POLITECNICO DI MILANO
Scuola di Ingegneria Industriale e dell'Informazione
Dipartimento di Elettronica, Informazione e Bioingegneria

POLITECNICO
MILANO 1863

IMPLEMENTATION OF ALGORITHMS FOR SOLVING MULTI-AGENT PATH FINDING PROBLEMS

Supervisor: prof. Francesco AMIGONI

Master thesis by:
Matteo ANTONIAZZI, ID 895712

Academic year 2018-2019

Abstract

Multi-Agent Path Finding (MAPF) is the problem of computing collision-free paths for a team of agents from their current locations to given destinations. Application examples include automated warehouse systems and video games.

Several different MAPF algorithms have been presented during the last years, each of which has its pros and cons.

In this thesis, we present a tool implementing some of the most used algorithms for solving the MAPF problem. The basic idea is to implement these algorithms in a common scholar programming language, such as Python, which gives the user the possibility to easily understand the key ideas behind these algorithms, allowing him/her to analyze and compare them in different environments and robots configurations.

We carry out some experimental activities in order to validate our implementations and compare the different algorithms.

Sommario

Multi-Agent Path Finding (MAPF) è il problema della pianificazione di percorsi senza collisioni per un team di agenti dalle loro posizioni attuali alle loro destinazioni. Esempi di applicazione includono sistemi di gestione di magazzini automatizzati e videogiochi.

Diversi algoritmi MAPF sono stati presentati negli ultimi anni, ognuno dei quali ha i suoi pro e contro.

In questa tesi, presentiamo uno strumento che implementa alcuni degli algoritmi più utilizzati per risolvere il problema del MAPF. L'idea di base è implementare questi algoritmi in un comune linguaggio di programmazione accademico, come Python, che offra all'utente la possibilità di comprendere facilmente le idee chiave alla base di questi algoritmi, consentendogli di analizzarli e confrontarli in diversi ambienti e configurazioni di robot.

Effettuiamo alcune attività sperimentali al fine di validare le nostre implementazioni e confrontare i diversi algoritmi.

Acknowledgments

I would like to thank my supervisor, Professor Francesco Amigoni, for giving me the possibility to work on this topic and for the assistance he provided me during the development of this work. He also had increased a lot my interest in the Artificial Intelligence field. A special thanks to my family for giving me the opportunity to complete my studies and for their presence and support during all these years. A thanks to all my classmates for the time spent together and for making my school career more enjoyable. And thanks also to all my closest friends for the encouragement and for helping me take my mind off in the most stressful periods.

Contents

Chapter 1

Introduction

The *multi-agent path finding* (MAPF) problem involves a graph and a number of agents. For each agent, a unique start location and a unique goal location are given, and the task is to find for all agents a path from their start location to their goal location with no collisions with obstacles or other moving agents. The paths must be found by minimizing a cumulative cost function (e.g., sum of path costs or maximum path cost). MAPF has practical applications in a lot of relevant modern applications including robotics [1], automated warehouse systems [9, 2], aviation [11], traffic controls [3, 14], video games [10], etc.

The purpose of the thesis is to implement a set of solving algorithms for the MAPF problem in a scholar programming language. We have used Python since it is simple and easy to understand. Our tool has been developed with the idea that it can be useful for those who want to have a general idea on how these algorithms work and on the solutions they generate. In fact, given a map defining the environment, agents configurations, and an algorithm, our tool generates the relative solution of the MAPF problem, with a graphic animation on the user interface.

The first step done has been to represent the MAPF problem, and implement a simple version of each one of the algorithms. Then, a Graphic User Interface has been implemented in order to have a visual representation of the problem. The third step has been adding to the algorithms some features and improvements, like the possibility to minimizing different objective functions, or choose if the agent remains at goal once reached it or it can disappear. In the end, some experimental activities have been carried out in order to check that the algorithms work in the correct way, and a simple comparison between them has been performed.

The thesis is structured as follows. In Chapter 2 we present the problem

definition and terminology. The MAPF framework we have implemented in Python is described in Chapter 3. Chapter 4 presents all the algorithms addressed by this thesis, giving a general idea on how they work (with reference to the corresponding papers). In Chapter 5 we discuss how those algorithms work in details and how they have been implemented in the Python language. The package structure and GUI implementation are presented in Chapter 6. In Chapter 7 experimental results are shown and commented. Chapter 8 concludes the thesis and suggests possible uses of the tool we presented.

Chapter 2

Problem Definition and Terminology

In this chapter we define the multi-agent path finding problem in general, and we discuss some of the approaches for solving it and some of the possible variants of the problem. We also include some basic terminology.

2.1 MAPF Definition

The input to a classical MAPF problem are a graph $G = (V, E)$ and k agents labeled $a_1, a_2, \ldots a_k$. The vertices V of the graph represents possible locations, and the edges E are the possible transitions between locations. Every agent a_i has a start vertex, $start_i$, and a unique goal vertex, $goal_i$.

Time is assumed to be discretized. At time step t_0 agent a_i is located at $start_i$. At every time step every agent is located in one of the graph vertices and can perform a single *action*. Each agent has two possible types of actions: wait or move. A *wait* action means that the agent stays in its current location for another time step. A *move* action means that the agent moves to an empty neighboring location. The maximum number of locations that a single agent can move to in one time step is defined by the branching factor, b_{base}. This thesis focuses on 4-connected grid, where $b_{base} = 5$, since every agent has 5 possible actions (*N, E, S, W,* or *wait*). Variants where agents can also move in diagonal direction ($b_{base} = 9$) exists, but are not treated here.

The overall goal of MAPF solvers is to find a *solution*, i.e., a plan for each agent, that can be executed without collisions. A *plan* for an agent is a sequence of actions that, when performed by the involved agent, they lead it at its goal position. In order to achieve this, a solver need to be able to

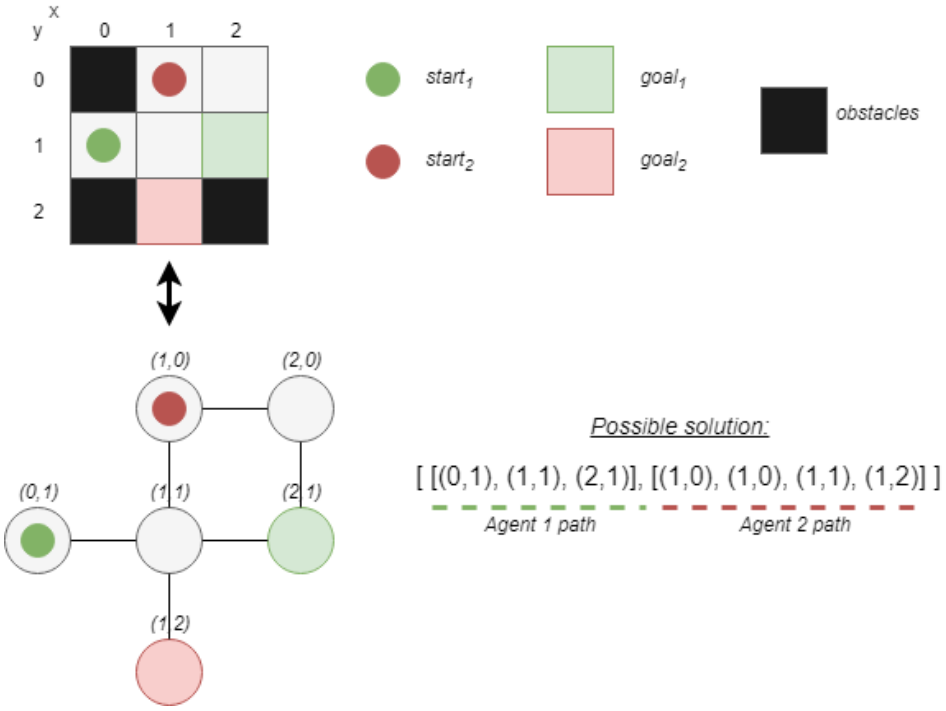


Figure 2.1: An example of a MAPF problem with two agents.

identify and resolve possible conflicts. The definition of what constitutes a conflict is discussed in Section 2.2. A solution is called *valid* iff there is no conflict between any two single-agent plans.

Figure 2.1 shows an example of a 2-agents MAPF problem. Both the map version and the graph version of the environment are shown, and the solution of problem is represented as list of agent positions.

2.2 MAPF Variants

As discussed in this work [16] exists various research papers that address the MAPF problem, and those consider different sets of assumptions about agents or objective of the problem. In this section we discuss all the assumptions made in our thesis.

Regarding the agent behaviour at its goal location, specifically in the time steps after it has reached its target and before the last agent has reached its target, we deal with all the possible assumptions.

- **Stay at goal**. Under this assumption, when an agent has reached its goal it waits and occupies this location as long as other agents are still travelling.

- **Disappear at goal**. Under this assumption, when an agent has reached its goal it disappears. We also add the possibility to specify a time that agents need to spend in the goal before disappearing. We called it *goal occupation time*, and range from 1, which indicate that the agent occupies its goal location only during the time step at which it arrives, to any positive integer (e.g., if *goal occupation time* = 3, means that the agent will have to stay at its goal for 3 time steps before disappearing and freeing that location).

In both cases wait actions at the goal cost zero. Note that if an agent has reached its goal, but later on is forced to move away, this might cause an increase in the total cost, since actions at the goal have zero cost only if the agent arrives without moving away later. Most prior work on classical MAPF assumed stay at goal [14, 15, 13, 12, 19], but recent work also considered the disappear at goal assumption [7].

The literature on MAPF includes several different definitions of what constitutes a conflict between plans. In this thesis we consider two types of conflicts.

- **Vertex conflict**. A *vertex conflict* occurs when two agents occupy the same location at the same time step. So, it is not possible for agents to share a vertex. This type of conflict is the basic type of conflict, and it is always checked during planning.

- **Edge conflict**. An *edge conflict* occurs when two agents pass through each other, or, more simply, when two agents traverse the same edge in opposite directions at the same time step. In our implementation, it is up to the user to include or not this type of conflicts.

Following conflicts, instead, are not considered. So, an agent is free to occupy a location that was occupied by another agent in the previous time step. This, of course, only if the agent is not involved in other types of conflicts. Most work on search-based MAPF algorithms [15, 5] forbid edge conflicts, but allow following conflicts. Some others, instead, allows edge conflicts [8].

In many cases the solution is required to minimizing a cumulative cost function. In this thesis we focus on two different objective functions [18].

- **Sum of costs (SOC)**. It is the sum of time steps required by each agent to reach its goal location. Given a MAPF solution it is the summation of all the single-agent plan lengths.

- **Makespan**. It is the number of time steps required for all agents to reach their goal locations. Given a MAPF solution it is the length of the longest plan.

When solving a MAPF problem it is up to the user to decide which objective function the solver will minimize during the planning.

2.3 Classes of MAPF Solvers

A first differentiation can be done by dividing the algorithms into two classes: *optimal* and *sub-optimal* solvers. The latter ones are usually used when the number of agents is large. In such cases, we aim to quickly find a solution, even if it is not the optimal one. Optimal solvers, instead, are usually applied to MAPF problems with a relatively small number of agents, and they find the minimal-cost solution. Solvers can also be *complete*, when they guarantee to find a solution whenever one exists.

Another way to divide these solvers is based on the approach used to solve the problem. Those approaches can be divided in two categories: decoupled approach and coupled approach. In the *decoupled approach* paths are planned for each agent separately. An example is HCA* [14]. Given an agent ordering, a path is planned for each agent such that it avoids conflicts with previously computed paths, by means of a reservation table. Decoupled approaches run very fast, but optimality and even completeness are not always guaranteed.

In the *coupled approach*, instead, the problem is formalized as a global search problem. This formulation can be solved by an A*-based algorithm [6] where the state space grows exponentially with the number of agents. Coupled approaches return the optimal solution at a significant computation cost.

There are also mixed approaches, like CBS [12], in which the high-level performs a global search while the low level performs strictly single-agent searches, similar to the decoupled approach. Another algorithm called ICTS [13] works in a similar way, and both those solvers are complete and optimal.

All these approaches belong to the class of *search-based solvers*, where the search-space can be represented by agent locations, like in the case of A*, or by conflicts, like in the case of CBS. Most of them are designed for the

sum of cost variant of MAPF, but can be easily modified to the makespan variant by slightly modifying the definition of the underlying search space.

Another possible class of algorithms, which is not treated in this thesis, is composed of the *reduction-based solvers*. This class of solvers reduces the MAPF problem to other known formalism, like the Boolean Satisfiability (SAT) [17], the Integer Linear Programming (ILP) [20], or the Answer Set Programming (ASP) [4]. Existing algorithms for solving these problems are usually highly efficient only on small problem instances. On large problem instances, the translation process has a very large overhead which makes these approaches inefficient. Many of these solvers are designed for the makespan variant of MAPF and they can't be easily modified to address the sum of costs variant.

Chapter 3

Implementing a MAPF Framework

Our goal is to produce a tool able to solve a MAPF problem with a chosen algorithm, and where it's possible to visualize the computed solution. In this section we discuss how we implemented the MAPF framework in the Python language, so, what we need for defining a MAPF problem, generating one and, in an abstract way, solve one, with an overview on the main classes and methods created.

3.1 Generic MAPF Problem Implementation

Let's start focusing on the main elements of a MAPF problem. The first two implemented classes, which represent the input of a MAPF problem, are *Map* and *Agent*. In our formulation, the graph $G = (V, E)$ is represented by a grid world, that we simply call *Map*. The vertices are represented by the grid cells, while the edges are represented by the possible moves allowed in a location.

A *Map* instance is defined by a height, a width, and a list of occupied positions. This class offers a method *neighbours()* which returns the list of valid locations adjacent the inserted one. Specifically, given a location (a vertex), it returns the list of location that are connected to the given one and are not obstacles. Since we focus on 4-connected grids, every location we'll have at most 4 possible neighbours. An *Agent* instance, instead, is defined by an id, a starting location and a goal location.

This two classes are both used in the formulation of a *ProblemInstance* (Figure 3.1). This class represents an instance of a MAPF problem. It contains the map of the problem and a list of agents. During the initialization

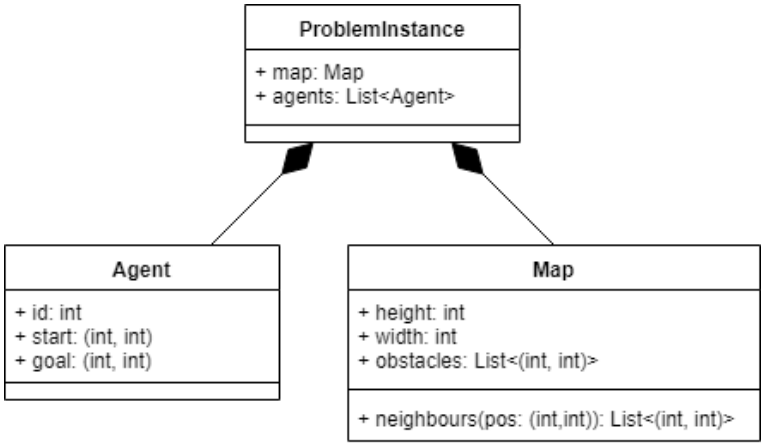


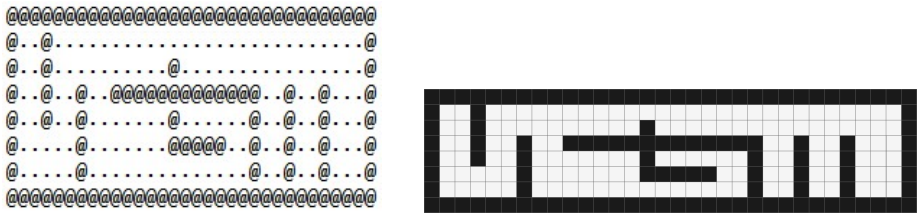
Figure 3.1: UML diagram of the MAPF problem definition.

of an instance of this class, it is checked that no agent is located outside the boundaries of the map, no agent initial or goal position overlaps an obstacle, and there are no duplicates positions between agents. The given set of agents is modified in the problem such that all the ids are consecutive integers in ascending order starting from 0. This is done because agent ids can be any value, but when solving a problem we want that these ids are consecutive integers in order to simplify the implementation of the various algorithms. The original set of agents is also kept, since some algorithm needs it.

3.2 Problem Instance Generation

Now we present how instances of MAPF problems can be generated. We have implemented some functions that take care of creating the map, agents, and more in general problem instances.

We start from the creation of the map. The map can be randomly generated or can be load from a file. For the first option, we implement a method called *generate_random_map()* that takes in input the desired size of the map, and an obstacle probability, and returns an instance of *Map*. The obstacle probability represents the probability of a cell of being an obstacle. For the second case, instead, we use files .map. These files are text files where the map structure is represented by symbols, using a dot ('.') for representing a free cell, and an at-sign ('@') for representing an occupied one. Figure 3.2 shows an example of .map files and its corresponding grid representation.



(a) (b)

Figure 3.2: (a) A .map file to upload. (b) Grid representation of the map loaded from the file.

For loading those files we implement the *load_map_from_file()* method, which takes in input the file path and returns a Map object.

Now, we proceed with the creation of the agents. Agents can be randomly generated or can be loaded from a file. When generated randomly, their start and goal locations are sampled uniformly from the free cells of the map. Method *generate_random_agents()* takes care of doing this. Instead, when we want to load them, we use .scen files. Those files represent a possible scenario, i.e., a list of source and target vertices, and it refers to a specific map. For doing this we implement the *load_scenario()* method, which returns a list of agents whose configurations are taken from the scenario file given in input.

For our experiments, we used the publicly available benchmark explained in [16]. It consists of 24 maps taken from maps of real cities, the video games Dragon Age Origins and Dragon Age 2, open grids with and without random obstacles, maze-like grids, and room-like grids. We used some of these maps in our implementation.

For each map, there are 25 (x2) benchmark sets available. One set of benchmarks contains the random scenarios, i.e., problems that are generated purely randomly, capped at 1000 problems per file. Here the individual problems for a given agent will all tend to be longer. The other set of benchmarks contains the even scenarios, i.e., problems that are evenly distributed in buckets of 10 problems with the same length. These problems will have an even mix of short and long problems.

In our implementation, we create a class *Reader* that takes care of loading the maps and the agents from the files. This class has three main attributes: the map number, the type of the scenario, and the number of the scenario file. (1) The number of the map refers to the map to be loaded, the correspondence is specified by a dictionary with as key an integer and as value the map name. (2) The type of the scenario can be “even” or “random” and it is used to decide which set of scenario use. (3) The number of the scenario

is used for the selection of the file among the 25. This class offers the same methods explained above about loading a map or a scenario, with some advanced features. It allows for example to select another set of agents from the same scenario file. In case we are using a random scenario file it will sample randomly another set of agents from the file. Instead, if we are using an even scenario file, it will sample agents from the next bucket keeping the order. This class allows also to reuse the previous loaded instances.

3.3 Generic Solver and Settings

Let's now describe how we will solve a MAPF problem, and which classes we will use. First, we create an abstract class that represents the abstract idea of what a solver will do. We called it *AbstractSolver* and will be used by all the algorithms (Figure 3.3).

This abstract class has only one attribute: an instance of *SolverSettings*. This class contains all the characteristics of the solver that will be used during the computation of the solution.

- *Heuristic string.* This is a string which defines the heuristic that the solver will use.
- *Heuristic object.* This is an instance of the *Heuristic class.* The role of this class is explained in Chapter 5.
- *Objective function.* This is a string that defines the objective function used for the problem, that means what objective the solver will try to minimize.
- *Stay at goal.* This is True if the agents never disappear once they reach the goal. Otherwise, it is False if the agents disappear.
- *Goal occupation time.* If the previous attribute *stay at goal* is False, this variable tells how many time steps the agents will stay at the goal before disappearing. The minimum value is 1.
- *Edge conflict.* If True, the solver will consider also the edge conflicts during the search.
- *Timeout.* Maximum amount of time for computing the solution.

All these attributes will be used by the actual solver in order to perform the search. The main method that this class offers is the method *solve()*, which, given a *ProblemInstance*, returns, if one exists, a solution. It also

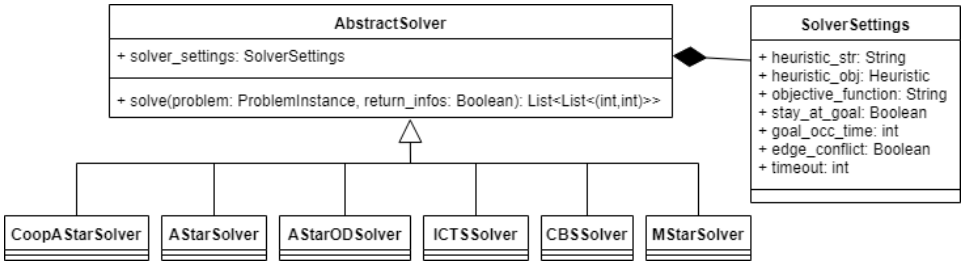


Figure 3.3: UML diagram relative to the abstract solver.

receives as parameter a Boolean value called *return_infos*. If this value is True, the method returns some output information in addition to the solution. This information is related to the cost of the solution, like the sum of costs and the makespan, or to the computation, like the number of generated nodes, the number of expanded nodes, and the time spent.

The *solve()* method is the same for each algorithm, and it just launches a thread that executes the *solve_problem()* method and computes the output information. The thread is started with a timeout, specified in the solver settings. If the thread is not finished within the timeout, the thread is stopped. In order to do this, we have declared a stop flag which will be regularly checked by the thread. This stop flag will be set as soon as the program reaches its timeout, and when it happens the thread breaks its execution, and so an empty solution will be found.

The *solve_problem()* method is different for each solver, and it is the core of the specific algorithm.

3.4 Solvers Output

The output of the solvers is composed of a list of agents positions. More precisely, a list of lists of agent positions, where output$[i]$ represents the path for agent i. Based on the settings of the solver we distinguish two possible outputs. We decide that when the stay at goal is True, we simply return for each agent the list of positions it occupies to reach the goal, including this last one only once even if the agent remains there forever. Instead, when it is False, we repeat the goal position as many times as the goal occupation time value is. An example of possible outputs is shown in Figure 3.4. We can see that in the first case the list of positions contains the goal state only one time, while in the other case it is repeated 3 times, since agents need to stay in the goal for 3 time steps before disappearing.

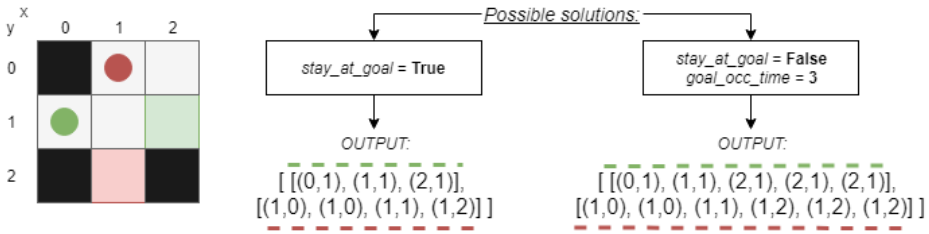


Figure 3.4: Examples of possible outputs of the solve method.

3.5 Paths Processing

In this section, we discuss some implemented methods useful when working with lists of paths:

- *check_conflicts()*. This method takes in input a list of paths and checks if it exists a conflict between them. It is up to the user to decide if edge conflicts are also checked. It returns the ids of the first two agents involved in a conflict.
- *calculate_soc()*. Given the list of paths, it returns the sum of cost value. Time spent in goal is not considered.
- *calculate_makespan()*. Given the list of paths, it returns the makespan value. Time spent in goal is not considered.

Chapter 4

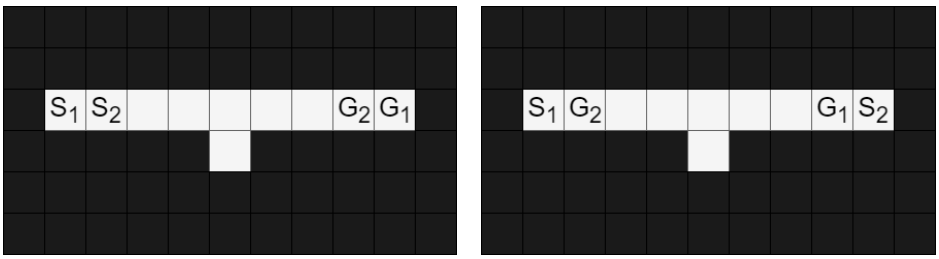
Algorithms for Solving MAPF Problems

In this section, we discuss how a MAPF problem can be solved and we present all the algorithms addressed by this thesis, theoretically describing them in details. In this thesis, we only focus on search-based solvers and, in particular, only on complete and optimal solvers. The only exception is the first algorithm, HCA*, which is neither complete nor optimal. We present the algorithms one at a time, discussing how they work and the properties they have.

4.1 Hierarchical Cooperative A* (HCA*)

This algorithm [14] is an example of a *decoupled approach*, that means that paths are planned for each agent separately. It can run very fast, but optimality and even completeness are not always guaranteed. The algorithm works in this way. Given an agent ordering, individual searches are performed one for each agent consecutively. Those are single-agent searches, for example A*, and need to take account of the planned routes of the previous agents. In order to store these reserved locations, a *reservation table* is used. This table represents the shared knowledge about each other's planned routes. It is a sparse data structure that stores all the busy locations with the respective time step.

For the following examples, shown in Figure 4.1, we consider the stay at goal assumption. In Figure 4.1(a) it is shown an example of a problem that cannot always be solved. In fact, if the first agent to plan is a_1, then the solver will find a solution. Agent a_1 will decide to go straight to the goal, while agent a_2 will wait in position (4,1) to let a_1 pass towards its goal.



(a) (b)

Figure 4.1: (a) Simple problem that shows the importance of the agent ordering. Based on which agent starts, the solution can or cannot be found. (b) Problem that can't be solved by Cooperative A* [14].

Instead, if the first agent to plan is a_2, then a solution can not be found. In this case, agent a_2 decide to go straight to the goal. Now, agent a_1 has no way to reach its goal, since a_2 once reached its goal will stay there and will obstruct the path of the other agent. In Figure it 4.1(b) is shown an example of a problem that cannot be solved. Here, no matter who is the agent to plan first, a solution cannot be found. The path of the first planned agent will block the other, which will not reach its destination.

4.2 A* - Based Search

This algorithm [6] is complete and returns the optimal solution. It is an example of a *coupled approach*, that means that the problem is formalized as a global search problem. In fact, it searches in a global search space which combines the individual states of all the k agents. So, a state is represented by the combination of all the single-agent states, and, during the expansion of the state, it considers the moves of all agents simultaneously. Figure 4.2 represents the basic idea of this algorithm. In case, for example, the branching factor is $b_{base} = 5$, each state has potentially 5^k possible children.

To solve a MAPF problem more efficiently with A*, one requires a non-trivial admissible heuristic. We now explain the two different admissible heuristics that can be used for the single-agent case. Then, the heuristic value for the multi-agent case will be computed based on the objective function we are minimizing. If we are minimizing the sum of costs (SOC), this value will be computed by doing the sum over all the heuristic values of the single-agent states. Instead, if we are minimizing the makespan, it will be the maximum value over all the heuristic values of the single-agent states. The two single-agent heuristics used in our algorithms are:

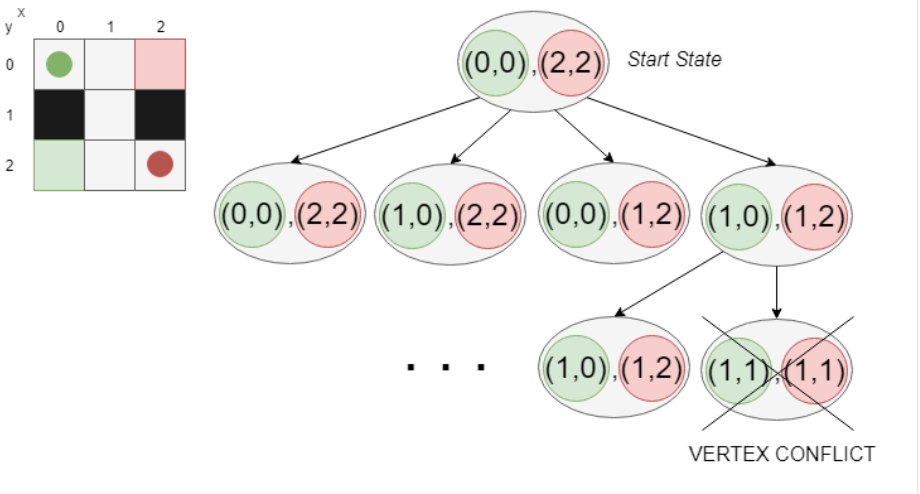


Figure 4.2: Basic idea of an A* search tree. We can see that a global state is composed by k single-agent states, here represented by their locations.

- **Manhattan distance**. The Manhattan distance between two points is the air line distance or, more specifically, is the sum of the absolute differences of their Cartesian coordinates.

- **Abstract distance with RRA***. This is the one used by Silver [14]. This heuristics can be viewed as perfect estimates of the distance to the destination. It is computed by running a single-agent search from a location to the goal with all the other agents removed. So, for each agent a_i we assume that no other agent exists and we precalculate its optimal individual path cost from start to goal. This is an admissible and consistent heuristic. Furthermore, the inaccuracy of the heuristic is determined only by the difficulty of interacting with other agents. Reverse Resumable A* algorithm [14] is used in order to avoid to launch a search every time I need to compute the heuristic. In fact, it keeps a table that stores the single-agent distances for every agent at every position. RRA* initialize the table by executing a modified A* in a reverse direction. Then, when an abstract distance is requested, RRA* checks whether it is already known or not. If so, the value can be returned immediately. If not, the RRA* search is resumed.

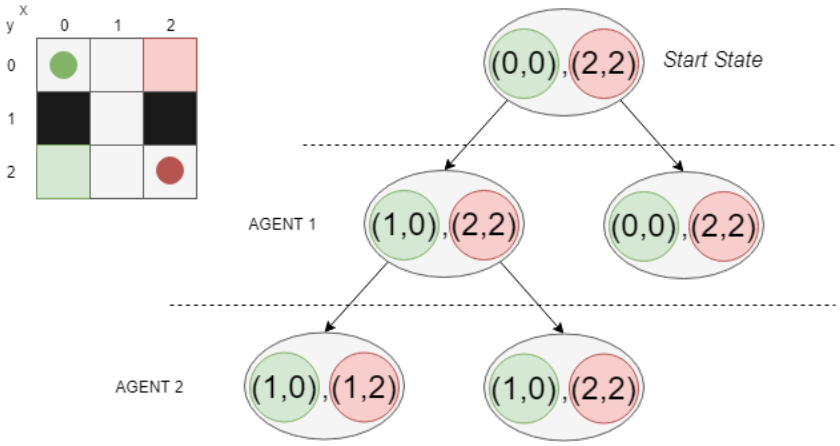


Figure 4.3: Basic idea of the A*+OD. We can see that agents are considered one at a time.

4.3 Standley's enhancements

Now we present two methods introduced by Standley [15] that substantially improve the basic A* settings: *operator decomposition (OD)* and *independence detection (ID)*.

4.3.1 Operator Decomposition (OD)

This first improvement reduces the branching factor by introducing *intermediate states* between the regular states. In this new representation, one agent is considered at a time, and when a state is expanded only one agent moves. For doing this, we need a new version of a state, in order to keep track of the agents that has already moved in that time step. This new state representation has a branching factor equal to b_{base}, since agents move one by one. So, now a state can be of two types: standard state or intermediate state. We refer to *standard states* as those in which no agent has moved, and so, every single-agent state that compose it has the same time step. Instead, we refer to *intermediate states* as those in which at least one agent has been assigned to a move, and so, the time steps of the single-agent states not all will be the same. This first improvement reduces also the total number of states generated by A*.

As an example, consider the MAPF problem shown in Figure 4.3. We can see that the first expansion consider only the first agent moves, and it will result in generating 2 children instead of the 4 children generated by the standard A*. Then, states in the frontier are sorted in the same way, and the one with the lowest f-value is pop and expanded. This time only

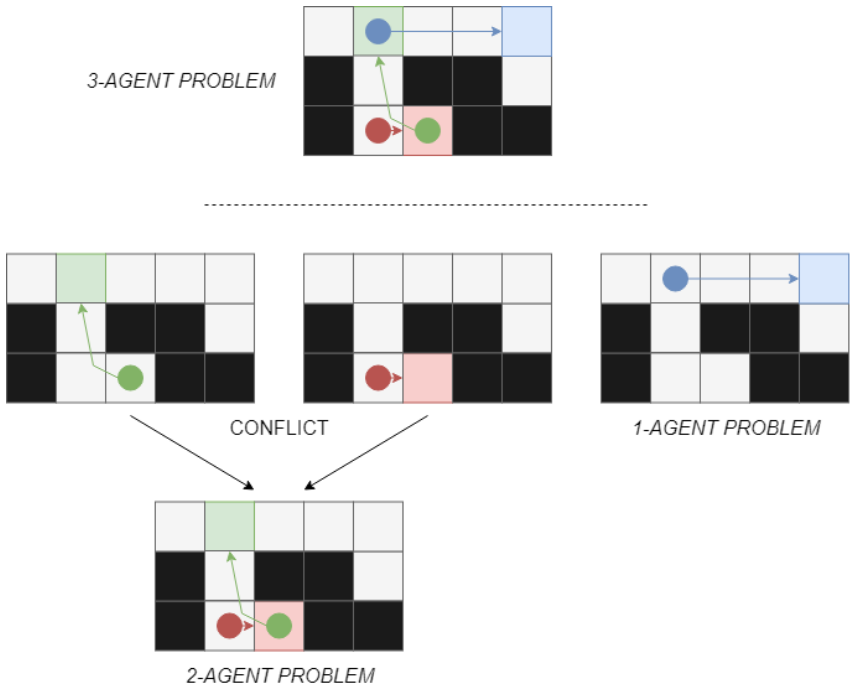


Figure 4.4: Example of Independence Detection framework. It reduces the complexity of the problem.

the moves of the second agent are considered. Note that, while standard and intermediate states are conceptually different, A* search treats them equivalently, so intermediate states derived from different standard states can be expanded in any order.

4.3.2 Independence Detection (ID)

ID detection is a general framework which runs as a base level and can use any possible MAPF solver on top of it. It is complete when coupled with a complete search algorithm, therefore it is usually used in conjunction with algorithms such as A*, ICTS, etc. Since these search algorithms are all exponential in the number of agents, they are effective only for small numbers of agents. In order to solve larger problems, ID divides agents into *independent groups* and solve sub-problems for these groups separately. Initially, each agent is placed in its own individual group and an optimal solution for each agent is computed without considering the other agents. Then, paths are checked to see if a conflict occurs between two groups. If so, all agents involved in the conflict are merged into a new group. Whenever a new group is formed, this new k-agent problem is solved using any optimal

MAPF solver. This process is repeated until no conflict between agents occurs. Standley observes that solving the largest group dominates the running time of solving the entire problem.

In Figure 4.4 it is shown an example of how the framework works. We can see that the 3-agent problem is initially split into three single-agent problems. Then, the solution paths are simulated and a conflict is found between two agents. Those are merged into a new group and solved together. So, instead of solving a 3-agent problem, we have solved a 2-agent problem and a 1-agent problem, with a reduction of the complexity.

4.4 Increasing Cost Tree Search (ICTS)

This algorithm [13] is based on the understanding that a *complete solution* for the entire problem is built from *individual paths* (one for each agent). ICTS is a two-level search algorithm.

- **High level**. At high level, it searches a tree called *increasing cost tree* (ICT). A node in the ICT consists of a vector of costs, $[C_1, C_2, \dots C_k]$, one per individual agent. This vector of costs represents all possible solutions in which individual path cost of agent a_i is exactly C_i (with C_i positive integer). The root of ICT is $[opt_1, opt_2, \dots opt_k]$, where opt_i is the cost of the optimal individual path for agent i which assumes that no other agents exists. A child is generated by adding a unit costs to one of the agents. An ICT node is a *goal* node if there is a non-conflicting complete solution such that the cost of the individual path for agent a_i is exactly C_i. Figure 4.5 shows an example of an ICT with three agents, all with individual optimal path costs of 10. Dashed lines mark duplicate children which can be pruned.

- **Low level**. The low level acts as a goal test for an ICT node. For each ICT node $[C_1, C_2, \dots C_k]$ visited by the high level, the low level is invoked. The task of the low level is to find a non-conflicting complete solution such that the cost of the individual path of each agent a_i is exactly C_i. A general approach to check whether an ICT node is a goal would be: **(1)** For every agent a_i enumerate all the possible individual paths with cost a_i. **(2)** Iterate over all possible ways to combine individual paths until a solution is found or all the combinations have been checked. If a solution exists, the solution is returned and the search is halted. Otherwise, the high level continues to the next ICT node. In Section 5.6 we implement a more efficient and effective algorithm for doing this.

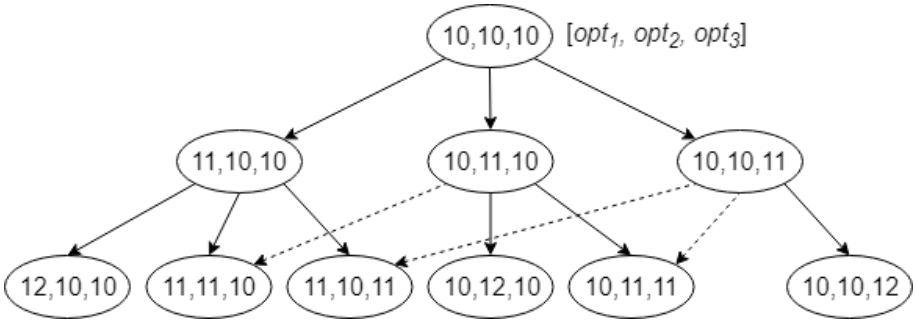


Figure 4.5: Example of ICT for three agents [13].

4.5 Conflict-Based Search (CBS)

This algorithm [12] is a continuum of *coupled* and *decoupled* approaches. In fact, CBS guarantees optimal solutions, like most coupled approaches, but the searches it performs are strictly single-agent. This approach is based on the use of constraints, which prevent agents from being in certain locations in certain time steps. CBS works at two levels.

- **High level**. At high level, it searches a tree called *constraint tree* (CT). Each node in the CT contains the following fields of data: **(1)** A set of constraints. The root contains an empty set and a child of a node inherits the constraints of the father plus a new one. **(2)** A solution. A set of k paths, one for each agent. The path for agent a_i must be consistent with the constraints of a_i. **(3)** The total cost of the current solution.

  A node is a goal node when the solution is valid, i.e., the set of paths have no conflicts with each other. Notice that the paths can be consistent with their constraints, but have conflicts together.

- **Low level**. The low-level performs single-agent searches. It considers one agent at a time, and searches for a path that is consistent with the constraints of the CT node. Any single-agent path-finding algorithm can be used here, like A* that is the one we used.

Example in Figure 4.6, shows how the algorithm works in a simple case. We start from the root node, which has no constraints. The low-level is called on this node and computes the paths by performing a single-agent search on both the agent separately. Those paths are then simulated, and since a conflict occurs, the solution is declared not valid. So, the node is expanded

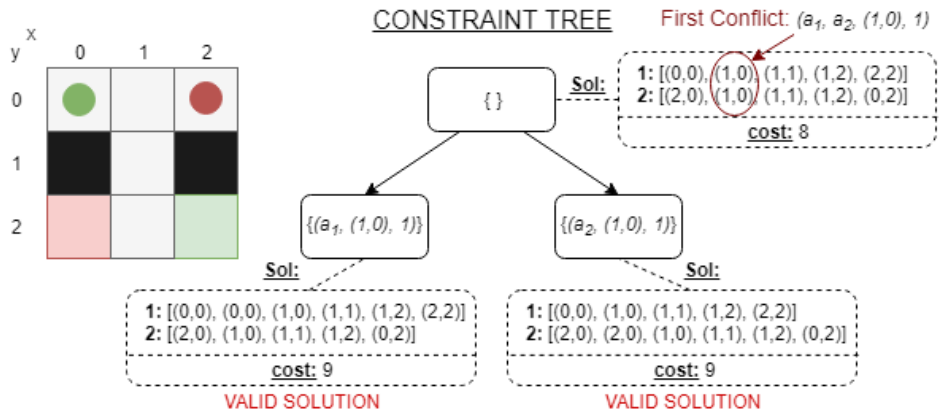


Figure 4.6: Example of CBS constraint tree.

and two children are generated by splitting the first found conflict into two new constraints. Then, the low-level is called on both of these nodes, and will take care of computing the paths satisfying the constraints. Since they have no conflicts, both of the solutions are valid, and so, they are both goal nodes. The algorithm returns only one solution, depending on which node it expands first.

4.6 M*

This algorithm [19] is an A*-based algorithm that dynamically changes the branching factor based on conflicts. The general idea is that a state generates only one child in which each agent follows its optimal move. This continues until a conflict occurs. In this case, the search dimensionality is increased, and all the ancestors nodes are placed back into the open list in order to be re-expanded again. This time, however, only the agents not involved in the conflict follow their optimal move. The conflicting agents, instead, will consider all their possible moves.

The main steps of the algorithm are the following: **(1)** Start the search and generate only nodes on optimal paths. **(2)** If a conflict occurs, backtrack and consider all ignored actions. **(3)** Repeat until no conflict is found.

In M* algorithm, the state contains a new field: the *collision set*. This set will contain all the agents that have been involved in a conflict in this node or in one of its successors. Then, if an agent is not in the collision set, it will follow its optimal policy, otherwise, it will consider all its possible moves.

Figure 4.7 shows an example of application of the M* algorithm. Initially,

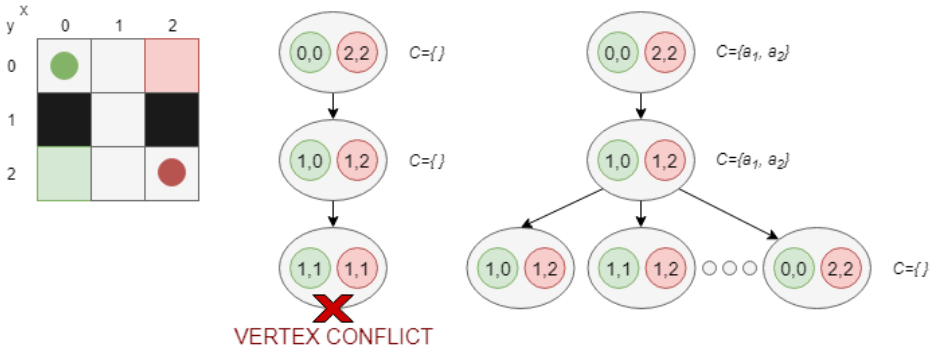


Figure 4.7: Representation of the basic idea of M* algorithm. Initially, only the optimal moves are taken. When a conflict occurs, the algorithm backtracks and considers for the agents involved in the conflict all their possible moves.

nodes are generated by following their optimal policy. As soon as a conflict occurs, we backtrack and consider, for the agents involved in the conflict, all their possible neighbours. So, we re-append the previous two (already expanded) nodes to the frontier and we update their collision set. Now, we pop a node from the frontier. Its collision set contains a_1 and a_2, so when expanding this node, we must consider all their possible combinations of neighbours like the usual A*.

Chapter 5

Algorithms Implementation

In this section, we discuss the previously mentioned algorithms in details by explaining how they have been implemented in the Python language. First, we discuss the single-agent case, then we continue with all the multi-agent algorithms one at a time.

5.1 Single-Agent Solver

Here, we make a quick review on the single-agent case, explaining how the classes *SingleAgentState* and *A** works, since those will be reused by the other MAPF solvers. Figure 5.1 shows their UML diagrams.

A *SingleAgentState* stores the agent position at a specific time step. As superclass of *State* it has a g-value, an h-value, an f-value, and the time step of the state. The g-value represents the number of time steps that the agent has spent away from its start location. Note that the number of time steps are counted until the time step in which it arrives at its goal without moving away. The h-value, instead, represents the heuristics value in that position, namely the estimated distance from the goal, and this value depends on the selected heuristic. The f-value is just the sum of these two values. Methods like *compute_heuristic()* and *compute_cost()* are used by the class to compute the h-value and the g-value of the state.

The main method of this class is *expand()* which simply expands the current state by computing all the possible neighbours' positions, creates a new state for each one of these and returns this list. The method *expand_optimal_policy()*, that will be used in the M* algorithm, returns the next optimal state, i.e., the next state we encounter by following the optimal policy.

Another important method is *goal_test()* which returns True if the agent

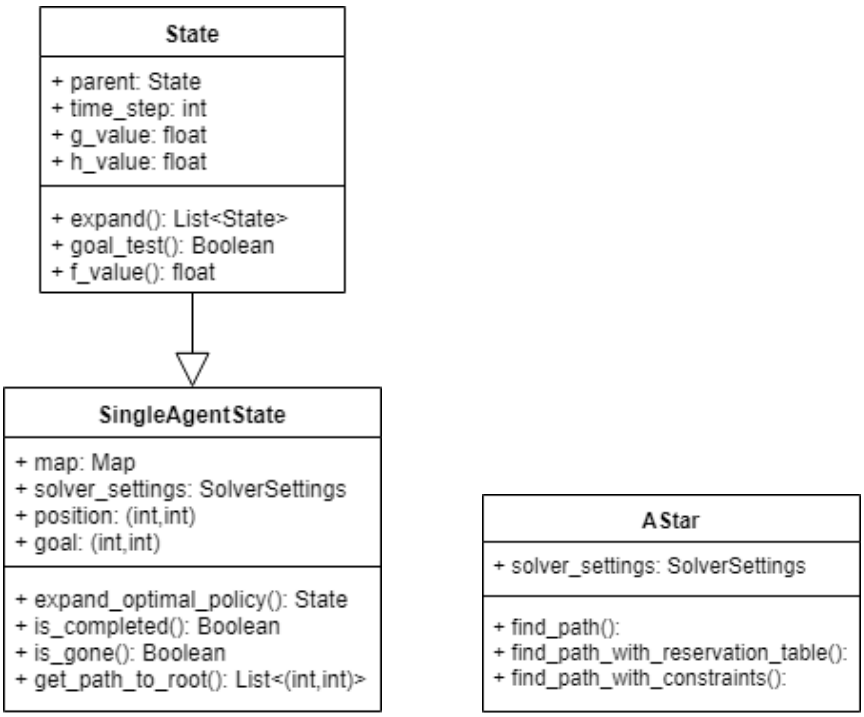


Figure 5.1: (a) UML diagram of the single-agent state class. (b) UML diagram of the single-agent A* star solver.

is on its goal position. Since in our case we implemented both the versions where the agent can stay at the goal or disappear after a number of time steps, we've implemented another method: *is_completed()*. This method keeps into account also the second version of the problem, by checking that the agent has spent the needed time stopped in the goal.

The *A** class takes care of solving a single-agent problem. In the method *find_path()* we implement the basic version of the A* algorithm, that is computing the path from a starting position to a goal position. Then, we implement two other methods that will be used by two of the MAPF algorithms. One, used by HCA*, solves the problem by taking into account a reservation table, that stores a list of prohibited positions. The other, used by CBS, solves the problem by taking into account a list of constraints.

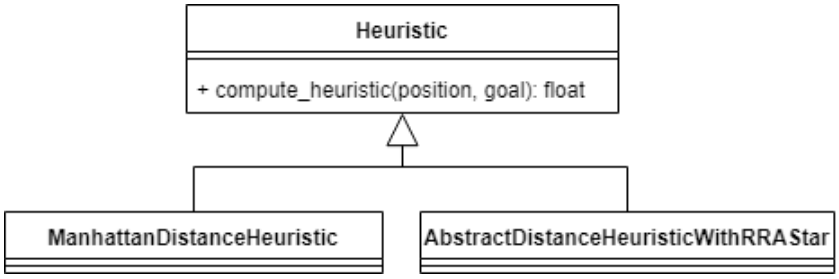


Figure 5.2: UML diagram of the classes involved in the computation of the heuristic.

5.2 Heuristic

Now, we explain how we implemented the use of the heuristic for the single-agents case. As already said, for the multi-agent case we just do the sum or maximum based on the objective function we are minimizing.

First, we create an abstract class which just offer one method: *compute_heuristic()*. This method takes in input the starting position and the goal position, and just returns the heuristic value between these two locations. This class is then extended by two different classes which represent the behaviour of the chosen heuristic (Figure 5.2).

- ***Manhattan distance heuristic***. This class just implements the *compute_heuristic()* method by returning the sum of the absolute differences of the Cartesian coordinate of the two locations.

- ***Abstract distance heuristic with RRA****. This class is the implementation of the heuristic viewed in Section 4.2. It computes the heuristic value by performing a single-agent search from the start to the goal with all the other agents removed. In order to efficiently do this, we implement the Reverse Resumable A* algorithm [14]. This algorithm executes a modified A* search in the reverse direction. So, when a heuristic value is requested, it just returns this value, if it is already present in the closed list, otherwise the RRA* search is resumed.

  More in details the algorithm does the following. First, we keep two lists for each agent: an *open list* containing the nodes not yet expanded and a *closed list* containing the nodes already expanded. We used two dictionaries in order to implement these, with the goal position of the agent as key, and the corresponding open/closed list of the agent as value of the dictionary. Then we initialize these two lists for each one

of the agents. We just perform a reverse A* search from the agent goal position to the agent starting position. In this way, the closed list will contain all the states expanded during the search. So, when a heuristic value is requested from a specific position to a goal position, the algorithm simply checks if it hass already been computed or not. It just watches if the position is present in the closed list of the agent involved, referring to the specific list using as key of the dictionary the agent goal. If so, it just returns the g-value of that state, otherwise, it resumes the search until that position is encountered. Pseudocode for the RRA* procedure is shown in Algorithm 1 and 2.

Algorithm 1 Reverse Resumable A*: part 1

```
1: procedure INITIALIZE_TABLE
2:     open_list ← dict()
3:     closed_list ← dict()
4:     for all A ∈ AGENTS do
5:         goal_pos ← A.get_goal()
6:         start_pos ← A.get_start()
7:         start_state ← SINGLEAGENTSTATE(start_pos, goal_pos)
8:         open_list[goal_pos] ← {start_state}
9:         closed_list[goal_pos] ← ∅
10:        RESUMERRA*(start_pos, goal_pos)
11:    end for
12: end procedure
```

The Manhattan distance is used as heuristic for the RRA* search, meeting the consistency requirements. So, the *start_state* in the first procedure is initialized with a *g-value* equal to zero and an *h-value* equal to the Manhattan distance between the *start_pos* and the *goal_pos*.

We called the initialization procedure *initialize_table()* since we imagine the two dictionaries of lists as a big table containing all the info for all the agents. Those lists will be used to calculate the abstract distance on-demand.

The implementation of the abstract method *compute_heuristic()* is done by simply checking if the requested heuristic value has been already computed. If so, it just returns it, otherwise it needs to be computed. In this case, we resume the search until the requested node has not been expanded. Note that, for each new node, we add it to the frontier if it is not already

present in the open list nor in the closed list, or if it is present in the open list but its f-value is less than the f-value of the node in the list.

Algorithm 2 Reverse Resumable A*: part 2

```
1: procedure ResumeRRA*(P, G)                    ▷ P is position, G is goal
2:     open ← open_list[G]
3:     closed ← closed_list[G]
4:     while open ≠ ∅ do
5:         node ← open.pop()
6:         closed ←(add)— node
7:         if node.position() = P then
8:             return success
9:         end if
10:        for all state ∈ node.expand() do
11:            if state ∉ open & state ∉ closed then
12:                open ←(add)— state
13:            end if
14:            if state ∈ open & f(state) < f(state in open) then
15:                open ←(add)— state
16:            end if
17:        end for
18:    end while
19: end procedure
```

```
1: procedure compute_heuristic(P, G)             ▷ P is position, G is goal
2:     if P ∈ closed_list[G] then
3:         return closed_list[G].get_node(P).g_value()
4:     end if
5:     if resumeRRA*(P,G) then
6:         return closed_list[G].get_node(P).g_value()
7:     end if
8:     return None
9: end procedure
```

5.3 Hierarchical Cooperative A* (HCA*)

This solver is one of the easiest to implement. We start implementing Cooperative A* (CA*), and then, we can simply obtain the Hierarchical version of the algorithm (HCA*) by using the previous explained heuristic computed with RRA*. Hierarchical Cooperative A*, in fact, uses a simple hierarchy containing a single domain abstraction, which ignores both the time dimension and the *reservation table*. Abstract distances can thus be viewed as perfect estimates of the distance to the destination, ignoring any potential interactions with other agents. Reverse Resumable A* (RRA*) is used in order to reuse the search data, and avoiding performing searches from scratch each time.

The task is decomposed into a series of single-agent searches. The individual searches take into account the planned routes of other agents by means of a *reservation table*. This table is used to mark the impassable positions and represents the common knowledge between the agents. A simple implementation of this table is to treat it as a 3-dimensional grid (two spatial dimensions and one time dimensions). We have implemented it using a dictionary, where the key is represented by a position, and the value is a list of time step. Those represents the time steps in which that position is busy. This is an efficient implementation, since only a small portion of the locations will be touched, without creating immediately all the grid.

The algorithm works like this: it takes each agent one at a time, and solves the single-agent problem considering only the reservation table for checking collisions. For example, we start from agent a_0, and we solve the problem without considering any other agent. We fill the reservation table with the position occupied by the first agent along the planned path, and then we pass to agent a_1. The problem related to this agent is solved considering the busy positions of the previous one, and so on for all the agents. Algorithm 3 shows the high-level of CA*.

In the case the stay at goal assumption is used, we also keep a list of the positions that represents the goals of the agents which paths have already been computed, *completed_pos*. This reminds that agents will occupy their goal forever. In fact, if a position is occupied by one agent that has reached its goal, this position will be busy from that time on. So, when we check that a position is free at a specific time step, we also check if at that position some agent will stay forever. If so, we just get from the dictionary the last time step present, which indicates that from that time step on that position will be busy.

The algorithm used for solving the single-agent instances is a simply A*

Algorithm 3 Cooperative A*

```
1: procedure SOLVE(P)                          ▷ P is the problem instance
2:     res_table ← dict()
3:     paths ← [ ]
4:     for all agent in P.agents do
5:         solver ← AStar()
6:         path ← solver.find_path_with_res_table(P.map, agent, res_table)
7:         paths ←(add)— path
8:         Update res_table with the found path
9:     end for
10:    return paths
11: end procedure
```

with the use of a *reservation table*. It works like the usual A*, but each time a new state (with a specific position and time step) is expanded the following checks are done: **(1)** We check that the position is not busy in that time step. We simply get from the reservation table the list of time steps in which that position is busy. If our time step is in that list, that state will be discarded. **(2)** This check is done only with stay at goal assumption. In this case, we also see if the position is in the *completed_pos* list. If so, we go to the reservation table and take the last time step. From that time step on, that position will be busy due to an agent that occupies its goal forever. So, if the time step is greater than that time step, the agent will collide, and so that state will be discarded. **(3)** If the position represents the goal position of that agent, it is also checked that no previous agent has planned to pass on it in the future. In this case, in fact, we cannot simply leave that agent there, but we have to delete that state and wait outside the goal until all previous agents have crossed it. **(4)** If the edge conflicts are considered, we also check that no agents pass through.

A note to add to this A* algorithm with the *reservation table* is that, in this case, when checking that the current state is not already present in the closed list, we need to compare not only the position, but also the time step. That's because we have to allow agents to wait in their current location, since it can happen, for example, that an agent has to wait in a certain position to let the other already planned agents to pass and clear its way. In the single-agent state, instead, this is not done and only the position is kept into account. That is because it is pointless for a single agent to wait in a location, since it only makes the cost of the solution increasing when, around, nothing changes.

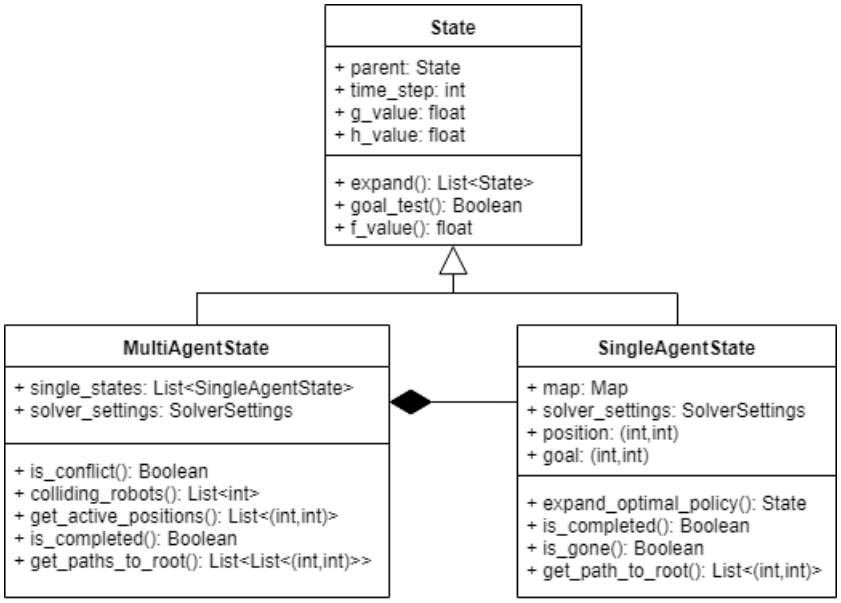


Figure 5.3: UML diagrams related to the multi-agent state.

5.4 A* - Based Search

It works like the single-agent A* with a new state representation. A state will not represent a single position, but the positions of all the agents together. We create a class called *MultiAgentState* (Figure 5.3), which is composed of k single-agent states and has all the characteristics of a state, like a g-value, an h-value, an f-value, and a time step. The first two values are computed in different ways based on the objective function we are minimizing. If we are minimizing the sum of costs (SOC), the g-value is the sum over all the g-values of the single-agent states. Instead, if we are minimizing the makespan, the g-value is the maximum value over all the g-values of the single-agent states. The same applies to the h-value. The *expand()* method works in this way. When expanding a state, all the single-agent states are expanded and all their combinations considered. Obviously, only the states with no conflicts are considered, while the others discarded.

The algorithm is like the usual A*. We keep a frontier with the states that will be expanded, and a closed list of states already visited used for detecting duplicates. We start by adding to the frontier the start state, the state where all the agents are located in their starting positions. Then, at each loop, the state with the minimum value is taken from the frontier and, if not already in the closed list, expanded. All valid states obtained from

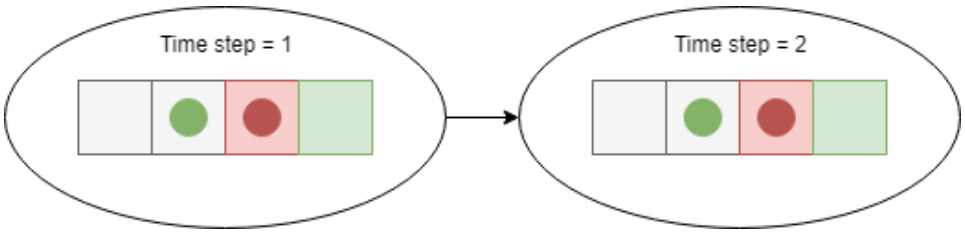


Figure 5.4: In this example, we can see two possible multi-agent states. In case the case we consider the stay at goal assumption, those are considered as duplicates, and so, it's pointless to expand the one on the right. Instead, in case agents disappear at goal, those are not considered as duplicates, since one agent needs to wait the other is gone before being able to reach its goal. With this configuration, only with the second assumption, a solution can be found.

the expansion are then returned and added to the frontier. The expanded state is added to the closed list and the loop is repeated. When the goal state is found, the search ends, and the paths from the goal states to the start states are computed and returned.

In order to analyze conflicts in a correct way, we add to the class *SingleAgentState* a method called *is_gone()* which simply checks if the state is active or not. In the case the method returns True, it means we are using the assumption where agents disappear at goal, and the agent has been removed from its goal since it has already spent the needed time there (defined in the *goal_occupation_time* variable). This because those agents will no longer block other agents to pass. It helps us to only consider the active agents (agents not yet disappeared) when searching for conflicts. The *get_active_positions()* method in the *MultiAgentState* class takes care of returning only the blocking positions.

Vertex conflicts are tested by ensuring to not have two equal active positions in the same state. If this happens, it means that two agents are at the same location at the same time step, and this is not allowed. For the edge conflicts, instead, we check to not have the case in which an agent a_{i} occupies a location previously occupied by another agent a_{j}, and that agent a_{j} now occupies the previous position of a_{i}.

We need to make a little difference in the algorithm in the case an agent can disappear at goal. In fact, in the standard case, two states with all the agents in the same positions are considered as duplicates. But, it can happen that one agent is waiting for another agent to disappear before continuing its path, and, this situation cannot be found if we don't change the duplicate detection. So, only in this case, we allow the expansion of states where one agent is spending its remaining time in the goal, even if all the agents are still in their current positions. We can see an example of it in Figure 5.4.

5.5 Standley's enhancements

In this section will explain how the two improvements have been implemented.

5.5.1 Operator Decomposition (OD)

In order to implement this algorithm, we create a new state class, called *ODState*. This class is child of the class *MultiAgentState*, so it inherits all properties and methods from it. It contains the list of single-agent states, and, in addition, a cursor which indicates the current agent to move. So, when an OD state expands, it simply expands only the single-agent state indicated by the cursor, leaving untouched the other states. We also add a method to differentiate a standard state from an intermediate state. Remember that the standard states are the ones where every agent has moved, while the intermediate states are the ones where only a subset of agents has moved. The method simply checks the value of the cursor. If the cursor is positioned on the first agent it means that all other agents have already been moved, and so, it is a standard state. Otherwise, in all the other cases, it is an intermediate state.

In order to guarantee the algorithm to find the optimal solution, we need to make a little adjustment. Consider the situation depicted in Figure 5.5. If the algorithm considers the agent a_1 first, then it might conclude that it has only one legal move (wait). However, the optimal solution requires that both agents make the move E (go East) first. So, the algorithm should allow agents to move into spaces occupied by agents who are not been already assigned to a move. In fact, if we allow agent a_1 to go in (1,1) and we will check the conflict only when agent a_2 will have also moved, then the case in which both agents move E first will be possible. To solve this problem, we simply decide to check conflicts only on already moved agents. So, when a new state is generated, we take all the agent before the current agent to move, and we look for collisions only between them.

There's also a difference in the detection of the edge conflicts in the OD case. In fact, we cannot take simply the previous state and compare the positions, but we need to get the previous standard state. Let's recall the example in Figure 5.5. We can see that the edge conflict in the state with agents positions (1, 1) and (0, 1) can be detected by comparing this state to its previous standard state, that in this case is the start state. It can be easily noticed that the two agents positions in those states are swapped, and so an edge conflict is present. If we consider only its previous state,

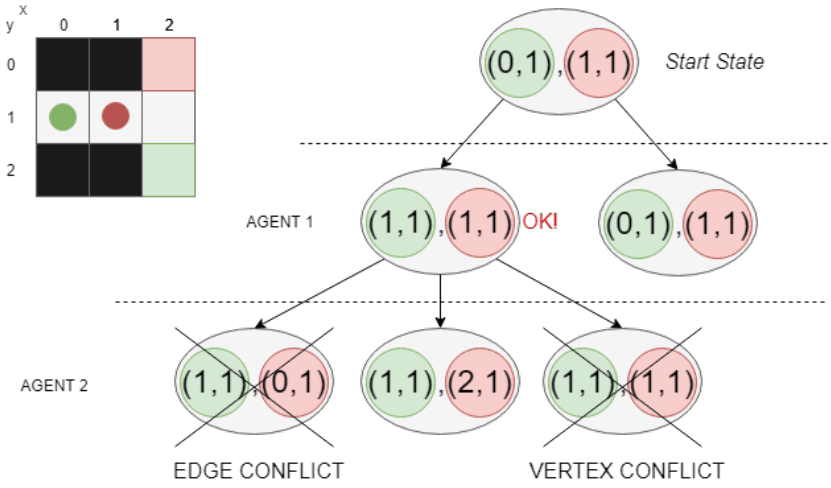


Figure 5.5: This example shows the collision detection in A*+OD.

there would be no way to detect it. So, we add to the *ODState* class a method that returns the previous standard state, and we will use this when searching for edge conflicts.

In general, the algorithm works like the standard A* algorithm with the exception that a state needs to be also standard in order to be a goal state. We also use the closed list to detect duplicates. Because duplicate intermediate states would have duplicate standard ancestors states, A* will never encounter duplicate intermediate states. For this reason, we only put standard states in the closed list.

Similarly to the basic multi-agent A*, we need to make a little fix in case agents can disappear at goal. Also here, we allow the expansion of states where there is at least one agent which is spending its remaining time in the goal, despite all the agents' positions are the same.

5.5.2 Independence Detection (ID)

Like all the other solvers, the *IDFramework* is an extension of the class *AbstractSolver*. Since it is a framework, it needs a MAPF solver on top of it in order to work. In fact, this algorithm solves each sub-problem one at a time by using that solver. The main steps of the algorithm are the following: **(1)** We assign every agent to a singleton group, and we create a new problem for each one of the groups. Then we solve each one of these problems using the top solver. **(2)** We simulate the execution of all paths in order to check the presence of some conflicts. **(3)** If no conflict occurs, then

the solution paths are found. Otherwise, we merge two conflicting groups into a new group, and we create a new problem instance with those two. **(4)** We solve the new problems and we repeat this procedure until no conflict occurs. A simple version of the procedure is shown in Algorithm 4.

In our specific implementation, each time two groups are merged, a new instance of the problem is generated containing the newly merged groups. Notice that every time a new problem is created, we need to solve only the new merged problem, since all the others were still the same.

Algorithm 4 Simple Independence Detection

```
1: assign each agent to a singleton group
2: solve each single problem with the top solver
3: simulate execution of all paths
4: while some conflict occur do
5:     merge two conflicting groups into a new single group
6:     solve the new merged problem with the top solver
7:     simulate execution of the paths
8: end while
9: solution ← paths of all groups combined
10: return solution
```

5.6 Increasing Cost Tree Search (ICTS)

Let's start with the high-level of this algorithm, which duty is to search for a minimal cost solution in a search space that spans combinations of individual agents' costs (one for each agent). Similarly to other algorithms, it starts with the frontier which contains only the start node, $[opt_1, opt_2, \dots opt_k]$, and continue to pop nodes from the frontier until the goal node is found. In our case the *ICTNode* is represented by a vector of costs: $[C_1, C_2, \dots C_k]$. This vector of costs represents all possible solutions in which path cost of agent a_i is exactly C_i. Nodes of the same level of ICT have the same total cost. The method *expand()* of an *ICTNode* generates all the possible children obtained incrementing by one the cost of one of the agents. For example, if a node has as costs vector $[C_1, C_2, \dots C_k]$, the k children will be $[C_1+1, C_2, \dots C_k]$, $[C_1, C_2+1, \dots C_k]$, $\dots [C_1, C_2, \dots C_k+1]$.

The *goal test* is done by calling the low-level search and verifying if there is a non-conflicting complete solution such that the cost of the individual path for each agent is exactly the one specified in the vector of costs.

In case the task is to minimize the makespan, the ICT works differently.

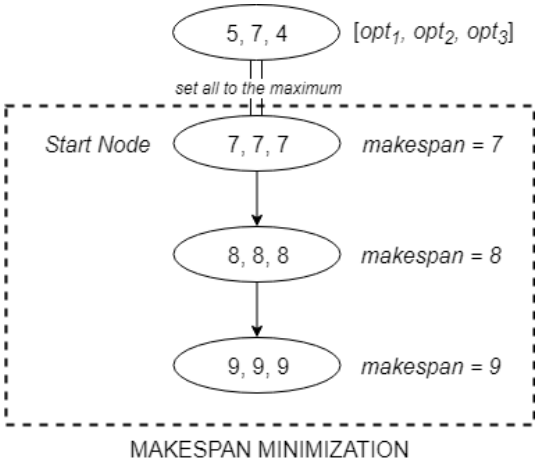


Figure 5.6: This example shows how the high-level of the ICTS works in the case we are minimizing the makespan as objective function.

For this case, there is no meaning to the individual cost of a single agent. The ICT will be linear instead of exponential, since each level of the tree will have only one node which corresponds to the makespan. For simplicity, we decide to keep the costs vector, but those values will be equal to the value of the desired makespan. Next, when a node is expanded, it will just increase the makespan cost by one, and so, only a new node will be generated. This node will have all the previous costs increased by one. For example, if a node has as costs vector $[C_1, C_2, \dots C_k]$, the child will have as costs vector $[C_1 + 1, C_2 + 1, \dots C_k + 1]$. Also, the start node will be different. It will be obtained by taking the maximum of the optimal costs, $max(opt_1, opt_2, \dots opt_k)$, and set all the values of the vector to that.

In Figure 5.6 it is represented an example of ICT with makespan minimization. We can see that from the point of view of the makespan this new start node is equal to the previous one.

Let's now analyze the low-level. The simplest idea is to search all the possible paths for every agent with the given cost, and then iterate over all the possible ways to combine these paths. This approach works, but it would be very computationally and time expensive. So, we implement an effective way to compute those paths.

First, we store these paths in a special compact data structure called *multi-value decision diagram (MDD)*. For example, let compute the MDD_i^c for agent a_i which stores all the possible paths of cost c. This structure will have a single *source node* at level 0, and a single *sink node* at level c. Every node at depth t corresponds to a possible location of a_i at time t, that is

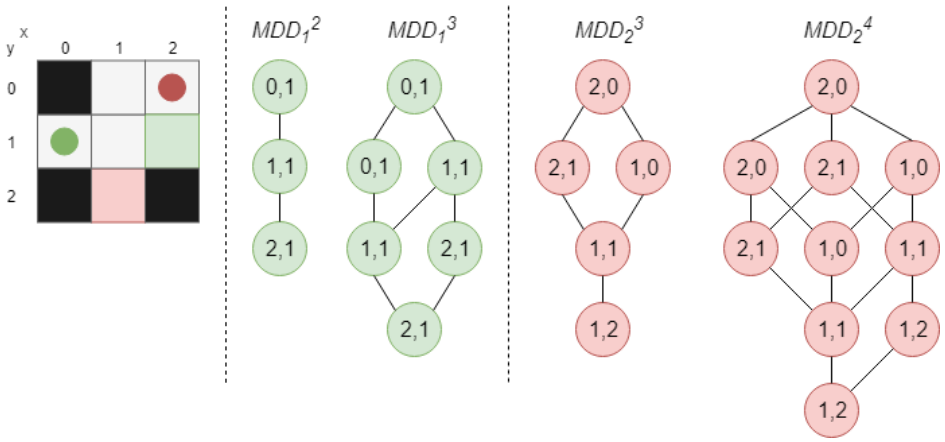


Figure 5.7: Example of construction of MDDs on a 2-agent problem.

on a path of cost c from the starting position to the goal position. Building those MDDs is very easy. We perform a breadth-first search from the start location of agent a_i down to depth c and only store this partial diagram.

Figure 5.7 illustrates the MDD_1^2 and MDD_1^3 for agent a_1, and MDD_2^3 and MDD_2^4 for agent a_2.

So, once we have built the corresponding MDD for each of the agents, we need to find a set of paths, one for each MDD that do not conflict with each other. In order to efficiently doing this, we build the *total MDD*. This diagram is similar to a single MDD, but here the state space is a k-agent search space. A node in this diagram corresponds to a set of k positions. This diagram is built by putting together all the single MDDs and creating a single diagram that is the composition of them. The implementation starts with the source node, which is the unification of all the single MDD source nodes. Then the possible children are obtained with the cross product of all the single MDD children nodes. We just create a list containing the lists of the MDD children nodes, and then, we iterate over all the elements and create all the possible combinations of nodes. For example, if the list is [[ABCD], [PQ], [XYZ]] where each sub-list represents the set of children of a single MDD node and each letter an MDD node, the generated nodes are: [APX], [APY], [APZ], [AQX], [AQY], [AQZ], [BPX], [BPY], ..., [DQZ]]. This process continues until we reach the sink node of the total MDD, composed by all the single sink nodes. Any possible conflict is checked during the construction of the diagram, so, if no total MDD is created, it means that there is no valid solution with those costs.

Now, we explain more in details the construction of the *total MDD*. We

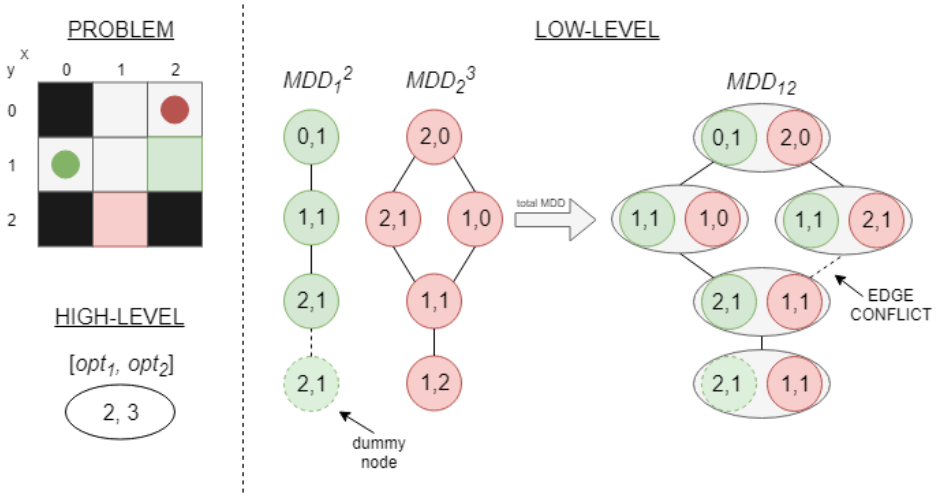


Figure 5.8: ICTS example. At the high-level we search in space composed by the paths costs, while at the low-level we verify the existence of at least one valid solution with those costs. It is also showed how the *total MDD* is computed.

define a *total MDD node* as a node composed by a list of single-agent MDD nodes. We start by putting in the frontier the start node, which is composed of the list of the single-agent MDD root nodes. We also need to set the length of this tree to the maximum length of the single-agent MDDs we want to merge. Now, we sort the frontier by time step, since we need to construct the tree incrementally from the root downwards. Then we pop a node from it and we expand this node. The expansion consists in generating the cross product of all the possible children of the single-agent MDDs, removing the nodes with conflicts. In case we are merging two single-agent MDDs with different size, and one has already reached its goal node, we add a dummy single-agent MDD node with the goal position. This is equal to the other nodes, but it is generated only to make the two diagrams to merge with equal depth. The generated nodes are added to the frontier and the process is repeated. The construction ends once reached the *sink total MDD node*, which is composed of the list of the single-agent MDD sink nodes. If the frontier is empty before the goal node is reached, it means that a solution with those costs doesn't exist, and all the nodes have been removed due to conflicts.

Let us assume, for example, that we want to verify if it exists a solution for the problem in Figure 5.8 with path cost equals 2 for agent a_1 and path cost equals 3 for agent a_2. First, we need to compute the single MDDs for the two agents: MDD_1^2 and MDD_2^3. Then we compute the *total MDD* by

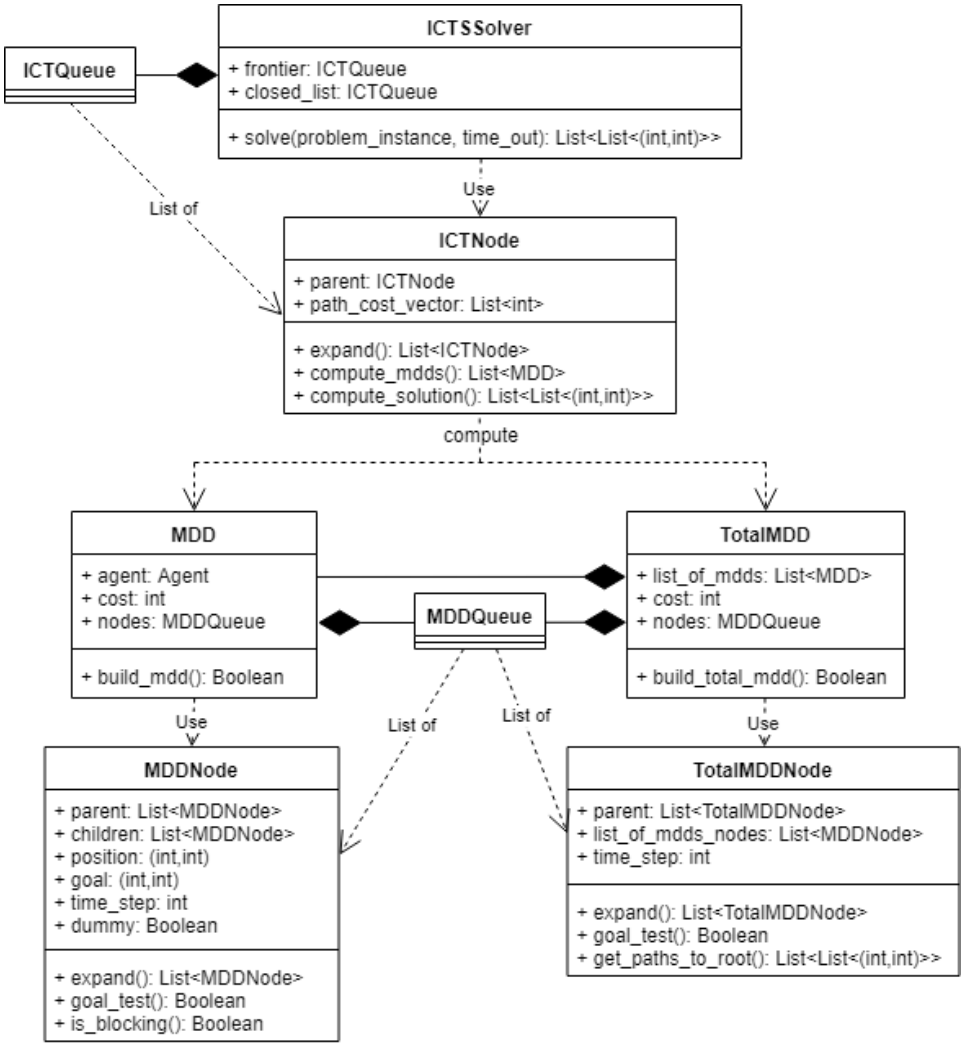


Figure 5.9: UML diagram for the ICTS solver.

merging those two diagrams. We start from the start node, and we continue to expand the children until we reach the goal node. Notice that we need to add a dummy node to MDD_1^2 in order to make the two diagrams equal in size. In the case edge conflicts are checked, node [(1,1),(2,1)] would not have any children since the only possible child will create an edge collision. Since the goal node is found, a solution exists, and the paths are reconstructed following the nodes from the leaves to the root. It's possible that more than one solution can be found. In this case, we just return the first one.

Let's now see the implemented classes for this algorithm. In Figure 5.9 we show the UML diagram. First, the class *ICTSSolver* takes care of performing

a search over the *ICT nodes*. The *ICTQueue* it's simply a queue keeping those nodes. Then, for each of these nodes, we compute the single-agent MDDs and the total MDD in order to check for the existence of a solution. The *MDD* class is responsible for building the MDD for the specified agent where the length of the path is given by the cost. This is done by performing a search over the *MDD nodes*. The *TotalMDD* class, instead, takes care of building the global MDD from the list of the single-agent MDDs. Here, the search is performed using instances of the *TotalMDD nodes* class, which are composed of lists of *MDD nodes*.

Algorithm 5 shows the main steps of the ICTS procedure.

Algorithm 5 ICTS algorithm

```
1: procedure SOLVE
2:     frontier ←add− {root of the ICT}
3:     while frontier ≠ ∅ do
4:         for all agent a_i do
5:             Build the corresponding MDD_i
6:         end for
7:         Search the k-agent MDD
8:         if k-agent MDD is goal then
9:             return Solution
10:        end if
11:    end while
12: end procedure
```

5.7 Conflict-Based Search (CBS)

This algorithm starts by calling the high-level search, which takes care of the construction of the constraint tree. This procedure, that is shown in Algorithm 4, starts with the root node, which is a node with no constraints. The high-level keeps all the possible nodes to expand in a frontier, and one at a time, pops the node with the smallest cost from it, and checks if that node is a goal node. If so, the solution is found, otherwise, the node is expanded and two different child nodes are generated. Those are obtained by adding new constraints to the parent constraints.

Let's now explain how the processing of a constraint tree node works. A CT node has a list of constraints that agents must satisfy. The low-level takes care of computing, for each agent a_i, the shortest path from its starting

Algorithm 6 High-level of CBS

```
1: procedure HIGH_LEVEL_SEARCH
2:     root_node ← ConstraintTreeNode()
3:     root_node.constraints ← ∅
4:     frontier ←add root_node
5:     while frontier ≠ ∅ do
6:         N ← best node from frontier
7:         solution ← N.low_level_search()
8:         if N is goal node then                ▷ solution has no conflicts
9:             return solution
10:        end if
11:        C ← first conflict in N                ▷ can be a vertex or edge conflict
12:        for all agent a_i in C do
13:            A ← ConstraintTreeNode()
14:            A.constraints ← N.constraints + pop constraint from C
15:            frontier ←add A
16:        end for
17:    end while
18: end procedure
```

position to its goal position, without violating any constraint associated with a_i. The search is done by performing a modified single-agent A*, which simply takes into account also a list of constraints. Those represents positions that cannot be occupied at a certain time by the agent. The search simply discards those positions. Once a consistent path has been found for each agent with respect to its constraints, these paths are then validated with respect to each other. The validation is performed by simulating the set of paths. If all agents reach their goals without any conflict, this node is declared as goal node. Otherwise, it means that a conflict has been found.

A vertex conflict is saved in the form (a_i, a_j, v, t), where a_i and a_j are the two agents involved in the conflicts, v represents the position of the vertex where the conflict occurs, and t the time step when it happens. From a conflict of this type, the two constraints added to the children are (a_i, v, t) and (a_j, v, t). The first prohibits agent a_i to go in position v at time step t, and the second similar, but for agent a_j.

In case also the edge conflicts are considered, we keep an additional list for those constraints. In case an edge conflict is found, the two children

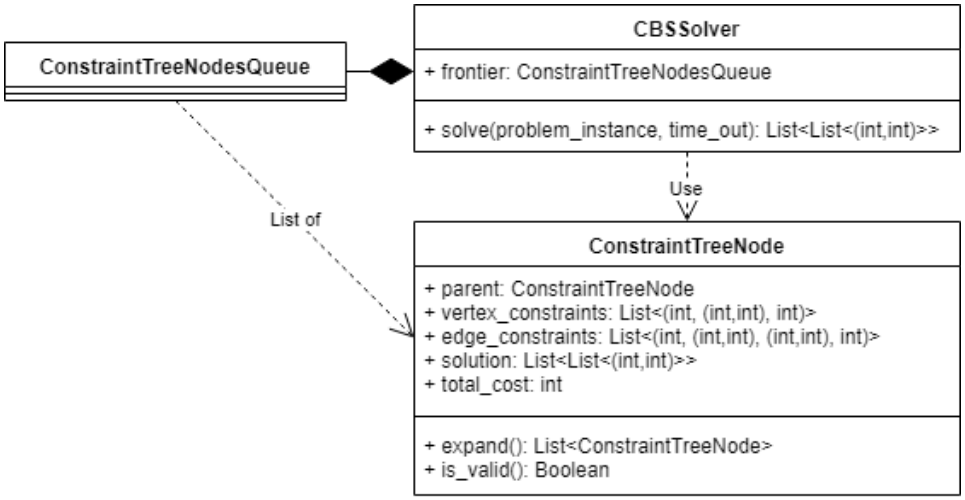


Figure 5.10: UML diagram for the CBS solver.

constraints will be in this form: (a_i, *pos_i*, *pos_f*, *ts_f*) and (a_j, *pos_i*, *pos_f*, *ts_f*). The first one says that agent a_i cannot go from *pos_i* to *pos_f* at the time step *ts_f*. This time step represents when it arrives in the second location, not when it starts moving.

Conflicts are checked using the *check_conflicts_with_type()* method. It returns a pair where the first element is a string representing the type of conflict found (vertex conflict or edge conflict), while the second is a list containing the two children constraints.

Notice that when the low level search is called on a node which is not the root, it recomputes the path only for the agent involved in the new constraint, since the other agents have the same constraints, and so, their solution paths are still the same.

Figure 5.10 shows the UML diagram for the CBS solver.

5.8 M*

M* is similar to the multi-agent A* implementation. The main difference is that M* restricts the set of possible successors of a vertex based on the collision set. Let's start with the *MStarState*. This class extends the class *MultiAgentState*, so it inherits all properties and methods from its parent. The only additional attribute is the *collision set*. This set contains the agents which are involved in a conflict in this state or in one of its successors. This set indicates the robots allowed to move in any possible direction. All other robots must follow their individual optimal paths. So, it's like A*, but

instead of considering all the possible actions for all agents, it considers all possible actions only for the agents not in the collision set, and the others will move according to their optimal paths.

The collision set is updated using a *backpropagation set* for each vertex. In our case, this set will simply contain the parent state.

Now we explain the algorithm in details. It starts with the start state where each agent is in its starting position and the collision set is empty. So, each agent will follow its individual optimal path. While the frontier is not empty, we pop a state from it and test if it is a goal state. If so, the solution is found and we just return it. Otherwise, we expand it. The *expand()* method takes for the agents not in the collision set their optimal move, while for the others will consider all their possible moves. All the possible combinations between those state are considered, and a set of children states is generated. For each one of these, we update its collision set based on the conflict it has, and we backpropagate that collision set to all the ancestor states. In the case their collision set changes due to new agents added to it, they must be re-inserted in the frontier in order to be re-expanded. Then, only the children states with no conflicts are added to the frontier. The procedure is shown in Algorithm 7 and 8.

Algorithm 7 M* Back Propagation function

```
1: procedure BACK_PROPAGATE(v_k, v_l) ▷ v_k is the parent, v_l is the new child
2:     C_k ← v_k.collision_set
3:     C_l ← v_l.collision_set
4:     if C_l ⊈ C_k then
5:         C_k ← C_k ∪ C_l
6:         if v_k ∉ frontier then
7:             frontier ←(add)— v_k                ▷ We need to re-expand the state
8:         end if
9:         for all v_m in v_k.back_set do
10:            BACK_PROPAGATE(v_m, v_k)            ▷ Backpropagate upward
11:        end for
12:    end if
13: end procedure
```

Algorithm 8 Pseudocode for M*

```
1: procedure SOLVE
2:     start_state ← MStarState()
3:     start_state.collision_set ← ∅
4:     frontier ←add start_state
5:     while frontier ≠ ∅ do
6:         N ← best state from frontier
7:         if N is goal state then
8:             return solution
9:         end if
10:        candidate_list ← [ ]
11:        for all single_agent_state s_i do
12:            if a_i in N.collision_set then          ▷ Consider all moves
13:                candidate_list ←add s_i.expand()
14:            end if
15:            if a_i not in N.collision_set then      ▷ Consider the optimal
16:                candidate_list ←add s_i.expand_optimal()
17:            end if
18:        end for
19:        candidate_states ← cartesian product of the candidate_list
20:        for all state in candidate_state do
21:            update state.collision_set
22:            BACK_PROPAGATE(N, state)
23:            if state.collision_set = ∅ then
24:                frontier ←add state
25:            end if
26:        end for
27:    end while
28: end procedure
```

Chapter 6

Package Structure and GUI

In this section, we show how the package has been structured and how the GUI has been implemented. The code is visible on GitHub at this link https://github.com/MatteoAntoniazzi/MAPF

6.1 Package Structure

The project package structure is shown in Figure 6.1. The *MAPF Solver* package contains all the requirements for solving a MAPF problem. First, the *Utilities* package is used for creating the problem and contains some common classes useful for the different algorithm. Here, we have all the classes needed for the set up of a MAPF problem, some used for solving single-agent problems, and others that help the multi-agent algorithms in their computation. Package *Search Based Algorithms* contains all the solvers treated in this thesis, while the *Heuristic* package contains the implementation of the two possible ways to compute the heuristic. The *GUI* package takes care of the visualization of the solution and everything related to the graphic aspects of our application. *Maps* is a folder containing all the maps and scenario files and *Experiments* contains some test functions useful for analyzing the algorithms.

6.2 GUI

For the implementation of the Graphic User Interface of our application, we decide to use the library Tkinter. It is Python's default GUI library. Tkinter is not the only GuiProgramming toolkit for Python. It is, however, the most commonly used one, and it is available on most Unix platforms, as well as on Windows systems.

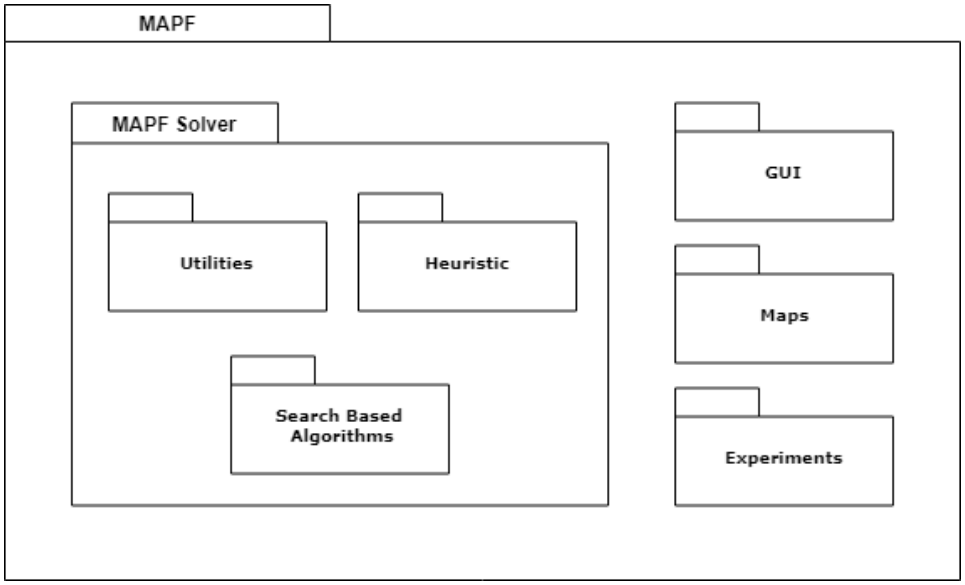


Figure 6.1: UML of the package structure.

6.2.1 MAPF Solution Representation

We start from the visualization of the map and the agent paths. Figure 6.2 shows the graphical output of our implementation.

Let's start from the map. The cells are simply rectangles built with the function *create_rectangle()* applied to the canvas. Method *create_text()* is used to write the letters 'S' and 'G' standing for start and goal. An oval is used to represent the agent. The animation function takes care of showing the agents movements to reach their goal. We simply move the oval in the map and colour its footsteps. Notice that it's possible to zooming in or zooming out the map. It is useful since some benchmark maps are very big and so cells are very small. This can be done by using the mouse wheel or the laptop touchpad. We also add a scrollbar for moving the map when zoomed alternatively to the drag and drop.

In the lower part of the visualization, we insert some info and buttons. In the left, we keep info on the computation of the solution, like the sum of costs, the makespan, the number of nodes generated or expanded during the solving, and the computation time. Notice that the number of nodes keep into account the high-level of the algorithm. For example, in the case of the ICTS solver, it counts the number of ICT nodes generated or expanded. Instead, in case we are using Cooperative A* those values will be zero, since no nodes are generated in the high-level of that algorithm. Then we also

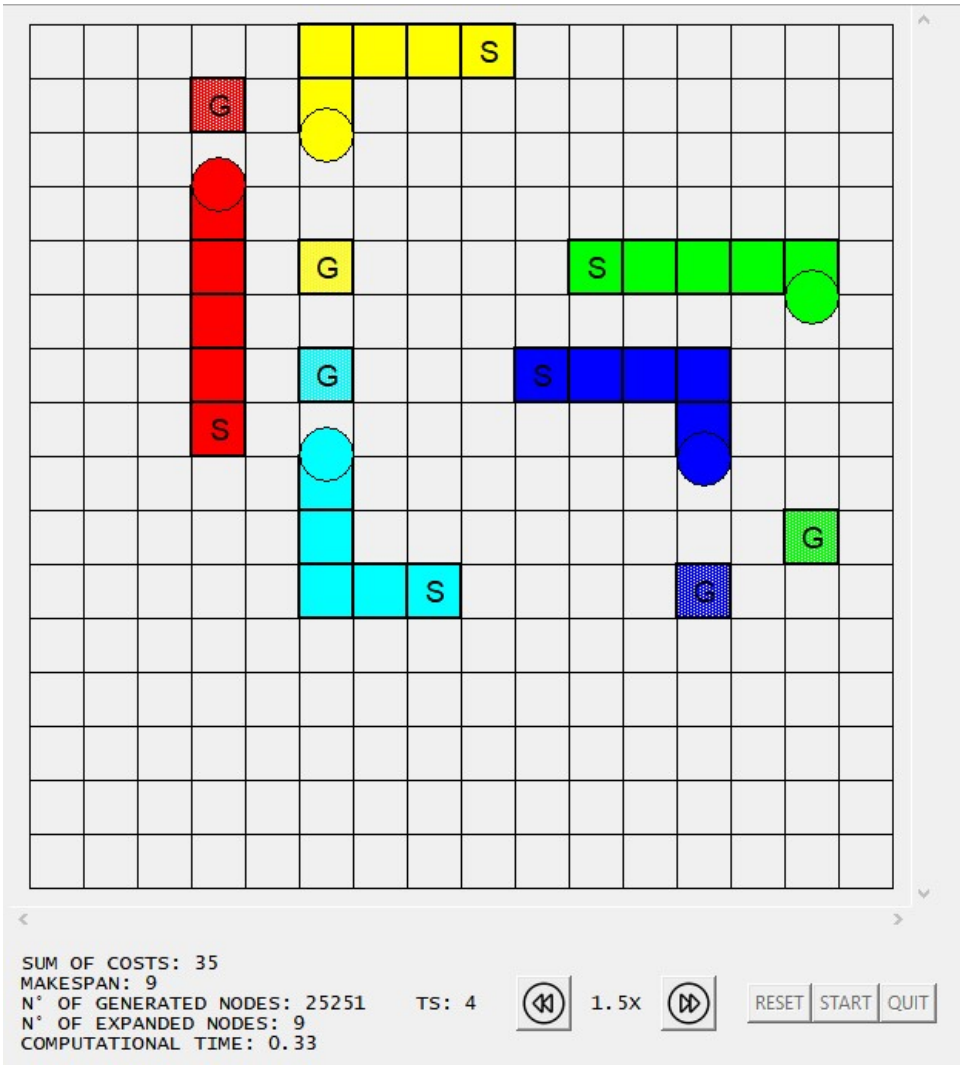


Figure 6.2: Visualization of a MAPF solution.

keep track of the actual time step of the computation.

In the right part, we have: two arrow buttons for the selection of the animation speed, one start button for starting the simulation, one reset button for restarting it, and one quit button for closing the frame.

6.2.2 Simulation tool

Let's now see how the simulation tool appear. Figure 6.3 shows the simulation menu of the tool. Here thew user can select all the inputs for our MAPF problem.

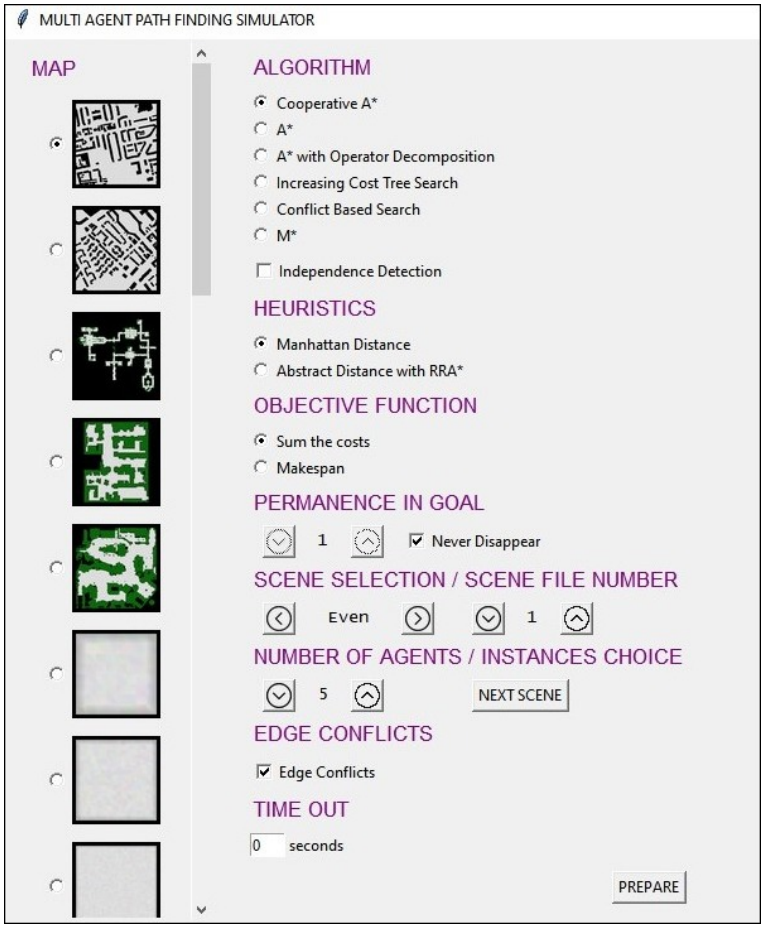


Figure 6.3: Simulation menu.

- *Map*. It is chosen from a set of predefined maps.
- *Alghorithm* used for solving the MAPF problem.
- *Independence detection framework*. If used the ID framework is used on top of the selected algorithm.
- *Heuristic* used by the solver.
- *Objective function* to minimize.
- *Permanence in goal*. If "never disappear" checkbox is selected, we assume agents stay at goal once reached it. Otherwise, they stay at the goal for a certain number of time steps before disappearing. This number, called *goal occupation time* can be selected by the user.

- *Scene selection.* Here, the user can choose which file to load. On the left, we can choose the type and on the right we can choose the number (from 1 to 25). See Section 3.2 for detailed information about the files organization.
- *Number of agents* to load in the map. The right button "next scene" is used to change agent instances from the same file.
- *Edge conflicts.* If selected, the solver considers also these conflicts.
- *Time-out.* Maximum time for running our algorithm.

The chosen scene file with the selected number of agents will be used to load a list of agents, that, with the loaded map, will create the problem instance. All the other info will be used as settings of the selected solver. Then, the *prepare* button just takes care of launching the simulation. Once computed the solution, this will be visualized on the right.

Chapter 7

Experimental results

In this section, we provide experimental results of the addressed algorithms. We perform these experiments mainly to validate our implementations, comparing our results with the ones obtained by the respective original papers and with an implementation that can be found on GitHub. In our case, we are not interested in the performance aspect, but we want only to verify that our algorithms work correctly.

7.1 Experimental problem settings

For the following experiments, we use a 4-connected grid. At each time step, every agent can perform a move action or wait. The stay at goal assumption is considered, so agents never disappear at the goal and they will block other agents from passing through it. Both vertex and edge conflicts are considered. As single-agent heuristic, we use the abstract distance heuristic computed with RRA*. Since we minimize the sum of cost, the total heuristic value is computed by doing the sum of the single-agent heuristics.

7.2 First Experiment

As first experiment, we try to reproduce an experiment performed by Sharon et al. [13] with some minor variations. The map is an 8 x 8 grid with no obstacles where the number of agents ranges from 3 to 14. We set a time limit of 450 seconds, a little higher than the time limit set in the paper. If an algorithm can't solve a problem instance within the time limit the search is halted. Agents are randomly generated, that means that start and goal locations are randomized with the only constraint that cannot exists two different start (or goal) locations which overlap. Note that we generate

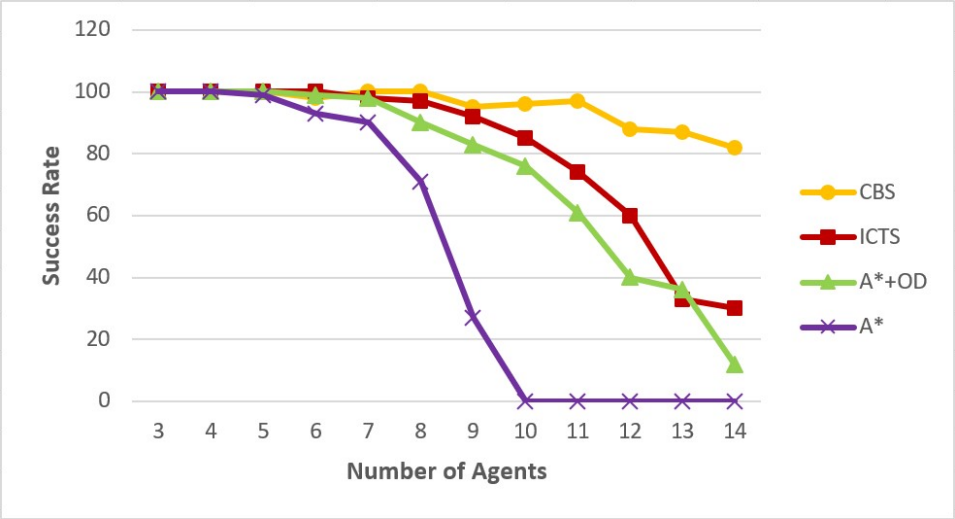


Figure 7.1: Success rate on an 8 x 8 grid with no obstacles.

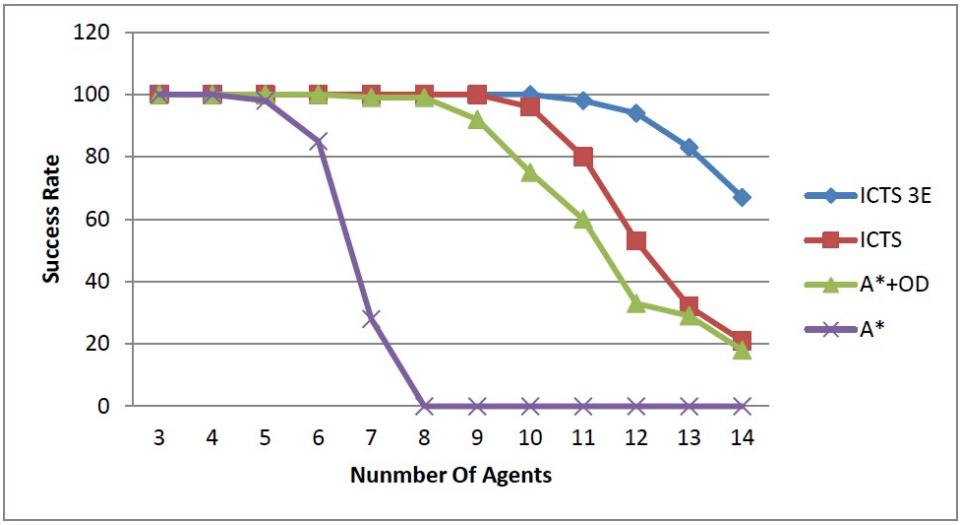


Figure 7.2: Results taken from Sharon et al.'s paper [13]. It shows the success rate on an 8 x 8 grid with no obstacles.

agents in a different way with respect to the paper, which uses the *coupling mechanism*. This procedure tries to reduce the noise and variance of the random generation, by means of a preprocessing phase on the agents. This consists in activating the ID framework on the agents and selecting only the k agents that conflict with each other.

In this experiment, we want to study the behaviour of the following algorithms: A*, A* with operator decomposition, and ICTS. For completeness, we include also the CBS algorithm. Both Figures 7.1 and 7.2 show the suc-

cess rate, i.e., the percentage of instances that are solved within the time limit, over 100 random instances. The first figure shows our results, while the second shows the ones obtained in the Sharon et al.'s paper.

We can see that A* doesn't work very well when the number of agents is more than 8. In fact, A* is limited by the exponential growth in the number of neighbours of a given vertex as the number of robots increases. A*+OD works better than A*, since it generates much less nodes. ICTS is able to solve more instances than both A* and A*+OD in the given time. CBS, in this case, outperforms all the other algorithms.

If we compare our results with those by Sharon et al. [13], we can see that, even if we generate agents randomly, the output charts are very similar. In our experiment A* chart collapse to 0 at 10 instead of 8, and ICTS can solve a bit more instances, but overall they have the same trend.

We now compare the number of nodes generated by the various algorithms. Notice that the number of generated nodes is calculated differently for each algorithm. For A* it is the traditional number of generated states. For A*+OD it includes both full states and intermediate states. For ICTS it counts the number of ICT nodes generated, while for CBS the number of nodes generated by the high level of the algorithm, i.e., the number of CT nodes.

Table 7.1 shows the number of generated nodes for each algorithm obtained in our implementation, while Table 7.2 shows the ones obtained in Sharon et al.'s paper. The cost column represents the average solution cost of the optimal solution. We can notice that the number of generated nodes in A* and A*+OD is higher in our case. This can be due to the fact that we use the standard version of A*, and so we don't perform duplicates detection in the frontier. This means that we add to the frontier nodes even if they are already present. This speeds up the process, but the count of generated nodes results slightly more. In ICTS, instead, the result is much smaller, probably due to the fact that the number of generated nodes of the paper corresponds to the summation of the number of total MDD nodes visited by all the calls to the low-level search.

We now perform the same experiment on a 3 x 3 grid. In this case, we slightly modify the A* code in order to avoid counting duplicate nodes in the count of the generated nodes. We add a duplicate detection check in the frontier. So, now, when a new node is generated we have two cases. If the same node is not already present in the frontier it is simply added to it. Otherwise, if the node is already present in the frontier, we compare the g-value of the two nodes, and only if the g-value of the new node is smaller, we replace this node to the other.

		Nodes generated			
k	Cost	A*	A*+OD	ICTS	CBS
3	15.4	770	140	1	2
4	20.6	5475	240	1	3
5	26.4	37275	435	4	8
6	32.1	287445	3570	5	15
7	38.4	946910	7615	16	45
8	42.7	NA	15640	12	200
9	45.7	NA	26340	55	2875
10	51.1	NA	33165	57	4500

Table 7.1: Nodes generated on a 8 x 8 grid.

		Nodes generated		
k	Cost	A*	A*+OD	ICTS
3	14.7	409	90	16
4	20.3	2756	303	31
5	26.1	>19631	933	94
6	29.9	>78432	2287	143
7	36.2	>176182	4762	372
8	41.0	NA	12935	645
9	46.7	NA	46565	3826
10	52.3	NA	>106181	24320

Table 7.2: Results on a 8 x 8 grid taken from Sharon et al.'s paper [13].

Table 7.3 shows our obtained results, while Table 7.4 is taken from the paper. The results show that ICTS is better than the A* variants for k less or equal than 7. Instead, if k is equal to 8 agents, both A* and A*+OD clearly outperform ICTS. An observation needs to be done for performance of A*+OD with respect to A*. In fact, as mentioned by Sharon et al.[13], A*+OD is usually better than A*, but in this case it is the opposite, because the graph is dense and so, the branching factor of both A* and A*+OD plays a smaller role than the depth of the search tree, which is larger for A*+OD because it has intermediate states.

Regarding the cost column, we notice that, even if agents are generated differently, we obtain almost the same average cost of the solutions. So, we can conclude that these results have validated our implementations of A*, A*+OD and ICTS.

		Nodes generated		
k	Cost	A*	A*+OD	ICTS
2	3.6	24.5	21	2
3	5.6	94	38	2
4	7.9	295	87	10
5	10.5	868	392	100
6	13.8	2342	1174	563
7	17.5	4563	7480	3534
8	23.2	9502	30373	NA

Table 7.3: Nodes generated on a 3 x 3 grid.

		Nodes generated		
k	Cost	A*	A*+OD	ICTS
2	3.6	23	17	1
3	5.5	90	38	2
4	7.8	294	102	6
5	10.7	980	425	31
6	13.8	2401	1383	204
7	18.6	6050	7105	2862
8	23.6	11055	35288	79942

Table 7.4: Results on a 3 x 3 grid taken from Sharon et al.'s paper [13].

7.3 Second Experiment

In this experiment, we want to test the A* algorithm and the CBS algorithm using this time the coupling mechanism for generating the agents. In order to do this, we applied the ID framework on top of A*. Then, we simply launch the problem with a large number of random agent instances, and whenever the ID framework detects a new collision group of k agents, we take it and put it in a bucket. Those will represents a possible agent configuration. We repeat this process until the bucket contains 100 different instances. The experiment is performed on a 8 x 8 grid and we set a time limit of 300 seconds. The number of agent range from 3 to 7. We stop the search at 7 agent, because using the A* algorithm for generating the agent instances, it takes too long. We compare our results with the ones obtained by Sharon et al. [12]. Those are shown in Table 7.5.

Notice that in this case we use the A* variant in order to avoid counting duplicate nodes. We can see that for A* the number of generated nodes of our experiment is very close to the those in the paper. For CBS, instead,

	Nodes generated			Nodes generated	
k	A*	CBS	k	A*	CBS
3	805	12	3	640	10
4	5530	23	4	3965	24
5	24198	46	5	21851	51
6	NA	120	6	92321	45
7	NA	237	7	NA	117

Table 7.5: Nodes generated on a 8 x 8 grid with agents generated using the coupling mechanism. On the left, there are the result of our experiments, while on the right the results taken from Sharon et al.'s paper are reported[12].

those are a bit higher, but overall coherent. Also this result validates our implementation of these algorithms.

7.4 Third Experiment

Now we want to see how the different algorithms behave when using the ID framework. We try to simulate the same experiment performed by Sharon et al. [13]. It consists of testing the algorithms on a number of agents ranging from 4 to 22 when the ID framework is activated. The map is, again, an 8 x 8 grid, and we set a time limit of 450 seconds. Agents are randomly generated, and we launch the experiment on 100 instances. Figures 7.3 and 7.4 show the number of instances solved for each algorithm.

We can notice that the ID framework allows an algorithm to solve more instances than the case where the framework is not used, since it breaks the problems into sub-problems simplifying the computation. This helps the algorithms to solve also problems where the number of agents is large. In fact, we can see how the trends shift to the right for all the algorithms. For example, A*, that when used alone it isn't able to solve any problem for 10 or more agents, with ID framework can solve a lot more instances. The chart for A* collapse now when the number of agents is approximately 22.

If we compare our results with Sharon et al.'s results, we can notice that our success rate is a bit lower for each algorithm, but overall the trends are similar. This difference can be due to the fact that we implement the simple version of the ID framework. In fact, as explained in [15], this can be improved by trying to find alternate paths before merging two agents. So, when two agents collided, instead of merging the two problems immediately, we should try to find alternate routes having the same cost. Even in this case we perform the same experiment on CBS, which gives the best performance.

Figure 7.3: Success rate on an 8 x 8 grid with no obstacles when ID framework is activated.

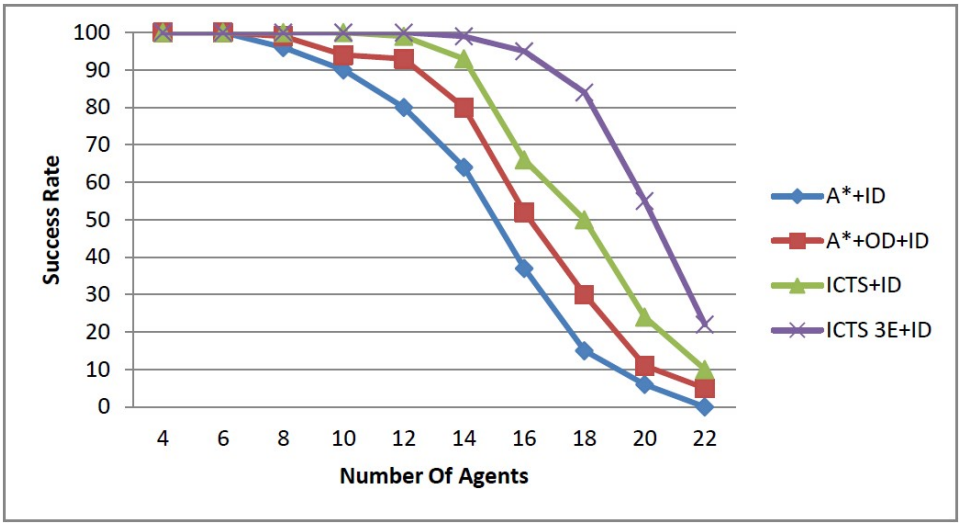


Figure 7.4: Results taken from Sharon et al.'s paper [13]. It shows the success rate on an 8 x 8 grid with no obstacles when ID framework is activated.

7.5 Fourth Experiment

As a fourth experiment, we want to compare the M* algorithm with the standard A* algorithm and its enhanced version A* with operator decomposition. We perform the same experiment of the one in the Wagner Choset's paper [19]. This time we use a 32 x 32 grid, with a 20% probability of a given cell being marked as an obstacle. Unique initial and goal positions for each robot are chosen randomly like in the previous experiments. Configurations,

Figure 7.5: Success rate on an 32 x 32 grid with obstacle probability of 20%.

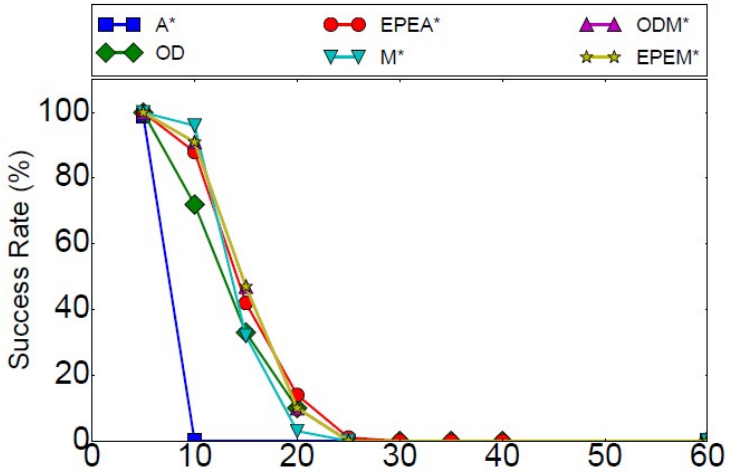


Figure 7.6: Results taken from Wagner and Choset's paper [19]. It shows the percentage of trials in which a solution was found within 5 minutes, in a 32 x 32 four-connected grid world.

where it's impossible for one agent to reach its goal due to some obstacles, are not considered. Time limit is set to 450 seconds and 100 random environments are tested for a given number of agents. Figures 7.5 and 7.6 show the success rate on those instances.

As expected, also in this case A* demonstrated the worst performance, being unable to find solutions for problems of 10 or more robots. M* solved the most problems showing performance substantially superior to A*. Comparing our results with those in the paper we can see that are very close. The only difference lies in the fact that in the paper for k equals to 15 the

k	Cost	Nodes	k	Cost	Nodes
3	15.8	3	3	15.7	4
4	21.0	6	4	21.0	6
5	26.3	14.5	5	26.4	15
6	31.5	24	6	31.4	26
7	36.7	42.5	7	36.9	39.5
8	42.4	74.5	8	42.4	70.6
9	47.7	165	9	47.6	149.5
10	53.2	208	10	53.1	197.4

Table 7.6: Nodes generated on a 8 x 8 grid. On the left there are the result of our experiments, while on the right the ones using the program taken from GitHub.

A*+OD performances are the same as the M* performances, and tends to be better for k equals to 20. In our experiments, instead, M* always gives us better results than A*+OD. This is due to the fact that our implementation of A*+OD is a bit slower. If we increase the time limit we would see an increase in the success rate, making the results more similar to those of the paper.

7.6 Fifth Experiment

For this experiment, we use this implementation of the CBS algorithm: https://github.com/teobellu/build-mapf. The goal of this experiment is to check that our CBS implementation works properly, by comparing the number of generated nodes of our algorithm with the program taken from GitHub. We use an 8 x 8 grid, with agents randomly generated, and for this experiment we allow edge conflicts. We launch the simulation on 100 random instances and we average the results. Table 7.6 shows the results obtained using the two implementations.

We can see that in both the implementation the obtained values are very similar. The costs are almost identical and the same for the generated nodes. We can conclude that the two implementations work in the same way, and so, this result validates our implementation of the CBS algorithm.

7.7 Sixth Experiment

For the last experiment, we want to analyze the performances of the Co-operative A*, the only sub-optimal algorithm we have implemented. For this experiment, we compare the results with the ones obtained by Silver

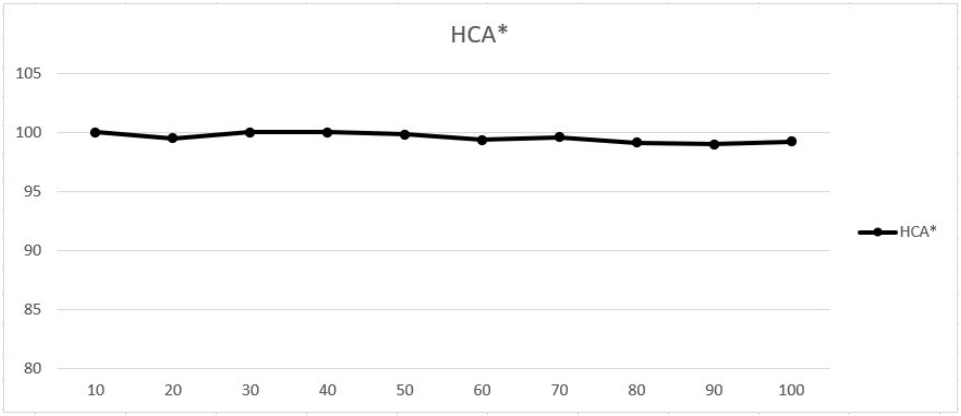


Figure 7.7: Percentage of agents that successfully reach their destination.

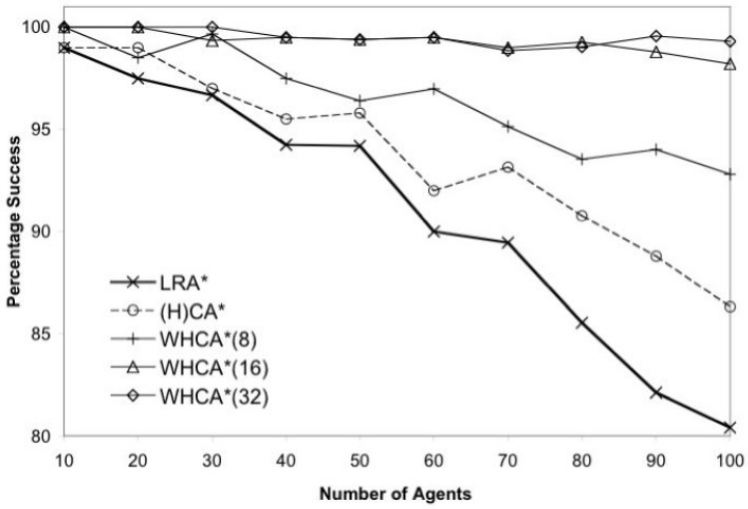


Figure 7.8: Results taken from Silver paper [14]. It shows the percentage of agents that successfully reach their destination.

[14]. This is performed by launching the algorithm on a series of 10 different maze-like environments. Each environment consists of a 32 x 32 grid, with obstacles placed down in 20% of the grid locations. Any disconnected sub-region is additionally filled with obstacles to guarantee a fully connected map. Agent positions are randomly generated like in all the previous experiments. An agent is considered successful if it is able to reach its destination within 100 turns. Figures 7.7 and 7.8 show the percentage of agents that successfully reach their destinations.

We can see that in our experiments the percentage of agents that has success is always near 100%.

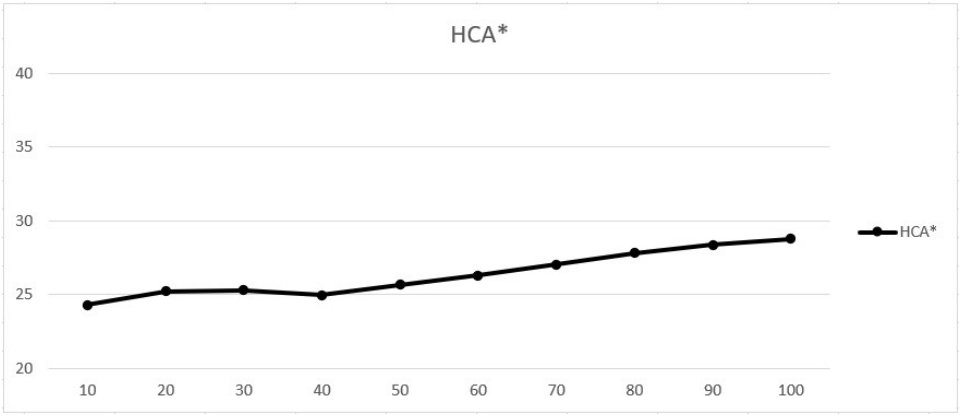


Figure 7.9: Average path length for each agent.

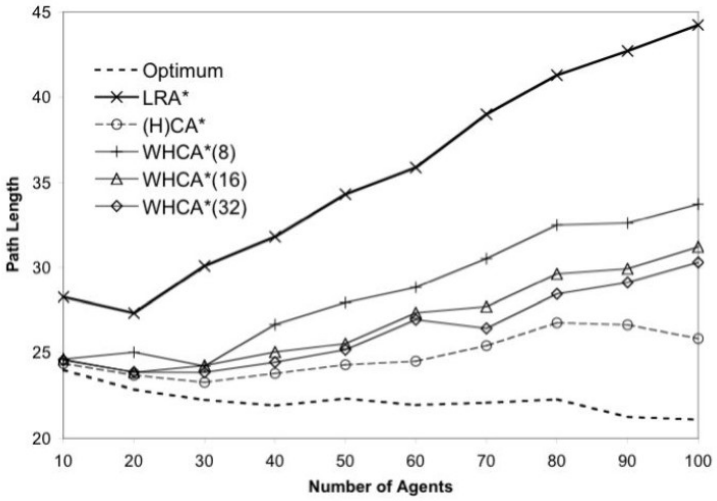


Figure 7.10: Results taken from Silver paper [14]. It shows the average path length for each agent.

We now compare the average path length (Figure 7.9 and Figure 7.10). In this case, our outcome results consistent with the paper's experiment with an average length path around 25/26 steps.

Chapter 8

Conclusions

In this thesis we have developed a tool containing the most common algorithms for solving the multi-agent path finding problem. We have seen that there are a lot of algorithms that solve this problem, each with its own pros and cons. For example the Cooperative A* is not an optimal algorithm, but is very useful when we want to find a solution fast, or when the number of agents is very large. Multi-agent A*, instead, finds the optimal solution, but is expensive since the search space grows a lot, especially when the branching factor is big, which value depends on the number of agents. For improving this algorithm we have seen the use of Operator Decomposition, which reduces the number of generated nodes, or of the M* algorithm, which tries to generate only the optimal nodes. Other algorithms, which work in different state spaces, like Increasing Cost Tree Search and Conflict-Based Search, have very good performance. Those are the ones that give the better results in our experiments. We have also seen the Independence Detection framework, which improves the performance of the algorithm used on top of it, and allows to solve problem instances where the number of agents is big. We have also performed a series of experiments in order to compare the results of our implementations of the algorithms to the ones from their original papers. Those tests appear to be mostly coherent with the results we were expecting, and they generally validate our work.

The goal of our tool is to put all these algorithms together, in such a way that users can understand the MAPF problem and the different ways to solve it. The Python language is a common and simple programming language, and students interested in learning those algorithms can easily read the code for understanding the various procedures that lead to the solution. Alternatively, they can test the algorithms on some benchmark maps, analyzing and comparing the different paths they generate.

Bibliography

[1] Maren Bennewitz, Wolfram Burgard, and Sebastian Thrun. Finding and optimizing solvable priority schemes for decoupled path planning techniques for teams of mobile robots. *Robotics and Autonomous Systems*, 41(2-3):89–99, 2002.

[2] Guy Desaulniers, André Langevin, Diane Riopel, and Bryan Villeneuve. Dispatching and conflict-free routing of automated guided vehicles: An exact approach. *International Journal of Flexible Manufacturing Systems*, 15(4):309–331, 2003.

[3] Kurt Dresner and Peter Stone. A multiagent approach to autonomous intersection management. *Journal of Artificial Intelligence Research*, 31(1):591–656, 2008.

[4] Esra Erdem, Doga Gizem Kisa, Umut Öztok, and Peter Schüller. A general formal framework for pathfinding problems with multiple agents. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, pages 290–296, 2013.

[5] 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 *Proceedings of the Tenth International Symposium on Combinatorial Search*, pages 29–37, 2017.

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

[7] Hang Ma, Daniel Harabor, Peter J. Stuckey, Jiaoyang Li, and Sven Koenig. Searching with consistent prioritization for multi-agent path finding. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence*, pages 7643–7650, 2019.

[8] Hang Ma, Craig Tovey, Guni Sharon, T. K. Kumar, and Sven Koenig. Multi-agent path finding with payload transfers and the package-exchange robot-routing problem. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 3166–3173, 2016.

[9] Ravi Narasimhan. Routing automated guided vehicles in the presence of interruptions. *International Journal of Production Research*, 37(3):653–681, 1999.

[10] Dennis Nieuwenhuisen, Arno Kamphuis, and Mark H. Overmars. High quality navigation in computer games. *Science of Computer Programming*, 67(1):91–104, 2007.

[11] Lucia Pallottino, Vincenzo Scordio, Antonio Bicchi, and Emilio Frazzoli. Decentralized cooperative policy for conflict resolution in multivehicle systems. *IEEE Transactions on Robotics*, 23(6):1170–1183, 2008.

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

[13] 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.

[14] David Silver. Cooperative pathfinding. In *Proceedings of the First Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 117–122, 2005.

[15] Trevor Scott Standley. Finding optimal solutions to cooperative pathfinding problems. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, pages 173–178, 2010.

[16] Roni Stern, Nathan R. Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne T. Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, T. K. Satish Kumar, Eli Boyarski, and Roman Bartak. Multi-agent pathfinding: Definitions, variants, and benchmarks. In *Proceedings of the Twelfth International Symposium on Combinatorial Search*, pages 151–159, 2019.

[17] Pavel Surynek. Towards optimal cooperative path planning in hard setups through satisfiability solving. In *PRICAI 2012: Trends in Artificial Intelligence*, pages 564–576, 2012.

[18] Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski. An empirical comparison of the hardness of multi-agent path finding under the makespan and the sum of costs objectives. In *Proceedings of the Ninth Annual Symposium on Combinatorial Search*, pages 145–147, 2016.

[19] Glenn Wagner and Howie Choset. M*: A complete multirobot path planning algorithm with performance bounds. In *Proceedings of the International Conference on Intelligent Robots and Systems*, pages 3260–3267, 2011.

[20] Jingjin Yu and Steven M. Lavalle. Planning optimal paths for multiple robots on graphs. In *Proceedings of the International Conference on Robotics and Automation*, pages 3612–3617, 2013.