Published in the Proc. of IEEE/RSJ Int'l Conf. on Intelligent Robots and Systems, 2011, pp. 3268-3275

Efficient and Complete Centralized Multi-Robot Path Planning

Ryan Luna and Kostas E. Bekris

***Abstract*— Multi-robot path planning is abstracted as the problem of computing a set of non-colliding paths on a graph for multiple robots. A naive search of the composite search space, although complete, has exponential complexity and becomes computationally prohibitive for problems with just a few robots. This paper proposes an efficient and complete algorithm for solving a general class of multi-robot path planning problems, specifically those where there are at most n-2 robots in a connected graph of n vertices. This paper provides a full proof of completeness. The algorithm employs two primitives: "push", where a robot moves toward its goal until no progress can be made, and "swap", that allows two robots to swap positions without altering the position of any other robot. Additionally, this paper provides a smoothing procedure for improving solution quality. Simulated experiments compare the proposed approach with several other centralized and decoupled planners, and show that the proposed technique improves computation time and solution quality, while scaling to problems with 100s of robots, solving them in under 5 seconds.**

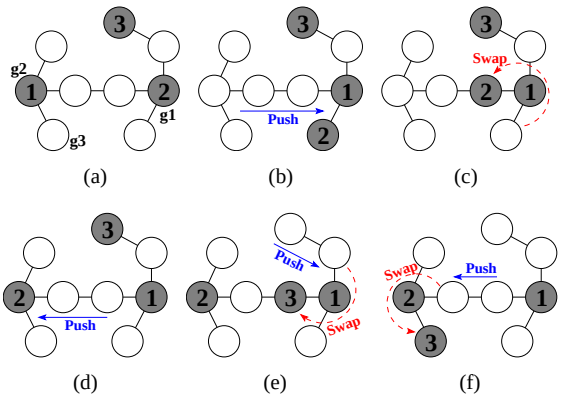


Fig. 1. A complete example of Push and Swap. (a) Start and goal configurations. (b) Robot 1 pushes to its goal, moving robot 2. (c) Robot 2 is blocked by 1. 2 swaps with 1. (d) Robot 2 pushes to its goal. (e) Robot 3 pushes and is blocked by 1. 3 swaps position with 1. (f) Robot 3 is blocked by 2. 3 swaps position with 2, and reaches its goal.

I. Introduction

Multi-robot path planning [1], [2] requires the computation of paths for multiple robots on a graph, where the robots must move from their start positions to unique goals while avoiding collisions. An efficient solution to this problem is relevant in many applications, such as warehouse management, intelligent transportation, (dis)assembly, autonomous mining, space exploration, as well as computer games.

A. Background

Traditional solutions to the multi-robot planning problem consider either a coupled or decoupled approach. In coupled techniques, the robots are considered a single composite system with many degrees of freedom, and the solution is found by searching the composite roadmap $G^n = G \times G \times \ldots \times G$, where G is the original graph and n is the number of robots. The coupled approach guarantees not only completeness, but optimality as well [3], [4]. These approaches, however, have exponential complexity in the number of robots. This complexity has inspired a number of approaches that attempt to prune the search space while maintaining completeness. One such method splits the multi-robot problem into a sequence of fully-coupled subcomponents where each subcomponent can be solved independently of all others [2]. A hybrid technique plans for each robot given future paths, and employs a coupled approach for choosing goals and avoiding deadlocks [5]. For graphs with specific topologies, efficient and complete approaches exist by searching a minimum spanning tree of the roadmap [1], or restricting the problem domain in grid-worlds (i.e., 4 or 8 connectivity) [6]. It is possible to take advantage of ad-hoc networks formed when robots are within communication range by sharing information and utilizing a coupled planner to compute trajectories for each connected component of the network [7].

In contrast, decoupled approaches compute individually optimal paths, and settle conflicts between the paths as they arise. These approaches compute sub-optimal solutions and are not usually complete. However, decoupled methods are typically able to compute solutions in times that are orders of magnitude faster than coupled planners, making them prevalent in the multi-robot literature. Prioritized planners compute paths sequentially for different robots in order of priority. Paths of high priority robots are considered moving obstacles that must be avoided by those of lower priority [8]. The choice of priorities has a significant impact on the solution quality [9], and searching the space of priorities can improve performance [10]. Another typical decoupled approach considers tuning the velocities of robots along the precomputed trajectories to avoid collisions [11], [12]. These approaches have evolved over time, and are now able to compute collision free paths for systems with dynamic constraints [13]. Decoupled approaches suffer from deadlocks, and a number of planners were created in order to reduce this. Techniques using incremental planning [14] or coordination graphs [15], [16] have shown to reduce these deadlocks.

A modern heuristic-search technique for solving multi-robot path planning considers a dynamic priority scheme for the robots during replanning, and windows the search

R. Luna and K. E. Bekris are with the Department of Computer Science and Engineering, University of Nevada, Reno, 1664 N. Virginia St., MS 171, Reno, NV, 89557 {rluna, bekris}@cse.unr.edu

in combination with a backwards A^* heuristic in order to improve search time and scalability [17]. This method was further improved with spatial abstraction for faster heuristic computation and lower memory requirements [18]. Other search techniques take advantage of the discrete space by creating a flow network within grid-worlds [19], or decompose the roadmap into subgraphs with specific properties [20].

B. Contribution

This paper proposes a new method for multi-robot path planning that is computationally efficient and complete for a very general class of problems, i.e., all instances where there are at most $n-2$ robots in a graph with n vertices.

The proposed method (Figure 1) is orders of magnitude faster when compared to traditional coupled A^*. In comparison to existing complete alternatives, the proposed method provides completeness for a much wider problem class. Furthermore, it makes no assumptions about the topology of the underlying graph [19], [6], [1]. Compared to a general decoupled algorithm [17], an efficient and complete centralized planner [1], and a coupled planner utilizing *optimal decoupling* [2], the proposed technique exhibits competitive solution times with no dependence on parameter selection. Empirical results show that the proposed approach consistently solves multi-robot path planning problems in times significantly faster than several existing coupled and decoupled approaches, and returns higher quality solutions.

The approach employs two basic primitives. The first primitive, "push", forces robots to clear a specified path for a robot to get to its goal. In harder instances, robots may be required to switch positions along their shortest paths. This is addressed by the second primitive called "swap". Once a robot cannot make progress towards its goal by pushing, it must swap positions with the next robot along its shortest path. This operation may force other robots to move in response; potentially all of them. Eventually, all the robots must be returned to their original positions, with just two robots swapping positions. This work shows that if it is not possible to execute a "swap" for two robots, then the problem is not solvable.

II. Setup and Notation

Consider a graph $\mathcal{G}(\mathcal{V},\mathcal{E})$ and n robots $\mathcal{R}$, where $n \leq |\mathcal{V}|-2$. An assignment $\mathcal{A} : [1,n] \rightarrow \mathcal{V}$ places the robots in unique vertices: $\forall i,j \in [1,n], j \neq i : \mathcal{A}[i] \in \mathcal{V},\ \mathcal{A}[i] \neq \mathcal{A}[j]$. The starting assignment is denoted as $\mathcal{S}$, and the goal assignment is denoted as $\mathcal{T}$. An action $\pi(\mathcal{A}_a,\mathcal{A}_b)$ is a change between two assignments $\mathcal{A}_a$ and $\mathcal{A}_b$ so that only one robot moves between neighboring vertices in the two assignments, i.e., $\exists\ i \in [1,n]$ and $\forall j \in [1,n], j \neq i$:

$$\mathcal{A}_a[i] \neq \mathcal{A}_b[i], (\mathcal{A}_a[i],\mathcal{A}_b[i]) \in \mathcal{E}, \mathcal{A}_a[j] = \mathcal{A}_b[j].$$

A path $\Pi = \{\mathcal{A}_0,\ldots,\mathcal{A}_k\}$ is a sequence of assignments, so that for any two consecutive assignments $\mathcal{A}_i$ and $\mathcal{A}_{i+1}$ in Π there is an action $\pi(\mathcal{A}_i,\mathcal{A}_{i+1})$. The objective of multi-robot path planning is to compute a solution $\Pi^* = \{\mathcal{S},\ldots,\mathcal{T}\}$, which is a sequence initiated with $\mathcal{S}$ and ending with $\mathcal{T}$.

III. Push and Swap

The PUSH_AND_SWAP method sets the current assignment $\mathcal{A}$ to the starting one $\mathcal{S}$ and starts building the solution path Π^* by inserting $\mathcal{S}$ into Π^* (line 1). It also initializes the set $\mathcal{U}$ to the empty set. This set will be used throughout this description, indicating a set of static vertices that are treated as obstacles. It contains vertices of robots that have reached their goals. Then for each robot r, PUSH_AND_SWAP tries first to push r to its goal $\mathcal{T}[r]$ by clearing its path from other robots (lines 2-5). If the push operation fails (line 5), then a swap operation is initiated with the robot that is blocking r's path (line 6). If the swap also fails, then the problem is not solvable (line 7). If r has not reached its goal, then the algorithm keeps applying push and swap operations (line 3). When r reaches its goal $\mathcal{T}[r]$, $\mathcal{A}[r]$ is inserted to the set $\mathcal{U}$ of static robots, whose position must be respected by all push operations (line 8). Eventually the algorithm returns Π^*, which is constructed by the calls to PUSH and SWAP.

Algorithm 1 PUSH_AND_SWAP ($\mathcal{G},\mathcal{R},\mathcal{S},\mathcal{T}$)

1: $\mathcal{A} \leftarrow \mathcal{S}$, $\Pi^* \leftarrow \{\mathcal{S}\}$, $\mathcal{U} \leftarrow \emptyset$
2: **for all** $r \in \mathcal{R}$ **do**
3: **while** $\mathcal{A}[r] \neq \mathcal{T}[r]$ **do**
4: $\mathtt{p}$ = SHORTEST_PATH $(\mathcal{G},\mathcal{A}[r],\mathcal{T}[r])$
5: **if** PUSH$(\Pi^*,\mathcal{G},\mathcal{A},r,\mathtt{p},\mathcal{U})$ == FALSE **then**
6: **if** SWAP$(\Pi^*,\mathcal{G},\mathcal{A},\mathcal{T},r,\mathcal{U})$ == FALSE **then**
7: **return** $\emptyset$ (i.e., Failure)
8: $\mathcal{U} \leftarrow \mathcal{U} \cup \mathcal{A}[r]$
9: **return** Π^* (i.e., Success)

A. Push Primitive

PUSH (Algorithm 2) attempts to move robot r along a given path $\mathtt{p}^*$. This operation will incrementally move any robot occupying a vertex in $\mathtt{p}^*$ away from this path, as long the occupied vertex doesn't belong to the set $\mathcal{U}$. A graphical example of the operation is shown in Figure 2.

Algorithm 2 PUSH($\Pi^*,\mathcal{G},\mathcal{A},r,\mathtt{p}^*,\mathcal{U}$)

1: t = last vertex in $\mathtt{p}^*$
2: $v \leftarrow$ vertex in $\mathtt{p}^*$ after $\mathcal{A}[r]$
3: **while** $\mathcal{A}[r] \neq t$ **do**
4: advance r along $\mathtt{p}^*$ until blocked, inserting intermediate actions into Π^*
5: **if** $\mathcal{A}[r] \neq t$ **then**
6: Mark $\mathcal{A}[r]$ and $\mathcal{U}$ as blocked on $\mathcal{G}$
7: $v_e \leftarrow$ reachable empty vertex to v on G
8: $\mathtt{p} \leftarrow$ SHORTEST_PATH$(\mathcal{G},v,v_e)$
9: **if** $\mathtt{p} == \emptyset$ **then** return FALSE
10: Mark $\mathcal{A}[r]$ and $\mathcal{U}$ as free on $\mathcal{G}$
11: move robots on $\mathtt{p}$ toward v_e; insert actions into Π^*
12: return TRUE

PUSH iterates as long as r has not reached the end of the given path (line 3), and it can still make progress without any need for swapping (line 9). If the vertices along $\mathtt{p}^*$ are not occupied, then r is moved along these vertices and the corresponding intermediate assignments are stored on the solution path (line 4). At this point, if r has traversed

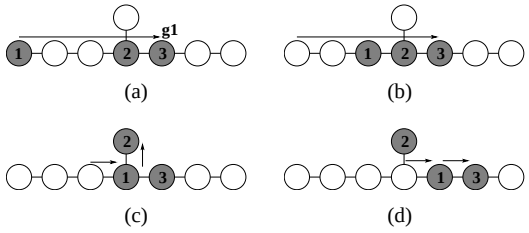


Fig. 2. Illustration of the push primitive. (a) Robot 1 advances along the empty vertices in its shortest path. (b) Robot 2 is pushed up, allowing 1 to advance. (c) Robot 3 is pushed forward, allowing robot 1 to reach its goal.

the entire path , the subproblem for r is solved (line 5). Otherwise, the next vertex v along $\mathtt{p}^*$ of r is occupied by another robot. In this case, the algorithm considers whether it is possible to push the robot blocking the path $\mathtt{p}^*$ out of the way of r, without altering the position of r or any of the robots in $\mathcal{U}$ (lines 6-11). To do this, PUSH computes the shortest path p between the robot occupying vertex v and the closest reachable empty vertex v_e to v on G. The vertex v_e is considered reachable if there exists a path between v and v_e that does not pass through any vertex in the set $\mathcal{U}$, or the vertex occupied by robot r (lines 6-8). If no path is found (line 9), then robot r cannot push the blocking robot out of its shortest path, and further progress cannot be made without swapping. If a path is found (line 10), then all of the robots along the shortest path between v and v_e are pushed one vertex forward towards v_e along p. In this way v is cleared for r to occupy. The pushing process is then repeated, given the new assignment of all robots.

B. Swap Primitive

SWAP switches the position of a robot r with the robot s adjacent to r along r's shortest path. After SWAP is finished, the only robots that have changed position are r and s.

Algorithm 3 SWAP ($\Pi^*, \mathcal{G}, \mathcal{A}, \mathcal{T}, r, \mathcal{U}$)

1: $\mathtt{p}^* \leftarrow$ SHORTEST_PATH ($\mathcal{G}, \mathcal{A}[r], \mathcal{T}[r]$)
2: $s \leftarrow$ robot on first vertex in $\mathtt{p}^*$ after $\mathcal{A}[r]$
3: success = FALSE
4: $\mathcal{S} \leftarrow$ {Vertices of degree ≥ 3, sorted by dist. from r}
5: **while** $\mathcal{S} \neq \emptyset$ **and** success == FALSE **do**
6: $v = \mathcal{S}$.POP (), $\Pi \leftarrow \emptyset$
7: p $\leftarrow$SHORTEST_PATH($\mathcal{G}, \mathcal{A}[r], v$)
8: **if** PUSH ($\Pi, \mathcal{G}, \mathcal{A}, \{r, s\}$, p, $\emptyset$) == TRUE **then**
9: **if** CLEAR ($\Pi, \mathcal{G}, \mathcal{A}, \mathcal{A}[r], \mathcal{A}[s]$) == TRUE **then**
10: success = TRUE
11: **if** success == FALSE **then** return FALSE
12: $\Pi^* = \Pi^* + \Pi$
13: EXECUTE_SWAP ($\Pi^*, \mathcal{G}, \mathcal{A}[r], \mathcal{A}[s]$)
14: $\Pi =$ Π.REVERSE (), exchanging paths for r and s
15: $\Pi^* = \Pi^* + \Pi$
16: **if** $\mathcal{T}[s] \in \mathcal{U}$ **then**
17: return RESOLVE ($\Pi^*, \mathcal{G}, \mathcal{A}, \mathcal{T}, \mathcal{U}, \mathtt{p}^*, r, s$)
18: **return** TRUE

To switch two robots, SWAP selects s, which is adjacent to r along r's shortest path (lines 1-2). For a vertex v in $\mathcal{G}$ with degree ≥ 3, the algorithm computes the shortest path from $\mathcal{A}[r]$ to v (line 7). Then, SWAP attempts to push r and s to v and one of its neighboring vertices (line 8). This is achieved by a call to the function PUSH, which will move the composite agent composed of adjacent agents r and s, that can also move all other agents indiscriminately; the set of static vertices passed to PUSH here is empty(line 8). If the pushing succeeds, then $\mathcal{A}[r] = v$ and $\mathcal{A}[s]$ is adjacent to v. For r and s to swap positions at v, two adjacent vertices of v (excluding the vertex occupied by s) must be evacuated (line 9). This is the objective of CLEAR, detailed later in this section. If two adjacent vertices of v cannot be cleared, then it is impossible for r and s to exchange positions at v, and another vertex must be checked.

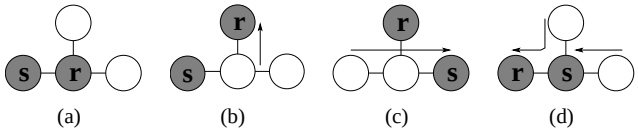


Fig. 3. Illustration of the swap primitive. Robots r and s swap positions.

If all vertices of degree ≥ 3 are exhausted and r and s cannot reach such a vertex with two empty neighbors, SWAP returns failure (line 11). If the swap can take place, the actions computed by MULTIPUSH and CLEAR are added to the global solution (line 12), and the swap is performed (line 13). The swap is shown in Figure 3. Once the swap is executed, the actions Π computed by MULTIPUSH and CLEAR must be reversed so that robots already at their goal will return there after the swap. Care must be taken during reversal not to undo the swap of r and s; paths executed by r will be executed by s and vice versa (line 14). The reversed set of actions is then added to Π^* (line 15). Finally, it may happen that s occupies a vertex in the set $\mathcal{U}$, indicating s was already at its goal (line 16). After the swap, r and s have switched positions, and s must be moved back to its goal to maintain the swap invariant. This is achieved by RESOLVE (line 17), detailed later in the section.

1) Clear Operation: The CLEAR operation (Algorithm 4) attempts to free two neighbors of a vertex v so that robots r and s can swap positions with one another. There are two cases to consider when freeing the neighborhood of v.

If there are already two empty neighbor vertices of v, then the clearing is trivially achieved (lines 1-2). Otherwise, PUSH is used to evacuate the neighbors. During the first step, CLEAR attempts to push all occupied neighbor vertices of v to other neighbors excluding v (Figure 4(a), lines 5-9). If two neighbors of v are freed during this process, CLEAR succeeds (line 8). Otherwise, if all neighbors of v are exhausted but there exists a single empty neighbor of v (Figure 4(b)), the algorithm considers moving an occupied neighbor of v through v and an empty neighbor where it may be possible to push further. Only if there is one empty neighbor is this feasible; note that if it is not possible to push a single robot away from v, it will not be possible to push a second one. To

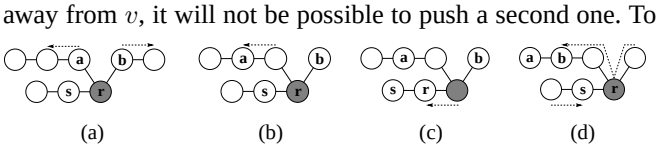


Fig. 4. CLEAR operation at vertex v (shaded). (a) Case 1. (b-d) Case 2.

achieve this, the vertex v itself must be cleared (Figure 4(c), lines 10-11). Then all occupied neighbors of v are checked to see if it is possible to clear them via an empty vertex of v (Figure 4(d), lines 14-17). If a single push succeeds, then two neighbors have been successfully cleared. CLEAR can then move the robots formerly occupying v and its neighbor back to their original positions and return success (line 18). If both steps 1 and 2 fail, then it is not possible to clear vertex v without additional swapping. Section IV details why these two cases are sufficient for clearing the neighborhood of v.

Algorithm 4 CLEAR ($\Pi^*, \mathcal{G}, \mathcal{A}, v, v'$)

1: $\mathcal{E} \leftarrow \{$free neighbors of $v\}$, $\mathcal{U} \leftarrow \{v, v', \mathcal{E}\}$
2: **if** $|\mathcal{E}| \geq 2$ **then** return TRUE
3: **for all** $n \in$ NEIGHBORS $(v) \setminus \{\mathcal{E}, v'\}$ **do**
4: **for all** $n' \in$ NEIGHBORS $(n) \setminus \{\mathcal{E}, v\}$ **do**
5: p $\leftarrow$SHORTEST_PATH $(\mathcal{G}, n, n')$
6: **if** PUSH$(\Pi^*, \mathcal{G},$ ROBOT$(\mathcal{G}, n)$, p, $\mathcal{U})$ **then**
7: $\mathcal{E} \leftarrow \mathcal{E} \cup \{n\}$
8: **if** $|\mathcal{E}|$ == 2 **then** return TRUE
9: **else if** $|\mathcal{E}|$ == 1 **then** break
10: p $\leftarrow$SHORTEST_PATH $(\mathcal{G}, v, v')$
11: **if** PUSH$(\Pi^*, \mathcal{G},$ ROBOT$(\mathcal{G}, v)$, p, $\mathcal{E})$ **then**
12: v'' = vertex held by robot that formerly held v'
13: $\mathcal{U} \leftarrow \{v, v', v'', \mathcal{E}\}$
14: **for all** $n \in$ NEIGHBORS $(v) \setminus \{\mathcal{E}, v'\}$ **do**
15: **for all** $n' \in$ NEIGHBORS $(\mathcal{E}) \setminus v$ **do**
16: p $\leftarrow$SHORTEST_PATH $(\mathcal{G}, n, n')$
17: **if** PUSH$(\Pi^*, \mathcal{G},$ ROBOT$(\mathcal{G}, n)$, p, $\mathcal{U} \cup \{n\})$ **then**
18: move ROBOT$(\mathcal{G}, v')$to v, ROBOT$(\mathcal{G}, v'')$to v'
19: return TRUE
20: return FALSE

2) Resolve Operation: The RESOLVE operation repairs the case in SWAP where a robot switches positions with another robot already at its goal. RESOLVE returns the robot s to its goal, while allowing r to make further progress.

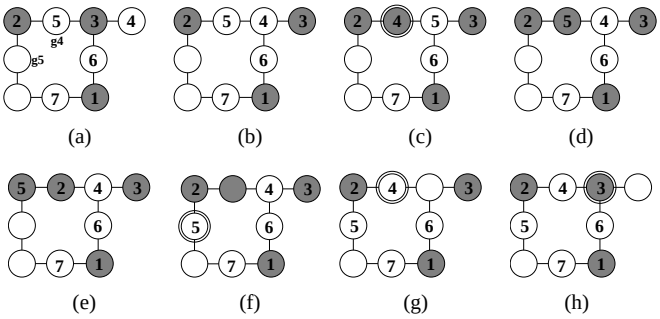


Fig. 5. RESOLVE: Vertices in $\mathcal{U}'$ are shaded. (a) After robots 1,2 and 3 have planned. (b) Robot 4 swaps with 3. (c) Resolve is invoked (3 was at goal). 4 swaps with 5 and reaches its goal. (d) Resolve called for 3 and 5; 5 swaps with 4. (e) 5 swaps with 2 invoking Resolve again. (f) 5 pushes to its goal; third Resolve is successful. (g) 4's goal is freed; second Resolve is successful. (h) 3's goal is free; first Resolve is complete.

RESOLVE first attempts to push robot r further along its shortest path (lines 2-5). If the push succeeds, then the goal of s will be free, allowing s to move to it. If this push fails, then r will have to swap positions again along its shortest path (line 8). After a successful swap, robot s can attempt to push from its current position to its goal (line 9). r continues to swap along its shortest path until the push for s succeeds. Under some circumstances, r may swap to its goal. In this case, r has swapped with a robot r' that was occupying its goal; r' must then continue the resolution process in order to free the goal of s. This is achieved with a recursive call to RESOLVE, replacing r with r' (lines 11-15).

Algorithm 5 RESOLVE ($\Pi^*, \mathcal{G}, \mathcal{A}, \mathcal{T}, \mathcal{U}, \mathrm{p}^*, r, s$)

1: t = vertex in p^* after $\mathcal{A}[r]$, $\mathcal{U}' \leftarrow \{\mathcal{U} \cup \{\mathcal{A}[s]\}\} \setminus \{\mathcal{T}[s]\}$
2: p =SHORTEST_PATH $(\mathcal{G}, \mathcal{A}[r], t)$
3: **if** PUSH $(\Pi^*, \mathcal{G}, \mathcal{A}, r,$ p$, \mathcal{U}')$ **then**
4: move s from $\mathcal{A}[s]$ to $\mathcal{T}[s]$
5: return TRUE
6: **else**
7: $r' = r$, $p^s \leftarrow \{\mathcal{A}[s], \mathcal{T}[s]\}$
8: **while** SWAP $(\Pi^*, \mathcal{G}, \mathcal{A}, \mathcal{T}, r', \mathcal{U}')$ **do**
9: **if** $\mathcal{A}[s]$ == $\mathcal{T}[s]$ **or** PUSH $(\Pi^*, \mathcal{G}, \mathcal{A}, s, p^s, \mathcal{U}')$ **then**
10: return TRUE
11: **else if** $\mathcal{A}[r']$ == $\mathcal{T}[r']$ **then**
12: $\mathcal{U}' \leftarrow \mathcal{U}' \cup \{\mathcal{T}[\mathrm{r}']\}$
13: r' = robot r' just swapped with
14: p^* =SHORTEST_PATH $(\mathcal{G}, \mathcal{A}[r'], \mathcal{T}[r'])$
15: return RESOLVE $(\Pi^*, \mathcal{G}, \mathcal{A}, \mathcal{T}, \mathcal{U}', \mathrm{p}^*, r', s)$
16: return FALSE

C. Post Processing

Post processing the sequence Π^* PUSH_AND_SWAP can yield significant improvements in path quality. Π^* may have redundant paths due to multiple calls to SWAP and subsequent reversals required for the swap invariant. In SMOOTH (algorithm 6), if a robot leaves a vertex v at step t, and returns at $t+j$, and no other robot occupies v during $(t, t+j)$, then the robot is free to remain at v during $(t, t+j)$.

Algorithm 6 SMOOTH (Π)

1: removed = TRUE
2: **while** removed == TRUE **do**
3: removed = FALSE
4: **for all** $\pi \in \Pi$.REVERSE () **do**
5: $r =$ ROBOT(π), $v =$ last vertex in π
6: $\pi' \leftarrow$next path in Π.REVERSE () containing v
7: **if** $\pi' \neq \emptyset$ **and** $r ==$ROBOT(π') **then**
8: **for all** $\pi'' \in \Pi(\pi', \pi]$ **do**
9: **if** ROBOT $(\pi'') == r$ **then**
10: **remove** (π'') from Π
11: **remove** portion of π' after v
12: removed = TRUE
13: return Π

SMOOTH accepts a solution path Π, and iterates over each action $\pi \in \Pi$ in reverse order (line 4). For each action π, the actions after π in the reversed sequence are checked for an occurrence of the final vertex v in π. If such an action π' exists, and π and π' are executed by the same robot r (lines 7-8), then all actions executed by r between π' and π' (including π) are removed from Π (lines 9-11). Additionally, π' is cut to end at vertex v (line 12). The smoothing process

continues until the entire solution sequence has been iterated through. SMOOTH continues to iterate over Π in reverse order until no paths are removed during an entire iteration.

IV. Analysis

This section proves the completeness of PUSH_AND_SWAP for instances where the number of robots is $\leq |\mathcal{V}| - 2$.

Theorem 4.1: PUSH_AND_SWAP is complete for multi-robot path planning problems where the number of robots n is less than or equal to $|\mathcal{V}| - 2$.

To prove this theorem, it must be shown that if two adjacent robots cannot swap vertices, then the planning problem itself has no solution. This implies that SWAP must be able to bring two agents to a degree of 3 or more and free two neighboring vertices in each solvable instance. Additionally, it must be shown that progress is always made when applying PUSH or SWAP and that a solvable instance will never become unsolvable with those primitives.

Lemma 4.2: PUSH can transfer a composite robot R made up of two adjacent robots to a vertex v in $\mathcal{G}$ if such a transfer is possible and necessary for SWAP.

Proof: Consider a composite robot R, a destination v for R on $\mathcal{G}$ and $\mathtt{p}$ a path from R to v. If there exists an alternate path $\mathtt{p}'$ from v to R, not passing through an internal vertex of $\mathtt{p}$, then v belongs to a loop that includes R. If v belongs to such a loop, and this loop contains at least an empty vertex or there is an empty vertex reachable from the loop, then R can simply PUSH the robots along its shortest path around the loop to reach v.

Otherwise if v does not belong to a loop with R, but there exists a single path $\mathtt{p}$ from the initial position of R to v, then let ρ be the number of vertices reachable from v without passing through any internal vertex of $\mathtt{p}$, and η be the number of robots along $\mathtt{p}$. If $\rho \geq \eta$, then it is possible to push all robots blocking $\mathtt{p}$ into vertices reachable by v. This completely frees $\mathtt{p}$ for R, allowing the composite robot to reach v. Otherwise, if $\rho < \eta$ then it is not possible for R to reach v; there are no free vertices to push the robots in $\mathtt{p}$.

If there are multiple paths from the initial position of R to v, then there must be at least one internal loop inside $\mathcal{G}$ that intersects a path from R to v. At this intersection, a vertex v' of degree 3 or more exists. For SWAP, this vertex is closer than the given destination vertex v, and will be expanded first. In this case, v' either belongs to a loop containing R, or a single path exists from the initial position of R to v', both of which are discussed earlier in this proof. Figure 6 shows this case. ■

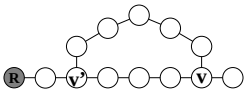


Fig. 6. A redundant loop from R's vertex to v. Using PUSH to move R to v is unnecessary for SWAP (v' is closer than v).

Lemma 4.3: CLEAR considers all essential cases when evacuating two vertices in the neighborhood of a vertex v for the purposes of swapping robots r and s at v. If CLEAR fails, then freeing v's neighborhood for SWAP is not possible.

Proof: There are 3 cases to consider when evacuating a robot a in the neighborhood of v, (Figure 7): 1) Push a toward a neighbor vertex that is not v or an empty neighbor of v. 2) Push a through v and an empty neighbor of v: v'. Robot a should end at a neighbor of v', excluding v or an empty neighbor of v. 3) Swap a with robots r and s in order to occupy a vertex opposite r and s.

PUSH can be used in case 1 by fixing the positions of r, s, and any free neighbor of v since PUSH exhaustively searches for reachable empty positions. Case 2 can also utilize PUSH, once v itself is cleared by having robot r push toward s one vertex, placing any free neighbor of v in $\mathcal{U}$. After v is free, a can be moved through v to the empty neighbor to attempt a PUSH. For this PUSH the set $\mathcal{U}$ is populated with the vertices occupied by robots r and s, and the vertex v.

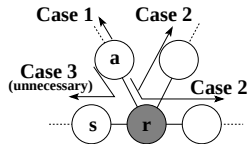


Fig. 7. Evacuating a from the neighborhood of a vertex v (shaded).

Case 3 is an extension of case 2, where robot a attempts to evacuate the neighborhood of v through the vertices occupied by robots r and s. This evacuation, however, requires a to swap position with r and s. If it is possible to swap a with r and s at v, then there must be two free vertices in the neighborhood of v, and it is not necessary to evacuate a. Therefore, for a to swap with r and s, a second swap vertex, $\dot{v}$ must be used. Note, if it is possible for a to swap with r and s at $\dot{v}$, then r and s could also swap at $\dot{v}$. Similarly, it may be possible for a to swap with r at v, and then swap with s at $\dot{v}$. If it is feasible for a and s to swap at $\dot{v}$, it is also possible for r and s to swap at $\dot{v}$. Because SWAP searches all possible vertices of degree 3 or more, $\dot{v}$ will be checked, making case 3 unnecessary. ■

Lemma 4.4: A multi-robot path planning problem is solvable if and only if SWAP can bring robots r and s to a vertex v with a degree ≥ 3 along with two empty vertices.

Proof: Consider the sequence of vertices along the shortest path of r to its goal $\mathcal{T}[r]$, where s is positioned between r and $\mathcal{T}[r]$. The ordering of r and s along this string of vertices has to be swapped in order for the problem to be solved. Even if r follows a different path to reach its goal $\mathcal{T}[r]$, the ordering of r and s along the shortest path will change if the problem is solvable. Thus, the problem is solvable if and only if the two robots can be swapped.

SWAP exhaustively searches all the vertices of degree 3 or more, checking whether it is feasible for adjacent robots r and s to reach a vertex using PUSH, apply CLEAR at the vertex, and execute a swap. From lemmas 4.2 and **??**, if SWAP is unable to move two adjacent robots to a vertex v of degree three or more to execute a swap operation, then the multi-robot path planning problem is not solvable. ■

Lemma 4.5: After a successful call to SWAP, the robots in $\mathcal{U}$ are assigned to the same vertices they were assigned before the call to SWAP, and at least one robot outside of $\mathcal{U}$ has made progress toward its goal.

Proof: SWAP attempts to switch the positions of a robot r not in $\mathcal{U}$ with the robot s blocking r's shortest path. Assuming a swap is possible, the swap will result with r and s switching positions, leaving all other robots intact. The other robots are guaranteed to be at the vertices they started in because the actions computed during SWAP is reversed.

Because r is swapped along its shortest path, it will make progress toward its goal. In the case that s belongs to $\mathcal{U}$, RESOLVE is employed to progress r further along its path to free the goal of s. Because RESOLVE is recursive, more than one robot may make progress towards its goal. ■

With these lemmas, the proof of Theorem 4.1 is as follows:

Proof: When planning for a robot r, the algorithm first attempts to move r towards its goal by pushing robots not in $\mathcal{U}$ away from its shortest path. $\mathcal{U}$ contains the vertices of robots that have previously reached their goals. If PUSH succeeds, then r reaches its goal. However, if r cannot make progress using PUSH, then there exists a robot s adjacent to r along r's shortest path that must be swapped with r.

To swap robots r and s, it is necessary to move both to a vertex v with degree ≥ 3, and clear two vertices neighboring v so that a swap (Figure 3) can be performed. SWAP requires two empty vertices in the neighborhood of v (achieved by CLEAR), and forms the basis for the constraint that at least two empty vertices must exist in $\mathcal{G}$ for completeness. SWAP performs these steps, and from Lemma 4.4, if a SWAP cannot be executed, then the problem cannot be solved.

Since PUSH never moves robots already at their goals, and Lemma 4.5 shows that SWAP allows r to make progress without moving any robot already at its goal, repeated calls to PUSH and SWAP will eventually bring robot r to its goal position, leaving those already at their goal intact. ■

Corollary 4.6: A robot r' may be pushed away from its initial vertex during the planning of other robots. Because of this displacement, it may seem that there is no path for r' to travel from its current vertex to its goal. Note, however, that it is possible for r' to get from its initial vertex to the current vertex using a series of PUSH and SWAP operations, and a similar set of operations will allow r' to return to its initial vertex. If the initial configuration is solvable, then it is possible for r' to reach its goal. The solvability of an instance depends only on the initial configuration; any configuration achieved through PUSH_AND_SWAP can be reversed.

V. EVALUATION

This section evaluates PUSH_AND_SWAP, and compares its performance against:

a) A coupled A^* planner which considers the fully composite robot when planning, and expands actions for a single robot at a time. This change allows the solution computed to be directly compared to PUSH_AND_SWAP, which also returns a sequential set of actions.

b) WHCA* [17], a modern decoupled planner that considers a planning window where a prioritized search takes place. This approach periodically changes the priority of each robot to avoid worst-case assignments. It also employs a backwards A^* heuristic to select the path that "best" completes the planning window during each replanning cycle.

c) A complete planner that considers movement through a spanning tree of $\mathcal{G}$ [1]. This planner is fast, deterministic, and complete for problems where the number of leaves in the spanning tree is greater than the number of robots.

	Coupled A^*		WHCA*(5)		Sp. Tree		Push/Swap	
Problem	Time	Size	Time	Size	Time	Size	Time	Size
Tree	2.54	15	1.18	34.4	1.56	15	0.42	18
Corners	3882	36	0.90	36	∞	n/a	0.88	50
Tunnel	413	53	∞	n/a	∞	n/a	2.68	81
String	198	20	1.46	36.1	1.97	38	0.42	26
Cycles	305	19	1.14	30.5	∞	n/a	0.53	34
Loop	∞	n/a	∞	n/a	∞	n/a	8.45	350
Connect	∞	n/a	∞	n/a	∞	n/a	2.63	86

TABLE I

THE COMPUTATION TIME (MS) AND SOLUTION LENGTH FOR THE BENCHMARKS. ∞ REPRESENTS A FAILURE TO COMPUTE A SOLUTION.

d) A technique that separates teams of robots into a sequences of fully coupled sub-components [2]. Each sub-component can be solved independently of the others by treating the current positions of all other robots as obstacles.

To evaluate the proposed approach, a series of challenging instances of multi-robot path planning were created. These problems include a set of small benchmarks as well as larger instances. All experiments are performed on a Core 2 Duo 2.5GHz machine with 4GB of memory. Results are given in terms of solution quality, computational feasibility, and algorithmic scalability. All computations for PUSH_AND_SWAP include the path smoothing process shown in section III-C.

A. Benchmark Problems

A series of small, benchmark problems (Figure 8) were created, ranging in size from 3 to 16 robots. Because of the small-scale of these problems, a comparison of the PUSH_AND_SWAP technique with the coupled A* as well as the decoupled WHCA* can be shown. The spanning tree planner is also applicable in some problems because number of leaves in the tree is greater than the number of robots.

Computation Time: Table I shows the time needed for the four approaches to compute their respective solutions. Times of more than 10 minutes are deemed a failure. The coupled approach computes high quality solutions for small problems, but becomes infeasible as the number of robots grows. WHCA* quickly computes solutions for 3 of the benchmarks, but the success rate for those problems is less than 50%. WHCA*'s inability to solve other problems can be attributed to a high degree of coordination necessary to compute the solution. The spanning tree planner can solve certain instances, but it has difficulty addressing these benchmarks because there are too few leaves in the spanning tree. In contrast, PUSH_AND_SWAP approach is able to quickly compute solutions to all of the benchmark problems.

Path quality: The solutions returned by PUSH_AND_SWAP and WHCA* are sub-optimal. WHCA* does poorly in the *tree* and *string* problems, computing solutions 2.5 times longer than the optimal. Only in the *corners* problem is the solution competitive with the other techniques. The spanning tree approach does well in the *tree* benchmark, but as the size of the environment increases, the quality of the solution decays considerably. The spanning tree approach had the poorest quality in the *string* benchmark. PUSH_AND_SWAP can solve all problems in single milliseconds, while achieving a solution comparable to the other suboptimal approaches.

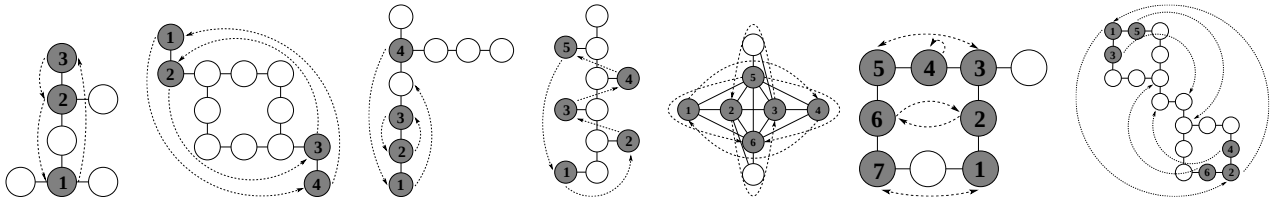


Fig. 8. A set of small benchmark problems. From left: Tree, Corners, Tunnel, String, Cycles, Loop, Connect. Arrows indicate desired goal positions.

	WHCA*(5)		Push and Swap	
Robots	Time (s)	Path Length	Time (s)	Path Length
24	0.011	1.86	0.001	2.00
48	0.152	2.00	0.009	2.00
72	0.909	2.10	0.029	2.00
96	3.56	2.07	0.067	2.00

TABLE II

ROTATION PROBLEM: TIME (SECONDS) AND PATH LENGTH.

B. Large Scale Problems

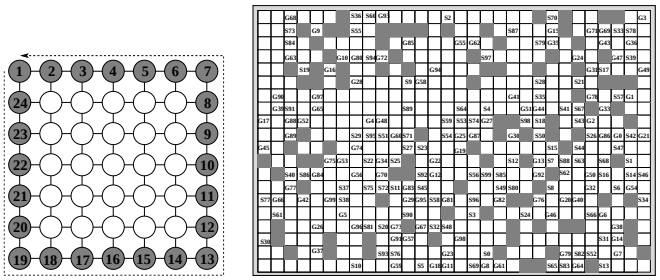
As the number of robots grows, the coupled approach becomes intractable, and decoupled planners suffer from deadlocks due to incompleteness.

Rotation Problem: The first large scale experiments test an environment with the robots arranged in a circular pattern, with the interior of the environment free (Figure 9(a)). The goal for each robot is adjacent to the start, with the final result achieving a rotation of all robots by one vertex. The spanning tree planner is not applicable because the graph does not generate enough leaves in a spanning tree. The configuration of the robots is *fully-coupled* and cannot be separated into smaller subsets, making the coupled A^* planner infeasible even for the smallest problem with 24 robots.

Table II shows the computation time for WHCA* with a window size of 5 compared with PUSH_AND_SWAP. PUSH_AND_SWAP computes virtually the same quality solutions as WHCA*, but in substantially less time. All experiments are averages over 20 runs. There was at least one failure for each WHCA* experiment due to randomness.

Random Problem: The next large-scale experiments evaluate the Coupled A^*, WHCA*, Spanning Tree, and PUSH_AND_SWAP techniques over a randomly populated grid world with 20% obstacle coverage. Robots are placed in random, mutually exclusive start and goal vertices. Figure 9(b) shows an example of this experiment with 100 robots. Similar to the *rotation* problem, the direct application of the coupled A^* is computationally infeasible. However, the random placement of the robots allows the problem to be split into a sequence of optimally decoupled composite robots [2]. Results given in this experiment for the coupled A^* approach include costs for optimal decoupling [2].

Computation time: Figure 10 (left) shows the time needed to compute solutions to the random problem with varying numbers of robots. WHCA* was executed with two window sizes, 8 and 16. PUSH_AND_SWAP shows a slight computational advantage over WHCA* with the window size of 8. However, the small window quickly degrades in its ability to compute a solution, and fails with 50 or more robots. WHCA* (16) is able to solve much larger numbers of robots, but suffers from the same decay as the number of robots



(a) (b)

Fig. 9. Large-scale experiments: (a) Rotation problem with 24 robots. (b) Random assignment of 100 robots. Robot i moves from Si to Gi.

increases. Additionally, the larger window size needs much more computation time due to the larger planning horizon.

When using the Coupled A^* planner, optimal decoupling [2] significantly improves computability by separating the problem into a sequence of largely singular composite robots. Even so, the optimal decoupling process has exponential complexity in the size of the roadmap and number of robots. This approach begins to decay after 50 robots, requiring substantial amounts of memory and computation time. Results for coupled A^* are given for problems with up to 50 robots.

The spanning tree planner is able to effectively operate in the random environment, solving the 100 robot instance in under 11 seconds. However, this time is still more than double the time of PUSH_AND_SWAP, which solves the 100 robot instance in about 4.5 seconds. PUSH_AND_SWAP was able to solve all instances of the random assignment problem in times faster than the other planners tested against.

It is important to note that WHCA* is not a complete algorithm. Figure 10 (middle) shows the percentage of successful random experiments for WHCA* compared to PUSH_AND_SWAP. The spanning tree technique is complete, and also has a 100% success rate. Optimal decoupling [2] has 100% success as well, but suffers from computational infeasibility after 50 robots.

Path Quality: PUSH_AND_SWAP achieves a solution quality that is noticeably better than the WHCA* and spanning tree approaches. Figure 10 (right) shows the ratio of the solution lengths for the various planners against the paths computed by PUSH_AND_SWAP. This graph shows that PUSH_AND_SWAP consistently achieves an average solution length 20% shorter than WHCA*, and more than three times shorter than the spanning tree approach.

With optimal decoupling, [2] the coupled A^* approach is not expected to compute the optimal solution for the entire problem, but each composite robot will yield its individually optimal solution. Surprisingly, using this approach does not

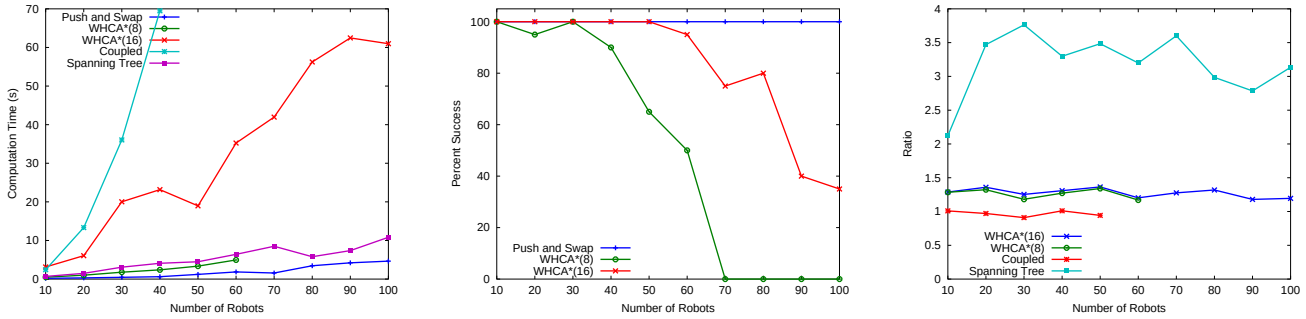


Fig. 10. Random grid experiment. All values are averages of 20 runs. (left) Computation time for varying numbers of robots using different techniques. (middle) Success rate for WHCA*. (right) The ratio of solution lengths for various techniques against those computed by PUSH_AND_SWAP.

significantly improve solution quality. The PUSH_AND_SWAP approach computes solutions just 3% longer.

VI. Discussion

This paper presented an efficient and complete approach for multi-robot path planning problems with at least two empty vertices in the graph. Through the combination of two basic primitives, the algorithm solves a broad set of problems at least as fast as a well established decoupled planner and a sophisticated complete approach without relying on parameter selection or graph topology. Experiments verify the advantages of the proposed technique against both coupled and decoupled approaches, showing improvements in computation time as well as solution quality. In comparison with a coupled A^* alternative combined with *optimal decoupling* [2], an efficient and complete approach [1], and a general decoupled method [17], PUSH_AND_SWAP exhibits computation times faster than all of these approaches, with path quality comparable to the optimal coupled A^* solution.

The proposed algorithm can potentially be extended to solve problems where there is only a single empty vertex by taking advantage of redundant loops in the graph (e.g., the 15-puzzle problem). This requires an extension to the swap primitive to use redundant loops while maintaining the invariant for all robots not involved in the swap.

The PUSH_AND_SWAP approach computes only a sequential solution and does not provide an optimal solution, but the solutions computed are comparable to a coupled planner. A significant extension of this work involves computing an optimal solution in terms of the total number of moves. It is interesting to investigate if an optimal solution can be achieved at a competitive computational cost. There are, however, competing notions of path optimality in multi-robot path planning, such as the sum of the path costs for all robots, or ideas related to Pareto optimality [21] which can be used to evaluate the quality of a solution.

References

[1] M. Peasgood, C. Clark, and J. McPhee, "A complete and scalable strategy for coordinating multiple robots within roadmaps," *IEEE Transactions on Robotics*, vol. 24, no. 2, pp. 282–292, 2008.

[2] J. van den Berg, J. Snoeyink, M. Lin, and D. Manocha, "Centralized path planning for multiple robots: Optimal decoupling into sequential plans," in *Robotics: Science and Systems V*, 2009.

[3] J.-C. Latombe, *Robot Motion Planning*. Boston, MA: Kluwer Academic Publishers, 1991.

[4] S. M. LaValle, *Planning Algorithms*. Cambridge, 2006.

[5] S. Qutub, R. Alami, and F. Ingrand, "How to solve deadlock situations within the plan-merging paradigm for multi-robot cooperation," in *Proc. of the Inter. Conf. on Intelligent Robots and Systems (IROS)*, vol. 3, 1997, pp. 1610–1615.

[6] K.-H. C. Wang and A. Botea, "Tractable Multi-Agent Path Planning on Grid Maps," in *Proceedings of the International Joint Conference on Artificial Intelligence IJCAI-09*, Pasadena, CA, USA, 2009, pp. 1870–1875.

[7] C. Clark, S. Rock, and J.-C. Latombe, "Motion planning for multiple robot systems using dynamic networks," in *Proc. IEEE Int. Conf. on Rob. and Autom. (ICRA)*, 2003, pp. 4222–4227.

[8] M. Erdmann and T. Lozano-Perez, "On multiple moving objects," in *IEEE Intern. Conference on Robotics and Automation (ICRA)*, 1986, pp. 1419–1424.

[9] J. van den Berg and M. Overmars, "Prioritized motion planning for multiple robots," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2005, pp. 2217–2222.

[10] M. Bennewitz, W. Burgard, and S. Thrun, "Finding and optimizing solvable priority schemes for decoupled path planning for mobile robots," *Robotics and Autonomous Systems*, vol. 41, no. 2, pp. 89–99, 2002.

[11] K. Kant and S. Zucker, "Towards efficient trajectory planning: The path-velocity decomposition," *International Journal of Robotics Research (IJRR)*, vol. 5, no. 3, pp. 72–89, 1986.

[12] P. O'Donnell and T. Lozano-Perez, "Deadlock-free and collision-free coordination of two robot manipulators," in *IEEE Int. Conf. Robotics and Automation (ICRA)*, 1989, pp. 484–489.

[13] J. Peng and S. Akella, "Coordinating multiple robots with kinodynamic constraints along specified paths," *Int. Journal of Robotics Research*, vol. 24, no. 4, pp. 295–310, 2005.

[14] M. Saha and P. Isto, "Multi-robot motion planning by incremental coordination," in *IEEE/RSJ Int'l Conference on Intelligent Robots and Systems (IROS)*, 2006, pp. 5960–5963.

[15] Y. Li, K. Gupta, and S. Payandeh, "Motion planning of multiple agents in virtual environments using coordination graphs," in *IEEE Int. Conf. Robotics and Automation (ICRA)*, 2005, pp. 378–383.

[16] K. E. Bekris, K. I. Tsianos, and L. E. Kavraki, "A decentralized planner that guarantees the safety of communicating vehicles with complex dynamics that replan online," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2007, pp. 3784–3790.

[17] D. Silver, "Cooperative pathfinding," in *The 1st Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE'05)*, 2005, pp. 23–28.

[18] N. Sturtevant and M. Buro, "Improving collaborative pathfinding using map abstraction," in *The Second Artificial Intelligence for Interactive Digital Entertainment Conference (AIIDE'06)*, 2006, pp. 80–85.

[19] K.-H. C. Wang and A. Botea, "Fast and Memory-Efficient Multi-Agent Pathfinding," in *International Conference on Automated Planning and Scheduling (ICAPS)*, Sydney, Australia, 2008, pp. 380–387.

[20] M. R. K. Ryan, "Graph decomposition for efficient multi-robot path planning," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 2007, pp. 2003–2008.

[21] R. Ghrist, J. M. O'Kane, and S. M. LaValle, "Pareto optimal coordination on roadmaps." in *Workshop on the Algorithmic Foundations of Robotics (WAFR)*, 2004.