Heterogeneous agents extension for Push & Rotate algorithm

Laurens Martin

Supervisor: Prof. dr. Peter Lambert
Counsellor: Ignace Saenen

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Department of Electronics and Information Systems
Chair: Prof. dr. ir. Rik Van de Walle
Faculty of Engineering and Architecture
Academic year 2016-2017

Permission for consultation

"The author(s) gives (give) permission to make this master dissertation available for consultation and to copy parts of this master dissertation for personal use. In the case of any other use, the copyright terms have to be respected, in particular with regard to the obligation to state expressly the source when quoting results from this master dissertation."

Laurens Martin, August 2017

Acknowledgements

I would like to thank my counsellor Ignace Saenen for providing tips and guidance as well as my supervisor prof. dr. Peter Lambert for his feedback. I'd like to express thanks to Stéphanie Carlier, Joris Heyse, Simon Houbracken and Pieter Malfroot for providing helpful technical advice, and to Lisa Martin and Kobe Geryl for proofreading.

Laurens Martin, August 2017

Abstract

Heterogeneous agents extension for Push & Rotate algorithm

by

Laurens Martin

Counsellor: Ignace Saenen

Supervisor: Prof. dr. Peter Lambert

Department of Electronics and Information Systems

Chair: Prof. dr. ir. Rik Van de Walle

Faculty of Engineering and Architecture, Ghent University

Summary

Many algorithms exist to try and solve the multi-agent pathfinding problem, but very little offer support for agents of differing sizes. In this dissertation a solver is introduced that tries to solve the multi-agent pathfinding problem for a combination of standard sized agents as well as agents equalling two by two standard sized agents. This is done by adjusting the Push & Rotate algorithm as introduced by B. De Wilde.

Keywords

pathfinding, multi-agent, heterogeneity, push and rotate

Heterogeneous agents extension for Push & Rotate algorithm

Laurens Martin, *Author,* Ignace Saenen, *Counsellor,* prof. dr. Peter Lambert, *Supervisor*

***Abstract*—Many algorithms exist to try and solve the multi-agent pathfinding problem, but very little offer support for agents of differing sizes. In this dissertation a solver is introduced that tries to solve the multi-agent pathfinding problem for a combination of standard sized agents as well as agents equalling two by two standard sized agents. This is done by adjusting the Push & Rotate algorithm as introduced by B. De Wilde.**

***Keywords*—pathfinding, multi-agent, heterogeneity, push and rotate**

I. Introduction

Multi-agent pathfinding (MAPF) is a way to calculate routes for multiple agents concurrently, while keeping unnecessary detours and conflicts between agents at a minimum. It might be used in warehouses managed by autonomous robots or in computer games featuring many different units. MAPF is an active research field and new approaches are introduced regularly, but few of these approaches take into account the diversity of the agents. Heterogeneous solvers are quite rare, and focus on single agent problems. In this paper we build upon Push and Rotate[1], a MAPF solver, to enable processing of agents of varying sizes.

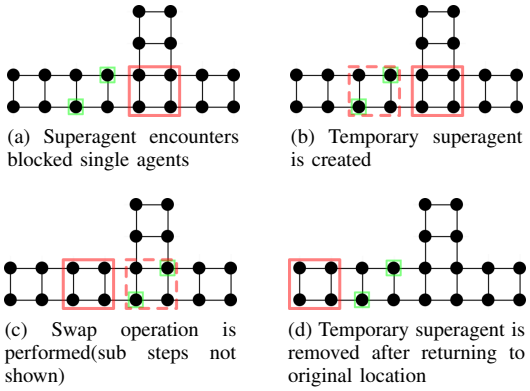
II. Algorithm

A. Base

The standard Push and Rotate algorithm relies on three base operations: 'Push' is used to try and move an agent forward along the path to its destination. Any agent encountered that has not yet reached its goal itself, is pushed out of the way. 'Swap' is used when a push operation fails. The agent currently being processed as well as the agent on the node that can't be cleared are moved towards a node with 3 or more neighbour nodes. Provided the necessary room can be made available, the two agents can swap positions through the use of such a node, allowing the original agent to continue along its path and the other agent to return to its original node. 'Rotate' is used for cycles within the graph, and can be seen as a special case of 'push' operations used when each agent except one can only move forward or backwards.

B. Modifications

Large agents, referred to as superagents, are represented on their own graph as individual agents, as well as a cluster of subagents on the original graph. These supergraphs are linked to the original graph, so that nodes on one graph can be linked to the matching nodes on the other graph. Moving a superagent one step forward corresponds to each subagent moving two nodes forward in the same direction on the original graph. In this scenario, to fully represent every possible position a two by two superagent could occupy on the original graph, it would require four supergraphs. While it is possible to represent these positions on just a single graph, these would require additional annotations to clarify which nodes can be occupied at a given time, as well as more drastic changes to the Push and Rotate algorithm, which was out of scope of this project.



(a) Superagent encounters blocked single agents

(b) Temporary superagent is created

(c) Swap operation is performed(sub steps not shown)

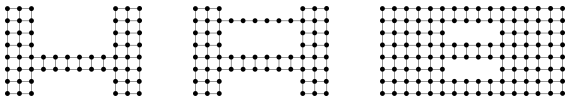
(d) Temporary superagent is removed after returning to original location

Fig. 1. Superagent swapping with blocked single agent(s)

Operations between agents of the same size needed only minor modifications. However, interactions between different sized agents need to be carefully modified to provide correct functionality.

A push operation where a single agent r tries to push a superagent S out of the way is trivial, as this comes down to finding a node the superagent can move to and moving the four subagents away in that direction at the same time. A superagent R trying to move towards a node and pushing one or more single agents s_1 up to s_4 can be done by pushing away those agents individually. In this scenario, care has to be taken on what order the subnodes are added to the list of blocked nodes.

Swapping a single agent r and superagent S is only possible if the required nodes surrounding the single agent are available. If these nodes are valid points points on the graph and are connected as needed, a swap can be performed, regardless of the current state of the other single agents occupying the neighbouring nodes. To allow this swap to happen, the single agent r is combined with any other single agents occupying the neighbouring nodes to form a temporary superagent R_t, which is then used to perform the 'swap' operation as normal. While in this temporary superagent state, the subagents assume the same shape as a normal superagent, meaning they can not



(a) Corridor (b) Shortcut (c) Multiple corridors

Fig. 2. Representation of the general layout of the different map types

share any of the subnodes occupied by R_t that are technically empty, with another single agent that is not part of the same superagent. After the initial superagent has reached its destination, the temporary agent R_t is returned to its original position and dismantled into its single agents (i.e. r and any of the valid neighbours). A representation of this process can be seen in figure 1.

To reduce potentially unsolvable conflicts between single and superagents, the order of processing is changed. The original Push & Rotate algorithm sorts agents by their subgraph assignment to guarantee that each agent will be able to reach its destination. For Heterogeneous Push & Rotate, agents are split into single and superagents, within each set the ordering as determined by the subgraphs remains intact. While this separation causes some problem instances to fail because individual agents might no longer reach their goal, it solves the problem of single agents trying to swap with blocked superagents.

III. Tests

A collection of different map types was used to test the performance of the algorithm. Types included a rectangle grid where each node was connected to its orthogonal neighbours, a map with a single corridor linking two zones, a corridor map with an additional shortcut only accessible to small agents and a map with three spaced out corridors for easy back and forth between zones. Representations of these map types can be seen in figure 2. For each map type, the width of the corridor is varied, as well as total number of agents and small to large agent ratio. For the corridor map, different lengths of corridor were tested as well.

Tests showed that Heterogeneous Push and Rotate is not able to handle interactions between the different-sized agents very well, as even for sparsely populated maps, the algorithm has trouble finding valid solutions (figure 3). While the full rectangle grid should theoretically provide best results, there's no noticeable improvement on this map compared to multi-corridor maps. When adding a shortcut to a corridor map, there's no noticeable improvement in number of solutions found, but calculation time does seem to decrease as would be expected.

Dissolving superagents into four individual agents allows one to execute PnR on the same problems as HPnR, allowing a rough comparison of calculation times. For those instances that HPnR manages to solve, calculation times are generally a bit higher than those of PnR (figure 4), with the difference becoming more noticeable as the number of agents increases. While processing a single superagent instead of 4 individual agents has a positive impact on performance, HPnR performs extra look-ups to check whether agents are part of superagents

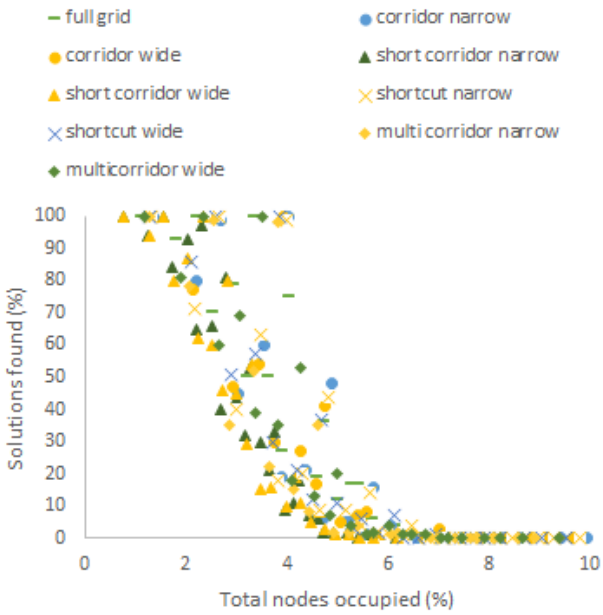


Fig. 3. Solutions found in relation to the total number of occupied nodes for different map types

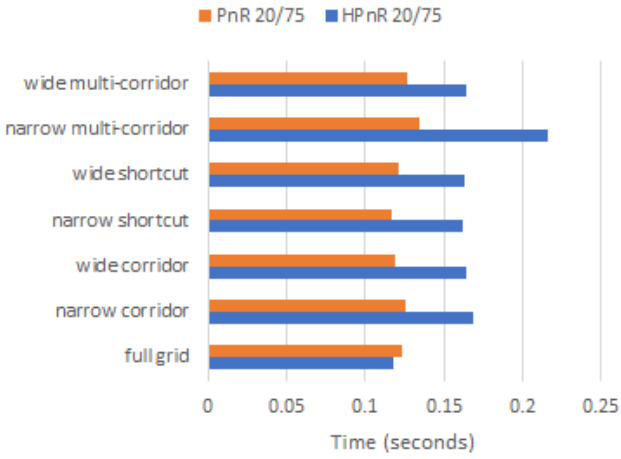


Fig. 4. Average time for a solution to be found using 20 superagents and 75 single agents for different map types

as well as to see if any subnodes might be occupied, and relies on the expensive 'swap' option more than standard PnR, as can be seen in figure 5. Therefore a decrease in performance is not wholly unexpected.

IV. Discussion

While the calculation time of HPnR falls within acceptable bounds compared to its homogeneous counterpart, the chance of finding a solution decreases drastically as the number of

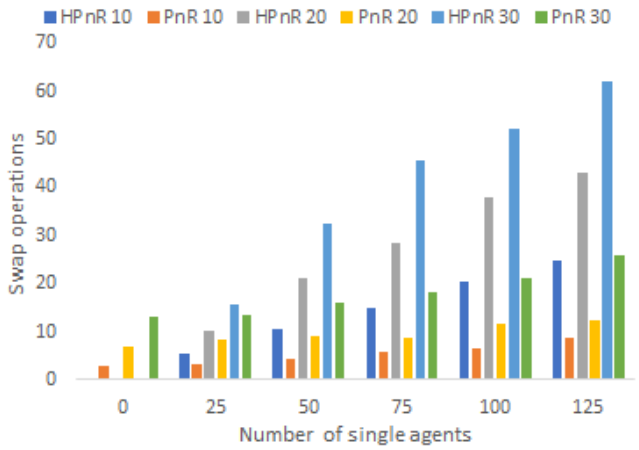


Fig. 5. Average number of swaps for problem instances on a 60 by 60 full grid map

mixed agents increases. Some failures were to be expected due to the more limiting nature of the modified operations in relation to the unmodified versions, yet the observed failure rate is too high to be solely attributed to these changes. It is suspected one of the subroutines causes synchronisation problems between single and superagents under certain conditions, resulting in a failure to correctly evaluate all movement options. However, more research is needed to confirm and solve the exact problem causing this issue. For the instances that HPnR manages to solve, calculation time and individual operations are roughly as to be expected, with 'push' operations being lower than those of normal PnR and 'swap' being higher. Both operations and look-up procedures for state checks and superagents can be further improved in a few different ways, yet any further research should logically focus foremost on tracking the issue causing the algorithm to fail in more complex situations.

V. Conclusion

In this paper a heterogeneous agents extension for Push and Rotate was proposed, but the resulting algorithm failed to obtain good results. More work is needed to create a more reliable version. The authors believe that with the right improvements and corrections to the subroutines as well as some optimisation to superagent checks, Heterogeneous Push and Rotate could become a viable solver for different-shaped multi-agent problems. However, in its current state, the algorithm is too inaccurate for practical applications.

References

[1] Boris de Wilde, Adriaan W. Ter Mors, and Cees Witteveen. "Push and Rotate: A Complete Multi-agent Pathfinding Algorithm". In: *J. Artif. Int. Res.* 51.1 (Sept. 2014), pp. 443–492.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

AHA* Annotated Hierachical A*

BFS Breadth First Search

CA Combinatorial Auction

CBS Conflict Based Search

DFS Depth First Search

FIFO First In, First Out

HPA* Near-Optimal Hierarchical Pathfinding

HPnR Heterogeneous Push and Rotate

JPS Jump Point Search

LIFO Last In, First Out

MA-CBS Meta-Agent Conflict Based Search

MAPF Multi-agent Pathfinding

PnR Push and Rotate

PnS Push and Swap

Chapter 1

Introduction

Path finding is something most of us do naturally in day-to-day life, automatically deciding which route will be most efficient to reach our goal. But path finding is a lot more widespread than simple personal route analysis. Planning transport routes for logistic companies, warehouse robots continuously storing and retrieving goods, computer characters moving across a map in a video game, they all rely on path finding to reach their destination in an efficient way. Robots, autonomous vehicles, route-managing software or other digital agents need explicit rules to determine which path would be most efficient to reach a destination.

In the standard pathfinding problem, one agent in particular is considered. This agent has to traverse a given environment, which may contain obstacles, to reach its destination. Obstacles are commonly static, often in the form of environmental factors such as walls or rivers, yet some solvers can process dynamic obstacles such as temporary barriers or competing agents trying to reach their own destination. In some scenarios, the planner might have knowledge about the routes of these other agents, but it might just as easily have no data about them at all. In any case, in the standard pathfinding problem, no attempt is made to synchronise movements between different agents, consequently collisions or deadlocks might occur.

Contrarily, in a Multi-agent Pathfinding (MAPF) problem, agent movement is coordinated either by making the agents aware of the movements of others or by having a global planner. In this manner, all agents can achieve their goal in a much more efficient way. Movement of one agent might affect the optimal route of others. When the paths of multiple agents start to overlap, it might lead to stalemates or bottlenecks. Consequently, it is in the agents best interest to gauge where other agents currently are and are going to be. In online situations, one might use prediction algorithms, real time communication or a global planner aware of every single agent to avoid these problems. In offline situations, preprocessing can be done to determine the optimal route for each agent, in relation to the routes of the others.

Pathfinding problems are commonly abstracted to represent the environment in a discrete form, generally with a graph. In such a case, the problem becomes NP-complete [1]. Generally, the edges represent parts of routes that can be taken, with nodes representing the possible locations where an agent can reside. In MAPF problems, it is assumed nodes can only be occupied by a single agent at a time, unless stated otherwise.

To determine the most efficient route, one needs to specify 'efficient' first. One may prefer the shortest, fastest or cheapest route. While in some scenarios, these criteria might lead to the same route, in many others, it will not. When planning a path across a graph, a cost is attributed to each edge. This cost relates to the criteria as selected by the user, with a cheaper cost meaning a more preferable step. Costs can be directly linked to distance, time needed or some other criterion, or it can be a weighted combination. In some scenarios, all edges have the same cost, in which case this data can be omitted and there only needs to be kept track of how many edges are in a given path. In that scenario, the shortest path is also the path with the fewest number of nodes that need to be traversed.

All agents have a start node as well as a goal node. In offline solutions, all agents are known at the start. In online solutions, agents might appear anytime during processing. Most MAPF solvers are offline.

Pathfinding is an old but still popular research field. New approaches to old problems appear regularly, trying to provide tighter bounds on the problem than those which are already available. Many of these approaches often focus on improvements for specific cases, such as for specific agent distributions or limiting themselves to certain types of graphs. They often expand upon the already well established base algorithms, tweaking and modifying certain aspects to optimise the solutions for given parameters. The most prominent example thereof is A* [2, 3] and its many variations, with A* itself being an extension of Dijkstra's Algorithm [4].

However, very few solvers take into account the potential diversity of agents, and simply assume the agents to be homogeneous, processing them all near-identically. Distinctions between agents are often seen as simply limiting factors in the possible paths, and left as potential future work. In this dissertation we will research how to process agents of different sizes effectively, using their size not only as limitation, but to make small and large agents work together to utilise the available room efficiently. As a base, the recently introduced MAPF algorithm, Push and Rotate (PnR)[5], is used. This dissertation will try to extend to allow for agents of different sizes to move across the graph without colliding, limiting each others paths or leaving unnecessary gaps.

Chapter 2

Literature Study

2.1 Single-agent solutions

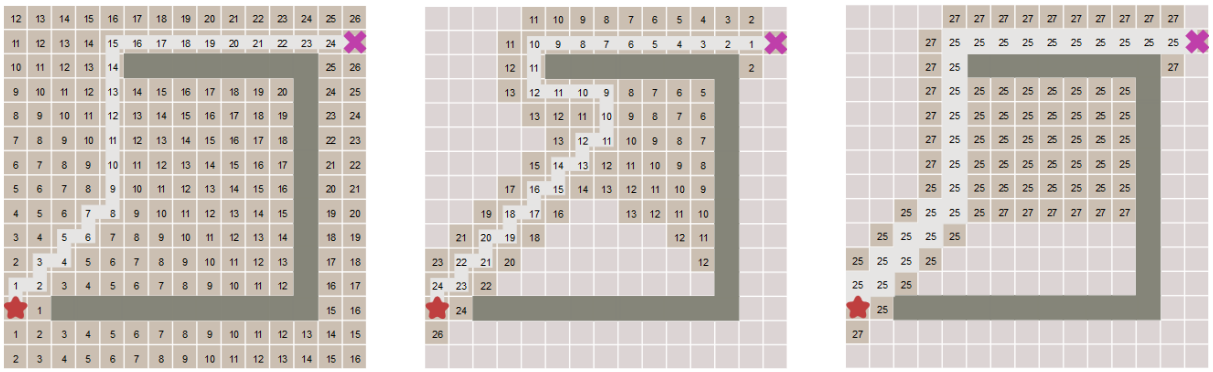
As it is not uncommon for MAPF solvers to rely on single-agent pathfinders to determine individual paths, we first take a look at the single-agent pathfinding techniques that are available. A brief overview of the different techniques is listed below, this is not meant to be an exhaustive list, but rather to give an indication of the different approaches one can take.

BFS

One of the most basic techniques is Breadth First Search (BFS). A node is taken from the frontier, i.e. the set of nodes scheduled for evaluation. If the selected node has any neighbours which are not yet part of the frontier, these are added. After evaluating the node and removing it from the frontier, the next node is selected using a first in, first out(FIFO) order. This process is repeated until the frontier is empty or the destination is reached, depending on implementation. While this technique is generally outperformed in the case of single-agent pathfinding, it can still be useful to create flow fields, distance maps or perform other general map analysis.

Dijkstra's algorithm

The BFS algorithm assumes a uniform cost for all movement steps. By introducing movement costs and implementing the frontier as a priority queue, Dijkstra's algorithm is obtained. The general idea is still the same, but nodes are sorted in the priority queue according to the path cost for moving from the start node to that node. Using a priority queue, this means that cheaper paths will be evaluated first. An example of how Dijkstra's algorithm works can be seen in figure 2.1.



(a) Dijkstra's algorithm (b) Greedy Search (c) A*

Figure 2.1: Illustrating the different solving techniques. Cells with a number indicate evaluated cells, white cells represent the final path from star to cross. Images obtained through A. Patel and Red Blob Games[6]

DFS

Similar to BFS, one can also use Depth First Search (DFS). In this algorithm, nodes are evaluated as they are added, using a last in, first out(LIFO) order, instead of the FIFO order used by BFS. The effectiveness of DFS is very situational, with a minimum bound equal to the minimum path length. However it does not guarantee an optimal solution and its upper bound is terrible, equalling the total number of nodes.

Greedy search

In greedy search, nodes are evaluated using a simple heuristic to estimate the distance between the current node and the goal node, such as manhattan or euclidian distance. Evaluated nodes are added to a priority queue, with nodes with a better heuristic score being processed first. Of course, obstacles may cause the heuristic to be inaccurate, consequently, greedy search generally finds a solution quickly, but as with DFS, it is not guaranteed to be the optimal solution. An example of how it can lead to suboptimal solutions is seen in figure 2.1.

A*

Combining greedy search and Dijkstras algorithm lead to the A*-algorithm[1], the most popular solution to the single-agent pathfinding problem.

The A*-star algorithm takes into account both the current path cost and the heuristic prediction to reach the goal from the current node. The exact algorithm is given in listing 1. The

[1] [6] provides an interactive way to learn about and understand the differences between BFS, Dijkstra's algorithm and A*.

total cost can be obtained by retrieving cost_so_far[goal], the path itself can be obtained by following came_from[goal] until we reach the start node, this gives us the path in reverse. An example can be found in figure 2.1.

Algorithm 1 A*

```
1: pq = new priority queue
2: pq.add(start,0)
3: came_from = new dictionary
4: cost_from_start = new dictionary
5: cost_from_start[start] = 0
6: while pq is not empty do
7:     current = pq.get
8:     if current is goal then
9:         break
10:    for neighbour in current.neighbours do
11:        new_cost = cost_from_start[current] + cost from current to neighbour
12:        if neighbour not in cost_from_start or new_cost < cost_from_start[neighbour] then
13:            cost_from_start[neighbour] = new_cost
14:            predicted_cost = new cost + heuristic cost from neighbour to goal
15:            pq.add(neighbour, predicted_cost)
16:            came_from[neighbour] = current
```

Introduced in 1968[2], A* is still the go-to algorithm for a quick and clean single-agent solver. Its simplicity and popularity have led to many variations and new versions are still regularly introduced[7, 8]. Some, like Near-Optimal Hierarchical Pathfinding (HPA*)[9], revert to potentially suboptimal solutions to reduce calculation time. Others, like Jump Point Search (JPS)[10], are restricted to certain types of graphs, but have no other downsides.

2.2 Multi-agent solutions

One can apply the single-agent techniques to the multi-agent problem as well. Generally, each agent is evaluated in turn, any conflicts get resolved as they occur or after all initial paths have been determined. However, to increase performance, over the years, various multi-agent specific approaches have been introduced to tackle the problem. When the agents are fully aware of the other agents paths, the term cooperative pathfinding is used.

Pebble method

In 1984, D. Kornhauser introduced the pebble method as a solution for MAPF problems[11]. It offers a complete but not necessarily optimal solution. The authors provide an upper and lower bound of $O(n^3)$ for what was then known as the pebble coordination problem. However due to

some parts lacking algorithmic implementations and a very theoretical approach, it was generally ignored and has only in recent years managed to capture the attention of other researchers as a basis for further solutions.

Amongst these solutions is the PnR algorithm, which uses Push and Swap (PnS) and Kornhauser's proofs to provide a complete solution for the MAPF problem. Both PnS and its improvement PnR will be discussed in more detail in chapter 3.

Directional maps

Instead of looking for an optimal solution to the MAPF problem, M.R. Janssen and N.R. Sturtevant introduced an algorithm[12] that combines aspects of traditional path planning with flocking approaches to create implicit cooperation and coordination between agents. When using directional maps, the algorithm looks at the movement of individual agents across individual nodes or cells. By combining the data of different agents across this cell, a direction vector is calculated for each cell, which results in a general direction map. Each cell on the map has its own vector, and path costs are calculated using these vectors. An agent trying to move across the cell in the same direction as the vector will incur no penalty, while trying to move in the opposite direction will incur the maximum penalty. Consequently, agents will no longer follow the shortest path to their destination, but will instead follow a general flow across the map, creating some small form of flocking behaviour. This approach is well-suited for environments were the agents are constantly moving, as the direction map can be updated dynamically, updating the direction vectors with each step.

Conflict Based Search

Another recently introduced solver is Conflict Based Search (CBS)[13]. This algorithm, created by G. Sharon et al., works on two levels: on the first level, conflicts between agents are detected and solved, on the second level, a path for a single agent is determined. CBS offers optimal solutions. The high level conflict tracking is done using a constraint tree with constraints relating to time and place for specific agents. At each node in this tree, single-agent searches for each agent are performed according to the constraints imposed by the node. CBS performs well in bottleneck scenarios, while being generally outperformed in more spacious environments.

To provide better solutions where CBS is outperformed by more traditional A* approaches, the authors introduced a variant called Meta-Agent Conflict Based Search (MA-CBS)[14], which combines groups of agents into a meta-agent should the number of conflicts amongst them exceed a certain bound. Furthermore, a few suboptimal variants have been introduced [15] to allow

usage of CBS in scenarios where the maximum runtime is rather limited.

2.3 Heterogeneous agents

The agents used in the solvers mentioned in sections 2.1 and 2.2 are implicitly assumed to be homogeneous, i.e. their characteristics are identical. When not all agents have the same attributes, for example different sizes or traversal speed, the problem becomes a bit more complex. Some methods might be easily extended for certain attributes. Traversal speed, for example, could be implemented by automatically marking a node occupied for a certain amount of time units, based on the agents speed. Other attributes like traversal capabilities require a different approach. The difficulty of implementing certain agent attributes depends on both the solver and the attribute itself. This heterogeneous aspect is rarely mentioned however and little actual research has been done on how these attributes would affect existing algorithms.

AHA*

D. Harbor and A. Botea introduced Annotated Hierachical A* (AHA*)[16] to offer a solution for heterogeneous agents. Cells in the grid are annotated with a terrain type as well as the size clearance for specific terrain types. This clearance n determines an n by n grid, with the annotated cell being the top left corner of this grid, in which no obstacles of that type are present. Size and traversal capabilities of an agent are then taken into account when calculating a route, allowing smaller agents to take shorter routes were possible. The basic version of the algorithm was able to process simple grids, but it was not able to handle more complex ones. Therefore the authors adopted a hierarchical approach: the original map is divided into clusters and entrances between clusters are determined as to create an abstract representation of the map. This abstraction causes AHA* to be able to process a lot more difficult problems while providing near-optimal solutions, while the search effort is lower than the traditional A* algorithm. An example of how AHA* processes a graph can be found in figure 2.2. AHA* is the only algorithm we encountered that actively tries to process agents with differing sizes on the same graph. However, it is focused on single-agent processing and seems to rely on large well connected grids with few obstacles.

Corridor Map Method

The corridor approach introduced by R. Geraerts and M. H. Overmars [17] seeks to avoid unnatural paths as created by A* and its variants, while still being able to run in real-time and being flexible for the specific needs of the agent. To do this, the solver works in two phases. In

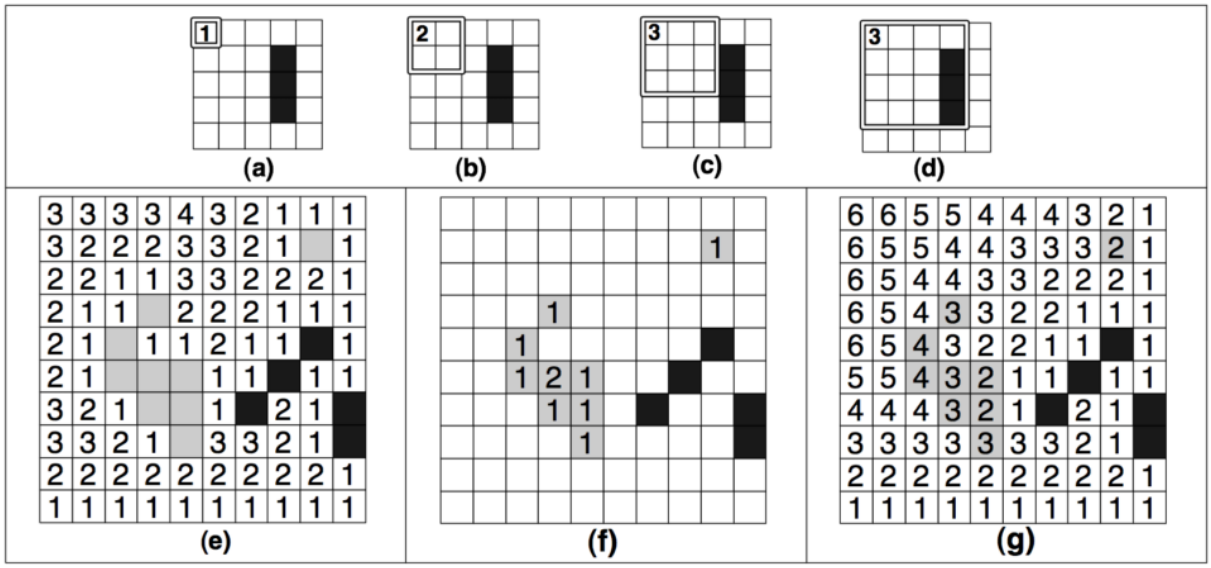


Figure 2.2: An example of AHA* processing. (a)-(d) Determining clearance for agents, (e)-(g) clearances for different environments. Image obtained through Harabor et al.[16].

the construction phase, the environment that has to be traversed is processed to create a corridor map, which consists of a collection of collision-free corridors through which the agent can move. This phase is time-intensive and should be done beforehand. Once the corridor-map is obtained, pathfinding for an agent with a specific start and goal location can be done using the querying phase. This phase runs in real-time and generates smooth and fairly natural paths for the agent to follow. Agents are modelled as a ball with a certain radius, thus allowing processing of agents of different sizes. The solver determines a backbone path, i.e. a rough outline of the path to follow, based on the specifics of the query. An attraction point is placed along the backbone path, this attraction point pulls the agent towards it. Real-time obstacles not included in the preprocessing phase can likewise be modelled using a repellent force. By integrating over time and updating the attraction point and if necessary any repellent forces at each time step, a smooth path along the corridors can be obtained. One can see some of the steps illustrated in figure 2.3.

Combinatorial Auction

O. Amir et al. took a less conventional approach to the MAPF problem and reformed it to a Combinatorial Auction (CA)[18]. In a CA, agents make bids for a combination of items that would benefit them, even though the individual items might prove useless. This logic can also be applied to the MAPF problem, where a subpath has no specific value for an agent, but a complete path from start to goal does. The CA approach allows for both cooperative and non-cooperative

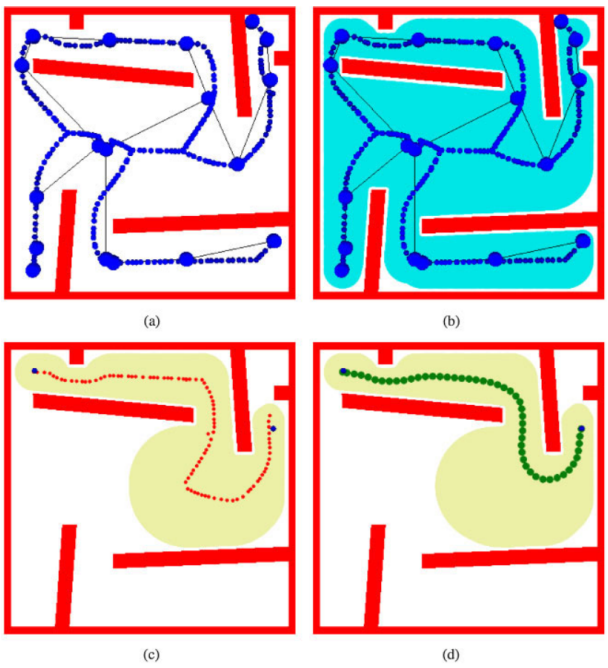


Figure 2.3: The different processing steps of the corridor map method. Image obtained through Geraerts et al.[17].

agents. Furthermore, CA allows easy use of heterogeneous agents, with agents valuing bundles that fit their attributes higher, and thus allow for increased bids. However, the attributes mentioned are fairly superficial, for example preference for shorter paths or accommodating certain time limits.

Using heterogeneity as an advantage

While these techniques adjust their solution to the specific attributes of any agent, no attempt is made to make agents use their differing attributes to their (common) advantage. Therefore it might be interesting to test if one can instead use these attributes to provide a better solution. In the ideal case, small agents would be able to use gaps between large agents as well as paths only accessible to them to reach their destination, allowing large agents to use the wide paths withouth much hindrance from smaller agents.

Chapter 3

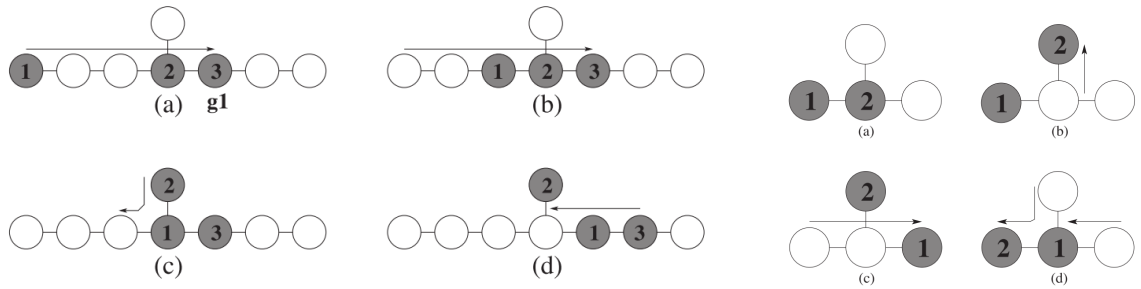
Push and Rotate

3.1 Push and Swap

The original Push and Swap (PnS) algorithm, introduced by Ryan Luna and Kostas E. Bekris [19], was claimed to be a complete algorithm for any problem where there are at most $n-2$ agents on a graph with n nodes. This statement was later disproven in [5], as will be more thoroughly explained in 3.2. The operations in Push and Rotate (PnR) are based on those introduced in PnS, therefore a brief overview of the algorithm as originally introduced in [19] is given. One can gather from the name that PnS is based around two main operations, the 'push' and 'swap' primitives. The algorithm loops over all agents, trying to push them to their goal. Should the push operation fail, a swap operation is tried instead. If the swap fails as well, the algorithm immediately ends as no solution can be found. If an agent reaches its destination, it is added to a collection $\mathcal{U}$ of finished agents and the process is restarted for the next agent.

Table 3.1: Overview of the different notations used in PnS and PnR

Type	Notation
Graph	$\mathcal{G}$
Start location of agent r	$\mathcal{A}[r]$
Goal location of agent r	$\mathcal{T}[r]$
Path from start to goal node	p^*
Set of agents that have reached their destination	$\mathcal{U}$
List of moves in the solution	Π
Cycle of nodes	c



(a) Push (b) Swap

Figure 3.1: The primitive operations of Push and Swap as illustrated in [19].

3.1.1 Push

The default 'push' primitive is used to try and move an agent r closer to its goal, as determined by its route p^*. This route is calculated using a single-agent solver such as A* and r's start and goal positions. The 'push' procedure continues as long as pushing the agent closer along its given path p^* is possible and the agent has not reached the goal node $\mathcal{T}[r]$. If the next node on the path is unoccupied, the agent can be moved forward without issue. Should another agent be occupying a node v that is part of the path towards $\mathcal{T}[r]$, the algorithm will try to move the agent occupying v out of the way, without affecting r or $\mathcal{U}$ (finished agents). This is done by calculating a path p from node v to the nearest empty vertex v_{empty}, with r and $\mathcal{U}$ as obstacles. If such a path can be found, all agents positioned upon that path are pushed forward towards v_{empty}, allowing the agent occupying v to move forward as well, freeing v for agent r. In the case that such a path cannot be found, we move on to the 'swap' primitive.

3.1.2 Swap

'Swap' comes into play when the 'push' operation is unsuccessful. This can be either because the blocking agent has already reached its goal node and therefore belongs to the set of blocked agents, or because the blocking agent can not be pushed to anywhere else. The 'swap' operation tries to move two agents r and s to a part of the graph where they can swap positions with each other. From agent r a path to a node v with degree ≥ 3 is calculated. Agent r is then pushed to v and s is pushed to a neighbour of v, through the use of a function called 'multipush'. This function is basically a collection of 'push' moves that does not take $\mathcal{U}$ into account and simply pushes everything out of the way. Two more nodes neighbouring v have to be cleared before the swap operation can continue, this is done using a 'clear' operation. If the 'clear' operation is successful, agents r and s are swapped and the agents moved through 'multipush' and 'clear' are moved back to their original position. If it is not possible to clear two other neighbouring

nodes, another node will be evaluated to perform the swap instead. If no applicable nodes can be found, the 'swap' operation fails.

3.1.3 Performance

Test scenarios were limited to a hundred agents or less, but the authors believe their algorithm to be scalable and is capable of solving instances were agents have to coordinate well in an acceptable time frame. The solution quality depends on the form of the graph. Random grids, where 'push' is the main operation, result in favourable solutions, while highly constrained graphs such as loops or corridors result in a lot of 'swap' operations, which reduces the quality of the solution. A parallel solution, allowing multiple agents to move at the same time, has been introduced as well[20]. However, as mentioned before, further research detected a few flaws in the design, which led to the creation of the Push and Rotate algorithm, which is discussed in the next section.

3.2 Push and Rotate

Boris de Wilde, Adriaan W. ter Mors and Cees Witteveen proposed the Push and Rotate (PnR) algorithm [5], which is partially build around the same operations as the PnS algorithm, and fixes some of the issues encountered when using PnS. The following problems with the original PnS algorithm were identified [21, 22]:

1. Polygon graphs: in graphs where every node has degree 2, no 'swap' operations are possible. A wrong priority ordering for the agents might result in a failure to solve an instance that is solvable.

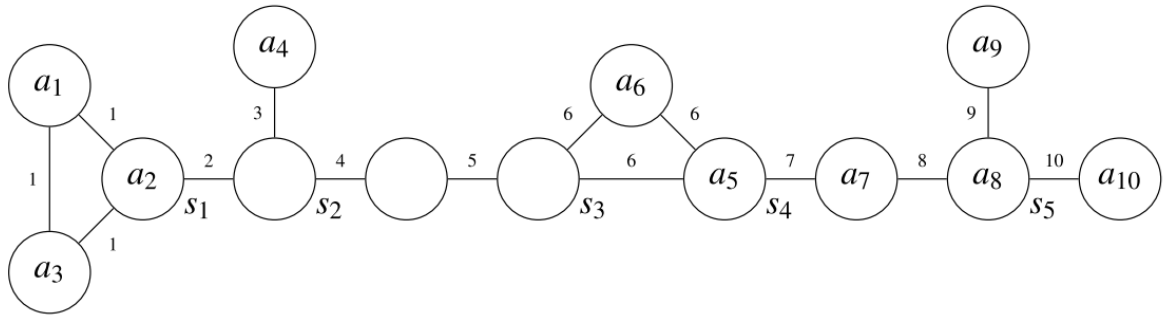
2. Incompleteness of the 'clear' operation: the original authors failed to account for two of the four possible scenarios.

3. 'Resolve' operation invoking 'swap': executing 'resolve' might lead to a high-prioirty agent being two steps removed from its destination, causing the algorithm to fail.

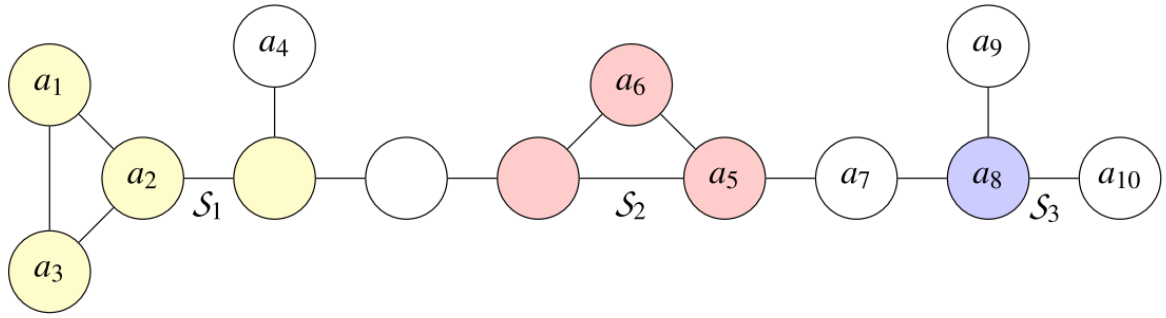
4. Failing on isthmus: in select scenarios a high priority agent might be required to move out of the way of a lower priority agent, but a swap is not possible due to the graph layout.

3.2.1 Subgraphs

Subgraphs were introduced to solve the problem of the planner failing because of an inaccessible isthmus (issue four in the list). Based on the problem decomposition theory of Kornhauser [11],



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



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

Figure 3.2: Determining the subgraphs, as illustrated in [5].

an algorithm was developed to split the problem graph into subgraphs and connecting isthmuses. Agents are then assigned to a certain subgraph, based on their proximity to it and the number of free nodes within and around the subgraph, note however that only agents which are restricted in their movement are assigned to a certain subgraph. This allows one to create priority relations between the different subgraphs, ensuring that agents are processed in an order that does not impose further constraints upon the solution. Calculation of subgraphs is partially based on the total number of occupied nodes in the complete graph, therefore the algorithm can only be run once the total number of agents is known. At this point, the exact distribution of agents is not yet required, only the graph layout and the number of free nodes. One could theoretically store the result of this step to obtain a small decrease in total run time for larger graphs, or store the result if a problem will be run multiple times for the same number of agents.

The next step in the process is assigning agents to subgraphs based on their start positions. Any agent on the inner part of a subgraph is automatically assigned to it. For agents on the joint vertices or the planks, criteria are more strict and assignment depends on the number of free nodes reachable.

Kornhauser[11] provides proof that any agent assigned to a subgraph is unable to move further than that subgraph or its planks. With this in mind, it is easy to see if the goal node of an agent does not belong to the same subgraph as its start node, the agent is unable to reach

its destination. This would mean the instance is unsolvable. Therefore the next step in the PnR algorithm is calculating the agent assignment using the goal nodes instead of the start nodes of the agents. If these distributions do not match up, the instance can immediately be declared unsolvable as some agents will be unable to reach their goal, no matter the order used to move agents.

Should the assignments match up, the priority of the subgraphs is then calculated based on agents that might restrict the movement of agents in other subgraphs. If an agent, assigned to a certain subgraph S_i, would limit the possibility of agents assigned to a subgraph S_j, subgraph S_j needs a higher priority than subgraph S_i to guarantee that the agents can reach their goals and the instance can be solved. A more detailed description of subgraph priority processing can be found in sections 3.1.2 and 3.1.3 of [5].

3.2.2 New Push and Swap

Both the 'push' and 'swap' operations are still being used in the PnR algorithm and remain largely unchanged.

Push is used to test if the next node on a path is accessible or can be made accessible. As with the PnS algorithm, we have a list of blocked agents $\mathcal{U}$, the current location $\mathcal{A}(r)$ of agent r and the node v the agent wants to move to. As in the original PnS, 'clear_vertex' Tries to find a path p along which agents can be moved to clear the node v without moving r or any agent in $\mathcal{U}$. If the 'clear_vertex' operation succeeds, the move is added to the solution. Algorithm 2 gives the pseudocode for the push operation.

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

```
1: if vertex v is occupied then
2:     U' ← U ∪ {A(r)}
3:     if clear_vertex(Π, G, A, v, U') = false then
4:         return false
5: move(Π, A,agent r to vertex v)
6: return true
```

The 'swap' operations remains relatively unaltered as well. As with PnS, it is used when 'push' fails, which happens most commonly because agent r is trying to push agent s, while s is already at its goal location and does not want to move. r and s will swap locations, after which r will continue its path and s will return to its goal node. Any node in the subgraph with degree 3 or higher is eligible to be used as swapping point. 'multipush', which can be seen as a series of 'push' operations, is used to bring agents r and s to this chosen node, ignoring any blocked agents. 'Clear' tries to free up two adjacent nodes to make the swap possible. After

swapping r and s, all moves that were made in 'multipush' and 'clear' are reversed (with the locations of agents r and s swapped) to return all agents to their original location. This means that agent s is now no longer on its goal location, but on the node where r started the 'swap' operation. However, after r has reached its goal, the path that was taken is iterated over in reverse to replace any finished agents that might have been moved through 'swap' operations, returning agents such as s to their goal destination. The pseudocode can be found in algorithm 3 Proof that the swap operation will succeed for any two agents in the same subgraph can be found in appendix B of [5].

Algorithm 3 swap($\Pi, \mathcal{G}, \mathcal{A}, r, s$)

```
1: S ← {vertex x ∈ f_s(r)| degree(x) ≥ 3}
2: for all vertex v ∈ S do
3:     A' ← A
4:     Π' ← []
5:     if multipush(Π', G, A', r, s, v) = true then
6:         if clear(Π', G, A', r, s, v) = true then
7:             Π ← Π + Π'
8:             A ← A'
9:             exchange(Π, G, A, r, s, v)
10:            reverse(Π, A, Π'_{r/s})
11:            return true
12: return false
```

3.2.3 Rotate

The 'rotate' operation comes into play when a cycle needs to be traversed. It builds upon the already established 'push' and 'swap' operations to efficiently get an agent through a cycle. Should the nodes in the cycle not be fully occupied, rotating is done using a series of 'push' movements. However, with a fully occupied cycle, a spot for a 'swap' operations needs to be found or created by clearing vertices. After a successful swap operation, the other movements can be reversed to return the agents to their original location. The exact steps can be found in algorithm 4.

3.2.4 Solve

The main solve loop can be found in algorithm 5. The outer while-loop continues as long as not every agent has reached its goal. An agent from the set of unfinished agents is taken, the shortest route is determined using a single-agent pathfinding algorithm (see section 2.1 for possible algorithms), taking into account any blocked agents in case the entire graph is a cycle (polygon outline). The while-loop on line 14 causes the selected agent to move one node forward

Algorithm 4 rotate($\Pi, \mathcal{G}, \mathcal{A}, c$)

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

with each iteration, until it has reached its destination. Cycles are solved using 'rotate' (lines 16-19), while normal path traversal is first tried using 'push' (line 21) and in case 'push' fails, 'swap' (line 22). Then, q, indicating the path the agent has just followed, is resolved. Any finished agents that were located on q have to be returned to their destination. In some cases the original destination node is unoccupied and a simple 'move' operation suffices, but should moving the finished agent back be more complex, the outer while loop is run again (line 34) with the finished agent as focus (line 30, the finished agent is assigned to r and the 'if' statement on line 7 evaluates to false, so no new agent is chosen when running the loop again). Once q has been entirely resolved, the loop continues with the next highest priority unfinished agent (line 8). Once every agent has reached its destination, the final set of moves Π is returned.

3.2.5 Performance

PnR performed considerably well in the test scenarios of the authors, and while it does not achieve the theoretical bound of $O(n^3)$ as calculated by Kornhauser, it manages to perform equally well as Bibox[23], an algorithm that reaches the $O(n^3)$ bound for biconnected graphs. It performs especially well on grid graphs with a lot of open spaces. The authors suspect that finding ways to sidestep the 'swap' operation through advanced use of rotations would reduce computation time even further.

Algorithm 5 solve($\mathcal{G}, A, \mathcal{A}, \mathcal{T}, \mathcal{S}, f, \prec$)

$\Pi \leftarrow []$
$q \leftarrow []$
$\mathcal{F} \leftarrow \emptyset$
$r \leftarrow \perp$
is_polygon $\leftarrow \forall v \in V : \text{degree}(v) = 2$
while $\mathcal{F} \neq A$ **do**
    if $r = \perp$ **then**
        $r \leftarrow$next_agent$(A \setminus \mathcal{F}, \prec)$
    if is_polygon **then**
        $p \leftarrow$shortest_path$(\mathcal{G}, \mathcal{A}(r), \mathcal{T}(r), \mathcal{A}(\mathcal{F}))$
    else
        $p \leftarrow$shortest_path$(\mathcal{G}, \mathcal{A}(r), \mathcal{T}(r), \emptyset)$
    $q \leftarrow q + [\mathcal{A}(r)]$
    while $\mathcal{A}(r) \neq \mathcal{T}(r)$ **do**
        $v \leftarrow$vertex after $\mathcal{A}(r)$ on p
        if $v \in q$ **then**
            $c \leftarrow$get_cycle(v, q)
            $q \leftarrow q - c$
            rotate$(\Pi, \mathcal{G}, \mathcal{A}, c)$
        else
            if push$(\Pi, \mathcal{G}, \mathcal{A}, r, v, \mathcal{A}\mathcal{F}) =$ false **then**
                swap$(\Pi, \mathcal{G}, \mathcal{A}, r, \mathcal{A}^{-1}(v))$
        $q \leftarrow q + [v]$
    $\mathcal{F} \leftarrow \mathcal{F} \cup \{r\}$
    $r \leftarrow \perp$
    while $|q| > 0$ **do**
        $v \leftarrow$ the last vertex on q
        $s \leftarrow \mathcal{A}^{-1}(v)$
        if $s \in \mathcal{F} \wedge v \neq \mathcal{T}(s)$ **then**
            $r \leftarrow \mathcal{A}^{-1}(\mathcal{T}(s))$
            if $r = \perp$ **then**
                move$(\Pi, \mathcal{A},$ agent s to vertex $\mathcal{T}(s))$
            else
                break inner loop, continue outer loop
        $q \leftarrow q - [v]$
return Π

Chapter 4

Heterogeneous PnR

4.1 Heterogeneity

The original PnR algorithm does not take into account agent-specific attributes. Agents might differ in size, movement speed, traversal capabilities etc. These attributes affect the possible paths an agent can take in the given graph. Including these attributes in the solver might result in a more efficient or optimal solution.

For agents differing in size, the current PnR solution would be to use a graph where each node is accessible by the largest agent, i.e. the graph would be fully determined by the size of the largest agent. However, there might be tighter passages in the environment inaccessible by this agent. That information would be completely lost in such a graph and all agent paths would be solved according to the possible routes of the largest agent instead. By taking the size information into account, one might be able to send the small agents through narrow, shorter paths, while only the agents that are too big to fit are forced along the wider, possibly longer routes.

Similar scenarios might occur in relation to the other agent attributes, though it appeared size would be most interesting to research. Therefore, the original algorithm will be modified so as to obtain advantageous solutions for the different-sized agents and allow them to move towards their goal while using space more efficiently than by simply processing the entire graph according to the size of the largest agent.

A quick overview of the different notations used in this section is given in table 4.1.

4.2 Representing large agents

The first thing to consider is how the different-sized agents will be represented.

Table 4.1: Overview of the different notations used in heterogeneous PnR

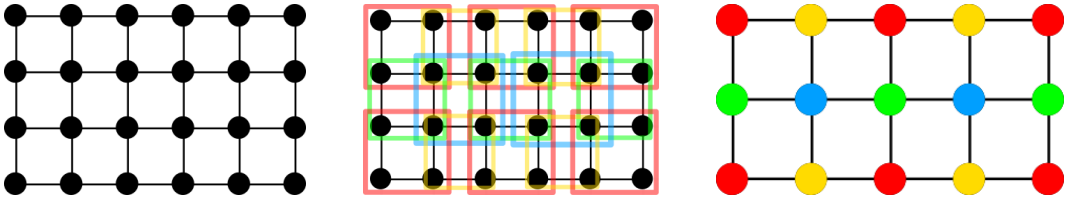
Type	Short name	Notation
the original graph	base graph	$\mathcal{G}$
a specific graph representing large agent positions	supergraph	$\mathcal{G}_1$ to $\mathcal{G}_n$
unspecified graph representing large agent positions	supergraph	$\mathcal{G}_k$
node on base graph	node	v
node on a supergraph	supernode	V
specific node on base graph as part of a supernode	subnode	v_1 to v_n
unspecified node on base graph as part of a supernode	subnode	v_k
agent on the base graph	single agent	r
agent on a supergraph	superagent	R
specific agent on base graph as part of a superagent	subagent	r_1 to r_n
unspecified agent on base graph as part of a superagent	subagent	r_k

4.2.1 Annotated nodes and agents

Perhaps the simplest solution is to add an annotation to both nodes and agents, indicating maximum allowed agent size and actual size respectively. The biggest advantage of this technique is that it allows one to process agents of any given size, no special distinction has to be made. Additionally, no assumptions are made about the graph layout or edge specifications, a trait which is lost in the other approaches. However, every agent would still occupy a single node. The graph would be defined by the size of the largest agent, meaning nodes which allow large agents to pass might be occupied by a single tiny agent.

4.2.2 Cluster of subagents

Another possibility is to still keep working on a single graph, but represent larger agents as a group of smaller subagents. This would allow one to use custom agent shapes, but makes it difficult to distinguish between a group of agents simulating a larger one and a singular small agent. An annotation can be added to indicate an agent is part of a larger whole, and data structures would allow one to keep track of which subagents are part of a given larger agent. To make this approach work, special movement operations would need to be implemented which can account for the special shapes these agents have. Moving an agent would come down to moving all subagents at the same time, in the same direction. This places some restrictions on the type of the graph that can be processed. The easiest way to make sure that a cluster of



(a) Original graph

(b) All possible locations for a superagent

(c) Indicating all possible locations on a single grid

Figure 4.1: Potential superagent locations and the graph obtained when marking all locations on one graph

subagents can fit in a new set of nodes is to use a rigid structure such as a square or hexagonal grid.

4.2.3 Superlevels

Expanding upon the subagent clusters, we consider creating new additional 'supergraphs' based on the original graph. As with the cluster version, a larger agent is represented as a group of subagents r_1 to r_n on the original graph $\mathcal{G}$, but it will also be represented as a single agent r on a processed superlevel $\mathcal{G}_1$ based upon the original graph. Creating this graph $\mathcal{G}_1$ is done in two steps: first, the nodes on the original graph are checked. Any group of nodes where a bigger agent of an arbitrary shape could fit, are indicated with a valid node on $\mathcal{G}_1$. After these nodes are created, edges between them have to be set up. To check if there is an edge between two nodes, the edges between the corresponding nodes on the original graph $\mathcal{G}$ have to be considered. If all necessary edges between these subnodes are present, an edge between the corresponding supernodes can be created as well.

While this process could be done for any given agent shape, generalising it is not trivial. Therefore let us consider the specific case of processing a graph to show us where we could fit an agent that has a size of two by two smaller agents (i.e. occupying two by two nodes) on the original graph $\mathcal{G}$.

This approach comes with a few considerable disadvantages though. Once again, it is not possible to process all graphs in such a way, and by using this technique we have to put considerable restrictions on the shape and attributes of the graph. To allow for efficient and algorithmic processing of the original graph, the graph needs to have a set pattern (grid, hexagonal) and edges have to indicate one set length.

The major issue with this type of implementation is that if a node indicates a location where an agent could be positioned, there is partial overlap with neighbouring nodes. In this project we try to process superagents of a size of two by two normal agents, as can be seen in the figures.

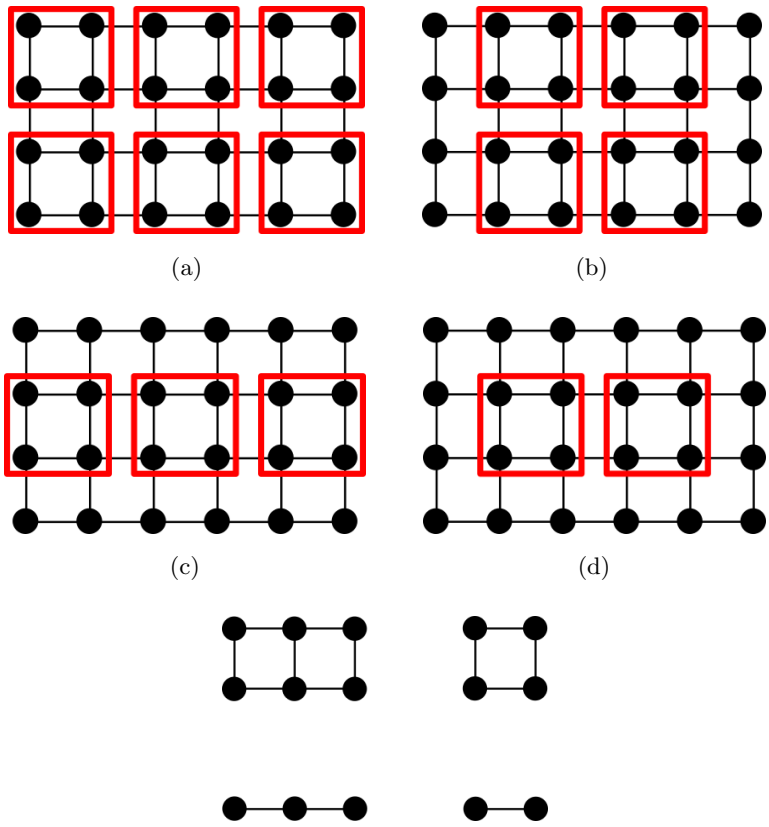


Figure 4.2: Ways of representing possible superagent locations through extra graphs, graphs (a) through (d) result in the graphs depicted in (e)

Any node that is fully connected will have overlap with at least 4 other nodes, potentially up to 8, as subnodes in corners might e shared with diagonal neighbours. Even in the most basic case where there is only one valid adjacent node, two of the four nodes will be shared. This introduces a lot of state checks when trying to move an agent along such a grid. Suppose $\mathcal{G}_1$ represents the graph where a single node n represents two by two subnodes n_1, n_2, n_3 and n_4 on $\mathcal{G}$. A node m directly adjacent to n would share two of its subnodes with n, for example any adjacent red and green nodes in figure 4.1(c). Looking again at figure 4.1(c), if an agent r currently residing on the leftmost green node would try to 'push' to the adjacent blue node, the middle green node as well as the two middle red nodes need to be checked, as they might be occupying one of the subnodes shared with the blue node. In such a case, it would be easier to work with the subagents directly. This situation would occur in any direction and similar problems arises regardless of the exact shape the larger agents might assume.

Because of these issues, another approach is considered: representing the locations of the larger agents not with a single graph $\mathcal{G}_1$ but with multiple graphs $\mathcal{G}_1$ to $\mathcal{G}_n$. In the two by two example, this would mean four graphs are used to indicate all possible positions for a large agent.

The different graphs can be seen in 4.2. Images (a) through (d) represent possible superagent locations on the original graph, while (e) shows the corresponding four resulting supergraphs. Two nodes next to each other on any of these graphs $\mathcal{G}_k$ would indicate that it is possible for two agents to be placed side by side, each occupying a single node, i.e. four nodes on the original graph. This is not the case in the single graph representation mentioned previously. Movement from one location to another would mean that when looking at the nodes on the original grid, any node that was previously occupied by a subagent r_k of agent r will no longer be occupied by any subagent of agent r. In simpler terms, moving from one node to another corresponds with a full move of 2 subnodes instead of a partial one of just a single subnode. The structural disadvantage remains, but it is no longer needed to check the other nodes on the graph, allowing the reuse of the PnR operations. In exchange, an efficient way to communicate between graphs on the same superlevel is needed. Additionally, it becomes harder to manage partial movements, as this corresponds with agents being removed from one graph and added to another. If superagents are only allowed to move in full steps, as is the case in the solution provided here, communication between graphs can be limited to checking if shared subnodes are free and making sure that any movement operations also clear the nodes on other supergraphs in addition to the supergraph currently being processed.

4.2.4 Multi-tiered superlevels

This implementation also puts certain restrictions on the shapes of the superagents. One can choose to process for example agents of a size 2x2 on the first set of superlevels, but this would mean that additional tiers of superlevels would need to represent agents that are multiples of these superagents, e.g. 4x2, 4x4 or 4x8 original agents. It would not be possible to process both agents of size 2x2 and 3x3 at the same time, as the supergraphs of these different superagents would have overlap between nodes. Using multiples would allow one to reuse the same methods, using the 2x2 superagent graphs as base graphs. However, one could choose to forgo the 2x2 superagents completely and pick 3x3 as the size of superagents in the first set of superlevels. This would allow processing of agents of sizes such as 6x3, 6x6 or 9x9 on later sets of superlevels. So, in effect, the algorithm would need to be adjusted to the specific use case. An analysis of which superagents would occur and how they relate to the small agents, size-wise, is needed for optimal performance. One does need to consider that as the number of tiers increases, the number of superlevels per tier would quickly grow to unmanageable proportions, as each of the superlevels would have multiple superlevels on the next tier, causing exponential growth. In solution presented here, focus lies on a base graph and a single supergraph representing nodes

consisting of two by two nodes on the base graph.

4.2.5 Graph limitations

The original PnR algorithm makes no assumption about the layout of the graph, but the superlevel implementation requires a set structure. Therefore it was chosen to test the heterogeneous extension of the algorithm on grid graphs, graphs which can be represented by nodes in an x rows by y columns fashion, and a node can only be connected to its direct neighbours. In practice, this allows one to efficiently target a specific node by its x and y coordinate within the grid. The number of neighbours for any of the nodes is at most four (or eight if diagonal movement would be allowed), which can be used to lower the upper bound drastically when calculating computation complexity when iterating over neighbours.

4.2.6 Free nodes

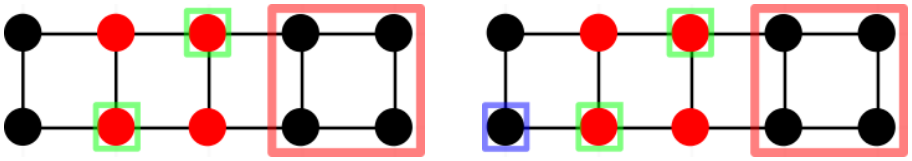
In the original PnR algorithm, to guarantee that the 'swap' operation succeeds, two free nodes are needed. This requirement carries over to the superlevels of the modified algorithm. Therefore, on the lower level, we will need two times the size of a large agent (in terms of number of subnodes occupied) number of nodes instead of simply two free nodes, should less nodes be available, the problem can immediately be deemed unsolvable.

The algorithm no longer uses a single graph, but a collection of graphs. There is the original or base graph, as well as one or more supergraphs. The solver presented here is limited to a single superlevel, but with a few adjustments, it could be expanded to work with multiple superlevels as well. These adjustments would consists mainly of checks to see if certain shared subnodes are occupied or not, as well as additional scenarios for the 'push' and 'swap' operations.

From here on onward, we will use lowercase letters to indicate nodes(v,u etc.) and agents(s,q etc.) on a the original graph, with uppercase letters representing nodes(V,U etc.) and agents(S,Q etc.) on supergraphs. To further distinguish nodes and agents on the original graph that result in a single supernode or agent on a supergraph, subscript is used, e.g. v_1 to v_4 represent the four different subnodes of a node V on the supergraph. For a complete overview, see table 4.1.

4.3 Modifying operations

Now that it is established how large agents will be represented, we investigate how the functions of PnR will have to be modified to process these agents successfully. First off, the 'move' operation is modified so that moving a superagent on a supergraph automatically moves the corresponding subagents on the original graph as well.



(a) Adding the red subnodes to the list of blocked nodes will cause the green single agents to be stuck on the red nodes

(b) Not adding the red subnodes to the list of blocked agents causes the green single agents to push the blue single agent onto the red subnodes while trying to move away from the subnodes themselves

Figure 4.3: Superagent pushing single agents: the red superagent is trying to move to the red nodes

4.3.1 Sub- and Superpush

In the situation where a small agent r, occupying node u, is trying to move to a node v and another agent s is being pushed away, it has to be checked whether s is part of a larger agent S, as in that case s can not be moved individually, all subagents s_k of S have to be moved at the same time. Instead, the algorithm switches to the supergraph on which S resides and tries to clear supernode V of which v is a subnode. To make sure that the original agent r remains unaffected, supernode U containing node u is determined and is added to the list of blocked nodes.

Pushing a large agent to a supernode that is occupied by one or more single agents proved troublesome to carry out with the basic 'push' functionality, as clearing nodes so the pushing agent can move forward, relies on a given set of blocked nodes. The subnodes that would need to be cleared, so as to make the entire supernode available, could not be added to this set of blocked nodes, as it would prevent some of the single agents currently occupying one of the subnodes, to move across another of the subnodes to another available node, as illustrated in figure 4.3(a). At the same time, these subnodes can not be left unblocked either, as this might result in other single agents being moved to any of these subnodes while the occupying subagents are being cleared, as illustrated in figure 4.3(b).

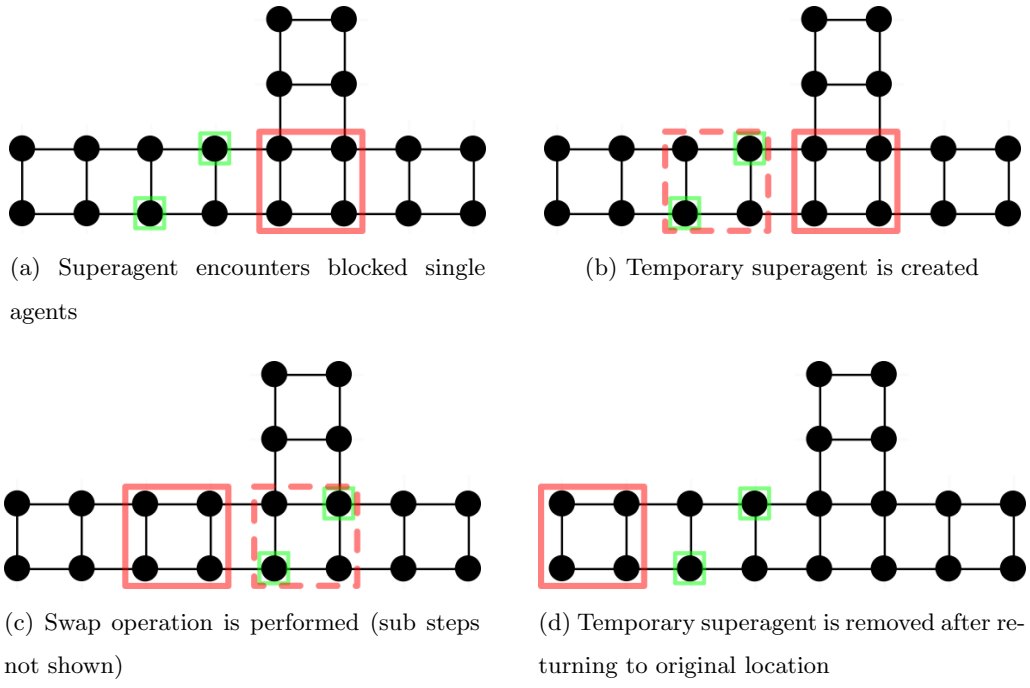
However, by changing the order in which agents are processed, i.e. processing superagents last, the scenario in which a superagent tries to push one or more single agents away no longer occurs. Instead, because the single agents have already reached their position, superagents are only able to perform 'swap' operations on single agents. Push operations with both parties being superagents can still occur, but these can be processed using a 'push' operation that is only slightly modified.

4.3.2 Swap and temporary agents

When a big agent R, which wants to move to node V, tries to swap with a smaller agent s_1, occupying a subnode v_1 of V, a temporary big agent S_t is created. This temporary agent consists of all agents currently occupying a subnode v_k of V. During the swap, these agents s_k will move together according to the movements of their temporary superagent S_t. After the swap movement is completed, and R has been to its goal location as per standard 'swap' operation, any agents that were already finished before R was being processed but had to move during the 'swap' operation, are returned to their goal nodes. This includes the temporary superagent S_t, which is moved back to its original location as well, after which it is decoupled into separate single agents once again. Because there is no rotation of any kind during these operations, this means that all agents which were part of the temporary agent end up at the exact same node as before the swap operation. An illustration of this process can be found in figure 4.4. It is important to note however, that this temporary superagent does not allow further mixing of single agents once it is created. That is to say, should this temporary agent encounter a single agent, this single agent could theoretically be absorbed into the temporary agent, provided the temporary agent does not already have four subagents. In this version the temporary agent will try to push the single agent away. This operation will fail because the single agent is part of the finished agents set (as determined by the order of processing agents), and will therefore try to perform a swap with the single agent and thus this single agent becomes a temporary superagent itself. In dense areas, this might lead to the creation of many sparse superagents, where each superagent is representing one or two single agents, but none can move because of the superagent conversion. A proper way to merge single agents into already existing super agents would prevent this problem. However finding a way to return these merged single agents to their respective goal nodes after all movement operations have been completed is not a trivial issue and out of scope of this project.

When the reverse situation occurs, where an agent r on the smaller grid is trying to move to a location currently occupied by an already finished superagent S, it might be possible that r is currently residing at a node v at the end of a corridor or pathway that is just a single node wide. In such a case, a swap with S might not be possible, as the room required for S is not available at v and consequently there is no valid supernode V containing v.

However, as mentioned previously, by changing the order in which agents are processed we alleviate this problem somewhat. By processing single agents first, as this allows single agents to push large agents out of the way to get to their destination. In cases where the superagent can not be pushed out of the way because of lack of space, the operation will still fail. It was



(a) Superagent encounters blocked single agents

(b) Temporary superagent is created

(c) Swap operation is performed (sub steps not shown)

(d) Temporary superagent is removed after returning to original location

Figure 4.4: Superagent swapping with blocked single agent(s)

considered to check if it is possible to create a temporary superagent here as well, providing the required nodes are available. In this case there would be the single agent s, currently being moved towards its goal location, encountering a superagent Q that can not be pushed, but potentially swapped with. This would mean that any single agent s_2, occupying a node v_2 that is part of the same supernode V as which s is residing on, would be forced into the 'swap' operation as well, regardless of their finished or unfinished state. If any of these agents s_i would have been finished, this would mean that at a later point, the 'swap' operation might have to be reversed so s_k is back at its goal v_k. This might result in an infinite loop in which agents are swapped to make them reach their goal. A solution for such a case consists of many edge cases, which require select solution methods. Instead, in this basic implementation, the swap will simply fail and the instance will be marked unsolvable.

4.3.3 Consequences of changing processing order

While it was already shown that we can no longer guarantee a solution for any graph with at least $n - 2$ free nodes, the change in processing order will further reduce the number of valid solutions because of blocked paths. In large open maps with well connected subgraphs, this effect should be barely noticeable however.

To summarise, single agents are processed first. This allows them to perform 'push' operations on the superagents, reducing the chance of failure because of superagents that are locked into position. Once the single agents have been processed, the solver tackles superagents. In

this phase, the interaction between superagents and single agents relies on the temporary superagents that are generated whenever a single agent is blocking the path of a superagent. Because of these temporary superagents taking up entire supernodes while they may contain anywhere from one to four single agents, a big part of the manoeuvrability of these single agents is lost. It is suspected this will lead to a decent amount of failed solutions, or alternatively to a high amount of 'swap' operations in the cases were a solution can be found.

Due to the changes in the operations, it can no longer be guaranteed that problems where the subgraph assignment of both start and goal locations match up, as discussed in 3.2.1, will actually be solvable. Instead, the algorithm might encounter situations were both 'push' and 'swap' fail for any possible path, and therefore in an unsolvable instance.

4.4 Preprocessing changes

The algorithm is given a list of single and superagents. Before solving can start, each superagent is converted to a corresponding collection of subagents and added to the list of agents on the base graph. During this conversion, the algorithm also checks for any clashing start and goal locations.

Besides the original graph, the algorithm also requires at least one superlevel graph. In general these can be derived from the original graph, but it is possible to enter custom superlevel graphs with more limited path options as well. For each graph entered, the subgraphs as discussed in 3.2.1 are determined (not to be confused with the base graph and supergraphs mentioned in this chapter), as these are needed for the ordering of all agents, as well as a to determine if the instance can be solved. To detect those graphs that can't be solved as soon as possible, the standard PnR validity check matching start and goal subgraph assignment is performed on each graph individually, as well as on the final list of single agents and super agents combined. Should any of these checks fail, the solver will be unable to provide a solution.

The algorithm creates maps to keep track of which subagents a superagent consists, as well as to which superagent on which supergraph a certain subagent belongs. Furthermore, each supergraph needs its own state, moves and list of blocked nodes to be able to determine possible paths, similar to the base graph. These collections are bundled into an extended state object and coupled with the corresponding supergraphs.

After all preprocessing is done, the algorithm iterates over all the agents of the base graph. Agents that are part of a superagent are not solved until all single agents have been processed. Because the superagents require paths that are at least twice as wide as a normal agent, there are no new restrictions on the solving of single agents. However, superagents that might have

been able to reach their destination if they were to be processed before single agents might no longer be able to reach their goal. A solution to this problem would benefit the algorithm greatly, but is not an easy issue to solve.

Chapter 5

Tests

5.1 Comparison algorithm

To test the performance of Heterogeneous Push and Rotate (HPnR), a selection of specific map layouts will be populated with a combination of both small (single) and big (super) agents. The ratio of single to super agents will be varied to see if any specific trends can be discerned. As we can perform no direct comparison to any other heterogeneous MAPF algorithm, we will use the standard PnR algorithm to solve the same instances. For heterogeneous push and rotate, 'push' and 'swap' operations are counted as one, regardless if it was a superagent or a single agent performing the operation. Each map will be tested over 100 iterations, with new randomly generated start and goal locations for both single and superagents in each iteration. Each iteration both HPnR and PnR will be executed, using the exact same list of agents. For PnR superagents will be split into their four individual subagents, their start and goal nodes remain unchanged. This means they will be able to move independently and pass through single node pathways, but will end up in the same spot as with HPnR.

5.2 Map types

5.2.1 Open grid

The open grid is a standard x by y grid. There are no obstacles, except other agents. Due to recursion in the algorithm that determines subgraphs, the size of an open grid is limited, the size of this grid is more limited than the other map types. A grid of 60 by 60 normal nodes (corresponding with a 30 by 30 supergraph) will be used to test how well the algorithm performs in what could be considered a best case scenario.

5.2.2 Corridor

A corridor is a narrow grid, were agents will often cross paths with one another. This layout should generally result in a lot of swap operations between agents. Using two wider areas at either side of the corridor should provide less conflicts as agents can move from one wide area to the other sequentially.

Two different widths of corridor will be tested, one being 4 nodes in width, allowing 2 superagents side by side, as well as one that is just 2 nodes wide, allowing only a single superagent. Additionally, the the length of the corridor will be varied as well: a short corridor of 16 nodes will be used, as well as a long corridor of 44 nodes.

5.2.3 Shortcut

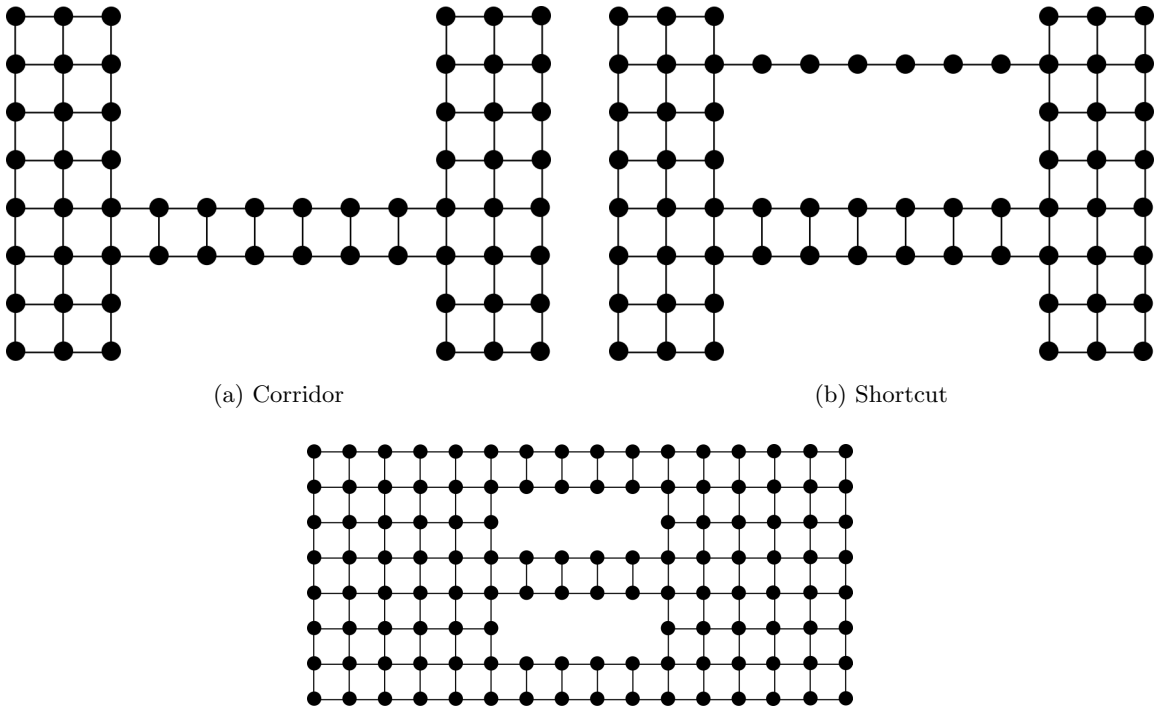
The shortcut map is used to test the efficiency of smaller pathways, only available to the small agents. This pathway is created between the two wider areas of the corridor map, parallel to the normal corridor. Small agents can theoretically forgo the wider corridor entirely in this scenario, while the large agents have no other choice but to use it. The shortcut map uses a corridor that is 44 nodes in length, the same as the long corridor map scenario. Both a two node wide as well as a four node wide corridor will be tested, but the shortcut will remain one node wide in both cases.

5.2.4 Multi-corridor

The multi-corridor map has tunnels at the top and the bottom of the map as well as one in the centre. This should reduce the number of swap operations that might be needed when trying to move from one section to the other. The map uses long hallways of 44 nodes long and varies between narrow corridors of 2 nodes wide and corridors of 4 nodes wide.

5.3 Agent distribution

For each map, different single and super agent distributions will be tested. Single agents will vary from 0 to 200 agents at a time, while superagents will range from 10 to 30. As superagents occupy 4 subnodes at a time, at most 320 nodes will be claimed at any given time. However, taking into account the temporary superagents that might be formed during ‘swap’, up to 960 nodes might be claimed as ’occupied’ in a worst case scenario.



(a) Corridor

(b) Shortcut

(c) Multi-corridor

Figure 5.1: Representation of the general layout of the different map types

Chapter 6

Results

To indicate the different series, the abbreviations for PnR or HPnR are used to indicate which solver was used. These are combined with a number to indicate how many superagents were part of the problem, or in case of PnR, the number of superagents that was transformed into four individual agents. To provide a direct comparison, the data from the PnR solver has been separated into two sets: one set uses only the data of those problems which HPnR was able to solve as well, while the other set contains the data of all instances that PnR was able to solve, regardless of HPnR's success or failure. Series simply marked as PnR X use the data from the first set and are used to compare most often. The second will be marked as 'All PnR solutions' to indicate the difference with the first set.

6.1 Full grid

First, the results for the full grid test scenario are examined.

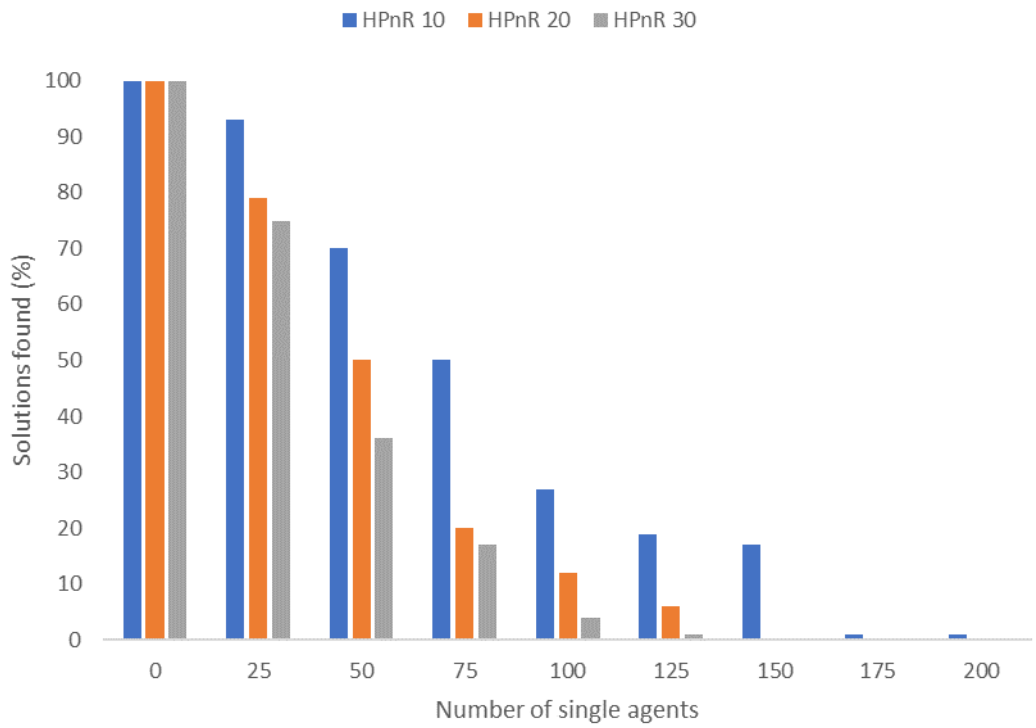


Figure 6.1: Success rate compared with agent distribution on the 60 by 60 full grid

From figure 6.1 one can see that the success rate rapidly drops as the number of agents increases. Instances where only superagents occur can be solved without difficulties, but as the number of single agents added increases, so does the failure rate. While some failures were to be expected due to the limitations of the heterogeneous operations, the success rate lies far below what was originally envisioned.

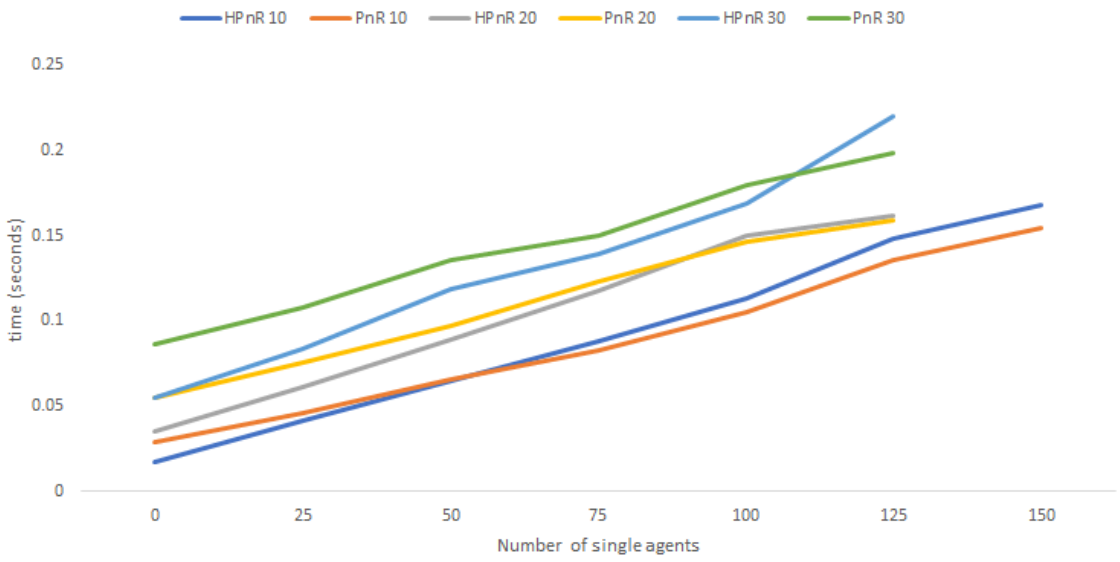


Figure 6.2: Average time needed to solve a problem instance on the 60 by 60 full grid

For those problems which HPnR managed to solve, calculation time between the solvers can be directly compared. It can be seen in figure 6.2 that while HPnR runs slightly faster when there are few single agents present, on average it does takes longer to solve a problem than PnR. The difference between the solvers becomes more pronounced as the number of agents increases. This can probably be attributed to the look-ups that are performed to see if an agent belongs to a superagent, or if a supernode is partially occupied by a single agent. The time won by having to determine one path for a superagent compared to the four individual paths that PnR calculates, does not seem to weigh up to these look-ups.

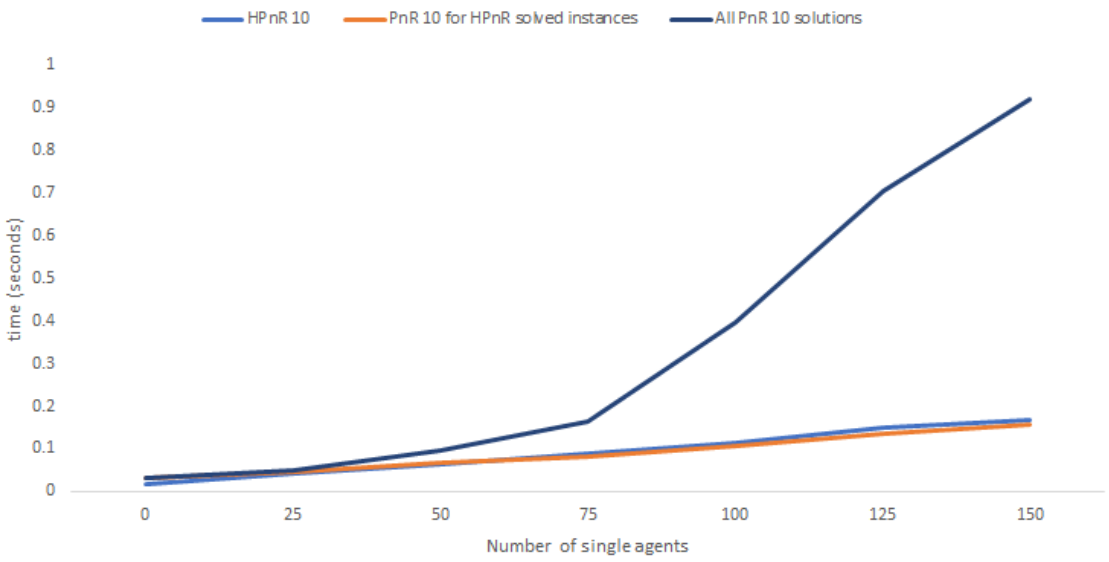
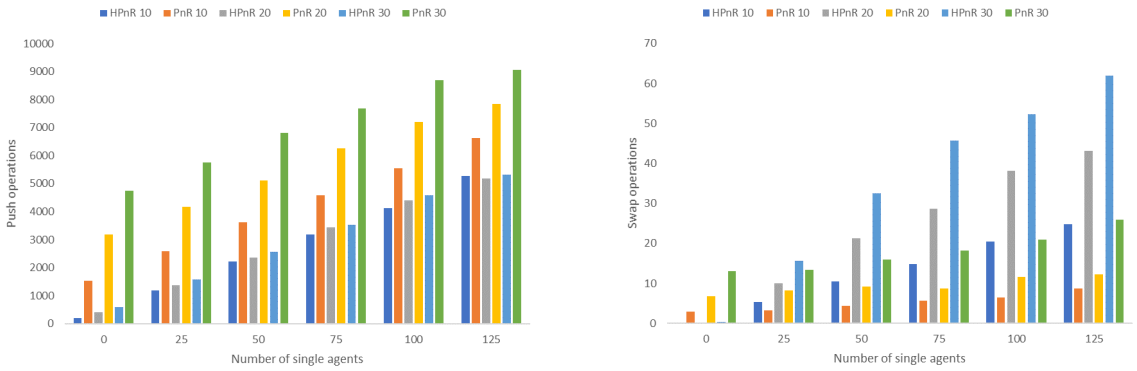


Figure 6.3: Average time needed to solve a problem instance on the 60 by 60 full grid, using 10 superagents, including all PnR solutions

When looking at all instances PnR managed to solve, instead of only those HPnR managed to solve as well, the average solution time increases drastically, as displayed in figure 6.3. This large discrepancy between these average solution times indicates that HPnR fails on those problems that are more intensive in path analysis and number of operations needed.



(a) Push (b) Swap

Figure 6.4: Average number of operations on the 60 by 60 full grid

Looking at the number of 'push' operations executed, as displayed in figure 6.4(a), expected behaviour can be observed. The number of pushes lies significantly lower for HPnR than for PnR. As the number of superagents on the map increases, the difference between the number of 'push' operations becomes even more distinct. This behaviour can be explained through two factors. First, a superagent being pushed counts as a single push operation and moves its subnodes two nodes forward at a time. Meaning one 'push' performed on a superagent can

replace eight normal 'push' operations. Secondly, through the interaction between single and superagents, as discussed in chapter 4, the 'push' operation in HPnR will fail more often than its non-heterogeneous counterpart.

In the case of 'swap', HPnR has the big lead in number of operations, shown in figure 6.4(b), again as expected. As more 'push' operations are expected to fail in HPnR, the algorithm has to rely heavier on 'swap' to solve conflicts. As 'swap' is a more expensive operation to perform than 'push', the larger number of swaps in HPnR might also be a factor in the higher computation time.

6.2 Corridor

6.2.1 Long narrow corridor

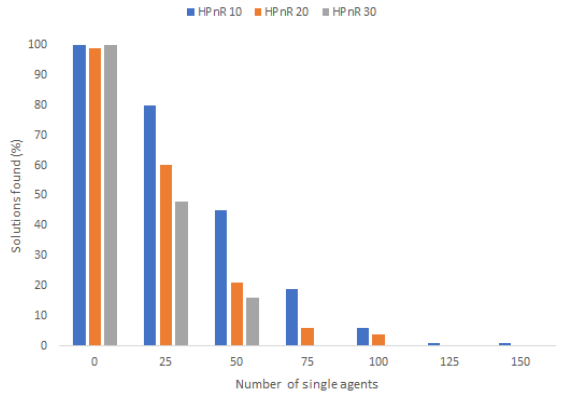


Figure 6.5: Success rate compared with agent distribution on the 80 by 80 narrow corridor map

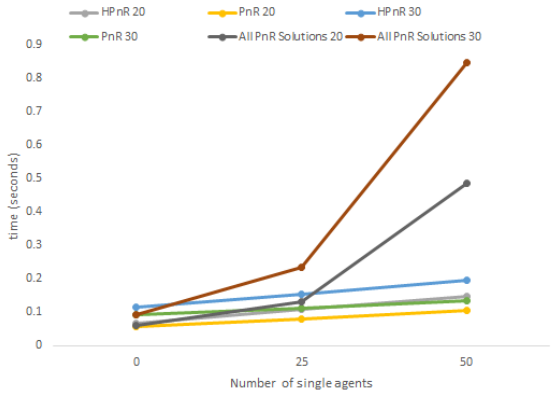
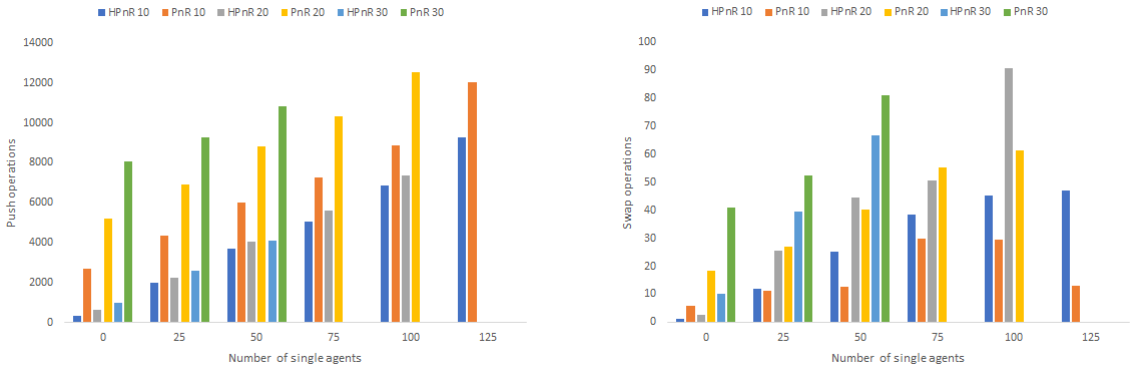


Figure 6.6: Average time needed to solve a problem instance on the 80 by 80 narrow corridor map

On the long narrow corridor map (44 nodes long, 2 nodes wide), efficiency of the algorithm decreases (figure 6.5), as the number of solutions found drops even quicker when increasing the number of agents when compared to the full grid. This was to be expected, as the narrow corridor provides no opportunity to move around others, requiring 'swaps' and clearing the entire length of the corridor of superagents. It is important to note that the narrow width of the corridor causes the supergraph to consist of two separate subgraphs, which can lead to instances being unsolvable, as discussed in [5]. This can already be seen for HPnR with twenty superagents and no single agents. One of these instances failed because of the unique path between the subgraphs.

Looking at the time to solve (figure 6.6), we observe the same trend as with a full grid.

While HPnR takes more time than PnR for the problems that both managed to solve, the average calculation time for all solutions found by PnR lies far above that, further cementing the idea that HPnR fails in the more complex problems which require more potential path evaluations and intrinsic movements.



(a) Push

(b) Swap

Figure 6.7: Average number of operations on the 80 by 80 narrow corridor map

The number of 'push' operations (figure 6.7(a)) shows the same trend as with the full grid map type. The difference in number of pushes between HPnR and PnR becomes slightly smaller when more agents are present on the map. The 'swap' operations, as depicted in figure 6.7(b), tell a different story. While the expected behaviour of a higher number of swaps for HPnR can be observed for 10 superagents, the 20 and 30 superagent scenarios show a variable number of swaps and lack a consistent trend. For 20 superagents, on maps with 25 to 75 single agents, the number is roughly equal. Only at 100 single agents, the expected pattern is observed. For 30 superagents, more swaps occur in PnR than in HPnR consistently, though the difference decreases as the number of single agents increases. A possible explanation for this is that many of the agents might be bundled up within the corridor already. This allows the temporary superagents as created by HPnR to move multiple single agents at a time, while PnR has to perform a swap with most of these single agents individually.

6.2.2 Long wide corridor

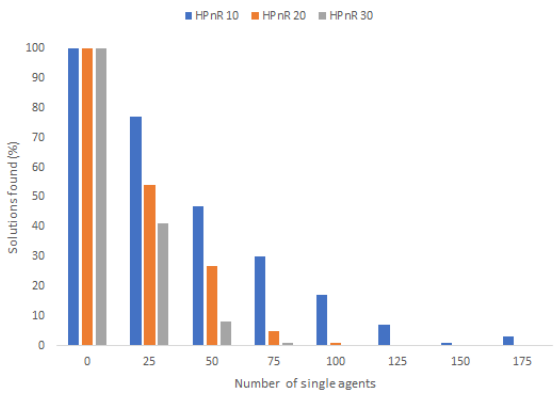


Figure 6.8: Success rate compared with agent distribution on the 80 by 80 wide corridor

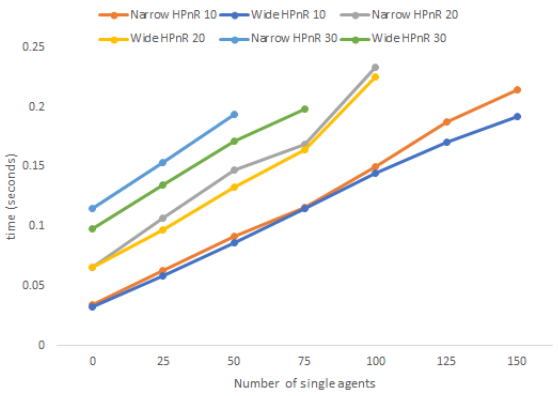
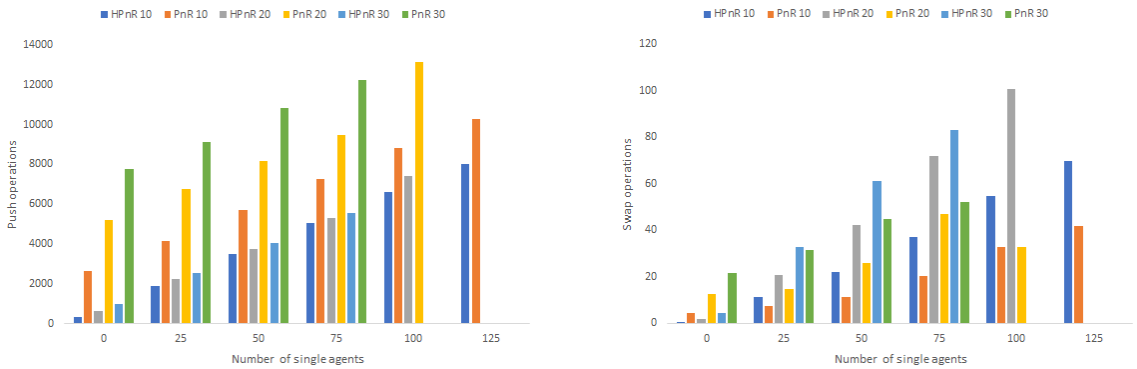


Figure 6.9: Average time needed to solve a problem instance on the 80 by 80 wide corridor

When it comes to number of operations, the wide corridor provides the results that were expected, as shown in figure 6.10. As we are now using a corridor that is 4 nodes wide, single agents have plenty of options to move to the side of other agents. 'push' operations are lower for HPnR, while 'swap' operations are higher. These results are very similar to those of the full grid, for the same reasons.

Performing a direct comparison between the narrow and wide corridor shows that the narrow corridor requires slightly more time to solve a problem, visible on figure 6.9. This is logical, as a more narrow corridor increases the chance of encountering a blocked agent and limits path options, forcing more 'swap' operations. As these operations are significantly more expensive to perform than the 'push' operation, this explains the higher solving time.



(a) Push

(b) Swap

Figure 6.10: Average number of operations on the 80 by 80 wide corridor map

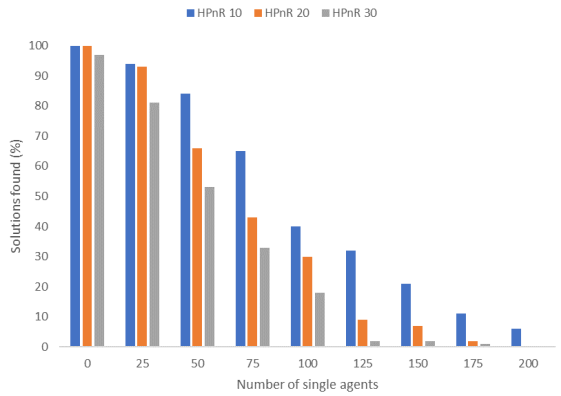


Figure 6.11: Success rate compared with agent distribution on the narrow, short corridor map

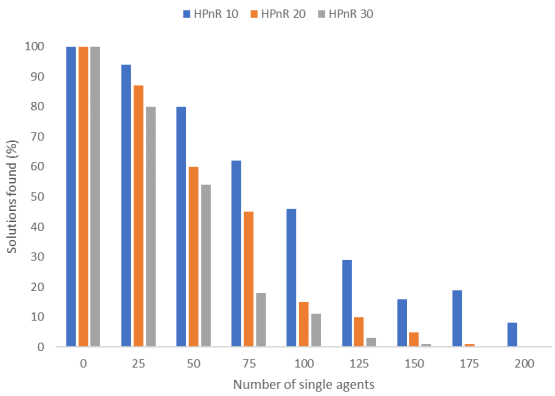


Figure 6.12: Success rate compared with agent distribution on the wide, short corridor map

6.2.3 Short corridor

Looking at figures 6.11 and 6.12, the short corridor maps seem to obtain a higher percentage of solutions found, most notably when more agents are present. While the full grid scenario offers the best case scenario for movement directions, the total number of available nodes in the short corridor scenario is significantly larger. It can be noted that the wide short corridor scenario results in slightly more failures than the narrow short corridor scenario. As the agents are randomly generated and there are only a hundred test instances for each map, this might be attributed to chance.

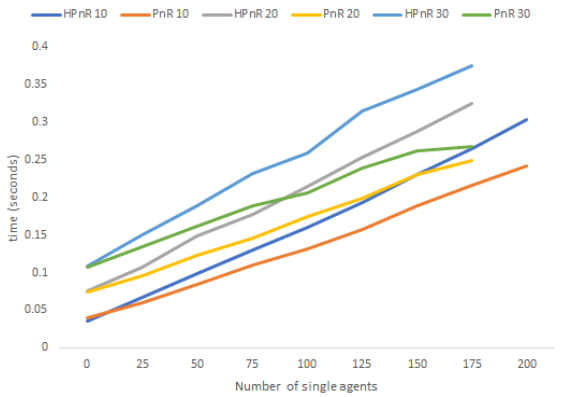


Figure 6.13: Average time needed to solve a problem instance on the narrow, short corridor

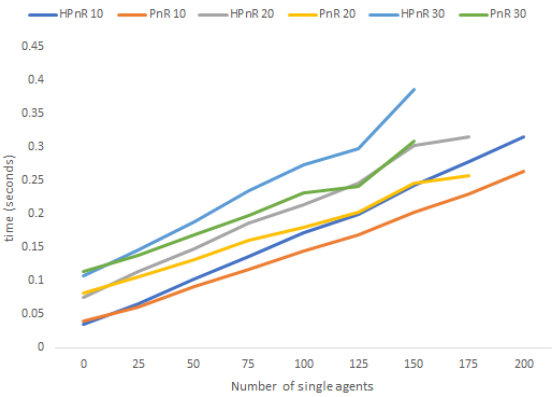


Figure 6.14: Average time needed to solve a problem instance on the wide, short corridor

The data representing the time to solve for instances in which both solvers succeeded shows no new trends (figures 6.13 and 6.14). As the number of single agents increases, the discrepancy between HPnR and normal PnR become bigger, in favour of PnR, as noted in the earlier test scenarios.

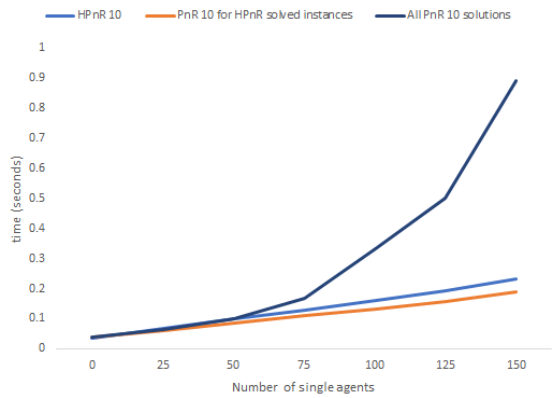


Figure 6.15: Average time needed to solve a problem instance on the narrow, short corridor

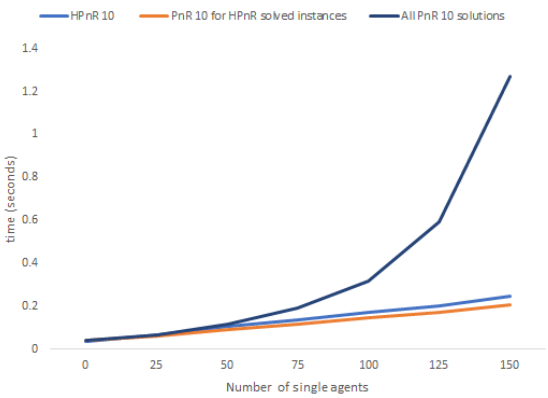


Figure 6.16: Average time needed to solve a problem instance on the wide, short corridor

Comparing the average time of instances that HPnR was able to solve to the average time of all instances shows a similar trend as with the full grid, as can be seen for 10 superagents in figures 6.15 and 6.16. While HPnR takes slightly more time than PnR on the solved instances, the average for all instances lies far above that. The difference becomes even more drastic as the number of agents increases.

6.3 Shortcut

The narrow and wide shortcut scenarios have the same length as the long corridor maps (44 nodes), so a direct comparison between the narrow (2 nodes) corridor and shortcut maps shows the effects of an added path only accessible to single agents.

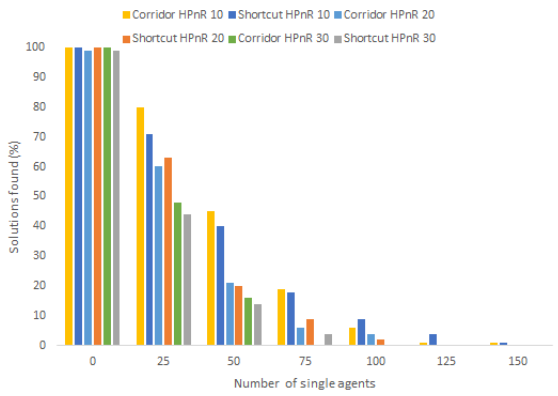


Figure 6.17: Comparison of the success rate for narrow corridor and shortcut

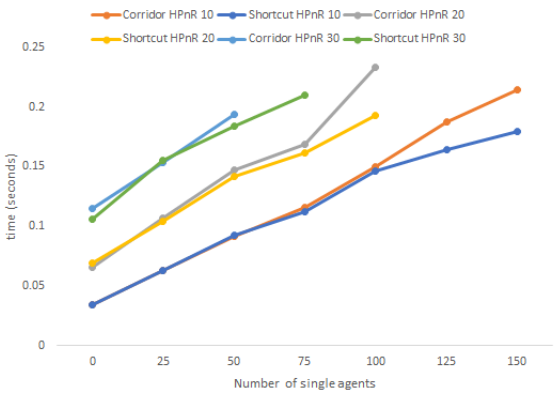
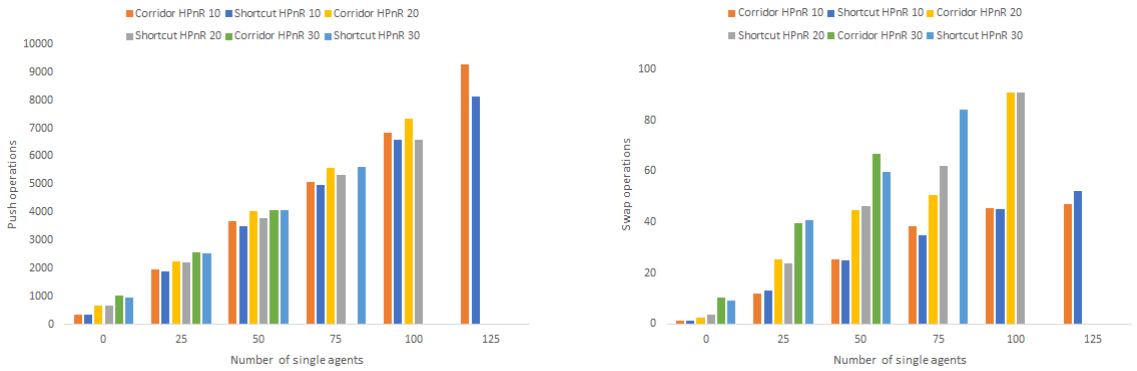


Figure 6.18: Comparison of the calculation time for narrow corridor and shortcut

Looking at the percentage of valid solutions, as indicated in figure 6.17, the difference between

a narrow shortcut and a narrow corridor is rather minimal, in some cases there were even more normal corridor problems that could be solved than shortcut scenarios. This indicates that the exclusive path does not provide benefit to the accuracy of HPnR.

Visualising the time to solve (figure 6.18), one can see that as the number of single agents increases, a difference in solve time becomes noticeable. The solver takes a bit longer to solve corridor instances than shortcut instances. This is as expected, though the effect is less pronounced than anticipated.



(a) Push

(b) Swap

Figure 6.19: Average number of operations on narrow maps

Looking at number of operations, depicted in figure 6.19, there's a small but consistent difference in number of 'push' operations, with the corridor map type requiring more operations than the shortcut map. In the case of 'swap', no real trend can be distinguished. While it was expected that the shortcut map would result in less operations, as some single agents can take a shorter path, the effect is rather small and only noticeable in the number of 'push' operations. As the maps are quite sparsely populated, the single agents will rely mostly on 'push' to reach their destination. Superagents can't use the provided shortcut, consequently processing of the superagents on both types of map is nearly identical and would bear little difference in number of operations.

6.4 Multi-corridor

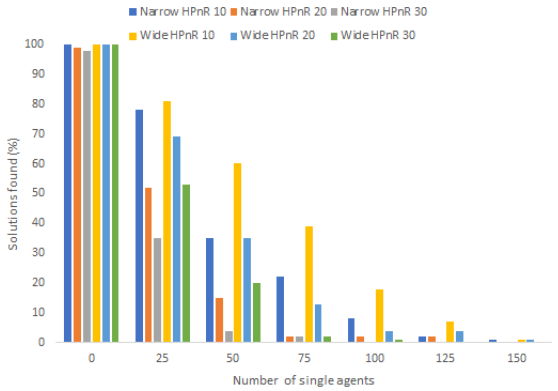


Figure 6.20: Comparison of the success rate for narrow and wide multi-corridors

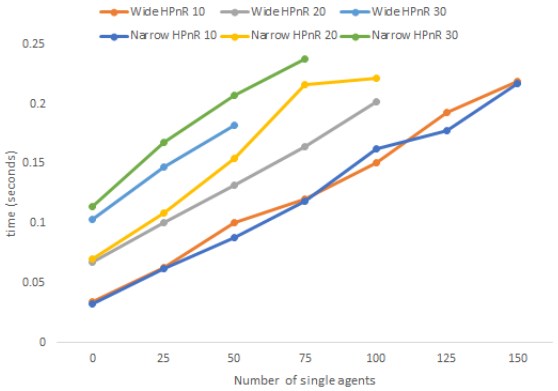


Figure 6.21: Comparison of the calculation time for narrow and wide multi-corridors

Comparing the narrow and wide multi-corridor scenarios with each other (figure 6.20) shows that on average HPnR can solve more problems on the wide map than on the narrow map. As with the normal corridor maps, wider corridors provide more space to move and less conflicts, thus this behaviour is as expected.

When looking at the solving times for both narrow and wide multi-corridors(figure 6.21), a small difference is noticeable. The narrow corridor requires a bit more time to be solved, as on average, longer paths need to be evaluated.

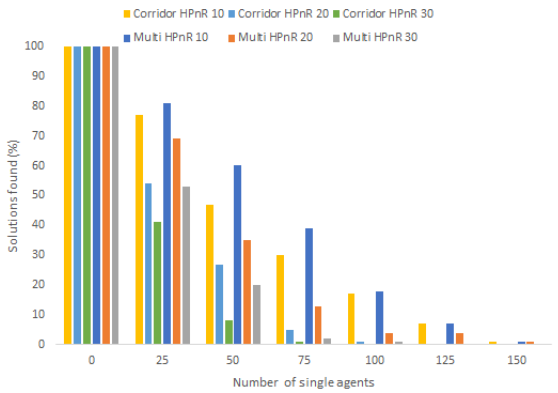


Figure 6.22: Comparison of the success rate for single and multi-corridors

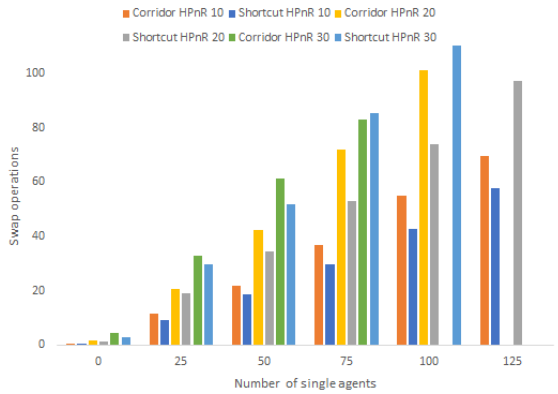


Figure 6.23: Comparison of number of swaps for single and multi-corridors

Comparing the multi-corridor to its single corridor variant, it can be seen on figure 6.22 that the extra space again results in more chance for a valid solution. While there was little difference in number of pushes, figure 6.23 shows there are a decent amount of 'swap' operations more in

single corridor solutions compared to multi-corridor problems.

6.5 Discussion

While the behaviour of the HPnR algorithm is in many cases as expected, it's capability to find solutions is drastically lower than expected. Even accounting for more limited movement and the restrictions induced by using temporary superagents, the algorithm seems to fail in fairly trivial cases. Figure 6.24 provides a visualisation of the percentage of solutions found in relation to the total percent of nodes occupied by agents. While the original PnR algorithm is able to process any problem that has more than two free nodes, HPnR fails to solve any problem with more than 7% of the map covered with agents. It was anticipated that HPnR would be able to solve most of the instances on large open spaces, with map congestion below 25% providing no major problems. In reality, accuracy of the algorithm is far below what was expected and it is clear that in its current form HPnR is not a reliable solver.

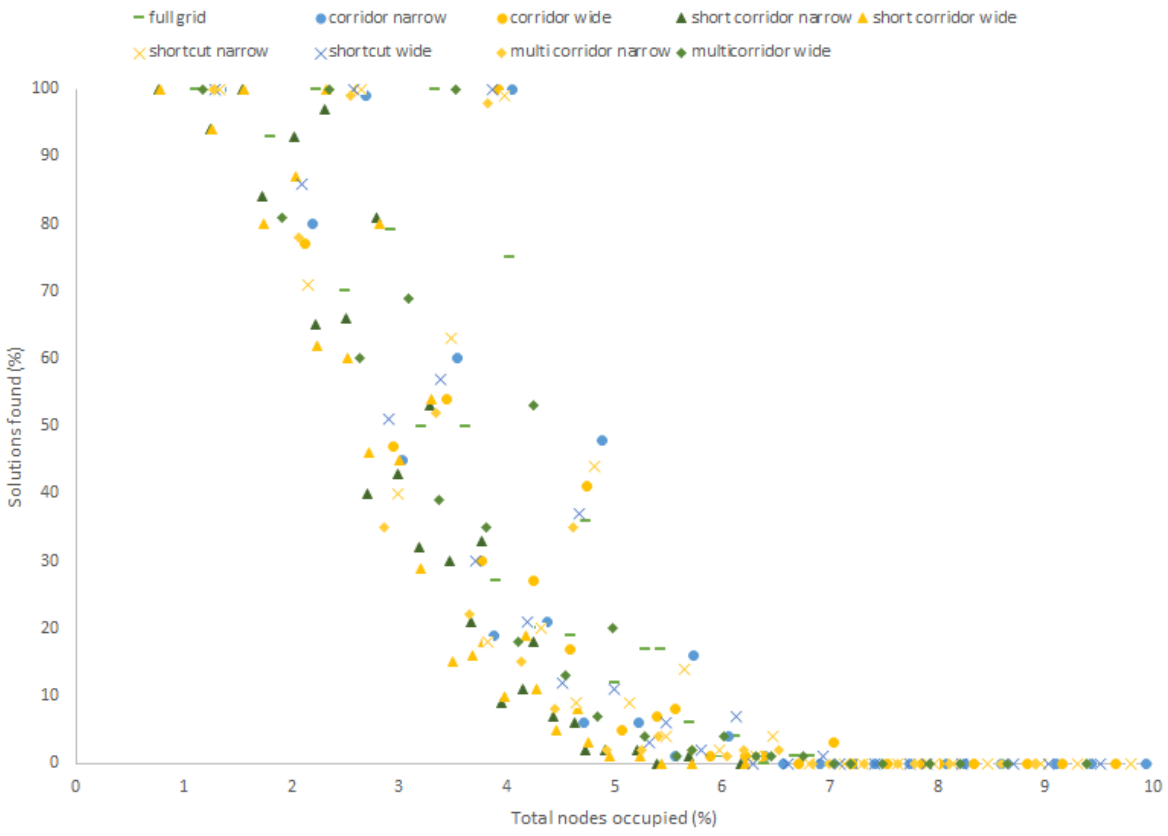


Figure 6.24: Overview of success rate for all scenarios

Based on the test results and some analysis of the code, it is suspected that the failure to process more complex problems lies in one of the subroutines used in the algorithm. Further

analysis is needed, but as of now, it is believed the problem lies with the clear_node operation, causing a mismatch between subagent and superagent location in specific circumstances. Alternatively, the swap node iterator, the code evaluating all nodes that could be used as swapping point, might be processing supernodes incorrectly, however as the problem occurs in sparsely generated maps, this seems less likely. To provide better solutions, the 'swap' operation would need to be overhauled so that temporary superagents are only created when absolutely necessary. Implementing the merging and disbanding of temporary superagents, in so far this would be possible, would further aid in providing solutions where agents are forced into bottlenecks.

Calculation time of the algorithm is within acceptable bounds, though it is believed that with optimisation, performance can be increased and solving times brought closer to the non-heterogeneous counterparts. To further increase effectiveness, the reliance on the 'swap' operation would need to be reduced. As mentioned in [5], swap is the most expensive operation in the algorithm. The current implementation of HPnR relies on swap to solve any conflict between a superagent and a single agent when processing the superagents, which has a negative impact on performance.

It is suspected that should the issue preventing HPnR from solving some of the more complex problems be found and fixed, HPnR would perform well on grids with well connected subgraphs, as calculation time is not drastically higher than that of PnR and as mentioned previously, could be reduced even further through optimisation. Expectations are a lot lower for graphs with many narrow paths, as processing agents in proper order becomes increasingly important in scenarios where the number and size of pathways is restricted. For HPnR to perform well in such scenarios, one would need to modify 'swap' to allow single agents to initiate swaps with superagents under specific circumstances, as well as find a way to efficiently order both single and superagents together. Without these changes, the current implementation would quickly block certain agents from moving to their destination through inefficient ordering and thus result in failure.

Chapter 7

Conclusion

In this dissertation an extension to the Push and Rotate algorithm that allows it to process heterogeneous agents was proposed. However, transforming Push and Rotate to a heterogeneous agent algorithm came at a serious cost and introduced some serious disadvantages. The original PnR algorithm is capable of processing any type of graph, regardless of shape or structure, as long as there are at least two free nodes. By introducing superagents that claim multiple nodes at a time, the algorithm puts restrictions on the shape of the graph, only allowing graphs in the shape of a grid to be processed correctly.

The introduction of superagents also removed the guarantee that any graph for which the subgraph assignments of both start and goal locations match up will be solvable. Instead, these assignments now only provide an early check to see if the algorithm might be able to solve the problem at hand and failure might occur at a later point in processing.

The heterogeneous adaption fails to solve problems reliably and is unable handle maps with a high concentration of agents. For those problems that HPnR does manage to solve, performance falls within acceptable bounds when compared to PnR solving a comparable problem. The solver used in our test cases uses only a single superlevel. It is reasonable to believe that extra superlevels would reduce the performance of the algorithm by a slight margin. Extra levels would translate to more state checks, more agents being affected by movement of other agents and potentially more conflicts that would need to be resolved.

Finally, considering the rather poor performance of HPnR, it might be interesting to approach the problem from a different angle than the one taken here, i.e. instead of taking a multi-agent solver and trying to add heterogeneity to it, taking a solver capable of processing heterogeneous agents and making it capable of processing multiple agents. An algorithm such as AHA* might lend itself well to this cause.

Bibliography

[1] Oded 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, Madhu Sudan, Luca Trevisan, Salil Vadhan, Avi Wigderson, David Zuckerman.* Ed. by Oded Goldreich. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1–5.

[2] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. "A formal basis for the heuristic determination of minimum cost paths". In: *IEEE Transactions on Systems Science and Cybernetics* SSC-4(2) (1968), pp. 100–107.

[3] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. "Correction to "A Formal Basis for the Heuristic Determination of Minimum Cost Paths"". In: *SIGART Bull.* 37 (Dec. 1972), pp. 28–29.

[4] E. W. Dijkstra. "A Note on Two Problems in Connexion with Graphs". In: *Numer. Math.* 1.1 (Dec. 1959), pp. 269–271.

[5] Boris de Wilde, Adriaan W. Ter Mors, and Cees Witteveen. "Push and Rotate: A Complete Multi-agent Pathfinding Algorithm". In: *J. Artif. Int. Res.* 51.1 (Sept. 2014), pp. 443–492.

[6] Amit Patel. *Red Blob Games - Introduction to A**. http://www.redblobgames.com/pathfinding/a-star/introduction.html. July 2017.

[7] Meir Goldenberg et al. "A * Variants for Optimal Multi-Agent Pathfinding". In: *SoCS* (2012), pp. 19–25.

[8] Rafia Inam and Daniel Cederman. "A * Algorithm for Graphics Processors". In: *Proceedings of the 3rd Swedish Workshop on Multi-Core Computing (MCC 2010)* (2010).

[9] Adi Botea, Martin Muller, and Jonathan Schaeffer. "Near optimal hierarchical path-finding". In: *Journal of game development* 1.1 (2004), pp. 7–28.

[10] Daniel Harabor and Alban Grastien. "Online Graph Pruning for Pathfinding on Grid Maps". In: *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*. AAAI'11. San Francisco, California: AAAI Press, 2011, pp. 1114–1119.

[11] D. Kornhauser, G. Miller, and P. Spirakis. "Coordinating Pebble Motion On Graphs, The Diameter Of Permutation Groups, And Applications". In: *Proceedings of the 25th Annual Symposium onFoundations of Computer Science, 1984*. SFCS '84. Washington, DC, USA: IEEE Computer Society, 1984, pp. 241–250.

[12] M Renee Jansen and Nathan R Sturtevant. "Direction Maps for Cooperative Pathfinding". In: (2008).

[13] Guni Sharon et al. "Meta-Agent Conflict-Based Search For Optimal Multi-Agent Path Finding". In: *Proceedings of the Fifth Annual Symposium on Combinatorial Search* (2012), pp. 97–104.

[14] Guni Sharon et al. "Conflict-based search for optimal multi-agent pathfinding". In: *Artificial Intelligence* 219 (2015), pp. 40–66.

[15] Max Barer et al. "Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem". In: *Frontiers in Artificial Intelligence and Applications* 263.SoCS (2014), pp. 961–962.

[16] Daniel Harabor and Adi Botea. "Hierarchical path planning for multi-size agents in heterogeneous environments". In: *2008 IEEE Symposium on Computational Intelligence and Games, CIG 2008* (2008), pp. 258–265.

[17] Roland Geraerts and Mark H. Overmars. "The corridor map method: a general framework for real-time high-quality path planning". In: *Computer Animation And Virtual Worlds* 19.August (2008), pp. 271–281.

[18] Ofra Amir, Guni Sharon, and Roni Stern. "Multi-Agent Pathfinding as a Combinatorial Auction". In: *Proceedings of the 29th AAAI Conference on Artificial Intelligence* JANUARY (2015), pp. 2003–2009.

[19] Ryan Luna and Kostas E. Bekris. "Push and Swap: Fast Cooperative Path-finding with Completeness Guarantees". In: *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume One*. IJCAI'11. Barcelona, Catalonia, Spain: AAAI Press, 2011, pp. 294–300.

[20] Qandeel Sajid, Ryan Luna, and Kostas E. Bekris. "Multi-Agent Pathfinding with Simultaneous Execution of Single-Agent Primitives". In: *SOCS*. 2012.

[21] Boris de Wilde. "Cooperative mutli-agent path planning". MA thesis. Delft University of Technology, 2012.

[22] Boris de Wilde, Adriaan W. ter Mors, and Cees Witteveen. "Push and Rotate: Cooperative Multi-agent Path Planning". In: *Proceedings of the 2013 International Conference on Autonomous Agents and Multi-agent Systems*. AAMAS '13. St. Paul, MN, USA: International Foundation for Autonomous Agents and Multiagent Systems, 2013, pp. 87–94.

[23] Pavel Surynek. "A novel approach to path planning for multiple robots in bi-connected graphs". In: *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*. IEEE. 2009, pp. 3613–3619.