Extending CBS to efficiently solve MAPFW

Noah Jadoenathmisier[1]
Supervisors: M.M. de Weerdt[1] , **J. Mulderij**[1]
[1]Delft University of Technology
N.J.M.Jadoenathmisier@student.tudelft.nl,
{M.M.deWeerdt, J.Mulderij}@tudelft.nl

Abstract

Multi-Agent Path Finding with Way-points (MAPFW) is the problem of routing agents through a graph past a set of waypoint to a goal location, without agents colliding, with the shortest combined path length. This problem has to the authors knowledge not been investigated yet even though it has implications in train scheduling problems and video game artificial intelligence. In this paper an extension to Conflict Based Search (CBS), an algorithm that solves Multi-Agent Path Finding problems without waypoints, is proposed to solve MAPFW problems, named CBSW. The effect of extending the Bypass optimization and the Prioritizing conflicts optimization for CBS to CBSW is investigated and a new optimization that improves the performance of CBSW in corridors is proposed. CBSW is compared with other MAPFW solvers that have been developed made simultaneously, and experimental result show a large speed up using CBSW on benchmarks with large maps or many waypoints.

1 Introduction

Multi-Agent Path Finding with Way-points (MAPFW) is an extension of the Multi-Agent Path Finding problem (MAPF). MAPF is the problem of finding a set of paths for a set of agents on a graph from start to goal locations in a way that the agents never are on the same vertex at the same time, and never cross the same edge at the same time [1]. In the MAPFW problem each agent also has a set of waypoint locations that it has to visit before ending at its goal location. This paper focuses on optimal solutions. A solution is optimal if the combined length of the paths for all the agents is minimal. Since MAPF is NP-hard [2], and MAPF is a restriction of MAPFW, it is MAPFW without waypoints, MAPFW is also NP-hard.

An optimised solver for MAPFW is currently needed because it can aid in the planning of trains [3]. This is now performed manually, which is very time consuming. An optimized solver could also be used in games for the computer's artificial intelligence [4].

The MAPF problem has been deeply researched. Algorithms created that solve MAPF problems include A*-OD-ID [5], Branch-and-cut-and-price [6], Multi-label A* [7], Increasing cost tree search [8] and Conflict Based Search (CBS) [9]. These all have their own strengths and weaknesses. To the authors knowledge no such research has been done yet to solve the MAPFW problem.

Because of the the promising results that have been achieved with CBS, and improvements thereof, CBS serves as the base of the algorithm that is presented in this paper. In the proposed extension, Conflict Based Search with Waypoints (CBSW), dynamic programming is used to calculate efficient heuristics. The Bypass optimization [10] and the Prioritizing conflicts optimization are used to speed up CBSW. A new optimization, the Corridor constraint optimization, is proposed, that helps with solving train routing problems. The performance is evaluated by comparing it with A*+OD+ID+W which is a solver based on A* with operator decomposition and independence detection, and BCP-W which is a solver based on Branch-and-cut-and-price. Both these solvers have been developed at the same time as CBSW and are currently the only other optimal MAPFW solvers [11] [12]. Experimental results show performance gains by using CBSW over the the A*-OD-ID-W and BCP-W in benchmarks that have many waypoints, and/or use large graphs.

In this paper, first the MAPFW is explained and a definition of MAPFW is given. Then CBS, the base algorithm for CBSW is explained. Next the new low-level solver that handles waypoints is explained. After this, different optimizations are proposed. Those optimizations are compared in the experimental results section. In this section CBSW is also compared to A*+OD+ID+W and BCP-W. Then a reflection is given about the reproducablility and ethical issues concerning this research. After this the limitations to this paper are stated. Finally a conclusion and suggestions for future work are given.

2 Problem definition

Multi-Agent Path Finding with Waypoints (MAPFW) is an extension of the Multi-Agent Path Finding problem (MAPF). In the MAPF problem [1] there is a set of k agents, labeled $a_1, \ldots, a_k$ and a graph $G = (V, E)$. Each agent a_i has a starting location s_i and a goal location g_i. Each agent is required to move from its start to its goal location. To get there, each time step every agent can either move to a neighbouring vertex on the graph, or stay on the same vertex. It is not allowed for two agents, a_i, a_j to be at the same vertex v at time t. This is called a vertex conflict, and denoted by $\langle a_i, a_j, v, t\rangle$. It is also not allowed for two agents to cross the same edge in opposite directions in the same time step. This is called an edge conflict, and with agent a_i starting on vertex v and moving to vertex u, and agent a_j starting at vertex u and moving to vertex v both starting at time t and arriving at time $t + 1$ is denoted by $\langle a_i, a_b, v, u, t\rangle$. The extension to MAPF assigns a set of way-point vertexes $wp_i = \{v_1, \ldots, v_k\}$ to each agent a_i. All agents need to pass all of the waypoints that they are assigned to on the path to their goal. There are no restrictions on the order in which the agents need to visit their waypoints.

3 Conflict based search

The basic CBS algorithm consists of two parts, the high, and low-level algorithm. In the high-level algorithm, a tree is constructed. Each node N consists of a solution $N.Sol$ that consists of all the paths of all the agents, a cost $N.Cost$ and a set of constraints $N.Const$. There are two types of constraints, vertex constraints, where agent a is not allowed to be on vertex v at time t, denoted by $\langle a, v, t\rangle$ and edge constraints where where agent a is not allowed to cross an edge (v, u) at time interval $\langle t, t+1\rangle$, denoted by $\langle a, v, u, t\rangle$. The root node of the tree is generated by generating a solution using the low-level algorithm. This algorithm plans for each agent individually, and does not take other agents into consideration. The root does not have any constraints. The cost is calculated by summing the lengths of all individual paths of all agents. If there is a conflict between 2 agents, 2 child nodes with constraints are generated. These constraints prevent the same conflict from occurring in the child nodes.

In case of a vertex conflict, $\langle a_i, a_j, v, t\rangle$, the child nodes both have a vertex constraint. Respectively $\langle a_i, v, t\rangle$ and $\langle a_j, v, t\rangle$. In case of an edge conflict $\langle a_i, a_b, v, u, t\rangle$, the child nodes both have an edge constraint $\langle a_i, v, u, t\rangle$ and $\langle a_j, u, v, t\rangle$ respectively. Both child nodes will use the low-level algorithm again to re-plan the route of the constrained agent honouring its constraints, including the new constraint. When no more conflicts are present, a solution is found. Nodes are expanded in a best first order based on their cost. This ensures that the solution that is found is optimal.

The low-level algorithm uses A* to find a path for an agent. The state space contains the location of the agent, and the time. The time is necessary because the constraints added to agents are only applicable on a specific time step. The heuristic used to speed up the search is the length of the shortest path to the goal from the current position, when ignoring all the constraints. This heuristic function can be easily precomputed and stored in memory for all grid locations and all goal locations.

The number of nodes that has to be expanded in the high-level search increases when the number of conflicts that occur because of the low-level search increases. To reduce the number of conflicts in the low-level search, a conflict avoidance table is used. While planning a path for an agent, if two or more nodes have the same cost, ties are broken by expanding the node that results in the lowest number of conflicts.

4 A new low-level solver

The low-level algorithm of CBS uses A* to find a path for an agent, that does not conflict with any of its constraints. This A* search needs a new state space, since for MAPFW problems, it is necessary to cross all the waypoints of the agent before ending at the goal location. Therefore the nodes of the A* search should contain:

- Time
- Location
- Unvisited waypoints

Adding waypoints to the state space greatly increases the number of elements in the search space for A* with a factor of 2^n for an agent with n waypoints. Using the heuristic of traditional CBS, the shortest path length to the goal, even small problems become infeasible because of this increase in the number of elements in the search space. Therefore a new heuristic is needed.

In MAPFW the length of the shortest path to the goal location, when ignoring all other agents also includes all the waypoints. Since traditional CBS uses the length of the shortest path to the goal as its heuristic, ignoring the constraints, CBSW uses the length of the shortest path that passes all waypoints and ends up at the goal location while ignoring the constraints as its heuristic.

Given n waypoints, the powerset contains 2^n combinations of unvisited waypoints. The heuristic can therefore not be pre-computed for all sets of unvisited waypoints for large n, as this would take to much computing time and memory. It would also not be beneficial, since in practice most unvisited waypoint combinations will never be considered.

To solve this problem only the following is pre-computed

- The distance from any location to any goal location
- The distance from any location to any waypoint location

Now, on every step of A*, the heuristic value can be computed using a modified Traveling Salesperson Problem (TSP)

solver [13]. This TSP solver receives a starting waypoint from the set of unvisited waypoints, the set of unvisited waypoints and the goal location. Then it returns the length of the shortest path that starts on the starting waypoint, crosses all the other waypoints and ends at the goal location, while ignoring other agents.

It is unknown which waypoint should be visited first, but it has to be one of the waypoints in the open set. Given one of these waypoints as a candidate for the next waypoint that should be visited, $wp1$, the minimum length of the path from the current positions past all waypoints to the goal, is calculated by taking the pre-computed minimum distance from the current position to $wp1$, and adding to that the minimum path length from that waypoint past all other waypoints to the goal location which can be computed with the TSP solver.

This length is calculated for all open waypoints as $wp1$. The shortest of these path lengths is the shortest path from the start to the goal past all waypoints, and is thus used as the heuristic value.

Algorithm 1: Waypoint heuristic

```
Result: Heuristic value
Input: current_location, unvisited_waypoints, goal;
best = inf;
foreach waypoint in open_waypoints do
    dist = distance(current, waypoint);
    dist += TSP(waypoint, open_waypoints, goal);
    best = min(best, dist)
end
return best
```

To save on computing time the TSP function is memoized [14]. This can be done with minimal memory overhead since this function is called repeatedly mostly with the same arguments. On randomly generated 64 by 64 4-connected grid based benchmarks with 20% obstacles as described in the results section. The memoization hit rate can be upwards of 99.9%.

This distance is a lower bound on the total distance that the agent should still travel when it also needs to avoid other agents, and therefore can be used as an admissible heuristic. For problems with many waypoints, but few conflicts, solving TSP dominates the run-time of the algorithm. Therefore careful attention should be payed to its implementation. Two TSP solving algorithms are tested and compared.

5 TSP solver

5.1 Branch and bound

The first algorithm that was tested to solve TSP uses Branch and bound [15]. Here, a bidirectional distance matrix is first generated. This matrix consists of all the distances between all the unvisited waypoints, and the unvisited waypoints and the goal location, collectively named cities. The distance from any city to itself is set to infinity and the distance from the goal location to any of the waypoints is also set to infinity, except for the starting waypoint. That distance is set to zero. This ensures that the shortest tour that is found goes from the start waypoint, past all the other waypoints to the goal location and back to the starting waypoint. Since the distance from the goal location to the starting waypoint is zero, the length of this tour is equal to the length of the shortest path from the starting waypoint past all the other waypoints to the goal location.

After generating the distance matrix, a search tree is generated. Every node n, of the tree contains a reduced distance matrix $n.dist$, a position $n.pos$, a cost $n.cost$ and , a depth $n.depth$. The cost contained in each node represents a lower bound on the length of the shortest tour. In the reduced distance matrix for n the cost are stored of going from each city to each other city minus the cost that has already been accounted for in $n.cost$. The position is the latest city that has been visited, and the depth is the amount of cities that have already been visited.

In a tour, every city has to be preceded by another city and, every city must be followed by another city. For every city, n, a lower bound can be given to the cost of the edge that connects n with the city before n in the tour. This lower bound is named $enter_n$. A lower bound can also be given to the cost of the edge that connects n with the city after n in the tour. This lower bound is named $exit_n$. $enter_n$ is calculated by taking the lowest value in the row corresponding to n in the distance matrix. $exit_n$ is calculated by taking the lowest value in the column corresponding to n in the distance matrix.

To reduce a distance matrix $distmat$, for every city c, $enter_c$ is subtracted from every element in the row corresponding to c in $distmat$. This is followed by subtracting $exit_c$ from every column corresponding to c in $distmat$ for every city c. The $reduction\_cost$ is the sum of $enter_c$ for all cities c added to the sum of $exit_c$ for all cities c.

The root node r of the search tree, contains the result of reducing the distance matrix as $r.dist$, and its cost is the $reduction\_cost$ of the distance matrix. The position is the starting waypoint and the depth is zero.

Nodes are expanded in a best first order in terms of their cost. When expanding a node n, all non-visited cities are considered. These non-visited cities are found by looking at $n.dist$. The non-visited cities are all the cities w for which the edge in $n.dist$ from $n.pos$ to w does not have a cost of infinity. For every city u that is not visited yet, a new node $child$ is generated. The depth of $child$ is d plus one, and the position of $child$ is u. The reduced distance matrix $n.dist$ copied to $child$. All the elements in the column of $n.pos$ and in the row of u in the distance matrix of $child$ are set to infinity. This matrix is reduced, with a reduction cost $reduction\_cost$. Finally, the cost of $child$ is set to the sum of $cost$, $reduction\_cost$, and the value corresponding to going

from $n.pos$ to u in $n.dist$.

If a node is expanded with a depth equal to the number of cities, its cost is the solution for the cost of the shortest tour, and, therefore equal to the length of the shortest path that starts on the starting waypoint, crosses all the other waypoints and ends at the goal location.

Algorithm 2: Branch and bound TSP

```
Result: Length of the shorthest path
Input: node;
queue = PriorityQueue();
queue.put(priority=node.cost, element=node);
do
    node = queue.get();
    if node.depth == node.dist.size-1 then
        return node.cost;
    foreach nextPos reachable from node.pos do
        nextNode = generateNewNode(node, nextPos);
        nextNode.reduceDistanceMatrix();
        nextNode.updateCost();
        queue.put(priority=nextNode.cost,
          element=nextNode);
    end
while not queue.isEmpty();
```

5.2 Dynamic programming

The second algorithm that was tested to solve TSP uses dynamic programming [16]. Here a queue filled with nodes is used to find the length of the shortest path from any waypoint, past all the other waypoints to the goal simultaneously by working backwards starting from the goal location. All the distances are stored in a hashmap.

All nodes n, consist of a set of visited waypoints $n.visited$ and, the last visited waypoint $n.last$. First for every waypoint w a node is generated with only w in its set of visited waypoints, and w as its last visited waypoint. Al of these nodes are also added to the distance hashmap. Here the distance for every node is given by the distance from the goal location to waypoint w corresponding to that node.

While the queue is not empty, nodes will be popped from the queue. Every-time a node n is popped from the queue, a new set of nodes is generated. This new set of nodes is generated by first finding the set of nodes that is unvisited, $unvisited$, by node n, by taking the difference of the set of all waypoints, and $n.visited$. For each waypoint w in $unvisited$, a new node is generated. Its set of visited waypoints is the disjunction of $n.visited$ and w. And the last visited waypoint for this new node is w.

For every node $child$ in this generated set of nodes, a cost, new_cost, generated by adding the cost of n as stored in the distance hashmap and, the distance between $last\_visited$ of n and $n.last$. If $child$ is not yet in the distance hashmap it is added with new_cost. If $child$ it already has an entry in the distance hashmap, the distance in the hashmap is updated to be the minimum of the old distance and new_cost.

When the queue is empty, the length of the shortest path from any waypoint, past all the other waypoints to the goal is stored in the distance hashmap. To find the distance starting from waypoint w, a node is generated that has the set of all waypoints as its set of visited waypoints, and w as its last visited waypoint. This node used to look up the distance in the distance hashmap.

6 Bypass optimization

The first optimization for CBSW that has been investigated is the Bypass optimization [10]. When using the Bypass optimization, CBSW is referred to as CBSW-B. When a conflict is being resolved, without the Bypass optimization, two new child nodes are always created and added to the high-level search tree. This results is a rapid growth of the number of nodes in the search tree. The Bypass optimization tries to reduce the number of nodes in the high-level search tree by avoiding splitting a node into two new child nodes.

Every time a node n with a conflict c is considered, if one of the generated child nodes is a helpful bypass to n, only that node is inserted into the high-level search tree. The other child node is discarded.

Node bp is a valid bypass to node n with conflict c, if bp does not have conflict c, $n.Cost == bp.Cost$ and bp is consistent will all of the constraints of node n.

Node bp is a useful bypass to node n with conflict c, if bp is a valid bypass to n in respect to conflict c and the number of conflicts in bp is strictly smaller than the number of conflicts in n.

If a useful bypass bp is found, because the parent node of bp had the lowest cost in the high-level search tree and bp has the same cost as his parent node, bp has the lowest cost of all the nodes in the high-level search tree. Because of this, it is possible to directly consider bp as the next node in the high-level search. This is repeated until no bypasses are found anymore.

7 Corridor constraints optimization

CBS performs well on problems where few conflicts occur. If there are one wide corridors in a problem, where two agents want to pass the corridor from opposite sides, lots of conflicts occur drastically limiting the performance of CBS.

A way to model train tracks using MAPFW problems, is to use corridors. This is because there can not be two trains passing a train track from opposite sides, just like there can not be two agents passing a MAPFW corridor. There can

be multiple trains on the same track, for example if they are moving in the same direction just like multiple agents can be in the same MAPFW corridor at the same time.

To enable CBSW to solve problems with corridors more efficiently, the corridor constraint optimization is proposed. When using the corridor constraints, CBSW is referred to as CBSW-C. This optimization has two parts. The first part is to find all the corridors in a the MAPFW problem that is being solved. This part is done as a pre-processing step and only has to be done once. The second part is to add a new type of constraint to child nodes when a conflict occurs in a corridor.

7.1 Finding corridors

To find all the corridors in a MAPFW problem, first all grid cells that are empty are considered. A grid cell is empty if it does not contain a wall, agent starting point, agent waypoint or agent goal location. For each of these empty cells, the number of direct neighbours that are a wall is counted. The direct neighbours are the cells above, below, left and right of a cell. Every MAPFW problem is considered to be completely surrounded by a layer of cells that are walls. A set $candidates$ is then filled with all empty cells that have at least two neighbours that are a wall.

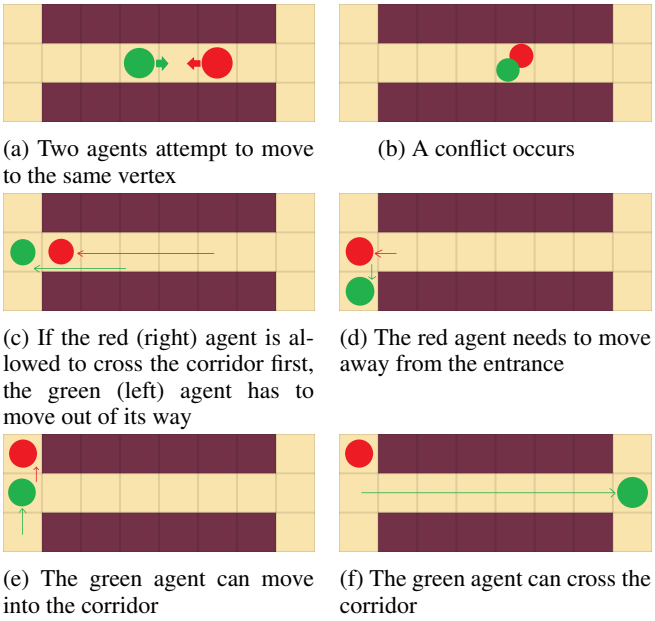
After creating the $candidates$ set, an empty set of corridors, $corrs$, is initialized. A corridor is a set of cells that are all direct or indirect neighbours. Two cells a and b in a set s are indirect neighbours when there is another cell c in s such that a is a (indirect) neighbour to c and b is a (indirect) neighbour to c.

All cells in $candidates$ are added to $corrs$ one by one. When adding cell a, if there are corridors in $corrs$ that have a cell that directly neighbours a, all corridors in $corrs$ that have a cell that directly neighbours a are merged, and a is added to that set. If no such corridor exist in $corrs$, a set containing only a is added to $corrs$.

For every corridor, the end points are found. This is done by finding the two cells in the corridor that only have one direct neighbour in the corridor. If no two such cells are found, the corridor is discarded. This only happens when a corridor forms a loop.

When the two endpoints of a corridor with cells $cells\_set$, are found, the corridor is ordered. A new ordered list $cells\_orderedlist$ is used for this. It is initialized with one of the end points of the corridor as its only element. While $cells\_set$ is not empty, the only cell in $cells\_set$ that is a direct neighbour to the last cell in $cells\_orderedlist$ is moved from $cells\_set$ to $cells\_orderedlist$. When $cells\_set$ is empty, $cells\_orderedlist$ contains all the cells of the corridor such that all two cells that are next to each other in $cells\_orderedlist$ are direct neighbours.

Given a ordered list l representing a corridor and a cell c that is a member of l, it is possible to calculate the distance from both endpoints of l to c. For the endpoint in the head of l, this is done by taking the position of c in l. For the endpoint in the tail of l, this is done by subtracting the position of c in l from the length of l minus one.



(a) Two agents attempt to move to the same vertex

(b) A conflict occurs

(c) If the red (right) agent is allowed to cross the corridor first, the green (left) agent has to move out of its way

(d) The red agent needs to move away from the entrance

(e) The green agent can move into the corridor

(f) The green agent can cross the corridor

Figure 1: Example of a corridor conflict and its resolution

7.2 New constraints

When a conflict is being resolved, and this conflict occurs in a corridor this is called a corridor conflict. When a corridor conflict is found, a new constraint, called an exit constraint, is added to the conflicting agents. This constraint uses the fact that it is not possible for both agents to cross the corridor starting from opposite sides at the same time. Exit constraints enforce that an agent can not cross a corridor completely without constraining it to much. An exit constraint prevents an agent a form leaving cor, with endpoints $entr$ and $exit$, at $exit$ between time t_s and t_e, if it entered cor at $entr$. This constraint is denoted by $\langle a, cor, exit, t_s, t_e \rangle$.

To guarantee optimally, four child nodes bust be generated when a corridor conflict is being resolved. In every child node one of the conflicting agents gets an exit constraint, and one, possibly the same agent gets a vertex or edge constraint depending on the nature of the conflict. This gives four combinations, and therefore four child nodes.

The t_s is set to the time of the conflict. t_e is set as the time of the conflict plus the distance from the conflict location to the entrance used by a of the corridor, plus the length of the corridor plus two (In case of an edge conflict that occurs in a corridor, the conflict time plus one is used here). This t_e is chosen because that is the earliest time that a is able to exit the corridor on $exit$ if the other agent crosses the whole corridor first. To get to the end of the corridor the other agent still

needs to travel the distance from the conflict location to the entrance used by a as seen in Figure 1c. Then it needs to move out of the way for a to be able to move into the corridor as seen in Figure 1d,1e. This takes two time steps. Finally agent a needs to cross the whole corridor which takes as many time steps as the corridor is long as can be seen in Figure 1f.

Even though the branching factor is increased from two to four, in practice often some of the child nodes get a significantly higher cost because of the stricter constraints. These child nodes will therefore not be expanded before all nodes with a lower cost have been expanded. This reduces the number of extra nodes in the search graph that get added because of the higher branching factor.

8 Prioritizing conflicts

The third optimization that has been investigated is the Prioritizing conflicts optimization [17]. Here an extension for this optimization that works with MAPFW problems is proposed. The performance of CBSW is highly dependant on the order in which conflicts are chosen to be resolved in the high-level algorithm. Without the Prioritizing conflicts optimization the first conflict that is found is used. It is better to prioritize conflicts that can not be resolved without adding cost to the solution, called cardinal conflicts. To identify cardinal conflicts, the primary data structure that is used in Increasing Cost Tree Search, called a multi-value decision diagram (MDD) is used [8]. A MDD is a directed acyclic graph, in which all paths that lead an agent to its goal within a given budget are stored.

The way that an MDD efficiently stores all paths is by using layers. Every layer contains a node for all possible states that an agent can be in, in the time-step corresponding to the depth of the layer. All nodes, with state s, except for the node in the final layer, are connected to all nodes in the next layer which have a state that can be reached from s. For MAPF problems the state only contains the location of an agent. For MAPFW problems states also contain the set of waypoints that has already been visited.

The first layer of a MDD for agent a, contains a single node, $startNode$. The position of a in the state of this node is the starting position of a. For MAPFW problems the set of visited waypoints is empty for the state of this node. The last layer of a MDD for a also contains a single node, $goalNode$. The position of a in the state of this node is the goal location of a. For MAPFW problems the set of visited waypoints in the state of this node is the set containing all the waypoints of a.

In order to construct an MDD for an agent a, a depth limited breadth first search is performed starting from a state with the start position of a, and an empty set of visited waypoints. The depth limit is set to the cost of the path of a in the high-level search. During this search every time a node is explored, the TSP based heuristic function is used to check if the length of the shortest path past all the waypoints that are not yet visited

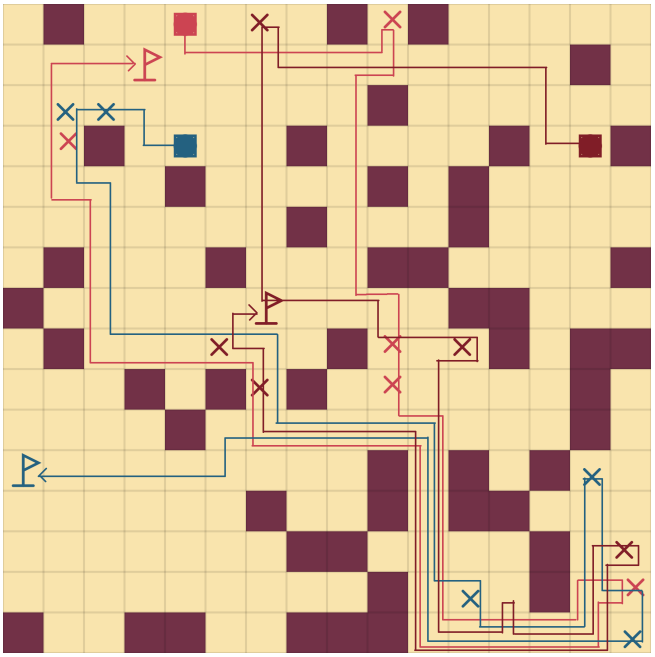


Figure 2: An example of a randomly generated maze of size 16×16 with three agents and five waypoints per agent including the paths of an optimal solution

in the state of that node, to the goal location is longer than there are time steps left before the depth limit is reached. If the shortest path is longer than there are time steps left before the depth limit is reached, the node is discarded. This reduces the number of nodes that are generated that do not lead to the $goalNode$ before the depth limit is reached.

When all nodes have been generated all nodes in the last layer are discarded except for $goalNode$. When this is done, all nodes that do not lead to another node, except for $goalNode$, are discarded. This is done by looping over all layers in a backwards order starting at the second to last layer.

When in this MDD the layer that corresponds to a time-step contains only one node, or all the nodes in that layer contain the same location in their state, a can not be on any other location on that time-step without increasing the length of its path. If the conflict location is, on the time of the conflict the only location that that can be used without increasing the length of the path for both agents, the conflict is a cardinal conflict. If no cardinal conflicts are found, conflicts where at least one of the agents has to use the conflict location at the time of the conflict, called semi-cardinal conflicts, are prioritized.

9 Experimental results

To assess the performance of CBSW extensive experiments are performed using a variety of different benchmarks. First the results of comparing different versions of CBS based MAPFW solvers, using different improvements are reported. Then the results of comparing CBSW with A*-OD-ID-W

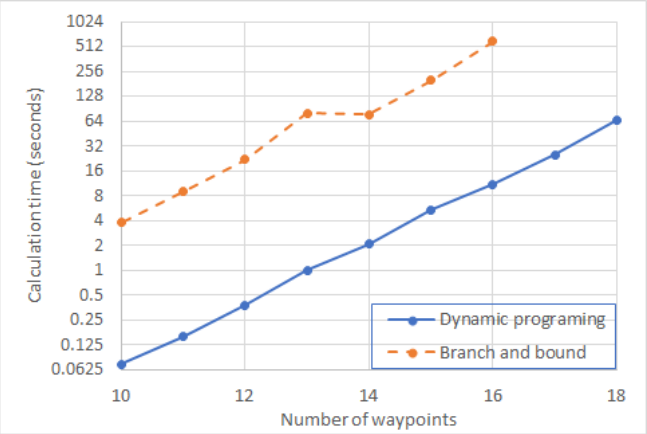


Figure 3: Comparison of the calculation time of the TSP solvers

and Branch-and-cut-and-price-W are shown. The A*-OD-ID-W algorithm is implemented by Steven Siekman and the Branch-and-cut-and-price-W algorithm is implemented by Andor Michels.

All benchmarks have been run on a 14 core Intel(R) Xeon(R) Gold 6248 CPU running at 2.50GHz with 8GB of ram. CBSW and A*-OD-ID-W have been implemented in Python 3.8 and Branch-and-cut-and-price-W has been implemented in C++.

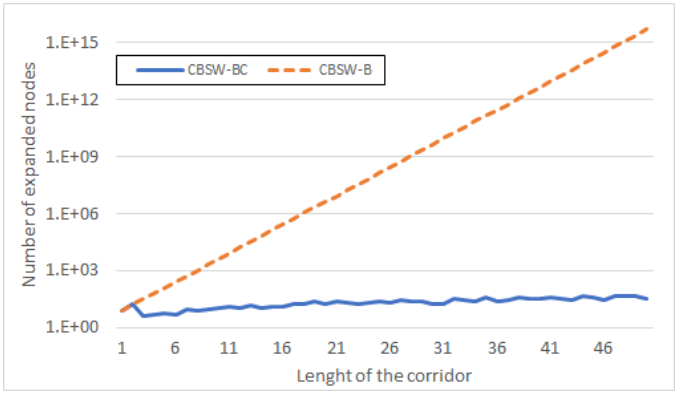
9.1 Benchmarks

To assess the performance of MAPFW solvers, randomly generated grid benchmarks have been used. In these benchmarks, agents are randomly placed in a $n \times n$ 4-connected grid. All the agents are assigned a random set of waypoint locations, and a random goal location assigned to them. 20% of the grid is filled with obstacles. An example of such a randomly generated map is Figure 2. 4-connected grids are often used for MAPFW benchmarks and closely resemble real-world path-finding problems [18].

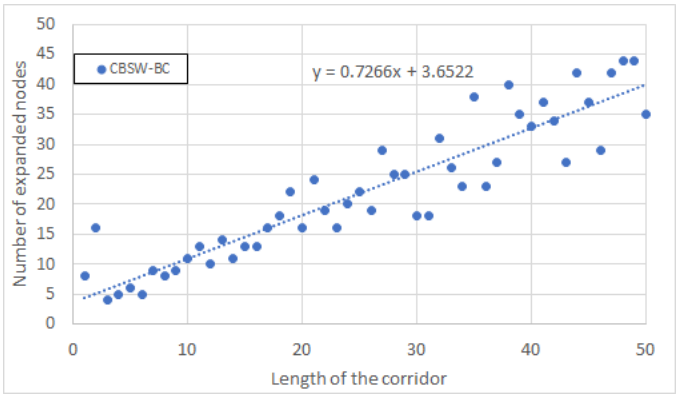
When using randomly generated benchmarks, a seed is used to ensure that all solvers get exactly the same benchmarks. To show the difference between CBSW with and without the corridor optimization, a maze with a single corridor is used.

9.2 Comparison of versions of CBSW

First the two TSP solvers are compared. Both the Dynamic programming TSP solver as the Branch and bound TSP solver are benchmarked on randomly generated TSP problems. Both solvers solved TSP problems with 10 to 18 cities. For each number of cities each solver solved 10 problems, and the average time is reported in figure 3. No times are reported for the Branch and bound TSP solver for 17 or 18 cities, because given the available memory of 8GB, instances with more than 16 cities could not be solved using the Branch and bound



(a) Comparing CBSW-B and CBSW-BC on a logarithmic scale



(b) Performance of CBSW-BC with a trend line on a linear scale

Figure 4: Nodes expanded in the high-level search on increasingly long corridors

based TSP solver. It can be seen that regardless of the number of cities, the Dynamic programming based TSP solver outperforms the Branch and bound based TSP solver. Because of this, the Dynamic programming based TSP solver is used in CBSW.

Second, The difference between CBSW with and without the Bypass optimization is investigated. CBSW with the Bypass optimization is used to solve a set of randomly generated benchmarks. The percentage of conflicts that can be bypassed is recorded. On 1000 conflicts, 51% could be bypassed. There is no extra overhead to the algorithm when using the Bypass optimization, because in the worst case, when no bypass is found, the same happens as in CBSW without the Bypass optimization. When a bypass is found only one child node is added to the high-level search tree, and therefore the search is faster. Because of this the Bypass optimization is used in CBSW. To make extra clear in the other comparisons that the Bypass optimization is used CBSW, CBSW with the Bypass optimization is referred to as CBSW-B.

CBSW-B is compared with CBSW-BC (CBSW with both the bypass and the corridor optimizations) in Figure 4. A logarithmic scale is used to show the number of nodes that are expanded in the high-level search tree. The benchmark used

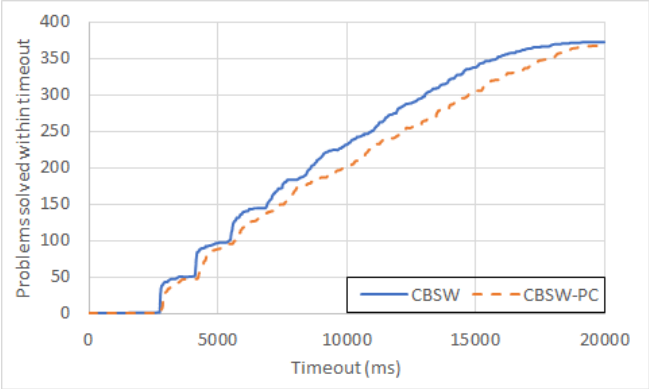


Figure 5: Problems solved within the time limit on random 32×32 maps with 13 waypoints per agent and 2 to 12 agents

Figure 6: A corridor maze of length 6

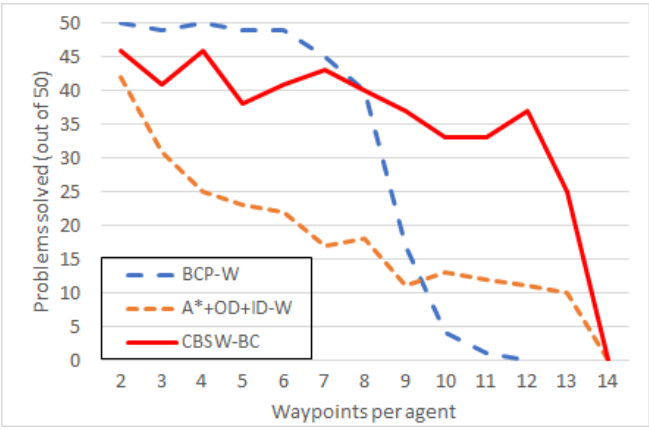
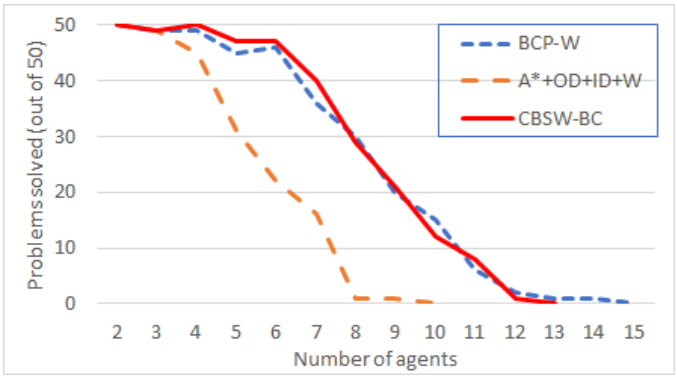
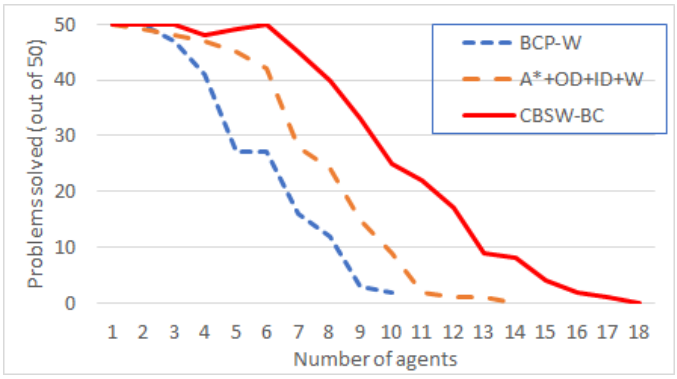


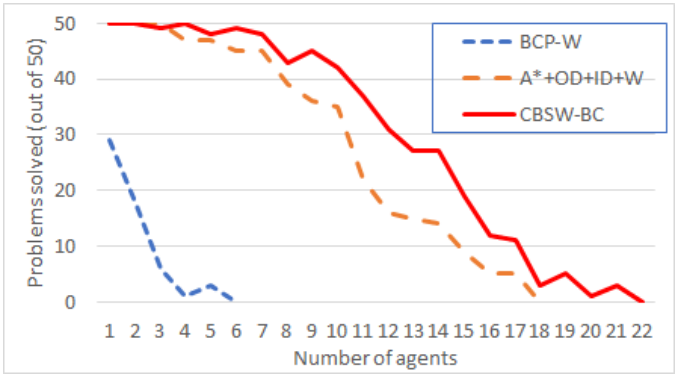
Figure 7: Benchmarks completed with a 20 second timeout with 10 agents on a 16×16 map with an increasing number of waypoints



(a) 16×16 maps



(b) 32×32 maps



(c) 64×64 maps

Figure 8: Benchmarks completed with a 20 second timeout with 10 waypoints per agent with an increasing number of agents on three map sizes

to test both solver consists of a single one wide corridor with an agent on both sides, and on both sides of the corridor there are three empty spaces for the agents to move around in. Both agents their goal location is on the opposite side of the corridor. An illustration of such a benchmark with a corridor of length six is shown in Figure 6.

Without the corridor optimization the number of nodes that is expanded grows exponentially with the length of the corridor. The exact amount of nodes that is expanded is 4×2^n where n is the length of the corridor. This exponential growth can

be seen in Figure 4a. In Figure 4b only the number of nodes that are expanded using CBSW-BC is shown. It can be seen that the number of expanded nodes grows linearly in respect to the corridor length. There is some variation from the trendline. This happens because when choosing nodes to expand in the high-level search, when two or more nodes have the same cost, ties are broken randomly.

The overhead of finding the corridor locations, and checking whether or not conflicts occur in a corridor is very minimal. Regardless of the length of the corridor, these extra calculations take less then a millisecond of time during the whole solve of the benchmark. Because of the improvements that the corridor optimizations bring to CBSW, in the comparison with other MAPFW solvers, CBSW-BC is used.

To asses the performance impact that the Prioritizing conflicts optimization has on MAPFW, first the number of nodes that is expanded in the high-level search algorithm is compared between CBSW without the Prioritizing conflicts optimization and CBSW with the Prioritizing conflicts optimization (CBSW-PC). On 100 randomly generated benchmarks on 32×32 grids with two to five agents and five waypoints per agent, CBSW expanded an average of 10.65 nodes and CBSW-PC expanded an average of 7.85 nodes. This shows that the Prioritizing conflicts optimization decreases the number of expanded nodes by 26%.

Even though the number of expanded nodes is reduced, the performance of CBSW-PC is worse than the performance of CBSW. In Figure 5, the number of randomly generated problems with 2 to 12 agents and 5 waypoints per agent on a 32×32 grid within a timeout is shown. Only problems solved by both solvers within 20 seconds are considered. The line for CBSW is consistently higher than the line for CBSW-PC. This indicates that within a given time limit CBSW solves more problems than CBSW-PC. The bumps in the graph are there because maps with a varying number of agents are used.

Experimental results on the same set of 32×32 benchmarks shows that the prioritization of conflicts causes 35% overhead. This explains why the performance of CBSW-PC is worse than the performance of CBSW. A large contributing factor to this overhead is the heuristic calculation used to create the MDDs. This heuristic is needed because, the creation of an MDD with a budget of b and waypoints wp on a $size \times size$ grid can expand up to $size \times size \times b \times 2^{wp}$ nodes. The heuristic is shown to be effective in practice in limiting the number of expended nodes.

Because the performance of CBSW-PC is worse than the performance of CBSW because of the added overhead, in the comparison with other algorithms, the Prioritizing conflicts optimization is not used.

9.3 Comparison with other algorithms

CBSW is compared with A*+OD+ID+W and Branch-and-cut-and-price-W. First it is assessed how the performance of the solvers changes on benchmarks with an increasing number of waypoints per agent. The grids used are 32×32 and all benchmarks use 10 agents. For every tested number of waypoints, all solvers are given the same set of 50 randomly generated challenges, it is shown in Figure 7 how many of these 50 challenges each solver was able to solve within 20 seconds per challenge.

For benchmarks with less than eight waypoints per agent BCP-W can be seen to outperform CBSW. This can be explained by comparing CBS and BCP (the non-waypoint versions). On MAPF problems BCP outperforms CBS [6]. The difference in programming language also affects the relative performance here in favour of BCP-W.

On benchmarks with eight to twelve waypoints, CBS outperforms BCP-W and A*+OD+ID+W. This can be explained by the difference in the way that CBSW and BCP-W handle waypoints. BCP-W needs to re-compute the heuristic, which also uses TSP, more often. Solving TSP with more waypoints takes exponentially longer. Therefore adding more waypoints quickly increases the time that BCP-W needs to solve MAPFW problems. Because of the efficient memoization of the heuristic function in CBSW, the increase in the number of waypoints has less effect on CBSW.

On benchmarks with more than 12 waypoints, the performance of CBSW quickly drops off, and with 14 waypoints the success-rate drops to 0%. The performance of A*+OD+ID+W is lower than the performance of CBSW regardless of the number of waypoints, but for problems with 10 or more waypoints, A*+OD+ID+W outperforms BCP-W. This is because A*+OD+ID+W is also capable of efficiently memoizing the results of its heuristic function. The success-rate of A*+OD+ID+W also drops to 0% at 14 waypoints. For both CBS-W and A*+OD+ID+W this can be explained, because at 14 waypoints, the time needed to solve TSP in the heuristic function for just one agent is 2.2 seconds. For 10 agents the time needed is thus $10 \times 2.2 = 22$ seconds. This means that it takes more time to compute the solutions to the TSP problems then there is timeout on the benchmark. Because both CBSW and A*+OD+ID+W need to calculate those heuristic values, problems with 14 or more waypoints could not be solved within a 20 second time limit.

Second it is assessed how the size of the maps used and the number of agents affects the performance of CBSW, A*+OD+ID+W and BCP-W. Three benchmarks are used for this. All benchmarks use 10 waypoints per agent, because that is the number of agents for which the three solvers had the most similar performance based on the previous benchmark. The three benchmarks use maps of 16×16, 32×32 and 64×64 respectively. The results can be seen in Figure 8. Here it is also reported for each number of agents, how many out of 50 challenges each solver was able to complete with a 20 second timeout.

On 16×16 benchmarks, CBSW and BCP-W perform very similarly as can be seen in Figure 8a. Both algo-

rithms linearly solve less and less problems as the number of agents increases. Both CBSW and BCP-W outperform A*+OD+ID+W on this benchmark regardless of the number of agents. The challenge with the largest number of agents that CBSW completed within the 20 seconds timeout has 13 agents.

On 32×32 benchmarks, CBSW outperforms both A*+OD+ID+W and BCP-W as can be seen in Figure 8b. Both CBSW and A*+OD+ID+W perform better on 32×32 maps than on 16×16 maps, while BCP-W performed worse. The number of agents in the hardest challenge that CBSW solved on this benchmark was 17.

On 64×64 benchmarks, CBSW also outperforms both A*+OD+ID+W and BCP-W as can be seen in Figure 8c. The performance of both CBSW and A*+OD+ID+W increased again while BCP-W performed worse on these larger maps. The number of agents in the hardest challenge that CBSW solved on this benchmark was 21. On these 64×64 maps, the performance difference between CBSW and A*+OD+ID+W is relatively small.

The increase in the performance of CBSW on larger maps can be explained by the decrease in the number of conflicts on larger maps. The less conflicts occur the better CBSW performs.

The small difference in performance between CBSW and A*+OD+ID+W on large maps can also be explained by the decrease in the number of conflicts on larger maps. If no conflicts occur, both CBSW and A*+OD+ID+W use A* to plan a path for all agents independently.

10 Responsible Research

To make this research reproducible, all source code for CBSW is made publicly available on GitHub. Together with Steven Siekman, we have made a website showing all results including more detailed graphs: mapfw.nl. This website has an API that can be used to automatically benchmark MAPFW solvers and an interface to design new benchmarks. To further improve the reproducibility of this research, and to allow further research, we have also released a Python client library. This library can be used to run all benchmarks on the mapfw.nl website, including the benchmarks that have been used in this paper. Both the website, and the client library provide tools for analysing the results of the benchmarks.

Extra attention is given to make sure that users of the client library only submit valid results. This is achieved by performing lots of checks in the client library that give warnings to the user when something appears to go wrong. An example of this is that some benchmarks have a default timeout. If the user explicitly requests another timeout from the client library, this will result in a warning.

All algorithm timing is handled by the client library, to make sure that the timing methodology is consistent between all users of the client library. To speed up test times, support for running benchmarks on multiple cores is present as well as options to enable profilers to profile your algorithm.

11 Limitations

All the implementations of CBSW used in this paper have been made in an interpreted language, Python 3.8. This made implementing CBSW faster, but the run-time of the algorithm is longer then if a compiled language was used. The difference in run-time between interpreted and compiled languages can be 4 orders of magnitude on some problems [19]. Therefore this paper shows the relative difference between the different optimizations to CBSW solvers, and does not give much inside to the performance that can be achieved using these algorithms in production code using a compiled language.

A*-OD-ID-W was also made in Python 3.8, and therefore the comparison between CBSW and A*-OD-ID-W gives a good indication of the difference between the two algorithms. Branch-and-cut-and-price-W was implemented in C++, and therefore not only algorithmic differences influenced the results of the benchmarks. Especially for easier benchmarks the difference in programming languages can make a big difference in the results of the benchmark.

The Bypass optimisation should not affect the optimally of the solutions generated by CBSW since in [10], it is said that that the bypass optimisation does not impact optimally. There is however no formal proof of this in this paper. Experimental results give no indication to believe that this optimization effects optimality, but a formal proof is still needed.

12 Conclusion

Adding waypoints to MAPF increases the difficulty of the problem. Using dynamic programming to calculate efficient heuristics, enables CBSW to decrease the amount of work that has to be done in the low-level solver. The Bypass optimization provides a large speedup on general problems, and the Corridor optimization provides a large speedup in problems with one wide corridors. The Prioritizing conflicts optimization does not improve the performance of CBSW because it adds a lot of overhead. As demonstrated empirically on standard benchmarks, CBSW has better performance on benchmarks that use many waypoints, or have large maps, than A*+OD+ID+W and BCP-W, and on small maps with a small number of waypoints BCP-W outperforms CBSW.

13 Future Work

Future work should (1) Investigate extending other optimal MAPF solvers to solve MAPFW problems. (2) Investigate

the effects of different and more advanced TSP solvers. (3) Investigate the effect of re-writing CBSW in a compiled language.

References

[1] Roni Stern et al. "Multi-agent pathfinding: Definitions, variants, and benchmarks". In: *Twelfth Annual Symposium on Combinatorial Search*. 2019.

[2] Jingjin Yu and Steven M LaValle. "Structure and intractability of optimal multi-robot path planning on graphs". In: *Twenty-Seventh AAAI Conference on Artificial Intelligence*. 2013.

[3] Jesse Mulderij et al. *Train Unit Shunting and Servicing: a Real-Life Application of Multi-Agent Path Finding*. 2020. arXiv: 2006.10422 [cs.MA].

[4] David Silver. "Cooperative Pathfinding." In: *AIIDE* 1 (2005), pp. 117–122.

[5] Trevor Scott Standley. "Finding optimal solutions to cooperative pathfinding problems". In: *Twenty-Fourth AAAI Conference on Artificial Intelligence*. 2010.

[6] Edward Lam et al. "Branch-and-cut-and-price for multi-agent pathfinding". In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI-19), International Joint Conferences on Artificial Intelligence Organization*. 2019, pp. 1289–1296.

[7] Florian Grenouilleau, Willem-Jan van Hoeve, and John N Hooker. "A Multi-Label A* Algorithm for Multi-Agent Pathfinding". In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 29. 1. 2019, pp. 181–185.

[8] Guni Sharon et al. "The increasing cost tree search for optimal multi-agent pathfinding". In: *Artificial Intelligence* 195 (2013), pp. 470–495.

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

[10] Eli Boyrasky et al. "Don't split, try to work it out: Bypassing conflicts in multi-agent pathfinding". In: *Twenty-Fifth International Conference on Automated Planning and Scheduling*. 2015.

[11] Steven Arend Pieter Siekman. "Extending A* to solve multi-agent pathfinding problems with waypoints". In: *TU Delft Repository* (June 2020).

[12] Andor Michels. "Multi-agent pathfinding with waypoints using Branch-Price-and-Cut". In: *TU Delft Repository* (June 2020).

[13] Gilbert Laporte. "The traveling salesman problem: An overview of exact and approximate algorithms". In: *European Journal of Operational Research* 59.2 (1992), pp. 231–247.

[14] Donald Michie. ""Memo" functions and machine learning". In: *Nature* 218.5136 (1968), pp. 19–22.

[15] AH Land and AG Doig. "An automatic method of solving discrete programming problems. Econometrica. v28". In: (1960).

[16] Stuart Dreyfus. "Richard Bellman on the birth of dynamic programming". In: *Operations Research* 50.1 (2002), pp. 48–51.

[17] Eli Boyarski et al. "ICBS: improved conflict-based search algorithm for multi-agent pathfinding". In: *Twenty-Fourth International Joint Conference on Artificial Intelligence*. 2015.

[18] Peter Yap. "Grid-based path-finding". In: *Conference of the Canadian Society for Computational Studies of Intelligence*. Springer. 2002, pp. 44–55.

[19] Juan-Julián Merelo-Guervós et al. "Ranking the Performance of Compiled and Interpreted Languages in Genetic Algorithms". In: (2016).