Published in the Proc. of Int'l. Joint Conference on Artificial Intelligence, 2011, pp. 294-300

Push and Swap: Fast Cooperative Path-Finding with Completeness Guarantees

Ryan Luna and Kostas E. Bekris *
University of Nevada, Reno
{rluna, bekris}@cse.unr.edu

Abstract

Cooperative path-finding can be abstracted as computing non-colliding paths for multiple agents between their start and goal locations on a graph. This paper proposes a fast algorithm that can provide completeness guarantees for a general class of problems without any assumptions about the graph's topology. Specifically, the approach can address any solvable instance where there are at most n-2 agents in a graph of size n. The algorithm employs two primitives: a "push" operation where agents move towards their goals up to the point that no progress can be made, and a "swap" operation that allows two agents to swap positions without altering the configuration of other agents. Simulated experiments are provided on hard instances of cooperative path-finding, including comparisons against alternative methods. The results are favorable for the proposed algorithm and show that the technique scales to problems that require high levels of coordination, involving hundreds of agents.

1 Introduction

Cooperative path-finding requires the computation of paths for multiple agents on a graph, where the agents must move from their unique start nodes to their unique targets while avoiding collisions. This problem is relevant to many applications, such as warehouse management, intelligent transportation, mining, space exploration, as well as computer games.

1.1 Background

The problem can be solved with a coupled approach, which plans for the composite graph $G^n = G \times G \times \ldots \times G$, where G is the original graph and n is the number of agents, or through a decoupled approach, where paths are computed individually and then conflicts are resolved. Integrated with complete search methods, such as A*, the coupled algorithm achieves completeness and optimality. Nevertheless, coupled planning becomes impractical due to the exponential dependency on n and soon requires far too much time and memory.

*This work is supported by NSF CNS 0932423. Any opinions, findings and conclusions expressed in this work are those of the authors and do not necessarily reflect the views of the sponsors.

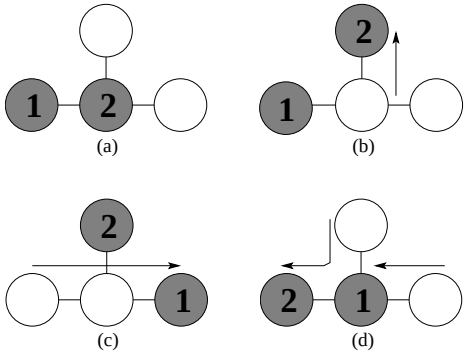

Figure 1: Illustration of the swap primitive. Agents 1 and 2 are to swap positions with one another.

In prioritized schemes, paths are computed sequentially and high-priority agents are considered moving obstacles by low-priority ones [Erdmann and Lozano-Perez, 1986]. Searching the space of prioritizations can assist in performance [Bennewitz *et al.*, 2002]. Such decoupled planners tend to prune states in which higher priority agents allow lower priority agents to progress, which may eliminate the only viable solutions. Modern decoupled approaches consider dynamic prioritization and windowed search [Silver, 2005], as well as spatial abstraction for improved multi-agent heuristic computation [Sturtevant and Buro, 2006]. Another modern search-based technique creates a flow network in a grid-world to significantly reduce the branching factor and the amount of replanning [Wang and Botea, 2008].

Certain methods aim to reduce the size of the search space while maintaining completeness. A hybrid technique plans for each agent given the future paths of other agents, but employs a coupled approach for assigning targets and avoiding deadlocks [Qutub *et al.*, 1997]. An alternative method manually decomposes a large graph into subgraphs of prespecified types [Ryan, 2007]. Then it plans between subgraphs before coordinating motion within each subgraph. For graphs with specific topologies, efficient strategies to solve multi-agent path planning problems exist [Peasgood *et al.*, 2008], [Surynek, 2009]. For specific domains in grid-worlds, there is a complete polynomial time algorithm [Wang and Botea, 2009]. A recent sequential approach computes the optimal decoupling of large-scale problems into fully-coupled subproblems [van den Berg *et al.*, 2009].

1.2 Contribution

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

The proposed method is many orders of magnitude faster compared against the traditional coupled A* approach, as it focuses its search into a necessary subset of operations to get a solution. In comparison to existing alternatives that provide completeness guarantees for certain problem subclasses, the proposed method provides similar guarantees for a much wider problem class. Furthermore, it does not make any assumptions about the topology of the underlying graph, as it does not require the graph to be a grid, as other methods do [Wang and Botea, 2008], [Wang and Botea, 2009], or a tree [Peasgood *et al.*, 2008]. Compared to a general decoupled algorithm devised to solve problems in a computationally efficient manner [Silver, 2005], the proposed solution exhibits competitive solution times while having no dependence on parameter selection. Most importantly, the proposed approach solves a much wider set of problems than the incomplete, decoupled alternative.

The approach employs two basic primitives. Note that easy instances, where there is no coupling between agents, can be solved by each agent moving along its shortest path to its goal. One of the two primitives of the proposed approach, called "push", is similar to this operation. An agent forces other agents to clear its shortest path to its goal and moves along this path. Harder problems, however, will require at least two agents to switch positions. This issue is addressed by the second primitive of the algorithm called "swap". Once an agent cannot make progress towards its goal by pushing, then it has to swap positions with the agent next to it along the shortest path. This operation brings the two agents in a part of the graph where the swap can take place. This may force other agents in the graph to move in response, potentially all of them. Eventually, all the agents must be returned to their original positions, with the two agents swapping positions. If it is not possible to execute the swap operation for these two agents, then the problem is not solvable.

Experiments verify the promised advantages by comparing computation time against an optimized coupled A* alternative, as well as the fast and general decoupled method for cooperative path-finding [Silver, 2005]. The comparison includes small-scale, hard instances of cooperative path finding where the agents are highly coupled and a significant number of swaps per agent has to be executed. The decoupled approach cannot solve many of these problems and the A* solution is orders of magnitude slower than the proposed algorithm. The experiments also include larger-scale instances where hundreds of agents are placed randomly. The A* approach on the composite graph cannot provide solutions for these problems within a reasonable amount of time. When the decoupled approach works, it comes up with fast solutions but it cannot solve all of the problems. The "push-and-swap" algorithm solves all the large-scale problems in a computationally efficient manner. The length of solutions is competitive to the length of solutions by the decoupled approach [Silver, 2005] but the solution paths lend themselves to smoothing.

2 Setup and Notation

Consider a graph $\mathcal{G}(\mathcal{V},\mathcal{E})$ and n agents $\mathcal{R}$, where $n \leq |\mathcal{V}|-2$. An assignment $\mathcal{A} : [1,n] \rightarrow \mathcal{V}$ places all the agents 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 will be denoted as $\mathcal{S}$, while the target assignment will be 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 agent moves between neighboring nodes 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 cooperative path-finding is to compute a path $\Pi^* = \{\mathcal{S},\ldots,\mathcal{T}\}$, which is a similar sequence initiated with $\mathcal{S}$ and finishing with $\mathcal{T}$.

3 Push and Swap

This section proposes a novel, complete algorithm for the cooperative path-finding problem.

3.1 Algorithm

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

```
1: A ← S
2: Π* ← {A}
3: U ← ∅
4: for all r ∈ R do
5:    while A[r] ≠ T[r] do
6:       if PUSH(Π*, G, A, T, r, U) == FALSE then
7:          if SWAP(Π*, G, A, T, r, U) == FALSE then
8:             return ∅ (i.e., Failure)
9:    U ← U ∪ A[r]
10: return Π* (i.e., Success)
```

Push And Swap: Algorithm 1 provides the high-level operation of the approach. The PUSH_AND_SWAP method sets the current assignment $\mathcal{A}$ to the starting one $\mathcal{S}$ (line 1) and initiates the solution path Π^* by inserting the starting assignment (line 2). It also initializes the set $\mathcal{U}$ of agents that have already reached their targets to the empty set. Then for each agent r (line 4), PUSH_AND_SWAP tries first to push r to $\mathcal{T}[r]$ by clearing its path from other agents (line 6). If the push operation for r fails, then a swap operation is initiated (line 7). If the swap also fails, then the problem is not solvable (line 8). If r has not reached its target, then the algorithm keeps applying push and swap operations on r (line 5). When r reaches $\mathcal{T}[r]$, it is inserted to the set $\mathcal{U}$ of static agents, whose position must be respected (line 9). Eventually the algorithm returns the solution path Π^*, which is constructed by the calls to the PUSH and SWAP algorithms.

Push Operation: The PUSH algorithm first computes the shortest path p^* between the current assignment $\mathcal{A}[r]$ of the input agent r and the target assignment $\mathcal{T}[r]$ (line 1). The method will keep iterating as long as r has not moved to $\mathcal{T}[r]$ (line 3) and it can still make progress towards $\mathcal{T}[r]$ without any need for swapping (lines 12-13). If the graph vertices

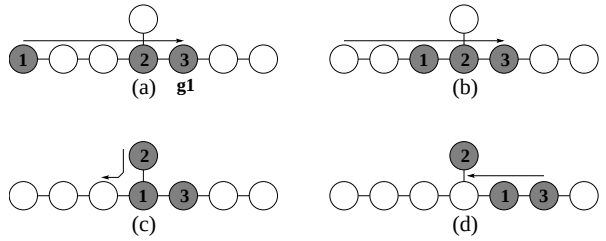

Figure 2: Illustration of the push primitive. Agent 1 pushes 2 twice in order to reach its target position.

along p^* are not occupied, then r is moved along these vertices and the corresponding intermediate assignments are stored on the solution path (lines 4-7). At this point, if r has already reached $\mathcal{T}[r]$, the subproblem for r is solved (line 8). Otherwise, the next vertex v along the shortest path p^* of r is occupied by another agent. Then the algorithm considers whether it is possible to push those agents along the shortest path p^* out of the way of r, without altering the position of r or any of the agents in $\mathcal{U}$, which have already reached their targets (lines 9-22). PUSH computes the shortest path p between the agent occupying vertex v and the closest empty vertex v_{empty} to v on G, given that $\mathcal{A}[r]$ and all vertices in $\mathcal{U}$ are obstacles (lines 9-11). If no path p can be found (lines 12-13), then agent r cannot push the agents out of its shortest path and cannot make progress without swapping. If a path is found (lines 14-22), then all of the agents along the shortest path are pushed one vertex forward towards v_{empty}, clearing v for r to occupy. Then the algorithm continues the pushing process given the new assignment of the agents.

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

```
1: p* ← SHORTEST_PATH(G, A[r], T[r])
2: v ← first vertex in p* after A[r]
3: while A[r] ≠ T[r] do
4:    while ∃ v and v is empty on G do
5:       A[r] = v
6:       Π* = Π* + {A}
7:       v ← next vertex in p*
8:    if A[r] ≠ T[r] then
9:       Mark A[r] and U as blocked on G
10:      v_empty ← closest empty vertex to v on G
11:      p ← SHORTEST_PATH(G, v, v_empty)
12:      if p == ∅ then
13:         return FALSE
14:      Mark A[r] and U as free on G
15:      v' ← last vertex on p before v_empty
16:      v'' ← v_empty
17:      while v'' ≠ v do
18:         r' ← agent for which A[r'] = v'
19:         A[r'] = v''
20:         Π* = Π* + {A}
21:         v'' = v'
22:         v' → previous vertex along p
23: return TRUE
```

Swap Operation: Overall, SWAP selects the agent s adjacent to the input agent r along r's shortest path to $\mathcal{T}[r]$ (lines 1-2), and switches their positions leaving agents already at their target positions intact. For a vertex v in $\mathcal{G}$ with degree ≥ 3, the algorithm computes the shortest path from $\mathcal{A}[r]$ to v (line 7). The algorithm then attempts to push agents r and s to v and one of its neighboring nodes correspondingly (line 9). This is achieved by a call to the function MULTIPUSH, which is analogous to PUSH. The difference is that instead of a single agent, MULTIPUSH moves a set of adjacent agents simultaneously. Furthermore, MULTIPUSH ignores the set $\mathcal{U}$ and is able to indiscriminately push all agents out of the specified path p. If the pushing operation succeeds, then $\mathcal{A}[r]$ is equal to v and $\mathcal{A}[s]$ is adjacent to v. In order for r and s to swap positions at v, any two adjacent vertices of v (excluding the vertex occupied by s) must be evacuated (line 10). This is the objective of function CLEAR. If two adjacent vertices of v cannot be cleared, then it is not possible for r and s to exchange positions at v, and another vertex is considered.

If the set of all vertices of degree ≥ 3 on $\mathcal{G}$ is exhausted and r and s cannot reach a vertex v, then SWAP returns failure (line 13). If the swap can take place, the sequence of actions Π computed in MULTIPUSH and CLEAR is added to the global solution (line 14), and the swap itself is performed (line 15). A graphical example of the swap primitive is shown in Fig. 1.

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

```
1: p* ← SHORTEST_PATH (G, A[r], T[r])
2: s ← agent on first vertex in p* after A[r]
3: success = FALSE
4: swap_vertices ← {All vertices of degree ≥ 3 on G}
5: while swap_vertices ≠ ∅ and success == FALSE do
6:    v = swap_vertices.POP ()
7:    p ←SHORTEST_PATH(G, A[r], v)
8:    Π ← ∅
9:    if MULTIPUSH (Π, G, A, T, {r, s}, p) == TRUE THEN
10:      IF CLEAR(Π, G, v, A[r], A[s]) == TRUE THEN
11:         success = TRUE
12: if success == FALSE then
13:    return FALSE
14: Π* = Π* + Π
15: EXECUTE_SWAP (Π*, G, A[r], A[s])
16: Π = Π.REVERSE (), exchanging paths for r and s
17: Π* = Π* + Π
18: if T[s] ∈ U then
19:    return RESOLVE (Π*, G, A, T, r, s)
20: return TRUE
```

Once the swap has been executed, the sequence of actions Π computed during the MULTIPUSH and CLEAR operations must be reversed so that the agents in $\mathcal{U}$ will be able to return to their targets at the end of SWAP. Note, that care has to be taken because r and s have swapped positions. Therefore, in order to successfully reverse the sequence of actions, paths executed by r will now need to be executed in reverse by s, and vice versa (line 16). The reversed sequence of actions is then added to the solution path Π^* (line 17).

Finally, it may happen that the agent s was in the set $\mathcal{U}$

and was already at its target position (line 18). In this case the assignment of s must be adjusted so that $\mathcal{A}[s]$ is equal to $\mathcal{T}[s]$. This is achieved by function RESOLVE (line 19). This step is discussed later in this section.

Clear Operation: CLEAR attempts to free two vertices in the neighborhood of a vertex v that has degree three or more for the purposes of SWAP. For brevity, a detailed algorithm is not given, but the general procedure is described here.

Assume that two adjacent agents, r and s wish to switch positions, and currently occupy vertex v and a neighbor of v. Two other vertices in the neighborhood of v must be freed in order for r and s to swap positions. There are three cases to consider when evacuating a agent a from the neighborhood of v, seen in Fig. 3. Case 1 involves simply pushing a to a neighboring vertex v or one of its neighbors. If case 1 fails to free two vertices, but there exists one free vertex in the neighborhood of v, case 2 can be employed. In case 2, r is pushed toward s to free v temporarily so that a can move through v to its unoccupied neighbor v'. Once a is at v', it can attempt another push away from v in an effort to clear the neighborhood.

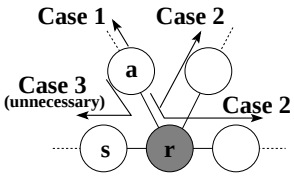

Figure 3: The three cases considered when evacuating a from the neighborhood of v (shaded).

Case 3 is an extension of case 2, where agent a attempts to evacuate the neighborhood of v through the vertices occupied by agents r and s. This case is unnecessary to consider when checking if a SWAP is feasible between r and s at v because it requires a to swap with r and s. If it is possible to swap a with r and s at v, then there must be two other free vertices in the neighborhood of v, and it isn't necessary to evacuate a. Therefore, for a to swap with r and s, a second swapping vertex, $\dot{v}$ must be employed. However, 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 can 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 to consider.

Resolve Operation: When RESOLVE is invoked, an agent r is swapped with an agent s, but s was already at its goal (e.g., $\mathcal{A}[s] \in \mathcal{U}$). There are a few cases to consider in order to get agent s back to its target, while allowing r to make progress:
1) Use PUSH to move r further along its path, freeing its current vertex, the target of s. Agent s can push to its target.
2) If a PUSH fails for r it needs to be swapped a second time with the agent blocking its shortest path. If the SWAP succeeds, then the PUSH from case 1 is repeated. If this fails, r is swapped again, repeating case 2.

It can happen that r swaps all the way to its target. In this case, r has swapped with a target r' that was occupying its target, and r' must then continue the resolution process in order to free the goal of s. If any SWAP from case 2 fails, then the resolution process is not possible, making the original SWAP that invoked RESOLVE invalid.

3.2 Completeness

Theorem 3.1. PUSH_AND_SWAP is complete for cooperative path-finding problems where the number of agents n is less than or equal to $|\mathcal{V}| - 2$.

The following discussion provides a sketch of the proof due to space limitations. This sketch brings together three ideas that are necessary to provide completeness in this formulation: 1) If SWAP fails, then the problem is not solvable, 2) Once an agent has reached its target, PUSH and SWAP always ensure this agent remains there, and 3) If the initial configuration is solvable, there always exists a path for an agent to reach its target, regardless of where it is pushed.

Lemma 3.2. A cooperative path-finding problem is solvable if and only if SWAP can bring agents r and s to a vertex v with a degree ≥ 3 along with two empty vertices.

Consider the sequence of vertices along the shortest path of r to its target $\mathcal{T}[r]$, 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 target $\mathcal{T}[r]$, 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 agents can be swapped.

SWAP exhaustively searches all the vertices of degree 3 or more, checking whether it is feasible for adjacent agents r and s to reach a vertex using MULTIPUSH, apply CLEAR at the vertex, and execute a swap.

There are two cases to consider when using MULTIPUSH to navigate a composite agent A composed of r and s to v: the initial position of the A lies in a cycle of the graph with v, or no such cycle exists. In the case of the cycle, assuming that at least one vertex in the cycle is free, agent A can simply traverse the cycle to arrive at v; MULTIPUSH does not exclude vertices in $\mathcal{U}$. If A does not lie in a cycle with v, then the feasibility becomes a "packing" problem. If the number of agents that lie along the shortest path for A to reach v is less than or equal to the number of reachable free vertices for those agents, then the "packing" problem will succeed, allowing A to reach v. Reachable, in this instance, is defined as all vertices that can be occupied without moving through a vertex in the path of A. If this problem is not solvable, then it is not possible for A to reach v.

From the construction of CLEAR, given in the algorithm description, all relevant possibilities to free the neighborhood of a vertex v are attempted. Therefore, if the set of vertices of degree 3 or more, reachable by the composite agent A is exhausted, and CLEAR fails for each vertex, then it is not possible to swap the positions of agents r and s, making the cooperative path-finding instance unsolvable.

Lemma 3.3. After the application of one iteration of PUSH and SWAP primitives, at least one agent has made progress toward its target, and agents at their targets will remain there.

PUSH, by construction, excludes vertices in the set $\mathcal{U}$ when searching for positions to push agents blocking a particular path. If a vertex in the path to PUSH along exists in $\mathcal{U}$, the method returns a failure, indicating a SWAP is necessary.

SWAP does not exclude any vertices in $\mathcal{G}$ when attempting to switch the positions of two adjacent agents. When applying MULTIPUSH and CLEAR in attempting to bring the two agents to a vertex where a swap can take place, potentially all other agents already at their target could be disturbed during this process. Therefore, all actions executed during MULTIPUSH and CLEAR are reversed after the swap is executed (line 16), bringing all agents that were at their target back to them.

When an agent a makes progress along its path using PUSH it isn't disturbing any other agents already at their targets, and progress is made in computing the solution. However, if a is blocked, then PUSH makes no progress in advancing a. Agent a must swap positions with the blocking agent to continue toward its target. By construction of SWAP any agents already at their targets that must be moved to accommodate the swap will be returned there during the action reversal. At the end of SWAP, a has swapped positions along its shortest path, making progress toward its target.

Lemma 3.4. If the initial agent configuration is solvable, any configuration reached using PUSH and SWAP operations will remain solvable.

An agent a' may be pushed away from its initial vertex during the planning of other agents. Because of this displacement, it may seem that there is no path for a' to travel from its current vertex to its target. Note, however, that it is possible for a' 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 a' to return to its initial vertex. In the worst case, a' will have to return to its initial vertex, then follow the original path to its target. Lemma 3.3 shows that an iteration of both the PUSH and SWAP operators will allow an agent to make progress in a solvable instance. If the initial configuration is solvable, then it is possible for a' to reach its target. The solvability of an instance depends only on the initial configuration, because any configuration achieved through PUSH_AND_SWAP can be reversed.

4 Evaluation

This section evaluates the performance of PUSH_AND_SWAP, and provides a comparison with the coupled A* and the decoupled WHCA* algorithm [Silver, 2005] in terms of scalability and robustness. In order to evaluate the proposed approach, a series of challenging instances of cooperative pathfinding were created. These problems include a set of small benchmarks that tend to be difficult for a decoupled planner, as well as much larger instances in a randomly generated maze environment. All experiments were performed on a Core 2 Duo 2.5GHz machine with 4GB of memory.

Benchmark problems: A series of small, benchmark problems (Fig 4) were created, ranging in size from 3 to 7 agents. Because of the small-scale of these problems, a comparison of the PUSH_AND_SWAP technique with a complete, coupled A* planner can be shown. Results are also provided for the decoupled WHCA* approach using a window size of 5. Table 1 shows the time needed for the three approaches to compute their respective solutions. Note that times of more than 60 minutes are declared a failure for the coupled A* approach. If the WHCA* gets stuck to a local minimum, this is also reported as a failure. As expected, the coupled approach is able to compute solutions for the smaller problems, but quickly grows infeasible as the number of agents exceeds just a handful. The WHCA* planner is able to quickly compute solutions for 3 of the 6 benchmarks; the 3 failures can be attributed to a high degree of coordination needed to solve those particular problems. PUSH_AND_SWAP is able to quickly compute solutions to all six benchmark problems, supporting the completeness property given in Section 3.

	Coupled A^*		WHCA*(5)		Push and Swap	
Problem	Time	Size	Time	Size	Time	Size
Tree	2.80	15	0.68	34.4	0.86	39
Corners	3882	36	0.53	36	1.30	60
Tunnel	417	53	∞	n/a	1.36	145
String	207	20	1.04	36.1	0.77	39
Loop/Chn	∞	n/a	∞	n/a	2.99	523
Connector	∞	n/a	∞	n/a	3.57	108

Table 1: The computation time (ms) and solution length for the benchmarks. ∞ represents a failure to compute a solution.

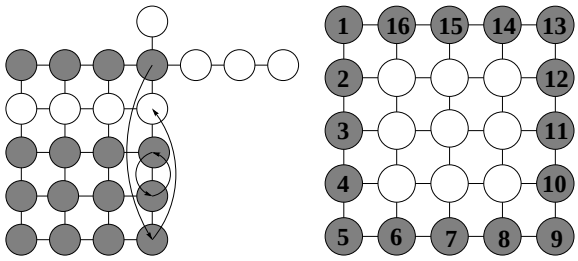

Figure 5: Left: The "stacks" benchmark problem with 16 agents. Left: Rotation problem with 16 agents. All agents must move one vertex counterclockwise in the loop.

One larger benchmark asks for multiple "stacks" of agents to reverse their position in each stack. The 16-agents version of the problem in Fig. 5 (a) is infeasible for the coupled A* planner. It also proved difficult for the decoupled WHCA* approach, which was consistently unsuccessful in computing a solution with various window sizes. PUSH_AND_SWAP, on the other hand, computed a solution in just 5.7 mS.

Scalability: As the number of agents increases, the failure rate of traditional decoupled techniques increases rapidly because the level of coordination required grows larger.

The first scalability experiment tests an environment with the agents arranged in a circular pattern, with the interior of the environment free. The target for each agent is adjacent to the initial position, with the final result achieving a rotation of all agents by one vertex. Figure 5(b) shows an example

	WHCA*(5)		Push and Swap	
# Agents	Time (ms)	Length	Time (ms)	Length
12	2.13	1.79	1.136	3.00
16	3.10	1.87	1.23	1.12
20	5.06	2.01	2.19	2.50
24	9.59	1.93	2.56	2.08

Table 2: The computation time (milliseconds) and average individual solution length for the rotation problem.

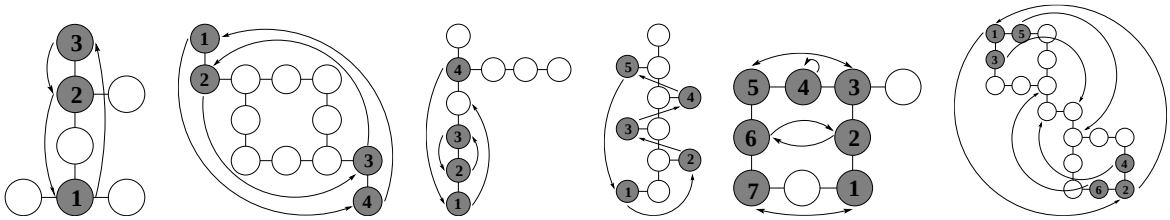

Figure 4: The set of benchmark tests. From left: Tree, Corners, Tunnel, String, Loop-Chain, Connector.

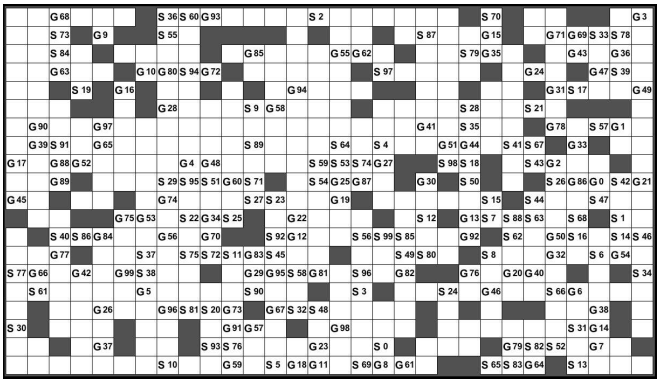

Figure 6: A randomly generated environment with 100 agents. Agent i must navigate from Si to Gi.

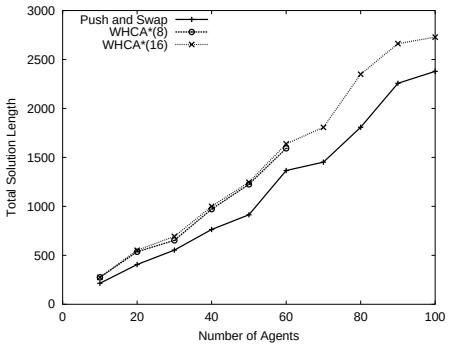

Figure 8: The average total solution length computed by the WHCA and PUSH_AND_SWAP approaches in the random placement problems. Values are averages of 20 runs.

of such an experiment with 16 agents. Table 2 shows the computation time for the WHCA* algorithm with a window size of 5 compared with the PUSH_AND_SWAP technique.

The next experiment compares WHCA* with the PUSH_AND_SWAP technique on a randomly populated grid-world with 20% obstacle coverage. Agents with random and mutually exclusive initial and target configurations are placed in this environment. Figure 6 shows an example of this experiment with 100 agents.

Figure 7(a) shows the computation time needed to compute a solution to the random placement problem with varying numbers of agents. WHCA* was executed with two window sizes, 8 and 16. The PUSH_AND_SWAP technique has computation times competitive with the WHCA* approach with the window size of 8. However, the small window size for coordination in the decoupled approach quickly degrades in its ability to compute a solution. The probability of WHCA* (8) to compute a solution decays rapidly after 40 agents in the environment. WHCA* with window size 16 is able to solve much larger numbers of agents in the same environment, but suffers from the same decay as the smaller window, just at a larger number of agents. Fig 7(b) shows the percentage of experiments for which the methods were able to compute a solution. Note that the PUSH_AND_SWAP technique has 100% success, regardless of the number of agents, with computation times well under 10 seconds for 100 agents.

The final scalability experiment focuses on tightly coupled problems requiring a high degree of coordination between agents. Such problems are not usually solvable by traditional decoupled techniques and require a more centralized approach. These experiments take the Loop-Chain benchmark problem and resize the loop so that more agents can be placed inside. For these benchmarks there is a high degree of coupling between all agents, and all agents will have to make individually suboptimal choices in order to solve the global problem. Figure 7(c) shows the computation time of the PUSH_AND_SWAP approach as increasing number of agents are solved in this type of problem. From the figure, it can be seen that the computation time for even 100 agents in this particularly difficult instance remains tractable. The WHCA* approach is unable to solve even the smallest version of this problem, shown in Fig. 4 (Loop-Chain).

Solution Quality: Although there are no solution quality guarantees when using PUSH_AND_SWAP, a direct application of the algorithm in all of the experiments provided qualities that were competitive in the smaller benchmark problems, and noticeably better in the random placement problems when compared to WHCA*. Tables 1 and 2 show that in the benchmark and single rotation problems, the total solution length computed by both methods were similar. Quantifiable improvements are seen in the random grid environment. PUSH_AND_SWAP is able to compute solutions roughly 20% shorter than WHCA* in both window sizes. Figure 8 shows the average solution length computed for each approach. Generally, solution quality for PUSH_AND_SWAP is dependent on the number of swaps that must be performed in order to solve the problem. In problems where PUSH can be liberally employed, like a random grid, the solution quality will be significantly better than highly constrained problems, like the Loop/Chain benchmark, where many swaps are performed in order to compute the solution.

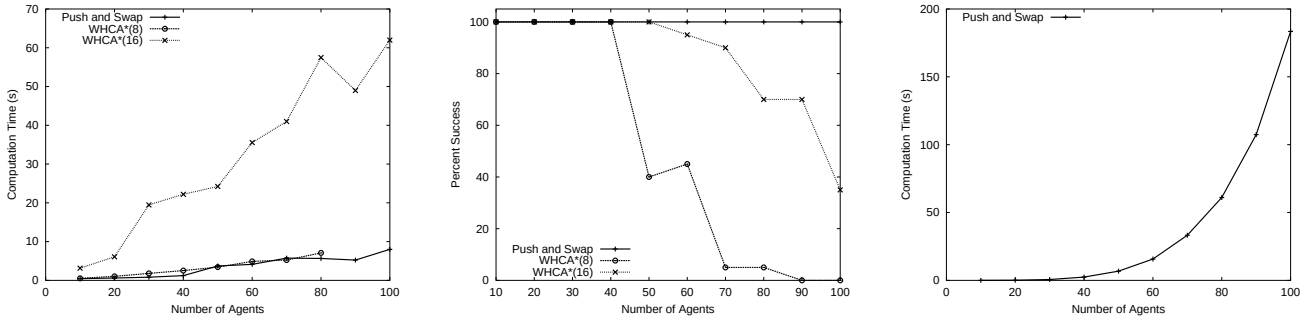

Figure 7: (left) The computation time for varying numbers of agents placed randomly in an environment. All values are an average of 20 runs. (middle) The success rate for WHCA and PUSH_AND_SWAP in the random experiments. All values are an average of 20 runs. (right) The computation time for the PUSH_AND_SWAP algorithm as the Loop-Chain example grows.

5 Discussion

This paper presented a computationally efficient and complete approach for solving instances of cooperative pathfinding for problems with at least two empty vertices in the graph. Through the combination of two basic primitives, the algorithm can solve a broad set of problems as fast as a well established decoupled planner, without any dependence on parameter selection. Unlike decoupled approaches, the proposed PUSH_AND_SWAP algorithm is able to solve instances where the agents are fully coupled, as well as problems where agents may need to repeatedly move away from their goal position in order to solve the global problem.

The proposed algorithm can potentially be extended to solve problems where there is only a single empty node as long as there are redundant loops in the graph (e.g., the case of the 15-puzzle problem). This requires an extension to the swap primitive to take advantage of cycles in the roadmap in order to maintain the invariant for all agents not swapping.

The PUSH_AND_SWAP approach computes a sequential solution and does not aim to provide path quality guarantees. Nevertheless, the agents that are pushed are always moving along the shortest paths to their goals. Some redundancy in motion is introduced by consecutive calls to the SWAP function. This redundancy can potentially be avoided by a more complex version of this function. In general, sequential solutions can be smoothed and parallelized during a post processing step, which is an interesting direction to investigate. It must be emphasized that there are competing notions of path optimality for cooperative path finding, such as the sum of the path costs for all agents or notions related to Pareto optimality, which can be used to evaluate the quality of a solution.

References

[Bennewitz *et al.*, 2002] M. Bennewitz, W. Burgard, and S. Thrun. Finding and optimizing solvable priority schemes for decoupled path planning for mobile robots. *Robotics and Autonomous Systems*, 41(2):89–99, 2002.

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

[Peasgood *et al.*, 2008] M. Peasgood, C. Clark, and J. McPhee. A complete and scalable strategy for coordinating multiple robots within roadmaps. *IEEE Transactions on Robotics*, 24(2):282–292, 2008.

[Qutub *et al.*, 1997] 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)*, volume 3, pages 1610–1615, 1997.

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

[Silver, 2005] David Silver. Cooperative pathfinding. In *The 1st Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE'05)*, pages 23–28, 2005.

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

[Surynek, 2009] Pavel Surynek. Making solutions of multi-robot path planning problems shorter using weak transpositions and critical path parallelism. In *International Symposium on Combinatorial Search*, 2009.

[van den Berg *et al.*, 2009] 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.

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

[Wang and Botea, 2009] 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*, pages 1870–1875, Pasadena, CA, USA, 2009.