Journal of Artificial Intelligence Research 51 (2014) 443-492 Submitted 05/14; published 10/14

Push and Rotate: a Complete Multi-agent Pathfinding Algorithm

Boris de Wilde BOREUS@GMAIL.COM
Adriaan W. ter Mors A.W.TERMORS@TUDELFT.NL
Cees Witteveen C.WITTEVEEN@TUDELFT.NL
Faculty of Electrical Engineering, Mathematics, and Computer Science
Mekelweg 4, 2628 CD Delft, The Netherlands

Abstract

Multi-agent Pathfinding is a relevant problem in a wide range of domains, for example in robotics and video games research. Formally, the problem considers a graph consisting of vertices and edges, and a set of agents occupying vertices. An agent can only move to an unoccupied, neighbouring vertex, and the problem of finding the minimal sequence of moves to transfer each agent from its start location to its destination is an NP-hard problem.

We present Push and Rotate, a new algorithm that is complete for Multi-agent Pathfinding problems in which there are at least two empty vertices. Push and Rotate first divides the graph into subgraphs within which it is possible for agents to reach any position of the subgraph, and then uses the simple `push`, `swap`, and `rotate` operations to find a solution; a post-processing algorithm is also presented that eliminates redundant moves. Push and Rotate can be seen as extending Luna and Bekris's Push and Swap algorithm, which we showed to be incomplete in a previous publication.

In our experiments we compare our approach with the Push and Swap, MAPP, and Bibox algorithms. The latter algorithm is restricted to a smaller class of instances as it requires biconnected graphs, but can nevertheless be considered state of the art due to its strong performance. Our experiments show that Push and Swap suffers from incompleteness, MAPP is generally not competitive with Push and Rotate, and Bibox is better than Push and Rotate on randomly generated biconnected instances, while Push and Rotate performs better on grids.

1. Introduction

Computer scientists and roboticists have long studied the problem of coordinating the motions of multiple moving objects. A general formulation of the problem is to find conflict-free trajectories in space and time for each of the objects. Cast as the warehouseman's problem, this problem was proved PSPACE-complete by Hopcroft et al. (1984). The problem's complexity can be reduced to NP-complete by assuming that agents can only move along a graph (Goldreich, 1993). The graph (or roadmap) can be given, for instance in applications such as automated guided vehicles following lines drawn on factory floors (Roszkowska & Reveliotis, 2008), but it can also be learned (Kavraki, Svestka, Latombe, & Overmars, 1996) or otherwise constructed (LaValle & Kuffner, 2001).

Application domains of multi-agent pathfinding include many forms of robotics, for instance mobile robots (Siméon, Leroy, & Laumond, 2002) and robot arms (Erdmann & Lozano-Pérez, 1987); the routing of automated guided vehicles, for instance at container terminals (Vis, 2006; Gawrilow, Köhler, Möhring, & Stenzel, 2007) or in warehousing or manufacturing (Narasimhan, Batta, & Karwan, 1999; Desaulniers, Langevin, Riopel, & Villeneuve, 2004); video games, such as role-playing games and real-time strategy games (Nieuwenhuisen, Kamphuis, & Overmars, 2007); airport taxi routing (Trüg, Hoffmann, & Nebel, 2004; Ter Mors, Zutt, & Witteveen, 2007) and collision avoidance of airplanes in flight (Šišlák, Pěchouček, Volf, Pavlíček, Samek, Mařík, & Losiewicz, 2008); but routing and multi-agent pathfinding also occurs in 'less obvious' domains such as planning for mining carts (Beaulieu & Gamache, 2006) or routing sheets of paper through a modular printer (Ruml, Do, Zhou, & Fromherz, 2011).

As remarked by Surynek (2011), the applicability of algorithmic approaches depends on the freedom the agents have in their respective application domains. Incomplete reservation-based approaches, where agents typically plan independently, minding the reservations of the others (Lee, Lee, & Choi, 1998), can be useful in case there is enough room in the infrastructure to guarantee (or at least to make it highly likely) that a solution can be found. In this paper, we discuss approaches that can also deal with more congested scenarios, in which the majority of the locations can be occupied by agents. An inspiration for such a situation are real-time strategy computer games, in which large numbers of units, both friendly and hostile, are competing for room to move around (see Figure 1).

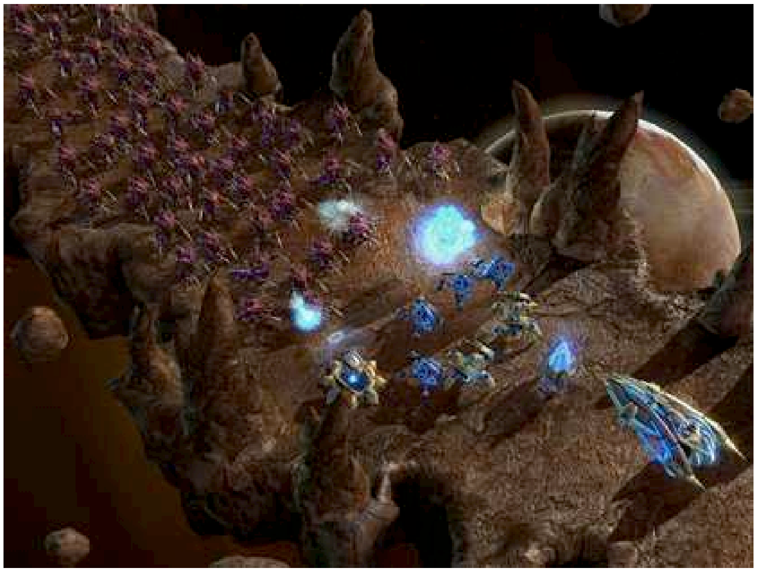


Figure 1: The Zerg attack the Protoss in Starcraft 2. The maps of the original Starcraft have been made available for research at `http://movingai.com/benchmarks/`.

Kornhauser (1984) gave a complete algorithm for the multi-agent pathfinding problem, as well as lower and upper bounds of $O(n^3)$ (where n is the number of vertices in the graph) for the number of moves required in a solution. In recent years, a number of approaches have appeared that solve subclasses of the multi-agent pathfinding problem (see Section 2.1). Röger and Helmert (2012) suggested that one of the reasons that the results from Kornhauser have not been more widely applied is:

> … the approach is not described in one place, and most of its parts are not described algorithmically. Therefore, the underlying algorithm must be derived from a number of proofs in the paper.

The Push and Swap algorithm (Luna & Bekris, 2011b) is a complete algorithm for instances having at least two unoccupied vertices, and is therefore the most general of the recent algorithms. However, we found a number of problems with the Push and Swap algorithm (De Wilde, Ter Mors, & Witteveen, 2013), and presented our own Push and Rotate algorithm to overcome these shortcomings.

Our primary aim in this paper is to provide a complete and understandable specification of an algorithm for the multi-agent pathfinding problem with at least two unoccupied vertices. More specifically, this paper presents the following contributions:

1. A specification of a complete algorithm for multi-agent pathfinding, which involves a reconstruction, and in some cases a refinement of some of the theoretical results from Kornhauser (1984).

2. An empirical evaluation of our algorithm, which includes a comparison with the Bibox algorithm (Surynek, 2009), an algorithm that assumes biconnected instances, and which is among the best performing of the recently developed approaches described in Section 2, both in terms of number of moves and CPU time required (judging by the empirical evaluations in Surynek, 2009; Wang & Botea, 2011).

3. A brief exploration of the possibilities to improve solution quality and reduce computation times through the use of heuristics, namely in the area of selecting which agent to plan next, and which path each agent should choose.

Part of the first contribution has appeared in a conference paper (De Wilde et al., 2013), namely Algorithms 1, 2, 3, and 8 and some of the proofs. The reconstruction of Kornhauser's theoretical results, as well as a complete listing of relevant algorithms and proofs are new to this paper, as are contributions two and three (a different, more restricted empirical evaluation was conducted for the conference paper).

This paper is organized as follows. In Section 2, we define the multi-agent pathfinding problem, discuss its complexity and a number of algorithmic approaches. In Section 3, we describe our Push and Rotate algorithm, and in Section 4 we prove its completeness and analyze its worst-case behavior. In Section 5, we describe heuristics for path and agent selection, before describing the experiments in Section 6. The experiments comprise a comparison with the Bibox and Push and Swap algorithms on different types of randomly generated instances. Finally, we finish with conclusions and future work in Section 7.

2. Background and Problem Statement

We consider a simple[1] connected[2] graph $\mathcal{G} = (V,E)$, a set of agents A, with $|A| < |V|$, an assignment function $\mathcal{A}$ of agents to vertices $\mathcal{A} : A \rightarrow V$, and a goal assignment of agents to vertices $\mathcal{T} : A \rightarrow V$. The functions $\mathcal{A}$ and $\mathcal{T}$ are total, injective, and non-surjective; they are total functions as the location

---

[1] A graph is *simple* if there is at most one edge between any two vertices; multiple edges between two vertices would not expand the solution space, since we assume that an agent must always move to an empty vertex.

[2] If the graph is not connected, then the Multi-agent Pathfinding problem can be considered for each of the components separately. If there exists an agent for which the destination location is in a different component from the start location, then the instance has no solution.

of each agent must be specified, injective because one vertex can hold only a single agent at a time, and non-surjective as we require that there are always more vertices than there are agents.

A move is to transfer an agent $a_i \in A$ from its current vertex $v = \mathcal{A}(a_i)$, to an adjacent, unoccupied vertex w, $\mathcal{A}^{-1}(w) = \perp$. We define the MULTI-AGENT PATHFINDING problem as an optimization problem: to find a sequence Π of moves that transforms the initial assignment to the goal assignment, such that $|\Pi| \leq |\Pi'|$ for any sequence of moves Π' that transforms $\mathcal{A}$ into $\mathcal{T}$.

The decision variant of the Multi-agent Pathfinding problem (i.e., does there exist a sequence of K moves that transforms the initial assignment to the goal assignment) was shown to be NP-complete (Goldreich, 1993)[3]. The NP-completeness of the problem holds in case there is only one agent with a destination location, with all other agents being obstacles that may be moved out of the way (Papadimitriou, Raghavan, Sudan, & Tamaki, 1994) (i.e., the goal assignment is a partial function with a domain of size 1). Our Push and Rotate algorithm that we will present in Section 3 will find move sequences for the Multi-agent Pathfinding problem with at least two unoccupied vertices, although it does not guarantee optimal solutions, as that would require exponential running time, assuming P $\neq$ NP.

Goraly and Hassin (2010) presented a variant of the Multi-agent Pathfinding problem in which there are m colors, and each of the p agents has a color. The goal configuration specifies for each vertex which color the occupying agent must have (or whether the vertex should remain empty); the authors show that feasibility can be decided in linear time (i.e., it can be decided whether an instance has a solution), also in the case that $m = p$. Goraly and Hassin build on the work of Auletta et al. (1999), who presented an algorithm for deciding the feasibility of Multi-agent Pathfinding problems on trees, in linear time.

Călinescu et al. (2008) also distinguish between different kinds of *chips*, namely unlabeled and labeled (where every label is unique), and they consider a different type of model where moving a chip along an empty path is considered a single move. They prove that even for unlabeled chips, the problem is APX-hard, and still NP-hard for instances on an infinite, rectangular grid. Finally, Wu and Grumbach (2009) considered directed graphs, which have the interesting property that a wrong move can put the problem in an unrecoverable configuration. Unrecoverable reconfigurations were also at the heart of the PSPACE-completeness proof of the Sokoban puzzle game (Culberson, 1999; Hearn & Demaine, 2005); Wu and Grumbach do not consider the complexity of the optimization problem, however, but instead prove that feasibility can be decided in $O(n^2m)$ time, where n is the number of vertices, and m the number of arcs.

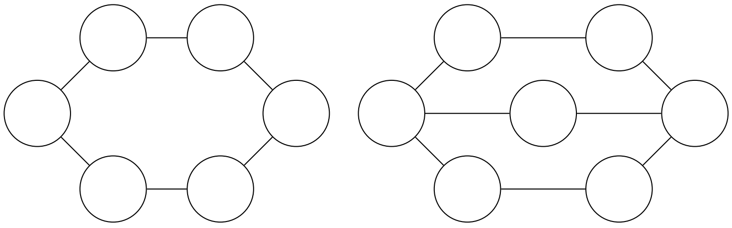
Early research into Multi-agent Pathfinding focused on the feasibility of reaching one assignment from another. Wilson (1974) studied simple biconnected[4] graphs with a single empty vertex, and he shows that any assignment can be reached from any other, except when the graph is bipartite, in which case there are two sets of assignments that can only reach the assignments in the same set. Wilson's theorem includes two exceptions that are not generally solvable: polygon (or cycle) graphs, and, remarkably, a single specific graph θ_0 (Figure 2).

Kornhauser et al. (1984) extend Wilson's result to general graphs with any number of unoccupied vertices and provide a polynomial-time decision procedure with an $O(n^3)$ bound on the number of moves required. The result by Kornhauser is based on the insight that a graph may be viewed as

---

[3] Goldreich proves the NP-completeness of the Shortest Move Sequence problem which is equivalent to our definition of MULTI-AGENT PATHFINDING, with the exception that Goldreich considers biconnected graphs.

[4] A *biconnected* graph is a connected graph with no *articulation vertices*, i.e., vertices whose removal will disconnect the graph; such graphs are also called nonseparable graphs.



(a) A polygon graph. (b) Wilson's θ_0 graph.

Figure 2: Biconnected graphs that are not generally solvable for a single empty vertex.

a tree of biconnected components that are linked by chains of zero or more vertices with degree 2 (called *isthmuses*). It turns out that if two of these components are linked by an isthmus that contains more vertices than the number of unoccupied vertices minus two, then it is impossible for agents in components on different sides of the isthmus to swap positions.

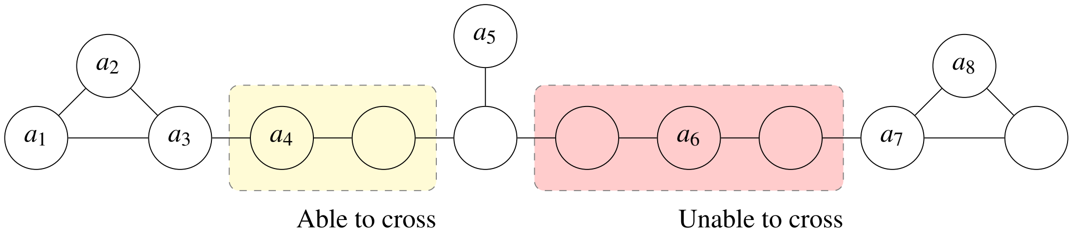


Figure 3: Illustration of isthmuses connecting nonseparable components.

In Figure 3, note that it is possible for any of the agents in $A_1 = \{a_1, \dots, a_6\}$ to swap positions, and also the agents in $A_2 = \{a_7, a_8\}$ can exchange positions, but no agent from A_1 can exchange position with any agent in A_2. This means to say that, if we take agents a_6 and a_7 as an example, then it is not possible to reach a configuration in which a_6 occupies the position of a_7 in Figure 3, while at the same time a_7 occupies the current position of a_6. The isthmuses that are impossible to cross induce a decomposition of the graph into smaller subgraphs, which can be solved in turn. We will demonstrate this decomposition in Section 3.1.

2.1 Algorithms

Standley (2010) proposed an algorithm that guarantees optimal[5] solutions for multi-agent pathfinding in which the agents move in an eight-connected grid. In each time step, an agent either moves to an adjacent grid cell or stands still, and the cost for a single agent is the number of time steps it takes to reach its goal; the cost of the entire solution is the sum of all agent plan costs. Moves are not only allowed to empty grid cells, but also in a cycle of agents where each agent moves to the cell of the next agent in the cycle. Hence, Standley's algorithm can also be applied in problem instances where the number of agents is equal to the number of vertices.

---

[5] In this section, we will refer to an approach as *optimal* if it returns a solution consisting of a minimal number of moves.

Standley's approach can be seen to improve on a standard A*-based approach in which the state is an n-tuple of grid locations, one for each of the n agents, and in which the moves of all agents are considered simultaneously in one timestep, so each state has potentially 9^n legal operators (nine for each of the eight moves plus the wait move). Standley's *operator decomposition* divides each time step such that one agent is considered at a time, thereby reducing the branching factor from 9^n to 9, although the search depth increases by a factor of n. Coupled with a perfect heuristics and a perfect tie-breaking strategy, the operator decoupling would reduce the number of required steps from $9^n \times d$, for a search depth d, to $9n \times d$. Hence, the operator decoupling scheme requires a good heuristic to save computation time compared to the standard approach.

In further work, the operator decomposition technique is employed to create an anytime algorithm, based on the maximum group size of agents for which optimal solutions are found (Standley & Korf, 2011). The anytime algorithm starts with a group size of one, and increases the group size by one step when a solution is found, looking for more efficient solutions. The stand-alone operator decomposition algorithm struggled to find solutions to problems larger than a 32×32 grid with ten agents, but the anytime algorithm was able to produce good-quality solutions for more than a hundred agents.

Another optimal approach is due to Sharon et al. (2011), which is called Increasing Cost Trees (ICT). In ICT, there is a high-level search through a tree where each node consists of a k-vector of individual path costs, and represents all possible solutions in which the cost for an agent equals the value in the cost vector. To create a child node, a unit cost is added to the cost of one of the agents. A low-level search is then performed to see if the new node can bring all agents to their destinations. ICT can be efficient in scenarios with low interaction between the agents, in which all agents can reach their destinations with a small number of extra moves; otherwise, Standley's operator decomposition is more efficient.

Another optimal approach by Sharon et al. (2012) is meta-agent constraint-based search. In constraint-based search, a high-level search is performed in a constraint tree, where nodes are constraints on individual agents, and a low-level search to find individual agent paths that respect the constraint of the high-level node. The constraint-based approach performed poorly on some types of problems, however, so the authors devised the meta-agent strategy, in which groups of agents with many internal conflicts are merged into one meta agent, in order to reduce the number of conflicts at the high-level constraint search.

Given the NP-completeness of the problem, many papers employ a sequential approach, in which agents are planned for one after the other, to obtain a polynomial-time algorithm that is not necessarily optimal. To ensure that the planning of agent n does not disrupt all the work done to put agents $1, \ldots, n-1$ into position, existing algorithms either reserve time slots on nodes, or previous agents are restored to their positions after the planning of agent n.

Reservation-based approaches are typically not complete, in the sense that the reservations made by the first n agents can make it impossible for agent $n+1$ to find a plan (even if a multi-agent plan for all $n+1$ agents could be found by other means), and as such these approaches are not particularly suited to instances in which the ratio between the number of agents and the number of empty vertices is high. In context-aware routing (Ter Mors, Witteveen, Zutt, & Kuipers, 2010), agent n finds an optimal route plan (i.e., a path plus the times at which it will arrive at each node in the path) around the reservations of the first $n-1$ agents, but it assumes that agents only enter the graph at the start of their first reservation, and leave the graph at their destination. Alternatively, it can be assumed that all start and destination locations are 'parking places', and can hold an infinite number of agents (Zutt & Witteveen, 2004).

Velagapudi, Sycara, and Scerri (2010) use a reservation-based system to create a distributed cooperative routing algorithm. Their problem definition is more general, as they simply assume a set of robots that are looking for trajectories within a given time horizon, a binary obstacle map O, and a function COLLISIONCHECK that takes two robot trajectories and returns *true* if these collide. In their distributed routing algorithm, once an agent has found a route, it broadcasts this route to all other agents. During route planning, an agent has to take into account the routes that higher-priority agents have broadcast. In addition, if an agent receives a higher-priority route that conflicts with its own route, it will have to re-plan.

Silver (2005) presents a windowed approach in which the agents only make reservations for a restricted time horizon. Silver claims the following three advantages. First, the agents can continue cooperating after they reach their destination vertices (instead of staying put and blocking their destination vertices). The second advantage is that the sensitivity to agent ordering (or prioritization) is reduced, as different agent priorities can be assigned for different time periods. Finally, there is no need to plan for long-term contingencies that may not occur.

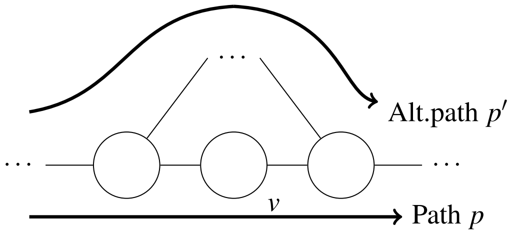


Figure 4: For the SLIDABLE class of instances, there must exist, for every path p and every vertex v on p, an alternative path p' connecting the predecessor and successor of v.

The MAPP algorithm (Wang & Botea, 2008) brings agents to their destination one by one along a pre-computed path, where lower-priority agents may be temporarily pushed aside. MAPP requires a number of restrictions on the instances for which it guarantees to find a solution; the SLIDABLE class of instances requires:

1. for each node in an agent's start-destination path (except for the start and destination), there must exist an alternative path connecting its predecessor and successor nodes (see Figure 4),
2. the first node (after the start node) in the first agent's path must be empty,
3. no target location may intersect with any of the paths and alternative paths of any of the agents.

Khorshid et al. (2011) present a Tree-based Agent Swapping Strategy (TASS): in each iteration of the algorithm, an agent is selected to be sent to its goal, which is accomplished by swapping it with other agents. Swapping two agents is accomplished by moving both agents to a junction — a node of degree three or more — and ensuring that at least two neighbors of the junction are empty. After the swap has been performed, both the empty vertices and the other agents that have moved are returned to their original locations, such that only the two swapping agents will have been affected.

To solve instances on general graphs, the authors first employ a graph-to-tree decomposition algorithm. However, the decomposition is not complete in the sense that a solvable instance may not have a solution any more after the transformation to a tree. The authors also provide conditions under which an instance is guaranteed to be solvable, namely that the distance between any two junctions may not be greater than the number of empty vertices minus two, which turns out to be identical to the result obtained by Kornhauser (1984) when restricted to trees.

Surynek's Bibox algorithm (2009) requires (and is complete for) biconnected graphs with at least two unoccupied vertices. The Bibox algorithm makes use of the fact that each biconnected graph can be viewed as an original cycle, extended by a number of *handles* (see Figure 5; Kornhauser shows how any biconnected graph can be decomposed into handles). Agents with a destination in the outermost handle are brought to their destination first, after which that handle does not need to be taken into account any more, and the algorithm proceeds with the next handle. Surynek also implemented the approach by Kornhauser (1984), and reported that Bibox produced both lower running times and shorter paths.

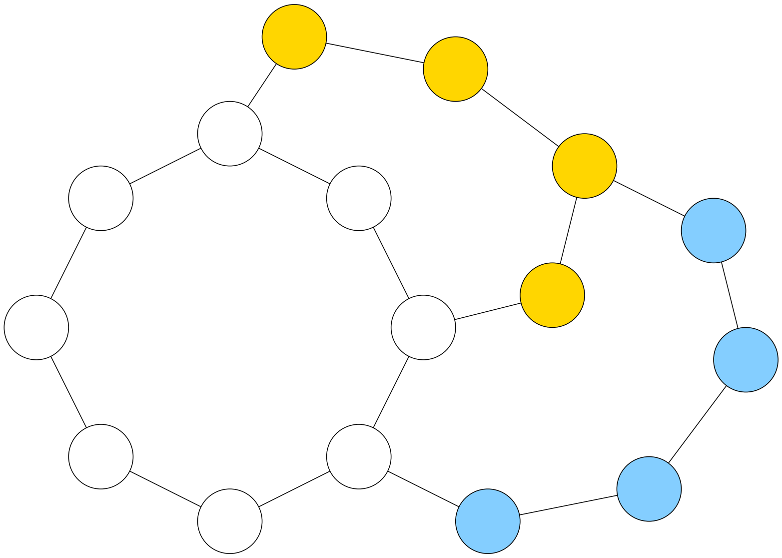


Figure 5: A graph consisting of an initial cycle, in white, and two handles in yellow and blue.

Finally, the Push and Swap algorithm (Luna & Bekris, 2011b) was presented as complete for any graph with two or more unoccupied vertices. The algorithm works by iteratively selecting agents in some unspecified priority order to move to their respective destination locations. For each agent, the algorithm will move it to its destination location along any shortest path. When other agents are encountered along this path, the action to be taken depends on the priority of the other agent. In case the other agent has lower priority, the algorithm will attempt to move it out of the way with the `push` operation. This can be accomplished by pushing the blocking agent forward along a shortest path (not containing any higher-priority agents) to an empty vertex. In case the blocking agent has a higher priority, the algorithm will attempt to exchange their positions using a `swap` operation that is similar to the swap operation in TASS (Khorshid et al., 2011). The higher-priority agent must be returned to its destination, for which the Push and Swap algorithm uses the `resolve` operation.

A parallel version of the Push and Swap was developed by Sajid, Luna, and Bekris (2012), with the aim of reducing the length of solutions produced by Push and Swap, in which all moves are sequential. The idea behind Parallel Push and Swap is to first resolve any dependencies between agents that require a `swap` operation, after which some of the `push` steps can be performed in parallel.

Although presented as complete, we showed that the Push and Swap algorithm is not complete, and contains the following shortcomings (De Wilde, 2012; De Wilde et al., 2013):

1. The algorithm does not identify *polygon* graphs, i.e., graphs consisting only of a single cycle. For a solvable polygon instance, Push and Swap will fail to find a solution if the wrong agent priority ordering is chosen; if an agent tries to move a higher-priority agent out of the way with a `swap`, the algorithm will fail, since a `swap` requires a vertex of degree ≥ 3, and in a polygon graph all vertices have degree 2 (see Figure 6 for an illustration).

2. To enable the `swap` operation, two neighboring vertices of a junction (in TASS terminology) must be emptied using the `clear` operation; the specification of `clear` by Luna and Bekris identifies only two of the four cases that must be distinguished.

3. When moving a vertex back to its destination after a `swap`, the `resolve` may invoke the `swap` operation again. Examples can be constructed in which these recursive calls result in a higher-priority agent being two steps removed from its destination, and Push and Swap may fail in those instances.

4. The Push and Swap algorithm does not take into account the result from Kornhauser (1984) that it is impossible for agents to swap if they are separated by an isthmus longer than the number of empty vertices minus two. The Push and Swap algorithm may fail in case an agent a_i must move out of the way for another agent a_j, and a_i has been assigned the higher priority, but a_i and a_j cannot swap.

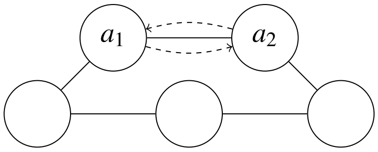


Figure 6: Regardless of which agent moves first, the second agent will try to reach its goal along the shortest path, which is blocked by the other, higher-priority agent. A `swap` will be attempted but there are no vertices of degree ≥ 3 to swap at.

The first shortcoming was resolved in a personal communication with the author: instead of always choosing the shortest path, an agent must choose the shortest path to its destination that does not contain any finished agents. The second shortcoming is rectified by our updated `clear` operation (Algorithm 12, Appendix A). To fix the third point, we changed the way in which an agent is returned to its position after a `swap`, and we introduce a new operation `rotate` to accomplish this. To address the fourth point, we studied the theory on problem decomposition by Kornhauser (1984), and developed new algorithms to decompose the problem into *subgraphs* (see Definition 4), assign agents to subgraphs, and determine the agent priorities to the extent that they are determined by the relation between the subgraphs. The result is an algorithm for multi-agent pathfinding that is complete for general graphs with at least two empty vertices.

3. Push and Rotate

Our Push and Rotate algorithm consists of a pre-processing phase, and a phase in which agents are moved to their destinations. In the pre-processing phase, we first divide the graph into subgraphs, and then we assign agents to the subgraphs. The definition of a subgraph (Definition 4 in Section 3.1) ensures that only agents assigned to the subgraph can swap positions[6]. The third and final step of pre-processing is to determine the order (priority) in which agents are planned for. For agents assigned to the same subgraph, any priority ordering is feasible (we test an ordering heuristic in Section 6.3), while for agents assigned to different subgraphs, it may be necessary to complete one subgraph before starting another, so a partial priority relation between the subgraphs is determined.

When it comes to the phase of moving agents to their destination locations, our algorithm works in a fashion similar to Push and Swap (Luna & Bekris, 2011b). First, we determine a shortest path for the agent to its destination, and then we attempt to move this agent forward along this path. If other agents are encountered along the way, then the action to be taken depends on whether the blocking agent has a higher priority. If it has a lower priority (meaning that we have not planned for the agent yet), then we can try to push this agent out of the way along a shortest path to an empty vertex. If this does not work, or if the agent has a higher priority (it has been planned for, and is occupying its destination location), then we attempt to exchange the position of the agents using a `swap` operation, which involves moving both agents to a vertex of degree three or higher, emptying two of its neighbouring vertices, performing the `exchange` operation (Figure 7), and reversing the appropriate moves to ensure that only the swapping agents are in a different position — namely each other's. After the `swap` operation, we must return the higher-priority agent to its destination. In Push and Swap, this results in recursive calls to `swap`, and ultimately undefined behavior (see De Wilde, 2012, for the analysis); in our Push and Rotate algorithm, we solve this problem by detecting whether there is a cycle of agents that want to move forward. If so, all these agents are advanced one step with the `rotate` operation.

If the `swap` operation fails, we (like Luna & Bekris, 2011b) conclude that the instance has no solution. It should be noted, however, that this conclusion can only be validly drawn for particular priority orderings. In Figure 8, for example, the instance is only solvable if agent a_4 (or a_5, or both) have a higher priority than agent a_3. Otherwise, if a_3 is moved to its destination vertex v first, a `push` operation for a_4 or a_5 will fail because a_3 has higher priority; the `swap` operation will fail, because a swap is impossible between a_3 and a_4. To see why a swap is impossible, note that it requires both

---

[6] We normally use the word *swap* to indicate the exchange of position of two agents occupying adjacent nodes. A broader meaning of the word swap is that of *any* two agents changing position in an assignment. For the latter meaning, Kornhauser interchangeably uses the terms 2-cycle, transposition, or swap (Kornhauser, 1984, p. 8). Both meanings of the word swap hold with regard to agents being able to swap iff they are assigned to the same subgraph. Note that when we write `swap`, we refer to the operation defined in Algorithm 5.

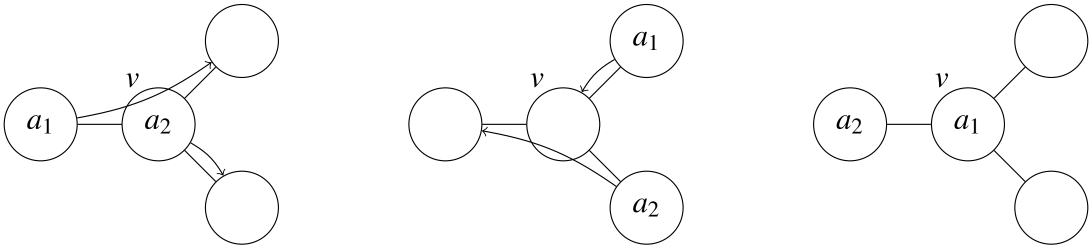


Figure 7: Sequence of states of the `exchange` operation.

agents to be at a vertex with degree three or more, with two neighbors unoccupied. The only two vertices of degree three or more that can be reached by both a_1 and a_4 are v and v'. With a_3 at v, there is only one empty neighbor to the left of v; if a_3 moves back to its start location and a_4 moves to v', then there is only one empty neighbor to the right of v'.

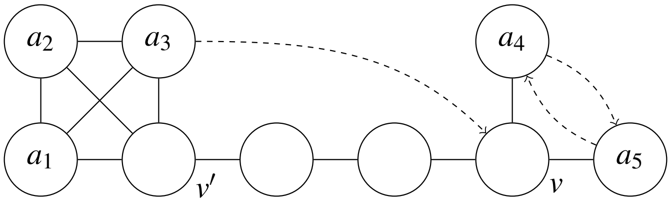


Figure 8: This instance is solvable iff the priority of a_4, a_5, or both is higher than the priority of a_3.

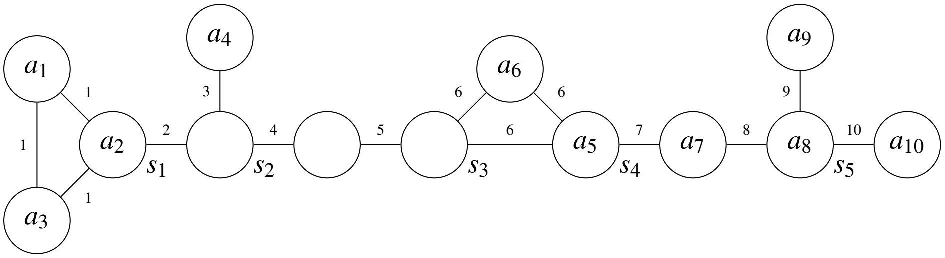
3.1 Problem Decomposition

Kornhauser (1984) proposed a decomposition of the graph into subgraphs, which we will reconstruct in Section 3.1.1, along with an algorithm to obtain the decomposition. The subgraphs are defined in such a manner that agents are either confined to one subgraph, or they are confined to an isthmus. In Section 3.1.2, we will pose the proposition — with the proof to come later — that the `swap` operation (Algorithm 5, Section 3.2) will succeed on two agents if and only if they are assigned to the same subgraph. The results of Section 3.1.2 are a refinement of the results by Kornhauser, who did not explicitly state all the details of the agent assignment problem. Finally, in Section 3.1.3, we consider the interactions between agents assigned to different subgraphs, in the sense that some subgraphs must be solved before others, in order to avoid the problems explained in Figure 8. Kornhauser does not specify any priority for the agents, and therefore there are no restrictions regarding moving agents with a higher priority, so the prioritization of subgraphs is unique to our approach.

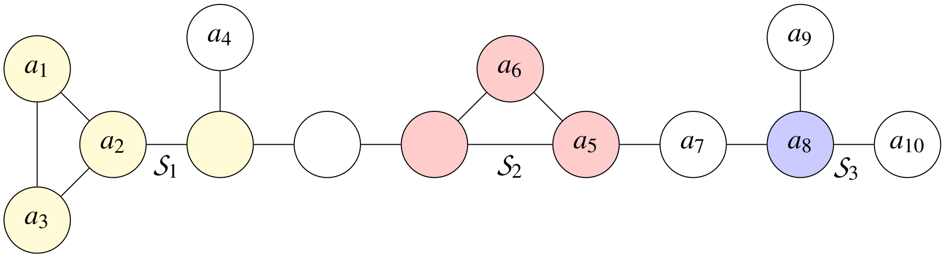
3.1.1 CONSTRUCTING SUBGRAPHS

Kornhauser (1984) remarked that any connected graph can be viewed as a tree of biconnected components. Within biconnected components, agents can exchange position, so the biconnected components play a central role in the construction of subgraphs. In Kornhauser's thesis, the first step to creating subgraphs is to divide the graph into biconnected components, defined by Kornhauser as follows (Kornhauser, 1984, p. 30):

Definition 1 (Biconnected Components)**.** *Let $\mathcal{G} = (V, E)$ be a simple connected graph. Two edges $e_1, e_2 \in E$ are* equivalent *if and only if $e_1 = e_2$ or there is a cycle in $\mathcal{G}$ containing e_1 and e_2; this is an equivalence relation. The equivalence classes, together with incident vertices, are the biconnected components of $\mathcal{G}$.*



(a) Ten biconnected components, and vertices $s_1, \ldots, s_5$ with degree 3 that join biconnected components.



(b) With $m = 3$, this graph has three subgraphs $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$.

Figure 9: Identifying biconnected components (9(a)) and subgraphs (9(b)).

Biconnected components consisting of one edge are called *trivial*. The graph in Figure 9(a) contains ten biconnected components, two of which are nontrivial (components numbered 1 and 6). Note that vertices $s_1, \ldots, s_5$ in Figure 9 are vertices[7] with degree 3 that join biconnected components, both trivial and nontrivial. On the basis of these vertices Kornhauser defines an equivalence relation that we can use to construct subgraphs.

Definition 2 (Join vertices)**.** *Let $\mathcal{G} = (V, E)$ be a simple connected graph. The set of* join vertices *$S \subset V$ consists of vertices of degree ≥ 3 which are common to at least two biconnected components.*

Note that the set of join vertices S cannot comprise the set of all vertices V: first suppose the graph consists exclusively of trivial biconnected components. This implies that the graph is acyclic, and there exist at least two vertices of degree 1, which are therefore not join vertices. In case the graph also contains a nontrivial biconnected component C_1, then if $S = V$, all vertices in C_1 must be connected to other biconnected components $C_2, \ldots, C_j$, the vertices of which would also all have to

---

[7] We follow the notation from Kornhauser here, and denote join vertices by s_i rather than denoting vertices by v_i as we do elsewhere.

be connected to *different* biconnected components. Of course, it is not possible to keep creating new biconnected components, and reusing an earlier one would create a cycle, meaning that all vertices involved are in a single biconnected component, not connecting different ones.

In the following, let m be the number of empty vertices.

Definition 3 (Reachability Equivalence)**.** *Given a simple connected graph $\mathcal{G} = (V, E)$ and its set of join vertices S, $s_1, s_2 \in S$ are equivalent if and only if*

1. *$s_1 = s_2$, or*
2. *s_1 and s_2 are in the same nontrivial biconnected component, or*
3. *there is a unique path between s_1 and s_2, and its length is $\leq m - 2$.*

The transitive closure of this relation is an equivalence relation.

In Figure 9(a), $m = 3$ and we have the equivalence classes $S_1 = \{s_1, s_2\}$, $S_2 = \{s_3, s_4\}$, and $S_3 = \{s_5\}$. Note that if we remove one agent from Figure 9, then m is 4, and condition 3 of Definition 3 ensures that s_2 and s_3 are in the same equivalence class, as well as s_4 and s_5. Hence, with 4 empty vertices, the graph in Figure 9 contains a single equivalence class $S_1 = \{s_1, \ldots, s_5\}$.

Definition 4 (Subgraph)**.** *Given an equivalence class S_i, a* subgraph *$\mathcal{S}_i = (V_i, E_i)$ is a subgraph of $\mathcal{G}$, induced by the set of vertices V_i consisting of:*

1. *the equivalence class S_i from Definition 3,*
2. *any vertices $v \in V$ such that v is in the same nontrivial biconnected component as some $s \in S_i$,*
3. *any vertices on a unique path between two join vertices $s_1, s_2 \in S_i$.*

Figure 9(b) shows that there are three subgraphs in our example, corresponding to the equivalence classes S_1, S_2, and S_3. An algorithm to obtain the division into subgraphs is given below in Algorithm 1.

Algorithm 1 `find_subgraphs`$(\mathcal{G}, m)$

```
1: $\mathcal{S} \leftarrow$ all nontrivial biconnected components in $\mathcal{G}$
2: $\mathcal{S} \leftarrow \mathcal{S} \cup \{\{v\} | v \in V \wedge \texttt{degree}(v) \geq 3 \wedge \nexists i : v \in V_i\}$
3: while $\exists \mathcal{S}_i, \mathcal{S}_j \in \mathcal{S} \mid \exists u, v(\min_{v \in \mathcal{S}_i, u \in \mathcal{S}_j} \texttt{distance}(v, u)) \leq m - 2$ do
4:     $\mathcal{S}_k = \mathcal{S}_i \cup \mathcal{S}_j \cup \{v' | v' \in \texttt{shortest\_path}(u, v)\}$
5:     $\mathcal{S} \leftarrow \{\mathcal{S} \setminus \{\mathcal{S}_i, \mathcal{S}_j\}\} \cup \{\mathcal{S}_k\}$
6: return $\mathcal{S}$
```

In line 1 of Algorithm 1, we first find all nontrivial biconnected components, which can be done in $O(|V| + |E|)$ time (Hopcroft & Tarjan, 1973). Next, we add all vertices of degree three or higher that are not part of a nontrivial biconnected component to the set $\mathcal{S}$. In the while loop starting on line 3, all elements of $\mathcal{S}$ that have vertices with distance $\leq m - 2$ will be joined into one subgraph.

Note that the shortest path between S_i and S_j in line 3 is always between two join vertices: two nontrivial biconnected components are connected by an isthmus, and so the vertices connecting the nontrivial component to the isthmus must be common to more than one biconnected component, namely the nontrivial biconnected component, and the trivial biconnected component which is the isthmus edge. Hence, line 3 corresponds to point 3 from Definition 3 (either u and v are join vertices, or two join vertices are encountered on the path from u to v), and line 4 corresponds to point 3 from Definition 4.

We note here that Kornhauser employs a slightly different definition of subgraph. He includes in his definition also vertices on a *plank* of a subgraph.

Definition 5 (Plank)**.** *Given an equivalence class* S_i *and the corresponding subgraph* $\mathcal{S}_i = (V_i, E_i)$*, a* plank *is a unique and maximal path in* $\mathcal{G}$ *from some vertex* $v \in (V \setminus V_i)$ *to a vertex* $s_j \in S_i$ *of length* $\leq m - 1$.

In Figure 9, the subgraph $\mathcal{S}_2$ has two planks: the plank on the left between s_2 and s_3, and the right plank between s_4 and s_5. Subgraph $\mathcal{S}_3$ even has three planks: it shares the plank between s_5 and s_4 with $\mathcal{S}_2$, and also the two outgoing edges from s_5 are planks.

As a result of Kornhauser's definition, subgraphs can overlap. We opted not to include planks in Definition 4, as we prefer to keep them separate, which is useful in illustrations as in Figure 9(b).

3.1.2 ASSIGNING AGENTS TO SUBGRAPHS

In this section, we will state two important results on the relation between agents and subgraphs:

1. Algorithm 2 below assigns to a subgraph those agents that are *confined* to the subgraph and its planks, i.e., agents that can *only* reach vertices of the subgraph and its planks.
2. In Proposition 3 we state that for any two adjacent agents assigned to the same subgraph, we can exchange their positions using our `swap` operation. This implies that an agent assigned to a subgraph can reach *all* of the vertices of the subgraph and its planks.

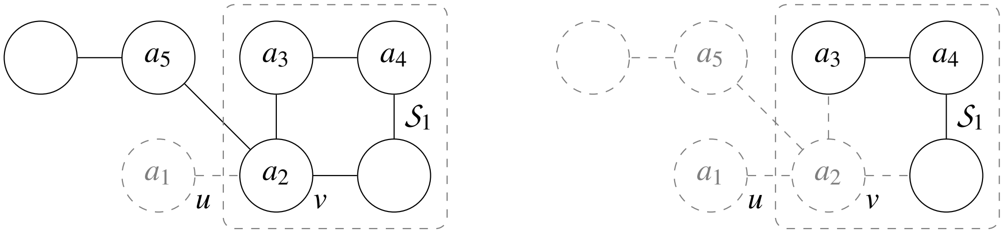
The implication of the second result is that individual subgraphs are *solvable* in case the goal positions of the agents assigned to a subgraph are inside the subgraph or its planks. The importance of the first result is that we can determine exactly which agents belong to a subgraph. Note that not all agents are assigned to a subgraph, as some agents are confined to an isthmus.

Algorithm 2 `assign_agents_to_subgraphs`($\mathcal{G}, A, \mathcal{A}, \mathcal{S}$)

```
for all $a_i \in A$ do
    $f(a_i) \leftarrow \perp$
for all $\mathcal{S}_i = (V_i, E_i) \in \mathcal{S}$ do
    for all $v \in V_i$ do
        $m'' \leftarrow$ number of unoccupied vertices reachable from $\mathcal{S}_i$ in graph induced from $V \setminus \{v\}$
        for all $u \notin V_i$ for which $\{u, v\} \in E$ do
            $m' \leftarrow$ number of unoccupied vertices reachable from $v$ in $(V, E \setminus \{u, v\})$
            if $((m' \geq 1 \wedge m' < m) \vee m'' \geq 1) \wedge a_i \leftarrow \mathcal{A}^{-1}(v) \neq \perp$ then
                $f(a_i) \leftarrow \mathcal{S}_i$
                Follow path from $u$ away from $v$ and assign the first $m' - 1$ agents on this path to $\mathcal{S}_i$
        if $\{u \notin V_i$ for which $\{u, v\} \in E\} = \emptyset \wedge (a_i \leftarrow \mathcal{A}^{-1}(v) \neq \perp)$ then
            $f(a_i) \leftarrow \mathcal{S}_i$
return $f$, the assignment of agents to $\mathcal{S}$
```

Algorithm 2 iterates over all vertices in all subgraphs, and decides whether an agent occupying a vertex should be assigned to the subgraph. We can distinguish two types of vertices in the subgraph: there are vertices that only have connections to vertices inside the subgraph, and vertices that are connected to vertices not in the subgraph — these vertices (which are join vertices, Definition 2) form the start of a plank. Agents on the former type of vertex are immediately assigned to the subgraph, whereas for plank vertices it is checked whether there are sufficient empty vertices to move any agents on the plank into the subgraph.

For the treatment of plank vertices, our algorithm can be viewed as a refinement from Kornhauser (1984), who only remarks that an agent is assigned to the subgraph "*if enough blanks are on the subgraph to one side of P [the pebble] to take P off the plank...*". We specify the values m' and m'' (Figure 10) that encode exactly when sufficient free space is available for an agent to be moved into the subgraph. The meaning of m' and m'' and the condition on line 8 of Algorithm 2 are explained further by the different cases in the proof of Proposition 1.



(a) $m' = m = 2$, meaning that a_2 will not be assigned (yet) to $\mathcal{S}_1$, and that a_1 will be assigned to $\mathcal{S}_1$.

(b) $m'' = 1$, meaning that a_2 will be assigned to $\mathcal{S}_1$.

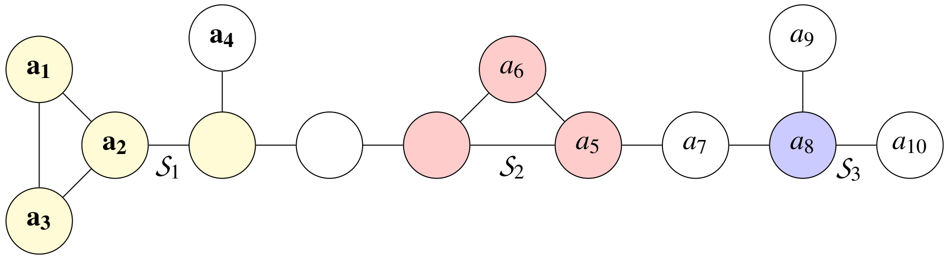
Figure 10: Illustration of m' and m''. Unreachable and removed parts of the graph have been grayed out.

Before we prove the correctness of Algorithm 2, let us first see how agents are assigned to subgraphs in our example. Figure 11(a) shows that agents a_1, a_2, a_3, and a_4 are assigned to $\mathcal{S}_1$. Agents $a_1, \ldots, a_3$ are clearly inside the subgraph, and will be assigned on the basis of line 12. Agent a_4 is on a plank, but not on a join vertex, so it will be assigned to $\mathcal{S}_1$ in line 10, since $m' = 3$.

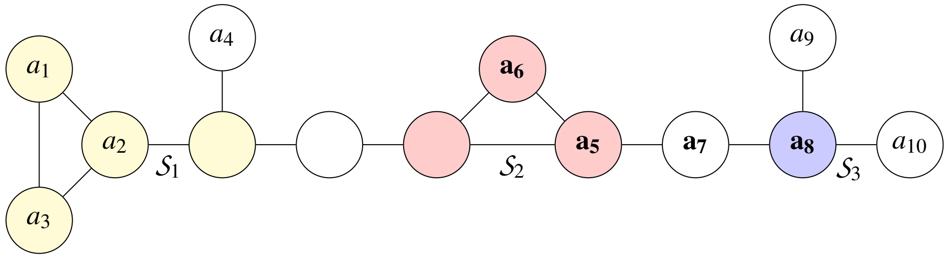
For subgraph $\mathcal{S}_2$ (Figure 11(b)), a_6 is on an inner vertex, so it will be assigned to $\mathcal{S}_2$ in line 12. Agent a_5 occupies a join vertex, and three empty vertices can be reached without making use of vertex $\mathcal{A}(a_5)$, so $m'' = 3$, and a_5 is assigned to $\mathcal{S}_2$ in line 9. Agents a_7 and a_8 are on a plank that starts at join vertex $\mathcal{A}(a_5)$; $m' = 3$, so the first two agents on the plank, starting from vertex $\mathcal{A}(a_7)$, are assigned to $\mathcal{S}_2$ in line 10.

For subgraph $\mathcal{S}_3$ (Figure 11(c)), both a_9 and a_{10} are on a (non-join) plank vertex, and with $m'' = 3$, both are assigned to $\mathcal{S}_3$ in line 10. Note that when $v = \mathcal{A}(a_8)$ (line 4) and $u = \mathcal{A}(a_7)$ (line 6), we have $m' = m'' = 0$, so a_8 is not assigned to $\mathcal{S}_3$.

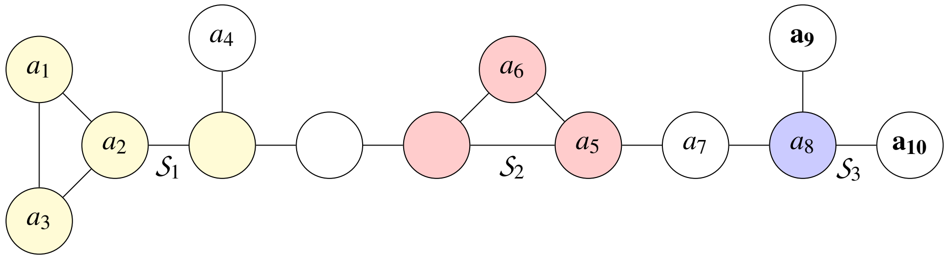
Proposition 1. *Algorithm 2 assigns agent a_i to subgraph $\mathcal{S}_j$ if and only if a_i is confined to $\mathcal{S}_j$ and its planks.*



(a) Agents a_1, a_2, a_3, and a_4 are assigned to $\mathcal{S}_1$ (the leftmost subgraph).

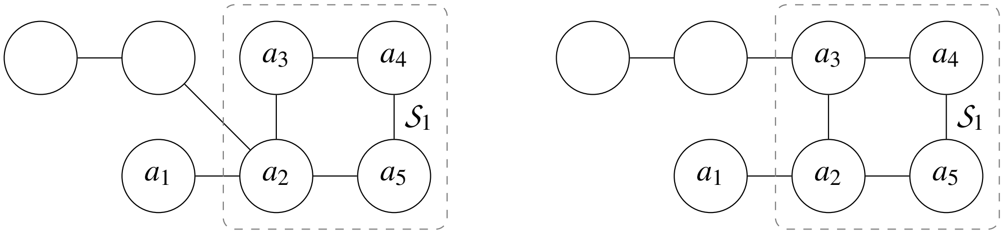


(b) Agents a_5, a_6, a_7, and a_8 are assigned to $\mathcal{S}_2$ (the middle subgraph).



(c) Agents a_9 and a_{10} are assigned to $\mathcal{S}_3$ (the rightmost subgraph).

Figure 11: Assignment of agents to subgraphs.



(a) For agent a_2, $m'' = 0$.

(b) For agent a_2, $m'' = 2$.

Figure 12: In Figure 12(a), a_2 is not assigned to $\mathcal{S}_1$; in Figure 12(b), a_2 is in $\mathcal{S}_1$, but a_3 is not.

Proof. The central idea of the proof is that if an agent can reach the 'inside' of a subgraph, then there are not sufficient empty vertices in the graph to move beyond one of its planks, as shown by Kornhauser (1984). We will consider the following four cases:

1. An agent a_i occupies a vertex v inside some subgraph $\mathcal{S}_j$, and v is not a plank vertex,
2. An agent a_i occupies a plank vertex $w \notin \mathcal{S}_j$ (i.e., not the join vertex),
3. An agent a_i occupies a join vertex v that is the start of a plank (i.e., $v \in \mathcal{S}_j$).
4. An agent a_i occupies a vertex v that is not in $\mathcal{S}_j$ nor on any of its planks.

Case 1: inner vertex: first note that a_i is assigned to $\mathcal{S}_j$ (line 12 in Algorithm 2). To reach any vertex not assigned to $\mathcal{S}_j$ or one of its planks, agent a_i must walk off one of the subgraph's planks. First, the agent must take at least one step to a join vertex that is the start of a plank, leaving behind one empty vertex. Then, walking to the end of the plank, which is $m-1$ long, requires $m-1$ empty vertices. To step off the plank, another empty vertex is required, but there are only m empty vertices in total, so a_i can reach at most the end of a plank.

Case 2: plank vertex w**:** recall that m' is defined as: the number of unoccupied vertices reachable from v in $(V, E\backslash\{u,v\})$, and here v is the start of the plank. $m'-1$ of these empty vertices can be used to move $m'-1$ plank agents into the subgraph. Consider agent a_i that is number $m'-1$ on the plank: it takes $m'-1$ empty vertices to move to the start vertex of the plank, and since m' empty vertices are available into which other agents can be moved, one more empty vertex is available for a_i to step off the plank and into the subgraph. Then, we can apply the first case to show that a_i cannot leave the subgraph or any of its planks.

Case 3: join vertex: agent a_i cannot always enter the subgraph, even when agents behind it on the plank can. Consider Figure 12(a), in which agent a_2 is on the start of two planks, and let $u = \mathcal{A}(a_1)$. Then $m' = 2$, but a_2 cannot enter the single subgraph $\mathcal{S}_1$, because it has to move out of the way to give the agents inside the subgraph space to move. In this case, $m' = m$, and all the empty vertices are behind a_2. Figure 12(b) shows that in case $m' = m$, the agent at the join vertex is only assigned to the subgraph in case empty vertices can be reached from $\mathcal{S}_1$ without using the join vertex. Hence, in Figure 12(b) agent a_2 is assigned to $\mathcal{S}_1$ since $m'' \geq 1$.

Note that if a_i is not assigned to $\mathcal{S}_j$, then it is also not confined to $\mathcal{S}_j$ or its planks: all empty vertices are directly behind a_i ($m' = m$), so it can move m steps away from $\mathcal{S}_j$, while the plank is only $m-1$ edges long.

In case the agent a_i at the join vertex is assigned to the subgraph, then there is at least one empty vertex that can be used to make room for a_i to step into the subgraph or onto another plank, and we can apply the reasoning of the first case to conclude that a_i is confined to the subgraph and its planks.

Case 4: vertex v outside $\mathcal{S}_j$ and its planks: clearly, a_i is not confined to $\mathcal{S}_j$ and its planks. To see that a_i won't be assigned to $\mathcal{S}_j$, note that it can only enter $\mathcal{S}_j$ via its planks. As the planks are $m-1$ edges long (all shorter planks are not connected to another part of the graph), a_i would need at least m steps to reach the start of a plank of $\mathcal{S}_j$. Since there are only m empty vertices in the graph, there are no empty vertices left to enter $\mathcal{S}_j$, and a_i is therefore not assigned to the subgraph.

□

An important property of subgraphs is that an agent can reach any vertex of the subgraph it is assigned to. Kornhauser (1984) proved *2-transitivity*: for any two pairs of agents a_1, a_2, and b_1, b_2 assigned to the same subgraph, it is possible to send a_i to the location of b_i (possibly moving other agents). In Section 3.2, we will prove that our `swap` operation will always succeed on two agents assigned to the same subgraph, which achieves 2-transitivity.

3.1.3 PRIORITIES OF SUBGRAPHS

The third stage of the decomposition process is to assign priorities to agents based on their membership to subgraphs. Algorithm 3 generates a partial order of subgraphs, and agents inherit the priority of the subgraph they are assigned to. For the completeness of our Push and Rotate approach, it is not necessary to differentiate between priorities of agents assigned to the same subgraph, although it is possible to do so in order to pursue better solution quality. Finally, agents that are not assigned to any subgraph receive the lowest priority, and are therefore planned last.

Algorithm 3 generates precedence constraints between two subgraphs in case one of the subgraphs contains an agent that will restrict the movements of agents in the other subgraph. Specifically, given two subgraphs $\mathcal{S}_i$ and $\mathcal{S}_j$, and an agent r assigned to $\mathcal{S}_j$, the priority relation $\mathcal{S}_i \prec \mathcal{S}_j$ is added in the following two cases[8]:

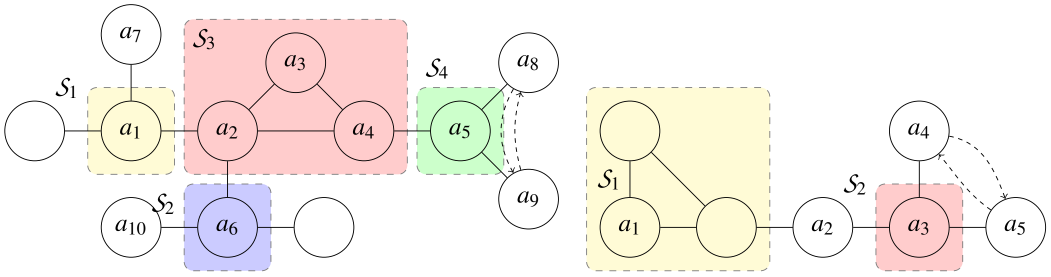
1. The goal position of agent r is the start vertex of a plank[9] of $\mathcal{S}_i$, or
2. The goal position of agent r is a vertex v' on a plank of $\mathcal{S}_i$, and all vertices of the plank between v' and the start of the plank are goal positions of agents not assigned to any subgraph.

The intuition behind both cases is that once agent r has been moved into its goal position, there will be no more empty vertices in the subgraph $\mathcal{S}_i$; otherwise, in the first case, agent r would be able

---

[8] A note on notation: in the coming algorithms, we use r and s to denote agents (as do Luna & Bekris, 2011b). In the examples so far we have used $a_1, \dots, a_k$, but having agents a_i and a_j can be confusing in combination with subgraphs $\mathcal{S}_i$ and $\mathcal{S}_j$.

[9] Recall that the start vertex of a plank is the vertex of the plank that belongs to the subgraph.



(a) $\mathcal{S}_4 \prec \mathcal{S}_3$ due to agent a_5.

(b) $\mathcal{S}_2 \prec \mathcal{S}_1$ due to agent a_2.

Figure 13: If agents from one subgraph have a goal location on the plank of another, then there might be a precedence relation between the subgraphs.

to move into $\mathcal{S}_i$ and would be part of S_i, by Proposition 1, which is a contradiction (in the second case: the agents not assigned to any subgraph could move into $\mathcal{S}_i$, so they would be assigned to $\mathcal{S}_i$). For the agents of subgraph $\mathcal{S}_i$ to move into their goal locations, empty vertices must be brought into $\mathcal{S}_i$. In case r has been moved into its goal, this is not possible: a `push` operation is not allowed, and a `swap` operation is not possible, by Proposition 3. Hence, subgraph $\mathcal{S}_i$ must have a higher priority than agent r, and therefore subgraph $\mathcal{S}_i \prec \mathcal{S}_j$.

An example of the first case is shown in Figure 13(a), in which for all agents their start location equals their destination location, except for agents a_8 and a_9 that want to exchange position. If a_5 has a higher priority than a_8 and a_9, then an algorithm based on the operations `push` and `swap` will not succeed: to plan a_8, a `push` may not move a_5 aside since a_5 has higher priority; a `swap` will fail because a_5 and a_8 are in different subgraphs, and therefore cannot swap. Hence, we must add the priority relation $\mathcal{S}_4 \prec \mathcal{S}_3$. An example of the second case is shown in Figure 13(b), in which the plank vertex of $\mathcal{S}_2$ is occupied by a_3. In this example, no solution using `push` and `swap` can be found in case a_2 has a higher priority than a_4 and a_5. When agents a_4 and a_5 are planned, it is not allowed to move agent a_3 — which is not assigned to any subgraph — away with a `push`, because agent a_2 behind it may not be moved by the `push` operation. An attempted swap between a_3 and one of the agents of $\mathcal{S}_2$ will fail because a_3 is not in $\mathcal{S}_2$. Hence, the priority relation $\mathcal{S}_2 \prec \mathcal{S}_1$ must be added.

Proposition 2. *If the priority relation between subgraphs is cyclic, then the instance is not solvable.*

Proof. Suppose by way of contradiction that we have two subgraphs $\mathcal{S}_i$ and $\mathcal{S}_j$ such that $\mathcal{S}_j \prec \mathcal{S}_i$, due to agent r from $\mathcal{S}_i$, and $\mathcal{S}_i \prec \mathcal{S}_j$, due to agent s from $\mathcal{S}_j$.

By definition of a subgraph, there is at most one connection between two separate subgraphs, so the goal locations of both r and s are on the same isthmus. For agent r to induce $\mathcal{S}_j \prec \mathcal{S}_i$, it either has to reach the start vertex v (from the perspective of $\mathcal{S}_j$) of the isthmus, or there may only be agents not assigned to $\mathcal{S}_j$ (so not s) between itself and v. Hence, agent s may not occupy any vertex on the isthmus between r and vertex v. The only way for s to induce $\mathcal{S}_i \prec \mathcal{S}_j$ would be if the goal location of s is behind (i.e., on the side of $\mathcal{S}_i$) the goal location of r.

Hence, the assumption that the priority relation is cyclic implies that in their goal locations r and s have swapped, however Kornhauser proved that only agents assigned to the same subgraph can swap positions (Kornhauser, 1984, p. 11). $\square$

As an example of Proposition 2, consider agents a_1 and a_2 in Figure 13(a). If the destination location of agent a_1 is the current location of a_2, then the precedence constraint $\mathcal{S}_3 \prec \mathcal{S}_1$ would have to be added. Similarly, if the destination of a_2 is the current location of a_1, then we have $\mathcal{S}_1 \prec \mathcal{S}_3$. In this configuration, agents a_1 and a_2 have swapped location, but since they are assigned to different subgraphs, this is not possible, by Proposition 3.

Algorithm 3 `subgraph_priority`$(\mathcal{G},\mathcal{S},\mathcal{T},f)$

```
 1: for all $\mathcal{S}_i = (V_i, E_i) \in \mathcal{S}$ do
 2:   for all $v \in S_i$ do
 3:     for all $u \notin V_i$ for which $\{u,v\} \in E$ do
 4:       Vertex $u$ should be the first vertex on the path from $\mathcal{S}_i$ to another subgraph $\mathcal{S}_j$, otherwise
          continue with the next $u$
 5:       $v' \leftarrow v$
 6:       while $\exists r : (\mathcal{T}(r) = v') \wedge (f(r) \neq \mathcal{S}_i)$ do
 7:         if $f(r) = \mathcal{S}_j$ then
 8:           $\mathcal{S}_i \prec \mathcal{S}_j$
 9:           Continue with next $v$ (line 2)
10:         $v' \leftarrow$ next vertex on path from $\mathcal{S}_i$ to $\mathcal{S}_j$
11: return The priority relation "$\prec$"
```

Algorithm 3 checks for every subgraph $\mathcal{S}_i$, and for every join vertex $v \in S_i$ connected to a node $u \notin V_i$ that is on a path (i.e., an isthmus) to a component $\mathcal{S}_j$, whether there exists an agent r assigned to $\mathcal{S}_j$ that restricts the movements of the agents assigned to $\mathcal{S}_i$. In order for the precedence constraint $\mathcal{S}_i \prec \mathcal{S}_j$ to be added, agent r should either occupy v, or it should occupy a node v' such that all nodes[10] from (not including) v' to v (including) are goal locations of agents not assigned to any subgraph. In either case, agent r, if planned first, would bottle up the agents assigned to $\mathcal{S}_i$, hence the latter agents should receive higher priority: $\mathcal{S}_i \prec \mathcal{S}_j$.

A final note on agents not assigned to any subgraph. Although Algorithm 3 does not assign any priority to these agents, these agents will be planned *after* all agents assigned to a subgraph.

3.2 The Operations `push`, `swap`, and `rotate`

In Section 3.3 we will describe the operation `solve` that moves all agents to their destinations, if possible, but in this section we will first present the main operations used in our algorithm: `push`, `swap`, and `rotate`. The `push` and `swap` operations are conceptually similar to the operations presented by Luna and Bekris (2011b), while the `rotate` operation moves agents in a cycle one step forwards. The `rotate` operation does not require that all agents move simultaneously.

A note on notation: in our algorithms we assume parameters are passed by reference, so the algorithm from which a 'subroutine' is called can see any changes made to the parameter by the subroutine. Below, Π stands for the set of generated moves, and $\mathcal{U}$ stands for the set of blocked

---

[10] If there are vertices in between v and v' that are empty in $\mathcal{T}$, then the agents in $\mathcal{S}_i$ would still have room to maneuver, even if r is at its goal location.

vertices, i.e., vertices that the algorithm may not use (any agents on these vertices will therefore remain in place).

Algorithm 4 `push`$(\Pi, \mathcal{G}, \mathcal{A}, r, v, \mathcal{U})$

```
1: if vertex $v$ is occupied then
2:     $\mathcal{U}' \leftarrow \mathcal{U} \cup \{\mathcal{A}(r)\}$
3:     if clear_vertex$(\Pi, \mathcal{G}, \mathcal{A}, v, \mathcal{U}')$ = false then
4:         return false
5: move$(\Pi, \mathcal{A}$, agent $r$ to vertex $v)$
6: return true
```

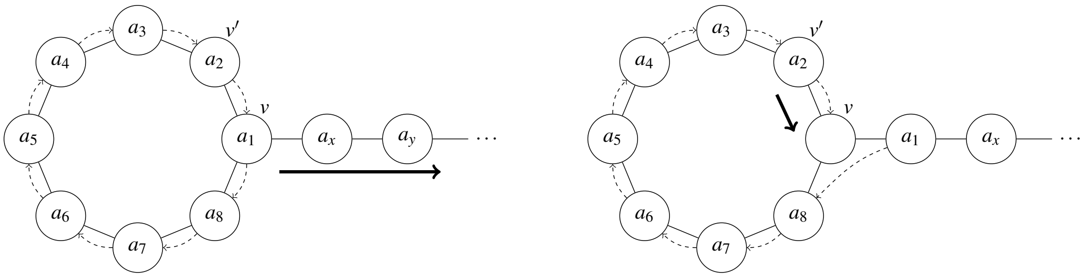
The `push` operation attempts to move an agent r, currently at location $\mathcal{A}(r)$, to location v, which is assumed to be adjacent to $\mathcal{A}(r)$. If successful, the move is recorded in the current sequence of agent moves Π (line 5). In case location v is occupied when `push` is called, the `clear_vertex` operation is used to try to clear v. In line 2, prior to calling `clear_vertex`, the current location of agent r is added to the set $\mathcal{U}$ of locations from which agents may not be moved.

The specification of the `clear_vertex` (Algorithm 10) has been moved to Appendix A (as well as some other auxiliary algorithms), so as not to disrupt the flow of text too much. What `clear_vertex` does is to find a shortest path from v to an unoccupied node u in the graph induced by $V \setminus \mathcal{U}$; if such a path exists, all agents on this path are moved one place towards u.

Algorithm 5 `swap`$(\Pi, \mathcal{G}, \mathcal{A}, r, s)$

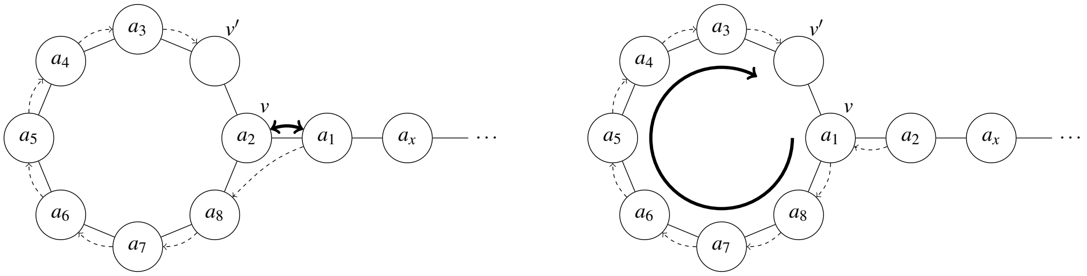
```
1: $S \leftarrow \{$vertex $x \in f_S(r) \mid$ degree$(x) \geq 3\}$
2: for all vertex $v \in S$ do
3:     $\mathcal{A}' \leftarrow \mathcal{A}$
4:     $\Pi' \leftarrow [\,]$
5:     if multipush$(\Pi', \mathcal{G}, \mathcal{A}', r, s, v)$ = true then
6:         if clear$(\Pi', \mathcal{G}, \mathcal{A}', r, s, v)$ = true then
7:             $\Pi \leftarrow \Pi + \Pi'$
8:             $\mathcal{A} \leftarrow \mathcal{A}'$
9:             exchange$(\Pi, \mathcal{G}, \mathcal{A}, r, s, v)$
10:            reverse$(\Pi, \mathcal{A}, \Pi'_{r/s})$
11:            return true
12: return false
```

The `swap` operation attempts to exchange the locations of two adjacent agents r and s, by moving them to a vertex of degree three or higher, performing the `exchange` operation there (see Figure 7 and Algorithm 13), and moving the agents back to where the `swap` was initiated (at the same time reversing the moves of all agents not involved — line 10, where $\Pi'_{r/s}$ stands for the sequence of new moves with the roles of r and s reversed). All vertices of degree three or higher that belong to the same subgraph as agent r (denoted $f_S(r)$) are eligible to perform the swap (we evaluate the vertices closest to r and s first). For a candidate swap node v, the `multipush` operation (Algorithm 11, Appendix A) attempts to bring both agents to v (line 5). Since all moves of `swap` will be reversed (with the exception of the `exchange` operation, and with the roles of r and s reversed), `multipush` does not take into account the set of blocked agents $\mathcal{U}$; these agents may be moved as well. The `multipush` operation is essentially a series of `push` operations, iteratively moving agents r and s a step closer to v, and moving other agents out of the way.



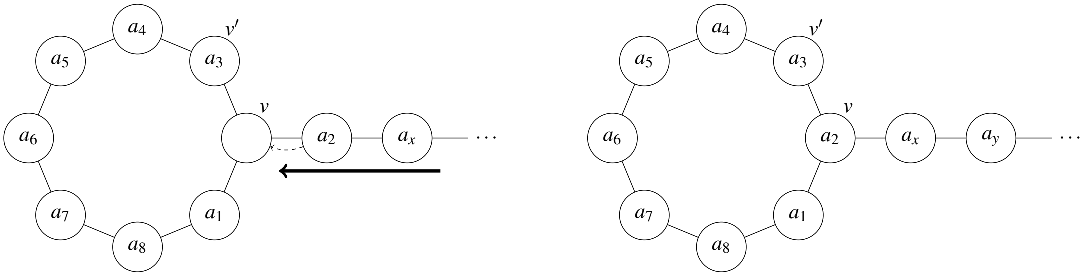
(a) Initial configuration

(b) a_1 moves out



(c) a_2 moves forward, swaps with a_1

(d) agents rotate, starting with a_3



(e) a_2 moves back into cycle

(f) Goal configuration

Figure 14: The steps of the `rotate` operation.

If `multipush` succeeds, the next step in Algorithm 5 is to try to ensure that v has two empty neighbors, by calling the `clear` operation. The idea behind `clear` is to push agents on vertices adjacent to v out of the way towards empty vertices, but the exact specification is quite intricate, due to the possible movements of r and s that have to be considered in order to allow the other agents to reach the unoccupied vertices. The specification and explanation of the `clear` operation is given in Appendix A. We now state the result that the `swap` operation will succeed for two agents assigned to the same subgraph (the proof is in Appendix B).

Proposition 3. *For two agents r and s on adjacent vertices in $\mathcal{G}$, the operation* `swap`$(\Pi, \mathcal{G}, \mathcal{A}, r, s)$ *will succeed if and only if r and s are assigned to the same subgraph $\mathcal{S}_i$.*

The `rotate` operation assumes a cycle c of locations, and moves all agents within that cycle forward one step. In case c is not fully occupied, performing the rotate is a trivial operation; otherwise, one agent must first be moved out of the cycle to provide room for the other agents to move. The steps of the `rotate` operation are illustrated in Figure 14. In Figure 14(a), we see an agent a_1 that will be temporarily pushed out of the cycle to make room for the others. It does not matter whether or not agents a_x and a_y, and others behind them, have a higher priority, since the moves performed to move a_1 out of the cycle will be reversed.

In Figure 14(c), agent a_2 moves forward into v', and a `swap` operation for agents a_1 and a_2 ensures that a_1 and a_2 will be in the right position after the rotate. Figure 14(d) depicts the actual rotation of the agents, with a_3 moving forward first, followed by a_4, etc. In Figure 14(e) agent a_2 moves back into the cycle to complete the rotate, and in Figure 14(f) the goal configuration is reached by returning the other agents (a_x and a_y in the figure) to their previous locations.

Algorithm 6 `rotate`$(\Pi, \mathcal{G}, \mathcal{A}, c)$

```
 1: for all vertices $v \in c$ do
 2:     if $v$ is unoccupied then
 3:         Move all agents in $c$ forward, starting with the agent moving to $v$
 4:         return true
 5: for all vertices $v \in c$ do
 6:     $r \leftarrow \mathcal{A}(v)$
 7:     $\Pi' \leftarrow [\,]$
 8:     if clear_vertex$(\Pi', \mathcal{G}, \mathcal{A}, v, c \setminus \{v\})$ = true then
 9:         $\Pi \leftarrow \Pi + \Pi'$
10:         $v' \leftarrow$ vertex in $c$ before $v$
11:         $r' \leftarrow \mathcal{A}(v')$
12:         move$(\Pi, \mathcal{A}$, agent $r'$ to vertex $v)$
13:         swap$(\Pi, \mathcal{G}, \mathcal{A}, r, r')$
14:         Move all agents in $c$ forward, starting with the agent moving to $v'$
15:         reverse$(\Pi, \mathcal{A}, \Pi'_{r/r'})$
16:         return true
17: return false
```

The specification of the `rotate` operation is given in Algorithm 6. Lines 1 to 4 deal with the trivial case of a cycle in which at least one location v is unoccupied: the agent going to v is moved first, after which there is room for the agents behind it to move in turn. For the main body of the algorithm, line 5 iterates over all vertices in the cycle, looking for a vertex v that can be cleared (Lemma 2 proves that, for a solvable instance, such a vertex can always be found), in line 8.

Once `clear_vertex` has succeeded, the algorithm then proceeds with the steps as illustrated in Figure 14; in lines 10 to 12, the next agent is moved to the vacated vertex, and in line 13 this agent swaps with the agent that has moved out of the cycle (Lemma 2 proves that this `swap` is possible for a solvable instance). Line 14 rotates the agents, and in line 15 the necessary moves are reversed.

3.3 The Push and Rotate Algorithm

In Algorithm 7 we present our Push and Rotate algorithm. It first determines the division into subgraphs, and it determines the assignment of agents to subgraphs for both the initial and the goal

Algorithm 7 Push and Rotate($\mathcal{G}, A, \mathcal{A}, \mathcal{T}$)

```
$m \leftarrow |V| - |A|$
$\mathcal{S} \leftarrow$ find_subgraphs($\mathcal{G}, m$)
$f \leftarrow$ assign_agents_to_subgraphs($\mathcal{G}, A, \mathcal{A}, \mathcal{S}$)
$f' \leftarrow$ assign_agents_to_subgraphs($\mathcal{G}, A, \mathcal{T}, \mathcal{S}$)
if $f = f'$ then
    $\prec \leftarrow$ subgraph_priority($\mathcal{G}, \mathcal{A}, \mathcal{S}, f$)
    return solve($\mathcal{G}, A, \mathcal{A}, \mathcal{T}, \mathcal{S}, f, \prec$)
return false
```

assignment of agents. If these assignment functions f and f' are not the same, then the instance is not feasible and the algorithm returns false. Otherwise, the main algorithm `solve` is called which returns a sequence of moves transforming the initial agent assignment into the goal assignment.

The idea behind the `solve` operation is to move agents to their destinations one by one, and a single agent is moved to its destination one step at a time, along a shortest path to its destination. In every iteration, first a `push` is tried to move the agent to the next location in its path, and if the `push` fails, then a `swap` is performed, which will succeed (as proved in Theorem 1, provided that there are at least two unoccupied vertices) as `solve` is only called on feasible instances.

When an agent has been moved to its destination, it is added to $\mathcal{F}$, the set of finished agents. The path it has traversed has been encoded in q, as shown in Figure 15(b). Since the `push` operation is not allowed to move agents in $\mathcal{F}$, these agents can only be moved by the `swap` operation (line 22) or the `rotate` operation (line 19). An agent in $\mathcal{F}$ can only be moved off its destination by the former operation. We refer to such an agent as a resolving agent, until it has returned to its destination. All resolving agents are contained in q, because the `swap` occurs along the path of an agent to its destination, which is then added to q. Note that when an agent in $\mathcal{F}$ is moved by the `rotate` operation, it is actually returned to its destination. The second stage (lines 26 to 35) of the `solve` operation aims to return resolving agents to their destinations, while shrinking q.

There now follows a more detailed description of Algorithm 8. First, some initialization is done: the sequence of moves Π is initialized to the empty list $[\,]$ (line 1), as is the path q of resolving agents (line 2). The set of finished agents $\mathcal{F}$ is initially empty (line 3), and r, the pointer to the agent that is selected in each iteration, is initially undefined, denoted $\perp$ (line 4). In line 5, we check whether the input graph $\mathcal{G}$ is a polygon. If so, then `is_polygon` gets the value true, otherwise false.

The outer while loop of line 6 iterates until the set of finished agents equals the set of agents. If no next agent has been selected yet ($r = \perp$, line 7), then in line 8 we choose a next agent with joint-highest priority of all agents in $A \setminus \mathcal{F}$. Note that, if $r \neq \perp$ in line 8, this can only mean that an agent has been selected in line 30 as part of resolving agents on q.

Next, we must choose a path along which r is moved to its destination. If the graph is a polygon (`is_polygon` = true, line 9), then we must choose a path that does not encounter any finished agents (line 10) — as no `swap` is possible in a polygon instance — otherwise we simply choose a shortest path (line 12).

The inner loop of line 14 will move r closer to its destination, one step with each iteration. If the next step on the path of r is also on q, then we have detected a cycle of resolving agents, and we

Algorithm 8 `solve`($\mathcal{G}, A, \mathcal{A}, \mathcal{T}, \mathcal{S}, f, \prec$)

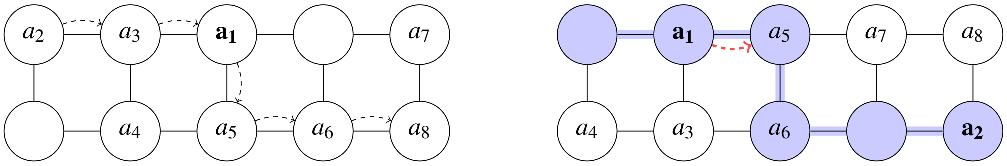
```
 1: $\Pi \leftarrow [\,]$
 2: $q \leftarrow [\,]$
 3: $\mathcal{F} \leftarrow \emptyset$
 4: $r \leftarrow \perp$
 5: $\texttt{is\_polygon} \leftarrow \forall v \in V : \texttt{degree}(v) = 2$
 6: while $\mathcal{F} \neq A$ do
 7:     if $r = \perp$ then
 8:         $r \leftarrow \texttt{next\_agent}(A \setminus \mathcal{F}, \prec)$
 9:     if $\texttt{is\_polygon}$ then
10:         $p \leftarrow \texttt{shortest\_path}(\mathcal{G}, \mathcal{A}(r), \mathcal{T}(r), \mathcal{A}(\mathcal{F}))$
11:     else
12:         $p \leftarrow \texttt{shortest\_path}(\mathcal{G}, \mathcal{A}(r), \mathcal{T}(r), \emptyset)$
13:     $q \leftarrow q + [\mathcal{A}(r)]$
14:     while $\mathcal{A}(r) \neq \mathcal{T}(r)$ do
15:         $v \leftarrow$ vertex after $\mathcal{A}(r)$ on $p$
16:         if $v \in q$ then
17:             $c \leftarrow \texttt{get\_cycle}(v, q)$
18:             $q \leftarrow q - c$
19:             $\texttt{rotate}(\Pi, \mathcal{G}, \mathcal{A}, c)$
20:         else
21:             if $\texttt{push}(\Pi, \mathcal{G}, \mathcal{A}, r, v, \mathcal{A}(\mathcal{F})) =$ false then
22:                 $\texttt{swap}(\Pi, \mathcal{G}, \mathcal{A}, r, \mathcal{A}^{-1}(v))$
23:         $q \leftarrow q + [v]$
24:     $\mathcal{F} \leftarrow \mathcal{F} \cup \{r\}$
25:     $r \leftarrow \perp$
26:     while $|q| > 0$ do
27:         $v \leftarrow$ the last vertex on $q$
28:         $s \leftarrow \mathcal{A}^{-1}(v)$
29:         if $s \in \mathcal{F} \wedge v \neq \mathcal{T}(s)$ then
30:             $r \leftarrow \mathcal{A}^{-1}(\mathcal{T}(s))$
31:             if $r = \perp$ then
32:                 $\texttt{move}(\Pi, \mathcal{A},$ agent $s$ to vertex $\mathcal{T}(s))$
33:             else
34:                 Break inner loop, continue outer loop
35:         $q \leftarrow q - [v]$
36: return $\Pi$
```

move r and all other agents involved in the cycle of q[11] forwards by performing a `rotate` operation, in line 19. Otherwise, we attempt to move agent r forwards with a `push` (line 21), and if that fails, a `swap` operation (line 22). After either the `push` or the `swap` succeeds, we append the vertex v that

---

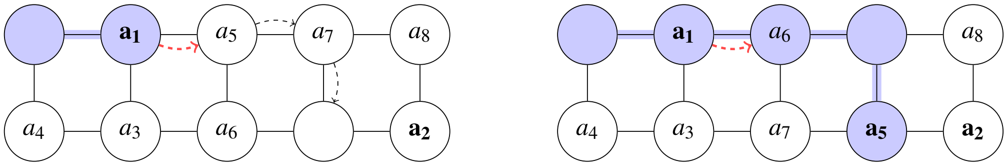
[11] The list q appended with v can consist of a cycle and a simple path emanating from the cycle, in the shape of a 'q', more or less, so the `get_cycle` method in line 17 only returns the cyclic part of q.

agent r has just moved to, to q (line 23). When the while loop of line 14 has completed, agent r is added to the set $\mathcal{F}$ of finished agents (line 24), and r is reset to $\perp$ (line 25).



(a) A shortest path of a_2 to its destination vertex through the destination vertex of a_1.

(b) The state directly after a_2 has reached its destination vertex, q is highlighted.



(c) $a_1 \in \mathcal{F}$ is encountered on q, hence $r \leftarrow a_5$ for the next iteration of the algorithm.

(d) q will continue to grow while a_5 moves to its destination vertex.

Figure 15: An example of how q changes while agents are being moved to their destination vertices.

After moving agent r to its destination, the while loop starting from line 26 iterates over the locations in q to see whether there are any agents in $\mathcal{F}$ that need to be returned to their goal location, starting from the last location v in q. If v contains a finished agent s (line 29), then that agent has been moved off its goal, and we check in line 31 whether its goal location is occupied by another agent r. If there is no such agent r, then we simply move agent s back to its goal location; otherwise, in line 34, we break the while loop from line 26, thereby starting a new iteration of the while loop from line 14 with the agent r.

4. Analysis of Push and Rotate

In this section, we will first prove the correctness and completeness of Push and Rotate, before analyzing the computational complexity in section 4.1. The correctness is proved in Theorem 1, which makes use of Proposition 3, and lemmas involving the `push` and `rotate` operations (the proofs of Lemma 1 and Proposition 3 are quite long and have been moved to Appendix B).

The following lemma proves that, for a solvable instance, if the `push` operation fails, then the two agents involved must belong to the same subgraph (and, because of Proposition 3, the `swap` operation will succeed).

Lemma 1. *Suppose* `push` *(Algorithm 4) is called in the context of Algorithm 7 for an agent r moving to vertex v. If* `push` *does not succeed, then r and $s = \mathcal{A}^{-1}(v)$ are assigned to the same subgraph.*

The next lemma shows that the `rotate` operation is sound.

Lemma 2. *For an instance $(\mathcal{G}, A, \mathcal{A}, \mathcal{T})$ with at least two empty vertices, the* `rotate` *operation moves all agents in a cycle forward by one step.*

Proof. Consider Figure 14; to see that the `rotate` operation will always succeed given two empty vertices, note the following:

- Because there are at least two empty vertices in $\mathcal{G}$, we can find a path p from v to an empty vertex. Since the operations required to clear v will be reversed at the end of the `rotate` operation, it is always possible to `push` agents along p.
- Since all agents in the cycle are assigned to the same subgraph, agents a_1 and a_2 can `swap`.

□

Theorem 1. *Push and Rotate is complete for the class of Multi-agent Pathfinding problems with at least two empty vertices.*

Proof. In this proof we will focus on the correctness of Algorithm 8, `solve`. That the approach of Algorithm 7 — of dividing the graph into subgraphs and solving the subgraphs sequentially — is sound was proven by Kornhauser (1984). Here, we will prove that Algorithm 8 returns a solution if one exists, and false otherwise. The idea behind the proof is:

1. In each iteration of `solve` in which an agent is selected that is not in $\mathcal{F}$ (line 8), an agent is added to $\mathcal{F}$.
2. In case a finished agent has been moved off its goal location, then its current location is adjacent to its goal location, and the current location is on a path q.
3. After a finite number of iterations, all vertices on q will have been processed, restoring out-of-position agents in $\mathcal{F}$ to their goal location.

To prove the first point, consider an agent r in the while-loop of line 6. First a shortest path p is determined to its goal. If the graph is a polygon (`is_polygon` = true, line 5), then a shortest path is found in the graph $\mathcal{G} \setminus A[\mathcal{F}]$. In each iteration of the while-loop in line 14, agent r is moved to v, the next vertex on p:

- If $v \in q$, then there is a cycle C in q, as q is constructed from vertices that have already been visited (line 23). The `rotate` operation will move any agents on C one step forward (i.e., in the direction of the C). Lemma 2 shows that this rotate operation is possible. The result of the `rotate` is that all agents in $\mathcal{F} \cap \mathcal{A}^{-1}(C)$ are returned to their goal positions (since a `swap` has moved these agents one step backwards along q) and agent r moves to v. Also, q will now be updated such that the cycle is removed.
- If `push` succeeds for agents r and s, then agent s will be pushed out of the way, and agent r will move to v.
- Otherwise the `swap` operation will be executed. If `push` does not succeed for r and s, then, due to Lemma 1, we have $f_{\mathcal{S}}(r) = f_{\mathcal{S}}(s)$, and this in turn implies that `swap` will succeed (Proposition 3), hence agents r and s will be swapped successfully.

To prove the second point, note that an agent in $s \in \mathcal{F}$ can only be moved off its goal location as a result of the `swap` operation, which moves it to a location adjacent to its goal location. To see that s cannot be moved further away[12] from its goal location by subsequent operations, note that all agents on q will be moved back to their destination after agent r, with which agent s has swapped, has reached its destination. If, during the processing of q, the current location v of s is encountered again, then a `rotate` operation is performed, returning s to its destination.

To prove the third and final point, consider an agent s that has been moved off its goal position $\mathcal{T}(s)$. Then, in line 30, we assign to r the agent occupying the goal position of s. If there is no such agent, we can return s to its goal position using a single `move`. Otherwise, we will break the loop that iterates over q — thus the number of iterations is finite — and start a new iteration of the loop from line 6, for the agent r on $\mathcal{T}(s)$. Once this agent r has been added to $\mathcal{F}$, a new processing loop for q is started, and this occurs at most $|A|$ times. □

4.1 Runtime Analysis

Let k denote the number of agents in the instance and n the number of vertices in the roadmap. In order to solve an instance, each agent needs to be sent to its goal position, along a shortest path of length $\leq n$.

For each step along this shortest path, the `rotate`, `push` or `swap` operation will be performed. Out of these operations, the runtime of `swap` operation is dominant[13], which simplifies the following equation:

$$t_{\texttt{solve}} = O(k \cdot n \cdot t_{\texttt{swap}})$$

The `swap` tries `multipush` and `clear` on all v with sufficient degree. While further analysis may show that only a very limited ($O(1)$) number of vertices need to be checked (indeed, such behavior was observed in our experiments), for now we will have the following:

$$t_{\texttt{swap}} = O(n \cdot (t_{\texttt{multipush}} + t_{\texttt{clear}}))$$

Both these operations require $O(n \cdot t_{\texttt{clear\_vertex}})$ time. The `clear_vertex` operation can find a free vertex with a simple breadth-first search, resulting in $O(|V| + |E|) = O(n^2)$. This leads to the following runtime complexity for the `solve` operation:

$$t_{\texttt{solve}} = O(n^5 \cdot k)$$

4.2 Solution Quality

Each of the k agents has to move along its (shortest) path (of length $\leq n$) towards its goal position. Just like in the runtime analysis, the `swap` operation (Algorithm 5) is a dominant factor in the output of the `solve` operation (Algorithm 8), as `push` moves at most k agents along a path of length at most k (as in the runtime analysis, the worst-case performance of `rotate` is determined by its call to `swap`). This yields the following expression for the number of moves that the `solve` operation outputs:

$$l_{\texttt{solve}} = O(k \cdot n \cdot l_{\texttt{swap}})$$

The `swap` operation moves two agents to a vertex v by using `multipush` (Algorithm 11), and then clears two neighbors of v. While many different attempts are made in the `clear` (Algorithm 12)

---

[12] Note that during the operation of the `swap` operation, agents may temporarily be moved out of the way, but these moves are reversed before the end of the `swap` operation.

[13] Actually, `rotate` calls `swap`, but its worst-case running time is also determined by the call to `swap`.

operation to clear the two neighbors of v, the total output of moves is not more than a constant times the number of moves generated by `clear_vertex` (Algorithm 10) plus some constant number of moves: $O(l_{\texttt{clear\_vertex}})$, which is $O(n)$, as `clear_vertex` pushes agents backwards along a path to an empty vertex. Since the `multipush` operation executes the `clear_vertex` for each step along the way, its output is a dominant factor.

$$l_{\texttt{solve}} = O(k \cdot n \cdot l_{\texttt{multipush}})$$

$$l_{\texttt{solve}} = O(k \cdot n^2 \cdot l_{\texttt{clear\_vertex}})$$

$$l_{\texttt{solve}} = O(k \cdot n^3)$$

Note that the `swap` operation makes too little progress (it advances an agent one step) to achieve the $O(n^3)$ bound that was shown to be possible by Kornhauser (1984). In Section 6, we will investigate to what extent this higher number of moves in the worst case manifests itself in practice, by comparing our algorithm to the Bibox algorithm (Surynek, 2009), which does achieve an $O(n^3)$ bound on the number of moves required (although Bibox only works on biconnected graphs).

5. Heuristics and Post-Processing

Push and Rotate does not guarantee an optimal solution, and we can try to improve solution quality through heuristics, and by processing the initial solution. In Section 5.1, we will discuss a post-processing step that detects unnecessary moves, but first we will discuss at which points in the Push and Rotate algorithm a heuristic might be used that can affect the solution quality (in the order in which they appear in the text):

1. From the `push` operation (Algorithm 4), the `clear_vertex` operation (Algorithm 10) is called, which must choose an empty node to move other agents into.
2. Similarly, the `swap` operation (Algorithm 5) must choose a vertex v of degree ≥ 3 at which to perform the swap.
3. In Algorithm 8 (`solve`), the order in which agents are planned is decided in line 8.
4. Also in Algorithm 8, p is assigned a shortest path to its destination, in line 12; if there are multiple shortest paths, then one must be chosen.

So far, we have only considered heuristics for the third and fourth points. With regard to the first two points, if we consider empty vertices or possible swap locations in a specific order, then a different plan can be found assuming that multiple nodes are viable. With regards to the third point, the agent ordering can be very important to solution quality as we shall show in Section 6. Note that the agent ordering is partially determined by Algorithm 3, which introduces precedence constraints between subgraphs. Within a subgraph, a heuristic can be used to order the agents.

Our agent-priority heuristic aims to limit the number of `swap` operations that are required. The heuristic first determines the diameter of the graph, and then chooses two vertices between which the distance is exactly the diameter. We first move agents away from one of these vertices[14], ensuring that all empty spaces are at this vertex and the vertices closest to it. Agents are ordered based on

---

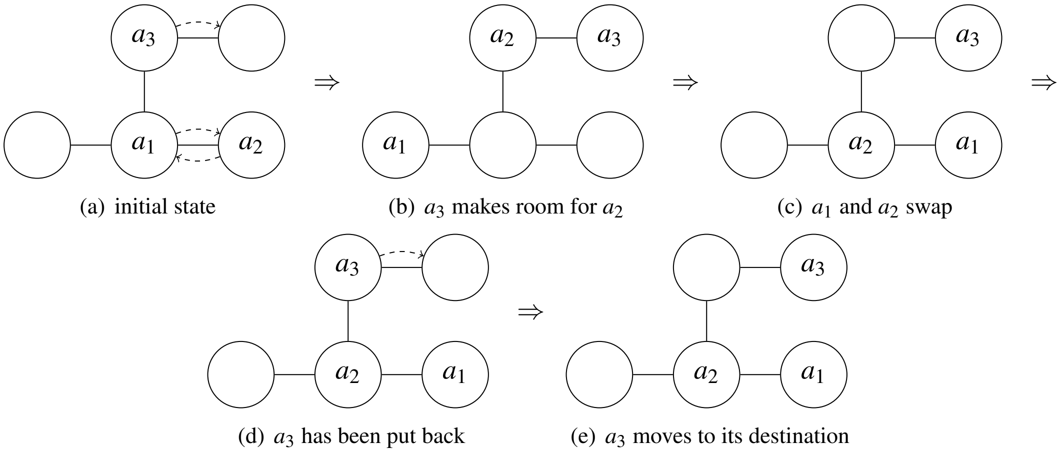
[14] The moves produced by this pre-processing are part of the solution.

the distance of their destination to this vertex, with the farthest agents receiving the highest priority. The idea is that, when a subset of the agents has been moved to their destinations, empty spaces will never get 'stuck' behind these finished agents. Remember that the `push` operation is not allowed to move agents with higher priority. So as long as empty spaces can be reached, the `push` operation can be performed, which is much cheaper than the otherwise required `swap` operation.

For the fourth point, we have implemented a heuristic that finds a path with the minimum number of agents in $\mathcal{F}$ on it (if there exist multiple such paths, we select a shortest one). Again, we aim to avoid the use of the costly `swap` operation.

5.1 Removing Redundant Moves

The Push and Rotate algorithm is capable of outputting redundant moves. An example is shown in Figure 16: as part of the `swap` between agents a_1 and a_2, agent a_3 will be moved into its goal location (in particular, the `clear` operation will move agent a_3 into its goal location, see Figure 16(b)). Since the moves that are generated by the `clear` operation will be executed in reverse after the exchange of position between agents a_1 and a_2, agent a_3 will be moved back to its initial position (Figure 16(d)). Finally, when agent a_3 is planned it is moved into its goal location for the second time. Clearly, the final two moves of the sequence are redundant.



(a) initial state (b) a_3 makes room for a_2 (c) a_1 and a_2 swap

(d) a_3 has been put back (e) a_3 moves to its destination

Figure 16: Example of redundancy: after the `swap` of a_1 and a_2, a_3 will be moved back.

A sequence of agent moves is redundant if the agent visits a vertex for a second time, and no agents have visited the vertex in the meantime. When redundant moves are removed from the solution, other moves may become redundant too. The approach taken by Luna and Bekris (2011a) is to keep iterating over all moves in the solution, until no more redundancies are discovered.

In order to remove redundancies efficiently, Algorithm 9 below uses a linked-list-like structure, encoded in the following functions. The $P_A : \Pi \to \Pi$ (previous-agent) and $N_A : \Pi \to \Pi$ (next-agent) functions are back and forward pointers respectively that form a doubly-linked chain of all the moves of a specific agent. The $P_V : \Pi \to \Pi$ (previous-vertex) and $N_V : \Pi \to \Pi$ (next-vertex) functions are similar pointers that form a chain of all the moves to a specific vertex.

Algorithm 9 starts with a single pass (lines 2 to 4) over all moves to find redundancies. All redundant moves are added to Q, and are processed in the while loop starting from line 5. First, in line 6,

Algorithm 9 `smooth`(Π)

```
1:  $Q \leftarrow$ empty queue
2:  for all $(a, v) = \pi \in \Pi$ do
3:      if agent$(P_V(\pi)) = a$ then
4:          $Q$.add$(P_V(\pi))$
5:  while $|Q| > 0$ do
6:      $\pi \leftarrow$ retrieve and remove next element from $Q$
7:      $\pi' \leftarrow N_A(\pi)$
8:      while $\pi' \neq N_V(\pi)$ do
9:          $\Pi \leftarrow \Pi \backslash \pi'$
10:         $P_A(N_A(\pi')) \leftarrow P_A(\pi')$, $N_A(P_A(\pi')) \leftarrow N_A(\pi')$
11:         $P_V(N_V(\pi')) \leftarrow P_V(\pi')$, $N_V(P_V(\pi')) \leftarrow N_V(\pi')$
12:         if agent$(P_V(\pi'))$ = agent$(N_V(\pi'))$ then
13:             $Q$.add$(P_V(\pi'))$
14:         $\pi' \leftarrow N_A(\pi')$
15: return $\Pi$
```

a redundant move $\pi = (a_i, v)$ is retrieved from Q. As π is redundant, this means that agent a_i's plan contains a number of moves to other vertices before returning to vertex v — without other agents having visited v in between. This means that from the sequence $[\pi, N_A(\pi), N_A(N_A(\pi)), \cdots, N_V(\pi)]$ all moves except for the first can be removed; this is achieved in the while loop of line 8.

Within the while loop of line 8, first a redundant move is removed from the sequence of moves (line 9), and then the pointers P_A and N_A, and P_V and N_V are updated in lines 10 and 11 respectively. Finally, it is checked whether other moves have become redundant as a result of this removal, in line 12. This check is done in constant time, and does not require looping over all moves again, as in the algorithm by Luna and Bekris (2011a).

5.2 Executing Moves in Parallel

The makespan of a multi-agent plan is the difference in time between the moment all agents have stopped moving and the moment the first agent started moving. Push and Rotate's initial output is a list of agent moves, with no specification of which agent moves can be performed at the same time, so the makespan is initially equal to the number of moves. In this section, we briefly discuss the `condense` function[15], which tries to reduce the makespan by executing as many agent moves in parallel as possible.

We can distinguish three models with regard to the allowed degree of parallelism. The least restrictive model allows an agent to move to a vertex from which another agent is just moving away. This, however, is not the Multi-agent Pathfinding problem as defined in Section 2, as it allows agent movement even if there are no empty vertices. The most restrictive model is that an agent is only allowed to move to an empty vertex, and this means that in each time step, only m moves can be performed in parallel, where m is the number of empty vertices. We currently employ the most restrictive model.

---

[15] We do not include an algorithmic description in this paper for `condense`. Although the algorithm is not very complicated, it is still somewhat lengthy to write down.

An intermediate model, also supported by our `condense` function, is that agents are allowed to move to vertices that are being vacated, as long as at the head of a chain of moving agents, there is an empty vertex. Note that this degree of parallelism does not conflict with the Multi-agent Pathfinding problem: it is still possible to serialize the moves such that each agent always moves to an empty vertex.

The idea behind the `condense` function is as follows: for every time step starting from the first, it inspects all empty vertices, and places all moves going into an empty vertex into the same time step. In case the intermediate model is employed, the empty-vertex variables are then updated to the vertices that have just been vacated, and it is checked whether there are any agents moving into these vertices. This process continues until no more moves into ‘emptying’ vertices can be found.

6. Experiments

In this section we compare the performance of Push and Rotate with that of MAPP, Push and Swap, and Bibox. We previously compared MAPP to Push and Rotate on a game map[16] from Baldur’s Gate II (De Wilde et al., 2013), and we repeat that experiment here (on a different map). Our experiments on game maps are very preliminary as they did not lend themselves to comparisons with other algorithms: MAPP produces too many moves, Push and Swap requires a lot of computation time, while Bibox cannot solve the instances that are not biconnected.

Push and Swap is conceptually similar to Push and Rotate, so we would expect its performance to be similar also. In addition, apart from the game map, all our experiments were conducted on biconnected instances (to allow a comparison with Bibox), so the fact that Push and Swap does not take into account subgraphs does not affect its success ratio for those instances. However, it turned out that the other source of incompleteness of Push and Swap, namely the fact that recursive calls to `swap` may fail — which we solved with the introduction of the `rotate` operation — resulted in Push and Swap solving only a fraction of all problem instances.

The main focus of our experiments is therefore on Bibox. Bibox is not complete, as it requires a biconnected graph, but it performs well compared to other approaches (Surynek, 2009). Moreover, it achieves a bound of $O(n^3)$ on the number of moves, which Kornhauser (1984) showed to be the lowest worst-case bound achievable for general graphs. Push and Rotate, by contrast, has a bound of $O(kn^3)$ on the number of moves, where k is the number of agents. We have compared to Bibox on two types of instances: instances from a random generator that is part of the Bibox code, and grid instances.

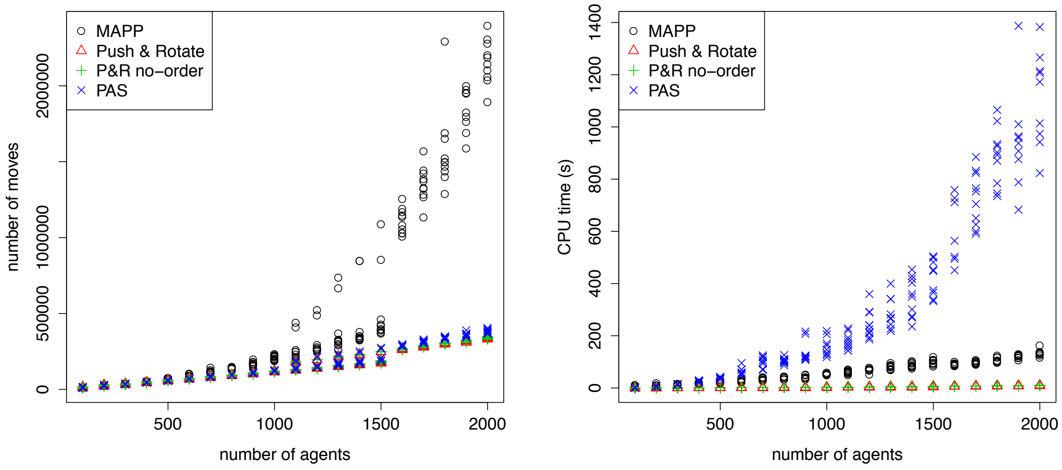
For most experiments, we have run Push and Rotate both with and without the agent-ordering heuristics from Section 5. The effectiveness of the heuristic is shown by the fact that it is almost always better, in terms of number of moves and therefore also in terms of CPU time, to use the agent-ordering heuristic with Push and Rotate.

6.1 Map AR0603SR from Baldur’s Gate II

The problem instances in the benchmark set are characterized by a large set of vertices, many of which are unoccupied. The map we chose for these experiments has 13765 vertices, and between 100 and 2000 agents (step size 100).

---

[16] A set of benchmark maps from the video game industry is available at `http://movingai.com/benchmarks/`.



(a) number of moves (b) CPU times

Figure 17: Comparison on map AR0603SR of Push and Rotate, Push and Swap (PAS), Push and Rotate without agent-ordering heuristic (P & R no-order) and MAPP.

Figure 17 shows the number of moves produced by Push and Rotate, MAPP, and Push and Swap (no Bibox, as the map was not biconnected), and the CPU times. Both Push and Rotate and Push and Swap produce very efficient plans (which are in fact always within a few percent of the lower bound of the sum of the shortest paths), but MAPP requires many more moves to find a solution, and therefore we did not try to include it in our further experiments. Note that for these instances, it does not matter whether or not the agent-ordering heuristic is employed for Push and Rotate.

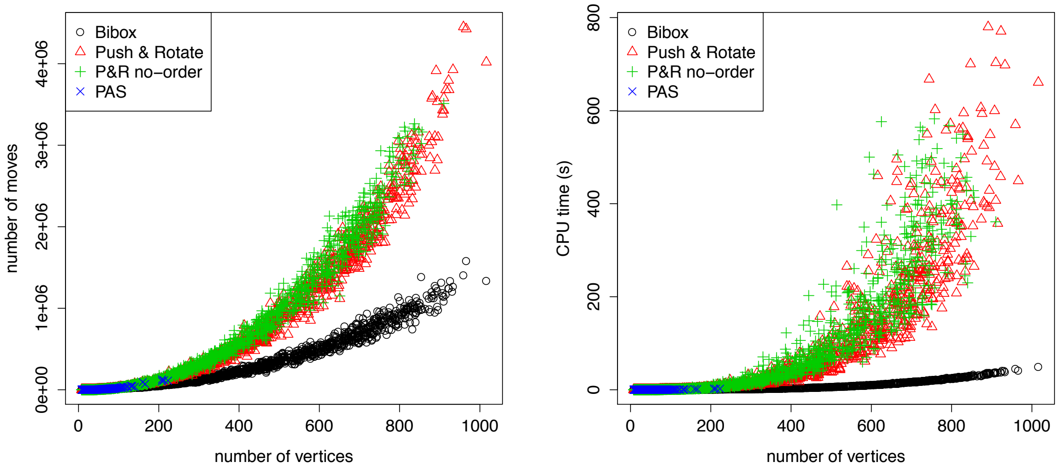
From Figure 17(b) it is interesting to note that Push and Swap is quite a bit slower for this type of instance, while for the instances in the remainder of our experiments section (grids and other biconnected instances) the C++ implementation of Push and Swap (when it did find a solution) was often a little bit faster than our Java implementation of Push and Rotate.

6.2 Random Biconnected Instances

Surynek's code[17] generates random instances by iteratively adding handles to an initial cycle (see Section 2.1) according to three parameters: the number of handles, the size of the initial cycle, and the maximum handle length (where the length of the next handle is uniformly chosen between 0 and the maximum handle length minus one, but set to 1 if it equals 0). In addition, one can choose the number of empty vertices, with a default value of 2. In our experiments we measure the number of moves, the CPU time, and the makespan, which is the number of time steps required to get all agents to their destination (in one time step an agent can perform one move, though multiple agents can move during one time step).

---

[17] We used the code available at `http://ktiml.mff.cuni.cz/~surynek/research/icra2009/`.



(a) number of moves

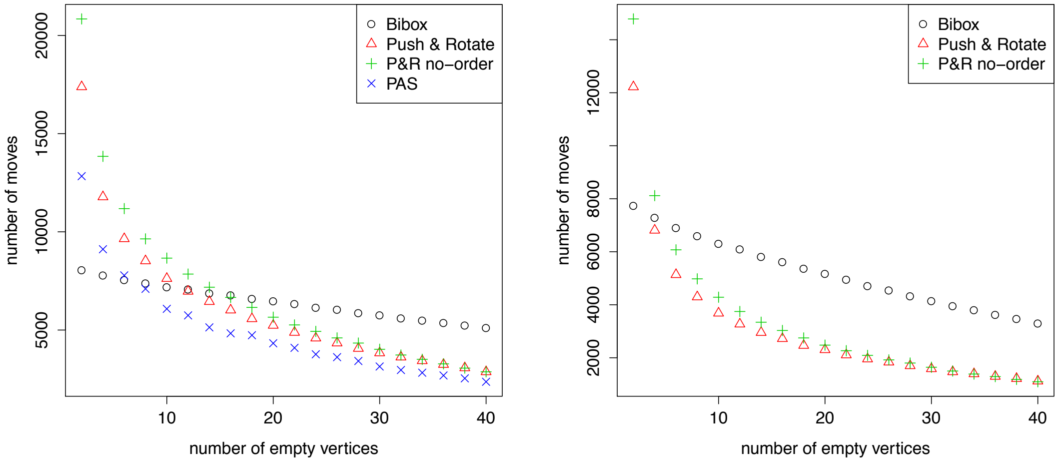
(b) CPU time

Figure 18: Comparison on random instances with parameters (handles, initial cycle, max handle length): (20, 20, 20) to (50, 50, 50), with 2 empty vertices.

Figure 18 compares Push and Rotate, Push and Swap and Bibox on instances generated according to a single variable *x* that ranged from 20 to 50, with steps of two, that was used for all three parameters (number of handles, initial cycle size, and maximum handle length), while the number of empty vertices was kept at two. Clearly the number of moves required for Push and Rotate rises much more quickly with instance size, and consequently CPU times increase as well. The results for Push and Rotate are slightly better in terms of makespan, but still much worse than Bibox. We can also see from the figures that the agent-ordering heuristic is useful, but not essential for the performance of Push and Rotate. Push and Swap only managed to solve 1.17% of these instances, so it is not really usable on such hard instances; later we shall see that it performs better if there are more empty vertices.

Bibox works exceptionally well for large handle sizes, which is unsurprising since the algorithm is based on the concept of filling handles: Bibox inserts vertices into their destination handle in the right order, so once at their destination, vertices are not moved much. Push and Rotate, on the other hand, has to bring agents into place using `push` and `swap`; to swap two agents in the middle of a handle, many agents have to be moved out of the way in order to bring them to a node at which they can swap. P & R's `rotate` operation is rarely used for this type of instance, as a path of resolving agents rarely intersects itself.

Figure 19 shows experiments on random biconnected instances with 40 handles, initial cycle size 5, maximum handle length 10, and an increasing number of empty vertices, ranging from 2 to 40 (step size 2), and 180 instances per step. On the hardest instances, with few empty vertices, Push and Rotate still produces more costly plans, but with a higher number of empty vertices, Push and Rotate produces much better plans. Such instances are easier for Push and Rotate, as first of all the



(a) average number of moves

(b) average makespan

Figure 19: Comparison on random instances with 40 handles, initial cycle 5, and maximum handle size 10.

`push` operation will succeed more often, and if a `swap` is still necessary, it will be easier to `clear` a node at which two agents can swap.

Bibox, on the other hand, requires exactly two empty vertices in its current implementation, and fills up the remaining empty vertices with dummy agents. Even though the moves involving these dummy agents are removed from the final solution, Bibox does not manage to benefit from more empty vertices to the extent that Push and Rotate does. Bibox still produces solutions more quickly, however, requiring a few tenths of a second for each instance; Push and Rotate requires around 6 seconds to solve the hardest instances, and around 1 second for the easiest ones. Figure 19 suggests that Push and Swap also performs very well, although it should be noted that the success ratio for Push and Swap is still only 53.9% for these experiments.

In terms of makespan (Figure 19(b)), the performance advantage of Push and Rotate over Bibox is even greater[18]. This is something we observed in all our experiments comparing Push and Rotate with Bibox. The main reason seems to be that our `condense` is more powerful than its counterpart in the Bibox algorithm. Table 1 shows the parallelism, which is the number of moves divided by the makespan, for Push and Rotate, Bibox, and Bibox plus our `smooth` (removing redundant moves) and `condense` functions ('Bibox++'). Without using our post-processing algorithms, Bibox has very poor parallelism, but after applying `smooth` and `condense` it is more parallel than Push and Rotate for the emptier instances. However, Bibox still has more moves, so the makespan of Push and Rotate and Bibox++ was about equal on average.

---

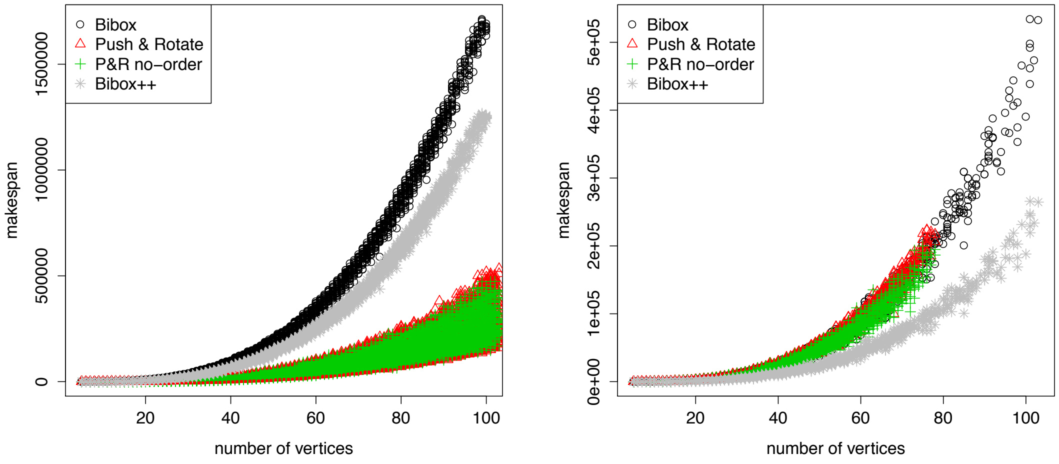
[18] The implementation we had of Push and Swap did not have any kind of condense feature, so its makespan would simply be the number of moves, which we did not include in the figure.

Empty	2	6	10	14	18	22	26	30	34	38	42	46	50
P & R	1.42	1.86	2.10	2.25	2.35	2.42	2.48	2.54	2.57	2.63	2.67	2.71	2.74
Bibox	1.03	1.07	1.10	1.12	1.15	1.18	1.20	1.23	1.26	1.29	1.33	1.36	1.39
Bibox++	1.22	1.49	1.69	1.88	2.05	2.25	2.44	2.64	2.83	3.07	3.29	3.5	3.69

Table 1: Parallelism of Push and Rotate, Bibox, and Bibox plus `smooth` and `condense`, on biconnected from Figure 19.

6.2.1 THE INITIAL CYCLE

The Bibox algorithm solves the handles of an instance by inserting agents into one endpoint of the handle, and using the 'lower' part of the graph (i.e., the initial cycle and the handles with a lower number) to move agents and empty vertices around. For the initial cycle, a different procedure is followed in which a `swap`-like operation is used to exchange the position of the agent at vertex *i* with the agent whose destination is *i*.



(a) initial cycle: $\{4,\ldots,100\}$, max handle size: 4

(b) initial cycle: 4, max handle size: $\{4,\ldots,100\}$

Figure 20: Makespan comparison for instances with an initial cycle and one handle, and two empty vertices.

Figure 20 shows, however, that this initial-cycle procedure is not as efficient as the processing of the handles, or indeed as Push and Rotate[19]. Figure 20(a) shows a comparison on instances in which the length of the initial cycle varied from 4 to 100, with a single handle of maximum size 4. Figure 20(b) shows experiments on similar types of graphs, but these with a small initial cycle of 4 vertices, and a maximum handle length that varies from 4 to 100. The number of empty vertices in both figures is two. In the first setting, Push and Rotate performs considerably better than

---

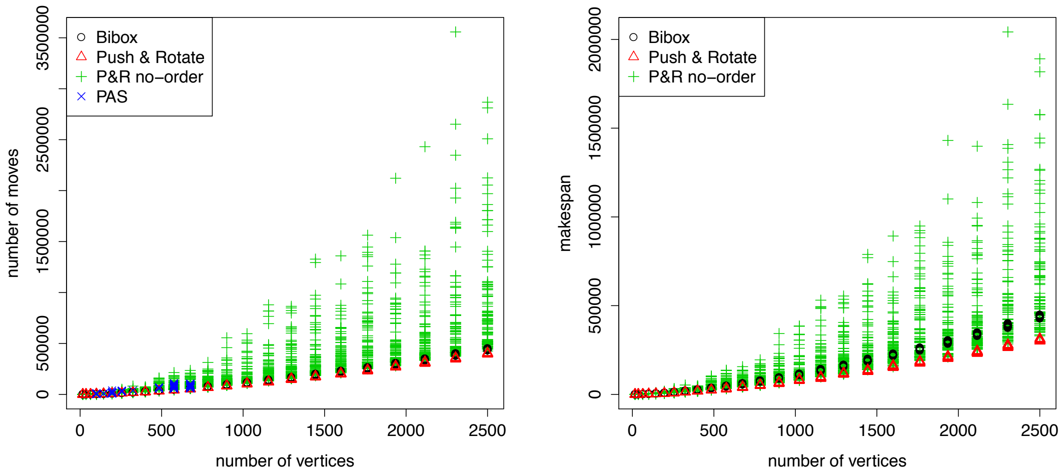
[19] Due to a scripting error, the experiments with a handle size of more than 80 were only run for Bibox(++).

Bibox, while in the second setting the relative performances are similar to the other experiments with random instances. For these experiments, removing redundant moves from the solution of Bibox using our `smooth` algorithm removed only a small percentage of moves, while trying to reduce makespan with `condense` did result in a significant reduction. However, even the condensed Bibox output was far worse, for large initial cycles, than the plans produced by Push and Rotate.

It is interesting to note that Bibox produces in excess of n^3 moves for instances with a large initial cycle — it produces around two million moves at 100 vertices — yet CPU times are less than a second for all instances. For Push and Rotate, by contrast, the CPU times grow with the number of moves; around 5 seconds for 50 vertices, and up to 70 seconds for instances of 100 vertices.

6.3 Grid Instances

For any problem instance, Bibox requires a specification of the handle decomposition of the graph. For the grid instances we generated, we adopted the decomposition suggested by Surynek (2011): the initial cycle consists of the four vertices in the top left corner of the grid, and handles are added first to the right of the initial cycle, then below the initial cycle, and finally the remaining vertices are added one by one, from top left to bottom right.



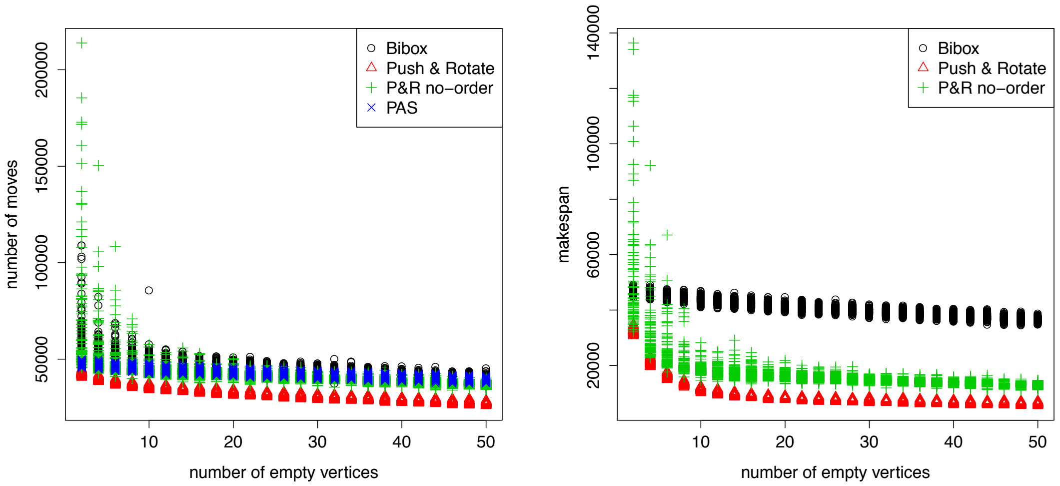
(a) number of moves

(b) makespan

Figure 21: Comparison on grid instances with sizes from 4 × 4 to 50 × 50, with 2 empty vertices.

Figure 21 shows that for grid instances with only two empty vertices, the performance of Push and Rotate relies heavily on the agent-ordering heuristic. With the heuristic enabled, Push and Rotate performs consistently better than Bibox; without it, the performance is usually much worse. For this set of experiments, Push and Swap solved only 31.9% of the instances, and very few beyond grid sizes of 25 × 25.

We also conducted experiments with increasing numbers of empty vertices, on 24 × 24 grid instances. These problems turn out to be relatively easy for Push and Rotate, but Bibox is unable to leverage the increased freedom of movement, and produces quite costly plans. Figure 22(a) shows



(a) number of moves

(b) makespan

Figure 22: Increasing number of empty vertices for 24 × 24 grid instances.

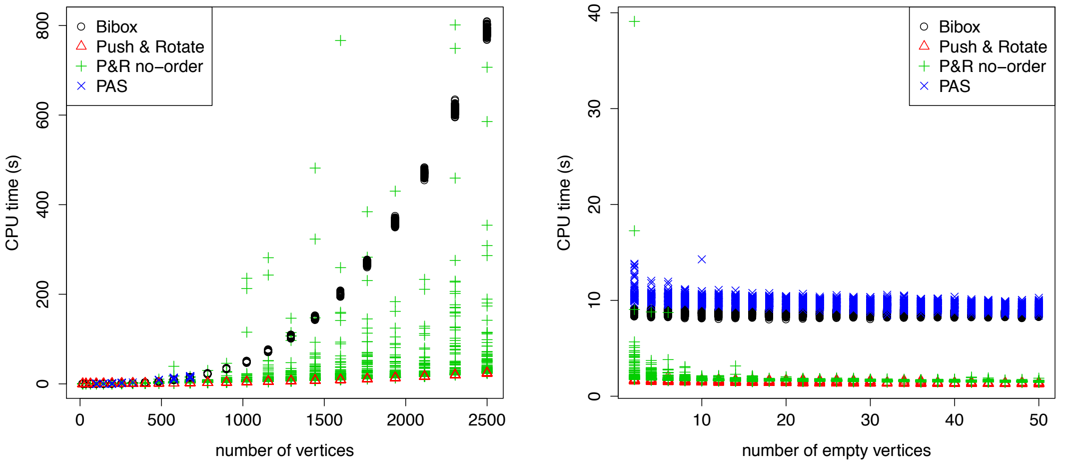
that for two empty vertices, Bibox produces around 10% more moves; for 40 empty vertices, it requires around 35% more moves. In terms of makespan, the difference between the algorithms is even larger: for two empty vertices, the span of the Push and Rotate solution is around 75% of Bibox's solution, while for 40 empty vertices it is less than 25%. These instances proved just easy enough for Push and Swap, and 98.1% of the instances were solved, with a performance comparable to that of Push and Rotate without the agent-ordering heuristic.

Finally, in Figure 23 we present the CPU times for the algorithms on grid instances; Figure 23(a) for the experiments of increasing grid sizes, and Figure 23(b) for the experiments with increasing numbers of empty vertices. First, an interesting thing to note is that Push and Rotate's good performance (heuristic enabled) on grids is also expressed in terms of lower CPU times, while Bibox's times grow steadily with the number of moves. Second, the algorithms are not sensitive, in terms of CPU times, to the number of empty vertices, even though all algorithms produce solutions containing fewer moves.

6.4 Experiment Conclusions

From the experiments in this section it is apparent that the performance of both algorithms depends considerably on the type of problem instance. Bibox is well-suited to the type of random instances generated by the Bibox program, while Push and Rotate performs really well on grid instances, as long as the agent-ordering heuristic is employed. Although we haven't been able to run Bibox on any game maps from `movingai.com`, we suspect that Push and Rotate will perform better on these instances: there are plenty of open spaces (essentially grids), and many free vertices, too, both of which play to the strengths of Push and Rotate.

With regard to the other algorithms on comparison, Push and Swap performed reasonably well when a plan was found, but for larger instances, especially with only a few empty vertices, a solution



(a) grid sizes from 4 × 4 to 50 × 50, 2 empty vertices (b) 24 × 24 grids with increasing number of empty vertices

Figure 23: CPU times on grid experiments.

was rarely found, so it is not practically usable. Similarly, the solutions produced by MAPP contain too many moves, and also it is only guaranteed to return a solution on instances of the SLIDABLE class. Finally, Push and Rotate without the agent-ordering heuristic often performs well, but it is never significantly better than employing the heuristic, and often significantly worse, so for the instances we've seen there is no reason not to use it.

With regard to CPU times, if the problem domain suits the algorithm, CPU times are low and do not grow at the same rate as the number of moves; otherwise, CPU times grow steadily with the number of moves. An exception seems to be the performance of Bibox on random biconnected instances with a small single handle and a large initial cycle: in a few tenths of a second, Bibox manages to produce millions of moves. It should be noted, however, that Bibox's post-processing algorithms are quite rudimentary and do not spend a lot of CPU time improving the solution. For Push and Rotate, on the other hand, the post-processing algorithms are important to solution quality, but they also make up about 1% to 5% of the total computation time.

7. Conclusions and Future Work

We have presented a complete and polynomial-time algorithm for the multi-agent pathfinding problem. Our approach is similar to that of Luna and Bekris (2011b); using simple the primitive operations `push`, `swap`, and, in our algorithm, `rotate`, an algorithm can be constructed that is both easy to understand and performs competitively. Although our Push and Rotate does not achieve the (best possible) worst-case bound of $O(n^3)$ moves (n the number of vertices in the graph), it produces competitive solutions when compared to Bibox, which does achieve the $O(n^3)$ bound (Surynek, 2011), but is restricted to biconnected graphs.

We argue that Push and Rotate is currently the algorithm of choice for multi-agent pathfinding problems. Bibox performs comparably well, but it is only applicable to biconnected graphs, and moreover the algorithm is not yet very mature, for instance in the way it handles more than two empty vertices, and its post-processing algorithms are comparatively basic. Finally, there is the work by Kornhauser (1984), which demonstrated the $O(n^3)$ bound on the number of moves. Surynek reports that his implementation of Kornhauser's work performs considerably worse (in terms of the number of moves) than his Bibox algorithm. Moreover, Kornhauser's algorithm is very difficult to understand and therefore to implement. A more accessible specification and implementation are under construction (Röger & Helmert, 2012), and we are in contact with the authors to compare their work with ours in due course.

A useful next step in multi-agent pathfinding would be to develop an algorithm that guarantees an $O(n^3)$ bound, performs (at least) competitively with Bibox and Push and Rotate, and is easy to understand and implement. Arguably, Push and Rotate ticks two of these boxes, so it would make sense to try and adapt it to reach the $O(n^3)$ bound. This would probably entail replacing the expensive `swap` operation, which can require $O(n^2)$ moves just to further one agent a single step, and make more use of rotations, as already done by the Bibox algorithm and our `rotate` operation.

Acknowledgements

This research was sponsored by the SUPPORT project from the Dutch Ministry of Economic Affairs.

Appendix A. Algorithms

In this appendix we present the algorithms used but not defined in the main text.

A.1 Operation `clear_vertex`

Algorithm 10 `clear_vertex`$(\Pi, \mathcal{G}, \mathcal{A}, v, \mathcal{U})$

```
 1: for all unoccupied vertex $u \in V$ do
 2:     $p \leftarrow$ shortest path in $\mathcal{G} \setminus \mathcal{U}$ from $u$ to $v$
 3:     if $p \neq \perp$ then
 4:         $x' \leftarrow \perp$
 5:         for all vertices $x$ on path $p$ (in order) do
 6:             if $x' \neq \perp$ then
 7:                 $r \leftarrow \mathcal{A}^{-1}(x)$
 8:                 move($\Pi, \mathcal{A}$, agent $r$ to vertex $x'$)
 9:             $x' \leftarrow x$
10:         return true
11: return false
```

Algorithm 10 is called from the `push`, `multipush`, and `rotate` operations, and, if successful, clears a vertex v. Algorithm 10 iterates over all empty vertices $u \in \mathcal{G}$ (line 1), and starts each iteration by finding a shortest path, avoiding all blocked vertices $\mathcal{U}$, from u to the vertex v that must be cleared (line 2). If such a path is found ($p \neq \perp$, line 3), then all agents on p are moved one step towards u (the for-loop from line 5). Note that choosing the right u in line 1 can impact both the running time of the `clear_vertex` operation, as well as the total number of moves of the solution produced by Push and Rotate.

A.2 Operation `multipush`

Algorithm 11 `multipush`$(\Pi, \mathcal{G}, \mathcal{A}, r', s', v)$

```
 1: $r \leftarrow$ agent $r'$ or $s'$ closest to $v$
 2: $s \leftarrow$ the other agent
 3: $p \leftarrow$ shortest path in $\mathcal{G}$ from $\mathcal{A}(r)$ to $v$
 4: if $p = \perp$ then
 5:     return false
 6: for all vertices $x$ on path $p$ do
 7:     $v_r \leftarrow \mathcal{A}(r)$, $v_s \leftarrow \mathcal{A}(s)$
 8:     if vertex $x$ is occupied then
 9:         $\mathcal{U} \leftarrow \{v_r, v_s\}$
10:         if clear_vertex$(\Pi, \mathcal{G}, \mathcal{A}, x, \mathcal{U})$ = false then
11:             return false
12:     move$(\Pi, \mathcal{A}$, agent $r$ to vertex $x)$
13:     move$(\Pi, \mathcal{A}$, agent $s$ to vertex $v_r)$
14: return true
```

Operation `multipush` (Algorithm 11) is called from the `swap` operation, and moves two agents r' and s' that are next to each other, to a node v (which is a node of degree three or more, at which r' and s' can be swapped). First, a shortest path is found in line 3, and then the algorithm proceeds to move r and s forwards along this path one step at a time, calling `clear_vertex` if the next vertex on the path is non-empty.

A.3 Operation `clear`

The `clear` operation (Algorithm 12) is called from `swap`, and attempts to clear two neighbors of a potential swap node. The algorithm works in four stages: in the first stage (Figure 24(a)) it simply attempts to push agents, that are occupying neighbors of v, away from v. If there are two vertices cleared in this stage, the operation is done. Otherwise, the other three stages need one unoccupied vertex next to v to work. If there is no unoccupied vertex after stage one, the `clear` operation fails.

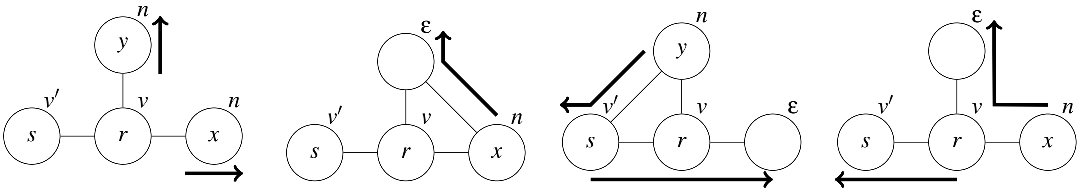
In the second stage (Figure 24(b)) a neighbor n is required that has a path to the already empty neighbor ε that does not go through v. The agent at n is first moved to ε, and if the `clear_vertex` operation subsequently succeeds on ε (with n, v, and v' blocked vertices), then `clear` succeeds.

The third stage (Figure 24(c)) checks whether an agent occupying a neighbor n of v may be able to vacate n by moving through the vertex v' that currently holds agent s. This is tried by moving agent r into the empty vertex ε, and agent s into v.

The idea behind the final stage (Figure 24(d)) is that it may be possible to create additional space behind the already empty vertex ε, similar to the second stage. In stage four, however, instead of using a direct connection between n and ε, the agent at n tries to move to ε through v, meaning that r and s must move backwards to make room.

Algorithm 12 `clear`$(\Pi, \mathcal{G}, \mathcal{A}, r', s', v)$

```
$r \leftarrow$ agent $r'$ or $s'$ on $v$ ; $s \leftarrow$ agent $r'$ or $s'$ not on $v$ ; $v' \leftarrow \mathcal{A}(s)$
$\mathcal{E} \leftarrow \{$unoccupied $n \in$ neighbours$(v)\}$
if $|\mathcal{E}| \geq 2$ then
    return true
for all $n \in$ neighbours$(v) \setminus (\mathcal{E} \cup \{v'\})$ do
    if clear_vertex$(\Pi, \mathcal{G}, \mathcal{A}, n, \mathcal{E} \cup \{v, v'\}) =$ true then
        if $|\mathcal{E}| \geq 1$ then
            return true
        $\mathcal{E} \leftarrow \mathcal{E} \cup \{n\}$
if $|\mathcal{E}| = 0$ then
    return false
$\varepsilon \leftarrow$ the vertex in $\mathcal{E}$
for all $n \in$ neighbours$(v) \setminus \{v', \varepsilon\}$ do
    $\Pi' \leftarrow [\,]$ ; $\mathcal{A}' \leftarrow \mathcal{A}$
    if clear_vertex$(\Pi', \mathcal{G}, \mathcal{A}', n, \{v, v'\}) =$ true then
        if clear_vertex$(\Pi', \mathcal{G}, \mathcal{A}', \varepsilon, \{v, v', n\}) =$ true then
            $\mathcal{A} \leftarrow \mathcal{A}'$ ; $\Pi \leftarrow \Pi + \Pi'$
            return true
        break
for all $n \in$ neighbours$(v) \setminus \{v', \varepsilon\}$ do
    $\Pi' \leftarrow [\,]$ ; $\mathcal{A}' \leftarrow \mathcal{A}$
    move$(\Pi', \mathcal{A}'$, agent $r$ to vertex $\varepsilon)$ ; move$(\Pi', \mathcal{A}'$, agent $s$ to vertex $v)$
    if clear_vertex$(\Pi', \mathcal{G}, \mathcal{A}', n, \{v, \varepsilon\}) =$ true then
        if clear_vertex$(\Pi', \mathcal{G}, \mathcal{A}', v', \{v, \varepsilon, n\}) =$ true then
            $\mathcal{A} \leftarrow \mathcal{A}'$ ; $\Pi \leftarrow \Pi + \Pi'$
            return true
        break
if clear_vertex$(\Pi, \mathcal{G}, \mathcal{A}, v', \{v\}) =$ false then
    return false
move$(\Pi, A$, agent $r$ to vertex $v')$
if clear_vertex$(\Pi, \mathcal{G}, \mathcal{A}, \varepsilon, \{v, v', \mathcal{A}(s)\}) =$ false then
    return false
$n \leftarrow$ any vertex from neighbours$(v) \setminus \{v', \varepsilon\}$ , $t \leftarrow \mathcal{A}^{-1}(n)$
move$(\Pi, \mathcal{A}$, agent $t$ through vertex $v$ to vertex $\varepsilon)$
move$(\Pi, \mathcal{A}$, agent $r$ to vertex $v)$ ; move$(\Pi, \mathcal{A}$, agent $s$ to vertex $v')$
return clear_vertex$(\Pi, \mathcal{G}, \mathcal{A}, \varepsilon, \{v, v', n\})$
```



(a) 1: `push` agents

(b) 2: move agent through empty neighbor

(c) 3: agent s makes room

(d) 4: agents r and s clear v

Figure 24: The four stages of the `clear` operation.

Note that the stages of `clear` omitted from the work of Luna and Bekris (2011b) are stages 24(b) and 24(c), so Luna and Bekris effectively considered the `clear` operation for trees.

A.4 Operation `exchange`

Algorithm 13 `exchange`$(\Pi, \mathcal{G}, \mathcal{A}, r', s', v)$

```
$r \leftarrow$ agent $r'$ or $s'$ on $v$
$s \leftarrow$ agent $r'$ or $s'$ not on $v$
$(v_1, v_2) \leftarrow$ two unoccupied neighbors of $v$
$v_s \leftarrow \mathcal{A}(s)$
move$(\Pi, \mathcal{A}$, agent $r$ to vertex $v_1)$
move$(\Pi, \mathcal{A}$, agent $s$ through vertex $v$ to vertex $v_2)$
move$(\Pi, \mathcal{A}$, agent $r$ through vertex $v$ to vertex $v_s)$
move$(\Pi, \mathcal{A}$, agent $s$ to $v)$
```

The exchange algorithm was illustrated in Figure 7 in Section 3.

Appendix B. Proofs

Lemma 1. *Suppose* `push` *(Algorithm 4) is called in the context of Algorithm 7 for an agent r moving to vertex v. If* `push` *does not succeed, then r and* $s = \mathcal{A}^{-1}(v)$ *are assigned to the same subgraph.*

Proof. Note that, as `push` is called in the context of Algorithm 7, the problem instance is solvable, and all agents with a higher priority than r have already been planned for.

To prove the lemma, we will prove the equivalent statement that if r and s are not assigned to the same subgraph, then `push` will succeed.

Case 1: r is not assigned to any subgraph By definition, r and s are not assigned to the same subgraph, and, by Proposition 3, this means that r cannot swap with any other agent. Note that, as a result of not being assigned to any subgraph, agent r is "trapped" on an isthmus (see Figure 25 for an illustration). All agents assigned to a subgraph have a higher priority than r, and are therefore in $\mathcal{F}$ by the time the call `push` is made[20].

---

[20] Note that Algorithm 8, and the operations that are called from it, make only a single call to `push`, in which the set of blocked vertices $\mathcal{U}$ is equal to the set of locations that currently hold an agent in $\mathcal{F}$.

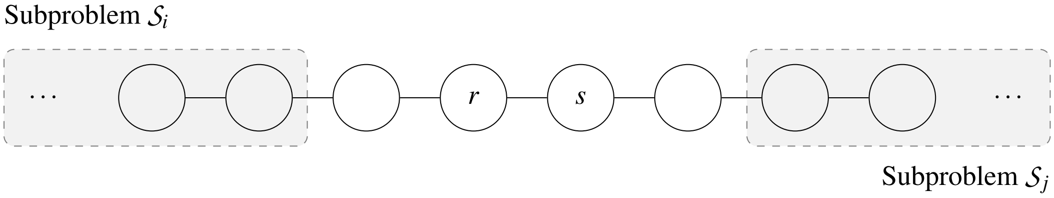


Figure 25: Agent r, trapped on the isthmus, can only move to $\mathcal{A}(s)$ using a `push`.

By way of contradiction, suppose that the instance is solvable yet `push` does not succeed. First, suppose that $s \in \mathcal{F}$; r cannot have the same goal location as s, so r's goal location must be 'behind' $\mathcal{A}(s)$. However, if it is possible for r to reach such a location, then it would be assigned to subgraph $f_{\mathcal{S}}(s)$, by Proposition 1, so a contradiction has been reached.

Next, suppose that $s \notin \mathcal{F}$. Since r has a priority greater or equal to s, we can conclude that s is also not assigned to any subgraph. If the instance is solvable yet `push`(r,s) does not succeed, then there exists an empty vertex 'beyond' s, but there is no path in $\mathcal{G} \setminus \mathcal{A}(\mathcal{F})$ that reaches it; i.e., the path to the empty vertex is blocked by agents in $\mathcal{F}$. However, if there would exist an assignment $\mathcal{A}'$ in which s would move out of the way for r, and all agents in $\mathcal{F}$ are returned to their goal location, then s will have swapped with the empty vertex that is within or beyond a subgraph. Hence, s would reach locations in which it would be assigned to the subgraph ($\mathcal{S}_j$ in Figure 25), so a contradiction has been reached.

Case 2: r is assigned to subgraph $f_{\mathcal{S}}(r)$ Since $f_{\mathcal{S}}(r) \neq f_{\mathcal{S}}(s)$, the blocking agent s can reach (at most) a plank vertex of $f_{\mathcal{S}}(r)$. Suppose that s is on the start of a plank of $f_{\mathcal{S}}(r)$; then, due to Algorithm 3, $f_{\mathcal{S}}(r) \prec f_{\mathcal{S}}(s)$. Hence, it is not possible that s, or any agents in subgraphs 'behind it', are in $\mathcal{F}$. Therefore, `push` will succeed in case the instance is solvable.

In case s is not on the start of a plank of $f_{\mathcal{S}}(r)$, then the goal location of r must be on a plank of $f_{\mathcal{S}}(r)$. Due to Kornhauser's thesis (1984), we know that r can reach any vertex of $f_{\mathcal{S}}(r)$ and its planks (the 2-transitivity result). Since the agents cannot swap (Proposition 3), agent r can only reach its goal by pushing agent s away, which is what `push` achieves without restrictions. Finally, note that $s \notin \mathcal{F}$; if s were at its goal position, then the instance would not be solvable: r and s cannot swap, so the goal location of r cannot be at or beyond $\mathcal{T}(s)$. □

Proposition 3. *For two agents r and s on adjacent vertices in $\mathcal{G}$, the operation* `swap`$(\Pi, \mathcal{G}, \mathcal{A}, r, s)$ *will succeed if and only if r and s are assigned to the same subgraph $\mathcal{S}_i$.*

Proof. First we show that if a swap between r and s succeeds then both agents are assigned to the same subgraph: $f_{\mathcal{S}}(r) = f_{\mathcal{S}}(s) \neq \varnothing$. After a successful swap, the only change in the assignment are the positions of agents r and s. Now consider the assignment $\mathcal{A}$ and the assignment $\mathcal{A}'$, which has the positions of agents r and s swapped:

$$\mathcal{A}'(a) = \begin{cases} \mathcal{A}(s) & \text{if } a = r \\ \mathcal{A}(r) & \text{if } a = s \\ \mathcal{A}(a) & \text{otherwise} \end{cases}$$

Hence, the subgraph of r prior to the swap must equal the subgraph of s after the swap. Furthermore, we know from Proposition 1 that any agent is confined to at most a single subgraph, so $f_{\mathcal{S}}(r) = f_{\mathcal{S}}(s)$.

To see that `swap` succeeding implies that $f_{\mathcal{S}}(r) \neq \varnothing$, note that the swap has occurred at some vertex v with degree ≥ 3. Vertex v is clearly part of some subgraph $\mathcal{S}_i$, and r and s have reached v with two of v's neighbors empty, so r and s are assigned to $\mathcal{S}_i$:

1. In case r and s are on a plank: $m' \geq 1 \wedge m' < m$, by choosing the right u (line 6 in Algorithm 2), see Figure 7.

2. In case either r and s are on an inner vertex: such an agent is assigned to $\mathcal{S}_i$ in line 12, Algorithm 2.

Now we show $\varnothing \neq f_{\mathcal{S}}(r) = f_{\mathcal{S}}(s)$ implies that `swap` will succeed for agents r and s. When one of the agents r and s occupies a vertex v with $\texttt{degree}(v) \geq 3$ and two empty neighbor vertices, and the other agent occupies a neighbor vertex of v, then the agents r and s can exchange positions by moving into the empty vertices in one order, and exiting them in the other order (see Algorithm 13). Hence, `swap` succeeds if and only if there exists a vertex v for which `multipush` and `clear` also succeed.

This leaves us to prove that if r and s are assigned to the same subgraph, then such a v always exists. The assignment criteria for agents to subgraphs guarantee that:

1. an agent is inside a biconnected component, or

2. an agent is less than m steps away from a vertex within $f_{\mathcal{S}}(r)$, with degree ≥ 3.

In case both agents are inside a biconnected component, `multipush` succeeds trivially. In a biconnected component there are two paths between any pair of vertices, so it is always possible to bring r and s to any vertex v with degree ≥ 3 in the biconnected component (note that the only blocked vertices that `multipush` considers are $\mathcal{A}(r)$ and $\mathcal{A}(s)$, since moves that put other agents out of place will be reversed later). Second, `clear` will always succeed for the same reason: there always exists one v with degree ≥ 3 such that the (at least two) unoccupied vertices can both be reached from v by at least two paths. Hence, there is at least one path from v to each of the empty vertices that is not blocked by r and s, so two neighbors of v can be emptied (possibly with r and s momentarily stepping aside as illustrated in Figures 24(c) and 24(d)).

In case the agents are not both inside a biconnected component, we must show that a vertex of degree ≥ 3 can be reached, and also that `clear` will succeed.

In case exactly one agent (say r) is inside a biconnected component, but agent s is not, then r already occupies a (join) vertex with degree ≥ 3. In case neither agent is inside a biconnected component, then the agents are either between two vertices of degree ≥ 3, or they are not occupying vertices that are assigned to the subgraph. Consider the two agents between two vertices with degree ≥ 3. There are not more than $m-2$ vertices between these two vertices, two of which are occupied by agents r and s. Suppose l_1 and l_2 steps are required to reach the vertices, and there are m_1 and m_2 free vertices on the respective sides of the agents. This leads to:

$$l_1 + l_2 \leq m - 4 \tag{1}$$

$$m_1 + m_2 = m \tag{2}$$

Assume one of the vertices is unreachable:

$$l_1 > m_1 = m - m_2 \tag{3}$$

Then the other vertex is reachable:

$$\begin{aligned} l_2 &\leq m - 4 - l_1 &(4)\\ l_2 &< m - 4 - (m - m_2) &(5)\\ l_2 &< m_2 - 4 &(6)\end{aligned}$$

If the agents are outside the vertices assigned to the subgraph, then the assignment criteria of agents to subgraphs state that it must be possible for any agent assigned to a subgraph to enter the subgraph while one additional empty vertex remains in the subgraph.

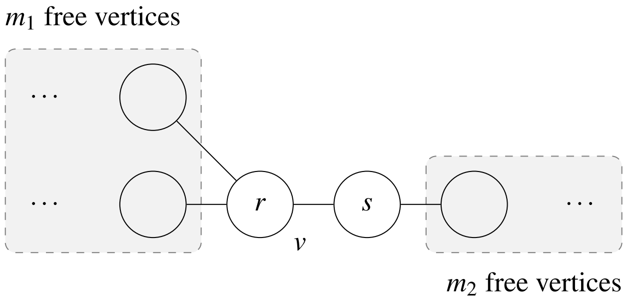


Figure 26: Illustration of reachable free vertices.

Consider the amount of free vertices on both sides of the two agents (see Figure 26).

1. Having ≥ 1 free vertices on one side of the two agents and > 1 on the other side is sufficient to make clearing two vertices easy.

2. In the case that all free vertices are on one side of the two agents, either the agents are not in the same subgraph, or another vertex v'' with degree ≥ 3 is reachable in $m-2$ steps. Moving the agents towards v'' will leave ≥ 2 free vertices 'behind' v'' and ≥ 1 free vertex 'behind' the trailing agent. This means that the first case applies to vertex v''.

3. Having exactly one free vertex on both sides of the agents implies that $m = 2$. This means that each subgraph is either a single vertex, or a biconnected component of the graph, since no two components are joined together in Algorithm 1. In the single vertex case, it is impossible for both agents to belong to the same subgraph, as moving agent s into this single-vertex subgraph will not leave sufficient free vertices reachable as is required in Algorithm 2.

   When the agents belong to the same biconnected component, stage three of the `clear` operation applies: it is possible to push the agents towards one of the free vertices, such that the trailing agent ends up on vertex v. This clears the path for an additional vertex next to v to push towards the other free vertex through the original position of the trailing agent. This clears vertices next to v.

Finally, to see that $\varnothing \neq f_S(r)$, note that a swap requires r and s to be adjacent to a vertex v of degree ≥ 3. Clearly, it is possible for both r and s to step off the vertex v and into one of the unoccupied vertices, which are either in $\mathcal{S}_i$, or are part of other planks of $\mathcal{S}_i$. Hence, r and s are confined to $\mathcal{S}_i$ and its planks by Proposition 1. $\square$

References

Auletta, V., Monti, A., Parente, M., & Persiano, P. (1999). A linear-time algorithm for the feasibility of pebble motion on trees. *Algorithmica*, *23*(3), 223–245.

Beaulieu, M., & Gamache, M. (2006). An enumeration algorithm for solving the fleet management problem in underground mines. *Computers & Operations Research*, *33*(6), 1606–1624.

Călinescu, G., Dumitrescu, A., & Pach, J. (2008). Reconfigurations in graphs and grids. *SIAM Journal on Discrete Mathematics*, *22*(1), 124–138.

Culberson, J. (1999). Sokoban is PSPACE-complete. In *Proceedings in Informatics*, Vol. 4, pp. 65–76. Citeseer.

De Wilde, B. (2012). Cooperative multi-agent path planning. Master's thesis, Delft University of Technology.

De Wilde, B., Ter Mors, A. W., & Witteveen, C. (2013). Push and Rotate: Cooperative multi-agent path planning. In *Proceedings of the twelfth international conference on autonomous agents and multiagent systems*, AAMAS, pp. 87–94, Saint Paul, Minnesota, USA.

Desaulniers, G., Langevin, A., Riopel, D., & Villeneuve, B. (2004). Dispatching and conflict-free routing of automated guided vehicles: An exact approach. *International Journal of Flexible Manufacturing Systems*, *15*(4), 309–331.

Erdmann, M., & Lozano-Pérez, T. (1987). On multiple moving objects. *Algorithmica*, *2*(1), 477–521.

Gawrilow, E., Köhler, E., Möhring, R. H., & Stenzel, B. (2007). *Mathematics - Key Technology for the Future*, chap. Dynamic Routing of Automated Guided Vehicles in Real-time, pp. 165–177. Springer Berlin Heidelberg.

Goldreich, O. (1993). Finding the shortest move-sequence in the graph-generalized 15-puzzle is NP-hard. Tech. rep. 792, Technion – Israel Institute of Technology. The work on this paper was completed in July 1984.

Goraly, G., & Hassin, R. (2010). Multi-color pebble motion on graphs. *Algorithmica*, *58*(3), 610–636.

Hearn, R. A., & Demaine, E. D. (2005). PSPACE-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. *Theoretical Computer Science*, *343*(1–2), 72–96.

Hopcroft, J. E., & Tarjan, R. E. (1973). Algorithm 447: efficient algorithms for graph manipulation. *Communications of the ACM*, *16*(6), 372–378.

Hopcroft, J. E., Schwartz, J. T., & Sharir, M. (1984). On the complexity of motion planning for multiple independent objects; PSPACE-hardness of the "warehouseman's problem". *The International Journal of Robotics Research*, *3*(4), 76–88.

Kavraki, L. E., Svestka, P., Latombe, J.-C., & Overmars, M. H. (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, *12*(4), 566–580.

Khorshid, M. M., Holte, R. C., & Sturtevant, N. (2011). A polynomial-time algorithm for non-optimal multi-agent pathfinding. In *Proceedings of the Fourth International Symposium on Combinatorial Search*, SoCS, pp. 76–83.

Kornhauser, D., Miller, G., & Spirakis, P. (1984). Coordinating pebble motion on graphs, the diameter of permutation groups, and applications. In *Proceedings of the* 25*th* *Annual Symposium on Foundations of Computer Science*, FOCS, pp. 241–250.

Kornhauser, D. M. (1984). Coordinating pebble motion on graphs, the diameter of permutation groups, and applications. Master's thesis, Massachusetts Institute of Technology.

LaValle, S. M., & Kuffner, J. J. (2001). Rapidly-exploring random trees: Progress and prospects. In Donald, B. R., Lynch, K. M., & Rus, D. (Eds.), *Algorithmic and Computational Robotics: New Directions*, pp. 293–308. A K Peters, Wellesley, MA.

Lee, J. H., Lee, B. H., & Choi, M. H. (1998). A real-time traffic control scheme of multiple AGV systems for collision-free minimum time motion: a routing table approach. *IEEE Transactions on Man and Cybernetics, Part A*, *28*(3), 347–358.

Luna, R., & Bekris, K. E. (2011a). Efficient and complete centralized multi-robot path planning. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, IROS, pp. 3268–3275, San Francisco, CA, USA. IEEE.

Luna, R., & Bekris, K. E. (2011b). Push and Swap: fast cooperative path-finding with completeness guarantees. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence – Volume One*, IJCAI, pp. 294–300. AAAI Press.

Narasimhan, R., Batta, R., & Karwan, H. (1999). Routing automated guided vehicles in the presence of interruptions. *International Journal of Production Research*, *37*(3), 653–681.

Nieuwenhuisen, D., Kamphuis, A., & Overmars, M. (2007). High quality navigation in computer games. *Science of Computer Programming*, *67*(1), 91 – 104. Special Issue on Aspects of Game Programming.

Papadimitriou, C., Raghavan, P., Sudan, M., & Tamaki, H. (1994). Motion planning on a graph. In *Proceedings of the* 35*th* *Annual Symposium on Foundations of Computer Science*, FOCS, pp. 511 –520.

Röger, G., & Helmert, M. (2012). Non-optimal multi-agent pathfinding is solved (since 1984). In *Proceedings of the Fifth Annual Symposium on Combinatorial Search*, SoCS, pp. 173–174. AAAI Press.

Roszkowska, E., & Reveliotis, S. A. (2008). On the liveness of guidepath-based, zone-controlled dynamically routed, closed traffic systems. *IEEE Transactions on Automatic Control*, *53*(7), 1689–1695.

Ruml, W., Do, M. B., Zhou, R., & Fromherz, M. P. (2011). On-line planning and scheduling: An application to controlling modular printers. *Journal of Artificial Intelligence Research*, *40*, 415–468.

Sajid, Q., Luna, R., & Bekris, K. E. (2012). Multi-agent pathfinding with simultaneous execution of single-agent primitives. In *Proceedings of the Fifth Annual Symposium on Combinatorial Search*, SoSC, pp. 88–96, Niagara Falls, Canada. AAAI.

Sharon, G., Stern, R., Felner, A., & Sturtevant, N. (2012). Meta-agent conflict-based search for optimal multi-agent path finding. In *Proceedings of the Fifth Annual Symposium on Combinatorial Search*, SoSC, pp. 97–104, Niagara Falls, Canada. AAAI.

Sharon, G., Stern, R., Goldenberg, M., & Felner, A. (2011). The increasing cost tree search for optimal multi-agent pathfinding. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*, IJCAI, pp. 662–667.

Silver, D. (2005). Cooperative pathfinding. In *Proceedings of the 1st Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE, pp. 117–122.

Siméon, T., Leroy, S., & Laumond, J.-P. (2002). Path coordination for multiple mobile robots: a resolution-complete algorithm. *IEEE Transactions on Robots and Automation*, *18*(1), 42–49.

Šišlák, D., Pěchouček, M., Volf, P., Pavlíček, D., Samek, J., Mařík, V., & Losiewicz, P. (2008). *AGENTFLY: Towards Multi-Agent Technology in Free Flight Air Traffic Control*, chap. 7, pp. 73–97. Birkhauser Verlag.

Standley, T. (2010). Finding optimal solutions to cooperative pathfinding problems. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, AAAI, pp. 173–178.

Standley, T., & Korf, R. (2011). Complete algorithms for cooperative pathfinding problems. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence – Volume One*, IJCAI, pp. 668–673. AAAI Press.

Surynek, P. (2009). A novel approach to path planning for multiple robots in bi-connected graphs. In *Proceedings of the 2009 IEEE International Conference on Robotics and Automation*, ICRA, pp. 3613–3619, Kobe, Japan.

Surynek, P. (2011). *Multi-Robot Systems, Trends and Development*, chap. Multi-Robot Path Planning, pp. 267–290. InTech - Open Access Publisher, Vienna, Austria.

Ter Mors, A. W., Witteveen, C., Zutt, J., & Kuipers, F. A. (2010). Context-aware route planning. In *Proceedings of the Eighth German Conference on Multi-Agent System Technologies*, MATES, Leipzig, Germany. Springer.

Ter Mors, A. W., Zutt, J., & Witteveen, C. (2007). Context-aware logistic routing and scheduling. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling*, ICAPS, pp. 328–335.

Trüg, S., Hoffmann, J., & Nebel, B. (2004). Applying automatic planning systems to airport ground-traffic control – a feasibility study. In *27th Annual German Conference on AI, KI*, pp. 183–197.

Velagapudi, P., Sycara, K., & Scerri, P. (2010). Decentralized prioritized planning in large multi-robot teams. In *Proceedings of the 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IROS, pp. 4603–4609, Taipei, Taiwan.

Vis, I. F. A. (2006). Survey of research in the design and control of automated guided vehicle systems. *European Journal of Operational Research*, *170*(3), 677–709.

Wang, K.-H. C., & Botea, A. (2008). Fast and memory-efficient multi-agent pathfinding. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling*, ICAPS, pp. 380–387.

Wang, K.-H. C., & Botea, A. (2011). MAPP: a scalable multi-agent path planning algorithm with tractability and completeness guarantees. *Journal of Artificial Intelligence Research*, *42*, 55–90.

Wilson, R. M. (1974). Graph puzzles, homotopy, and the alternating group. *Journal of Combinatorial Theory, Series B*, *16*(1), 86–96.

Wu, Z., & Grumbach, S. (2009). Feasibility of motion planning on directed graphs. In *Theory and Applications of Models of Computation*, Vol. 5532 of *Lecture Notes in Computer Science*, pp. 430–439. Springer.

Zutt, J., & Witteveen, C. (2004). Multi-agent transport planning. In *Proceedings of the Sixteenth Belgium-Netherlands Artificial Intelligence Conference*, BNAIC, pp. 139–146, Groningen.