Multi-Agent Pathfinding with Simultaneous Execution of Single-Agent Primitives

Qandeel Sajid
University of Nevada, Reno
Reno, NV 89557 USA
sajid.qandeel@gmail.com

Ryan Luna
Rice University
Houston, TX 77005 USA
rluna@rice.edu

Kostas E. Bekris
University of Nevada, Reno
Reno, NV 89557 USA
bekris@cse.unr.edu

Abstract

Multi-agent pathfinding is a challenging combinatorial problem that involves multiple agents moving on a graph from a set of initial nodes to a set of desired goals without inter-agent collisions. Searching the composite space of all agents has exponential complexity and does not scale well. Decoupled methods are more efficient but are generally incomplete. There are, however, polynomial time algorithms, which utilize single or few-agents primitives with completeness guarantees. One limitation of these alternatives is that the resulting solution is sequential, where only one agent moves at a time. Such solutions are of low quality when compared to methods where multiple agents can move simultaneously. This work proposes an algorithm for multi-agent pathfinding that utilizes similar single-agent primitives but allows all agents to move in parallel. The paper describes the algorithm and its properties. Experimental comparisons suggest that the resulting paths are considerably better than sequential ones, even after a post-processing, parallelization step, as well as solutions returned by decoupled and coupled alternatives. The experiments also suggest good scalability and competitive computational performance.

Introduction

Exciting applications in warehouse management, intelligent transportation, robotics and computer games involve many agents that must move from an initial to a goal set of locations in the same environment. This problem is often abstracted using a graph to represent the workspace where no two agents can occupy a node or an edge simultaneously. The problem has been shown to be NP-Complete [Ratner and Warmuth (1986)] and searching the composite space has exponential complexity in the number of agents. The alternative is to apply a decoupled approach where individual paths are found first and intersections are resolved later. While computationally efficient, this approach is generally incomplete and fails in highly constrained challenges.

Alternative approaches, however, provide desirable properties. An early theoretical study refers to the problem as the "pebble motion on a graph" and outlines a decision algorithm, which shows that in the worst case n^3 moves are sufficient for a solution, where n is the number of vertices in the underlying graph [Kornhauser, Miller, and Spirakis (1984)]. Recently, concrete sub-optimal, polynomial complexity algorithms have been proposed for computing the actual path in important subcases of the general problem [Khorshid, Holte, and Sturtevant (2011); Luna and Bekris (2011)]. These methods employ single-agent primitives, or more accurately primitives involving a small, constant number of agents. They return, however, sequential paths, where only one agent moves at a time. Sequential solutions are typically of low quality compared to those where multiple agents move simultaneously.

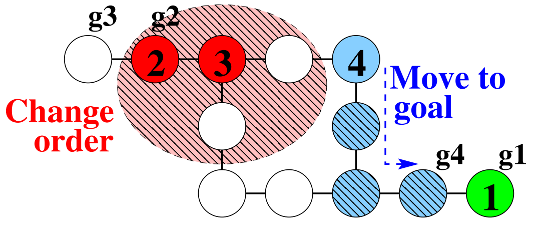

Figure 1: Agent 4 can move towards its goal while agents 2 and 3 change order locally so as to reach their goals. In a sequential solution only one agent moves at a time.

This paper proposes an approach for computing a solution where agents can move simultaneously using similar single-agent primitives. An example of these simultaneous movements is depicted in Figure 1 in which agent 4 moves towards its goal while agents 2 and 3 exchange positions. Comparisons with recent pathfinding algorithms, including both incomplete/decoupled and complete/coupled planners, suggest that the proposed method scales as well as an established decoupled algorithm [Silver (2005); Sturtevant and Buro (2006)], finds solutions as fast as a recent complete, sequential algorithm [Luna and Bekris (2011)], and that the quality of the solution is comparable to a recent anytime, optimal algorithm [Standley (2010); Standley and Korf (2011)]. The proposed technique is shown to be effective not only in large, sparse pathfinding contexts, but also in small, highly constrained situations where even the best optimal search algorithms often fail. The paper discusses the properties of the algorithm and argues that its completeness for trees. The experiments include graphs with loops, on which the proposed method is shown experimentally not to fail.

Related Work

Coupled methods consider the agents as a composite system with high dimensionality and search the composite space with complete, optimal planners (e.g., A* and variants [Hart, Nilsson, and Raphael (1968); Korf (1985)]). Unfortunately, the naive, coupled approach quickly becomes intractable due to its exponential complexity. This led to techniques that prune the search space while maintaining desirable properties. One recent technique searches for optimal paths in an iterative deepening manner, showing that a breadth-first search is feasible [Sharon et al. (2011)]. Another approach modifies state expansion using operator decomposition and segmenting instances into independent subproblems [Standley (2010); Standley and Korf (2011)]. There is also work on minimizing the maximum dimension of independent subproblems [van den Berg et al. (2009)].

Certain complete alternatives cede optimality to achieve computational efficiency. A study of the problem refers to is as the "pebble motion on graphs". It shows that a cubic number of moves as a function of the number of graph vertices is sufficient and that the decision problem can be solved in polynomial time [Kornhauser, Miller, and Spirakis (1984)]. Recent methods provide concrete algorithms for finding paths for specific types of instances. For trees, there is a linear time check for the solvability of an instance [Masehian and Nejad (2009)]. Based on this result, a tree-based technique has been proposed that employs single-agent primitives [Khorshid, Holte, and Sturtevant (2011)]. Similar single-agent primitives have been employed by a polynomial complexity method that is complete on general graphs with two more vertices than agents [Luna and Bekris (2011)]. Another method considers the minimum spanning tree of the graph and is complete if there are more leaves in the tree than agents [Peasgood, Clark, and McPhee (2008)]. There is also a complete approach specifically for bi-connected graphs with two more vertices than agents [Surynek (2009)]. For "slidable" grid-based problems, there is a polynomial time solution [Wang and Botea (2011)]. Another method segments the graph into subgraphs by planning high-level operations between subgraphs to reduce the branching factor [Ryan (2008)].

Decoupled techniques compute individual paths and resolve collisions as they arise. While faster, they are not guaranteed to be optimal or complete. Certain methods are based on heuristic search. One creates a flow network within grid-worlds [Wang and Botea (2008); Jansen and Sturtevant (2008)] and coordinates the actions where the flows intersect to reduce the number of paths and their intersections. WHCA* [Silver (2005); Sturtevant and Buro (2006)] utilizes dynamic prioritization, windowed cooperative search and a perfect single-agent heuristic to compute scalable solutions. Prioritized planners compute paths sequentially for different agents in order of priority, where high priority agents are moving obstacles for the lower priority ones [Erdmann and Lozano-Perez (1986)]. For trees, a hierarchical prioritization-based planner exists to coordinate sequential actions [Masehian and Nejad (2010)]. Other decoupled planners employ a plan-merging scheme to coordinate actions and detect deadlocks [Saha and Isto (2006)].

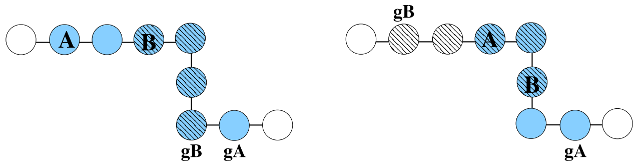

Figure 2: Nodes g_A and g_B are the goals for agents A and B. Left: The start and goal of B are along the shortest path of A. Right: The shortest path of each agent goes through the other's assignment.

Problem Setup

The proposed work borrows ideas from suboptimal, polynomial complexity methods that employ single-agent primitives to solve the multi-agent path-finding problem [Kornhauser, Miller, and Spirakis (1984); Luna and Bekris (2011); Khorshid, Holte, and Sturtevant (2011)]. In contrast to these methods, it achieves a solution where agents move in parallel. It specifically extends the path planning method, which works for general graphs with at least two empty nodes [Luna and Bekris (2011)]. In this context, there are two basic operations: (a) a "push" method that transfers individual agents towards their goals, while clearing the path from other obstructing agents, and (b) a "swap" operation that changes the order between two agents and allows progress when the "push" operation fails.

Consider a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, where $n = |\mathcal{V}|$, and k agents $\mathcal{R}$, where $k \leq n - 2$. An assignment $\mathcal{A} : [1, k] \rightarrow \mathcal{V}$ places the agents in unique vertices:

$$\forall i, j \in [1, k], j \neq i : \mathcal{A}[i] \in \mathcal{V},\ \mathcal{A}[i] \neq \mathcal{A}[j].$$

The starting assignment is denoted as $\mathcal{S}A$, and the goal assignment is denoted as $\mathcal{G}A$. A parallel action $\lambda(\mathcal{A}_a, \mathcal{A}_b)$ is a change between two assignments $\mathcal{A}_a$ and $\mathcal{A}_b$ so that each agent can move only between neighboring vertices in the two assignments:

$$\forall\ i \in [1, k] :\ \ \mathcal{A}_a[i] = \mathcal{A}_b[i] \text{ or } (\mathcal{A}_a[i], \mathcal{A}_b[i]) \in \mathcal{E}.$$

A parallel path $\Pi = \{\mathcal{A}_0, \ldots, \mathcal{A}_q\}$ is a sequence of assignments, so that for any two consecutive assignments $\mathcal{A}_i$ and $\mathcal{A}_{i+1}$ in Π there is a parallel action $\lambda(\mathcal{A}_i, \mathcal{A}_{i+1})$. The objective is to compute a parallel solution $\Pi^* = \{\mathcal{S}A, \ldots, \mathcal{G}A\}$, which is a sequence initiated at $\mathcal{S}A$ and ending at $\mathcal{G}A$.

The following notion is important for the description of the algorithm. In particular, it describes when it is necessary to employ the "swap" operation.

Definition (Dependency) *A "dependency" between agents A and B arises if (a) the start and goal nodes of B are along the shortest path to the goal of A, or (b) the shortest path of each agent goes through the current node of the other.* Examples of situations where dependencies between agents arise are shown in Figure 2.

An important intuition is that it is not necessary to reason over all possible paths but only over shortest paths. If multiple shortest paths exist, then it is sufficient to reason about

Sets	Definition
$\mathcal{R}$	All agents
$\mathcal{P}$	Pushing agents
$\mathcal{U}$	Agents at their goals
$\mathcal{S}$	Agents involved in swaps
$\mathcal{I}$	Vertices reserved for swap operations
$\mathcal{C}$	One individual agent or composite agent
$\mathcal{M}$	Agents that have moved in this iteration
$\mathcal{H}$	Previous pushers in a recursive call
$\mathcal{T}$	Vertices reserved by a specific swap group

Table 1: Definitions of commonly used sets in the algorithm.

one of them. Such dependencies can be resolved implicitly during a "push" operation (agent A moves along a different path). If a dependency between two neighboring agents cannot be resolved by "push", then the "swap" operation will exhaustively consider all possible ways to resolve it.

Simultaneous Action Multi-Agent Pathfinding

At a high level, the proposed PARALLEL_PUSH_AND_SWAP method (Algorithm 1) operates by allowing agents to execute either a PUSH or a SWAP primitive to progress toward its goal. All agents are grouped into one of two sets: (i) the "pushing" agents $\mathcal{P}$, which try to individually make progress along their shortest paths, and (ii) the prioritized "swapping" pairs $\mathcal{S}$, which need to execute a swap action to remove a dependency between them. Initially all agents are in $\mathcal{P}$. When an agent reaches its goal, it is inserted into set $\mathcal{U}$, indicating it should not be disturbed by "pushers" $\mathcal{P}$. The various sets of agents used by the algorithm are defined in Table 1.

Algorithm 1 PARALLEL_PUSH_AND_SWAP ($\mathcal{R}$)

```
1: $\mathcal{P} \leftarrow \mathcal{R}, \mathcal{U} \leftarrow \emptyset, \mathcal{S} \leftarrow \emptyset, \mathcal{I} \leftarrow \emptyset$
2: while $|\mathcal{U}| \neq |\mathcal{R}|$ do
3:     $\mathcal{M} \leftarrow \emptyset$
4:     for all $s \in \mathcal{S}$ do
5:         if SWAP $(s, \mathcal{M})$==FAIL then return FAIL
6:     for all $p \in \mathcal{P}$ do
7:         $\mathcal{H} \leftarrow \emptyset$
8:         PUSH$(p, \mathcal{M}, \mathcal{H}, \mathcal{S}, \mathcal{U}, \emptyset, false)$
9:         if $p$ is at goal then $U.insert(p)$
10: return SUCCESS
```

PARALLEL_PUSH_AND_SWAP iterates first over the swapping pairs $\mathcal{S}$ (line 4) and then the "pushers" $\mathcal{P}$ (line 6). SWAP has higher priority, since its objective is to remove dependencies, which have been detected but not resolved by PUSH. Each step of the algorithm maintains a set $\mathcal{M}$, indicating which agents have been scheduled to move during the last iteration (line 3). This is needed as agents are trying to move in parallel and they should not try to push another agent, which has already been assigned an action for the current step. If the overall problem has no solution, this will be detected by SWAP, which detects that a dependency cannot be resolved (line 5). When all agents have reached their goals, success is reported (line 10). A complete example of this algorithm is shown in Figure 3.

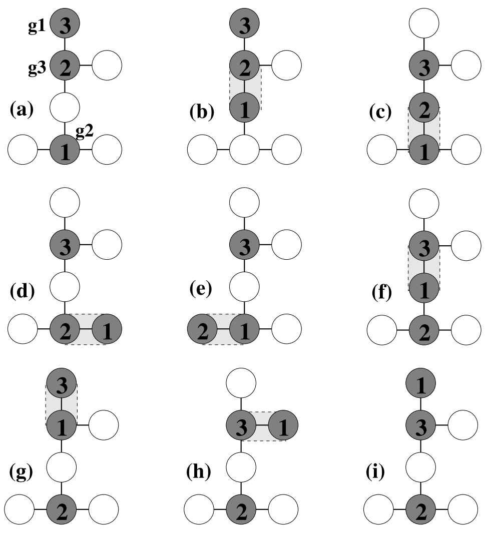

Figure 3: A solution of the simple "tree" benchmark using the proposed approach. Swap groups are shaded. (a-b) Agents 1 and 2 push toward their goals and a dependency is detected. (c-e) Agents 1 and 2 form a swap pair, and move to a vertex of degree $\geq$ 3. Agent 3 reaches its goal. (f-i) Agent 1 swaps with agent 3 to reach its goal. Agent 2 reaches its goal using PUSH. Details on how a swap between takes place can be found in [Luna and Bekris (2011)].

Swap Operation

When computing a solution for a pathfinding instance, it is necessary to ensure that all inter-agent dependencies are resolved. Agents that have a dependency cannot use PUSH to reach their goal since their paths are diametrically opposed. SWAP (algorithm 2) explicitly removes a dependency through a coordinated set of actions which results in the dependent agents exchanging vertices. There are three steps required when swapping two agents: the agents must go to a vertex with degree $\geq$ 3, evacuate two neighbors of this vertex, and then perform a primitive swapping operation.

Definition (Swap Vertex) *A "swap vertex" is a vertex of degree $\geq$ 3 used by a pair of agents for* SWAP. A dependent pair of agents reserves this vertex, preventing lower priority agents from occupying it. When an agent from the pair reaches the swap vertex, its partner is presumed to occupy a neighboring vertex, and the pair is said to occupy the swap vertex. The "swap area" is a set of vertices that includes a reserved swap vertex and its neighbors. Only higher priority agents may enter a swap area. A "swap group" refers to all agents involved in a SWAP for a particular swapping pair (i.e., additional agents occupying a "swap area").

SWAP is a coordinated set of actions among agents in a swap group, $\mathcal{C}$, which requires several time steps to complete. Therefore, $\mathcal{C}$ can vary in size depending on the state of the primitive. When SWAP is initially invoked, $|\mathcal{C}| = 2$; the set contains only the dependent pair of agents. In this case, the agents are either using PUSH to reach their swap vertex (line 2-3), or have just reached it and must begin freeing two vertices in the swap area using CLEAR (line 5). If the swap area has two empty vertices, the pair of agents will exchange vertices using SWAP_PRIMITIVE (line 6), at which time the dependency is resolved and the pair is re-inserted into $\mathcal{P}$. If CLEAR is unsuccessful, SWAP will assign a new swap vertex if $\mathcal{C}$ has the highest priority (line 8), otherwise SWAP will be invoked again at the same swap vertex. If the highest priority pair exhausts all vertices of degree ≥ 3, a swap is not possible between the dependent agents and there is no solution for this instance (lines 7-8). The process of freeing two vertices in the swap area may span several time steps, during which SWAP will be invoked for $|\mathcal{C}| > 2$ (lines 9-18), and the swap group will be coordinating their actions to free the swap area.

Algorithm 2 SWAP ($\mathcal{C}, \mathcal{M}$)

```
1: if $|\mathcal{C}| == 2$ then
2:     if $\mathcal{C}$ not on swap vertex with degree $\geq 3$ then
3:         PUSH($\mathcal{C}, \mathcal{M}, \emptyset, \mathcal{S}, \mathcal{U}, \emptyset, true$)
4:     else
5:         $attempt$=CLEAR($\mathcal{C}, \mathcal{M}, \mathcal{C}.swapVert.$)
6:         if $attempt$==SUCCESS then SWAP_PRIMITIVE ($\mathcal{C}$)
7:         else if $attempt$==FAIL then
8:             return FIND_NEW_VERTEX($\mathcal{C}$)
9: else
10:    $a \leftarrow \mathcal{C}.end()$
11:    $attempt$=PUSH($a, \mathcal{M}, \emptyset, \mathcal{S}, \mathcal{U}, \mathcal{C}.area, true$)
12:    if $attempt$==FAIL then
13:        $\mathcal{C} \leftarrow \mathcal{C}.pop()$
14:        return FIND_NEW_VERTEX($\mathcal{C}$)
15:    else if $attempt$==SUCCESS then
16:        if $a$ is at $tempGoal$ then
17:            erase $a.tempGoal$, $\mathcal{C} \leftarrow \mathcal{C}.pop()$
18:        else PUSH($\mathcal{C} \setminus a, \mathcal{M}, \emptyset, \mathcal{S}, \mathcal{U}, \emptyset, true$)
19: return SUCCESS
```

During the CLEAR phase, agents occupying the swap area are included in the swap group ($|\mathcal{C}| > 2$). This formulation allows the status of the swap to be implicitly encoded across several time steps. Due to space constraints a formal algorithm for CLEAR is not provided, but a brief description follows. If there are two empty vertices in the swap area, CLEAR is trivially successful. Otherwise there are agents occupying the neighborhood of the swap vertex. There are two methods to move these agents: The first uses PUSH to move these agents to any vertex away from the swap area. If this PUSH frees two vertices then CLEAR succeeds. On the other hand, if after this step there are no empty vertices in the swap area, CLEAR fails. For the case when there is a single empty vertex in the swap area, a second phase for CLEAR is attempted in which PUSH is used to move an agent in the swap area through the empty vertex via the swap vertex and out of the swap area. To perform this step, it is necessary to move the swapping pair away from the swap vertex first, then the occupying agent can move to the free neighbor. At the formerly empty vertex, the occupying agent can check whether it is possible to push to a vertex immediately outside the swap area using FEASIBLE. If this motion is valid, the outside vertex becomes the agent's temporary goal until the vertex is reached (lines 15-18). When this motion is finished the swapping pair can move back to the swap vertex (lines 10-14), at which time CLEAR is successful.

Push Operation

PUSH attempts to move an agent, or a composite agent $\mathcal{C}$, one vertex along a specific path. The order of the agents executing PUSH determines which can be "pushed" away from their current position to accommodate the motion of "pushers". PUSH entails two steps: (a) computing a path for the pushing agent(s) that respects the priorities of all other agents, and then(b) attempting to make progress along this path. Under certain condition, PUSH can change the set that an agent belongs to. If an agent $\mathcal{C}$ in $\mathcal{P}$ cannot push along its shortest path, then PUSH creates a swapping pair out of $\mathcal{C}$ and the agent currently blocking its movement and adds the pair into $\mathcal{S}$ as the lowest priority pair. If instead the pushing agent $\mathcal{C}$ already belongs to $\mathcal{S}$ (PUSH is invoked by SWAP) and it must push one or more members of a lower priority pair in $\mathcal{S}$, the lower priority group is dissolved and reinserted into $\mathcal{P}$. PUSH is detailed in algorithm 3.

Algorithm 3 PUSH ($\mathcal{C}, \mathcal{M}, \mathcal{H}, \mathcal{S}, \mathcal{U}, \mathcal{T}, swap$)

```
1: if $\mathcal{C} \in \mathcal{H}$ then return FAIL
2: if ($\mathcal{C} \in \mathcal{U}$ and swap==false) or $\mathcal{C} \in \mathcal{M}$ then
3:     return PAUSE
4: $\mathcal{U} = \mathcal{U} \setminus \mathcal{C}$
5: if $\mathcal{C}$ has temporary goal then
6:     $\pi$= SHORTEST_PATH ($\mathcal{C}, \mathcal{C}.tempGoal, \emptyset$)
7: else
8:     $\pi$=SHORTEST_PATH ($\mathcal{C}, \mathcal{C}.goal, \mathcal{H}_{|\mathcal{H}|-1} \cup \mathcal{T}$)
9:     if FEASIBLE($\mathcal{C}$, $\pi$, $\mathcal{M}, \mathcal{H}, \mathcal{S}, \mathcal{T}, \mathcal{I}, swap$)==FAIL then
10:        while $\pi = \emptyset$, $c_n \in \mathcal{H}$ do
11:            $\pi$=SHORTEST_PATH ($\mathcal{C}, c_n.goal, \mathcal{H}_{|\mathcal{H}|-1} \cup \mathcal{T}$)
12:    if FEASIBLE($\mathcal{C}$, $\pi$, $\mathcal{M}, \mathcal{H}, \mathcal{S}, \mathcal{T}, \mathcal{I}, swap$)==FAIL then
13:        $\mathcal{Y} = closest\_empty\_vertices(\mathcal{C})$
14:        $e$: argmin$_{y \in \mathcal{Y}}$ (shortest distance to $\mathcal{C}.goal$)
15:        $\pi$=SHORTEST_PATH ($\mathcal{C}, e, \mathcal{H}_{|\mathcal{H}|-1} \cup \mathcal{T}$)
16: $attempt$ = FEASIBLE($\mathcal{C}$, $\pi$, $\mathcal{M}, \mathcal{H}, \mathcal{S}, \mathcal{T}, \mathcal{I}, swap$)
17: if $attempt$== SUCCESS then
18:     move $\mathcal{C}$ one vertex along $\pi$, $\mathcal{M} \leftarrow \mathcal{M} \cup \{\mathcal{C}\}$
19: return $attempt$
```

PUSH employs the following parameters. $\mathcal{C}$ is the set of agents executing the push. Typically, $|\mathcal{C}| = 1$. Nevertheless, SWAP can also invoke PUSH for a pair of agents, in which case $|\mathcal{C}| = 2$. In that case, the input parameter $swap$ is set to TRUE and the parameter $\mathcal{T}$ corresponds to the vertices reserved for swapping agents in $\mathcal{C}$ ($\mathcal{T} \subset \mathcal{I}$). Otherwise, $swap$ is $false$ and $\mathcal{T} = \emptyset$. PUSH operates in a recursive manner, checking the feasibility of movement for each agent individually. Each agent processed is inserted into the set $\mathcal{H}$.

PUSH begins by verifying that a push is indeed valid by first checking that $\mathcal{C}$ is not pushing itself recursively, that is has not already made an action this round, or that it is already at its goal and not involved in swap ($\mathcal{C} \notin \mathcal{H}$, $\mathcal{C} \notin \mathcal{M}$, $\mathcal{C} \in \mathcal{U}$ respectively). If any of these conditions is true, the push does not proceed (lines 1-3). PAUSE indicates that the push does not fail, but cannot proceed during this step. It can also happen that an agent is displaced from its goal during SWAP. PUSH removes $\mathcal{C}$ from $\mathcal{U}$ before checking any action to ensure that an agent previously pushed from its goal by other agents (likely due to a swap) will return to its goal (line 4). Presuming these checks succeed, PUSH then computes the path π that $\mathcal{C}$ will move along (lines 5-15). For the special case that $\mathcal{C}$ is assigned a temporary goal (i.e., during SWAP when moving to the swap vertex or for agents clearing a swap area), the shortest path, ignoring the positions of all other agents is computed (lines 5-6). Otherwise, there are three paths to consider for a push, all of which must avoid agents already involved in this push (the set $\mathcal{H}$): the shortest path to the goal of $\mathcal{C}$ ("optimistic"), the shortest path to the goal of an agent already pushing $\mathcal{C}$ ("go with the flow"), and the path to the closest empty vertex ("likely to succeed"). With each push the motivation is to move the agent closer to its goal either directly by finding a path to the goal or indirectly by finding a path to a location close to the goal when the direct path to the goal cannot be found. As a last resort when a path to the first two option cannot be found, the agent is pushed out of the way to the closest empty. PUSH uses an auxiliary method, FEASIBLE (algorithm 4), to verify feasibility of each path (and recursively call PUSH if necessary), which is described later. When the first feasible path is discovered, $\mathcal{C}$ is moved one step along this path and success is returned (lines 17-18). If no valid path is found for $\mathcal{C}$, the last computed value of FEASIBLE is returned (line 19).

Algorithm 4 FEASIBLE ($\mathcal{C}, \pi, \mathcal{M}, \mathcal{H}, \mathcal{S}, \mathcal{T}, \mathcal{I}, swap$)

```
1:  if $\pi == \emptyset$ then return FAIL
2:  $v \leftarrow \pi.pop()$
3:  if $v$ is reserved by $m \in \mathcal{M}$ then
4:      return PAUSE
5:  else if $v \in \mathcal{I}$ then
6:      if $(swap == false)$ then return PAUSE
7:      if CHECK_PRIORITY($\mathcal{C}$, group using $v$)==FAIL then
8:          return FAIL
9:      else if $\mathcal{T} \neq \emptyset$ then return FAIL
10: else if agent ($a$) is on $v$ and $swap == false$ then
11:     if $a \in \mathcal{S}$ then return FAIL
12:     $\pi$= SHORTEST_PATH ($\mathcal{C}, \mathcal{C}.goal, \emptyset$)
13:     $\pi$' = SHORTEST_PATH ($a, a.goal, \emptyset$)
14:     $\mathcal{ESV} \leftarrow \{$vert. of degree $\geq 3$, sorted by dist. from $\mathcal{C}\}$
15:     if DEPEND($\mathcal{C}, a, \pi, \pi', \mathcal{ESV}.pop()$)==TRUE then
16:         SETUP_SWAP($\mathcal{C}, a, \mathcal{ESV}.pop()$)
17:         $\mathcal{S} \leftarrow \mathcal{S} \cup \{\mathcal{C}, a\}$
18:         return FAIL
19: else if (agent ($a$) is on $v$ and $swap == true$) then
20:     if $a \in \mathcal{S}$ then
21:         if CHECK_PRIORITY($\mathcal{C}$, group of $a$)==FAIL then
22:             return FAIL
23: else if ($v$ is free) then return SUCCESS
24: return PUSH($a, \mathcal{M}, \mathcal{H} \cup \mathcal{C}, \mathcal{S}, \mathcal{U}, \mathcal{T}, swap$)
```

FEASIBLE validates a path computed by PUSH, and determines whether it is possible for $\mathcal{C}$ to move along π to vertex v during the next time step. It isn't possible for an agent to move into a vertex already reserved by an agent for the next time step (lines 3-4). If v belongs to another agent's "swap area", the priorities between the $\mathcal{C}$ and the swap group using v is compared in CHECK_PRIORITY. If the $\mathcal{C}$ has higher priority the other group is dissolved and the agents are pushed back into $\mathcal{P}$; otherwise, $\mathcal{C}$ just waits for the other group to finish (lines 5-8, 19-22). Also, an agent cannot push into its own "swap area". This case arises when clearing agents during SWAP (line 9). At this point, if v is free, it is safe to move to it (line 23). This leaves just one last case: v is occupied, and $\mathcal{C} \in \mathcal{P}$ (lines 10-18). It is possible now to detect whether a SWAP is necessary between $\mathcal{C}$ and the agent occupying v by checking whether a dependency between these two agents exists using DEPEND. Recall that a dependency between two agents occurs when an agent must move through the start and goal of another agent, or two agents have diametrically opposing shortest paths. If these agents are dependent, FEASIBLE forms a swap group with these two agents, and assigns the new group a swap vertex (lines 15-17). Since a SWAP is required between these two agents, any attempt to PUSH $\mathcal{C}$ along the path will fail (line 18). If the agents are not dependent, the success of FEASIBLE depends on whether it is possible to move the agent occupying v in a recursive PUSH (line 24). $\mathcal{C}$ is appended to the set $\mathcal{H}$, indicating to the agent of the recursive PUSH that $\mathcal{C}$ is pushing it.

Analysis

This section argues the completeness of the PARALLEL_PUSH_AND_SWAP for trees when $|\mathcal{R}| \leq n-2$ and $|V| = n$.

Lemma 1: If the agents being pushed have no "dependency", the FEASIBLE *will not form any new dependencies. Push does not change the "order" of agents.*

FEASIBLE moves agents in three ways. Agents will first try to push along the shortest path to their goal. In this case, no dependency is introduced between an agent and its pushers as they will remain in the same order relative to their goals. If that fails, or the agent is already at its goal, then it pushes towards its pushers' goal. Then the agent and its pushers are pushed in a chain towards the original pusher's goal and no new dependency arises. In the last case, an agent may need to move towards the empty vertex closest to its goal. This also does not introduce any new dependency because in this case the pushers will also follow the same path.

Lemma 2: Only necessary swaps are detected by the algorithm. The swaps will be resolved if the problem is solvable.

If there is a dependency between two agents, it will be detected by FEASIBLE. FEASIBLE checks for both types of dependencies between a pusher and a pushee. Nevertheless, on graphs that contain loops and if only the shortest paths of agents are considered, then false dependencies may be detected depending on the position of agents along the loop. To prevent false positives, the dependency between a pair of agents involved will be also checked at a vertex of degree ≥ 3 (an example of a false positive is shown in Figure 4). If the dependency remains, the swap is necessary. If there is a dependency involving multiple agents, (e.g., agent a_1 dependence on its successor and their is a dependency

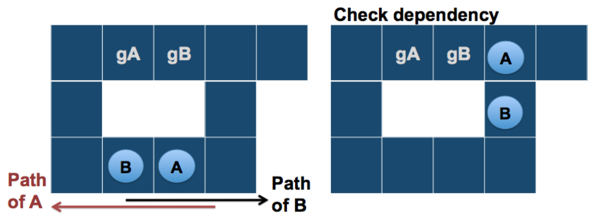

Figure 4: False positives can occur when two agents are in the correct order but find shortest paths in the opposite direction of each other due to being equidistant from their goals. In the figure, agents A and B have a dependency due to crossing paths but it is found to be a false positive when checked at a vertex of degree ≥ 3. Note that the agents are not actually moved to that vertex but their dependency is checked as if they had in their current order.

on successors until a_k which depends on a_1), then PARALLEL_PUSH_AND_SWAP will also detect this dependency. This is discovered by FEASIBLE, which is able to identify that an agent being pushed is in the list of pushers $\mathcal{H}$. In this case, FAIL is returned, which causes the agent to look for an alternative direction, such as push towards an empty vertex.

If the problem is solvable, the swap pair with the highest priority will move to a vertex of degree ≥ 3. Eventually all swap pairs will acquire the highest priority. Even if a swap pair is dissolved, the algorithm will be able to recreate it later. CLEAR will exhaust all possible ways to free two adjacent vertices in a specific vertex of degree ≥ 3. If CLEAR fails for this vertex, it will exhaustively consider all vertices.

Lemma 3: Pairs of agents will swap at most once on trees.

Swap pairs are formed when there is a dependency between two agents that needs to be resolved. In a graph with a loop it is possible that once a dependency between a pair (a,b) is resolved, that a will swap with most of the other agents in the loop, while b swaps with the rest. This will cause an implicit swap between a and b, recreating the dependency between them once again. In a tree, no implicit swap can arise and agents will swap at most once. The experimental results suggest that the number of implicit swaps occurring between pairs of agents is bounded (e.g., on problems of the form shown in Figure 5). Future work will aim to show that the number of swaps is finite for general graphs with loops.

Theorem: All trees with $|\mathcal{R}| \leq n-2$ and $|V| = n$ that can be solved will be solved through PARALLEL_PUSH_AND_SWAP.

Proof: For the top pushing agent r, the algorithm will attempt to move it to its goal, pushing along the way other agents closer to their goals, or r's goal, or to an empty vertex closest to their goal. This will allow r to push any agent other than those involved in swap. If a dependency is found, then a swap is necessary. The pair with the dependency will move to the vertex, clear two neighboring vertices, and continue to switch positions. Because the swap primitive requires at least two empty vertices, it is necessary that $|\mathcal{R}| \leq n-2$ and $|V| = n$. Otherwise, the problem cannot be solved with PARALLEL_PUSH_AND_SWAP. If a dependency is not found, r makes progress towards its goal.

Based on Lemmas 1 and 2, FEASIBLE will only move agents in a way that unnecessary swaps will not occur. If a necessary swap is needed, it will be detected and resolved. According to Lemma 3, the same pair will not be swapped more than a finite number of times. Each swap removes a dependency between agents. At some point all agents will be independent of one another. Without any dependencies, all agents will reach their goal by using FEASIBLE. In this way, the PARALLEL_PUSH_AND_SWAP algorithm guarantees that every agent will be brought to its goal. The displacement of an agent from its goals, either for a push or swap operation, will not prevent the agent from returning back to it.

PARALLEL_PUSH_AND_SWAP is able to solve all cases on trees with $|\mathcal{R}| \leq n-2$ agents. It can solve trees with $|\mathcal{R}| \leq n-1$ agents when no swap is required.

Evaluation

The proposed method is compared against the following:

1. The sequential equivalent [Luna and Bekris (2011)], which returns solutions where one agent moves at a time.

2. A "parallelized" version of the output returned by method 1 above, where a post-processing step removes states when agents remain in the same vertex. The removal is accepted if the solution is still valid (i.e., no inter-agent collisions). The search repeats until no possible states can be removed from the solution without invalidating it. A scalable and optimal method for parallelization is not known [Surynek (2009)]. Greedy methods could also be employed, but the path quality is poor in comparison.

3. The WHCA* approach [Silver (2005); Sturtevant and Buro (2006)], which is a decoupled planner that uses windowed cooperative search and dynamic prioritization. For all tests, WHCA* had a timeout of 10 minutes since typically when it finds a solution, it does so rather quickly.

4. The ODA* Standley (2010); Standley and Korf (2011), an A*-based solution with operator decomposition, which splits the state expansion step of a classical A* search into a series of "operators", one for each agent. A state is not advanced until each agent is assigned a valid operator. This method bring the branching factor down to a constant factor, while increasing the depth of the solution by an amount linear in the number of agents. ODA* had a timeout of 1 minute due to its significant memory requirements.

The algorithms are tested on cooperative pathfinding problems, ranging from small benchmarks, where all agents share pair-wise dependencies, to larger scale problems. All experiments were executed on an Intel Core i5 2.66Ghz with 4GB of memory. These algorithms were evaluated based on path quality, computation time, and success rates.

Small-size Benchmark Problems The benchmarks used here are those from the sequential work on Push and Swap [Luna and Bekris (2011)] plus the two loop problem shown in Figure 5. These benchmarks involve anywhere between 3 to 16 agents.

Computation Time: The solutions returned from all five techniques are shown in Table 2. PARALLEL_PUSH_AND_SWAP takes more time than the sequential

Problems	Parallel PS			Sequential PS			Sequential PS w/Parallelization			WHCA* (5)			ODA*		
	Time	Steps	Success	Time	Steps	Success	Time	Steps	Success	Time	Steps	Success	Time	Steps	Success
Tree	1.04	9	100%	0.60	39	100%	3.97	20	100%	**0.43**	16.1	36.1%	2.17	**6**	100%
Corners	2.64	21	100%	1.33	68	100%	29.2	27	100%	**0.47**	10.2	11.1%	16.10	**8**	100%
Tunnel	9.53	44	100%	**1.62**	159	100%	1574	49	100%	–	–	0%	184	**6**	100%
Connect	7.70	37	100%	**3.10**	126	100%	1167	**29**	100%	–	–	0%	–	–	0%
String	2.38	15	100%	0.53	39	100%	8.56	23	100%	0.73	14	62.7%	**.45**	**8**	100%
Two Loop	4527	**1015**	100%	**77.4**	5269	100%	–	–	0%	–	–	0%	–	–	0%

Table 2: Computation time (ms), solution length (# of steps) and success rates for benchmarks. WHCA* and ODA* do not always return. Averages shown over twenty repetitions per case. Bold values are the best results in terms of time and steps taken for each case.

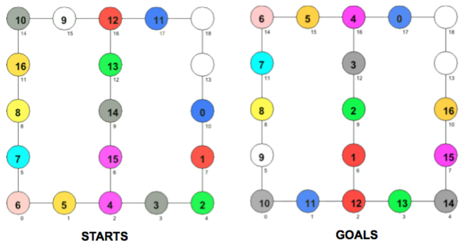

Figure 5: Two loop problem.

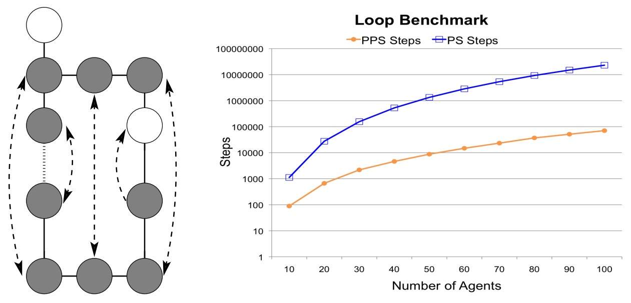

Figure 6: A difficult problem where agents must rotate around a loop to get to their goal. A swap can take place only at one vertex. The graph shows the length of the solution for the sequential and parallel algorithms as the loop grows in a logarithmic scale.

Push and Swap, as well as WHCA* (with window size of 5) for these small scale problems. Note, however, that WHCA* is incomplete and does not always return a solution. Furthermore, the PARALLEL_PUSH_AND_SWAP approach is able to always compute solutions faster than the sequential algorithm with the parallelization step and ODA*.

Solution Length: The number of steps taken by PARALLEL_PUSH_AND_SWAP, is smaller than the sequential Push and Swap which often requires more than two times the number of steps. The path found by parallelizing the sequential solution still did worse than PARALLEL_PUSH_AND_SWAP. Although WHCA* produces a competitive solution for *corners*, its success rate is low. Anytime-ODA* finds the optimal solution for three of the maps, but it requires considerably more computation time. It also failed to find the solution in some maps. Overall, PARALLEL_PUSH_AND_SWAP produces better solutions than alternatives with competitive times.

Large Scale Problems This paper considers two types of scalable problem instances: "loop" and "random" problems. Experiments were also run on "Moving AI" benchmark maps [Sturtevant (2012)].

Loop Problems: n-2 agents are placed in a loop-like graph with n nodes. An example is provided in Figure 6. This problem is challenging, as all the agents share pairwise dependencies. Two techniques are evaluated: PARALLEL_PUSH_AND_SWAP and the sequential method. WHCA* and ODA* were unable to compute solutions for any instance of this family of problems. The parallelization of the sequential solution was significantly slower than the time it took to compute solutions with PARALLEL_PUSH_AND_SWAP. PARALLEL_PUSH_AND_SWAP is capable of producing increasingly better solutions especially as the number of agents in the loop increases.

Random Problems: These involve a 20x30 grid map where 20% of cells are obstacles. This map is populated with 5-100 agents (in increments of 5) with 100 random start and goal configurations for each number of agents creating a total of 2000 tests for each algorithm. Figure 7(left) shows that as the problem size increases, the time PARALLEL_PUSH_AND_SWAP takes in comparison to Push and Swap decreases. ODA* takes much more time than the PARALLEL_PUSH_AND_SWAP but is significantly better than WHCA* with window size 20. WHCA* with window size 10 approaches the efficiency of PARALLEL_PUSH_AND_SWAP but as the number of agents increases, its success ratio drops. ODA* computes better paths at a significantly higher computation cost and with lower success rate. As in Figure 7 (middle), the number of steps computed by PARALLEL_PUSH_AND_SWAP is noticeably lower than that computed by the sequential Push and Swap. WHCA* computes solutions similar in size to that of PARALLEL_PUSH_AND_SWAP but its success rate drops fast. PARALLEL_PUSH_AND_SWAP computes better solutions at competitive times and with 100% success rate.

"Moving AI" Map: A hundred start/goal location pairs were generated on the arena map from the game Dragon Age [Sturtevant (2012)]. The number of agents is increased by fifty until there are two hundred agents. As shown in Figure 8(left), WHCA* takes too much time to be visible in the graph. ODA* was unsuccessful on these maps (Figure 8(right)). PARALLEL_PUSH_AND_SWAP took less steps to solve the problem than WHCA* (both window sizes) and the sequential Push and Swap (Figure 8(middle)). As shown in Figure 8(right), PARALLEL_PUSH_AND_SWAP has a consistent success ratio of 100%, while the rates for WHCA*

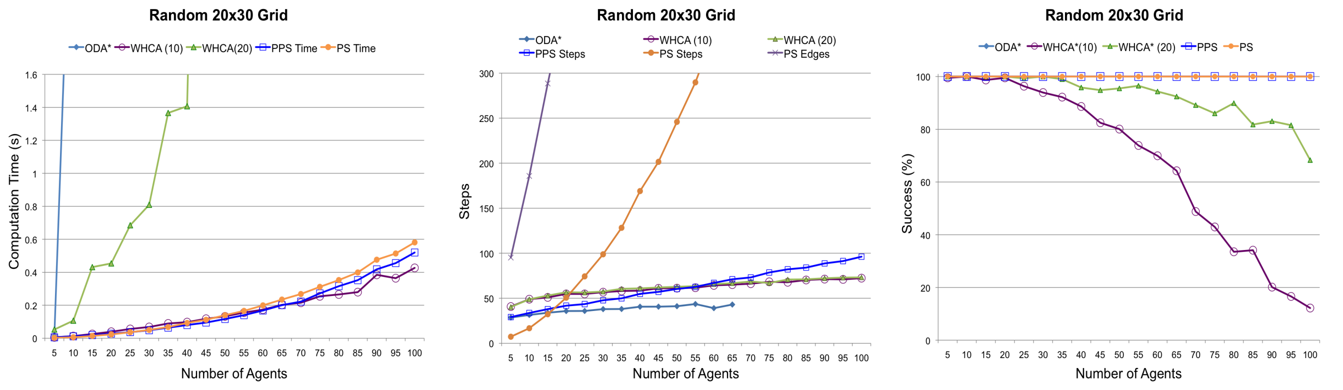

Figure 7: (left) Computation time. (middle) Solution length (including number of edges traversed by Sequential Push and Swap). (right) Success ratio. 20x30 grid-world with 4-connectivity and 20% obstacle density.

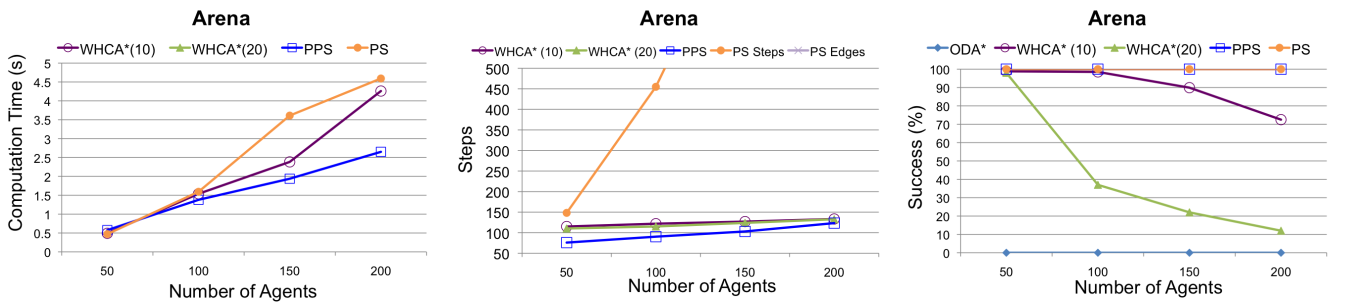

Figure 8: (left) Computation time. (middle) Solution length (including number of edges traversed by Sequential Push and Swap). (right) Success ratio. The algorithms were executed on the "Moving AI" arena map. ODA* always failed. Computation time taken by WHCA* (20) and the number of edges for Sequential Push and Swap were too high to be shown.

and ODA* drop quickly.

Discussion

This work presents an algorithm for multi-agent pathfinding with simultaneous actions. Related work utilizing similar primitives returned only sequential solutions. Experimental comparisons show the advantages of the approach both on tightly constrained instances, as well as larger, more general scenarios. Solution lengths for the proposed method are competitive with anytime complete approaches, and significantly better than sequential solutions. The time needed to compute these solutions is also competitive. The proposed method never failed in the tested environments.

It is interesting to investigate optimal algorithms using the same set of primitives, although this is expected to lead to an exponential complexity result. The agent ordering, and the priorities assigned by the algorithm affect computation time and solution quality, and it may prove useful to select them in a more informed manner. Another extension involves relaxing the restriction requiring two empty vertices in the input graph to guarantee completeness.

Acknowledgments: The work of Q. Sajid and K. Bekris is supported by NSF CNS 0932423. Q. Sajid was also supported by NV NASA EPSCOR. The work of R. Luna is supported by NSF IIS 0713623. Any opinions expressed in this paper are those of the authors and do not necessarily reflect the views of the sponsors.

References

Erdmann, M., and Lozano-Perez, T. 1986. On Multiple Moving Objects. In *ICRA*, 1419–1424.

Hart, E. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 2:100–107.

Jansen, M. R., and Sturtevant, N. 2008. Direction Maps for Cooperative Pathfinding. In *AIIDE*.

Khorshid, M. M.; Holte, R. C.; and Sturtevant, N. 2011. A Polynomial-Time Algorithm for Non-Optimal Multi-Agent Pathfinding. In *SOCS*, 76–83.

Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27:97–109.

Kornhauser, D.; Miller, G.; and Spirakis, P. 1984. Coordinating pebble motion on graphs, the diameter of permutation groups and applications. In *FOCS*, 241–250.

Luna, R., and Bekris, K. E. 2011. Push and Swap: Fast Cooperative Path-Finding with Completeness Guarantees. In *IJCAI*.

Masehian, E., and Nejad, A. H. 2009. Solvability of Multi Robot Motion Planning Problems on Trees. In *IROS*, 5936–5941.

Masehian, E., and Nejad, A. H. 2010. A Hierarchical Decoupled Approach for Multi Robot Motion Planning on Trees. In *ICRA*, 3604 – 3609.

Peasgood, M.; Clark, C.; and McPhee, J. 2008. A Complete and Scalable Strategy for Coordinating Multiple Robots within Roadmaps. *IEEE Transactions on Robotics* 24(2):282–292.

Ratner, D., and Warmuth, M. K. 1986. Finding a shortest solution for the n x n extension of the 15-puzzle is intractable. In *The Fifth National Conference on Artificial Intelligence (AAAI'86)*, 168–172.

Ryan, M. R. K. 2008. Exploiting Subgraph Structure in Multi-Robot Path Planning. *Journal of AI Research* 31:497–542.

Saha, M., and Isto, P. 2006. Multi-Robot Motion Planning by Incremental Coordination. In *IROS*, 5960–5963.

Sharon, G.; Stern, R.; Goldenberg, M.; and Felner, A. 2011. The Increasing Cost Tree Search for Optimal Multi-Agent Pathfinding. In *IJCAI*, 662–667.

Silver, D. 2005. Cooperative Pathfinding. In *AIIDE*, 23–28.

Standley, T., and Korf, R. 2011. Complete Algorithms for Cooperative Pathnding Problems. In *IJCAI*, 668–673.

Standley, T. 2010. Finding Optimal Solutions to Cooperative Pathfinding Problems. In *AAAI*, 173–178.

Sturtevant, N., and Buro, M. 2006. Improving Collaborative Pathfinding Using Map Abstraction. In *AIIDE*, 80–85.

Sturtevant, N. 2012. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*.

Surynek, P. 2009. A Novel Approach to Path Planning for Multiple Robots in Bi-connected Graphs. In *ICRA*, 3613–3619.

van den Berg, J.; Snoeyink, J.; Lin, M.; and Manocha, D. 2009. Centralized path planning for multiple robots: Optimal decoupling into sequential plans. In *Robotics: Science and Systems*.

Wang, K.-H. C., and Botea, A. 2008. Fast and Memory-Efficient Multi-Agent Pathfinding. In *ICAPS*, 380–387.

Wang, K.-H. C., and Botea, A. 2011. MAPP: A Scalable Multi-Agent Path Planning Algorithm with Tractability and Completeness Guarantees. *Journal of AI Research* 42:55–90.