Multi-Agent Path Finding with Temporal Jump Point Search

Shuli Hu,[1] Daniel D. Harabor,[2] Graeme Gange,[2] Peter J. Stuckey,[2] Nathan R. Sturtevant[3]

[1]School of Information Science and Technology, Northeast Normal University, China
[2]Faculty of Information Technology, Monash University, Australia
[3]Department of Computing Science, Alberta Machine Intelligence Institute (Amii), University of Alberta, Canada
husl903@nenu.edu.cn,{daniel.harabor, graeme.gange, peter.stuckey}@monash.edu, nathanst@ualberta.ca

Abstract

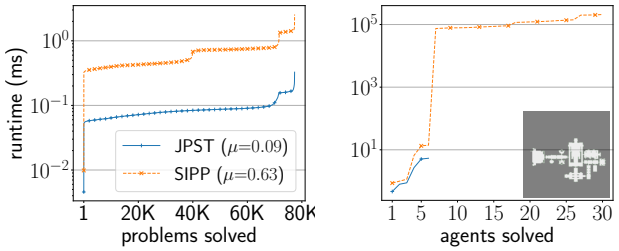
Temporal Jump Point Search (JPST) is a recently introduced algorithm for grid-optimal pathfinding among dynamic temporal obstacles. In this work we consider JPST as a low-level planner in Multi-Agent Path Finding (MAPF). We investigate how the canonical ordering of JPST can negatively impact MAPF performance and we consider several strategies which allow us to overcome these limitations. Experiments show our new CBS/JPST approach can substantially improve on CBS/SIPP, a contemporary and leading method from the area.

Introduction

Multi-Agent Path Finding (MAPF) is a combinatorial planning problem that asks us to coordinate a team of moving agents. In the most common setup, *classical MAPF* (Stern et al. 2019), the agents are situated on a 2D grid and each must move from start to target, all while avoiding fixed obstacles in the environment and avoiding collisions with other agents. The same type of grid-based setup appears as a necessary ingredient in a variety of important industrial MAPF applications, such as warehouse logistics (Wurman, D'Andrea, and Mountz 2008), mail sortation (Kou et al. 2020) and computer games (Silver 2005).

Leading methods for solving optimal MAPF, such as those in the CBS family (Sharon et al. 2015; Gange, Harabor, and Stuckey 2019; Li et al. 2021), take a decomposition-based approach to the problem. They assign individually optimal paths to every agent and then re-plan pairs of agents with conflicting (i.e., mutually incompatible) paths. Temporal constraints are added that ensure re-planned agents find new individually optimal paths, thus avoiding the conflict. An important consideration is how to efficiently compute single agent paths among temporal obstacles.

Temporal Jump Point Search (JPST) is a recent method for optimal grid-based pathfinding with temporal obstacles (Hu et al. 2021). This problem is closely related to re-planning in classical MAPF and of independent interest in robotics applications. JPST is shown to be more than one order of magnitude faster than SIPP (Phillips and Likhachev 2011), a contemporary and leading technique. A natural question is whether JPST can be directly applied in MAPF for similar performance gains. Surprisingly, the answer is no. In Figure 1a we show the raw performance of SIPP and JPST on an identical set of single-agent path planning problems. Obviously JPST is far superior. But when we replace SIPP by JPST (inside CBS), in Figure 1b, we see that JPST fails to find any solution after the first 6 agents, while SIPP solves problems with up to 30 agents. What is happening?



(a) SAPF (b) MAPF

Figure 1: For Single-Agent Path Finding (SAPF), we create a CBS Conflict Tree and use JPST and SIPP to solve the identical set of path planning problems arising at each node. For MAPF, we use JPST and SIPP as low-level planners in CBS. We add agents until CBS timeout (5 mins). We test on *lt_gallowstemplar_n*, a game map from Dragon Age Origins.

In this paper we show that the canonical move ordering of JPST, which makes the search fast, can also produce paths that are more likely to collide. The result is an overall increase in the size of the CBS conflict tree, which makes MAPF problems harder to solve. To address the issue we investigate a variety of different strategies including tie-breaking, bypassing and segment replanning. Experiments show that our revised CBS/JPST planner improves upon and can solve substantially more problems than CBS/SIPP.

Background

Conflict-Based Search (CBS) (Sharon et al. 2015) is a two-level search algorithm for classical MAPF. At the high-level, CBS searches (in a best-first way) a binary constraint tree (CT) where each node is a complete assignment of paths to agents (i.e. a plan). CBS assigns individually optimal paths to each agent and resolves collisions between pairs of *conflicting* agents by introducing collision-avoiding *constraints*.

CBS always expands the CT node with lowest f-cost. If this *current node* is conflict-free then the search terminates, having found a least-cost feasible plan. Otherwise, the current node must contain at least one pair of agents a_1 and a_2 that are in collision. For simplicity we assume they both use the same vertex v at time t (edge collisions are also possible). The node generates two child nodes. In one child CBS adds a constraint $\neg(a_1, v@t)$ to say a_1 cannot use vertex v at time t. In the other child CBS adds a similar constraint for a_2. CBS then computes (e.g., by calling A*) new individually optimal and collision-avoiding paths for a_1 and a_2.

Most CBS implementations rely on a *Conflict Avoidance Table* (CAT): a data structure that stores the currently assigned path of each agent. When replanning an agent a, CBS finds the shortest path given the current constraints on a and tie-breaks in favour of the least conflicts with the CAT. This strategy avoids introducing unnecessary conflicts and is known to substantially improve CBS performance.

Another common enhancement is (*opportunistic*) *bypassing* (Boyarski et al. 2015). After CBS replans an agent a_1, if the new conflict-avoiding path has the same cost for a_1 as in the parent CT node, CBS simply adopts the new path in the parent CT node. Since bypassing resolves conflicts without branching it can substantially improve CBS performance.

Temporal Jump Point Search (JPST) (Hu et al. 2021) is a fast and optimal A* algorithm for pathfinding in grids with temporal obstacles. The first key idea is to search in a *canonical* fashion. When moving across the grid, JPST takes vertical (V) moves as early as possible, then horizontal (H) moves, and then wait (W) moves. Other non-canonical moves are pruned. Figure 2 shows several examples.[1]

The second key idea of JPST is *jumping*. After pruning, the branching factor of many successor nodes is reduced to zero or one. JPST processes these nodes recursively, which allows the search to consider multiple grid locations in a single expansion step. The recursion stops when an obstacle impedes further progress or when the target is detected or when a type of node called a *jump point* is reached. Jump points are grid nodes with a successor that would normally be pruned by the VHW-ordering but for which the local VHW-path is invalid (due to a static or temporal obstacle). Figure 2d shows an example. Here the obstacle prevents the search from taking the vertical move earlier.

Safe Interval Path Planning (SIPP) (Phillips and Likhachev 2011) is an algorithm for optimal SAPF with temporal obstacles. The key idea (also adopted in JPST) is to represent the time dimension for a grid location v with a set of contiguous intervals $[t_i, t_j]$ (initially $i = 0$ and $j = \infty$). Temporal obstacles can block location v: from time t_o to t_{o+k}. This divides a safe interval in two: $[t_i, t_o)$ and $(t_{o+k}, t_j]$. SIPP tracks the earliest arrival of the agent in each safe interval. Later arrivals are pruned. This strategy eliminates temporal path symmetries and can be much more efficient than conventional time-expanded grid A*. When run under CBS, SIPP can also make use of a CAT, but it simply stores which SIPP nodes (location + time interval) are traversed by the path of an agent. It counts a conflict whenever two agents paths share a SIPP node. This is an approximation since it is possible for two agents to use the same SIPP node without conflicting; i.e. by one leaving before the other arrives. But, since this is only a heuristic, this is acceptable.

[1]Note that the start node (at time 0) is a special case: all available grid moves are considered canonical.

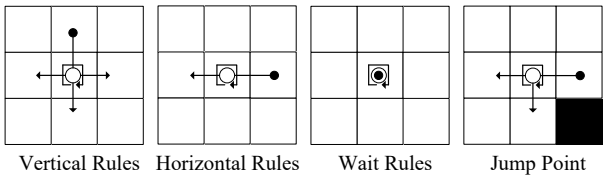


Figure 2: Canonical pruning with JPST. We show the current node (white circle), the parent node (dark circle) and VHW-canonical successors (arrows). Other moves are pruned.

Canonical Paths in MAPF

To understand how JPST can end up being unable to solve problems solvable by SIPP even though as a low level path planner it is much faster, we examine Figure 3a. Here we show the unique VHW-canonical paths for agents A_1 (orange) and A_2 (green). The paths conflict from (6,3) until (1,1). The standard CBS approach is to select the earliest conflict position (6,3)@5 and branch. When replanning A_2 CBS finds a new unique VHW-canonical path, with the same cost as before, which turns at (6,4) to (5,4), and results in another conflict with A_1. If CBS expands this child first it finds another similar conflict at (4,4). Indeed JPST needs to add constraints at each of the locations marked with a cross before finding a VHW-canonical path that does not conflict with A_1. Note that a CAT does not help JPST here, since at every replanning step there is *exactly one* VHW-canonical path; i.e., there are no ties to break. Thus the main advantage of JPST, removing path symmetries to speed up SAPF, makes conflict resolution harder in MAPF. Replanning A_1 is somewhat easier. After adding a constraint at (6,3)@5, it will then find a conflict at (6,4)@4. Adding a constraint on this location increases the path length for A_1.

Contrast this to the resolution of the same conflict using SIPP and a CAT. When replanning A_2, CBS will immediately find there is (at least) one shortest path which never conflicts with another agent; e.g. (6,8), (6,4), (2,4), (2,2), (0,2), (0,1). Also, since A_2's optimal cost is unchanged, CBS can use the new path as a bypass in the parent node. This not only resolves the conflict but also avoids any branching.

Idea

We now consider several bypassing strategies that help us fruitfully combine CBS and JPST. The main idea, *explicit bypassing*, aims to find an equivalent-cost non-VHW path, which can be adopted in the CBS parent node to resolve a conflict. During the bypassing process we can explore which segment of the path to replan, and consider alternative replanning methods (i.e., not JPST).

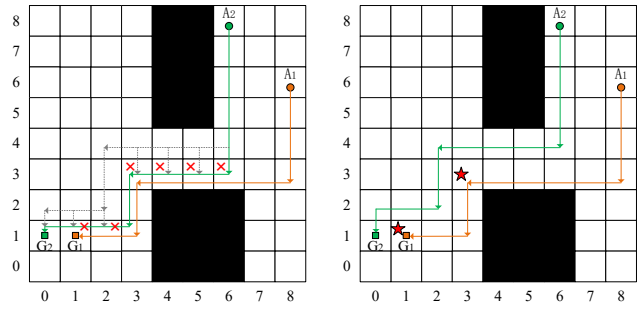


Figure 3: Examples of canonical replanning with JPST.

Explicit Bypassing The main difficulty with canonical replanning is finding an equally good path, which is not VHW canonical. We propose to find such paths using a modified bypassing procedure. The obvious approach is to add temporal obstacles at all collision points when doing the bypass check (as opposed to just the selected conflict location). This essentially "fast-forwards" the usual CBS process on the branch that leads to the bypassed path. We can do better than this by noting that we only need to add temporal obstacles at all jump points on the path of agent a_1 which collide with the current path of a_2. Examining Figure 3b we can see that adding temporal obstacles at (3,3)@8, and (1,1)@12 (the start and target are both always jump points), allows a_2 to discover the green VHW canonical bypass path.

Lemma 1 *Adding temporal obstacles at each conflicting jump point will create a VHW canonical bypass path, if a bypass path exists.*

Proof: Suppose agent a_1 has current VHW canonical path p_1, and agent a_2 has current VHW canonical path p_2. Let O be the set of vertex@time pairs in common. Let J be the subset of O which are jump points on path p_1. Suppose there exists a bypass path p of equal length to p_2 when all pairs in O are treated as temporal obstacles. We show there is an equi-length VHW path when only pairs in J are treated as temporal obstacles. Suppose p has a HV, WV or WH transition; that is, it may not be canonical. Either (a) we can invert the order of these operations to get an equi-length canonical path p', or (b) there must be some obstacle that prevents the reordering from being feasible. If the obstacle is permanent, then clearly p at this point is VHW canonical. If the obstacle is temporary then we can show it must be in J. We consider the case where we have a HV move in p, the other cases are similar. The temporal obstacle $v@t$ must be vertical to the starting point to prevent the inversion. Now on the original path p_1 the move from v cannot be horizontal, otherwise it would intersect the HV part of p, so it must be vertical or wait. If it is a wait then v must be a jump point (since we have to move off it, or it is the target). If it is vertical then on the original path p_1 the move into v cannot be vertical since it would intersect with the HV part of p. Hence $v@t$ is a jump point on p_1, and is in J. Thus, there is an equi-length VHW only using pairs in J. □

During explicit bypassing we add (due to Lemma 1) temporal obstacles at all shared jump points in the path. If there are no shared jump points we do not run explicit bypassing.

Replanning End Normally when we branch on a conflict, agents are replanned from the beginning of their path. This is required because we need to guarantee that the selected path is a shortest possible path. But when we are performing explicit bypassing checks we do not have to follow this restriction. Every time we choose the earliest conflict to branch, so there is no conflict before the current conflict location. Therefore, we can instead try replanning from the previous jump point $p@t'$ closest to the conflict $v@t$. For the end point of the bypass, we consider three options: (J) to the next jump point which is not in a straight line from $p@t'$ (since there is no alternate equal length path in this case); and (M) replanning from the previous jump point to the furthest point on the current path $m@t''$ which is Manhattan optimal distance from the conflict (i.e. Manhattan distance from v to m equals $t'' - t$); and (G) replanning from the previous jump point to the goal location. Replanning optimal segments appears previously in path smoothing for games (Sturtevant 2008).

Replanning Method Since the problem arises from the restriction to VHW canonical paths, one possibility is to relax the method used in the replanning phase. We consider several options: (A) A* and the CAT to replan the agent, removing the canonical restriction on paths; (S) SIPP and the CAT, hoping to more easily find a non-canonical path of the same length; (J) still use JPST replanning because it is faster.

Experiments

We evaluate experimentally the relative merits of using different planning and bypass methods, after planning initial paths with JPST. We use standard benchmark problems (Stern et al. 2019), where each map has n independent problems, and the goal is to simultaneously solve problems $1, \ldots, k$ for maximal k. There are 25 such `scen`(ario) files per map, each having a set of randomly generated problem instances that are evenly distributed by path length. We choose the first scenario file. All the algorithms are implemented in C++ and based on code from `libMultiRobotPlanning`, a freely available pathfinding library due to Wolfgang Hönig.[2]. We use Ubuntu 20.04 LTS on an Intel(R) i7-8700 CPU with 32GB RAM, and set a 5 mins timeout for each problem.

In Table 1 we report the max number of agents solved per map by each algorithm. Best performances are in bold, except when all algorithms have the same performance. The first four result columns are baseline algorithms used for solving single-agent problems given path constraints at the low-level of CBS. JPST is much faster than A*, so it is able to solve more problems overall, but JPST does worse than SIPP. Note that canonical A* (CA*), which first tries paths in VHW order like JPST, is also worse than A* showing how canonical paths make it harder, but not as bad as JPST compared to SIPP since it can still use a conflict avoidance table.

[2]https://github.com/whoenig/libMultiRobotPlanning

					JPST									
map	JPST	SIPP	A*	CA*	N	A-J	S-J	J-J	A-M	S-M	J-M	A-G	S-G	J-G
Berlin_1_256	54	54	54	54	54	54	54	54	54	54	54	54	54	54
Boston_0_256	59	56	52	58	44	59	59	69	59	59	77	78	77	**83**
brc202d	39	39	**40**	39	23	**40**	**40**	**40**	**40**	**40**	**40**	**40**	**40**	**40**
den312d	33	25	25	33	25	33	33	**36**	**36**	**36**	**36**	**36**	**36**	**36**
den520d	23	23	23	23	23	23	23	23	23	23	23	23	23	23
empty-16-16	26	35	35	28	18	30	30	28	**38**	**38**	37	**38**	**38**	37
empty-32-32	33	33	33	33	33	33	33	33	38	35	33	**45**	36	33
empty-48-48	56	56	56	56	53	56	56	56	56	56	56	56	56	56
empty-8-8	20	20	25	20	20	24	24	22	**26**	**26**	22	**26**	**26**	22
ht_chantry	27	**30**	27	27	27	**30**	**30**	27	**30**	**30**	27	**30**	**30**	**30**
ht_mansion_n	**49**	**49**	**49**	46	46	**49**	**49**	**49**	**49**	**49**	**49**	**49**	**49**	**49**
lak303d	25	25	25	25	25	25	25	25	25	25	25	**34**	**34**	**34**
lt_gallowstemplar_n	6	**30**	6	6	6	6	6	6	11	6	6	26	11	6
maze-128-128-10	19	21	21	19	19	19	19	19	19	19	19	**26**	**26**	19
maze-128-128-2	**18**	**18**	17	**18**	16	**18**	**18**	**18**	**18**	**18**	**18**	**18**	**18**	**18**
maze-32-32-2	16	16	16	16	14	**17**	**17**	**17**	**17**	**17**	**17**	**17**	**17**	**17**
maze-32-32-4	**25**	22	**25**	24	20	22	22	**25**	**25**	**25**	**25**	**25**	**25**	**25**
orz900d	39	**41**	24	24	23	40	39	40	**41**	**41**	**41**	27	40	**41**
ost003d	23	23	23	23	23	23	23	23	23	23	23	23	23	23
Paris_1_256	81	67	67	67	72	98	98	98	**118**	**118**	**118**	**118**	**118**	**118**
random-32-32-10	42	52	47	42	38	43	43	42	**54**	53	43	**54**	**54**	53
random-32-32-20	46	45	45	40	28	**47**	**47**	**47**	**47**	**47**	**47**	**47**	**47**	**47**
random-64-64-10	18	18	18	18	18	18	18	18	18	18	18	18	18	18
random-64-64-20	62	**66**	62	62	48	**66**	**66**	**66**	**66**	**66**	**66**	**66**	**66**	**66**
room-32-32-4	13	15	13	15	13	15	15	15	16	15	15	16	**19**	**19**
room-64-64-16	28	28	28	28	28	28	28	28	28	28	28	28	28	28
room-64-64-8	19	19	19	19	19	19	19	19	19	19	19	**20**	19	19
w_woundedcoast	29	35	29	37	29	29	29	29	**41**	40	40	**41**	40	40
Total solved agents	928	961	904	900	805	964	963	972	1035	1024	1022	**1079**	1068	1054

Table 1: Comprehensive results of different algorithm variants on the MAPF benchmark problems. Results reported are the number of agents solved per map. JPST explicit bypassing variants are two letter codes: (A) using A* for replanning, (S) using SIPP, (J) using JPST; and replanning to (J) the next following jump point, (M) the end point of Manhattan optimal path from the conflict, and (G) to the goal location. The column 'N' mean JPST version without (even opportunistic) bypassing.

The remaining columns show the results of the JPST variants. The best performance overall is JPST with A* for bypass replanning from the previous jump point to the goal, which does slightly better (11 problems) than JPST with SIPP for bypass replanning from the previous jump point to the goal. The reason A* is better than SIPP even in replanning is that the SIPP CAT is less precise than the A* CAT because it counts a conflict whenever two agents use the same SIPP node, but this may not necessarily lead to a conflict. Clearly the longer the segment we replan the better, from the previous to the next jump point, being worse than to the furthest Manhattan optimal from the conflict, being worse than to the goal. Figure 4 shows CBS success rates (% problems solved) on two maps (one small, one large), using all 25 scenario files. We see that JPST is clearly better.

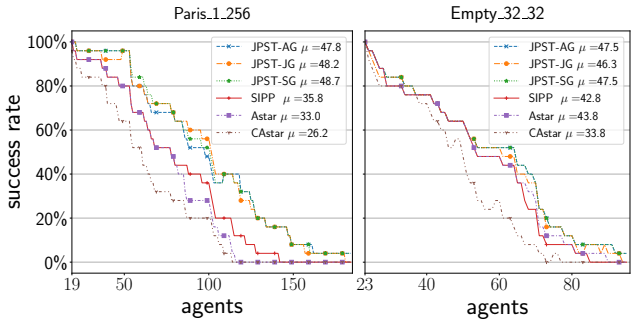


Figure 4: Success rates using all 25 scenario files per map.

Conclusion

Even though JPST is an order of magnitude faster than SIPP for low level path planning with temporal obstacles, when used as a drop-in replacement for SIPP in CBS it actually degrades performance. This is because it cannot take advantage of the CBS Conflict Avoidance Table (CAT), and this makes it harder to resolve collisions among agents. We consider how explicit bypassing strategies can overcome this weakness and we show the resulting algorithm (CBS+JPST) is substantially better than alternatives. Explicit bypassing is not valuable for SIPP, since SIPP uses the CBS CAT to try to reduce conflicts, and most bypasses will be discovered by the usual opportunistic bypassing strategy.

Acknowledgements

Research at Northeast Normal University is supported by National Natural Science Foundation of China (NSFC) under Grant No.62106040. Research at Monash University is supported by the Australian Research Council under grants DP190100013 and DP200100025 and also by a gift from Amazon. This work was funded by the Canada CIFAR AI Chairs Program. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) and support from NSF grant No.1815660.

References

Boyarski, E.; Felner, A.; Sharon, G.; and Stern, R. 2015. Don't Split, Try To Work It Out: Bypassing Conflicts in Multi-Agent Pathfinding. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 47–51.

Gange, G.; Harabor, D.; and Stuckey, P. J. 2019. Lazy CBS: implicit conflict-based search using lazy clause generation. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, 155–162.

Hu, S.; Harabor, D. D.; Gange, G.; Stuckey, P. J.; and Sturtevant, N. R. 2021. Jump Point Search with Temporal Obstacles. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, 184–191.

Kou, N. M.; Peng, C.; Ma, H.; Kumar, T. S.; and Koenig, S. 2020. Idle Time Optimization for Target Assignment and Path Finding in Sortation Centers. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, 9925–9932.

Li, J.; Harabor, D.; Stuckey, P. J.; Ma, H.; Gange, G.; and Koenig, S. 2021. Pairwise symmetry reasoning for multi-agent path finding search. *Artificial Intelligence*, 301: 103574.

Phillips, M.; and Likhachev, M. 2011. SIPP: Safe Interval Path Planning for Dynamic Environments. In *2011 IEEE International Conference on Robotics and Automation*, 5628–5635. IEEE.

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

Silver, D. 2005. Cooperative Pathfinding. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, 117–122.

Stern, R.; Sturtevant, N. R.; Felner, A.; Koenig, S.; Ma, H.; Walker, T. T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. K. S.; Boyarski, E.; and Bartak, R. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. *Symposium on Combinatorial Search (SoCS)*, 151–158.

Sturtevant, N. 2008. Memory Efficient Pathfinding Abstractions. In *AI Game Programming Wisdom 4*, 203–217.

Wurman, P. R.; D'Andrea, R.; and Mountz, M. 2008. Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses. *AI Magazine*, 29(1): 9–20.