# Push, Stop, and Replan: An Application of Pebble Motion on Graphs to Planning in Automated Warehouses

Miroslav Kulich[1], Tomáš Novák[2], and Libor Přeučil[1]

***Abstract*— The pebble-motion on graphs is a subcategory of multi-agent pathfinding problems dealing with moving multiple pebble-like objects from a node to a node in a graph with a constraint that only one pebble can occupy one node at a given time. Additionally, algorithms solving this problem assume that individual pebbles (robots) cannot move at the same time and their movement is discrete. These assumptions disqualify them from being directly used in practical applications, although they have otherwise nice theoretical properties. We present modifications of the Push and Rotate algorithm [1], which relax the presumptions mentioned above and demonstrate, through a set of experiments, that the modified algorithm is applicable for planning in automated warehouses.**

Warehouses are used by industries to store assembly parts or goods to be sold. These warehouses often already have a computer system that tracks the position of the product in the racks, but the goods are usually moved to and from the racks by human beings. They navigate through the space between the racks searching for the correct rack and then they search for the item. When assembling an order composed by several different items, they usually spend a lot of the time walking around the warehouse. Companies like Amazon or SwissLog have therefore implemented solutions for automated warehouses where robots bring whole racks to picking stations. Here the human workers pick up the desired items and pack them into boxes according to orders. Such a system requires the robots to be able to navigate in the warehouse and avoid obstacles which could be static – such as walls and racks – or dynamic, mainly other robots and occasionally human beings.

The problem of coordinating a fleet of robots is called multi-agent pathfinding, which is known to be NP-complete for a discrete graph and PSPACE-complete for real environments [2], [3]. Traditional planning methods can be categorized into centralized and decentralized. Approaches from the first category systematically search a composite configuration space built as a Cartesian product of particular robots' configurations and provide optimal solutions [4]–[6]. On the other hand, the thorough search is very time consuming and only small instances can be thus solved. Although many authors developed advanced search space pruning, which decreases the time complexity [7]–[10], the computational complexity is still high: problems with tens of robots are solved in minutes.

Decoupled approaches plan paths for individual robots independently from each other, which is followed by coordination of the robots [11], [12]. Alternatively, prioritized planning is used, which computes trajectories sequentially for individual robots based on their priorities. Robots with already determined trajectories are considered as moving obstacles to be avoided by robots with lower priorities [13]–[15].

Several computationally efficient heuristics have been introduced recently [16]–[18]. Furthermore, another stream of research is based on a solution of the problem called pebble motion on graphs, the planning problem where only one agent moves at a time (the 15-puzzle is the most famous example of this problem). Luna and Bekris [19] present a complete heuristics for general problems with at most $n-2$ robots in a graph with $n$ vertices based on the combination of two primitives - "push" forces robots towards a specific path, while "swap" switches positions of two robots if they are to be colliding. An extension which divides the graph into subgraphs within which it is possible for agents to reach any position of the subgraph, and then uses "push", "swap", and "rotate" operations is presented in [1].

The main shortcoming of the pebble-motion solving algorithms is that individual agents cannot move at the same time. Therefore, the real usage of such a solution in a warehouse would be time-wasting and ineffective. All of the abovementioned algorithms also assume one type of robots, while robots in an automated warehouse might be split into those with and those without racks and assume a constant time of node-to-node movements, which is not possible in a real application. In this paper, we propose a modification of the Push and Rotate (P&R) algorithm [1], which is designed for the use in real warehouse applications by allowing parallel movement of two types of robots (loaded and unloaded) and accounts for non-constant node-to-node movement time. The proposed algorithm moves robots on the shortest possible trajectories to their destinations and in case of a conflict it uses a modified *push* operation or one of the newly proposed operations: *stop* and *replan*.

The rest of the paper is organized as follows. First, the problem is described in Section I. The planning algorithm is detailed in Section II, while experimental results are presented and discussed in Section III. Section IV is finally dedicated to concluding remarks.

Miroslav Kulich and Libor Přeučil are with Czech Institute of Informatics, Robotics, and Cybernetics, Czech Technical University in Prague, Prague, Czech Republic {kulich,preucil}@cvut.cz

Tomáš Novák[2] is with the Department of Cybernetics, Faculty of Electrical Engineering, Czech Technical University in Prague, Prague, Czech Republic tomasnovakmail@gmail.com

This work has been supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No 688117, by the Technology Agency of the Czech Republic under the project no. TE01020197 "Centre for Applied Cybernetics", and by the European Regional Development Fund under the project Robotics for Industry 4.0 (reg. no. CZ.02.1.01/0.0/0.0/15 003/0000470).

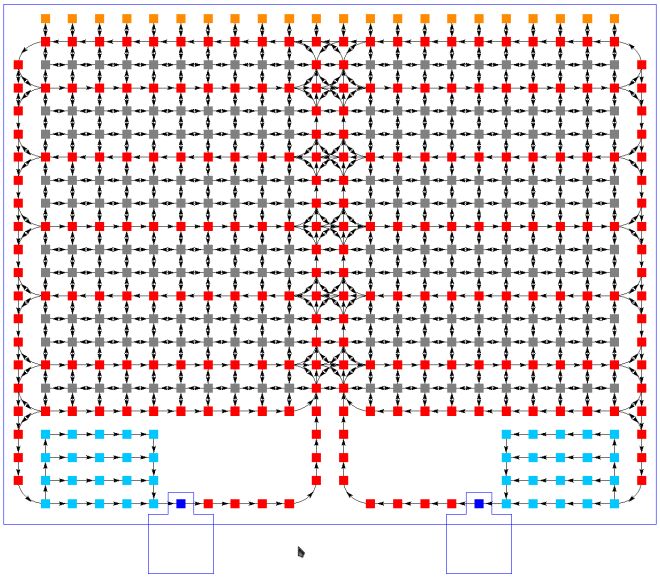


Fig. 1: A map of a real warehouse.

## I. Problem definition

The complete solution for robot management in an automated warehouse is typically composed of three layers. The highest layer is the warehouse management system (WMS), which creates a queue of tasks that are supposed to be accomplished from an order that is currently processed. The order may consist of several items that are stored at different locations. Their position is thus determined, and the manager adds the information which rack has to be brought to which picking station. The middle layer processes the queue from WMS and determines an appropriate robot for each task. The lowest layer, the path-planning algorithm, coordinates the movement of the robots by computing their collision-free trajectories given their start and goal positions.

The path planning in the last layer is addressed in this paper. Specifically, we assume a connected, directed graph $G = (N, E)$, where $N$ is a set of nodes and $E$ is a set of connections between them. For each robot $r_i \in R$ start $s_i$ and goal $g_i$ nodes are given as well as a subset of edges $E$ where the robot is allowed to operate. The aim is to find a set of non-colliding trajectories from $s_i$ to $g_i$ for each robot while minimizing some global cost function. A typical cost function is a sum of robots' travel times or a plan completion time.

Fig. 1 illustrates a map of a typical warehouse: nodes on roads (red), positions of racks (grey), picking stations (blue polygons), maintenance nodes/charging stations (orange), queues before picking stations (light blue). Note that robots with loaded racks cannot go to grey nodes if they are occupied by some racks, while unloaded robots can. Two types of edges are distinguished: straight lines (we call them *default*) and turns, which allow robots to concurrently move and rotate *spline*.

## II. Proposed algorithm

### A. Warehouse-related requirements

Several challenges appear when designing an algorithm for real continuous environments instead of for a discrete world of graphs. On the other hand, one can employ several simplifications in comparison to general graph algorithms using typical properties of a warehouse to simplify the topology of the associated graph. The main challenges that had to be resolved during the algorithm design are discussed in this section.

*1) Non-constant time of movement between nodes:* Contrary to standard graph algorithms, where robots move between nodes discretely, they can also occupy space between the nodes. This movement is represented by a sequence of time steps containing necessary information about the robots: time, position, rotation, velocity and a set of occupied nodes. A model of a robot's movement is necessary to generate these sequences. The precision of this model defines the usability of the real scenario; however, in the proposed algorithm we only use a very simple model and propose a modification to deal with its inaccuracy in the real world scenario.

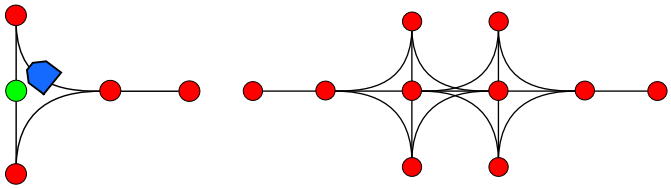
*2) Node/edge conflicts, mainly at spline edges and complicated junctions:* A set of conflicting nodes and edges that no robot can be present at the same time is defined for every node and edge. This is mainly the case of spline edges and complicated junctions as in Fig. 2.

*3) Parallel movement of robots:* is necessary to reduce the makespan of the solution and make it viable for usage in a real warehouse. To achieve this goal, all robots are moved simultaneously at time-step. When one robot happens to conflict with another robot, one of the operations described in Section II-B is invoked to resolve the conflict. All of the operations involve only the robots in conflict and the positions of the task-accomplished robots. The consideration of a simultaneous movement of robots causes an increase in the complexity of the algorithm, but also adds more flexibility.

*4) Simplifications:* A real warehouse environment increases the complexity of the algorithm significantly; however, the graph of the warehouse has properties that can be used to simplify the planning algorithm. It is assumed that the graph is always biconnected. The decomposition in the P&Rotate algorithm thus always ends up with one component, and, therefore, it can be omitted. Moreover, the *swap* operation always successfully completed as proved in [1] and the *rotate* operation can be thus also omitted.

*5) Runtime requirements:* Robots in a warehouse are not given their tasks at once, but rather the goals are assigned one by one from the warehouse management system. When a goal is assigned to the robot, the time to find the solution should be minimized, which is a challenging requirement to meet. The algorithm should thus be able to reuse the already evaluated solution and add a new robot's movement as fast as possible to it.

*6) On-the-fly processing:* The algorithm produces plans on the fly. This means that first parts of plans are available



(a) Spline shortcuts (the green node is in conflict with the robot).

(b) Complex spline junctions

Fig. 2: Complex spline junctions at the corners of aisles in the warehouse.

and robots can start moving according to them before the planner finishes. The only problem is in case of collision when the planner virtually moves the time backward to solve the collision. In the extreme case when one collision resolution immediately causes a new conflict, the time can be moved backward significantly. Therefore, some solution buffer has to be introduced guaranteeing that the virtual time in the planner is not behind the real-time. To do that, a statistical evaluation to calculate the size of the solution buffer has been done. Nevertheless, it can happen that the buffer would get too small during the movement of the robots; the movement would have to be paused in this situation.

### B. Algorithm description

The proposed planning algorithm consists of three parts: the single robot path planning phase, the initial trajectory generation, and the robot maneuvering phase, and uses operations *stop*, *push*, and *replan* to modify the trajectories in case of conflict. Moreover, the current state and time of the warehouse is maintained.

The priority of each robot is, initially, equivalent to length of the shortest paths from its origin to its destinations Besides this, a temporary priority of each robot is initialized to these constant priorities, incremented by the *push* operation and reset to the value of the main priority. The *stop* operation tries to resolve the conflict by stopping one of the robots, while the *push* operation pushes robot with a lower temporary priority out of the path of the other conflicting robot.

*1) Single robot path planning phase:* The shortest paths for all robots from their start to their goal positions are generated on the graph $G$ making use of the standard A* algorithm [20]. The used heuristic function is the Euclidean distance to the goal, while the determination of edge cost depends on the edge type. We define the cost $C_l$ for spline edges and $C_s$ for edges to storage location nodes bellow, while the default cost $C_d = 1$ is used for all other edges.

Traveling through spline edge is shorter than travel trough two default edges, but longer than traveling through one; thus $C_d < C_l < 2C_d$ and is set to 1.5. The edges that end in the storage location nodes are in most cases traveled by the robots without racks. To enforce their preference to travel under the racks and leave more space on the road nodes for robots with racks, the cost $C_s$ must be $0 < C_s < C_d$ and $0 < C_s < \frac{C_l}{2}$, and is set to 0.1. The robots that carry a rack cannot use edges starting or ending at the storage location nodes except initial and goal state of the robots.

*2) Initial trajectory generation phase:* The generated paths are processed by the robot model. The output for each robot $r_x$ is a list of time steps $J_r = \{j_1, j_2, \ldots, j_n\}$, where $n$ is the number of time steps in the path. The output data depends on the model parameters. Because we need to discretize continuous movements, the time steps are samples of real trajectory movement; thus sample frequency must be defined reasonably. The algorithm directly processes the time samples. Therefore the computational difficulty grows with the sampling frequency. Choosing a too small number of samples could lead to failure in case that there are not at least 2 samples between two nodes – each where a robot occupies one of the two nodes on the edge. The robots could come into a conflict in moments that were not captured by the samples; thus the algorithm would not detect it. It is reasonable to have at least ten samples for each edge. To make sure the sampling frequency $F_s$ is sufficient, it should meet the condition $F_s > \frac{l_{min}}{v_{max}} \times 10$, where $l_{min}$ is the minimum length of an edge and $v_{max}$ is the maximal robot velocity.

**Algorithm 1:** Robot maneuvering algorithm

**Data:** generated trajectories $J$
**Result:** Trajectories $J$, modified to be collision-free.

```
1  solved ← false
2  while !solved do
3      if conflict_detect() then
4          resolve_crash(crash)
5      end
6      if check_finished() then
7          solved ← true
8      else
9          resolve_priority_reset()
10         resolve_replan()
11         state_shift(1)
12     end
13 end
```

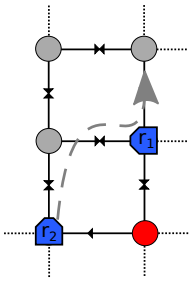
*3) Robot maneuvering phase:* This phase is described in Algorithm 1. The loop (line 2) is repeated until no conflict exists. First, it is checked whether there is a conflict in the current state of robots (line 3). Detected conflicts are immediately resolved by the *resolve_crash* Algorithm. After that, it is checked whether all robots reached their final destination (line 6). When the problem is not solved yet, temporal priorities are reset, and paths of robots blocked by already finished robots are replanned (lines 9–10). Moreover, the time is incremented, and all robots are moved accordingly (line 11).

The *conflict_detect* procedure searches through all the blocked nodes and edges and checks whether they are occupied by some robot other than the currently tested one.

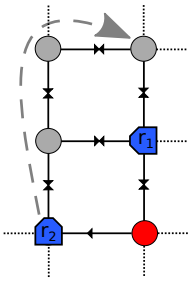
The *resolve_crash* algorithm solves one crash at the time. If several conflicts occur at the same time, only the first one is resolved; however, all the operations will cause the time to move one back; thus the other conflicts will also be resolved. The algorithm consists of two steps: it is first decided which operation from the set $\{stop, push, replan\}$ will be used, and this operation is executed.

The decision is done in several steps. As we do not allow to move a robot which is already in its final destination, replanning of the other robot in the conflict is invoked by the *replan* operation. When no robot is finished, it is tested whether *stop* can be used by checking two conditions for both robots. The first one is that the other robot's path does not cross the node that the tested robot would be stopped at. In Fig. 3a, the robot $r_2$ has a path planned in a way, that stopping the robot $r_1$ would not help to resolve the conflict. If the path was planned differently (Fig. 3b), the *stop* operation helps to resolve the conflict completely.

The other ond condition is implemented to prevent deadlock situations that might arise from *stop*. When the robot is stopped, it is put into the `idle` state and waits until the first edge on its path is empty. To avoid possible deadlocks, the robot that is supposed to continue without stopping cannot have any idle robots on its way. When no other operation is selected, the *push* operation is chosen.




(a) If $r_1$ is stopped, the robot $r_2$ would still crash into it.



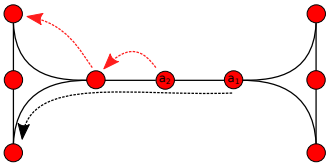

(b) *stop* operation is possible. Stopping robot $r_1$ allows robot $r_2$ to move towards its goal.

Fig. 3: Examples showing situations when the *stop* operation is possible and when it is not.

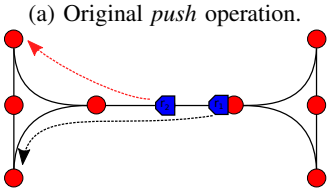
### C. Operations

The *stop* operation (Algorithm 2) aims to stop one robot so that the other one can continue without disruption. It is decided which robot should be stopped (line 1) first. The algorithm already gets a set of one or two robots, which can be stopped and selects the one with a lower temporary priority. The time is then shifted back until the $r_{stop}$ is not occupying the conflicting node (line 2). $r_{stop}$ is thus in some other node $n_s$, where it will be stopped.

The wait sequence is generated (line 3) as a sequence of $steps\_shift$ time steps where the robot $r_{stop}$ stands still at the node $n_s$ ending with the time step with a special $idle$ flag. If this special time step is the next step, it is checked during the state shift (Algorithm 1, line 11) whether the edge that the robot $r_{stop}$ is going to move at is not in conflict



(a) Original *push* operation.



(b) New *push* operation.

Fig. 4: Comparison of the original and new *push* operation on simple case.

with any other robot. The wait sequence is extended to the moment there is no conflict. The generated wait sequence is then added to the path after the current state extending the planned trajectory $J_{r_{stop}}$ (line 4).

**Algorithm 2:** *stop* operation

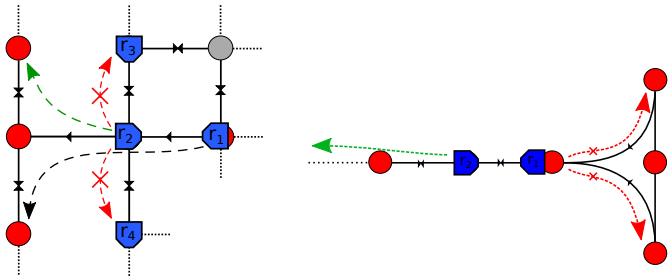
**Data:** Robots that crashed $r_1$ and $r_2$, robot model $L$
**Result:** Stops one of the robots and resolves the conflict.

1 $[r_{stop}, r_{go}] \leftarrow$ Decide which robot to stop
2 $steps\_shift \leftarrow$ Shift time back until $r_{stop}$ is not occupying the conflicting node
3 $J_{wait} \leftarrow$ Generate wait sequence using $L$.
4 Update $J_{r_{stop}}$ with $J_{wait}$

The *push* operation is based on the same operation from the P&R algorithm, which is used for the movement of agents even when there is no conflict. In the proposed algorithm, only the part of the operation that is invoked when the next node is occupied by an agent is considered, because it is used for resolving conflicts only and not moving the robots itself. The original *push* moves a robot only by one node; thus the operation can be executed several times for a single robot if the robots have a conflict on a long isthmus as is illustrated in Fig. 4a with the red arrows. The new version moves the agents arbitrarily far when moving on an isthmus or when the closest nodes cannot be used for example if they are occupied by finished robots (Fig. 5a).

The new *push* (Algorithm 3) decides first which robot $r_{push}$ will be pushed away and which robot $r_{go}$ will continue on its path (line 1). The robot to be pushed is the one with a lower temporary priority, as it is less likely that the robot was pushed recently and the higher-priority robot has more likely a longer trajectory to travel through. In some situations, the chosen robots cannot be pushed due to a direction of adjacent edges, see, e.g., Fig. 5b. The other robot is selected in this case.

Next, *push* finds the closest node to $r_{push}$ that is not on the path of $r_{go}$. List of nodes and edges that are on a path of




(a) The robot $r_2$ cannot be pushed to the closest nodes, because those are occupied by finished robots $r_3$ and $r_4$ and must be pushed to the top left red node.

(b) It is impossible to push the robot $r_1$, because the only exiting edge ends on a node occupied by the robot $r_2$, which is trying to push it. The robot $r_2$ must be pushed.

Fig. 5: Admissibility of the *push* operation.

$r_{go}$ are thus created (lines 3 and 4). Moreover, a list of nodes that are forbidden to expand during the search is determined. First, the node where $r_{go}$ is going to wait is added to the node list to prevent robot $r_{push}$ from being pushed through this node and all nodes with finished robots are added since it is not possible to move them (line 5). The path is then found (line 7) using a modified version of Dijkstra's algorithm [21] that accounts for the blocked edges and nodes with forbidden expansion. The state of the algorithm is shifted back until the robot $r_{push}$ does not occupy a node (line 8) and all its future time steps are removed from its planned trajectory to be later replaced with the push trajectory (line 9).

The trajectory is generated using the robot model, and the $reset$ flag is added to the last generated step (line 10). The temporary priority of $r_{push}$ is increased by $r_{go}$ (line 13) to push other robots that might get into conflict with $r_{push}$ while being pushed. This way only a robot with a very high priority would be able to push this robot back. Priorities ensure that the non-finished robot with the highest priority always moves towards its destination. When the push is finished, the robot $r_{push}$ resets its temporary priority when the time step with the $reset$ flag is encountered in the Algorithm 1 (line 9).

As the push of $r_{push}$ is executed on directed edges, moving back to the original position using the same path might be impossible. Instead, the algorithm generates a trajectory from the last node of $push\_path$ to the goal node of $r_{push}$ (line 11) using *replan*. The trajectories are added together to form a new trajectory of the robot $r_{push}$.

The algorithm stops $r_{go}$ at the last visited node before the conflict. First, we need to shift the time to a state where $r_{go}$ is at this node. However, the state was shifted back before thus the state must be shifted by $(steps\_shift - t_{back\_to\_node})$, where $t_{back\_to\_node}$ is the number of steps from robot $r_{go}$ leaving the previous node before the state shift back (line 14). This shift can be either forward or backward. Similarly to *stop*, the operation generates the wait sequence $J_{wait}$ for $r_{go}$ with a minimal wait of $(steps\_shift - t_{back\_to\_node})$ steps to ensure that $r_{push}$ will get to the state of conflict. The special $idle$ flag is added to the last step of the wait (line 15). The operation adds the $J_{wait}$ sequence into the $J_{r_{go}}$ trajectory right after the current step (line 16) and the operation is complete.

**Algorithm 3:** The *push* operation.

**Data:** Robots in conflict, robot model $L$, graph $G$
**Result:** Resolves conflict with *push* operation.

1 $[r_{push}, r_{go}] \leftarrow$ Decide which robot to let go and which robot to push.
2 $t_{back\_to\_node} \leftarrow$ Number of steps from when $r_{go}$ left last node.
3 $blocked\_nodes \leftarrow$ Nodes in path of $r_{go}$.
4 $blocked\_edges \leftarrow$ Edges in path of $r_{go}$.
5 $no\_expansion\_nodes \leftarrow$ A node where $r_{go}$ waits or nodes with already finished robots.
6 $n_p \leftarrow$ Last node that $r_{push}$ occupied.
7 $push\_path \leftarrow$ Find a path to closest node to $n_p$ with respect of $blocked\_nodes$, $blocked\_edges$ and $no\_expansion\_nodes$ on graph $G$.
8 $[steps\_shift, t_{now}] \leftarrow$ Shift state back until $r_{push}$ is occupying a node.
9 $J_{r_{push}} \leftarrow J_{r_{push}}(j_0, \ldots, j_{(t_{now})})$.
10 $J_{push} \leftarrow$ Generate trajectory using $push\_path$ and $L$.
11 $J_{replanned} \leftarrow$ Generate trajectory from last node of $push\_path$ to goal node of $r_{push}$ using *replan*.
12 $J_{r_{push}} \leftarrow J_{r_{push}} \cup J_{push} \cup J_{replanned}$.
13 temporary priority of $r_{push} \leftarrow$ temporary priority of $r_{push}+$ temporary priority of $r_{go}$.
14 Shift state by $(steps\_shift - t_{back\_to\_node})$.
15 $J_{wait} \leftarrow$ Generate wait sequence with at least $(t_{back\_to\_node} - steps\_shift)$.
16 Update $J_{r_{go}}$ with $J_{wait}$.

The *replan* operation (Algorithm 4) is used when one of the robots in conflict is finished (Fig. 6). The finished robots cannot be moved, thus *push* and *stop* would not help in this case. The operation plans a new trajectory of the robot $r_x$ from its current node $n_c$ to the goal node $n_g$ while avoiding all nodes that the finished robots occupy. The property of a graph that by removing any storage location nodes, the graph will not become disconnected, is considered.

At first, the algorithm identifies which of the robots is not finished (line 1) to select the one that needs to be replanned. All nodes occupied by finished robots are added to the list $nodes\_to\_avoid$ (line 2) to ensure that they are avoided. The solution state is shifted back until the robot $r_x$ is not occupying the conflicting node (line 3). The operation removes the planned trajectory of the robot $r_x$ from the current state to replace it further with the replanned one (line 6). The algorithm then calculates the shortest path from the currently occupied node $n_c$ to the goal node $n_g$ using A* algorithm (line 7) assuming that all the nodes from $nodes\_to\_avoid$ are set as unreachable. A trajectory for $r_x$ describing its movement from node $n_c$ to its goal node $n_g$

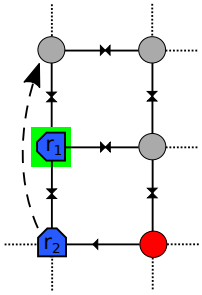



Fig. 6: Both operations *stop* and *push* would fail, because the robot $r_1$ is finished and cannot be moved. This situation requires the *replan* operation.

is generated (line 8) and added to the planned trajectory $J_{r_x}$ (line 9).

Note that *replan* operation removed a part of the planned trajectory and replaced it with a new one. If $r_x$ was performing the *push* operation, the *reset* flag for resetting the priority would be deleted. The algorithm resets the priority (line 10) to avoid this loss because the change of the trajectory causes the robot is no longer performing the *push* operation.

**Algorithm 4:** *replan* operation

**Data:** Robots in conflict, robot model $L$, graph $G$
**Result:** A new trajectory $J_{r_x}$ of a non-finished robot $r_x$.

1 $r_x \leftarrow$ The robot that is not finished.
2 $nodes\_to\_avoid \leftarrow$ Nodes occupied by finished robots.
3 $t_{now} \leftarrow$ Shift state back until $r_x$ is not occupying the current node.
4 $n_c \leftarrow$ Node occupied by $r_x$.
5 $n_g \leftarrow$ Goal node of $r_x$.
6 $J_{r_x} \leftarrow J_{r_x}(j_0, \ldots, j_{(t_{now})})$.
7 $P \leftarrow$ Calculate the shortest path from $n_c$ to $n_g$ avoiding nodes in $nodes\_to\_avoid$.
8 $J_{replan} \leftarrow$ Generate trajectory using $P$ and $L$.
9 $J_{r_x} \leftarrow J_{r_x} \cup J_{replan}$
10 temporary priority of $r_x \leftarrow$ main priority of $r_x$.

### D. Algorithm limitations

As already mentioned, limitations of the proposed algorithm are caused by simplifications made and specialization on the real environment. Specifically, the directed graph must be connected and biconnected at prospective goal nodes except maintenance nodes.

Another limitation is the maximal number of robots in a given map. This number should be theoretically the same as in the P&R algorithm, i.e., $n - 1$ where $n$ is the number of robots. Because no robot can finish on road nodes, the number of robots is thus $n - n_r$, where $n_r$ is the number of road nodes. In the warehouse used during development and testing of the algorithm, the rate of robots to nodes (ignoring pick-station, isthmuses leading to and from pick-stations and queue parts of graph) is approximately $0.52$.

### E. Algorithm advantages

The calculation of trajectories for a high number of robots in a big warehouse is computationally demanding. Also, the goals for robots do not have to be known in advance, and some might be added later on. The proposed algorithm solves both of these issues as discusses below.

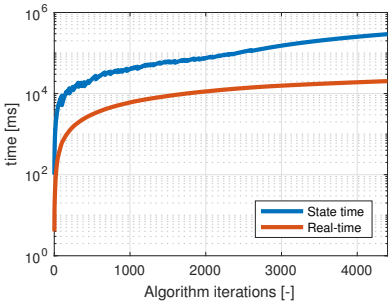
The algorithm moves the state forward in time, and only when any conflict occurs, it moves the state back in time. For one operation, the time the state is moved back is the maximal time of the conflicted robots moving from the last node on the edge. However, if the operation causes another conflict before the operation is finished (e.g., stopping one robot causes a new conflict with another robot), the state could be moved back again. The amount of back-shifting is not limited, but long shifts are highly improbable. The algorithm can start running, buffer a solution for some time and then the robots can start moving in real time with s low risk of the solution state moving behind the real state of the warehouse. Implementation of the safety stop of the system when the state of the robots gets close to the state of the solution should be, of course, implemented. The buffering time must be decided by numerous simulations, and the available computational power must also be considered. The discussion about practical values of the buffering time is made in Section III.

The algorithm allows for tasks being added during the calculation. The state of the solution must be moved back to the time when the new robot is supposed to start moving and the robot is simply added with its shortest path to its destination. It might cause new conflicts in previously calculated trajectories, but for conflicts that it does not affect, there is no need for recalculation, while the trajectories of these robots are already collision-free. One issue that might occur is that due to the impact of the newly added robot, some robots will perform operations that are no longer needed. For example, the newly added robot $r_1$ affects another robot $r_2$ that in the previous calculation would get into conflict with the robot $r_3$. In the previous iteration, the robot $r_2$ pushed the robot $r_3$, and this trajectory was added to its trajectory. In the next iteration, the robot $r_1$ stops the robot $r_2$, and it will not get into conflict with the $r_3$, but while the robot $r_3$ has the trajectory of the operation already calculated, it will still perform it. This might lead to unnecessary movements and delays.

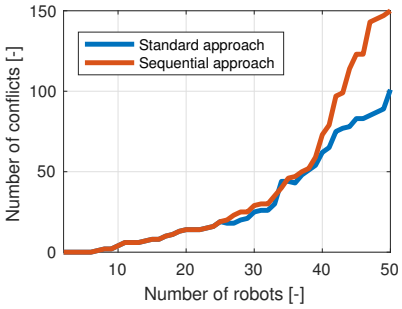
## III. Experiments

The goal of the experiments is to assess the usability of the proposed algorithm in practice. The experiments were performed on a computer with an Intel i7-4771 processor and 8GB RAM running Linux Mint.

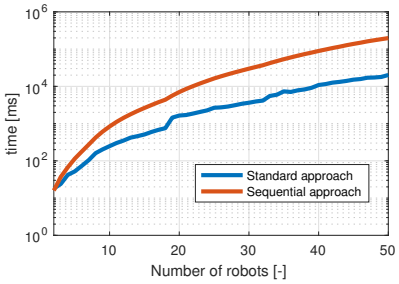
The warehouse map displayed in Fig. 1 was used for experiments. A task for $50$ robots (22 carrying a rack and 28 without it) with various distances between start and goal nodes was created randomly. Maintenance and storage




(a) Real-time during the state time of the algorithm during the calculation of the *sequential* approach.




(b) The number of conflicts for the *sequential* and *standard* approaches.




(c) Calculation time of the *sequential* and *standard* approaches..

location nodes were used as start and goal positions. The algorithm was executed with 2 to 50 robots to study the influence of the growing number of robots on the algorithm performance.

### A. Execution delay

One of the main advantages of the algorithm is discussed in Section II-E: the algorithm moves forward in time, and calculated trajectories can be performed before the algorithm finishes. As the algorithm can move back in time, the risk of the execution being before the calculation must be addressed.

The algorithm was executed with 50 robots and the time of the solution state was compared to real-time. In Fig. 7a one can see that the difference between state time and real-time grows (logarithmic scale was used for easier visual comparison); thus it is highly unlikely that the lines ever cross. In this case, the buffering time can be very small; thus the robots can start moving towards their goals instantly.

The more difficult the problem is (bigger warehouse, more robots), the higher is the risk of the execution catching up with the algorithm. This can be overcome with more computational resources and reasonable buffering time. One can also assume that not all robots will move at the same time. Some robots might be charging at the maintenance stations while others might be waiting in queue for a picking station.

### B. Two approaches comparison

The ability of the algorithm to add robots to the plan during the calculation was also discussed in Section II-E. The extreme case of adding robots one by one to the beginning of the solution was tested to assess the impact on the results. The algorithm assumes that $k$ robots have already their plans and then a plan for the $k+1$th robot is calculated. This approach is named *sequential*, while the approach of assuming all $n$ robots at once is named *standard*.

The number of conflicts to be solved for each amount of robots from 2 to 50 for both approaches was recorded and compared first. One can see in Fig. 7b that the sum of conflicts for the *sequential* approach is comparable with the *standard* approach for the amount of robots ranging from 2 to 39. This shows that the newly added robots in this task only cause a few new conflicts with the comparison with the full calculation. With the growing number of robots, however, the probability of long parts of the trajectories of robots being replanned due to the addition of new robots is growing. The conflicts that have been solved previously might be thrown away with the trajectory, and thus new conflicts must be calculated. This result also confirms the expected property that the number of conflicts grows exponentially with the number of robots.

Perhaps the most significant impact of the *standard* approach is on the solution time. Running the algorithm multiple times through the whole plan demands significantly more computational resources. In each run, fewer resources are needed since most conflicts were already solved. However, the cumulative value of calculation time for the *sequential* approach is always significantly higher as seen in Fig. 7c (a logarithmic scale is used for better visualization). In less extreme cases, adding a task during the calculation still causes a delay. However, this is not as significant than the recalculation of the whole solution.

For example, 40 robots start moving at the same time, and the algorithm gets far in front of the real execution. After a few seconds, when the algorithm is almost finished, a task for a robot is added 2 seconds in advance to the real execution. The algorithm will go back and use the already calculated trajectories; thus only newly caused conflicts need to be calculated again. Some extreme cases might occur; thus it is very important to cautiously choose the optimal time reserve when adding a task for a robot.

### C. Solution quality

To measure the quality of generated trajectories, we compute a theoretical lower bound as a sum the shortest path length for each robot obtained by A* that ignores all other robots in the assignment. The ratio of the sum of trajectory steps for all robots provided by the proposed algorithms in comparison to this bound is shown in Fig. 8. This represents the effect of the operations on the trajectories with an increasing number of robots. There are no conflicts between the first 6 robots; thus their cumulative trajectory length is the same as for the lowest threshold trajectories. The cumulative trajectory length grows with the increasing

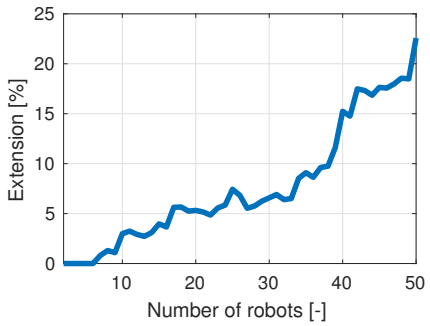



Fig. 8: Extension of the sum of trajectories lengths compared to the sum of the lengths of the lower-bound trajectories.

number of conflicts. The figure is highly correlated with Fig. 7b which represents the number of conflicts with the increasing number of robots.

## IV. Conclusion

A novel planning algorithm for coordination of a robotic fleet in an automated warehouse based on a pebble motion on a graph algorithm, Push and Rotate, is presented. The algorithm allows a continuous movement of robots on their trajectories instead of a discrete movement between nodes. It also takes into consideration the rotation of the robots and their different velocities. One of the main achievements is that all the robots can move in parallel, which was not possible with the original P&R. This is achieved by an implemented system of priorities that assures that the robot with the longest trajectory always moves towards its final destination. Whenever the robots are involved in a push operation, the accumulation of priorities overcomes problems with deadlocks. The computational time for the solution that has to be calculated before the result can be executed was another challenge. The algorithm solves this issue inherently by taking the shortest trajectories to destinations. The trajectories are only modified during the calculation. Thus if the algorithm is faster than real-time, the solution can be executed before the calculation is finished. The algorithm also allows adding robots during the calculation.

In future work, we want to study both the theoretical and practical aspect of the algorithm more thoroughly. From the theoretical point of view, the aim will be to prove the completeness of the algorithm as well as to derive bounds for the buffering time. Moreover, we want to perform experiments in more challenging setups, i.e., scenarios in larger maps and involving more robots. We are also preparing experiments in a real environment.

## References


[1] B. DeWilde, A. Ter Mors, and C. Witteveen, "Push and Rotate: A complete Multi-agent Pathfinding algorithm," *Journal of Artificial Intelligence Research*, vol. 51, pp. 443–492, 2014.

[2] J. Hopcroft, J. Schwartz, and M. Sharir, "On the Complexity of Motion Planning for Multiple Independent Objects; PSPACE- Hardness of the "Warehouseman's Problem"," *The International Journal of Robotics Research*, vol. 3, no. 4, pp. 76–88, Dec. 1984.

[3] O. Goldreich, "Finding the Shortest Move-Sequence in the Graph-Generalized 15-Puzzle Is NP-Hard," in *Studies in Complexity and Cryptography. Miscellanea on the Interplay between Randomness and Computation - In Collaboration with Lidor Avigad, Mihir Bellare, Zvika Brakerski, Shafi Goldwasser, Shai Halevi, Tali Kaufman, Leonid Levin, Noam Nisan, Dana Ron,*, ser. Lecture Notes in Computer Science, O. Goldreich, Ed. Springer, 2011, vol. 6650, pp. 1–5.

[4] J.-C. Latombe, *Robot Motion Planning*. Norwell, MA, USA: Kluwer Academic Publishers, 1991.

[5] S. M. Lavalle, "Rapidly-Exploring Random Trees: A New Tool for Path Planning," *Technical Report (Computer Science Deptartment, Iowa State University)*, vol. 11, 1998.

[6] M. R. Ryan, "Exploiting Subgraph Structure in Multi-Robot Path Planning," *Journal of Artificial Intelligence Research*, pp. 497–542, 2008.

[7] J. van den Berg, J. Snoeyink, M. C. Lin, and D. Manocha, "Centralized path planning for multiple robots: Optimal decoupling into sequential plans," in *Robotics: Science and Systems V, University of Washington, Seattle, USA, June 28 - July 1, 2009*, J. Trinkle, Y. Matsuoka, and J. A. Castellanos, Eds. The {MIT} Press, 2009.

[8] A. Geramifard, P. Chubak, and V. Bulitko, "Biased Cost Pathfinding." in *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2006, pp. 112–114.

[9] M. Peasgood, C. M. Clark, and J. McPhee, "A Complete and Scalable Strategy for Coordinating Multiple Robots Within Roadmaps," *Robotics, IEEE Transactions on*, vol. 24, no. 2, pp. 283–292, Apr. 2008.

[10] K.-H. C. Wang and A. Botea, "Fast and Memory-Efficient Multi-Agent Pathfinding," in *ICAPS*, 2008, pp. 380–387.

[11] S. LaValle and S. Hutchinson, "Optimal motion planning for multiple robots having independent goals," *IEEE Transactions on Robotics and Automation*, vol. 14, no. 6, pp. 912–925, 1998.

[12] T. Simeon, S. Leroy, and J.-P. Lauumond, "Path coordination for multiple mobile robots: a resolution-complete algorithm," *IEEE Transactions on Robotics and Automation*, vol. 18, no. 1, pp. 42–49, 2002.

[13] J. van den Berg and M. Overmars, "Prioritized motion planning for multiple robots," in *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2005, pp. 430–435.

[14] M. Bennewitz, W. Burgard, and S. Thrun, "Optimizing schedules for prioritized path planning of multi-robot systems," in *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, 2001, pp. 271 – 276 vol.1.

[15] M. Cap, P. Novak, A. Kleiner, M. Selecky, and M. Pechoucek, "Prioritized Planning Algorithms for Trajectory Coordination of Multiple Mobile Robots," *IEEE Transactions on Automation Science and Engineering*, vol. Special Is, 2015.

[16] K. Chiew, "Scheduling and routing of autonomous moving objects on a mesh topology," *Operational Research*, vol. 12, no. 3, pp. 385–397, Nov. 2010.

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

[18] D. Silver, "Cooperative Pathfinding." in *The 1st conference on Artificial Intelligence and Interactive Digital Entertainment*, 2005, pp. 117–122.

[19] R. Luna and K. E. Bekris, "Efficient and complete centralized multi-robot path planning," in *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, Sep. 2011, pp. 3268–3275.

[20] K. Beevers, "Boost graph library: A* heuristic search - 1.64.0," http://www.boost.org/doc/libs/1_64_0/libs/graph/doc/astar_search.html, accessed: May 9, 2017.

[21] J. Siek. (2017) Dijkstra's shortest paths. [Online]. Available: http://www.boost.org/doc/libs/1_60_0/libs/graph/doc/dijkstra_shortest_paths.html