Auton Agent Multi-Agent Syst (2018) 32:693–740
https://doi.org/10.1007/s10458-018-9391-2

A complete multi-robot path-planning algorithm

Ebtehal Turki Saho Alotaibi[1] · **Hisham Al-Rawi**[1]

Published online: 13 June 2018

Abstract In the domain of multi-robot path-planning problems, robots must move from their start locations to their goal locations while avoiding collisions with each other. The research problem that we addressed is to find a complete solution for the multi-robot path-planning problem. Our first contribution is to recognize the solvable instances of the problem with our solvability test; the theoretical analysis has already been provided to show the validity of this test. Our second contribution is to solve this problem completely, in polynomial time, with the Push and Spin (PASp) algorithm. Once the problem was solved, we found decisions within the complete solution that may improve the performance of the complete algorithm. Hence, our third contribution is to improve the performance by selecting the best path from the set of complete paths. We refer to the improved version of our algorithm as the improved PASp algorithm. In terms of the completeness evaluation, the mathematical proofs demonstrate that the PASp is a complete algorithm for a wider class of problem instances than the classes solved by the Push and Swap (PAS), Push and Rotate (PAR), Bibox or the tractable multi-robot path-planning (MAPP) algorithms. Moreover, PASp solves any graph recognized to be solvable without any assumptions. In addition, the theoretical proof of the PASp algorithm showed completive polynomial performance in terms of total-path-lengths and execution time. In our performance evaluation, the experimental results showed that the PASp performs competitively, in reasonable execution time, in terms of number of moves compared to the PAS, PAR, Bibox and MAPP algorithms on a set of benchmark problems from the video-game industry. In addition, the results showed the scalability and robustness of PASp in problems that can be solved only by PASp. Such problems require

Electronic supplementary material The online version of this article (https://doi.org/10.1007/s10458-018-9391-2) contains supplementary material, which is available to authorized users.

✉ Ebtehal Turki Saho Alotaibi
e.t.otaibi@gmail.com; etalotaibi@imamu.edu.sa

Hisham Al-Rawi
h@it-online.org

1 Computer Science Department, Al-Imam Muhammad Ibn Saud Islamic University, Riyadh, SA, Saudi Arabia

high levels of coordination with an efficient number of moves and short execution time. In grid and bi-connected graphs with too many cycles, PASp required more moves and more time than the PAS, PAR and Bibox algorithms. However, PASp is the only algorithm capable of solving such instances with only one unoccupied vertex. Furthermore, adding heuristic search and smooth operation to the improved PASp showed significant further improvement by reducing the number of moves for all problem instances. PASp produced the best plans in a bit higher time. Finally, the PASp algorithm solves a wider class of problems and performs more completely in very complex/crowded environments than other state-of-art algorithms. Additionally, the Spin operation introduces a novel swapping technique to exchange two items and restore others in a graph for industrial applications.

Keywords Multi-robot · Path planning · Algorithms

1 Introduction

Formally, the multi-robot path-planning problem (MPP) has consisted of a graph and a set of robots. In such problems, each robot must reach its destination in a minimum time with a minimum number of moves. The MPP is known to be a hard decision problems solvable in polynomial space (PSPACE hard) where the complexity of the problem increases exponentially with the size of the configuration space [1]. Because the problem of finding the shortest path of one robot for any problem is NP-hard [2], Goldreich [3] showed that finding the shortest sequence of moves to reach one arrangement of robot from another is NP-hard by reducing 3-Exact-Cover to the MPP. The primary motivations for the multi-robot path-planning problem are to enable robots to complete the tasks of moving objects or carrying out specific tasks within a limited free space. These tasks include but are not limited to:

- Transportation [4], in which robots, in the form of unmanned vehicles, are programmed to follow specific traffic rules.
- Mining [5], in which teams of robots work together to explore unsafe places with the goal of retrieving minerals.
- Planetary exploration [6], in which teams of robotic rovers would be able to intelligently investigate extraterrestrial bodies.
- Autonomous warehouse management [7], in which teams of robots cooperate to retrieve and pack goods in warehouses.
- Search and rescue [8], in which a team of robots work to explore rubble after a disaster using special sensors to locate and rescue buried victims.
- Monitoring and surveillance [9], in which a set of robots could explore a position, locate the positions of specific targets and update the position information to aid surveillance. However, these are not the only motivations. Many tasks in virtual space can also be considered as problems of path-planning for multiple robots, such as:
- Data transfer [10] at communication nodes with limited buffers.
- Computer games [11] in which robots try to find a collision-free path between their starting position and an end goal.

Given the importance of the multi-robot path-planning problem and its applications, it is not surprising that many powerful algorithms have been devised. Kornhauser et al. [12] have provided a complete centralized procedure to solve all MPP problems in a bi-connected graph, in other form of the problem known as Pebble Motion in Graph (PMG). In the same paper, a feasibility test was proposed for polygons and a graph containing a bridge, showing that it is impossible for robots to swap their locations if they are separated by a bridge longer than the number of unoccupied vertices minus two. The Push and Rotate (PAR) algorithm [13] is a complete version of the Push And Swap (PAS) algorithm [14]. The PAS algorithm is intended to be a complete algorithm for problems with at least two unoccupied vertices. For each robot *r*, the PAS algorithm finds the shortest path linking the start location to its goal location. When another robot *s* is detected on a vertex *v* in *r*'s path, *r* has the ability to push *s* away if *s* has a lower priority. If *s* has a higher priority, PAS will switch to the swap algorithm. Finally, each of the two robots is affected by the swap algorithm, which is resolved by moving them back to their previous locations. There have been some reported failure cases of PAS, and PAR was modified accordingly. In addition, PAR introduces the rotate algorithm to resolve cascade moves. The rotate algorithm is invoked to move robots forward in a cycle. The Bibox algorithm [15] solves bi-connected graphs completely. Initially, Bibox decomposes the problem into many handles and one original cycle. Then, robots with goal locations at the outer handles are solved earlier and locked (excluded from the search space). This results in a smaller problem. The PAS and PAR algorithms appear to be the most suitable in the problem domain because they do not impose as many restrictions on the roadmap. Therefore, our contribution in this work is to:

1. Define a feasibility test for all instances allowing a parallelism level defined with at least simultaneous rotations,
2. Design a complete polynomial algorithm solving any solvable graph with no assumptions,
3. Provide an empirical evaluation of our algorithm, including comparisons with the Push and Swap [14], Push and Rotate [13], Bibox [15], and MAPP [16] algorithms, and
4. Provide a brief exploration of the possibilities to improve solution quality times through the use of smooth post-processes and heuristics namely in the area of selecting which path each agent should choose.

The first and part of the second contribution have appeared in a conference paper [17], namely, the main algorithm and a shallow explanation of its execution in figures. The complete listing of relevant algorithms and proofs, as well as the algorithm improvements and the complete evaluations, are new to this paper.

This paper is organized as follows: after this brief introduction, in Sect. 2, the details of closely related work are described. In Sect. 3, the multi robot path-planning problem is defined from the literature. In Sect. 4, the feasibility test and its proof are introduced. In Sect. 5, the Push and Spin algorithm is discussed in detail, followed by Sect. 6 with a discussion of its completeness, solution quality and time analysis. In Sect. 7, the post-process and heuristic searches are discussed. In Sect. 8, the experiments are introduced with a comparative analysis of the selected methods on different types of instances. Finally, conclusions and future work are presented in Sect. 9.

2 Background

Kornhauser et al. [12] has provided a complete centralized procedure to solve all MPP problems in a bi-connected graph. The solutions he has created have upper and lower bounds of O(n^3) moves for graphs with n vertices and require O(n^3) time. However, there is no single algorithmic description of the Kornhauser procedure in the literature. Instead, some work focuses on implementing and improving Kornhauser's ideas. Additionally, Kornhauser proves a feasibility test of the problem instance in the form of a tree of bi-connected (non-separable)

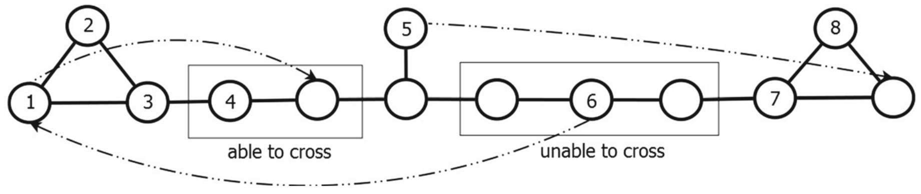


Fig. 1 Graph with solvable and unsolvable instance based on Kornhauser result

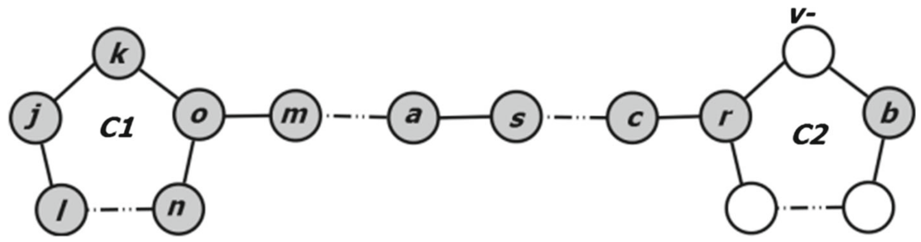


Fig. 2 Solvable, unrecognized instance by Kornhauser

components that are linked by chains of vertices with degree two (called isthmuses). The test is:

> It is impossible for agents to swap if they are separated by an isthmus longer than the number of empty vertices minus two.

To clarify the Kornhauser result, consider sequential moves only. Figure 1 illustrates an instance containing both solvable and unsolvable instances based on the Kornhauser condition. Note that it is possible for any two of the robots in A1 = {a1..., a6} to swap their positions. The same is true for the robots in A2 = {a7, a8} because they too can exchange their positions. For example, robots 5 and 6 can swap their positions and robots 7 and 8 can swap their positions. The condition to be met is that no robot from A1 can exchange positions with any robot in A2. To understand this, again referring to Fig. 1, if robot 7 wants to swap its position with robot 6, it is possible only if they move either to the component at the left side of the vertex occupied by robot 6 or to the component at the right side to execute the swap operation. This swapping would invariably fail because during the swapping, robots passing the bridge to the left component would stack all robots on it. Hence, after this, there will be essentially no possibility to swap unless there are at least enough free nodes to accommodate all opponent robots on the bridge as well as the two swapping robots. The same thing would occur if the swapping robots pass the bridge to the right component.

Even though the Kornhauser procedure provides a complete solution for a wide class of problems, its feasibility test doesn't distinguish solvable instances from unsolvable ones. This is because it considers sequential moves only. Hence, it forces us to the conclusion that there exists a wider class of solvable instances not unrecognized by Kornhauser. We can see it vividly in the instance containing a number of unoccupied nodes equal to the maximum bridge length, and the problem allows simultaneous moves. In that case, the instance is solvable. However, as the bridge length increases, the number of unrecognized solvable instances increases. For example, the Chain-of-Cycles instance described in Fig. 2 is a solvable instance in the simultaneous moves boundaries even though the number of unoccupied nodes equals the maximum bridge.

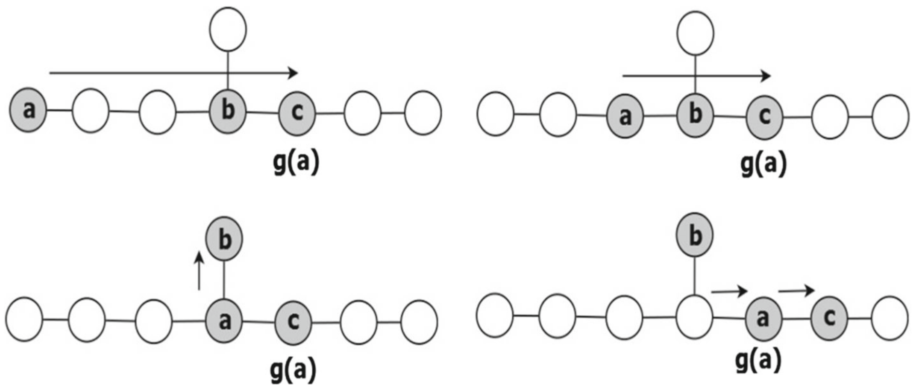


Fig. 3 Push operator: robot *a* faces *b* through its $p*$, and b has lower priority than a, robot a pushes b away from its $p*$

The anomaly is that a wide range of solvable instances with one unoccupied vertex still exists that is excluded from the work for the same class of instance described above. It is understandable that the problem is solvable even if there is one unoccupied node. The feasibility test in Sect. 4 covers these instances.

In the following context, we will review three powerful algorithms. We selected the Push and Swap and Push and Rotate algorithms because they provide complete solutions with few assumptions about the graph to be solved. The Bibox algorithm was selected also because it provides complete solutions with a proven minimum number of moves.

2.1 Push and Swap

The Push And Swap (PAS) algorithm [14] provides a complete MPP algorithm for problem instances with at least two unoccupied vertices. However, it considers the priorities of the robots, which can cause a limitation, and was solved later in another publication [18]. For each robot a, the PAS algorithm finds the shortest path $p*$ linking the start location of a to its goal location and advances the robot through $p*$ with Push And Swap algorithms. PAS solves the problem of n vertices and k robots in O(n^4) time.

2.1.1 Push

If the next vertex in the shortest path $p*$ of robot a is unoccupied, the Push algorithm will advance robot a; if another robot b is detected on vertex v in a's path, a has the ability to push it away if it has a lower priority (Fig. 3). If b has a higher priority, the push operator fails; then, PAS will switch to the swap operator.

2.1.2 Swap

When another robot b is detected on a vertex v in a's path and b has higher priority than a, both robots will advance to the nearest vertex u whose degree is equal to or greater than three. If the vertex u has fewer than two unoccupied neighbor vertices, two occupied vertices should be cleared regardless of the occupying robots' priorities (Fig. 4). Treating robots a and b as obstacles, then, robot a will swap with robot b (Fig. 5). Finally, each robot is affected by the clear operation, which will be resolved by returning it to its previous location.

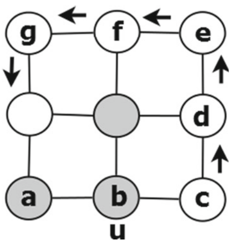


Fig. 4 Robots *a* and *b* are advanced to vertex *u* to achieve swap operation, vertex *u* contains only one unoccupied neighbor, therefore, it should clear the other neighbor. To do it, it should not move *a*, *b* or the unoccupied neighbor, hence, all those vertices will be considered as obstacles for the cleared robots

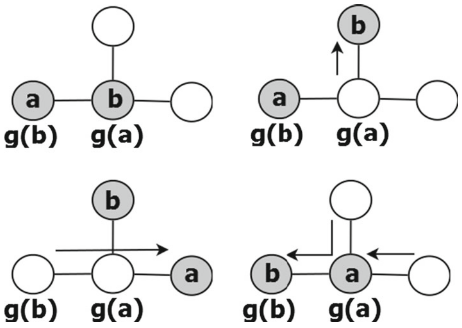


Fig. 5 Swap operator: Robot *a* faces *b* through its SP and *b* has higher priority than *a*, robot *a* and *b* are moved to vertex *u* with degree more than two and swap

2.2 Push and Rotate

The Push and Rotate (PAR) algorithm [13] is a complete version of the PAS algorithm. It solves the problem of *n* vertices and *k* robots with O($k.n^3$) moves in O($k.n^5$) time. It solves any solvable instance recognized by the Kornahuser procedure as instances with a number of unoccupied vertices greater than the number of the bridge length minus two. PAR has addressed four issues contradicting PAS claims. These contradictions will be described in this section exactly as it was presented there.

1. Push and rotate proved that the multi-robot path-planning problem can be solvable when Swap fails. The Polygon graph with two unoccupied vertices is a graph conforming to Push and Swap requirements. However, the MPP problem is solvable in the polygon even though Swap fails due to the absence of a vertex of degree higher than two (Fig. 6).

The isthmus graph is another example proving that the problem can be solved when Swap fails. When a graph satisfies Push and Swap requirements, under some configuration of robot ordering, the isthmus contains a vertex of degree equal to or greater than three; it is solvable but Swap fails (Fig. 7).

2. Push and rotate proved that there are some possibilities that the Clear operation doesn't consider.
3. The Resolve algorithm in PAS executes recursively to return each robot affected by the Swap operator to its previous location, which may lead to a swap with another robot. In this case, the Resolve algorithm will turn to the swapped robot to resolve it. This will

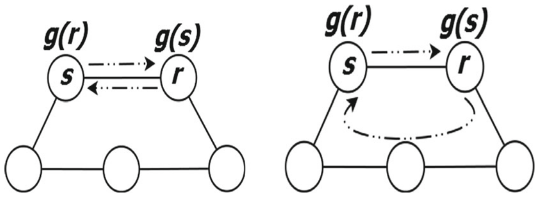
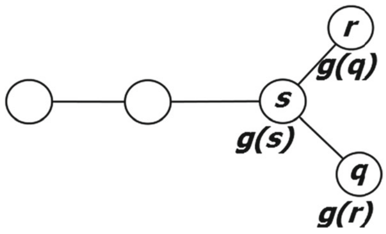


Fig. 6 Polygon graph satisfies Push and Swap requirements (two unoccupied vertices), robot *s* pushes robot *r* to reach its goal position through its $p*$, then, *s* is finished robot, *r* tries to push *s* to reach its goal through its $p*$ and it fails, swap operator will fail also since there is no vertex with degree more than two in Polygon (right), hence, this graph will be unsolvable in Push and Swap context while it is solvable if there is an alternative path for the robot to follow it (left)

Fig. 7 Graph with isthmus is solvable in Push and Swap context if robot *s* has lower priority among others only



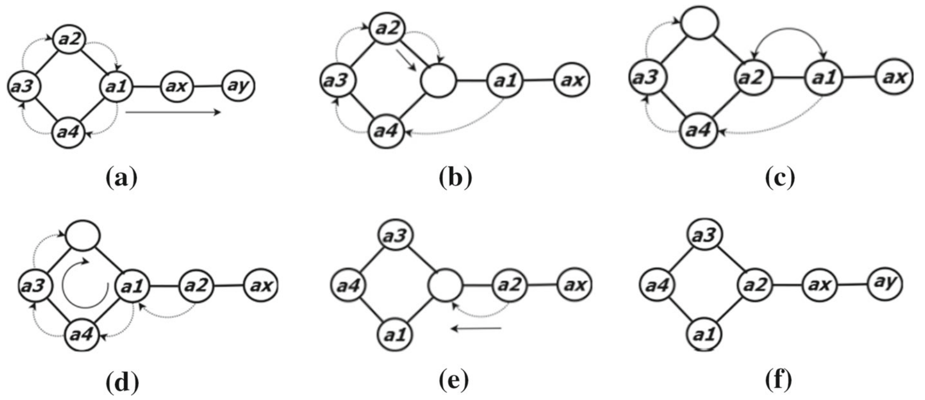


Fig. 8 Rotate operator execution, in the worst case, all vertices in the cycle are occupied

also change the last robot's locations, resulting in moving the robots two steps away from their location. This possibility is not considered by the Resolve algorithm.

4. Push and Rotate proved that new redundant moves result after executing the Smooth operator in PAS that may require executing Smooth again.

In addition, PAR introduces the rotate operator to resolve cascade moves. The rotate operator is called to move robots forward in a cycle. To do so, one vertex is cleared by swapping it with a neighboring vertex. Then, all the robots are rotated clockwise so that any robot affected by the clear operator will be resolved. Figure 8 describes rotation procedure.

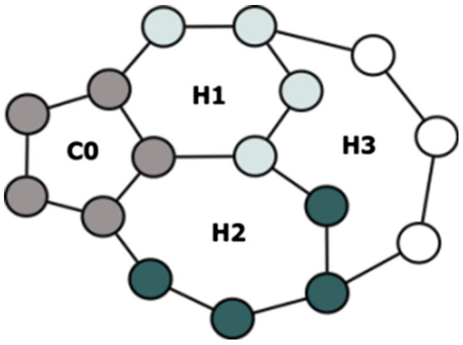


Fig. 9 Decomposing biconnected graph to *i* handles (H_i) and original cycle (C_0)

2.3 Bibox

The Bibox algorithm [15] solves bi-connected graphs completely. Initially, Bibox decomposes the problem into many handles and one original cycle (Fig. 9). Then, robots with goal locations at the outer handles are solved first, then locked and excluded from the search space. This results in a smaller problem; now, the original cycle will be solved in different ways. Bibox solves the problem of n vertices in O(n^3) time and O(n^3) moves. However, Bibox always solves the graph considering the worst scenario. If there are more than two free vertices, it fills those vertices with dummy robots, solves them, and removes their solutions from the final solution. This permits the algorithm to utilize additional free vertices and hence allows improvements.

2.3.1 Solve handle

Solving a handle means starting from the vertex at the beginning of the handle and bringing the robots whose goal vertex is the handle into the handle one by one until all vertices in the handle are finished. If a robot is located outside the handle, bring it to the entering vertex, u, lock the handle to protect the finished vertices from any arbitrary moves, move it to its goal with the rotate operator, and unlock the handle. The idea is that the handle keeps the finished vertices at the beginning, which means they will be rotated and reverse rotated to ensure their robots are back in their locations (Fig. 10). If a robot is located in the handle, rotate the completed vertices until the robot reaches the entering vertex. Advance the robot to any free outer vertex, rotate it in the reverse direction along the path previously traversed to protect finished vertices, and continue with the robot as in the outer robot case (Fig. 11).

2.3.2 Solve original cycle

The original cycle will be solved at the end, when the problem consists of a cycle with two unoccupied entering vertices and all robots inside have their goals inside also. For each two robots that want to swap, rotate them to the entering vertex, exchange their locations, and reverse rotate to restore the previous locations (Fig. 12).

In conclusion, we have presented an exhaustive literature review and explained the modalities of the major algorithms critical to the working of our proposed technique. With such solid work already existing, it is not sensible to propose our entire contribution from scratch (reinventing the wheel). Rather, our proposed work builds on Kornhauser's work. In addition, the Push and Swap and Push and Rotate algorithms appear most favorable for our work

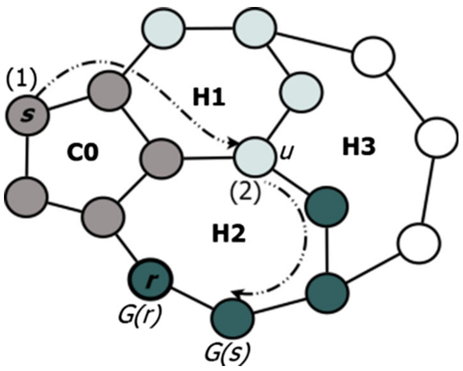


Fig. 10 SolveHandle Case 1, if the robot outside the handle

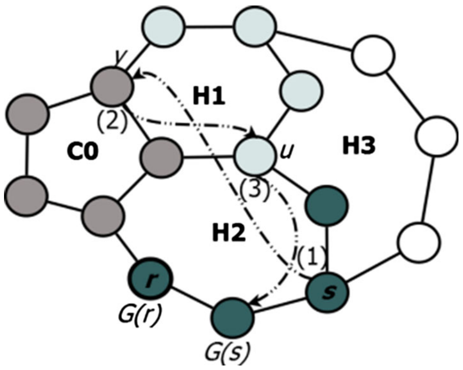


Fig. 11 SolveHandle Case 2, if the robot inside the handle

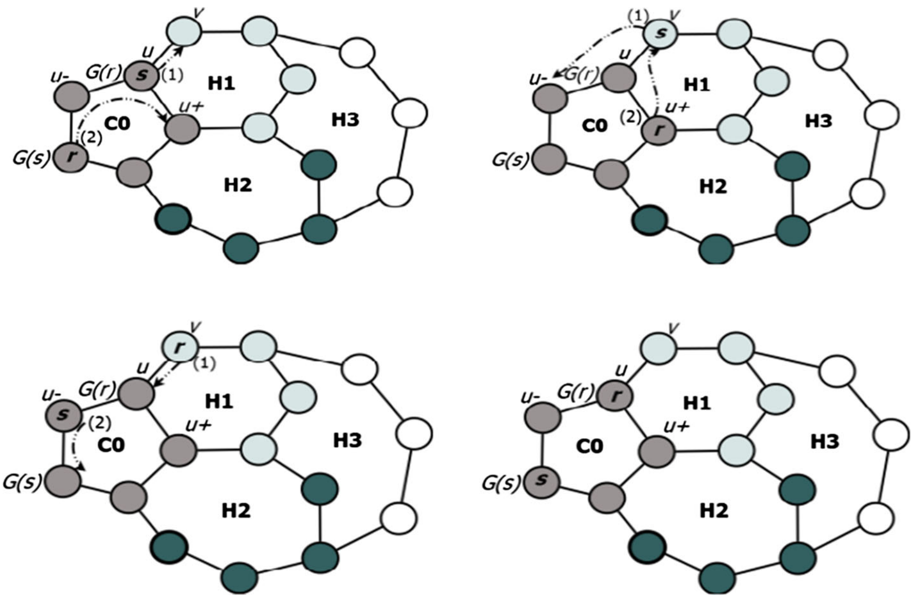


Fig. 12 Solve original cycle

because they impose fewer restrictions on the roadmap. Furthermore, the Bibox algorithm is also suitable because it is powerful for a smaller class of instances, with higher performance in term of execution time and path length produced. Despite the fact that these algorithms seem most appropriate for working with our proposed model, they have their own issues that we intent to resolve with our work as our major contribution based on those algorithms. The major limitations that we attempt to resolve are:

- Kornhauser does not recognize a wide set of instances solvable by simultaneous rotation.
- Push and Rotate reported four failure cases contradicting Push and Swap completeness.
- The Push and Swap algorithm failed to solve the class of instances in which the maximum bridge length equals the number of unoccupied vertices even though it contains two unoccupied nodes.
- Because Push and Rotate is limited to the solvable instances recognized by Kornhauser, it failed to solve a solvable instance with the number of unoccupied nodes equal to the bridge length.
- Push and Rotate produces infinite resolution if there is no ordering of robots.
- All of the moves generated by the Push and Swap and Push and Rotate algorithms are reversed at the end except the moves generated by exchanging the swapping robots.
- Bibox is limited in application because it provides a complete solution for only a bi-connected graph with two free nodes.
- Bibox always solves the graph considering the worst scenario. If there are more than two free nodes, it fills those nodes with dummy robots, solves them, and removes their solutions from the final solution. This permits the algorithm to utilize additional free nodes and hence permits any improvement.
- There is a wide class of solvable instances with only one unoccupied node, which is excluded from the Push and Swap, Push and Rotate and Bibox assumptions.

This is a long list of issues that can be resolved for further optimization. Our work focuses on overcoming these limitations with a new technique to be explained in the next section. There, we define a feasibility test to evaluate the possible solvability of the earlier instance. In addition, we introduce a Push and Spin algorithm that remains based on the principle of pushing and swapping robots but incorporates our proposed suggestions. We will prove through analysis and experimental validation that this new algorithm is complete for any of the class of instances passing the feasibility test.

3 Problem statement

Wilson [19] proposed an efficient decision procedure to solve 15-puzzles, 14-pebbles with one unoccupied vertex, without considering the number of moves. Therefore, his solution requires exponentially many moves. Kornhauser et al. [12] generalized Wilson's solution to solve the Pebble Motion on Graph (PMG) problem with n-1 pebbles in $O(n^3)$ moves. In that work, he defined PMG for a number of pebbles smaller than the number of vertices. A move consists of transferring a pebble to an adjacent unoccupied vertex. In this scenario, the problem is to decide whether or not one arrangement of the pebbles is reachable from another. The PMG problem further attempts to find the shortest sequence of moves needed to achieve the rearrangement wherever possible. In the same paper, Kornhauser also provided solvability conditions for a graph with two unoccupied vertices. One variant of PMG was defined by Yu and Rus [20], who allowed parallel rotation. They defined a problem of finding a sequence of simple moves and rotations that took an initial configuration to the goal configuration, thereby generalizing the problem into Pebble Motion with Rotation (PMR). The PMG problem can easily be mapped to the Multi-robot Path-Planning (MPP) problem. Mächler [21] revealed the differences between PMG and MPP. He showed that PMG models usually assume that there is a central planner attempting to minimize a sequential execution of moves on the graph, while there are mainly two distinctions in MPP models: (1) whether the robots execute in parallel or sequentially, and (2) whether there is a centralized planner or the planning is distributed. Therefore, MPP can be thought of as a generalization of the PMG. It is also evident from the literature that several techniques have emerged to solve sequential MPP. Surynek [15], Luna and Bekris [14], de Wild et al. [13], and many others are working in this domain to solve such types of problem. As noted earlier, parallel moves are one of the distinctions between PMG and MPP. MPP with parallel moves (MPPp) can be defined as a chain of robots that can be moved simultaneously as long as there is an empty vertex at the head on the chain. This particular problem was studied by Ryan [22], who separated subgraphs on the basis of MPPp. Yu and LaValle [23] added parallel rotation to the fully occupied cycle in MPP with parallel moves and rotations (MPPpr) as a natural result for parallel movement.

The model proposed in our work, referred to as Multi-robot Path Planning with spinning (MPPs), is a type of Multi-robot Path Planning with parallel moves and rotations (MPPpr). Its inherent constraint is that parallel moves are allowable only when rotating robots in a cycle (spinning). To illustrate the problem, visualize a roadmap $\mathcal{G} = (V, E)$, which is a connected graph. Some important variables to be used are: a set of robots $\mathcal{R}$, a set of vertices V, an initial assignment of robots to vertices $\mathcal{J}$: $\mathcal{R} \rightarrow V$, and a target assignment of robots to vertices Φ: $\mathcal{R} \rightarrow V$. The functions $\mathcal{J}$ and Φ are total, injective, and non-surjective. Their totality can be judged from the fact that the position of each robot must be specified, injective because one vertex can hold only a single robot at a time, and non-surjective as we require that there always be more vertices than robots. The path of robot i is therefore defined as a map from a time-step to a vertex: p_i: $\digamma + \rightarrow V$. A path p_i is feasible for a robot $r_i \in \mathcal{R}$ if it satisfies the following properties:

1. $p_i(0) = \mathcal{J}(r_i)$, the start position, at time-step zero, of the robot i, is the initial assignment of robot r_i to its start vertex.
2. For each robot i, there exists a finite number of time-steps $k_i \in \digamma +$ such that in the last time-step, the robot i reaches its goal vertex $p_i(k) \equiv \Phi\ (r_i)$.
3. For any $0 \leq k$, $(p_i\ (k), p_i(k+1)) \in$ E or $p_i(k) = p_i(k+1)$. If $p_i(k) = p_i(k+1)$, then the robot r_i stays at vertex $p_i(k)$ between the time steps k and $k+1$.

We say that two paths p_i, p_j are in collision if there exists $k \in \digamma +$ such that $p_i(k) = p_j(k)$ is a collision on a vertex (Fig. 13a), or $(p_i(k), p_i(k+1)) = (p_j(k+1), p_j(k))$ is a collision on an edge (Fig. 13b).

3.1 Problem (MPPs)

Given $(\mathcal{G}, \mathcal{R}, \mathcal{J}, \Phi)$, the problem is to find a set of paths $\mathrm{P} = \{p_1, \ldots, p_n\}$ such that each p_i is a feasible path for the respective robot r_i and no two paths p_i, p_j collide.

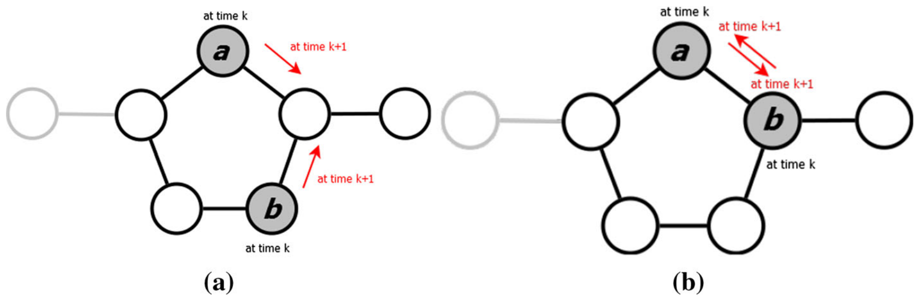


Fig. 13 Collision cases: **a** collision on vertex, **b** collision on an edge

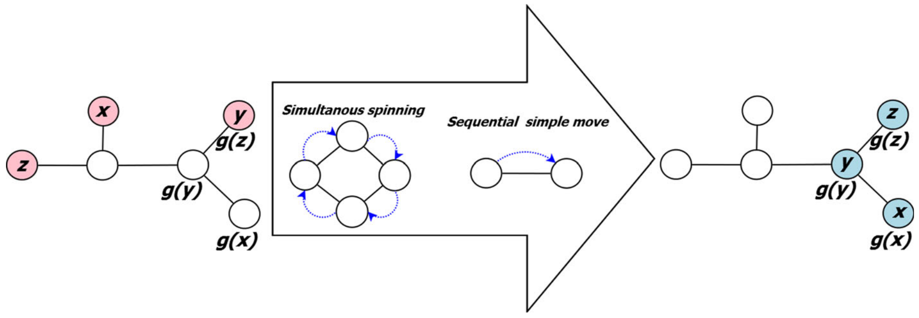


Fig. 14 Multi-robot path planning problem with spinning

Based on the fact that PMR is a generalization of the PMG problem, and due to the availability of parallel rotation [20], MPP is a generalization of PMG because it allows distributed and parallel computations [21]. We can therefore conclude that PMR is a centralized parallel MPP. Furthermore, because MPPs and MPPpr are centralized parallel MPP, we can follow PMR definitions; MPPs allow two types of *moves* of pebbles. In a *simple move*, a robot may move to an adjacent empty vertex in a time-step. In a *spinning move*, robots occupying all vertices of a cycle can rotate simultaneously (clockwise or counterclockwise), such that each robot moves to the vertex previously occupied by its (clockwise or counterclockwise) neighbor (Fig. 14).

4 Feasibility test

1. Polygons are solvable if and only if the start and finish positions differ by a cyclic permutation.
2. The problem allowing parallel rotation would be considered solvable only if the instance contains a cycle and the number of free nodes is greater than or equal to the maximum bridge length.

$\exists$C $\in$ G: C is a cycle $\wedge$ number of free nodes in G $\geq$ max (bridge length) in G $\rightarrow$ G is solvable.

Proof The proof of the first test is provided by Kornhauser [12]. The correctness of the proof of the second test is guaranteed because it is constructed based on Kornhauser's proof. The principal change here is to relax the swap operation. By doing so, instead of swapping

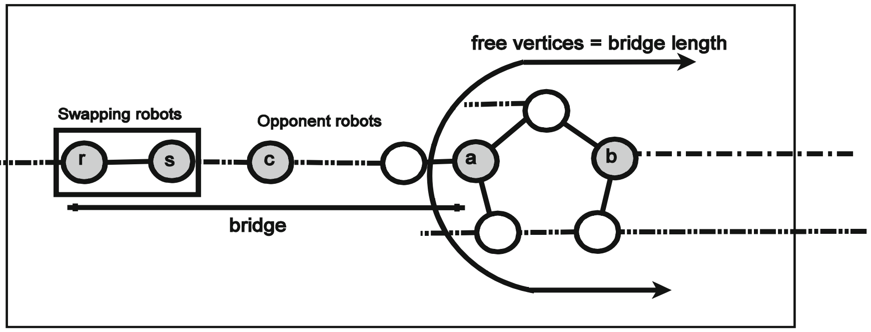


Fig. 15 Feasibility Test

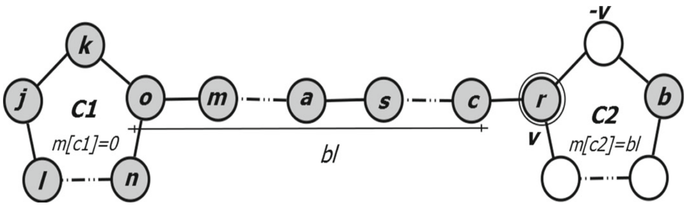


Fig. 16 Feasibility test, Case 1

a vertex, the algorithm achieves a successful swap by requiring the availability of a cycle without any additional empty neighboring vertices. The swapping robots *r* and *s* would pass to the nearest swapping cycle, pushing away opponent robots on their way (Fig. 15). If the graph contains empty vertices equal to the longest bridge length, then the entire path would be cleared. Once the swapping robots reach the entrance vertex of the cycle, the full parallel spinning would be sufficient to swap the swapping robots' positions, as is described in the next section. All out-of-position robots affected by the execution of robot *r* would be restored.

To be accurate, there are two main cases for the current position and goal position of robot *r*, with all other situations being subcases of these two. The cases are:

Case 1 Current position of the robot *r* and its goal position are at a cycle. According to its definition, parallel spinning will always be able to pass a robot occupying a vertex in the cycle to reach its goal on the same cycle even if the cycle is fully occupied.

Case 2 Current position of the robot *r* is at a cycle and its goal position is at another cycle. Let G be an instance containing two cycles C_1 and C_2, a bridge with length *bl*, and free vertices *m* equal to *bl*. In the worst case, the bridge is fully occupied, with the robot *r* at the join vertex of C_2 and its goal position inside C_1. Consider three main configurations in this case;

1. All free vertices are inside C_2 while C_1 is fully occupied (Fig. 16).

In this case, robot *r* would occupy the join vertex *v*, which is the entrance to C_2. However, the solvability would still be guaranteed by moving robot *r* away one step to $-v$, followed by shifting all robots in the bridge to C_2 and passing robot *r* toward C_1 through the free bridge. Once *r* reaches the head of the bridge, which is the entrance to C_1, the robot will be treated as Case 1.

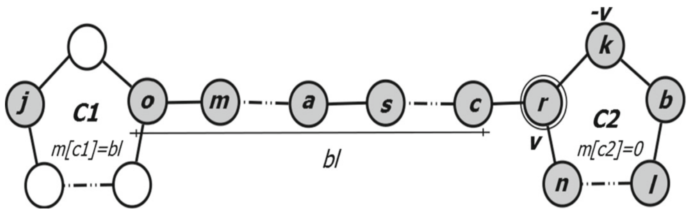


Fig. 17 Feasibility test, Case 2

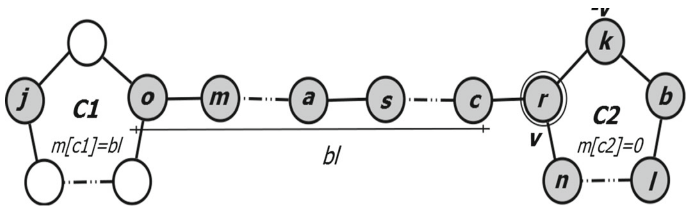


Fig. 18 Feasibility test, Case 3

2. The free vertices are divided between C_1 and C_2 equally (Fig. 17).

In this case, robot *r* would occupy the join vertex *v*, which is the entrance to C_2. However, the solvability of the problem would still be guaranteed by moving robot *r* away one step to $-v$ followed by shifting half of the robots in the bridge to C_2. Once this is achieved, robot *r* would be passed towards C_1 through half the free bridge and all opponent robots in the second half would be shifted toward C_1. Once *r* reaches the head of the bridge, which is the entrance to C_1, the robot will be treated as Case 1.

3. All free vertices are inside C_1 while C_2 is fully occupied (Fig. 18).

In this case, robot *r* would occupy the join vertex *v*, which is the entrance to C_2. However, the solvability of the problem would still be guaranteed by shifting all of the robots in the bridge to C_1 and then passing robot *r* toward C_1 through the free bridge. Once *r* reaches the head of the bridge, which is the entrance to C_1, the robot will be treated as Case 1.

These results can be generalized for any division of the free vertices. Furthermore, they can also be generalized to grid and bi-connected graphs when they contain a set of cycles connected by a bridge of length one (an edge). Hence, one empty vertex is sufficient to make theses graphs solvable. Furthermore, this result can clearly be applied to any instance, e.g., a tree attached at only one end to a bi-connected component, the grids, the bi-connected components, a tree with leaves of cycles, the connectors, the corners, the loop-chain, etc.

5 Push and spin

Push and spin uses the same version of the Push operation proposed by the Push and Rotate algorithm, and introduces a new swapping technique. The principal idea of this novel swapping technique is based on the following fact:

If we rotate a cycle a number of rotations equal to the degree of the cycle, the cycle will have returned to its initial configuration

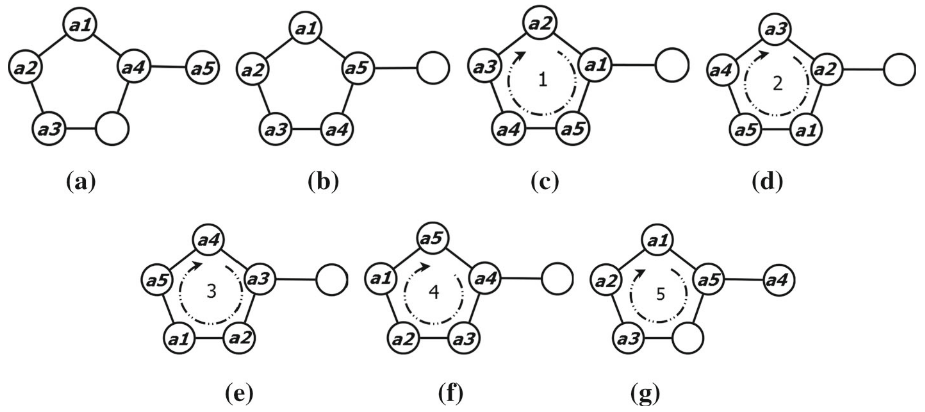


Fig. 19 Swapping robots a4 and a5 by using full rotation idea without affecting others

While abiding by this fact, any new algorithm need not reverse all its moves because all occupying robots would already be back to their previous locations before the rotations started.

Based on this fact, if two consecutive robots are pushed into the cycle, they will swap their positions if the cycle is rotated a number of rotations equal to its degree. Furthermore, all robots in the cycle will have returned to their previous positions with no reverse or resolve operations required. Figure 19 shows two robots $a4$ and $a5$ wanting to swap their positions using the rotation idea. First, these robots will be pushed inside the cycle (Fig. 19a), and then the cycle will be rotated in clockwise direction by a number of rotations equal to its degree (Fig. 19b–g). The final configuration will show that robots $a4$ and $a5$ have been swapped while other robots have already returned to their previous positions.

To understand the Push and Spin algorithm, the following predicates will be used.

$A(s)$: the vertex occupied by robot s.

$A^{-1}(v)$= the robot occupying vertex v.

m: the number of unoccupied nodes in graph $\mathcal{G}$.

m' (u, v): the number of unoccupied nodes in the portion of $\mathcal{G}$ behind v and excluding edge (u, v). If the portion of $\mathcal{G}$ behind v is already connected with the part of $\mathcal{G}$ behind u by a path other than (u, v), m' will equal m except u and v.

next (s) = the vertex next to A(s) on the shortest path of s.

Push $(s, p, \emptyset)$ pushes robot s blindly through p. The Push operation will push away all robots on p even though they are finished robots because the operation will consider the finished set as obstacles.

Vertex is available: not occupied by any robot.

Path is reachable: a path linking that vertex is connected, but it is not necessary that all its nodes be available.

Finished robot: robot at its goal node.

Definition (*Simultaneous rotation*) The rotation of robots in the cycle follows the least restrictive parallelism model that allows a robot to move to a vertex from which another robot is moving away even if there are no empty vertices.

5.1 Push and Spin algorithm

The Push and Spin algorithm (Algorithm 1) takes a connected graph $\mathcal{G}$ as its input. In this scheme, the assignment function is from an occupying robot to an occupied vertex; $\mathcal{G}$ is an invertible one-to-one function while the goal function is from the robot to its goal vertex; Φ is an invertible one-to-one function and set of robots $\mathcal{R}$. The Push and Spin algorithm returns a list of solutions, Π; the ID of each index in this list is the time-step, and the index itself contains a list of moves made at that time-step. In sequential moves, each index in the list contains an ID and a list with only one move. However, in the case of parallel moves, many moves will be recorded at the same time-step. List Π takes all robots from their start positions to their goal positions. The Push and Spin algorithm will extract all of the cycles in the graph. The extract-cycles operation computes all non-trivial bi-connected components. These components can be obtained in linear time $O(|V|+|E|)$ [24]. If there are no cycles in the graph, the algorithm will switch to the Push and Swap[1] algorithm. Once cycles are extracted, robots will be selected in an arbitrary manner and the Push and Spin algorithm will calculate the shortest path connecting robot a with its goal by an A* search. This is done by iteratively applying Push and Spin operators that move robot a toward its goal. We use the same Push version, modified by the Push and Rotate algorithm [13], pushing the robot one step only. If pushing robot a to its goal by the Push operation fails (line 15), the result will be that a finished robot from $\mathcal{U}$ will be blocking a. In such a case, the algorithm would switch to the next operator, the Spin operation (line 18). This operator will swap a and the robot blocking a. In addition, if the Spin operation fails, all the attempts to advance the robot through its shortest path have expired and robot a is congested in its position. In this case, the algorithm will consider this robot as a Resolved robot and add it to the *ResolveStack* line 19). Note that robots in the *ResolveStack* will not be executed unless one robot reaches its goal. If the robot a reaches its goal, it will be added to the finished robots set $\mathcal{U}$, and the robots affected by its solution—in the *ResolveStack*—will be re-executed in a LIFO order.

5.2 Push operation

The Push operation (Algorithm 2) attempts to advance robot a to its destination node, v, and to insert all produced moves sequentially into Π (i.e., in different lists with different IDs). If the destination vertex v is occupied by another robot (line 3), the Push operation has the right to push non-finished robots (i.e., robot $\notin \mathcal{U}$) without resolving them because at its execution time, the shortest path will be calculated from a robot's current position to its goal position irrespective of its current position. To push the conflicted robot away, the Push operation, on an ad hoc basis (line 9), would block the current position of robot a and the finished robot at $\mathcal{U}$ (line 4). As a result, the shortest-path algorithm will consider those blocked robots as obstacles. Then, the operation will find the nearest reachable free vertex to which to push the conflicted robot (line 5) while calculating the shortest path to it (line 6) and advancing the conflicted robot to it. If there is no path linking the conflicted robot with that node, (e.g., the conflicted robot is a finished robot), the Push operation will exit with a false return. If all conflicted robots are pushed away and the robot a reaches its goal (line 11), the operation will exit with true.

[1] To cover an instance of a tree.

Algorithm.1 `Push and Spin` ($\mathcal{G}, \mathcal{A}, \Phi, \mathcal{R}$)

```
1.  Mark all nodes type in $\mathcal{G}$ as "B"
2.  $\mathcal{C}$ = extract_cycles ($\mathcal{G}$)
3.  if $\mathcal{C} = \emptyset$
4.      return Push And Swap ($\mathcal{G}, \mathcal{A}, \Phi$)
5.  $ResolveStack = \emptyset$
6.  $\Pi = \emptyset$
7.  $\mathcal{U} = \emptyset$
8.  $a = \emptyset$
9.  while $\mathcal{U} \neq \mathcal{R}$
10.     if $a = \emptyset$
11.         $a = next\ (\mathcal{R}/\mathcal{U})$ ⊲ select next robot in $\mathcal{R}$ and not in $\mathcal{U}$
12.     $p^*$ = shortest-path($\mathcal{G}$, $\mathcal{A}(a)$, $\Phi(a)$)
13.     while $\mathcal{A}(a) \neq \Phi(a)$
14.         $v \leftarrow p^*$.next ($a$)
15.         if Push ($\Pi, \mathcal{G}, \mathcal{A}, a, v, \mathcal{U}$) failed then
16.             $p^*$.blocked($a, b$) ⊲ $b$ is the conflicted robot will be on next vertex on $p^*$
17.             $\Pi^* = \Pi$
18.             if Spin ($\Pi^*, \mathcal{G}, \mathcal{A}, \Phi, a, b, p^*, \mathcal{U}^*, \mathcal{C}, ResolveStack$) failed then
19.                 $ResolveStack.push\ (a)$
20.                 $a = \emptyset$
21.                 break
22.             else
23.                 $\Pi = \Pi \cup \Pi^*$
24.     if $\mathcal{A}(a) = \Phi(a)$
25.         $\mathcal{U} = \mathcal{U} \cup a$
26.         $a = ResolveStack.pop$
27. return $\Pi$
```

Algorithm.2 `Push` ($\Pi, \mathcal{G}, \mathcal{A}, r, v, \mathcal{U}$)

```
1.  if $\mathcal{A}[r] \neq v$ then
2.      Advance $r$ to $v$ until blocked, inserting intermediate actions into $\Pi$ in sequence indexes
3.      if $\mathcal{A}[r] \neq v$ then
4.          Mark $\mathcal{A}[r]$ and $\mathcal{U}$ as blocked on $\mathcal{G}$
5.          $v_e \leftarrow$ reachable empty vertex to $v$ on $\mathcal{G}$
6.          $p \leftarrow$ shortest-path($\mathcal{G}, v, v_e$)
7.          if $p = \emptyset$ then
8.              return false
9.          Mark $\mathcal{A}[r]$ and $\mathcal{U}$ as free on $\mathcal{G}$
10.         Move robots on $p$ toward $v_e$; insert actions into $\Pi$ in sequence indexes
11. return true
```

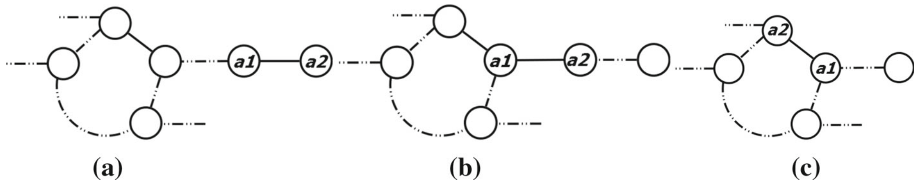


Fig. 20 Main three cases of swapping robots locations related to the cycle

5.3 Spin operation

All situations of swapping robots positions related to the swapping cycle can be summarized into three main cases. The first is the bb-case (Fig. 20a), which occurs when the swapping robots are on the bridge. The second case, known as the cb-case (Fig. 20b), occurs when one swapping robot is on the cycle and the other is on the bridge. The last case, called the cc-case (Fig. 20c), is encountered when the swapping robots are inside the cycle. All cases use the same swapping technique with some differences in their details. The Spin operation checks the swapping robots' positions and executes the corresponding case.

Spin operation (Algorithm 3) works as follows. If the robots are inside the cycle, they will be swapped by cc-case (Algorithm 4). If the swapping cannot occur, the Spin operation will exit with false (line 3). On some occasions we may have a situation where the robots are on the bridge, or one is on the bridge and the other on the cycle, and the problem instance has passed the feasibility test. In such a case, there will always be at least one swapping cycle containing sufficient unoccupied nodes to receive all opponent robots and the swapping robots and execute swap. In this case, the swapping cycle should be calculated by the compute-swapping-cycles operation. For each extracted cycle, the compute-swapping-cycles operation[2] will compute m', the total vertices in the part of the graph remaining when the edge between the robot and the swapping robot is blocked; m'' will be the number of occupied vertices of m'. The total vertices able to receive the opponent robots will be m'–m''. If these vertices are sufficient to receive all opponent robots plus the swapping robots, then the cycle is the swapping cycle. Afterwards, the swapping robots should be swapped by the bb-case operation (Algorithm 4). As with the previous case, if robots cannot be swapped, the Spin operation will exit with false. Otherwise, the Spin operation will try to resolve the swapped robot (line 9) and exit with true.

Algorithm.3 `Spin` ($\Pi^*, \mathcal{G}, \mathcal{A}, \Phi, s, r, p^*, \mathcal{U}, \mathcal{C}, ResolveStack$)

```
1. if $\mathcal{G}$.type ($\mathcal{A}[s]$) = "$C$" AND $\mathcal{G}$.type ($\mathcal{A}[r]$) = "$C$" then
2.     if cc-case ($\Pi^*, \mathcal{G}, \mathcal{A}, s, r, \mathcal{U}, C_e$) failed then
3.         return false
4. else
5.     $\mathcal{C}$ = compute swapping cycles ($\mathcal{G}, \mathcal{A}, \mathcal{C}, s, r$)
6.     if bb-case ($\Pi^*, \mathcal{G}, \mathcal{A}, s, r, \mathcal{U}, \mathcal{C}$) failed then
7.         return false
8. Resolve ($\Pi^*, \mathcal{G}, \mathcal{A}, \Phi, s, r, p^*, r, \mathcal{U}, ResolveStack$)
9. return true
```

[2] Details of the compute swapping cycle operation are in Appendix A.3 of Supplementary material.

5.4 cc-Case operation

In its first step, the cc-case operation (Algorithm 4) will select a join vertex *v* joining the cycle with the rest of the graph (line 1). This will be followed by selecting one of its neighbors, *u*, such that *u* is not implied to the cycle (line 2). To check its eligibility to receive the swapping robots, the prepare cycle operation[3] will borrow an unoccupied node either from the cycle or out of the cycle to ensure that the vertex *u* is not occupied. However, we must keep in mind that any moves made by the prepare cycle operation will be reversed[4] at the end.

Now, the instance containing a cycle with occupied nodes and the unoccupied join node's neighbor, *u*, (Fig. 21a) needs to be addressed. In this case, *partial_spin* will be calculated as the number of nodes in the cycle between the vertex occupied by the swapper robot *s* and the join vertex (line 8). When this process is completed, the cycle will be rotated simultaneously a number of rotations equal to *partial_spin* (Fig. 21b). When the swapper robot *s* reaches the join vertex *v* (line 9), the operation will multi-push the swapper robot *s* (line 10) one step toward *u* and the swapped robot *r* one step toward *v*. This will leave one unoccupied vertex at *v*—(Fig. 21c) and execute one rotate (line 10), so that the unoccupied vertex will be vertex *v* (Fig. 21d). As a result, swapper robot *s* will be pushed back (line 11) to the join vertex *v* (Fig. 21e). The final step of the rotations will be to rotate the cycle as many times as the number of rotations, which will be its degree minus the *partial_length* that we calculated previously minus one (line 12), (Fig. 21f). Upon completion of all these operations successfully, the moves done in the borrowing steps will be reversed. This will be done because at this point, we will be certain that the operation has backed up the instance to exactly its previous configuration (Fig. 21g). Note that if one of these operators fails, the operation will exit directly with false and the caller algorithm will totally destroy all moves resulting from this attempt.

Algorithm.4 `cc-case` (Π^*, $\mathcal{G}$, $\mathcal{A}$, s, r, $\mathcal{U}$, $\mathcal{C}_e$)

```
1.  $v$ = entering vertex : $v \in \mathcal{C}_e$ : $\mathcal{G}$.degree($v$) $\geq 3$
2.  $u \leftarrow \mathcal{G}$.adj[$v$] : $u \notin \mathcal{C}_e$
3.  start_reversing_index = $\Pi^*$
4.  if prepare cycle ($\Pi^*$, $\mathcal{G}$, $\mathcal{A}$, $s$, $r$, $v$, $u$, $\mathcal{C}_e$) failed then
5.      return false
6.  end_reversing_index = $\Pi^*$
7.  back_index = 0
8.  partial_spin $\leftarrow$ shortest_path($\mathcal{G}$, $\mathcal{A}[s]$, $v$)
9.  if execute spin ($\mathcal{G}$,$\mathcal{A}$,$\mathcal{C}_e$, length (partial_spin), $\Pi^*$) then
10.     if multipush ( $\Pi^*$, $\mathcal{G}$, $\mathcal{A}$, $s$, $u$, $r$, $v$, $\emptyset$) AND execute_spin ($\mathcal{G}$,$\mathcal{A}$,$\mathcal{C}_e$, 1, $\Pi^*$) then
11.         if Push ($\Pi^*$, $\mathcal{G}$, $\mathcal{A}$, $s$, $v$, $\emptyset$) then
12.             if execute spin ($\mathcal{G}$,$\mathcal{A}$,$\mathcal{C}_e$, length ($\mathcal{C}_e$) – length (partial_spin) - 1, $\Pi^*$) then
13.                 if reverse (s, r, start_reversing_index, end_reversing_index,
                    back_index ,cc, $\Pi^*$) then
14.                     return true
```

[3] Details of the compute prepare cycle operation are in Appendix A.1 of Supplementary material.

[4] Details of the reverse operation are in Appendix A.4 of Supplementary material.

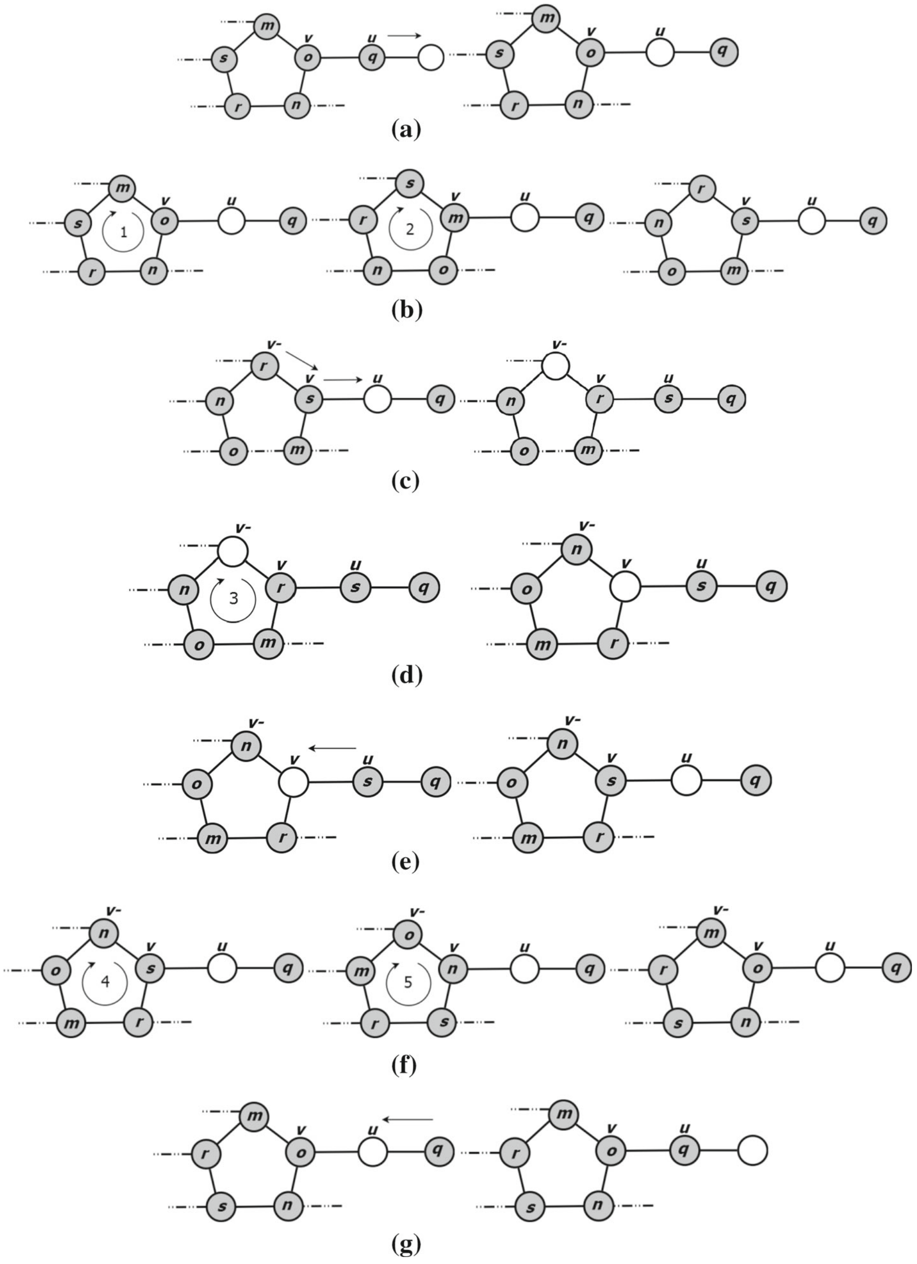


Fig. 21 Execution of cc-case operation. **a** cc-case after preparing the join node's neighbor (u). **b** cc-case after executing partial rotation. **c** cc-case after multipushing the swapping robot *s* and *r* to *v* and *u*. **d** cc-case after one additional rotate. **e** cc-case after pushing swapper robot *r* back to vertex *v*. **f** cc-case after full rotation. **g** cc-case after reverse borrowing operator

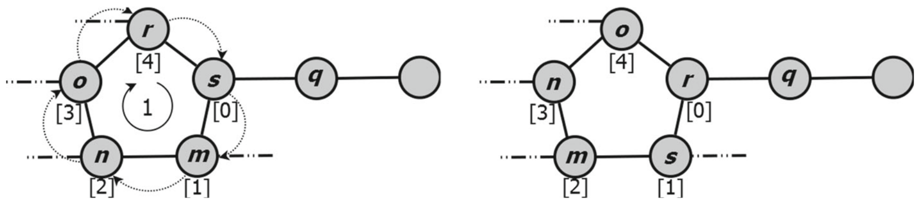


Fig. 22 Execute spin one time

5.5 Execute spin operation

As described earlier, our problem definition allows simultaneous rotation of robots in a cycle. The advantage of this model is that it is the least restrictive model because it allows robots to move even if there is no unoccupied node available. In the execute spin operation (Algorithm 5), for each iteration, the cycle will spin all robots occupying its nodes simultaneously one time clockwise. A list of moves will be initialized in each iteration (line 2), and for every occupied vertex in the cycle (line 4), the occupying robot will be moved to its neighbor vertex in the clockwise direction. This task will be performed by numbering the vertices in the cycle increasing in the clockwise direction (line 6). Even if a neighbor vertex is occupied, the operation would ensure the availability of the neighbor vertex because all robots would be moved simultaneously (Fig. 22). For the same reason, the robot will be pushed blindly—∅ in the finished-robots set—to the next vertex (line 7). Finally, the list of moves generated in the same iteration will be added to the solution list in the same index, because an index means a time-step (line 9).

Algorithm.5 `execute spin` ($\mathcal{G}$, $\mathcal{A}$, $\mathcal{C}_e$, spin number, Π^*)

```
1. for iteration = 0 → spin number
2.     $\Pi^{**} = \emptyset$
3.     for every $v \in \mathcal{C}_e$
4.         if $\mathcal{A}^{-1}[v] \neq \emptyset$ then ⊲ occupied node
5.             $index = \mathcal{C}_e^{-1}[v]$
6.             $u \leftarrow \mathcal{C}_e[(index+1)\% \text{length}(\mathcal{C}_e)]$ ⊲ neighbor in clockwise direction.
7.             if Push ($\Pi^{**}$, $\mathcal{G}$, $\mathcal{A}$, $\mathcal{A}^{-1}[v]$, $u$, $\emptyset$) failed then
8.                 return false
9.     insert $\Pi^{**}$ into same index in $\Pi^*$
10. return true
```

5.6 bb-Case operation

In its first step, the bb-case operation (Algorithm 6) will select a join vertex v to the swapping cycle C that joins the cycle with the rest of the graph (line 1). It will then select one of its neighbors, u, such that u is not implied to the swapping cycle C (line 2). The operation will push the swapping robots s and r blindly toward the nodes u and v, respectively, with the multipush[5] operator blindly. This will push all opponent robots and the swapping robots

[5] Details of the multi-push operation are in Appendix A.2 of Supplementary material.

toward the swapping cycle regardless of any finished robot. This is because the set of finished robots sent by Ø (line 4) and all moves generated by this operator will be reversed at the end (Fig. 23a). Once this step is completed, the swapping robot will again be multi-pushed to the swapping cycle (line 6). The goal of doing this process in two steps is to save the index of the moves to be reversed (Fig. 23b). The swapping cycle will be rotated a number of rotations equal its degree (line 7), (Fig. 23c–f). At this stage, the swapper robot will be at the join node. The operation will push it one step out of the cycle (line 8), (Fig. 23g), and the cycle will continue its rotation (line 9), (Fig. 23h). Upon completion the robots will swap the moves generated by multipush in the first step and all other robots will reverse their moves (line 11), (Fig. 23i, j).

```
Algorithm.6 bb-case ($\Pi^*$, $\mathcal{G}$, $\mathcal{A}$, $s$, $r$, $\mathcal{U}$, $\mathcal{C}$)
1.  $v$ ← join vertex to $\mathcal{C}$ : $\mathcal{G}$.degree($v$) $\geq 3$
2.  $u$ ← $\mathcal{G}$.adj[$v$] : $u \notin \mathcal{C}$
3.  start_reversing_index = $\Pi^*$
4.  if multipush ( $\Pi^*$, $\mathcal{G}$, $\mathcal{A}$, $s$, $v$, $r$, $u$ , $\emptyset$) then
5.      back_index = $\Pi^*$
6.      if multipush ( $\Pi^*$, $\mathcal{G}$, $\mathcal{A}$, $s$, $\mathcal{C}$[1], $r$, $\mathcal{C}$[0] , $\emptyset$) then
7.          if execute spin ($\mathcal{G}$,$\mathcal{A}$, $\mathcal{C}$, length ($\mathcal{C}$)-1, $\Pi^*$) then
8.              if push ( $\Pi^*$, $\mathcal{G}$, $\mathcal{A}$, $s$, $u$, $\emptyset$) then
9.                  if execute spin ($\mathcal{G}$,$\mathcal{A}$, $\mathcal{C}$, 1, $\Pi^*$) then
10.                     end_reversing_index = $\Pi^*$
11.                     if reverse ($s$, $r$, start_reversing_index,
                        end_reversing_index, back_index,bb, $\Pi^*$) then
12.                         return true
```

5.7 Resolve operation

As mentioned earlier, the Resolve operation was proposed initially by the Push and Swap algorithm. Push and Rotate reported the original Resolve operation results in a recursive resolve case and therefore proposed re-executing the algorithm to resolve the problem. In addition, the Push and Rotate resolving mechanism results in infinite executions of the problem in the case of no order between robots. In our proposed algorithm we cannot use their resolving approach. Push and Spin assumes that merging both ideas may solve both issues. Recalling the Push and Swap resolve technique, Resolve operation in Push and Swap executes recursively to return each robot affected by the Swap operator to its previous location. This may lead to a swap with another robot. In this case, the Resolve operation will turn to the swapped robot to resolve it, which also may change the last robots' locations. This will result in moving the robots two steps away from their locations, which is not considered by Push and Swap Resolve operation. The Push and Rotate algorithm stacks all resolving robots in a list and tries to re-execute them after a robot in turn reaches its goal. This maintains the idea that all resolving robots will remain one step away their goals. The Resolve operation (Algorithm 7) will attempt to resolve the swapped robot by moving the swapping robot one

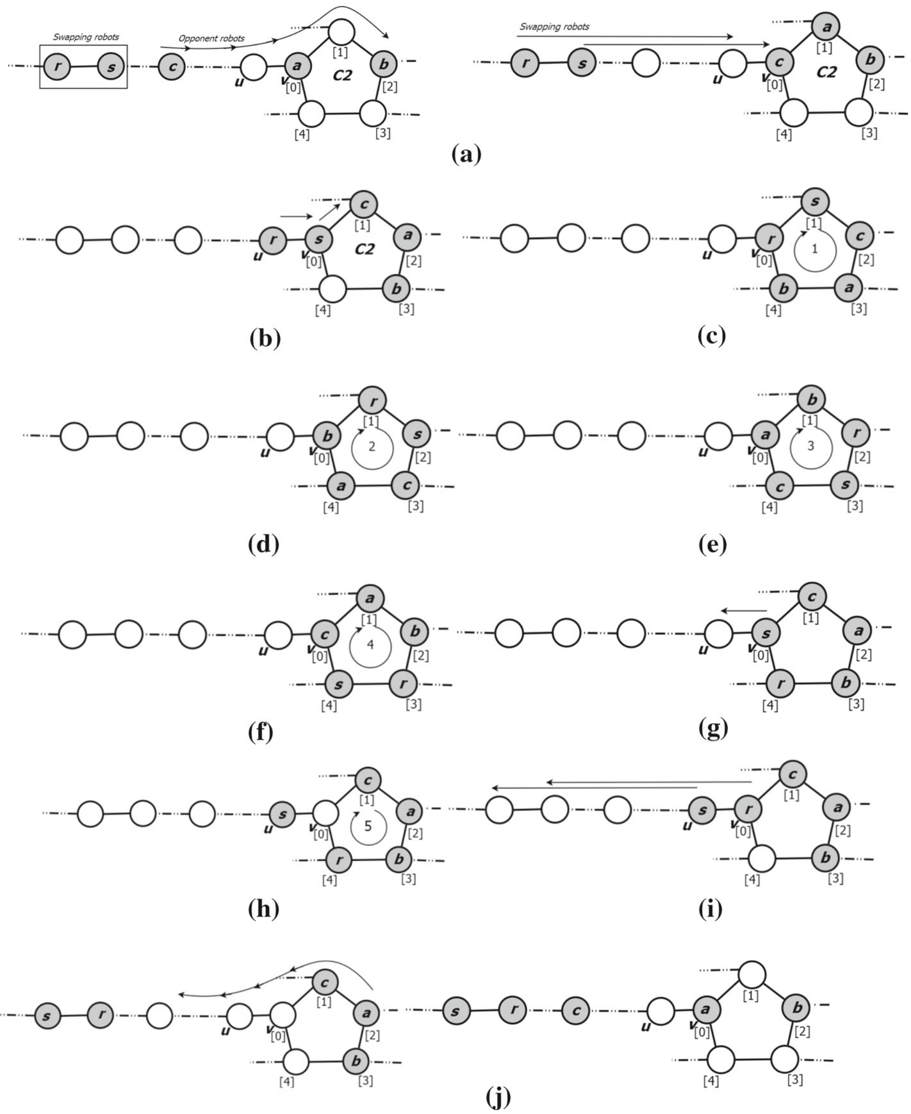


Fig. 23 Execution of bb-case operation. **a** bb-case multipushes the swapping robots blindly will push all the opponent robot forward. These moves will be reversed. **b** bb-case multipushes the swapping robots into the swapping cycle, this move will not reversed. **c** First rotation in bb-case. **d** Second rotation in bb-case. **e** Third rotation in bb-case. **f** Fourth rotation in bb-case. **g** bb-case pushes swapper robot out. **h** Fifth rotation in bb-case. **i** bb-case reverses the moves generated by multipush in **a**. **j** bb-case reverses the moves generated by multipush in **a**

step toward its goal and re-push the swapped robot to its goal. If this fails, the swapped robot will be stacked to be re-executed after the swapper robot reaches its goal. Note that it will be removed from finished robot set $\mathcal{U}$ so it will be re-executed naturally in the Push and Spin algorithm.

```
Algorithm.7 Resolve ($\Pi^*$, $\mathcal{G}$, $\mathcal{A}$, $\Phi$, $s$, $p^*$, $r$, $\mathcal{U}$, $ResolveStack$)
1. $v = p^*$. next($s$) ⊲ after spin, $s$ location will be at $r$ location
2. if Push ($\Pi^*$, $\mathcal{G}$, $\mathcal{A}$, $s$, $v$, $\mathcal{U}$) then
3.         if Push ($\Pi^*$, $\mathcal{G}$, $\mathcal{A}$, $r$, $\Phi[r]$, $\mathcal{U}$) failed then
4.                 $ResolveStack$.push( $r$ )
5.                 $\mathcal{U}$.remove ( $r$ )
```

6 Analysis

6.1 Proof of completeness

Lemma 1 *Push (s, v) operation* (*Algorithm* 2) *always advances robot s one step to v=next(s) in a connected G successfully if the vertex v is unoccupied by any finished robot and if there is at least one free vertex u in the rest of the graph and if all robots in the path between v and u are non-finished robots.*

$$A^{-1}(v) \notin \mathcal{U} \wedge m' \geq 1 \wedge \exists u : u \text{ is free} \wedge \ \forall r \in \text{shortest-path } (v, u) : r \notin \mathcal{U} \rightarrow \text{Push } (r, v)$$

Proof Actually, there are many cases for vertex v=next(s). However, we would limit ourselves to the only case that makes the left hand side of implication true.

Let $r1$ be the blocking robot at vertex v such that $r1$ is non-finished robot. Let m' be equal to one so there is an unoccupied vertex u in the remaining part of the graph. Assume that the paths between vertex v and u are fully occupied by non-finished robots (Fig. 24a).

By the definition of Push operation (Algorithm 2), it would always able to push non-finished robot away the pushed robot's path. While there is a chain of robots between v and u (Fig. 24a), one push will move all robots one step occupying vertex u and freeing vertex v. Once vertex v is free, robot $r1$ can move to (Fig. 24b). Hence, robots s will be pushed to node v successfully.

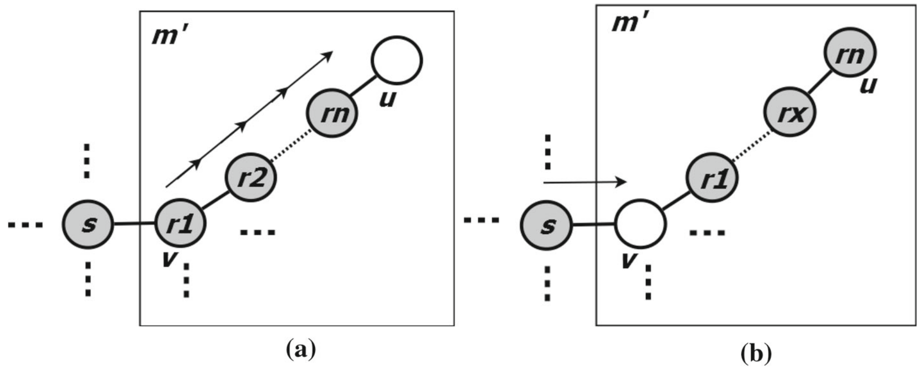


Fig. 24 Lemma 1, Push operation clears the path of non-finished robots (**a**) and pushes robots (**b**)

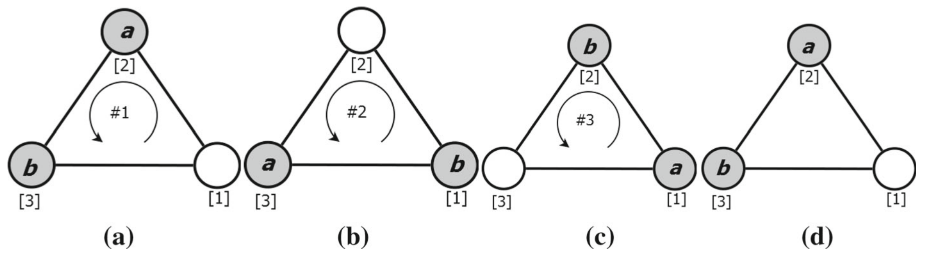


Fig. 25 Lemma 3, base case. **a** Configuration C_1. **b** Configuration C_2. **c** Configuration C_3. **d** Configuration C_1

Lemma 2 *Multipush (s, r) operation*[6] *always advances the compost robot R, composed of r and s robots, one step to v = next(R) in a connected G successfully if the vertex v is unoccupied by any finished robot and if there is at least one free vertex u in the rest of the graph and if all robots in the path between v and u are non-finished robots.*

Proof This implication can be proved directly based on Lemma 1, once the fronted robot is pushed successfully, it would leave an unoccupied vertex which would be the destination of the follower robot.

Lemma 3 *If the cycle has been rotated in any direction a number of rotations equal to its degree, it will restore the initial configuration C.*

$$C = C + \text{full rotation}$$

Proof Base Case Let robots a and b be located on index 2 and 3 of the smallest cycle respectively in configuration C_1. The degree of the smallest cycle would be set at three. If the cycle has been rotated three times, the configuration would be changed to C_2, C_3 and C_1 again. That would mean that robots a and b would be back to indexes 2 and 3 (Fig. 25)

$$C_1 = C_1 + three\text{rotations.}$$

Inductive Hypothesis Let's consider a cycle with degree k, robot a and b be located at indexes $k-1$ and k respectively in configuration $C_{[k]1}$. This would mean that if the cycle has been rotated k times, the configuration would be changed to $C_{[k]2}$, $C_{[k]3}$,...,$C_{[k]\mathrm{k}}$, $C_{[k]1}$ again. By this implication, robots a and b would be back to indexes $k-1$ and k respectively (Fig. 26).

$$C_{[k]1} = C_{[k]1} + k \text{ rotations.} \tag{1}$$

Inductive Step On this base, we have to prove in the cycle with degree $k+1$. Consider robots a and b be located at indexes k and $k+1$ respectively. The initial configuration of the cycle is with degree $k+1$ is $C_{[k+1]1}$. If the cycle has been rotated $k+1$ time, the configuration would be changed to $C_{[k+1]2}$, $C_{[k+1]3}$,...,$C_{[k+1]\mathrm{k+1}}$, $C_{[k+1]1}$ again, robots a and b will be back to indexes k and $k+1$ respectively.

Based on the hypothesis, after k rotations, as shown in Fig. 27, the initial configuration $C_{[k]1}$ of the cycle with degree k will be restored while a and b will be at index k. One more rotation $(k+1)$ would be enough to restore the initial configuration $C_{[k+1]1}$.

$$C_{[k+1]1} = C_{[k]1} + 1 \text{ rotation.} \tag{2}$$

[6] Details of Multipush operation is in Appendix A.2 of Supplementary material.

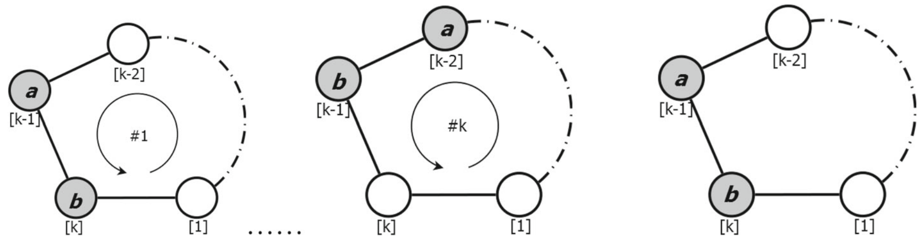


Fig. 26 Lemma 3, hypothesis

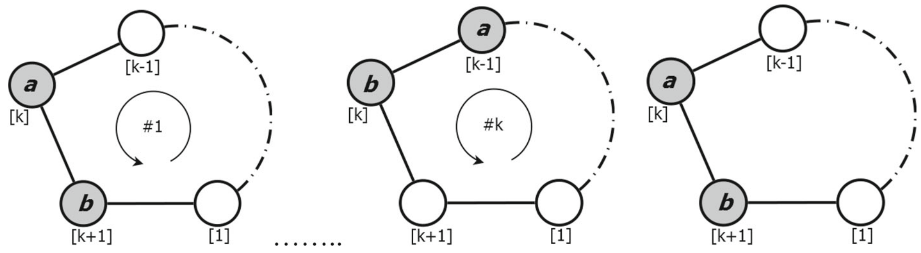


Fig. 27 Lemma 3, induction

From (2) in (1)

$$C_{[k+1]1} = C_{[k]1} + k \text{ rotations} + 1 \text{ rotation}.$$

Hence, robots a and b will reach indexes k and $k+1$ respectively and the initial configuration $C_{[k+1]1}$ of the cycle with degree $k+1$ will be restored (Fig. 27).

Lemma 4 *Prepare cycle operation*[7] *will always evacuate vertex u, which is a join vertex neighborhood that does not imply to the cycle C without affecting swapping robots adjacency if the problem instance satisfies the solvability condition.*

$\exists v, u : v$ is join vertex of C, $u \in \text{adj}\,[v] \wedge u \notin C \wedge \max\,(\text{bridge length}) \leq m \rightarrow$ prepare_cycle(C).

Proof Lets assume that there is an outer neighbor of the cycle C join vertex v; u, and the bridge is only the edge connecting v by u (Fig. 28). Further assuming that the instance satisfies solvability condition. There should be at least one unoccupied nodes in the graph, prepare cycle operation will push the robot $A^{-1}(u)$ to the nearest one blindly. Let w is the nearest unoccupied vertex to u there are two case for w position regarding the cycle C:

- $w \notin C$.

If the graph is connected, and there is an occupied vertex u as well as one unoccupied vertex w which are connected by path p. This path is in the worst case, fully occupied. In such case, prepare cycle will make blind push for robot $A^{-1}(u)$ to w successfully. From Lemma 1 push operation will always push robot $A^{-1}(u)$ to the free vertex w if the path to it is free or occupied by non-finished robots, prepare cycle will send empty finished robot set. Hence, prepare cycle will always evacuate vertex u in this case (Fig. 29).

- $w \in C$.

[7] Details of prepare cycle operation is in Appendix A.1 of Supplementary material.

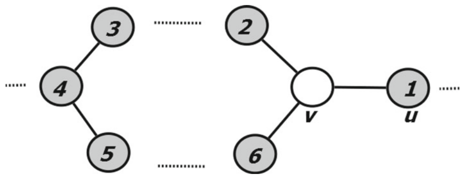


Fig. 28 Lemma 4, prepare cycle will evacuate vertex u

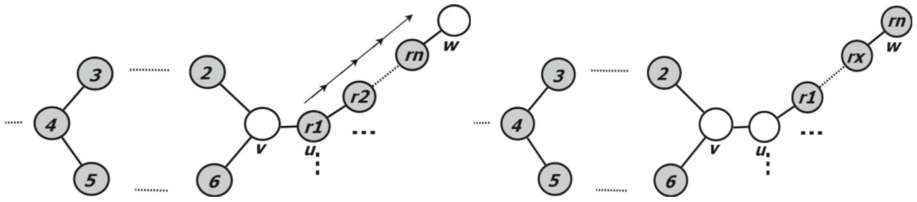


Fig. 29 Lemma 4 Case 1

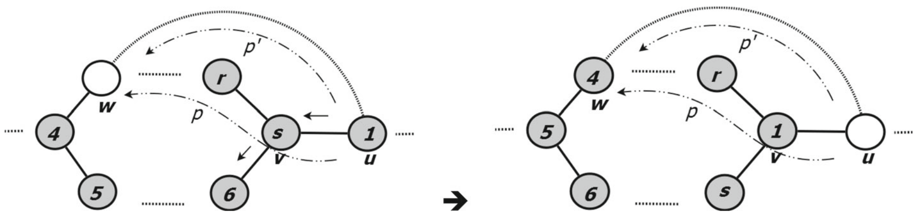


Fig. 30 Lemma 4, Case 2 without protecting swapping robots adjacency

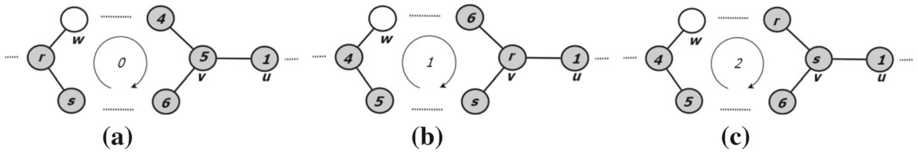


Fig. 31 Lemma 4, main three cases should be considered before blind pushing robot A^{-1}(u) inside the cycle

In this case, there are two subcases. However, both are treated in the same way. If w is inside the cycle, that would mean there is definitely a path p from u then v to w. Let p' be the other path connecting u to w and the swapping robots on the join node. prepare cycle operation will always consider path p, even if there is shorter one. The operation will first evaluate the location of swapping robots and makes the number of rotations enough to alienate them far of the join node. Without this rotation, evacuating u will result in Fig. 30 case. This is in conflict with the definition of swapping robots positions. Actually, the number of rotations required to alienate swapping robots from v varies between $\{0, 1, 2\}$ only. Figure 31a shows when the cycle requires zero rotation, Fig. 31b shows when the cycle requires one rotation and Fig. 31c shows when the cycle require two rotations. After that, prepare cycle will push $A^{-1}(u)$ to vertex w blindly. The success of this operation is always guaranteed based on Lemma 1. That is due to the fact that prepare cycle will send an empty finished-robot-set to Push operation, then, prepare cycle will always evacuate vertex u in this case (Fig. 32).

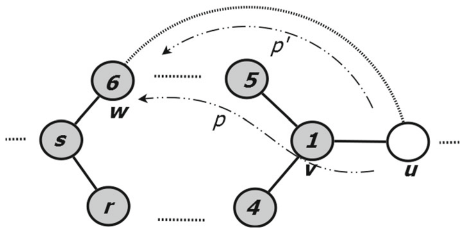


Fig. 32 Lemma 4 Case 2

Lemma 5 *In compute swapping cycle operation*[8] *if the instance containing a cycle satisfies the solvability condition—for the bridge length bl, there should be at least m free nodes where m ≥ bl—then, there is at least one cycle able to receive swapping robots.*

$$\max\text{ (bridge length)} \leq m \rightarrow \text{compute swapping cycle}$$

Proof The proof of this lemma is direct, as shown in from the proof of the feasibility test (Sect. 5). There is always a way to pass a robot to any position in the graph if the graph satisfies the solvability condition. Hence, compute swapping cycle selects the best cycle fit all opponent robots and the swapping robots.

Proposition 2 *If prepare cycle operation evacuates vertex u, then, cc-case operation (Algorithm 4) always swaps swapping robots positions and returns other out-of-position robots to their previous positions.*

$$\text{prepare cycle} \rightarrow \text{cc-case}$$

Proof Based on Lemma 4, prepare cycle would always evacuates vertex u which is a join vertex neighborhood that does not imply to the cycle if the problem instance passes the feasibility test. Hence, we have to prove that cc-case always (1) restores the initial configuration except swapping robots positions, and (2) exchanges swapping robots' positions.

1. Let C_1 be the graph configuration before applying prepare cycle, C_2 be the graph configuration resulting after applying prepare cycle on the graph and m be the sequence of moves generated by prepare cycle to transfer C_1 to C_2. The robots affected by other operations during rotation (i.e., swapping robots) would be excluded from Lemma 3 result. Indeed, we would be concerned in this part on the robots moved by m and full rotation operation only. cc-case uses prepare cycle operation to clear the vertex u which generates m.

$$C_2 = C_1|m \tag{3}$$

Then, cc-case operation would rotate the cycle for a number of rotations equal to its degree Hence the configuration C_2 would be restored definitely (Lemma 3).

$$C_2 = C_2 + \text{full rotation} \tag{4}$$

But, cc-case would execute push and multipush operations on the swapping robots r and s during rotation.

$$C_{2/s} \leftrightarrow_r = C_2 + \text{full rotation} + \text{multipush}_{s,r} + \text{push}_s \tag{5}$$

From (5) in (3) we conclude that cc-case would always be able to restore C_1—except the swapping robots s and r configuration—by full rotating the cycle and reversing m.

$$(C_2 + \text{full rotation} + \text{multipush} + \text{push})\ |m^{-1} = C_{1/s} \leftrightarrow_r$$

[8] Details of compute swapping cycle operation is in Appendix A.3 of Supplementary material.

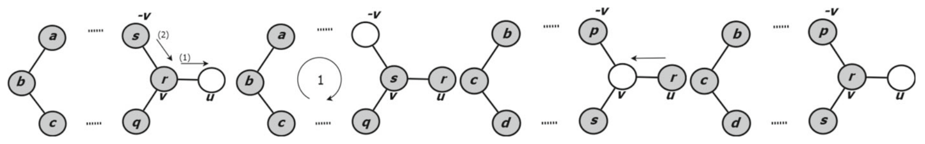


Fig. 33 Proposition 2, exchange swapping robots

2. cc-case would rotate the cycle partially until the swapping robots reach the join vertex v.

Once, in desired configuration C_i, cc-case multipushes the fronted robot r toward u successfully in time t_1 (Lemma 4). The follower robot s would be sent towards the vertex v—which was occupied in t_1 by the fronted robot r. Upon successfully completing the procedure in $t_{2,}$ it would leave unoccupied node; $-v$. cc-case would rotate the cycle one more rotation which will make the unoccupied vertex v. In this way, it can push the fronted robot r back to vertex v (Fig. 33) and the rotates the cycle a remaining number of rotations. At the end, and based on the first part of this proof, all robots will return to their positions. Furthermore, the swapping robot would have exchanged their positions.

Proposition 3 *If compute swapping cycle operation finds a swapping cycle, then, bb-case operation (Algorithm 6) always swaps the swapping robots positions and return all out-of-position robots to their previous positions.*

$$\text{compute swapping cycle} \rightarrow \text{bb-case}$$

Proof Based on Lemma 5, compute swapping cycle would always find at least one desired cycle C. This cycle would be able to receive swapping robots if the instance containing a cycle satisfies the solvability condition. Hence, we would have to prove that bb-case always (1) multipushes the swapping robots toward the swapping cycle blindly, (2) while exchanging swapping robots' positions and (3) simultaneously restoring initial configuration except swapping robots positions.

1. Let C_1 be the graph configuration. Since compute swapping cycle guarantees to find a swapping cycle SC (Lemma 5), bb-case will multipush the swapping robots s and r at the bridge toward the join vertex v of SC blindly. As clearly evident, multipush will not consider any finished robot in the swapping robots' way and it will push them to SC as well. Based on Lemma 2 and Proposition 1, the swapping cycle SC would always be able to receive the swapping robots and all the robots in their way. This proves that the multipush always succeeds in compute swapping cycle. C_2 is the resulted graph configuration and m is the sequence of moves transfers C_1 to C_2.

$$C_2 = C_1|m \tag{6}$$

Once the swapping robots reach the join vertex v, bb-case would push the fronted robot r to the vertex next to v in clockwise direction; $+v$ and the follower robot s to v.

Let C_3 be the resultant graph configuration after pushing swapping robots inside the swapping cycle SC. bb-case will rotate the swapping cycle SC for a number of rotations that is equal to its degree minus one. When performed, the fronted robot r will be at the join vertex v and the follower robot s would be at the vertex previous to the join node; $-v$. Since multipush would insert all robots in the bridge in the swapping robots way to the swapping cycle, the neighbor vertex of v on the bridge u would be unoccupied. In this way bb-case would push the fronted robots r to u and execute the remaining rotation which would restore

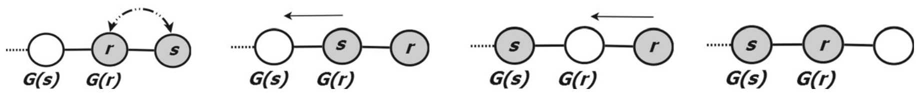


Fig. 34 Proposition 4, Stage 1 of resolving technique

all robots in the cycle into configuration C_3 and set the follower robot to v. At this point, configuration C_3 would have been restored except the swapping robots.

$$C_{2/s} \leftrightarrow_r = C_3 + \text{full rotation} + \text{push}_r \tag{7}$$

2. When the swapping robots would be at v and u, their positions would be exchanged. bb-case would reverse the sequence of moves generated by multipush while exchanging swapping robots moves.

From (7) in (6) we conclude that bb-case always multipushes the swapping robots toward the swapping cycle blindly. It would exchange swapping robots' positions and restore initial configuration except swapping robots positions.

$$\left(C_3 + \text{full rotation} + \text{push}_r\right) |m^{-1} = C_{1/s} \leftrightarrow_r$$

Proposition 4 *After execution of Spin operation (Algorithm 3), the position of the swapper robot will be exchanged with the blocking robot and all robots will be at their positions except the swapped robot in the worst case.*

Proof Spin operation evaluates the position of the swapping robots,

- If the swapping robots are inside a cycle, cc-case would always swap robots positions and would return other out-of-position robots to their previous positions (Proposition 2).
- If the swapping robots are on the bridge, bb-case would always swap the robots positions and would return all out-of-position robots to their previous positions (Proposition 3).

The last part to prove, is that at the end of the second and third case, the swapper robot s would be at the swapped robot r position. This may be the goal of the swapped robot r (Fig. 34). In such case, the Resolve operation would try to push robot s one more step in its shortest path. In this way, it would resolve robot r by pushing it back to its goal, if not, this would be because the swapper robots s faces another finished robot in its shortest path, hence, robot s needs to finish its execution and the swapped robot will be resolved in stage 2 in the next algorithm. Hence, the swapped robot is out-of-position at the end of Spin operation in the worst case.

Proposition 5 *After execution of one iteration in Push And Spin operation (Algorithm 1), robot r will have been solved and all robots will be at their positions except the swapped robot in the worst case.*

Proof Push And Spin algorithm would find the shortest path connecting the current position of robot r with its goal position based on the reachability. This would be independent of the path availability which would be guaranteed in the connected graph. Consequently, for each step, the robot would advance to next vertex in the shortest path toward its goal position. This next vertex may be:

- Unoccupied, the robot would simply advance to it by push operation successfully (Lemma 1).

- Occupied by non-finished robot s, the robot would be advanced to it by push operation and the occupying robot s would be pushed away successfully (Lemma 1).
- Occupied by finished robot s, after execution of Spin operation, the position of the swapper robot r would be exchanged with the blocking robot s and all robots would be at their positions except swapped robot in the worst case (Proposition 4).

If all cases would succeed, the robots would be assumed to have reached its goal at the end of its iteration in Push And Spin algorithm. In this way, all robots would be at their positions except the swapped robot in the worst case.

Theorem 1 *Push and Spin algorithm is complete for any graph contains a cycle and a number of free nodes more than or equals maximum bridge length.*

Proof The idea behind the proof is:

1. In each iteration of Push and Spin algorithm (Algorithm 1) in which a robot is selected that is not in $\mathcal{U}$ (line 8), a robot would be added to $\mathcal{U}$.
2. In case a finished robot s has been moved off its goal location by Spin operation, then its current location would be adjacent to its goal location. Resolve operation would try to make one push for swapper robot r in order to evocate the goal of robot s. When that would happen, it would push robot s back to its goal. If it would fail, it would schedule robot s in ResolveStack to be re-executed after swapper robot r reaches its goal.
3. After a finite number of iterations, all robots in ResolveStack would have been processed, restoring out-of-position agents in $\mathcal{U}$ to their goal location.

To prove the first point, for non-finished robot r, after execution of the iteration of robot r in Push and Spin algorithm (line 12), robot r would have been solved and all robots would be at their positions except the swapped robot in the worst case (Proposition 4), robot r will be added to $\mathcal{U}$.

To prove the second and third point, Push And Spin would execute push algorithm to push robot s in Fig. 35a. In its shortest path, robot s would face a finished robot r in its shortest path. Spin algorithm will exchange their positions successfully (Proposition 4) (Fig. 35b). Furthermore, Spin algorithm would try to resolve robot r by pushing robot s one step in its shortest path and it would fail since robot s would be facing another finished robot in its shortest path, hence. This would make the Spin operator to exit with stacked robot; $\{r\}$ (Fig. 35c). Push And Spin algorithm would continue to plan robot s's path. It would execute push algorithm to push robot s in its shortest path making robot s face finished robot q in its shortest path. Spin algorithm would have exchanged their positions successfully (Fig. 35d). After this, Spin operator would try to resolve robot q by pushing robot s one step in its shortest path. This attempt would fail since robot s would be facing another finished robot in its shortest path making the Spin operator to exit with stacked robot; $\{q, r\}$ (Fig. 35e).

Push And Spin algorithm would continue to plan the path of robot s. For meeting that objective, it would execute push algorithm to push robot s in its shortest path. Robot s would be facing a finished robot t in its shortest path, Resultantly, Spin algorithm would have exchanged their positions successfully (Fig. 35f). Spin algorithm would try to resolve robot t by pushing robot s one step in its shortest path (Fig. 35g) and it will succeed in pushing robot s one step and resolving t (Fig. 35h). In this manner, Spin operator would exit without stacked robot. At the end of robot s execute, it would be added to finished robots set. Push And Spin algorithm would re-execute stacked robots in last-in-first-out manner since the latest swapped robot q will be able to move to the last evocated vertex before robot t goal (Fig. 35i). This would give us the desired unoccupied vertex which is the goal of earliest stacked robot r (Fig. 35j).

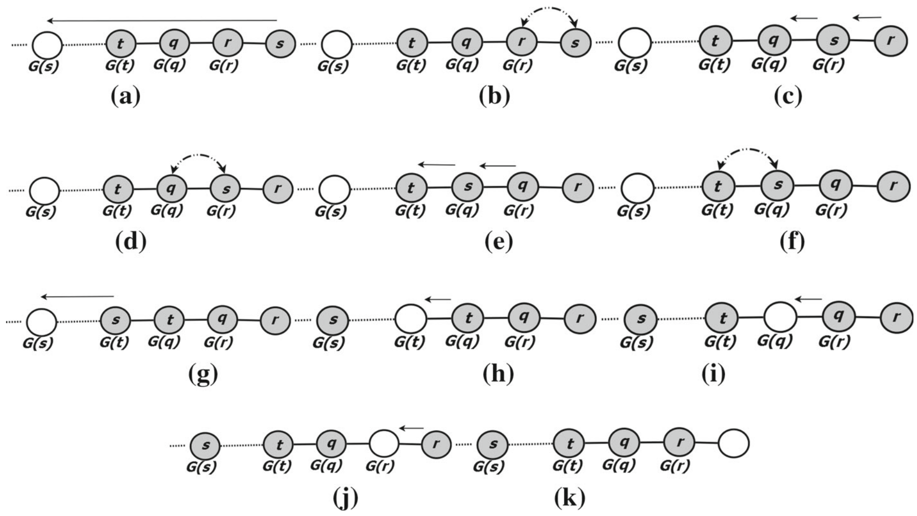


Fig. 35 Thereom 1, Stage 2 of resolving technique

6.2 Runtime analysis

Let k denote the number of robots in the instance and n be the number of vertices in the roadmap. In order to solve an instance, each robot needs to be sent to its goal position by Push and Spin algorithm (lines 13–23) to solve the robot or to resolve the robot, it will take at most k. In the iteration of the robot, Push and Spin algorithm computes the shortest path for the robot to reach its goal position. The length of this path (or any simple path in the graph) is upper bounded by n. For each step along this shortest path, Push and Spin algorithm operation will do a Push or Spin algorithm. Out of these algorithms, the runtime of Spin algorithm is dominant, which simplifies the following equation:

$$t_{\text{Push and Spin}} = O(k \cdot n \cdot t_{\text{Spin}})$$

Spin algorithm executes either cc-case algorithm or compute swapping cycle, bb-case, and resolve algorithms resulting the following equation:

$$t_{\text{Spin}} = O\left(t_{\text{cc-case}} + t_{\text{compute swapping cycle}} + t_{\text{bb-case}} + t_{\text{resolve}}\right)$$

- cc-case takes O(n^2), the runtime of prepare cycle operation is dominant, it pushes a robot on u to clear this will take O(n^2) in the worst time
- compute swapping cycle operation takes O($c.n^2$) because it evaluates the availability of each cycle O(C) and calculates the shortest path to it O($|n|+|e|$) = O(n^2).
- bb-case algorithm takes O(n^3), the runtime of multipush operation is dominant, it takes O(n^2) time to push the robot toward its goal, this push will be repeated along the robot path which is upper bounded by n.
- resolve algorithm takes O(n^2) time to push the swapping robot and swapper robot one step. Hence, the time complexity of Spin algorithm is:

$$t_{\text{Spin}} = O\left(n^3\right) \rightarrow t_{\text{Push and Spin}} = O(k \cdot n^4)$$

6.3 Solution quality analysis

For the same reasons as above, the Spin algorithm is a dominant factor in the output of the Push and Spin algorithm. This yields the following expression for the number of moves that the Solve algorithm outputs:

$$l_{\text{Push and Spin}} = O(k \cdot n \cdot l\text{Spin})$$

as described in time analysis, Spin algorithm executes either cc-case algorithm or compute swapping cycle, bb-case, and resolve algorithms resulting the following equation:

$$l_{\text{Spin}} = O\left(l_{\text{cc-case}} + l_{\text{compute swapping cycle}} + l_{\text{bb-case}} + l_{\text{resolve}}\right)$$

- cc-case takes O(n^2), the output of execute spin operation is dominant and it moves all robots in the cycle of n vertices O(n) moves for a number of rotations. This in its worst case would equal the cycle size O(n). In addition, execute spin operation produces O(n) makespan.
- Compute swapping cycle operation doesn't take any action, hence, no moves will be produced.
- bb-case operation takes O(n^2), the output of multipush operation is dominant, it takes O(n) moves in each push of the robot one step toward its goal to push the opponent robots away, this output will be produced for each step in the path to the goal produces O(n) moves.
- resolve operation produces O(n) moves to push the swapping robot and swapper robot one step which require push all opponent robots away O(n). Hence, the path quality of Spin operation is:

$$l_{\text{Spin}} = O\left(n^2\right) \rightarrow l_{\text{Push and Spin}} = O(k \cdot n^3)$$

7 Post-processing and heuristic

The goal of the research problem that we tackle is to find the complete solutions for MPPs. The theoretical analyses already performed demonstrate the completeness of the algorithm. However, the Push and Spin algorithm does not guarantee the optimality of the solution. It is noteworthy here that we did not improve the performance metrics (i.e., using heuristics, smooth methods, etc.) in the previous sections, although various methods exist to optimize the completed solution. In this section, two optional methods are used for this purpose.

7.1 Smooth operation

The Smooth operation is a component invoked on the complete solution at the end of the algorithm. It reduces the generated redundant moves. In this work, we have implemented the same Smooth operation introduced in the Push and Swap algorithm; however, because our solution contains simultaneous moves, the operation has been modified accordingly. Smooth operation (Algorithm 8) takes the list of solutions Π, which contains a list of lists of moves. Each list is indexed with its time-step in Π; hence, the sequential moves represent a list of one move, and the simultaneous moves represent a list of all moves executed at the same time-step. Smooth operation iterates over Π in a reverse manner.[9] If the index contains simultaneous

[9] The description here is written as in the Push and Swap publication because we have implemented the same smooth operation of the Push and Swap algorithm.

moves (list of length more than one), the operation will skip it (line 6), because there will be no redundant moves generated by executing the Spin operation. If the index contains sequential moves (list of length one), for each action π, the actions after π in the reversed sequence are checked for an occurrence of the final vertex v in π. If such an action π' exists (line 9), and π and π' are executed by the same robot r (line 10), then all actions executed by r between π and π' (including π) can be removed from Π (lines 11–13). Additionally, π' can be adjusted to end at vertex v (line 14). The smoothing process continues until the entire solution sequence has been iterated through, and the iteration begins anew. Smooth continues to iterate over Π in reverse order until no paths are removed.

Algorithm.8 `smooth` (Π)

```
1.  removed = TRUE
2.  while removed == TRUE
3.      removed = FALSE
4.      for all $\pi \in \Pi$.reverse ()
5.          if $\pi$.size > 1 then
6.              continue
7.          $r$ = robot($\pi$)
8.          $v$ = last vertex in $\pi$
9.          $\pi' \leftarrow$ next path in $\Pi$.reverse () containing $v$
10.         if $\pi' \neq \emptyset$ and $r$ == robot($\pi'$) then
11.             for all $\pi'' \in \Pi$ ($\pi, \pi'$)
12.                 if robot($\pi''$) == $r$ then
13.                     Remove ($\pi''$) from $\Pi$
14.             Remove portion of $\pi'$ after $v$
15.             removed = TRUE
16.  return $\Pi$
```

7.2 Heuristic search

After designing a complete polynomial algorithm, we found that within the complete solution, alternative complete decisions may appear that may improve the performance of the algorithm in addition to its completeness. Hence, we designed a new version of the complete Push and Spin algorithm (improved Push and Spin algorithm) that inherits the completeness of the original algorithm but with better performance.

One of the major factors affecting multi-robot path-planning algorithms is the total path length. It is calculated as the total number of moves that transform the initial configuration of robots into the goal configuration. Hence, in the complete algorithm, finding the shortest path as an alternative to the connected path will provide a better solution in terms of reducing the total path length. This has a direct effect on the traversal times of robots.

Furthermore, in general in the Push and Spin algorithm, as described earlier, the Spin operation represents the dominant factor in both execution time and total path length metrics. Hence, these metrics can be improved by reducing the number of situations that lead to the need to invoke the Spin operation.

The Spin operation will be called when robot a finds a (blocking) planned-robot n (Fig. 36) in its connected path p. Hence, one way to improve the path and eliminate Spin operation calls is to try to find another path $p*$ such that $p*$ contains the minimum number of planned-robots. However, if there is no other path, we must maintain completeness by using path p.

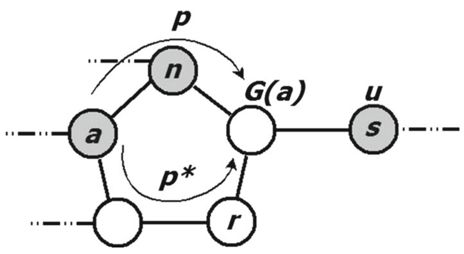


Fig. 36 Alternative path computation

These ideas have been applied to the improved Push and Spin algorithm by replacing the *connected-path* algorithm in Algorithms 1 and 2 with the *best-path* algorithm (Algorithm 9). In this algorithm, we introduced an evaluation function that evaluates the available paths according to their lengths and the availabilities of planned-robots. Furthermore, the execution time factor will be improved by reducing Spin cases, but may suffer by evaluating different available paths.

Algorithm.9 best-path (r , *path-set*, G, $\mathcal{U}$, w_1, w_2)

```
1- min = |G|
2- index = -1
3- For each path $p_i \in$ path-set :
        a.  best-path = $w_1 . |p_i| + w_2 .$ number of planned-robot ($p_i$)
        b. If min > best-path:
                i. min = best-path
        ii. index= $i$
4- return index
```

8 Results and discussions

The research problem that we aimed to tackle is to find the complete solution for the multi-robot path planning problem. The first contribution is to recognize the solvable instances of the problem by our solvability test; the theoretical analysis is already provided to show the validity of this test. The second contribution is to solve this problem completely, in polynomial time, by the Push and Spin algorithm. Once the problem is solved, we found within the complete solution different decisions that may improve the performance side of the complete algorithm. Hence, the third contribution is to improve the performance side by selecting the best path from the set of complete paths. The improved version of our algorithm is referred to as the improved Push and Spin algorithm.

8.1 Experimental framework

The experiments were run on the evaluation environment implemented by us [21].[10] We have compared the performance of Push and Spin (PASp) with that of MAPP tractable multi-agent path planning algorithm for undirected graphs, Push and Swap (PAS), Push and Rotate (PAR),

[10] The link of the tools: https://pushandspin.wordpress.com/evaluation-tools/.

Bibox algorithms[11] and the improved Push and Spin (PASp+) with weighted path heuristic and smooth operation. The Push and Spin algorithm has been implemented in C++.

8.1.1 The evaluation factors

- The total number of moves

Push and Swap, Push and Rotate, Bibox and MAPP algorithms work on MPP problems against the Push and Spin algorithm works on MPPr problem. For the purpose of removing any biasedness, we unified the metrics here and calculated the path length as the total number of sequential moves. Therefore, the single parallel rotation of n robots here is calculated as n moves.

- The CPU time

The execution time is calculated as the real time between the algorithm start time to the end time. Bibox algorithm contains handle decomposition preprocess. Also in its current implementation, it assume that the cycles are defined earlier i.e., before the execution start time. Therefore we have ignored the execution time of the preprocess of both Push and Spin algorithm which is extract cycle time and Push and Rotate algorithm which is problem decomposition time.

- The makespan

Makespan is the number of time steps required to get all robots to their destination. In Push and Spin algorithm, the makespan always equal to the path length except in the moves generated by single rotation. In that case, they will be calculated as one move since it takes single timestep.

8.1.2 Type of instances

Furthermore, we have tested the algorithms on four types of instances;

- Large public benchmark instances

Sturtevant [22] describes a new repository that has been placed online[12] to improve the evaluation of grid-based problems. The repository allows researchers to use the same problems and test sets thereby increasing the reproducibility of published results. In the repository, two main sets are described; Commercial Game Benchmarks and Artificial Benchmarks. Each map has been coded in a way to be able to parse. The maps are attached with set of scenarios describing robots locations on the map.

- Random bi-connected instances

Surynek's code[13] in the current implementation doesn't include handles decomposition preprocess, Instead, it generates random bi-connected graphs according to three parameters; h the number of handles, C_0 the size of the initial cycle, and l the maximum handle length (Fig. 37). In addition, one can choose the number of empty vertices, with a default value of 2. To be able to compare the algorithms with Bibox, we have used the same generator to

[11] Our source code is in: https://pushandspin.wordpress.com/source-code/.

[12] http://movingai.com/benchmarks/.

[13] We used the code available at http://ktiml.mff.cuni.cz/˜surynek/research/icra2009/.

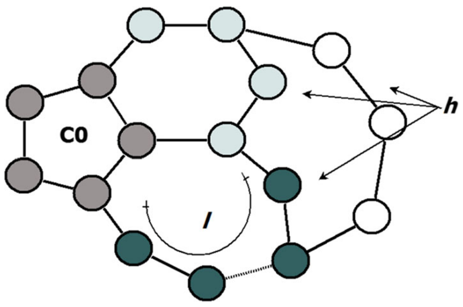


Fig. 37 Initial cycle (C0), maximum handle length (l) and number off handles (h) in Surynek graphs

generate the same bi-connected instance for evaluation. Since there are two sets of parameter, our evaluation tool generates two sets of scenarios; fixed graph parameter variable number of robots (FG-VR) and fixed number of robots variable graph parameter (FR-VG). Surynek code generates scenarios of each instance by filling the generated graph with the given number of robots randomly.

- Random instances

The class we argue Push and Spin would the only one able to solve, is the class of graphs containing maximum bridge length less than or equal free nodes. This can be generalized to the fact that the Push and Spin is able to solve grid and bi-connected graphs with only one free node too. To validate our argument, we have designed a random generator which generates this type of graphs, the procedures for generating random graphs described as follow:

```
1. Parameter $n$: the number of vertices, $bl$: the maximum bridge length.
2. $i=0$
3. $G=\phi$
4. while $i<n$
    a. Select random $b \in [1, bl]$
    b. Create bridge with length $b$
    c. Select random $c_1 \in [3, n-b]$, $c_2 \in [3, n-b-c_1]$
    d. Create two non trivial biconnected components with size $c_1$ and $c_2$
    e. $G' \leftarrow$ Append the two cycles by the bridge in random connection points.
    f. Append the $G'$ by $G$
    g. $i=i+b+c_1+c_2$
```

Once the graph is created, the scenario is generated by calling the procedure of generating random scenario with parameter (n, $n-|R|$[14]). Hence, the generated scenarios will be; fixed graph size variable bridge length and robot number (FG-VR) and fixed bridge length and robot number for variable graph size (FR-VG) scenarios.

- Random grid instances

We have used the same Surynek generator to generate grid instances of $n \times n$ size with constant value of the parameter $(h, C_0, l) = ((n-1)^2 - 1, 4, 4)$. Hence, the generated scenarios will be; fixed grid size variable number of robots (FG-VR) and fixed number of robots variable grid size (FR-VG).

[14] Because the number of free nodes should be equal to the longest bridge in this instances.

8.2 Experimental results

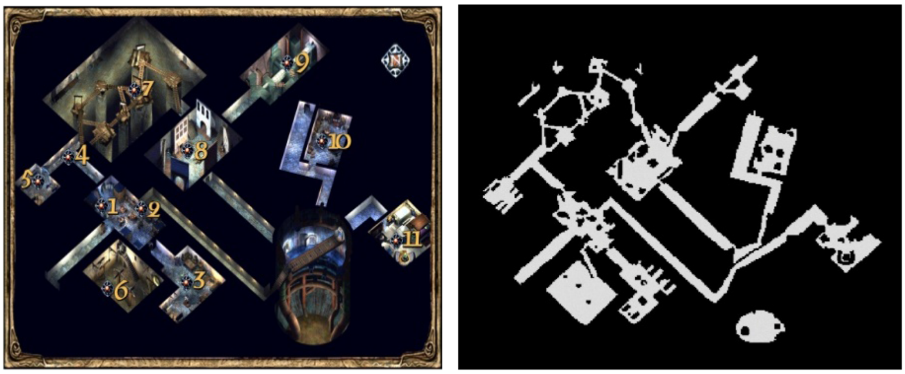
The discussion part of the experimental results will be limited to:

- Investigation of the completeness of the Push and Spin algorithm for wider use.
- Annotation of the theoretical worst-case bounds with the performance of the algorithm on different instance classes. Often in real world, algorithms perform significantly better than their worst-case time complexity would indicate.
- Test the heuristics and confirm statistically that they are capable of improving the solution length, with the execution being longer when using the heuristic [23].
- Show the degree of improvement in total path length of Push and Spin algorithm yields.
- Provide an insight into the performance of the algorithm for the purpose of allowing it to be compared with other approaches. For this purpose, instances from a public benchmark set [22] have been included, since these have been used to test other algorithms [13].

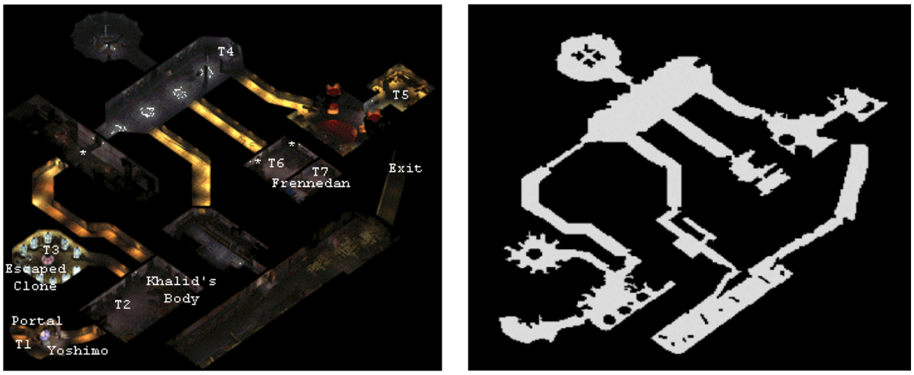
We can theoretically predict the performance differences between the Push and Spin, MAPP and Bibox algorithms by considering the differences between their techniques. MAPP imposes alternative paths for each vertex in the robots' shortest paths; when two planned-robots collide, MAPP would switch the robot to the alternative-cached path. This would require a high number of unoccupied vertices resulting in better performance in such instances. Bibox solves the problem incrementally; it solves the handles one by one and removes them from the search space. Additionally, it always solves the problem in its worst scenario by filling the free vertices with dummy robots which are cleared from solutions after obtaining final result. As a result in this algorithm, the congestion rate, unlike for other algorithms, will not affect the performance of the Bibox algorithm. Hence, it can be seen that Bibox would perform better when the graph size is low or the graph topology contains higher handle lengths [10].[15]

The Push and Swap as well as Push and Rotate algorithms are too close to the Push and Spin algorithm. It should be noted that irrespective of the experiment setting, these algorithms would produce approximately similar performance in their original implementations (without heuristics and smooth operations). The results would be a bit better for the Push and Rotate algorithm since it would use robots ordering heuristics in its original implementation, which reduces the number of swapping cases. The main difference is observed when two planned-robots collide; both the Push and Swap and Push and Rotate algorithms will look for the nearest swapping vertex (a vertex of degree three or more) and clear the path to it. This operation would require O(n^2) moves. On the other hand, when the same case occurs, the Push and Spin algorithm will look for the nearest swapping cycle (a cycle with empty vertices equal to the facing robots and swapping robots) and clear the path to it by pushing all facing robots toward it. This operation would require O(n^2) moves too. Therefore, the performance may differ when the graph contains closer-full cycles. This demonstrates the ability of the Push and Spin algorithm to work when the cycle is full. For the same scenario, both the Push and Swap and Push and Rotate algorithms would try to clear the neighbors of the entrance vertex of the cycle (the entrance vertex will definitely be a vertex of degree three), and they will fail if the cycle is full. To handle such a performance difference, we apply the Push and Spin, Push and Swap, and Push and Rotate algorithms in different evaluation models. We expect differences in the Push and Spin, Push and Swap, and Push and Rotate performance when the graph topology changes, and also, when the congestion rate increases.

[15] Our evaluation tools allow the user to set the handle length as well as the graph size and the congestion rate.



(a)



(b)

Fig. 38 Two maps selected from Baldurs Gate II. **a** AR0306SR map of Baldurs Gate II (left), its coded map (right). **b** AR0603SR map of Baldurs Gate II (left), its coded map (right)

8.2.1 Large public benchmark instances

We have tested two Commercial Game maps from Baldurs Gate II of size 512X512, AR0307SR (Fig. 38a) and AR0603SR (Fig. 38b). The parser will read the coded maps of AR0306SR and AR0603SR into graphs where every element in the map is converted into a vertex if it not obstacles (land). This would result in graphs that are characterized by a large set of vertices, many of which are unoccupied.

Since the graph topology here is random, we cannot predict a difference in the behavior of the Push and Spin, Push and Swap, and Push and Rotate algorithms. The Push and Spin, and Push and Swap algorithms would produce competitive performance, and the Push and Rotate algorithm perform better than both of those in terms of total-path-length and execution time. This better performance is due to the fact that it uses a robot ordering technique. MAPP would perform better in terms of the resulting total-path-length when the number of robots is low. However, it would perform worse when the congestion increases since it imposes a high number of alternative-free paths. On the other hand, MAPP uses cached-alternative paths, and hence, the action that would be taken when the collision occurs is to switch to an alternative path. This operation would require less execution time compared to other algorithms.

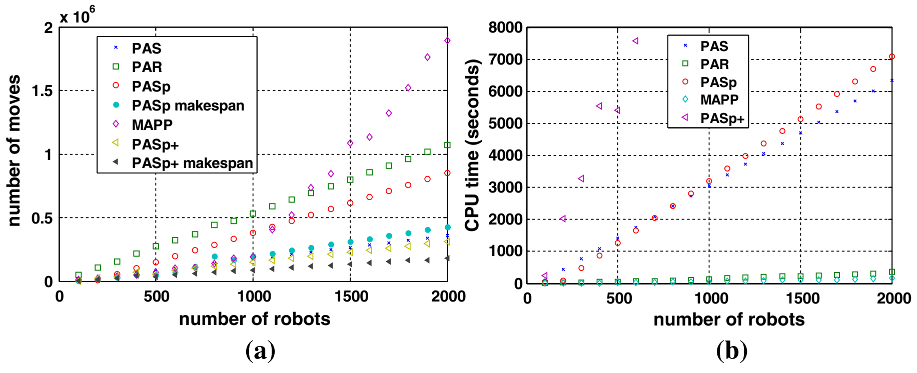


Fig. 39 Comparison on map AR0603SR of PASp, PAS, PAR and MAPP, **a** number of moves, **b** CPU time

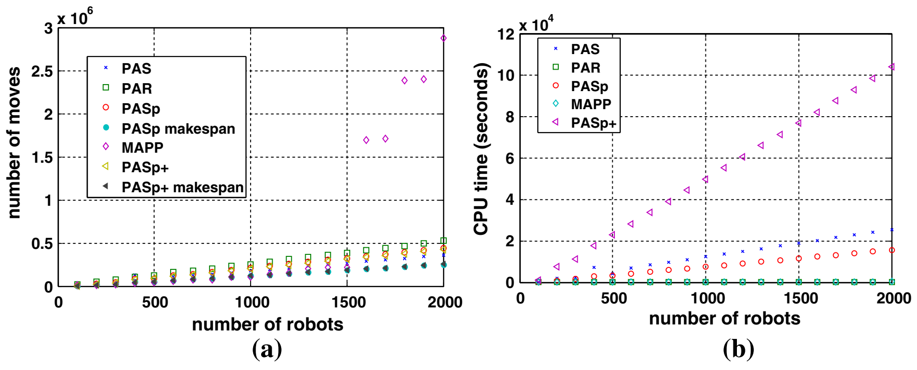


Fig. 40 Comparison on map AR0307SR of PASp, PAS, PAR and MAPP, **a** number of moves, **b** CPU time

Figures 39 and 40 show the number of moves produced by Push and Spin, Push and Swap, Push and Rotate, MAPP, and Push and Swap (no Bibox, as the maps were not bi-connected). These also show the CPU times on AR0603SR and AR0307SR maps respectively. The figures show that Push and Swap produces very efficient plans due to lower number of moves. At the same time, Push and Spin and Push and Rotate produce bit higher number of moves because they impose new techniques to cover wider solvable area which affects their plan quality. In addition, Push and Spin produces too low makespan. This variance agrees with the theoretical computation of the upper bound of the number of moves for Push and Swap, Push and Rotate and Push and Spin algorithms. MAPP requires many more moves to find a solution, and therefore we did not try to include it in our further experiments. On the other hand, the heuristics and post-process show significant improvement since PASp+ provides the lowest number of moves among all others. Consequently, the makespan of PASp+ shows, also, more improvement.

On the contrary, Push and Swap is the slowest for this type of instances. Push and Spin takes higher time to find the solution than Push and Rotate and MAPP. Push and Rotate uses robot ordering technique which beside making the algorithm complete also improves the solution in term of time and number of moves. MAPP is the fastest in this aspect because when collision occurs, it switches the robot to the alternative path. In addition, even though the heuristic and post-process improve the path length in PASp+ significantly, they consume

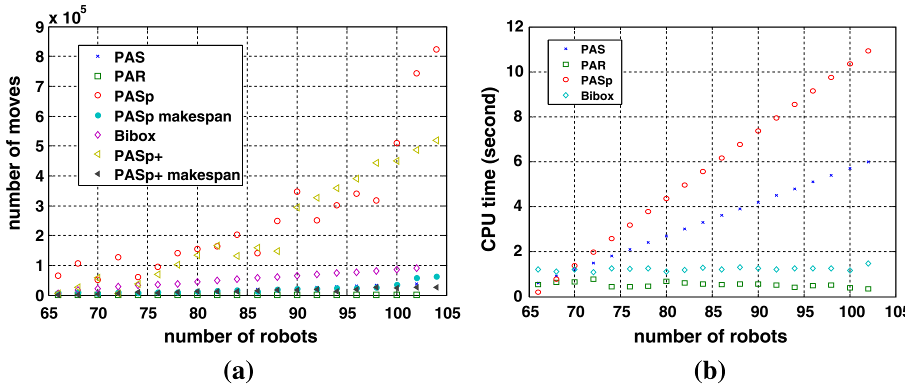


Fig. 41 Comparison on random instances with 20 handles, initial cycle 5, and maximum handle size 10, **a** number of moves, **b** CPU time

more time to evaluate the paths and smooth it, this is one of the drawback of the nature of optimization techniques [23]. Moreover, by taking the highest registered moves of Push and Spin, 8.2×10^5 in Fig. 39a, it would be clear to note that it is significantly lower than the upper bound of the number of moves for that scenario ($2000 \times 578{,}73^3$).

8.2.2 Random bi-connected graph

As described earlier, Surynek generator generates two set of experiments each of them vary for a specific parameter. Figure 41 shows FG-VR scenarios, Fig. 41a shows the number of moves produced by Push and Spin, Push and Swap, Push and Rotate and Bibox and Fig. 41b shows their CPU time. All experiments are performed on Bi-connected graph of 105 nodes with 20 handle, 5 nodes for initial cycle and 10 nodes for maximum handle length. We keep on increasing the number of robots, ranging from 65 to 103 (step size 2).

In Fig. 41a, Push and Spin produces the highest number of moves among others.[16] However, the heuristics and post process in PASp+ improve the number of moves. Bibox produces low number of moves but still higher than Push and Rotate. In addition, the makespan of Push and Spin produces too low number of moves almost equaling those produced by Push and Rotate. Figure 41b shows that Push and Spin takes the highest computation time than Push and Swap and Bibox. Push and Rotate is the fastest due the use of ordering robot technique. It is clear that Bibox is constantly fast at uniform rate because it solves the instance always in its worst scenario by filling the free nodes by dummy robots and clear their solutions from final result.

On the other hand, Fig. 42 shows FR-VG scenarios, Fig. 42a shows the number of moves produced by Push and Spin, Push and Swap, Push and Rotate and Bibox and Fig. 42b shows their CPU time. All of these are achieved on Bi-connected graphs generated according to a single variable x that ranged from 3 to 30, with steps of two, that was used for all three parameters (number of handles, initial cycle size, and maximum handle length). The number of empty vertices was kept at two (and one exceptionally, since Push and Rotate able to solve instances with single empty vertex).

[16] The CPU time of PASp+ is removed from Fig. 37b since it is too high which show no variant between the CPU time of other algorithms.

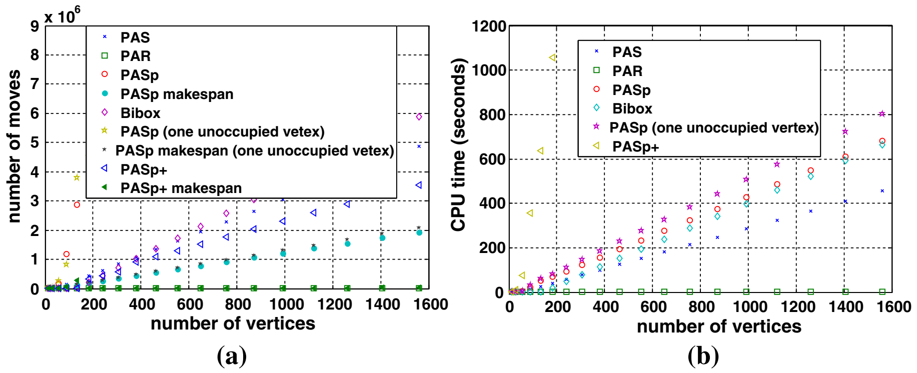


Fig. 42 Comparison on random instances with parameters (handles, initial cycle, max handle length): (3, 3, 3) to (30, 30, 30), with 2 empty vertices and 1 empty vertex for PASp, **a** number of moves, **b** CPU time

In Fig. 42a, it can be seen that Push and Spin produces the highest number of moves for two unoccupied nodes experiments. Bibox produces low number of moves but still higher than Push and Swap and Push and Rotate algorithms. In addition, heuristic and post-process of Push and Spin produces too low number of moves almost equaling those produced by Push and Rotate. For the instances with one empty vertex, Push and Spin is the only one which can solve it, the results show higher number of moves than the instance with two empty vertices solved by Push and Spin due to higher congestion. Its makespan is also higher for the same reason. Figure 42b shows that Push and Spin takes the high computation time, moreover, the improved version takes the highest time due to the heuristic search and post-process time, then, Bibox and Push and Swap take lower time. Push and Rotate is the fastest in these type of instances due to the use of agent ordering heuristic in that pure implementation. In addition, Push and Spin solves one empty vertex biconnected graph slower than two empty vertices biconnected graph due to the congestion. In addition, the highest number of moves generated by Push and Spin algorithm in Fig. 42a, 4.6×10^7 is significant lower than the upper bound calculated for that scenario (1599×1600^3).

8.2.3 Random graph

Our random generator generates two set of experiments each of them varying for specific parameter. Figure 43 shows FG-VR scenarios, Fig. 43a shows the number of moves produced by Push and Spin and Fig. 43b shows its CPU time, all on random graph of size 100 and a number of empty nodes and maximum bridge length ranging from (1, 1) to (100, 100) (step size 5). The number of moves and the CPU time increases with increase in the number of robots. This shows the scalability of Push and Spin algorithm. Furthermore, it shows, in the contrast of Bibox, Push and Spin can benefit of the existence of empty nodes in the graph. In addition, the heuristic and post-process in PASp+ show better performance in term of number of moves and makepan but they consume more time for improvement.

Figure 44 shows FR-VG scenarios, Fig. 44a shows the number of moves produced by Push and Spin and Fig. 44b shows its CPU time. These are calculated on random graphs of different sizes ranging from 53 to 156 (step size 5) with 50 number of empty nodes and maximum bridge length. Figure 44 confirms the scalability and the robustness of Push and Spin. With increase in graph size with the same number of empty nodes (same congestion rate), Push and Spin is able to solve these instance. Moreover, the heuristic and post-process in PASp+

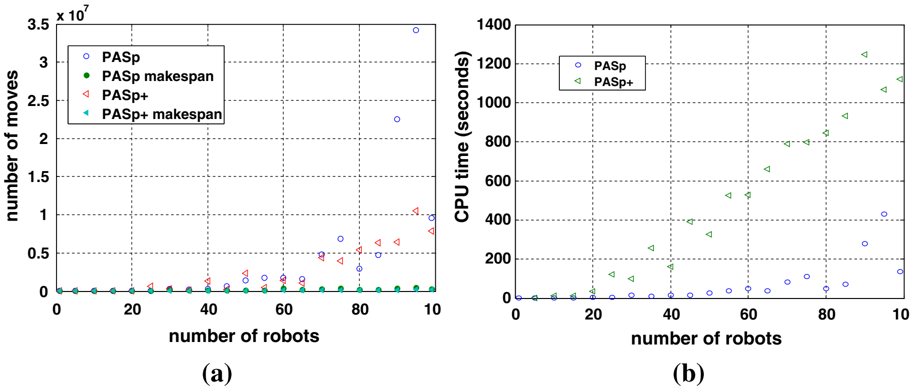


Fig. 43 Comparison on random non-tree graph with 100 vertices and max. bridge and free vertices (1, 1) to (99, 99), **a** number of moves, **b** CPU time

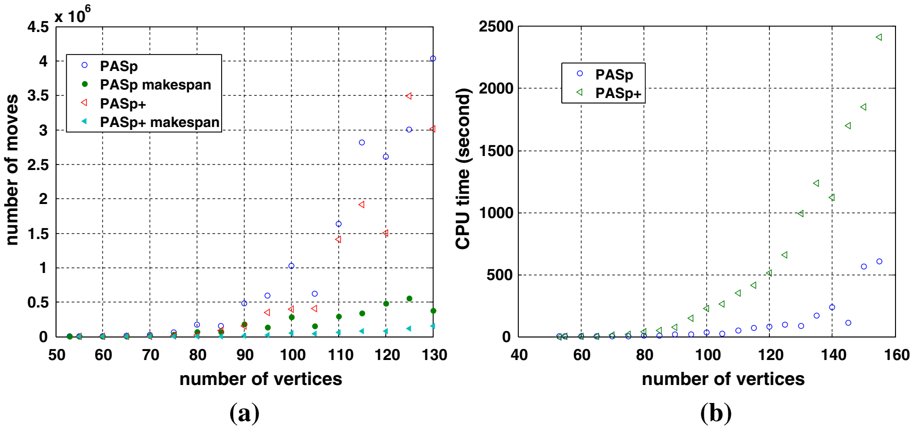


Fig. 44 Comparison on random non-tree instances with maximum bridge and free vertices (50, 50) and graph size from 53 to 156, **a** number of moves, **b** CPU time

show better performance in term of number of moves and makepan but they consume more time for improvement. The highest number of moves generated by Push and Spin algorithm in Fig. 44a, 3.4×10^7 is much lower at about 35.7% than the upper bound calculated for that scenario (95×100^3). Note that none of the algorithms including Push and Swap, Push and Rotate, Bibox, MAPP are able to solve such type of instances when maximum bridge length equals the number of empty nodes.

8.2.4 Random grid instances

As described earlier, Surynek generator generates two set of experiments each of them vary for a specific parameter. Figure 45 shows FG-VR scenarios, Fig. 45a shows the number of moves produced by Push and Spin, Push and Swap, Push and Rotate and Bibox. On the other hand, Fig. 45b shows their CPU time measured on grid graph of 24×24 vertices, the an increasing number of robots, ranging from 376 to 575—24×24 with one unoccupied vertex—(step size 10).

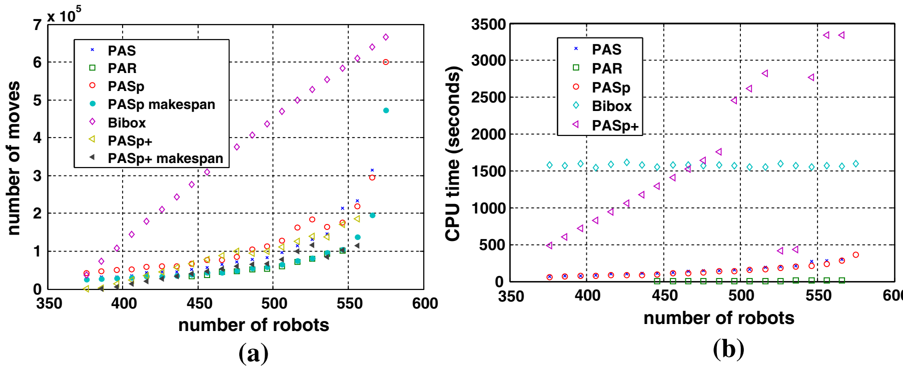


Fig. 45 Increasing number of empty vertices for 24 × 24 grid instances, **a** number of moves, **b** CPU time

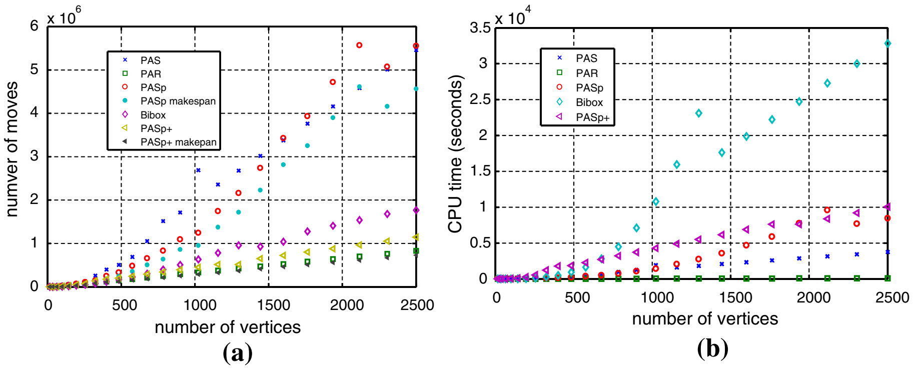


Fig. 46 Comparison on grid instances with sizes from 4 × 4 to 50 × 50, with 2 empty vertices, **a** number of moves, **b** CPU time

Figure 45a shows that Bibox produces the highest number of moves among all of the algorithms. Push and Swap, Push and Spin produce low number of moves but still higher than Push and Rotate. In addition, the makespan of Push and Spin produces very low number of moves and it is closely to those produced by Push and Rotate. In addition the improved version of Push and Spin (PASp+) shows better performance in term of number of moves and makespan. Figure 45b shows that Bibox takes the highest computation time than both Push and Swap and Push and Spin. Push and Rotate is the fastest in these type of instances due to fact that Push and Rotate orders the robots before the solution thereby improving the number of moves. It is clear that Bibox is fast and about constant time because it solves the instance always in its worst scenario by filling the free nodes by dummy robots and clear their solutions from final result. On the other hand, the Fig. 45b shows too much time to improve Push and Spin algorithm.

On the other hand, Fig. 46 shows FR-VG scenarios, Fig. 46a represents the number of moves produced by Push and Spin, Push and Swap, Push and Rotate and Bibox respectively. Figure 46b shows their CPU time, computed on grid size ranging from 4 × 4 to 50 × 50 (step size 2), while the number of empty vertices was kept at two.

As can be seen from Fig. 46a, both Push and Spin as well as Push and Swap produce the highest number of moves among others for two empty node experiments. Bibox produces

Table 1 Impact of heuristic and postprocess on Push and Spin performance

Instance		Path reduction (average of %)	Makespan reduction (average of %)	Spin calls reduction (max of %)
Large public benchmark instances	AR0603SR	42.4	47.5	25.0
	AR0307SR	21.6	18.7	28.3
Random bi connected instances	FG-VR	20.7	28.4	24.3
	FR-VG	47.8	54.0	72.7
Random instances	FG-VR	32.8	49.3	25.3
	FR-VG	42.9	63.5	26.1
Random grid instances	FG-VR	20.3	7.6	29.3
	FR-VG	44.9	52.1	56.1

low number of moves but till higher than Push and Rotate, This is due to the fact that Push and Rotate orders the robots before the solution time which improve the number of moves. For the number of moves factor, the improved Push and Spin algorithm shows competitive performance. Figure 46b shows that Bibox takes the highest computation time, the improved Push and Spin PASp+, Push and Spin and Push and Swap are bit faster. Push and Rotate is the fastest in these type of instances since its robot ordering technique reduces the number of swaps, which reduce the computation time.

The implemented heuristic aims to avoid the paths containing finished robot in order to eliminate Spin operation calls, the smooth operation of post-process aims to remove the redundant moves in the final solution. The effect of these improvements is summarized in Table 1. It shows for each instance described earlier, the average of the percentages of the improved path by heuristic and smooth operation (produced by PASp+) to the original path (produced by PASp). In addition, it shows the average of the percentages of the improved makespan by heuristic and smooth operation to the original makespan. Finally, it shows the maximum percentage of the number of Spin operation calls in PASp+ to the number of Spin operation calls in PASp.

9 Conclusions

In summary, this paper succeeds in presenting three contributions to the field. First, it provides a solvability test to decide early whether or not the problem is solvable. Second, it proposes a new complete algorithm called Push and Spin. Third, it presents a heuristic approach to computing a favorable set of moves with which to move a robot towards its destination vertex in the goal configuration to optimize the complete algorithm.

The Push and Spin algorithm allows simultaneous rotation to cover all solvable instances that are proven to contain as many or more unoccupied vertices than the longest bridge length in the graph. In addition, our algorithm eliminates the tedious process of carrying out reverse moves, as occurs in the Push and Swap and Push and Rotate algorithms, by applying the fact that fully spinning the cycle will restore all the out-of-position robots.

The mathematical proofs demonstrate that Push and Spin is a more complete algorithm for a wider class of problem instances than the class solvable by the Push and Swap, Push and Rotate, Bibox or MAPP algorithms, and solves any graph recognized to be solvable without any assumptions. The following interesting observations can be made from the experiments:

- The original versions of the Push and Spin, Push and Swap, and Push and Rotate algorithms provide closer-to-linear performance on the public benchmark instances when the congestion rate increases. However, the Push and Rotate algorithm provides slightly better performance due to the use of a robot-ordering heuristic that improves the performance. Alternatively, MAPP provides better total-path-lengths when the congestion rate is low. However, its total-path-length grows exponentially when the congestion rate increases. MAPP is the fastest algorithm because it uses cached-alternative paths. The improved Push and Spin (PASp+) provides the best performance in terms of the total-path-lengths. However, it requires a longer time to optimize the solution due to the differences between complete and optimization algorithms that we have previously reported [17]. The complete algorithms provide complete non-optimal solutions, while the optimization algorithms provide optimal incomplete solutions.
- Although graph topology is not one of the factors affecting the performance of the Push and Spin, Push and Swap, and Push and Rotate algorithms, the experiments demonstrate that all three algorithms perform better on grid and bi-connected graphs than on any other graph; this is because the worst-case scenario is produced rarely. The grid and bi-connected graphs contain closer cycles, and hence, a shorter path-to-clear before the swapping vertex (or swapping cycle).
- In general, Push and Rotate provides the best performance for the grid and bi-connected graphs due to the use of the robot-ordering technique. However, it is incomplete for solving grid and bi-connected graphs with one and two unoccupied vertices. Moreover, Bibox provides good performance for bi-connected graphs and worse performance for grid graphs because, as expected, it is affected by the handle length, which is fixed to two in grid graphs. Push and Spin is complete for solving all grid and bi-connected graphs with too high a congestion rate up to one unoccupied vertex. It provides the worst performance among all of the algorithms in bi-connected graphs and a performance that is close to the Push and Swap algorithm in grid graphs.
- The results of the previous point lead us to conclude that Push and Rotate provides the best performance, while Push and Spin provides complete performance in bi-connected and grid graphs. Hence, improving the Push and Spin algorithm with heuristics and post processing may result in better and more complete solutions. The results of the improved Push and Spin on grid and bi-connected graphs indicate this.
- The Push and Spin algorithm is complete for a wider class of problem instances than the class solvable by the Push and Swap, Push and Rotate, Bibox or MAPP algorithms. It solves any graph recognized to be solvable without any assumption needed.
- The improved Push and Spin algorithm combines the advantages of completeness and optimization techniques by applying optimization methods on a set of complete solutions produced by the Push and Spin algorithm. It produces complete optimal solutions with longer execution times.
- Push and Spin scales well for all graphs solvable only by it. The results show that it often performs significantly better (almost 35.7%) than its worst-case time complexity. In addition, in contrast with the Bibox algorithm, Push and Spin offers wider room for improvement.

9.1 Limitations and future works

Although this work has proposed a new complete algorithm for the multi-robot path-planning problem, there are limitations and open research problems that still need to be investigated and solved in future studies.

One of the major factors limiting the performance of the Push and Spin algorithm is the number of executing spin operations needed, which is an operation in which the robot tries to push a planned-robot out of its way. Hence, ordering the robot to execute a move in such a way to reduce the need to push a planned-robot will reduce the number of spin operation invocations; this will improve the performance of the Push and Spin algorithm. Two methods may be used for such a purpose:

- The robot-ordering technique proposed by the Push and Rotate algorithm, which orders robots based on the distance between their current and goal positions. The problems will be solved earlier for a robot far from its goals.
- Decomposing the problem using the Bibox decomposition technique, which solves robots with outer goals first, and once they are planned, their goal vertices will be excluded from the search spaces.

In addition, the criteria for selecting the swapping cycle in the compute-swapping-cycle operation affect the total-path-length in the solution. The compute-swapping-cycle operation explores all the extracted cycles one by one to evaluate a robot's position in relation to the swapping robots and to verify if it can receive all the opponent robots and swapping robots. Here, many heuristics may be implemented to reduce the total-path-length and traversal time of robots. For example:

- In the current definition of the compute-swapping-cycle operation, the algorithm may select a cycle too far away; this would still be able to execute the spin algorithm successfully, though it would generate a higher number of moves. Therefore, adding a distance factor between the swapping robots and the swapping cycle as an additional criterion for selecting swapping cycles yields a better solution.
- In addition, the compute-swapping-cycle operation may select the nearest swapping cycle, though it may be large. This will then produce a large number of moves, because the cycle should be spun a number of spins equal to its degree. Hence, adding a cycle-size factor as an additional criterion for selecting swapping cycles would reduce the number of spins, and hence, the total path length.

Other ideas that could open new horizons include implementing a pattern database and employing a full parallel model for executing robots. We wish to expand these results to real test-bed environments with a larger team of robots.

Acknowledgements We would like to thank both reviewers of autonomous agents and multi-agent systems journal for their insightful comments and critical review which for sure improved the clarity and quality of the work.

References

1. Bhaduri, A. (2009). A mobile robot path planning using genetic artificial immune network algorithm. In *World congress on nature and biologically inspired computing* (pp. 1536–1539). IEEE.
2. Papadimitriou, C. H., Raghavan, P., Sudan, M., & Tamaki, H. (1994). Motion planning on a graph. In *1994 Proceedings. 35th Annual symposium on foundations of computer science* (pp. 511–520). IEEE.
3. Goldreich, O. (2011). *Finding the shortest move-sequence in the graph-generalized 15-puzzle is NP-hard. Lecture notes in computer science* (pp. 1–5). Berlin Heidelberg: Springer.
4. Dresner, K., & Stone, P. (2005). Multiagent traffic management: An improved intersection control mechanism. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems* (pp. 471–477). ACM.
5. Roberts, J. M., Duff, E. S., & Corke, P. I. (2002). Reactive navigation and opportunistic localization for autonomous underground mining vehicles. *Information Sciences, 145*(1), 127–146.

6. Leitner, J. (2009). Multi-robot cooperation in space: a survey. In *Advanced technologies for enhanced quality of life. AT-EQUAL'09* (pp. 144–151). IEEE.
7. Guizzo, E. (2008). Three engineers, hundreds of robots, one warehouse. *IEEE Spectrum, 45*(7), 26–34.
8. Macwan, A., Vilela, J., Nejat, G., & Benhabib, B. (2015). A multirobot path-planning strategy for autonomous wilderness search and rescue. *IEEE Transactions on Cybernetics, 45*(9), 1784–1797.
9. Tang, Z., & Ozguner, U. (2005). Motion planning for multitarget surveillance with mobile sensor agents. *IEEE Transactions on Robotics, 21*(5), 898–908.
10. Zheng, T., Liu, D., & Wang, P. (2004). Priority based dynamic multiple robot path planning. In *Proceedings of 2nd international conference on autonomous robots and agents*.
11. Nieuwenhuisen, D., Kamphuis, A., & Overmars, M. H. (2007). High quality navigation in computer games. *Science of Computer Programming, 67*(1), 91–104.
12. Kornhauser, D., Miller, G., & Spirakis, P. (1984). *Coordinating pebble motion on graphs, the diameter of permutation groups, and applications*. Master's thesis, M.I.T., Cambridge.
13. de Wilde, B., ter Mors, A. W., & Witteveen, C. (2014). Push and rotate: A complete multi-agent pathfinding algorithm. *Journal of Artificial Intelligence Research, 51,* 443–492.
14. Luna, R., & Bekris, K. E. (2011). Push and swap: Fast cooperative path-finding with completeness guarantees. In *IJCAI* (pp. 294–300).
15. Surynek, P. (2009). A novel approach to path planning for multiple robots in bi-connected graphs. In *IEEE international conference on robotics and automation* (pp. 3613–3619). IEEE.
16. 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.
17. Alotaibi, E. T. S., & Al-Rawi, H. (2016). Multi-robot path-planning problem for a heavy traffic control application: A survey. *International Journal of Advanced Computer Science and Applications, 7*(6), 10.
18. Sajid, Q., Luna, R., & Bekris, K. E. (2012). Multi-agent pathfinding with simultaneous execution of single-agent primitives. In *SOCS*.
19. Wilson, R. M. (1974). Graph puzzles, homotopy, and the alternating group. *Journal of Combinatorial Theory, Series B, 16*(1), 86–96.
20. Yu, J., & Rus, D. (2015). Pebble motion on graphs with rotations: Efficient feasibility tests and planning algorithms. In *Algorithmic foundations of robotics XI* (pp. 729–746). Berlin: Springer.
21. Mächler, P. (2012). *Pebbles in motion polynomial algorithms for multi-agent path planning problems*. Master of Science in Computer Science: University of Basel, Basel.
22. Ryan, M. R. K. (2008). Exploiting subgraph structure in multi-robot path planning. *Journal of Artificial Intelligence Research, 31,* 497–542.
23. Yu, J., & LaValle, S. M. (2013). Multi-agent path planning and network flow. In *Algorithmic foundations of robotics X* (pp. 157–173). Berlin: Springer.
24. Hopcroft, J. E., & Tarjan, R. E. (1971). *Efficient algorithms for graph manipulation*. Stanford, CA: University of California.