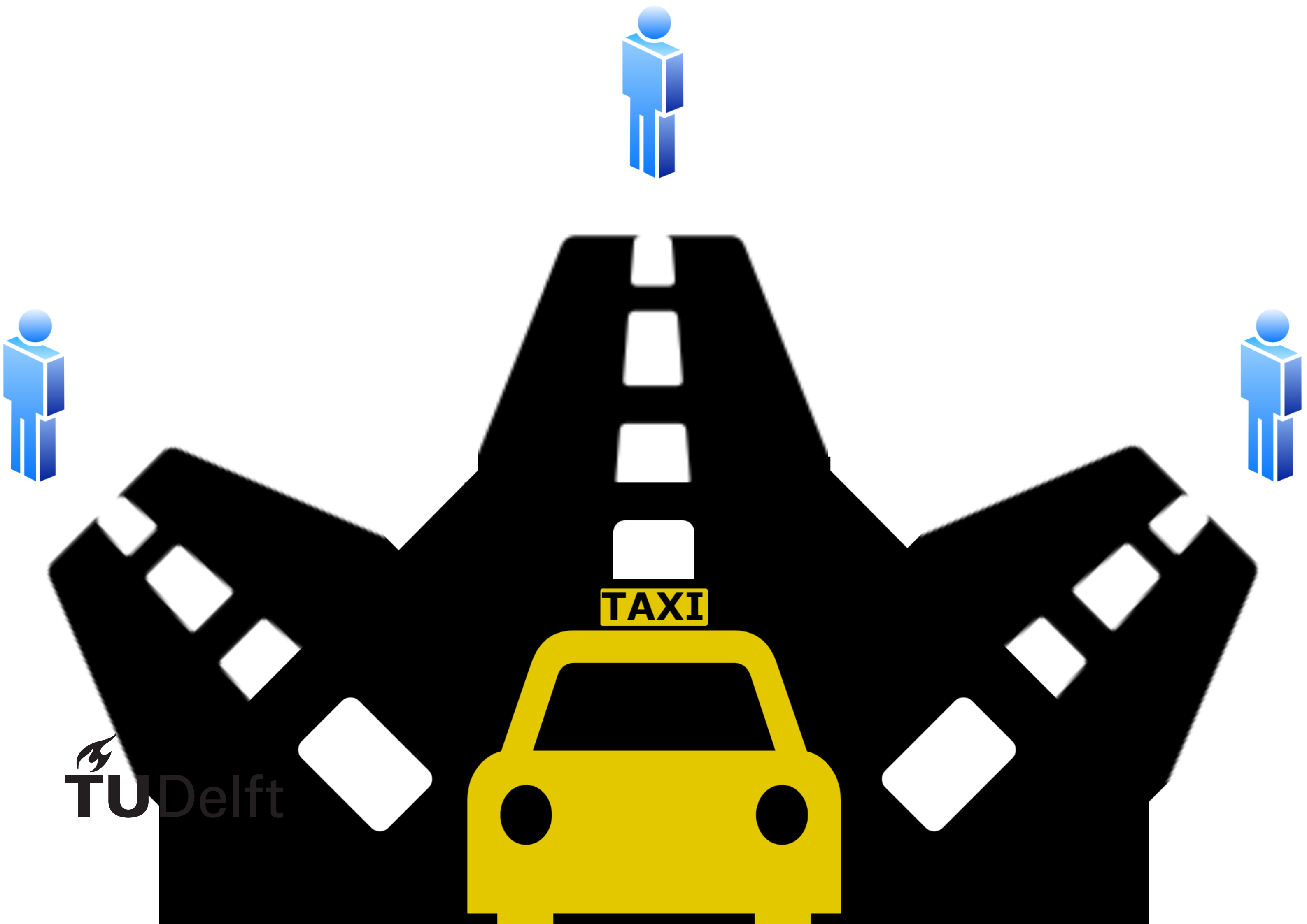
Cooperative multi-agent multi-target catching

Sander Waij



Cooperative multi-agent multi-target catching

by

Sander Waij

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday January 31, 2022 at 2:00 PM.

Student number:	4476085	
Project duration:	December 2, 2020 – January 31, 2022	
Thesis committee:	Dr. E. Demirović,	TU Delft, supervisor
	Dr. F. A. Oliehoek,	TU Delft

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TUDelft

Preface

Algorithms have always been interesting to me. Similar to how I like solving puzzles as efficiently as possible, finding good algorithms to solve the problem is half the fun. Therefore, I am glad to have been able to do my thesis within the Algorithmics group of the Delft University of Technology (TU Delft).

The biggest finding of this thesis is the significant increase of efficiency with targets that want to cooperate, both by not actively fleeing from the agents as well as sharing their path to the agents. This way, targets are assigned far more efficiently and agents get to their targets more efficiently. Furthermore, two variations are introduced to allow for more applications in the real world.

I would like to thank the people that helped me throughout my thesis. First of all, my supervisor, Emir Demirović, for the weekly meetings and helping me figure out the direction of this project. Secondly, my family and friends, for listening to my struggles and keeping me going. Finally, the additional member of the thesis committee, Frans Oliehoek, for taking the time and effort to be on the thesis committee and to give advice during the green light meeting.

Sander Waij
Delft, January 2022

Contents

1 Introduction **1**

2 Problem Definition **3**
2.1 Formal description . . . 3
2.2 Informal description . . . 4

3 Related work **7**
3.1 Single-agent pathfinding . . . 7
3.2 Single-agent chasing . . . 7
3.3 Multi-agent pathfinding . . . 8
3.4 Multi-agent chasing . . . 9
3.5 Multi-agent multi-target catching . . . 9
3.6 Map types . . . 11

4 Contributions **13**
4.1 Overview program . . . 13
4.2 Target path . . . 13
4.2.1 New formal description . . . 15
4.2.2 New informal description . . . 15
4.2.3 Using the target's path . . . 15
4.2.4 Example . . . 15
4.2.5 Algorithm for the intercept location . . . 16
4.2.6 Advanced algorithm for the intercept location . . . 17
4.2.7 Proof efficiency intercept location . . . 18
4.3 Variants . . . 20
4.4 Scalability . . . 20

5 Experiment **23**
5.1 Summary experiment . . . 24
5.2 Maps . . . 24
5.3 Target path difference . . . 24
5.4 Strategy difference . . . 27
5.5 Gap and stayput . . . 27
5.6 Variation 1 . . . 28
5.7 Stops . . . 31

6 Conclusion and Future work **35**

1 Introduction

Single-agent pathfinding is the act of finding a path between two locations. How to do this efficiently has been researched extensively, with many applications, such as GPS itinerary planning, robotics, and games. An example of single-agent pathfinding can be seen in Figure 1.1.

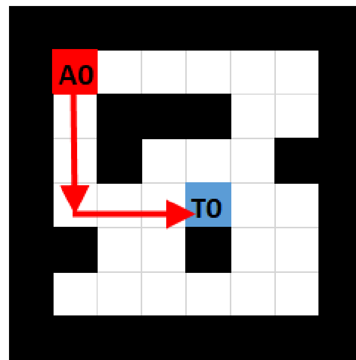


Figure 1.1: An example of single-agent pathfinding. The agent $A0$ found the red path to target $T0$.

This may be extended to have multiple agents find a path simultaneously. When multiple agents are doing their own pathfinding simultaneously you may need to consider avoiding collisions, depending on the problem definition. Sometimes the assignments indicating which agent should go to which location are not specified beforehand. This means that the assignments should be established by the algorithm. The complexity here is that not all agents can go to their nearest target location, as that would most likely mean that multiple agents are going to the same target location, leaving some target locations unreached. Therefore, an assignment algorithm should be used that provides each agent with their own target location (Xie et al., 2017). Examples of use cases of multi-agent pathfinding are robots running a warehouse, where they need to store items in the correct places while avoiding colliding with other robots, or autonomous cars that need to avoid other cars or obstacles (Stern et al., 2019).

This multi-agent pathfinding can be extended by allowing each target location to move. In these cases, it is called multi-agent multi-target catching. In this case, whenever the targets move, a better assignment could occur than what was previously the best. This makes the assignment algorithm a lot more complex. An example of such a case can be seen in Figure 1.2.

This multi-agent multi-target catching is the problem that we will be exploring in this thesis. It is not extensively studied, most of the literature is about non-moving targets or single-agent chases. For the literature that is presenting work on multi-agent multi-target chases, the algorithms best ensure that targets are being caught even if these targets are actively fleeing from the agents. However, the cases in which these targets are cooperative are not considered yet. An example of a case where the

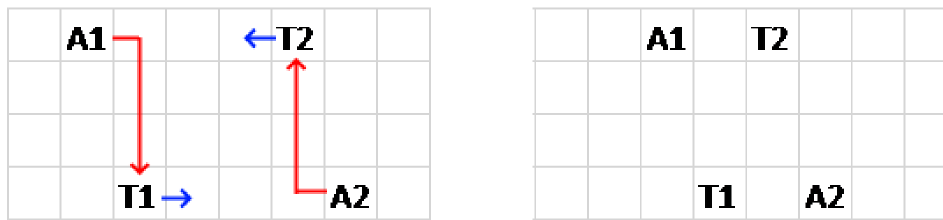


Figure 1.2: An example of how moving targets can change the best assignment. This is done on a 4-connected gridmap, i.e. no diagonals. In their starting position in the left picture, the distance of A1 to T1 is equal to its distance to T2. The same holds for A2. Thus, either assignment is fine. The assignment A1 to T1 and A2 to T2 is chosen. The agents are then moving along to the red arrow towards their assigned targets. The targets are moving along to the blue arrows. After one iteration, the situation is as shown in the right picture. There, the distance of both agents to their targets is still 4, while the distance to the other target is 2 for both. Thus, it is better to change the assignment to A1 to T2 and A2 to T1. Example by Xie et al., 2017.

target is cooperative is potential passengers wanting to get into a taxi. These people can be asked to wait somewhere, but sometimes they do not mind to keep walking if that is more efficient. It would not make sense for these passengers to start actively avoiding any taxi. They would rather create their own path, maybe past a few stores, and get picked up along that path as quickly as possible. This changes the type of problem from a multi-agent pathfinding problem into a cooperative multi-agent multi-target catching problem. Whenever the targets are cooperative such as these passengers, a lot more information can be acquired to make the assignments and the movements of the agents much more accurate.

The main contribution of this thesis is making catching cooperative targets more efficient. The main research question will therefore be: “Does having cooperative targets improve the efficiency of catching these targets compared to catching non-cooperative targets?” The changes that make these targets cooperative are that they are not actively avoiding the agents and that they are allowing the agents to see their path. The location in which each of the agents can catch each target can then be calculated. The assignments of agents to targets can then be based on these locations. Instead of moving to the current location of their target, the agents can now go to this previously-calculated location to catch their target. We have also created a few variants of the problem. One is a case where the agents do not disappear whenever they catch a target. Another is the addition of stops on the maps. Agents cannot stop on any other location than these whenever the stops are active. If there are no suitable locations on the target’s path, their path will be altered to include an efficient location to be intercepted in.

This thesis is divided into 5 different sections. First, we will discuss the problem in a more detailed fashion in the Problem Definition section. Then we will show what is already done in this field in the Related work section. We will show the main contributions in the Contributions section. The experiments and results will be shown in the Experiments section. The thesis will be concluded with the Conclusion.

2

Problem Definition

2.1. Formal description

The domain of the multi-agent multi-target catching problem can be formally described as a graph $G = (V, E)$, where V consists of all accessible locations and E consists of all the edges between every two vertices depicting locations that are directly adjacent, either horizontally, vertically, or diagonally. These edges have a distance of 1 for horizontal and vertical neighbors and $\sqrt{2}$ for diagonal neighbors.

The agents can be described as $Ag = \{A_0, ..., A_{n-1}\}$, where n is the number of agents. Similarly, the targets can be described as $T = \{T_0, ..., T_{m-1}\}$, where m is the number of targets.
The assignment can be described as the function $as : Ag \rightarrow T$, where $as(A_0) = T_1$ means that agent A_0 is assigned to target T_1.
Locations of agents and targets at a certain time can be described as the function $loc : Ag \times It \rightarrow V$, such that $loc(A_0, It) = V_1$ means that agent A_0 is in the location depicted by vertex V_1 at iteration It. Thus, the starting locations can be written as $loc(A_0, 0) = V1$, which means that agent A_0 has starting location V_1.

An action, or move, is a function $a : Ag_i \times Ag \times T \times It \times loc \rightarrow V$, in such a way that $a(A_0, \{A_0\}, \{T_0\}, 0, loc) = V_1$ means that considering agents $\{A_0\}$, targets $\{T_0\}$, iteration 0, and $loc(A_0, 0) = V_0$, the best move for A_0 is to go from V_0 to V_1.
A single-agent plan is a sequence of actions that an agent plans to do, denoted as $\pi_i = (a_0, ..., a_{n-1})$, where i is the agent number and n is the number of steps needed to eventually catch the target.
Every entity has a maximum of one move per iteration. Agents have exactly one move per iteration. Targets have no move once every *stayput* + 1 iterations to ensure agents can always catch the targets.

Whenever an agent and their target are in the same location in a graph, they both disappear. Thus, $Ag = \{Ag - A_i | loc(A_i) == loc(T_j), as(A_i) == T_j\}$ and $T = \{T - T_j | loc(A_i) == loc(T_j), as(A_i) == T_j\}$. A chase is done whenever no targets are left. The value of the solution can be viewed from either a summation or makespan perspective.

For summation, its value is

$$summation = \sum_{A_i \in Ag} |\pi_i| \tag{2.1}$$

For makespan, its value is

$$makespan = \max_{A_i \in Ag} |\pi_i| \tag{2.2}$$

With these notations, the example in Figure 1.1 can be formally described as:
$G = (V, E)$, where

$$V = \{(1,1), (1,2), (1,3), (1,4), (1,6), (2,1), (2,4), (2,5), (2,6), (3,1), (3,3), (3,4), (3,5), (3,6), (4,1), (4,3), (4,4), (4,6), (5,1), (5,2), (5,3), (5,4), (5,5), (5,6), (6,1), (6,2), (6,4), (6,5), (6,6)\}$$

and

$$E = \{((1,1),(1,2)),((1,1),(2,1)),((1,2),(1,3)),((1,3),(1,4)),((1,4),(2,4)),((1,6),(2,6)),$$
$$((2,1),(3,1)),((2,4),(2,5)),((2,4),(3,4)),((2,5),(2,6)),((2,5),(3,5)),((2,6),(3,6)),$$
$$((3,1),(4,1)),((3,3),(3,4)),((3,3),(4,3)),((3,4),(3,5)),((3,4),(4,4)),((3,5),(3,6)),$$
$$((3,6),(4,6)),((4,1),(5,1)),((4,3),(4,4)),((4,3),(5,3)),((4,4),(5,4)),((4,6),(5,6)),$$
$$((5,1),(5,2)),((5,1),(6,1)),((5,2),(5,3)),((5,2),(6,2)),((5,3),(5,4)),((5,4),(5,5)),$$
$$((5,4),(6,4)),((5,5),(5,6)),((5,5),(6,5)),((5,6),(6,6)),((6,1),(6,2)),((6,4),(6,5)),((6,5),(6,6))\}$$

with as format (x-coordinate, y-coordinate).

The other variables are filled in as follows:
$Ag = \{A_0\}$
$T = \{T_0\}$
$as(A_0) = T_0$
$loc(A_0, 0) = (1,1)$
$loc(T_0, 0) = (4,4)$

$$\pi_0 = \{a(A_0, Ag, T, 0, loc) = (1,2),$$
$$a(A_0, Ag, T, 1, loc) = (1,3),$$
$$a(A_0, Ag, T, 2, loc) = (1,4),$$
$$a(A_0, Ag, T, 3, loc) = (2,4),$$
$$a(A_0, Ag, T, 4, loc) = (3,4),$$
$$a(A_0, Ag, T, 5, loc) = (4,4)\}$$

2.2. Informal description

Catching multiple targets with multiple agents consists of two different parts. Firstly, the agents must be assigned a target to move towards. Secondly, the agent must get to the target.

The assignment of targets to agents is a problem that generally has multiple viable solutions. This is because there are multiple ways for a solution to be considered optimal. A solution may be considered optimal whenever all of the targets are caught as quickly as possible. However, another possible parameter to optimize is the distance that all agents need to traverse before catching the target. To illustrate this, Figure 2.1 shows that the optimal solution changes when optimizing for the other parameter. The optimal assignment is computed of the two agents, A0 and A1, to targets T0 and T1. On the left, it is optimized for speed, being done in MAX(5,6) = 6 iterations, while for the right assignment, the targets are caught in MAX(1,7) = 7 iterations. The right assignment is optimized for the least number of agent moves. This assignment has 1 + 7 = 8 agent moves, while the left assignment has 5 + 6 = 11 agent moves. In this example, the targets are assumed to not move. This is different than in a real case, so the agents will not generally catch the targets at the locations indicated. However, the assignment in its original form also only takes the current location of the targets into account, so how the assignment is done will be the same as in this figure.

While the agents are going towards their target, a different assignment of agents and targets may become better. This is caused by targets moving into the direction of different agents and agents moving in the direction of different targets as well. An example of this can be found in Figure 2.2. Here, after the first move, A0 would be better off going after T1 instead of T0 which it was originally planning to do, and A1 should go after T0 instead of T1 as this assignment would give 2 iterations instead of 4 and 4 agent steps instead of 8. Because of cases like this, it is beneficial to recalculate the assignment every few iterations to make the agents move as efficiently as possible.

Whenever an agent reaches a target, they both disappear and are unable to move or catch/be caught. All entities are allowed to be in the same location, collisions are ignored. An agent can only catch a target that is assigned to it.

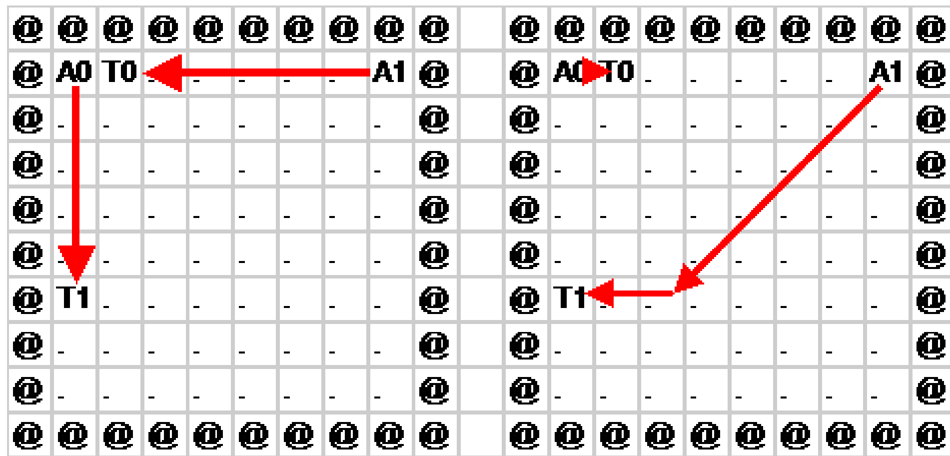


Figure 2.1: Optimizing for speed (left), with MAX(5,6)=6 iterations (left) versus MAX(1,7)=7 iterations (right), and optimizing for least number of agent moves (right), with 5+6=11 moves (left) versus 1+7=8 moves (right).

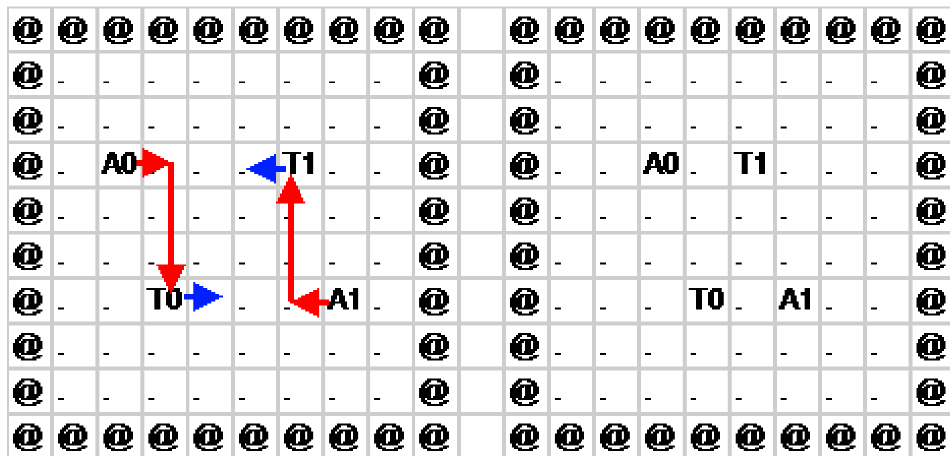


Figure 2.2: Using a 4-connected gripmap (no diagonals). In the left image, the moves can be seen. The right image show the resulting locations after one move. In the right image, changing targets gives a better result. Example by Xie et al., 2017.

What differentiates examples such as the dynamic taxis from police chases is that the potential passengers want to get caught up by the taxis, while the criminals fleeing from the police do not want to get caught. This distinction allows for cooperation between agent and target, making the chases even more efficient. This will be the cornerstone of this thesis.

3 Related work

Cooperative multi-agent multi-target catching has come from years of previous research in the field. This started from regular pathfinding to catching a moving target, first with a single agent and later with multiple agents. The main components of cooperative multi-agent multi-target catching are the assignment of agents to targets and the movement of the agents. The advantage over multi-agent multi-target catching is that when using cooperative targets, the agents will know where the target will be in later iterations. This can lead to more accurate assignments and thus more accurate movements.

3.1. Single-agent pathfinding

Before getting agents to catch multiple moving targets, a single agent should first be able to find their way to a stationary target efficiently. This can be done with Dijkstra's algorithm shown in Dijkstra et al., 1959. Problem 2 in Dijkstra's paper gives us an algorithm that finds the path of the minimum total length between two given nodes.

Hart et al., 1968 has a heuristics approach to single-agent pathfinding in their algorithm A*. This means that estimates are used to determine the shortest path. A commonly used evaluation function for determining the cost of an optimal path through node n is $f(n) = g(n) + h(n)$, where $g(n)$ is the cost to get from the starting node to n and $h(n)$ is the cost to get from n to the preferred goal node of n. $g(n)$ and $h(n)$ can be estimated, which will together give an estimation for $f(n)$. Whenever the estimate for $h(n)$ is a lower bound for $h(n)$, A* is admissible.

Koenig and Likhachev, 2006 improves on this A* algorithm with Adaptive A*, which updates these h-values after each search with the increased information to make them more precise.

3.2. Single-agent chasing

Some targets will not stay in the same place. For a moving target, a single-agent chase is needed.

Ishida and Korf, 1991 presents an algorithm in which heuristics of distance between two points are used in a single-agent single-target chase. As the chase continues, these distance values are getting more accurate. Because heuristics are used, calculating the moves in this chase is done quicker in exchange for fully being optimal. In order to guarantee that the agents catch these moving targets, the targets must be slower than the agents or occasionally make an error in avoiding the agent.

Koenig et al., 2007 presents another algorithm for a single-agent chase called MT-Adaptive A*. This extends the previously-mentioned Adaptive A* (Koenig and Likhachev, 2006) by changing the goal state between searches. It thus updates the h-values between searches like in Adaptive A*, but now also corrects them for the change in goal state.

Sun et al., 2008 presents an algorithm generalizing Adaptive A* called Generalized Adaptive A* (GAA*). This algorithm allows for decreasing the action costs over time by correcting these h-values for those after those decreases.

3.3. Multi-agent pathfinding

Stern et al., 2019 presents an overview of multi-agent pathfinding (MAPF) variations. They state that the overarching goal of multi-agent pathfinding is to find a solution that can be executed without collisions. They define five different types of conflicts, but they acknowledge that it is "certainly not a complete set of all possible conflicts". They also mention that there are two common assumptions for how agents behave at their targets, being either staying at the target and or disappearing whenever the target is reached. The objective functions makespan and sum of costs are also defined, being the number of time steps needed for all agents to reach their target and the sum of time steps required by each agent to reach its target, respectively.

Sharon et al., 2015 introduces a Conflict Based Search (CBS) algorithm for multi-agent pathfinding. It uses single-agent searches at its core and creates a conflict tree to find the conflicts between agents. Furthermore, Sharon et al., 2015 introduces Meta-Agent CBS (MA-CBS) which groups agents together that often conflict, treat them as a single composite agent, and work out the conflict separately. This makes it more flexible than their normal CBS algorithm.
Gange et al., 2019 remarks that CBS still does not work well with maps with many agents. This is partly due to it not recognizing subproblems and thus continuously solving for the same agents in different branches of the tree. With their algorithm Lazy CBS, they solve this problem by storing the reasons for conflicts.

Stern, 2019 shows the heuristics approach to pathfinding. One of those is A^*. This algorithm expands neighbors starting in the source node and maintains their lowest-cost path found, parent node on that path, and a heuristic estimate of the cost of the path from this vertex to the target vertex. The heuristic estimate is the shortest path to the target vertex, disregarding any possible collisions that should be avoided. However, as both the size of the search space and the branching factor, here being the average outgoing degree of a vertex, are exponential in the number of agents, problems with a large number of agents cannot be solved with A^*.
One way to combat this lack of scalability is by decoupling the multi-agent pathfinding to separate pathfinding problems with as little interaction as possible. One of the ways to do this is by prioritized planning. This solves the single-agent pathfinding problems in sequence, while not being allowed to create a conflict with plans made by earlier agents. The problem with this approach is that it is not complete nor optimal.

Optimal algorithms, i.e. algorithms that output the best solution possible, are A^*-based (Standley, 2010, Wagner and Choset, 2015), Constraint Programming (CP) (Barták et al., 2017, Surynek, 2010), Conflict-based search (CBS) (Sharon et al., 2015), and Increasing cost tree search (ICTS) (Sharon et al., 2013) algorithms. The first two are effective for small graphs, while the latter two are effective for large graphs.
Approximately optimal multi-agent pathfinding algorithms are generally fast but do not always give the optimal solution. These algorithms accept a parameter $\epsilon > 0$ and return a solution that is at most $1 + \epsilon$ times the cost of an optimal solution. These are generally based on optimal algorithms and thus the algorithms mentioned here will also be based on the optimal algorithms previously mentioned. As such, an A^*-based approximately optimal algorithm is *Weighted* A^* that uses a $g + (1 + \epsilon)h$ evaluation function to choose which node to expand (Pohl, 1970). No approximately optimal ICTS-based algorithm was found, but there is a version in which moving an agent across different edges can have different costs. This algorithm is based on the Extended ICTS algorithm (Walker et al., 2018). Enhanced CBS is an approximately optimal MAPF algorithm based on CBS Barer et al., 2014). Suboptimality can be introduced in the pathfinding of a single agent, by using an approximately optimal algorithm such as A^*-based algorithms. However, suboptimality can also be introduced in the conflict search, by using focal search, a heuristic search framework introduced by Pearl and Kim, 1982. Lastly, the paper mentions eMDD-SAT, an approximately optimal MAPF algorithm from the CP family that models the MAPF problem as an SAT problem.

3.4. Multi-agent chasing

Hollinger et al., 2009 presents a bounded approximation algorithm using implicit coordination. They use a reward function for all different possible actions for every searcher (in our case called an agent). The reward is higher whenever the target is quickly caught. The solver uses a finite-horizon path enumeration, meaning that it will plan a finite number of steps ahead and chooses the best path to that horizon.

3.5. Multi-agent multi-target catching

Multi-agent multi-target catching is the act of multiple agents chasing and catching multiple moving targets as efficiently as possible. The difference with the multi-agent chasing of the previous section is that this has multiple moving targets instead of one. One of the challenges is assigning the correct agent to each target. In the scenario where the targets are moving collisions are usually ignored, i.e. there can be more than one entity in a single location.

Xie et al., 2017 shows three different assignment criteria: summation cost criterion, which aims to minimize the total distance between the agent and its assigned target, makespan criterion, which aims to minimize the number of iterations needed to catch all targets, and mixed-cost criterion, which is a combination of both.
The strategy based on the summation cost criterion uses a Hungarian algorithm (Kuhn, 1955) that selects the assignments that, together, have the lowest summed distance. This makes sure that the amount of fuel used is the least possible.
How this works is as follows:
First, turn the assignments into a matrix of the number of agents by the number of targets and fill the cells with the distance between the corresponding agent and the corresponding target. If the matrix is not a square matrix, make it a square matrix and fill the remaining row(s) or column(s) with zeros. This way, you end up with an $n \times n$ matrix, where n is the maximum of the number of agents and the number of targets.
Second, the Hungarian algorithm will be performed on this matrix:

1. The smallest entry in each row must be subtracted from all of the entries in the row. This makes that smallest entry equal to 0.

2. The smallest entry in each column must be subtracted from all entries in the column. This makes that smallest entry also equal to 0.

3. Cover all zeros in the matrix with a minimum number of lines. If the number of lines is equal to n, there exists an optimal assignment among the cells with zeros. The agents and targets corresponding to those cells will be assigned to each other. If the number of lines is less than n, continue with step 4.

4. Subtract the value of the smallest element not covered in step 3 from all of the uncovered elements (including itself). Thus, creating an extra zero. Go back to step 3.

The strategy based on the makespan criterion minimizes the number of iterations needed by removing assignments that require a high number of steps until a perfect bipartite graph is no longer possible, i.e. not every agent is assigned to exactly one target. The check of whether a perfect bipartite graph is still possible is done with the Hopcroft-Karp algorithm (Hopcroft and Karp, 1973). Assignments with a high number of steps take up a high number of iterations and therefore using these assignments will not minimize the number of iterations. It then uses a combination of remaining assignments that together form a perfect bipartite graph.
The last strategy, based on the mixed-cost criterion, uses a combination of the two aforementioned strategies. It removes the assignments with a higher number of steps and obtains the assignments that are equal to or below the minimum number of iterations needed, just like in the MKS strategy. For these remaining assignments, it chooses the combination of assignments that, together, have the lowest summed distance, calculated with the Hungarian algorithm, as is done in the DIS strategy.

Another way to solve the multi-agent multi-target catching problem is by using multi-agent reinforcement learning (MARL). Every agent will perform a series of decisions based on what it sees and knows about the environment. Their behavior can be programmed, but often the behaviors are learned online (Busoniu et al., 2008). This is done by trial-and-error and evaluated by the reward values that the agents receive whenever they get to a new state. Just like the other solutions here, reinforcement learning is more difficult for multiple agents. In this case, the reason is defining the learning goal. MARL can be divided into three groups, fully cooperative, fully competitive, and a mix of the two (Zhang et al., 2021). A fully cooperative setting would be most suitable for our problem. For cooperation, the agents need to keep track of what information other agents are learning as well. Since the agents are learning the behaviors of the other agents and targets online, they are not as exact as when using the strategies mentioned in Xie et al., 2017 but are more flexible.

Sometimes agents cannot see the whole map at once nor are they controlled centrally. This can be modeled using a decentralized partially observable Markov decision process (Dec-POMDP) (Bernstein et al., 2002). In the example of Oliehoek, 2012, as can be seen in Figure 3.1, the basic idea of the model can be found, using a two-agent case. Each agent will take action independently, which leads the environment to change. The environment then rewards the agents based on their new state and actions. Finally, the agents will observe their new state.
In a Dec-POMDP the planning often takes place before execution in a centralized offline phase. The joint plan will be computed and then sent to all of the agents. The execution phase after will be online, but decentralized as illustrated in Figure 3.1. The only way the agents can communicate in this phase is by using states of the environment. As they have limited observations, the planning phase is the most important.

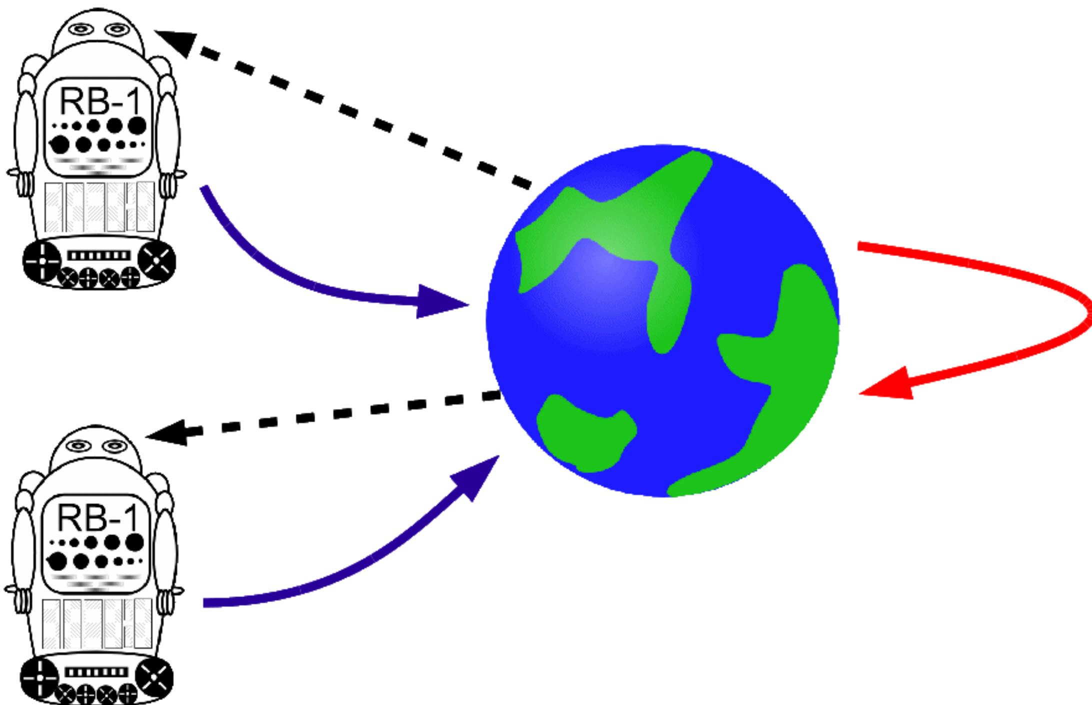


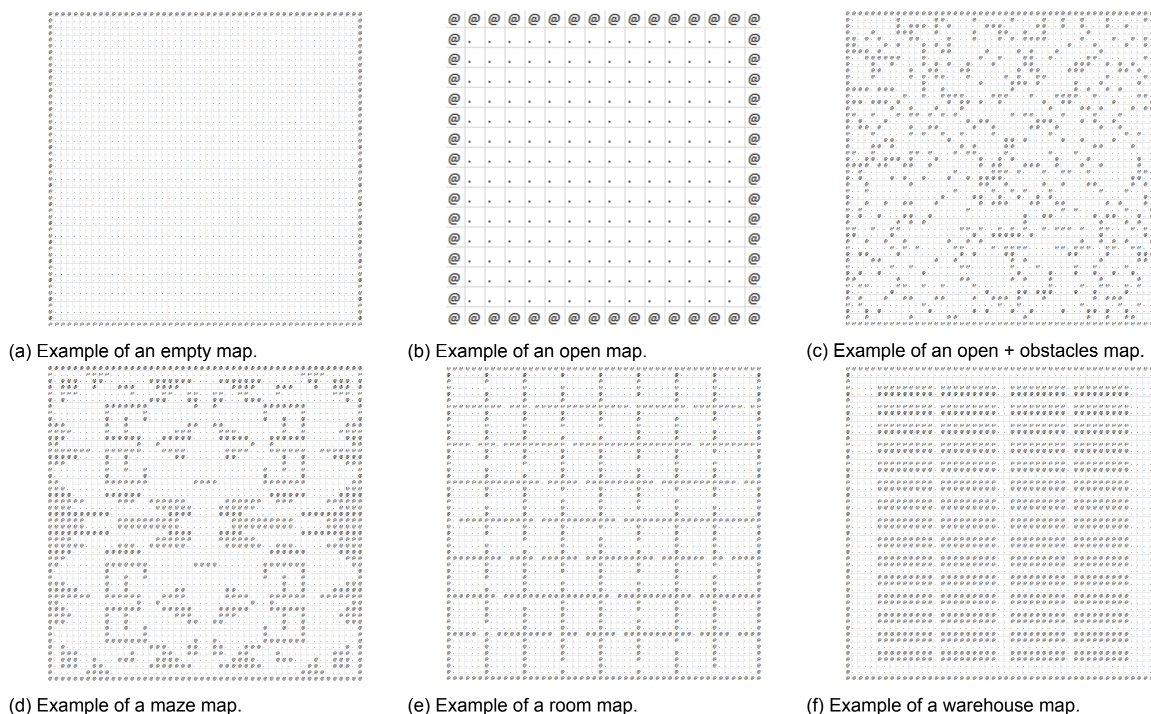
Figure 3.1: Execution of a decentralized POMDP. Made by Oliehoek, 2012.

Another problem that is a subclass of Multi-agent Markov Decision processes (MMDPs) that needs a decentralized approach is Spatial Task Allocation Problems (SPATAPs) (Claes et al., 2015). Claes et al., 2015 introduces the idea of Guéant et al., 2011 to the field of online planning in multi-agent systems. The key idea is to predict the future actions of the other robots based on their current location. For this, the robots have to communicate their location to the other robots. Unlike in the Dec-POMDP, this is possible in SPATAPs mid-execution, as the planning phase is done at every time step. This way

a robot does not need to take every target into account in the planning, only the ones that are predicted to not be caught by other robots. The planning is done online at every time step, allowing the algorithm to react to changes in the environment. Claes et al., 2015 uses an algorithm for settings with negative interactions, which discounts rewards for going to a target that another robot is predicted to be going towards as well. They combine this with the Phase-Approximation approximation algorithm to make this online planning not as badly scalable.
Claes et al., 2017 expands on this even further by showing that Monte Carlo Tree Search (MCTS) methods are effective in SPATAPs, especially for more complex problems. Additionally, they extend SPATAPs with a drop-off point to make the model more realistic.

3.6. Map types

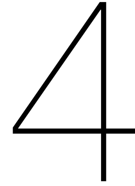
Different types of maps also yield different results. For that reason, we made sure that multiple types of maps were represented. Stern et al., 2019 displays a vast array of map types, for which we ensured to include at least one of each in our experiments.
Being able to run the algorithms on all of these map types also ensures that the algorithm works for every map in which all locations can reach all other locations. These different types consist of: Empty, Open (small empty map with a lot of agents and targets), Open + obstacles (empty map with randomly spread obstacles, also called Random), Maze, Room, and Warehouse. Examples of these types of maps can be found in Figure 3.2.



(a) Example of an empty map. (b) Example of an open map. (c) Example of an open + obstacles map.

(d) Example of a maze map. (e) Example of a room map. (f) Example of a warehouse map.

Figure 3.2: Examples of all the types of maps.



Contributions

In the assignment strategies of Xie et al., 2017, the location of the target at that moment in time is used for both assigning agents to targets and for determining which direction each agent should move in. This means that the agents will always be chasing the targets and the time it takes them to catch the target entirely depends on the distance between the target and the agent and the difference in speed. However, since we are picturing scenarios where the targets are cooperative, the targets can help the agents get to them quicker. The main way this thesis allows the agents to be quicker is by adding the knowledge of the targets' paths to the agents.
The advantage of using this exact method over something like the multi-agent reinforcement learning is how optimal it is. With the cooperation of the targets every move is known beforehand, so we better make use of it. MARL, Dec-POMDP, and SPATAPs are all more flexible when not all information is known. However, they are generally not as precise when all moves are known.

In this chapter, we will discuss the additions of this thesis to the current literature and their impact. We will first create an overview of the program. We will then highlight the importance of knowing the target's path and show how this path can be used. The different variants created will be introduced in Section 4.3. Lastly, the scalability of the program is discussed.

4.1. Overview program

In the pseudocode of Algorithm 1, an overview of the program is shown. In iteration -1, the paths of the targets are determined. This is to ensure that those are known before any calculations or moves are done. After iteration -1, the general sequence of events per iteration is:

1. Assign targets to agents
2. Move all agents
3. Move all targets

The assignments are only done once every number of iterations equal to the variable gap. This saves time and often does not diminish the results as explained in Section 5.5. The agents can move every iteration. The targets, however, have to stop for one iteration after a number of iterations equal to the variable $stayput$ has gone by. This indicates the speed difference between the agents and the targets, where a lower $stayput$ indicates a higher speed difference, with 1 being the lowest $stayput$ at twice the speed.

4.2. Target path

In the literature, the targets are often trying to not get caught for as long as possible. This means that they react to where each agent is going and alter their path in such a way as to make it as difficult as possible for the agents to catch them. In our use cases, the targets want to get caught, so it would not make sense for them to actively try to get away from the agents. Therefore, the targets will instead

```
1  for All runs do
2  |   iteration ←-1
3  |   gap_counter ←gap
4  |   stayput_counter ←0
5  |   while not all targets caught do
6  |   |   if iteration = -1 then
7  |   |   |   Determine paths of targets
8  |   |   else
9  |   |   |   if gap = gap_counter then
10 |   |   |   |   Determine distance and #steps between all agents and targets
11 |   |   |   |   Assign targets to agents according to assignment strategies, distances, and
   |   |   |   |     #steps
12 |   |   |   |   gap_counter ←1
13 |   |   |   else
14 |   |   |   |   gap_counter ←gap_counter + 1
15 |   |   |   end
16 |   |   |   for Agent agent ∈ all_agents do
17 |   |   |   |   Move towards assigned target
18 |   |   |   |   if On location of assigned target then
19 |   |   |   |   |   Catch target
20 |   |   |   |   |   Disappear
21 |   |   |   |   end
22 |   |   |   end
23 |   |   |   if stayput = stayput_counter then
24 |   |   |   |   for Target target ∈ all_targets do
25 |   |   |   |   |   Move along own path
26 |   |   |   |   |   if On location of assigned agent then
27 |   |   |   |   |   |   Get caught by agent
28 |   |   |   |   |   |   Disappear
29 |   |   |   |   |   end
30 |   |   |   |   end
31 |   |   |   |   stayput_counter ←0
32 |   |   |   else
33 |   |   |   |   stayput_counter ←stayput_counter + 1
34 |   |   |   end
35 |   |   end
36 |   |   iteration ←iteration + 1
37 |   end
38 end
```

Algorithm 1: The overview of the whole program

take the shortest path to a few random locations.
The path of a target, given the target has a predefined path and will not deviate from it, is very valuable for an agent to know. With this information, along with the knowledge of their difference in speed, agents can calculate in which location they can intercept their target. While the literature uses the current location of the target for determining the movement of the agent and the assignment process, the location in which the agent can intercept the target can now be used. This makes both the moves of the agent and the assignment much more accurate.
For this, line 10 of Algorithm 1 does not determine the distance and number of steps between the agent and the target, but between the agent and the location in which it can first catch the target.

4.2.1. New formal description

The formal description of the cooperative variant of the multi-agent multi-target chase is very similar to the non-cooperative one.
The objective values of this variant are the same as of the non-cooperative variant: catching all of the targets with values for both total agent distance (Equation 2.1) and number of iterations until the last target is caught (Equation 2.2).
Just like in the non-cooperative variant, the domain can be described as a graph $G = (V, E)$ with V depicting the locations and E the edges of adjacent locations. The difference is that the agents can now access the path of the targets.
This path of a target can be described as a list $path = \{V_k, ..., V_l\}$, where both V_k and V_l are independent locations, along with all other locations in the list. For each of the locations in the list, the next one in the list (if applicable) must be a neighboring location, either vertically, horizontally, or diagonally. When a target moves, it will go from its current location in the path to the next location in the list.

4.2.2. New informal description

The problem definition of the chase while using the targets' paths as references of where to go is quite similar to the original problem described in Chapter 2. The difference is that an agent can now access the path of the targets and change their strategy based on this extra knowledge.

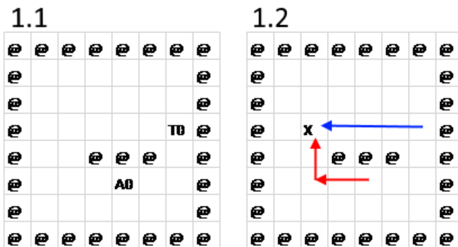
4.2.3. Using the target's path

The original assignment function uses the current locations of the targets, calculates the number of steps and distance to each agent using Dijkstra's shortest path algorithm (Dijkstra et al., 1959), and inputs these number of steps and distances into their respective assignment strategies. The assignment function of the cooperative variant does not use the current location of the target as its reference point but instead uses the first interceptable location where the agent can catch the target on their path.
The moves of the agents are also not in the direction of the target, but rather in the direction of this previously-mentioned first intercept location.

4.2.4. Example

In Figure 4.1, an example is shown for how much of an improvement knowing the path of the target can be. A 4-connected gridmap is used, meaning that every move will be either up, right, down, or left and therefore always be a distance of 1.
The path of the target, using the notation explained in Subsection 4.2.1 is:
$path = \{(6,3), (5,3), (4,3), (3,3), (2,3), (2,4), (2,5), (2,6), (1,6)\}$.
In the top half of the Figure, when the agent knows the path of the target, it will go to the left, as it knows it can intercept the target at the location X (location $(2,3)$) shown. This chase will last for 4 iterations, as the agent needs 4 iterations to get to location X. In the bottom section, which will happen if the agent does not know the path of the target, the agent will go right, as that is the fastest way to go from its own location (location $(4,5)$) to the current location of the target (location $(6,3)$). In picture 2.3, only the agent will take a step as with a *stayput* of 4, the target must stay in the same location for one iteration after every 4 iterations. The chase continues in picture 2.4 and ends up lasting for 12 iterations. As the agent moves every iteration with a distance of 1, the agent distance is equal to the number of iterations,

being 4 with path knowledge, and 12 without. This means that in this example, not knowing the target's path costs the agent three times as much time and distance.



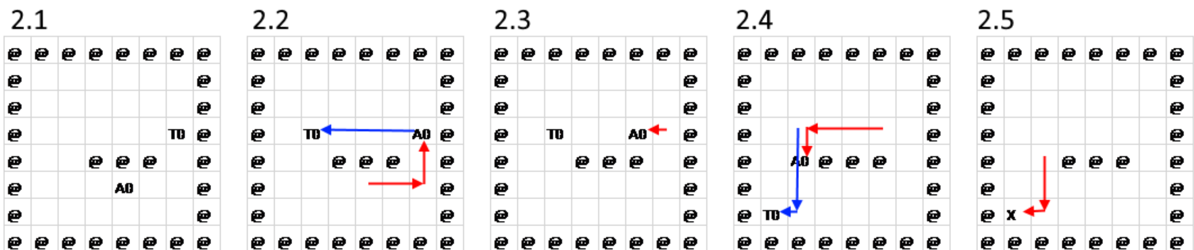


Figure 4.1: The difference between an agent knowing the target's path (top) and not knowing the target's path (bottom). Using a 4-connected gridmap (no diagonals). The *stayput* parameter is 4, so after every 4 moves, the target has to stand still for one iteration. This happens in 2.3 . The red arrows are in the direction of the moves that the agent is making. The blue arrows are in the directions of the target's moves.

4.2.5. Algorithm for the intercept location

This pseudocode for the algorithm for determining the first interceptable location can be found in Algorithm 2. The simplified version where every target path has at least one stopping location and does not allow changes is explained here. The version that does allow changes is explained in Subsection 4.2.6.

The algorithm has a for loop, initialized at line 2, that goes over the indices of the path. It stores the location indicated by this index in loc. The number of agent steps is determined at line 7 by taking a Dijkstra's shortest path from the agent's current location to loc with distances of all edges set to one to mimic the number of steps. This is done with the algorithm $steps_to_location$. The number of target steps is the same as the index of the location, as the path starts with the target's current location and every location in the path is adjacent to the previous location. These target steps are stored in $target_steps$ at line 8. However, since the target has to stop after every $stayput$ iterations, the number of iterations for the target to arrive is not equal to the number of steps. Thus, the iteration of arrival is calculated at line 10 and stored in $target_arrives$. The iteration of the departure of the target is calculated at line 12. This iteration is increased by one as the agent can move earlier than the targets, allowing them to catch the target just before they leave.
Line 16 checks if the agent arrives at the location earlier than or at the same time as when the target is leaving location loc, i.e. $agent_steps \leq target_leaves$, and if location loc can be stopped in. If both of these checks are true, the agent can catch the target in this location. Whenever this happens, the location loc will be returned as the first interceptable location. If there is no stoppable location for which the agent can intercept the target, the algorithm will return the last stoppable location after the for loop at line 20. This last stoppable location is maintained at line 5.

The algorithm is optimized to have the least number of steps needed for each target. This could, however, mean that there is another location further on the path that is closer to the agent but for which

```
Input  : A path path of the target
         The agent agent for which the first intercept location is checked
         A function stoppable that takes as input a location and outputs whether the location
         can be stopped in
Output: The first interceptable location
1  last_stoppable_location ←-1
2  for i ←0 to path_size do
3  |  loc ←i^th location in path
4  |  if stoppable(loc) then
5  |  |  last_stoppable_location ←loc
6  |  end
7  |  agent_steps ←steps_to_location(agent_location, loc)
8  |  target_steps ←i
9  |  This is the iteration at which the target arrives at this location
10 |  target_arrives ←(target_steps + (MAX(target_steps + target_stop_counter - 1, 0) / stayput))
11 |  This is the iteration at which the target leaves this location
12 |  target_leaves ←(target_steps + (target_steps + target_stop_counter) / stayput) + 1
13 |  The next two variables are not used in this simplified algorithm
   |   but show what the values are when using this location
14 |  iterations_to_catch ←MAX(agent_steps, target_arrives)
15 |  distance_to_catch ←distance_to_location(agent_location, loc)
16 |  if stoppable(loc) & agent_steps ≤ target_leaves then
17 |  |  return loc
18 |  end
19 end
20 return last_stoppable_location
```

Algorithm 2: The algorithm to find the first interceptable location on the target's path

the target takes longer to travel. This will be elaborated on further in Chapter 5.

4.2.6. Advanced algorithm for the intercept location

The advanced pseudocode as explained in this subsection is for when a target is allowed to go off of its original path. The pseudocode is shown in Algorithm 3.
Just like with the simplified version shown in Figure 2, the for-loop goes over all of the indices in the path and stores the corresponding location to loc. This is done at line 6. Just like in the simplified version, the agent steps, target steps, iteration of arrival of the target, and iteration of departure of the target are stored. This is done at lines 10, 11, 13, and 15, respectively.

Now what is different here is that for every index in the $path$, it is checked whether a different location can be inserted on that index as a more favorable intercept location. The target is already on the 0th location on its path, so this will be checked for index 1 and further. Every location will be checked in the for-loop starting at line 20. Line 21 will then filter out all locations in which cannot be stopped, as those are not suitable intercept locations. For all of these locations the agent steps, target steps, iteration of arrival of the target, and iteration of departure of the target are stored at lines 22, 23, 24, and 25, respectively. Both the number of iterations until the catch happens and the distance of the agent to that location are multiplied by the $penalty$ to discourage leaving the original path. This is done in lines 26 and 27.

If this is indeed a better location than the original i^{th} location on the path and the agent can get there earlier or at the time of the target leaving, store this location and its minimum_iterations and minimum_distance. After this is done for all of the locations on the map, it is checked whether the target can stop on the original i^{th} location and the agent can get there on time. If this is the case, it is checked whether the original location is at least as good as the best location found outside of the path, factoring in the penalty for going off of the original path. If this is the case, the algorithm will return the original

loc at line 39. If the other location is better, that location is added to the path in the correct location as stored in index_to_insert and that location is returned at line 42. If the agent cannot catch the target in any original or alternative location, the algorithm returns the last stoppable location on the original path at line 46. This last_stoppable_location is maintained at line 8.

4.2.7. Proof efficiency intercept location

In terms of the number of iterations, this first interceptable location will always be equal or lower than when the current target location is used, which is when no target path is used. The proof for this claim is as follows:

Theorem 1. *Given a target t with its path $path$, an agent a, and the map that they are in, the number of iterations it takes for agent a to catch the target t in the first intercept location is always equal or lower than when the agent chases the target by going after their current location.*

Proof. Proof by contradiction:
Assume agent a catches target t quicker by going after t's current location rather than going towards the first intercept location.
Denote the location that the agent catches the target in using the current location as l_c and the first intercept location as l_f.
Since l_f is the first intercept location, this is the first location where the number of agent steps is lower than or equal to the iteration in which the target leaves the location.
This means that there cannot be a location earlier on the path that the target can be caught in when it is there at those iterations.
When going after t's current location, no path is known and thus no alterations to the target's path are made.
With no earlier location on $path$ that target t can be caught in, and the agent going towards l_f in the shortest way possible, there is no way to catch the target faster than the agent going to l_f using the shortest path. Thus, agent a cannot catch target t quicker by going after t's current location.
This is a contradiction.

□

The reason that this also works with multiple agents and multiple targets is that the assignments are based on every combination of agents and targets.

Theorem 2. *Given a set of targets $T = \{t_0, ..., t_n\}$, a set of agents $A = \{a_0, ..., a_m\}$, and the first intercept locations l_{f_ij} for each pair of t_i and a_j, using other locations for the intercept location will not make the makespan lower.*

When not swapping assignments, no intercept location can get lower. The reason for this is that for every assignment, the previous theorem holds.
When swapping assignments, the following proof holds:

Proof. Proof by contradiction:
Assume using a further location on the target's path than l_{f_ij} for a target and agent results in a lower makespan when swapping assignments.
This means that for the assignment t_k - a_l with the highest makespan, an agent a_h can be paired with t_k to create a lower makespan, while a combination of the other agents and targets can be paired to have equal or lower makespan than the assignment t_k - a_h.
The way the makespan assignment works is that it removes every assignment that has the highest makespan until no assignment is possible anymore. This means that without assignment t_k - a_l either t_k takes longer to be caught than when using assignment t_k - a_l or a_l takes longer to catch another target than t_k.
Since using the first intercept location for a single agent-target pair uses the lowest makespan, as proven in the proof above, using locations later than the first intercept locations does not change that

Input : A path $path$ of the target
The agent $agent$ for which the first intercept location is checked
A function $stoppable$ that takes as input a location and outputs whether the location can be stopped in
The penalty $penalty$ for going off the original path
An algorithm $better_for_assignment$ that takes as input the number of iterations and distance for one location and the number of iterations and distance of another location and returns true if the first location is better for the current assignment strategy

Output: The first interceptable location

```
1  minimum_iterations ←max_value
2  minimum_distance ←max_value
3  new_location ←-1
4  index_to_insert ←-1
5  for i ←0 to path_size do
6      loc ←i^th location in path
7      if stoppable(loc) then
8          last_stoppable_location ←loc
9      end
10     agent_steps ←steps_to_location(agent_location, loc)
11     target_steps ←i
12     This is the iteration at which the target arrives at this location
13     target_arrives ←(target_steps + (MAX(target_steps + target_stop_counter - 1, 0) / stayput))
14     This is the iteration at which the target leaves this location
15     target_leaves ←(target_steps + (target_steps + target_stop_counter) / stayput) + 1
16     iterations_to_catch ←MAX(agent_steps, target_arrives)
17     distance_to_catch ←distance_to_location(agent_location, loc)
18     if i > 0 then
19         previous_loc ←(i − 1)^th location in path
20         for other_location ∈ map do
21             if stoppable(other_location) then
22                 new_agent_steps ←steps_to_location(agent_location, other_location)
23                 new_target_steps ←steps_to_location(previous_loc, other_location)
24                 new_target_arrives ←(new_target_steps + (MAX(new_target_steps +
                    target_stop_counter - 1, 0) / stayput))
25                 new_target_leaves ←(new_target_steps + (new_target_steps +
                    target_stop_counter) / stayput) + 1
26                 new_iterations_to_catch ←MAX(new_agent_steps, new_target_arrives) *
                    penalty
27                 new_distance_to_catch ←distance_to_location(agent_location, loc) * penalty
28                 if new_agent_steps ≤ new_target_leaves &
                    better_for_assignment(new_iterations_to_catch, new_distance_to_catch,
                    minimum_iterations, minimum_distance) then
29                     minimum_iterations ←new_iterations_to_catch
30                     minimum_distance ←new_distance_to_catch
31                     new_location ←other_location
32                     index_to_insert ←i
33                 end
34             end
35         end
36     end
37     if stoppable(loc) & agent_steps ≤ target_leaves then
38         if better_for_assignment(iterations_to_catch, distance_to_catch, minimum_iterations,
            minimum_distance) then
39             return loc
40         else
41             add_location_to_path(new_location, index_to_insert)
42             return new_location
43         end
44     end
45 end
46 return last_stoppable_location
```

Algorithm 3: The algorithm to find the first interceptable location on the target's path

either t_k takes longer to be caught than when using assignment t_k - a_l or a_l takes longer to catch another target than t_k.
Thus, the makespan is not lower when swapping assignments after using later intercept locations than the first intercept locations.
This is a contradiction. □

As explained in 4.2.5, the first intercept location algorithm is not necessarily optimized for the agent distance. It is possible for agents to stay closer to their starting location and still catch the target, but this means that the target has to travel further. Since the target is always slower than the agent, letting the target travel further for the agent's benefit would increase the number of iterations needed a lot quicker than when an agent needs to travel further. This could introduce a potentially significant increase in the number of iterations needed. This is why we have decided that the first intercept location is more appropriate.

4.3. Variants

We have decided to make different variants for the problem. The main variant makes both the target and the agent disappear whenever an agent catches a target. Whenever there is an equal number of agents and targets, every agent must catch exactly one target. This is called variation 0 in the figures.

Another variant we have implemented only makes the target disappear, while allowing the agent to immediately be assigned another target. This is a less realistic variant for many use cases. For example, for taxis picking up passengers, as the agent (taxi) cannot immediately look for another target (passenger). This also skews the results, as the assignment algorithms are not designed to have one agent catch multiple targets, but rather have each agent catch one target. This is called variation 1 in the figures. For this variation, line 20 of Algorithm 1 is omitted and the agent will not disappear anymore.

Both previously mentioned variants have two different types of maps used. This map type determines whether or not a target or agent can stop in all locations or only in some predetermined ones. In the latter case, the agents cannot catch targets in locations that are not these stop locations either. These maps with only specific stop locations are created to simulate a bit more realism, as in real life a lot of locations are often not safe to stop in. For instance, a passenger cannot stop in the middle of the highway to get picked up. When searching for a location to catch the target, these locations will therefore not be taken into consideration. If there are no locations in which the target can stop on the original path of the target, or another location close to that path is a lot more efficient, the target will change their path to go through that location instead.

All in all, this means that the following variants are available:

- Variation 0 without stops
- Variation 0 with stops
- Variation 1 without stops
- Variation 1 with stops

4.4. Scalability

To make sure that the storage does not get flooded with paths from each location to every other location, the paths should be stored intelligently. In our code, this is done by storing for each location to every other location what direction to go in next. After being in the new location in the direction that was just returned, the new direction will be checked to get one step closer to the end location. This is visualized in Figure 4.2. This is the route of the agent shown in 1.1. The 'X' shown when in location (4,4) means that it should not move, as the agent is already in the correct location. For comparison, the most straightforward way of storing this path would be to store the full list containing "(1,1), (1,2), (1,3), (1,4), (2,4), (3,4), (4,4)" in the cell with (1,1) as starting location and (4,4) as end location.

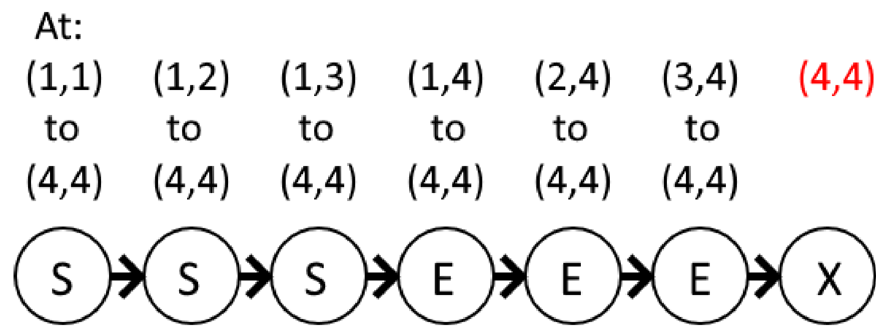


Figure 4.2: The path done by the agent in Figure 1.1 can be stored using one direction for each location passed. Instead of storing the full path from (1,1) to (4,4) in the matrix cell depicting the path from (1,1) and to (4,4), the agent will request the value of the matrix cell depicting from every new location to (4,4), returning the direction to go into next.

The time complexity of the Hungarian algorithm and the summation strategy is $O((a+t)^3)$ where a is the number of agents and t is the number of targets (Grinman, 2015, Edmonds and Karp, 1972). The Hopcroft-Karp algorithm has a time complexity of $O((a+t)^{2.5})$, and together with the binary search of $O(log(a+t))$ used to determine what the lowest maximum steps are such that there is still a viable assignment, the time complexity is $O((a+t)^{2.5}log(a+t))$ (Hopcroft and Karp, 1973). The mixed strategy is a sequence of the makespan strategy and the summation strategy, meaning that the time complexity is the same as the highest of the two. In this case, that is the summation strategy with $O((a+t)^3)$.

The memory is a problem for bigger maps. This is most likely due to the distances and number of steps between two locations being stored for each location to each other location. One way this is already alleviated is by storing only the side where the first location is more to the top left than the second. Unfortunately, even then a 2D array is needed of the number of locations by the number of locations. Thus, the space complexity of this problem is $O(n^2)$, where n is the number of locations. Another way to compress these paths is by using compression techniques, such as the Compressed Path Database as shown in Zhao et al., 2020 or even further compression with for instance wildcards and redundant symbols as described in Chiari et al., 2019. This will be reserved for future work.

5 Experiment

The overall goal of the experiment is to figure out what the impact is of the additional knowledge of the targets' paths, different assignment strategies, different maps, different number of agents (which always equals the number of initial targets), and all of the different variables.

When measuring the impact of the target path knowledge, the different assignment strategies, and all of the different variables, it will be compared to instances where all of the other influences stay the same, including the map, the agents' and targets' starting locations and the targets' original paths. For the number of agents and the different maps, all but the starting locations and paths will stay the same, as this is in both cases the changing factor.

For all combinations of variables and maps, 100 runs are done. We have created nine different maps with different features to best represent all maps. The maps used are the ones mentioned in Section 3.6 with an alternative map for the types random, room, and warehouse. With these maps, the algorithm is tested for how much distance the agents have traveled (see equation 2.1) and for how many iterations are needed for all targets to get caught (see equation 2.2). The variables are as follows:

- Map: empty, open, original (maze-like), random, random2, room, room2, warehouse, warehouse2
- Target path used: true/false
- The three assignment strategies: summation (DIS), makespan (MKS), and mixed (MIX)
- The initial number of targets and agents: 1, 2, 5, 10, 20, and 40
- The number of iterations between one iteration the targets cannot move (stayput): 1, 2, 4, 8, and 16
- Every number of iterations (gap) the assignments get recalculated: 1, 2, 4, 8, and 16
- Variation: Variation 0, Variation 1
- Stops: true (specific stopping locations)/false (no specific stopping locations)

With the nine different maps, two different target path used values, three different assignment strategies, six different numbers of targets and agents, five different stayput values, five different gap values, two different variations, and two different stops (latter two as stated in Section 4.3), and 100 runs per different combination, the total number of runs is $9 \times 2 \times 3 \times 6 \times 5 \times 5 \times 2 \times 2 \times 100 = 3{,}240{,}000$.

There will first be a summary of all the results in this chapter in Section 5.1. After that, the impact of the different variables will be shown within their own section, starting with the maps in Section 5.2. This is followed by the target path usage in Section 5.3 and strategy in Section 5.4. Gap and stayput are grouped together in Section 5.5 for their similarity in results. Lastly, the variations are explained with variation 1 in Section 5.6 and stops used in Section 5.7.

5.1. Summary experiment

The maps have quite a big impact on the performance. Knowing the targets' paths significantly lowers the number of iterations and agent distance needed to catch all targets similarly for all maps except for room-type maps that do not decrease as significantly as the other.
In general, knowing the targets' paths leads to about half the number of iterations needed and two-thirds of the agent distance.
Choosing the right strategy is also important, as the distance-based strategy is the worst of the three for the number of iterations, but the best for agent distance. On the flip side, the makespan-based strategy is the best in terms of the number of iterations but the worst for agent distance. The mixed strategy is tied lowest with the makespan-based strategy for the number of iterations and almost as low as the distance-based strategy for agent distance.

Higher gap values are slightly increasing the number of iterations and agent distance when not using the targets' paths. When the targets' paths are used, the different values of the gap and stayput have no impact.
Higher stayput values have a more significant increase in number of iterations and agent distance needed to catch all targets when the targets' paths are unknown. When these paths are known, there is next to no impact.
Not making agents disappear after catching a target, i.e. variation 1, causes the number of iterations and agent distance to decrease for each strategy. The makespan-based strategy takes the least advantage of freeing up an agent after catching a target.
Using specific stopping locations generally increases the number of iterations needed. The agent distance is impacted less when not using the targets' paths and is about the same when the targets' paths are used.

5.2. Maps

Using different maps has an impact on the performance. Figure 5.1 shows the average number of iterations for all number of agents for each of the maps. The entities can stop anywhere and the paths of the targets are not used. As can be seen, the open map consistently needs the least number of iterations to have all targets caught, followed by the empty map. This makes sense, as in those maps there are no obstacles any of the entities need to go around. The smaller space in the open map compared to in the empty map leads to fewer iterations needed. The map with consistently the highest number of iterations needed is the room2 map. In this map, there is often only one way to reach a room which rarely starts by going in the same direction as the room itself. This leads to a high number of iterations to even get close to a target.

Figure 5.2 has the same setup as Figure 5.1, but now with the agents using the paths of the targets. In addition to showing that using the paths here is a lot more efficient in getting the targets caught quickly, most of the patterns between the different map results are similar. The only maps that stand out in their pattern are the room maps that stay higher relative to the other maps. Similar to why the room2 map was higher in Figure 5.1, both of the room maps did not have as much impact from the knowledge of the targets' paths because getting to a room often needs a lot of iterations already. Most of the time, there is only one way to get to a certain room so it does not really matter whether the agents know the targets' paths because the agent needs to get to that room anyway.

5.3. Target path difference

The difference between the agent knowing the path of the target or not for the "original" maze-like map with no stops and using the mixed strategy can be seen in Figures 5.3 and 5.4. This map is chosen because the results in Section 5.2 have shown "original" to be near the average of all of the maps. As can be seen, the number of iterations is about twice as low when knowing the targets' paths. The agent distance is also lower, being about 1.5 times as effective.

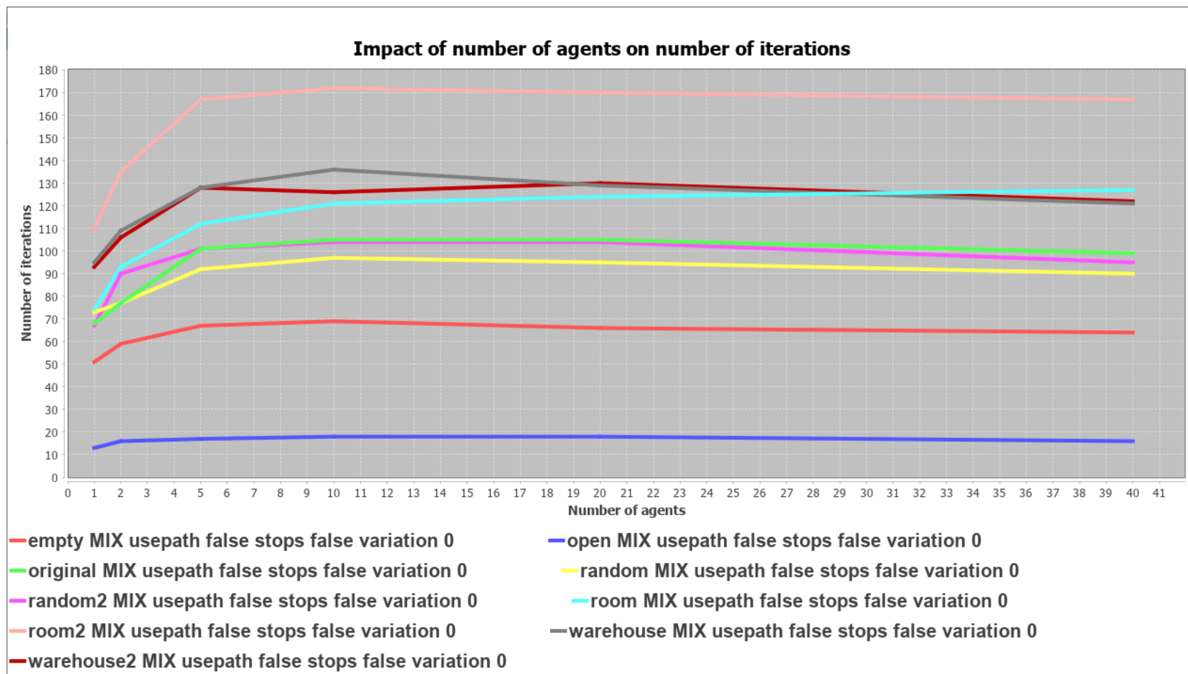


Figure 5.1: Impact of the type of map without stops and without using the targets' paths, with the mixed strategy.

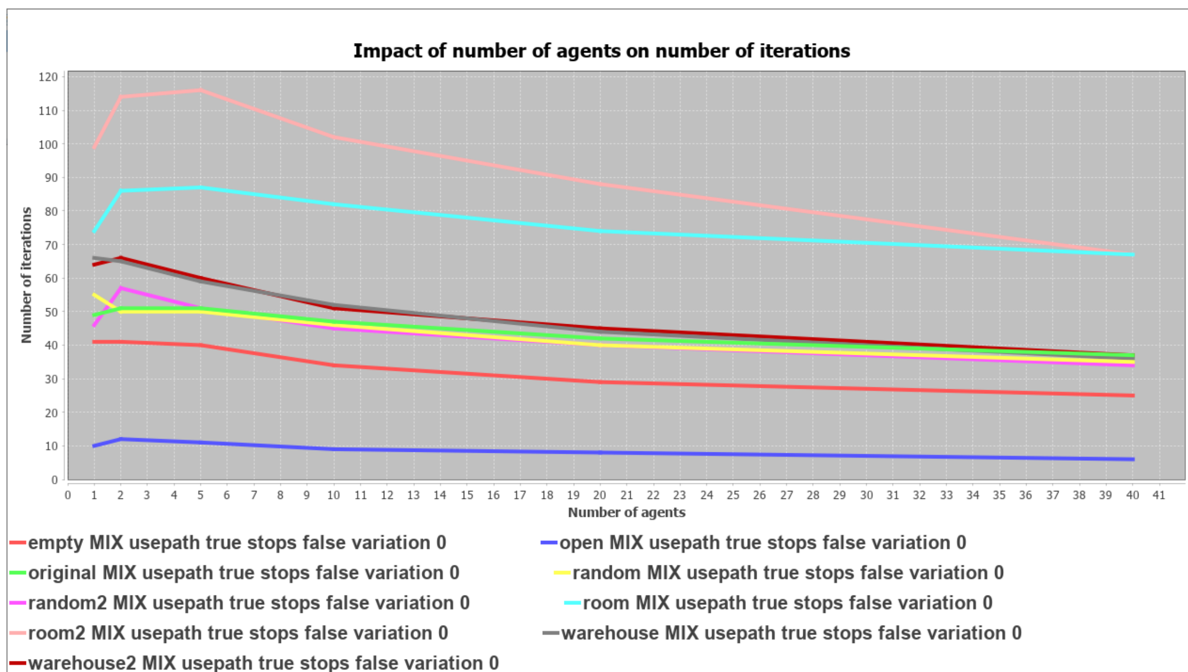


Figure 5.2: Impact of the type of map without stops, but with using the targets' paths, with the mixed strategy.

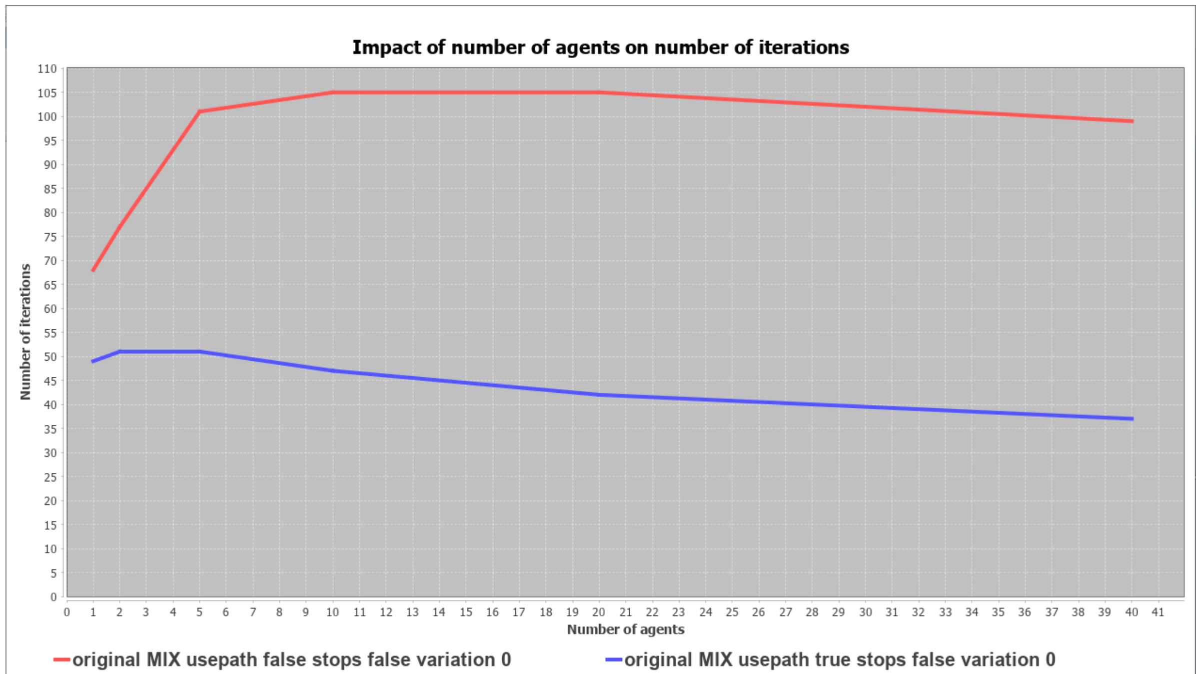


Figure 5.3: Difference between an agent NOT knowing the path of the target (in red) and an agent knowing the path of the target (in blue) on the "original" maze-like map without stops. Shown by measuring the number of iterations for each number of agents.

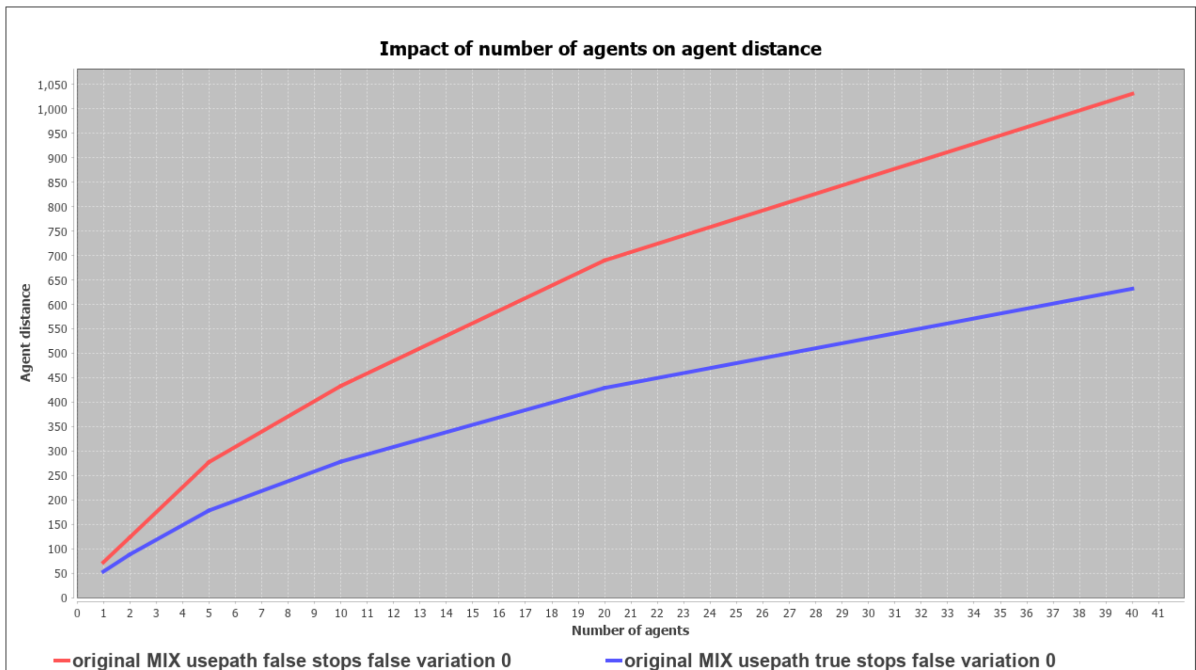


Figure 5.4: Difference between an agent NOT knowing the path of the target (in red) and an agent knowing the path of the target (in blue) on the "original" maze-like map without stops. Shown by measuring the agent distance for each number of agents.

5.4. Strategy difference

Figure 5.5 shows the difference in the resulting number of iterations for each of the three strategies on the “original” maze-like map without stops. Here can be seen that the distance-based strategy has the highest number of iterations, which is to be expected as this strategy is the only one of the three not optimizing for the lowest number of iterations. The number of iterations for the makespan-based and the mixed strategy is the same for all number of agents. This is because the mixed strategy first optimizes the makespan before optimizing the resulting possible assignments for the distance.
For this same configuration, the difference in agent distance can be found in Figure 5.6. Here can be seen that the makespan assignment strategy is the worst for the agent distance. This makes sense as this strategy is not optimizing for the agent distance, while the other two are. The mixed strategy is very close to the optimal distance-based strategy in terms of agent distance.

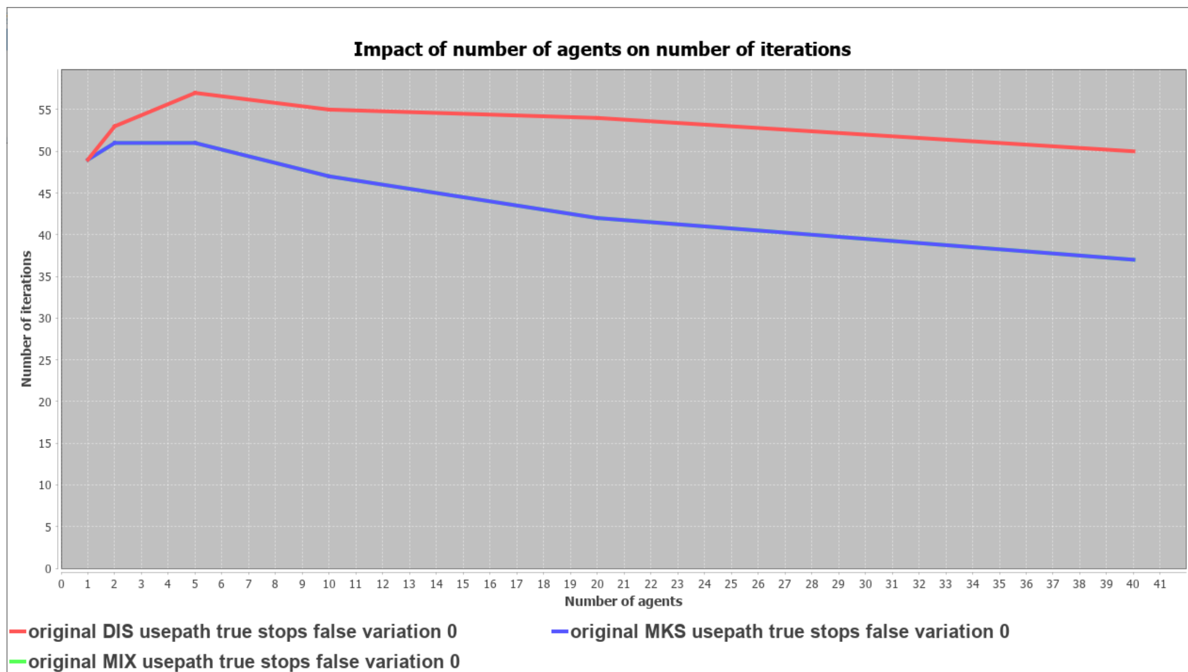


Figure 5.5: Difference in resulting number of iterations for each of the strategies on the “original” maze-like map without stops. The red line indicates the distance-based strategy, the blue line indicates the makespan-based strategy, and the green line, which is equal to the blue line, indicates the mixed strategy.

As mentioned in Chapter 4.2.1, the algorithm for determining the location to intercept the target in is optimized for the least number of steps for the target. The earliest location for which the agent can catch the target will be the intercept location, which makes sure that the target is caught as early as possible, but not necessarily with the lowest agent distance. Later locations on the path of the target may be closer to the agent than the first interceptable location, but it takes more iterations for the target to get to that location. Given that the target is slower than the agent, this difference in the number of iterations may be quite significant. If this is not the target that is caught last, this could be a free decrease in agent distance, but this is something for future work.

5.5. Gap and stayput

The results shown above are only shown in terms of the number of iterations or agent distance for every number of agents. However, we also mentioned that we would show results for the gap and the stayput.

The results for the different gaps on the “original” maze-like map without stops can be seen in Figures 5.7 and 5.8. Both the number of iterations and agent distance without using the target paths are very slowly increasing for higher gaps. This is to be expected, as higher gaps mean fewer recalcula-

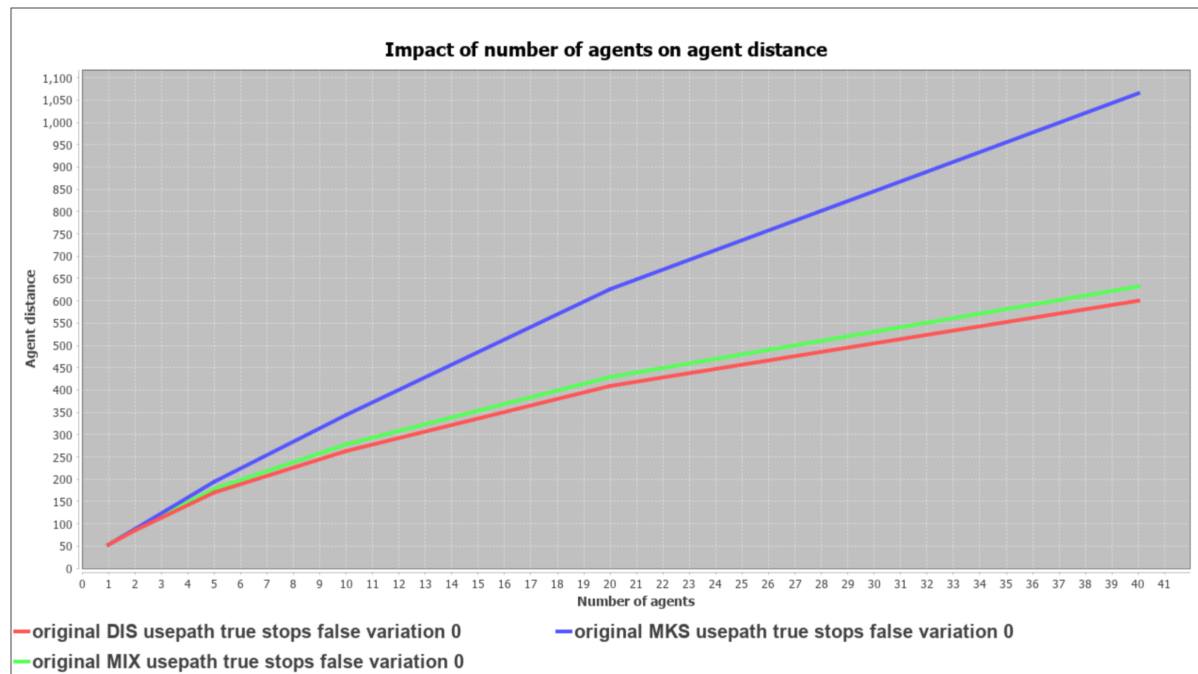


Figure 5.6: Difference in resulting agent distance for each of the strategies on the "original" maze-like map without stops. The red line indicates the distance-based strategy, the blue line indicates the makespan-based strategy, and the green line indicates the mixed strategy.

tions for the assignments. This means less often the most accurate information and thus less often the most optimal assignments, leading to a higher number of iterations and agent distance. When the target paths are used, all the agents know where they can catch any target. The assignment is based on these locations and therefore there is no need for reassignment, rendering the gaps useless. The results show this by not showing any difference in number of iterations or agent distance for different values of gap.

The results for the different stayput values on the "original" maze-like map without stops can be seen in Figures 5.9 and 5.10. Without the knowledge of the targets' paths, the number of iterations and agent distance needed for all targets to get caught ramp up with higher stayput values. In this case, agents are generally following the targets, which makes them rely on their higher speed to catch the targets. When the stayput value is high, the targets become quicker. It takes the agents therefore longer to catch up to the targets, explaining the higher number of iterations and agent distance for higher stayput values.
When the agents do know the path of the targets, they can calculate where the target will be caught based on the stayput value. Since the targets start anywhere and go anywhere, a faster target could both mean that the target approaches the agent faster or gets away from the agent faster. Regardless of that, the agent will calculate where they can intercept the target the earliest and go to that location. Therefore, the stayput will have next to no influence on the number of iterations or agent distance.

5.6. Variation 1

When agents do not disappear whenever they catch a target, as used in variation 1, the number of iterations and agent distance is a bit different than whenever they do disappear. This difference can be seen in Figures 5.11 and 5.12. For the iterations, all are lower than with variation 0. The distance-based strategy improves the most out of the change to variation 1. The reason for this is that the distance-based strategy often tries to catch each target as quickly as possible, which for this variation frees one of the agents up to catch another target. The makespan-based strategy, on the other hand, only tries to get the latest catch as early as possible, which means the closest targets will not necessarily be caught quickly. This variant, therefore, does not improve the results for this strategy as much. For the agent

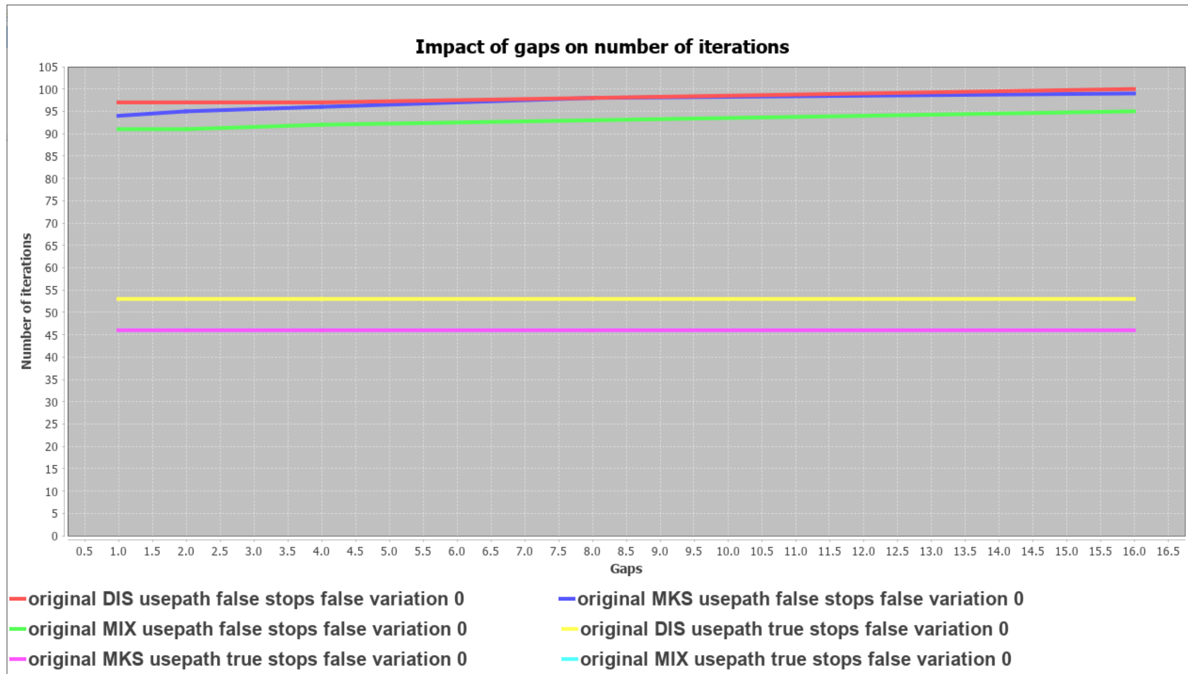


Figure 5.7: Results for different gaps on the "original" maze-like map without stops in terms of number of iterations. The red, blue, and green lines are the results for the distance-based, makespan-based, and mixed strategy without using the target paths, while the yellow, magenta, and cyan (directly under the magenta line) lines are the results for the distance-based, makespan-based, and mixed strategy with the target paths used.

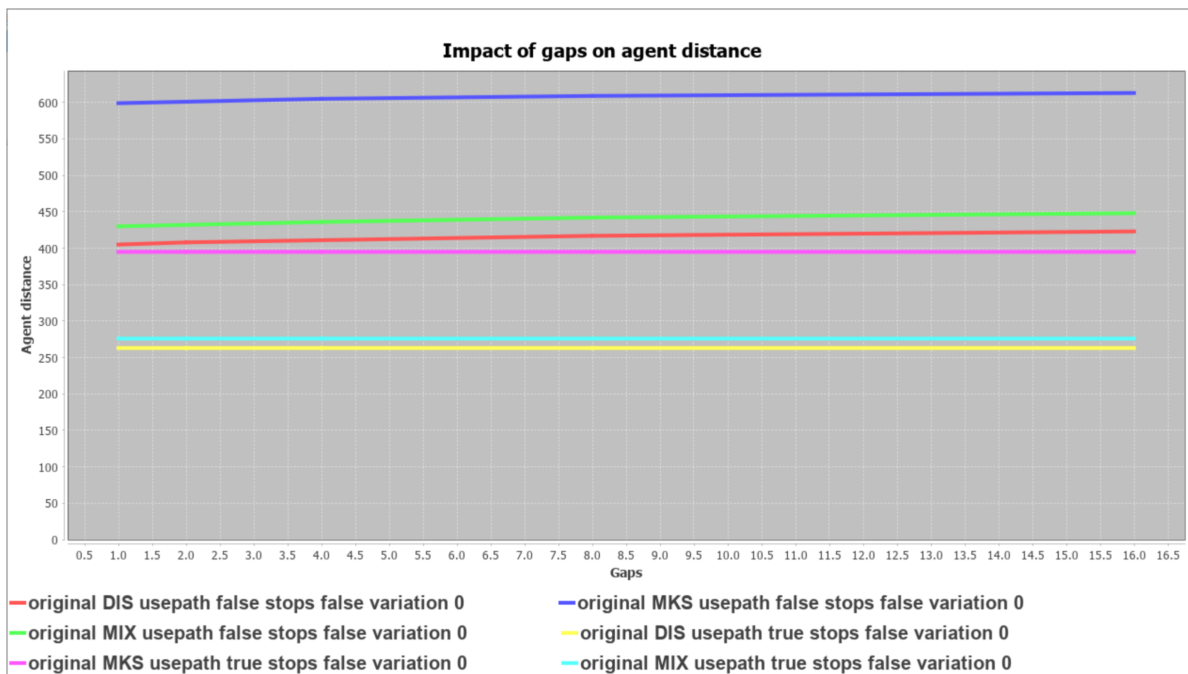


Figure 5.8: Results for different gaps on the "original" maze-like map without stops in terms of agent distance. The red, blue, and green lines are the results for the distance-based, makespan-based, and mixed strategy without using the target paths, while the yellow, magenta, and cyan lines are the results for the distance-based, makespan-based, and mixed strategy with the target paths used.

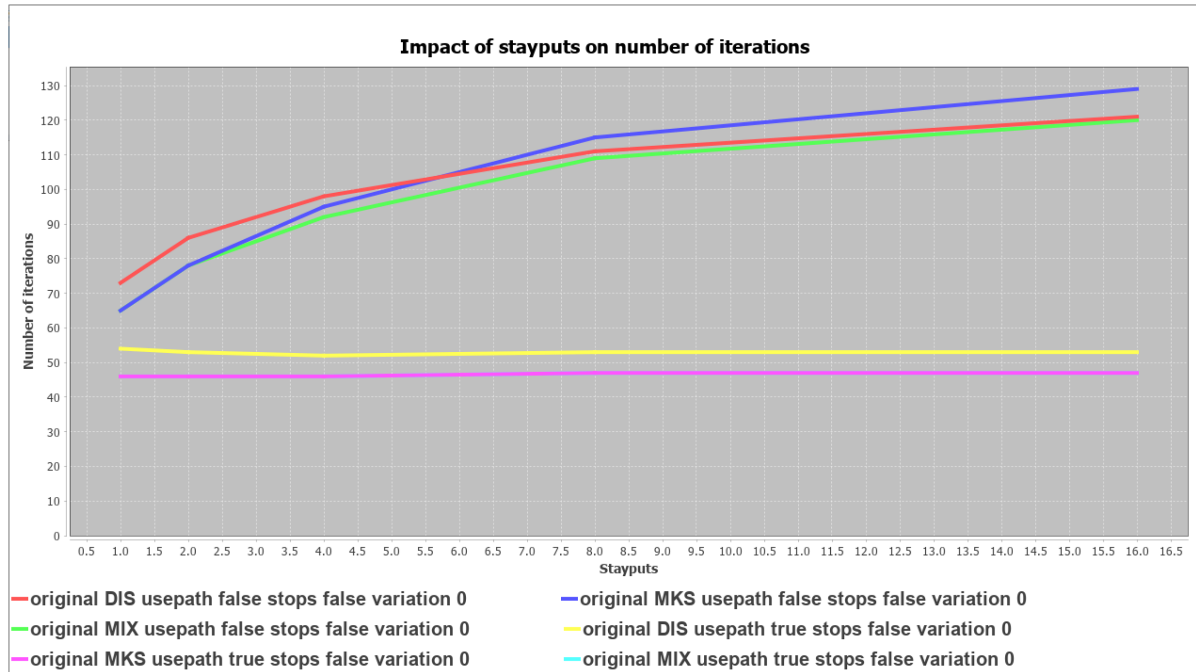


Figure 5.9: Results for different stayput values on the “original” maze-like map without stops in terms of number of iterations. The red, blue, and green lines are the results for the distance-based, makespan-based, and mixed strategy without using the target paths, while the yellow, magenta, and cyan (directly under the magenta line) lines are the results for the distance-based, makespan-based, and mixed strategy with the target paths used.

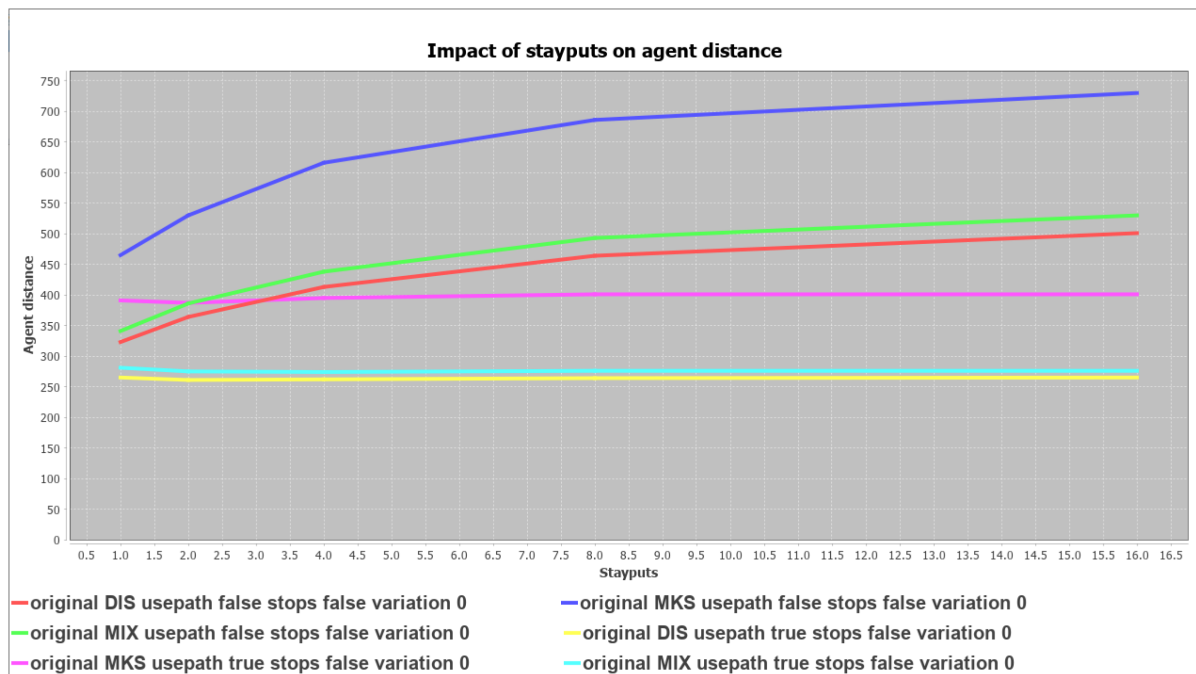


Figure 5.10: Results for different stayput values on the “original” maze-like map without stops in terms of agent distance. The red, blue, and green lines are the results for the distance-based, makespan-based, and mixed strategy without using the target paths, while the yellow, magenta, and cyan lines are the results for the distance-based, makespan-based, and mixed strategy with the target paths used.

distance, all strategies with variation 1 are lower than their counterpart with variation 0. However, the makespan-based strategy with variation 1 is still worse than the distance-based and mixed strategy with variation 0. This is most likely due to the makespan-based strategy not being optimized for agent distance, nor getting much value out of freeing up agents because the strategy does not necessarily catch closest targets quickly.

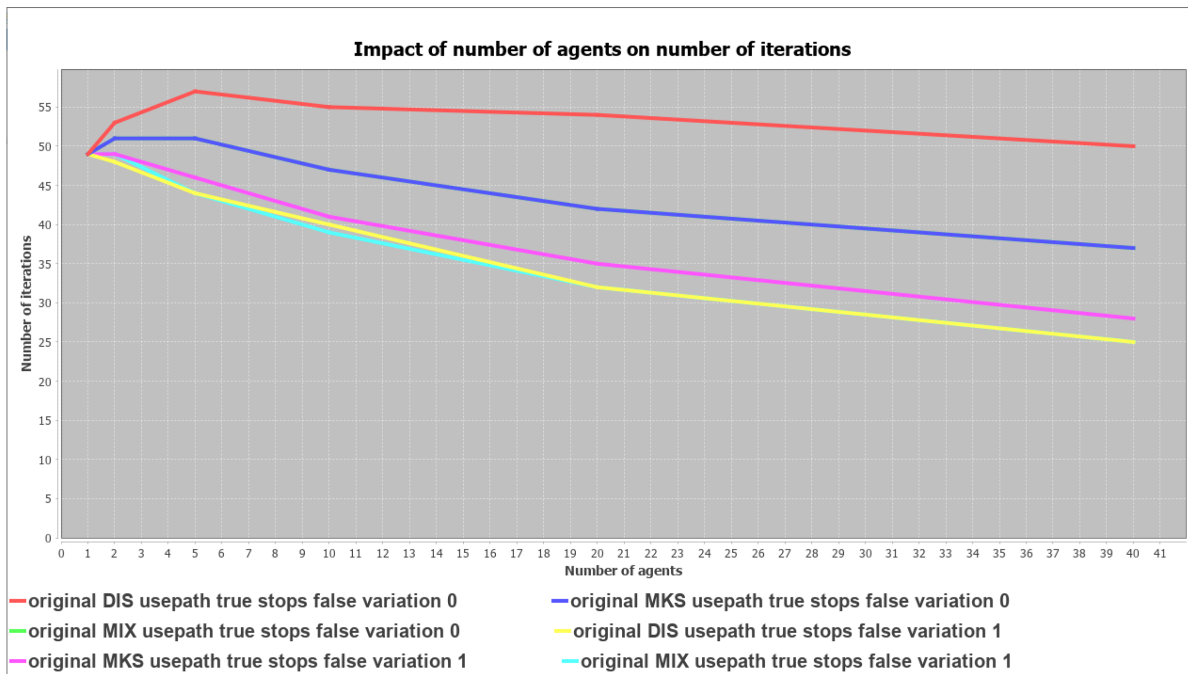


Figure 5.11: The difference in iterations for both variants on the "original" maze-like map without stops. The red, blue, and green (directly under the blue) line are the distance-based, makespan-based, and mixed strategy with variation 0 where the agents disappear whenever they catch a target. The yellow, magenta and cyan lines are the distance-based, makespan-based, and mixed strategy, respectively, for the variation where the agents do not disappear whenever they catch a target.

5.7. Stops

Figures 5.13 and 5.14 show the difference in using only specific stopping locations or allowing agents and targets to stop anywhere. This is done for all of the three assignment strategies DIS (summation), MKS (makespan), and MIX (mixed), using no paths of the targets, all on the "original" maze-like map. The figure shows that having no specific stopping location is generally marginally better, both in terms of the number of iterations and agent distance needed. The reason for this is that the agents are generally following the targets when no paths are used. Normally, they can instantly catch the target whenever they get to the same location as them. However, in this case, they may need to follow them for a few more steps to get to a location in which they can stop and catch them.

Figures 5.15 and 5.16 also show the difference in using only specific stopping locations or allowing to stop anywhere, for all three assignment strategies on the "original" maze-like map, but now when the targets' paths are known. Similar to when no paths were known, allowing to stop anywhere is generally better in terms of the number of iterations. The difference between having specific stops and allowing the agents and targets to stop anywhere is bigger when target paths are known. The reason for this is that the algorithm determining the first intercept location (see Section 4.2) also takes into account whether the location found can be stopped in.
For the agent distance, however, the results are about the same regardless of being allowed to stop anywhere or not. The reason for this is that a target could be going towards the agent, which would make specific stops and therefore a later stopping location advantageous for the agent distance, as the target will therefore stop closer to the agent's starting position. The target could also be moving away from the agent's starting location, making the later intercept location disadvantageous for the agent's

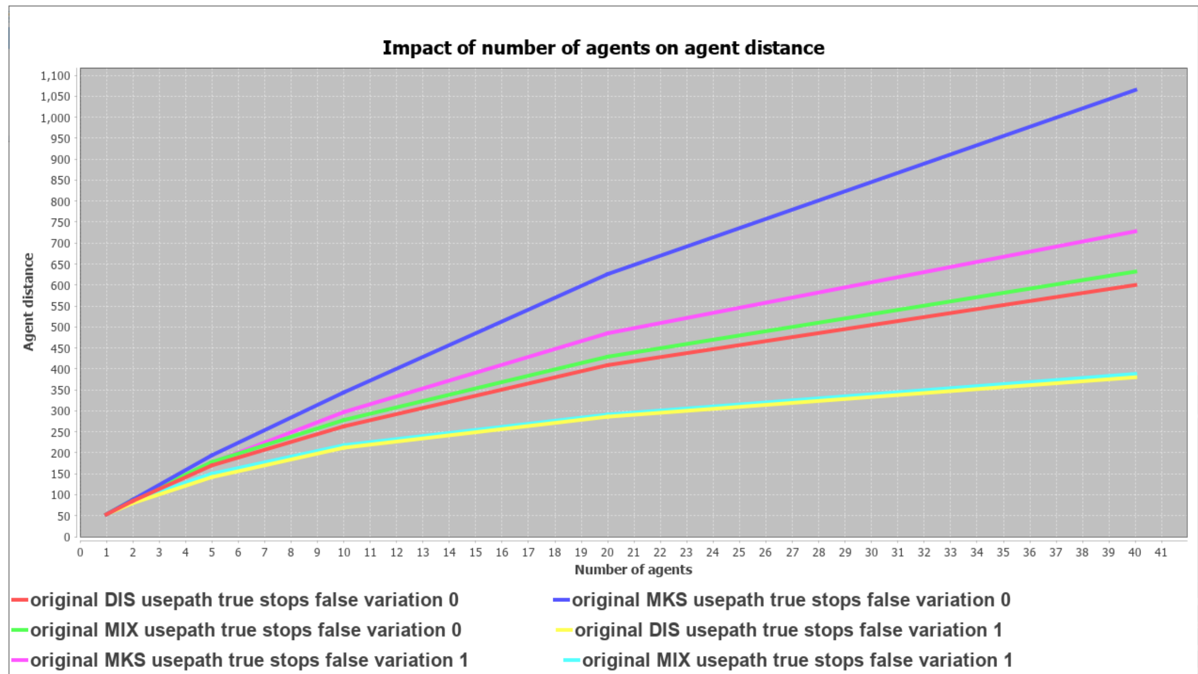


Figure 5.12: The difference in agent distance for both variants on the “original” maze-like map without stops. The red, blue and green line are variation 0 where the agents disappear whenever they catch a target. The yellow, magenta and cyan lines are the distance-based, makespan-based, and mixed strategy, respectively, for the variation where the agents do not disappear whenever they catch a target.

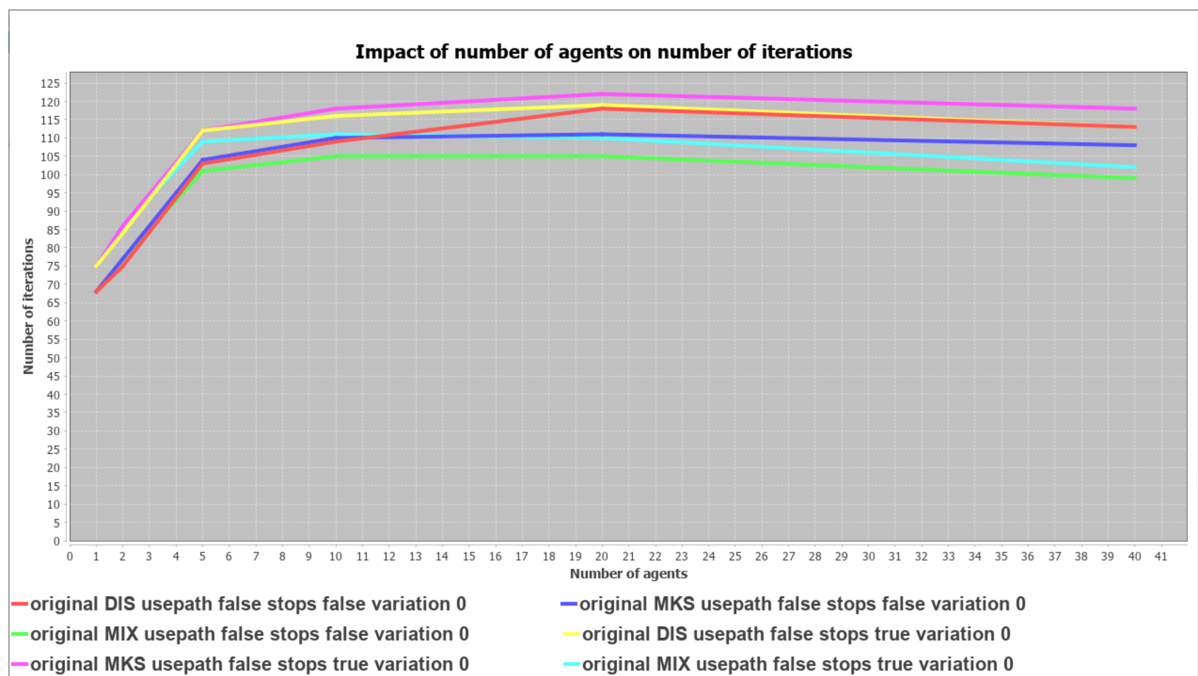


Figure 5.13: Impact in terms of number of iterations of the different strategies shown with both maps with specific stopping locations (stops) and maps where all locations can be stopped in (nostops). No target paths are used, measured on the “original” maze-like map. The red, blue and green line are the distance-based, makespan-based, and mixed strategy when no specific stops are used, while the yellow, magenta, and cyan lines are the distance-based, makespan-based, and mixed strategy, respectively, where the specific stopping locations are used.

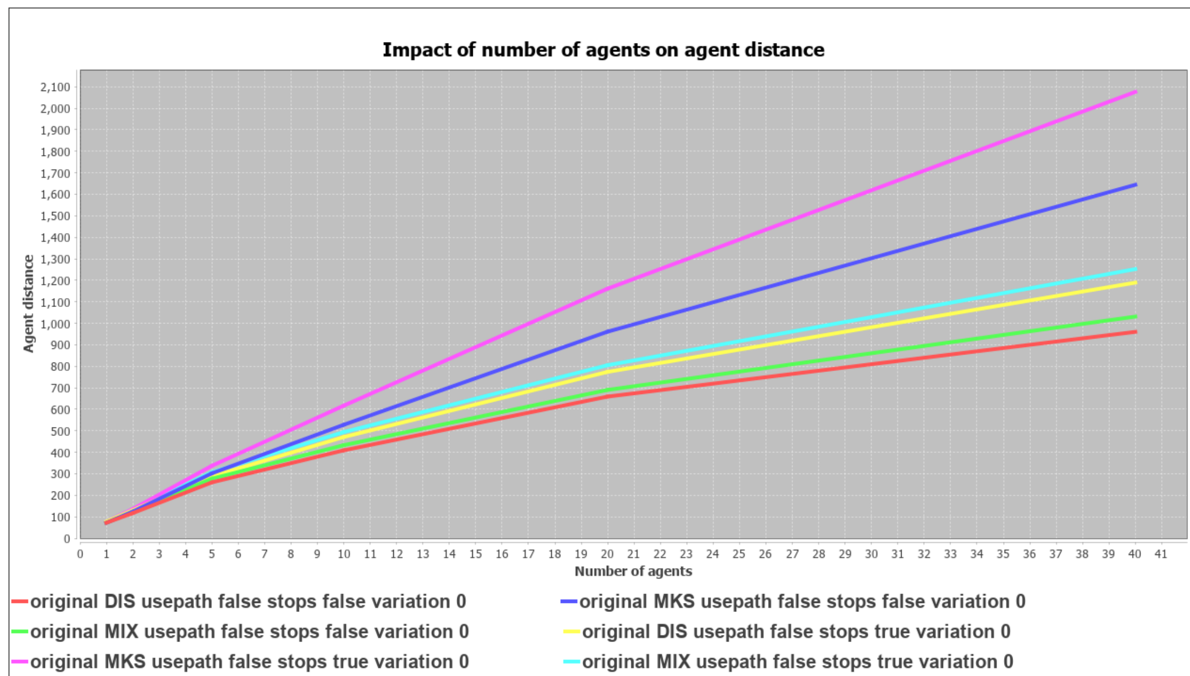


Figure 5.14: Impact in terms of agent distance of the different strategies shown with both maps with specific stopping locations (stops) and maps where all locations can be stopped in (nostops). No target paths are used and on the “original” maze-like map. The red, blue and green line are the distance-based, makespan-based, and mixed strategy when no specific stops are used, while the yellow, magenta, and cyan lines are the distance-based, makespan-based, and mixed strategy, respectively, where the specific stopping locations are used.

distance traveled.

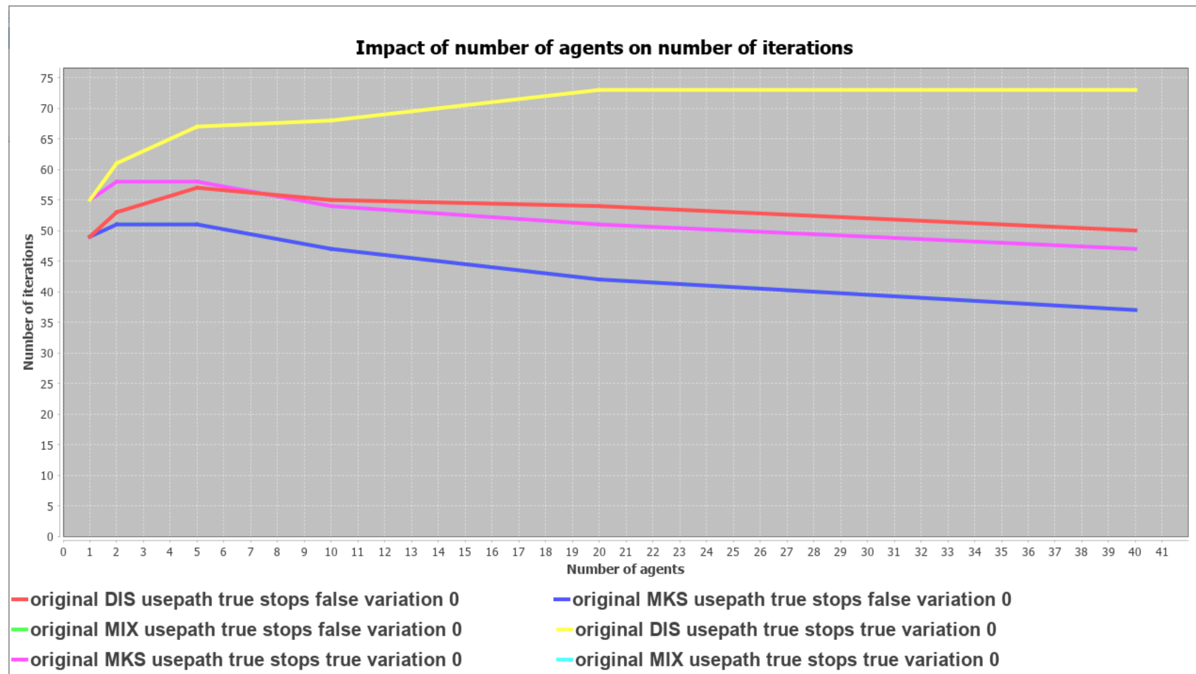


Figure 5.15: Impact in terms of number of iterations of the different strategies shown with both maps with specific stopping locations (stops) and maps where all locations can be stopped in (nostops). Target paths are used and on the “original” maze-like map. The red, blue and green (equal to the blue line) lines are the distance-based, makespan-based, and mixed strategy when no specific stops are used, while the yellow, magenta, and cyan (directly under the magenta line) lines are the distance-based, makespan-based, and mixed strategy, respectively, where the specific stopping locations are used.

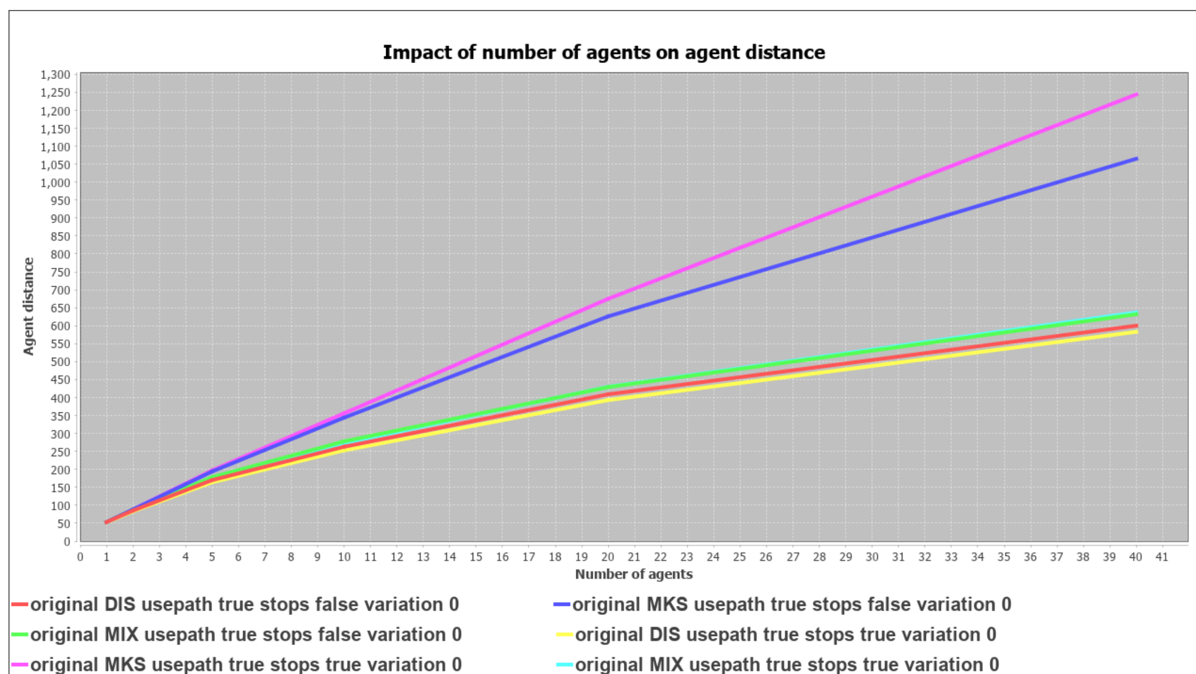


Figure 5.16: Impact in terms of agent distance of the different strategies shown with both maps with specific stopping locations (stops) and maps where all locations can be stopped in (nostops). Target paths are used and on the “original” maze-like map. The red, blue and green lines are the distance-based, makespan-based, and mixed strategy when no specific stops are used, while the yellow, magenta, and cyan lines are the distance-based, makespan-based, and mixed strategy, respectively, where the specific stopping locations are used.

6 Conclusion and Future work

We have introduced a way for agents to more efficiently catch cooperative moving targets. Thus, the research question ”Does having cooperative targets improve the efficiency of catching these targets compared to catching non-cooperative targets?” can be answered with ”Yes”. The way we have increased the efficiency is by using the targets’ paths. When targets can share their path, agents can calculate where they can intercept them and plan accordingly. For all types of maps, both the number of iterations and agent distance are reduced significantly. The number of iterations needed is about half of what is needed when no target paths are known. The agent distance is about two-thirds.
When knowing the targets’ paths, recalculation of which agent needs to catch which target is not necessary anymore, nor does the speed difference between the agents and targets matter much anymore. Variations in which agents can repeatedly catch different targets improve the performance a little. Variations with specific stopping locations increase the number of iterations needed slightly while staying relatively even in terms of agent distance when targets’ paths are known.
For future work, compression techniques can be used to be able to perform cooperative multi-agent multi-target catching on bigger maps.

Bibliography

Barer, M., Sharon, G., Stern, R., & Felner, A. (2014). Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. *Seventh Annual Symposium on Combinatorial Search*.

Barták, R., Zhou, N.-F., Stern, R., Boyarski, E., & Surynek, P. (2017). Modeling and solving the multi-agent pathfinding problem in picat. *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, 959–966.

Bernstein, D. S., Givan, R., Immerman, N., & Zilberstein, S. (2002). The complexity of decentralized control of markov decision processes. *Mathematics of operations research*, *27*(4), 819–840.

Busoniu, L., Babuska, R., & De Schutter, B. (2008). A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, *38*(2), 156–172.

Chiari, M., Zhao, S., Botea, A., Gerevini, A. E., Harabor, D., Saetti, A., Salvetti, M., & Stuckey, P. J. (2019). Cutting the size of compressed path databases with wildcards and redundant symbols. *Proceedings of the International Conference on Automated Planning and Scheduling*, *29*, 106–113.

Claes, D., Oliehoek, F., Baier, H., Tuyls, K., et al. (2017). Decentralised online planning for multi-robot warehouse commissioning. *AAMAS'17: PROCEEDINGS OF THE 16TH INTERNATIONAL CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS*, 492–500.

Claes, D., Robbel, P., Oliehoek, F., Tuyls, K., Hennes, D., & Van der Hoek, W. (2015). Effective approximations for multi-robot coordination in spatially distributed tasks. *Proceedings of the 14th international conference on autonomous agents and multiagent systems (AAMAS 2015)*, 881–890.

Dijkstra, E. W. et al. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, *1*(1), 269–271.

Edmonds, J., & Karp, R. M. (1972). Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM (JACM)*, *19*(2), 248–264.

Gange, G., Harabor, D., & Stuckey, P. J. (2019). Lazy cbs: Implicit conflict-based search using lazy clause generation. *Proceedings of the International Conference on Automated Planning and Scheduling*, *29*, 155–162.

Grinman, A. (2015). The hungarian algorithm for weighted bipartite graphs. *Massachusetts Institute of Technology*.

Guéant, O., Lasry, J.-M., & Lions, P.-L. (2011). Mean field games and applications. *Paris-princeton lectures on mathematical finance 2010* (pp. 205–266). Springer.

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, *4*(2), 100–107.

Hollinger, G., Singh, S., Djugash, J., & Kehagias, A. (2009). Efficient multi-robot search for a moving target. *The International Journal of Robotics Research*, *28*(2), 201–219.

Hopcroft, J. E., & Karp, R. M. (1973). An n5/2 algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, *2*(4), 225–231.

Ishida, T., & Korf, R. E. (1991). Moving target search. *IJCAI*, *91*, 204–210.

Koenig, S., & Likhachev, M. (2006). A new principle for incremental heuristic search: Theoretical results. *ICAPS*, 402–405.

Koenig, S., Likhachev, M., & Sun, X. (2007). Speeding up moving-target search. *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, 1–8.

Kuhn, H. W. (1955). The hungarian method for the assignment problem. *Naval research logistics quarterly*, *2*(1-2), 83–97.

Oliehoek, F. A. (2012). Decentralized pomdps. *Reinforcement learning* (pp. 471–503). Springer.

Pearl, J., & Kim, J. H. (1982). Studies in semi-admissible heuristics. *IEEE transactions on pattern analysis and machine intelligence*, (4), 392–399.

Pohl, I. (1970). Heuristic search viewed as path finding in a graph. *Artificial intelligence*, *1*(3-4), 193–204.

Sharon, G., Stern, R., Felner, A., & Sturtevant, N. R. (2015). Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, *219*, 40–66.

Sharon, G., Stern, R., Goldenberg, M., & Felner, A. (2013). The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence*, *195*, 470–495.

Standley, T. (2010). Finding optimal solutions to cooperative pathfinding problems. *Proceedings of the AAAI Conference on Artificial Intelligence*, *24*(1).

Stern, R. (2019). Multi-agent path finding–an overview. *Artificial Intelligence*, 96–115.

Stern, R., Sturtevant, N., Felner, A., Koenig, S., Ma, H., Walker, T., Li, J., Atzmon, D., Cohen, L., Kumar, T., et al. (2019). Multi-agent pathfinding: Definitions, variants, and benchmarks. *arXiv preprint arXiv:1906.08291*.

Sun, X., Koenig, S., & Yeoh, W. (2008). Generalized adaptive a. *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 1*, 469–476.

Surynek, P. (2010). An optimization variant of multi-robot path planning is intractable. *Proceedings of the AAAI Conference on Artificial Intelligence*, *24*(1).

Wagner, G., & Choset, H. (2015). Subdimensional expansion for multirobot path planning. *Artificial Intelligence*, *219*, 1–24.

Walker, T. T., Sturtevant, N. R., & Felner, A. (2018). Extended increasing cost tree search for non-unit cost domains. *IJCAI*, 534–540.

Xie, F., Botea, A., & Kishimoto, A. (2017). A scalable approach to chasing multiple moving targets with multiple agents. *IJCAI*, 4470–4476.

Zhang, K., Yang, Z., & Başar, T. (2021). Multi-agent reinforcement learning: A selective overview of theories and algorithms. *Handbook of Reinforcement Learning and Control*, 321–384.

Zhao, S., Chiari, M., Botea, A., Gerevini, A. E., Harabor, D., Saetti, A., & Stuckey, P. J. (2020). Bounded suboptimal path planning with compressed path databases. *Proceedings of the International Conference on Automated Planning and Scheduling*, *30*, 333–341.