Multi-agent pathfinding with waypoints using Branch-Price-and-Cut

Andor C. Michels[1] ,
Supervisors: Jesse Mulderij[1] , **Mathijs de Weerdt**[1]
[1]Delft University of Technology

Abstract

In the multi-agent pathfinding (MAPF) problem, agents have to traverse a graph to a goal location without running into each other. Currently, the Branch-and-Cut-and-Price-MAPF (BCP-MAPF) algorithm is the state-of-the-art algorithm for solving MAPF problems, which uses Branch-Price-and-Cut (BPC) to solve a linear minimization problem. The multi-agent pathfinding with waypoints (MAPFW) problem is a variation on the classical MAPF problem. MAPFW can be useful for e.g. navigating warehouses and scheduling trains. However, little to no research exists on MAPFW. This research proposes two extensions to the BCP-MAPF algorithm to incorporate waypoints and reviews their performance.

1 Introduction

Multi-agent pathfinding (MAPF) is the problem of multiple agents moving on a graph without occupying the same vertex or edge as another agent. Solving this problem can be very important for e.g. automated warehouses [28], airport logistics [21] and train scheduling [1].

Numerous algorithms exist with different areas of application. The current state-of-the-art algorithms for the basic MAPF problem are Conflict-Based Search (CBS) [25] and Branch-and-Cut-and-Price (BCP) [18, 17]. However, in most real-life situations, agents are required to do more than move to a certain location. For example, the multi-agent pickup and delivery (MAPD) problem [19] assigns one 'waypoint' to each agent, which has to be passed on its way to the end location. The MAPD problem can be generalized to an arbitrary amount of waypoints such that the agent can do multiple tasks or pick up multiple items and bring them to a location in a warehouse scenario. We will call this extension multi-agent pathfinding with waypoints (MAPFW). Despite its usefulness, little to no research has been done on this extension of MAPF.

The aim of this research is to find how the BCP MAPF algorithm can be effectively extended to include waypoints. We are looking specifically to BCP because it is the current state algorithm for solving MAPF problems.

This paper provides: (1) an analysis of extending BCP-MAPF with waypoints; (2) two possible algorithms for solving the MAPFW problem using Branch-Price-and-Cut and; (3) experiments that compare the performance of said algorithms on four different grid-based maps.

2 MAPFW problem statement

This section will define the MAPFW problem. Let $G = (V, E)$ be an undirected graph containing a set of vertices V and a set of edges $E \subset V \times V$. Let A be a set of agents a, with a start location $s_a \in V$, an end location $g_a \in V$ and a set of waypoints $W_a \subset V$. At every timestep $t \in \mathbb{Z}^{\geq}$, an agent can do one of two actions: either move to a neighbouring vertex, or stay on its current vertex.

A solution to the MAPFW problem has every agent a starting on vertex s_a at $t = 0$, moving according to the previously described actions, and stopping at vertex g_a, while passing every waypoint $w \in W_a$ at least once. At no timestep t two agents may cross the same edge (edge conflict) and at no timestep t two agents may be at the same vertex (vertex conflict). For every agent a, the sequence of actions is the path p_a with cost $c(p_a)$, equal to the amount of actions. An optimal solution to the MAPFW problem is a solution where $\sum_{a \in A} c(p_a)$ is minimised.

3 Background

This section summarizes the literature on different subjects used throughout this research. First, Linear Programming and Integer Linear Programming are introduced, followed by the solving technique Branch-Price-and-Cut. Finally, a Linear Programming formulation of the traveling salesman problem is provided.

3.1 LP and ILP

In Linear Programming (LP), a linear function, subject to certain linear constraints, is minimized. A simple example of a LP problem is the following:

$$\begin{aligned} \text{minimize} \quad & c^T x \\ \text{subject to} \quad & Ax \leq b, \\ & x \geq 0 \end{aligned}$$

Solving a linear program can be done in polynomial time [16], but it is often solved using the exponential simplex method [22], since it's more efficient in practice.

In Integer Linear Programming (ILP), a linear program is solved while fixing the variables to integers. In contrast with LP, ILP is NP-complete.

However, because of the broad applicability of ILP [13, 15] and the large amount of research done on ILP [8, 7, 2], there exist very efficient industrial solvers. Therefore, it can be useful to reduce an NP-hard problem to an ILP program.

3.2 BPC

Branch-Price-and-Cut (BPC) is a technique for solving ILP problems.

In the early research on BPC, it was named Branch-and-cut-and-Price (BCP). Desrosiers et al. [11] have proposed to change the name to Branch-Price-and-Cut, as it better reflects the order of operations. This naming scheme has been adopted by several researchers [9, 12], but the old naming scheme has been used by the creators of the BCP-MAPF algorithm [18, 17]. In this paper, we will use BPC for the solving technique, and BCP-MAPF for the MAPF algorithm invented by Lam et al. [18, 17].

BPC contains 3 main components: the master problem, the pricer and the separators.

In the master problem, the linear program is solved with a subset of the constraints. When a fractional solution is found, the algorithm branches and creates a new constraint such that a fractional variable is either rounded up or down. Then the new linear program is solved again. This process is repeated until an integer solution is found.

Because the search space of ILP problems is generally very large, it can be useful to limit the number of variables in the master problem. For the MAPFW problem, time can be infinitely expanded, making the search space infinitely large, meaning that limiting the number of variables is necessary. The pricer generates variables that could result in valid solutions for the master problem that could be better than the current solutions. This way a lot of irrelevant, unused variables do not have to be taken into account. This is generally referred to as column generation. Since the relevant search space is often exponential in size, the set of variables needs to be expanded a lot. Therefore, the pricer must be relatively efficient.

When branching, two branches are created that can in turn branch. Often, a lot of these branches need to be partially searched in order to verify an optimal solution. The constraints that are necessary for the problem are referred to as problem constraints, whereas the constraints that are not strictly necessary are called redundant constraints. Redundant constraints can be useful by cutting away extra fractional solutions.

When the master problem has found a solution that does not adhere to one of the redundant constraints, the separators will add one of the constraints, a 'cut', to the master problem. This is generally referred to as row generation.

Not all branches have to be fully searched. The current best sub-optimal solution can be used as an upper bound, since any solution with a higher cost is less optimal. This greatly reduces the size of the search space.

3.3 BCP-MAPF

BCP-MAPF is a reduction of MAPF to an ILP problem that can be effectively solved using BPC [18, 17]. It works by eliminating conflicts between different agents. BCP-MAPF is optimal and complete. This part will describe the BCP-MAPF algorithm in terms of the elements of BPC.

For every agent $a \in A$ and every path $p \in P_a$, define $c_p \in \mathbb{R}^{\geq}$ as the constant cost of path p, and λ_p as a variable that represents the extent to which a path is chosen. The master problem is represented as follows:

$$\begin{aligned} \text{minimize} \quad & \sum_{a \in A} \sum_{p \in P_a} c_p \lambda_p \\ \text{subject to} \quad & \sum_{p \in P_a} \lambda_p \geq 1 && \forall a \in A \\ & \lambda_p \geq 0 && \forall a \in A, p \in P_a \end{aligned}$$

In this problem, λ_p will always be between 0 and 1 since it cannot be lower and selecting multiple paths per agent will never minimize the cost function.

In the MAPF problem, 2 paths cannot cross the same vertex or the same edge at the same time. Therefore 2 problem constraints are added to the problem. Let $G^t = (V^t, E^t)$ be the time-extended graph of $G = (V, E)$, where $v = (l, t) \in V^t$ represents location $l \in V$ at time $t \in \mathbb{Z}^{\geq}$. If $e \in E^t$ is the edge from (l_1, t) to $(l_2, t+1)$, then $e' \in E^t$ is the edge from (l_2, t) to $(l_1, t+1)$. To enable staying in place, for every vertex $v = (l, t)$, there is an edge e from (l, t) to $(l, t+1)$.

For every agent $a \in A$, every path $p \in P_a$ and every vertex $v \in V^t$, define $x_v^p \in \{0, 1\}$ as a constant indicating if path p uses vertex v. For every agent $a \in A$, every path $p \in P_a$ and every edge $e \in E^t$, define $y_v^p \in \{0, 1\}$ as a constant indicating if path p uses edge e.

Vertex conflicts caused by 2 paths using the same vertex can be resolved by allowing only one of the paths. This is done with the following linear constraint:

$$\sum_{a \in A} \sum_{p \in P_a} x_v^p \lambda_p \leq 1 \quad \forall v \in V^t$$

Similarly, edge conflicts caused by 2 paths using the same edge can be resolved with a linear constraint:

$$\sum_{a \in A} \sum_{p \in P_a} (y_e^p + y_{e'}^p) \lambda_p \leq 1 \quad \forall e \in E^t$$

BCP-MAPF contains 7 redundant constraints that greatly reduce the execution time of the algorithm. These redundant constraints are also used in the MAPFW-algorithms described in this paper. However, since they do not specifically add to the extension with waypoints, they are not further discussed. More information on these constraints can be found in [18] and [17].

The pricer generates single-agent shortest paths for all agents on the time-extended graph with edge weights adjusted relative to conflicts found in the master problem.

3.4 TSP

The Traveling Salesman Problem (TSP) is the problem of finding the shortest route through all points in a set. Several well-documented ILP formulations exist for the TSP problem, one of which is the Miller-Tucker-Zemlin (MTZ) formulation [20].

Let n be the number of nodes. Let x_{ij} be the edge from vertex $i \in \{1, ..., n\}$ to vertex $j \in \{1, ..., n\}$ and let d_{ij} be the cost of edge x_{ij}. Take u_i as arbitrary non-negative integers. The MTZ model is represented as follows:

$$
\begin{array}{lll}
\text{minimize} & \sum\sum_{0 \le i \ne j \le n} d_{ij} x_{ij} & \\
\text{subject to} & \sum_{\substack{i=0 \\ i \ne j}}^{n} x_{ij} = 1 & \forall j \in \{1, ..., n\} \\
& \sum_{\substack{j=0 \\ j \ne i}}^{n} x_{ij} = 1 & \forall i \in \{1, ..., n\} \\
& u_i - u_j + n x_{ij} \le n - 1 & 1 \le i \ne j \le n \\
& x_{ij} \in \{0, 1\} & \forall i, j \in \{1, ..., n\} \\
& u_i \in \mathbb{Z}^{\ge} & \forall i \in \{1, ..., n\}
\end{array}
$$

A lot of research exists on tightening the relations and finding cutting planes for the MTZ formulation [3, 4, 10, 24, 26]. There also exist specific branch-and-cut algorithms for time-dependent variations on MTZ [6].

4 Analysis

The extension of the MAPF-problem with waypoints can be seen as a variation of TSP with a start and end location that are not necessarily the same. Therefore, to find a valid extension of BCP-MAPF with waypoints, we have to find where to solve this variation of TSP.

In the original BCP-MAPF algorithm, two large subproblems are solved. In the pricer, valid paths are found. In the ILP formulation, these paths are used to find an optimal solution. Solving TSP can be done in either of these subproblems.

This section analyzes the MAPFW problem by discussing important aspects found when extending BCP-MAPF with waypoints in certain ways: first by merging the MTZ formulation in the master problem, second by using TSP to find the optimal order of waypoints.

4.1 BCP-MAPF with MTZ

Since the minimization problems of BCP-MAPF and MTZ are similar, the two models can be combined in a single model.

Take the same definitions as the BCP-MAPF model. For every agent $a \in A$, define $W_a \subset V$ as the set of waypoints agent a has to visit, and define $s_a \in V$ and $g_a \in V$ as the start and goal vertices of agent a. For every agent $a \in A$, define $p_{vw} \in P_a$ as the path from v to w, where $v \in W \cup s$ and $w \in W \cup g$, and define $\lambda_{vw} = \lambda_{p_{vw}}$ as a variable representing to what extent p_{vw} is chosen. For every agent $a \in A$ and every path $p \in P_a$, define u_p as an arbitrary positive integer. The MAPFW problem can be modelled as follows:

$$
\begin{array}{ll}
\text{minimize} \sum_{a \in A} \sum_{p \in P_a} c_p \lambda_p & \\
\text{subject to:} & \\
\lambda_{vw} \ge 0 & \forall a \in A, p_{vw} \in P_a \\
\sum_{\substack{v \in W_a \cup s \\ v \ne w}} \lambda_{vw} \ge 1 & \forall w \in W_a \cup g \\
\sum_{\substack{w \in W_a \cup s \\ w \ne v}} \lambda_{vw} \ge 1 & \forall v \in W_a \cup s \\
u_v - u_w + (|W_a| + 2)\lambda_{vw} \le |W_a| + 1 & \forall v \in W_a \cup s, \forall w \in W_a \cup g \\
\lambda_{vw} \in \{0, 1\} & \forall v \in W_a \cup s, \forall w \in W_a \cup g \\
u_v \in \mathbb{Z}^{\ge} & \forall v \in W_a \cup s \cup g \\
\sum_{a \in A} \sum_{p \in P_a} x_v^p \lambda_p \le 1 & \forall v \in V \\
\sum_{a \in A} \sum_{p \in P_a} (y_e^p + y_{e'}^p) \lambda_p \le 1 & \forall e \in E
\end{array}
$$

However, such a formulation has several significant drawbacks. The first drawback is that the pricer has to generate paths to and from every waypoint. Note that the cost of a path in the MAPF problem depends on the starting time, since different starting times result in different conflicts on the graph. MTZ can theoretically work with this time-dependency. However, this also implies that for every waypoint, a path needs to be generated for each possible starting time, i.e. all possible end times of previous paths. This results in a worst-case time complexity for the pricer of $\mathcal{O}(|W|!)$. Similarly, the space complexity will be $\mathcal{O}(|W|!)$, since every path is passed to the master problem to verify whether it is part of an optimal solution. As noted earlier, the pricer is called frequently and should therefore be relatively efficient. This means that generating segments of the final path in the pricer is not viable.

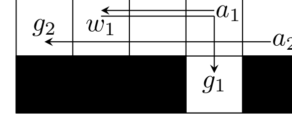

Figure 1: Example of a situation where a concatenation of shortest paths does not work. Both agents a_1 and a_2 need to move to goal locations g_1 and g_2 respectively, but agent a_1 had to pass waypoint w_1 before stopping at g_1.

The second drawback is that the shortest path is not necessarily the concatenation of shortest paths from point to point. When separate paths are created, a conflict in a certain path does not impact a previous path. In this way, the shortest path to a waypoint can create a situation in which the optimal solution is no longer reachable.

Such a situation is demonstrated in Figure 1. Agent a_1 moves to waypoint w_1 before going to goal g_1. Agent a_2 moves directly to g_2. While moving to w_1, agent a_1 will not get a conflict when using the shortest path. Therefore, the shortest path to w_1 generated by the pricer will always stay the same. When moving from w_1 to g_1, there is a conflict with agent a_2. This will eventually result in agent a_2 moving back to its start location so a_1 can pass to g_1, since there is no other possibility. However, the optimal solution has a_1 first

step aside to g_1 to let a_2 pass, before going to w_1. This means the algorithm is not optimal.

Both these problems can theoretically be removed by implementing the shortest path problem in the master problem. ILP formulations exist for time-continuous shortest path problems [23]. However, with the search space being every possible vertex in time and without easy pricing the variables, the algorithm will get a significant time-hit when running on larger instances.

To summarize, we derive three important points:

- Generating path segments in the pricer and combining them in the ILP formulation is not viable.
- The shortest path through waypoints is not the concatenation of shortest paths from waypoint to waypoint.
- Solving the pathfinding in the ILP formulation is not viable.

Combining these aspects shows that solving TSP in the ILP formulation is not efficient, meaning that solving TSP in the pricer shows more promise.

4.2 Waypoint order using TSP

Solving the shortest path algorithm with ordered waypoints on a time-extended graph can easily be done with a variation of the A* algorithm by using the heuristic to steer the agent to the next waypoint and keeping track of which waypoints have been visited.

Therefore it can be useful for the pricer to first compute the best order of waypoints using a traditional TSP algorithm and second compute the shortest path through the ordered waypoints. This way, the problem of the shortest path not being the concatenation of shortest paths, as described in section 4.1, is avoided.

However, TSP would calculate the order of waypoints for the shortest concatenation of shortest paths, which is not necessarily the same order as the general shortest path through the waypoints.

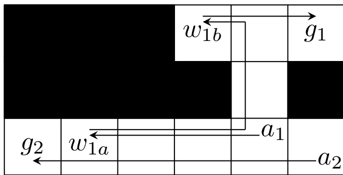

Figure 2: Example of a situation where determining the order of waypoints using TSP does not work. Both agents a_1 and a_2 need to move to goal locations g_1 and g_2 respectively, but agent a_1 had to pass waypoints w_{1a} and w_{1b} in arbitrary order before stopping at g_1.

An example is demonstrated in Figure 2. Initially, the shortest paths are $a_1 \rightarrow w_{1a} \rightarrow w_{1b} \rightarrow g_1$ and $a_2 \rightarrow g_2$. When agent a_1 moves from w_{1a} to w_{1b}, there is a conflict with agent a_2. Two situations are created: (1) agent a_2 waits at its starting location so a_1 can pass, similar to what happens in Figure 1, resulting in a total cost of $11 + 11 = 22$; (2) Agent a_1 waits out the conflict and moves towards w_{1b} later, resulting in TSP determining that the order $a_1 \rightarrow w_{1b} \rightarrow w_{1a} \rightarrow g_1$ is faster, resulting in a total cost of $15 + 5 = 20$. The branching algorithm will eventually choose the latter option because of its lower cost. However, the optimal cost with path $a_1 \rightarrow w_{1a} \rightarrow w_{1b} \rightarrow g_1$ and $a_2 \rightarrow g_2$ is $13 + 5 = 18$, if agent a_1 starts by moving up and down to let agent a_2 pass. Therefore, this is not an optimal algorithm.

The main takeaway of this part is that the previous observation of the sequence of shortest paths not being the overall shortest path, also has to be taken into account when finding the optimal order of waypoints.

This problem can be mitigated by always calculating paths from the start. A TSP solution can iteratively add waypoints to find the optimal order while calculating paths with the so far constructed order. More on this solution can be found in section 5.2.

5 BCP-MAPFW

In this section, 2 extensions of BCP-MAPF with waypoints are proposed. First, an algorithm that solves TSP in A* using a heuristic that steers agents towards all waypoints. Second, an algorithm that uses the TSP principle shown in section 4.2.

5.1 BCP-MAPFW-A*

BCP-MAPFW-A* is an optimal and complete algorithm for MAPFW that uses A* in the pricer to find shortest paths with multiple waypoints on the time-extended graph. The A* algorithm is a variation on the A* algorithm from the original BCP-MAPF algorithm that uses a heuristic to steer the agent towards the waypoints. This way, the complete paths are generated in the pricer, avoiding all problems found in section 4.1.

Define t and g as the current and goal locations respectively. Let W' be the set of yet unvisited waypoints and let d_{xy} be the expected cost of going from location x to location y. There are generally only a handful of y locations for which the distance needs to be calculated, specifically the waypoints and the goal location. Therefore, this can be pre-computed with a floodfill to any $x \in G$. The heuristic function h is:

$$h(t,g,W) = \begin{cases} h_0(t,g,W), & \text{if } |W| = 0 \\ h_1(t,g,W), & \text{if } |W| = 1 \\ h_2(t,g,W), & \text{if } |W| \geq 2 \end{cases}$$
$$h_0(t,g,W) = d_{tg}$$
$$h_1(t,g,W) = d_{tw} + d_{wg}$$
$$h_2(t,g,W) = \min_{w_1 \in W'} \Big(d_{tw_1} + \min_{w_2 \in W' - w_1} d_{w_2 g} + \sum_{w_3 \in W' - w_1} \min_{w_4 \in W'} d_{w_3 w_4} \Big)$$

This heuristic contains three situations: (1) There are no waypoints left to go to; (2) There is 1 waypoint left to go to; (3) There are multiple waypoints to go to. In case (1), the heuristic leads directly towards the goal location, similar to a classic A* heuristic. In case (2), the heuristic leads first to the waypoint, then to the goal. As long as the expected costs are optimistic, it is trivial to show that these heuristics are optimistic. Case 3 can be interpreted as: Take a waypoint $w_1 \in W'$. The

heuristic consists of the cost to w_1, the lowest cost from any waypoint in $W' - w_1$ to the goal and the sum of lowest costs from any waypoint in W' to every waypoint in $W' - w_1$. Take the minimum of all $w_1 \in W'$.

The cost to w_1 is the minimum cost needed to go to that waypoint. The lowest cost from any waypoint in $W' - w_1$ to the goal is the minimum cost needed to go to the goal, knowing that w_1 is not the last waypoint visited. The lowest cost from any waypoint in W' to every waypoint in $W' - w_1$ is the minimum cost needed to visit that waypoint. The minimum of the sum of these costs will always be optimistic.

Because all heuristics are optimistic, the A* algorithm is optimal.

5.2 BCP-MAPFW-TSP

BCP-MAPFW-TSP is an optimal and complete algorithm for MAPFW that uses a variation on TSP to find the optimal order of waypoints in the pricer. The TSP algorithm uses the principle shown in section 4.2.

```
Data: start s, goal g, waypoints W
Result: the shortest path from s to g through every
        waypoint in W
if |W| = 0 then
 | return A*({s, g})
PriorityQueue q;
q.push({s}, h(s, g, W));
while q is not empty do
 | (order, path) = q.pop;
 | if |W| = |order| + 1 then
 |  | q.push(order + g, A*(order + g));
 | else if |W| = |order| + 2 then
 |  | return path;
 | else
 |  | foreach w ∈ W - order do
 |  |  | q.push(order + w, A*(order + w) + h(w, g,
 |  |  |   W - order));
 |  | end
end
return no path;
```

Algorithm 1: TSP algorithm for BCP-MAPFW-TSP

The algorithm uses an A* implementation with ordered waypoints that uses the heuristic function $h_{A^*} = d_{sw_1} + d_{w_1 w_2} + \cdots + d_{w_n g}$ to steer from one waypoint to the next. Here d_{xy} is pre-computed using a floodfill, similar to the previous algorithm. This is a relatively efficient way of finding the shortest path through ordered waypoints.

The TSP algorithm itself is heavily based on a standard A* algorithm where the cost to each waypoint is re-calculated every cycle. The pseudocode for the TSP variation can be found in Algorithm 1. The h function referred to in the algorithm is the heuristic function from section 5.1.

Because of the principle shown in section 4.2, this algorithm generates the shortest path through the waypoints, meaning that this MAPFW solution is optimal. However, the constant re-calculating of the whole path can cause a performance-hit.

6 Experiments

The experiments compare the proposed MAPFW algorithms to each other and explore strengths and weaknesses.

Both BCP-MAPFW algorithms are implemented with SCIP [14] as BPC solver and CPLEX as LP solver. Both are single-threaded. The experiments are run on an AMD Ryzen 1700x at 3.4GHz. The algorithms will be benchmarked on 4 different grid-based maps, available on mapfw.nl[1]: 'random16x16' (16x16), 'cohenwh' (53x22), 'den206d' (190x50), 'lak503d' (194x194). The maps are in order of increasing size. The first map is a map filled with random obstacles, randomly generated for each run. The second map is a model of a warehouse by [5]. The last two are benchmark maps from the moving AI lab repository[2] [27]. All benchmarks are generated and collected by the MAPFW benchmarking server.

The experiment is divided in two parts. First, both algorithms are run on all maps with 5 waypoints and a varying amount of agents, with randomized locations and a time limit of 1 minute. For each number of agents, 50 runs are done, until no more solutions are found within the time limit.

Second, the first experiment is repeated, but with 5 agents and a varying number waypoints.

7 Results

Figure 3 and 4 show the results from the experiments as the percentage of instances solved within the time limit. From these graphs alone, three interesting observations can be made.

First, BCP-MAPFW-A* is heavily influenced by the size of the map. It outperforms BCP-MAPFW-TSP in the two smallest maps, while on lak503d, it is significantly outperformed by BCP-MAPFW-TSP. This is presumably because of the lack of direction that the heuristic function can give when waypoints are in opposite directions, in contrast to the directed heuristic of BCP-MAPFW-TSP. BCP-MAPFW-A* outperforms BCP-MAPFW-TSP on smaller maps because of the inefficiency of re-calculating intermediate paths.

Second, the performance of BCP-MAPFW-TSP on random16x16 and lak503d is similar, which is probably correlated to the lack of corridors, parts where agents cannot pass each other, which are more present on cohenwh and den206d.

Third, BCP-MAPFW-TSP cannot solve instances with more than 9 waypoints, regardless of the map, while BCP-MAPFW-A* is more dependable on the map and can solve instances with upto 17 waypoints on random16x16. This is because when waypoints are close together, BCP-MAPFW-A* can steer in their general direction, while BCP-MAPFW-TSP treats them separately. The inefficiency of re-calculating intermediate paths also has an impact here.

When the time limit is reached and the algorithm has found a (sub-optimal) solution, this solution is returned. In general, this solution is relatively close to an optimal solution, sometimes returning an optimal solution. Because of the lack

[1] https://mapfw.nl/
[2] https://movingai.com/

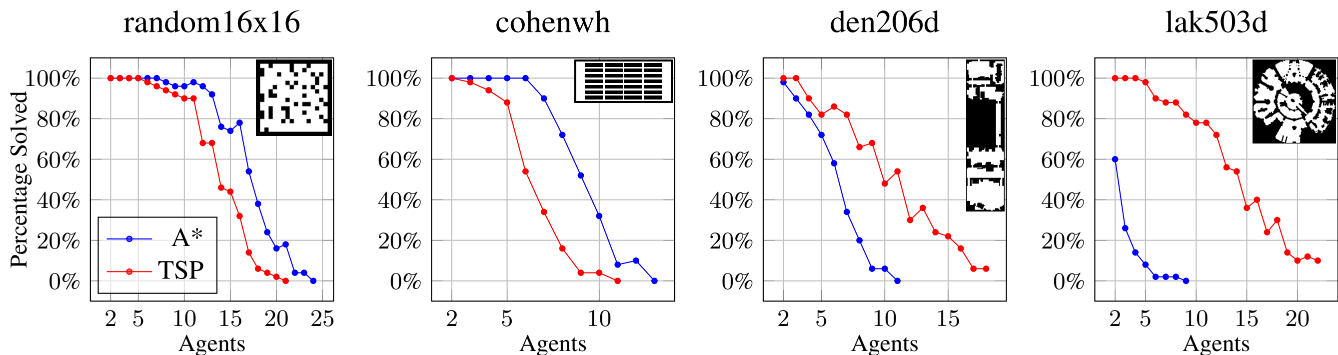

Figure 3: Percentage solved per number of agents. Higher is better.

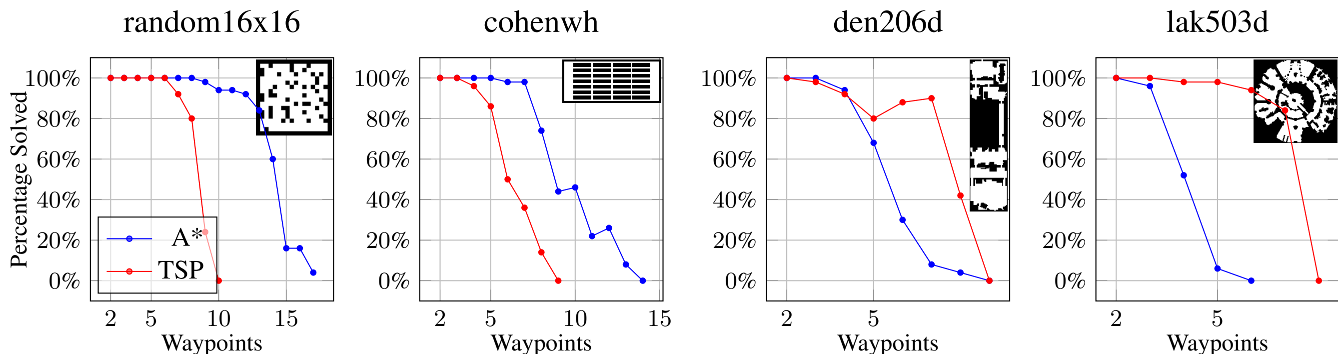

Figure 4: Percentage solved per number of waypoints. Higher is better.

of optimal solutions found by other algorithms in these instances, no exact significant numbers can be published on the performance of these sub-optimal solutions.

8 Responsible research

To improve research repeatability, this section contains information of this researches research environment.

The code used in this research is available on github[3], as is the source code of the original BCP-MAPF algorithm[4]. Both CPLEX and SCIP are freely available with an academic license. All benchmarks and benchmark results are available online[5].

9 Future research

One way of extending BCP-MAPF that has not been discussed in this paper is dynamically adding orders of waypoints to the pricer as the minimum cost of all paths becomes larger than the minimum cost of other orders. This way, the number of paths in the master problem is limited to only the paths that could result in an optimal solution, and the pricer can run the relatively efficient ordered waypoint A* algorithm that is used in section 5.2. A set of N best solutions to the TSP problem can be solved using a variation of the TSP algorithm in the same section.

[3]https://github.com/ACMichels/bcp-mapfw

[4]https://github.com/ed-lam/bcp-mapf

[5]https://mapfw.nl/

10 Conclusion

This paper provided two algorithms based on BCP-MAPF for solving the multi-agent pathfinding with waypoints problem, with a comparison between the two algorithms. In conclusion, BCP-MAPFW-A* works best on smaller maps and is more scalable to more waypoints, while BCP-MAPFW-TSP is better scalable to larger maps but has a tighter limit on the number of waypoints.

11 Acknowledgements

I thank Jesse Mulderij for the invaluable knowledge he provided. Without it, this project would not have been finished in the time that it has. I also thank Noah Jadoenathmisier and Stef Siekman for setting up the benchmarks on the benchmarking server.

References

[1] D. Atzmon, A. Diei, and D. Rave. Multi-train path finding. In *Twelfth Annual Symposium on Combinatorial Search*, 2019.

[2] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. Savelsbergh, and P. H. Vance. Branch-and-price: Col-

umn generation for solving huge integer programs. *Operations research*, 46(3):316–329, 1998.

[3] T. Bektaş and L. Gouveia. Requiem for the miller–tucker–zemlin subtour elimination constraints? *European Journal of Operational Research*, 236(3):820–832, 2014.

[4] G. Campuzano, C. Obreque, and M. M. Aguayo. Accelerating the miller–tucker–zemlin model for the asymmetric traveling salesman problem. *Expert Systems with Applications*, 148:113229, 2020.

[5] L. Cohen, S. Koenig, S. Kumar, G. Wagner, H. Choset, D. Chan, and N. Sturtevant. Rapid randomized restarts for multi-agent path finding: Preliminary results. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1909–1911, 2018.

[6] J.-F. Cordeau, G. Ghiani, and E. Guerriero. Analysis and branch-and-cut algorithm for the time-dependent travelling salesman problem. *Transportation science*, 48(1):46–58, 2014.

[7] R. J. Dakin. A tree-search algorithm for mixed integer programming problems. *The computer journal*, 8(3):250–255, 1965.

[8] G. B. Dantzig and P. Wolfe. Decomposition principle for linear programs. *Operations research*, 8(1):101–111, 1960.

[9] G. Desaulniers, J. G. Rakke, and L. C. Coelho. A branch-price-and-cut algorithm for the inventory-routing problem. *Transportation Science*, 50(3):1060–1076, 2016.

[10] M. Desrochers and G. Laporte. Improvements and extensions to the Miller-Tucker-Zemlin subtour elimination constraints. *Operations Research Letters*, 10(1):27–36, 1991.

[11] J. Desrosiers and M. E. Lübbecke. Branch-price-and-cut algorithms. *Wiley encyclopedia of operations research and management science*, 2010.

[12] F. G. Engineer, K. C. Furman, G. L. Nemhauser, M. W. Savelsbergh, and J.-H. Song. A branch-price-and-cut algorithm for single-product maritime inventory routing. *Operations Research*, 60(1):106–122, 2012.

[13] C. A. Floudas and X. Lin. Mixed integer linear programming in process scheduling: Modeling, algorithms, and applications. *Annals of Operations Research*, 139(1):131–162, 2005.

[14] A. Gleixner, M. Bastubbe, L. Eifler, T. Gally, G. Gamrath, R. L. Gottwald, G. Hendel, C. Hojny, T. Koch, M. E. Lübbecke, S. J. Maher, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, F. Schlösser, C. Schubert, F. Serrano, Y. Shinano, J. M. Viernickel, M. Walter, F. Wegscheider, J. T. Witt, and J. Witzig. The SCIP Optimization Suite 6.0. Technical report, Optimization Online, July 2018.

[15] I. E. Grossmann and J. Santibanez. Applications of mixed-integer linear programming in process synthesis. *Computers & Chemical Engineering*, 4(4):205–214, 1980.

[16] L. G. Khachiyan. A polynomial algorithm in linear programming. In *Doklady Akademii Nauk*, volume 244:5, pages 1093–1096. Russian Academy of Sciences, 1979.

[17] E. Lam and P. L. Bodic. New valid inequalities in branch-and-cut-and-price for multi-agent path finding. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, page (in print), 2020.

[18] E. Lam, P. L. Bodic, D. Harabor, and P. Stuckey. Branch-and-cut-and-price for multi-agent pathfinding. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, page (in print), 2019.

[19] H. Ma, J. Li, S. Kumar, and S. Koenig. Lifelong multi-agent path finding for online pickup and delivery tasks. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 837–845, 2017.

[20] C. E. Miller, A. W. Tucker, and R. A. Zemlin. Integer programming formulation of traveling salesman problems. *ACM*, 7(4):326–329, 1960.

[21] R. Morris, C. S. Pasareanu, K. Luckow, W. Malik, H. Ma, T. S. Kumar, and S. Koenig. Planning, scheduling and monitoring for airport surface operations. In *Workshops at the Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[22] J. C. Nash. The (dantzig) simplex method for linear programming. *Computing in Science & Engineering*, 2(1):29–31, 2000.

[23] A. B. Philpott. Continuous-time shortest path problems and linear programming. *SIAM journal on control and optimization*, 32(2):538–552, 1994.

[24] T. Sawik. A note on the miller-tucker-zemlin model for the asymmetric traveling salesman problem. *Bulletin of the Polish Academy of Sciences. Technical Sciences*, 64(3), 2016.

[25] G. Sharon, R. Stern, A. Felner, and N. Sturtevant. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219:40–66, 2015.

[26] H. D. Sherali and P. J. Driscoll. On tightening the relaxations of miller-tucker-zemlin formulations for asymmetric traveling salesman problems. *Operations Research*, 50(4):656–669, 2002.

[27] N. R. Sturtevant. Benchmarks for grid-based pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):144–148, 2012.

[28] P. R. Wurman, R. D'Andrea, and M. Mountz. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI magazine*, 29(1):9–9, 2008.