Delft University of Technology
Master of Science Thesis in Computer and Embedded System Engineering

Decentralized Multi-Agent Conflict Resolution in Path Planning

Marco Nicola Stroia

Decentralized Multi-Agent Conflict Resolution in Path Planning

Master of Science Thesis in Computer and Embedded System Engineering

Embedded Systems Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Van Mourik Broekmanweg 6, 2628 XE Delft, The Netherlands

Marco Nicola Stroia

5 July 2024

Author
Marco Nicola Stroia (M.N.Stroia@student.tudelft.nl)
(marcostroianicola@gmail.com)

Title
Decentralized Multi-Agent Conflict Resolution in Path Planning

MSc Presentation Date
18th July 2024

Graduation Committee

Koen Langendoen	Delft University of Technology
Mathijs de Weerdt	Delft University of Technology
Tom van Groeningen	Sioux Technologies Netherlands

Abstract

Path finding is an important component in solving a wide array of engineering problems, ranging from video games to real-life applications such as automated warehouse management and autonomous vehicles. Path finding algorithms are designed to solve complex problems, and in order to do so, assumptions are necessary to simplify the problems. While these assumptions are important, using them makes the obtained algorithms less applicable to a real-life scenario, and as such, verifying how lifting some of them would affect the obtained results is worth pursuing.

Two main assumptions were identified and subsequently lifted. First, classic multi-agent path finding algorithms use a centralized approach, where solutions are computed before execution. This results in a long computation period followed by execution. Lifting this assumption results in a decentralized approach where agents solve conflicts on the go, while approaching their target. The second assumption made by state of the art algorithms is that agents participating in a multi-agent path finding problem share a common goal: minimizing a global cost function. This is not always applicable, as in a real-life scenario participating agents can have selfish goals. This assumptions has been lifted by allowing agents to negotiate their paths by trading with the other participants to create better solutions for themselves.

The Selfish Localized Pathfinding (SLP) algorithm has been designed to lift these assumptions. It describes a decentralized algorithm that allows participating agents to negotiate their paths, which makes a good candidate for an application closer to real-life.

The SLP algorithm has been tested in order to evaluate its performance, both in terms of its ability to solve a set of test cases, and in terms of the cost incurred by the participating agents. SLP performed well in varied domains. SLP solved significantly more cases than Conflict Based Search, a centralized state of the art path finding algorithm. This comes at the expense of an increase in the path lengths obtained by the algorithm. This downside is offset by the significant decrease in the time required to solve problems, which can be divided in small clusters due to the decentralized approach of the algorithm. On the whole, SLP can provide a good alternative to Conflict Based Search.

Preface

Automated Guided Vehicles (AGVs) are an increasingly more present technology in our day to day life. Autonomous cars are approaching real-life practicability at a rapid rate, and warehouse stock is managed by robots. The company Sioux Technologies is also interested in the progress made in the area of AGVs. One important component of AGV operation is path planning, as navigating environments without collisions is essential. Multiple algorithms exist that can solve path finding problems relating to multiple agents. Conflict Based Search (CBS) is one of these algorithms, and it stands as the base for multiple other path finding algorithms. CBS is an algorithm constructed on a set of assumptions, including but not limited to a centralized entity computing solutions for all agents and agents altruistically willing to extend their paths for the benefit of the whole group. For this thesis, the impact of lifting some of these assumptions has been studied, examining how their removal influences the path finding performance, both in terms of path length and solving speed.

I want to thank my professor, Koen Langendoen for his advice throughout my research. I would also like to thank my company supervisors and colleagues from Sioux Technologies for their help and guidance during my research. Furthermore, I would like to thank my friends and family for their support and for listening to my grievances throughout the process.

Marco Nicola Stroia

Delft, The Netherlands
5th July 2024

Contents

Chapter 1

Introduction

Automated Guided Vehicles (AGVs) are an increasingly more present technology in our day to day life. Autonomous cars are approaching real-life practicability at a rapid rate, and warehouse stock is managed by robots (Figure 1.1), reducing human effort [7].

Figure 1.1: **Warehouse robots. Source:** [4]

An important aspect of AGV operations is the ability to plan paths between two different locations. Without careful planning, a dense network of robots will inevitably lead to collisions, which can have consequences ranging from small property damage, in the case of warehouse robots, to loss of life, in the case of autonomous cars.

One party interested in AGV path finding is the company Sioux Technologies B.V.. This research has the goal of furthering the in-house knowledge on path finding algorithms of Sioux.

Path finding problems can be divided into two types, based on the number of agents that have to be accounted for when solving the problem. In this context, the term "agent" is an abstraction of a possible real-life AGV implementation, such as a warehouse robot and an autonomous car, meant to reduce the complexity of path finding problem definitions. The first type is single-agent path finding (SAPF) problems. A SAPF problem accepts a start location, a target location and an environment as an input, with the desired result of a path going from the start to the end location that avoids collisions between the agent and the environment, e.g. an autonomous warehouse robots should not hit walls or the shelves used to store goods. The second type of problems are multi-agent path finding (MAPF) problems. These problems accept multiple start and target locations, each pair corresponding to a different agent. Similarly to SAPF problems the desired output is a set of paths connecting all start and target locations. An additional requirement of MAPF problems besides avoiding collisions with the given environment, is the need of avoiding collisions between different agents. These possible collisions are referred to as “conflicts”, and a description of these conflicts will be given in Section 2.3.

The literature contains research on multiple methods of solving MAPF problems. The current state of the art is defined by Conflict Based Search (CBS) [11], which led to additional research building on top of it, such as Enhanced Conflict Based Search (ECBS) [1] and Explicit Estimation Conflict Based Search (EECBS) [9]. These methods have been proven to find fast and optimal solutions, but they are designed to solve problems given a particular set of constraints and assumptions, which might significantly differ from real-life conditions.

1.1 Problem Statement

Assumptions are an essential way to simplify complex scenarios into abstract, clearly defined problems. While important, assumptions might create unfeasible or undesirable conditions when bringing the solutions back into the real world. The current MAPF state of the art solutions are also built on top of a set of assumptions. While not all assumptions will be listed, the following paragraphs present them in decreasing order of importance to this research.

The output is computed in a centralized manner prior to execution [11] (**the centralized assumption**). In more detail, all agents share the information with a central authority that is tasked with computing the paths for all agents, given a set of start and target locations. Computing MAPF solutions in a centralized manner before execution is infeasible in most real-world scenario. For example, given a system of autonomous cars in the city, planning the routes before execution would require all cars to leave at the same time.

The cost metric used for choosing a solution is a shared global metric [11], such as the sum of the path length of each agent (**the altruistic assumption**). In order for this metric to be useful, the agents must be truthful to each other when declaring their costs, and they must be altruistic and collaborative, prioritizing minimizing the global cost, as opposed to reducing their own cost. In a real-life, non-ideal, scenario the altruism of agents cannot be assumed. For example, individuals in different cars would be more interested in reaching their destination earlier, than in ensuring fairness with the other traffic participants.

All agents are willing to share information with the central authority, e.g. start location, goal, planned path (**the shared knowledge assumption**). Depending on the manner in which the central authority is implemented, this can mean that agents might share information with all other agents. This might be undesirable, as agents might consider sharing information unnecessary, and might choose to only share information with other agents in close proximity that influence their decisions in the short term.

While these (and other) assumptions result in efficient algorithms and satisfactory solutions, it is relevant to ask how removing them would influence the results in order to verify how a MAPF algorithm would perform in a scenario closer to real-life. Thus the following problem statement arises:

> How can these three assumptions be lifted while still solving multi-agent path finding problems? How will lifting them affect path-finding performance?

1.2 Contribution

The main goal of this research is proposing an alternative approach to solving MAPF problems that lifts the assumptions listed in Section 1.1.

To lift **the centralized assumption**, we propose that the agents plan their own SAPF solution before execution, and solve smaller MAPF problems while executing the planned path, when the possibility of running into other agents arises. The MAPF problems are constructed by creating clusters of nearby agents, treating their current position as a start location, with the new solutions replacing the old solutions. A visual representation of the process can be seen in Figure 1.2.

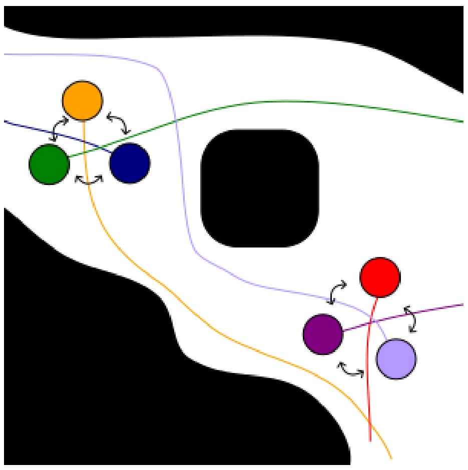


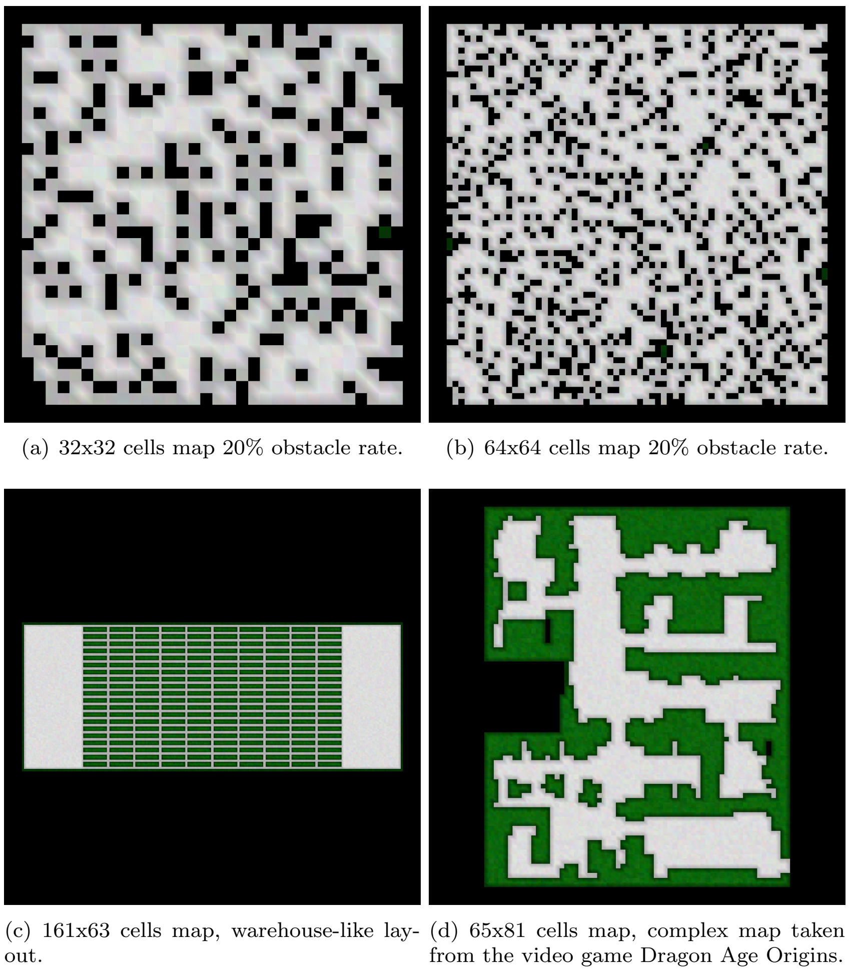
Figure 1.2: **MAPF cluster solving.**

As **the altruistic assumption** cannot be guaranteed, an approach that would more closely resemble real-life selfish agents is required. A possible solution to this problem is a system in which each agent can negotiate during the MAPF problem solving process, affecting its output. These negotiation rounds give agents a chance to obtain solutions that benefit themselves. More details on the negotiation tactics will be given in Chapter 4.

Given the local nature of the proposed solutions, where clusters of agents in close proximity resolve conflicts locally, the **general knowledge assumption** does not hold. Agents only need to share information within their own cluster (at a particular moment in time). This provides increased privacy, and reduces the potential communication load that complete information would generate.

Another consequence of solving MAPF problems for localized clusters is that agents that reach their destination might block the way for other agents, if the paths of others have not been considered. To account for this, agents remain active participants in problem solving until all agents reach their goal. This way, agents are still able to move from their goal location, if necessary, solving deadlocks.

Finally, in order to evaluate the performance and results of the proposed approach, we use the set of benchmarks developed by Nathan Sturtevant [13]. The benchmarks contain a varied set of scenarios, thus allowing us to thoroughly evaluate the algorithms in different environments, ranging from maps of varying sizes with randomized obstacles (Figure 1.3(a), Figure 1.3(b)), to warehouse-like layouts (Figure 1.3(c)), and video game-like complex environments (Figure 1.3(d)).



(a) 32x32 cells map 20% obstacle rate.

(b) 64x64 cells map 20% obstacle rate.

(c) 161x63 cells map, warehouse-like layout.

(d) 65x81 cells map, complex map taken from the video game Dragon Age Origins.

Figure 1.3: **Benchmark map examples [13].**

1.3 Thesis outline

This thesis will be structured as follows: the Background chapter provides information on concepts associated with multi-agent path finding algorithms and other aspects that played a role during research. Following it immediately, the Related Work chapter covers previously done work related to path finding algorithms, that will be built on later.Chapter 4 (Design of the Selfish Localised Pathfinding (SLP) Algorithm) theoretically covers the new research being done, with the Evaluation chapter showing empirical results supporting the theory. These are then followed by the Conclusion chapter, which also covers possible future work and limitations in the research.

Chapter 2

Background

2.1 Single-Agent Path Finding

Single-agent path finding is one of the main requirements for multiple applications, ranging from autonomous vehicles to video games. Before tackling a path finding problem, one must know the components that such a problem is comprised of and different ways in which they can be represented. A 2013 survey by Souissi et al. identifies some important aspects that go into solving such a problem, which will be described in the following subsections [12].

2.1.1 Path Planning Classifications

Souissi et al. classify the multiple types of path planning problems in four levels, going from a high level classification to a low level one. In summary, these are presented as follows:

- **Level 1**
  - *Holonomic problems*: Path finding agents can move in all degrees of freedom, simplifying the problem and ignoring real-life aspects such as velocity and size.
  - *Nonholonomic problems*: Agents cannot move in all degrees of freedom, and additional aspects should be accounted for when path planning, e.g. a car cannot freely spin in place, and certain maneuvers are required for it to turn around.
  - *Kinodynamic problems*: These types of problem more closely resemble a real-life scenario, where agents need to satisfy certain space and velocity requirements.
- **Level 2**
  - *Path planning that requires environment modelling*: Some path finding algorithms, such as A* [10], require a complete model of the environment before path planning can commence.
  - *Path planning that does not require environment modelling*: Some other algorithms, such as Rapidly exploring Random Trees (RTTs) [8], do not require a precomputed environment model to start execution

- **Level 3**
  - *Offline Algorithms*: The environment does not change during path execution.
  - *Online Algorithms*: The environment might change during path execution.
- **Level 4**
  - *Deterministic*: In such algorithms, given the same initial conditions, they will output the same results.
  - *Probabilistic*: In order to satisfy certain time constraints, some algorithms choose to give up deterministic solution in favor of faster computation times, that might not result in optimal results.

2.1.2 Environment Modelling

Souissi et al. also identify two categories in which environment modelling can be classified: methods using cells to model the space and methods using precomputed paths between points to model the environment.

- Using cells
  - Regular Grids: These grids are comprised of identical cells, e.g. squares, triangles, hexagons.
  - Irregular Grids: In this case, the cells constructing the grid are not identical. The benefit of this type of grid comes from the ability to increase grid resolution only in places where this is required (e.g. small obstacles). This comes at the expense of potentially worse solutions when the grid cells are too big, as bigger cell would reduce the resolution at which the environment is modelled.
  - Navigation Mesh: A method of representing a space in a reduced number of polygons, where certain areas require less detail. It is mostly used in video game programming.
- Using precomputed paths
  - Visibility graphs: The environment is modeled by computing the paths between all obstacles corners that are visible from each other. By computing straight edges between each obstacle (and the start and target positions), the new graph can be traversed in order to reach the goal.
  - Voronöı diagram: A new graph is constructed by computing new edges that are at an equal distance to nearby obstacles. This results in a simplified representation that reduces the chance of obstacle collision, but might result in suboptimal paths.

The Selfish Localised Pathfinding (SLP) Algorithm, introduced in Chapter 4, is a deterministic, online algorithm, that solves a holonomic problem, using a modelled environment. In terms of environment modelling, as a requirement of Sioux Technologies, the suggested algorithm will make use of regular grid of cells.

2.2 Single Agent Path Finding Algorithms

Single-agent path finding algorithms are an essential component to finding solutions in a multi-agent path finding context. For the purpose of this research, a SAPF solution, or a path, is defined as a sequence of grid cells that an agent has to traverse to reach its goal. The following two sections describe two SAPF algorithms that have been used while designing the SLP Algorithm.

2.2.1 A*

A* is an algorithm that can be used to find the optimal path through a graph [10]. It traverses the graph starting at a certain node, going neighbour to neighbour, until the goal is found, or until all possible moves are exhausted. A* operates by assigning every expanded node n, three values, $g(n)$, $h(n)$, and $f(n)$. $g(n)$ represents the traversed distance from the start. $h(n)$ represents the **estimated** distance to the goal, and is computed using a heuristic function. In the case of path finding the heuristic function is usually the Manhattan distance between the node and the target location. $f(n)$ is the sum of the two, and is used when selecting the next move with the lowest cost. On top of this, the algorithm keeps track of an Open and Closed queue, with the former keeping track of the possible next nodes to expand, and the latter keeping track of the already expanded notes, as to not expand them again.

2.2.2 PEA*

Partial Expansion A* (PEA*) [16] is an improvement on the A* algorithm. Similarly to A*, it finds an optimal path for a single agent, using two queues, Open and Closed. PEA* aims to reduce the overall runtime of the algorithm by spending less time on node expansion by reducing the number of expanded graph nodes, also reducing memory cost in the process.

The main difference between PEA* and A* is the addition of a forth value that the algorithm keeps track of: the stored cost $F(n)$. Whenever a graph node is first checked, including the start node, its $F(n)$ value is set to its $f(n)$ value.

$F(n)$ is then used to retrieve the node with the smallest cost at every step of the algorithm. When a node is selected, its neighbours get checked. If any of the neighbours' $f(n)$ value is equal to the current $F(n)$, the neighbour gets added to the Open queue. This aims to only add nodes to the Open queue when the solution is nearing the target, i.e if $f(n) = g(n) + h(n)$ moving one step closer to the goal would result in $f(n') = (g(n) + 1) + (h(n) - 1) = f(n)$.

This is not always possible, as lateral or backwards moves might be essential to solving path finding problems. To account for this, the $F(n)$ value of the currently checked node is updated to store the minimal $f(n)$ value of any the nodes neighbours, for values higher than the current $F(n)$. If no such value is found, then the current node is added to the Closed list and considered fully expanded.

By only adding the nodes that approach the goal location to the Open list, the algorithm reduces the number of operations related to the addition/retrieval of nodes from the list, and in consequence runtime, when compared to A*.

2.3 Multi-Agent Path Finding

According to the 2019 paper "Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks" by Sharon et al. multi-agent path finding (MAPF) is "the fundamental problem of planning paths for multiple agents, where the key constraint is that the agents will be able to follow these paths concurrently without colliding with each other" [13]. The paper aims to provide structure to MAPF research, and, as a result, defines a set of notations and classification criteria for MAPF problems.

The survey defines a MAPF problem as a tuple $\langle G, s, t \rangle$, where $G = (V, E)$, with V being a set of vertices and E being a set of non-directional edges connecting them. s and t are functions defined on the domain of participant agents, mapping them to their starting and target positions, respectively. The research summarized in the survey also generally uses a discrete time model, in which time can be separated into time steps, thus allowing agent state to be described as occupying a vertex v at a time step t.

Solving such a problem will produce a MAPF solution. This solution consists of a set of SAPF solutions, each one of them corresponding to a different agent.

Five types of conflicts are formally defined in the paper. These can be seen in Figure 2.1 and can be shortly defined as:

- **Edge Conflict**: two agents cannot traverse the same edge at the same time.
- **Vertex Conflict**: two agents cannot occupy the same vertex at the same time.
- **Following Conflict**: an agent a_1 cannot occupy vertex v at time step $t + 1$ if an agent a_2 occupied it at timestep t.
- **Cycle Conflict**: a set of agents create a cycle of following conflicts, with each occupying a position another one occupied one time step prior.
- **Swapping Conflict**: two agents occupying two adjacent vertices swap places in two consecutive time steps.

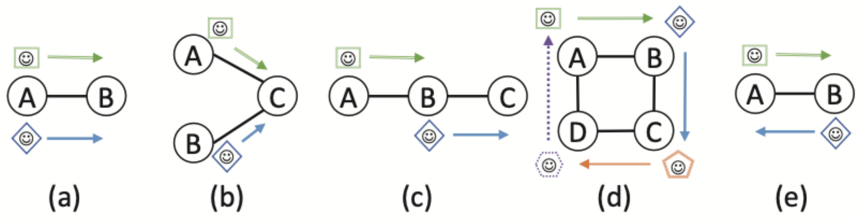


Figure 2.1: **An illustration of common types of conflicts. From left to right: an edge conflict, a vertex conflict, a following conflict, a cycle conflict, and a swapping conflict. Adapted from [13].**

Sharon et al. define the following types of tasks and agents:

- **Classical MAPF**: Each agent has a specific target, which it has to reach without colliding with other agents.
- **Anonymous MAPF**: Given n agents with n targets, any of the agents should reach any of the targets, as long as all targets are satisfied.
- **Colored MAPF**: Agents are separated into teams with their own set of targets, where each team solves an anonymous MAPF problem.
- **Online MAPF**: A sequence of MAPF problems have to be solved in the same graph, as they occur.

All the research presented by this report relies on a discrete time/space model, where agents occupy an entire cell at a time and move instantly from one cell to the next. Nonetheless, this assumption sometimes falls short when real-life considerations have to be considered, and alternatives are worth discussing. Sharon et al. define two MAPF variants that account for such real-life considerations:

- **MAPF with large agents**: Agents, cells and obstacles are considered as entities that occupy more than a point-sized area, and as such new factors have to be accounted for when computing a path.
- **MAPF with kinematic constraints**: Agents movements might be limited by certain physical limitations, and this newly defined set of possible agent actions can influence the found solution.

Another important element listed in the paper is a set of clearly defined benchmarks. These allow for easy comparison and evaluation of algorithms. The benchmarks contain four types of 4-grid connected maps:

- **Open $N \times N$ grids**: maps that have no obstacles of any kind.
- **$N \times N$ grids with obstacles**: maps similar to open maps, where certain moves are considered impossible.
- **Warehouse grids**: maps that try to replicate warehouse environments, with long narrow corridors.
- **Dragon Age Origins (DAO) maps**: maps taken from the video game Dragon Age Origins, which have been made publicly available. These are the largest types of maps and feature complex environments.

Upon reaching their target, agents can display two types of behaviour:

- **Stay at target**: Agents wait at their destination until all other agents finish execution, and become static obstacles throughout future time steps.
- **Disappear at target**: Agents that reach their target "dissapear", and no longer have any effect on the planning of other participating agents.

Given these characteristics defined by Sharon et al. the solution proposed in SLP Algorithm solves the following problem: a classical MAPF problem, in which edge and vertex conflicts are considered. The agents do not disappear once they reach their target, but are still allowed to move once their destination is reached.

2.4 Environment and Agent Characteristics

This research has been done in collaboration with Sioux Technologies Netherlands, as part of their "Maze Solver System". In consequence, some environment and agent characteristics have been set by the company.

The environment is defined as a maze where each cell can be separated by its neighbours by an opaque wall, taller than the agents operating inside the maze. To more closely reflect this real scenario, the developed algorithm will encode obstacles as the costs between two cells in the maze, with a wall being encoded as an arbitrarily high number, considered an impassable edge. This contrasts the usual encoding, where obstacles are defined as cells. This encoding has little impact on how path finding algorithms are implemented, which can be minimally altered to support it, but allows for more variation when creating a maze.

The abstract agent representation used by the path finding algorithm described in Chapter 4 has to satisfy a few requirements imposed by its real-life counterpart. One important characteristic of the maze solving robot developed by Sioux (Figure 2.2) is its ability to rotate around its axis. This allows for sharp turns and quick changes in direction without the need of additional space to perform them. This added degree of freedom eases the transition from an abstract solution to a real-life implementation, without the need for additional algorithmic complexity. Another characteristic of the robot that influences the algorithm is the relationship between its size and the maze. When observing the top down projection of the robot and maze, the robot is (approximately) a circle of a diameter almost as big as the size of a maze cell. Consequently, when solving the path finding problem, this means that no more than one agent can be present in a graph node at any point, thus requiring the algorithm to consider vertex conflicts. Similarly, because of the size of the robot and cells, two robots cannot swap places with each other from neighbouring cells. This creates a need for MAPF algorithms to consider edge conflicts. Vertex and edge conflicts are the only types of conflicts the Sioux implementation considers, with the other conflict types described in Section 2.3 not being accounted for.

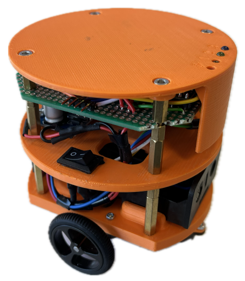


Figure 2.2: **Maze Solver Robot developed by Sioux Technologies B.V. Netherlands.**

Chapter 3

Related Work

3.1 Conflict Based Search

Multiple methods of solving multi agent path finding problems have been studied. The most prevalent among these is Conflict Based Search (CBS) [11]. The main principle on which CBS operates is separating the conflict resolution and the path finding into two levels: the low level and the high level.

The low level handles the path finding for a single agent, using a known algorithm, such as any of the algorithms presented in Section 2.2. These algorithms have to be modified to allow for conflict checking and waiting, two operations that would not be necessary in single agent path finding (in a fixed environment).

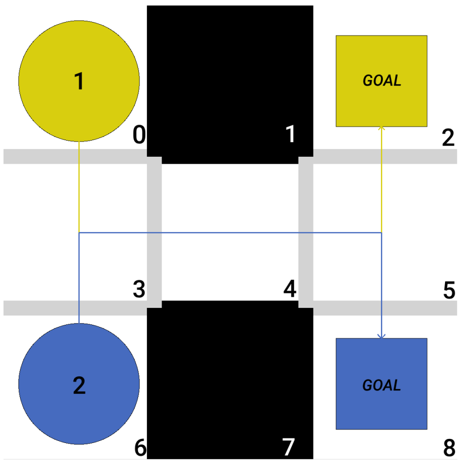
At the high level, CBS defines a data structure called the "conflict tree". This tree stores nodes containing the following information:

- a set of conflicts between agent paths, identified in the parent node and its ancestors
- a set of individually computed paths, each path corresponding to one agent
- a cost, defined as the sum of path lengths of all individual paths

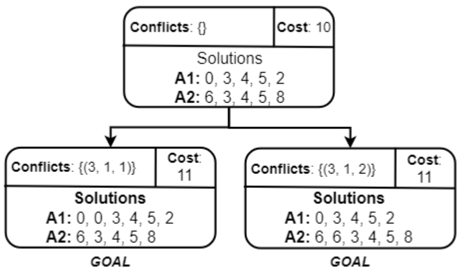
A visual representation of this data structure can be seen in Figure 3.1.

At the first step of the algorithm, a SAPF solution is computed for each agent. These solutions are added to a set, their sum of cost is computed and a conflict tree node is created and added to the tree. On every subsequent step, the node with the lowest stored cost is selected, and its set of individual solutions is searched for Edge or Vertex conflicts, as described in Section 2.3. If no conflict is found, then the solution has been found, and the algorithm returns the set of solutions, with each solution corresponding to one agent.

Otherwise, new SAPF solutions are computed for each agent accounting for the new found conflict as follows, with each agent separately accounting for the conflict. For example, if two agents participate in the problem, two new nodes will be created, with two different solution sets. The algorithm continues this process until a solution has been found or all options have been exhausted, at which point the problem is deemed unsolvable.



(a) Maze state.



(b) Conflict tree of the maze state. Conflicts are noted as {Conflicting Node, Timestep, Agent Solving the Node}.

Figure 3.1: **Conflict tree example.**

Assumptions are essential for CBS to work as described. The following list contains three of the important assumptions that are generally assumed by CBS, with more details about them given in Chapter 2:

- Agents exist and move in a discrete time/space environment. In more detail, the time is divided into "timesteps", in which agents occupy a certain position in a grid-like map.
- Agent moves happen instantaneously between two time steps, i.e an agent occupies a grid cell at timestep t, and it can immediately occupy an adjacent grid cell at timestep $t + 1$.
- A grid cell can only be occupied by one agent at a time.

An example of an environment applying these assumptions can be seen in Figure 3.2.

Even so, the leveled approach of CBS allows modifications to be easily implemented, both in the literature and in the work presented in Chapter 4. The following two subsections present works that describe CBS modifications, aiming to improve the runtime performance of the CBS algorithm, by removing the need for an optimal solution.

3.1.1 Bounded Suboptimal and Enhanced CBS

The 2014 article "Suboptimal Variants of the Conflict-Based Search Algorithm for the Multi-Agent Pathfinding Problem" by Barer et al. [2] aims to decrease algorithm runtime by removing the solution optimality requirement. Compared to CBS, which is proven to obtain the solution that minimizes the chosen cost metric, the algorithms described in the article obtain bounded-suboptimal solutions. In this context, a bounded-suboptimal solution is defined as a solution

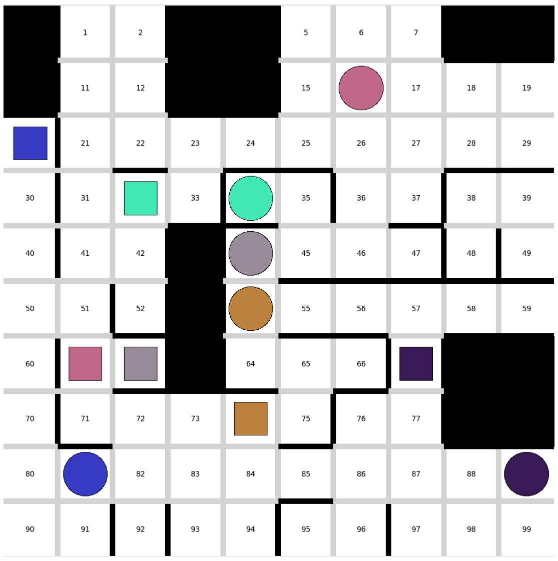


Figure 3.2: **Example of a maze that can be solved using CBS.**

where the cost is at most w times the optimal cost. w is referred to as the sub-optimality bound. Barer et al. present two algorithms: Bounded Suboptimal CBS (BCBS) and Enhanced CBS (ECBS).

BCBS relaxes the optimality of CBS levels separately. It defines the bounds w_L and w_H, for the low level and high level, respectively. These bounds are defined such that $w_L * w_H = w$, this ensures that the final results are bounded by w.

BCBS works by storing elements in an additional queue for both the high and low levels. These additional queues, named Focal queues, store elements from the CBS conflict tree and low-level Open queue, such as the queue described in Section 2.2.1.

A Focal queue stores elements n for which: $f_1(n) \leq w * f_{1_{min}} * f_2$, where f_1 and f_2 are arbitrary functions, and $f_{1_{min}}$ is the minimal f_1 value in the conflict tree or low-level open queue. Using carefully chosen functions [2], searching the Focal queue ensures that if a solution is found, its cost will be at most w times the optimal.

Choosing w_L and w_H is not trivial, and different choices might result in varying algorithm performance [2]. As such, Barer et al. define an algorithm that removes this complexity, by not dividing the suboptimality factors between the high and low levels: ECBS.

Two main aspects differentiate BCBS and ECBS. Firstly, ECBS sets $w_L = w$, allowing the low-level algorithm to perform the boundary estimation. The second difference lays in the way in which the Focal list is generated for the high level.

An additional value is computed when generating each high-level node n: $LB(n)$. This value represents the sum of all $f_{1_{min}}$, generated when performing all of the low-level searches required to generate n. The algorithm then defines the value LB, which stores the minimum $LB(n)$ of all the unchecked nodes in the conflict tree. The Focal list is generated as follows: $Focal = \{n|n \in$

$ConflictTree, n.cost \leq LB * w\}$. Similarly to BCBS, when a solution is found in the high-level Focal list, it is guaranteed to have a cost at most w times the optimal one.

These two algorithms have been evaluated experimentally, and have been proven to provide better runtime performance than an unmodified version of CBS [2]. Tthe proposed algorithm presented in Chapter 4 also provides sub-optimal results because of its local nature. As a consequence, we opted to not use BCBS or ECBS instead of CBS. This ensures that the expected detrimental effects on solution optimality are more easily measured, with no interference from other factors.

3.1.2 Explicit Estimation CBS

The 2021 article "EECBS: A Bounded-Suboptimal Search for Multi-Agent Path Finding" by Li et al. identifies and addresses issues present in the high-level search of ECBS.

In short, it replaces the high-level focal search with a modified version of Explicit Estimation Search [14]. This algorithm uses an additional function to estimate the cost of the children of a node from the CBS conflict tree. This estimate is then used to choose which conflict tree node should be expanded next, as nodes that are likely to generate children with lower cost, are also more likely to generate better final solutions.

While the algorithm provides performance improvement while still satisfying the suboptimality bound defined by ECBS, it remains incompatible with the algorithm presented in Chapter 4 for the same reasons as listed in Section 3.1.1.

3.2 Self-Interested Agents

The main goal of the algorithm proposed in Chapter 4, is solving a MAPF problem in a decentralized manner accounting for self-interested agents, thus removing some of the assumptions listed in Chapter 1. The two following works address these aspects, and have inspired the proposed algorithm.

3.2.1 Decentralized Multi-Agent Path Finding for UAV Traffic Management

The 2022 article "Decentralized Multi-Agent Path Finding for Traffic Management" by Ho et al. proposes an algorithm that allows multiple agents to negotiate conflict free paths [5].

The algorithm looks at every pair of agents, examining the number of conflicts between each pair. Afterwards, sequential pairwise negotiations are used to "resolve" each pair, with the pair with the highest number of conflicts being chosen first. A pair being resolved is defined by creating a conflict free plan for the two agents.

The algorithm proposes two methods of concluding such a pairwise resolution:

- The prioritization approach: the path of the agent who "won" the negotiation is declared as an obstacle for all other agents.

- The negotiation approach: the path of the agent that has the smallest number of conflicts with the other agents is declared as an obstacle for all other agents, as its small number of conflicts is considered to have a smaller impact on the rest of the agents.

Additionally, the paper proposes that the entire set of agents can be separated into multiple independent clusters, that will solve their internal conflicts in a decentralized manner, abiding to cluster specific rules. This is then followed by communication between clusters, where a separate round of negotiation takes place.

While the approach presented by Ho et al. does not directly translate into the solution proposed it inspired the negotiation approach presented in Section 4.2, and how paths can be treated as obstacles by other agents.

3.2.2 Decentralized multi-agent path finding framework and strategies based on automated negotiation

"Decentralized multi-agent path finding framework and strategies based on automated negotiation" by Keskin et al. [6] looks into how a self-interested decentralized solution would perform when solving MAPF problems.

Agents start execution by only knowing their start and target locations, and the path to follow. Throughout the execution the agents check their surroundings for other agents in a regular grid map, checking cells around the agent location. When agents detect each other, they begin a negotiation process, that aims to solve possible conflicts, while at the same time allowing agents to express their interests.

The paper presents two methods of negotiation: "Path-Aware" and "Heatmap". The former uses a bilateral negotiation protocol to generate paths that account for agent preferences, with the latter using the density of the environment (i.e. the number of agents in an area) to generate negotiation bids. The article concludes that when compared to EECBS, in scenarios where agents disappear once they reach their destination, the Heatmap method results in an increase in problem solving rates. These results came at the cost of an increase in the solution suboptimality when compared to the baseline, i.e. longer paths.

Two aspects of this research stand out as undesirable. First, the increase in problem solving rate is present in scenarios where agents disappear once they reach their goal, while this assumption is not applicable to the real world. Second, the Heatmap negotiation approach results in solutions where agents tend to prefer low density regions and thus take detours, without taking into account the self-interested desires of the agents.

Nonetheless, the research done by Keskin et al. inspired the algorithm proposed in Chapter 4 in two ways: Agents starting execution being aware of just their own individual path until a possible conflict arises, and the local negotiation approach between small clusters of agents.

Chapter 4

Design of the SLP Algorithm

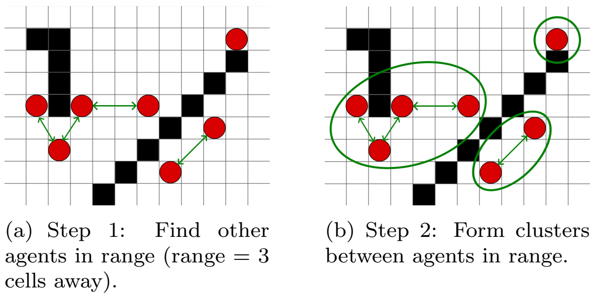
This research proposes an algorithm that solves MAPF problems while lifting the assumptions listed in Chapter 1: the Selfish Localised Pathfinding (SLP) algorithm. It is an algorithm that allows local clusters of neighbouring agents to solve MAPF problems while advancing towards their goal, while also allowing them to negotiate solutions, selfishly affecting the final output. It consists of two steps. Step 1 is presented in Section 4.1 and it handles defining local problems between nearby agents in an environment. Step 2, presented in Section 4.2, solves the local problems, using a modified version of the CBS algorithm that allows the participating self-interested agents to negotiate a better solution for themselves. Section 4.3 describes some of the implementation details relevant to the algorithm.

4.1 Local Conflict Resolution

Traditionally, a centralized MAPF algorithm is used to compute agent paths before execution. This ensures that once execution starts, it will finish without the need to check for additional conflicts, and it is also required for obtaining optimal solutions. Alternatively, we propose a decentralized algorithm that starts execution based on a SAPF solution, solving the conflicts that occur on the go.

Once a SAPF solution is computed for each agent, they start execution. In practice, ideally, this solution could be computed by either the agent itself, given it has enough computational power, or by another entity that the agent communicates with remotely, i.e. using the cloud [15]. While the proposed algorithm has been conceptualised as, and should be able to function as, a distributed algorithm, it has been implemented in a centralized manner for the purpose of this research, because of technical limitations. Details on this implementation can be found in Section 4.3.

The agents move one cell per timestep, checking for other nearby agents in range at every step. This can be done in one of two ways: checking for all agents in a certain range, similar to how radio communication works, or checking for agents in field of view, while still within a certain range, akin to using LIDAR. When agents are detected nearby, clusters are formed between each agent, its neighbours and the neighbours of its neighbours. While this process usually results in small clusters (depending on the used range), in the worst case scenario a cluster that consists of all agents can form. Figure 4.1 shows the cluster forming process when only detecting nearby agents in field of view.


(a) Step 1: Find other agents in range (range = 3 cells away).

(b) Step 2: Form clusters between agents in range.

Figure 4.1: **Cluster forming process.**

After a cluster is formed, a MAPF problem can be defined and solved using Algorithm 1 described in Section 4.2, planning paths from the location of each agent to its target. One option of solving the problem is choosing one agent in the cluster as a leader. This leader then gets the required information from all the agents in the cluster and is responsible for solving the problem (either locally or remotely), and returning the new paths to the other agents. Both the leader and the other agents then replace their remaining paths with the newly obtained ones, and continue execution. The process of moving and solving new MAPF problems is repeated until all agents reach their goal, at which point the whole problem is considered solved.

4.2 Self-Interested CBS

In a real-life scenario, the agents solving a MAPF problem might not be willing to altruistically minimize a global cost function, and might prefer minimizing their own path length.

We propose a modification to the CBS algorithm that allows agents to modify the flow of the algorithm, such that the obtained output is beneficial to them. Each agent starts the execution with a given amount of “negotiation points” or NPs (this amount can be set depending on the environment, as a larger map might require more NPs to influence results). Then, at each CBS Conflict Tree node generation, each agent can use some of its NPs to modify the costs of the nodes in a way that benefits it, i.e. increase the cost of undesirable nodes such that they are selected later, or decrease the cost of desirable nodes to make them more likely to be chosen earlier. In more detail, given the perceived increase in cost in a node caused by conflict solving, each agent offers some of its NPs, altering the cost of a node based on the fact that the lowest cost node always gets selected first. Figure 4.2 shows an example of how the modification would change the CBS conflict tree.

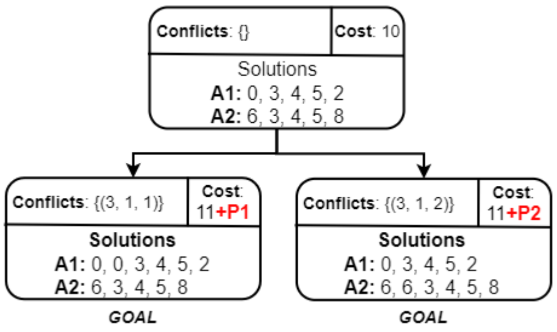


Figure 4.2: **Self-Interested Modification To CBS. The cost in red are the NPs added by the agents. If P1 $\leq$ P2 then the node in which agent 1 (who lost the negotiation by using fewer NPs) solves the conflict by waiting a time step is chosen (thus increasing its cost). Using the $\leq$ operator also accounts for ties, where which solution is chosen is not important.**

We base the amount of NPs on the "influence" a new path has on an agents perceived cost. This influence is computed by comparing the length of the new path to the original SAPF solution computed at the first step of the algorithm and to the current path. Three different paths have been defined:

- The original path: the path computed at the first step of the algorithm, going from the starting position to the target position.
- The current path: the path from the current position of the agent to the target, before a new solution accounting for conflicts is computed
- The new path: the path from the current position of the agent to the target, after a new solution accounting for conflicts is computed

Figure 4.3 shows the paths used when computing the influence.

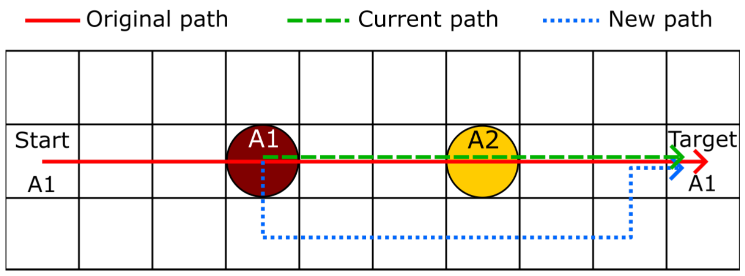


Figure 4.3: **Different paths used when computing the NPs given in a negotiation: the "original path" (solid red arrow), the "current path" (dashed green arrow), and the "new path" (dotted blue arrow).**

We propose two ways of computing the influence, based on the current path and original path, respectively:

$$\textbf{Current path influence} = \frac{\text{Length of the new path}}{\text{Length of the current path}} - 1 \tag{4.1}$$

and

$$\textbf{Original path influence} = \frac{\text{Length of the new path}}{\text{Length of the original path}} - 1 \tag{4.2}$$

The influence takes a positive value if the new path is longer than the current or original path, and a negative one if the new path is shorter. Equation 4.3 shows how this influence gets converted to the NPs used for negotiation.

$$\text{NPs} = \begin{cases} -1 \times \min(\text{A}, |\text{influence}| \times \text{A}), & \text{if influence} < 0 \\ \min(\text{A}, \text{influence} \times \text{A}), & \text{otherwise} \end{cases} \tag{4.3}$$

$$\text{where A} = \text{agent's available NPs}$$

Examining Equations 4.1, 4.2 and 4.3 we reason how the two types of influence affect the flow of the algorithm.

The **current path influence** only accounts for the current and future actions of the agent. It takes positive values when the new path is longer than the current path. This causes the agent to increase the cost of the current conflict tree node, as it wants to delay its selection. If the new path is shorter than the current path (because the current path is not updated to the current state of the environment and might take unnecessary detours), the NP amount is negative, and the agent will decrease the cost of the current conflict tree node, as a shorter path is more desirable.

The **original path influence** attempts to account for the state of the agent in its entire path from start to target, using the new path length as an estimate for the distance to the target. The influence takes a positive value if the new path is longer than the original path. This causes the agent to increase the cost of the current conflict tree node, delaying its selection. If the new cost is smaller than the original, the influence takes a negative value and reduces the cost of the conflict tree node, speeding up its selection. When the new path length approaches zero, i.e. the agent is getting close to its target, the absolute value of the influence will increase, so the agent will negotiate strongly to keep approaching its target. In summary, if a new path is significantly longer than the original, the agent will attempt to significantly delay the conflict tree node selection, and will try to significantly decrease the conflict tree node cost if the new path is significantly smaller than the original. This approach causes the algorithm to punish or reward changes that significantly differ from the original path depending on the situation, only spending NPs when necessary.

As the computed amount of NPs might be too large depending on the path length, e.g. increasing the cost by 1000 for a 10 cell long path, and disproportionately affect the algorithm flow, we propose capping the maximum NPs given during a single negotiation to an arbitrary amount, that can be chosen to reflect the environment, i.e. bigger testing environments might require bigger caps:

$$\text{capped NPs} = \text{sign(NPs)} \times \min(|\text{NPs}|, \text{NP cap}) \tag{4.4}$$

One important thing to note is that during the first step of the SLP algorithm described in Section 4.1, the two influence metrics will act identically, as the original path is identical to the current path. This also allows the described negotiation to be used in a modified version of CBS, that doesn't use the local conflict resolution described in Section 4.1.

The starting NP amount limits the influence a single agent can have on path planning, as the NPs can run out while it is spent for negotiation, thus stopping an agent from further influencing the algorithm. At the same time, multiple agents participate in a negotiation and use their NPs. As the result of a negotiation might not be beneficial to all participants, we define a "winner" of each negotiation: the agent with the lowest influence value out of all the participants when a final solution is found. Once a winner is decided, all the spent NPs are summed into a pool, and is divided between the "loser" agents. The amount of NPs that each agent receives is inversely proportional with the influence of their new path, i.e. the agent that has the largest influence (bigger, thus worse, new path length), will receive the most NPs back from the NP pool. This way, the same amount of NPs stays within the system, and agents that were badly affected by the negotiation have more negotiation power for future interactions. In practice, the leader elected for each cluster, as presented in Section 4.1, can be responsible for redistributing the NPs and informing the other agents about it.

Additionally, alongside returning a set of new paths, the modified CBS algorithm returns a set of conflicts, containing the conflicts that were accounted for in the conflict tree node. These conflicts are remembered by the agents that participated in the negotiation. They are then verified during future negotiations to ensure that future path plans do not go against previously decided constraints. The details of this functionality can also be seen in Section 4.3.

The pseudo-code of the modified CBS algorithm and the NPs calculation can be seen in Algorithm 1 and Algorithm 2, respectively.

Algorithm 1 Self-Interested CBS

```
Require: List of agents, (Optional) list of previous solutions, List of constraints
 1: OPEN ← {}                                              ▷ Stores nodes to be checked
 2: CLOSED ← {}               ▷ Stores nodes that should not be added to open again
 3: node$_{root}$ ← New CBS Node
 4: if SAPF solutions for agents are already computed then
 5:     node$_{root}$.paths ← list of previous SAPF solutions
 6: else
 7:     node$_{root}$.paths ← list of newly computed SAPF solutions
 8: end if
 9: node$_{root}$.cost ← sum of individual path lengths(node$_{root}$.solutions)
10: OPEN ← OPEN ∪ node$_{root}$
11: while OPEN $\neq \emptyset$ do
12:     node ← OPEN.pop()                                 ▷ Get node with lowest cost
13:     conflict ← the first conflict found in the node.paths
14:     if there is no new conflict then
15:         redistributeNPs(agents)
16:         return node.paths, node.conflicts
17:     end if
18:     for agent ∈ list of agents do                ▷ Each agent solves the conflict
19:         newNode ← New CBS Node                                ▷ Allocate the node
20:         // Assign all the values to the new node.
21:         newNode.conflicts ← newNode.conflicts ∪ {conflict}
22:         newNode.paths ← node.paths
23:         newNode.paths[agent] ← SAPF solution handling new conflict
24:         newNode.cost ← sum of individual path lengths(newNode.paths)
25:         if newNode ∈ CLOSED then
26:             continue         ▷ node has the same solutions as a CLOSED node
27:         end if
28:         for agent$_i$ ∈ list of agents do                 ▷ The self-interested step
29:             newNode.cost←newNode.cost + NPsToGive(agent$_i$)
30:             agent$_i$.available_NPs←agent$_i$.available_NPs - NPsToGive(agent$_i$)
31:         end for
32:         OPEN ← OPEN ∪ {newNode}
33:         CLOSED ← CLOSED ∪ {newNode}
34:     end for
35: end while
```

Algorithm 2 Redistribute NPs

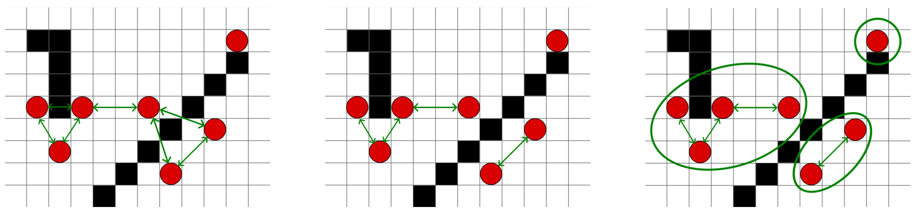
The centralised approach to redistributing NPs. In a decentralised approach the new values should be returned to Algorithm 1 such that they can be communicated to the agents.

```
Require: List of agents
1: NP_sum $\leftarrow$ sum of NPs given by all agents
2: winner $\leftarrow$ agent with the lowest influence
3: agents $\leftarrow$ agents $\setminus\{$winner$\}$
4: influences $\leftarrow$ the influence of each agent $\in$ agents
5: influence_sum $\leftarrow$ sum(influences)
6: for agent $\in$ agents do
7:     influence $\leftarrow$ agent.influence / influence_sum
8:     agent.available_NPs $\leftarrow$ agent.available_NPs + influence $\times$ NPs_sum
9: end for
```

4.3 Implementation Details

While the theoretical concepts presented in Sections 4.1 and 4.2 could be applied in a decentralized manner, for the purpose ease of implementation they were tested and evaluated in a centralised manner, where the state of all agents, their available NPs, their paths and their positions are kept in one place. Additionally, the negotiation is also computed in a centralised manner.

In a decentralized implementation, agents would detect each other, form clusters and elect a leader. For the purpose of our tests, the clusters are formed by checking the state of all agents and forming clusters as such: each agent checks if another agent is located at most d cells away, with the Manhattan distance used as a metric. After the agents in range are found, a Bresenham line [3] is traced between each of them, containing all the cells between the agents. The line is then checked and if an obstacle exists between any consecutive cells, the agents cannot "detect" each other, and they are not considered neighbours. A visual representation of the cluster forming process can be seen in Figure 4.4.



(a) Step 1: Find other agents in range.(range = 3 cells away)

(b) Step 2: Remove connections between agents not in field of view.

(c) Step 3: Form agent clusters.

Figure 4.4: **Implemented cluster forming process.**

While the modified algorithm showcased in Section 4.2 can be easily executed both by a cluster and a centralized computation unit, the way in which the NPs is redistributed is not immediately clear. In a decentralized approach, the leader can compute the amount of NPs each agent should be refunded and inform each agent. In our centralized approach, the algorithm only has to change the stored state of each of the negotiating agents.

Another responsibility of the algorithm is keeping track of which agent negotiated its path with which agent, i.e if two agents negotiated and after one move they are still in range, they shouldn't negotiate again. This is done by keeping track of a map for each agent, storing the number of time steps it shouldn't handle negotiations with each other agent, equal to the path length. Whenever an agent takes part in a negotiation, it resets the values for all the other agents in its map, as the previous commitments are no longer guaranteed. At this point, if agent A1 keeps track of the value for agent A2, and vice versa, their values would be different, as one of them possesses more knowledge about the environment. If the values are different, the two agents should renegotiate, even if they negotiated before.

Chapter 5

Evaluation

5.1 Simulation Setup

The performance of the SLP algorithm was evaluated using multiple benchmarks, comparing it to the state of the art. This evaluation uses a subset of the benchmarks defined by Nathan Sturtevant [13], a subset used for multiple works of research regarding MAPF [13]. These benchmarks define various environments for testing, and provide multiple scenarios (start/target location combinations). One benefit that they provide is they ensure the problems are solvable. This comes at the cost of a reduced number of scenarios, and, as a consequence, a possibleh increase in the variance of the data points obtained while testing. Even so, the tests relating to each map use the same scenarios, and the relation between the obtained results can still be used to draw conclusions about the evaluated algorithms.

Three metrics were used to evaluate the different algorithm variants: the algorithm success rate of solving each test case, the average increase of path length, and the runtime of the algorithm.

The algorithm success rate is based on all the agents reaching their target (i.e. moving from their start to their goal as described in Chapter 4) in a certain amount of compute time. Since generating the maze data structure takes a non-negligible amount of time, the time is only measured after the generation of the maze is completed. The increase in path length for each agent is obtained by comparing the number of steps taken/waited required to finish a MAPF problem to the length of the path obtained by an optimal SAPF solution from its start location to its target, that ignores all other agents. While not ideal, the SAPF solutions were used as a comparison because optimal MAPF solutions (computed by the CBS algorithm) could not be feasibly computed, as obtaining the solutions would require computation time outside of scope of this research. The runtime shows the time spent on solving MAPF problems. In the case of centralized algorithms, the runtime will represent the average time the “difficult” (i.e. accounting for a lot of agents) MAPF problem required, while in the case of decentralized it will show the average time required to solve each “simpler” (i.e. accounting for small clusters of agents) MAPF problem.

In order to test the performance of the algorithms they have been implemented using C++, with the simulation handling the maze and agent state being implemented in Python. The tests were run on a PC using a 13th Gen Intel®Core™i7-13850HX 2100 Mhz processor and 32GB of RAM.

The C++ functions implementing CBS are wrapped and used as a Python library by the simulation. The Python simulation is responsible for storing the state of the maze (e.g. agent location, agent planned paths), handling the nearby agent detection and moving agents through their planned path.

The evaluation presented in this chapter allocated 30 seconds for solving each case. The simulation was ran on 20 different agent start/target locations, for a varying number of agents, e.g. test case 1 for 1 agent, test case 2 for 1 agent, ..., test case 1 for 2 agents, etc.. The algorithms were tested on multiple environments. The obtained results show similar trends on all tested maps, and consequently, this research will only present results related to the following three relevant cases: a 32x32 map with 20% of the space occupied by randomized obstacles (Figure 1.3(a)) representing a small, but reasonably uncrowded map, a 65x81 map from the video game Dragon Age Origins (Figure 1.3(d)) representing a large uncrowded space with a complex topography, and a 161x63 warehouse-like map Figure(1.3(c)), a crowded map mainly consisting of tight corridors.

The SLP algorithm also requires the configuration of a few parameters. For these tests, the range used for cluster formation described in Figure 4.4 was 3, i.e. agents could be detected in cells to which the Manhattan distance was at most 3. For negotiation, the agents start algorithm execution with 100,000 negotiation points (a large number that would result in the NPs rarely running out), with a negotiation point cap of 1,000, i.e. they give out at most 1,000 units per negotiation step.

This chapter will be structured into four parts, in which the proposed algorithms will be compared to algorithms that are part of the state of the art: an Unmodified version of the CBS algorithm (**CBS**), and the ECBS algorithm, using a suboptimality bound of 1.1(**ECBS 1.1**). The first two parts showcasing the influence of the two components of the SLP algorithm will be presented separately, a decentralized MAPF solver with no negotiation (**Local-CBS**) and a centralized algorithm allowing for self-interested agents (**CBS-with-negotiation**), respectively. The third part presents the effect lifting all assumptions has on MAPF problem solving, showing the results obtained by evaluating SLP in its entirety, looking at the two negotiation tactics described in Chapter 4 separately: **SLP with current path negotiation** and **SLP with original path negotiation**. The chapter ends with a evaluation of performance of each of the presented algorithms in terms of run time.

5.2 Local Conflict Resolution

The most important aspect of the research, and the first step of the SLP algorithm, is lifting the assumptions of a centralized solver, and as such, it is important to verify how it affects the performance of the algorithm. Figure 5.2 shows how a Local-CBS compares to CBS and ECBS 1.1. It is immediately clear that in terms of solving rate, Local-CBS outperforms CBS, but it is still clearly outperformed by ECBS 1.1, which can handle larger problems more easily. This is expected, as ECBS 1.1 and Local-CBS provide bounded suboptimal and unbounded suboptimal solutions, respectively. Even though Local-CBS is

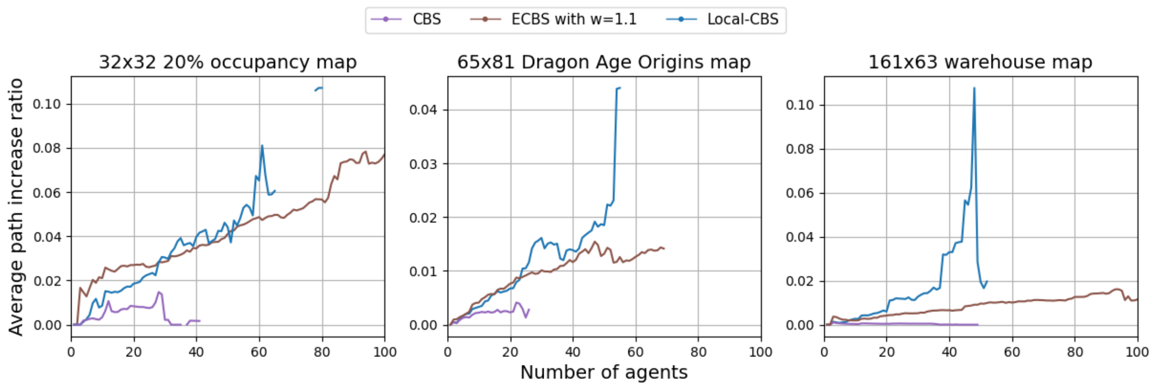


Figure 5.1: **Average path length increase ratio for CBS, ECBS 1.1, and Local-CBS.**

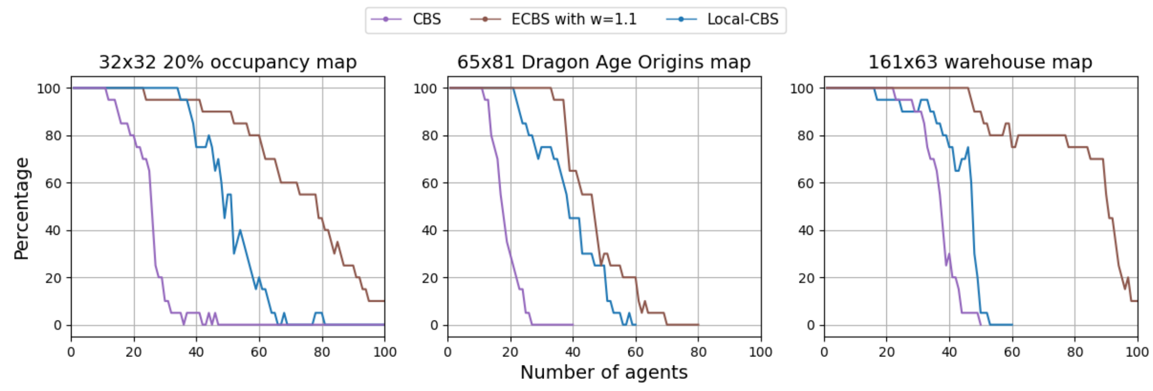


Figure 5.2: **Success rates CBS, ECBS 1.1, and Local-CBS.**

an unbounded algorithm, it has a lower solving rate than ECBS 1.1. This can be caused by its lack of information necessary to solving problems caused by its local nature, thus resulting in a longer time required to solve problems.

It is also important to see how the solution quality compares for the three approaches. Figure 5.1 shows the average path length increase ratio per agent for the three approaches. The smallest increase is noticed in CBS, as expected, as it outputs optimal solutions. ECBS 1.1 shows a higher, but steady and predictable increase, which is guaranteed by its bounded suboptimality. Local-CBS shows the highest cost increase, as the lack of complete information provides no way of bounding the cost increase. The largest difference can be observed on the warehouse map, where it reaches approximately 10% increase in cost. While this fits in the 1.1 suboptimality bound set for ECBS 1.1, the increasing trend is only stopped by the algorithm not being able to solve more complex cases, and would have probably continued to increase, if the algorithm was given more time to run.

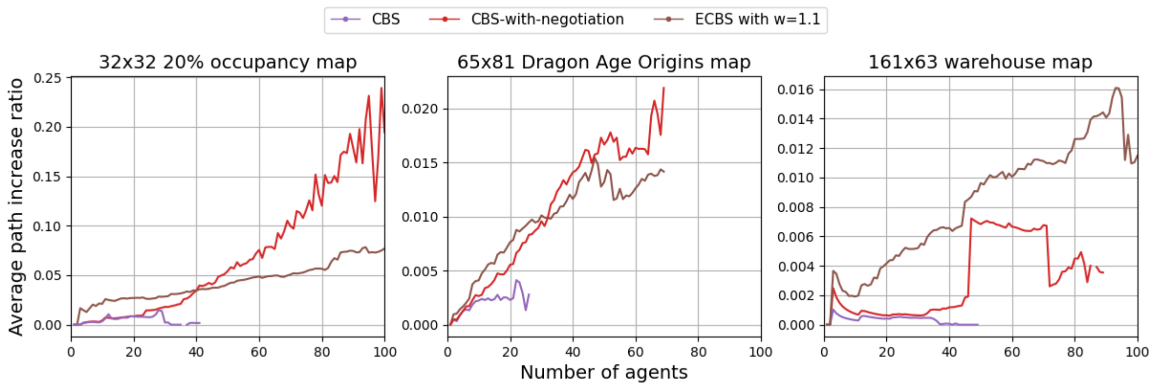


Figure 5.3: **Average path length increase ratio for CBS, CBS-with-negotiation, and ECBS 1.1.**

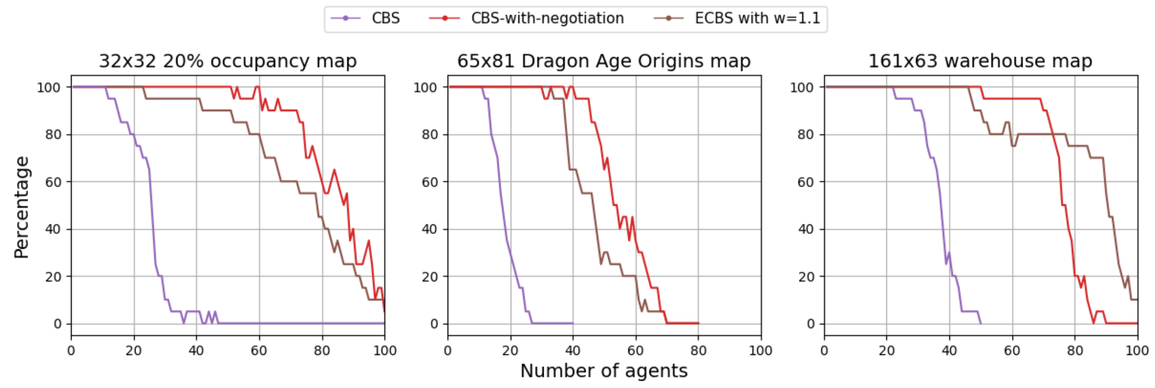


Figure 5.4: **Success rates CBS, CBS-with-negotiation and ECBS 1.1.**

5.3 Self-Interested CBS

After evaluating an altruistic decentralized algorithm, a centralized self-interested algorithm has been evaluated. This modification lifts the assumption that all agents participating in a MAPF problem are willing to collaborate in order to minimize a global cost function.

Making the agents self-interested and adding negotiation to the algorithm removes the optimality guarantee of the algorithm, as the usual execution flow is altered. This is immediately visible in Figure 5.3. Adding negotiation to the CBS algorithm significantly increases the average cost incurred by the participating agents when compared to CBS. When compared to ECBS 1.1, the algorithm performs differently given different domains. In the case of a small 32x32 map, ECBS 1.1 outperforms CBS-with-negotiation by a wide margin and in the case of a larger map, CBS-with-negotiation is still outperformed by a smaller margin. An interesting result appears when examining the warehouse case: the CBS-with-negotiation algorithm greatly outperforms ECBS 1.1 in terms of path length, even when accounting for when the success rate difference between the two is not significant. One potential explanation for this phenomenon is the fact that CBS-with-negotiation uses an optimal low-level solver, thus finding

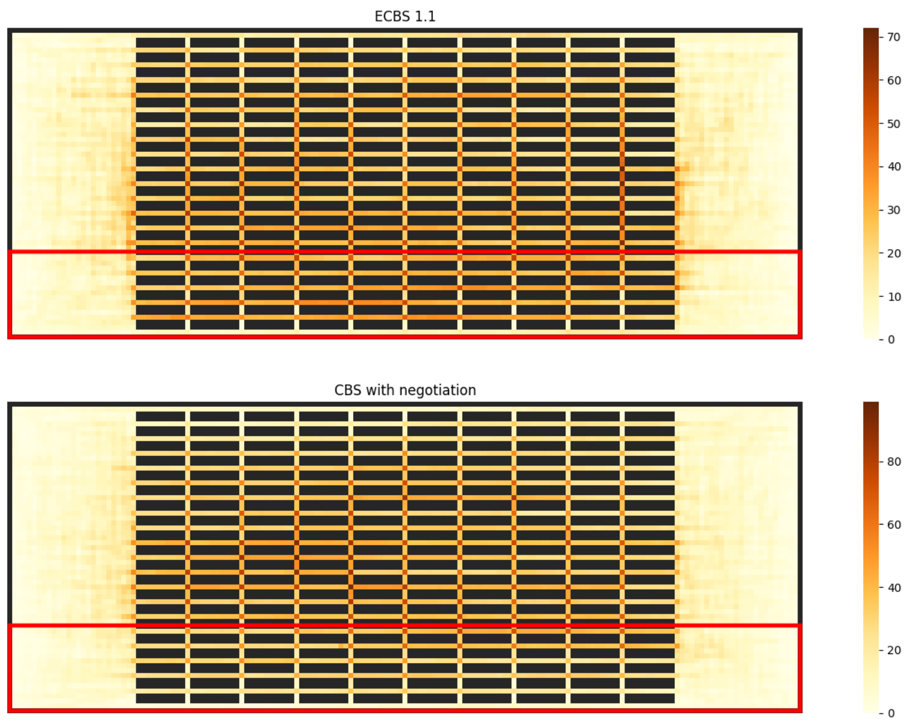


Figure 5.5: **Heatmaps showing how agents using ECBS 1.1 and CBS-with-negotiation traverse the warehouse map, in cases solving problems for 60 agents. One area where differences are visible is highlighted by a red rectangle.**

shorter solutions. Figure 5.5 supports this hypothesis. For agents using ECBS 1.1 as a solver, the steps taken are more evenly distributed throughout the map, with the CBS-with-negotiation solutions traversing the central area of the map more than its edges (this difference is the most visible in the lower parts of the map).

In terms of success rate in the given time frame, CBS-with-negotiation manages to solve a significantly larger number of cases compared to CBS. This is expected, as CBS-with-negotiation lifts the optimality requirement, thus spending less time on finding solutions. ECBS 1.1 and the modified CBS algorithm perform similarly, with ECBS 1.1 performing slightly worse for a higher number of agents in the warehouse map. One possible reason for this is the crowded nature of the map. While Figure 5.3 shows that the modified version of CBS obtains better results in terms of path length because of the low-level algorithm used, the same algorithm might cause the lower success rate. As all agents might try to take routes closer to optimal, the algorithm might require multiple negotiation rounds before finding a solution, passing the time limit and reducing the success rate. Opposite to this, ECBS uses a suboptimal low-level algorithm, finding suboptimal paths faster. Figure 5.4 shows the success rates of the three examined algorithms.

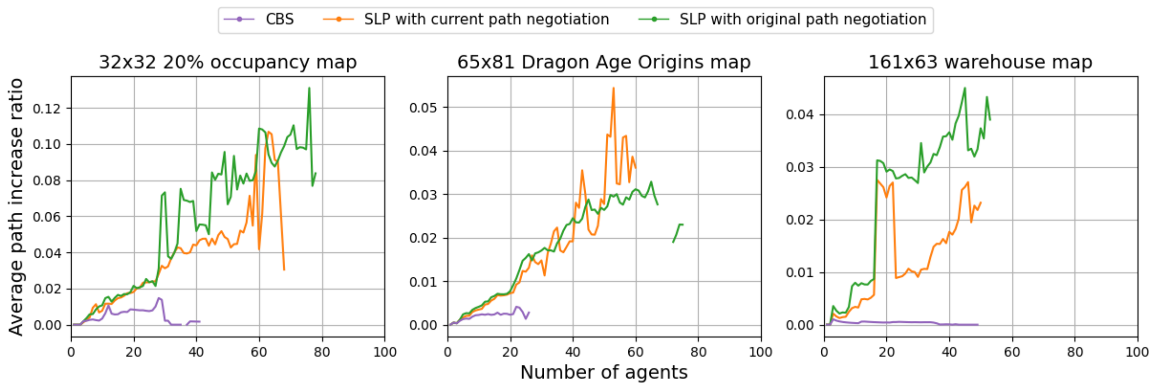


Figure 5.6: **Average path length increase ratio for CBS, SLP using current path negotiation, and SLP using the original path negotiation.**

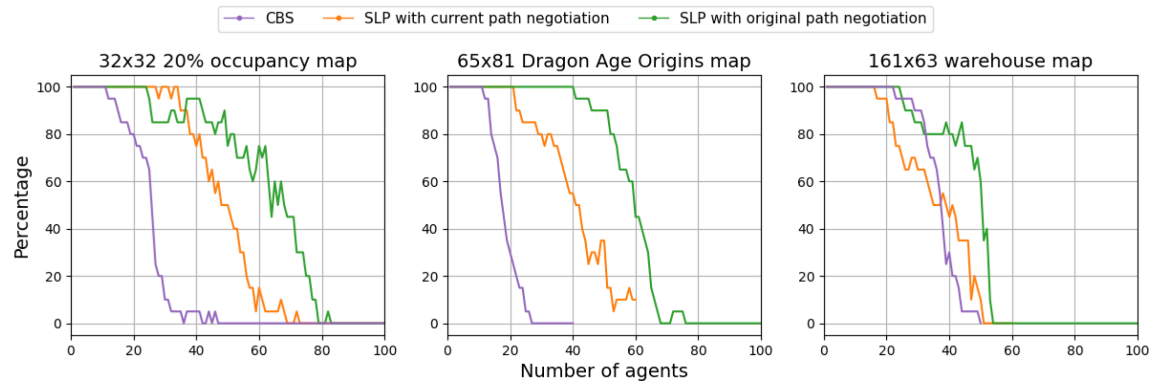


Figure 5.7: **Success rates CBS, SLP using current path negotiation, and SLP using the original path negotiation.**

5.4 SLP Algorithm

The next step is evaluating the SLP algorithm, which combines the elements evaluated in Section 5.2 and 5.3. We analyze how the two types of negotiation perform in a decentralized algorithm, comparing them to CBS.

Figure 5.6 shows the average path length increase for the compared algorithm. The figures show the two decentralized algorithms obtain longer paths in general. This can be explained by the algorithms providing solutions using incomplete information, as not knowing the paths of all agents might result in routes being planned through crowded areas, thus requiring more rounds of negotiation and rerouting. Similarly to Section 5.3, the warehouse environment produces results that stand out. While in the other environments the two variants perform similarly, a large difference can be observed for the warehouse environment, starting close to the 20-agent mark. One possible explanation for this is offered by success rate visible in Figure 5.7. Around the same number of agents, the success rate of the algorithm using current path negotiation starts to go down. This can possibly signal that the algorithm can only solve "simpler" problems, that result in shorter average paths, explaining the difference. This mirrors the contradictory trend observed in path length increase in Section 5.3, reinforcing that the particularly crowded nature of the map has an influence on the results. Another conclusion that can be drawn from the results in Figure 5.6 and 5.7 is that using original path negotiation generally results in better metrics, indifferent of the map, and it is likely preferable as a general solution.

5.5 Timing analysis

The last evaluation was done on time required by each examined algorithm to solve MAPF problems. We separate these results into two groups, the centralized variants and the decentralized variants.

The centralized algorithms compute just one MAPF problem, creating a plan that is then executed without the need for more planning. Figure 5.8 shows how they perform in terms of time required to solve the problems.

As expected, the CBS algorithm has the steepest increasing slope, going towards the 30 second hard limit the fastest (with the immediate stagnation being explained by the algorithm not being able to solve more complex problems). When comparing ECBS 1.1 with the CBS-with-negotiation, they perform similarly, with no clear distinction of a "better" algorithm, as they outperform each other in different domains.

The decentralized approach computes multiple small MAPF problems, while agents advance towards their destination. Because of this, the timing evaluation for the decentralized approach showcases how well the algorithms would perform in an online scenario, where the agents stopping execution for long periods of time to recompute paths would be undesirable. Figure 5.9 shows these results.

A clear trend is apparent, SLP using original path negotiation performs better than the other versions on all environments. Additionally, its performance is more stable compared to the other two variants, that show significant spikes in runtime. These two aspects, combined with the previous results, make it the best candidate for a local self-interested algorithm.

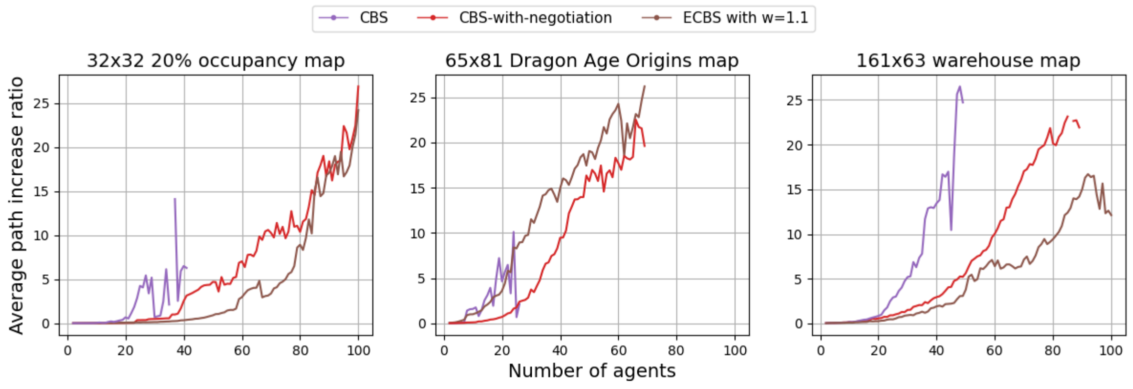


Figure 5.8: **Timing for centralized algorithms.**

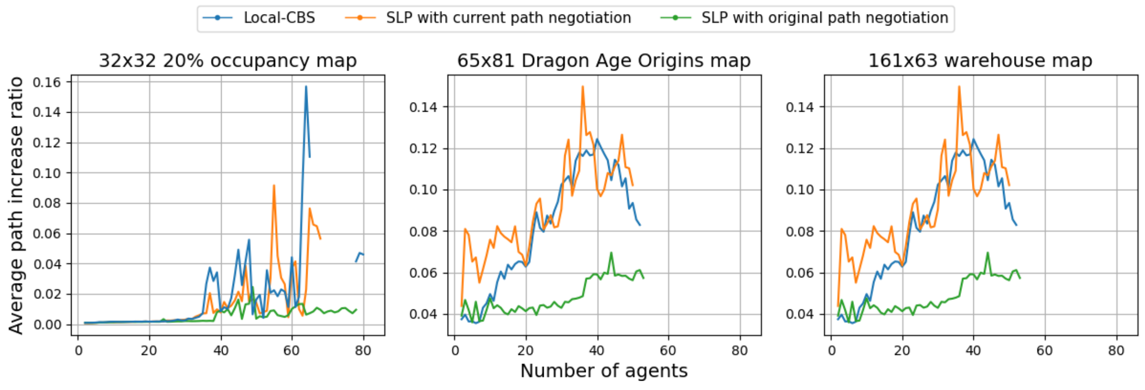


Figure 5.9: **Timing for decentralized algorithms.**

Chapter 6

Conclusions

6.1 Conclusions

Path finding is an important component in solving various problems, ranging from automated warehouse management to autonomous vehicles. These types of problems require additional considerations when compared to normal path finding, as multiple agents traveling to their target result in a dynamic and complex environment. To account for these complexities, the state of the art in multi-agent path finding algorithms uses various assumptions, simplifying the problem.

The aim of this work is lifting some of the assumptions used by other MAPF solvers. It lifts the assumption of a centralized solver, that can use information about all participating agents to produce MAPF solutions, by replacing it with a decentralized solver, where agents only use information about other agents in their proximity to solve problems. It also lifts the assumption that all agents participating in a MAPF problem are willing to obtain solutions that minimize a global cost. In a real-life scenarios agents might be selfish, and, as a consequence, the altruistic approach to cost was replaced, allowing agents to negotiate solutions that benefit them at the expense of others. These assumptions were lifted with the purpose of creating an algorithm that more closely fits real-life conditions would perform, with the goal of answering the research question presented in Chapter 1:

How can the assumptions listed in Chapter 1 be lifted while still solving multi-agent path finding problems? How will lifting them affect path-finding performance?

The SLP algorithm lifts all assumptions listed in Chapter 1. It is a decentralized algorithm, in which agents can express their interests at various points during execution, affecting their path. SLP is comprised of two important components. The first component defines a decentralised approach to path finding. Nearby agents can form clusters for which MAPF problems are solved, as opposed to accounting for all agents in the environment. The second component of SLP allows agents that participate in the path finding process to negotiate with each other, affecting the results. Agents can use their "negotiation points" to avoid solutions that would result in an increase in path length for themselves, selfishly passing on the cost to other agents with less negotiation power. Separately, this components result in an increase in problem solving success rate, which comes at the cost of an increased average path length. Using both components of SLP together combines their strengths and weaknesses, and results in solutions that lie between the solutions obtained by using its components separately. One important thing to note is that using the second component alone, in a centralised algorithm, can be treated as a single negotiating cluster. As such, the differences between the two components can indicate that clusters of varying sizes might need further research, aiming for a "sweet spot" between the solving speed provided by small clusters and the information provided by larger clusters.

Nonetheless, the timing analysis of the decentralized algorithms (Figure 5.9) shows that the SLP algorithm using original path negotiation is a good fit for an online algorithm, managing to compute paths for small clusters in a very short amount of time. This solution comes at the cost of an increase in the average path length of the agents. The proposed algorithm offers solutions within 3% and 4% of the SAPF path length for the Dragon Age Origins and warehouse maps, respectively. It creates slightly worse solutions for the 32x32 map with randomized obstacles, within 12% the SAPF solution. Despite the slight increase in path length, SLP using the original path greatly outshines CBS in terms of the percentage of solved problems (Figure 5.7), while also providing an algorithm applicable to a decentralized context.

6.2 Future Work

One limitation of the results presented in this research is the lack of a deep analysis of some of the algorithm parameters. The evaluation presented in Chapter 5 uses a negotiation points cap of 1,000 and a maximum amount of negotiation points of 100,000, with the agents using a detection range of 3 units. These parameters can be explored further, as they affect each negotiation step, and the whole test, respectively. For example, the maximum NP value of 100,000 results in scenarios where agents are not likely to run out of NPs, so test cases where agents have a stricter negotiation budget might offer new insights. Another limitation in the analysis is the small sample size of the benchmarks, as generating more test cases will provide a more accurate image of the performance of the proposed algorithms.

One other possible avenue of future research is a more thorough evaluation of other negotiation strategies. While this research proposes two metrics, other ways of computing the negotiation point amount agents are willing to trade can be developed. For example, the original path influence metric can be modified to use a changing value instead of the original path, reducing the value by 1 every time a step is taken by the agent, estimating where the agent "should" be.

Additionally, the heatmaps shown in Chapter 5 (Figure 5.5) exposes another research direction. Different algorithms will result in different types of solutions, which might traverse the environments in different ways. This means that certain algorithms are a better fit for certain environments, and how to choose an algorithm should be studied.

Finally, while the SLP algorithm is conceptualized to run in an online manner, the simulation does not fully account for that: the time steps and moves are considered instantaneous, and the time each clusters takes to solve conflicts is not accounted for. The timing analysis presented in Figure 5.9, shows that negotiations are fast, and the SLP algorithm could fit the required role, but additional testing/simulation would be required to translate it into a fully applicable real-world solution that accounts for physical constraints (e.g. size, wheel slippage) and other requirements, such as designing communication protocols and limitations regarding implementing the algorithm on an embedded system. As a consequence, while a prototype has been developed by Sioux (Figure 2.2), no testing was done using real hardware, which would showcase potential problems in the presented approach.

Bibliography

[1] Max Barer, Guni Sharon, Roni Stern, and Ariel Felner. Suboptimal Variants of the Conflict-Based Search Algorithm for the Multi-Agent Pathfinding Problem, 2014.

[2] Max Barer, Guni Sharon, Roni Stern, and Ariel Felner. Suboptimal Variants of the Conflict-Based Search Algorithm for the Multi-Agent Pathfinding Problem, 2014.

[3] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.

[4] Freepik. Free Vector — Automatic Logistics Delivery isometric composition. `https://www.freepik.com/free-vector/automatic-logistics-delivery-isometric-composition_6123995.htm#fromView=search&page=1&position=22&uuid=e76692d8-2930-4dd5-9c2f-c0c65a92bb69`.

[5] Florence Ho, Ruben Geraldes, Artur Goncalves, Bastien Rigault, Benjamin Sportich, Daisuke Kubo, Marc Cavazza, and Helmut Prendinger. Decentralized Multi-Agent Path Finding for UAV Traffic Management. *IEEE Transactions on Intelligent Transportation Systems*, 23:997–1008, 2 2022.

[6] M. Onur Keskin, Furkan Cantürk, Cihan Eran, and Reyhan Aydoğan. Decentralized multi-agent path finding framework and strategies based on automated negotiation. *Autonomous Agents and Multi-Agent Systems*, 38, 6 2024.

[7] Joshua Laber, Ravindra Thamma, and E Daniel Kirby. The Impact of Warehouse Automation in Amazon's Success, 2020.

[8] Steven M. LaValle. Rapidly-exploring random trees : a new tool for path planning. *The annual research report*, 1998.

[9] Jiaoyang Li, Wheeler Ruml, and Sven Koenig. EECBS: A Bounded-Suboptimal Search for Multi-Agent Path Finding. 10 2020.

[10] M J D Powell, S Lasdon, J D Schoeffler, B P Dzielinski, R E Gomory, Peter E Hart, Nils J Nilsson, and Bertram Raphael. The gradient projection method for nonlinear programming, pt. I, linear constraints. *Research Analysis Corp*, 19:874–890, 1968.

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

[12] Omar Souissi, Rabie Benatitallah, David Duvivier, AbedlHakim Artiba, Nicolas Belanger, and Pierre Feyzeau. Path planning: A 2013 survey. In *Proceedings of 2013 International Conference on Industrial Engineering and Systems Management (IESM)*, pages 1–8, 2013.

[13] Roni Stern, Nathan Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, T. K. Satish Kumar, Eli Boyarski, and Roman Bartak. Multi-agent pathfinding: Definitions, variants, and benchmarks. 6 2019.

[14] Jordan T Thayer and Wheeler Ruml. Bounded suboptimal search: a direct approach using inadmissible estimates. pages 674–679, 07 2011.

[15] Maksym Vette. Serverless cloud based Automated Guided Vehicle control. *repository.tudelft.nl*, 2023.

[16] Takayuki Yoshizumi, Teruhisa Miura, and Toru Ishida. A* with Partial Expansion for large branching factor problems, 2000.