2020 IEEE International Conference on Robotics and Automation (ICRA)
31 May - 31 August, 2020. Paris, France

Optimal Sequential Task Assignment and Path Finding for Multi-Agent Robotic Assembly Planning

Kyle Brown Oriana Peltzer Martin A. Sehr Mac Schwager Mykel J. Kochenderfer

***Abstract*— We study the problem of sequential task assignment and collision-free routing for large teams of robots in applications with inter-task precedence constraints (e.g., task A and task B must both be completed before task C may begin). Such problems commonly occur in assembly planning for robotic manufacturing applications, in which sub-assemblies must be completed before they can be combined to form the final product. We propose a hierarchical algorithm for computing makespan-optimal solutions to the problem. The algorithm is evaluated on a set of randomly generated problem instances where robots must transport objects between stations in a "factory" grid world environment. In addition, we demonstrate in high-fidelity simulation that the output of our algorithm can be used to generate collision-free trajectories for non-holonomic differential-drive robots.**

I. Introduction

Consider a factory environment with an array of manufacturing stations and a fleet of autonomous mobile robots. The *project* consists of an initial set of objects and a prescribed set of manufacturing operations that will incrementally transform those initial objects into a final object. The goal is to assign and route robots to transport objects between stations so that the project makespan—the time from start to completion—is minimized. We call this problem precedence-constrained multi-agent task assignment and path-finding (PC-TAPF).

PC-TAPF generalizes the multi agent pathfinding (MAPF) problem with task assignment, sometimes called the anonymous MAPF [1]. Whereas anonymous MAPF involves a *one off* task assignment problem (each robot performs no more than one task) in which all tasks are independent, PC-TAPF involves *sequential* task assignment (each robot may be assigned to a sequence of tasks) and incorporates temporal precedence constraints between tasks (e.g., task A and task B must both be completed before task C may begin). Both problems involve collision-free routing, but the PC-TAPF routing problem is affected by the same precedence constraints that are present in the assignment problem. Although minimum-makespan anonymous MAPF problems with homogeneous agents can be solved in polynomial time [1], the PC-TAPF problem is NP-hard. The difference lies in the "cross-schedule dependencies" that arise from the inter-task precedence constraints [2]. As the ratio of tasks to agents increases, the sequential assignment problem approaches a traveling salesman problem. As the breadth of the task dependency graph (i.e., the degree to which it is parallelizable) increases, the route-planning problem becomes increasingly congested and potentially ill-matched with optimistic task assignment. However, PC-TAPF problems often exhibit useful structure that can be exploited to efficiently find optimal solutions.

Various methods have been proposed for solving problems that are mathematically similar to PC-TAPF. Some perform joint task assignment and collision-free route-planning, but do not handle dependencies between tasks [3]–[7]. Others handle tasks with temporal dependencies, but ignore the routing problem or assume that agents will never collide [8]–[10]. To the best of our knowledge, no solver has been proposed to optimally solve the combined sequential task assignment and routing problems with inter-task precedence constraints and collision-free constraints.

In this article, we propose a hierarchical algorithm for optimally solving PC-TAPF problems. The first level assigns tasks to robots by solving a relaxed problem that ignores non-collision constraints. The task assignments are passed down to a Conflict-Based Search (CBS) level, which searches over a constraint tree for an optimal collision-free set of paths [11]. At each node of the constraint tree, CBS calls a lower level routine that incrementally constructs a joint route plan by iterating over a dependency graph and repeatedly calling a modified version of A* [12]. We evaluate the runtime of the algorithm and its component parts on a suite of problem instances that vary in size and structure, and show that even large problem instances (i.e. 40 robots, 60 tasks) can be efficiently solved to optimality in most cases. We also demonstrate our solver with a fleet of non-holonomic differential-drive robots in high-fidelity simulation.

II. Background

Many variants of multi-agent pathfinding (MAPF) [13] problems exist in the literature. Related problems include MAPF with Deadlines (MAPF-DL) [7], multi-agent pick-up and delivery (MAPD) [14], and combined target-assignment and path-finding (TAPF) with teams of heterogeneous agents [5], [15]. Optimal solvers for path-finding problems are often based on Conflict-Based Search (CBS) [11], Network flow [15] or subdimensional expansion [4]. Conflict-Based Search with Task Assignment (CBS-TA) addresses heterogeneous anonymous MAPF problems [3]. CBS-TA searches over a

*This work was supported by Siemens and the National Science Foundation.

[1]Kyle Brown and Oriana Peltzer are with the Stanford University Department of Mechanical Engineering, {kjbrown7, peltzer}@stanford.edu.

[2]Martin A. Sehr is with Corporate Technology, Siemens Corporation, Berkeley, CA 94704, USA, martin.sehr@siemens.com.

[3]Mac Schwager and Mykel Kochenderfer are with the Stanford University Department of Aeronautics and Astronautics,{mykel, schwager }@stanford.edu.

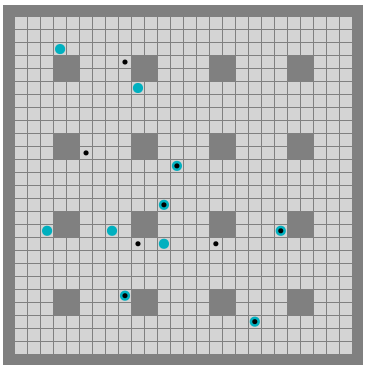


Fig. 1: The factory environment. Dark gray regions represent obstacles (manufacturing stations). Blue disks represent robots, and smaller black disks represent objects.

forest of CBS constraint trees, where each tree in the search forest corresponds to a different task assignment matrix. Our algorithm is similar, in the sense that it solves a relaxed "assignment" problem at the top level and queries a lower-level route-planner, iterating through all possible assignments in best-first order until there is zero gap between the lower bound from the assignment problem and the best route plan.

Several types of vehicle routing and scheduling problems (VRSPs) incorporate temporal dependencies—such as precedence, synchronization, and time-windowing constraints—between tasks. Bredström and Rönnqvist introduce a traveling repairman VRSP with precedence and synchronization constraints [10]. Dohn, Rasmussen, and Larsen introduce a formulation for modeling general temporal dependencies between tasks [8]. Exact methods for VRSPs with temporal dependencies usually involve solving a mixed integer linear program (MILP) [8]. We do the same at the top level of our hierarchical algorithm. Because such solution methods are often intractable for large problem instances, many decentralized approaches accept a degree of suboptimality in exchange for savings in computation time. These include algorithms like M+, wherein robots negotiate with each other in a "task market [16].

III. Problem Statement

The factory environment is modeled as a two-dimensional grid world. Manufacturing stations are regularly spaced throughout the environment, and each station is surrounded by an array of pick-up and drop-off zones, at which objects may be collected and deposited, respectively. The discrete state x_i^t of robot i at time t corresponds to a grid cell in the environment. At each time step, each robot may remain in place or move in any of the compass directions to arrive in an adjacent grid cell at the next time step. Cells corresponding to manufacturing stations may not be entered. robot i may collect object j if they occupy the same grid cell (i.e., $x_i^t = o_j^t$, where o_j^t denotes the object state), and the object moves with the robot until the robot deposits it. The environment is shown in fig. 1.

A *project specification* S defines the set of manufacturing operation that must be performed. Each operation consumes one or more input objects and produces (except in the case of the no-output terminal operation) a single output object. Precedence constraints arise whenever the output of one operation is required as an input to another operation. The project specification also defines the pick-up and drop-off locations of all associated objects. Figure 2a depicts a simple project specification. The project *makespan* is defined as the time from start ($t = 0$) to completion of the terminal operation.

TABLE I: Notation

n	number of robots	$G = (\mathcal{V}, \mathcal{E})$	operating schedule
m	number of objects	$v \in \mathcal{V}$	schedule vertex
x_i	state of robot i	$(v \to v') \in \mathcal{E}$	schedule edge
o_j	state of object j	$v.t^0$	vertex start time
$p_i = x_i^{0:t}$	trajectory of agent i	$v.\Delta t$	vertex duration
		$v.t^F$	vertex end time

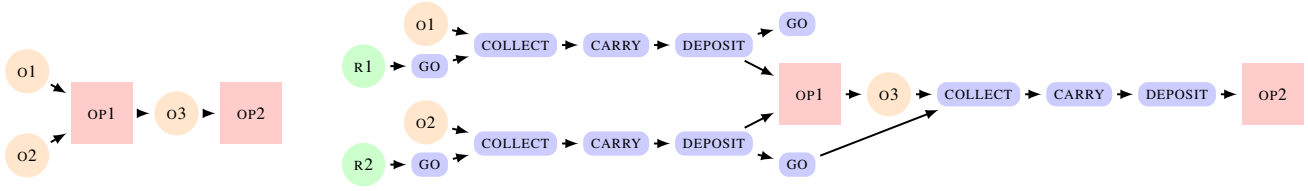
A. Operating Schedules and Route Plans

The *operating schedule* $G = (\mathcal{V}, \mathcal{E})$ is a directed acyclic graph (DAG). Each vertex $v \in \mathcal{V}$ of the graph corresponds to one of the following discrete high-level events or activities: initial object condition OBJECT_AT, initial robot condition ROBOT_AT, manufacturing operation OPERATION, and the robot actions GO, COLLECT, CARRY and DEPOSIT. An edge $(v \to v') \in \mathcal{E}$ denotes a precedence constraint, meaning that the activity associated with v' may not begin until the activity associated with v has been completed. The required topological structure of the schedule is intuitive: In order to perform a transport task, a robot must *go* to the appropriate pick-up location, *collect* an object, *carry* it through the factory, and finally *deposit* it at the drop-off location. A manufacturing operation may only begin once all input objects have been deposited at the prescribed drop-off points. An output object becomes available for collection only once the associated operation has been completed. An example operating schedule is shown in fig. 2b.

Each vertex v has a start time $v.t^0$ and completion time $v.t^F$. For root nodes only (i.e., ROBOT_AT vertices and the initial OBJECT_AT vertices), start times are fixed. The completion time of every vertex is equal to $v.t^0 + v.\Delta t$, where Δt is the duration of the activity. For ROBOT_AT and OBJECT_AT nodes, $\Delta t = 0$ by definition. Each COLLECT, DEPOSIT and OPERATION node may have a fixed positive Δt. For navigation nodes (GO and CARRY), Δt is a variable that depends on how quickly the robot travels from the node's start location to its destination location.

An operating schedule is *valid* if and only if (a) all vertices have the correct number of incoming and outgoing edges to the appropriate neighbor vertices and (b) there are no cycles in the graph. An example of a cycle would be if robot i were assigned to first deliver object j and then deliver one of object j's prerequisites, which would require traveling backward through time. Constructing a valid schedule corresponds to solving a sequential task assignment problem.

A *route plan* is denoted by $P = (p_1, \ldots, p_n)$, where



(a) (b)

Fig. 2: (a) An example project specification and (b) a possible corresponding schedule graph with $n = 2$ robots. robot 1 is assigned to deliver object 1, and robot 2 is assigned to first deliver object 2 and then deliver object 3 once Operation 1 is complete.

$p_i = (x_i^0, x_i^1, \ldots, x_i^t) = x_i^{0:t}$ is the trajectory of agent i from time 0 to t. A route plan is *consistent* if and only if it satisfies all environment constraints in addition to the constraints associated with the operating schedule (i.e., each robot must be at the relevant pick-up and drop-off locations at the time steps specified by the operating schedule). These latter constraints may be understood as boundary conditions that partition an individual robots trajectory into disjoint segments. A route plan is *valid* if and only if it is consistent and it contains no *conflicts* between trajectories. A *state-conflict* between agents i and j is said to occur at time t if $x_i^t = x_j^t$. An *action conflict* occurs if $(x_i^t = x_j^{t+1}) \wedge (x_i^{t+1} = x_j^t)$, where the two agents switch places in a single time step (which would require that they pass through each other).

A *solution* consists of a operating schedule and corresponding route plan. A solution is valid if and only if the schedule and the route plan are both valid. A valid solution is optimal if and only if no other valid solution has a lower *makespan*. The project makespan T is defined as the total number of timesteps from the beginning of a project ($t = 0$) until the terminal operation is completed.

IV. Methods

We propose a four-level hierarchical planning algorithm to optimally solve PC-TAPF problems. Each of the four levels is described in this section. A graphical summary of the algorithm is given in fig. 3.

A. Level 1: Sequential Next-Best Assignment Search

Sequential Next-Best Assignment Search (NBS) computes a valid operating schedule by solving a sequential task assignment problem. The assignment problem is a relaxation of the full PC-TAPF problem, because it ignores the constraint that robots must not collide. The solution to the assignment problem is an *optimistic* operating schedule, and its makespan is a lower bound on the makespan of the optimal solution to the full PC-TAPF problem. The optimistic schedule is passed to Level 2, CBS, which returns a corresponding minimum-makespan valid route plan. The makespan of this route plan (which may be higher than the lower bound, since CBS must respect the collision-free constraints) constitutes an upper bound on the makespan of the optimal valid solution. NBS tries all possible assignments in best-first order until no assignments remain with better makespan than the best route plan found by CBS up to that point. NBS is summarized in algorithm 1.

The relaxed problem is formulated as a mixed integer linear program (MILP). We re-cast the sequential assignment problem as a one-off assignment problem by introducing m "dummy robots", where the jth dummy robot is shorthand for "the robot that just delivered object j." We encode the discrete decision variable as an binary *assignment matrix* $A \in \mathbf{B}^{(n+m)\times m}$, where $A_{ij} = 1$ indicates that robot i is assigned to transport object j. The first n rows of A correspond to the real robots, and the final m rows correspond to the dummy robots. Hence, if $A_{ij} = 1$ and $A_{j+n,k} = 1$, then robot i is assigned to deliver object j and then to deliver object k. Because the jth dummy robot is a placeholder for the robot that already delivered object j, constraints must be added so that dummy robot j cannot be assigned to object j or any of its prerequisites (as this would lead to a cyclic schedule graph).

To simplify notation, we define t_i^0, τ_j^0, and τ_j^F as the start time of robot i, the pickup time of object j, and the delivery time of object j, respectively. The MILP formulation is given by

$$\begin{aligned}
\text{minimize} \quad & \tau_m^F && (1)\\
& A_{ij} \in \{0,1\} \quad i \in 1:n+m, j \in 1:m && (2)\\
& \textstyle\sum_{i=1}^{n+m} A_{ij} = 1, \quad j \in 1:m && (3)\\
& \textstyle\sum_{j=1}^{m} A_{ij} \leq 1, \quad i \in 1:n+m && (4)\\
& A_{j+n,k} = 0, \quad j \in 1:m, \quad k \in \text{PRED}(j) && (5)\\
& t_i^0 = 0, \quad i \in 1:n && (6)\\
& t_{j+n}^0 = \tau_j^F, \quad j \in 1:m && (7)\\
& \tau_j^0 \geq 0, \quad j \in 1:m && (8)\\
& \tau_j^0 \geq \tau_k^F + \text{OP}.\Delta t, \quad \text{OP} \in S.operations, \\
& \quad j \in \text{OP}.inputs, \ k \in \text{OP}.outputs && (9)\\
& \tau_j^F \geq \tau_j^0 + \text{COLLECT}_j.\Delta t + d(s_j, g_j) \\
& \quad + \text{DEPOSIT}_j.\Delta t, \quad j \in 1:m && (10)\\
& \tau_j^0 - (t_i^0 + d(x_i^0, s_j)) \geq -M(1 - A_{ij}), \\
& \quad i \in 1:n, \quad j \in 1:m, && (11)
\end{aligned}$$

where (1) defines the project makespan, (2) constrains the elements of the assignment matrix to be binary, (3) ensures that each task is assigned to exactly one robot, (4) ensures that each robot (including dummies) is assigned to no more

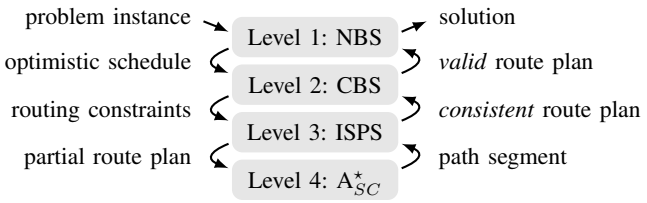


Fig. 3: A schematic overview of our algorithm.

than one task, (5) prevents each dummy robot j from being assigned to any task $k \in \text{PRED}(j)$ among the predecessors of its parent task, (7) constrains the start time of each dummy robot to match the completion time of its parent task, (6) defines the start times for all non-dummy robots, (8) ensures that no object can be collected before time $t = 0$, (9) enforces all task precedence constraints that arise from the operations in the project spec, (5) constrains the time between task start and task completion to be separated by at least collection time plus the minimum travel time between the pick-up and drop-off locations plus the deposit time, and (11) uses the big M method to constrain the start time of task j to be at least the start time of the assigned robot plus the minimum travel time $d(x_i^0, g_j)$ from the robot initial location to the object pick-up location. The MILP can be solved with any off-the-shelf solver. We use Gurobi in our experiments [17].

Algorithm 1 Sequential Next-Best Assignment Search (NBS)

```
1: procedure NBS(PC-TAPF)
2:     MILP ← FORMULATE_MILP(PC-TAPF)
3:     (S*, T*) ← (nothing, ∞)                    ▷ best solution
4:     T_ ← 0                                     ▷ lower bound
5:     while T* > T_
6:         G, T_ ← SOLVE(MILP)
7:         if T* > T_
8:             S, T ← CBS(PC-TAPF, G)             ▷ Call Level 2
9:             if T < T*
10:                (S*, T*) ← (S, T)
11:            MILP.constraints ← (G ≠ S.G)
12:    return S*, T*
```

B. Level 2: Conflict-Based Search

CBS is a general framework for multi agent path finding [11]. It operates by best-first search over a binary constraint tree, where each node of the constraint tree contains a set of constraints (the root node constraint set is empty). A *state constraint* $c_S = (i, x, t)$ specifies that robot i may not occupy position x at time t, and an *action constraint* $c_A = (i, x, x', t)$ specifies that robot i may not move from position x to position x' at time t.

When CBS receives a operating schedule from NBS, it initializes the root node of the binary constraint tree. Beginning at the root node, CBS passes the operating schedule and the current node's constraint set down to the next level of planner. ISPS returns a *consistent* route plan that also respects all constraints passed in by CBS, or an infeasible flag if no consistent route plan can be found. If the route plan is invalid (i.e., it has at least one conflict), CBS *branches* by generating a pair of mutually exclusive constraints and adding each to a new search node that also inherits all the constraints of its parent search node. CBS calls ISPS for each child search node to recompute a new solution that respects the added constraint, then adds each search node to a priority queue with priority defined by its makespan. CBS continues until (1) a valid solution is obtained, (2) the cost of the best node in the queue exceeds the upper bound on the optimal solution, or (3) the priority queue is exhausted. In the latter two cases, CBS returns no solution.

C. Level 3: Incremental Slack-Prioritized Search

Algorithm 2 Incremental Slack-Prioritized Search (ISPS)

```
1: procedure ISPS(G, constraints)
2:     P ← ([x_1^0], ..., [x_n^0])                ▷ init route plan
3:     C ← ∅                                      ▷ closed set
4:     Q ← INITQUEUE(G, C)                        ▷ priority queue
5:     while Q ≠ ∅
6:         v ← DEQUEUE(Q)                         ▷ pop vertex from queue
7:         p ← A*_SC(P, v, constraints)           ▷ plan path segment p
8:         v.t^F ← p.t^F                          ▷ update vertex final time
9:         P ← ADD_PATH(P, p)                     ▷ update route plan
10:        C ← {v} ∪ C                            ▷ update closed set
11:        G ← UPDATESCHEDULE(G)
12:        Q ← INITQUEUE(G, C)
13:    return P

14: procedure UPDATESCHEDULE(G)
15:    for v ∈ TOPOLOGICAL_SORT(G)
16:        for v' ∈ PRED(v)
17:            v.t^0 ← max(v.t^0, v'.T)
18:        v.t^F ← max(v.t^0 + v.Δt, v.t^F)
19:    for v ∈ REVERSE_TOPOLOGICAL_SORT(G)
20:        for v' ∈ SUCC(v)
21:            v.SLACK ← min(v.SLACK, v'.t^0 + v'.SLACK − v.t^F)
22:        v.t^F ← max(v.t^0 + v.Δt, v.t^F)
23:    return G

24: procedure INITQUEUE(G, C)
25:    for v ∈ VERTICES(G)
26:        if v ∉ C ∧ PRED(v) ⊆ C
27:            Q ← ENQUEUE(Q, v → v.SLACK)
28:    return Q
```

The Incremental Slack-Prioritized Search (ISPS) module receives a operating schedule and a set of constraints from CBS. ISPS incrementally constructs a consistent plan by traversing the schedule graph in topological order and calling $\text{A}^\star_{SC}$ to compute a path segment for each vertex. The key idea behind ISPS is that it takes advantage of *slack* in the operating schedule. Slack denotes the amount of room for delay in a vertex's final time $v.T$ before the makespan would increase. Vertices with high slack can afford significant delay without affecting the project completion time. Vertices with zero slack are on the *critical path*. If they are delayed by even a single timestep, the entire project will be delayed.

ISPS begins by adding all root vertices (vertices with no predecessors) of the schedule graph to a priority queue Q where they are prioritized by their slack. The lowest slack vertex is popped from the queue, and $\text{A}^\star_{SC}$ is called to plan a path segment corresponding to that vertex. When planning is complete for that vertex, it is placed in the *closed set* $\mathcal{C}$. The path segment is added to the partial route plan, and ISPS calls a subroutine to update the operating schedule and recompute the slack of all vertices.When all of a vertex's predecessors are in the closed set, that vertex is added to the priority queue.

Only GO and CARRY nodes require actual path-planning from an initial location to a target location.[1] All other nodes in the operating schedule serve only as checkpoints, and can be automatically closed as soon as they become active. To avoid scheduling explicit waiting periods between GO nodes and COLLECT nodes, the planner extends the planning horizon of each GO node to the time step at which the assigned object become available. ISPS terminates when either (1) all schedule nodes are closed, (2) $\text{A}^\star_{SC}$ fails to find a feasible path, or (3) the cost of the route plan exceeds the upper bound cost maintained by NBS. ISPS is summarized in algorithm 2.

D. Level 4: Slack-and-Collision-aware Tie-breaking A*

Slack-and-Collision-aware Tie-breaking A* ($\text{A}^\star_{SC}$) combines a custom cascading search heuristic with the well-known A* graph-search algorithm [12]. As in regular space-time A*, $\text{A}^\star_{SC}$ maintains a priority queue of search states, where each search state corresponds to a path p. At every iteration, $\text{A}^\star_{SC}$ pops the lowest-cost search state from the queue and expands it by trying all possible one-step actions beginning from the terminal state of the corresponding path. When a search state satisfies the termination criteria (i.e., the path reaches the goal location), it is returned to $\text{A}^\star_{SC}$.

For brevity, we do not include the pseudocode of the entire $\text{A}^\star_{SC}$ algorithm. However, algorithm 3 shows the pseudocode of GET_HEURISTIC_COST(P, p, v), a subroutine that computes the four element heuristic cost $\mathbf{c} \in \mathbf{R}^4_+$ of a path p. This custom heuristic cost determines the order in which $\text{A}^\star_{SC}$ will expand search states within the algorithm. Ties are broken in cascading fashion. This cascaded tie-breaking behavior is the key to the efficiency of our path-planner in computing make-span optimal route plans while simultaneously avoiding conflicts.

The first element $\mathbf{c}_1$ of the cost tuple is the *delay* cost. It measures the amount by which a path is guaranteed to delay the entire project, which is 0 until the slack disappears. The second cost element $\mathbf{c}_2$ counts the number of times that a path conflicts with the global route plan P. The third element $\mathbf{c}_3$ is equal to the path length plus the remaining distance to the goal location, and the fourth element $\mathbf{c}_4$ is simply the distance to the goal location. Hence, the default behavior (based on $\mathbf{c}_3$ and $\mathbf{c}_4$) of $\text{A}^\star_{SC}$ is to move to the goal location as quickly, but it will avoid conflicts at the expense of path length until the slack runs out, at which time path length again become the dominant criterion for optimality. The only way that $\text{A}^\star_{SC}$ will return a path that creates conflicts is if it runs out of slack and there is no other path of equal or shorter length with fewer conflicts.

E. Repairing Route Plans

When ISPS has finished computing a consistent route plan for the entire operating schedule, the solution is checked for conflicts. If there are conflicts, ISPS tries to repair the "broken" solution by iterating through the operating schedule a second time. The difference the second time through is that a full route plan (as opposed to a partial route plan) is available to populate the conflict table which is used in $\text{A}^\star_{SC}$ for computing the conflict cost COUNT_CONFLICTS(p, P). This allows paths that are planned early in the topological ordering to be adjusted to avoid potential conflicts with plans that are planned later in the ISPS procedure.

[1] GO and CARRY are functionally equivalent–the path-planning process for them is identical. We simply call them different things to acknowledge that the agent is actually transporting an object during the CARRY phase.

Algorithm 3: GET_HEURISTIC_COST Subroutine of $\text{A}^\star_{SC}$

```
1: procedure GET_HEURISTIC_COST(p, P, v)
2:     h ← d(s, v.goal_state)                          ▷ "cost-to-go" heuristic
3:     c1 ← max(0, p.t^F + h − (v.t^F + v.SLACK))      ▷ delay cost
4:     c2 ← COUNT_CONFLICTS(p, P)                      ▷ number of conflicts along path
5:     c3 ← p.length + h                               ▷ lower bound total travel distance
6:     c4 ← h                                          ▷ minimum remaining travel time
7:     return c
```

F. Theoretical Properties

It can be shown that NBS will search the full space of possible valid project schedules in best-first order. If the combination of CBS, ISPS, and $\text{A}^\star_{SC}$ is optimal and complete, the full algorithm can also be guaranteed to find the optimal solution if the problem is feasible. Sharon, Stern, Felner, *et al.* have already shown that CBS is optimal and complete under the assumption that the lower level path planner is also optimal and complete [11].

However, our lower level path planner (ISPS combined with $\text{A}^\star_{SC}$) is incomplete because it is possible for $\text{A}^\star_{SC}$ to prematurely consume slack in one vertex such that a downstream vertex is forced to unnecessarily delay the project. Completeness could be ensured by removing the slack term from line 3 of algorithm 3, but this would negate the planner's key advantage: its ability to avoid conflicts by consuming slack. We are preparing an extension of ISPS and $\text{A}^\star_{SC}$ that will incrementally anneal the slack cost term when necessary, thus achieving completeness without ruining the planner's efficiency.

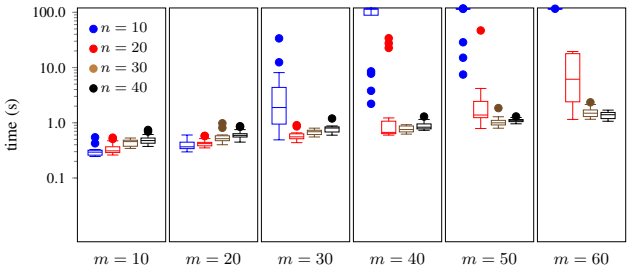
V. EXPERIMENTS

We evaluate our solver on a set of 384 randomly generated PC-TAPF problem instances—16 instances for every combination of number of robots $n \in \{10, 20, 30, 40\}$ and number of objects $m \in \{10, 20, 30, 40, 50, 60\}$. The grid world environment is shown in fig. 1. Initial robot locations are selected at random. The initial and destination locations of all objects are selected at random from the designated pick-up and drop-off zones surrounding each manufacturing station. For each problem instance, we evaluate the runtime of (a) the full algorithm, (b) the NBS MILP solver alone, and (c) a single iteration of ISPS. The results are summarized in figs. 4a to 4c.

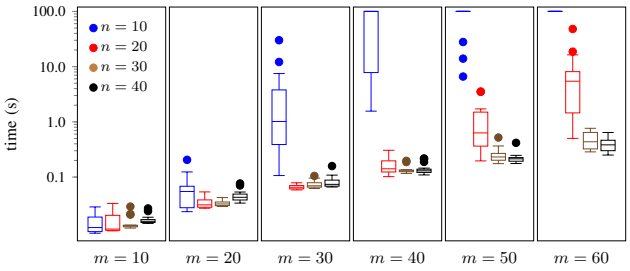
Figure 4a shows that the PC-TAPF solver runs very fast for most problem instances, even when m and n are large. However, NBS struggles when the ratio of tasks to robots becomes high. This is unsurprising, as the MILP approaches a traveling salesman problem when $m \gg n$. The NBS MILP solver timed out (at 100 seconds) on 40 of the problem instances (including all instances with $m = 60$,

$n = 10$) before finding an optimal solution. In these cases, the suboptimal MILP solution was still passed to CBS.

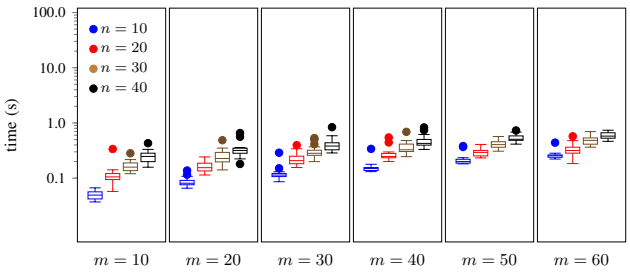
ISPS (as demonstrated by fig. 4c) scales gracefully with problem size. The route planner's efficiency is largely due to the ability of ISPS and $A^{\star}_{SC}$ to exploit slack in the operating schedule, thereby removing the burden of conflict-resolution from CBS. Figure 5 shows that only nine of the 384 problem instances required any CBS branching at all. On two problem instances, the solver reached our arbitrary limit of 100 branching operations in a single CBS iteration. Despite the incompleteness of ISPS and $A^{\star}_{SC}$, only three problem instances (besides those on which the solver reached a time or iteration limit) were not solved optimally.



(a) Runtime statistics for the full PC-TAPF solver.



(b) Runtime statistics for the task assignment MILP solver.



(c) Runtime statistics for ISPS.

Fig. 4: Box plots summarizing the computation time for the full PC-TAPF solver, the NBS MILP Solver and ISPS. Each individual plot represents 16 trials with the indicated values of m (labeled along the x-axis) and n (increasing from left-to-right and labeled by color). Runtime is plotted on a log scale, and all three figures are plotted on the same scale.

In addition to the experiments above, we include a video to demonstrate our solver's performance in a high-fidelity simulator with multiple non-holonomic differential drive robots. After solving the PC-TAPF problem, we use geometric path-planning and convex optimization to convert the optimal

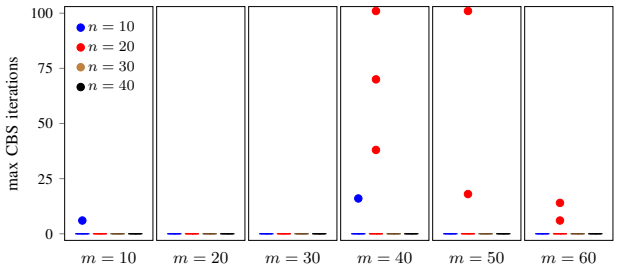


Fig. 5: Box plots summarizing the maximum number of CBS iterations in any full iteration of the PC-TAPF solver. Plots are organized as in figs. 4a to 4c. Only nine of the 384 problem instances required any CBS branching.

route plan into dynamically feasible trajectories that allow each robot to pass in and out of each grid cell in the planned sequence at precise intervals. To stabilize each robot about its reference trajectory, we implement a hybrid nonlinear feedback control policy that switches between control laws (including the tracking control law of Lee, Song, Lee, *et al.* [18]) depending on the current stage of the reference trajectory. Our simulator is built on Webots (`http://www.cyberbotics.com`).

VI. Conclusions

We introduced the PC-TAPF formulation and presented a hierarchical algorithm that takes advantage of slack in the operating schedule to efficiently and optimally solve many PC-TAPF problem instances. Though the algorithm performs well empirically, there are several weaknesses that need to be addressed. ISPS (and hence, the entire algorithm) is not complete. We are preparing an extension of ISPS that will make the full algorithm optimal and complete. Moreover, we have observed that the computational cost of NBS can grow intractably large for certain classes of problems. This makes it attractive to develop bounded-suboptimal PC-TAPF solvers with better runtime guarantees.

Real manufacturing applications are likely to include large sub-assemblies that are too big to be transported by a single robot. We are currently extending our approach to "collaborative transport" scenarios, in which some transport tasks must be handled by teams of robots. Another interesting avenue of future work would be to extend our work to a heterogeneous robot fleets, wherein robots might vary in size, shape, speed, and ability to service certain tasks.

We are extending our algorithm to a paradigm where projects arrive at the factory continuously, and wherein better performance may be attained by partial re-planning of the operating schedule and route plan to optimally balance multiple projects simultaneously. A related theme in our ongoing work is robustness to uncertainty. We are interested in incorporating open-loop robustness to potential failure and delay, as well as closed-loop robustness during execution of the PC-TAPF solution. Our algorithm currently assumes a deterministic environment, but several efforts from the MAPF and TAPF literature offer a promising starting point for incorporating robustness to uncertainty [19], [20].

REFERENCES

[1] J. Yu and S. M. LaValle, "Multi-agent path planning and network flow," in *Algorithmic Foundations of Robotics X*, 2013, pp. 157–173.

[2] G. A. Korsah, A. Stentz, and M. B. Dias, "A comprehensive taxonomy for multi-robot task allocation," *The International Journal of Robotics Research*, vol. 32, no. 12, pp. 1495–1512, 2013.

[3] W. Hönig, S. Kiesel, A. Tinka, J. W. Durham, and N. Ayanian, "Conflict-based search with optimal task assignment," in *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2018.

[4] G. Wagner, H. Choset, and N. Ayanian, "Subdimensional expansion and optimal task reassignment," *Symposium on Combinatorial Search*, pp. 177–178, 2012.

[5] H. Ma and S. Koenig, "Optimal target assignment and path finding for teams of agents," in *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2016.

[6] J. Yu and S. M. LaValle, "Optimal multi-robot path planning on graphs: structure and computational complexity," *ArXiv*, 2015.

[7] H. Ma, G. Wagner, A. Felner, J. Li, T. K. Satish Kumar, and S. Koenig, "Multi-agent path finding with deadlines," in *International Joint Conference on Artificial Intelligence (IJCAI)*, vol. July, 2018, pp. 417–423.

[8] A. Dohn, M. S. Rasmussen, and J. Larsen, "The vehicle routing problem with time windows and temporal dependencies," *Networks*, vol. 58, no. 4, pp. 273–289, 2011.

[9] D. Bredstrom and M. Ronnqvist, "A branch and price algorithm for the combined vehicle routing and scheduling problem with synchronization constraints," *SSRN Electronic Journal*, pp. 1–21, 2011.

[10] D. Bredström and M. Rönnqvist, "Combined vehicle routing and scheduling with temporal precedence and synchronization constraints," *European Journal of Operational Research*, vol. 191, no. 1, pp. 19–31, 2008.

[11] G. Sharon, R. Stern, A. Felner, and N. Sturtevant, "Conflict-based search for optimal multi-agent path finding," in *AAAI Conference on Artificial Intelligence (AAAI)*, 2012.

[12] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[13] D. Silver, "Cooperative pathnding.pdf," *Proceedings of the First AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pp. 117–122, 2005.

[14] H. Ma, T. K. Kumar, J. Li, and S. Koenig, "Lifelong multi-agent path finding for online pickup and delivery tasks," in *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS*, vol. 2, 2017, pp. 837–845.

[15] J. Yu and S. M. LaValle, "Optimal multirobot path planning on graphs: complete algorithms and effective heuristics," *IEEE Transactions on Robotics*, vol. 32, no. 5, pp. 1163–1177, 2016.

[16] S. Botelho and R. Alami, "Multi-robot cooperation through negotiated task allocation and achievement," *IEEE International Conference on Robotics and Automation (ICRA)*, no. May, pp. 1234–1239, 1999.

[17] L. L. C. Gurobi Optimization, *Gurobi optimizer reference manual*, 2019.

[18] T. C. Lee, K. T. Song, C. H. Lee, and C. C. Teng, "Tracking control of unicycle-modeled mobile robots using a saturation feedback controller," *IEEE Transactions on Control Systems Technology*, vol. 9, no. 2, pp. 305–318, 2001.

[19] W. Honig, S. Kiesel, A. Tinka, J. W. Durham, and N. Ayanian, "Persistent and robust execution of mapf schedules in warehouses," *IEEE Robotics and Automation Letters*, vol. 4, no. 2, pp. 1125–1131, 2019.

[20] H. Ma, W. Hönig, L. Cohen, T. Uras, H. Xu, T. K. Kumar, N. Ayanian, and S. Koenig, "Overview: a hierarchical framework for plan generation and execution in multirobot systems," *IEEE Intelligent Systems*, vol. 32, no. 6, pp. 6–12, 2017.