CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Delay-tolerant Multi-Agent Path Finding

Delay-tolerant methods for solving multi-agent path finding problems in autonomous systems

Master's thesis in Computer science and engineering

JOAKIM GYLLENSKEPP
KEVIN NORDENHÖG

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

Delay-tolerant Multi-Agent Path Finding

Delay-tolerant methods for solving multi-agent path finding problems in autonomous systems

JOAKIM GYLLENSKEPP
KEVIN NORDENHÖG

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Delay-tolerant Multi-Agent Path Finding
Delay-tolerant methods for solving multi-agent path finding problems in autonomous systems
JOAKIM GYLLENSKEPP
KEVIN NORDENHÖG

Supervisor: Elad M. Schiller, Computer Science and Engineering
Examiner: Peter Ljunglöf, Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2019

Delay-tolerant Multi-Agent Path Finding
Delay-tolerant methods for solving multi-agent path finding problems in autonomous systems
JOAKIM GYLLENSKEPP
KEVIN NORDENHÖG
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Multi-agent path finding algorithms find collision-free paths that agents should follow according to a schedule. The schedules found by multi-agent path finding algorithms are strict, meaning that when a robot in an autonomous system is delayed and fall behind schedule, the schedule is invalidated and the movement of all agents are paused until a new schedule is found. Computing these schedules is inherently slow due to the hardness of multi-agent path finding problems. There are methods, such as MAPF-POST, which find the delay tolerance of a given schedule and, in some cases, allow agents to be delayed without doing expensive replanning.

We propose a modification to the multi-agent path finding algorithm Conflict-based search, which makes the schedule found by the algorithm tolerant to delays. The delay tolerance that the schedule should have can be selected by the user. Our results show that the delay tolerant conflict-based search can successfully reduce the number of recalculations in a system with delays, at the cost of a higher run-time.

We also propose a method called refined component stalling, which avoids recalculations by purposefully delaying a subset of the agents in order to retain a valid schedule. Our experiments show that this method successfully reduces the number of agents affected by delays and the need for recalculations, at the cost of giving the agents longer paths.

Our contributions reduces the time spent on recalculations significantly, which means that the time it takes for the agents to follow a schedule may be faster compared to existing multi-agent path finding algorithms, which sometimes spends a lot of time on recalculations. This is especially clear when agents move fast, since the longer path length of our contributions are less noticeable.

Keywords: Computer science, multi-agent path finding, path finding, simulation, conflict-based search, autonomous systems.

Acknowledgements

We want to thank our supervisor Elad Schiller for his guidance, support, and great commitment throughout the project. We also want to thank Peter Lindh and the rest of the employees at Squeed for their help and friendliness. Lastly, we would like to thank Peter Ljunglöf for taking his time to examine our work.

Joakim Gyllenskepp and Kevin Nordenhög, Gothenburg, June 2019

Contents

List of Figures

List of Tables

1

Introduction

1.1 Background

Automated storage systems where robots operate on a grid can be found in automated warehousing, cross-docking, car parking systems, and container handling [1]. One system used for automated warehousing is AutoStore [2], where robots move in a 4-directional manner along the x and y directions of a grid, which is located on top of an array of storage units. This allows the robots to access any cell in the grid, where they can place or grab a storage bin from the storage unit.

The autonomous robots that move around in these systems should move fast for efficiency, while simultaneously avoiding collisions with each other. Multi-agent path finding (MAPF) can be used to find a schedule with collision-free paths that the robots can follow. These paths need to be strictly followed in order to avoid collisions, and if a robot falls behind the schedule it is invalidated and all robots are stopped while a new schedule is found by replanning. Replanning should be avoided since MAPF problems are NP-hard and sometimes very time-consuming [3].

In order to reduce the amount of replanning needed, Hönig *et al.* [3] proposed MAPF-POST which can post-process the schedule output by a MAPF algorithm, and create a plan execution schedule that, in some cases, allows agents to be delayed. The impact of this method is limited by how much room for errors there is in the existing schedule, e.g., if there is a lot of room for error in the schedule, larger delays are allowed without causing replanning.

In this thesis, we propose methods that can reduce the impact of delays and replanning in systems where autonomous robots move according to a MAPF algorithm. Furthermore, we have designed and implemented a simulator that can be used to evaluate MAPF algorithms in a system where delays are present. The simulator uses grid-based maps similar to AutoStore, where agents may move in a 4-directional manner to adjacent positions in the grid every time unit.

1.2 Thesis statement

Consider an automated system where autonomous robots move around according to a MAPF schedule. If the speed of the robots is slow enough, we assume that they are guaranteed to keep up with the schedule without falling behind. If we increase the speed of the robots, we assume that, at a certain point, some robots are bound to fall behind the schedule and get delayed.

Assume that we have a system where robots may be delayed. How do MAPF algorithms perform in these systems? Can we improve the MAPF algorithms to work better?

1.3 Our contribution

In this project, we have proposed a model for delay-prone multi-agent path finding systems, as well as solutions that advance state-of-the-art multi-agent path finding. We have implemented the model in an evaluation environment that simulates agents that can be delayed. This was done in order to evaluate our novel algorithms Delay-Tolerant Conflict Based Search (DT-CBS) and Refined Component Stalling, that were made in order to handle agent delays.

DT-CBS is an algorithm that is based on the state-of-the-art MAPF algorithm CBS. DT-CBS finds a schedule that tolerates a predefined amount of agent delay. DT-CBS successfully reduces the number of recalculations compared to CBS, at the cost of an increased run-time of the algorithm and slightly longer paths. This resulted in a faster plan execution in cases where the system was executed fast and CBS encountered many recalculations, which happens when the probability for delays are high, the paths are long, or when there are many agents in the system.

Refined Component Stalling handles delays that happen during plan execution, by purposefully delaying a subset of the agents such that the delay will be tolerated. Refined Component Stalling reduces the need for recalculations, but it increases the path-lengths of the agents during plan execution. This means that Refined Component Stalling can be beneficial in instances where the MAPF problem is hard to solve, i.e., big maps with many agents that otherwise requires many recalculations.

2

Background

Here we present existing well-known algorithms used for both single-agent and multi-agent path finding. Single-agent path finding refers to finding an appropriate path for a single vehicle, whereas multi-agent path finding finds an appropriate set of non-colliding paths for a number of vehicles such that each vehicle can travel from its origin to its destination.

2.1 Definitions

Definition 1. Multi-agent path finding problem. Multi-agent path finding problems are specified by a graph $G = (\mathcal{V}, \mathcal{E})$ and a set of agents $a_1, .., a_k$. Each agent a_j has a start position $s_j \in \mathcal{V}$ and destination $g_j \in \mathcal{V}$. The goal of a MAPF algorithm is to find a valid conflict-free solution to a MAPF problem.

Definition 2. Solution to a MAPF problem. The solution to a MAPF problem consists of a set of non-colliding paths $\mathcal{P}_1, .., \mathcal{P}_k$, one for each agent. Each path $\mathcal{P} = \{s_j^0, .., g_j^T\}$ contains the schedule for agent j from the start position s_j^0 at time 0 to the destination g_j^T at time T.

Definition 3. Optimal MAPF solution. A MAPF algorithm is optimal if the solution found is guaranteed to be the solution with the lowest possible cost for the given problem, otherwise it is suboptimal.

Definition 4. Bounded suboptimal solver. Bounded suboptimal solvers let the user specify a bound w, letting the solver only return solutions with cost less or equal to $(1 + w) \cdot C$, where C is the cost of the optimal solution.

Definition 5. Unbounded suboptimal solver. Unbounded suboptimal solvers offer no guarantee on the quality of the solution. The cost of the solution returned can be very expensive even if cheap solutions exist.

Definition 6. Time. Time is defined as a finite discrete integer where T is equal to the length of the longest path in the solution.

Definition 7. Conflict. In a MAPF algorithm a conflict is when two agents occupy the same position at the same time. A conflict between agent a_i and a_j at position v at time t is denoted by the tuple (a_i, a_j, v, t).

Definition 8. ϵ delay-tolerance. Consider that there is a solution $\mathcal{SOL}$ to a MAPF problem. When $\mathcal{SOL}$ is executed, one or more agents are delayed by at most ϵ. We say that $\mathcal{SOL}$ is ϵ delay-tolerant if these delays do not cause conflicts between agents, regardless of when they happen.

Definition 9. SIC and makespan. The solution of a MAPF problem is evaluated by the sum of individual costs (SIC) or makespan. The SIC is the total time costs over all agents, whereas makespan is the maximum time cost of a single agent. An optimal solution C refers to a solution with the lowest possible cost for a given problem, i.e., the optimal solution can change depending on if it is optimized after SIC or makespan.

2.2 Single-agent path finding algorithms

Single-agent path finding is used to find the shortest paths between two vertices in a graph. MAPF algorithms often utilize single-agent path finding algorithms when finding paths for multiple agents.

The most commonly used single-agent algorithm by MAPF algorithms is A* [4]. A* works in a similar manner as the well-known algorithm Dijkstra's algorithm, that finds the shortest path between two vertices by assigning values to all the vertices in the graph based on the distance from the start vertex [5]. Dijkstra's selects the closest unvisited vertex to the start vertex and assign values (distances) to its neighbors, overwriting any values if the new value is lower and saving the vertex as the *previous vertex*, and when done the vertex is marked as *visited*. This process repeats until the goal vertex is marked *visited* and the shortest path can be found by backtracking the vertices saved as the *previous vertex*.

Instead of simply searching the graph in cost order, A* uses a heuristic in order to prioritize vertices closer to the goal vertex and assigns values to those vertices first. This allows A* to often visit fewer vertices and save time while still finding the optimal path. The A* algorithm can be seen in Algorithm 1.

2.3 Multi-agent path finding algorithms

Given a graph and a set of agents with unique start and goal nodes, a multi-agent path finding algorithm finds a path for each agent such that the agents will not collide with each other. MAPF can be approached using different methods, there are search-based solvers, reduction-based solvers, rule-based solvers, hybrid solvers, and also AI-based solvers [3, 6]. This report only covers search-based MAPF solvers.

In order to give an overview of existing search-based solvers, this section describes multiple well-known search-based MAPF algorithms, brought up by Felner *et al.* [7].

Algorithm 1 A* search algorithm

Variables:
$closedSet = \{\}$ ▷ Contains the explored vertices
$openSet = \{\}$ ▷ The frontier, containing the vertices to explore
$cameFrom$ = empty map ▷ Maps vertices to its lowest cost neighbor
$cost = \emptyset$ ▷ Cost from start to this vertex
$priority = \emptyset$ ▷ Heuristic score
Add *start* to *openSet*
$cost[start] = 0$
$priority[start]$ = HeuristicCost(*start*, *goal*)
▷ Calculates the Manhattan distance
while *openSet* not empty **do**
    current = vertex in *openSet* with the lowest priority value
    if *current* = *goal* **then**
        return (*cameFrom*, *current*) ▷ Used for reconstructing the path
    remove *current* from *openSet*
    add *current* to *closedSet*
    for each *neighbor* of *current* **do**
        if *neighbor* in *closedSet* **then**
            continue ▷ ignore already evaluated neighbors
        $neighborcost = cost[current]$+ distance(*current*, *neighbor*)
        ▷ tentative cost for the neighbor
        if *neighbor* not in *openSet* **then**
            add *neighbor* to *openSet*
        else if $neighborcost \geq cost[neighbor]$ **then**
            continue
        $cameFrom[neighbor] = current$
        $cost[neighbor] = neighborgscore$
        $priority[neighbor] = cost[neighbor]$+ HeuristicCost(*neighbor*, *goal*)

The most important algorithm to consider in order to understand our contribution is conflict-based search presented in Section 2.3.2.

2.3.1 Cooperative A*

Cooperative A* (CA*) is a suboptimal unbounded MAPF algorithm that plans the path of all agents individually in succession based on some predefined order [8]. Each path is calculated using A* and is written down in a reservation table so that the next agent in line cannot occupy the same position at the same time. Because CA* calculates the paths of the agents in succession, a solution from a finished agent may prevent any solution from another agent. For example, if the goal of a finished agent is in a corridor other agents can be blocked from going through that corridor, making CA* an incomplete search algorithm.

CA* performs a series of k single agent searches, where k is the number of agents, meaning that run-time of CA* is linear in the number of agents. This is very fast compared to most optimal MAPF algorithms whose run-time is exponential in the number of agents, but at the cost of being suboptimal and incomplete.

Algorithm 2 Cooperative A* [8]

1: **Variables:**
2: $constraints = \emptyset$ ▷ Positions occupied by other agents at specific time units
3: $schedule = \emptyset$ ▷ Contains paths for agents that finished pathfinding
4: **for each** agent k **do**
5: $schedule[k] = \text{A*}(k, constraints)$
6: ▷ Single-agent A* which considers *constraints*
7: **for each** (*position*, *time*) **in** $schedule[k]$ **do**
8: $constraints$ += (*position*, *time*) ▷ Add new schedule to *constraints*

2.3.2 Conflict-based search

Conflict-based search (CBS) as introduced by Sharon *et al.* [6] aims to solve the MAPF problem optimally. The run-time of CBS is exponential in the number of conflicts, as opposed to other MAPF algorithm which are often exponential in the number of agents [9]. This can be beneficial in some environments but also unfavorable in some. Sharon *et al.* demonstrates that CBS outperforms many MAPF algorithms in environments with bottlenecks but suffers in open space environments [6].

CBS is divided into high level and low level. The high-level search uses a *constraint tree* in order to store the constraints for all the agents. A constraint is a specified position that an agent is not allowed to occupy at a specified time. Each node in the constraint tree stores constraints, a schedule containing the paths of all the agents that satisfies the constraints, and the SIC of the schedule. Each child inherits

Algorithm 3 Conflict-based search, high-level [6]

1: **Variables:**
2: $Root.constraints = \emptyset$ ▷ Root node has no constraints
3: $Root.solution$ = find shortest paths for each agent separately using low-level
4: $Root.cost$ = SIC($Root.solution$) ▷ Cost is Sum of Individual Costs (SIC)
5: $OPEN = \emptyset$ ▷ $OPEN$ is a priority queue containing nodes to expand
6: insert root to $OPEN$
7: **while** $OPEN$ not empty **do**
8: $P \leftarrow$ best node from $OPEN$ ▷ Best node has lowest cost (SIC)
9: Validate $P.solution$ until conflict ▷ Check for conflicts in solution
10: **if** no conflict **then**
11: **return** $P.solution$ ▷ P is the goal node
12: $C \leftarrow$ first conflict (a_i, a_j, v, t) in P
13: **for each** conflicting agent a_c in C **do**
14: $A \leftarrow$ new node
15: $A.constraints \leftarrow P.constraints + (a_c, v, t)$ ▷ Old constraints + new
16: $A.solution \leftarrow P.solution$
17: Find new $A.solution$ fulfilling new constraints with low-level
18: $A.cost$ = SIC($A.solution$)
19: insert A to $OPEN$

the constraints of the parent and one new constraint, starting with the root node without any constraints. The low-level calculates the shortest paths for the agents of a node which fulfills its constraints by running single-agent path finding for each agent independently, and saves the solution and SIC in the node. The high-level search decides what node to expand, which is the node with the lowest total SIC. The root node will be expanded first as it is the only node.

When expanding a node a validation process starts, where all paths are simulated. If all agents reach their goals without any conflict, node N is considered the goal node and the node's solution is returned. If there is a conflict during the validation, the node is considered a non-goal node and is split into two new children each adding a new constraint for one of the conflicting agents. If the conflict $C = (a_i, a_j, v, t)$ is found during the validation, one of the nodes would give agent a_i the constraint (a_j, v, t) and the other node would give agent a_j the constraint (a_i, v, t). If the number of conflicting agents are more than two at one time during the validation, the node is split into k children (k being the number of conflicting agents). The high-level CBS can be seen in Algorithm 3.

2.3.3 Greedy-CBS

Greedy-CBS (GCBS) is an unbounded suboptimal version of CBS where both the high- and low-level are relaxed and more flexible [10]. For the high-level, this means that instead of expanding the nodes with the lowest cost, GCBS expands the solu-

tions closest to a goal node. Therefore GCBS has a better performance than CBS, i.e., it expands fewer nodes and finds a solution faster. Barer et al. [10] tested five different heuristics in order to determine how close a solution is to the goal:

- h_1: number of conflicts
- h_2: number of conflicting agents
- h_3: number of pairs with at least one conflict
- h_4: vertex cover (a combination of h_2 and h_3, where a graph is defined with the conflicting agents as the nodes and the pairs as the edges)
- h_5: alternating heuristic (alternate between the different heuristics in a round robin fashion)

They found that the different heuristics have their own advantages and works great in some instances. h_5 provides the best performance, h_4 provides solutions with the lowest costs and h_3 is the most robust across different environments.

In order to relax the low-level a suboptimal low-level path finding algorithm can be used. However, it is important to not use an algorithm that provides too long paths since this would add to the number of conflicts, further increasing the high-level's load.

2.3.4 Bounded-CBS

Bounded-CBS (BCBS) is a bounded suboptimal version of CBS that modifies the high level of CBS by utilizing focal search in order to apply a cost bound [10]. Focal search keeps two lists, one for OPEN nodes (regular OPEN nodes from A*) and one for FOCAL nodes. The list of FOCAL nodes keeps track of the OPEN nodes that still satisfy the cost-bound and are worth expanding. Focal search uses two functions f_1 and f_2, f_1 compares the value to the bound and determines which nodes should be in FOCAL, and f_2 is used to choose which FOCAL node to expand.

2.3.5 Increasing cost tree search

Increasing cost tree search (ICTS) is an optimal MAPF algorithm that solves the problem by using an increasing cost tree (ICT) [11]. Sharon *et al.* demonstrates that, compared to other MAPF algorithms, ICTS performs well in environments with many open areas and worse in dense environments [11]. The performance of ICTS scales according to the difference in cost between the optimal multi-agent solution and the sum of all optimal single-agent solutions, which corresponds to the depth of the ICT.

The ICT is a tree where each node contains costs $[C_1, .., C_k]$ associated with each

agent k. At the root, the sum of the costs is equal to sum of the length of the optimal single-agent paths for each agent. Each node has a branching factor equal to the number of agents, where each child increases the cost of one agent by one. The high-level search does a breadth-first search on the ICT where for each node the low-level search checks if it is the goal node. Breadth-first search ensures that the first solution that is found is optimal since nodes at the same level of the ICT has the same total cost. The low-level search does a goal test for each node in the ICT. The goal test searches for a combination of non-conflicting single-agent paths where the cost for each agent is equal to the cost associated with the node. If there is a valid solution it is returned, otherwise the algorithm continues with the next node given by the high-level search.

2.4 Multi-agent path finding optimizations

Here we present some existing techniques that improve the performance of multiple MAPF algorithms.

2.4.1 Independence detection

Independence detection is a technique used to increase the speed of complete MAPF algorithms [9]. It divides agents into independent groups such that there is an optimal solution for each group and that there are no conflicts between groups. Independence detection essentially divides the MAPF problem into several smaller MAPF problems when possible. The run-time of solving most MAPF problems are exponential in the number of agents and dividing them into smaller groups can therefore significantly improve the overall performance.

The algorithm starts by assigning each agent to its own group and plans the paths for each agent independently. Then it simulates the execution of the paths until a conflict occurs, where it merges the two conflicting groups and plans the paths for the new group using a MAPF algorithm. This is repeated until a final conflict-free solution is found.

The algorithm for simple independence detection is described in Algorithm 4. A more refined version which is called full independence detection uses multiple optimizations and can speed up the run-time of complete search algorithms exponentially.

2.4.2 MAPF-POST

Hönig *et al.* [3] presents the algorithm MAPF-POST which post-processes a MAPF schedule into a plan execution schedule that can be executed on real robots. MAPF-

Algorithm 4 Simple Independence detection as explained by Standley (2012) [12]

```
1: assign each agent to a singleton travel group
2: plan a path for each travel group with A*
3: iterate through all paths until a conflict occurs
4: while conflicts occur do
5:     merge two conflicting travel groups into a single travel group
6:     use MAPF to generate paths for the new travel group
7:     iterate through all paths until a conflict occurs
8: solution = paths of all travel groups combined
9: return solution
```

POST provides three properties that are desirable in real systems. Firstly it considers kinematic constraints, such as velocity limits. Secondly, it guarantees a safety distance between agents. Thirdly it utilizes slack to avoid replanning due to imperfect plan execution, i.e., it may avoid replanning when an agent is delayed from the MAPF plan.

In this project, we have taken inspiration from the third property of MAPF POST. We will therefore only elaborate on how MAPF-POST utilizes slack to avoid replanning, and we will not explain the other properties which are not very relevant for our work.

Consider the case where two agents are scheduled to pass the same location v at different times. If the first agent passes v at time t, and the second agent passes v at time $t + \epsilon$, the slack that exists between the agents at location v is ϵ. Now consider that the agents may be delayed during plan execution. As long as the first agent is delayed by less than ϵ, it is guaranteed that the two agents will not collide at v, in which case the delay can be tolerated and replanning can be avoided. By looking at every position where a pair of agents pass and finding the slack ϵ between them, it is possible to generate a plan execution schedule that sometimes allows agents to be delayed while avoiding replanning. We say that a schedule is ϵ delay-tolerant when the minimum slack for any position is ϵ.

3

System

Here we present the system developed and used in this project. Firstly, we describe the architecture that is used for running path finding algorithms and executing their schedules in a system where delays are present. Secondly, we present the simulation model and the system that we use for the evaluation of MAPF algorithms.

3.1 Architecture

The architecture contains two layers, a global planner and a local planner. The global planner executes path finding on a static map using normal MAPF algorithms and finds a preliminary plan execution schedule that the agents should follow. During plan execution, the local planner handles deviations that occur in a dynamic system and strives to adapt the schedule such that it will not be invalidated by the deviation.

3.1.1 Global planner

The global planner will run a MAPF algorithm which generates a path from the starting point to the destination for each agent on a static map. The global planner is the first thing that is executed when launching the system and finds the initial schedule intended to be followed by all the agents. For example, the MAPF algorithms presented in Chapter 2.3 can be used as global planners.

3.1.2 Local planner

After the initial schedule has been computed by the global planner, a phase called plan execution starts, where agents follow the paths in the initial schedule. During plan execution, the system is dynamic and deviations may occur. The local planner handles these deviations, which would normally prevent the agents from following their given paths, for example when an object blocks the planned path of an agent or when an agent is delayed.

The local planner is executed every time a deviation occurs. Depending on what local planner is used, it might modify the existing schedule to accommodate for the deviation, provide a new schedule or decide to keep the old schedule if the deviation can be tolerated by the system.

Here we present an example of a local planner. Consider a set of agents following a plan execution schedule. Suddenly a deviation occurs, and one agent is delayed. The local planner detects the delay, but it finds that the delay is small enough not to cause any collision. The local planner then allows the agents to continue following the plan execution schedule since the delay was not big enough to cause any collision. On the other hand, if the delay would be too high the local planner would pause all the agents while computing a new schedule.

3.2 Evaluation environment

Here we present the models that are used for evaluation of MAPF algorithms. The step execution model describes the maps that we are using and how agents may move around in the maps. The agent deviation model covers our definition of deviations and how they appear in our system. Finally, we briefly present how we have implemented the simulation model in the simulator which we use for evaluation.

3.2.1 Simulation model

The architecture needs to be implemented in a non-deterministic simulator in order to evaluate the performance of the local planner. Due to the computation costs of simulators such as V-Rep [13] and Gazebo [14] that considers kinematic constraints and vehicular dynamics, we will develop an elegant non-deterministic grid-based simulator to experiment with the selected algorithms. The idea is to have a model that can be used to evaluate MAPF algorithms in a simple system to get an overview of their performance, the model is not meant to investigate how the algorithms work in real systems. Instead of kinematic constraints and vehicular dynamics, which are the cause of delays in real systems, delays are modeled through random motion failure where the agents simply do not move for one time unit.

Figure 3.1 briefly describes the simulator. To start the simulation, the paths for all agents are computed offline by the global planner. The schedule is sent to the agent deviation model, which applies deviations such that the next system state will differentiate from the expected one, according to some binomial distribution. The agent deviation model outputs the changes to be made from the current system state to the next system state. These changes are applied in the step execution stage which outputs the new system state. The deviation detection checks whether the new system state deviates from the intended state, and if there is a deviation it is handled by the local planner which will find new paths when necessary.

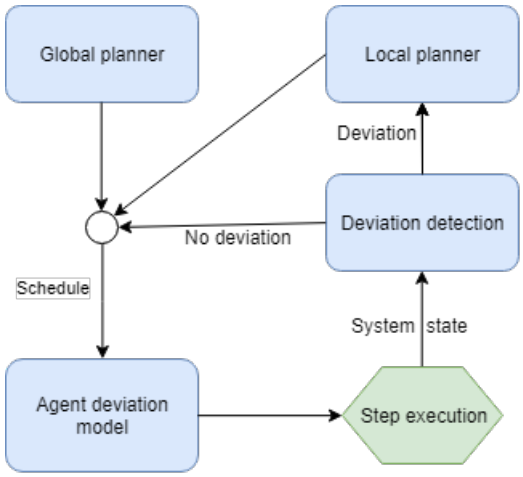


Figure 3.1: Description of simulator

Step execution model

The model uses discrete time steps, where each step in time corresponds to one action for each agent on the grid. Agents may at each time step wait at their current position or move in a 4-directional manner to adjacent positions on the grid, as long as the position is not an obstacle and the action does not result in a collision, as seen in Figure 3.2. To clarify, the 4 directions that an agent may move in is west, north, east, and south, i.e., agents may not move diagonally. We define a collision as two agents that at the end of the step occupy the same node in the grid, or when two agents swap positions during two parallel steps. That is, when two agents at one point in time occupies two adjacent positions, they may not simultaneously move to the other agent's position such that their positions swap during one time step.

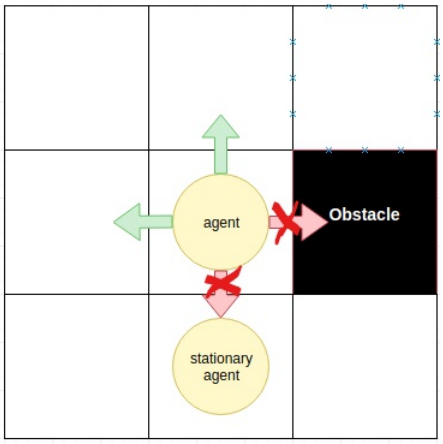


Figure 3.2: A 3x3 map with two agents and one obstacle. The non-stationary agent may move to non-occupied positions, as shown by the arrows.

Agent deviation model

To the end of representing a dynamic system behavior, our simulation model considers non-deterministic step execution. Specifically, our model considers agents that are randomly delayed as well as the appearance of an obstacle, such as a container We model these phenomena via a random motion failure, i.e., the agent simply does not change location, and locations where obstacles may randomly appear at any point in time to where the agents are not allowed to move.

These random motion failures are based on a binomial distribution, where for every action each agent may be delayed according to a user-specified probability p. As each delay is seen as an independent occurrence, the binomial distribution gives the expected delay that the agents will have after taking m steps, and if m is the makespan this gives the expected delay of a given schedule. This distribution allows the user to select the probabilities of random motion failure, which means that the user can choose the delay according to the system they are interested in.

These random failures is our way to model much more complex phenomena and still keep the computational costs affordable, and significantly lower than the ones of V-Rep and Gazebo [13, 14].

The term deviating agents refer to agents that are affected by these random motion failures in a way that prevents them from following the planned schedule.

Extensions to the model

The model described in this section is a simple one that can be extended. Possible such extensions are agents that may occupy more than one location, non-uniform cost functions for moving between positions, and agents with unequal speed. It is possible to extend the distribution that is used for random motion failures if a more realistic distribution is required.

3.2.2 Simulation system

The simulation system is implemented using Python and the method used for the visual presentation of the simulation system is Matplotlib [15], which is a large 2D plotting library in Python used to draw the map and the agents on the screen.

The map is represented as a 2-dimensional grid. Each cell in the grid is either empty, an obstacle, or a dynamic obstacle. A dynamic obstacle is when obstacles may randomly appear according to the agent deviation model. It is possible to draw a map and use it for simulation, by specifying the grid size and the positions of the static and dynamic obstacles in a configuration file. The number of agents and their start and goal positions are also specified in the configuration file.

4

Algorithms

In this section, we present our contributions that aim to improve the performance of MAPF algorithms in systems where agents may be delayed.

4.1 Delay-tolerant CBS

An optimal MAPF algorithm will always arrive at the solution with the lowest possible cost and therefore provide the best performance in a static system, which does not have any delays. But in a dynamic system where delays are present, an optimal solution might be sensitive to delays, which would force the system to recalculate the paths as often as delays appear. If delays are common in the system a schedule that is not cost-optimal but more tolerant to delays can be more beneficial and reduce the need for recalculations during runtime.

Here we present a method that finds an ϵ delay-tolerant solution, i.e., the solution tolerates that agents are delayed by up to ϵ time units. This algorithm was inspired by MAPF-POST introduced by Hönig *et al.* [3], as described in Section 2.4.2, which can be used to find the ϵ delay-tolerance of a given solution. In our algorithm, the delay bound ϵ can be selected by the user. A high ϵ tolerates higher delays but may result in a more expensive lower quality solution.

The method works by adding a tail of size ϵ to each agent, where agents are not allowed to step on the tails of other agents. This allows for a delay of ϵ without causing any collisions. We present this method for Cooperative A*, which can be found in Algorithm 5. Because Cooperative A* is suboptimal and incomplete, Algorithm 5 will not provide the best result with regards to the quality of the schedule. However, it is a fast and simple algorithm that demonstrates the method.

One of our contributions is a modified algorithm for CBS, called Delay-Tolerant CBS (DT-CBS), that makes the solution found by CBS delay-tolerant. The implementation of the algorithm can be found in Algorithm 6, which shows that by extending CBS with the same method as Algorithm 5, by adding a tail of length ϵ to the agents, an ϵ delay-tolerant solution can be found. When two agents collide in normal CBS, a constraint is added at the position of collision at time t. The way algorithm 6 works is that it, in case of a collision with an agent or its tail, adds

Algorithm 5 Tail algorithm for maximizing window size for Cooperative A*, the differences from Cooperative A* are seen in the boxed lines.

1: **Variables:**
2: $constraints = \emptyset$ ▷ Positions occupied by other agents at specific times
3: $schedule = \emptyset$ ▷ Contains paths for agents that finished pathfinding
4: $\boxed{tail = \epsilon}$ ▷ The solution tolerates a delay of at most ϵ
5: **for each** agent k **do**
6: $schedule[k] = \text{A*}(k, constraints)$
7: ▷ Single-agent A* which considers *constraints*
8: **for each** ($position$, $time$) **in** $schedule[k]$ **do**
9: **for each** $\boxed{i = -tail;\ i < tail;\ i++}$ **do**
10: $\boxed{constraints \mathrel{+}= (position, time + i)}$
11: ▷ Add new schedule to *constraints*

multiple constraints at the position of collision at time $t = [t-\epsilon, .., t+\epsilon]$. By adding constraints from time $t - \epsilon$ to $t + \epsilon$, the algorithm ensures that another agent or its tails won't occupy the position at time t or during the time the current agent's tail occupy the position. The constraint from time t to $t+\epsilon$ ensures that the conflicting agent won't collide with the tail of the current agent. The constraints from time $t-\epsilon$ to t is used to make sure that the agent which the constraints are set for cannot occupy the position at time t, since it would otherwise be possible for that agent to arrive at an earlier point in time which could cause its tail to remain at time t.

Consider that agent a_i on time t_i collides with the tail of agent a_j at position v, where a_j was at time t_j. Due to this conflict one node is generated for each agent. On the first node a set of constraints are added that says that agent a_i may not visit position v from time $t_j - \epsilon$ to $t_j + \epsilon$. Similarly on the second node, the constraints says that agent a_j may not visit position v from time $t_i - \epsilon$ to $t_i + \epsilon$. The solution for each node is calculated and put into $OPEN$.

4.2 Component stalling

Here we introduce a method that has the goal of stopping only a subset of all agents in the event of a deviation. We decide which subset to stop depending on the component of the deviating agent.

Definition 10. Component. We define a component as a set of agents whose paths intersects directly or indirectly. Meaning that if one agent's path intersects with two other agents, whose paths do not overlap directly, all three agents would be in the same component.

Definition 10 entails that the paths from agents within a component is separated from the paths of agents from other components. Algorithm 7 demonstrates the

Algorithm 6 Find an ϵ delay-tolerant solution using CBS, the differences from CBS are seen in the boxed lines.

Variables:
$\boxed{tail = \text{the wanted delay tolerance of the solution}}$
$Root.constraints = \emptyset$ ▷ Root node has no constraints
$Root.solution =$ find paths for each agent using low-level
$Root.cost = \text{SIC}(Root.solution)$ ▷ Cost is Sum of Individual Costs (SIC)
$OPEN = \emptyset$ ▷ $OPEN$ is a priority queue containing nodes to expand
insert root to $OPEN$
while $OPEN$ not empty **do**
    $P \leftarrow$ best node from $OPEN$ ▷ Best node has lowest cost (SIC)
    $\boxed{\text{Validate } P.solution \text{ until conflict with another agent or its tail}}$
    if no conflict **then**
        return $P.solution$ ▷ P is the goal node
    $\boxed{C \leftarrow \text{first conflict } (a_i, a_j, v, t_i, t_j) \text{ in } P}$ ▷ $t_i = t_j$ if collision is not with tail
    for each conflicting agent a_c in C **do**
        $A \leftarrow$ new node
        for each $\boxed{x \text{ from } -tail \text{ to } tail}$ **do**
            $\boxed{A.constraints \leftarrow P.constraints + (a_c, v, t_j + x)}$
                ▷ a_c should not visit v during $t_j \pm tail$
        $A.solution \leftarrow P.solution$
        Invoke low-level to update $A.solution$
        $A.cost = \text{SIC}(A.solution)$
        insert A to $OPEN$

construction of such components given a complete MAPF solution. When an agent in a component is deviating from its schedule, we refer to its component as the affected component.

Algorithm 7 Creating components

calculate paths using a MAPF-algorithm
assign each agent to its own component
iterate through all paths until agents from separate components overlap
while component paths overlap **do**
    merge overlapping components
    iterate through all paths until agents from separate components overlap
return components

Now consider the case when one agent is delayed by δ time units from its schedule. We propose a method called component stalling, presented in Algorithm 8, as a local planner that manages this delay by stopping the execution of the agents in the affected component and delaying all non-deviating agents by δ until the deviant catches up. Consequently, all the non-deviating agents in the affected component are delayed by the same amount of time as the deviant. This would make all agents in

the component equally delayed, and they can maintain their relation to each other and follow the original schedule even after the delay has occurred. The affected component has a solution independent from any other components, and for that reason it is not necessary to stop agents from other components. This allows the agents outside of the affected component to keep running since the components are calculated offline, letting some agents reach their destination faster and possibly save time.

Algorithm 8 Delay component

1: **Variables:**
2: δ = amount of deviation (in time steps)
3: *solution* = old paths from the current time and positions
4: stops the execution of affected component
5: delay all agents except the deviant in *solution* with δ time steps
6: **return** *solution*

Algorithm 8 has a computational cost that is constant in the number of deviations, since it simply delays agents, i.e., the algorithm runs fast and agents do not need to stop and wait for an expensive MAPF algorithm. Because agents are delayed and their paths do not change, the components will remain the same after the algorithm has finished running. Given a delay of δ, the makespan of the solution will at most increase by δ, when the agent with the longest path is in the affected component. Similarly, the sum of individual cost will always increase by $\delta * k$, where k is the number of agents in the affected components.

In order to reduce the cost when handling a delay using component stalling, it is possible to minimize the component sizes with a algorithm we call Refined component stalling presented in Algorithm 9. Components can be minimized by bounding the amount of accepted stalling. To do this agents would only be placed in the same component if the overlaps of their paths are within the bounded time of each other. This would allow the local planner to stall fewer agents possibly resulting in better performance. However, since there is a bound on the amount of stalling, this approach only works for delays less than the bound within each component. When the bound is exceeded a recalculation is needed, as the paths otherwise can affect agents outside of the affected component. Using this method you can bound the stalling by the expected amount of delay of a schedule, ensuring that as long as the number of delays is within expectations the components are not bigger than necessary.

The component size can be reduced further by also accounting for from what time step a delay occurs when calculating the components. Consider an intersection where two or more agents paths overlap, if the paths are considered from any time before the intersection the agents will be considered to be in the same component. However, if the paths are considered from any point in time after the intersection, the agents would not be in the same component unless they overlap again later in the schedule. Considering this principle, it is possible to reduce the component size by calculating the components for all time steps before the deviation occurs. At

Algorithm 9 Creating components using a time bound

1: **Variables:**
2: *bound* = amount of allowed stalling
3: calculate paths using a MAPF-algorithm
4: assign each agent to its own component
5: iterate through all paths until agents from separate components overlap within $\pm bound$
6: **while** components occupy the same position within $\pm bound$ **do**
7: merge overlapping components
8: iterate through all paths until agents from separate components overlap within $\pm bound$
9: **return** components

the time of the deviation, the local planner can then simply use the component corresponding to the current time step, i.e., the components becomes smaller during the later stages of plan execution since less paths intersect. The calculation can happen offline, either before or during plan execution.

5

Evaluation

Here we describe the method that will be used to evaluate the algorithms and the system. Firstly, we present the research questions that we aim to answer. Secondly, we show the evaluation criteria which will be used to compare the results of our experiments. Thirdly, we go through the experiment plan and the experiments that we will use to answer our research questions. Lastly, we present the evaluation environment which shows the problem instances that the experiments will solve.

5.1 Research questions

- Q1. Consider a multi-agent system where agents follow a path according to a MAPF algorithm. Assume that there is a known probability p for an agent to be delayed, according to the model in Section 3.2.1. Optimal MAPF schedules are invalidated when at least one agent is delayed. In this case, recalculation is needed. How often is it necessary to recalculate the paths?
- Q2. Consider a system where delays are present according to the model in Section 3.2.1, where each delay requires all agents to stop and recalculate the schedules before resuming. How can one mitigate the impact of these delays? Can we reduce the number of recalculations? Can we reduce the number of agents that needs to pause due to these delays?
- Q3. When agents are delayed and fall behind schedule, is it worthwhile to use the refined component stalling algorithm presented in Section 4.2?
- Q4. Is it worthwhile to find a delay-tolerant schedule using the DT-CBS algorithm presented in Section 4.1?
- Q5. Is it worthwhile to use the DT-CBS algorithm that finds a delay-tolerant schedule presented in Section 4.1 together with refined component stalling presented in Section 4.2 to mitigate the impact of delays?

5.2 Evaluation criteria

During the evaluation the algorithms are compared with CBS according to these evaluation criteria:

- The execution time of the path finding algorithm that finds the initial schedule to be followed. We do not measure the number of operations but rather the run-time of the algorithm on a mid-ranged cost computer. The execution time of a schedule is highlighted in Figure 5.1.
- Makespan of the initial schedule, i.e., the longest path length of any agent in the schedule. The makespan gives a measurement on how long it takes to execute the entire schedule in a system.
- SIC of the initial schedule, i.e., the sum of the path lengths of all agents in the schedule. This divided by the makespan gives an idea of the average number of agents that are moving concurrently, which also affects the number of delays in the system.
- The number of nodes generated by CBS and its delay-tolerant version, which correlates to the memory usage of the algorithm.
- Success rate, i.e., how often the algorithm successfully solves a given problem within 5 minutes. In small maps with a lot of agents, the success rate is expected to decrease.

For some experiments, the evaluation will take place in a system where delays are present and where recalculations may be needed. In these cases the evaluation criteria will also include the following:

- Number of time slots where at least one delay occurs. This is equal to the number of recalculations of algorithms not tolerant of delays.
- Number of recalculations that are necessary due to delays. If this is lower than the number of time slots where at least one delay occurs, it means that the algorithm successfully reduced the number of recalculations needed due to delays.
- The total execution time of all recalculations. This shows how long agents need to stop in order to handle delays that occur during run-time. In Figure 5.1, the total execution time of all recalculations is the sum of the two recalculation times.
- The stretch factor, i.e., after plan execution, how much did the makespan and sum of individual costs increase due to delays? This shows how the algorithms that handle delays affect the path lengths.

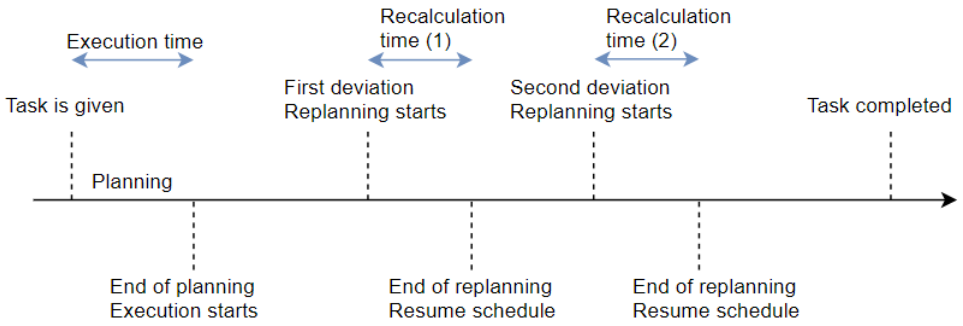


Figure 5.1: The execution time and recalculation time of a schedule.

5.3 Experiment plan

Here we present the experiments that we intend to use to evaluate our work:

5.3.1 Frequency of recalculation with MAPF

This experiment evaluates how often existing optimal MAPF algorithms are required to do recalculations in a system with delays. The experiment will take place in the evaluation environment presented in Section 5.4, where we will look at the number of time slots where at least one delay occurs, which is equal to the number of recalculations needed. We will also investigate the makespan of the solutions since it affects the number of deviations. The experiment will only consider CBS and no other optimal MAPF algorithm, since all optimal MAPF algorithms should arrive at the same solutions.

Multiple tests will be performed and the number of recalculations for different probabilities of delays will be tested. Delays may occur according to the probabilities $p = [0.1\%, 1\%, 2.5\%, 5\%, 7.5\%, 10\%, 12.5\%, 15\%]$, and these delays will be managed by stopping all agents and doing recalculation with an optimal MAPF algorithm. Only an 8x8 map with no obstacles will be used, with the intention to reduce the computational need while getting an overview of how optimal MAPF performs in regards to recalculations in a system with delays.

5.3.2 Evaluation of CBS and DT-CBS

This experiment evaluates the performance of our implementations of CBS and DT-CBS. The results will be used to compare how the state-of-the-art MAPF algorithm CBS performs in comparison to DT-CBS presented in Section 4.1. The performance of DT-CBS will be evaluated for delay tolerance $\delta = [1, 3]$.

The experiment will take place in the evaluation environment according to the evaluation criteria presented in Section 5.2. Both an 8x8 map without obstacles and a 21x20 map with obstacles resembling a warehouse will be tested. Note that this experiment will not investigate how CBS performs in an environment where delays are present and where recalculations are needed but instead only test the execution of the algorithms.

5.3.3 Component stalling used with CBS and DT-CBS

This experiment evaluates the refined component stalling presented in Section 4.2 for component bound $\delta = 1$, where CBS is used for recalculation when a delay exceeds δ. Furthermore, the experiment evaluates the performance of DT-CBS used together with refined component stalling. It will be evaluated for delay tolerance $\epsilon = [1, 3]$ and component bound $\delta = 2 \cdot \epsilon$. When selecting δ, we reason that DT-CBS will have a higher run-time, and that a higher δ will lead to fewer recalculations. Furthermore, the bound δ needs to be bigger than ϵ, since a delay tolerant schedule would only stall agents when the delays exceed ϵ, otherwise making the refined component stalling redundant.

The experiment will take place in the evaluation environment with an 8x8 map with no obstacles, where the schedule will be executed in a system where delays occur based on the probability $p = 5\%$. The main objective of this experiment is to investigate if refined component stalling can reduce the number of agents affected by delays. Measurements will be taken according to the second part of the evaluation criteria, which considers the time needed for recalculations and the stretch factor.

5.4 Evaluation environment

The algorithms will be evaluated through measurements taken during simulation. The simulation will run on different configurations with changes to map layout and the number of agents.

Two map layouts will be tested. The first map is based on the experiments conducted by Sharon *et al.* for CBS [16], it is an 8x8 map with no obstacles where the number of agents will range from $k = [3, 13]$. The other map is meant to resemble a warehouse that contains rows of shelves with narrow corridors in between them, as well as a

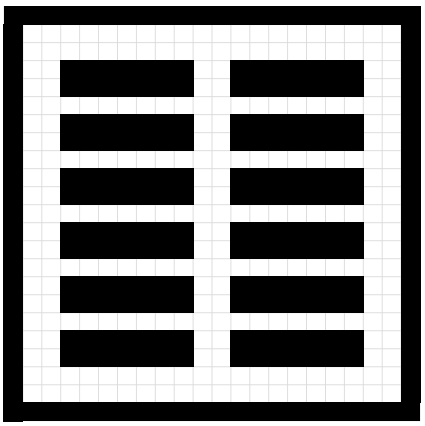


Figure 5.2: Warehouse map

number of wider pathways, which can be seen in Figure 5.2. The number of agents in the warehouse map will also range from $k = [3, 13]$.

Each configuration will be run on 100 instances where the start and goal positions of the agents will be randomized. Note that the same 100 instances will be used for the different algorithms to keep the evaluation consistent. For each instance, the global planner will run a MAPF algorithm and compute a schedule. If an instance cannot be solved within 5 minutes it is halted, and the instance will only affect the success rate and no other measurements.

The simulator presented in Chapter 3 can be used to execute the schedule in a system where delays occur according to some probability p. These delays will be managed by the local planner which will change the schedule when necessary, such that no collisions occur.

6

Results

Here we present an experimental evaluation of our contributions presented in Chapter 4, which is used to validate whether or not our contributions improve the performance of MAPF algorithms in a system where agents may be delayed.

The experiments are used to answer the research questions introduced in Section 5.1. How often do existing MAPF algorithms need to do recalculation due to delays when executing in a system where agents may be delayed? How can one mitigate the impact of these delays? Can we reduce the number of recalculations? Can we reduce the number of agents that needs to be paused? Is it worthwhile to use the component stalling algorithm presented in Section 4.2? Is it worthwhile to use our algorithm DT-CBS presented in Section 4.1? Is it worthwhile to combine component stalling with DT-CBS?

From our experiments, where there is a probability of delay $p = 0.05$ on an 8x8 map with 3 to 13 agents, we could observe that MAPF algorithms such as CBS require up to three recalculations during plan execution. The experiments show that DT-CBS can reduce the number of recalculations needed to zero, at the cost of a 10-100 times higher run-time, 5-14% higher makespan, and 2-10% higher SIC. Furthermore, we observe that Refined Component Stalling used together with CBS reduces the number of recalculations to zero, and reduces the number of agents that needs to be paused due to delays by more than half, at the cost of 4-11% higher makespan and SIC.

Looking at the results that show the plan execution time of DT-CBS and stalling, we point out that in some cases DT-CBS is faster than CBS when agents move fast, i.e., when the step time is small. We highlight that in the difficult case when there are more than 10 agents that move every 100ms, we can see that the plan execution of DT-CBS used together with Refined Component Stalling is between 16-49% faster than CBS. This is a hard case for CBS since there are many recalculations and every recalculation is time-consuming, but this shows that there are cases where delay-tolerant path finding provides better performance.

6.1 Frequency of recalculations with MAPF

In this section, we present the results of how often existing MAPF algorithms need to do recalculation when executing in a system where agents may be delayed according to a set of different delay probabilities. The number of recalculations would be the same for any MAPF algorithm, but for this experiment we use CBS. The experiment does not cover the entire evaluation criteria, it simply looks at the number of recalculations done and the makespan in an 8x8 map. During our experiments we could observe that even for small problems where the makespan is 8-10 it is necessary to do at least one recalculation for 5-13 agents if the probability of delay is 5% during plan execution.

6.1.1 Anticipated results

We expect that recalculations will rarely happen when there are few agents and when the probability of delay is low, but as the number of agents and the probability of delay increases we believe there will be instances where at least one recalculation is necessary for every problem instance. Note that for MAPF algorithms recalculation is done every time there is a delay, i.e., the number of recalculations is equal to the number of time slots where at least one delay occurs.

6.1.2 Actual results

Figure 6.1 shows the average makespan that the agents have on the 8x8 map with no obstacles. The median makespan is very similar to the average makespan, but we do not show it to save space. Observe that the makespan increases with the number of agents.

The number of recalculations made by the optimal MAPF algorithm can be found in Figure 6.2. As the probability of a delay increases, the number of unique delays seems to increase in a linear fashion. Similarly, the number of delays becomes higher as the number of agents grows. When the probability of delay is 5% or more and there are at least 5 agents we observe that on average at least one recalculation is needed for each problem instance.

6.1.3 Difference between the anticipated and actual result

The result were as expected, the number of recalculations increased as the number of agents and the probability increased. One thing worth mentioning is that the makespan increases by up to 27% as the number of agents grow, which also affects the average number of delays.

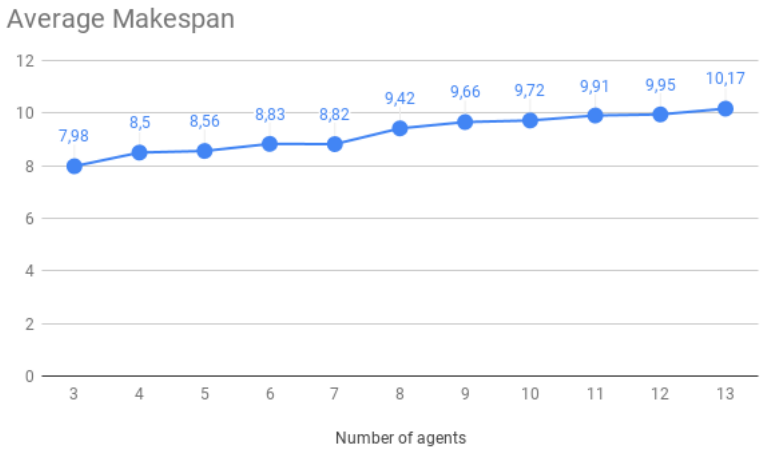


Figure 6.1: The resulting makespan after executing a schedule in a system with delays using CBS for recalculation.

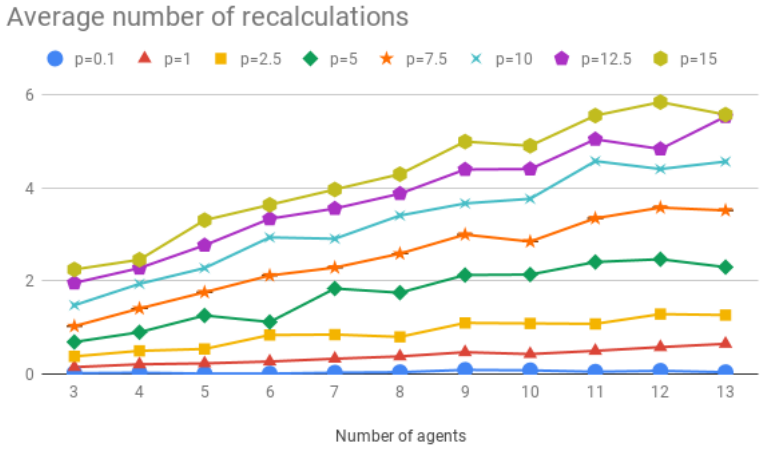


Figure 6.2: Number of recalculations when executing a MAPF schedule, which is equal to the number of time slots where at least one delay occurs.

6.2 Evaluation of CBS and DT-CBS

This section presents the experimental comparison between the standard CBS and DT-CBS according to the first part of the evaluation criteria, which looks at the general performance of the algorithms and the quality of the schedule it generates. The experiments evaluate DT-CBS for delay tolerance $\epsilon = [1, 3]$. The experiments take place in the 8x8 map with no obstacles, as well as the 21x20 map with obstacles resembling a warehouse.

From the experiments, we observe that DT-CBS has a significantly higher run-time than CBS, which becomes worse as ϵ increases. Furthermore, the run-time has a very high variance, i.e., the time it takes to solve the problem instances varies a lot, and this variance is higher for DT-CBS. The makespan and SIC increase slightly, but not significantly, for DT-CBS compared to CBS.

6.2.1 Anticipated results

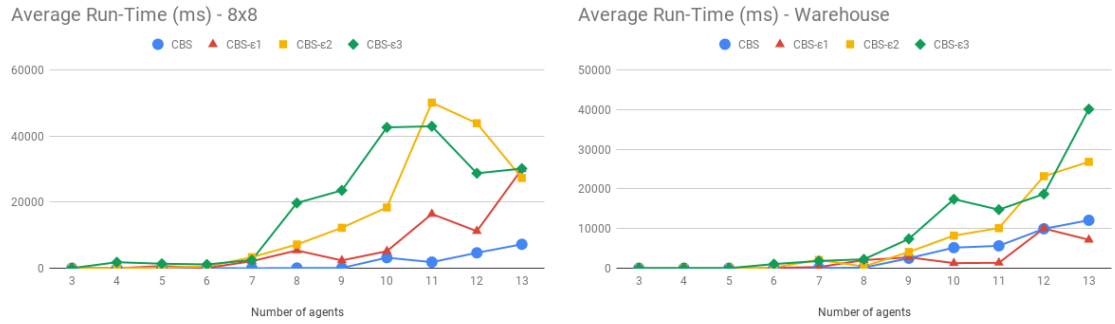
Since DT-CBS is an adaptation of regular CBS, where agents essentially occupy more than one location at any point in time during the execution of the algorithm, we expect to see more conflicts and therefore more nodes generated before finding the solution. Because CBS is exponential in the number of conflicts, we should also see an increase in run-time as ϵ gets bigger. However, we anticipate that the number of recalculations needed when executing the schedule is reduced compared to regular CBS since the schedule should be delay-tolerant for up to ϵ delays. Since this experiment only covers the algorithms and not the plan-execution of their schedules, the results on the number of recalculations will be covered in Section 6.3.

Because we expect the run-time to be higher for DT-CBS than CBS, we believe that the success rate of DT-CBS will be lower, as the success rate is based on if the algorithm can solve the given problem within five minutes.

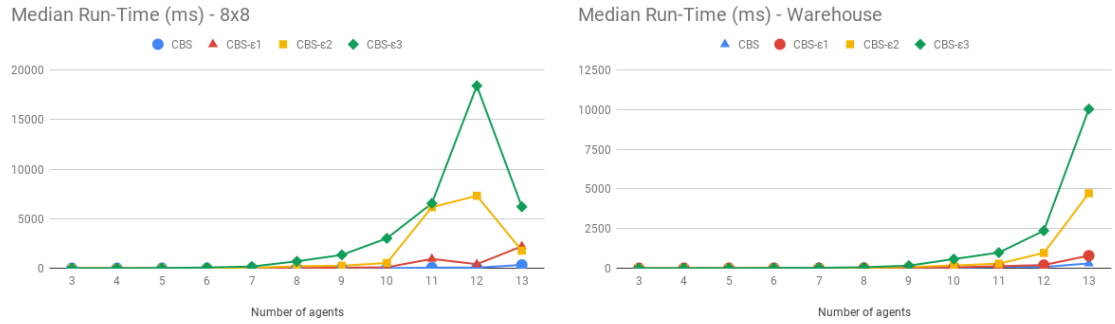
We also expect to see a bigger makespan and SIC for the schedules provided by DT-CBS compared to the ones provided by CBS. This is because CBS provides optimal schedules while DT-CBS puts further requirements on the solutions, the accepted solutions should be a bit more costly.

6.2.2 Actual results

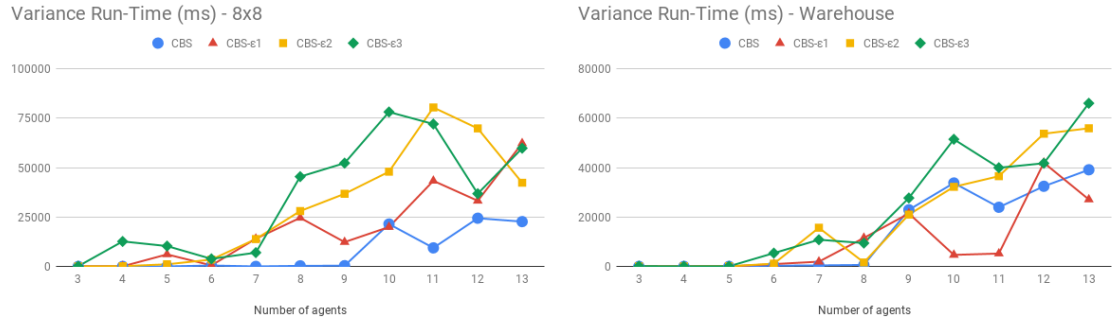
From Figure 6.3 we can see that both the average and median runtime of DT-CBS is higher than regular CBS and that a bigger ϵ seems to contribute to a longer run-time. We can also observe from the figure that the variance in run-time for both algorithms are very high, both from the variance graph and by observing the big differences between the values in the median and average run-time graphs. Furthermore, while still significant, the variance is less notable in the warehouse map compared to the



(a)



(b)

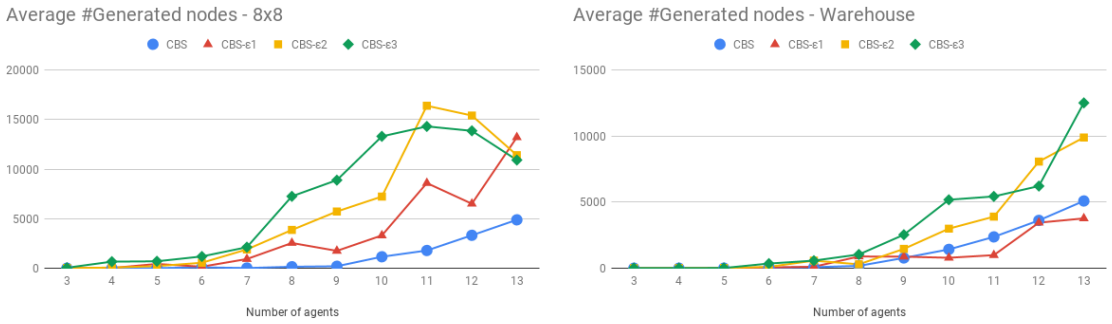


(c)

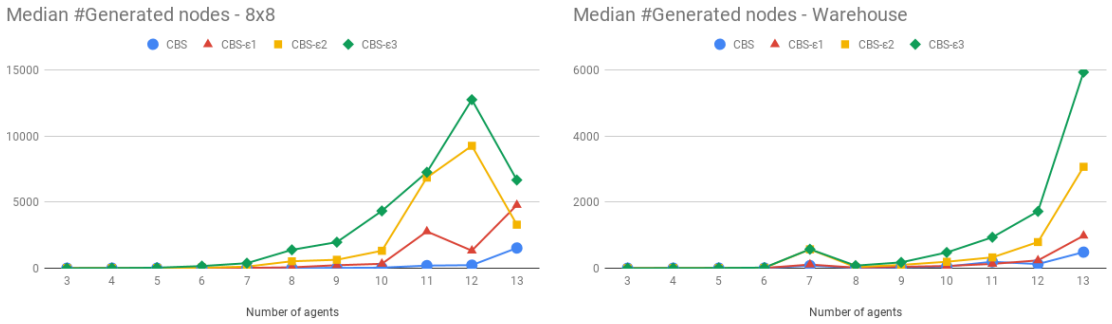
Figure 6.3: Average and median run-time together with the variance, on the 8x8 map (left) and the warehouse map (right).

8x8 map.

Similarly as the run-time, Figure 6.4 shows that the number of nodes generated by DT-CBS is significantly higher than CBS, and the number of nodes necessary to find a solution increases with ϵ. The number of nodes scales according to the number of conflicts while executing the algorithm, which means that DT-CBS encounters significantly more conflicts during its execution. We can also observe a similar discrepancy between the values seen in median and average graphs as with the run-time, indicating a high variance on the number of nodes.

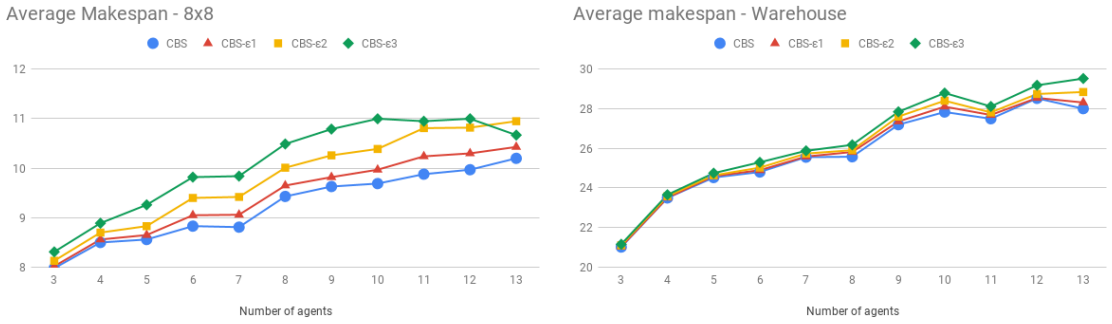


(a)



(b)

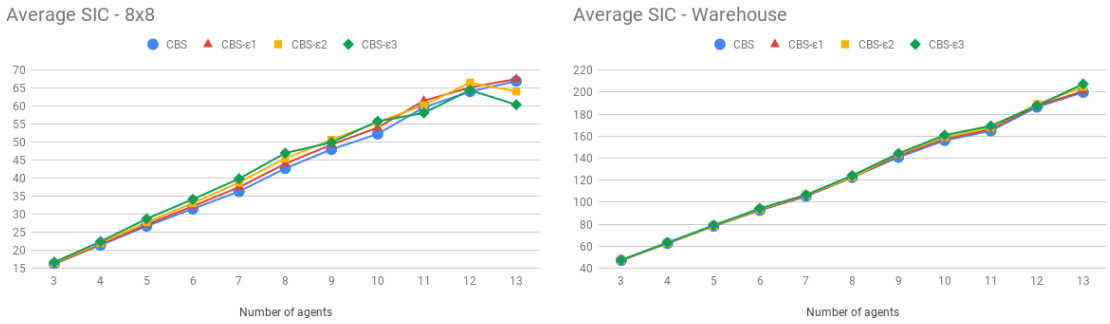
Figure 6.4: Average and median number of generated nodes on the 8x8 map (left) and the warehouse map (right).



(a)

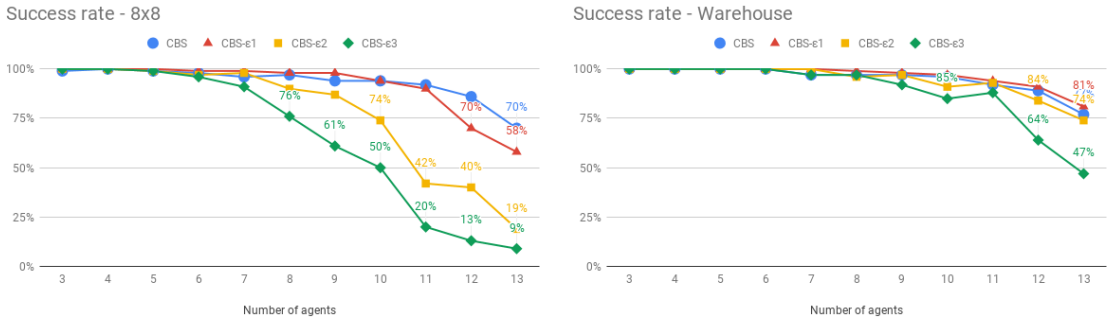
Figure 6.5: Average makespan on the 8x8 map (left) and the warehouse map (right).

Looking at Figure 6.5 we can see that the difference in average makespan between CBS and DT-CBS using $\epsilon = 3$ goes from about 0.5 to 1.5 for 3 to 13 agents. Meaning that a schedule from DT-CBT for 13 agents using $\epsilon = 3$ is only expected to add up to 1.5 time steps to the total execution time. Similarly, we can observe the average SIC in Figure 6.6 , where the difference between the algorithms seems to be somewhere between 0 and 5 for both maps and does not seem to increase significantly more when using more agents.



(a)

Figure 6.6: Average SIC on the 8x8 map (left) and the warehouse map (right).



(a)

Figure 6.7: The success rate on the 8x8 map (left) and the warehouse map (right).

Looking at the success rate visualized in Figure 6.7, we can observe a significant decrease for DT-CBS at $\epsilon = 2$ and $\epsilon = 3$ on the 8x8 map, while CBS and DT-CBS with $\epsilon = 1$ are successful in most of the cases even for $k = 13$. In the result for the warehouse map however, the success rate is much more similar for both algorithms and we only see DT-CBS with $\epsilon = 3$ diverge at $k = 12$ and $k = 13$.

6.2.3 Difference between the anticipated and actual result

As expected the results demonstrated a difference between the run-time of the two algorithms, where DT-CBS takes a significantly longer time. Similarly, our expectations of the increase in generated nodes were correct.

The big variance in run-time and the number of nodes generated was not something we directly foresaw but still did not find unexpected. The variance is most likely a consequence of CBS being exponential in the number of conflicts and not in the number of agents. Algorithms that are exponential in the number of agents would most likely have lower variance and therefore more consistent results. But since the number of conflicts is highly dependent on the map and the placement of the agents, our results exhibit high variance. However, we also observe the positive ramifications

of this by comparing the results from the warehouse map and the 8x8 map where we see that the runtime and the number of nodes generated are often smaller on the warehouse map, even though there are more positions for the algorithm to process on the map.

The reason for the better performance on the warehouse map likely is that CBS, as mentioned in Section 2.3.2, favors environments with bottlenecks over open maps without obstacles. And since the corridors in the warehouse map create bottlenecks for the agents and that the 8x8 map does not have any obstacles, we can see a faster run-time, fewer nodes generated and higher success rate of the algorithms on the warehouse map. A bigger map with more open tiles also reduces the probability of conflicts, as the agents have more open tiles to choose from when calculating their paths. Furthermore, adding a tail of $\epsilon = 3$ to the agents is a significant percentual increase in blocked tiles on the 8x8 map area, as one agent then covers up to 4 tiles which are half the height or width of the map. And all agents for $k = 13$ and $\epsilon = 3$ can cover up to $4 \cdot 13 = 52$ positions on the map, while the entire 8x8 map consists of $8 \cdot 8 = 64$ tiles in total. This is likely to be one of the reasons for the low performance and success rate for DT-CBS on the 8x8 map.

Throughout the results, a fluctuation can be observed when the number of agents gets higher, especially for $k = [12, 13]$ and $\epsilon = [2, 3]$ on the 8x8 map. We believe this is a consequence of the low success rate when using these settings. The low success rate means a smaller set of test cases contributing to the average and median values, making the result less reliable. It also means that the hard problems solved by CBS and DT-CBS with $\epsilon = 1$, which increases the average and median, most likely are the problems that fail. This results in a lower average and median value since only the easy problems are considered.

The results from Figure 6.5 and Figure 6.6 shows that, as expected, the SIC and makespan grows as ϵ gets bigger. However, we can see that the increase in makespan is a bit more substantial than the SIC increase. We expect this is because CBS and DT-CBS prioritize nodes based on SIC and therefore optimizes the solution based on the SIC value.

6.3 Component stalling used with CBS and DT-CBS

This section covers the experimental evaluation of Refined Component Stalling used in conjunction with both CBS and DT-CBS, in accordance with the evaluation criteria. The experiment is used to validate whether or not our contributions can decrease the impact of delays, in a system where agents may be delayed with the probability $p = 0.05$.

The first question that the experiment answers is whether or not CBS can be used together with Refined Component Stalling to reduce the impact of delays. The

second question is whether or not DT-CBS used together with Refined Component Stalling can reduce the impact of delays even further.

The experiments show that both Refined Component Stalling and DT-CBS significantly reduces the number of recalculations necessary compared to CBS. Furthermore, Refined Component Stalling can successfully decrease the number of agents affected by delays. Compared to CBS, we observe that DT-CBS and stalling increase the makespan and SIC slightly, but not by a lot. When Refined Component Stalling is used together with DT-CBS, the experiment shows that the number of stalls becomes lower as the delay tolerance ϵ increases. The results also shows that the plan execution time for DT-CBS and stalling in some cases is faster than that of CBS when agents move fast, i.e., when the step time is small.

6.3.1 Anticipated results

Because DT-CBS has a delay tolerance of ϵ, it allows significantly more delays than standard CBS. This means that DT-CBS will allow more delays before stalling, reducing the number of stalls during plan execution. Furthermore, the number of recalculations will most likely be reduced significantly since more stalls are allowed because the component bound of DT-CBS is $\alpha = 2 \cdot \epsilon$ compared to $\alpha = 1$ of CBS. If the number of recalculations for DT-CBS is close to zero, the total time for recalculations will be zero during plan execution. On the other hand, we expect standard CBS to have many recalculations and therefore a high recalculation time during plan execution.

Following our expectation that CBS has more stalls than DT-CBS because DT-CBS allows more delays, the stretch factor for both the SIC and makespan should be larger for standard CBS, since a stall is usually more costly than a delay in terms of SIC and makespan because it affects many agents. Furthermore, we anticipate that the increased stretch factor of standard CBS should be larger for makespan than SIC, since a stall affects the makespan more than SIC.

DT-CBS is expected to generate more costly initial schedules in regards to makespan and SIC, although the stretch factor is expected to be smaller for DT-CBS. We also expect the average number of time slots where at least on delay occurs will be quite similar for standard CBS and DT-CBS, since this should have a correlation with the SIC and makespan.

We expect to see a difference between the number of agents affected by delays when comparing Refined Component Stalling with stalling, and recalculation of all agents. This is because the bound α in Refined Component Stalling limits the component size, which is related to the number of agents affected by delays, as explained in Section 4.2.

Following our expectation that DT-CBS and Refined Component Stalling reduces the number of recalculations, we expect to see the time it takes for CBS to execute

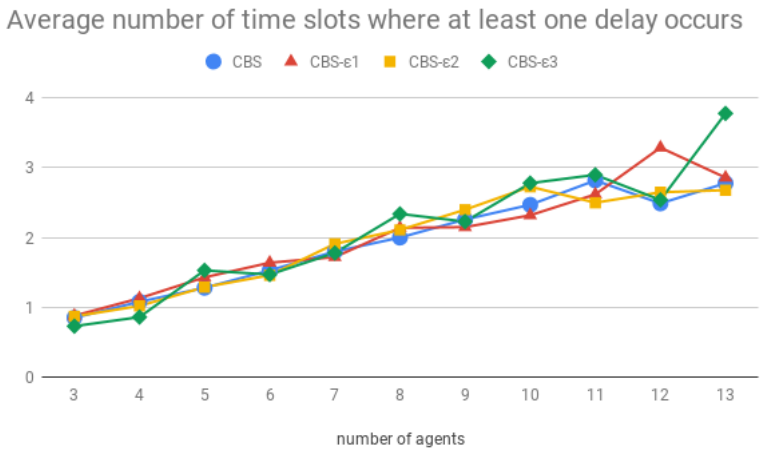


Figure 6.8: Average number of time slots where at least one delay occurs for stalling with CBS and DT-CBS

a schedule, including the recalculation times, to be longer than when using DT-CBS and stalling in some settings. The time for executing a schedule can be calculated by the equation $m \cdot t_s + t_r$, where m is the makespan, t_s is the step time and t_r is the recalculation time. Since the average makespan of DT-CBS is larger than the makespan of CBS and the average recalculation time is expected to be larger for CBS, we anticipate this result to heavily depend on the step time. When using a small step time we should see more influence on the result from the recalculation time, and when using large time steps we expect to see more impact from the average makespan.

6.3.2 Actual results

The performance of CBS and DT-CBS can be found in Section 6.2.2, where the run-time can be seen in Figure 6.3, the makespan in Figure 6.5, the SIC in Figure 6.6, number of nodes generated in Figure 6.4, and success rate in Table 6.7. Here we present the post plan-execution results of CBS and DT-CBS used together with stalling.

Figure 6.8 shows that the number of delays is quite similar for CBS and DT-CBS, and that increases with the number of agents. We can also see some fluctuations at $k = [12, 13]$ agents.

Figure 6.9 shows that the average number of stalls for CBS is significantly higher than DT-CBS. As ϵ grows the number of stalls required by DT-CBS seems to go towards zero. Similarly, Figure 6.10 shows that the number of recalculations for standard CBS goes from 0,18 to 0,75 as the number of agents increases, while DT-CBS has close to zero recalculations for all the tests. Note that the results show a peak at 11 agents for both the number of recalculations and the number of stalls.

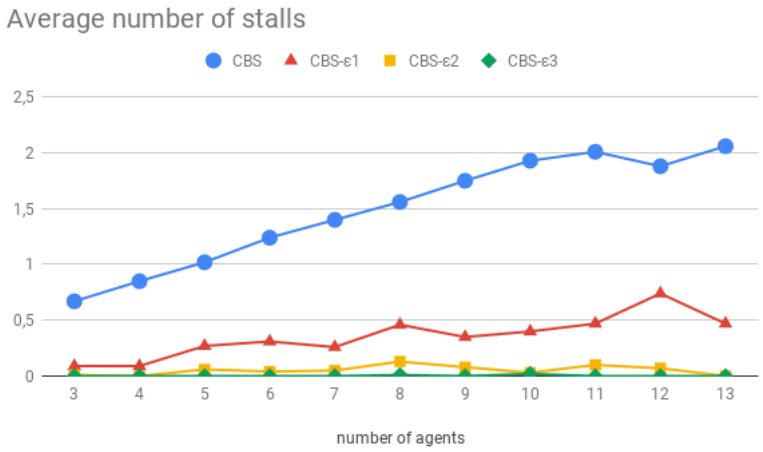


Figure 6.9: Average number of stalls for standard CBS and DT-CBS with stalling bound $\alpha = \max(1, 2 \cdot \epsilon)$.

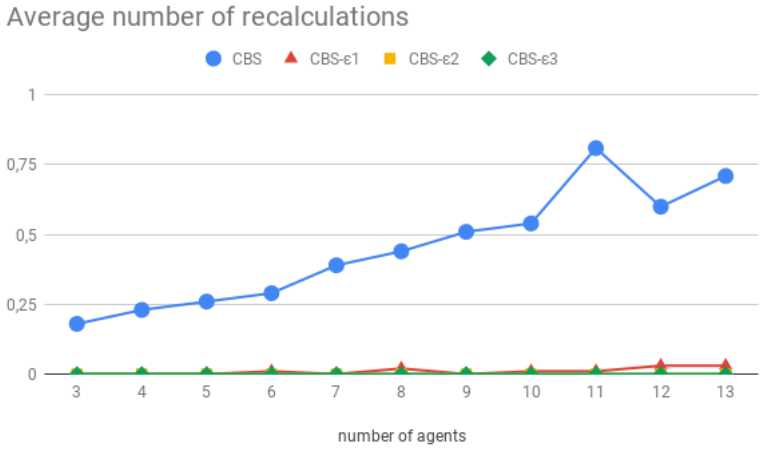


Figure 6.10: Number of recalculations for stalling used together with standard CBS and DT-CBS

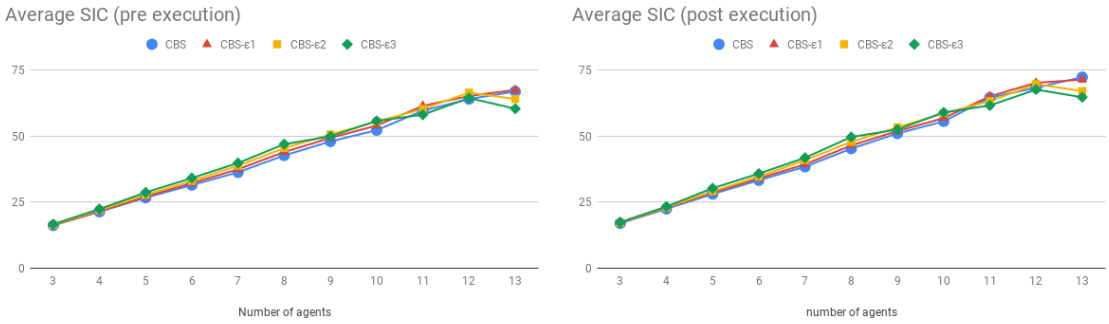


Figure 6.11: SIC before executing the schedule to the left. SIC after executing the schedule with stalling and standard CBS and DT-CBS to the right.

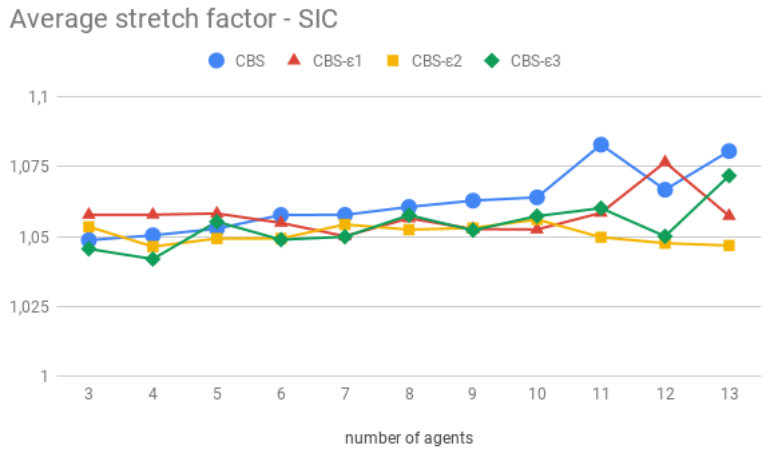


Figure 6.12: Stretch factor of SIC for stalling used together with standard CBS and DT-CBS

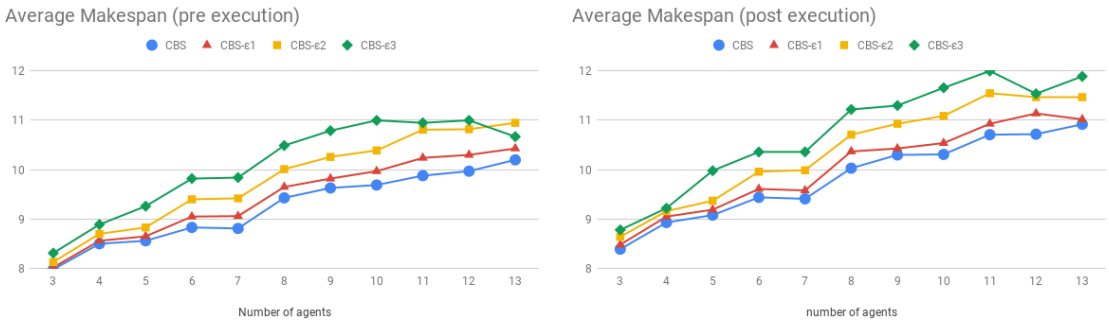


Figure 6.13: Makespan before executing the schedule to the left. Makespan after executing the schedule with stalling and standard CBS and DT-CBS to the right.

Figure 6.11 shows the average SIC before and after plan execution. We can observe that the SIC after plan execution does not differ very much between CBS and DT-CBS. However, the SIC of DT-CBS with $\epsilon = 3$ is between 2% to 10% larger than CBS for $k = [3, 10]$ agents. Note that after $k = 10$, the SIC of DT-CBS with

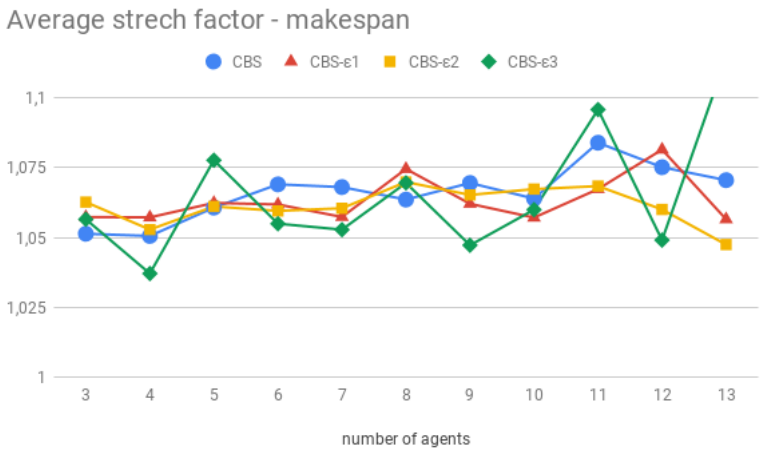


Figure 6.14: Stretch factor of makespan for stalling used together with standard CBS and DT-CBS

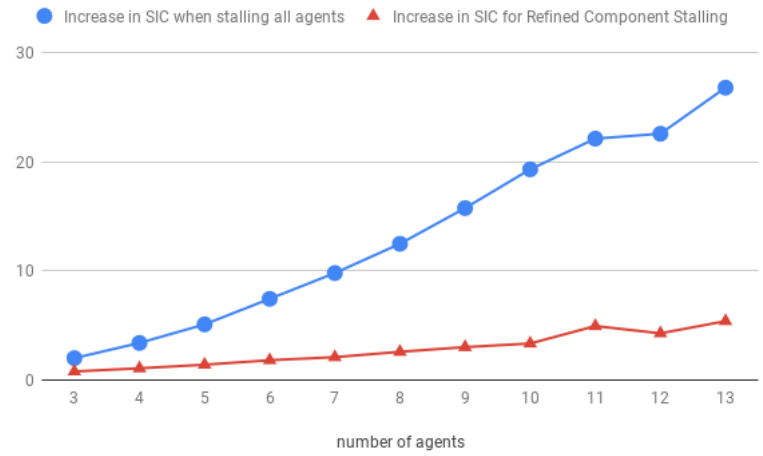


Figure 6.15: SIC when stalling all agents and when using refined components

$\epsilon = [2, 3]$ dips and becomes smaller than CBS. Figure 6.12 highlights the differences of the SIC in Figure 6.11, where we observe that the SIC increases by about 5% to 10% during plan execution. The stretch factor of CBS generally seems to be a bit higher than DT-CBS, although not by too much. Note that there are some peaks in the result, e.g., at $k = 13$ agents DT-CBS with $\epsilon = 1$ has a higher stretch factor than CBS.

Figure 6.13 shows the average makespan before and after plan execution. After plan execution, we can see that the average makespan is 3% to 13% higher for DT-CBS than CBS. Observe that the makespan increases by 4% to 9% during plan execution, where the increase seems to grow slightly bigger as the number of agents increases, as highlighted by the stretch factor in Figure 6.14. The stretch factor seems slightly larger for CBS than DT-CBS, but note that there are some peaks which makes the results hard to read, especially for DT-CBS with $\epsilon = 3$.

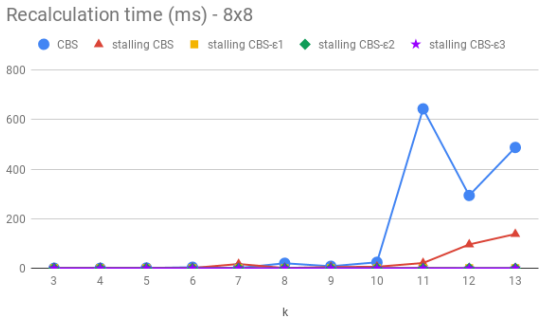


Figure 6.16: The average total time needed for recalculations during plan execution in an 8x8 map, where the probability of delay is 5%.

Figure 6.15 shows the difference in increased SIC when stalling all agents and when utilizing Refined Component Stalling as delays occur. We can see that by using Refined Component Stalling we only attribute to around 20% to 40% of the increase in SIC compared to stalling all the agents (100%).

Figure 6.16 shows the recalculation time of CBS, CBS with stalling, and DT-CBS with stalling. The recalculation time is very low for few agents, but for $k = [10, 13]$ agents we observe that the recalculation time increases slightly for CBS and for CBS used together with stalling, while the recalculation time of DT-CBS used together with stalling stays at zero because it does not do any recalculations.

Figure 6.17 shows the total time it takes to execute the schedule, calculated by $m \cdot t_s + t_r$, where m is the makespan, t_s is the step time, and t_r is the recalculation time. Note that this does not include the time it takes to calculate the initial schedule, but it includes the recalculation time. In the slow system with a step time of 500ms, CBS consistently outperforms the delay tolerant algorithms when it comes to plan execution time. Observe that the total plan execution time is generally faster for CBS in the fast system with a step time of 100ms as well, but there are some cases when DT-CBS and stalling have a faster plan execution time than CBS, which partly depends on the higher recalculation time of CBS in these cases, as seen in Figure 6.16.

6.3.3 Difference between the anticipated and actual result

As expected, the number of delays were quite similar for CBS and DT-CBS. Similarly to other experiments, we believe that the fluctuation towards $k = [12, 13]$ are caused by a drop in success rate which means that the average is calculated from fewer tests.

According to our prediction, the number of stalls and recalculations were significantly lower for DT-CBS than standard CBS. This implies that the total time for recalculation of DT-CBS during plan execution is zero in most cases, in small maps with 3-13 agents. On the other hand, the average number of recalculations for CBS is lower than expected, where we can see it per average requires less than one recal-

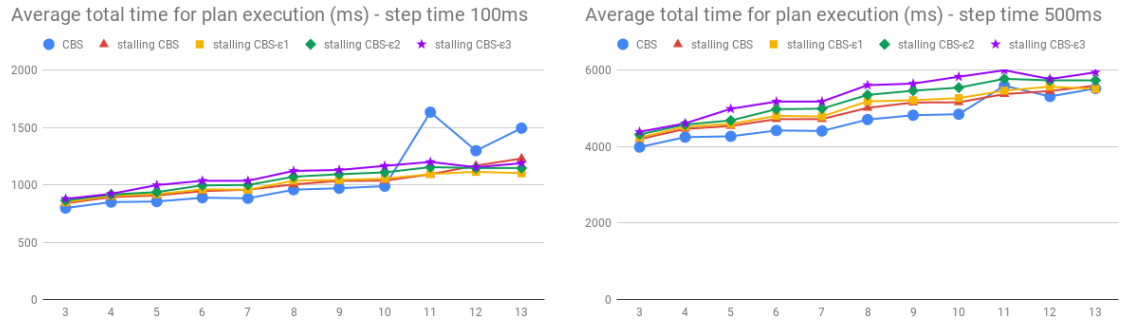


Figure 6.17: The total time it takes to execute the schedule, calculated by $m \cdot t_s + t_r$, where m is the makespan, t_s is the step time, and t_r is the recalculation time. The graph to the left has $t_s = 100ms$ and to the right $t_s = 500ms$.

culation. There is a spike in the number of recalculations for CBS at $k = 13$ agents, which we believe may be caused because the number of delays also has a small spike for $k = 13$ agents.

We predicted that the stretch factor of SIC and makespan would be higher for CBS than DT-CBS, which turned out to be true, although the difference was smaller than we predicted.

We can see that the stretch factors spike at $k = [11]$ for CBS and $k = 12$ for DT-CBS with $\epsilon = 1$. For DT-CBS, we believe that the cause is a spike in the number of delays at the same time, which makes sense since the number of delays should have a correlation to the SIC, since a delay increases the SIC. For CBS, we also observe that the number of delays increases at $k = 11$, which causes the number of stalls to peak at $k = 11$, further increasing the SIC. Furthermore, the stretch factor of the makespan of DT-CBS with $\epsilon = 3$ fluctuates a lot, which correlates to the number of delays, although the number of delays affected the stretch factor more than anticipated. We believe that the component bound $\alpha = 2 \cdot \epsilon$ may be the cause since a higher component bound results in larger component groups which cause more agents to be affected by a stall. Since stalls affect the makespan significantly, it does make sense that DT-CBS with higher ϵ should give higher spikes in the makespan.

From the results, we conclude that Refined Component Stalling can reduce the number of agents affected by a delay, compared to stalling all agents. We can draw this conclusion by comparing the SIC since each agent that is stalled for one time unit should add 1 to the SIC after plan execution. Since we only see a 20% to 40% increase in SIC compared to stalling all agents, we can see that we have successfully reduced the number of agents affected by a delay by using Refined Component Stalling.

We expected the recalculation time of CBS with stalling to be longer than DT-CBS with stalling, as it turns out the recalculation time was mostly close to zero due to the number of recalculations being small. But we can see that when recalculations

occurred, the recalculation time was higher for CBS than DT-CBS as anticipated. Similarly, due to the lack of recalculations it was hard to see if the plan execution time of DT-CBS was better than CBS, but we note that in the cases where recalculations occurred we could see that DT-CBS actually had a faster plan execution than CBS, in the case where the system had a lower step time and agents moved fast. This suggests that DT-CBS used together with stalling performs better in systems where the agents move fast.

We believe that the total time for plan execution would have a more consistent result if a larger map was used. This depends on two factors, the recalculation time and the makespan. In a larger map, there would occur more recalculations due to longer path lengths, which would increase the recalculation time so that it would be more consistent. Furthermore, according to our results, the makespan seems to increase more proportionately in smaller maps than larger maps for DT-CBS compared to CBS, which means that DT-CBS will affect the makespan less for larger maps and it will get a better plan execution time.

7

Discussion and conclusion

In this chapter, we include a discussion about the project and the results and present our conclusions.

7.1 Discussion

The results show that the time required for plan execution for DT-CBS and component stalling performs better than CBS in some cases. More specifically, they seem to outperform CBS in hard cases where there are many agents and lots of delays, although more extensive experiments would be required to be certain that this is always the case.

As shown in the results, Refined Component Stalling could successfully reduce the number of agents affected by delays, and it also decreases the number of recalculations needed. The impact that Refined Component Stalling has on the number of agents affected is heavily dependent on the component bound δ. The experiments we carried out covered $\delta = 1$ for CBS, and $\delta = 2 \cdot \epsilon$ for DT-CBS with delay tolerance $\epsilon \in \{1, 2, 3\}$. In order to get a clearer picture of exactly how δ affects the performance of Refined Component Stalling, more thorough testing would be needed.

We found that DT-CBS, which finds a delay-tolerant schedule for a MAPF problem, has a higher computational cost compared to CBS, which finds a non delay-tolerant solution. Although still worse than CBS, DT-CBS seemed to perform better relatively to CBS in the warehouse map compared to the 8x8 map with no obstacles. Furthermore, DT-CBS showed a clear improvement in reducing the number of recalculations and the impact of delays in the system during plan execution. This can be considered as a trade-off between a worse performance during the offline calculations for improved performance during the plan execution. This can be very beneficial if the schedule for the next task can be precomputed in parallel with plan execution of the current task.

Consider a system where agents have to finish a set of tasks. During the time that the current task is being executed, it is possible to precompute the schedule of the next task. This is only possible if the time required for plan execution of

the task is higher than the run-time of the algorithm. By precomputing schedules with DT-CBS, we remove the disadvantage of DT-CBS, which is the increased run-time, while we keep the advantage of fewer recalculations. This means that the system very rarely needs to pause, either for recalculation or for computing task schedules. To further remove the need for recalculations, DT-CBS can be combined with component stalling which acts as a fail-safe, such that if the delay exceeds the delay tolerance, it is still not necessary to do recalculation.

Now assume that it is possible to precompute the schedule of the next task. From our results, we observe that the plan execution time of DT-CBS in some cases outperform the plan execution time of CBS. Our experiments were limited to small maps, due to time restrictions. We believe that in larger maps where there is an increase in makespan and the number of recalculations, DT-CBS may consistently outperform CBS during plan execution, especially in a fast system where the step time is low.

One problematic aspect of CBS is that the run-time is unpredictable, as CBS is exponential in the number of conflicts rather than the number of agents and the map size. Given two very similar problems in the number of agents and the map size, one problem may finish almost instantly while the other one will run for a long time. This is clear when looking at the success rate in our results, where we observe that the success rate of DT-CBS is worse than CBS. We believe that it would be interesting to look at other MAPF algorithms with more predictable run-times and explore if these could be made delay-tolerant while preserving consistent run-times.

We saw a pattern that when the success rate decreased, the results became less consistent, which decreases the reliability of results where the success rate is low. In order to counteract this, it would better to have more test cases than 100. Furthermore, running more than 100 instances would give a more consistent evaluation, as the results changed significantly depending on which problem instances were included in the evaluation. In the end, we decided to limit the number of problem instances to 100 anyway, as the tests are very time consuming, but we used the same problem instances for all tests.

7.2 Risk analysis and ethical considerations

The area of the studied problem considers safety-critical of self-driving agents and in general transport is a very important sector in society. The exact studied problem deals with abstract agents that solve a planning problem. The plan then needs to be carried out by a safety-critical system. We note that there is another system that carries out the plan and it has an autonomous ability to always perceive safety regardless of the plan provided by the studied algorithms. The studied algorithms do consider improved performance and some of them also consider optimization. As such, there is a clear benefit for society since these improvements can imply reduced costs of future systems.

Increased performance and reduced cost in systems that are using MAPF algorithms, such as autonomous warehouses, means that energy consumption potentially can be reduced. By improving the efficiency of such systems, there would likely be less energy, time and money wasted both in existing autonomous systems and future systems. This could result in making these systems more common, increasing the growth of autonomous systems, as the benefits of them become clearer.

Furthermore, something that cannot be neglected is that there are a lot of people affected when these systems are automated, e.g., employees currently working in warehouses that have a risk of losing their jobs to automation. Although this is true to some extent, new jobs are certainly created within maintenance, development, and also logistics since these systems still require some kind of human interference when things go wrong. The process of constantly improving the efficiency of existing infrastructure can be seen as a necessary part of the development of society.

7.3 Conclusion

Multi-agent path finding algorithms can find collision-free paths for a set of agents with different start and goal positions. When one of these agents is delayed, all paths are generally invalidated and the agents need to pause for recalculation of their paths.

We have successfully modified the MAPF algorithm CBS so that the paths found by CBS is tolerant to delays. The modified algorithm, which we call DT-CBS, allows the user to select the delay tolerance that the paths should achieve. Our experiments show that the number of recalculations needed when using DT-CBS is significantly lower compared to CBS, at the cost of an increased run-time of the algorithm.

We have introduced Refined Component Stalling, which can be used to decrease the impact of a delay by avoiding recalculation while only affecting a subset of all agents, at the cost of slightly longer path lengths. More specifically, when an agent is delayed, the nearby agents that it may collide with due to the delay are stalled to compensate for the delay. The number of agents affected by a delay can be tuned through a parameter which we call the component bound, and with a smaller bound fewer agents are affected by delays, at the cost of an increased number of complete recalculations.

In this project, we have explored the area of delay tolerance in multi-agent path finding. We have concluded that DT-CBS used together with stalling in some cases have a lower plan execution time than CBS when the agents move fast, which suggests that DT-CBS and stalling may work more efficiently for fast systems. Further development is required to find less computational expensive alternatives to CBS, such as expanding more recent suboptimal variants of CBS to become delay-tolerant.

Bibliography

[1] Altan Yalcin. *Multi-Agent Route Planning in Grid-Based Storage Systems*. PhD thesis, European University Viadrina, Germany, Sept 2017.

[2] Autostore. `https://autostoresystem.com/`. Accessed: 2019-05-29.

[3] Wolfgang Hönig, T. K. Satish Kumar, Liron Cohen, Hang Ma, Hong Xu, Nora Ayanian, and Sven Koenig. Summary: Multi-agent path finding with kinematic constraints. In *International Joint Conference on Artificial Intelligence 2017*, pages 4869–4873, 2017.

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

[5] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[6] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219:40 – 66, 2015.

[7] Ariel Felner, Roni Stern, Solomon Eyal Shimony, Eli Boyarski, Meir Goldenberg, Guni Sharon, Nathan R. Sturtevant, Glenn Wagner, and Pavel Surynek. Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges. In *The International Symposium on Combinatorial Search*, 2017.

[8] David Silver. Cooperative pathfinding. *AIIDE*, 1:117–122, 2005.

[9] Trevor Standley. Independence detection for multi-agent pathfinding problems. In *AAAI*, 2012.

[10] M Barer, Guni Sharon, Roni Stern, and A Felner. Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. *Frontiers in Artificial Intelligence and Applications*, 263:961–962, Jan 2014.

[11] Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence*, 195:470 – 495, 2013.

[12] Trevor Scott Standley. Finding optimal solutions to cooperative pathfinding problems. In *Proceedings of the National Conference on Artificial Intelligence*, volume 1, Jan 2010.

[13] E. Rohmer, S. P. N. Singh, and M. Freese. V-rep: A versatile and scalable robot simulation framework. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1321–1326, Nov 2013.

[14] N. Koenig and A. Howard. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 3, pages 2149–2154, Sep. 2004.

[15] John D. Hunter. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[16] Guni Sharon, Roni Stern, Ariel Felner, and Nathan Sturtevant. Conflict-based search for optimal multi-agent path finding. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, AAAI'12, pages 563–569. AAAI Press, 2012.