MAPF-LNS2: Fast Repairing for Multi-Agent Path Finding via Large Neighborhood Search

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@usc.edu, {zhe.chen, daniel.harabor, peter.stuckey}@monash.edu, skoenig@usc.edu

Abstract

Multi-Agent Path Finding (MAPF) is the problem of planning collision-free paths for multiple agents in a shared environment. In this paper, we propose a novel algorithm MAPF-LNS2 based on large neighborhood search for solving MAPF efficiently. Starting from a set of paths that contains collisions, MAPF-LNS2 repeatedly selects a subset of colliding agents and replans their paths to reduce the number of collisions until the paths become collision-free. We compare MAPF-LNS2 against a variety of state-of-the-art MAPF algorithms, including Prioritized Planning with random restarts, EECBS, and PPS, and show that MAPF-LNS2 runs significantly faster than them while still providing near-optimal solutions in most cases. With a time limit of just 5 minutes, MAPF-LNS2 solves 80% of the random-scenario instances on all 33 maps with the largest numbers of agents from the MAPF benchmark, which clearly defines the new state-of-the-art suboptimal MAPF algorithm.

1 Introduction

MAPF (Stern et al. 2019) is the problem of planning collision-free paths for multiple agents in a shared environment while minimizing their travel times and NP-hard to solve optimally. It is the core problem of many applications, such as warehouse automation, traffic management, and robotics. Existing MAPF algorithms include *systematic search algorithms* (that are exponential-time but guaranteed to find optimal or bounded-suboptimal solutions), *rule-based algorithms* (that are usually polynomial-time and complete), and *prioritized algorithms* (that run fast empirically but are neither complete nor optimal). When facing challenging MAPF instances, however, the first two types of algorithms suffer from either memory-outs or time-outs while the last type suffers from incompleteness. One successful technique that can improve the chance of finding solutions is to restart the search with a new random seed (Bennewitz, Burgard, and Thrun 2001; Cohen et al. 2018).

In this work, we propose a different way to improve the chance of finding solutions. Instead of giving up the previous search effort and restarting from scratch, we make use of the infeasible set of paths produced by a MAPF algorithm and try to repair it via Large Neighborhood Search (Shaw 1998). We call the new algorithm MAPF-LNS2. MAPF-LNS2 starts from a set of paths that have collisions and repeatedly replans subsets of paths to reduce the overall number of collisions until the paths become collision-free. By using Prioritized Planning (PP) (Silver 2005) with an efficient single-agent pathfinding algorithm to plan and replan paths and a variety of heuristics to select subsets of paths, MAPF-LNS2 can solve easy instances as fast as PP and hard instances significantly faster than PP and other MAPF algorithms. Even when MAPF-LNS2 fails to find collision-free paths within the time limit, it usually returns paths with only a few collisions, which is acceptable in many applications (Belov et al. 2020). The main contributions of this work are twofold:

1. We propose an efficient single-agent pathfinding algorithm SIPPS based on SIPP (Phillips and Likhachev 2011) for finding a short path that avoids collisions with a given set of paths and minimizes the number of collisions with another given set of paths. We show that SIPPS runs 5 times (or more) faster than *Space-Time A**, an A*-based algorithm widely used by many MAPF algorithms. Thus, we demonstrate that SIPPS can speed up not only MAPF-LNS2 but also many other MAPF algorithms, such as EECBS (Li, Ruml, and Koenig 2021).
2. We propose a suboptimal MAPF algorithm MAPF-LNS2 that is fast, scalable, and memory-efficient. Although it lacks theoretical guarantees, it empirically significantly outperforms a variety of state-of-the-art MAPF algorithms, including Prioritized Planning with random restarts, EECBS, and PPS (Sajid, Luna, and Bekris 2012), in terms of both success rates and runtimes. MAPF-LNS2 solves 80% of the random-scenario instances with the largest number of agents from the MAPF benchmark suite with a time limit of just 5 minutes, which, to our knowledge, has not been achieved by any existing algorithms. Moreover, when given a longer time limit of one hour, MAPF-LNS2 can scale to 8,000 agents on a congested warehouse map.

2 Background

Definition 1 (Multi-Agent Path Finding)**.** We are given a connected graph $G = (V, E)$, a set of m agents $A = \{a_1, \cdots, a_m\}$, and a start vertex $s_i \in V$ and a target vertex

$g_i \in V$ for each agent $a_i \in A$. At each discrete timestep, an agent either moves to an adjacent vertex or waits at its current vertex. A *collision* happens when two agents occupy the same vertex or traverse the same edge in opposite directions at the same timestep. A *plan* is a set of paths $\{p_1, \cdots, p_m\}$ that move the agents from their start vertices to their target vertices. Each agent remains at its target vertex after it completes its path. A plan is *feasible* if it contains no collisions and *infeasible* otherwise. Our task is to find a feasible plan (also called a *solution*) with a small *sum of costs* $\sum_{i=1}^{m} |p_i|$, i.e., the sum of the travel times of the agents. $\square$

Due to the wide applications of MAPF, numerous MAPF algorithms have been proposed in recent years. State-of-the-art optimal and bounded-suboptimal algorithms, such as Lazy CBS (Gange, Harabor, and Stuckey 2019), BCP (Lam and Le Bodic 2020), and EECBS (Li, Ruml, and Koenig 2021), usually deploy a strategy called CBS (Sharon et al. 2015) that uses single-agent pathfinding algorithms to plan paths for each agent first and resolves collisions afterwards. They find solutions with quality guarantee but do not scale to large problems as their runtimes are exponential in the number of agents. State-of-the-art unbounded-suboptimal algorithms include prioritized algorithms, such as prioritized planning (Erdmann and Lozano-Perez 1987) and PBS (Ma et al. 2019), and rule-based algorithms, such as PPS (Sajid, Luna, and Bekris 2012), PIBT (Okumura et al. 2019), and WSCaS (Wang and Rubenstein 2020). Prioritized algorithms plan paths based on a priority ordering of the agents where lower-priority agents need to avoid collisions with higher-priority agents. They are simple and run extremely fast but can fail to find any solutions for challenging problems due to their incompleteness. Rule-based algorithms move agents toward their target vertices via simple movement rules. Many of them are polynomial-time and complete in theory, but they can still fail to find solutions within a reasonable time for large problems.

The idea of MAPF-LNS2 is that, when a MAPF algorithm fails, we obtain an infeasible plan from the algorithm and repair it. For instance, for a CBS-style algorithm, each high-level search node contains a plan, so we pick the plan with the minimum number of collisions. For a prioritized algorithm, it fails when there is no path for an agent that avoids collisions with the paths of higher-priority agents. We obtain the already-planned paths and plan paths for the remaining agents that minimize the number of collisions (instead of avoiding collisions) with the already-planned paths.

3 MAPF-LNS2

Large Neighborhood Search (LNS) (Shaw 1998) is a popular local search technique to improve solution quality. Starting from a given solution, it *destroys* part of the solution, called a *neighborhood*, and treats the remaining part of the solution as fixed. It then *repairs* the solution and replaces the old solution if the repaired solution is better. This procedure is repeated until some stop criteria are meet. MAPF-LNS (Li et al. 2021a) is an anytime MAPF framework that uses LNS to improve the quality of a solution obtained from a MAPF algorithm over time. It repeatedly selects a subset of agents and replans their paths. Motivated by this work, we propose MAPF-LNS2 that can efficiently find a solution (instead of improving a given solution) for a MAPF instance.

To begin with, MAPF-LNS2 calls a MAPF algorithm to solve the problem and obtains a (partial or complete) plan from the MAPF algorithm. For each agent that does not yet have a path, MAPF-LNS2 plans a path for it that minimizes the number of collisions with the existing paths. Details of finding such paths are introduced in Section 4. MAPF-LNS2 then repeats a repairing procedure until the plan P becomes feasible. At each iteration, MAPF-LNS2 selects a subset of agents $A_s \subseteq A$ by a neighborhood selection method (see Section 5) and removes their paths from P. Let us denote the removed paths as P^-. It then calls a modified MAPF algorithm to replan the paths of the agents in A_s that minimize the number of collisions with each other and with the paths in P. Specifically, MAPF-LNS2 uses a modified Prioritized Planning (PP) as the modified MAPF algorithm.[1] PP assigns a random priority ordering to the agents in A_s and replans their paths one at a time according to the ordering. Each time, it calls a single-agent pathfinding algorithm (see Section 4) to find a path for an agent that minimizes the number of collisions with the new paths of the higher-priority agents in A_s and with the paths in P (which are the paths of the agents not in A_s). Let us denote the new paths of the agents in A_s as P^+. Finally, MAPF-LNS2 replaces the old plan $P \cup P^-$ with the new plan $P \cup P^+$ iff the *number of colliding pairs* (CP) of the paths in the new plan is no larger than that of the old plan.

4 Pathfinding with Dynamic Obstacles

To make MAPF-LNS2 efficient, we need an efficient single-agent pathfinding algorithm that can finds a shortest path for an agent that minimizes the number of collisions with a given set of paths. Here, we formulate a more general problem called Pathfinding with Mixed Dynamic Obstacles (PMDO). We use half-open interval notation $[a, b)$ to represent the contiguous set of integers $\{x \mid a \leq x \wedge x < b\}$.

Definition 2 (Pathfinding with Mixed Dynamic Obstacles)**.** We call (v, t), (e, t), and $(v, [t, \infty))$ a vertex, edge, and target obstacle indicating that vertex $v \in V$, edge $e \in E$, and vertex $v \in V$ are occupied at timestep t, from timestep $t-1$ to timestep t, and at and after timestep t, respectively. Given a graph $G = (V, E)$, a start vertex $s \in V$, a target vertex $g \in V$, and two finite sets of obstacles $\mathcal{O}^h$ (called *hard obstacles*) and $\mathcal{O}^s$ (called *soft obstacles*), our task is to find a path p from s to g that does not collide with any hard obstacles. We assume that s at timestep 0 is not occupied by any hard obstacles, and g at timestep ∞ is not occupied by any hard obstacles either, i.e., there is finite time from which no more hard obstacles occupy g. The objective is to minimize the number of *soft collisions*, i.e., collisions with the soft obstacles, and break ties by the travel time $|p|$. $\square$

[1] We have tried to adapt another two MAPF algorithms to replan the paths, namely Greedy CBS (Barer et al. 2014) and PBS (Ma et al. 2019). But they both perform worse than PP empirically.

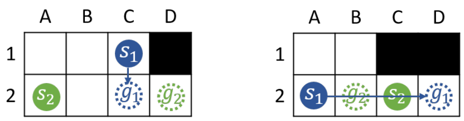

Figure 1: Examples. a_1 follows the arrow without waiting.

4.1 Space-Time A*

A straightforward algorithm for PMDO is space-time A*, which is widely used by many MAPF algorithms such as ID (Standley and Korf 2011) and ECBS (Barer et al. 2014). Space-time A* performs an A* search on a time-expanded graph where each state in the graph is defined by a vertex v and a timestep t, representing the agent being at vertex v at timestep t. The agent can move from state (v, t) to state (v', t') iff $((v, v') \in E \vee v = v') \wedge t' = t + 1$ holds and the move action does not collide with any obstacles in $\mathcal{O}^h$. In addition to the regular g-, h-, and f-values, each node also maintains a c-value that represents the number of soft collisions of the partial path from the root node to the current node. To find the optimal solution of a PMDO instance, we sort the nodes in the open list in ascending order of their c-values, breaking ties by their f-values.

While space-time A* can solve PMDO correctly, unfortunately it cannot do so efficiently. Consider the instance shown in Figure 1(left). If agent a_2 needs to plan a path that minimizes the number of collisions with the path of agent a_1, space-time A* needs to expand all the nodes that have zero collisions before finding the optimal path that has one collision with a_1 at C2 at timestep 2 (because a_1 reaches C2 at timestep 1 and remains there forever). However, there is a potentially infinite number of nodes that have zero collisions as the time dimension is unbounded. So space-time A* may not return a solution in finite time. Although one can fix this issue by using space-time A* restricted to states with timesteps no greater than the maximum of the timesteps of the obstacles and switching to standard A* (without the time dimension and wait actions) afterward (Ma et al. 2019), the total number of nodes it has to expand is still significant.

4.2 SIPPS

Safe Interval Path Planning (SIPP) (Phillips and Likhachev 2011) is a fast variant of space-time A* that uses time intervals instead of timesteps to represent the time dimension of the problem. It performs an A* search on a time-interval graph where each state in the graph is defined by a vertex and a safe (time) interval, representing that a particular vertex is free of hard obstacles during the time interval. For each state at vertex v with safe interval $[a, b)$, SIPP always prefers the (partial) path that arrives at v the earliest the possible within $[a, b)$ and then waits inside $[a, b)$ if necessary, since this allows SIPP to prune paths that arrive at v at a later time within $[a, b)$ without losing optimality. SIPP runs significantly faster than space-time A* empirically (Phillips and Likhachev 2011; Li et al. 2021b), yet it cannot handle soft obstacles. We thus generalize SIPP to *Safe Interval Path Planning with Soft constraints* (SIPPS) for solving PMDO.[2]

Algorithm 1: SIPPS

1 $\mathcal{T} \leftarrow$ *buildSafeIntervalTable*$(V, \mathcal{O}^h, \mathcal{O}^s)$;
2 *root* $\leftarrow$ *Node*$(s, \mathcal{T}[s][1], 1)$; // *1 is the index*
3 $T \leftarrow 0$; // *Lower bound on travel time*
4 **if** $\exists t : (g, t) \in \mathcal{O}^h$ **then** $T \leftarrow \max\{t \mid (g, t) \in \mathcal{O}^h\} + 1$;
5 compute g-, h-, f-, and c-values of *root*;
6 $Q \leftarrow \{root\}$, $P \leftarrow \emptyset$; // *Initialize open and closed lists*
7 **while** Q *is not empty* **do**
8 $\quad n \leftarrow Q.pop()$; // *Node with the smallest c-value*
9 $\quad$**if** $n.is\_goal$ **then return** *extractPath*(n);
10 $\quad$**if** $n.v = g \wedge n.low \geq T$ **then**
11 $\quad\quad c_{future} \leftarrow |\{(g, t) \in \mathcal{O}^s \mid t > n.low\}|$;
12 $\quad\quad$**if** $c_{future} = 0$ **then return** *extractPath*(n);
13 $\quad\quad n' \leftarrow$ a copy of n with is_goal set to *true*;
14 $\quad\quad c(n') \leftarrow c(n') + c_{future}$;
15 $\quad\quad$INSERTNODE(n', Q, P); // *Algorithm 3*
16 $\quad$EXPANDNODE$(n, Q, P, \mathcal{T})$; // *Algorithm 2*
17 $\quad P \leftarrow P \cup \{n\}$;
18 **return** "No Solution";

Safe Intervals A *safe interval* for a vertex is a contiguous period of time during which (1) there is no hard vertex/target obstacles, and (2) there is either a soft vertex/target obstacle at every timestep or no soft vertex/target obstacles at any timestep. We build a safe interval table $\mathcal{T}$ that maps each vertex $v \in V$ to a sequence of safe intervals $\mathcal{T}[v]$. To build $\mathcal{T}[v]$, we look at all hard and soft vertex and target obstacles at v and divide interval $[0, \infty)$ into a minimum set of disjoint safe intervals in $\mathcal{T}[v]$ in chronological order. We do not consider edge obstacles here as they are handled elsewhere.

SIPPS Nodes A SIPPS node n consists of four elements, namely a vertex $n.v$, a safe interval $[n.low, n.high)$ where $n.low$ is also called the *earliest arrival time*, an index $n.id$ indicating that the safe interval is (a subset of) the id-th safe interval in $\mathcal{T}[n.v]$ (i.e., interval $\mathcal{T}[n.v][n.id]$), and a Boolean flag $n.is\_goal$ indicating whether the node is a goal node (set to *false* by default). The f-value of node n is the sum of its g-value and h-value, where, the g-value is set to $n.low$, and the h-value is a lower bound on the minimum travel time from vertex $n.v$ to vertex g. Each node n also maintains a c-value, which is the number of the soft collisions of the partial path from the root node to node n, i.e., $c(n) = c(n') + c_v + c_e$, where n' is the parent node of n, c_v is 1 if the safe interval of n contains soft vertex/target obstacles and 0 otherwise, and c_e is 1 if $((n'.v, n.v), n.low) \in \mathcal{O}^s$ and 0 otherwise. If n is the root node (i.e., n' does not exist), $c(n) = c_v$.

Main Algorithm Algorithm 1 shows the pseudo-code of SIPPS. To begin with, we first build $\mathcal{T}$ and generate the root node at start vertex s with the first safe interval $\mathcal{T}[s][1]$ from $\mathcal{T}[s]$ and index 1 [Lines 1 and 2]. T is a lower bound on the travel time [Line 3]. If we have hard vertex obstacles at tar-

[2]To our knowledge, SCIPP (Cohen et al. 2019) is the only existing SIPP variant that handles soft obstacles. However, it cannot solve PMDO as it cannot handle hard or soft edge obstacles.

Algorithm 2: EXPANDNODE$(n, Q, P, \mathcal{T})$

1 $\mathcal{I} \leftarrow \emptyset$;
2 **foreach** $v : (n.v, v) \in E$ **do**
3 $\quad \mathcal{I} \leftarrow \mathcal{I} \cup \{(v, id) \mid \mathcal{T}[v][id] \cap [n.low + 1, n.high + 1) \neq \emptyset, id \in \mathbb{N}\}$;
4 **if** $\exists id : \mathcal{T}[n.v][id].low = n.high$ **then**
5 $\quad \mathcal{I} \leftarrow \mathcal{I} \cup \{(n.v, id)\}$; *// Indicates wait actions*
6 **foreach** $(v, id) \in \mathcal{I}$ **do**
7 $\quad [low, high) \leftarrow \mathcal{T}[v][id]$;
8 $\quad low \leftarrow$ earliest arrival time at v within $[low, high)$ without colliding with edge obstacles in $\mathcal{O}^h$;
9 $\quad$ **if** low *does not exist* **then continue**;
10 $\quad low' \leftarrow$ earliest arrival time at v within $[low, high)$ without colliding with edge obstacles in $\mathcal{O}^h \cup \mathcal{O}^s$;
11 $\quad$ **if** low' *exists* $\wedge low' > low$ **then**
12 $\quad\quad n_1 \leftarrow Node(v, [low, low'), id)$;
13 $\quad\quad$ INSERTNODE(n_1, Q, P) ; *// Algorithm 3*
14 $\quad\quad n_2 \leftarrow Node(v, [low', high), id)$;
15 $\quad\quad$ INSERTNODE(n_2, Q, P); *// Algorithm 3*
16 $\quad$ **else**
17 $\quad\quad n_1 \leftarrow Node(v, [low, high), id)$;
18 $\quad\quad$ INSERTNODE(n_2, Q, P); *// Algorithm 3*

get vertex g, then T is set to one plus the maximum timestep of all hard vertex obstacles at g [Line 4] since the agent cannot complete its path before any hard vertex obstacle at g. Q and P are regular open and closed lists [Line 6]. Q sorts its node in ascending order of their c-values, breaking ties in ascending order of their f-values. At every iteration, we pop a node n from Q [Line 8] and return its corresponding path if it is a goal node [Line 9]. Function *extractPath*(n) constructs a path by recursively backtracking the parent nodes until the root node is found. The resulting sequence of vertices are the reversed sequence of vertices visited by the path, with the time being the earliest arrival time of each node. If the difference between the earliest arrival times of two adjacent nodes is larger than one, we add wait actions in between accordingly. If n is at target vertex g with $n.low \geq T$ [Lines 10 to 15], it can be a goal node, but its c-value does not consider the number of additional soft collisions c_{future} that the agent would encounter if it stays at g forever after timestep $n.low$. We thus terminate only when c_{future} is 0 and generate a goal node that considers c_{future} otherwise. We last expand n [Line 16] and insert it to the closed list P [Line 17].

Expanding Nodes When expanding a node n (see Algorithm 2), we first store all reachable vertex-index pairs from vertex $n.v$ at a timestep within interval $[n.low, n.high)$ in $\mathcal{I}$ [Lines 1 to 5]. A vertex-index pair (v, id) is reachable iff the agent can move to v at a timestep within $\mathcal{T}[v][id]$ (i.e., $\mathcal{T}[v][id]$ overlaps with $[n.low + 1, n.high + 1)$) or wait at v from interval $[n.low, n.high)$ to interval $\mathcal{T}[v][id]$ (i.e., $n.high = \mathcal{T}[v][id].low$). For each vertex-index pair $(v, id) \in \mathcal{I}$ [Line 6], we use $[low, high)$ to represent the corresponding interval [Line 7]. We rewrite low to the earliest arrival time at v within $[low, high)$ without colliding with any hard edge obstacles. We jump to the next iteration if low does not exist [Line 9]. We then find the earliest arrival time low' at v within $[low, high)$ without colliding with any hard or soft edge obstacles. If low' exists and $low' > low$ [Lines 11 to 15], then the agent would collide with a soft edge obstacle if it arrives at v before low' and would not if it arrives at or after low'. Thus, we generate two child nodes, one with safe interval $[low, low')$ and one with safe interval $[low', high)$. The former child node has one more collision than the latter one. If low' does not exist or $low' = low$ [Lines 16 to 18], we generate one child node as usual.

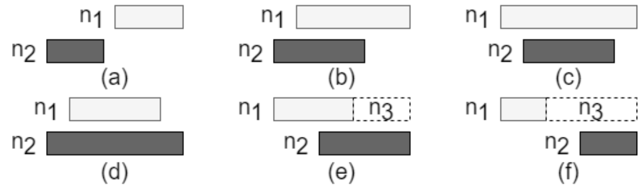

Figure 2: All possible combinations of the relative positions of the safe intervals of two nodes n_1 and n_2 with the same identity. The timeline is from left to right. Without loss of generality, we assume that $c(n_1) \leq c(n_2)$.

Inserting Nodes We say two nodes n_1 and n_2 have the same *identity*, denoted as $n_1 \sim n_2$, iff $n_1.v = n_2.v$, $n_1.id = n_2.id$, and $n_1.is\_goal = n_2.is\_goal$. We say n_1 *(weakly) dominates* n_2, denoted as $n_1 \succeq n_2$, iff $n_1 \sim n_2$, $[n_1.low, n_1.high) \supseteq [n_2.low, n_2.high)$, and $c(n_1) \leq c(n_2)$. We are interested in the dominance relationship because, if node n_1 dominates node n_2 (e.g., Figure 2(c)), we can prune n_2 without loss of completeness. Moreover, we know from Lines 6 to 18 in Algorithm 2 that a node n satisfies $n.high < \mathcal{T}[n.v][n.id].high$ iff it is generated in Line 12, i.e., there is a *twin node* n' with $n' \sim n$, $[n'.low, n'.high) = [n.high, \mathcal{T}[n.v][n.id].high)$, and $c(n') = c(n) - 1$. That is to say, if situations in Figures 2(e) and (f) occur, although n_1 does not dominates n_2, there exists a twin node n_3 of n_1 such that $n_1 \sim n_2 \sim n_3$, $[n_1.low, n_1.high) \cup [n_3.low, n_3.high) \supseteq [n_2.low, n_2.high)$, and $c(n_3) < c(n_1) \leq c(n_2)$. We can thus prune n_2. Therefore, we generalize the definition of the dominance as follow. We say node n_1 *(weakly) dominates* node n_2, denoted as $n_1 \succeq n_2$, iff $n_1 \sim n_2$, $n_1.low \leq n_2.low$, and $c(n_1) \leq c(n_2)$. We can prune a node if it is dominated by another node. For situations when two nodes with the same identity have overlapping intervals but no dominance relationship (see Figures 2(b) and (d)), intersection of the two intervals would be explored twice if we expand both nodes. We know that the node with the earlier interval always has the larger c-value (otherwise the two nodes would have a dominance relationship), so we can shrink the earlier interval by resetting its upper bound to the lower bound of the later interval. This can avoid the duplicate search effort without loss of completeness. The only unconsidered situation is the one shown in Figure 2(a), in which case we have to keep both nodes. In order to make SIPPS efficient, we use this analysis to avoid generating duplicate nodes in SIPPS. The pseudo-code is shown in Algorithm 3.

Algorithm 3: INSERTNODE(n, Q, P)

compute g-, h-, f-, and c-values of n;
$\mathcal{N} \leftarrow \{q \in Q \cup P \mid q \sim n\}$; *// Nodes identical to n*
foreach $q \in \mathcal{N}$ **do**
  if $q.low \le n.low \wedge c(q) \le c(n)$ **then** *// $q \succeq n$*
    return; *// No need to generate n*
  else if $n.low \le q.low \wedge c(n) \le c(q)$ **then** *// $n \succeq q$*
    delete q from Q or P; *// Prune q*
  else if $n.low < q.high \wedge q.low < n.high$ **then**
    if $n.low < q.low$ **then** $n.high = q.low$;
    else $q.high = n.low$;
insert n to Q;

Heuristics To achieve high efficiency, most MAPF algorithms use the distance $d(n.v, g)$ (i.e., the length of the shortest path) from $n.v$ to g as the h-value of node n when they plan paths for single agents, where the distance table d is computed during preprocessing. Such a heuristic is informed as long as the travel time of the optimal path p^* is not too larger than $d(s, g)$. Unfortunately, this is not always true for PMDO for two reasons: (1) T (which is a lower bound on $|p^*|$) can be substantially larger than $d(s, g)$ due to hard obstacles at target vertices; and (2) $T' = \max\{t \mid (g, t) \in \mathcal{O}^h \cup \mathcal{O}^s\} + 1$ (which is a lower bound on $|p^*|$ when p^* has zero soft collisions) can be substantially larger than $d(s, g)$. Therefore, we compute the h-value of a non-goal node n by

$$h(n) = \begin{cases} \max\{d(n.v, g), T' - g(n)\}, & c(n) = 0 \\ \max\{d(n.v, g), T - g(n)\}, & c(n) \ge 1 \end{cases}$$

The h-value of a goal node is, of course, 0.

Theoretical Analysis Below are two theorems for SIPPS. The proofs are omitted as they follow the proofs for SIPP.

Theorem 1. *SIPPS guarantees to return a path if one exists and "No Solution" otherwise.*

Theorem 2. *SIPPS guarantees to return a shortest path with zero soft collisions if one exists.*

One limitation of SIPPS is that, if no zero-soft-collision path exists, SIPPS may return a path with its number of soft collisions larger than the minimal one because the c-value ignores the soft collisions that occur when the agent waits within a safe interval that contains soft vertex/edge obstacles. Nevertheless, this approximation is acceptable since the number-of-collisions minimization itself is an approximation of the CP minimization used by MAPF-LNS2.[3] We have considered to directly minimize CP in SIPPS, but it is extremely inefficient as we have to keep track of the set of agents that the partial path from the root node to each node collides with, which substantially increases the search space.

Applications Although SIPPS is designed for MAPF-LNS2, it can be used by a broad family of MAPF algorithms as PMDO is a common problem that needs to be solved by the low levels of many MAPF algorithms. Examples include optimal algorithms ID (Standley and Korf 2011) and CBS (Sharon et al. 2015), bounded-suboptimal algorithm ECBS (Barer et al. 2014), and prioritized algorithm PBS (Ma et al. 2019) as well as their variants. With small changes in the priority function used by the open list of SIPPS (e.g., in CBS, prioritizing nodes with the smallest f-values and breaking ties by the c-values), SIPPS can largely speed up these MAPF algorithms while preserving their solution quality guarantees. Moreover, unlike space-time A*, SIPPS can be applied to continuous-time settings, so it can also speed up Continuous-time CBS (CCBS) (Andreychuk et al. 2019) and allows one to generalize CCBS to its suboptimal variants, e.g., continuous-time ECBS.

5 Neighborhood Selection

The selection of good neighborhoods is critical to the success of LNS. Here, we present three neighborhood selection methods and introduce adaptive LNS that intelligently combines these methods. Each method derives from a different motivation. Although there might be multiple implementations for each motivation, we present the one that works well for us and leave the exploration for other implementations for future work. We denote the current plan as P, the neighborhood as A_s, and the size of A_s as a predefined parameter N. $G_c = (V_c, E_c)$ is the *collision graph* where $V_c = \{i \mid a_i \in A\}$ and $E_c = \{(i, j) \mid p_i \in P$ collides with $p_j \in P\}$. We denote the degree of $v \in V_c$ as $deg(v)$.

Collision-Based Neighborhoods A straightforward idea to generating neighborhoods that can potentially reduce CP is to select a subset of agents whose current paths collide with each other. To implement this idea, we first select a random vertex v from V_c with $deg(v) > 0$ and find the largest connected component $G'_c = (V'_c, E'_c)$ of G_c that contains v. There are two cases: (1) If $|V'_c| \le N$, we put all agents a_v with $v \in V'_c$ in A_s, and repeatedly add additional agents that might collide with some agents in A_s to A_s until $|A_s| = N$. At each iteration, we select a random agent from A_s and let it perform a random walk starting from a random position on its path and stop when it hits another agent, which is then added to A_s. (2) Else, we select N vertices from V'_c via a random walk on G'_c and put the corresponding agents in A_s.

Failure-Based Neighborhoods The second idea is to reason about why we failed to find collision-free paths for some agents in the previous iterations. Finding a path for an agent a_i that does not collide with a given set of paths is an essential problem that is repeatedly solved in PP. Thus, previous work on PP has already studied the failure reasons of this problem thoroughly (Cáp et al. 2015). Briefly speaking, there are two scenarios, namely **(A)** a_i is blocked by the agents from the given set of paths "sitting" at their target vertices surrounding a_i (see Figure 1(left)), i.e., all possible

[3] Empirically, We ran MAPF-LNS2 on the random map with 400 agents using the setup described in Section 6 and collected the results of 84,739 SIPPS runs. Among them, more than 95% of runs find the minimum-collision paths, and 4% of runs find paths that contain only one more collision than the minimum (where the minimum-collision paths are found by space-time A*). In fact, although space-time A* guarantees to find minimum-collision paths, their numbers of colliding pairs are occasionally larger than those by SIPPS.

paths for a_i to reach g_i are blocked by some target obstacles, and **(B)** a_i is "run over" by the given set of paths at (or around) s_i during early timesteps (see Figure 1(right)), i.e, the agent has no way to go. Therefore, the failure-based neighborhood focuses on an agent a_i that has collisions and a set of agents whose paths visit s_i or whose target vertices are on some path from s_i to g_i. Formally, we first select an agent $a_i \in A$ with a probability proportional to $deg(i)$ and add a_i to A_s. We then collect two sets of agents $A^s = \{a_j \in A \mid p_j \in P \text{ visits } s_i\}$ and $A^g = \{a_j \in A \mid p \text{ visits } g_j\}$,where p is the path from s_i to g_i that minimizes $|A^g|$. There are three cases: (1) If $|A^s \cup A^g| = 0$, we terminate and return A_s, because we are guaranteed to find a path for a_i that does not collide with any other agents as a_i can sit at s_i until all other agents reach their target vertices and then move to g_i via path p. (2) Else if $|A^s \cup A^g| < N - 1$, we add the agents in $A^s \cup A^g$ to A_s and then repeatedly add additional agents to A_s whose target vertices are visited by the paths of some agents in A_s until $|A_s| = N$. At each iteration, we select a random agent a_j from A_s and collect the agents whose target vertices are visited by $p_j \in P$. We select a random agent from the collected agents and add it to A_s. (3) Else, we add $N - 1$ agents to A_s using the following rule. If $|A^s| = 0$, we add $N - 1$ random agents in A^g to A_s; Else if $|A^g| \geq N - 1$, we add the agent in A^s that visits s_i the earliest and $N - 2$ random agents in A^g to A_s; Else, we add all agents in A^g and the first $N - 1 - |A^g|$ agents in A^s (in ascending order of the timesteps when their paths visit s_i) to A_s. This rule prefers the agents in A^g slightly over the agents in A^s because we find empirically that Scenario (A) occurs more frequently than Scenario (B).

Random Neighborhoods Generating neighborhoods randomly may sound naïve, but it has been shown to be extremely effective for many problems (Demir, Bektas, and Laporte 2012; Song et al. 2020; Li et al. 2021a). We therefore design a random neighborhood method that selects N agents, each a_i with a probability proportional to $deg(i) + 1$.

Adaptive LNS (ALNS) ALNS (Ropke and Pisinger 2006) is a strong variant of LNS. It makes use of multiple neighborhood selection methods by recording their relative success in improving solutions and generating the next neighborhood by the most promising method. Formally, we maintain a weight w_i for each neighborhood selection method i that represents its relative success in reducing the CP. Initially, all $w_i = 1$. At each iteration, we select a method i with probability $w_i / \sum_j w_j$ to generate a neighborhood and replan the paths. After the replan, we set w_i to $\gamma \cdot \max\{0, c^- - c^+\} + (1 - \gamma) \cdot w_i$, where c^- and c^+ are the CPs of the plans before and after the replan, respectively, and $\gamma \in [0, 1]$ is a user-specified reaction factor that controls how quickly the weights react to the changes in the relative success in reducing the CP. We use $\gamma = 0.1$ in our experiments. The weights for the other methods remain the same.

6 Experiments

We compare MAPF-LNS2 against a representative set of scalable state-of-the-art MAPF algorithms, namely bounded-suboptimal algorithm EECBS, prioritized algorithms PP and PP with random restarts (PP^R) (where we repeatedly rerun PP with a random priority ordering until it finds a solution), and rule-based algorithm PPS. In addition, in order to show the effectiveness of SIPPS for speeding up MAPF algorithms other than MAPF-LNS2, we implement a variant of EECBS (denoted as EECBS*) that replaces space-time A* with SIPPS. Both PP and PP^R use SIPP to plan paths for single agents (they do not use SIPPS as their underlying single-agent problem does not have soft obstacles). If not specified, MAPF-LNS2 uses PP to find initial plans, ALNS to generate neighborhoods of size $N = 8$, and SIPPS to plan paths for single agents. We use the random-scenario instances on all 33 maps from the MAPF benchmark suite (see https://movingai.com/benchmarks/mapf/), yielding 25 instances per map per number of agents. We conduct experiments on Amazon EC2 "m4.xlarge" instances with 16 GB memory. If not specified, the time limit is 5 minutes. Due to space limit, we report results only on the random map *random-32-32-20* in Experiments 1-3 and the warehouse map *warehouse-20-40-10-2-2* in Experiment 5.

m	Success rate		Runtime (s)		Runtime per call (ms)	
	A*	SIPPS	A*	SIPPS	A*	SIPPS
250	**1.00**	**1.00**	3.37	**0.64**	5.49 ± 17.19	1.11 ± 1.79
300	**1.00**	**1.00**	15.99	**2.67**	10.9 ± 28.72	1.94 ± 2.79
350	0.88	**1.00**	>68	**9.25**	15.83 ± 43.52	2.75 ± 3.72
400	0.68	**0.88**	>162	**>78**	15.28 ± 40.95	3.04 ± 4.23

Table 1: Comparing SIPPS against A* on the random map.

m	Success rate				Runtime (s)			
	R	F	C	A	R	F	C	A
250	**1.00**	**1.00**	**1.00**	**1.00**	0.80	**0.59**	0.79	0.64
300	**1.00**	**1.00**	**1.00**	**1.00**	13.13	3.41	3.09	**2.67**
350	**1.00**	0.96	**1.00**	**1.00**	32.57	>22	**9.11**	9.25
400	0.48	0.60	0.76	**0.88**	>192	>155	>128	**>78**

Table 2: Comparing ALNS (denoted as A) against random (denoted as R), failure-based (denoted as F), and collision-based (denoted as C) neighborhoods on the random map.

m	Success rate			Runtime (s)		#runs		Init
	PP	PP^R	MAPF-LNS2	PP^R	MAPF-LNS2	PP^R	MAPF-LNS2	CP
100	0.56	**1.00**	**1.00**	0.02	**0.01**	179	**105**	0.6
200	0.08	**1.00**	**1.00**	6.69	**0.14**	47,114	**262**	5
300	0.00	0.00	**1.00**	>300	**2.67**	-	**1,285**	61
400	0.00	0.00	**0.88**	>300	**>78**	-	-	316

Table 3: Comparing MAPF-LNS2 against PP and PP^R on the random map. We omit the runtime of PP as it is equal to the runtime of PP^R and MAPF-LNS2 for any instance it has solved. *#runs* is the average number of times for which we run SIPP(S). *Init CP* is the average CP of the initial plan.

Experiment 1 compares PMDO algorithms. Table 1 compares MAPF-LNS2 with SIPPS against MAPF-LNS2 with space-time A* (or A* for short) in terms of *success rates* (i.e., percentages of instances solved within the time limit), average runtimes (with 5 minutes for unsolved instances), and average runtimes per SIPPS/A* call with their standard deviations. SIPPS clearly dominates A* with a speedup of more than 5 times. It is also more stable, e.g., the largest runtime per call for SIPPS is 51ms while that of A* is 524ms

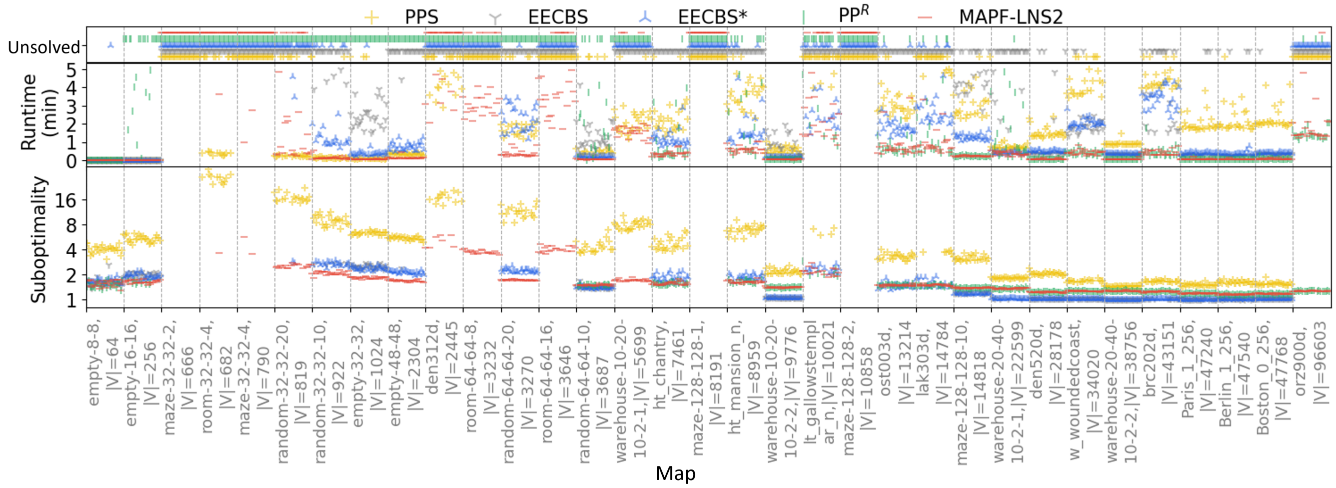

Figure 3: Runtime and solution quality on all maps. Suboptimality is overestimated by $\sum_{i=1}^{m}|p_i|/\sum_{i=1}^{m}d(s_i,g_i)$.

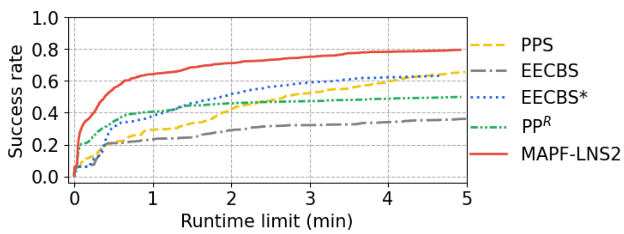

Figure 4: Success rates on all maps.

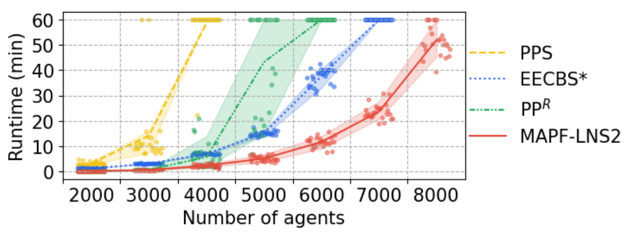

Figure 5: Runtime on the warehouse map. Each dot represents the runtime on one instance, with each line and filled area representing the mean and 0.1-quantile values over the 25 randomly generated instances for each number of agents.

(not shown in the table). Such difference is larger on larger maps.

Experiment 2 compares neighborhood selection methods. Table 2 compares MAPF-LNS2 with ALNS against MAPF-LNS2 with individual neighborhood selection methods. As expected, ALNS performs overall the best as it combines the strengths of the other methods and explores a larger variety of neighborhoods. We also experimented with different neighborhood sizes $N = 4, 8, 16, 32$ but omit the results since the observation is similar to that in the previous work (Li et al. 2021a): there is no global winner, and larger neighborhoods have larger chances to find better solutions but require more time to replan, resulting in fewer iterations eventually.

Experiment 3 compares MAPF-LNS2 against other PP-based algorithms, namely PP and PP^R. MAPF-LNS2 can be viewed as a PP-based algorithm as it uses PP to both find initial plans and replan. As shown in Table 3, MAPF-LNS2 performs the best. It rapidly reduces the CP of the initial plan generated by PP, and, as a result, largely improves the success rate of PP. This LNS framework is a more efficient framework than random restarts since MAPF-LNS2 requires significantly fewer single-agent runs than PP^R, which in turn results in significantly higher success rates and lower runtimes. Although MAPF-LNS2 failed to solve 3 instances with 400 agents, its final plans have only 1, 1, and 2 colliding pairs (not shown in the table).

Experiment 4 compares MAPF-LNS2 against state-of-the-art algorithms PP^R, PPS, and EECBS (with a suboptimality of 5) as well as our EECBS* (also with a suboptimality of 5).[4] We use instances on all 33 maps from the benchmark suite with the largest number of agents available in the random scenario, i.e., $m = \min\{0.5|V|, 1000\}$ for each map. Figure 3 shows the results for each instance, and Figure 4 summarizes the success rates. MAPF-LNS2 solves more than 60% of instances within 1 minute and 80% of instances within 5 minutes. Its success rate is always the highest for all runtime limits. The instances that MAPF-LNS2 fail to solve mostly come from highly congested maps, such as the maze and room maps, which are not solved by the other algorithms as well in most cases. Although PPS solves a few instances that are not solved by MAPF-LNS2, its solution quality is always substantially worse than that of MAPF-LNS2 (and other algorithms).

[4]We picked 5 as the suboptimality bound because we intended to choose a large enough suboptimality bound such that, if EECBS fails to solve an instance that MAPF-LNS2 has solved, it is due to the scalability limit of EECBS rather than it using a too-small suboptimality bound.

EECBS* finds solutions slightly better than MAPF-LNS2 on some instances, yet its runtime is always larger. We did not use EECBS*/PPS to find initial plans for MAPF-LNS2 because, whenever PP can find solutions, it always finds them faster than EECBS*/PPS, and whenever it cannot, MAPF-LNS2 can repair the plan rapidly and result in better performance than EECBS*/PPS eventually. In addition, the difference in the success rates and runtimes of EECBS and EECBS* clearly shows the advantage of SIPPS over space-time A*, especially on large maps. The success rate of EECBS* is almost twice as large as that of EECBS with a time limit of 5 minutes. In terms of the memory usage, the memory usage of PPS and EECBS(*) increases fast over time (as they generate longer and longer paths or keep a larger and lager search frontier), while that of PP^R and MAPF-LNS2 stays stable. Thus, PP^R and MAPF-LNS2 usually end up with a substantially smaller memory usage after 5 minutes than PPS and EECBS(*).

Experiment 5 examines a longer time limit of an hour. Figure 5 shows that MAPF-LNS2 still performs the best. It plans collision-free paths for 3,000 agents within a minute, 5,000 agents within 5 minutes, and 8,000 agents within a hour.

7 Summary

We propose a suboptimal algorithm MAPF-LNS2 that solves MAPF by repeatedly repairing the colliding paths in a given set of paths. MAPF-LNS2 solves 80% of the most challenging MAPF-benchmark instances within a time limit of just 5 minutes, which significantly outperforms a variety of state-of-the-art MAPF algorithms. In addition, the single-agent path planner SIPPS used by MAPF-LNS2 runs 5 times (or more) faster than space-time A* and can be used to speed up a variety of MAPF algorithms. For example, it almost doubles the success rate of EECBS within 5 minutes in our experiments.

Acknowledgments

The research at the University of Southern California was supported by the National Science Foundation under grant numbers 1409987, 1724392, 1817189, 1837779, and 1935712 as well as a gift from Amazon. The research at Monash University was partially supported by the Australian Research Council under Discovery Grants DP190100013 and DP200100025 as well as a gift from Amazon. 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

Andreychuk, A.; Yakovlev, K. S.; Atzmon, D.; and Stern, R. 2019. Multi-Agent Pathfinding with Continuous Time. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 39–45.

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 Annual Symposium on Combinatorial Search (SoCS)*, 19–27.

Belov, G.; Du, W.; de la Banda, M. G.; Harabor, D.; Koenig, S.; and Wei, X. 2020. From Multi-Agent Pathfinding to 3D Pipe Routing. In *Proceedings of the Thirteenth International Symposium on Combinatorial Search (SoCS)*, 11–19.

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.

Cáp, M.; Novák, P.; Kleiner, A.; and Selecký, M. 2015. Prioritized Planning Algorithms for Trajectory Coordination of Multiple Mobile Robots. *IEEE Transactions on Automation Science and Engineering*, 12(3): 835–849.

Cohen, L.; Uras, T.; Kumar, T. K. S.; and Koenig, S. 2019. Optimal and Bounded-Suboptimal Multi-Agent Motion Planning. In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, 44–51.

Cohen, L.; Wagner, G.; Chan, D. M.; Choset, H.; Sturtevant, N. R.; Koenig, S.; and Kumar, T. K. S. 2018. Rapid Randomized Restarts for Multi-Agent Path Finding Solvers. In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, 148–152.

Demir, E.; Bektas, T.; and Laporte, G. 2012. An adaptive large neighborhood search heuristic for the Pollution-Routing Problem. *European Journal of Operational Research*, 223(2): 346–359.

Erdmann, M.; and Lozano-Perez, T. 1987. On multiple moving objects. *Algorithmica*, 2(1-4): 477.

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 (ICAPS)*, 155–162.

Lam, E.; and Le Bodic, P. 2020. New Valid Inequalities in Branch-and-Cut-and-Price for Multi-Agent Path Finding. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 184–192.

Li, J.; Chen, Z.; Harabor, D.; Stuckey, P. J.; and Koenig, S. 2021a. Anytime Multi-Agent Path Finding via Large Neighborhood Search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.

Li, J.; Chen, Z.; Zheng, Y.; Chan, S.-H.; Harabor, D.; Stuckey, P. J.; Ma, H.; and Koenig, S. 2021b. Scalable Rail Planning and Replanning: Winning the 2020 Flatland Challenge. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 477–485.

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.

Okumura, K.; Machida, M.; Défago, X.; and Tamura, Y. 2019. Priority Inheritance with Backtracking for Iterative Multi-agent Path Finding. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 535–542.

Phillips, M.; and Likhachev, M. 2011. SIPP: Safe interval path planning for dynamic environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 5628–5635.

Ropke, S.; and Pisinger, D. 2006. An Adaptive Large Neighborhood Search Heuristic for the Pickup and Delivery Problem with Time Windows. *Transportation Science*, 40(4): 455–472.

Sajid, Q.; Luna, R.; and Bekris, K. E. 2012. Multi-Agent Pathfinding with Simultaneous Execution of Single-Agent Primitives. In *Proceedings of the Annual Symposium on Combinatorial Search (SoCS)*.

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

Shaw, P. 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)*, volume 1520, 417–431.

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

Song, J.; Lanka, R.; Yue, Y.; and Dilkina, B. 2020. A General Large Neighborhood Search Framework for Solving Integer Programs. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS)*.

Standley, T. S.; and Korf, R. E. 2011. Complete Algorithms for Cooperative Pathfinding Problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 668–673.

Stern, R.; Sturtevant, N. R.; Felner, A.; Koenig, S.; Ma, H.; Walker, T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. K. S.; Boyarski, E.; and Bartak, R. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, 151–159.

Wang, H.; and Rubenstein, M. 2020. Walk, Stop, Count, and Swap: Decentralized Multi-Agent Path Finding With Theoretical Guarantees. *IEEE Robotics and Automation Letters*, 5(2): 1119–1126.