Push and Rotate: Cooperative Multi-Agent Path Planning

Boris de Wilde, Adriaan W. ter Mors, and Cees Witteveen
Algorithmics Group, Delft University of Technology, Mekelweg 4, Delft, The Netherlands
boreus@gmail.com, {a.w.termors,c.witteveen}@tudelft.nl

ABSTRACT

In cooperative multi-agent path planning, agents must move between start and destination locations and avoid collisions with each other. Many recent algorithms require some sort of restriction in order to be complete, except for the Push and Swap algorithm [7], which claims only to require two unoccupied locations in a connected graph. Our analysis shows, however, that for certain types of instances Push and Swap may fail to find a solution.

We present the Push and Rotate algorithm, an adaptation of the Push and Swap algorithm, and prove that by fixing the latter's shortcomings, we obtain an algorithm that is complete for the class of instances with two unoccupied locations in a connected graph. In addition, we provide experimental results that show our algorithm to perform competitively on a set of benchmark problems from the video game industry.

Categories and Subject Descriptors

F.2.2 [**Nonnumerical Algorithms and Problems**]: Routing and Layout; I.2.11 [**Distributed Artificial Intelligence**]: Multiagent Systems

General Terms

Algorithms

Keywords

Planning, Distributed problem solving

1. INTRODUCTION

In cooperative multi-agent path planning, there is a set of of agents each with a unique start and destination location, and the goal is to find paths that are collision-free in space and time, while optimizing a global quality measure such as total time, or total number of steps taken. Cooperative multi-agent path planning can be applied in domains like control of automated guided vehicles [13], robotics [9], and video games (cf. [8]).

Finding optimal solutions to multi-agent path planning problems is NP-hard (cf. [2]). Recent advances in optimal solving include the work by Standley and Korf [11] and Yu and LaValle [17]. The former propose *operator decoupling*, a technique that reduces the branching factor of exhaustive search, while the latter present ILP formulations of a related multi-commodity flow problem, which can be used to solve multi-agent pathfinding problems. Both papers report experiments with up to 50 or 60 agents.

To tackle larger instances, *decoupled* approaches plan for each agent separately, for instance by first planning a path for each agent, and then coordinating the velocities such that two agents never occupy the same location simultaneously [4]. The decoupled approach is not complete in general, and in congestion-prone domains deadlocks would occur frequently. In video games, for example, it is common that many agents must traverse maps with choke points like bridges or narrow corridors, where quick passage through the congested area is important.

A number of algorithms have appeared in recent years to tackle these crowded domains, with varying degrees of completeness guarantees. The WHCA* algorithm by Silver [10] is a prioritized approach that takes other agents into account for a finite time horizon. Within this horizon, an agent must plan around the reservations of higher priority agents; beyond the horizon, the plan is continued directly to the goal. At regular intervals, the horizon is shifted forward and a new partial route is computed. It is possible that the algorithm ends up in a situation from which no further progress will be made. In the FAR algorithm [14], agents utilize a flow annotation on the map to plan their route, and try to break any deadlocks that occur using heuristic procedures.

Several other algorithms are complete for restricted classes of instances. The MAPP algorithm [15] works on problem instances that are called SLIDABLE: these are instances in which for every three consecutive vertices in an agent's path to its destination, an alternative path exists that does not make use of the 'middle' vertex. In addition, SLIDABLE specifies minimal interference conditions between the start and goal locations of the agents, and the number of unoccupied vertices must be at least two. In [5], the algorithm TASS (tree-based agent swapping strategy) is presented that is complete for trees (and often works well on instances that are close to trees, but loses its completeness in a tree-decomposition step), whereas the BIBOX algorithm [12] is complete for biconnected graphs (and two unoccupied locations).

The Push and Swap algorithm [7] is presented as complete for a connected graph with at least two unoccupied locations. However, there exist several classes of instances for which Push and Swap may[1] not find a solution, even if one exists. Our contribution in this paper is to analyze these shortcomings of Push and Swap and to prove that, by fixing these shortcomings, we obtain a complete algorithm, which we call Push and Rotate.

This paper is organized as follows. In Section 2, we provide some background on the problem and its definition, which finds

[1]For some problem instances, whether or not Push and Swap finds a solution depends on the order in which the agents are planned.

Appears in: *Proceedings of the 12th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2013), Ito, Jonker, Gini, and Shehory (eds.), May, 6–10, 2013, Saint Paul, Minnesota, USA.*
Copyright © 2013, International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). All rights reserved.

its origins in the *pebble motion* literature. In Section 3, we describe the Push and Swap algorithm, and demonstrate for which classes of instances it may fail. Section 4 presents our Push and Rotate algorithm, and we prove that our solutions to the shortcomings of Push and Swap result in a complete algorithm. In Section 5 we present a brief experimental validation of our algorithm, and in Section 6 we conclude with some ideas for future work. Finally, the proofs of the completeness and correctness of the algorithms are in the appendix.

2. BACKGROUND

The problem we consider in this paper was previously defined as coordinating pebble motion on graphs by Kornhauser [6]: let G be a graph with n vertices with $k < n$ pebbles numbered $1,\ldots,k$ on distinct vertices. A move consists of transferring a pebble to an adjacent unoccupied vertex. Goldreich proved that finding the shortest sequence of moves to reach one arrangement of pebbles from another is NP-hard [2].

Kornhauser showed that determining reachability can be done in polynomial time, and requires at most $O(n^3)$ moves. This result is based on the fact that a graph may be viewed as a tree of biconnected (non-separable) components that are linked by chains of vertices with degree 2 (called *isthmuses*). If two of these components are linked by a chain that contains more vertices than the number of unoccupied vertices minus two, then it is impossible for an agent to cross this isthmus; see Section 3.2 for an example.

In this paper, we use the term agents rather than pebbles, and we use the following notation: We consider a connected graph $\mathcal{G} = (V,E)$, a set of agents $\mathcal{R}$, an initial assignment of agents to vertices $\mathcal{S} : \mathcal{R} \to V$, and a goal assignment of agents to vertices $\mathcal{T} : \mathcal{R} \to V$. A sequence of assignments of length L, $\Pi = [\Pi_1, \Pi_2, \cdots, \Pi_L]$, is a solution of the instance when $\Pi_1 = \mathcal{S}$, $\Pi_L = \mathcal{T}$, and each assignment Π_{k+1} differs from the previous assignment Π_k such that exactly one agent moves to an adjacent and unoccupied vertex for each $k \geq 1$.

3. PUSH AND SWAP

In this section, we first explain the ideas behind the Push and Swap algorithm, and then show in which cases it may fail to find a solution. The Push and Swap algorithm works by iteratively selecting agents in some unspecified *priority* order to move to their respective destination locations. For one agent, the algorithm moves 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 attempts to move it out of the way with the `push` operation. This can be accomplished by pushing the blocking agent forward along the shortest path (not containing any higher-priority agents) to an empty vertex.

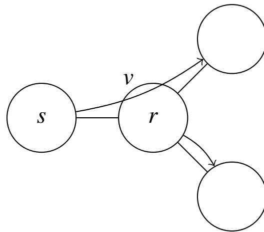


Figure 1: Vertex *v* with degree 3 can be used to swap *r* and *s*.

In case the blocking agent has already been planned for (i.e., it has higher priority and is already occupying its goal location), the algorithm attempts to exchange their positions using the `swap` operation[2]. A swap can be accomplished by moving both agents to a part of the graph where there is enough room for them to exchange positions. Figure 1 shows a location v that meets the requirements for a swap: it has degree three, and it is adjacent to two unoccupied locations. All vertices with a degree of at least three are eligible for a swap in the Push and Swap algorithm, and they are tried in order of proximity. To effect the swap, first the two swapping agents are moved to the vertex with sufficient degree using a variant of the `push` operation. Next, the `clear` operation is invoked that tries to clear two neighbouring locations of agents. If successful, the agents exchange position, and then all the moves made to enable this exchange are reversed ensuring that only the positions of the two swapping agents are affected.

After the `swap` operation has successfully completed, one agent r may be moved off its goal position. To move it back, the `resolve` operation is used. It is first attempted to `push` the agent s currently occupying its goal position (with which r has just swapped) towards the goal of s. If this is successful, the goal position of r is free and it can move there. Otherwise, the `swap` operation is invoked to move s towards its destination. If s thus reaches its destination without freeing the goal location of r, a recursive call to `resolve` is made to bring an agent s', that is now occupying r's goal, back to its goal.

In the next sections 3.1 to 3.4, we demonstrate the shortcomings of the Push and Swap algorithm that render it incomplete, i.e., we show that there exist instances for which Push and Swap fails to find a solution, even if one exists.

3.1 Polygons

In the pebble motion literature, a *polygon* is defined as an instance in which all vertices have degree 2. In Figure 2, the agents r and s, in their initial locations, are occupying each other's goal locations. First note that this instance can be solved by moving both agents either clockwise or counter-clockwise, one step for one agent and four steps for the other agent. Without loss of generality, suppose that agent r has the highest priority. Agent r will push agent s out of the way and reach its destination location. When the algorithm next plans for agent s, the shortest path to its goal location is blocked by agent r. Since the `push` operation may not move r, due to its higher priority, a `swap` between the agents is attempted. However, no node with degree three or more exists in this graph, and Push and Swap fails to solve the instance.

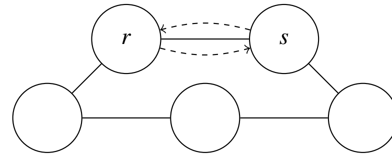


Figure 2: Polygon instance (cycle) in which no swap operation is possible.

3.2 Isthmuses

In [16, 6], it is shown that for a biconnected graph[3] with two unoccupied vertices, any arrangement of pebbles can be reached from any other. In case two biconnected components are joined by

[2] The `swap` operation can also be invoked in case a lower-priority agent cannot be pushed out of the way, for instance if the planning agent has a goal location at the end of a cul-de-sac.

[3] A graph is biconnected if the graph remains connected after removal of any vertex.

a chain of vertices of degree two, called a bridge or isthmus, then reachability depends on the length of the bridge, and the number of unoccupied vertices in the graph.

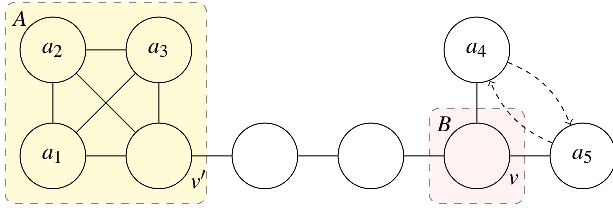


Figure 3: An isthmus connects two subproblems[5] A and B; for agents in subproblem A, no vertices beyond vertex v in subproblem B can be reached.

The Push and Swap algorithm does not take the notion of isthmuses into account, and can fail accordingly. In Figure 3, suppose that a_1's destination is location v, and a_4 and a_5 are currently occupying each other's goal locations. To exchange position, they need to use vertex v. If a_1 is planned first, however, then it will occupy v and cannot be moved away. First, a_4 (or a_5) will try a `push`, but it is not allowed to push an agent with higher priority. Second, the `swap` operation will fail, because a swap is impossible between a_1 and a_4, as that requires both agents to be at a vertex with degree three or more, with two neighbours unoccupied. The only two vertices of degree three or more that can be reached by both a_1 at a_4 are v and v'. With a_1 at v, there is only one empty neighbour to the left of v; if a_1 moves back to its start location and a_4 moves to v', then there is only one empty vertex to the right of v'. Hence, Push and Swap fails if a_1 is planned before a_4 and a_5.

3.3 Incomplete Clear

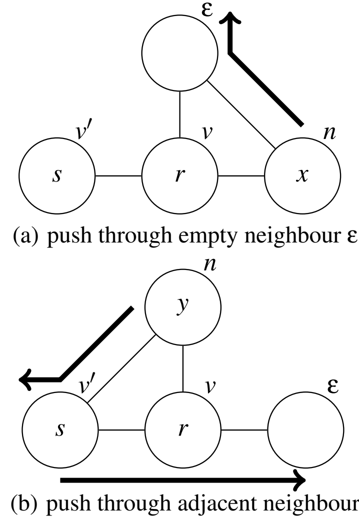
To execute a `swap`, two agents must be brought to a vertex of degree three or higher, with two empty neighbours. The `clear` operation attempts to clear two neighbouring vertices, basically by distinguishing the various cases that the operation might encounter, and dealing with them case by case. In the original description in [7], however, the cases listed in Figure 4 have not been considered. When executing the `clear` operation, it first looks if any neighbours are already unoccupied. If there are two or more unoccupied neighbouring vertices, the `clear` operation does not need to clear additional vertices and it returns true. In the case that there is one unoccupied neighbour, this vertex is considered an obstacle throughout the execution of the rest of the operation (vertex ε in Figure 4). In Figure 4(a), a solution would be to move agent x to ε, and then to push x away to clear ε again, but the `clear` operation does not consider this option.

With regard to Figure 4(b), Luna and Bekris [7] state that it is unnecessary to consider the case of clearing node n by pushing it along the path (v, v'), but they did not consider the possibility of moving y to v' along the edge (n, v'). To clear vertex n in Figure 4(b), agent r should move to ε, and agent s into v.

3.4 Recursive Resolve

When a planning agent s encounters a higher-priority agent r that is already at its goal location, a `swap` operation is used to exchange the position of the agents. As a result, agent r is moved off its goal location, and to move back there the `resolve` operation is invoked.

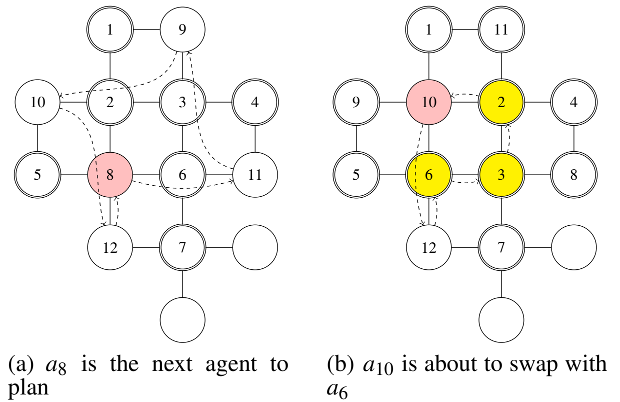
[5]In Algorithm 1, Section 4.1, we show how to construct *subproblems*, which are parts of the graph within which agents assigned to the subproblem can exchange positions.



(a) push through empty neighbour ε

(b) push through adjacent neighbour

Figure 4: Cases that the original `clear` does not solve.



(a) a_8 is the next agent to plan

(b) a_{10} is about to swap with a_6

Figure 5: Instance in which recursive calls to `resolve` result in an invalid state.

A problem arises when a `swap` operation is invoked on an agent that is already resolving, as is the case in Figure 5. In Figure 5(a), the agents that are already at their destination are indicated with a double circle, while a dotted arc indicates the destination of an agent not at its destination. The next agent to plan is a_8, at the pink vertex. Push and Swap starts by swapping a_8 with a_6, and then with a_{11}, leaving a_8 at its destination, a_6 at the start location of a_8, and a_{11}, which the `resolve` operation will now send forward to its goal, occupying the goal location of a_6. After a number of steps, the following situation will be reached (Figure 5(b)): there are three resolving agents a_2, a_3, a_6 (yellow vertices), and the planning agent is a_{10}. Now agent a_{10} will swap with agent a_6, and since $\mathcal{T}[a_6] \in \mathcal{U}$, `resolve` will be invoked with $s = a_6$ and $r = a_{10}$, while agent a_6 is two steps away from its destination. Lines 4 ("move s from $\mathcal{A}[s]$ to $\mathcal{T}[s]$") and 7 ("$p^s \leftarrow \{\mathcal{A}[s], \mathcal{T}[s]\}$") of the `resolve` algorithm [7] clearly require agent s to be on a vertex that is adjacent to its destination vertex, hence an illegal state has been reached.

4. PUSH AND ROTATE

In this section, we present Push and Rotate, and show how it solves Push and Swap's problems listed in the previous section. The case of *polygon graphs* are treated in the main algorithm (al-

gorithm 4 in Section 4.2) by selecting, for the next planning agent r, a shortest path to its goal that does not encounter any finished agents. The problems with *recursive resolve* are dealt with by the `rotate` operation in Lemma 4.3, in case there is a cycle among resolving agents. The extended `clear` operation is similar to the one described in Luna and Bekris [7], extended with the cases described in Section 3.3, and here we do not discuss it in further detail.

First, in Section 4.1, we show how to decompose the problem into subproblems, such that agents within a subproblem can exchange positions, but agents between subproblems cannot, and that by prioritizing subproblems we can ensure that our algorithm finds a solution if one exists. This solves the *isthmus* problem.

4.1 Problem decomposition

The problem decomposition we discuss in this section consists of three stages. In the first stage we identify disjoint parts of the graph that we call *subproblems*, the second stage is to assign agents to the subproblems, and the third stage is to define a priority ordering between the subproblems, such that agents assigned to higher-priority subproblems must move first in order for a solution to be found.

Due to [16], we know that for a biconnected graph with two unoccupied vertices, any arrangement of agents can be reached from any other. In case two biconnected components are connected by an isthmus, it depends on the length of the isthmus and on the number of unoccupied vertices, whether agents from one component can reach all locations in the other. In algorithm 1, we are interested in finding subproblems of the graph, such that agents assigned to the same subproblem can exchange locations (using the `swap` operation). In algorithm 1, let $m = |V| - |\mathcal{R}|$ denote the number of unoccupied vertices.

Algorithm 1 Division into subproblems

1: $\mathcal{C} \leftarrow$ all nontrivial biconnected components in $\mathcal{G}$
2: $\mathcal{C} \leftarrow \mathcal{C} \cup \{v \in V \mid \texttt{degree}(v) \geq 3 \wedge v \notin \mathcal{C}\}$
3: **while** $\exists \mathcal{C}_i, \mathcal{C}_j \in \mathcal{C} \mid (\min_{v \in \mathcal{C}_i, u \in \mathcal{C}_j} d(v,u)) \leq m-2$ **do**
4: $\quad \mathcal{C}_k = \mathcal{C}_i \cup \mathcal{C}_j \cup \{v' \in \texttt{shortest_path}(u,v)\}$
5: $\quad \mathcal{C} \leftarrow (\mathcal{C} \setminus \{\mathcal{C}_i, \mathcal{C}_j\}) \cup \{\mathcal{C}_k\}$

In line 1 of algorithm 1, we first find all non-trivial (i.e., of size at least three) biconnected components, which can be done in time $O(|V| + |E|)$ [3]. Next, all remaining vertices of degree three or higher are added as components of size 1. In while loop starting on line 3, all pairs of components that have distance less than or equal to $m-2$ (in line 3, where $d(v,u)$ denotes the length of the shortest path between v and u) are then joined into one subproblem[6]. The value of $m-2$ was found by Kornhauser [6] to be the maximum distance between biconnected components such that agents in one can still swap with agents in another (and later rediscovered by Khorshid to hold for trees — [5], Tree Solvability Condition 3).

Algorithm 2 assigns agents to subproblems. The idea is that an agent is part of a subproblem if it can reach the subproblem *and* there are enough empty vertices near the subproblem to be able to swap with other agents from the subproblem. The algorithm iterates over all vertices in all subproblems, and decides whether an agent occupying a vertex should be assigned to the subproblem. In case an agent is on a vertex v inside a subproblem $\mathcal{C}_i$ that is not connected to any vertex outside the subproblem, then this agent is assigned to $\mathcal{C}_i$ (line 10). In case v is connected to a $u \notin \mathcal{C}_i$, then in line 4, the value of m' indicates the number of unoccupied vertices

Algorithm 2 Assigning agents to subproblems

1: **for all** $\mathcal{C}_i \in \mathcal{C}$ **do**
2: $\quad$ **for all** $v \in \mathcal{C}_i$ **do**
3: $\quad\quad$ **for all** $u \notin \mathcal{C}_i$ for which $(u,v) \in \mathcal{G}$ **do**
4: $\quad\quad\quad m' \leftarrow$ number of unoccupied vertices reachable from v in $\mathcal{G} \backslash \{u\}$
5: $\quad\quad\quad m'' \leftarrow$ number of unoccupied vertices reachable from $\mathcal{C}_i$ in $\mathcal{G} \backslash \{v\}$
6: $\quad\quad\quad$ **if** $(m' \geq 1 \wedge m' < m) \vee m'' \geq 1$ **then**
7: $\quad\quad\quad\quad$ Assign agent on position v to $\mathcal{C}_i$ (if any)
8: $\quad\quad\quad$ Follow path from u away from v and assign the first $m'-1$ agents on this path to $\mathcal{C}_i$
9: $\quad\quad$ **if** $\{u \notin \mathcal{C}_i$ for which $(u,v) \in \mathcal{G}\} = \emptyset$ **then**
10: $\quad\quad\quad$ Assign agent on position v to $\mathcal{C}_i$ (if any)
11: **return** Assignment of agents to $\mathcal{C}$

that can be reached if the edge (u,v) is not followed. If $m' \geq 1$, the agent on v can be assigned to $\mathcal{C}_i$ (line 7), unless all unoccupied vertices are outside the subproblem ($m' = m$, line 6); in Figure 6, for instance, if v is the vertex containing a_8, and u is either of the vertices holding a_9 and a_{10}, then all unoccupied vertices are left of the subproblem ($m' = m$), and a_8 is not assigned to subproblem C.

In addition to agents inside the subproblem, the first $m'-1$ agents encountered on a path away from the subproblem are also added to the subproblem, since these agents are able to enter the subproblem while leaving one unoccupied vertex available for movement (line 8). For the vertices v and u in Figure 6, m' equals 3, so both agents a_7 and a_8 are assigned to the subproblem B. Note that all agents $\{a_5, \ldots, a_8\}$ are assigned to subproblem B, and can exchange positions within, but not all can occupy a vertex in the subproblem simultaneously.

In case $m' = m$, an agent occupying vertex v is only assigned to $\mathcal{C}_i$ if it is possible to move an unoccupied vertex into the subproblem without moving the agent out of the subproblem. Hence, if $m'' \geq 1$ (at least one vertex is reachable from component $\mathcal{C}_i$, if we do not use the agent's current vertex v), such agents are also assigned to the subproblem (lines 5, 7).

Algorithm 3 Priority relation between subproblems

1: **for all** $\mathcal{C}_i \in \mathcal{C}$ **do**
2: $\quad$ **for all** $v \in \mathcal{C}_i$ **do**
3: $\quad\quad$ **for all** $u \notin \mathcal{C}_i$ for which $(u,v) \in \mathcal{G}$ **do**
4: $\quad\quad\quad$ Vertex u should be the first vertex on the path from $\mathcal{C}_i$ to another subproblem $\mathcal{C}_j$, otherwise continue with the next u
5: $\quad\quad\quad v' \leftarrow v$
6: $\quad\quad\quad$ **while** $r \leftarrow \mathcal{T}^{-1}[v'] \mid r$ does not belong to $\mathcal{C}_i$ **do**
7: $\quad\quad\quad\quad$ **if** Agent r belongs to $\mathcal{C}_j$ **then**
8: $\quad\quad\quad\quad\quad \mathcal{C}_i \prec \mathcal{C}_j$
9: $\quad\quad\quad\quad\quad$ Continue with next u (line 3)
10: $\quad\quad\quad\quad v' \leftarrow$ next vertex on path from $\mathcal{C}_i$ to $\mathcal{C}_j$
11: **return** The priority relation "$\prec$"

The third stage of the decomposition process, listed in algorithm 3, is to assign priorities to agents based on their membership to subproblems (agents assigned to the same subproblem receive the same priority). Given two subproblems $\mathcal{C}_i$ and $\mathcal{C}_j$, and an agent r assigned to $\mathcal{C}_j$ with goal position $\mathcal{T}[r]$, the priority relation $\mathcal{C}_i \prec \mathcal{C}_j$ is added in either of the following two cases:

1. $\mathcal{T}[r]$ is on the edge of subproblem $\mathcal{C}_i$.
2. Agent r at $\mathcal{T}[r]$ locks an unassigned agent on the edge of $\mathcal{C}_i$.

[6]Note that by *component*, we mean a biconnected component in the graph, and by *subproblem* a construct consisting of one or more components, to which we assign the agents.

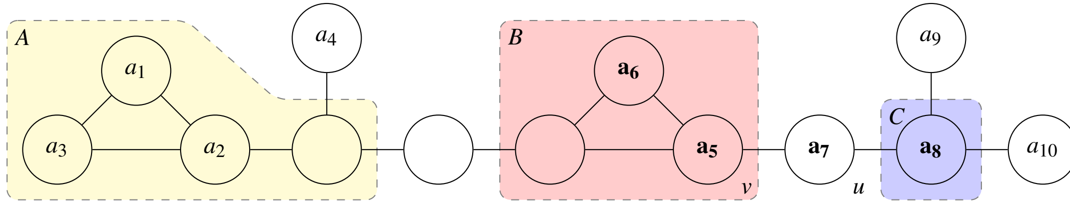


Figure 6: Decomposition of a problem: the agents assigned to subproblem B are a_5, a_6, a_7 and a_8.

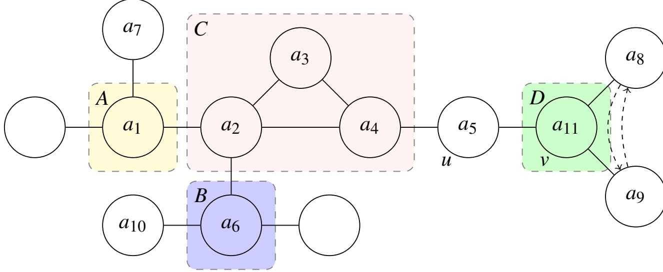


Figure 7: If agent a_5 is assigned a higher priority than agents a_8 and a_9, then a `swap` between the latter two is impossible.

As an example of the second case, consider the instance of Figure 7, in which for all agents their start location equals their destination location, except for agents a_8 and a_9, which want to exchange positions. Note that agent a_{11} is not assigned to any subproblem by Algorithm 2. If agent a_5 has high priority, then it will lock agent a_{11} and agents a_8 and a_9 in their positions. Algorithm 3, when considering (v,u) as named in the figure, will first find, on line 6, that agent a_{11} does not belong to D on, and in the next iteration, on line 7, find that agent a_5 belongs to C. Hence the relation $D \prec C$ is added on line 8. Note that the unassigned agents (in the example only a_{11}) always have lowest priority.

4.2 The `solve` Operation

The Push and Rotate algorithm continues by calling operation `solve` (algorithm 4). Like Push and Swap, it plans the agents one by one. The while loop In line 6 iterates until the set $\mathcal{F}$ of agents that have been planned for equals the set of agents $\mathcal{R}$. In line 8, we randomly select a non-finished agent with (equal) highest priority, and determine an arbitrary shortest path in line 12 (or a shortest path avoiding finished agents, in case the graph is a polygon (line 10)). The while loop from line 13 to line 21 moves agent r forward one step at a time. The basic procedure is to try a `push`, in line 18, and if that fails to try a `swap`, in line 19. If `swap` fails, then the instance has no solution (Theorem 1), and the algorithm returns false.

In line 21, the latest vertex v is added to the path q, which records the path traversed by agent r. Before we explain lines 15 and 16, we first show how q is processed, in lines 24 to 32. In line 27, it is checked whether in the trail of agent r, there exists an agent $s \in \mathcal{F}$ that has been moved off its goal location. In case it is not possible to `move` s to its goal location, then we assign to r the agent occupying its goal location, and return to the loop from line 6. If it is detected, in line 15, that q forms a cycle, then all agents on that cycle are rotated one step, using the `rotate` operation (see Figure 8 and Lemma 4.3).

4.3 Correctness and Completeness

We now present the main results for the correctness and the completeness of Push and Rotate; the proofs are in the appendix.

Algorithm 4 `solve`$(\mathcal{G},\mathcal{R},\mathcal{S},\mathcal{T})$

```
1: Π ← [ ], q ← [ ]
2: A ← S
3: F ← ∅
4: r ← empty
5: c ← ∀v ∈ V : degree(v) = 2
6: while F ≠ R do
7:    if r = empty then
8:       r ← next agent in R \ F
9:    if c then
10:      p ← shortest_path(G, A[r], T[r], A[F])
11:   else
12:      p ← shortest_path(G, A[r], T[r], ∅)
13:   while A[r] ≠ T[r] do
14:      v ← vertex after A[r] on p
15:      if v ∈ q then
16:         rotate(Π, G, A, q, v)
17:      else
18:         if push(Π, G, A, r, v, A[F]) = false then
19:            if swap(Π, G, A, r, A⁻¹[v]) = false then
20:               return false
21:      q ← append(q, v)
22:   F ← F ∪ {r}
23:   r ← empty
24:   while |q| > 0 do
25:      v ← the last vertex on q
26:      s ← A⁻¹(v)
27:      if s ∈ F ∧ A[s] ≠ T[s] then
28:         if r ← A⁻¹(T[s]) = empty then
29:            move(Π, A, agent s to vertex T[s])
30:         else
31:            Break inner loop, continue outer loop
32:      remove v from q
33: return Π
```

THEOREM 1. *Push and Rotate is complete for the class of multi-agent path planning problems in which there are two or more unoccupied vertices in each connected component.*

Push and Rotate works by calling algorithms 1 to 4 in order. The idea behind the proof is as follows:

1. In each iteration of `solve`, 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 on a path q.
3. After a finite number of iterations, all vertices on q have been processed, restoring out-of-position agents in $\mathcal{F}$ to their goal location.

To prove and state the correctness and completeness results, we use the following function and predicates:

$\|\mathcal{R}\|$	100	500	1000	1500	2000
P & R	9151.3	45494.9	90194.4	137208.3	184471.9
MAPP	9209.7	54458.9	154304.2	310645.4	497610.6

Table 1: MAPP and P & R on map 411.

$\mathtt{c}(r)$ the subproblem to which agent r is assigned.

$\mathtt{blocked}(r,s)$ agent s occupies the next location on the path of agent r.

$\mathtt{push}(r,s)$ if $\mathtt{blocked}(r,s)$, then the `push` operation can push agent s away from the next vertex on the path of r.

$\mathtt{swap}(r,s)$ if r and s are adjacent (i.e. $(\mathcal{A}[r], \mathcal{A}[s]) \in \mathcal{G}$), then there is a sequence of moves for which only agents r and s swap positions.

The next two lemmas show that either the `push` or the `swap` operation, when called from Algorithm 4 on a solvable instance, will succeed in moving an agent a step closer to its goal.

LEMMA 4.1. $\neg\mathtt{push}(r,s) \rightarrow \mathtt{c}(r) = \mathtt{c}(s)$

LEMMA 4.2. $\mathtt{c}(r) = \mathtt{c}(s) \iff \mathtt{swap}(r,s)$

It follows immediately from Lemma 4.2 that an agent not assigned to a subproblem cannot swap with an agent assigned to some subproblem. Corollary 4.1 shows that also in case two agents are not assigned to any subproblem, no swap is possible.

COROLLARY 4.1. *An agent that is not assigned to any subproblem cannot swap with any agent.*

In the proofs of Lemma 4.1 and Lemma 4.2, we make use of the fact that the assignment of an agent to a subproblem does not change as the agent moves around the graph:

THEOREM 2. *If assignment $\mathcal{A}'$ can be reached from assignment $\mathcal{A}$, the assignment of agents to subproblems is equal for $\mathcal{A}$ and $\mathcal{A}'$.*

Finally, we demonstrate the correctness of the `rotate` operation, as illustrated in Figure 8. Vertex v' is first cleared to allow r to step out of the cycle. The empty vertex created in the cycle allows agents to rotate, but first agent r' moves into v', and r and r' swap positions. Then each agent in the cycle will move one step forward, and finally r' steps back into the cycle at v'.

LEMMA 4.3. *The* `rotate` *operation, when invoked by* `solve`*, moves all agents in a cycle forward by one step.*

5. EMPIRICAL VALIDATION

We tested our algorithm on a set of benchmark problems from the video game industry[7], and compared the results with those of the MAPP algorithm. Table 1 shows a sample of this comparison, and clearly Push and Rotate produces significantly fewer moves than MAPP as the congestion on the map increases.

Figure 9 shows map 411, in which locations and agents are exactly one pixel wide, so although there are some narrow passages, there are no isthmuses. For this map, there are benchmark instances with up to 2000 agents, while the total number of locations is 14098.

[7]The maps can be downloaded from `movingai.com/benchmarks`. To compare with the exact same set of start and destination locations, we downloaded the scenarios provided by Wang and Botea at `http://users.rsise.anu.edu.au/~cwang/scenarios.zip`.

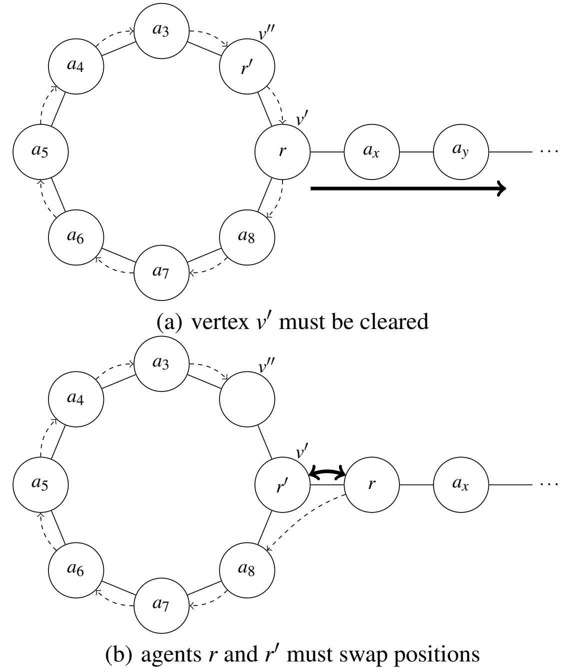


Figure 8: Illustration of the `rotate` operation.

Figure 10 in [15] also shows the performance of the incomplete WHCA* [10] and FAR [14] algorithms; the latter produces much better plans than MAPP, though still worse than Push and Rotate. To allow the reader to compare our approach to the aforementioned, Figure 10 shows the number of moves and CPU times produced by Push and Rotate on map 411. The computation times of MAPP, FAR, and WHCA* are shown in Figure 12 of [15]. For 1000 agents, FAR, MAPP and WHCA* require, respectively, 1, 10, and 60 seconds (P&R 4s.); for 1600 agents, the algorithms require 5, 80, and 320 seconds (P&R 7s.)[8].

6. CONCLUSIONS AND FUTURE WORK

In this paper we presented Push and Rotate, a complete algorithm for the cooperative multi-agent path planning problem, by overcoming the shortcomings we identified in Luna and Bekris's Push and Swap [7]. For the complete specifications of all operations `push`, `swap`, `rotate`, and `clear`, we refer the reader to [1].

Our empirical validation showed that Push and Rotate finds shorter plans than MAPP, while being applicable to a larger class of instances. Push and Rotate also produced better plans than the incomplete FAR and WHCA* algorithms, and its running times are competitive with FAR, the fastest of the algorithms tested in [15].

An area for future work is to try to improve both the speed of the algorithm and the quality of the produced solutions through the use of heuristics. In [1], preliminary investigations showed that choosing shortest paths that avoid finished agents could improve both run time and solution quality. Heuristically determining the relative priorities of agents within subproblems is another possibility.

7. ACKNOWELDEMENTS

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

[8]MAPP, FAR, and WHCA* were run on a 2.8 GHz Intel Core 2 Duo iMac with 2GB of RAM; Push and Rotate was programmed in Java and ran on an Intel i7 870 at 2.93 GHz with 8GB of RAM.

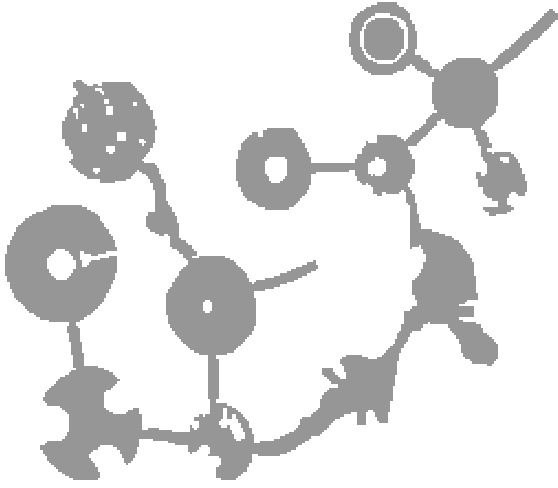


Figure 9: Map 411 of Baldur's Gate II.

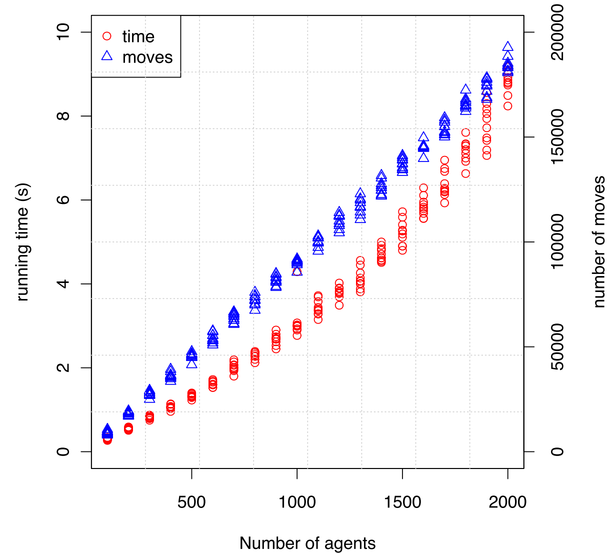


Figure 10: Number of moves and CPU time for map 411.

8. REFERENCES

[1] Boris de Wilde. Cooperative multi-agent path planning. Master's thesis, Delft University of Technology, The Netherlands, August 2012.

[2] Oded Goldreich. Finding the shortest move-sequence in the graph-generalized 15-puzzle is np-hard. In Oded Goldreich, editor, *Studies in Complexity and Cryptography. Miscellanea on the Interplay between Randomness and Computation*, volume 6650 of *Lecture Notes in Computer Science*, pages 1–5. Springer Berlin / Heidelberg, 2011. Original version published in 1984.

[3] J. E. Hopcroft and R. E. Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Commun. ACM*, 16(6):372–378, June 1973.

[4] K. Kant and S. W. Zucker. Toward efficient trajectory planning: The path-velocity decomposition. *The International Journal of Robotics Research*, 5(3):72–89, Fall 1986.

[5] M. M. Khorshid, R. C. Holte, and N. R. Sturtevant. A polynomial-time algorithm for non-optimal multi-agent pathfinding. In *Symposium on Combinatorial Search (SoCS-2011), AAAI Fourth Annual*, pages 76–83, 2011.

[6] D. Kornhauser, G. Miller, and P. Spirakis. Coordinating pebble motion on graphs, the diameter of permutation groups, and applications. In 25th *Annual Symposium on Foundations of Computer Science*, pages 241–250, 1984.

[7] R. Luna and K. E. Bekris. Efficient and complete centralized multi-robot path planning. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 3268 –3275, sept. 2011.

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

[9] M. R. K. Ryan. Exploiting subgraph structure in multi-robot path planning. *Journal of Artificial Intelligence Research*, 31(1), 2008.

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

[11] Trevor Standley and Richard Korf. Complete algorithms for cooperative pathfinding problems. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence - Volume Volume One*, IJCAI'11, pages 668–673. AAAI Press, 2011.

[12] P. Surynek. A novel approach to path planning for multiple robots in bi-connected graphs. In *Robotics and Automation, 2009. ICRA '09. IEEE International Conference on*, pages 3613–3619, May 2009.

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

[14] K.-H. C. Wang and A. Botea. Fast and memory-efficient multi-agent pathfinding. In *Internationl Conference on Automated Planning and Scheduling ICAPS*, pages 380–387, 2008.

[15] K.-H. C. Wang and A. Botea. MAPP: a Scalable Multi-Agent Path Planning Algorithm with Tractability and Completeness Guarantees. *Journal of Artificial Intelligence Research JAIR*, 42:55–90, 2011.

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

[17] Jingjin Yu and Steven M. LaValle. Planning optimal paths for multi-agent systems on graphs. *CoRR*, abs/1204.3830, 2012.

APPENDIX

PROOF OF THEOREM 1. Let r be the next agent in the while-loop of line 6. First a shortest path is determined to its goal. If the graph is a polygon ($c =$ true, line 5), then a shortest path is found in the graph $\mathcal{G} \setminus \mathcal{A}[\mathcal{F}]$. In each iteration of the while-loop in line 13, 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 21). The `rotate` operation will move any agents on C one step for-

ward. Lemma 4.3 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 $\mathtt{push}(r,s)$, then agent s will be pushed out of the way, and agent r moves to v.
- Otherwise the `swap` operation will be executed. $\neg\mathtt{push}(r,s) \rightarrow \mathtt{c}(r) = \mathtt{c}(s)$ (Lemma 4.1), and $\mathtt{c}(r) = \mathtt{c}(s) \leftrightarrow \mathtt{swap}(r,s)$ (Lemma 4.2), hence agents r and s will be swapped successfully.

If an agent s has been moved off its goal position $\mathcal{T}[s]$, then in line 28 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` (note that s is at most one location removed from its goal; as soon as a vertex on q is encountered a second time, a `rotate` is performed). Otherwise, we break the loop that iterates over q, and start a new iteration of the loop from line 6, for the agent r on $\mathcal{T}[s]$, extending q from that location.

Once all agents in the current subproblem are in $\mathcal{F}$, there will be no more agents r occupying $\mathcal{T}[s]$ in line 28 (since they are occupying $\mathcal{T}[r]$), and all out-of-position agents in $\mathcal{F}$ can be moved back to their goal locations. □

PROOF OF LEMMA 4.1. We will prove the equivalent

$$\mathtt{c}(r) \neq \mathtt{c}(s) \rightarrow \mathtt{push}(r,s)$$

If r is not assigned to any subproblem, then it cannot swap with positions with any other agent (Corollary 4.1), so a `push` succeeds if the instance has a solution.

Otherwise, r is assigned to some component $\mathtt{c}(r)$. Since $\mathtt{c}(r) \neq \mathtt{c}(s)$, the blocking agent s can, at most, reach a vertex on the edge of $\mathtt{c}(r)$. Suppose that s is on the edge of $\mathtt{c}(r)$; then, due to Algorithm 3, $\mathtt{c}(r) \prec \mathtt{c}(s)$. Hence, it is not possible that s, or any agents in subproblems 'behind it', are in $\mathcal{F}$. Therefore, `push` succeeds in case the instance is solvable.

In case s is not on the edge of $\mathtt{c}(r)$, then the goal location of r must be on an isthmus between $\mathtt{c}(r)$ and $\mathtt{c}(s)$. Again, the agents cannot swap (Lemma 4.2), so only a `push` can succeed. In its goal position, agent r is also assigned to $\mathtt{c}(r)$ (Theorem 2), and in Algorithm 2, line 8, we can see that there are sufficient empty vertices to reach its goal position. Finally, note that $s \notin \mathcal{F}$; if s is at its goal position, then $\mathtt{c}(r) \prec \mathtt{c}(s)$ due to lines 6–8. □

PROOF OF LEMMA 4.2. First we show $\mathtt{swap}(r,s) \rightarrow \mathtt{c}(r) = \mathtt{c}(s)$. After a successful swap, the only change in the assignment are the positions of agents r and s. Now consider the assignments $\mathcal{A}$ and $\mathcal{A}'$, which have the positions of agents r and s swapped:

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

By Theorem 2, the assignment to subproblems according to $\mathcal{A}$ is equal to the assignment to subproblems according to $\mathcal{A}'$, since there is a sequence of moves (the swap) leading from $\mathcal{A}$ to $\mathcal{A}'$.

For the proof of $\mathtt{c}(r) = \mathtt{c}(s) \rightarrow \mathtt{swap}(r,s)$, we refer the reader to [1]. Here, we briefly sketch the idea behind three possible cases:

1. Both agents are inside a biconnected component. From each vertex in the biconnected component, there are two paths to each of the empty vertices in $\mathcal{G}$ that are vertex disjunct within the biconnected component; if r and s move to such a vertex to `swap`, they can block at most one path, so two neighbours of the vertex can be cleared.
2. Both agents are between two biconnected components, on an isthmus of length $\leq m-2$ (see algorithm 1, line 3). This maximum isthmus length guarantees that at least one of the vertices of degree 3 or higher at the endpoints of the isthmus can be reached by the agents, while leaving sufficient unoccupied vertices available to swap there.
3. Both agents are on an isthmus connected to the subproblem. The assignment criteria for agents to subproblems (at most $m'-1$ agents on the isthmus are assigned, Algorithm 2 line 8) ensure that it is always possible for any agent to enter the subproblem while an unoccupied vertex remains in the subproblem, which allows the swap. □

PROOF OF COROLLARY 4.1. From Algorithm 1, it is clear that all vertices of degree 3 or higher are part of a subproblem. From line 8 in Algorithm 2, it follows that an agent that is not assigned to a subproblem cannot reach a vertex assigned to the subproblem such that it still contains an empty vertex. Hence, it cannot reach a vertex v of degree 3 with two of v's neighbours unoccupied. □

PROOF OF THEOREM 2. We show that a single move of agent r from vertex v_x to vertex v_y (or the other way around), does not change the assignment of agent r. From Algorithm 2, we can infer that the assignment of an agent to a subproblem depends on the reachability of empty vertices (specifically, on m' and m'' in lines 4 and 5) from the subproblem (and on the total number of empty vertices m). The idea behind this proof is that this reachability is not affected by any single move, since an agent move leaves behind an empty vertex.

In particular, we show that for each case of membership of v_x and v_y to some subproblem C_i, moving between v_x and v_y will not alter the assignment of agent r in Algorithm 2.

case $v_x, v_y \in C_i$**:** agent r is assigned to C_i; w.l.o.g., let $v_x = v$ in line 2. If $\exists u \notin C_i$ for which $(u,v) \in \mathcal{G}$, then both $m' \geq 1$ and $m'' \geq 1$ because v_y is empty, so r is assigned to C_i in line 7. If such a u does not exist, then r is assigned to C_i due to line 10.

case $v_x \in C_i, v_y \notin C_i$**:** suppose that agent r is at v (u and v as in line 3). Note that $m' < m$, because the vertex u is empty.

- **case** $m' \geq 1$**:** r is assigned to C_i in line 7; if r is at u, then $m' \geq 2$, since v is now unoccupied, and r will be assigned to C_i in line 8.
- **case** $m' = 0$**:** note that $m'' \leq m'$, (always, not just in this case), so r is not assigned to C_i; if r is at u, then $m' = 1$, but in line 8, only the first $m'-1$ agents are assigned to C_i, so r is not.

case $\forall i,j\ v_x \notin C_i, v_y \notin C_j$**:** if neither vertex is in a subproblem, then r's membership to a subproblem C_i depends on whether it is within the first $m'-1$ agents on the path from subproblem C_i (line 8). Agent r will not exchange position with another agent by moving between v_x and v_y, so whether or not it is among the first $m'-1$ agents from C_i does not change. □

PROOF OF LEMMA 4.3. Consider Figure 8; to see that, for a solvable instance, the `rotate` operation will always succeed, 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. The moves that clear v' will be reversed at the end of the `rotate` operation.
- All agents in the cycle are assigned to the same subproblem, since any agent not assigned to this subproblem would have been pushed away (Lemma 4.1). This means that agents r and r' can `swap` (Lemma 4.2). □