We want to select an agent whose path could be shorter if some other agents were not blocking its way because replanning them together has a chance to reduce the flowtime. Specifically, we select an agent a_k that has the largest *delay* (= its travel time minus the distance between its start and target vertices). We maintain a tabu list to avoid selecting the same agent repeatedly. We add agent a_k to the neighborhood set A_s and let it perform a restricted random walk, which allows it to take only the actions that could lead to a path shorter than its current one, ignoring collisions with the paths in P. The agents that agent a_k has collided with during the random walk are added to A_s because they prevent agent a_k from reaching its target vertex earlier without collisions.

The second destroy heuristic is based on map topology. We are interested in the agents that visit the same *intersection vertex*, i.e., a vertex with a degree greater than 2, because a different ordering of the agents to traverse through the intersection vertex could lead to solutions of different qualities. Specifically, we begin by collecting all intersection vertices and selecting a random one from it. We add the agents to A_s whose paths visit the selected vertex.

The neighborhood size is another important factor for the success for LNS. We use a user-specified parameter N to determine the number of agents in a neighborhood. We add only the first N agents found by a destroy heuristic to A_s. If fewer than N agents are added to A_s, for the first destroy heuristic, we randomly select an agent in A_s and repeat the procedure; for the second destroy heuristic, we select a neighboring intersection vertex and repeat the procedure. We use $N = 16$ in our experiments.

3 EMPIRICAL EVALUATION

We evaluate MAPF-LNS on two maps from the MAPF benchmark suite [8], namely warehouse-10-20-10-2-1 of size 161 × 63 (denoted by warehouse) and den520d of size 256 × 257. We use the "random" scenarios, yielding 25 instances for each map and each number of agents. The algorithms are implemented in C++, and the experiments are conducted on Ubuntu 20.04 LTS on an Intel Xeon 8260 CPU with a memory limit of 8 GB and a time limit of 1 minute.

We first compare MAPF-LNS with anytime BCBS [1]. MAPF-LNS uses bounded-suboptimal MAPF solver EECBS [3] with a suboptimality factor of 2 to generate the initial solutions and prioritized MAPF solver Prioritized Planning (PP) [7] with a random priority ordering to replan. Table 1 reports the success rate (= percentage of solved instances within 1 minute), the runtime to the first solution, the number of iterations, the sum of delays of all agents for the initial and final solutions, and the (overestimated) suboptimality of MAPF-LNS (= flowtime divided by the sum of the distances from the start vertex to the target vertex of all agents). In 1 minute, MAPF-LNS explores thousands of neighborhoods and reduces the sum of delays by up to 2 times. It is significantly better than anytime BCBS in terms of scalability, runtime to the first solution, and speed of improving solutions.

Table 2: Results of MAPF-LNS on harder instances. All instances are solved within 1 minute.

	Initial solver	m	Sum of delays		Suboptimality
			Initial	Final	
warehouse	PPS	250	13,199	635	≤1.03
		300	18,587	1,400	≤1.06
		350	25,539	3,979	≤1.14

	Initial solver	m	Sum of delays		Suboptimality
			Initial	Final	
den520d	PP	700	20,713	4,473	≤1.04
		800	25,885	7,408	≤1.05
		900	31,888	12,186	≤1.08

When we consider only the instances for which we can find optimal solutions, the actual suboptimality of MAPF-LNS is far better. We run optimal MAPF solver CBS [2] for 1 minute on each instance and find optimal solutions for 57 instances. Among the optimally solved instances, MAPF-LNS finds optimal solutions for 45 instances and <0.01%, <0.1%, and <0.2% worse-than-optimal solutions (in terms of flowtime) for 48, 56, and 57 instances, respectively.

We also examine MAPF-LNS on harder instances that anytime BCBS cannot solve within 1 minute. Table 2 reports the results. Since EECBS cannot solve many of them either, we use the more efficient MAPF solvers PP and Parallel-Push-and-Swap (PPS) [5] (a rule-based MAPF solver) to find initial solutions. The quality of the solutions generated by PP and PPS is usually much worse than that of EECBS (e.g., see the difference between the initial sums of delays on the warehouse instances with 250 agents in Tables 1 and 2). However, as shown in Table 2, even when starting from a low-quality solution, MAPF-LNS can rapidly improve it, reduce its sum of delays by up to 6 times, and quickly converge to a near-optimal solution whose flowtime is at most 14% larger than optimal.

ACKNOWLEDGMENTS

Jiaoyang Li performed the research during her visit to Monash University. The research at the University of Southern California was supported by the National Science Foundation (NSF) under grant numbers 1409987, 1724392, 1817189, and 1837779 as well as a gift from Amazon. The research at Monash University was supported by the Australian Research Council under Discovery Grant DP190100013 and DP200100025 as well as a gift from Amazon.

REFERENCES

[1] Liron Cohen, Matias Greco, Hang Ma, Carlos Hernández, Ariel Felner, TK Satish Kumar, and Sven Koenig. 2018. Anytime Focal Search with Applications.. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 1434–1441.

[2] Jiaoyang Li, Graeme Gange, Daniel Harabor, Peter J. Stuckey, Hang Ma, and Sven Koenig. 2020. New Techniques for Pairwise Symmetry Breaking in Multi-Agent Path Finding. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. 193–201.

[3] Jiaoyang Li, Wheeler Ruml, and Sven Koenig. 2021. EECBS: Bounded-Suboptimal Search for Multi-Agent Path Finding. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*.

[4] Stefan Ropke and David Pisinger. 2006. An Adaptive Large Neighborhood Search Heuristic for the Pickup and Delivery Problem with Time Windows. *Transportation Science* 40, 4 (2006), 455–472.

[5] Qandeel Sajid, Ryan Luna, and Kostas E. Bekris. 2012. Multi-Agent Pathfinding with Simultaneous Execution of Single-Agent Primitives. In *Proceedings of the Symposium on Combinatorial Search (SoCS)*. 88–96.

[6] Paul Shaw. 1998. Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*. 417–431.

[7] David Silver. 2005. Cooperative Pathfinding. In *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*. 117–122.

[8] Roni Stern, Nathan R. Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, T. K. Satish Kumar, Eli Boyarski, and Roman Bartak. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*. 151–159.

[9] Jingjin Yu and Steven M. LaValle. 2013. Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 1444–1449.