Greedy Priority-Based Search for Suboptimal Multi-Agent Path Finding

Shao-Hung Chan[1], **Roni Stern**[2], **Ariel Felner**[2], **Sven Koenig**[1]

[1] University of Southern California
[2] Ben-Gurion University of Negev
shaohung@usc.edu, roni.stern@gmail.com, felner@bgu.ac.il, skoenig@usc.edu

Abstract

Multi-Agent Path Finding (MAPF) is the problem of finding collision-free paths, one for each agent, in a shared environment, while minimizing their sum of travel times. Since solving MAPF optimally is NP-hard, researchers have explored algorithms that solve MAPF suboptimally but efficiently. Priority-Based Search (PBS) is the leading algorithm for this purpose. It finds paths for individual agents, one at a time, and resolves collisions by assigning priorities to the colliding agents and replanning their paths during its search. However, PBS becomes ineffective for MAPF instances with high densities of agents and obstacles. Therefore, we introduce Greedy PBS (GPBS), which uses greedy strategies to speed up PBS by minimizing the number of collisions between agents. We then propose techniques that speed up GPBS further, namely partial expansions, target reasoning, induced constraints, and soft restarts. We show that GPBS with all these improvements has a higher success rate than the state-of-the-art suboptimal algorithm for a 1-minute runtime limit, especially for MAPF instances with small maps and dense obstacles.

Introduction

Multi-Agent Path Finding (MAPF) (Stern et al. 2019) is the problem of finding collision-free paths, one for each agent, in a shared environment while minimizing the sum of travel times of each agent at its goal. It has many applications, such as autonomous warehouses (Wurman, D'Andrea, and Mountz 2008), unmanned aerial vehicles (Ho et al. 2019), and autonomous vehicles (Li et al. 2023). Unfortunately, solving MAPF optimally is NP-hard (Yu and LaValle 2013), and optimal algorithms (Boyarski et al. 2015; Li et al. 2019, 2020) thus require exponential time. Consequently, researchers have explored algorithms that trade off optimality and runtime.

Prioritized algorithms are simple-yet-efficient for solving MAPF suboptimally. These algorithms decouple MAPF into several single-agent path-finding problems that minimize the travel time of the agent. Overall, priorities are assigned to agents such that those with lower priorities have to avoid collisions with those with higher priorities. Prioritized Planning (PP) (Silver 2005) and Priority-Based Search (PBS) (Ma et al. 2019) are the leading prioritized algorithms. PP assigns a unique priority to each agent, while PBS lazily assigns priorities to colliding agents. Although PP and PBS are efficient, they are neither optimal nor complete. That is, they may fail to find solutions for some solvable MAPF instances.

Several improvements have been proposed for PP, such as assigning priorities to agents according to the distances from their start to their goal locations (Berg and Overmars 2005) and restarting PP with randomly-assigned priorities upon failure (Bennewitz, Burgard, and Thrun 2001). However, few techniques have been developed to improve the effectiveness of PBS (Boyarski et al. 2022). We close this gap by proposing several improvements for PBS by adopting a range of techniques and concepts from optimal and bounded-suboptimal search algorithms. Specifically, we present *Greedy PBS* (GPBS), which is a version of PBS that prioritizes nodes in the search tree according to the number of collisions between their paths. This has been done before in the Conflict-Based Search (CBS) framework (Barer et al. 2014; Walker, Sturtevant, and Felner 2020) but never for PBS. We also add to GPBS the *partial expansions* from EPEA* (Goldenberg et al. 2014) and the *target reasoning* from CBS (Li et al. 2020). We also introduce a novel technique for prioritizing nodes in the search tree, called *induced constraints*, which is based on counting the number of priorities they represent (as opposed to counting the number of collisions between their paths). To handle situations where GPBS could not find a path for an agent for a set of priorities, we introduce the *soft restart* technique, which removes all priorities before restarting the search but keeps the paths of the agents instead of restarting the search from scratch.

Empirically, we compare GPBS with other prioritized algorithms, namely PP and PBS. We also compare it with all our improvements to the state-of-the-art bounded-suboptimal algorithm, namely Explicit Estimation Conflict-Based Search (EECBS) (Li, Ruml, and Koenig 2021), and suboptimal algorithms, namely PBS with the merging technique (PBS w/m) (Boyarski et al. 2022) and Large Neighborhood Search (MAPF-LNS2) (Li et al. 2022). Results over 6 maps with up to 2,000 agents show that GPBS with all our improvements has a higher success rate in all cases than these state-of-the-art algorithms for a 1-minute runtime limit. For all MAPF instances, GPBS outperforms PBS w/m,

Copyright © 2023, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

EECBS, and MAPF-LNS2 with a success rate that is 41%, 30%, and 10% higher, respectively. For MAPF instances with small maps and dense obstacles (namely, the Random, Maze, and Room maps described in the Empirical Evaluation section), the improvements become 67%, 30%, and 19%, respectively. A study confirms that the combination of all proposed techniques yields the best results.

Preliminaries

In this section, we define MAPF formally and describe prioritized algorithms that solve it suboptimally.

Multi-Agent Path Finding

We use the MAPF definition by Stern et al. (2019). A MAPF instance is composed of an undirected and connected graph $G = (V, E)$, also known as a map, and a set of k agents $A = \{a_1, \ldots, a_k\}$. Each agent $a_i \in A$ has a unique start vertex $s_i \in V$ and a unique goal vertex $g_i \in V$. Time is discretized into timesteps. At each timestep, an agent is allowed to either move to an adjacent vertex or wait at its current vertex. A *path* p_i of an agent $a_i \in A$, starting at its start vertex s_i and ending at its goal vertex g_i, is a sequence of vertices, indicating where agent a_i is at each timestep. We assume that an agent eventually waits at its goal vertex permanently. The path *cost* of path p_i, denoted as c_i, is the number of timesteps needed by agent a_i to move from its start vertex to its goal vertex, ignoring the timesteps when it permanently waits at its goal vertex. A vertex collision or, equivalently, *vertex conflict* occurs between the pairs of the agents a_i and a_j iff the two agents stay at the same vertex at the same timestep. An edge collision or, equivalently, *edge conflict* occurs iff the two agents traverse the same edge $(u, v) \in E$ in opposite directions at the same timestep. We call a pair of agent indices (i, j) a *conflicting pair* if there exists at least one conflict between paths p_i and p_j. A *solution* of a MAPF instance is a set of paths $\{p_1, \ldots, p_k\}$ in which there is no conflict between any pair of paths (i.e., no conflicting pair). We evaluate the solution quality via the *sum of (path) costs* (SOC), defined as $\sum_{i=1}^{k} c_i$. We say that a solution is *optimal* iff the SOC is minimum, and *suboptimal* otherwise. In this paper, we target suboptimal algorithms that have high success rates for a 1-minute runtime limit.

Prioritized Algorithms

One way of solving MAPF is to decouple it such that the algorithm finds one minimum-cost path for each agent individually and then resolves any conflicts between the paths. Another way of solving MAPF is to prevent conflicts between paths by using *priority constraints*, or *constraints* for short. A constraint $i \prec j$ expresses that agent a_i has a higher priority than agent a_j. In this case, we find path p_i *before* path p_j, and path p_j must avoid any conflict with path p_i. A *priority ordering* $\prec$ is a strict partial order on the set of agent indices $\{1, 2, \ldots, k\}$. A *total priority ordering* is a special case where the partial order is total, which is equivalent to assigning a unique priority to each agent. A path p_j *satisfies a priority ordering* iff it has no conflicts with all paths p_i that satisfy $i \prec j$ in the priority ordering.

Low-Level Single-Agent Path Finding To find a path that satisfies a priority ordering for an agent a_i from its start vertex s_i to its goal vertex g_i, Li et al. (2022) proposed the Safe Interval Path Planning with Soft Constraints (SIPPS) algorithm. Each SIPPS node n contains (1) a vertex $v(n)$, (2) a safe interval $[t_{low}, t_{high})$ indicating that the agent satisfies the priority ordering if it occupies vertex v from timestep t_{low} to $t_{high} - 1$, and (3) a number of conflicts $c(n)$ between the partial path from the start vertex s_i to $v(n)$ and the paths of the agents that do not have higher priorities than agent a_i. The f-value of SIPPS node n is the sum of its g-value and h-value, where the g-value is set to timestep t_{low} and the h-value is a lower bound on the minimum number of timesteps that it takes the agent to move from $v(n)$ to goal vertex g_i. To find a minimum-cost path, SIPPS always expands the SIPPS node with the minimum f-value in each iteration. Instead, SIPPS can also always expand the SIPPS node n with the minimum number of conflicts $c(n)$ in each iteration, which results in a suboptimal but minimum-conflicting path. For PP, $c(n)$ is always 0 since each path must avoid conflicts from the already-planned paths (of agents with higher priorities).

Prioritized Planning Based on the classical *prioritized planning scheme* (Erdmann and Lozano-Pérez 1986), Prioritized Planning (PP) for MAPF (Silver 2005) first chooses a total priority ordering and then iteratively finds a minimum-cost path for each agent according to that priority ordering. The minimum-cost path of agent $a_i$is not allowed to conflict with the already-planned paths of all agents a_j that have higher priorities than agent a_i (i.e, all agents a_j with constraint $j \prec i$). If, for the given total priority ordering, PP is unable to find a path for an agent, then it randomly selects a new total priority ordering and restarts from scratch (Bennewitz, Burgard, and Thrun 2001). Although PP is incomplete and finds suboptimal solutions, it finds solutions fast and with close-to-optimal SOCs in practice.

Priority-Based Search Priority-Based Search (PBS) (Ma et al. 2019) is a two-level algorithm inspired by CBS. On the high level, it performs a search on the *Priority Tree* (PT). Each PT node N contains (1) a priority ordering with a set of constraints $i \prec_N j$ for some or of all the agent pairs $a_i, a_j \in A$ and (2) a set of minimum-cost paths $paths(N) = \{p_1(N), \ldots, p_k(N)\}$ that satisfy the priority ordering, i.e., there is no conflict between any two minimum-cost paths $p_i(N)$ and $p_j(N)$ if either constraint $i \prec_N j$ or $j \prec_N i$ holds. Initially, the root PT node contains an empty priority ordering (i.e., no constraint between any two agents) and one minimum-cost path for each agent. When a PT node N is expanded, if no conflicting pair exists, then N is declared a goal node. Otherwise, PBS selects a conflicting pair (i, j) and splits N into two child PT nodes. The two child PT nodes extend the priority ordering with additional constraints $i \prec j$ and $j \prec i$, respectively. PBS then executes a low-level search to replan the individual minimum-cost paths of these two child PT nodes to satisfy the new priority ordering. When generating a child PT node N_1 of PT node N, PBS first uses a topological sort to obtain an ordering of the set of agent indices $\{1, \ldots, k\}$ that is consistent with the

priority ordering. Then, according to this ordering, PBS uses a low-level search to find an individual minimum-cost path for each agent one by one while satisfying the constraints. While there are many ways to search the PT, PBS performs a depth-first search by selecting the child PT node with the lowest SOC in the next iteration. If the low-level search cannot find a path for an agent that satisfies the priority ordering of the child PT nodes of PT node N, PBS deletes this child PT node. If PT node N is fully expanded but no child PT node is generated, then it is a *dead-end* and PBS *backtracks* to its parent PT node.

Greedy Priority-Based Search

PBS is not guaranteed to find optimal solutions due to the fixed priority ordering in each PT node (Ma et al. 2019). Still, PBS prefers PT nodes with smaller SOCs on the high level and SIPPS nodes with smaller f-values on the low level. Its search strategies are borrowed from algorithms that guarantee optimal solutions. However, since PBS does not guarantee optimality, expanding nodes according to costs is unnecessary and time-consuming.

Thus, we suggest to "greedily" minimize the number of conflicts on both levels during the search, resulting in our new approach: *Greedy Priority-Based Search (GPBS)*. On the high level, GPBS uses a depth-first search on the PT but expands the child PT nodes in an order of increasing numbers of conflicting pairs. This is different from PBS, which expands them in an order of increasing SOCs. On the low level, GPBS uses SIPPS, which always expands the SIPPS node with the minimum number of conflicts (and breaks ties in favor of the SIPPS nodes with the smallest f-value) during each iteration. Thus, given a PT node N, the low-level search will find a path for each agent with the minimum number of conflicts that satisfies the priority ordering. In short, GPBS uses the numbers of conflicts and conflicting pairs as heuristics to guide the search on its low and high levels, respectively.

GPBS Enhancements

In this section, we introduce several techniques to speed up GPBS further. Some of these techniques have been introduced before in different algorithmic frameworks but have yet to be incorporated into the PBS framework.

Partial Expansions (PE)

During each iteration, PBS and GPBS generate two child PT nodes. However, since they perform a depth-first search on the high level and only expand the second child PT node during backtracking, they may generate child PT nodes that will never be expanded (e.g., because a solution was found). Such generated-but-not-expanded PT nodes are called *surplus nodes* (Goldenberg et al. 2014). To limit the number of generated surplus nodes, we apply the *partial expansion* (PE) technique from EPEA* (Goldenberg et al. 2014) to GPBS, denoted as GPBS(PE). That is, when expanding a PT node N with the conflicting pair (i, j), we only generate one child PT node N_1 and continue the depth-first search from PT node N_1. The second child PT node will be generated only after backtracking back from N_1 to N. Since GPBS needs to replan the paths of one or more agents due to the newly added constraint while generating a child PT node, PE is particularly effective in GPBS.

Given a PT node N, for any agent $a_i \in A$, we define its *lower-priority agents* $A_i^L(N)$ as the set of agents that contains agent a_i and all agents with lower priorities than a_i. That is,

$$A_i^L(N) = \{a_i\} \cup \{a_l \mid i \prec_N l\}.$$

Suppose GPBS expands PT node N and generates its child PT node N_1 with the additional constraint $j \prec i$. To satisfy the constraints in the priority ordering, the maximum number of agents that need replanning is $|A_i^L(N)|$. Similarly, the maximum number of agents that need replanning while generating the other child PT node N_2 with the additional constraint $i \prec j$ is $|A_j^L(N)|$. If backtracking does not occur in PT node N (because a solution is found in the subtree of PT node N_1), then GPBS(PE) will not generate PT node N_2 and thus save the time for replanning at most $|A_j^L(N)|$ agents.

Target Reasoning (TR)

We adopt target reasoning (Li et al. 2020) to choose the constraint to add to the priority ordering of the child PT node to expand first. Li et al. (2020) defined *target conflict* as a special case of vertex conflict where one agent collides with another that already stays at its goal vertex. They also showed that resolving target conflicts earlier in CBS can speed up the search. This technique is called *target reasoning* (TR). Although GPBS(PE) is a suboptimal algorithm, we find out that selecting and resolving target conflicts earlier than other conflicts can also speed up its search. On the high level, we first select a conflicting pair that contains a target conflict while expanding a PT node. To resolve the target conflict, we add the constraint $i \prec j$, with agent a_j waiting at its goal vertex g_j permanently, to the priority ordering of the child PT node to expand first. The rationale is that agents waiting at their goal vertices permanently may have multiple target conflicts with other agents, and thus we resolve them at once by replanning their paths.

Induced Constraints (IC)

We propose induced constraints as a technique for choosing the conflicting pair for expansion. Similar to the definition of the lower-priority agents, given a PT node N with its priority ordering, we can also define the *higher-priority agents* of a_i, denoted as $A_i^H(N)$, as the set of agents that contains agent a_i itself and those agents having higher priorities than agent a_i. That is,

$$A_i^H(N) = \{a_i\} \cup \{a_h \mid h \prec_N i\}.$$

Suppose GPBS expands PT node N and generates one of its child PT node N' with the additional constraint $i \prec j$. In this case, agent a_i and all agents with priorities higher than agent a_i have priorities higher than agent a_j and all agents with priorities lower than agent a_j. Thus, GPBS implicitly introduces the constraints $h \prec l$ for any two agents $a_h \in A_i^H(N)$ and $a_l \in A_j^L(N)$. Of course, some of these

constraints might have already been introduced during a previous iteration and thus are not new. We call the set of new constraints introduced by constraint $i \prec j$ the *induced constraints* (IC), denoted as $IC_{i \prec j}(N)$. That is,

$$IC_{i \prec j}(N) = \{h \prec l \mid a_h \in A_i^H(N) \wedge a_l \in A_j^L(N) \wedge h \not\prec_N l\}.$$

Intuitively, the more induced constraints we add when generating a child PT node of N, the fewer conflicting pairs GPBS(PE) needs to resolve under its subtree. Also, the number of expanded SIPPS nodes may decrease since the IC reduce the search space. Thus, we add the constraint $i \prec j$, with the largest $|IC_{i \prec j}(N)|$ among all conflicting pairs (i, j) in PT node N, to the priority ordering of the child PT node to expand first. Meanwhile, we want to minimize the effort of replanning the paths of agents in order to speed up the search. Thus, if there is more than one constraint with the maximum $|IC_{i \prec j}(N)|$, then we break ties in favor of one with a smaller number of lower-priority agents $|A_j^L(N)|$.

Restart Techniques

The restart technique can be used to solve combinatorial problems faster (Ruan, Horvitz, and Kautz 2002). In MAPF, previous research shows that randomly choosing a total priority ordering for the agents and restarting the search from scratch can speed up PP (Bennewitz, Burgard, and Thrun 2001). We call this restart technique *random restarts* (RR). GPBS uses SIPPS, which minimizes the number of conflicts, and finds a path for each agent one by one in the root PT node. Thus, a different order of agents that GPBS finds paths in the root PT node may result in a different PT, which affects the conflicting pairs in the root PT node and thus also the runtime of GPBS. Thus, an intuitive way of applying RR to GPBS is to restart the search when PT node N is a dead-end and the number of backtracks exceeds a user-specified threshold. However, RR has two drawbacks: (1) we must replan the paths of all agents, which increases the runtime on the low level, and (2) the resulting paths may have more conflicting pairs, which increases the runtime on the high level. Thus, we propose another restart technique, called *soft restarts (SR)*. If a PT node N is a dead-end and the number of backtracks exceeds a user-specified threshold, SR first removes all the constraints in the priority ordering of PT node N but keeps its set of paths $paths(N)$. Then, it continues the search. That is, PT node N becomes the new root PT node. Empirically, using SR without backtracking (i.e., setting the user-specified threshold to 0) results in better performance.

Implementation

Algorithm 1 shows the pseudo-code of our implementation of GPBS with all improvements, namely PE [Lines 8 and 32], TR [Lines 11 to 13], IC [Lines 14 to 23], and SR [Lines 43 and 46]. We use a stack, denoted as STACK[1], to store the PT nodes. For each PT node N, we use a priority queue (denoted as $conflicts(N)$) to store the conflicting pairs in $path(N)$ of PT node N. We prioritize conflicting pairs that have target conflicts if TR is used (breaking ties randomly).

Algorithm 1: GPBS with PE, TR,IC, and SR

```
   Input: a MAPF instance, runtime limit T
 1 Root ← new PT node, ≺_Root ← φ
 2 paths(Root) ← Find a path for each agent
 3 conflicts(Root) ← Find conflicting pairs
 4 STACK ← {Root}
 5 while STACK ≠ φ and runtime ≤ T do
 6   N ← top PT node in STACK
 7   if conflicts(N) = φ then return paths(N)
 8   if N has not yet been partially expanded then
 9     N_1 ← new child PT node of N, ≺_N1 ← ≺_N
10     (i, j) ← conflicts(N).top
11     if p_i(N) and p_j(N) has a target conflict and
         TR is used then
12       if c_j(N) < c_i(N) then Add i ≺ j to ≺_N1
13       else Add j ≺ i to ≺_N1
14     else if IC is used then
15       A^H, A^L ← zero vectors of size k
16       IC ← k × k zero matrix
17       forall (i, j) ∈ conflicts(N) do
18         A^L[i], A^L[j] ← |A_i^L(N)|, |A_j^L(N)|
19         A^H[i], A^H[j] ← |A_i^H(N)|, |A_j^H(N)|
20         IC[i][j] ← |IC_{i≺j}(N)|
21         IC[j][i] ← |IC_{j≺i}(N)|
22       (i, j) ← argmax_{(i,j)∈conflicts(N)} IC[i][j],
           breaking ties in favor of a smaller A^L[j]
23       Add i ≺ j to ≺_N1
24     else
25       Add either i ≺ j or j ≺ i to ≺_N1 randomly
26     paths(N_1) ← Find paths that satisfy ≺_N1
27     if paths(N_1) are found then
28       conflicts(N_1) ← Find conflicting pairs
29       Add N_1 to the top of STACK
30     else Delete PT node N_1
31     Set N to be partially expanded
32   else
33     Remove N from STACK
34     N_2 ← new child PT node of N, ≺_N2 ← ≺_N
35     i ≺ j ← retrieve constraint for generating the
         first child PT node
36     Add j ≺ i to ≺_N2
37     paths(N_2) ← Find paths that satisfy ≺_N2
38     if paths(N_2) are found then
39       conflicts(N_2) ← Find conflicting pairs
40       Add N_2 to the top of STACK
41     else Delete PT node N_2
42     Set N to be fully expanded
43   if N has been fully expanded and N has no child
       PT node and SR is used then
44     Remove all constraints from ≺_N
45     Remove all PT nodes from STACK
46     Add PT node N to STACK
47 return "No solution"
```

[1]If SR is used, then we can get rid of STACK since there is no backtracking. However, we left it in to show how to implement Algorithm 1 without SR, namely by removing Lines 43 to 46.

If TR and IC are used together, we prioritize conflicting pairs that have target conflicts, and only compute the numbers of IC if the top conflicting pair does not have a target conflict (i.e., all the conflicting pairs in *conflicts*(N) do not have target conflicts).

To start the search, we generate the root PT node *Root* with an empty priority ordering. We find one path for each agent and store them in *paths*$(Root)$. We also find all the conflicting pairs and store them in *conflicts*$(Root)$. Then, we add *Root* to STACK [Lines 1 to 4]. While STACK is not empty in each iteration and the runtime has not exceeded the user-specified runtime limit T, we expand the PT node N that is on the top of STACK. If there is no conflicting pair in *conflicts*(N), then we return the set of paths *paths*(N) [Line 7]. Otherwise, if PT node N has not yet been partially expanded, we generate a child PT node N_1 and initialize its set of paths and priority ordering with those of its parent PT node N. If the first conflicting pair in *conflicts*(N) contains a target conflict and TR is used, we add the constraint $i \prec j$ (with respect to $j \prec i$) to the priority ordering $\prec_{N_1}$ of child PT node N_1 if path costs $c_j(N)$ (with respect to $c_i(N)$) is smaller than $c_i(N)$ (with respect to $c_j(N)$), meaning that agent a_i (with respect to a_j) collides with agent a_j (with respect to a_i) at goal vertex g_j (with respect to g_i) [Lines 11 to 13]. Otherwise, if IC is used, for each conflicting pair (i, j) in *conflicts*(N), we compute its numbers of higher-priority agents, lower-priority agents, and IC for constraint $i \prec j$ and for constraint $j \prec i$. We then add the constraint $i \prec j$ with the maximum value (and break ties in favor of a smaller number of lower-priority agents) to priority ordering $\prec_{N_1}$ of child PT node N_1 [Lines 14 to 23]. After adding a constraint $i \prec j$ to the priority ordering $\prec_{N_1}$ of child PT node N_1, we update the set of paths *paths*(N_1) so that it satisfies the priority ordering $\prec_{N_1}$. We first find the lower-priority agents of agent a_j, and then use SIPPS that minimizes the number of conflicts to find a path for each agent in the lower-priority agents. We then add N_1 to STACK iff this could be done successfully and delete it otherwise.

If PT node N has already been partially expanded (meaning that its first child PT node has already been generated), then we remove it from STACK. We then generate the second child PT node N_2 and initialize its set of paths and priority ordering with those of its parent PT node N. We retrieve the constraint $i \prec j$ was added to the priority ordering of the first child PT node of N, add constraint $j \prec i$ to the priority ordering $\prec_{N_2}$ of PT node N_2, and update the set of paths *paths*(N_2) so that it satisfies the priority ordering $\prec_{N_2}$ [Lines 33 to 41]. If SR is used and if PT node N has been fully expanded but no child PT node was generated, then we perform SR by first removing all constraints from priority ordering $\prec_N$ and all PT nodes from STACK, and then adding PT node N back to STACK [Lines 43 to 46]. Lastly, since GPBS is an incomplete algorithm, even if there is a solution to the MAPF instance, GPBS may fail to find it, which results in returning "No solution" [Line 47]. However, GPBS with SR will keep cycling in the while loop until its runtime exceeds the user-specified runtime limit T.

Empirical Evaluation

We run an experimental evaluation in order to (1) support our design choices for TR and IC within GBPS, (2) evaluate the impact of different combinations of GBPS improvements, and (3) compare the success rates and SOCs of GPBS with all the improvements with PP, PBS, state-of-the-art bounded-suboptimal algorithm, namely Explicit Estimation Conflict-Based Search (EECBS) (Li, Ruml, and Koenig 2021), and state-of-the-art suboptimal algorithms, namely PBS with the merging technique (PBS w/m) (Boyarski et al. 2022) and Large Neighborhood Search (MAPF-LNS2) (Li et al. 2022).

Although the PP and PBS algorithms use an A* search on their low levels in the previous research (Silver 2005; Ma et al. 2019), we use SIPPS on their low levels to minimize the path cost, resulting in PP_p and PBS_p, respectively (the subscript "p" stands for "path costs"). Also, rather than the path cost, we create a PBS variant that uses a SIPPS version on it low level to minimize the number of conflicts, resulting in PBS_c (the subscript "c" stands for "conflicts"). Although there are PP variants that incorporate machine learning approaches (Zhang et al. 2022), the time required for building the dataset and training the model is more than 1 minute. Thus, such algorithms are not included in our experiments. To evaluate the impact of our improvements, we perform a study that compares GPBS, GPBS with PE (GPBS(PE)), GPBS with PE and TR (GPBS(PE,TR)), GPBS with PE, TR, and IC (GPBS(PE,TR,IC)), GPBS with PE, TR, IC, and RR with user-specified threshold 200 (since it performs the best among the thresholds 0, 20, 50, 100, 200, and 400) (GPBS(PE,TR,IC,RR)), and GPBS with PE, TR, IC, and SR with user-specified threshold 0 (since it performs the best among the thresholds 0, 10, 20, and 50) (GPBS(PE,TR,IC,SR)).

Experimental Setup

We evaluate all algorithms on MAPF instances with 4-neighbor grid maps from the MAPF benchmark suite (Stern et al. 2019). We use three small maps of size 32×32 with dense obstacles, denoted as "Dense", namely maps `random-32-32-20` (denoted as "Random"), `maze-32-32-2` (denoted as "Maze"), and `room-32-32-4` (denoted as "Room"). The number of agents on the Random map ranges from 200 to 400 in increments of 50, and the number of agents on the other maps ranges from 100 to 300 in increments of 50. We also use three large maps, namely maps `warehouse-10-20-10-2-1` of size 161×63 (denoted as "Warehouse(M)"), `warehouse-20-40-10-2-1` of size 321×123 (denoted as "Warehouse(L)"), and `den520d` of size 530×481 (denoted as "Game"). The number of agents on the Warehouse(M) map ranges from 200 to 1,000 in increments of 200, and the number of agents on the other maps ranges from 1,000 to 2,000 in increments of 200. Figure 4 shows the maps. We use the `random` scenario from the benchmark suite, which yields 25 MAPF instances for each map and each number of agents. Since the benchmark suite provides only MAPF instances with at most 1,000 agents, we create 25 MAPF instances, each with 1,200

	Random	Maze	Room	*Dense*	Warehouse(M)	Warehouse(L)	Game	*Total*
PP_p	0.20	0.01	0.20	0.14	0.38	0.08	0.97	0.31
PBS_p	0.10	0.03	0.13	0.09	0.26	0.00	0.00	0.08
PBS_c	0.46	0.08	0.27	0.27	0.54	0.23	0.80	0.41
PBS w/m	0.34	0.10	0.29	0.24	0.58	0.31	0.89	0.43
EECBS	0.74	0.39	0.70	0.61	0.60	0.34	0.48	0.54
MAPF-LNS2	0.94	0.47	0.78	0.73	0.83	0.38	**1.00**	0.73
GPBS	0.80	0.59	0.77	0.72	0.80	0.29	0.81	0.67
GPBS(PE)	0.86	0.61	0.92	0.80	0.82	0.39	**1.00**	0.76
GPBS(PE,ITR_1)	0.69	0.33	0.58	0.53	0.66	0.40	**1.00**	0.62
GPBS(PE,ITR_2)	0.86	0.66	0.94	0.82	0.86	0.40	**1.00**	0.78
GPBS(PE,TR)	0.91	0.70	0.94	0.85	0.82	0.39	**1.00**	0.79
GPBS(PE,TR,IIC)	0.86	0.67	0.87	0.80	0.80	0.41	0.99	0.76
GPBS(PE,TR,IC)	0.93	0.72	0.93	0.86	0.92	0.41	0.99	0.81
GPBS(PE,TR,IC,RR)	0.97	0.73	0.94	0.88	0.86	**0.45**	**1.00**	0.82
GPBS(PE,TR,IC,SR)	**1.00**	**0.74**	**1.00**	**0.91**	**0.93**	0.42	**1.00**	**0.84**

Table 1: Success rates over all MAPF instances with the same map. Column *Dense* contains the success rates over all MAPF instances with the Random, Maze, and Room maps. Column *Total* contains the success rates over all MAPF instances. The numbers in bold are the highest success rates in all columns.

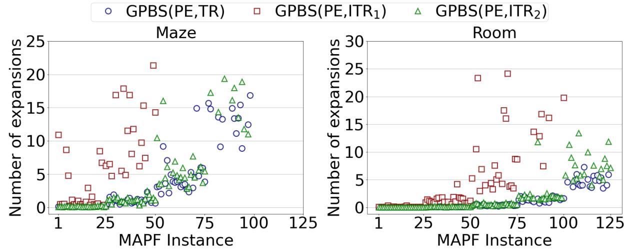

Figure 1: Numbers of SIPPS node expansions (in millions) of GPBS(PE,TR), GPBS(PE,ITR_1), and GPBS(PE,ITR_2) over all MAPF instances that are successfully solved by respective algorithms in Maze and Room maps.

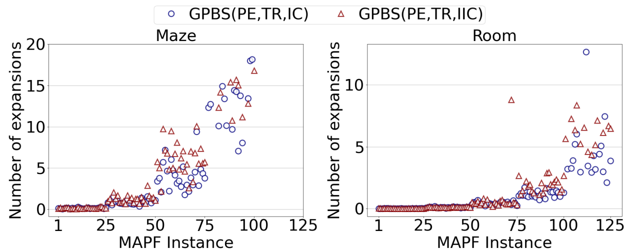

Figure 2: Numbers of SIPPS node expansions (in millions) of GPBS(PE,TR,IC) and GPBS(PE,TR,IIC) over all MAPF instances that are successfully solved by respective algorithms in Maze and Room maps.

to 2,000 agents for the large maps. The start and goal vertices of all agents are chosen randomly. We implement all algorithms in C++ (compiled with `GCC-11.3.0`) and conduct experiments on `CentOS Linux` and an `AMD EPYC 7302 16-core` processor with 16 GBs of memory. All of our experiments use a 1-minute runtime limit.

Evaluation of TR and IC

TR (1) adds the constraint $i \prec j$, with agent a_j waiting at its goal vertex g_j permanently, to the priority ordering of the child PT node to expand first, and (2) resolves target conflicts earlier than other conflicts. To evaluate TR, we create two variants, namely *Inverse TR 1* (ITR_1) and *Inverse TR 2* (ITR_2). ITR_1 adds constraint $j \prec i$ to the priority ordering of the child PT node to expand first, and ITR_2 resolves target conflicts later than other conflicts. IC adds the constraint $i \prec j$ with the largest $|IC_{i\prec j}(N)|$ to the priority ordering of the child PT node of PT node N to expand first. To evaluate IC, we create a variant called *Inverse IC* (IIC). This variant adds a constraint with the minimum number of IC in each iteration during PE.

As shown in Table 1, GPBS(PE,TR) has higher success rates than GPBS(PE,ITR_1) and GPBS(PE,ITR_2) in dense MAPF instances. However, the success rates of all these algorithms are about the same on MAPF instances with large maps since they have more free space. Also, GPBS(PE,TR,IC) has higher success rates than GPBS(PE,TR), while GPBS(PE,TR,IIC) has lower success rates. Figures 1 and 2 show the numbers of SIPPS node expansions on the low level for each MAPF instance that an algorithm can successfully solve. The MAPF instances are sorted in increasing order of their number of agents. GPBS(PE,TR) and GPBS(PE,TR,IC) have smaller numbers of SIPPS node expansions than their inverse versions, resulting in higher success rates.

Performance Comparison

Table 1 shows the success rates of the algorithms among all MAPF instances with the same map. The total success rate increase from 0.67 to 0.76 when we add PE to GPBS, further to 0.79 if we add TR, even further to 0.81 if we add IC, and reaches the best to 0.84 when we add SR. We compare GPBS with the prioritized algorithms,

	Random		Maze		Room		Warehouse(M)		Warehouse(L)		Game	
	#call	#exp	#call	#exp	#call	#exp	#call	#exp	#call	#exp	#call	#exp
PP_p	285.2	22.4	217.7	29.8	310.1	26.5	24.3	22.4	5.7	27.3	1.8	4.2
PBS_c	13.7	13.4	12.7	27.0	21.6	20.4	1.3	10.9	**1.5**	26.5	**1.5**	6.1
PBS w/m	18.2	13.5	9.6	21.6	20.9	18.6	1.1	9.4	**1.5**	27.5	**1.5**	5.9
EECBS	3.2	5.5	7.0	15.7	3.5	7.1	3.6	8.9	9.8	**19.9**	9.8	7.7
MAPF-LNS2	3.7	2.7	3.3	10.3	6.6	5.4	1.0	7.5	1.7	31.5	**1.5**	**3.8**
GPBS(PE,TR,IC,SR)	**0.7**	**1.1**	**1.1**	**7.5**	**0.6**	**1.3**	**0.7**	**6.3**	**1.5**	27.0	**1.5**	4.2

Table 2: Average numbers of calls to SIPPS (in thousands) and average numbers of SIPPS node expansions (in millions). The numbers in bold are the lowest numbers in each column.

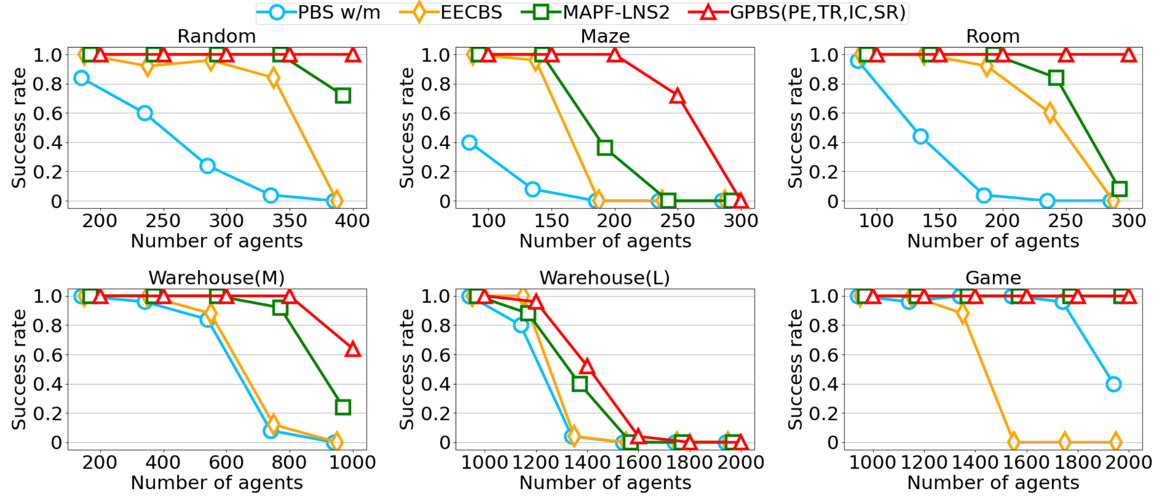

Figure 3: Success rates of PBS w/m, EECBS, MAPF-LNS2, and GPBS(PE,TR,IC,SR) over all MAPF instances with the same map and number of agents.

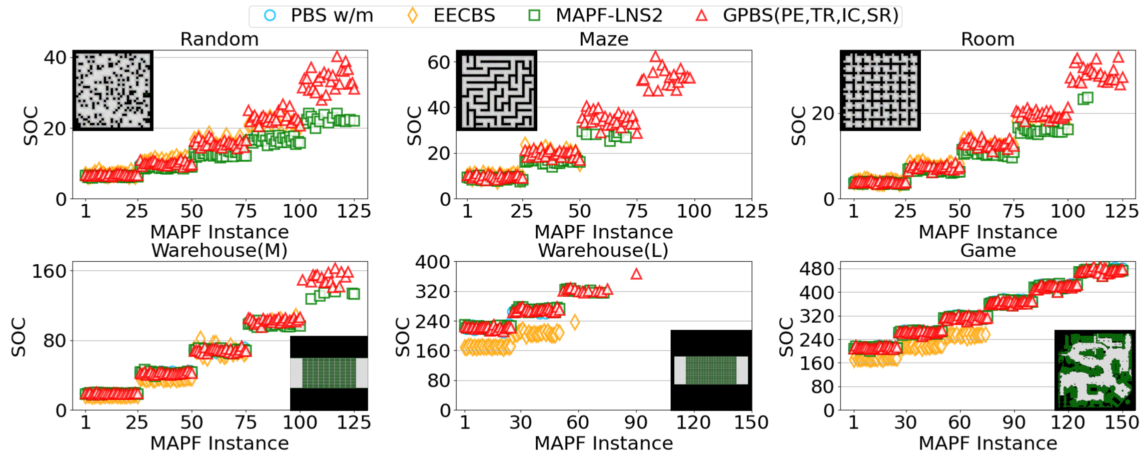

Figure 4: SOCs (in thousands) of PBS w/m, EECBS, MAPF-LNS2, and GPBS(PE,TR,IC,SR) over all MAPF instances that each algorithm can successfully solve, sorted in increasing numbers of their agents.

namely PP and PBS, all of them using SIPPS for the low-level search. We compare PP_p, PBS_p, and PBS_c. Table 1 shows that GPBS has a higher total success rate than all of the prioritized algorithms and GPBS(PE,TR,IC,SR) has an even higher success rate, especially for dense MAPF instances. Table 1 also shows the success rates of the state-of-the-art prioritized algorithm (PBS w/m), the state-of-the-art bounded-suboptimal algorithm with large user-specified suboptimality bound (EECBS), and the state-of-the-art sub-optimal algorithm (MAPF-LNS). We choose 5 as the user-specified suboptimality bound for EECBS (i.e., the SOC of the solution found by EECBS is at most 5 times larger

than minimal), which is large enough to enable it to scale to large MAPF instances (Li et al. 2022). We choose 8 as the neighborhood size of MAPF-LNS2, which is the default value in Li et al. (2022). GPBS(PE,TR,IC,SR) has higher success rates than PBS w/m, EECBS, and MAPF-LNS2 with improvements of 41%, 30%, and 10% over all MAPF instances, respectively. For dense MAPF instances, these improvements become 67%, 30%, and 19%, respectively. Table 2 shows the numbers of calls to SIPPS and the numbers of SIPPS node expansions of each algorithm, averaged over all MAPF instances with the same map. GPBS(PE,TR,IC,SR) replans the paths of fewer agents than the other algorithms, which results in fewer calls to SIPPS and fewer SIPPS node expansions and thus speeds up the search.

Figure 3 shows the success rates over all MAPF instances with the same map and number of agents. GPBS(PE,TR,IC,SR) has higher success rates than the other algorithms, especially for MAPF instances with dense obstacles and large numbers of agents. Figure 4 shows the SOCs over all MAPF instances with the same map among those MAPF instances that each algorithm can successfully solve. GPBS(PE,TR,IC,SR) finds solutions with higher SOCs than the other algorithms on dense MAPF instances. It has similar SOCs as the ones of EECBS (e.g., MAPF instances 1 to 100 on the Random map, 1 to 50 on the Maze map, and 1 to 100 on the Room map), which indicates that the SOCs are at most 5 times larger than minimal. GPBS(PE,TR,IC,SR) has similar SOCs as MAPF-LNS2 on large maps. These maps have more free space than the small maps, resulting in fewer conflicts that need to be resolved, and thus both algorithms find similar solutions. To compare RR and SR, Figure 5 shows an example where SR solves a MAPF instance successfully while RR with a user-specified threshold of 0 fails. When a PT node is a dead-end and both algorithms restart the search, their numbers of conflicting pairs may increase, as shown in the blue peaks (in around iteration 200, 600, and 1,000) and the red curve (in around iteration 400) on the top of Figure 5. However, RR requires finding a path for each agent from scratch, indicating a higher number of calls of SIPPS than SR, as shown on the bottom of Figure 5.

Relation between GPBS and MAPF-LNS2

In each iteration, both GPBS(PE,SR) and MAPF-LNS2 maintain one path for each agent, select a subset of agents, and replan their paths while avoiding the current paths of the other agents. In GPBS(PE,SR), this set of agents includes one of the agents corresponding to a conflict and its lower-priority agents, while in MAPF-LNS2, this set of agents is the "neighborhood," which is selected with a neighborhood selection strategy. Thus, GPBS with our improvements can be viewed as a special case of MAPF-LNS2 with an intelligent neighborhood selection strategy and an adaptive neighborhood size. Figure 5 shows the numbers of conflicting pairs and the cumulative numbers of calls to SIPPS when solving a MAPF instance with MAPF-LNS2 and GPBS(PE,TR,IC,SR), respectively. In each iteration, since MAPF-LNS2 selects a fixed number of agents and uses PP_c for replanning, it needs to replan the paths of all agents in

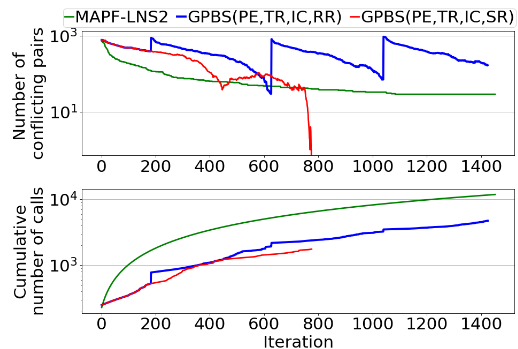

Figure 5: Numbers of conflicting pairs of agents and the cumulative numbers of calls to SIPPS. We use MAPF-LNS2, GPBS(PE,TR,IC,RR), and GPBS(PE,TR,IC,SR) to solve a MAPF instance with 250 agents on the Maze map. Both MAPF-LNS2 and GPBS(PE,TR,IC,RR) reached the runtime limit.

the neighborhood (Li et al. 2022). Such behavior reduces the number of conflicting pairs at the beginning of the search. On the other hand, since GPBS(PE,TR,IC,SR) selects a conflicting pair and assigns constraints accordingly, although it resolves fewer conflicting pairs than MAPF-LNS2 at the beginning of the search, the cumulative number of calls can be smaller than that of MAPF-LNS2. Table 2 shows that GPBS(PE,TR,IC,SR) has fewer numbers of calls to SIPPS than MAPF-LNS2 averaged over all MAPF instances on the same maps.

Conclusion

We proposed Greedy PBS (GPBS) that aims to minimize the number of collisions among agents as opposed to minimizing the sum of the travel times of all agents during the search. Based on GPBS, we proposed a set of techniques to speed up the search further. We adopted partial expansions and target reasoning from other algorithms and proposed techniques for selecting which constraints to impose during the search and for restarting. Our empirical evaluation showed that GPBS, with all our techniques, has higher success rates than several state-of-the-art algorithms. Future work includes developing a more sophisticated restart strategy, adopting incremental search on the low level of GPBS, and extending our techniques to other algorithms.

Acknowledgments

The research at the University of Southern California was supported by the National Science Foundation (NSF) under grant numbers 1409987, 1724392, 1817189, 1837779, 1935712, 2121028, and 2112533, the United States-Israel Binational Science Foundation (BSF) under grant numbers 2019703 and 2021643, as well as a gift from Amazon Robotics. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies, or the U.S. government.

References

Barer, M.; Sharon, G.; Stern, R.; and Felner, A. 2014. Suboptimal Variants of the Conflict-Based Search Algorithm for the Multi-Agent Pathfinding Problem. In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, 19–27.

Bennewitz, M.; Burgard, W.; and Thrun, S. 2001. Optimizing Schedules for Prioritized Path Planning of Multi-Robot Systems. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 271–276.

Berg, J. P. v. d.; and Overmars, M. H. 2005. Prioritized Motion Planning for Multiple Robots. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 430–435.

Boyarski, E.; Chan, S.-H.; Atzmon, D.; Felner, A.; and Koenig, S. 2022. On Merging Agents in Multi-Agent Pathfinding Algorithms. In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, 11–19.

Boyarski, E.; Felner, A.; Stern, R.; Sharon, G.; Tolpin, D.; Betzalel, O.; and Shimony, S. E. 2015. ICBS: Improved Conflict-Based Search Algorithm for Multi-Agent Pathfinding. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 740–746.

Erdmann, M.; and Lozano-Pérez, T. 1986. On Multiple Moving Objects. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 1419–1424.

Goldenberg, M.; Felner, A.; Stern, R.; Sharon, G.; Sturtevant, N. R.; Holte, R. C.; and Schaeffer, J. 2014. Enhanced Partial Expansion A*. *Journal of Artificial Intelligence Research (AIJ)*, 50(1): 141–187.

Ho, F.; Salta, A.; Geraldes, R.; Goncalves, A.; Cavazza, M.; and Prendinger, H. 2019. Multi-Agent Path Finding for UAV Traffic Management. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, 131–139.

Li, J.; Chen, Z.; Harabor, D.; Stuckey, P. J.; and Koenig, S. 2022. MAPF-LNS2: Fast Repairing for Multi-Agent Path Finding via Large Neighborhood Search. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 10256–10265.

Li, J.; Gange, G.; Harabor, D.; Stuckey, P. J.; Ma, H.; and Koenig, S. 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.

Li, J.; Harabor, D.; Stuckey, P. J.; Ma, H.; and Koenig, S. 2019. Symmetry-Breaking Constraints for Grid-Based Multi-Agent Path Finding. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 6087–6095.

Li, J.; Hoang, T. A.; Lin, E.; Vu, H. L.; and Koenig, S. 2023. Intersection Coordination with Priority-Based Search for Autonomous Vehicles. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*.

Li, J.; Ruml, W.; and Koenig, S. 2021. EECBS: Bounded-Suboptimal Search for Multi-Agent Path Finding. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 12353–12362.

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 (AAAI)*, 7643–7650.

Ruan, Y.; Horvitz, E.; and Kautz, H. A. 2002. Restart Policies with Dependence among Runs: A Dynamic Programming Approach. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, 573–586.

Silver, D. 2005. Cooperative Pathfinding. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (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.; Barták, R.; and Boyarski, E. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, 151–159.

Walker, T. T.; Sturtevant, N. R.; and Felner, A. 2020. Generalized and Sub-Optimal Bipartite Constraints for Conflict-Based Search. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 7277–7284.

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

Yu, J.; and LaValle, S. M. 2013. Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 1443–1449.

Zhang, S.; Li, J.; Huang, T.; Koenig, S.; and Dilkina, B. 2022. Learning a Priority Ordering for Prioritized Planning in Multi-Agent Path Finding. In *Proceedings of the Symposium on Combinatorial Search (SoCS)*, 208–216.