FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Matej Husár

Improved graph pruning for multi-agent pathfinding using heuristics

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis:	RNDr. Jiří Švancara, Ph.D.
Study programme:	Computer Science - Artificial Intelligence

Prague 2025

I would like to thank my supervisor Mgr. Jiří Švancara, Ph.D., who helped me solve the given topic, for his highly professional and expert guidance throughout the entire period of my research, for his valuable advice, prompt communication, help, willingness, constant support, and human approach. I would also like to thank my family and closest friends for their support and creation of favorable conditions throughout my studies, as well as during the writing of this thesis.

Title: Improved graph pruning for multi-agent pathfinding using heuristics

Author: Matej Husár

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: RNDr. Jiří Švancara, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: In this work, we focus on improving the solution finding process of the multi-agent pathfinding (MAPF) problem by extending graph-pruning techniques through heuristic-based selection of ground vertices, which they rely on. Finding an solution to a MAPF problem which is represented by collision-free paths for all agents from start to goal positions becomes NP-hard when optimality is required. Conventional algorithms struggle with scalability on large maps, motivating alternative approaches. Building on our previous work, we propose five progressively more sophisticated heuristics for selecting ground vertices based on agent paths. These approaches introduce various improvements, such as minimizing spatial and temporal conflict on vertices and edges while adding agent prioritization. Experimental evaluations demonstrate that the RPS algorithm presented in the end significantly outperforms the others, achieving the highest success rate and the fastest computation times and, in many cases, solving instances independently without the need for further solving. Our results confirm that careful ground vertex selection can substantially improve MAPF solving efficiency by graph-pruning techniques, especially on large-scale maps.

Keywords: Multi-agent pathfinding, SAT, Pruning heuristics, Ground vertices

Název práce: Vylepšení ořezávání grafu pro multiagentní plánování cest pomocí heuristik

Autor: Matej Husár

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: RNDr. Jiří Švancara, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: V této práci se zaměřujeme na zlepšení procesu hledání řešení problému multiagentního plánování cest (MAPF) rozšířením technik ořezávání grafu pomocí heuristického výběru klíčových vrcholů, na kterých tyto techniky závisejí. Nalezení řešení MAPF problému, které je reprezentováno bezkolizními cestami všech agentů z počátečních do cílových pozic, se při požadavku na optimálnost stává NP-úplným problémem. Tradiční algoritmy narážejí na problémy se škálovatelností na velkých mapách, což motivuje hledání alternativních přístupů. Navazujeme na naši předchozí práci a navrhujeme pět postupně sofistikovanějších heuristik pro výběr klíčových vrcholů na základě jednotlivých cest agentů. Tyto přístupy zavádějí různá vylepšení, jako je minimalizace prostorových a časových konfliktů vzniklých na vrcholech a hranách pričemž zavádí prioritizaci agentů. Experimentální vyhodnocení ukazuje, že algoritmus RPS představený na závěr výrazně překonává ostatní přístupy, dosahuje nejvyšší úspěšnosti a nejrychlejších výpočetních časů a ve většině případů řeší instance samostatně bez nutnosti dalšího řešení. Naše výsledky potvrzují, že pečlivý výběr klíčových vrcholů může výrazně zlepšit efektivitu řešení MAPF pomocí technik ořezávání grafu, zejména na rozsáhlých mapách.

Klíčová slova: Multiagentní plánování cest, SAT, Heuristiky pro ořezávání, Klíčové vrcholy

Contents

Introduction

Multi-agent pathfinding (MAPF) is a versatile and important computational problem that is used, for example, in video game control [1], airplane taxiing [2], traffic junctions [3] but also finds its application in the field of warehouse automation [4]. The general task of multi-agent pathfinding is to find paths for two or more agents, from their starting to their goal destination, in such a way that the agents do not block each other or generally cause any collisions between each other [5]. Finding such a solution is a demanding problem, to which, if we add the requirement of optimality, the problem becomes NP-hard [6, 7]. Conventional algorithms can solve it relatively quickly, unless we start to rapidly enlarge the maps on which the agents are to move. In these cases, conventional algorithms become overwhelmed and their calculation time starts to grow exponentially.

In our study, we are focusing primarily on solution optimality in terms of the total time, which will elapse from the start to the moment when all the agents are present at their corresponding final goal position. For finding the solution, we are using a reduction-based approach based on the translation to a Boolean satisfiability (SAT).

In our previous work, we have worked with the idea of reducing the graph by pruning vertices that no agent may need on its path, which allowed us to reduce the number of variables entering the SAT solver, thereby demonstrably reducing the overall calculation time needed to find a solution. For small maps, which can be solved by common algorithms, this method of graph reduction brought the smallest improvement because it is highly likely that agents have to use the entire space for their movement. On the other hand, for large maps, where the agent generally has more space to move around, which represents a significant complication for basic algorithms, pruning approaches turned out to be more efficient, which translated into a significant acceleration of the overall calculation.

In this work, we focus on improving the previous pruning approach using heuristics in terms of selection of ground vertices. The overall performance of pruning approaches depends on the selection of the so-called ground vertices around which the pruning itself is performed. In the original work, these ground vertices were represented by the random shortest paths of the agents from their start to their corresponding goal. This approach turned out to be sufficient to achieve an overall improvement, but at the same time, it represented a place where the given approach could be further improved. In this work, we will propose five progressively more complex approaches to selecting ground vertices based on agent paths, each of which will focus on improving the previous one. We will test individual approaches against each other, while comparing them in terms of the overall computational speed they bring. The great results of the final algorithm for finding ground vertices led us to focus on its ability to find the overall solution independently, without the help of graph-pruning approaches.

1 Definitions

1.1 Multi-Agent Path Finding (MAPF)

1.1.1 MAPF Instance

Definition 1. *A Multi-Agent Path Finding (MAPF) instance is defined as a pair (G, A). The underlying graph structure is represented as a $G = (V, E)$ where V represents the set of vertices of all possible positions that agents can occupy and E represents the set of all possible transitions between these positions. A denotes the set of all agents, while every agent $a_i \in A$ is specified by a pair (s_i, g_i) of two positions from V, where s_i denotes the corresponding starting position of the agent while g_i denotes the corresponding goal position of the agent.*

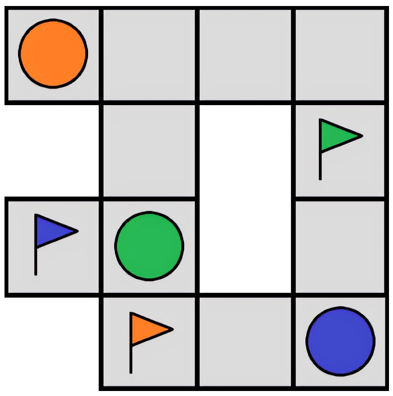

Figure 1.1 Simple grid-based MAPF instance example, where colored circles represent individual agents, and flags with corresponding colors represent their respective goal positions.

The figure 1.1 shows a lattice instance, but the MAPF problem can be solved on any graph.

Definition 2. *A plan for agent i is defined as an ordered sequence of vertices that specifies the concrete position of agent at each discrete time step in the graph. This sequence of agent steps is denoted by π_i, where the expression $\pi_i(t) = v$ indicates that agent i occupies the vertex v at the time step t. The length of the plan, representing the total number of time steps, is denoted by $|\pi_i|$.*

Definition 3. *A plan π_i for agent $a_i = (s_i, g_i)$ is said to be valid if and only if it satisfies the following conditions: the agent starts at its initial vertex, i.e., $\pi_i(0) = s_i$, and ends at its goal vertex, i.e., $\pi_i(|\pi_i|) = g_i$. Furthermore, for every time step $t = 0, \dots, |\pi_i| - 1$, the agent either remains stationary, i.e., $\pi_i(t) = \pi_i(t+1)$, or moves to an adjacent vertex, i.e., $(\pi_i(t), \pi_i(t+1)) \in E$.*

From the above definitions, we can see that time is discrete for each time step, and therefore for each agent, it is true that, in order to find a valid path, it either moves along the edge to the neighboring vertex or remains at the current position it occupied in the previous time step. We consider all agents to be homogeneous and therefore it is true that crossing one edge takes the same time for each of the agents. Since each edge is of the same length, it is also true that all agents move around the map at the same speed.

Definition 4. *A joint plan* Π *is defined as a collection of valid individual plans, one for each agent, such that all plans are of equal length. The joint plan* Π *is considered valid if it contains no conflicts between any pair of agents throughout its whole length. Thus, each solution to the given MAPF instance is determined by the corresponding existence of the valid joint plan. The overall length of the joint plan is denoted by* $|\Pi|$*.*

The definition of a valid joint plan requires that the plans of all agents be of equal length. We can achieve this result simply by saying that an agent with a shorter plan can simply wait at its destination or can stop somewhere else on the way. This is important to consider because when an agent reaches its goal location, it is not removed from the graph but is still on it, while the other agents must reach their destinations without collisions. The entire joint plan ends when all agents arrive at their destinations and are on them at the same time.

The definition 4 describes that a solution to a MAPF instance is represented by a valid joint plan, but that does not necessarily mean that a solution must always exist. Consider, for example, a simple case in which we have a graph with two components and one agent. The agent's starting position would lie in one component, and the target position would lie in the other component. So, it is clear that such an example will not have a solution.

1.1.2 Types of collisions

Finding or solving the plan for the one agent is straightforward, so the main idea is to be able to solve such plans for multiple agents. However, in a multi-agent environment we encounter new problems, which are collisions between agents that arise when they interact with each other. There are of course several types of conflicts that can occur between agents when moving around the map.

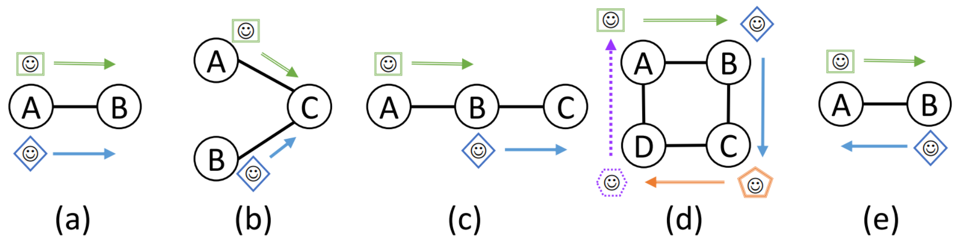

Figure 1.2 Types of possible collision conflicts that can occur between agents in the MAPF environment where (a) represents the *edge conflict*, (b) represents the *vertex conflict*, (c) represents the *following conflict*, (d) represents the *cyclic conflict* and (e) represents the *swapping conflict*. [8]

Definition 5. *An edge conflict (illustrated in Figure 1.2(a)) arises when two agents* a_i *and* a_j *try to change their corresponding positions at timestep* t *by the same edge. Formally,* $\pi_i(t) = \pi_j(t) \wedge \pi_i(t+1) = \pi_j(t+1)$ *and simultaneously* $\pi_j(t) \neq \pi_j(t+1)$*.*

Definition 6. *A vertex conflict (illustrated in Figure 1.2(b)) arises when two agents a_i and a_j share their corresponding positions on the graph at any given time step t. Formally, $\pi_i(t) = \pi_j(t)$.*

Definition 7. *A following conflict (illustrated in Figure 1.2(c)) arises when the agent a_i attempts to move to a position that was occupied shortly before by the second agent a_j. Formally, $\pi_i(t+1) = \pi_j(t)$.*

Definition 8. *A cyclic conflict (illustrated in Figure 1.2(d)) arises when a group of k agents $a_i, a_{i+1}, \ldots, a_{i+k}$ where $k > 2$ are trying to move sequentially in a closed loop. Formally, $\pi_i(t+1) = \pi_{i+1}(t) \wedge \pi_{i+1}(t+1) = \pi_{i+2}(t) \wedge \cdots \wedge \pi_{i+k-1}(t+1) = \pi_{i+k}(t)$.*

Definition 9. *A swapping conflict (illustrated in Figure 1.2(e)) arises when two agents a_i and a_j change their corresponding positions using the one concrete edge in the same timestep. Formally, $\pi_i(t+1) = \pi_j(t) \wedge \pi_i(t) = \pi_j(t+1)$ and simultaneously $\pi_i(t) \neq \pi_i(t+1)$.*

It is clear from the definitions themselves that prohibiting some of the conflicts listed will automatically prohibit some of the others. For example, if we prohibit pursuit conflict, we will automatically prohibit exchange conflict as well as cyclic conflict. However, the opposite implication is not true. Another example is that if we, for example, prohibit vertex conflict, we will automatically prohibit edge conflict. The opposite is also not true.

If we wanted to consider some real-life example where the moving agents are physical objects, such as airplanes, cars, ships, or robots, we would have to automatically prohibit vertex, edge, and exchange conflicts because these agents would have to share a common physical space, which is not possible in reality. On the other hand, we certainly allow pursuit and cyclic conflicts, since physical objects can move behind each other.

In this work, we will use the conflict setting mentioned above in order to simulate a real environment. We will call such a setting *parallel motion*. In other setting which, in comparison with parallel motion, also prohibits following conflicts, which at the end implies that the cyclic one is also automatically prohibited, it is referred to as *"pebble motion"* [9].

1.1.3 Cost functions

So far, we have only discussed instances and the valid solution itself, but we have not said anything about the quality of a valid solution, or about how difficult it is to find a valid solution in terms of the required computational power.

Two evaluation functions are used most often to evaluate the quality of a solution, namely *Makespan* [10], which describes the moment when all agents are at their target position and *Sum-of-costs* [11], which describes the sum of the time steps needed for all agents to be at their target positions and thus not need to move any further.

To define these two functions, we still need to introduce the notation T_i, which denotes the number of time steps required for agent a_i to reach its target position without having to leave it later.

Definition 10. *For agents A the Makespan of their plan Π is defined as:*

$$Mks(\Pi) = \max_{a_i \in A} T_i$$

Definition 11. *For agents A the Sum-of-costs of their plan Π is defined as:*

$$SoC(\Pi) = \sum_{a_i \in A} T_i$$

Moreover, finding a valid solution may require finding a solution that tries to minimize one of the two defined evaluation functions. It may not be obvious at first glance that optimizing a solution using one or the other function often leads to a different solution [12].

There exist many algorithms for finding a feasible solution to a MAPF problem in polynomial time, but finding a solution that is either makespan or sum-of-costs optimal falls into the class of NP-hard problems [6, 7]. Even deciding whether a solution exists in T time steps is an NP-complete problem [13].

1.1.4 Time-expanded graph

The paths of agents through a graph do not always have to be acyclic. It often happens that an agent visits a vertex more than once during its journey. A *time-expanded graph* [14] is used to better visualize the movement of agents through a graph.

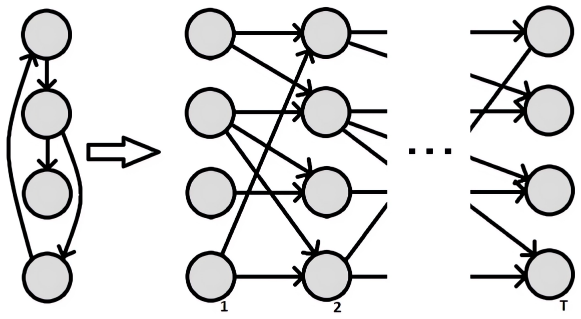

Figure 1.3 An example of transforming a directed graph into a time-expanded graph with T layers.

Definition 12. *Let $G = (V, E)$ be an undirected graph and $n \in \mathbb{N}$. Then, the time-expanded graph with $n + 1$ time layers created from G is a directed graph $Exp_T(G, n) = (V \times \{0, 1, \ldots, n\}, E')$, where $E' = \{([u, t], [v, t + 1]) \mid \{u, v\} \in E; t = 0, 1, \ldots, n - 1\} \cup \{([v, t], [v, t + 1]) \mid v \in V; t = 0, 1, \ldots, n - 1\}$.*

We create a time-expanded graph by adding T copies of the graph G on top of the original one, which creates additional $T \times |V|$ vertices, which can be imagined as individual time layers. We connect these layers with edges such that when there is an edge $(u, v) \in E$, then for all layers $i \in \{0, \ldots, T\}$ we add an edge (u_i, v_{i+1}) (such edges represent movement between vertices) and for each vertex $u \in V$ we add an edge (u_i, u_{i+1}) (these edges represent waiting at a vertex). Agents move along such a graph by starting at the first layer and moving to the next layer with each time step. It is obvious that we can easily create such a graph from a directed graph 1.3.

2 Solving MAPF

In general, there are two main ways how MAPF can be solved: optimal and sub-optimal. Sub-optimal algorithms represent an important category, often focusing on the speed of finding a solution at the expense of its optimality. In this paper, we will also demonstrate and test two sub-optimal approaches, while we will mainly focus on the optimal approach. In general, optimal solvers fall into two main categories.

Search-based solvers

In this category the Conflict-Based Search (CBS) [15] algorithm is the most notable one while in this regard is considered to be the state-of-the-art approach together with its improvements [16, 17]. CBS by its name works in a way that is trying to find the solution by eliminating the conflicts between the agents. For finding the paths corresponding to agents, the algorithm uses the single-agent A^* approach. When a conflict arises between two agents, the algorithm tries to resolve it in one of two ways. It either forbids the first agent in the conflict position in the conflicting time or does the same for the second agent. This decision-making process creates a binary constraint tree that CBS searches over while finding the optimal solution.

Reduction-based solvers

Reduction-based solvers address a MAPF instance by translating it into a chosen established formalism. Prominent approaches in the literature include translations to SAT [18] and ASP [19], although alternative formalisms offer distinct advantages, as outlined in a recent survey[20]. It is well established that search-based and reduction-based solvers demonstrate complementary strengths across different instance types[21], while search-based methods are particularly effective in large sparse environments, reduction-based techniques tend to perform best in small densely occupied ones. However, the scalability of reduction-based approaches is limited in large environments, motivating various enhancements aimed at reducing problem size. One such strategy involves the construction of a subgraph by eliminating potentially unnecessary vertices [22]. In this work, we extend the subgraph method while still using the reduction to SAT. Both key concepts will be discussed in greater detail in the next sections.

2.1 Solving MAPF by reduction to SAT

In this section, we will show how finding a solution to the MAPF problem can be expressed as solving the general SAT problem by using the reduction. In addition to finding a valid solution, we will look for one that is makespan optimal. We can achieve this by limiting the length of the plan while trying to find a solution to it. In many cases this leads to failure because the problem cannot be solved because of its restriction, so in that case the increase of the given length limit [23] will take place and the finding process repeats.

Let us first show how to reduce a finite-step MAPF to a SAT. We consider finding a solution to a parallel-motion MAPF with a conflict setting and a makespan T. Let us define two sets of variables: $\forall v \in V, \forall a_i \in A, \forall t \in \{0, \dots, T\}$: $At(v, i, t)$, where this means that agent a_i is located (is at position) at vertex v at time t, which corresponds to the position in the time-expanded graph (12); a $\forall (u, v) \in E, \forall a_i \in A, \forall t \in \{0, \dots, T-1\}$: $Pass(u, v, i, t)$ means that agent a_i starts traversing edge (u, v) at time t and arrives at vertex v at time $t+1$, which corresponds to one edge in the time-expanded graph (12). Due to this behavior, we do not declare variables for time T. The agent a_i can also stay at its vertex, so we must add an edge (v, v) to E for each vertex $v \in V$. $Pass(v, v, i, t)$ which means that agent a_i does not leave the vertex v at time t, but stays on it. Conditions by which we can describe the MAPF problem are designed as follows:

$$\forall a_i \in A : At(s_i, i, 0) = 1 \tag{2.1}$$

$$\forall a_i \in A : At(g_i, i, T) = 1 \tag{2.2}$$

$$\forall a_i \in A, \forall t \in \{0, \dots, T\} : \sum_{v \in V} At(v, i, t) \leq 1 \tag{2.3}$$

$$\forall v \in V, \forall t \in \{0, \dots, T\} : \sum_{a_i \in A} At(v, i, t) \leq 1 \tag{2.4}$$

$$\forall u \in V, \forall a_i \in A, \forall t \in \{0, \dots, T-1\} :$$
$$At(u, i, t) \implies \sum_{(u,v) \in E} Pass(u, v, i, t) = 1 \tag{2.5}$$

$$\forall (u, v) \in E, \forall a_i \in A, \forall t \in \{0, \dots, T-1\} :$$
$$Pass(u, v, i, t) \implies At(v, i, t+1) \tag{2.6}$$

$$\forall (u, v) \in E : u \neq v, \forall t \in \{0, \dots, T-1\} :$$
$$\sum_{a_i \in A} (Pass(u, v, i, t) + Pass(v, u, i, t)) \leq 1 \tag{2.7}$$

For a better understanding of the individual conditions, we will describe them in turn. Conditions (2.1) and (2.2) ensure that the starting and ending positions are valid for all agents. The condition (2.3) describes the fact that each agent must be at most one vertex at a time. On the other hand, (2.4) ensures that each graph vertex can be used concurrently, at the same time, by just one agent. To ensure the correct movement of agents around the graph, we have the conditions (2.5), (2.6), and (2.7). Condition (2.5) states that an agent can leave a vertex only along an edge that leads from a given vertex, and condition (2.6) ensures that the agent arrives at the target vertex along a given edge at time $t+1$. The last condition is (2.7), which ensures that two agents cannot swap places on the same edge. The interpreted fulfillment of these conditions corresponds to finding a solution for MAPF with makespan T. [24]

As we have already mentioned, to find a makespan-optimal solution, we gradually increase makespan T until we generate a set of satisfiable conditions. Such an approach will unambiguously find a makespan-optimal solution because when a solution is found, we are guaranteed that there is no solution with a smaller makespan.

2.2 Methods for speeding up the solving

In this subsection, we will focus on already well established methods for speeding up the overall computation time of finding a solution. We will first show and describe basic pre-processing methods, and then focus on the more complex graph pruning method of speeding up the computation.

2.2.1 Use of basic pre-processing

The problem of finding the optimal solution to a MAPF is often non-trivial and requires considerable computational power, while its calculation takes a significant amount of time. An improvement in the terms of reduction of the total number of variables, which describe the problem would be helpful not only for the makespan optimal model but also for the sum-of-costs optimal model. This would result in the computation speed up because the SAT solver would be working with a smaller variable count.

We can speed up the overall computation by better specifying the initial makespan value. Instead of starting with $T = 1$, we can easily determine the initial T as the longest of the shortest paths between the starting position and the goal position of all agents, formally, $LB_{mks} = \max_{a_i \in A} dist(s_i, g_i)$.

The overall calculation speed can be further improved by doing the so-called *pre-processing* [25] for each variable $At(v, i, t)$ that describes that agent a_i occupies the vertex v at time t. With this in mind, we can for some vertices easily in advance determine whether a given agent can be present there or not. A simple example is times 0 and T, because we can say with certainty that all agents must be at their starting position at time 0 and at time T they must be at their goal position again. From this we can easily conclude that if d is the distance between the vertex v and the starting position s_i of the anent a_i, then the agent cannot certainly be at the vertex v at times $0, \ldots, d-1$. This follows from the fact that the agent does not have enough time to travel the distance given to the vertex v. However, the same applies in the direction away from the goal. If the goal position g_i of the agent a_i is distanced d steps away, then we can say that the given agent certainly cannot be present at the given vertex at times $T - d + 1, \ldots, T$ simply because it would not have time to reach its goal position in the remaining time. We can leverage this knowledge of the vertex occupancy to apply the pre-processing calculation also to the $Pass(u, v, i, t)$ variables such that if the vertex is not occupied by the agent at the time t it means that the edges that are connected to this vertex are certainly also not going to be used at the given time t.

Figure 2.1 shows the pre-processing calculation for a single agent with a makespan value of 5. The calculation is made using a wave that propagates from the agent's starting position, which iteratively increases to the size of the makespan. The same calculation is then performed with the wave starting at the agent's target position. Using these waves, it is easy to determine where the agent can be furthest from the starting position at time t and also where it must be at time t, in order to still reach its target position. In this example, pre-processing immediately finds the optimal solution. The numerical values in the left part of the figure (separated by an arrow) tell us where the agent can be furthest from

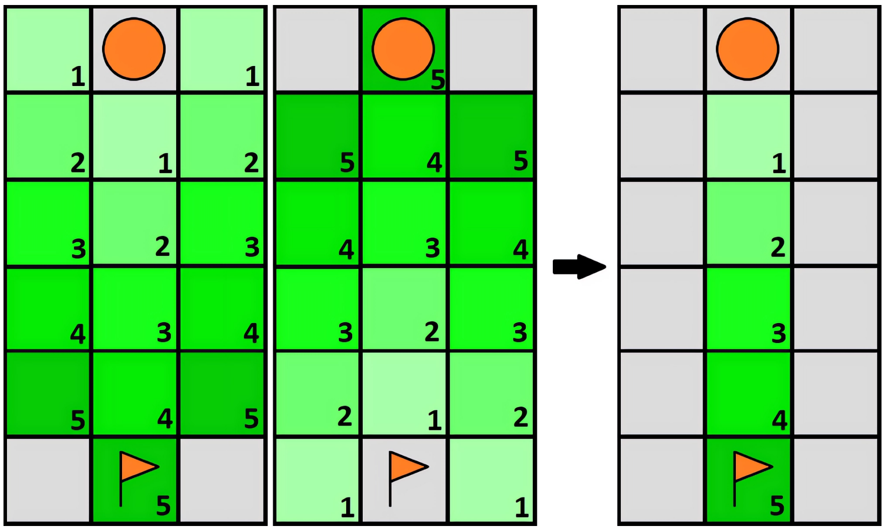

Figure 2.1 An example of basic pre-process computation for a single agent with $T = 5$.

the starting position, respectively, from the target position. On the other hand, in the right part of the figure, the numerical values show us which vertices the agent can be located at at a given time step.

Let us again look at the time-expanded graph (12), which helps us solve the MAPF problem with respect to the makespan optimality. A possible way to visualize it is that the agents do not move through a common one, but each agent has its own time-expanded graph on which it moves. These individual graphs are then connected to each other using conditions that guarantee non-conflict.

One of the advantages of a time-expanded graph is its ability, together with pre-processing, to detect unreachable vertices, which helps us in finding a solution from the perspective of graph connectivity, but also from the perspective of the possible respective path of the agent from the starting position to the goal, for each agent individually. We can use this property to our advantage since we know where the agent cannot be at a given time and, conversely, where it must be. This knowledge can significantly speed up the calculation of the SAT solver.

Another, proved option for reducing vertices and thus input variables to the SAT solver is to remove vertices from the graph before calculating the pre-processing. This approach is shown to be useful in speeding up the calculation when solving maps that have many vertices, and therefore their calculation can be extremely demanding and inefficient when using common algorithms, as shown in our previous work [22], which, to be self contained, we will briefly describe in the next subsection.

2.2.2 Graph Pruning

The subgraph method [22] is a framework developed to enhance the scalability of reduction-based solvers by eliminating vertices from the graph and then systematically adding them back if the solution does not exist without them. Given an MAPF instance $\mathcal{M} = (G, A)$ and a specified set of ground vertices, the method constructs a relaxed instance denoted as $\mathcal{M}_{k,m}$. The parameter k defines the maximum graph distance from the ground vertices to the included vertices in the subgraph, while m determines the cost increase added to the lower-bound metric LB_{mks} which together makes the total allowed cost T.

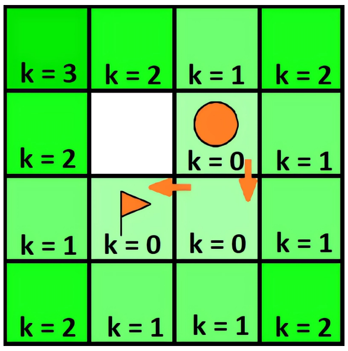

Figure 2.2 Labeled single agent instance which for each vertex shows to which k-restricted subgraph it belongs. Figure layout is taken from [22].

In its original formulation, ground vertices are chosen as those lying on the shortest random paths from the start position of each agent s_i to its goal position g_i. The value of k controls the number of additional vertices incorporated into the subgraph, effectively expanding its coverage, while m specifies the increase added to the lower-bound metric. An illustrative case involving a single agent is shown in Figure 2.2. As k increases, the subgraph expands by forming layers around the initially selected ground vertices, which corresponds to the base case when $k = 0$.

$$\begin{tikzcd}
 & & & \mathcal{M}_{0,0} \arrow[dl] \arrow[dr] & & \\
 & & \mathcal{M}_{1,0} \arrow[dl] \arrow[dr] & & \mathcal{M}_{0,1} \arrow[dl] \arrow[dr] & \\
 & \mathcal{M}_{2,0} \arrow[dl] \arrow[dr] & & \mathcal{M}_{1,1} \arrow[dl] \arrow[dr] & & \mathcal{M}_{0,2} \arrow[dl] \\
\mathcal{M}_{3,0} \arrow[dr] & & \mathcal{M}_{2,1} \arrow[dl] \arrow[dr] & & \mathcal{M}_{1,2} \arrow[dl] & \\
 & \mathcal{M}_{3,1} \arrow[dr] & & \mathcal{M}_{2,2} \arrow[dl] & & \\
 & & \mathcal{M}_{3,2} & & &
\end{tikzcd}$$

Figure 2.3 MAPF instance relaxations for $k_{max} = 3, m_{max} = 2$ taken from the original paper [22].

In the original paper, four different strategies were presented, which search the relaxed instance $\mathcal{M}_{k,m}$ described above in different ways. An example of such an instance is shown in Figure 2.3, which was taken from the original paper. The Algorithm 1 shows the pseudocode of a generic subgraph method for finding a solution, where the generic method *Relax()* can be replaced with one of the four relaxation methods which are briefly described below.

Algorithm 1 Generic subgraph method.

```
1: function GENERIC MAPF RELAXATION($\mathcal{M}$)
2:     $LB = \max_{a_i \in A} |SP_i|$
3:     $(k, m) \leftarrow Initial\_Candidate()$
4:     while not solve_MAPF($\mathcal{M}_{k,m}$) do
5:         $(k, m) \leftarrow Relax()$
6:     end while
7:     return $LB + m$
8: end function
```

Baseline

The Baseline relaxation strategy **B** operates by retaining the entire original graph G effectively selecting a sufficiently large value of $k_{\max}$ while iteratively increasing only the parameter m. This approach corresponds to the standard behavior of vanilla SAT-based makespan-optimal solvers, which do not restrict the graph but instead progressively relax the cost bound to find a feasible solution.

Prune-and-cut

The Prune-and-cut relaxation strategy **P** begins with the smallest possible instance, which is $\mathcal{M}_{0,0}$. If the instance is determined to be unsatisfiable, the parameter k is increased by powers of two until the upper bound $k_{\max}$ is reached. Only after exhausting all values of k does the parameter m increase by one, at which point k is reset to 0 and the process repeats.

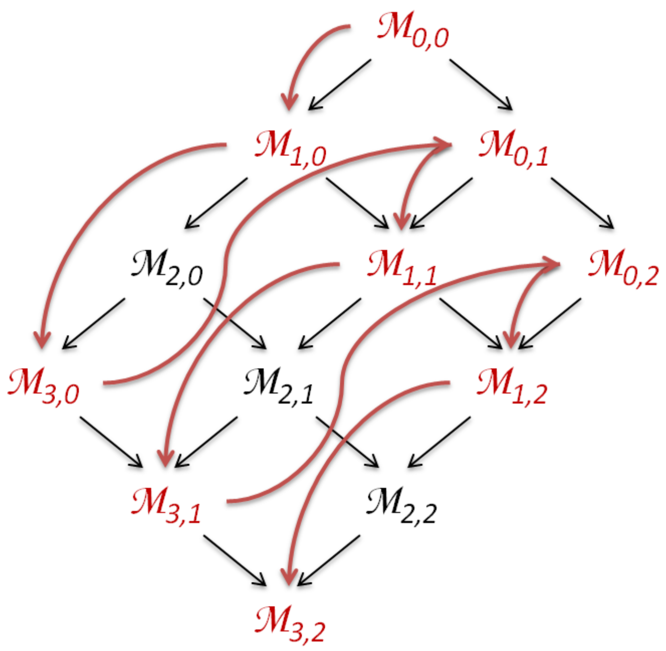

Figure 2.4 The traversal of the instance relaxation lattice by *Prune-and-cut* strategy, where just the relaxed instances which are highlighted are being solved. This figure was taken from the original paper [22].

The instance relaxation lattice traversal strategy of the Prune-and-cut algorithm is shown in Figure 2.4. This strategy has been formally proven to guarantee the discovery of an optimal solution, and empirical evaluations have shown that it consistently outperforms the baseline approach in terms of efficiency.

Makespan-add

The Makespan-add relaxation strategy **M** begins with the instance $\mathcal{M}_{1,0}$ so that $m = 0$ and $k = 1$ is chosen for the increased probability of finding the solution. If the instance is determined to be unsatisfiable, the parameter m is incremented by one until the instance is not satisfiable and the solution is fined. This strategy has been formally proven to be both suboptimal and incomplete, but in the majority of cases, this simple strategy is sufficient to find a solution, while the substantial reduction in the number of vertices within the graph often results in significantly improved computational efficiency, enabling faster solution discovery.

Combined

The Combined relaxation strategy **C** begins with the initial candidate instance $\mathcal{M}_{0,0}$, where both parameters are set to zero, such as $k = 0$ and $m = 0$. If the instance is determined to be unsatisfiable, both parameters are incremented simultaneously by one. This approach reduces the total number of solver invocations by avoiding exhaustive exploration of all possible subgraph reductions. However, this efficiency gain comes at the cost of optimality because it has been formally proven that the Combined strategy is complete, in the sense that it guarantees to find a solution if exists, but it does not necessarily find an optimal one.

Number of shortest paths to consider

A subsequent study [26] investigated an extension of the subgraph method by incorporating multiple shortest paths per agent into the set of ground vertices, rather than restricting to a single path. However, experimental results indicated that this increase in the number of included vertices led to a significant increase in the computation time. Consequently, in this work, we adopt the original strategy of selecting a single shortest path for each agent when constructing the ground set of vertices.

3 Ground Vertices Selection

The heuristics presented in the following chapters describe different single-path strategies for path selection and ordering to optimize the pruning process. Based on our previous work, we decided to focus only on the single-path strategies for the ground vertices selection. These strategies provide various methods for refining the search space based on agent movement preferences, priority sorting mechanisms, and conflict prevention techniques. The individual algorithms will be compared with each other in the next chapter, focusing on various aspects of their performance and behavior.

3.1 Agent single-paths strategies

In this section, we will introduce individual single-path algorithms for finding the shortest paths. We will show basic approaches that do not change the order of agents for which individual paths are searched, namely *Baseline*, *Random*, *Without Crossing* and *Without Crossing at the Same Times*. Next, we will introduce a more advanced approach, *Recursive Path Search*, which already considers the order in which individual paths for agents will be searched, and we will show and explain how such an ordering can greatly facilitate finding a conflict-free solution.

3.1.1 Biased

The *Biased* algorithm is a variant of the Breadth-First Search (BFS) algorithm, where the order of node expansion is fixed, leading to a preference in path selection. In this case, we chose the BFS algorithm because it has already been used to determine the reachability of the vertices by the agents. Unlike traditional BFS, which explores all neighboring nodes in an arbitrary or dynamically determined order, the *Biased* algorithm enforces a strict expansion sequence: left, up, right, and down from the goal. This predefined order influences the reconstruction of paths, as the direction from which a node was reached determines the backtracking process.

Algorithm Execution: The algorithm begins by enqueuing the starting position and then iteratively expands the nodes in a breadth-first manner. At each step, the next node to be expanded is selected according to the fixed order of movement. This ordering imposes a directional preference, meaning that when multiple paths are available, the agent will always choose the one that aligns with the predefined expansion sequence. As a result, the paths that are found by the algorithm exhibit a structured bias, reflecting a consistent movement pattern across different environments.

Path Reconstruction and Bias: During path reconstruction, each visited node maintains a reference to the direction from which it was first reached. Since the expansion order is strictly enforced, this reference is inherently biased. If multiple paths exist with equal costs, the algorithm consistently selects the one that aligns with the left-down-right-up preference. Consequently, the paths generated by different agents following the same biased strategy tend to be similar, as the movement preference dictates a common traversal pattern.

Implications of Bias: The biased nature of the algorithm results in predictable and structured paths. While this property may be beneficial in scenarios requiring uniformity, it can also introduce inefficiencies by restricting alternative, potentially less conflicting, paths that do not align with the fixed movement order. Furthermore, when applied in multi-agent pathfinding, agents using the *Biased* algorithm tend to cluster along similar routes, increasing the likelihood of congestion in shared environments, but lowering the number of used vertices, which is beneficial in reduction-based solving of multi-agent pathfinding [22].

Efficiency Considerations: The computational complexity remains identical to standard BFS, as each node is visited once and all neighbors are expanded in constant time. However, the fixed expansion order simplifies implementation and ensures deterministic behavior across different runs. This makes the *Biased* algorithm well suited for applications where predictability and uniform movement patterns are desirable.

In general, the *Biased* algorithm extends BFS by introducing a movement preference, resulting in common path structures shaped by the agent's inherent expansion order. This approach is particularly useful in environments where uniformity and deterministic path generation are preferred, though it may limit flexibility in exploring alternative routes, which leads to a higher number of collisions on paths.

3.1.2 Random

The *Random* algorithm is a variation of the Breadth-First Search (BFS) algorithm, similar to the *Biased* approach. However, instead of following a fixed expansion order for path reconstruction, it introduces randomness in selecting the predecessor node when backtracking to construct the final path. This results in a shortest path that varies between executions (in testing, the seeding of the random generator was used to stabilize the results of the testing between runs), unlike the deterministic behavior of the *Biased* algorithm.

Algorithm Execution: As with standard BFS, the algorithm explores the state space in a breadth-first manner, ensuring that the shortest path to each node is discovered. However, during the path reconstruction phase, instead of choosing the first discovered parent (as in *Biased*), the predecessor is randomly selected from all possible parents that led to the given node with the same shortest cost. This introduces stochasticity into the path selection process.

Path Reconstruction and Randomness: Since multiple shortest paths may exist in a grid-like environment, the random selection of predecessors leads to different path variations in the same situation. The generated paths remain valid from the point of view being shortest, but lack the structured preference seen in the *Biased* algorithm. This randomness ensures that no single traversal pattern dominates, resulting in a diverse set of paths.

Implications of Randomized Path Selection: The introduction of randomness prevents agents from following the same movement pattern repeatedly, reducing congestion in multi-agent scenarios which potentially leads to fever collision between agents, but potentially leads to a greater number of used vertices in final paths combined.

Efficiency Considerations: The computational complexity remains equivalent to BFS, as node expansion and shortest path discovery follow the same principles. The only additional computational overhead occurs during path reconstruction when a random selection is made among valid predecessors. Despite this, the algorithm remains efficient and practical for applications where diverse path generation is preferred.

Overall, the *Random* algorithm extends BFS by introducing stochasticity in path reconstruction, resulting in different shortest paths, which can lead to safer traversal through the environment from a collision point of view. This randomness eliminates agent movement bias, making it useful in applications that require variation in traversal patterns.

Biased and Random algorithms in use

Both proposed algorithms *Biased* and *Random* are able to find optimal paths for each agent in terms of their length. However, these approaches are greedy in terms of finding the shortest path and also in terms of the order of agents for which they search for a given path. Both the *Biased* and *Random* algorithms do not look at other agents, they completely ignore them, which of course can lead to unwanted conflicts.

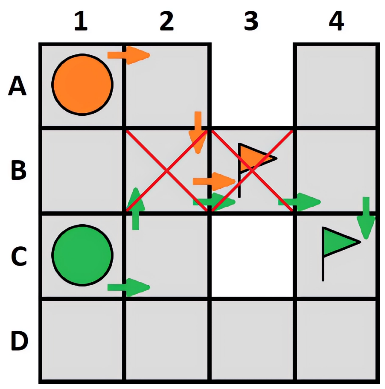

Figure 3.1 Conflict caused by *Biased* and *Random* algorithms.

Let us consider a simple instance, which is shown in Figure 3.1. The instance consists of two agents, with the first agent, orange, having a starting position at ($A1$), and its goal at position at ($B3$). The second, green agent, has a starting position on the ($C1$) position while its goal is located at position ($C4$). The individual paths of the agents from the corresponding start to the goal are shown in the figure in the corresponding color of the individual agents. The paths shown on this figure were obtained by actual run of the individual algorithms with the given instance.

Both algorithms in this case found the same correct path in terms of length, but in terms of conflict, the given paths are incorrect. The *Biased* algorithm first found the correct conflict-free shortest path for the orange agent, but since it does not take other agents into account when finding the path, when finding the path for the green agent it follows purely its biased rules, which end up finding the incorrect shortest path in terms of conflict. The *Random* algorithm has the potential to find a conflict-free path for the green agent, but due to its randomness, in this case it chose the same incorrect shortest path in terms of conflict as the *Biased* algorithm.

In the next two subsections, we will describe two proposed algorithms whose aim is to mitigate this unwanted greediness in the term of finding the shortest path without looking at other agents positions on their paths to at least some degree.

3.1.3 Without Crossing

The *Without Crossing* algorithm is an adaptation of the classical A^* search algorithm that employs the *Without Crossing Ranking* (WCR) heuristic Algorithm 7, to determine the expansion priority of the vertex. The primary objective of this algorithm is to ensure that all the final paths are the shortest possible ones while its secondary objective is to prevent crossing with the other agents paths computed before, thus greedily preventing conflicts.

The algorithm maintains an **open list**, which serves as a priority queue where nodes are sorted according to the WCR heuristic Algorithm 7. This heuristic not only considers the standard cost function $f(v) = G[v] + H[v]$ where $G[v]$ represents the known cost from the start to the vertex v and $H[v]$ is the estimated remaining cost based on the Manhattan distance, but also incorporates vertex occupancy constraints to prioritize nodes that are used less frequently, which is achieved by the tie-breaking condition on the usage of the vertices by the agents computed before.

Algorithm Execution: The algorithm initializes by adding the starting node to the open list with its corresponding cost. At each iteration, the node with the highest priority (determined by WCR) is expanded. The priority queue ensures that nodes with lower heuristic values are processed first, promoting efficient pathfinding. For each expanded node, the algorithm examines its neighboring nodes and updates the cost of reaching those nodes and adds them to the open list if it has not been visited or if a lower-cost path has been found. The search continues until the goal node is reached, at which point the optimal path by the general length and by the algorithm path goal of minimal crossing is reconstructed by tracing back through the predecessors of each expanded node.

Conflict Avoidance: A key feature of this algorithm is its ability to prevent crossing paths by leveraging the WCR heuristic. The heuristic prioritizes nodes that minimize conflicts based on past occupancy data, ensuring that paths are selected in a way that minimizes interference between multiple agents. By doing so, the algorithm avoids situations in which two agents attempt to potentially pass through the same node at the same time. This behavior will be shown and described alongside *Without Crossing at the Same Times* algorithm in detail later with the accompanying visualization in Figure 3.2.

Computational Efficiency: The use of a priority queue ensures that the most promising nodes are expanded first, maintaining the efficiency of A^* search while enhancing its applicability in multi-agent pathfinding scenarios. By integrating WCR into the priority queue sorting, the algorithm ensures better coordination among agents while maintaining time optimality in path selection.

Overall, the *Without Crossing* algorithm extends the classical A^* search by integrating conflict-aware heuristics, making it well suited for navigation tasks in constrained environments where multiple agents operate simultaneously.

3.1.4 Without Crossing at the Same Times

The *Without Crossing at the Same Times* algorithm extends the *Without Crossing* variant of A^* by incorporating temporal constraints to ensure that agents do not occupy the same position at the same timestep. This is achieved by replacing the WCR-based priority queue sorting with the *Without Crossing at the same times Ranking* (XCR) heuristic Algorithm 8, which considers both spatial and temporal conflicts.

Algorithm Execution: The core structure remains the same as the *Without Crossing* algorithm, where nodes are expanded based on a priority queue. However, instead of using the WCR heuristic for sorting, the XCR heuristic is applied, which incorporates an additional time-dependent vertex occupancy function. This structure $V_{\text{occ}}[v][t]$, tracks how often a vertex v is occupied at a specific timestep t, ensuring that agents do not reach the same node simultaneously.

Temporal Conflict Avoidance: When comparing two nodes for expansion, the XCR heuristic determines the priority. If both nodes have the same distance heuristic value, the algorithm further prioritizes nodes that have a lower occupancy count at the given time step. This additional constraint ensures that agents not only avoid spatially crossing paths but also prevent simultaneous occupation of the same vertex. This behavior will be shown and described alongside previous *Without Crossing* algorithm in detail later with the accompanying visualization in Figure 3.2.

Efficiency Considerations: By integrating temporal information into the sorting mechanism, the algorithm achieves path optimality with greedy conflict avoidance. While ensuring safe navigation, it maintains computational efficiency by leveraging the priority queue mechanism of A^* search. The additional time constraint does not significantly increase computational complexity, as it only adds a lookup operation to the vertex occupancy table.

Overall, the *Without Crossing at the Same Times* algorithm enhances the original *Without Crossing* approach by avoiding both spatial and temporal conflicts with agents computed before, making it more suitable for multi-agent pathfinding in time-sensitive environments.

WCR and XCR algorithms in use

The *Without Crossing* and *Without Crossing at the Same Times* algorithms are equally optimal in terms of the length of the computed paths compared to the *Biased* and *Random* algorithms, but they bring an improvement during path computation in terms of agent awareness compared to the *Biased* and *Random* algorithms. This agent awareness in both algorithms consists in the fact that when searching for the shortest path, a tie-break condition is used for path-length equal vertices, specific to each algorithm, which leads to the general selection of a better shortest paths in terms of conflicts.

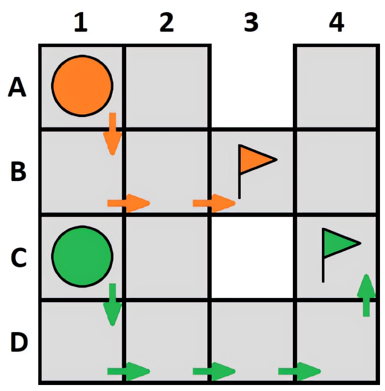

Figure 3.2 Conflict avoidance by WCR and XCR algorithm copared to algorithms *Biased* and *Random* from figure 3.1.

Figure 3.2 shows the performance of the *Without Crossing* and *Without Crossing at the Same Times* algorithms on the same instance where the conflict shortest path selection of the *Biased* and *Random* algorithms were shown. As we can see, unlike the previous algorithms, algorithms based on A^*, which are using the WCR and XCR heuristics, find the optimal solution both in terms of path length and also in terms of conflicts.

In this case, both algorithms were able to find a solution because they searched for the orange agent first and then for the green one. However, both the WCR and the XCR algorithms are greedy in terms of the order in which the shortest paths are searched for the agents. If in this case the path for the green agent was searched first and the path that leads through the vertex $(B3)$ was chosen, this would lead to conflicting pathfinding for both WCR and XCR. In this case, both algorithms would have to bring the orange agent into conflict with the green agent at the vertices $(B2)$ and $(B3)$, which would lead to a collision. This is the same problem as the *Biased* and *Random* algorithms had, as shown in Figure 3.1. So in this case, there would be no improvement. In the next subsection, we will look at how to handle this problem, which is caused by the order in which agents are presented to the pathfinding algorithm.

Another very important shortcoming of the WCR and XCR algorithms is that both approaches still focus only on finding the shortest path with the least number

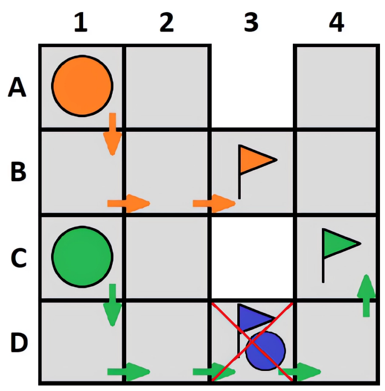

Figure 3.3 Conflict caused by WCR and XCR algorithm when third agent with short start-goal distance is introduced.

of collisions with previous computed agents. The consequence of this problem is simply illustrated in Figure 3.3, which shows what happens when we add another blue agent to both algorithms, which has its starting and goal positions at the same vertex, namely $(D3)$. It does not even matter in what order the individual algorithms would search for paths for this trio of agents because they would always fail on the minimal conflict criterion. If the shortest paths were searched in the order (blue, green, orange), this would lead to a conflict between the green and orange agents at positions $(B2)$ and $(B3)$. If they were searched in the order (green, blue, orange), this would lead either to a conflict at position $(D3)$ between the green and blue agents, or at positions $(B2)$ and $(B3)$ between the green and orange agents, depending on which path the green agent would choose at the beginning. As a last option, there could be a combination (orange, green, blue), which would also lead to a conflict, namely between the green and blue agents at position $(D3)$. In this example, we have shown that there is no correct order for searching paths for a given trio of agents, as long as the given algorithms only search for the shortest paths. To solve this problem, we must find a path longer than its shortest path for some agent, so that there are no conflicts, and at the same time, the makespan condition of the optimal solution is met. In the next subsection, we will show an algorithm that aims to solve both problems of the WCR and XCR algorithms.

3.1.5 Recursive Path Search

The proposed combination of Algorithm 2 and Algorithm 3 efficiently computes the makespan-long paths for a set of agents navigating a shared environment represented as a graph with the goal of being as collision-free as possible. Given a set of agents, each with a specified start and goal position, and a predefined makespan T, the algorithm determines a valid path from the perspective of time constraint for all agents while trying to minimize conflicts with other agents. So for the given environment G and the set of agents A the algorithm outputs a mapping that assigns to each agent $a \in A$ a sequence of positions over time: $\forall a \in A, \forall t \in \{0, \dots, T\}, \text{Paths}[a][t] = v \in V$.

The algorithm first initializes the data structures to track occupancy constraints. Specifically, $V_{\text{occ}}[v][t]$ records the number of times vertex v is occupied at time

Algorithm 2 Find Paths in Makespan for Agents

```
1: Input: $G = (V, E)$ is a valid graph representing the shared environment
2: Input: $A$ is the valid set of agents. Each agent is represented as a pair
   $a = (a_s, a_g)$
3: Input: $T \geq LB$ is the length of each agent final path
4: Return: $\forall a \in A, \forall t \in \{0, \ldots, T\} : Paths[a][t] = v \in V$
5: function FindPaths($G, A, T$)
6:     $\forall v \in V, \forall t \in \{0, \ldots, T\} : V_{\text{occ}}[v][t] \leftarrow 0$
7:     $\forall e \in E, \forall t \in \{0, \ldots, T\} : E_{\text{occ}}[e][t] \leftarrow 0$
8:     $\forall a \in A, \forall v \in V : R[a][v] \leftarrow SP_{\text{length}}(a_s, v)$
9:     $\forall a \in A, \forall t \in \{0, \ldots, T\} : Paths[a][t] \leftarrow null$
10:    for all agents $a \in A$ in descending order sorted by $R[a][g_i]$ do
11:        for $c \leftarrow 0$ to $T$ do
12:            $\forall v \in V, \forall t \in \{0, \ldots, T\} : Visited[v][t] \leftarrow false$
13:            if RPS($R[a]$, $V_{\text{occ}}$, $E_{\text{occ}}$, $Visited$, $a_s$, $a_g$, $T$, $c$, $Paths[a]$) then
14:                break
15:            end if
16:        end for
17:    end for
18:    return Paths
19: end function
```

t, which was already introduced in the XCR algorithm for determining vertex collision, while $E_{\text{occ}}[e][t]$ is newly introduced for this algorithm, which tracks edge usage over time. Edge usage tracking is used to determine the swapping conflict, which previous algorithms were lacking of. The shortest path length from the start position of each agent to every reachable vertex is calculated and stored in $R[a][v]$, which forms the basis for path planning. Initially, all paths are set to *null*, and agents are sorted in descending order according to the shortest path length from their starting position to their goal. This sorting of the agents is newly introduced in this algorithm compared to the previous ones, which were greedy in this sense. Implications of this sorting approach are shown and discussed later in the Figure 3.4.

Each agent is processed sequentially, trying to compute a feasible path using the recursive function $RPS()$. The search begins with a strict constraint of zero allowed conflicts, incrementally relaxing this constraint if no valid path is found. The function $RPS()$ operates recursively to explore feasible paths for an agent from its current position c to its goal g while respecting the remaining time budget T_{left} and the given conflict allowance. If the agent reaches its goal with no remaining time steps, the path is recorded as valid, and the function returns success. Otherwise, the algorithm marks the current position as visited and proceeds with further exploration.

To prioritize efficient pathfinding, two queues are maintained: a priority queue containing preferred moves and a conflict queue storing moves that introduce conflicts. A neighbor n of the current position c is classified in one of these queues based on whether visiting n in $T_{\text{next}} = T_{\text{left}} - 1$ introduces a conflict in the occupancy matrices V_{occ} and E_{occ} determined by the Algorithm 9. The priority

Algorithm 3 Recursive Path Search Algorithm

```
1: Input: $R[v]$ (shortest time in witch each vertex can be visited by an agent)
2: Input: $V_{\text{occ}}[v][t]$ (tracks how many times a vertex is occupied in given time)
3: Input: $E_{\text{occ}}[e][t]$ (tracks how many times an edge is occupied in given time)
4: Input: $Visited[v][t]$ (tracks if the vertex vas already visited in given time)
5: Input: Current position c = $(c_x, c_y)$
6: Input: Goal position g = $(g_x, g_y)$
7: Input: $T_{\text{left}}$ number of time steps left to stand on goal position
8: Input: $conflicts$ number of allowed conflicts left on the path
9: Input/Output: $Path[t]$ agent final path (computed during recursion)
10: Return: $TRUE$ if goal $g$ was reached with $T = 0$ steps left else $FALSE$
11: function RPS($R$, $V_{\text{occ}}$, $E_{\text{occ}}$, $Visited$, $c$, $g$, $T_{\text{left}}$, $conflicts$, $Path$)
12:     if $c = g$ and $T_{\text{left}} = 0$ then
13:         $Path[0] \leftarrow g$
14:         $V_{\text{occ}}[g][0] \leftarrow 1$
15:         return true
16:     end if
17:     $Visited[c][T_{\text{left}}] \leftarrow true$
18:     if $T_{\text{left}} = 0$ then return false
19:     Initialize $Q_{\text{priority}}$ and $Q_{\text{conflict}}$
20:     $T_{\text{next}} \leftarrow T_{\text{left}} - 1$
21:     for all neighbor $n$ of $c$ do
22:         if not $Visited[n][T_{\text{next}}]$ and $|R[g] - R[n]| \leq T_{\text{next}}$ then
23:             if IsConflict($c, n, T_{\text{next}}, V_{\text{occ}}, E_{\text{occ}}$) then Append $n$ to $Q_{\text{conflict}}$
24:             else Append $n$ to $Q_{\text{priority}}$
25:         end if
26:     end for
27:     Sort $Q_{\text{priority}}$ using Manhattan($a, b, g$)
28:     Sort $Q_{\text{conflict}}$ using ConflictsManhattan($a, b, g, T_{\text{next}}, V_{\text{occ}}$)
29:     for all $n$ in $Q_{\text{priority}}$ do
30:         if RPS($R[a]$, $V_{\text{occ}}$, $E_{\text{occ}}$, $Visited$, $n$, $g$, $T_{\text{next}}$, $conflicts$, $Path$) then
31:             $Path[T_{\text{left}}] \leftarrow c$
32:             $V_{\text{occ}}[c][T_{\text{left}}] \leftarrow V_{\text{occ}}[c][T_{\text{left}}] + 1$
33:             $E_{\text{occ}}[(c, n)][T_{\text{left}}] \leftarrow E_{\text{occ}}[(c, n)][T_{\text{left}}] + 1$
34:             return true
35:         end if
36:     end for
37:     if $conflicts = 0$ then return false
38:     for all $n$ in $Q_{\text{conflict}}$ do
39:         if RPS($R[a]$, $V_{\text{occ}}$, $E_{\text{occ}}$, $Visited$, $n$, $g$, $T_{\text{next}}$, $conflicts - 1$, $Path$) then
40:             $Path[T_{\text{left}}] \leftarrow c$
41:             $V_{\text{occ}}[c][T_{\text{left}}] \leftarrow V_{\text{occ}}[c][T_{\text{left}}] + 1$
42:             $E_{\text{occ}}[(c, n)][T_{\text{left}}] \leftarrow E_{\text{occ}}[(c, n)][T_{\text{left}}] + 1$
43:             return true
44:         end if
45:     end for
46:     return false
47: end function
```

queue is sorted using a Manhattan distance heuristic described in Algorithm 5, while the conflict queue is sorted based on a conflict-aware heuristic described in Algorithm 6.

The function then recursively explores the priority neighbors first, as they provide the most optimal paths. If no valid path is found, the algorithm attempts to explore moves from the conflict queue, decreasing the remaining allowed conflicts. If a valid path is discovered, the position and edge usage statistics are updated, and the function returns success. Otherwise, if no valid moves remain, the function terminates with failure.

The complexity of the algorithm depends on the number of agents $|A|$, number of vertices $|V|$ and on the makespan T. In the worst case, the exponential number of nodes can be expanded; however, heuristic-based A^* vertices sorting significantly reduces the branching of the search, improving runtime efficiency. The algorithm ensures that all agents obtain feasible paths while minimizing conflicts and optimizing movement.

Recursive Path Search algorithm in use

The *Recursive Path Search* algorithm is equally optimal in terms of the makespan length of the computed paths compared to the *Without Crossing* and *Without Crossing at the Same Times* algorithms, but brings an improvement in the term of each agent path length. *Recursive Path Search* algorithm allows agents to diverge from their shortest paths as long as they are able to travel to their corresponding goal locations in a given makespan.

It also brings an improvement during path computation in terms of agent swapping awareness compared to the *Without Crossing* and *Without Crossing at the Same Times* algorithms. This agent swapping awareness in algorithm consists in the fact that when searching for the makespan-long path, a tie-break condition is used for path-length equal vertices not just in terms of vertex occupancy but also in swapping occurrence, which are in this case both specified as conflict.

The last but very important improvement compared to the previous algorithms comes in the form of agent sorting. While all previous algorithms were greedy in terms of agent path search execution, this algorithm is making use of the information about the reachability of the vertices by the agents from which it can determine the length of the shortest path for the each agent. With information about the length of the shortest paths, it sort in the descending order the agents by their corresponding shortest path length before the computation of the single makespan-long paths for each of them begins. From this approach, we can see that the agent with the longest shortest path is going to be computed first because he has no time to diverge from its shortest path, while the agent with the shortest path has the most time to diverge from it, to make the space for the more priority agent to pass first.

Combining these three key improvements together leads to a better general selection of the makespan-long paths in terms of conflicts, while making use of the spare time of the agents whose shortest path length is smaller than makespan. This improvement can be seen in the *Recursive Path Search* algorithm solution to the example instance introduced in Figure 3.3 where the *Without Crossing* and *Without Crossing at the Same Times* algorithms failed to find a suitable one.

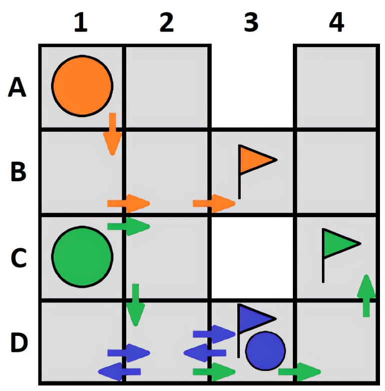

Figure 3.4 Conflict avoidance by RPS algorithm copared to algorithms WCR and XCR from figure 3.3.

In Figure 3.4 we can see the final paths that the *Recursive Path Search* algorithm efficiently found for this given instance. On them we can see that the green agent was computed first, because he has the longest shortest path. The path for the orange agent was then computed, while the path for the blue agent was computed at the end. At the beginning the green agent had two valid options to choose from, ether would be sufficient. He chose to go on the bottom route, through the blue agent start and goal position, which he did not have information about. Then the orange agent can be easily computed. The blue agent is computed lastly with enough time to step aside and make a clean route for the green agent, because he knows about his presence, because the green agent was computed before. If the green agent at the beginning chose the top path through orange goal position, the instance would also be solved because the orange agent would be able to wait and let pass the green agent through its goal and then make a move toward it, while the blue agent at the end would just be standing at his finish.

3.2 Vertex A* Sorting Heuristics

Vertex A^* sorting heuristics play a crucial role in determining the order in which neighboring positions are explored during the pathfinding process. By prioritizing certain moves over others, the search algorithm can be significantly optimized, reducing unnecessary backtracking and improving overall efficiency. Two primary heuristics are employed in the *Recursive Path Search* algorithm, which are sorted by the Manhattan distance comparison Algorithm 5 and the Manhattan distance with conflict comparison Algorithm 6, while the others are used in the basic A^* based searches such as *Without Crossing* and *Without Crossing at the Same Times*. These heuristics ensure that the agent moves towards its goal in an optimal manner while also considering potential conflicts with other agents.

Manhattan distance

The Manhattan distance heuristic is a fundamental metric used to evaluate the priority of two candidate positions relative to the agent's goal. Given two positions $a = (a_x, a_y)$ and $b = (b_x, b_y)$, the heuristic computes the Manhattan distance from each position to the goal $g = (g_x, g_y)$:

$$h(a) = |a_x - g_x| + |a_y - g_y| \tag{3.1}$$

$$h(b) = |b_x - g_x| + |b_y - g_y| \tag{3.2}$$

Algorithm 4 Manhattan distance Computation

```
1: Input: Positions v = $(v_x, v_y)$
2: Input: Goal position g = $(g_x, g_y)$
3: Return: Manhattan distance $h(v)$ between $v$ position and goal $g$
4: function ManhattanDistance($v, g$)
5:     $h_\text{v} \leftarrow |v_x - v_x| + |a_y - g_y|$
6:     return $h_\text{v}$
7: end function
```

A position a is given higher priority over b if its computed heuristic value is lower, i.e.,

$$h(a) < h(b). \tag{3.3}$$

This sorting mechanism ensures that the search first explores paths that move an agent closer to its goal, minimizing detours. The Manhattan heuristic is particularly effective in grid-based environments where movement is constrained to axis-aligned directions. It is utilized in the recursive function $RPS()$ to order the priority queue Q_priority, thereby guiding the search towards the goal in an optimal manner.

Algorithm 5 Manhattan distance Comparison

```
1: Input: Positions a = $(a_x, a_y)$
2: Input: Positions b = $(b_x, b_y)$
3: Input: Goal position g = $(g_x, g_y)$
4: Return: $TRUE$ if $a$ has higher priority than $b$ else $FALSE$
5: function Manhattan($a, b, g$)
6:     $h_\text{a} \leftarrow$ ManhattanDistance($a, g$)
7:     $h_\text{b} \leftarrow$ ManhattanDistance($b, g$)
8:     if $h_\text{a} < h_\text{b}$ then return true
9:     return false
10: end function
```

Manhattan distance with conflicts

While the Manhattan distance heuristic effectively prioritizes movement towards the goal, it does not account for conflicts that may arise due to multiple agents competing for the same space. To mitigate this, an extended heuristic is introduced that incorporates vertex occupancy information.

Given two positions a and b, if the Manhattan heuristic ranks them equally, a secondary comparison is made based on the occupancy count stored in $V_\text{occ}[v][t]$.

This value tracks how many times a vertex has been occupied at a given time step t. The heuristic is computed as follows:

$$a_{\text{used}} = V_{\text{occ}}[a][t], \quad b_{\text{used}} = V_{\text{occ}}[b][t]. \tag{3.4}$$

If $a_{\text{used}} < b_{\text{used}}$, position a is prioritized over b, as it has historically encountered fewer conflicts. This strategy ensures that the search favors paths that minimize collisions while maintaining proximity to the goal.

This enhanced heuristic is crucial in scenarios where multiple agents compete for limited space, as it balances goal-oriented movement with conflict resolution. The function $ConflictsManhattan()$ applies this logic within $RPS()$ by sorting the conflict queue Q_{conflict} accordingly. By integrating conflict awareness into the heuristic, the algorithm achieves better coordination among agents while preserving computational efficiency.

Overall, the combination of Manhattan distance and conflict-aware heuristics provides an effective sorting mechanism that enhances pathfinding in multi-agent environments. By prioritizing movement towards the goal while accounting for occupancy constraints, the algorithm significantly improves path efficiency with regards to conflicts.

Algorithm 6 Manhattan distance with vertex conflicts Comparison

```
1: Input: Position a = $(a_x, a_y)$
2: Input: Position b = $(b_x, b_y)$
3: Input: Goal position g = $(g_x, g_y)$
4: Input: Time $t$
5: Input: $V_{\text{occ}}[v][t]$ (tracks how many times a vertex is occupied in given time)
6: Return: $TRUE$ if $a$ has higher priority than $b$ else $FALSE$
7: function ConflictsManhattan($a, b, g, t, V_{\text{occ}}$)
8:     if Manhattan($a, b, g$) then return true
9:     if Manhattan($b, a, g$) then return false
10:    $a_{\text{used}} \leftarrow V_{\text{occ}}[a][t]$
11:    $b_{\text{used}} \leftarrow V_{\text{occ}}[b][t]$
12:    if $a_{\text{used}} < b_{\text{used}}$ then return true
13:    return false
14: end function
```

Without Crossing Ranking A^* heuristic (WCR)

The A^* heuristic *Without Crossing Ranking* (WCR) is a sorting mechanism designed to prioritize nodes during pathfinding while preventing unnecessary crossings or conflicts. This heuristic extends the traditional A^* search algorithm by incorporating occupancy constraints, ensuring that paths are chosen not only based on cost, but also on the history of vertex usage.

Given two positions a and b, together with a goal position g, the heuristic evaluates which of the two should be prioritized based on two main factors: the estimated total cost to the goal and the frequency of vertex occupation. The first step involves computing the heuristic cost for each position. The cost function is defined as:

$$f_v = G[v] + \text{ManhattanDistance}(v, g) \tag{3.5}$$

where $G[v]$ represents the accumulated cost from the starting position to the vertex v, and the Manhattan distance provides an admissible estimate of the remaining cost to reach the goal. This cost function follows the principle of A^* by combining actual and estimated costs.

The comparison proceeds as follows:

- If $f_a < f_b$, then the position a is considered to have a higher priority, and the function returns **true**.
- If $f_a > f_b$, then b is preferred and the function returns **false**.
- If both positions have the same estimated cost, the heuristic introduces an additional tie-breaking criterion based on occupancy.

To prevent excessive congestion at frequently used nodes, the function examines the recorded vertex occupancy values $V_{\text{occ}}[v]$, which track how often each position has been occupied. If position a has been used less frequently than position b, then a is prioritized, returning **true**. Otherwise, b retains priority.

Algorithm 7 Without Crossing Ranking A^* heuristic

```
 1: Input: Position a = $(a_x, a_y)$
 2: Input: Position b = $(b_x, b_y)$
 3: Input: Goal position g = $(g_x, g_y)$
 4: Input: $G[v]$ (tracks cost of the path from start to a vertex)
 5: Input: $V_{\text{occ}}[v]$ (tracks how many times a vertex is occupied)
 6: Return: $TRUE$ if $a$ has higher priority than $b$ else $FALSE$
 7: function WCR($a, b, g, G, V_{\text{occ}}$)
 8:     $f_{\text{a}} \leftarrow G[v]+$ ManhattanDistance($a, g$)
 9:     $f_{\text{b}} \leftarrow G[v]+$ ManhattanDistance($b, g$)
10:     if $f_{\text{a}} < f_{\text{b}}$ then return true
11:     if $f_{\text{a}} > f_{\text{b}}$ then return false
12:     $a_{\text{used}} \leftarrow V_{\text{occ}}[a]$
13:     $b_{\text{used}} \leftarrow V_{\text{occ}}[b]$
14:     if $a_{\text{used}} < b_{\text{used}}$ then return true
15:     return false
16: end function
```

This approach improves path selection by discouraging the use of congested or highly visited areas while maintaining optimal path efficiency. By integrating both cost estimation and historical usage data, the heuristic ensures a more balanced and conflict-free navigation strategy, reducing unnecessary crossings and improving overall pathfinding performance.

Without Crossing at the same times Ranking A^* heuristic (XCR)

The A^* heuristic *Without Crossing at the same times Ranking* (XCR) is an extension of the A^* *Without Crossing Ranking* heuristic (WCR). This variant refines the sorting mechanism by ensuring that two agents do not cross paths at the same timestep. Although WCR already incorporates occupancy constraints, XCR introduces a time-sensitive component to prevent simultaneous conflicts.

Algorithm 8 Without Crossing at the same times Ranking A^* heuristic

```
1: Input: Position a = $(a_x, a_y)$
2: Input: Position b = $(b_x, b_y)$
3: Input: Goal position g = $(g_x, g_y)$
4: Input: $G[v]$ (tracks cost of the path from start to a vertex)
5: Input: $V_{\text{occ}}[v][t]$ (tracks how many times a vertex is occupied in given time)
6: Return: $TRUE$ if $a$ has higher priority than $b$ else $FALSE$
7: function XCR($a, b, g, t, G, V_{\text{occ}}$)
8:     return WCR($a, b, g, G, V_{\text{occ}}[t]$)
9: end function
```

Similarly to WCR, the function prioritizes positions a and b based on the estimated total cost to the goal. The heuristic cost function remains the same as in the WCR where $G[v]$ denotes the accumulated cost from the starting position to the vertex v, and the Manhattan distance estimates the remaining distance to the goal. The fundamental logic of WCR is preserved, with the algorithm prioritizing the position that minimizes this heuristic cost.

However, XCR enhances the decision-making process by introducing a time-dependent vertex occupancy check. Instead of using a static $V_{\text{occ}}[v]$ to track vertex usage, XCR evaluates $V_{\text{occ}}[v][t]$, which records how frequently a vertex is occupied at a specific timestep t. This ensures that path assignments avoid simultaneous occupation of the same vertex.

The XCR function calls the WCR directly with the modified occupancy data at time t:

$$\text{XCR}(a, b, g, t, G, V_{\text{occ}}) = \text{WCR}(a, b, g, G, V_{\text{occ}}[t]) \tag{3.6}$$

By extending WCR with a time-dependent constraint, XCR prevents agents from occupying the same position in the same timestep, reducing potential conflicts in dynamic environments. This modification makes the heuristic more robust in multi-agent pathfinding scenarios, improving spatial coordination while maintaining optimal path selection.

3.3 Swapping condition

In multi-agent pathfinding, conflicts can arise not only from two agents attempting to occupy the same position, but also from agents attempting to swap positions simultaneously. To handle these cases, a swapping condition is introduced to determine whether the transition from a current position c to a next position n at a given time step t results in a conflict. Two primary types of conflict are considered when assessing a valid movement:

- **Vertex Conflict:** A conflict occurs when an agent attempts to move to a vertex that is already occupied at time t, i.e., if the vertex occupancy tracker $V_{\text{occ}}[n][t]$ is greater than zero.

- **Edge Conflict:** A conflict occurs when two agents attempt to swap positions at the same time, i.e., one moves from c to n while another moves from n to c. This is detected by checking the edge occupancy tracker $E_{\text{occ}}[(c,n)][t]$, which keeps track of the transitions between the connected nodes.

The function $IsConflict()$ evaluates both of these conditions simultaneously. If a vertex or an edge conflict is detected, the function returns **true**, indicating that the transition is invalid. The condition is formalized as follows:

$$\text{IsConflict}(c,n,t) = \begin{cases} \text{true}, & \text{if } V_{\text{occ}}[n][t] > 0 \text{ or } E_{\text{occ}}[(c,n)][t] > 0, \\ \text{false}, & \text{otherwise.} \end{cases} \tag{3.7}$$

The swapping condition plays a critical role in ensuring a collision-free movement in a constrained environment. It is used within the main pathfinding logic to prevent agents from making conflicting moves, ensuring smooth navigation. Specifically, this function is applied to the conflict resolution mechanism within $RPS()$, where an agent's movement is evaluated before execution. If $IsConflict()$ returns **true**, alternative paths or waiting strategies are considered to dynamically resolve the conflict.

Algorithm 9 Checks for edge or vertex conflict between two vertices in given time

```
1: Input: Current position c = $(c_x, c_y)$
2: Input: Next position n = $(n_x, n_y)$
3: Input: Time of potential conflict $t$
4: Input: $V_{\text{occ}}[v][t]$ (tracks how many times a vertex is occupied in given time)
5: Input: $E_{\text{occ}}[e][t]$ (tracks how many times an edge is occupied in given time)
6: Return: $TRUE$ if transition from $c$ to $n$ in given timestep $t$ results in conflict
   else $FALSE$
7: function IsConflict($c, n, t, V_{\text{occ}}, E_{\text{occ}}$)
8:     if $V_{\text{occ}}[n][t] > 0$ or $E_{\text{occ}}[(c,n)][t] > 0$ then return true
9:     return false
10: end function
```

By incorporating both vertex and edge conflicts, the algorithm effectively prevents agents from occupying the same space or swapping positions simultaneously. This approach enhances the efficiency and reliability of multi-agent coordination, reducing unnecessary backtracking and ensuring smoother navigation in complex environments.

3.4 Determining Solution from Shortest Paths Strategies

After computing the shortest paths for all agents using one of the defined search strategies, we can verify whether these paths are valid. This validation process ensures that all conditions described in Equations (2.1)–(2.7) are satisfied, preventing conflicts such as agents occupying the same position simultaneously or swapping positions through the same edge.

The validation procedure begins with the fundamental assumption that the pathfinding process inherently satisfies the conditions that ensure that each agent starts at its designated starting position (Equation (2.1)) and reaches its goal at the final timestep (Equation (2.2)). Similarly, the constraint ensuring that each agent occupies only one position at any given time (Equation (2.3)) is inherently guaranteed by the pathfinder, as it generates individual paths without self-overlaps.

The first explicit check ensures that no two agents occupy the same vertex in the same timestep, as required by Equation (2.4). This is verified by iterating over all agents at every timestep and comparing their positions. If two agents are found at the same vertex simultaneously, the solution is deemed invalid. This check is necessary because individual shortest paths are computed independently in some instances, so then the potential conflicts between agents must be resolved by the next stage.

Next, the validation process ensures that agents move only through valid edges and that transitions between vertices comply with Equation (2.5) and Equation (2.6). These conditions are implicitly satisfied by the pathfinder, as agents can only traverse edges that are part of the graph structure and must always arrive at their next position at the expected timestep.

Finally, an important constraint to check is that no two agents will change their corresponding positions using the same edge in the same timestep, as required by Equation (2.7). This is verified by iterating through all agents at every timestep and checking whether two agents have exchanged their positions between two consecutive timesteps. If such a swap is detected, the solution is invalid as it violates the restriction that an edge can only be occupied by a single directed movement per timestep.

If all conditions are satisfied, the solution is valid, which means that all agents can follow their computed paths without conflicts. This verification ensures that the multi-agent system can execute the computed paths in a collision-free manner while maintaining adherence to the underlying graph structure and movement constraints.

4 Experimental Evaluation

In this chapter, we will look at a comparison of the individual reduction-based graph prunning strategies Makespan-add [22], Prune-and-cut [22], and Combined [22], all combined with the proposed pathfinding heuristics which were described in the previous chapter while focusing on their overall speedup in finding a solution to an MAPF instance compared to the basic reduction-based Baseline approach.

4.1 Instances

For testing of the proposed algorithms, grid-based maps representing different types of environment were used. For each type of map, the sizes of 32×32, 64×64, and finally 128×128 were included to progressively increase computational complexity. In addition, maps of large cities were included in the tests with their size 256×256 to truly test the capabilities of the pathfinding heuristics. In the end, the large warehouse map 164×340 was also added to the test portfolio to test the capabilities of algorithms on the probable map type for this type of problem solving. Agents were placed incrementally on these maps, increasing in number each time the algorithm successfully solved a given instance. Each computation started with 5 agents, and if the algorithm was able to solve the instance within a time limit of 30 seconds, an additional 5 agents were introduced to the total limit of 100 agents when the computation was restarted. Conversely, if the algorithm failed to solve a given instance within the specified time limit, it was assumed that it would be incapable of solving an instance with a greater number of agents and thus the experiment was terminated.

The first type of map tested was *empty*, specifically empty-32-32, empty-64-64, and empty-128-128. Instances for the empty-32-32 map were obtained from movingai.com [8]. The instances for the remaining maps, empty-64-64 and empty-128-128, were generated by randomly distributing the start and goal positions of the agents.

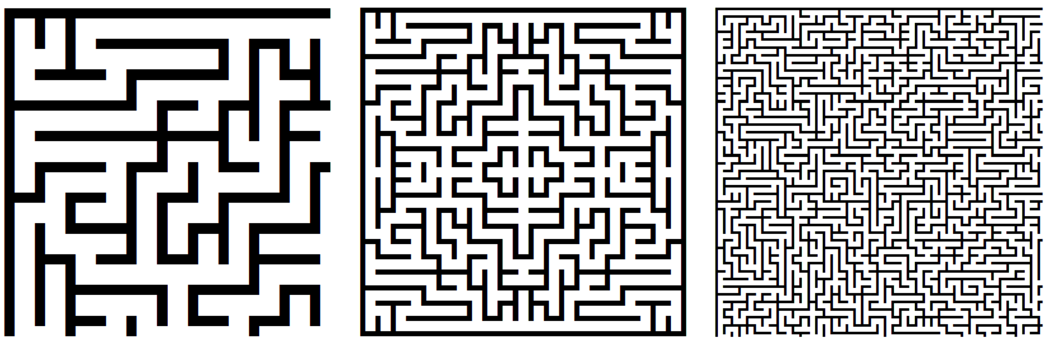

Figure 4.1 Layouts of the maze-type maps. From left to right: maze-32-32-2, maze-64-64-2, and maze-128-128-2.

The next type of maps used in our experiments was *maze*. Their respective visualizations are shown in Figure 4.1. The maps maze-32-32 and maze-128-128, along with their respective instances, were obtained from movingai.com [8]. The maze-64-64 map was created by mirroring the maze-32-32 map, first along the horizontal axis and then along the vertical axis. The individual instances for this map were subsequently generated by randomly distributing the start and goal positions of the agents.

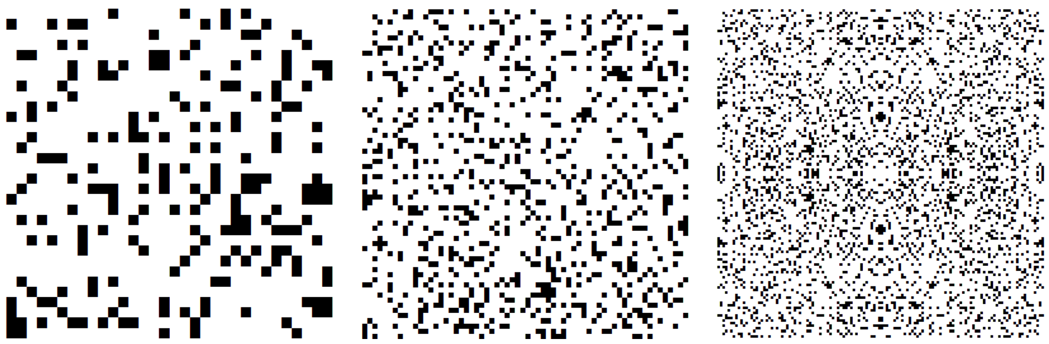
Figure 4.2 Layouts of the random-type maps. From left to right: random-32-32-20, random-64-64-20, and random-128-128-20.

The next category included *random* maps, illustrated in Figure 4.2. The random-32-32-20 and random-64-64-20 maps, along with their instances, were obtained from movingai.com [8]. The random-128-128-20 map was created similarly to the previous case, by mirroring the random-64-64-20 map, first along the horizontal axis and then along the vertical axis. The individual instances for this map were subsequently generated by randomly distributing the start and goal positions of the agents.

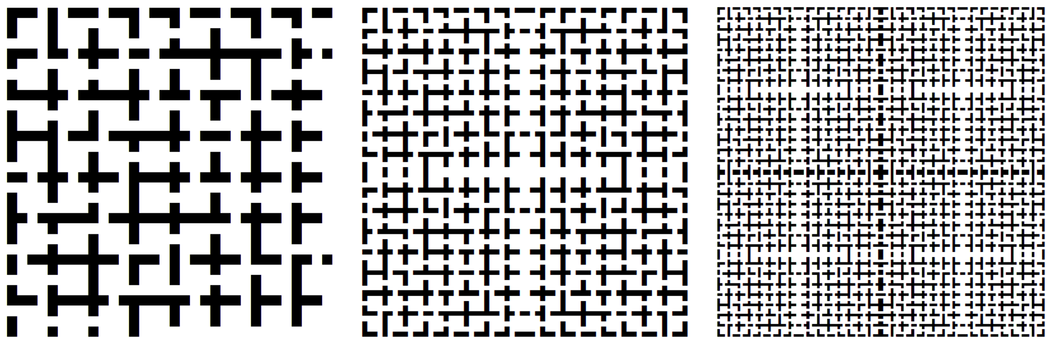
Figure 4.3 Layouts of the room-type maps. From left to right: room-32-32-4, room-64-64-4, and room-128-128-4.

The most challenging basic environment for the agents was the *room* map type, shown in Figure 4.3. This time, only the room-32-32-4 map, along with its instances, was obtained from movingai.com [8]. The remaining two maps, room-64-64-4 and room-128-128-4, were generated by mirroring the room-32-32-4 and room-64-64-4 maps, respectively. The instances for both mirrored maps were then generated by randomly distributing the start and goal positions of the agents.

The environments most challenging due to their large size for agents were the types of map of the *city* family, shown in Figure 4.4. All three maps along with their instances were obtained from movingai.com [8]. In the testing was also added the largest *warehouse* type map, shown in Figure 4.5 from movingai.com [8] with the corresponding instances.

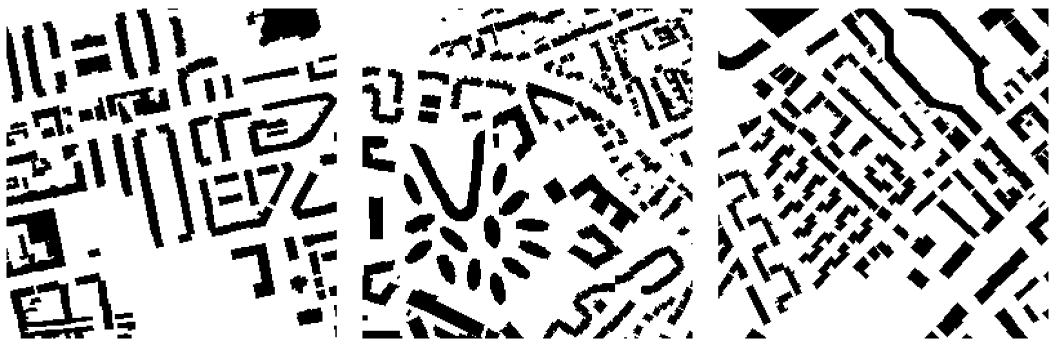
Figure 4.4 Layout of the large maps. From left to right: Berlin_1_256, Paris_1_256, and Boston_0_256.

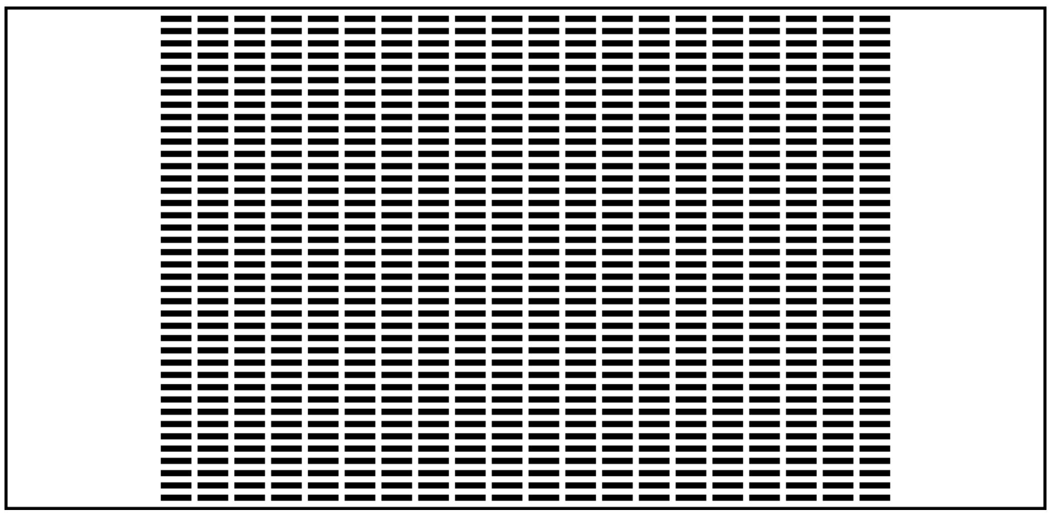
Figure 4.5 Layout of the large Warehouse-20-40-10-2-2 map.

Empirical evaluation

To test and compare the proposed path selection algorithms, we reimplement the subgraph methods and the reduction-based MAPF solver. For the underlying reduction-based MAPF solver, MAPF-encodings [27] project was used, which for SAT solving uses Kissat [28]. For the tasks management and execution the BS-thread-pool [29] library was used. All of the code is implemented in C++ (compiled using g++ with the C++20 standard) and run on a PC with AMD Ryzen™ 9 5900X CPU and a limit of 56 GB of RAM. A description of the test environment, input files, and results can be found in the attachments A.1.

4.2 Results

When comparing the algorithms, we focus on their various properties. Of all the properties, we focus most on the overall calculation speed, and we also look at the number of agents they can solve in the form of success rate. In individual tables and graphs, we refer to the reduction-based graph prunning strategies by using the initials of their names, the so-called B refers to the Baseline algorithm, M refers to the Makespan-add algorithm, P to the Prune-and-cut algorithm and C to the Combined algorithm [22]. We split the results into two groups to separate optimal and sub-optimal algorithms while comparing their results coherently. Next, we will be interested in the success of individual pathfinding algorithms in terms of their ability to solve a given instance without the need to use a reduction-based graph pruning strategy. We will refer to this metric as Solved in preprocess and it will express the percentage of times a given pathfinding algorithm was able to solve a given instance on its own. To the newly proposed pathfinding heuristics we refer in the individual tables and graphs by they corresponding short name such as Bia for the *Biased* algorithm, Ran for the *Random* algorithm, WCR for the *Without Crossing* algorithm, XCR for the *Without Crossing at the Same Times* algorithm and RPS for the final *Recursive Path Search* algorithm.

4.2.1 Success rate of Sub-optimal algorithms

Sub-optimal algorithms represent an important part of reduction-based approaches in terms of their speed. It turns out that sub-optimal algorithms such as Makespan-add and Combined can often find a solution faster than their optimal counterparts, but at the expense of some suboptimality [22]. Because of this fact, we decided to look at this type of algorithms separately and try out the proposed heuristics on them and observe how they affect their ability to find solutions, both in terms of overall success rate and in terms of their suboptimality. We will soon take a closer look at the results of the proposed heuristics in combination with the Combined algorithm, while we will also take a closer look at their impact in combination with the Makespan-add algorithm.

Combined strategy combined with pathfinding heuristics

The individual results of pathfinding algorithms in combination with the sub-optimal algorithms Combined and Makespan-add can be seen in Tables 4.1 and 4.2 respectively. The tables show two important metrics according to which we evaluate a given combination of algorithms, namely Success Rate and Sub-Optimality which are separately displayed for each map type and their corresponding size. Success Rate represents the proportion of the number of solved instances with all instances that were solved by at least one of the tested approaches. So, the Success Rate 1 means that the given combination of algorithm and pathfinding heuristic found a solution within the time limit for each instance that was presented to it, which we aim for. Sub-Optimality then represents the percentage of cases in which the given approach found a solution that was sub-optimal. In this case, this means for us that the closer this number is to 0 that we aim for, the better the given combination of algorithm and heuristic.

In Table 4.1 we can see the results of the comparison of individual combinations of heuristics with the suboptimal algorithm Combined by the metrics described above. From the table we can see that the Success Rate for maps of size 32 is not completely dominantly biased towards any algorithm, however, the RPS algorithm performs best on average out of all of them what was expected. The highest Success Rate is achieved for maps of type empty and maze, while at the same time it achieved complete success on the random map with all other algorithms. The Success Rate on the room map is the most interesting, because it gives us insight into how individual algorithms work, because, due to the complexity of this map, the reduction-based graph pruning Combined algorithm is essential for finding

Map Size	Type	Metric	B	C				
				Bia	Ran	WCR	XCR	RPS
32	empty	Succes rate	0.63	0.775	0.805	0.7	0.79	**1**
	maze		0.714	0.733	0.77	0.752	0.839	**0.932**
	random		0.985	**1**	**1**	**1**	**1**	**1**
	room		0.875	**0.95**	0.91	0.895	**0.95**	0.935
	empty	Sub-Optimality	-	0.039	**0**	**0**	0.032	**0**
	maze		-	0.627	0.091	0.117	0.12	**0**
	random		-	0.005	**0**	**0**	0.01	**0**
	room		-	0.043	0.049	0.034	0.074	**0.022**
64	empty	Succes rate	0.1	0.41	0.38	0.355	0.375	**1**
	maze		0.134	0.279	0.291	0.279	0.285	**0.994**
	random		0.077	0.332	0.316	0.281	0.321	**1**
	room		0.2	0.455	0.43	0.375	0.435	**1**
	empty	Sub-Optimality	-	**0**	**0**	**0**	**0**	**0**
	maze		-	0.02	**0**	**0**	0.078	**0**
	random		-	0.046	**0**	**0**	**0**	**0**
	room		-	**0**	**0**	**0**	**0**	**0**
128	empty	Succes rate	0	0.175	0.19	0.2	0.145	**1**
	maze		0	0.026	0.037	0.016	0.021	**1**
	random		0.02	0.175	0.16	0.15	0.14	**1**
	room		0.025	0.175	0.19	0.175	0.175	**1**
	empty	Sub-Optimality	-	**0**	**0**	**0**	**0**	**0**
	maze		-	**0**	**0**	**0**	**0**	**0**
	random		-	**0**	**0**	**0**	**0**	**0**
	room		-	**0**	**0**	**0**	**0**	**0**

Table 4.1 Comparison of individual combinations of heuristics with the suboptimal algorithm Combined [22] by the specified metric.

the solution. In this case, the most successful algorithms were also Bia and XCR, followed by the RPS algorithm, and at the end, the Ran algorithm ended up with the worst success rate. The highest success of the Bia and XCR algorithms can be easily explained by their behavior, which in this case is beneficial for the Combined algorithm. The paths found by these two algorithms differ from the others in that they generally use fewer vertices, which is beneficial for reduction-based graph pruning strategies [22]. The paths found by the Bia algorithm are specific in that they all have the same preference, which has the indirect consequence that they often go through the same vertices. The WCR algorithm tries to find paths that cross as little as possible throughout their entire length, which ultimately makes the sum of all used vertices on the paths in the graph larger than for the XCR algorithm, which allows paths to cross, while only trying not to cross them at the same time when several agents would pass through the same vertex at the same time. This results in the XCR algorithm using fewer vertices overall than the WCR algorithm, which ultimately makes it easier for the Combined algorithm to find a solution. The RPS algorithm still finds the best paths in terms of the number of collisions in this case, but at the cost of using more vertices, while still using fewer of them than the Ran algorithm, which is expected and also correct.

Sub-optimality is strongly dominated by the RPS algorithm for all map sizes 32, 64 and 128. From the individual results, we can see that the larger the map, the more important it is which pathfinding algorithm is used. We can imagine this as a weight of the problem. The smaller the map, the more likely it is that a great portion of vertices from it will have to be used to solve a given instance, and therefore it is very important that the reduction-based solver is as optimal as possible, because solving such an instance will require solving a larger number of conflicts that arise between individual agents in a small space. So the more weight from the problem is needed to be carried by the reduction-based solver. On the other hand, for large maps, it is again important that the pathfinding itself avoids as many conflicts as possible, because for them it is much simpler and more efficient on large maps where fewer agents are trying to move. Hence, the more weight from the problem needs to be carried by the pathfinder to lesser the burden from the reduction-based solver. From this we can clearly determine that the most advanced RPS pathfinding algorithm is the most successful of all the presented algorithms as expected because it was able to reduce the suboptimality of the Combined algorithm to zero in almost all cases. It failed to do so only for a room map of size 32, while still achieving the best result.

When we look at the success rate of individual algorithm combinations on maps of size 64 and 128, we can see that the RPS algorithm dominates the other approaches in a striking way. This huge difference between the individual algorithms can easily be explained by the sheer number of instances that the individual algorithm were able to solve. The ability to frequently find a solution using the RPS algorithm alone without the need to call a reduction-based solver increases with increasing map sizes, which we will show in more detail and describe later in Table 4.4. The actual speed of finding a solution using the RPS algorithm will be shown and described in more detail later in Figures 4.6 and 4.7.

Makespan-add strategy combined with pathfinding heuristics

Another sub-optimal reduction-based graph pruning strategy tested combined with the proposed pathfinding algorithms was the Makespan-add strategy, the individual results of which are given in Table 4.2. As with the Combined algorithm, in this case we were most interested in the achieved success rate on individual map types, and we also focused on the achieved sub-optimality of individual strategy combinations. As with the sub-optimal Combined strategy, the Makespan-add pathfinding heuristic RPS achieved by far the best results, either in terms of achieved success rate or in terms of achieved sub-optimality among all pathfinding approaches. As in the previous case, the RPS algorithm lags behind in terms of achieved success rate only just on the room map of size 32. The performance it achieved on this map was not very far from the best algorithm on this map of this size, which was WCR.

All in all, from the achieved results of individual pathfinding heuristics in combination with the sub-optimal reduction-based graph pruning strategies, we can conclude that while each of the proposed strategies has its own specifics, it turns out that the RPS strategy is by far the best from all tested. This strategy achieves much better results than the others, both in terms of success rate and in terms of achieved sub-optimality. We can therefore conclude that the RPS algorithm represents a substantial improvement over the other algorithms, while

Map Size	Type	Metric	B	M				
				Bia	Ran	WCR	XCR	RPS
32	empty	Succes rate	0.63	0.645	0.66	0.625	0.625	**1**
	maze		0.714	0.776	0.77	0.758	0.795	**0.944**
	random		0.985	0.995	0.995	1.	0.995	**1**
	room		0.875	0.93	0.92	**0.935**	0.915	0.925
	empty	Sub-Optimality	-	**0**	**0**	**0**	**0**	**0**
	maze		-	**0**	**0**	**0**	**0**	**0**
	random		-	**0**	**0**	**0**	0.005	**0**
	room		-	0.027	0.044	0.011	0.049	**0.005**
64	empty	Succes rate	0.1	0.24	0.24	0.225	0.23	**1**
	maze		0.134	0.235	0.229	0.223	0.251	**0.994**
	random		0.077	0.199	0.204	0.209	0.214	**1**
	room		0.2	0.32	0.32	0.295	0.3	**1**
	empty	Sub-Optimality	-	**0**	**0**	**0**	**0**	**0**
	maze		-	**0**	**0**	**0**	**0**	**0**
	random		-	**0**	**0**	**0**	**0**	**0**
	room		-	**0**	**0**	**0**	**0**	**0**
128	empty	Succes rate	0	0.155	0.165	0.19	0.135	**1**
	maze		0	0.021	0.032	0.021	0.016	**1**
	random		0.02	0.11	0.105	0.09	0.075	**1**
	room		0.025	0.08	0.11	0.095	0.085	**1**
	empty	Sub-Optimality	-	**0**	**0**	**0**	**0**	**0**
	maze		-	**0**	**0**	**0**	**0**	**0**
	random		-	**0**	**0**	**0**	**0**	**0**
	room		-	**0**	**0**	**0**	**0**	**0**

Table 4.2 Comparison of individual combinations of heuristics with the suboptimal algorithm Makespan-add [22] by the specified metric.

showing its dominance across all environments with different map sizes, while its performance gains rise with growing map sizes.

4.2.2 Success rate of Optimal algorithm

In the previous section, we focused on combinations of pathfinding heuristics and sub-optimal algorithms. Sub-optimal algorithms often have their contribution in the form of higher performance or speed, which can be beneficial in some situations, but at the expense of the optimality of the solution found. When searching for a solution to a MAPF problem, the optimality of the solution found is in the most cases of our most interest, and therefore algorithms that guarantee optimality are often the ones that we are trying to improve. The performance of pathfinding methods on sub-optimal algorithms gives us insight into how individual combinations of pathfinding heuristics will behave with an algorithm that guarantees optimality. The optimal algorithm mentioned in this case is the Prune-and-cut [22] strategy, which, like the sub-optimal Combined and Makespan-add strategies, relies heavily on finding suitable paths, which can then be used to perform graph pruning.

Map Size	Type	Metric	B	P				
				Bia	Ran	WCR	XCR	RPS
32	empty	Succes rate	0.63	0.785	0.795	0.7	0.79	**1**
	maze		0.714	0.714	0.77	0.745	0.783	**0.925**
	random		0.985	0.99	0.985	0.985	0.985	**1**
	room		0.875	**0.89**	0.85	0.835	0.865	0.87
64	empty	Succes rate	0.1	0.41	0.38	0.355	0.375	**1**
	maze		0.134	0.263	0.279	0.268	0.274	**0.983**
	random		0.077	0.332	0.316	0.281	0.332	**1**
	room		0.2	0.455	0.43	0.38	0.435	**1**
128	empty	Succes rate	0	0.175	0.19	0.2	0.145	**1**
	maze		0	0.026	0.037	0.026	0.021	**1**
	random		0.02	0.175	0.16	0.15	0.14	**1**
	room		0.025	0.17	0.19	0.175	0.175	**1**

Table 4.3 Comparison of individual combinations of heuristics with the optimal algorithm Prune-and-cut [22] by the success rate percentage metric.

In Table 4.3 we can see the results of the comparison of individual combinations of heuristics with the optimal algorithm Prune-and-cut using the success rate metric, which was already described above. In the contracts with the algorithms Combined and Makespan-add the table now does not contain the result for the sub-optimality, because the algorithm tested guarantees that the solution find is always optimal. When we compared the success rate of individual algorithm combinations for Combined strategy for maps of size 32, we were able to see that the results were not completely dominated by any algorithm; however, the RPS algorithm performed the best on average of all of them, as expected. In this case, however, the RPS algorithm shows much better results compared to the others. He wins in almost every map category on every map size except for the room one with a size of 32. The Success Rate on this room map is the most interesting because it gives us insight into how individual algorithms work, because, due to the complexity of this map, the reduction-based graph pruning Prune-and-cut algorithm is essential for finding the solution. In this case, the most successful algorithm was Bia, closely followed by the RPS algorithm, and in the end, the WCR algorithm ended up with the worst success rate. Due to the nature in which the Prune-and-cut strategy operates, it shows that Prune-and-cut is more sensitive to the paths provided than the Combined pruning strategy, which can be seen from the better results of the pathfinding algorithm RPS. The RPS algorithm with the Prune-and-cut strategy combination finds in most cases the solution to the given instance, which shows that the RPS algorithm is even more suited for the optimal strategy than for the sub-optimal ones, in the tightly packed instances, while still showing strong dominance on maps with greater number of vertices such as 64 and 128.

Overall, based on the results obtained from individual pathfinding heuristics combined with the optimal reduction-based graph pruning strategy, it can be concluded that although each proposed approach exhibits unique characteristics, the RPS strategy consistently outperforms all others evaluated. This method delivers a significantly superior outcome in terms of success rate and scalability.

Consequently, the RPS algorithm can be regarded as a notable advancement over alternative strategies, demonstrating its superiority across different types of map with varying sizes, with its performance benefits becoming increasingly pronounced as the map size grows.

4.2.3 Solved by pathfinding

In previous sections, we have discussed the combinations of pathfinding algorithms with reduction-based graph pruning strategies, focusing on the improvements that these strategies can provide in combination with them. In this section, on the other hand, we will focus purely on the performance of individual pathfinding algorithms. In subsection 3.4, we described how we can evaluate individual paths found by pathfinding algorithms and determine whether the paths themselves represent solutions to the given instance for which they were searched. With this approach, we can greatly speed up the overall calculation of the solution search, because if a given pathfinding algorithm found paths that meet all the solution conditions, we would not have to use any reduction-based method to calculate it. When testing the full combinations of pathfinding algorithm with reduction-based graph pruning strategy, we took advantage of this fact, and before each call to the reduction-based strategy, we checked whether the paths found by the tested pathfinding algorithm already met the MAPF solution conditions.

Map Size	Type	Metric	P				
			Bia	Ran	WCR	XCR	RPS
32	empty	Solved by pathfinding	0.05	0.065	0.075	0.07	**1**
	maze		0.006	0.012	0.025	0.025	**0.845**
	random		0.04	0.045	0.065	0.035	**0.6**
	room		0.025	0.03	0.035	0.035	**0.55**
64	empty	Solved by pathfinding	0.095	0.105	0.135	0.1	**1**
	maze		0.028	0.034	0.028	0.034	**0.972**
	random		0.031	0.036	0.082	0.046	**1**
	room		0.035	0.06	0.04	0.035	**1**
128	empty	Solved by pathfinding	0.14	0.15	0.18	0.115	**1**
	maze		0.021	0.032	0.021	0.016	**1**
	random		0.095	0.085	0.075	0.05	**1**
	room		0.055	0.095	0.065	0.05	**1**

Table 4.4 Percentage of instances solved by the concrete pathfinding algorithm without the use of reduction-based graph pruning Prune-and-cut [22] strategy.

In Table 4.4 we can see the individual results of the tested pathfinding algorithms indicating their ratio of solved instances without the need to call the reduction-based graph pruning Prune-and-cut strategy. From the results, we can see the absolute dominance of the RPS algorithm across all map types and their individual sizes. The RPS algorithm was able to independently solve 100% of instances on the empty map in the all three sizes, namely 32, 64 and 128. In general, the most difficult instances turned out to be those located on the smallest maps, namely 32, due to the ratio between the free vertices and the number of agents, since we chose 100 agents as the maximum tested for each map. The second most difficult environment of size 32 turned out to be the maze map, where the RPS algorithm was able to independently solve 85% of instances. The next most difficult environment turned out to be the map of type random of size 32, where the RPS algorithm was able to solve 60% of the instances independently. The most difficult environment turned out to be the room map of size 32, where the RPS algorithm solved 55% of the instances independently, which turns out to be a very good result, because the room map is the most difficult and difficult to solve of all the maps, which can also be seen from the previous results of other algorithms tested. The RPS algorithm was able to solve almost all instances independently on maps of size 64, achieving 100% of success on maps of the empty, random, and room types. The only map of size 64 in which the RPS algorithm did not find a solution independently for all instances is the maze map, where it managed to achieve 97% of success. On larger maps of size 128, the RPS algorithm demonstrated its ability to search for solutions on large maps efficiently, achieving a full 100% success rate in all environments tested.

From the results achieved for the RPS algorithm, we can confidently conclude that it has demonstrated an impressive ability to solve the vast majority of presented instances independently, without the need to use a reduction-based solver algorithm, and in a negligibly short time compared to other reduction-based approaches. From this we can generally conclude that the RPS algorithm has proven to be a suitable indicator of difficult instances that cannot be solved by it. All other instances that the RPS algorithm was able to solve independently can be considered simple in the future, because the RPS algorithm can solve them optimally in a few milliseconds, as we will show in the next subsection in Figures 4.6 and 4.7. In future work, we therefore propose to focus only on instances that are difficult in terms of not finding a solution using the RPS algorithm. These instances are worth closer examination, while further optimizations should be primary focused on these RPS-hard instances.

4.2.4 Overall solution finding speedup

In the previous subsection, we focused on the very important ability of the tested pathfinding algorithms to find optimal solutions on given instances without the subsequent need to use a reduction-based solver algorithm. We have shown that the RPS algorithm achieves respectable results in this regard. In this subsection, we will focus on an extremely important aspect of the tested approaches, which is their overall speed of finding the optimal solution. We will therefore look at how individual pathfinding algorithms affect the Prune-and-cut algorithm in terms of the overall computation time, and thus the finding of the optimal solution.

For a more detailed comparison of the running speed of the individual algorithms, we refer to Figures 4.6 and 4.7, each of which shows a different map size, one for normal maps in sizes 32, 64, and 128, while the other is for the large town maps with the large warehouse map. They show us the number of instances solved by a given time-limit. Each color represents one algorithm, and the lower the curve, the better it is for the given algorithm combination.

Normal maps

In Figure 4.6 we can see the individual time complexity of the combinations tested of the optimal Prune-and-cut algorithm with pathfinding algorithms on normal maps. This graph shows the results of empty, maze, random, and room map types in their corresponding sizes 32, 64, and 128.

On the individual curves in the graphs we can clearly see that the Baseline algorithm is the slowest, while its calculation time is clearly growing rapidly, which is based on the results shown in our original work [22], while the speed curve of the other combinations of the algorithms Bia, Ran, WCR and XCR is much shallower. Among these four algorithms, not one is significantly better than the others, which can be seen from the way their individual corresponding curves constantly overlap.

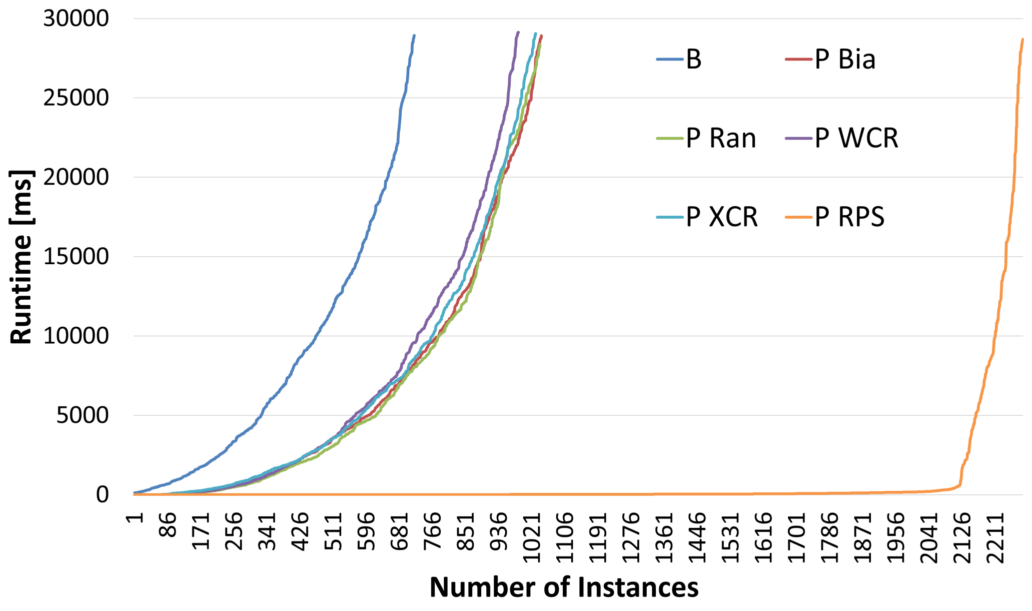

Figure 4.6 Number of instances solved by the tested algorithm combinations by a given time-limit on normal map types.

The most important result was achieved by the RPS algorithm. It was able to solve by far the largest number of instances of all the algorithms, while it was also by far the fastest from all of them. From its corresponding P-RPS curve we can see that the total actual time complexity of the RPS algorithm is only on the order of a few milliseconds on classical maps. The practically constant part of the given curve represents all cases where the solution of the given instance was found by the RPS algorithm without the help of the reduction-based graph pruning Prune-and-cut strategy. From this we can easily see the efficiency of the RPS algorithm itself, proving that it is not only able to find the solution on its own in a large percentage of cases, but also manages to do so in negligible time compared to reduction-based methods, which proves its general usability and time-saving nature.

Large maps

Due to the impressive results achieved by the RPS algorithm, we decided to test it on much larger and more complex maps with a much larger number of agents. We tested individual algorithms on 256×256 city maps, the layout of which is shown in Figure 4.4. We also added a large warehouse map to the test, whose size is 164×360 and its layout is shown in Figure 4.5. Due to the huge size of the given maps, we decided to raise the maximum allowed limit of agents in one instance to 900, which turned out to be correct because the RPS algorithm was comfortably able to solve many more agents on these maps than the original 100.

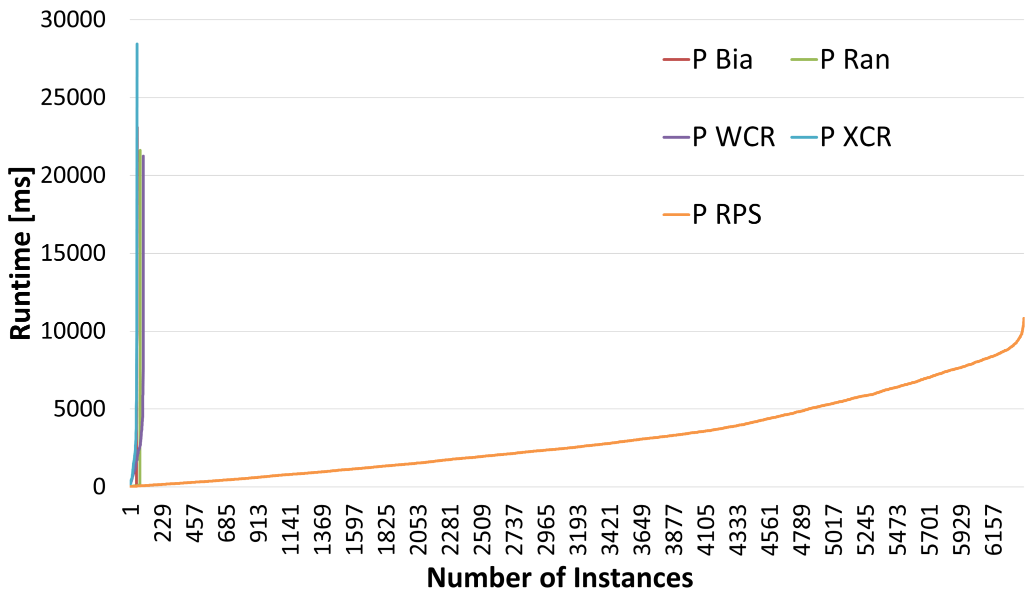

Figure 4.7 Number of instances solved by the tested algorithm combinations by a given time-limit on large map types.

On normal-sized maps, the RPS algorithm achieved by far the best results, while on large maps this advantage was even more pronounced, as can be seen in Figure 4.7. The other algorithms were able to solve only a tiny fraction of the number of instances that the RPS algorithm was able to solve in these test environments. In the vast majority of cases, the RPS algorithm was able to reach the limit of 900 agents on a given map, which demonstrates its ability to cope effectively with large environments. All instances solved by the combination of the RPS and Prune-and-cut algorithms were solved purely by the RPS algorithm because if the given RPS algorithm could not find a solution on its own, then the Prune-and-cut algorithm was unable to solve the given instance, due to its extreme complexity, either in terms of the number of vertices or by the cheer number of agents.

Conclusion

In this work, we focused on accelerating the overall calculation of multi-agent pathfinding (MAPF) using graph-prunning algorithms, focusing on heuristics that were aimed at increasing the performance of these graph-prunning approaches. Our main intention was to show that the calculation method using graph-prunning methods can be significantly accelerated and improved using heuristics primarily focused on selecting ground vertices, around which graph-prunning methods then build their calculation, which is most important on large maps with a large number of vertices, since for small maps it would be difficult to find vertices that we could remove from the graph.

We proposed five progressively more complex approaches for selecting ground vertices based on agent paths, each of which focused on improving the previous one in some way or another. The first approach was the *Biased* algorithm, which looked for paths that had a bias in the form of a preference for the direction of motion, which led to a smaller total number of used vertices at the expense of greater number of conflicts caused. This was followed by the *Random* algorithm, which has the potential to find a conflict-free path for the agent, but due to its randomness there are cases where it chooses the same incorrect shortest path in terms of conflict as the Biased algorithm. Both *Biased* and *Random* approaches are greedy in terms of finding the shortest path as they do not look at other agents, which the next *Without Crossing* (WCR) algorithm aimed to improve. It introduced a spatial tie-breaking condition for vertices selection while finding a path for the agent. This approach was further extended by the *Without Crossing at the Same Times* (XCR) algorithm, which while doing the pathfinding tries to avoid both spatial and temporal conflicts on the vertices with agents computed before.

The most complex approach tested was the *Recursive Path Search* (RPS) algorithm. This algorithm brought up three key improvements compared to previous approaches. The first improvement was in form of the ability to use the whole given makespan time, so the agents were not constrained by their corresponding shortest-path length on their paths. The second improvement was introduced in the form of an extension of the tie-break condition by the detection of agent swapping during pathfinding. The last but very important improvement compared to the previous algorithms comes in the form of agent priority, which determines in which order they will be processed by the pathfinding algorithm, which was achieved by sorting them by their corresponding shortest paths length in descending order.

Finally, in the experimental part, we focused mainly on the achieved success rate of individual algorithms and also on the computational time that the tested algorithms needed to solve the given instances. In both cases, the RPS algorithm achieved by far the best result compared to other approaches and was even able to solve most of the test instances on its own, without calling the reduction-based solver.

In our future work, we would like to explore the potential application of the subgraph methods with the sum-of-costs objective function.

Bibliography

1. SILVER, David. Cooperative Pathfinding. In: YOUNG, R. Michael; LAIRD, John E. (eds.). *Proceedings of the First Artificial Intelligence and Interactive Digital Entertainment Conference, June 1-5, 2005, Marina del Rey, California, USA*. AAAI Press, 2005, pp. 117–122.
2. MORRIS, Robert; PASAREANU, Corina S.; LUCKOW, Kasper Søe; MALIK, Waqar; MA, Hang; KUMAR, T. K. Satish; KOENIG, Sven. Planning, Scheduling and Monitoring for Airport Surface Operations. In: MAGAZZENI, Daniele; SANNER, Scott; THIÉBAUX, Sylvie (eds.). *Planning for Hybrid Systems, Papers from the 2016 AAAI Workshop, Phoenix, Arizona, USA, February 13, 2016*. AAAI Press, 2016, vol. WS-16-12. AAAI Technical Report. Available also from: `http://www.aaai.org/ocs/index.php/WS/AAAIW16/paper/view/12611`.
3. DRESNER, Kurt M.; STONE, Peter. A Multiagent Approach to Autonomous Intersection Management. *J. Artif. Intell. Res.* 2008, vol. 31, pp. 591–656. Available from DOI: `10.1613/JAIR.2502`.
4. WURMAN, Peter R.; D'ANDREA, Raffaello; MOUNTZ, Mick. Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses. *AI Mag.* 2008, vol. 29, no. 1, pp. 9–20. Available from DOI: `10.1609/AIMAG.V29I1.2082`.
5. STERN, Roni; STURTEVANT, Nathan R.; FELNER, Ariel; KOENIG, Sven; MA, Hang; WALKER, Thayne T.; LI, Jiaoyang; ATZMON, Dor; COHEN, Liron; KUMAR, T. K. Satish; BARTÁK, Roman; BOYARSKI, Eli. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In: SURYNEK, Pavel; YEOH, William (eds.). *Proceedings of the Twelfth International Symposium on Combinatorial Search, SOCS 2019, Napa, California, 16-17 July 2019*. AAAI Press, 2019, pp. 151–158. Available from DOI: `10.1609/SOCS.V10I1.18510`.
6. YU, Jingjin; LAVALLE, Steven M. Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs. In: DESJARDINS, Marie; LITTMAN, Michael L. (eds.). *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA*. AAAI Press, 2013. Available also from: `http://www.aaai.org/ocs/index.php/AAAI/AAAI13/paper/view/6111`.
7. RATNER, Daniel; WARMUTH, Manfred K. NxN Puzzle and Related Relocation Problem. *J. Symb. Comput.* 1990, vol. 10, no. 2, pp. 111–138. Available from DOI: `10.1016/S0747-7171(08)80001-6`.
8. STERN, Roni; STURTEVANT, Nathan R.; FELNER, Ariel; KOENIG, Sven; MA, Hang; WALKER, Thayne T.; LI, Jiaoyang; ATZMON, Dor; COHEN, Liron; KUMAR, T. K. Satish; BARTÁK, Roman; BOYARSKI, Eli. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In: SURYNEK, Pavel; YEOH, William (eds.). *Proceedings of the Twelfth International Symposium on Combinatorial Search, SOCS 2019, Napa, California, 16-17 July 2019*. AAAI Press, 2019, pp. 151–159. Available also from: `https://aaai.org/ocs/index.php/SOCS/SOCS19/paper/view/18341`.

9. KORNHAUSER, Daniel; MILLER, Gary L.; SPIRAKIS, Paul G. Coordinating Pebble Motion on Graphs, the Diameter of Permutation Groups, and Applications. In: *25th Annual Symposium on Foundations of Computer Science, West Palm Beach, Florida, USA, 24-26 October 1984*. IEEE Computer Society, 1984, pp. 241–250. Available from DOI: `10.1109/SFCS.1984.715921`.

10. SURYNEK, Pavel. Compact Representations of Cooperative Path-Finding as SAT Based on Matchings in Bipartite Graphs. In: *26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2014, Limassol, Cyprus, November 10-12, 2014*. IEEE Computer Society, 2014, pp. 875–882. Available from DOI: `10.1109/ICTAI.2014.134`.

11. SHARON, Guni; STERN, Roni; GOLDENBERG, Meir; FELNER, Ariel. The Increasing Cost Tree Search for Optimal Multi-Agent Pathfinding. In: WALSH, Toby (ed.). *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*. IJCAI/AAAI, 2011, pp. 662–667. Available from DOI: `10.5591/978-1-57735-516-8/IJCAI11-117`.

12. SURYNEK, Pavel; FELNER, Ariel; STERN, Roni; BOYARSKI, Eli. An Empirical Comparison of the Hardness of Multi-Agent Path Finding under the Makespan and the Sum of Costs Objectives. In: BAIER, Jorge A.; BOTEA, Adi (eds.). *Proceedings of the Ninth Annual Symposium on Combinatorial Search, SOCS 2016, Tarrytown, NY, USA, July 6-8, 2016*. AAAI Press, 2016, pp. 145–147. Available also from: `http://aaai.org/ocs/index.php/SOCS/SOCS16/paper/view/13972`.

13. SURYNEK, Pavel. On the Complexity of Optimal Parallel Cooperative Path-Finding. *Fundam. Informaticae*. 2015, vol. 137, no. 4, pp. 517–548. Available from DOI: `10.3233/FI-2015-1192`.

14. SURYNEK, Pavel. Time-expanded graph-based propositional encodings for makespan-optimal solving of cooperative path finding problems. *Ann. Math. Artif. Intell.* 2017, vol. 81, no. 3-4, pp. 329–375. Available from DOI: `10.1007/s10472-017-9560-z`.

15. SHARON, Guni; STERN, Roni; FELNER, Ariel; STURTEVANT, Nathan R. Conflict-based search for optimal multi-agent pathfinding. *Artif. Intell.* 2015, vol. 219, pp. 40–66. Available from DOI: `10.1016/J.ARTINT.2014.11.006`.

16. BOYARSKI, Eli; FELNER, Ariel; STERN, Roni; SHARON, Guni; TOLPIN, David; BETZALEL, Oded; SHIMONY, Solomon Eyal. ICBS: Improved Conflict-Based Search Algorithm for Multi-Agent Pathfinding. In: YANG, Qiang; WOOLDRIDGE, Michael J. (eds.). *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*. AAAI Press, 2015, pp. 740–746. Available also from: `http://ijcai.org/Abstract/15/110`.

17. GANGE, Graeme; HARABOR, Daniel; STUCKEY, Peter J. Lazy CBS: Implicit Conflict-Based Search Using Lazy Clause Generation. In: BENTON, J.; LIPOVETZKY, Nir; ONAINDIA, Eva; SMITH, David E.; SRIVASTAVA, Siddharth (eds.). *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS 2019, Berkeley, CA, USA, July*

11-15, 2019. AAAI Press, 2019, pp. 155–162. Available also from: https://ojs.aaai.org/index.php/ICAPS/article/view/3471.

18. Barták, Roman; Svancara, Jirí. On SAT-Based Approaches for Multi-Agent Path Finding with the Sum-of-Costs Objective. In: Surynek, Pavel; Yeoh, William (eds.). *Proceedings of the Twelfth International Symposium on Combinatorial Search, SOCS 2019, Napa, California, 16-17 July 2019*. AAAI Press, 2019, pp. 10–17. Available from doi: 10.1609/SOCS.V10I1.18497.

19. As'in Ach'a, Roberto Javier; López, Rodrigo; Hagedorn, Sebastián; Baier, Jorge A. A New Boolean Encoding for MAPF and its Performance with ASP and MaxSAT Solvers. In: Ma, Hang; Serina, Ivan (eds.). *Proceedings of the Fourteenth International Symposium on Combinatorial Search, SOCS 2021, Virtual Conference [Jinan, China], July 26-30, 2021*. AAAI Press, 2021, pp. 11–19. Available from doi: 10.1609/SOCS.V12I1.18546.

20. Surynek, Pavel. Problem Compilation for Multi-Agent Path Finding: a Survey. In: Raedt, Luc De (ed.). *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*. ijcai.org, 2022, pp. 5615–5622. Available from doi: 10.24963/IJCAI.2022/783.

21. Svancara, Jirí; Atzmon, Dor; Strauch, Klaus; Kaminski, Roland; Schaub, Torsten. Which Objective Function is Solved Faster in Multi-Agent Pathfinding? It Depends. In: Rocha, Ana Paula; Steels, Luc; Herik, H. Jaap van den (eds.). *Proceedings of the 16th International Conference on Agents and Artificial Intelligence, ICAART 2024, Volume 3, Rome, Italy, February 24-26, 2024*. SCITEPRESS, 2024, pp. 23–33. Available from doi: 10.5220/0012249400003636.

22. Husár, Matej; Svancara, Jirí; Obermeier, Philipp; Barták, Roman; Schaub, Torsten. Reduction-based Solving of Multi-agent Pathfinding on Large Maps Using Graph Pruning. In: Faliszewski, Piotr; Mascardi, Viviana; Pelachaud, Catherine; Taylor, Matthew E. (eds.). *21st International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2022, Auckland, New Zealand, May 9-13, 2022*. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), 2022, pp. 624–632. Available from doi: 10.5555/3535850.3535921.

23. Kautz, Henry A.; Selman, Bart. Planning as Satisfiability. In: Neumann, Bernd (ed.). *10th European Conference on Artificial Intelligence, ECAI 92, Vienna, Austria, August 3-7, 1992. Proceedings*. John Wiley and Sons, 1992, pp. 359–363.

24. Barták, Roman; Svancara, Jiri; Vlk, Marek. A Scheduling-Based Approach to Multi-Agent Path Finding with Weighted and Capacitated Arcs. In: André, Elisabeth; Koenig, Sven; Dastani, Mehdi; Sukthankar, Gita (eds.). *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2018, Stockholm, Sweden, July 10-15, 2018*. International Foundation for Autonomous Agents and Multiagent Systems Richland, SC, USA / ACM, 2018, pp. 748–756. Available also from: http://dl.acm.org/citation.cfm?id=3237494.

25. BARTÁK, Roman; ZHOU, Neng-Fa; STERN, Roni; BOYARSKI, Eli; SURYNEK, Pavel. Modeling and Solving the Multi-agent Pathfinding Problem in Picat. In: *29th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2017, Boston, MA, USA, November 6-8, 2017*. IEEE Computer Society, 2017, pp. 959–966. Available from DOI: `10.1109/ICTAI.2017.00147`.

26. SVANCARA, Jirí; OBERMEIER, Philipp; HUSÁR, Matej; BARTÁK, Roman; SCHAUB, Torsten. Multi-Agent Pathfinding on Large Maps Using Graph Pruning: This Way or That Way? In: ROCHA, Ana Paula; STEELS, Luc; HERIK, H. Jaap van den (eds.). *Proceedings of the 15th International Conference on Agents and Artificial Intelligence, ICAART 2023, Volume 1, Lisbon, Portugal, February 22-24, 2023*. SCITEPRESS, 2023, pp. 199–206. Available from DOI: `10.5220/0011625100003393`.

27. SVANCARA, Jirí. *MAPF-encodings* [`https://github.com/svancaj/MAPF-encodings`]. GitHub, 2025.

28. BIERE, A.; FAZEKAS, K.; FLEURY, M.; HEISINGER, M. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020. In: BALYO, T.; FROLEYKS, N.; HEULE, M.; ISER, M.; JÄRVISALO, M.; SUDA, M. (eds.). *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*. University of Helsinki, 2020, vol. B-2020-1, pp. 51–53. Department of Computer Science Report Series B.

29. SHOSHANY, Barak. A C++17 Thread Pool for High-Performance Scientific Computing. *SoftwareX*. 2024, vol. 26, p. 101687. Available from DOI: `10.1016/j.softx.2024.101687`.

List of Figures

List of Tables

A Attachments

A.1 Electronic attachments

A.1.1 List of electronic attachments

1. The `src` folder containing the source codes (`.cpp`, `.hpp`).
2. The `Experiments` folder containing the test instances.
   (a) The `Tests` folder containing the test instances of normal maps.
   (b) The `BigMaps` folder containing the large maps test instances.
   (c) The `results_maker.py` file containing the Python script used to generate the results table.
3. The `makefile` file containing the build and compile instructions.
4. The `Tests-Results.csv` file containing the results of the normal instances, from which the tables for this work were made.
5. The file `BigMaps-Results.csv`, which contains the results of the large instances, from which the tables for this work were made.
6. The `build`, `env` folders containing additional supporting files, while file `README.md` contains a brief description and setup process of the project.

A.1.2 Instructions for running the experiments

The project can be compiled using the `make` command.

Search Algorithms

- `b` – Baseline
- `m` – MakespanAdd
- `p` – PruneAndCut
- `c` – Combined

Pathfinding Algorithms

- `b` – Biased
- `r` – TrulyRandom
- `w` – WithoutCrossing
- `x` – WithoutCrossingAtSameTimes
- `R` – RecursivePaths

Running Experiments

Experiments can be executed by running the following command:

```
./build/mapf_experiments -{b|m|p|c} -{b|r|w|x|R} {time limit per
    instance in seconds} {relative path to tests folder}
```

Generating Results

To create a `<Tests folder name>-Results.xlsx` file containing the experiment results, use the Python script located in the `Experiments` folder:

```
python Experiments/results_maker.py --data_path Experiments/<Tests
    folder name>
```

Examples

Example of running experiments:

```
./build/mapf_experiments -mp -wx 30 Experiments/Tests
```

Example of generating the `Tests-Results.xlsx` file:

```
python Experiments/results_maker.py --data_path Experiments/Tests
```

Additional Notes

Please note that experiments involving large maps or a high number of agents may require significant memory resources.

A.1.3 Input files descriptions

```
type octile
height 4
width 4
map
....
@.@.
..@.
@...
```

Example of an input file `random-4-4.map` containing a map of size 4×4 with four obstacles.

```
version 1
0   random-4-4.map   4   4   0   0   2   3   3.41421356
4   random-4-4.map   4   4   0   2   3   0   2.74329455
```

Example of an input file `random-4-4-even-1.scen` containing two agents, where the fifth and sixth columns together represent the starting position coordinates and the seventh and eighth columns together represent the goal coordinates for that agent.

A.1.4 Output files description

```
room-32-32-4.map  room-32-32-4-even-1.scen  205  5   101  OK
room-32-32-4.map  room-32-32-4-even-1.scen  299  10  101  OK
room-32-32-4.map  room-32-32-4-even-1.scen  329  15  101  OK
room-32-32-4.map  room-32-32-4-even-1.scen  347  20  101  OK
room-32-32-4.map  room-32-32-4-even-1.scen  386  25  107  OK
room-32-32-4.map  room-32-32-4-even-1.scen  400  30  107  OK
room-32-32-4.map  room-32-32-4-even-1.scen  418  35  107  OK
room-32-32-4.map  room-32-32-4-even-1.scen  442  40  107  NO solution
room-32-32-4.map  room-32-32-4-even-1.scen  591  40  107  OK
room-32-32-4.map  room-32-32-4-even-1.scen  460  45  108  OK
room-32-32-4.map  room-32-32-4-even-1.scen  466  50  108  Timed out
room-32-32-4.map  room-32-32-4-even-2.scen  666  5   80   OK Preprocess
room-32-32-4.map  room-32-32-4-even-2.scen  246  10  80   NO solution
room-32-32-4.map  room-32-32-4-even-2.scen  396  10  80   OK
                                    ...
```

Example of a log output file `*_log.txt`. The individual columns represent, from left to right, the map name, the file name containing the agents, the number of map vertices used (left uncut from the graph), the number of agents, the tested makespan, and the solving result.

```
room-32-32-4.map room...even-1.scen  205  5   101  101  2  153    156
room-32-32-4.map room...even-1.scen  299  10  101  101  2  368    371
room-32-32-4.map room...even-1.scen  329  15  101  101  3  659    662
room-32-32-4.map room...even-1.scen  347  20  101  101  3  1205   1208
room-32-32-4.map room...even-1.scen  386  25  107  107  3  2556   2559
room-32-32-4.map room...even-1.scen  400  30  107  107  3  4677   4680
room-32-32-4.map room...even-1.scen  418  35  107  107  4  10707  10711
room-32-32-4.map room...even-1.scen  591  40  107  107  3  10390  10395
room-32-32-4.map room...even-1.scen  460  45  108  108  4  10475  10479
room-32-32-4.map room...even-2.scen  172  5   80   80   2  0      4
room-32-32-4.map room...even-2.scen  396  10  80   80   2  1071   1074
room-32-32-4.map room...even-2.scen  468  15  93   93   3  1477   1481
room-32-32-4.map room...even-2.scen  495  20  93   93   3  2452   2456
                                    ...
```

Example of the output file `*_results.txt` which contains the calculated results. The individual columns represent, from left to right, the map name, the file name containing the agents, the number of map vertices used (left uncut from the graph), the number of agents, the lower bound on the makespan, the size of the makespan of the found solution, pre-processing pathfinding heuristics time, SAT solver time, and the total solution time.