UCLouvain

École polytechnique de Louvain

Design and Implementation of a Multi-Agent Pathfinding library in C++ for Automated Logistic Warehouses

Authors: **Arthur Mahy, Julien Mans**
Supervisors: **Pierre Schaus, Vianney Coppé**
Readers: **Pierre Schaus, Vianney Coppé, Raphaël Negri**
Academic year 2022–2023
Master [120] in Computer Science and Mathematical Engineering

Abstract

This thesis elaborates on an associated c++ library in which the most common optimal algorithms for solving the Multi-Agent PathFinding (MAPF) problem were implemented. It contains a summary of these methods, implementation details and numerical experiments to compare their performance and check the impact of the various enhancements on success rate and running time. The implemented algorithms and their enhancements include A* in space or space-time search, in single-agent and multi-agent search, with and without operator decomposition; Independence Detection (ID) with or without Conflict Avoidance Table (CAT), as well as its Enhanced version (EID); Conflict Based Search (CBS) with or without CAT, with basic splitting or disjoint splitting.

Acknowledgments

We thank our promotor Professor P. Schaus as well as our co-promotor V. Coppe for his implication, feedback and presence throughout this master thesis.

We also thank all our professors and UCLouvain as a whole for those five years of formation leading to this master thesis, marking the boundary between our learning journey and our future careers as engineers.

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

The global problem of warehouse automation is to move inventory pods from their storage locations to the inventory stations where the needed goods are removed from the inventory pods (to be boxed and eventually shipped to customers) and then back to storage. One of the underlying problems is to plan the trajectory of the robots from their current position to their target (station or storage) through an optimal path and avoiding collisions between individual robots. The problem is that there are many inventory pods and many robots sharing the same environment. This is known as the Multi-Agent Path Finding (MAPF) problem.

The general goal of this master thesis was to design and implement a C++ library[1] allowing to solve the Multi-Agent Path Finding problem using various objective functions, distance heuristics and algorithms with various togglable enhancements and relying on useful abstractions so that component can be easily reused to build more complex algorithms. Among those algorithms, an important goal was to implement Conflict Based Search along with its most common enhancements: Disjoint Splitting and the use of a Conflict Avoidance Table.

Besides CBS and essential components of it such as A* and RRA* (for the Optimal heuristic), we also implemented Independence Detection with the ability to use other search algorithm as a low-level search, as well as Cooperative A* which we didn't spend much time on as it is not optimal.

Finally, we ran all those algorithms and combinations on many scenarios to quantify exactly how efficient the enhancements are and how algorithms compare to each other, identifying which one is adapted for which type of map.

[1] Accessible from `https://github.com/JMans15/TFE_MAPF`

Chapter 1

Literature Review

1.1 Artificial Intelligence [28]

This writing falls within the field of Artificial Intelligence (AI). But what is it really? So far, there is no generally accepted definition of Artificial Intelligence because there are so many different ones. In fact, it is an "umbrella term". "Umbrella term" refers to a word or phrase that encompasses a broad or general category, encompassing multiple specific subcategories or related concepts. It is used to group together various related topics or ideas under a single overarching term. Technologies enter and exit the AI 'umbrella' regularly.

The AI topics are numerous; we can mention cognitive modelling, optimisation, knowledge representation, machine learning, data mining, natural language processing, neural networks, planning, scheduling, reasoning, robotics, game theory, search, computer vision, perception, and many others. These topics can be categorised into four types of systems.

- **Human thinking.** The Cognitive Modelling approach aims to develop systems that can think like humans. It involves automating activities associated with human thinking, such as decision-making, problem-solving and learning [8]. The goal is to construct theories that explain how humans think. This approach leverages the synergy between computer models from AI and experimental techniques from psychology, introspection and brain imaging. However, existing AI approaches are typically not directly based on cognitive models, making it challenging to translate these theories into computer programs and often resulting in performance issues.

- **Human acting.** The approach of systems that act like humans is often associated with the Turing Test. This approach focuses on creating machines that can perform tasks requiring intelligence when performed by humans [18].

The Turing Test involves testing whether a program can achieve human-like performance in cognitive tasks. The Total Turing Test goes beyond mere cognitive tasks and includes interactions with people and objects. Achieving human-like performance requires various AI techniques such as natural language processing, knowledge representation, automated reasoning and machine learning. However, the Turing Test is not considered very relevant to practical AI applications.

- **Rational thinking.** The "Law of Thought" approach involves studying mental faculties using computational models [11]. This approach relies on abstract "laws of thought" based on logic and mathematical logic. It requires formal representations of problems and knowledge and utilises formal reasoning systems to derive solutions. However, this approach is not always directly applicable to practical AI because of the difficulty in fully formalising real-world complex problems. While problems may be solvable "in principle" according to formal logic, they may not be easily solvable "in practice".

- **Rational acting.** The rational agent approach is “the branch of computer science that is concerned with the automation of intelligent behaviour” according to Luger and Stubblefield (1993) [24]. This approach focuses on creating agents that consistently make decisions that align with what is considered "the right thing." These agents perceive information from the environment, employ knowledge and reasoning to determine suitable actions and execute them to achieve their goals. In the real world, it can be challenging to precisely define the objective or goal of an agent. This difficulty gives rise to the value alignment problem, which entails the challenge of ensuring that the goals programmed into the machine align accurately with our true preferences. The rational agent is not limited to human actions or human environments and does not solely focus on human thought processes. Furthermore, this approach is not confined to using only the laws of logic. Instead, the primary criterion is producing rational behaviour. The rational agent is considered the standard model of AI and is the chosen approach in many research endeavours in this field. It is the approach adopted by our library and the continuation of this writing.

1.2 What is an agent ? [28]

Agents are used to provide a consistent viewpoint on various topics in the field of AI and require essential skills to perform tasks that require intelligence. An agent is an entity that interacts with its environment. It perceives information from its environment through sensors and performs actions using actuators. Sensors allow

the agent to gather data from the environment and actuators are responsible for executing actions or making changes in the environment. Together, the agent's sensors and actions enable it to engage with and respond to its surroundings.

For example, a robot agent could use sensors like cameras, infrared and bumpers, while actuators could include grippers, wheels, lights, speakers and motors. In contrast, a software agent relies on software functions as its sensors and actuators for perceiving information and executing actions.

As previously mentioned, a rational agent does "the right thing" by performing the action that leads to the best outcome. Therefore, it becomes necessary to define what constitutes "the right thing." Determining how to measure the "best outcome" is crucial. We refer to this as the performance measure of an agent. The performance measure embodies the criterion for success of an agent's behaviour. For each possible percept sequence, a rational agent should choose an action that is expected to maximise its performance measurement. This decision is based on the evidence provided by the percept sequence as well as any existing knowledge the agent has. We can describe an agent with the PEAS system.

1.3 Pathfinding

Pathfinding is a classic problem in AI and mainly involves planning a route to a destination that avoids obstacles. Some of the most well-known applications of pathfinding algorithms include: [17]

- Computer Games: Currently, pathfinding finds its most widespread application in video games, where the computer guides simulated opponents across the game map. This proves to be the most intricate use of pathfinding as the computer must factor in various considerations such as the desirability of certain paths regardless of their length (some paths may be riskier). It must also handle multiple dynamic objects (real-time strategy (RTS) games). In certain instances, the computer may control these dynamic objects too necessitating an advanced program to account for five different entities and optimise their paths to minimise overall latency, further complicated by considering unit priorities and other relevant factors.

- Map Applications and GPS Navigation: GPS navigation systems use pathfinding algorithms to calculate the shortest or fastest routes for users to reach their desired destinations.

- Robotics and Autonomous Vehicles: Pathfinding is essential for robots and autonomous vehicles to navigate their environments efficiently and safely.

Robots need to plan optimal paths to reach their destinations while avoiding obstacles and potential hazards.

- Exploration Robots: Pathfinding is finding a growing and intriguing application in exploring hard-to-reach or hazardous locations, which are challenging for humans to access but hold great potential for profitable exploration by robots if they can navigate without getting lost or stuck. This trend is particularly evident in planetary exploration, with NASA increasingly opting to send robots instead of humans into space. Moreover, there is a noticeable effort to deploy robots for exploring Earth's difficult-to-reach regions, including deserts and polar areas, with the aim of searching for signs of life and intriguing anomalies.

- Network Routing: In computer networks, pathfinding algorithms are used to find optimal routes for data packets to travel from a source to a destination, reducing latency and congestion.

- Air Traffic Management: Pathfinding algorithms help manage and optimise air traffic by determining safe and efficient routes for aircraft-towing vehicles.

- Virtual Reality and Augmented Reality: Pathfinding is employed to enable virtual characters or augmented reality elements to navigate virtual environments realistically.

- Resource Allocation in Logistic Warehouses: Pathfinding algorithms can be used to optimise the movement of materials or products within manufacturing facilities, reducing production time and costs. An automated logistic warehouse is a facility that uses advanced technologies and robotic systems to perform various tasks related to inventory management, order fulfilment, and storage of goods. In an automated logistic warehouse, the traditional manual labour involved in moving, sorting, and organising items is replaced or complemented by automated machinery and systems.

1.3.1 PEAS description of an agent

In this section, a description of the rational agents involved in a pathfinding problem is provided using the PEAS system. This system is used to categorise similar agents together and delivers the performance measure with respect to the environment, actuators and sensors of the respective agent. PEAS stands for a Performance measure, Environment, Actuator, Sensor [28]. A pathfinding problem involves finding the optimal path between a starting point and a destination in a

given environment, while avoiding obstacles and minimising certain criteria such as distance travelled or travel time.

- **Performance measure.** The performance measure in a pathfinding problem can be defined by criteria such as the length of the path found, the time taken to find the path, the efficiency of the path in terms of avoiding obstacles or the resource consumption.
- **Environment.** The environment in a pathfinding problem consists of a defined space or map where the agent operates. It includes obstacles, boundaries and the locations of the start and destination points. The properties of the environment are elaborated in section 1.3.2.
- **Actuators.** The actuators refer to the actions that the agent can take to navigate in the environment. These actions typically include moving to a neighbouring position and waiting at its current position.
- **Sensors.** The sensors allow the agent to perceive information about the environment and its current state. This can include the ability to detect obstacles, identify the current position of the agent or gather information about neighbouring cells.

1.3.2 Environment properties

We can describe the environment in which an agent interacts with various properties [28]. In the context of the pathfinding problem, this section explains some of these properties.

- **Single-agent vs. multi-agent.** The environment is considered multi-agent if it includes multiple agents, each of which is also maximizing a performance measure that depends on the actions of other agents. While the main objective of this thesis is the multi-agent pathfinding (MAPF), single-agent pathfinding will be necessary to solve the MAPF problem. In scenarios with a single agent, the A* algorithm is typically sufficient to solve the problem effectively. However, when dealing with multiple agents, special attention is required to prevent conflicts in the solution. For instance, it's essential to avoid situations where two agents are required to occupy the same space simultaneously or where agents consume more of a limited resource than what is currently available.
- **Determinisic vs. stochastic.** The environment is deterministic if the next environment state is completely determined by the current state and the

executed action. There is no randomness or uncertainty involved. It is the case in this writing.

- **Static vs. dynamic.** If the environment is allowed to change while the agent is choosing an action, the environment is dynamic. In future algorithms in this paper, we assume that the environment will not change while the search is being conducted (static environment).

- **Discrete vs. continuous.** This distinction pertains to the way time is managed. In most MAPF algorithms, it is assumed that time is divided into discrete time steps and each action has a duration of one time step. However, some algorithms have been presented in the article [6] to solve MAPF problems with continuous time.

- **Fully vs. partially observable.** An environment is fully observable when the sensors can detect all aspects that are relevant to the choice of action. In **Cooperative Pathfinding**, it is assumed that each agent has complete knowledge of all other agents and their planned routes. In the complementary problem of **Non-Cooperative Pathfinding**, agents have no knowledge of each other's plans and must anticipate their future movements. In **Antagonistic Pathfinding**, each agent strives to reach its individual goal while actively preventing other agents from reaching theirs. Agents in this scenario are not aware of the planned paths of other agents [32]. This paper focuses exclusively on the Cooperative Pathfinding problem and the environment is consequently fully observable.

1.3.3 More distinctions within the context of MAPF problems

In addition to the properties of the environment as described in the previous section, we can also make the following distinction within the context of MAPF problems.

- **Centralised vs. distributed.** Latombe (1991)[19] and Fujimura (1991)[13] classified the various approaches to multi-agent pathfinding into two distinct categories. **Decoupled or distributed planning** involves breaking down the task into independent or weakly-dependent problems for each agent (Erdmann & Lozano-Perez 1987 [12]). Every agent possesses its own computing power and various communication paradigms can be assumed such as message passing or broadcasting. By contrast, **centralised planning** assumes a single central computing power and needs to find a solution for all agents simultaneously. However, this approach faces a significant challenge because it is PSPACE-hard, meaning that solving the problem requires a prohibitive

amount of computational resources as the search space quickly becomes very large. The centralised setting also encompasses scenarios where each agent has its own CPU but complete knowledge sharing is assumed and a centralised problem solver governs all the agents. This paper focuses solely on centralised approaches (except the CA* and HCA* algorithms that has been implemented and that are decoupled planning).

- **Optimal vs. Sub-optimal.** Algorithms designed to tackle the MAPF problem can also be categorised into these two groups: optimal solvers and sub-optimal solvers with respect to the cost function. Finding an optimal solution for the MAPF problem is NP-hard as the state space expands exponentially with the number of agents. Sub-optimal solvers are commonly employed when dealing with a large number of agents. The focus of this paper is to address the problem of finding an optimal solution for the MAPF problem.

1.4 Overview of existing algorithms

Many algorithms exist to solve the MAPF problem, each one with its strengths and weaknesses, and the choice of algorithm depends on factors such as optimality requirements, scalability, computational resources and specific problem characteristics.

Here is a non exhaustive list of existing algorithms along with their brief descriptions.

- A*: The A* algorithm is a state-of-the-art algorithm, introduced in 1968 by Hart *et al.* in 1968 [15], for finding the shortest path between two nodes in a graph. It is used in a wide variety of applications, such as route planning, robot navigation, and game AI.

  It is an informed search algorithm, meaning that it uses additional information, such as the estimated distance to the goal, to guide its search.

  It works by maintaining an "open" list of nodes, in which one has to find the node that has the best chances at being on the optimal path. It does so by using a distance function called the *heuristic*, estimating the distance from a node to the origin, along with an *evaluation function*, keeping track of the length of the path taken to get to the node.

  Once a node is selected from the open list, it is expanded, the evaluation function updates its cost, its neighbours are added to the open list and their distance to the goal is estimated by the heuristic.

When an expanded node is the target node (or belongs to the set of target nodes), the algorithm is terminated.

The A* algorithm is very flexible and different heuristics and definitions of the nodes and states can yield efficient algorithms for a variety of problems. Defining vertices from the positions of multiple agents can turn the algorithm into a multi-agent algorithm, further adding a temporal dimension can make it work on moving maps or with more complex constraints, ...

- Local Repair A* (LRA*): [33] LRA* is a family of algorithms that are widely used in the video game industry. Each agent in LRA* searches for a route to the destination using the A* algorithm, ignoring all other agents except for its current neighbours. The agents then begin to follow their routes until a collision is imminent.

  When a collision is imminent, the agent that is about to collide with another agent recalculates the remainder of its route. This is called a brute-force re-planner. However, cycles are possible using this technique, so it is common to add some modifications to escape such problems.

  One modification that can be added to LRA* is to increase the agent's agitation level every time it is forced to reroute. Random noise is then added to the distance heuristic in proportion to the agitation level. As the agents behave more and more randomly, it is hoped that they will escape from the problematic area and try different routes, not unlike some known local minima-avoiding strategies in the field of optimisation.

  LRA* is known to have several severe drawbacks in difficult environments. For example, if bottlenecks occur in crowded regions, they may take arbitrarily long to be resolved. While caught in a bottleneck, agents constantly reroute in an attempt to escape, requiring a full re-computation of the A* search almost every turn. This leads to visually disturbing behaviour that is perceived as unintelligent.

- Cooperative A* (CA*): [33] CA* is a decentralised algorithm for solving the Cooperative Pathfinding problem that decouples the problem into a series of single agent searches. The individual searches are performed in three dimensional space-time, and take account of the planned routes of other agents. A wait move is included in the agent's action set, to allow it to remain stationary (only take a time step). After each agent's route is calculated, the states along the route are marked into a reservation table. Entries in the reservation table are considered impassable and are avoided during searches by subsequent agents. The reservation table represents the agents' shared knowledge about each other's planned routes. It is a sparse

data structure marking off regions of space-time. The choice of data structure is independent from the state space of the agents themselves. This allows, individual agents to vary in speed or size, the reservation table being capable of marking off any occupied region. As for the implementation, a simple one is to use a 3D grid which uses a lot of unnecessary space. An hash table hashing on a randomly distributed function of the (x, y, t) key would waste a lot less space. It is important to note that being decoupled and greedy, this algorithm will not be able to solve certain classes of problem, a greedy solution for one agent could indeed prevent another to take the optimal route. It is also sensitive to the ordering of the agents, requiring sensible priorities to be selected for good performance, for example using Latombe's prioritized planning (1991). Any admissible heuristic can be used in CA*. The base case is to use the Manhattan distance. However, this can give poor performance in more challenging environments. Ideally, a better heuristic should be used to help reduce the required computation effort.

- Hierarchical Cooperative A* (HCA*): [16] Abstraction, in search, problem solving, and planning, works by replacing one state space by another (the "abstract" space) that is easier to search. The results of the search in the abstract space are used to guide search in the original space. A natural application of this technique is to use the length of the abstract solution as a heuristic for A* in searching in the original space. However, there are two obstacles to making this work efficiently. The first is a theorem [38] stating that for a large class of abstractions, "embedding abstractions," every state expanded by blind search must also be expanded by A* when its heuristic is computed in this way. The second obstacle arises because in solving a problem A* needs repeatedly to do a full search of the abstract space while computing its heuristic. (Holte, 1996)[16] introduces a new abstraction-induced search technique, "Hierarchical A*," that gets around both of these difficulties: first, by drawing from a different class of abstractions, "homomorphism abstractions," and, secondly, by using novel caching techniques to avoid repeatedly expanding the same states in successive searches in the abstract space. Hierarchical A* outperforms blind search on all the search spaces studied

- Windowed Hierarchical Cooperative A* (WHCA*): [33] One issue with CA* and HCA* is how they terminate once the agents reach their destination. If an agent sits on its destination, for example in a narrow corridor, then it may block off parts of the map to other agents. Ideally, agents should continue to cooperate after reaching their destinations, so that an agent can move off its destination and allow others to pass. A second issue is the sensitivity

to agent ordering. Although it is sometimes possible to prioritise agents globally, a more robust solution is to dynamically vary the order of agents, so that every agent will have the highest priority for a short period of time. Solutions can then be found which would be unsolvable with an arbitrary, fixed agent order. Thirdly, CA* and HCA* must calculate a complete route to the destination in a large, three dimensional state space. When a single agent is searching for a path, it often plans and executes its plan at the same time. This helps to avoid wasting time planning for long-term contingencies that may never happen. WHCA* develops a similar idea for co-operative search. A simple solution to all of these issues is to window the search. The cooperative search is limited to a fixed depth specified by the current window. Each agent searches for a partial route to its destination, and then begins following the route. At regular intervals, the window is shifted forwards and a new partial route is computed. To ensure that the agent heads in the correct direction, only the cooperative search depth is limited to a fixed depth, whilst the abstract search is executed to full depth. A window of size w can be viewed as an intermediate abstraction that is equivalent to the base level state space for w steps, and then equivalent to the abstract level state space for the remainder of the search. In other words, other agents are only considered for w steps (via the reservation table) and are ignored for the remainder of the search.

An additional benefit of windowing is that processing time can be spread across all agents. By overlapping the windows in a search, the searches can be smoothly performed concurrently. With n agents and a window size of w, recalculating routes at the midpoint of each window, only $\frac{2n}{w}$ searches need to be performed per turn.

- Independence Detection (ID): One problem of currently known algorithms for the MAPF problem is that they are NP-hard, in fact their space and time complexities are exponential against the number of agents[1]. A trivial idea is then to try and split big problems into independent smaller ones.

  Independence detection is a technique introduced by Trevor Standley in 2010 [35] for decomposing a large multi-agent pathfinding problem into smaller ones that can be solved independently.

  ID is a high level algorithm that is used in conjunction with a complete search algorithm such as A* or CBS. In order to solve larger problems, ID partitions the agents into several smaller travel groups in such a way that the optimal paths found for each independent travel group do not conflict with the paths

[1] e.g. CBS's high level search is $\mathcal{O}(2^{kC^3})$ (k the number of agents, C the optimal cost) as derived in [14]

of other travel groups. Therefore, the paths for all travel groups constitute a solution to the entire problem.

One can actually see ID as a family of algorithms, as there are several possible improvements over the most simple implementation of it.

- Simple Independence Detection (SID): First, assign each agent to its own travel group. Then, find a path for each agent's travel group. Some of these paths probably contain conflicts with the paths of other travel groups. By simulating the actions of every agent following these preliminary paths, we can discover conflicts among the paths[2]. We then merge the first two travel groups whose paths are found to conflict and find a new set of paths for the merged travel group using the complete global search algorithm. We can repeat this process of merging two conflicting travel groups into a larger travel group until no conflicts are found. We can be sure that this algorithm will terminate because in the worst case, it will merge all the agents into a single travel group and find paths for this group using the global search algorithm. There are three eventualities.

  SID could first find a set of travel groups that are independent, in which case the algorithm terminates with a valid solution. Else, the algorithm could find a travel group for which no solution can be found, in which case the problem has no solution. Finally, the algorithm could combine all of the agents into a single travel group, in which case SID didn't find any independence and must delegate the entire problem to the global search algorithm.

- Illegal Move Table: The running time of the algorithm being dominated by the path planning for the largest group, one intuitive refinement is to avoid doing unnecessary merges, thereby hopefully minimising the size of the largest group.

  To be useful the new path must avoid any conflict with other agents, this can be done making use of an illegal move table keeping track of moves that would result in conflicts. This is not dissimilar to CA*'s reservation table, the difference being that illegal **moves** are here stored instead of reserved **tiles**.

  To ensure optimallity, the alternative paths for a travel group must have the same total cost as the initial paths for that travel group, this can be done by bounding the cost of the alternative path during the search.

[2]Which is not actually how we implemented this, see the dedicated ID implementation section

– Conflict Avoidance Table (CAT): Another intuitive refinement is to give our algorithm means of avoiding future conflicts.

  As the global search algorithm searches paths for a travel group, it is important for the algorithm to avoid other agents' paths in order to reduce the likelihood of future travel group merges and re-plan. This is done using a CAT, which keeps track of current tentative moves of agents in groups not being planned.

  There are then several ways of using said table depending on our goal. First if optimallity is required, one chooses the path with the fewest CAT violations *amongst paths of minimal cost*. If optimallity is not as important as running time, then one can choose the path with the overall fewest CAT violations. Other heuristics and combinations can also be considered to find compromises.

– Full Independence Detection (ID): The full ID algorithm starts by assigning each agent to its own travel group. It then finds an initial path for each travel group independently, guided by the conflict avoidance table. Next, ID looks for conflicts within its current set of paths. Upon detecting a conflict, ID attempts to find an alternative set of paths for one of the conflicting travel groups, ensuring that the new paths do not conflict with the other travel group. If this fails, it repeats this process with the other of the conflicting travel groups. If both attempts to find alternative paths fail, ID merges the conflicting groups and cooperatively plans a set of paths for the new travel group. All paths are planned with a constantly updated conflict avoidance table to minimise future conflicts.

Standley [35] also gives several possible optimisations:

– Group Order for Replanning: Choosing which group to replan the path of can greatly affect the running time, a possible heuristic is to replan the group that took the least time to plan in the first place

– Initial Path Choice: Another important variable int ID's running time involves the initial paths used. ID starts with a path for each agent, and then resolves the conflicts among those agents. A choice with fewer initial conflicts will lead to fewer replans and group merges. To achieve this, one can find a path for every agent in two passes. First we find a path for every agent in turn while avoiding conflicts with the paths of all previously planned agents whenever possible, using the conflict avoidance table. On the second pass, we then find another path for every agent that tries to avoid conflicts with the newest path found for every

other agent (either from the first pass or the current second pass). This ensures that the final path we find for every agent maximally avoids a path found for every other agent on either the first pass or the second pass.

  - Prioritise Avoiding Large Groups: When finding paths, some travel groups are more important to avoid than others. Large groups in particular are harder to replan, and if replanning fails, we risk creating an even larger travel group. By storing a cost for every move in the conflict avoidance table, and incrementing the number of conflicts (int he CAT) with existing paths that occur on the best path to a node by the cost at every node expansion rather than simply incrementing by 1, we can effectively bias the algorithm to avoid large travel groups.
  - Avoid Futile Replanning: When an agent collides with an agent that is stopped at its goal, it's impossible to find an alternative path for the second agent's travel group that has the same optimal cost and avoids this collision, because all paths for the second agent's travel group will still result in the second agent being stopped at its goal. Therefore, replanning the second agent's group is futile, and we can avoid wasting time recomputing new paths for such groups.

- Conflict-Based Search (CBS): CBS was introduced by Sharon *et al.* in 2015 [30], the key idea of CBS is to grow a set of constraints and find paths that are consistent with these constraints. If these paths have conflicts, and are thus invalid, the conflicts are resolved by adding new constraints. CBS works in two levels. At the high level, conflicts are found and constraints are added. The low level finds paths for individual agents that are consistent with the new constraints

  At the high level, CBS searches in a tree called the *constraint tree* (CT). A CT is a binary tree. Each node N in the CT consists of:

  - A set of constraints: Each of these constraints belongs to a single agent. The root of the CT contains an empty set of constraints. The child of a node in the CT inherits the constraints of the parent and adds one new constraint for one agent.
  - A solution: A set of k paths, one path for each agent. The path for agent a_i must be consistent with the constraints of a_i . Such paths are found by the low-level search.
  - The total cost: The total cost of the current solution (summed over all the single-agent path costs). This cost is referred to as the *f-value* of node N.

The set of valid nodes in the CT is defined as the set of nodes for which the solution doesn't violate any of the constraints. The high level performs a best-first search on the CT where nodes are ordered by their costs.

Given the list of constraints for a node N of the CT, the low-level search is invoked. The low-level search returns one shortest path for each agent, a_i that is consistent with all the constraints associated with a_i in node N. The newly populated node is then *validated*, namely it is considered as a candidate to be a goal node. If it is, its associated solution is returned, else it is a non-goal node and the new conflicts create new child nodes.

Given a non-goal CT node N whose solution includes a conflict $C_n = (a_i, a_j, v, t)$[3] we know that in any valid solution, at most one of the conflicting agents (a_i and a_j) may occupy vertex v at time t. Therefore, at least one of the constraints (a_i, v, t) or (a_j, v, t) must be added to the set of constraints in N.constraints. To guarantee optimality, both possibilities are examined and node N is split into two children. Both children inherit the set of constraints from N. Note that for a given CT node N, one does not have to save all its cumulative constraints. Instead, it can save only its latest constraint and extract the other constraints by traversing the path from N to the root via its ancestors. Similarly, with the exception of the root node, the low-level search should only be performed for agent a_i which is associated with the newly added constraint. The paths of other agents remain the same as no new constraints are added for them.

It can occur that more than one new conflict occurs during the low-level search, in which case two approaches can be considered. One could split N into more than one children, or equivalently only consider the first occurring conflict, the others going to be caught by later validations.

As for the inner workings of the low-level search, it can be seen as a general single agent path finding problem where only the node's constraints are considered and other agents ignored. Further tie-breaking policies can be applied in case of identical f-values, e.g. making use of the CAT so that the new paths tend to create less new conflicts.

An important variation of CBS is Disjoint Splitting [22]. By default, CBS splits nodes into two new negative constraints, each forbidding one of the two agents (a_i, a_j) from being in the conflicting vertex (v) at the conflicting time (t). This method can lead to duplicate search effort as all the cases in which neither a_i nor a_j is on v at t are covered by both branches. We say that the splits are not *disjoint*. Disjoint Splitting solves this problem making

[3]This is referring to vertex constraints only for the sake of clarity, however this also applies to edge constraints

use of *positive constraints*. Using disjoint splitting, one of the node would prevent a_i from being in v at t, and the other would force a_i to be in v at t, thus implicitly preventing a_j from being there and creating two disjoint subproblems.

- Push and Swap (P&S): P&S is a decentralised algorithm introduced by Luna and Bekris in 2011 [25].

  This algorithm doesn't offer optimallity guarantees but was found to perform very well in the introducing paper [25], both regarding perfect success rate and running time.

  It relies on two simple manoeuvres, *push* and *swap*, to resolve conflicts between agents.

  When a conflict arises, one agents takes priority and pushes the other to an adjacent free cell. If that's not possible, they swap positions. P&S is scalable and suitable for large-scale MAPF instances but may not always find optimal solutions and is not adapted to problems where the swap operation is not possible (e.g. in the presence of tight corridors where agents can't cross each other). It is however proven to be complete [25] for cooperative path-finding problems for all instances where there are at most $n - 2$ agents in a graph with n vertices, it also doesn't make any assumption regarding the topology of the map.

- Prioritized Planning (PP): PP is a class of algorithms that assign priorities to agents based on a predefined ordering. Agents plan their paths sequentially, taking into account higher priority agents' positions to avoid conflicts. PP algorithms are relatively simple and efficient but may result in suboptimal solutions.

1.5 Position of our library in relation to the literature

This thesis is positioned in the AI literature as follows. A C++ library has been implemented to solve pathfinding problems that involve planning paths for rational agents, each with an initial and a final position. The performance measure for these agents is to avoid collisions among themselves and minimise fuel or time. The environment is deterministic and time is considered discretely. It is always assumed that each agent has complete knowledge of all other agents and their planned routes (cooperative pathfinding) and a single central computing power is responsible for finding solutions for all agents simultaneously (centralised planning). Algorithms

such as various versions of A*, ID, and CBS have been implemented; they are complete and optimal. This work was carried out with the aim of scheduling paths for automated shuttles in a smart logistics warehouse for Visio Ing Consult [4].

Chapter 2

General description of the library

The library implements various algorithms and variations/optimisations

It is structured in classes and uses abstractions to enable combinations of different components e.g. using different algorithms for low level searches, change the problem type from single agent to multi agent while using the same code for the A* algorithm, ...

Effectively, the map[1] is first parsed and turned into a graph $G(V, E)$. The graph is stored as an adjacency list; an array `arr` of vectors where `arr[i]` contains the indices of all of vertex `i`'s neighbors. This thus models a *directed* graph, even though all of our algorithms don't support such graphs (such as those using RRA*) and the map textual format doesn't allow for it.

One can then choose any combination of solver (high-level and/or low-level algorithms with variations), problem type, cost function and heuristic before solving the problem.

For example, one can define a problem and two search types

```
auto problem =
    std::make_shared<MultiAgentProblem>(
        g,          // The graph
        starts,     // Starting points of agents
        targets,    // Targets of agents
        objective   // Objective function
    );
auto astarsearch =
    std::make_shared<GeneralAStar>(
        heuristic,  // Heuristic
        true,       // Use space-time search
        true        // Use OD
```

[1] In the same format as those available at [1]

```
    );
auto cbssearch =
    std::make_shared<ConflictBasedSearch>(
        heuristic,  // Heuristic
        false,      // Do not use a CAT
        false       // Do not use DS
    );
```

And then solve the problem using Independence Detection and :

- A simple AStar search as a low-level search
- A CBS as a low-level search and a Conflict Avoidance Table

```
solution = IndependenceDetection(
    problem,        // The problem to solve
    astarsearch,    // The low-level search algorithm
    false           // Whether to use a CAT
).solve()
solution = IndependenceDetection(problem, cbssearch, true).solve()
```

Or simply solve the problem with a classical AStar

```
solution = GeneralAStar(heuristic, false, false).solve(problem);
```

Figure 2.1 depicts a simplified version of the include graph focusing on the main algorithms we implemented.

2.1 Automated test suite

Our library features an automated test suite using `boost` library's framework. It was used throughout the implementation to ensure that the already implemented features remain functional.

2.2 Solution visualisation

An application enabling to solve any scenario from [1] using any algorithm was created in parallel of this library. It uses the GTK framework and is entirely implemented in rust, it is available at `https://github.com/JMans15/MAPF-visu`. It has not received a lot of polishing but it is functional, an example use follows on figure 2.2. The "Next" and "Prev" buttons allow to visualise the solution timestep by timestep to better see how conflicts are handled.

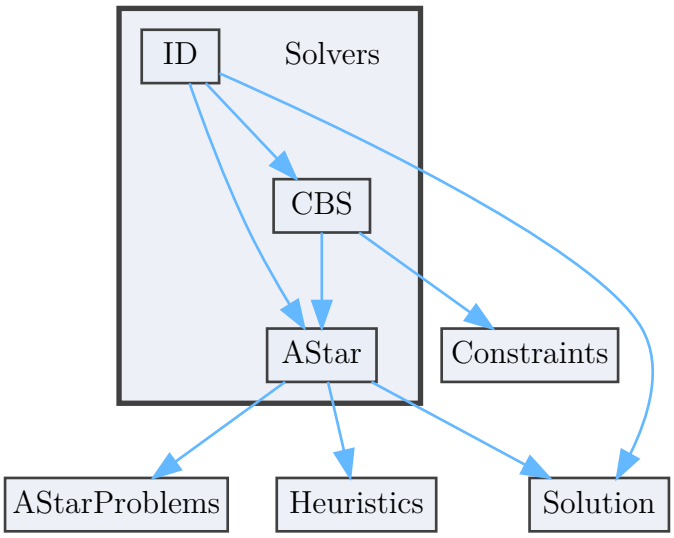


Figure 2.1: Layout of the implemented solvers

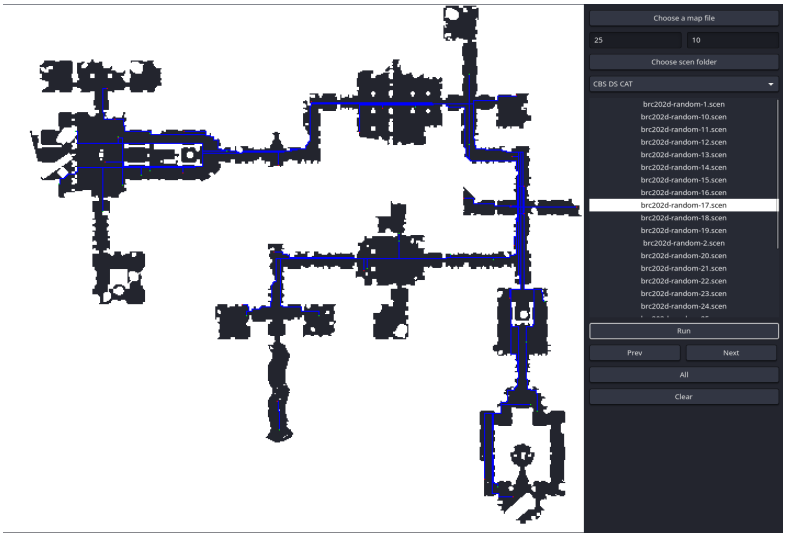


Figure 2.2: Example use of the visualiser

2.3 Documentation

Documentation of the library is available at `https://jmans15.github.io/TFE_MAPF/index.html`, it is generated by doxygen and private members are also documented as it serves as a description of the inner workings of the library rather than a classical user documentation. The "Files" section contains the layout of our directories and is much more clear to navigate. The "Data Structures" section should be accessed through links in the "Files" section as it is not as structured.

Chapter 3

Description of the problems

In this chapter, we describe the files in the "Problems" folder of the library, specifically SingleAgentProblem.h and MultiAgentProblem.h. We start with the simplest problem and then progressively make it more complex. We then move on to describing the structures present in the "Constraints" folder. Finally, we discuss the complexity of the MAPF problem.

The library operates on the **discrete time assumption** for every problem it solves. Time is divided into discrete time points.

3.1 Single-Agent Pathfinding

3.1.1 Simplest single-agent pathfinding problem

The following section describes the most basic form of the single-agent problem. Single-agent pathfinding is the problem of finding a path between two vertices in a graph. The inputs for the simplest single-agent problem are :

- `std::shared_ptr<Graph> graph` : a directed graph $G(V, E)$. The vertices V of the graph are possible positions for the agent and the edges E are the possible transitions between positions.
- `int start` : a starting position for the unique agent. $start \in V$
- `int target` : a target position for the agent. $target \in V$

A solution to this problem is a path for the agent, which can be seen as a sequence of {move} actions such that if the agent performs this sequence of actions starting from *start*, it will end up at *target*. In other words, a path can also be a list of vertices such that the first is *start* and the last is *target*. This is the representation that will be used in this library. This list must, of course, respect

the transitions E of the graph.

The objective function to minimise is the cost of the path, which is the length of the list of visited vertices reduced by 1. In fact, in this library, we assume that all move costs are 1 and do not depend on the graph. Given the objective function of this problem, there is no need to allow the agent to stay in place. An optimal solution has the lowest path cost among all solutions.

The optional inputs for this problem are :

- `int agentId = 0` : an integer to identify the agent.
- `int maxCost = INT_MAX` : an upper bound on the cost. The solution of this problem must have a cost inferior or equal to $maxCost$.

3.1.2 Single-agent pathfinding with constraints

Now that the foundations have been laid with the simplest single-agent problem, this section describes the modifications made to incorporate constraints of the type "agent a cannot occupy position p at time t". These constraints can be seen as objects moving on the map, where a vertex cannot be used by an agent at a certain time.

The problem includes vertex constraints as well as edge constraints. **Vertex constraints** prevent an agent from occupying a position at a specific time point, while an **edge constraint** prevents an agent from moving from one position to another between two consecutive time points. The problem also distinguishes between hard constraints and soft constraints. A **hard constraint** must be strictly adhered to in a solution for it to be feasible, whereas a **soft constraint** is desirable to satisfy but not mandatory. The implementation of these constraints and their sets will be explained in detail in section 3.2.3.

- `HardVertexConstraintsSet setOfHardVertexConstraints`
- `HardEdgeConstraintsSet setOfHardEdgeConstraints`
- `SoftVertexConstraintsMultiSet setOfSoftVertexConstraints`
- `SoftEdgeConstraintsMultiSet setOfSoftEdgeConstraints`

With the addition of these constraints, it becomes important to incorporate a temporal dimension into our problem by introducing a start time. It is no longer a problem where the solution remains the same for any start time, as it was the case for the problem described in the previous section (3.1.1).

- `int startTime = 0` : the agent is occupying the position *start* at the time $startTime$.

Given these "objects that move on the map," it is also important to allow the agent to stay in place between two time points, and a path can now be seen as a sequence of {move, wait} actions.

Now that we allow the agent to stay in place between two time points, two objective functions can be introduced. One considers the wait cost as 1, while the other considers the wait cost as 0. The objective function to minimise becomes an input for this type of problem.

- `ObjectiveFunction objective` : the objective/cost function to minimise. It can be equal to `Fuel` to minimise the total amount of distance travelled by the agent (wait cost of 0) or `Makespan` to minimise the total time for the agent to reach its goal (wait cost of 1).

3.2 Multi-Agent Pathfinding

3.2.1 Simplest multi-agent pathfinding

The multi-agent pathfinding (MAPF) problem is a generalisation of the single-agent pathfinding problem for $k \geq 1$ agents. We are given a set of agents each with respective start and goal positions. The task is to find paths for all agents while avoiding collisions between agents. One thing to note is that an agent does not disappear from the map/release the vertex once it reaches its destination; it stays on the map until all agents have reached their targets. The inputs for this problem are :

- `std::shared_ptr<Graph> graph` : a directed graph $G(V, E)$. The vertices V of the graph are possible locations for the agents and the edges E are the possible transitions between locations.
- `std::vector<int> starts` : a starting position for each agent. $\forall i \in \{0, 1, ..., k-1\}, starts_i \in V$
- `std::vector<int> targets` : a target position for each agent. $\forall i \in \{0, 1, ..., k-1\}, targets_i \in V$

A solution to this problem is a path for each agent i, which is a list of vertices such that the first is $start_i$ and the last is $target_i$. This list must, of course, respect the transitions E of the graph. Between successive time points, each agent can perform a move action to a neighbouring vertex or a wait action to stay idle at its

current vertex. Furthermore, a solution is only feasible if there are no collisions between the paths of different agents.

There are multiple ways to define the absence of collisions between agents. We define two types of conflicts: vertex conflict and edge conflict. A **vertex conflict** occurs when two or more agents occupy the same vertex/position at the same time, while an **edge conflict** occurs when two or more agents traverse the same edge between successive time steps. A solution is feasible if it does not contain any vertex conflicts or edge conflicts. It is our decision to consider these two types of conflicts in this library. However, it would not be complicated to add other types of conflicts since the implementation of the algorithms uses sets of conflicts and `calculateSetOfConflicts()` functions that would simply need to be modified to add more conflicts. We could have also avoided follow conflicts, for example. A **follow conflict** occurs when one agent closely follows another agent along their paths, potentially leading to collisions. It happens when agents' paths overlap or when one agent follows the exact sequence of vertices of another agent.

The MAPF problem is an optimisation problem, which means it has an objective function to minimise. The library offers three commonly used objective functions in the field.

- `ObjectiveFunction objective` : the objective/cost function to minimise. It can be equal to `Fuel` to minimise the total amount of distance travelled by all agents, `SumOfCosts` to minimise the sum of the time steps required for every agent to reach its goal and remain at the goal position without leaving it again or `Makespan` to minimise the total time for the last agent to reach its goal.

The optional inputs for this problem are :

- `std::vector<int> agentIds = {0, 1, ..., k-1}` : an integer to identify each agent. $\forall i \in \{0, 1, ..., k-1\}, agentIds_i$ is the id of the agent at index i.
- `int maxCost = INT_MAX` : an upper bound on the cost. The solution of this problem must have a cost inferior or equal to $maxCost$.

3.2.2 Multi-agent pathfinding with external constraints

Similarly to the single-agent pathfinding problem described in section 3.1.2, time-dependent external constraints can be added to the problem. As a result, an optional start time parameter is also introduced to the problem. Section 3.2.3 describes how these constraints and their corresponding sets are implemented.

- `HardVertexConstraintsSet setOfHardVertexConstraints`
- `HardEdgeConstraintsSet setOfHardEdgeConstraints`
- `SoftVertexConstraintsMultiSet setOfSoftVertexConstraints`
- `SoftEdgeConstraintsMultiSet setOfSoftEdgeConstraints`
- `int startTime = 0`

3.2.3 Constraints structures

This section provides a description of the constraints and the corresponding sets found in the "Constraints" folder of the library.

Vertex constraint

The file VertexConstraint.h contains the implementation of the `VertexConstraint` class. Its constructor takes three mandatory parameters and one optional parameter:

- `int agent`
- `int position`
- `int time`
- `bool positive = false` : either it is a negative or a positive constraint.

This structure can be used in two types of constraint sets: hard constraints and soft constraints.

In a set of hard constraints, a negative constraint signifies that the specified agent cannot be at the specified position at the given time. In contrast, a positive constraint indicates that the agent must be at the specified position at the given time. It is important to note that all implemented algorithms solely consider problems with negative constraints. Positive constraints are exclusively used within the Conflict Based Search with Disjoint Splitting algorithm, as described in section 6.3.

On the other hand, in a set of soft constraints, the positive parameter is not applicable. In this case, the structure always denotes that the specified agent is at the specified position at the given time, and all other agents (except the specified agent) are expected to avoid this position at that time step.

Edge constraint

While the file EdgeConstraint.h contains the implementation of the `EdgeConstraint` class, its implementation and meaning are very similar to the `VertexConstraint`

class. The parameters for the `EdgeConstraint` class are as follows:

- `int agent`
- `int position1`
- `int position2`
- `int time`
- `bool positive = false`

This structure can also be used in the two types of constraint sets: hard constraints and soft constraints.

In a set of hard constraints, a negative constraint indicates that the specified agent is not allowed to move from the first position to the second position between the previous time step and the specified time step. Conversely, a positive constraint specifies that the agent has to move from the first position to the second position between these two time steps. Similar to vertex constraints, all implemented algorithms handle only problems with negative edge constraints, and positive edge constraints are only used in the algorithm described in section 6.3.

This structure in a set of soft constraints always indicates that the specified agent occupies the edge between the first position and the second position from the previous time step up to the specified time step, and all other agents (except the specified agent) are expected to avoid this edge between these two time steps.

Set of constraints

The types of constraint sets contained in certain problems are defined in the ConstraintsSet.h file. They are `SoftVertexConstraintsMultiSet`, `SoftEdgeConstraintsMultiSet`, `HardVertexConstraintsSet` and `HardVertexConstraintsSet`.

The **sets of hard constraints** are implemented using `std::unordered_set` along with equality and hashing functions. These functions take into account all the attributes of a constraint (except `bool positive` as these sets only contain negative constraints) and consider two constraints equal if their attributes are equal.

Using an `std::unordered_set` is preferable to an `std::set` in this situation for several reasons. First of all, it has a lower time complexity and efficiently handles large data collections. Specifically, the average time complexity for search, insert and erase operations in an `std::unordered_set` is constant O(1), while in an `std::set` it is logarithmic O(log n). This means that operations on an `std::unordered_set` are generally faster when dealing with a large number of

elements. Secondly, the non element ordering. A `std::set` automatically sorts its elements in a specific order (by default, ascending order). However, this sorting feature can result in a performance overhead during insertion and deletion operations. On the other hand, an `std::unordered_set` does not organise its elements in a specific order, which can lead to faster operations. Lastly, the usage of a hashing function in an `std::unordered_set` enables efficient distribution of elements within the data structure. This hashing function facilitates quick access to the desired elements, which is particularly advantageous for handling large data collections. [2][3]

The **sets of soft constraints** are implemented using `std::unordered_multiset` along with equality and hashing functions that consider two constraints equal if they have the same position and time. Given the representation of a soft constraint, when we want to count the number of violated soft constraints in algorithms, for example, by placing an agent a at position p at time t, we need quick access to all constraints stating that an agent is at position p at time t. This cannot be efficiently achieved with an `std::unordered_set` as described for sets of hard constraints. The lack of ordering in `std::unordered_set` is one of its limitations because it can make iterating over elements in a specific order more difficult. Implementing the equality and hashing functions in such a way that two soft constraints are considered equal if they have the same position and time using an `std::unordered_set` was considered. However, since this type of set contains only one instance of each element, it would allow for a maximum of one constraint concerning position p at time t. This is not practical considering how these sets are used in algorithms (see the Conflict Avoidance Table in section 5.4). We want to keep track of all agents that are at position p at time t. For example, if agent 1 and agent 2 occupy position p at time t, and both of these constraints are in a set, if agent 1 changes its path and no longer passes through position p at time t and we remove the corresponding constraint from the set, we still want the constraint corresponding to agent 2 to remain present in the set.

Chapter 4

Traditional approach

This chapter presents the traditional approach to solving a pathfinding problem, namely the A* algorithm with multiple problem representations that vary the transition functions as well as the states (in the "AStar", "AStarProblems" and "States" folders). Subsequently, two heuristic functions will be described along with the Reverse Resumable A* algorithm, which is necessary for one of the heuristics ("Heuristics" folder).

4.1 Generic A* [29]

A* is a widely used informed search algorithm that combines the advantages of both uniform cost search (which is also called Dijkstra algorithm) and greedy best-first search. It is commonly used for pathfinding and graph traversal problems like the MAPF problem. Generic A* refers to a general formulation of the A* algorithm that can be applied to various problems, heuristics, transitions functions, states... Its implementation, which can be found in our library in the AStar.h file, is explained in this section and its pseudo code is the algorithm 1.

What is a search ?

A search is the process of looking for a sequence of actions that leads to a goal (a specific state of the environment) starting from an initial state. In our case, we are seeking the best sequence of actions because the problem has an objective function to minimise. States describe distinguishable stages during the problem-solving process. In our case, a state mainly consists of the agents' positions at a certain time step. An action transports the agents from one state to another by applying an operator to a state. The state space is the space of all possible states. It is never constructed entirely but only explored. This is what a search algorithm like A* does. Each node corresponds to a state. The search begins at the root node, which

contains the initial state `prob.getStartState()`, and each edge corresponds to an action. We expand a node n as follows: use `prob.getSuccessors(s)` to get the resulting states where s is the state in node n, generate new nodes with these states (called child nodes or successor nodes), and attach each of these nodes to the current node as its parent. We start by expanding the root node and then always expand the most promising node until reaching a goal state (checked with `prob.isGoalState(s)`).

Abstract class `AStarProblem`

The A* algorithm requires the search problem to be formulated in a specific way, including the representation of a state, an initial state, goal states or a goal test, a transition model to determine the successor states for a given state and an action cost function. This formulation can be found in the file AStarProblem.h in the form of an abstract class with the pure virtual functions `getStartState()`, `getSuccessors(std::shared_ptr<S> s)` and `isGoalState(std::shared_ptr<S> s)`. Several classes can then inherit from this abstract class and implement its pure virtual functions (see section 4.2).

Abstract class `Heuristic`

Informed search algorithm may have access to a heuristic function. A heuristic function (denoted $h(n)$) estimates the cost of the cheapest path from node n to a goal. Performance of informed searches like A* depends on the quality of the heuristic.

Generic A* manipulates an instance of the `Heuristic` class, which is an abstract class containing a pure virtual function `heuristicFunction(std::shared_ptr<S> s)` that returns an integer – an estimation of the cost of the least expensive path from state s to a goal, $h(s)$. Implementations of the classes derived from this abstract class are described in section 4.3.

Node

The implementation of a node can be found in the file Node.h. A state represents a physical configuration (the positions of agents on the map), while a node is a data structure constituting part of the search graph. A node n has the following attributes:

- `std::shared_ptr<Node<S>> parent` : a pointer to the parent node
- `std::shared_ptr<S> state` : a pointer to the state it contains

- `int cost` : the path cost (from the initial state/root node) $g(n)$
- `int heuristic` : its heuristic value $h(n)$
- `int violationCount = 0` : the number of soft constraints that have been violated on the path from the root node leading up to this node (useful in a problem with soft constraints)

Closed list

While traversing the state space, we may encounter and visit the same sequence of states multiple times. Failure to detect redundant paths can turn solvable problems into unsolvable ones. Redundant paths are avoided by maintaining a look-up table containing all the visited states, called a closed list, in such a way that each state is only explored once. Graph searches avoid redundant paths using a closed list, while tree searches do not.

The implementation of A* in this library is a graph search, and the closed list is implemented using `std::unordered_map<std::shared_ptr<S>, int>`, a map with the state as the key and the path cost as the value, which helps to preserve the lowest cost path among repeated states.

Open list

Which node to choose next ? Which node is the most promising one ? Choose one of the unexpanded node. This choice is the essence of the search. For that, the algorithm uses an open list (also called frontier or fringe) which is a set of some unexpanded nodes, a set of generated nodes which have not been visited (goal-tested) and which parents have been visited. In a best-first search (which is the case of the A* algorithm), the open list is a priority queue with an evaluation function $f(n)$. The search selects the node n that minimises $f(n)$. For information, in a breadth-first search the open list is implemented with a FIFO queue and with a LIFO queue (also called stack) in a depth-first search.

The A* algorithm chooses the (estimated) cheapest path through the current node. The evaluation function $f(n)$ used in the open list to determine the next node to select is defined the following way.

$$f(n) = g(n) + h(n)$$

where $g(n)$ is the exact cost from initial state to node n and $h(n)$ is the estimated cost from node n to a goal, as just explained.

In our library, the open list is implemented using a `std::multiset<std::shared_ptr<Node<S»,NodeComparator<S»`. The order in which nodes are removed is determined by the `NodeComparator` function contained within Node.h. This comparator is responsible for comparing two nodes to determine their priority in the open list during the search process. It first prioritizes nodes based on their combined cost and heuristic values, denoted as $f(n) = g(n) + h(n)$. If two nodes have the same $f(n)$ value, their violation counts are considered to give preference to solutions that violate the fewest soft constraints. If the violation counts are also equal, the function then compares their heuristic values $h(n)$.

Algorithm 1 `solve()` of `AStar` (generic A*)

```
1:  state ← prob.getStartState()
2:  node ← Node(state, 0, h(state))                    ▷ We initialise Node(state, cost, heuristic)
3:  add node to openList
4:  closedList[state] ← 0
5:  while not openList.empty() do
6:      node ← openList.pop()
7:      state ← node.getState()
8:      if node.getCost() > closedList[state] then
9:          continue
10:     end if
11:     if prob.isGoalState(state) then
12:         return retrieveSolution(node)
13:     end if
14:     for (successor, edgeCost, edgeViolations) in prob.getSuccessors(state) do
15:         successorCost ← node.getCost() + edgeCost
16:         successorViolations ← node.getViolationsCount() + edgeViolations
17:         if (state is not in closedList or successorCost < closedList[state]) and (successorCost ≤
    prob.getMaxCost()) then
18:             closedList[successor] ← successorCost
19:             successorNode ← Node(successor, successorCost, h(successor), node, successorViolations)
20:                                    ▷ We initialise Node(state, cost, heuristic, parent, violationCount)
21:             add successorNode to openList
22:         end if
23:     end for
24: end while
25: return empty solution
```

Upper bound on the cost of a solution

To account for the upper bound on the cost of a solution, the algorithm adds a node to the open list only if its cost $g(n)$ is less than or equal to `prob.getMaxCost()`.

Optimality, completeness and complexities of A*

To establish the optimality of A*, one can define specific terms and properties related to the heuristic function.

- $h(n)$ is admissible if $h(n)$ never overestimates the cost to reach a goal ; we could say that $h(n)$ is optimistic. The consequences of this are that

$f(n) = g(n) + h(n)$ never overestimates the cost of a solution through node n and that $h(n) = 0$ if n is a goal.

- $h(n)$ is consistent if, for every node n and successor node n' obtained with action a, the estimated cost of reaching goal from n ($h(n)$) is not greater than the addition of the cost of getting to n' ($c(n, a, n')$) and the estimated cost of reaching goal from n' ($h(n')$). This is the following triangle equality.

$$h(n) \leq c(n, a, n') + h(n')$$

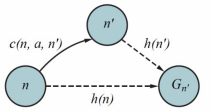


Figure 4.1: Condition for a heuristic to be consistent [29]

- If $h(n)$ is consistent, then it is also admissible.
- A* is optimal (which means it finds a least cost solution) for tree search if $h(n)$ is admissible and for graph search if $h(n)$ is consistent.

Regarding the completeness, space complexity and time complexity of A*, we can state the following.

- A* is complete, which means it finds a solution if one exists.
- Its time and space complexity is $O(b^d)$. It is better than other search algorithms. Compared to other algorithms that search from root using the same heuristic, no other optimal algorithm is guaranteed to expand fewer nodes than A* using graph search. However, A* is still problematic because of its space complexity.

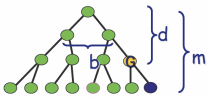


Figure 4.2: Search tree variables (b is the maximum branching factor, d is the depth of the least-cost solution and m is the maximum number of actions in any path (may be infinite)) [29]

4.2 Several transition functions and states

In the "AStarProblems" folder, there are 6 classes that inherit from the abstract class `AStarProblem`. Each class has its corresponding state class that inherits from the abstract class `State`, located in the "States" directory. These 6 implementations allow variation in the following aspects within the Generic A* algorithm:

- space search or space-time search
- single-agent search or multi-agent search
- for the multi-agent search, standard A* or A* with Operator Decomposition

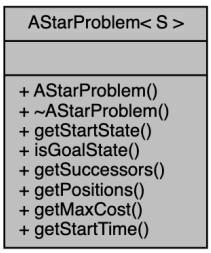


Figure 4.3: Diagram for `AStarProblem` abstract class

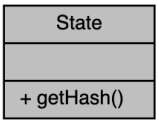


Figure 4.4: Diagram for `State` abstract class

4.2.1 Space search and space-time search

This distinction lies in how two states are considered equal in the closed list of the Generic A* algorithm. A space state does not consider time, while a space-time state includes an attribute `timestep`, representing the number of timesteps since the agent started. Time is thus included in the `getHash()` function of a space-time state but not in that of a space state.

A space-time search is necessary for the problems described in sections 3.1.2 and 3.2.2, meaning for problems involving external constraints that depend on time.

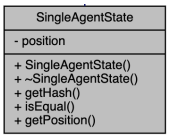


Figure 4.5: Diagram for `SingleAgentState` class

Figure 4.6: Diagram for `SingleAgentSpaceTimeState` class

A space-time search expands a significantly larger number of nodes and is therefore resource-intensive. Thus, it's essential to pair it with a highly effective heuristic.

4.2.2 Single-agent search

A single-agent state contains the position of one agent, while a multi-agent state contains the positions of all agents.

The simplest implementation of `getSuccessors()` is found in `SingleAgentAStarProblem` in algorithm 2. It is suitable for solving the problem described in section 3.1.1. It forces the agent to move to a neighbouring position in the next time step, where the cost of movement is 1, and there are 0 violated soft constraints caused by this movement as the problem does not contain any.

Algorithm 2 `getSuccessors(state)` of `SingleAgentAStarProblem`

1: $successors \leftarrow$ empty vector
2: **for** $newPosition$ in $prob.getGraph().getNeighbors(state.getPosition())$ **do**
3: $\quad successor \leftarrow SingleAgentState(newPosition)$
4: $\quad edgeCost \leftarrow 1$
5: $\quad edgeViolations \leftarrow 0$
6: $\quad$ add $(successor, edgeCost, edgeViolations)$ to $successors$
7: **end for**
8: **return** successors

The `getSuccessors()` function in `SingleAgentAStarProblemWithConstraints`, suitable for solving the problem described in section 3.1.2, differs in the following way. In addition to allowing movement to a neighboring position, it allows the agent to stay in place (wait action) between two successive time steps. If the objective function is `Fuel`, the wait cost is 0, whereas it is 1 in the case of `Makespan`. The move cost is always 1. Since the problem includes constraints, the successor is added to the list of successors only if it doesn't violate any hard constraints of

the problem. Furthermore, the number of violated soft constraints is counted and stored as $edgeViolations$.

4.2.3 Multi-agent search

In the case of a multi-agent search, there are two possibilities for implementing the transition function. In the **Standard A*** approach, transitioning from one state to another involves applying an action (move or wait) to all agents, effectively advancing by one time step. In the **A* with Operator Decomposition (OD)** approach, transitioning from one state to another entails applying an action to a single agent, requiring the passage through k states to progress by one time step, where k is the number of agents.

Standard A* [34]

In a four-connected grid world, each standard state potentially leads to 5^k legal successor states. Each of these legal successor states can be perceived as a solution to a constraint satisfaction problem, where each agent must be assigned an action from {N, S, E, W, and wait}, while adhering to constraints between sets of legal moves (no vertex conflict and no edge conflict). We have implemented this using a recursive function that assigns an action to each agent in a predefined order. This implementation can be likened to a backtracking search that finds all solutions of the CSP. In practice, most of the 5^k action combinations are valid, and the number of nodes added to the open list during each node expansion is exponential with respect to the number of agents k.

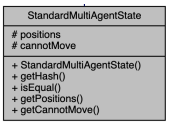


Figure 4.7: Diagram for `StandardMultiAgentState` class

A* with Operator Decomposition [34]

This technique reduces the branching factor of A*. In this representation, each state potentially leads to only 5 legal successor states ({N, S, E, W, and wait}). Since it takes passing through k states to progress by one time step, we will define 2 types of states. A standard (or regular) state is one where all agents have been assigned their actions. Conversely, when a state is not standard, it is considered intermediate. Every k intermediate states, we encounter a standard state. Assigning an action to the last unassigned agent in an intermediate state leads to a standard state which

has advanced to the next time step. These two types of states are considered in the same way within the Generic A*.

An Operator Decomposition state no longer solely comprises the current position of each agent (`positions`). It also includes information about the agent that needs to be assigned in the next state (`agentToAssign`) as well as `prePositions`, the positions of the agents in the last standard state. Note that `positions` and `prePositions` are equal at standard states.

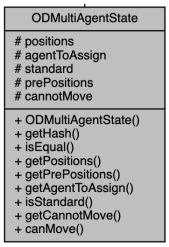


Figure 4.8: Diagram for `ODMultiAgentState` class

For A* with OD to guarantee optimal solutions, the `getSuccessors()` function requires that a new action assigned to an agent is legal if it is consistent (does not lead to a conflict) with the existing assignments of actions to agents in a state, but `getSuccessors()` allows agents to move into positions occupied by agents who are yet to be assigned actions within a state.

When employed alongside an perfect heuristic and a perfect tie-breaking strategy, the A* search algorithm generates a total of bd nodes, wherein b and d are specified in figure 4.2. The standard A* algorithm exhibits a branching factor of roughly 5^k and reaches a depth of t (representing the number of time steps within the optimal solution). Consequently, when the A* search algorithm is applied to the standard state space in conjunction with an perfect heuristic, it yields an estimated count of $(5^k)t$ nodes. However, A* with OD will generate no more than $5(kt)$ nodes in the same scenario, due to the reduction of its branching factor to 5 and and its depth only increases to kt. This is an exponential savings.

Modification in the `getSuccessors()` function for the `SumOfCosts` objective function

In the `getSuccessors()` function, we expand a node and we have to provide the cost from going to the current node to the successor node. When the objective function is `Fuel`, the wait cost is 0 and the move cost is 1. And when the objective function is `Makespan`, the cost is 1 when we progress by one time step and 0 otherwise.

But for the `SumOfCosts` objective function, a clever trick had to be devised. The move cost is consistently 1. However, the wait cost needs to be 0 when an agent

is at its target position and will no longer move and 1 otherwise. The challenge lies in determining when an agent will stop moving. To address this, two cases are anticipated and `getSuccessors()` is adjusted accordingly. (1) The wait cost is set to 0, and the agent is constrained to stay at its target position, effectively indicating it has stopped moving. (2) The wait cost is set to 1, allowing the agent to continue moving. This can be observed in lines 21 and 22 of algorithm 3. So, a multi-agent state will contain information about whether an agent can still move (once it reaches its final position) in the variable `cannotMove`. It's important to permit the agent to move even if it's at its target position. This accommodates scenarios where the agent needs to move to clear a path for another agent before returning to its target position.

Algorithm 3 very simplified `getSuccessors()` of `ODMultiAgentAStarProblem` in a four-connected grid

```
 1: successors ← empty vector
 2: if not SumOfCosts objective function then
 3:     add N to successors
 4:     add S to successors
 5:     add E to successors
 6:     add W to successors
 7:     add Waits to successors
 8: else if SumOfCosts objective function then
 9:     if agentToAssign can move then
10:         if agentToAssign is not at its target position then
11:             add N to successors with moveCost = 1
12:             add S to successors with moveCost = 1
13:             add E to successors with moveCost = 1
14:             add W to successors with moveCost = 1
15:             add Waits to successors with waitCost = 1
16:         else if agentToAssign is its target position then
17:             add N to successors with moveCost = 1
18:             add S to successors with moveCost = 1
19:             add E to successors with moveCost = 1
20:             add W to successors with moveCost = 1
21:             add Waits to successors with waitCost = 0 and we force the agent to not move in the future
22:             add Waits to successors with waitCost = 1 and the agent can still move in the future
23:         end if
24:     else if agentToAssign cannot move then
25:         add Waits to successors with waitCost = 0
26:     end if
27: end if
28: return successors
```

4.2.4 `GeneralAStar` class

Section 4.2 has described 6 different transition functions and states for the Generic A* (`AStar`). The `GeneralAStar` class has been implemented to execute one of these 6 versions of A* for a given problem.

When an instance of texttttGeneralAStar is created, it inherently requires

a heuristic to be provided. However, if no other parameters are given, the `fixedParameters` is set to false. When the `solve(problem)` method is called, the most appropriate version of A* for the problem will be executed. This means that a space search will be performed for a problem without external constraints, while a space-time search will be carried out for a problem with external constraints. Additionally, A* with Operator Decomposition will always be favoured. This approach seems logical, but if `problem` is a single-agent problem, a single-agent search will be launched, and a multi-agent search will be launched otherwise.

On the contrary, if, during the creation of a `GeneralAStar` instance, the `spaceTimeSearch` and `operatorDecomposition` parameters are provided, the parameters are fixed, and all problems will be solved using the same type of search: space-time search if `spaceTimeSearch` is true, A* with OD if `operatorDecomposition` is true, and Standard A* otherwise.

4.3 Heuristics

In the "Heuristics" directory, you will find classes that derive from the abstract class `Heuristic`. Two heuristics have been designed: `Manhattan` and `OptimalDistance`. Both of these heuristics have been adapted to accommodate single-agent search, multi-agent search, objective functions that sum individual costs of all agents (Sum of Individual Costs - SIC heuristic for the `SumOfCosts` and `Fuel` objective functions), and objective functions that consider the maximum individual cost (Maximum Individual Cost - MIC heuristic for the `Makespan` objective function).

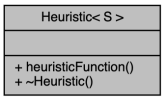


Figure 4.9: Diagram for `Heuristic` abstract class

The concept of a heuristic was introduced in section 4.1. Let's just recall that a heuristic function estimates the cost from a given state to the goal state. It's crucial to choose an admissible and consistent $h(n)$ for A* to be optimal. These terms were also explained in section 4.1.

One way to create a heuristic function is to simplify the original problem by relaxing some constraints. The cost of the optimal solution of the relaxed problem is always a consistent (and admissible) heuristic for the original problem. Our library offers 2 heuristics for the A* algorithm.

```
enum TypeOfHeuristic { Manhattan, OptimalDistance }
```

4.3.1 Manhattan distance heuristic

The Manhattan distance heuristic is the optimal solution of the problem relaxed by ignoring walls and other agents. In this relaxed problem, agents can move through walls and collide.

In this heuristic, the agent can only move in the four cardinal directions, as if navigating around rectilinear city blocks. This heuristic estimates the distance from the position of the current state to the target position by summing the horizontal and vertical moves on the grid necessary to reach the objective. This calculation is straightforward and can be performed in $O(1)$ time using the formula `abs(x_1 - x_2) + abs(y_1 - y_2)`.

4.3.2 Optimal distance heuristic [32]

The Optimal Distance heuristic represents the optimal solution of the problem relaxed by ignoring other agents. Agents are allowed to collide in this relaxed scenario. This heuristic aims at finding the shortest path between the current position of an agent and its target position, accounting for walls. The inaccuracy of the heuristic is determined only by how much the agent must deviate from the direct path to avoid other agents, or possible external constraints. In the case of a single-agent search, this heuristic is thus perfect! It proves very useful for effectively guiding a single-agent space-time search, as used in CBS.

In this heuristic, we solve a sub-problem for each agent and if a search is performed from scratch each time around, then overall performance may in fact be reduced. To avoid this, we initiate a Reverse Resumable A* (RRA*) search per agent in addition to the A* search for which this heuristic is being employed. RRA* will be explained in section 4.4.

4.3.3 Comparison of the heuristics and necessity of the Optimal distance heuristic

Both of these heuristics are admissible and consistent because they are optimal solutions of relaxed versions of the original problem. Therefore, A* with these heuristics guarantees optimality. This holds true for all objective functions; in the relaxed problems, an agent never waits, and thus wait costs are never considered.

Since the Optimal Distance heuristic arises from a less relaxed problem than the Manhattan heuristic, it offers a tighter bound on the distance and thus provides better guidance for the search. It can be stated that the Optimal Distance heuristic dominates the Manhattan heuristic. The most dominant heuristic is the most informed one. We say that h_2 dominates h_1 if, for any node n, $h_2(n) \geq h_1(n)$. In other words, h_2 is always better than h_1. However, the Optimal Distance heuristic

Table 4.1: Single-agent search from node (7) to node (17) in the AssignmentIA-Course.map that can be found in the library's GitHub repository

Heuristic	Type of search	Number of visited nodes	Number of nodes left in the fringe
Manhattan	Space	25	5
Manhattan	Space-time	177	49
Optimal distance	Space	19	10
Optimal distance	Space-time	19	47

can be more computationally expensive and not achievable in $O(1)$ time, but thanks to RRA*, this is mitigated.

The Optimal Distance heuristic is crucial in our library for the following reason. The Conflict-Based Search (CBS) algorithm described in section 6.1 requires launching single-agent searches with external constraints. These external constraints necessitate the search to be space-time. The number of visited nodes in a space-time search grows significantly due to the addition of an extra dimension. As a result, a highly effective heuristic like Optimal Distance becomes essential to better guide the search in this expanded space with an additional dimension.

To compare heuristics, it is common to compare the number of generated nodes. The table 4.1 illustrates, with a very small problem, the exponential growth in the number of visited nodes during a space-time search and demonstrates how the Optimal Distance heuristic addresses this issue. Among other observations, it's noticeable that the number of visited nodes in a single-agent SPACE-TIME search with the Optimal Distance heuristic is equal to the number of visited nodes in a single-agent SPACE search with the Optimal Distance heuristic.

4.4 Reverse Resumable A* [32]

This section describes the Reverse Resumable A* (RRA*) algorithm, which is essential for the implementation of the Optimal Distance heuristic. This algorithm is implemented in the "AStar" file.

The RRA* algorithm involves conducting a modified single-agent space A* search in reverse direction. The search begins at the agent's goal position and proceeds towards the agent's initial position. Unlike a regular A* search that stops at the goal position, RRA* continues its search until a specific node is expanded. Because the search is reversed, if an agent can move from vertex 1 to vertex 2, there must also be an edge in the reverse direction, enabling the agent to move from vertex 2 to vertex 1.

One notable property of A* with a consistent heuristic is that the optimal

distance from the start to a node becomes known once that node is expanded [26]. With the search proceeding in reverse, this implies that the optimal distance from a node to the goal position is known once the RRA* search visits that node. For the RRA* search, the Manhattan heuristic is employed, ensuring the consistency criterion is met.

Whenever a request is made for the optimal distance from the current position to the goal position, RRA* checks if a state with the current position exists in its closed list. If such a state is present, the optimal distance for that state is already determined and can be promptly returned in $O(1)$. If the state is not found, the RRA* search is resumed until the node associated with that state is expanded.

Chapter 5

Independence Detection

As discussed before, ID is an attempt at minimising the complexity of complete search algorithms such as A* or CBS by splitting large groups into smaller independent ones. Our library implements various versions and improvements, but we will first focus on the "Simple Independence Detection" (SID) version, and then build upon it.
Figure 5.1 is derived from the actual include graph of the project and represents the layout of our implementation of SID along with the abstractions we used

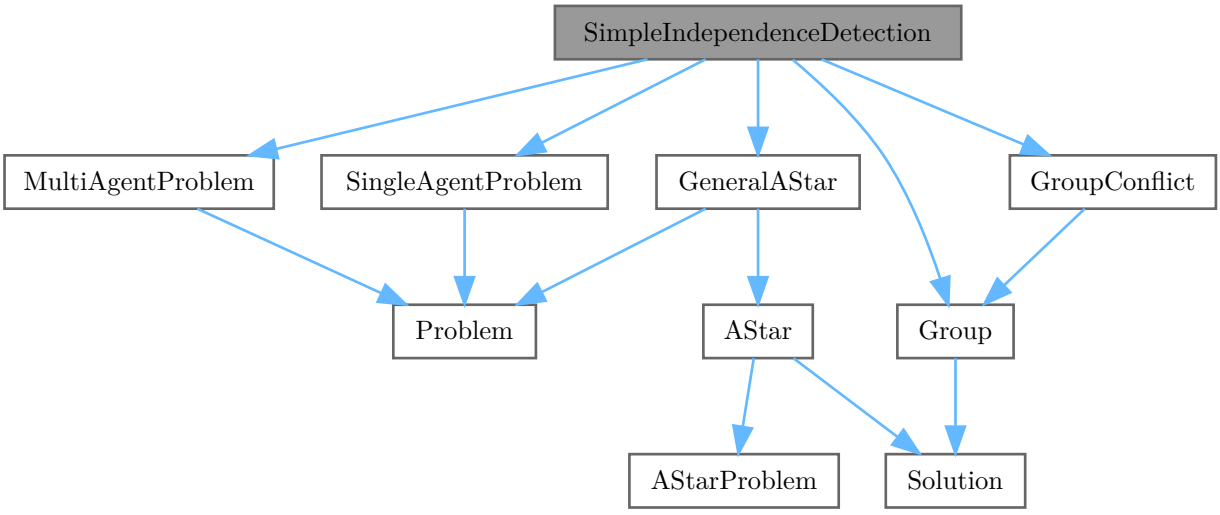
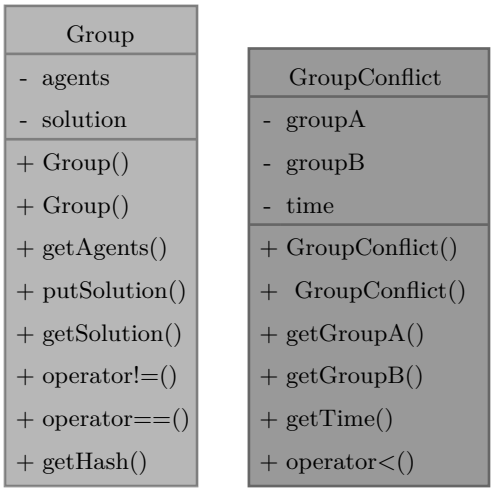


Figure 5.1: SID's simplified include graph, considering A* low-level search

5.1 Group

Groups are a central part of ID and are stored in a data structure of the same name, which contains the agents that are part of the group along with the current



(a) Group (b) GroupConflict

Figure 5.2: Group and GroupConflict data structures

solution.
Group conflicts are also stored in their own data structure and contain the two conflicting groups along with the time at which the conflict occurred
Figure 5.2 shows the attributes and methods of these classes

5.2 Simple Independence Detection

Algorithm 4 describes the general steps performed by the implementation.

Algorithm 4 `Simple Independence Detection[35]`

1: assign each agent to a singleton travel group
2: plan a path for each travel group with low level search
3: **repeat**
4: Find all collisions and identify the first one that occurs
5: merge two conflicting travel groups into a single one
6: plan paths for the new travel groups using low level search
7: **until** the current set of conflict is empty
8: *solution* ← path of all travel groups combined
9: **return** *solution*

The function thus starts by computing the paths of each agent individually, then enters the main loop.
In the main loop, we actually don't do the same thing as in the source paper[35]. In said paper the movement of agents is simulated and conflicts between groups

are detected while in our implementation we find all the conflicts between the computed paths and then find the first one, this avoids having to simulate all the displacements to find the first collision. A set of conflicts is also maintained so that we don't have to find all of them again every time.
When at least one conflict is left in the conflict set, the two groups are merged and the new paths are planed using the low level search, the complexity of which exponentially grows with the size of the group.
When all the paths can be simulated from start to finish without collisions, the global solution is extracted from the different groups and is returned.
In some cases this algorithm can be very efficient, particularly when few agents intersect or in low density areas, but if many merges occur the groups can grow very quickly which lessens the benefits of the procedure. The following section describes an enhanced version of the algorithm that aims at keeping the groups as small as possible.

5.3 Enhanced version of ID

Algorithm 5 describes the Enhanced ID (EID).

Algorithm 5 `Independence Detection[35]`

```
 1: assign each agent to a singleton travel group
 2: plan a path for each travel group with low level search
 3: repeat
 4:     Find all collisions and identify the first one that occurs, between travel groups G1 and G2
 5:     if G1 and G2 haven't conflicted before then
 6:         fill illegal move table with the current paths for G2
 7:         find alternative paths for G1 that do not conflict with G2 and satisfy cost limits
 8:         if search for such paths failed then
 9:             try the opposite, re-planing G2 so that it avoids G1 and satisfies cost limits
10:         end if
11:     end if
12:     if both re-planning attempts failed then
13:         merge G1 and G2 into a single group and compute new paths with low level search
14:     end if
15: until no conflict occurs
16: solution ← path of all travel groups combined
17: return solution
```

The global shape of the algorithm is the same as alg. 4, but it differs in the way it handles conflicts. Instead of merging the two groups this one first tries to re-plan one of them so that it avoids the other one for the reminder of the search, while ensuring optimality by explicitly setting an upper bound to the cost of the new solution.

5.4 Conflict Avoidance Table

Another important factor in the running time of the algorithm is how many conflicts happen. In fact, in the perfect scenario where no conflict can happen between the agents, ID only runs one single agent search per agent, and the less conflict happen, the smaller the groups remain. One way of reducing the likelihood of creating conflicts when planning a path is to make use of a Conflict Avoidance Table (CAT). This table stores all the current tentative moves of the agents that are not part of the group. When planning paths, ID will keep track of the number of CAT violations for each path candidate and will use this new metric as a tie-breaking heuristic, doing so keeps optimality guarantees.
If optimality is not as important as running time, it is also possible to prioritize CAT violations over path cost, but optimality guarantees are then lost.

Chapter 6

Conflict Based Search

6.1 Global algorithm

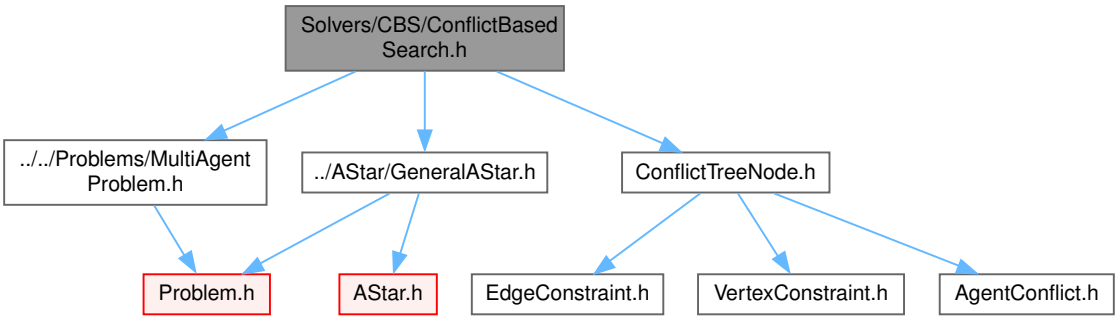


Figure 6.1: CBS include graph

Figure 6.1 contains the include graph for file `ConflictBasedSearch.h`, this represents the structure of our code. As presented before CBS uses a tree structure to store intermediate solutions, each node splitting in two other nodes to add a constraint. To represent this tree, we defined its nodes as in figure 6.2.

Each node most importantly contains the constraint it introduces (either an edge constraint or a vertex constraint), the current solution, the current solution's cost and the computed set of conflicts present in the solution.

Instead of storing the entire solution in each node, only the newly re-planned path is actually stored to save memory, this allows for impressively low memory usage as we will see later, the other paths are retrieved from the parent nodes when needed.

The high level algorithm searches in this structure for a node where the solution is valid, namely where the set of conflicts is empty.

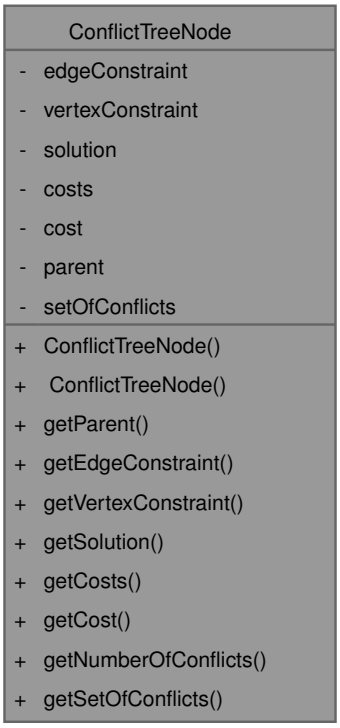


Figure 6.2: ConflictTreeNode class

For each node the general A* algorithm is called to solve one single agent problem per agent, taking into account all the external constraints added so far.

6.2 Conflict Avoidance Table

The intuition behind the CAT for CBS is not unlike the one for CAT for ID. Here, in addition to adding hard constraints necessary for the CBS algorithm, we also add the whole paths of the agents as soft constraints. This means that the low level searches will tend to avoid the paths of other agents (if possible while keeping optimality guarantees), minimising the risk of future conflicts.

6.3 Disjoint Splitting CBS [22]

6.3.1 Positive constraints

As stated before, disjoint splitting makes use of *positive* constraints, where we only used *negative* ones so far. To model this positive constraint, we use two distinct methods for two different points of view. First, a negative constraint is added for all

the agents so that they avoid the cell, as the low-level search only supports negative constraints. Then, we ensure that our agent remains at its current space-time position. As the agent is actually already there, one only has to ensure that he will not be re-planed out of it. This is done by an alternative re-planing strategy in which we only re-plan parts of the path that lie between positive constraints (and/or start/target position), taking the timespan into account. This way, positive constraints are never disturbed as a consequence of a later re-plan. So these two actions have for consequence that other agents will avoid the conflicting vertex, and we prevent our agent from not being at the conflicting space-time position by ensuring he never moves from it. This combination is equivalent to a theoretical positive constraint.

6.3.2 Problem with the Fuel objective

But this brings up the question of which objective function to use for the single agent subproblems. If the search couldn't find a path that fits between the fixed timesteps, the new node isn't added to the open list, so we ideally want to minimise the makespan or at least consider solutions with a *small enough* makespan. Bu to keep optimality the newly-planned path must minimise the same function as the global search. This is fine for makespan and SumOfCosts as they are equivalent for one agent, but this poses a problem for the Fuel objective. Indeed, minimising the fuel for the subproblems could very well yield solutions that don't fit in the current span, while such solutions exist. We thus would need a way of finding the solution minimising fuel consumption among the ones fitting in the span, which is not possible to model using simple tie-breaking strategies as such a solution could have a higher fuel cost than the global minimum. Because of this, our implementation of CBS with disjoint splitting is not optimal for the Fuel objective, as we use the global objective functions for the subproblems as well and this could result in the optimal solution not being explored.

Chapter 7

Performances

7.1 General notes

7.1.1 Problems sources

All the tests were performed on maps and scenarios from [1]

7.1.2 System specifications

CPU	16 threads, 8 cores @ 3.2Ghz Ryzen 7 5800h
RAM	2x16 GiB of CL20-22-22 3200MT/s DDR4
SWAP	8 GiB of lz4 compressed zram + 26 GiB on SSD

7.1.3 Constants among plots

The first plot comparing objective functions and heuristics which required to run all the algorithms and variants with all combinations objective functions and heuristics, on all the problems was made with a maximum of 50 agents and 30 seconds of timeout for each instance and was run from an arch linux TTY with nothing else running on the computer.
The other plots were made using `SumOfCosts` and the `Optimal` heuristic with a maximum of 100 agents with a 60 seconds timeout and ran alongside KDE Plasma, physical memory usage was limited to 28 GiB with a soft limit at 26 GiB[1] using systemd-run to avoid crashes.
The instances were executed concurrently, the number of threads being determined based on memory usage for each algorithm independently so that swap is used as little as possible only to avoid crashes (e.g. 3 for A* and 12 for CBS).

[1] Soft limits allow and promote usage of swap while hard limits kill the task entirely

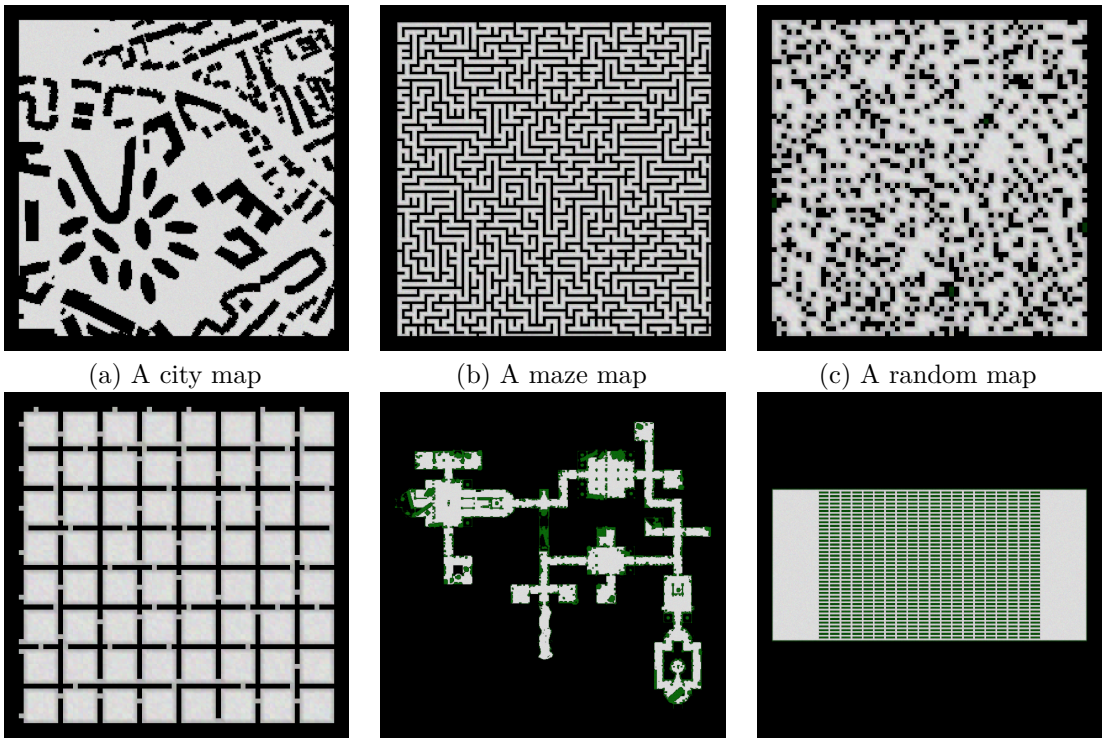
Note that the running time plots only consider successful searches, as not doing so yields plots symmetrical to the success rate ones. This results in the rightmost part of the graphs being based on less data samples and thus subject to more volatility. Finally, the y axes of all success rate plots are in logarithmic scale for the sake of readability, except for the Operator Decomposition plot where it's not needed.

7.1.4 Note on performance plots for some maps

Despite our plots going as far as 100 agents, some maps don't have scenarios with that much agents. This is notably the case of our empty map, it only allow 64 agents and is small enough that the 64 agents problem gas a 0 success rate, this is to be considered while analysing the following plots.

7.1.5 Map types

In the following section, some plots mention different "map types". Figure 7.1 gives examples of what each map type looks like. Empty maps are not included since they are empty.



(a) A city map (b) A maze map (c) A random map

(d) A room map (e) A videogame map (f) A warehouse map

Figure 7.1: Examples of map of each type

7.2 Objective functions and Heuristics

Figure 7.2 compares the success rates and running times of all combinations of heuristics and objective functions, averaged over all problems and algorithms.

We first observe that the optimal heuristic has a higher success rate overall, this is expected as it is more likely to loacally prune off unfeasible and suboptimal paths[2]. It is also expectedly slower, as it is more computationally expensive, Manhattan being $\mathcal{O}(1)$.

One also observes that the Fuel objective function performs notably worse than the other ones when it comes to the success rate. This can be explained by the fact that the cost for the *wait* action is zero and that more nodes with equal costs are generated in the A* algorithm.

The `Makespan` objective having a slightly better success rate is also expected as it gives more freedom to the agents, agents taking less time are less constrained as long as they don't take more time than the one that takes the longest.

For the remaining of the tests, we will only consider the `Optimal` heuristic along with the `SumOfCosts` objective as it is the most common combination, it also has the best minimal performance as `Makespan` was found to perform very bad with the A* algorithm.

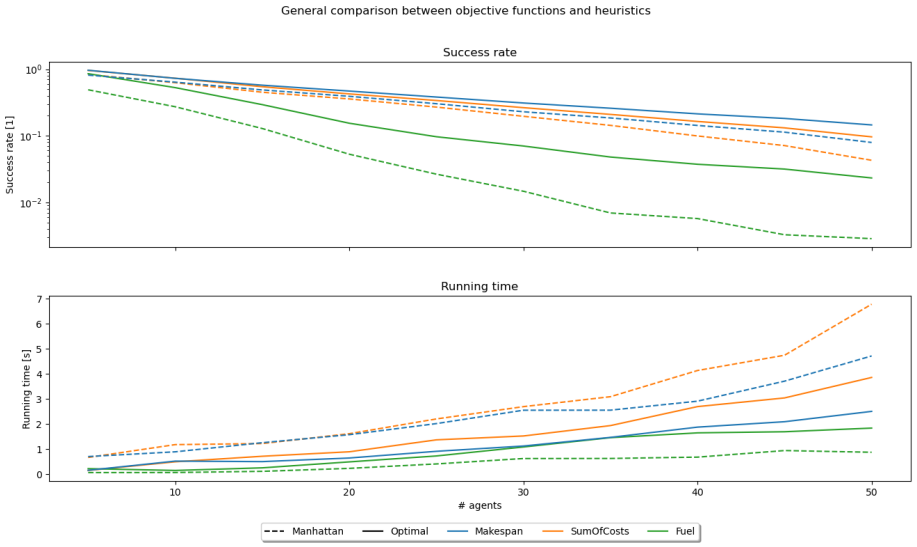


Figure 7.2: Average performance of objective functions and heuristics

[2]The Manhattan heuristic doesn't yield impossible or suboptimal solutions *in the end*, but the Optimal heuristic will never engage in paths from which the target is unreachable as the lower-level search will take obstacles into account.

7.3 A* and Operator Decomposition

Figure 7.3 compares how the A* algorithm performs with and without Operator Decomposition. It is clear that this enhancement hugely upgrades the algorithm. When the plain A* algorithm failed, it was most of the time due to either memory usage or timeout. As OD aims at lowering the number of explored nodes, it is expected that it mitigates those problems.

One also sees that this algorithm doesn't fare very well overall, having a quickly very low (though non zero) success rate.

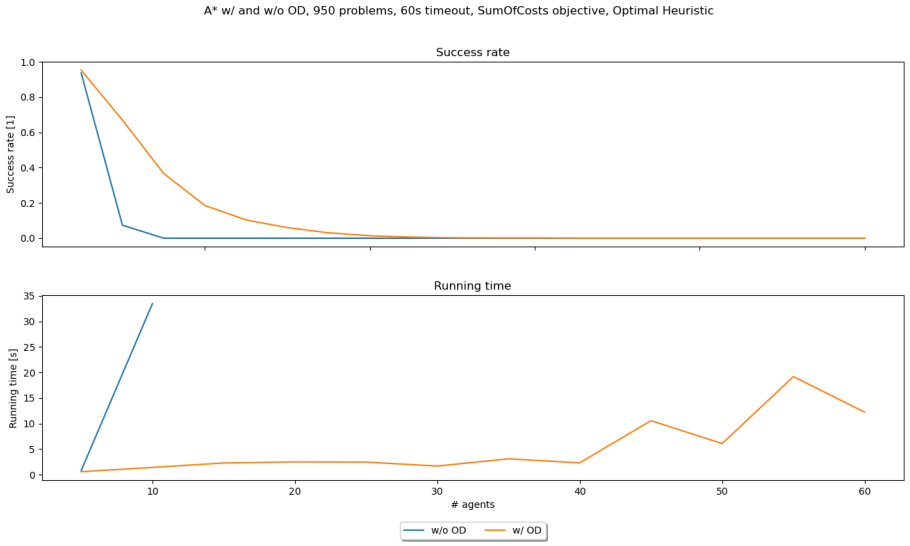


Figure 7.3: Performance comparison between A* w/ and w/o Operator Decomposition

7.4 Performance of ID variants using A* low level search

The first thing one can notice is that the usage of a CAT greatly improves the success rate, creating a visible branching in the plots, while not having a noticeable impact on running times.

The same can thus not really be said about the EID from what one sees in this figure. Figure 7.5 hopefully provides more insight by comparing their performance on specific map types. This second figure indeed tells us that EID is less suited for problems where it is more difficult for groups to avoid each other, namely maps with corridors or tight bottlenecks. However it seems to have a positive effect on

less dense maps presenting less tight passageways like warehouses and empty maps. On other types of maps, the overhead induced by the re-planning attempts seem to grow more important than their benefits. This could very well be a consequence of an error in our implementation.

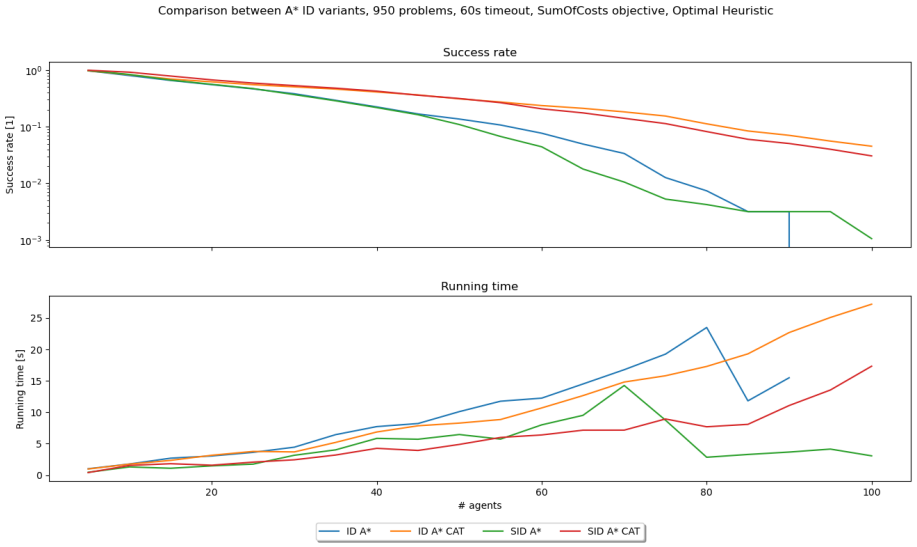


Figure 7.4: Performance comparison between variants of ID using A* low-level search

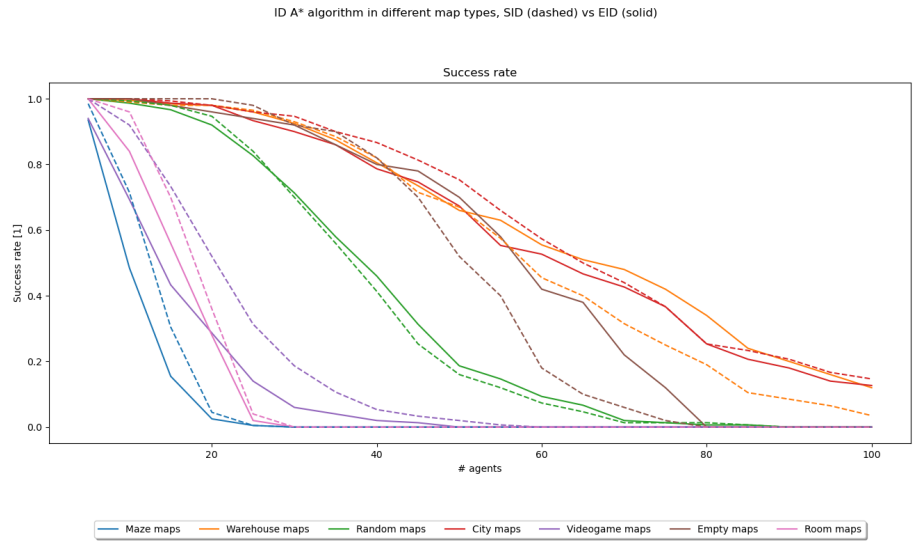


Figure 7.5: Performance comparison between EID and SID using A* low-level search on different map types

7.5 Performance of ID variants using CBS low level search

Figures 7.6 and 7.7 contain the same information as in the previous section but for the CBS low-level search. Globally, the same observations can be made, apart from the advantage of CAT being less noticeable but present.

However the situation with EID is even worse, its benefits being at best very small and at worse very negative. Note that it maybe because of CBS being less suited as an ID sub-level search algorithm as it shines in bigger problems, the accumulation of overheads coming from EID and CBS itself could explain this difference

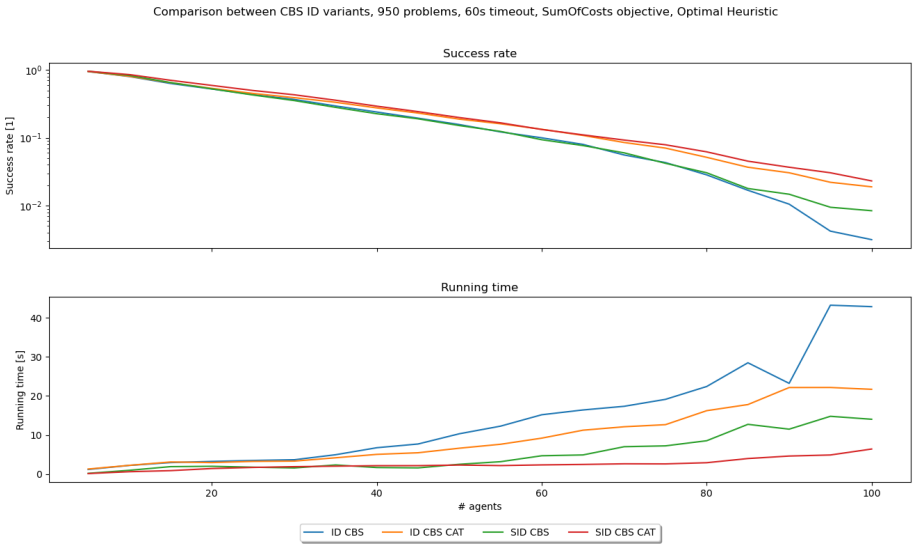


Figure 7.6: Performance comparison between variants of ID using CBS low-level search

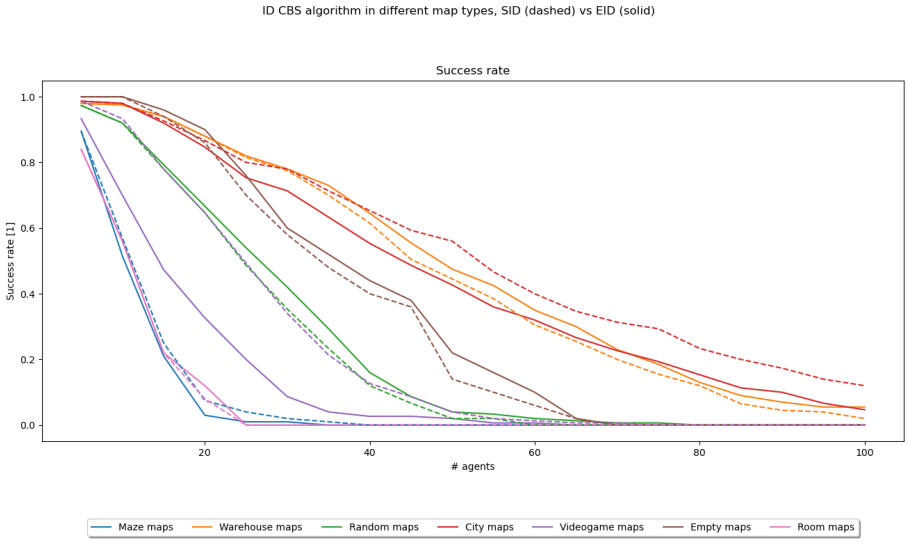


Figure 7.7: Performance comparison between EID and SID using CBS low-level search on different map types

7.6 Performance of ID, A* vs CBS

Figure 7.8 shows the respective performances of CBS and A* low-level searches used in the SID algorithm. Regarding success rate, there's a clear advantage for A*, while CBS is best at running fast. The difference in success rates can be explained by the fact that CBS best shines when dealing with problems with more agents, suffering for a bigger overhead for smaller problems. The ID algorithm specifically aiming at lowering the size of the travel groups it is then expected that A* performs better, the plots being limited to 100 agents and 60s of timeout also hides the behaviours of those methods with bigger problems.

Figure 7.9 further showcases the superiority of the A* low-level search in all map types when used in ID.

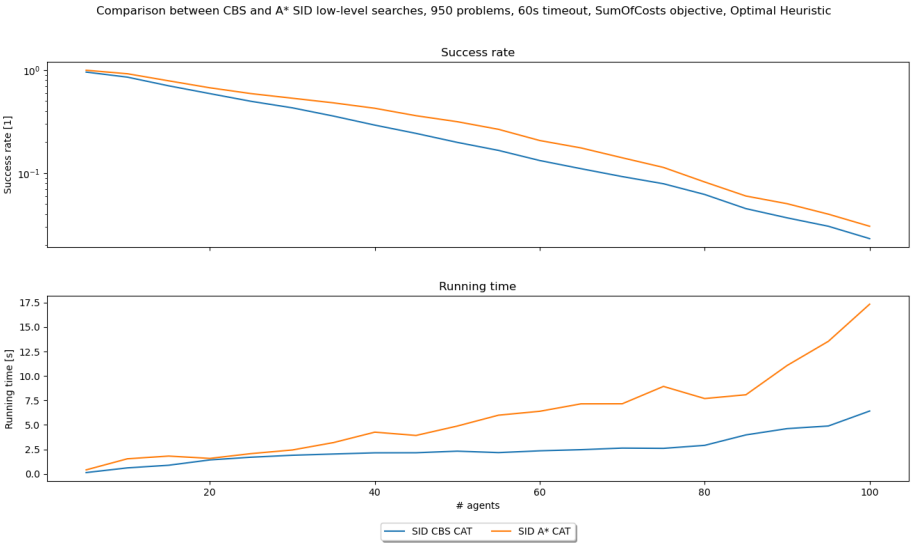


Figure 7.8: Performance comparison between SID with CBS and with A* low-level searches

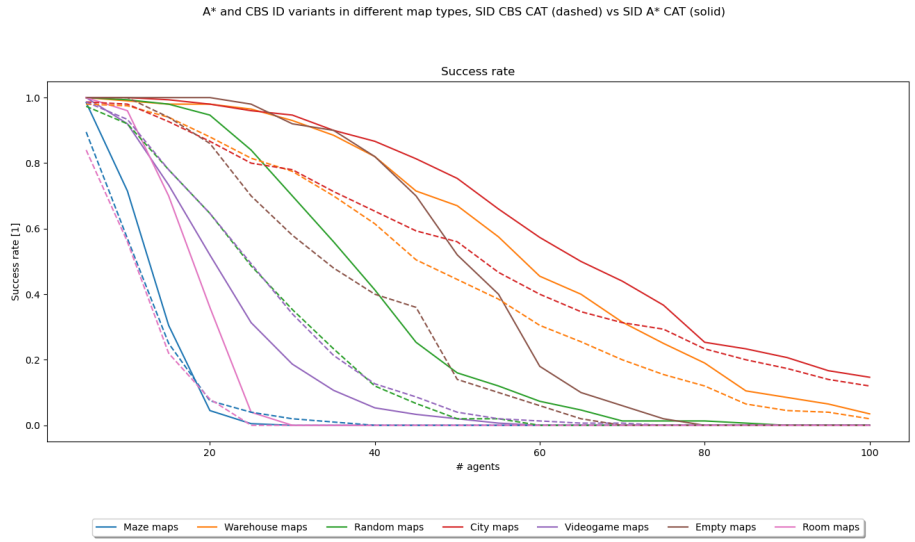


Figure 7.9: Performance comparison between SID with CBS and with A* low-level searches on different map types

7.7 Performance of CBS variants

Figure 7.10 shows how CBS enhancements affect the success rate and running time of the algorithm, it is clear that both enhancements have a positive effect on success rates, the combination of both clearly dominating the other curves. The enhancement also barely affect running times, which is expected since these don't imply a lot of overhead and also reduce the amount of work needed to find a solution.

On figures 7.12 and 7.11, one can see the effect of DS and CAT independently on different map types. The usage of CAT seems to be beneficial in all map types, except for "room" maps where large rooms are separated by walls with only very small doors, it is thus more difficult to avoid the paths of other agents as they all go through the same small doors.On the other hand, it seems to be most beneficial in "empty" maps, where the contrary holds. Indeed, it is way easier to avoid paths of other agents in such maps as there is plenty of room to do so.

As for Disjoint Splitting it is truly beneficial for all map types, as it affects how conflicts are handled it doesn't rely on agents being able to avoid other agent's whole paths. It can also be said that DS has a greater effect on maps inducing a lot of conflicts, such as rooms and the empty map. This is expected as DS effectively eliminates duplicate search efforts between branches ant thus helps a lot more when a lot of branching occurs.

Finally, figure 7.13 shows how the combination of both enhancement is globally profitable. They also seem to have a collaborative effect as the gap between solid and dashed curves seem to be greater than the sum of their independent gaps. This is most visible for "Warehouse" and "City" maps.

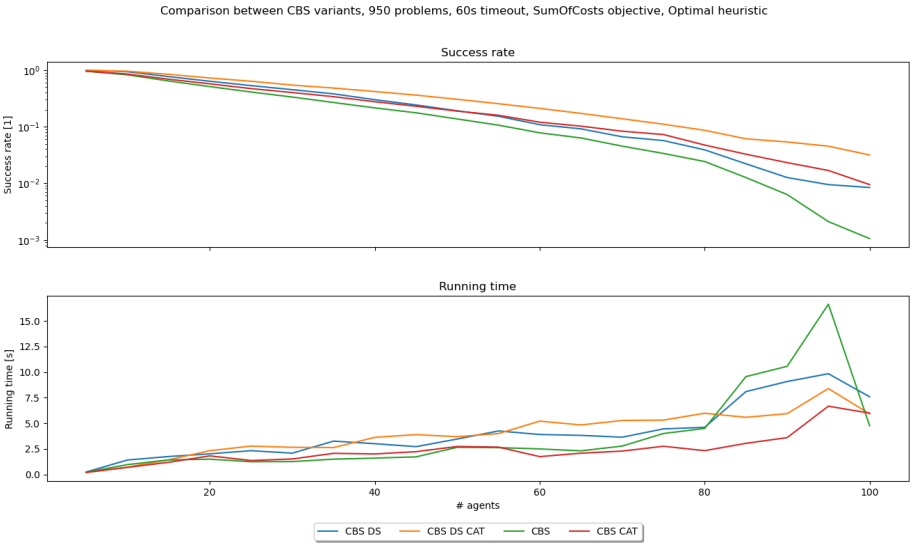


Figure 7.10: Performance comparison between CBS variants

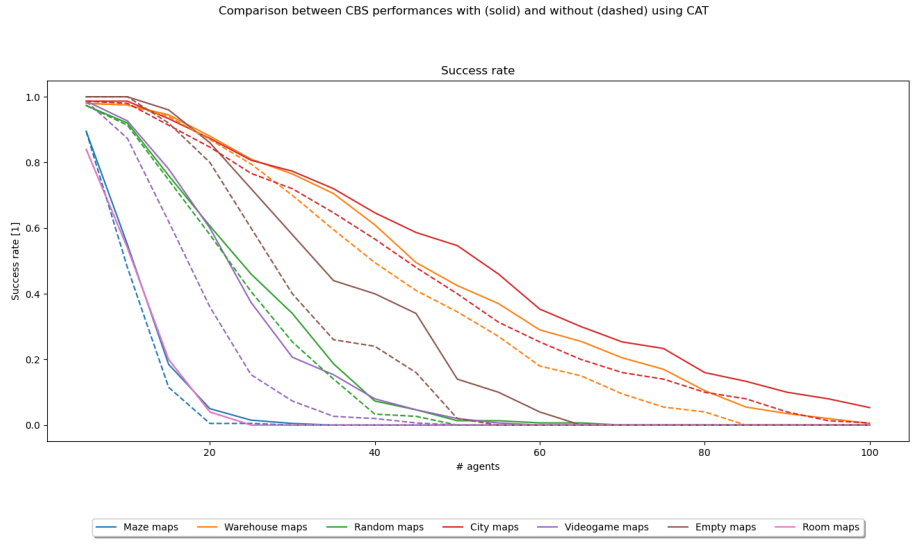


Figure 7.11: Effect of CAT on different map types

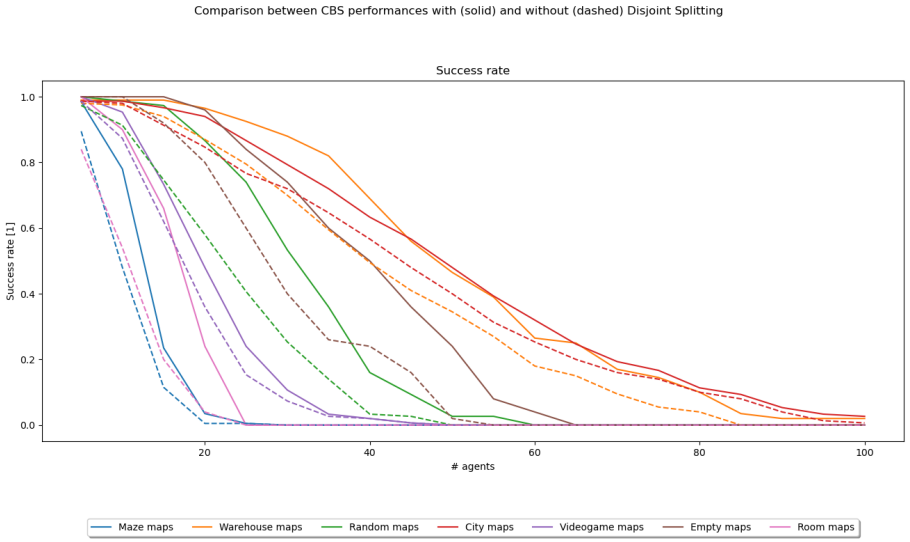


Figure 7.12: Effect of DS on different map types

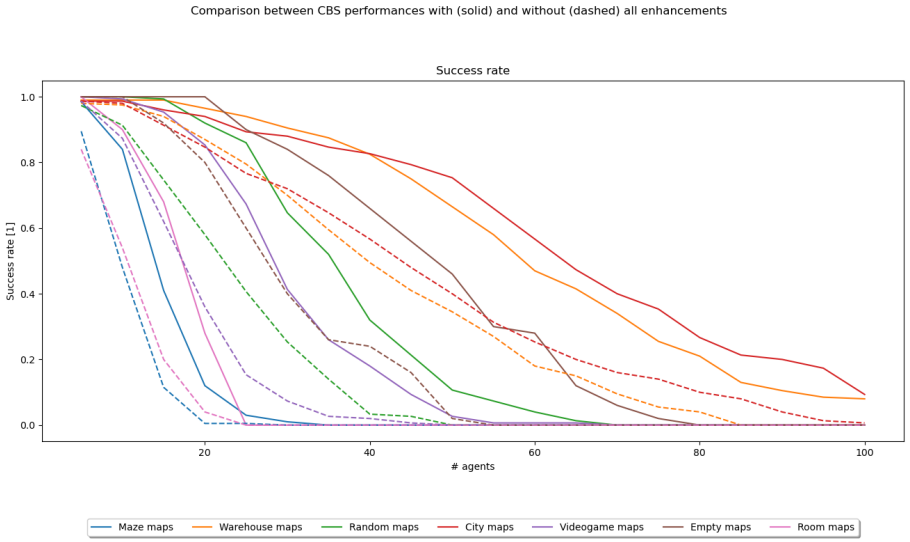


Figure 7.13: Effect of both enhancements on different map types

7.8 Best variant of each algorithm

To help us figure out what's the best performing algorithm in our library, 7.14 summarises the best performing variants of each algorithm overall. One sees that

the best variants of CBS and ID both perform very well and that the difference is quite tight.

Figure 7.15 provides much more nuance and allows to identify the best suited algorithms for some types of maps. For most map types, the difference is still minimal, but some types are highlighting some interesting differences. First ID seems superior when it comes to the empty map. This could be explained by the presence of smaller travel groups that manage to avoid each other quite well, while having a lot of internal conflicts.

As for "Videogame" maps, there's a clear advantage for CBS. On these maps, ID could be disadvantaged by the tighter passageways between the bigger zones, preventing travel groups to remain of a manageable size and thus exposing A*'s weakness against bigger problems.

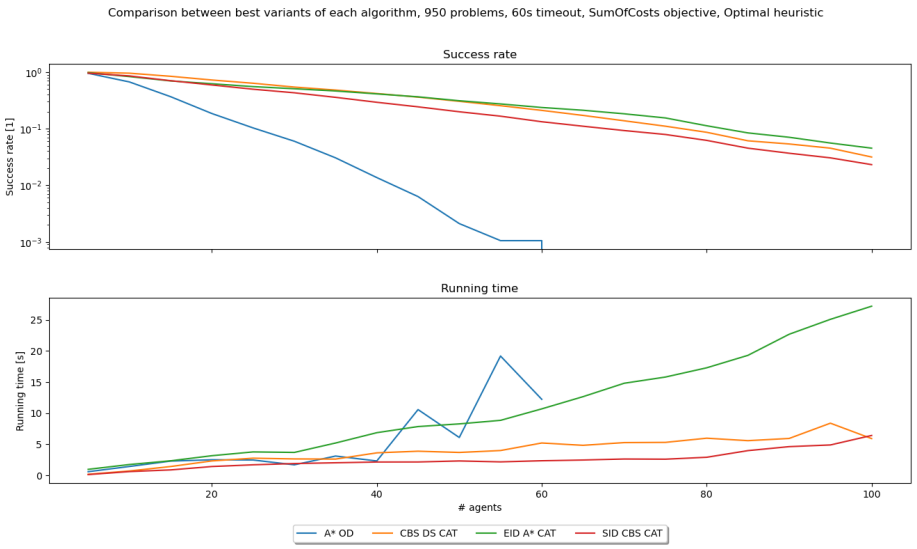


Figure 7.14: Performance comparison between best variants of each algorithm

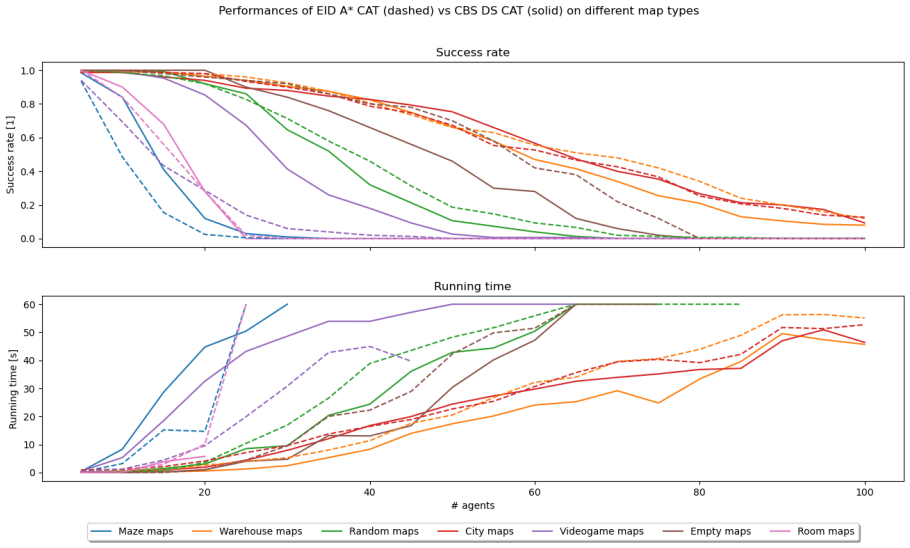


Figure 7.15: Performance comparison between two best algorithms on different types of maps

7.9 Conclusions

EID with the A* sub-level search using a CAT and CBS with Disjoint Splitting using a CAT are thus two candidates for the title of best algorithm in our library, this depends on the type of map in which the problem takes place. An other important thing to consider is resource usage, A* and thus the ID variant that uses it are very memory hungry. During the collection of our test data, ID A* had to be run on 3 threads only to remain in the 32 GiB of physical RAM (the usage of swap being avoided for performance bias reasons), while CBS could run on 14 threads out of 16 and still barely use a couple of GiB of RAM.

For this reason and the fact that the only map where ID has a clear advantage is the least realistic one (not many problems take place on empty maps), we tend to give the title to CBS which was also the main goal of this master thesis.

Chapter 8

Further work

This thesis has been an implementation work from scratch of complete and optimal algorithms to solve the MAPF problem. It could serve as a foundation in the future for making improvements to the algorithms already implemented. Here is a non-exhaustive list of ways to enhance this library.

- Safe Interval Path Planning (SIPP) could be implemented. It is a specialisation of A*. Therefore, the Generic A* algorithm could be reused, and other transition functions and state would need to be defined. [27]
- In A* with OD, we could consider when two multi-agent states are truly equal (`getHash()` and `isEqual()`). In our library, two multi-agent states are considered equal if everything they contain is equal. We could sometimes safely treat states that are not equal but have the same post-move positions (positions of the already assigned agents) as duplicates. [34]
- Also in A* with OD, we put standard nodes as well as intermediate nodes in the closed list. The intermediate descendants of a standard node form a tree rooted at the standard node because a move assignment to an agent cannot be changed until all agents have been assigned moves. Because duplicate intermediate nodes would have duplicate standard ancestor nodes, A* will never encounter duplicate intermediate nodes. Therefore, it would be better to put only standard nodes onto the closed list. [34]
- It would be very straightforward to add follow conflicts (or other types of conflicts) to the problem, especially in ID and CBS algorithms. In addition to adding edge conflicts and vertex conflicts to the set of conflicts, the `calculateSetOfConflicts()` and `updateSetOfConflicts()` functions could also detect follow conflicts.

- A valuable research effort would be to explore how to modify the algorithms if the problem contains capacity constraints on areas of the map, such as (S, k) constraints where at most k agents can be in the set of vertices S at any time. Relevant papers include [37] for vertex capacity constraints and [7] for k-robust MAPF (allowing robots a delay of k time steps while ensuring collision-free movement).

- ICTS could be implemented. [31]

- We have already improved CBS at the high level with disjoint splitting. But there are still numerous ways to enhance CBS, particularly regarding the conflict tree nodes to be processed, such as using heuristics to guide the search [20]. Currently, we resolve conflicts with the smallest time step first, but we could prioritise certain conflicts based on other criteria [9]. We could also consider symmetry-breaking constraints ([23][21] or mutex propagation (generalized symmetry breaking) [39]). An article on ICBS has also been written and could be implemented, which discusses cardinal conflicts [10].

- Meta-Agent CBS could be implemented: not restricting the low level of CBS to a single-agent search and allowing groups of joint agents. [30]

- Directed graphs could be considered. Our graph structure supports directed graphs, but the map format used by [1] doesn't, as well as some of our algorithms (e.g. it is trivial for RRA*).

- Finally, in our library, we have assumed discrete time. We could solve the MAPF problem with continuous time by generalising CBS (CCBS) [6][5] or using MDDs + reduction to SAT [36].

Conclusion

To conclude, we implemented A* for single agent and multi agent problems, Independence Detection, Cooperative A* and CBS, each one with various enhancements and variations. We bundled them all in a single library, designed to enable for re-using important building blocks for the more complex and higher-level algorithms and to be straightforward to use. We also produced a complete documentation using doxygen, a separate solution visualisation program, and collected data on the success rates and running times of the implemented algorithms. With this data, we then proved and quantified their efficiency, the impact of various parameters, enhancements and of the map type, and how they compare to each other in different scenarios.

Bibliography

[1] https://movingai.com/benchmarks/mapf/index.html.

[2] C++ standard library documentation for set. https://en.cppreference.com/w/cpp/container/set. [Online; Access on June 18, 2023].

[3] C++ standard library documentation for unordered_set. https://en.cppreference.com/w/cpp/container/unordered_set. [Online; Access on June 18, 2023].

[4] Visio ing consult. http://www.visioingconsult.be/fr/accueil.html. [Online; Access on August 11, 2023].

[5] Anton Andreychuk, Konstantin Yakovlev, Eli Boyarski, and Roni Stern. Improving continuous-time conflict based search, 2021.

[6] Anton Andreychuk, Konstantin Yakovlev, Pavel Surynek, Dor Atzmon, and Roni Stern. Multi-agent pathfinding with continuous time. *Artificial Intelligence*, 305:103662, 2022.

[7] Dor Atzmon, Roni Stern, Ariel Felner, Glenn Wagner, Roman Barták, and neng-fa Zhou. Robust multi-agent path finding. 07 2018.

[8] Richard Ernest Bellman. *An Introduction to Artificial Intelligence : Can Computers Think?* Boyd & Fraser Publishing Company, San Francisco, 1978.

[9] Eli Boyarski, Ariel Felner, Pierre Le Bodic, Daniel Damir Harabor, Peter James Stuckey, and Sven Koenig. f-aware conflict prioritization & improved heuristics for conflict-based search. In *AAAI Conference on Artificial Intelligence*, 2021.

[10] Eli Boyarski, Ariel Felner, Roni Stern, Guni Sharon, David Tolpin, Oded Betzalel, and Eyal Shimony. Icbs: Improved conflict-based search algorithm for multi-agent pathfinding. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, page 740–746. AAAI Press, 2015.

[11] Eugene Charniak and Drew McDermott. *Introduction to Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, 1985.

[12] Michael Erdmann and Tomás Lozano-Pérez. On multiple moving objects. *Algorithmica*, 2(1–4):477–521, nov 1987.

[13] K. Fujimura. *Motion planning in Dynamic Environments*. Springer-Verlag, New York, NY, 1991.

[14] Ofir Gordon, Yuval Filmus, and Oren Salzman. Revisiting the complexity analysis of conflict-based search: New computational techniques and improved bounds, 2021.

[15] 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.

[16] Robert Holte, M. Perez, Robert Zimmer, and Alan MacDonald. Hierarchical a*: Searching abstraction hierarchies efficiently. pages 530–535, 01 1996.

[17] Lev Krayzman, Nilay Kumar, and Scott Lawrence. Pathfinding infopages. `https://mbhs.edu/~lpiper/pathfinding/`. [Online; Access on July 27, 2023].

[18] Raymond Kurzweil. *The Age of Intelligent Machines*. MIT Press, Cambridge, Massachusetts, 1990.

[19] Jean-Claude Latombe. *Robot Motion Planning*. Kluwer, Dordrecht, The Netherlands, 1991.

[20] Jiaoyang Li, Ariel Felner, Eli Boyarski, Hang Ma, and Sven Koenig. Improved heuristics for multi-agent path finding with conflict-based search. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 442–449. International Joint Conferences on Artificial Intelligence Organization, 7 2019.

[21] Jiaoyang Li, Graeme Gange, Daniel Harabor, Peter J. Stuckey, Hang Ma, and Sven Koenig. New techniques for pairwise symmetry breaking in multi-agent path finding. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 193–201. AAAI Press, 2020.

[22] Jiaoyang Li, Daniel Harabor, Peter J. Stuckey, Ariel Felner, Hang Ma, and Sven Koenig. Disjoint splitting for multi-agent path finding with conflict-based

search. *Proceedings of the International Conference on Automated Planning and Scheduling*, 29(1):279–283, May 2021.

[23] Jiaoyang Li, Daniel Harabor, Peter J. Stuckey, Hang Ma, and Sven Koenig. Symmetry-breaking constraints for grid-based multi-agent path finding. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):6087–6095, Jul. 2019.

[24] George F. Luger and William A. Stubblefield. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. Benjamin/Cummings, Redwood City, California, second edition, 1993.

[25] Ryan Luna and Kostas E. Bekris. Push and swap: Fast cooperative path-finding with completeness guarantees. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume One*, IJCAI'11, page 294–300. AAAI Press, 2011.

[26] N. J. Nilsson. Principles of artificial intelligence. 1980.

[27] Mike Phillips and Maxim Likhachev. Sipp: Safe interval path planning for dynamic environments. In *2011 IEEE International Conference on Robotics and Automation*, pages 5628–5635, 2011.

[28] Stuart Russel and Peter Norvig. *Artificial Intelligence : A Modern Approach*. Pearson, fourth edition, 2022.

[29] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison Wesley, fourth edition, 2011.

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

[31] 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.

[32] David Silver. Cooperative pathfinding. Technical report, Department of Computing Science, University of Alberta Edmonton, Canada, 2005.

[33] David Silver. Cooperative pathfinding. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 1(1):117–122, Sep. 2021.

[34] Trevor Standley. Finding optimal solutions to cooperative pathfinding problems. Technical report, Computer Science Department, University of California, Los Angeles, 2010.

[35] Trevor Scott Standley. Independence detection for multi-agent pathfinding problems. In *MAPF@AAAI*, 2012.

[36] Pavel Surynek. Sparse real-time decision diagrams for continuous multi-robot path planning. In *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 91–96, 2021.

[37] Pavel Surynek, T. Kumar, and Sven Koenig. *Multi-agent Path Finding with Capacity Constraints*, pages 235–249. 11 2019.

[38] Marco Valtorta. A result on the computational complexity of heuristic estimates for the a$*$ algorithm. *Information Sciences*, 34(1):47–59, 1984.

[39] Han Zhang, Jiaoyang Li, Pavel Surynek, T.K. Satish Kumar, and Sven Koenig. Multi-agent path finding with mutex propagation. *Artificial Intelligence*, 311:103766, 2022.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**