# POLITECNICO DI TORINO

**Corso di Laurea Magistrale**
**in Ingegneria Matematica**

Tesi di Laurea Magistrale

# Decentralized Algorithms for Multi-Agent Pathfinding


**Relatori**
prof. Fabio Fagnani
prof. Giacomo Como
prof.ssa Sara Bernardini
*firma dei relatori*

. . . . . .
. . . . . .

**Candidato**
Andrea Bertolini

*firma del candidato*

. . . . . .

Anno Accademico 2021-2022

# Summary

Multi-Agent Pathfinding (MAPF) is the problem of planning the trajectory of many agents, from their start location to their respective destination, while sharing the same environment and avoiding collisions. It is part of the AI field of Planning and unifies concepts of Applied Mathematics, Computer Science, and Robotics.

MAPF started being studied during the last decades of the 20th century, but only after 2000 some practical solutions were presented. Over the last few years, it acquired more interest, especially from the application point of view, thanks to the development of advanced Robotics Systems. Well-known accomplishments include warehouse automation in Industry 4.0. During the last decade, many companies began to use AI-driven mobile robots' motion planning in fulfillment centers. Worldwide companies count up to tens of thousands of these robots for Pick & Place jobs, continuously moving inside warehouses extended for up to 80 000 $m^2$ without generating conflicts between each other. This automation brought not just efficient results from the production point of view, but also economically in terms of hundreds of thousands of new job positions.

Specific planning algorithms are fundamental for fast, ordered, and functional agents' operability. Since planning on previously unknown domains requires some ways to build a navigation map of the environment, algorithms are divided into graph-based and grid-based models. Researchers and companies mainly focus on the latter case, grid-based planning, which means the environment is abstracted as a grid whose nodes may contain at most one agent. Time is discretized, and each robot can make a single action, *move* or *wait*, at each step. For multi-agent systems, the two main pathfinding techniques are *centralized* (*coupled*) or *decentralized* (*decoupled*). Centralized algorithms aim at finding a solution from the point of view of a higher entity. This centralized planner tries to reduce a global system objective function by treating all agents as a whole composite object in the ensemble environment. Oppositely, decentralized methods plan agents' paths independently and then apply specific techniques to solve possible occurring conflicts. Each agent tries to optimize its user objective function.

In this work, we focus on decentralized grid-based solvers. Many ideas can be exploited to develop decentralized methods, such as priorities, abstractions, and hierarchies. Moreover, specific techniques exist that avoid conflicts between agents.

The first objective of this project is to make a thorough review of the literature, focusing on similarities and differences between the literature algorithms.

Secondly, inspired by a specific real-world problem, we develop a simple method and five optimized variants, taking inspiration from the approaches available in the literature or exploiting new ideas. In the first simple algorithm, there is no cooperation, so agents

act *selfishly*, and we apply a basic random-style collision avoidance technique to let them escape imminent conflicts. Optimized variants exploit ideas of computational optimization, intelligent movements, and congestion awareness to obtain better performances.
The specific real-world problem requires that all these developed algorithms work in an *online* setting, dealing with agents coming at any time into the system. Useful techniques are then put together in a final optimized version that we compare with some of the best-working literature's *cooperative* decentralized algorithms. During the experimental analysis, we focus on different variables, mainly number of avoided conflicts and search method function calls, sum of time steps, maximal path length, and time consumption.
We implement these methods and simulate systems using Object Oriented Programming with `Python` programming language. Both real and synthetic benchmark domains are taken directly from the research community website. Data from simulations are visualized and analyzed with `MATLAB` and `R` software. Finally, `PowerPoint` is used to enhance the charts.

Our final goal is to establish connections to well-known Applied Mathematics concepts: *Game Theory* and *Network Traffic Flow*. We consider Game Theory to understand the best choice an agent should make when close to possible conflicts and to be pushed in less congested areas while still looking for its shortest path. Network Traffic Flow helps us to quantify the performance of decentralized algorithms with respect to the centralized counterpart. Indeed, solutions coming from decentralized algorithms for MAPF can be interpreted as *user* optimum traffic assignments, while centralized versions reflect the *system* point of view.

In conclusion, with this work, we deeply examine different techniques to improve the performance of basic decentralized algorithms. After a complete experimental phase, we show decentralized algorithms based on cooperation outperform selfish ones. Finally, we present Game Theory and Network Traffic Flow as ways to improve decentralized algorithms and quantitatively compare them to centralized methods, opening the doors to interesting future connections. To the best of our knowledge, these two concepts have never been considered when developing MAPF solvers.

# Contents

# List of Tables

# List of Figures

# Part I

# Introduction

# Chapter 1

# Introduction to the General Problem

In this chapter, we present the general Multi-Agent Pathfinding framework. In the first section, we make a historical review of the main ideas developed to solve Multi-Agent Pathfinding problems. In the second section, we first focus on the formal definition of classical MAPF, with the main properties of the case. Then, we give an overview of these years' studied variants.

## 1.1 MAPF: Literature Review

### 1.1.1 Historical Introduction

Starting from the mid-20th century, the study of *collective behavior* for multi-agent systems acquired increasing interest, both from the research community and from the practical point of view of companies.
By ***collective behavior***, we mean the agents' behavior when sharing the same environment. By ***cooperation***, instead, we mean the subclass of collective behaviours characterized by the cooperation between agents (Cao et al. [1997]).

The first studies on collaboration go back to the 1940's when researchers started analyzing the behavior of many robots within the same environment. At the end of the 1980's, many new areas are involved in the increasing results of cooperative robotics. Connections between the development of innovative robotics systems, theoretical results from other fields, and other studied problems are fundamental for the overall progress.

### 1.1.2 Grid-Based Multi-Agent Pathfinding

One of the main studied problems in cooperative robotics is the *path planning* and *motion planning* problem, i.e., to plan the motion of many agents in the same environment without collisions between robots.

***Multi-Agent Path Finding (MAPF)*** is a specific multi-agent planning problem. ***The task is to plan paths for a group of agents to a set of target destinations, with the constraint that the agents will be able to follow these paths concurrently without colliding with each other*** (Silver [2005]).

In *classical MAPF*, agents are assumed to occupy exactly one vertex, having no volume, no shape, and move at a constant speed. *By contrast*, motion planning algorithms consider these properties too: at each time step, an agent belongs to a ***configuration*** instead of just a vertex. A **configuration** specifies the agent's location, orientation, velocity, etc., and an edge between two configurations represents kinematic motion. Several notable *MAPF variants* are steps towards closing this gap between classical MAPF and motion planning (Ma and Koenig [2017], Stern et al. [2019]).

MAPF started being studied in the last decades of the 20th century. Only at the beginning of the 21st century some practical solutions were presented. Today, the main fields of interest are the gaming industry, automation for Industry 4.0, traffic management for smart mobility, air traffic control, autonomous driving, underground mining, military field, disaster rescue, and monitoring.

Since planning on previously unknown domains requires some ways to build a *navigation map* of the environment, algorithms are divided into *graph-based models* and *grid-based models* (Zelinsky [1992]). Researchers and companies mainly focus on the latter case, grid-based planning, which means the environment is abstracted as a grid whose nodes may contain at most one agent. This document focuses on grid-based abstraction. Time is *discretized*, and each robot can make a single action, *move* or *wait*, at each step.

### 1.1.3 From Single-Agent to Multi-Agent Solvers

MAPF is a generalization of the *single-agent problem*, also known as the *shortest path search problem*. In the middle of the 20th century, a fast, optimal, and complete [1.2.3] heuristic search method [5], called ***A* Algorithm*** (Hart et al. [1968]), was developed, outperforming the well-known *Dijkstra's Algorithm* (Dijkstra [1959]). During the last decade of the 20th century and the first years of the new century, A*'s time complexity was improved. The choice of the heuristic function [1.1.5] is the key to improving the speed of the search method (Holte et al. [1996]), and for many years the research community followed this intuition to produce better algorithms.

Single-agent techniques can be adapted to the multi-agent case in the following way. We build an environment composite of the single-agent environments and treat the set of agents as an "extended" single entity. Here, we can apply the single-agent methods. Without building an appropriate search space, the traditional formulation of the pathfinding problem as a single-agent search on a bi-dimensional space is not suitable (Erdmann and Lozano-Perez [1987]). Such algorithms take the name of ***centralized algorithms***. In other words, *centralized planning* considers all agents together.

The majority of centralized algorithms are not only optimal but also complete.

Centralized methods do not consist only of search algorithms. Many researchers have tackled multi-agent path planning problems from the point of view of mathematical optimization problems, e.g., exploiting linear and quadratic programs. These specific centralized methods often have good theoretical properties like optimality, but the computational complexity limits their applications to small teams with few obstacles.
In this document, we focus on the algorithmic view of MAPF.

Even though we can extend single-agent methods to multi-agent systems, MAPF is much more challenging than the single-agent case, being a *PSPACE-hard problem* (Hopcroft et al. [1984], Surynek [2010]) [1]. The main drawback of such methods is the time and memory consumption. They are exponential in the dimension of the composite configuration space, i.e., in the number of agents. Moreover, the simple application of single-agent methods like A* to ensemble spaces for the multi-agent case is not enough to find a solution. Indeed, when adapting such methods we must establish a way to avoid conflicts.

To deal with this problem, one could think about finding the optimal path for each agent, then combine these optimal paths while avoiding conflicts. So the idea is to subdivide the main problem into single-agent search problems, and each agent programs its path from the origin to the corresponding destination. Then we combine the single results removing possible conflicts. The associated methods are called ***decentralized algorithms*** (also called ***coupled*** or ***distributed***) [2]. These methods are computationally efficient from the time point of view: the time required for producing the solution scales well with the number of agents. On the other hand, they are sub-optimal (or bounded sub-optimal, in the best cases) and incomplete. So the main characteristic of these alternative methods is to look for a trade-off between completeness and optimality of the solution to improve the performance (Wang et al. [2008]).

To conclude, we know in advance there is no universal winner between centralized and decentralized algorithms. It depends on the specific problem. Interestingly, during the first decade of 2000, there was an explosion in the development of MAPF decoupled algorithms.
In this document, we consider decentralized algorithms, generally preferred in real applications w.r.t centralized ones.

---

[1] One problem is said to belong to ***class P*** if there exists a polynomial complexity algorithm to solve it. Complexity ***class NP*** contains class P and other computational problems. The most likely conjecture is that $P \neq NP$. Today, we have no certainty about this. The problems that could be used as separators between the two classes are the ***NP-complete*** problems, i.e., the "most difficult problems" inside the NP class (Crescenzi et al. [2012]).

[2] The main partition of MAPF solvers is in *centralized* and *decentralized*. *Decentralized approaches* are sometimes referred to as *decoupled* (and *centralized* ones are *coupled*) or *distributed*. Some articles establish a partition in *centralized/decentralized*, *coupled/decoupled*, and *distributed*, e.g., Sharon et al. [2015], Wiktor et al. [2014]. In this document, we follow the binary partitioning of *centralized* (i.e., *coupled*) and *decentralized* (i.e., *decoupled*, *distributed*), such as Silver [2005], Wagner and Choset [2011], Luna and Bekris [2011], Ryan [2008a], van den Berg et al. [2009].

### 1.1.4 MAPF Partition

We can partition MAPF algorithms both from the point of view of the specific technique used as a basis (Felner et al. [2017]) and from the point of view of how agents interact with each other.

#### MAPF partition: basis technique

- *Search-Based Algorithms*: these methods are based on search; leading examples are CA* algorithm and its variants, HCA* and WHCA* (Silver [2005]).
- *Rule-Based Algorithms*: these algorithms include specific movement rules for different scenarios; examples of rule-based algorithms are push and swap, parallel push and swap, and push and rotate (Luna and Bekris [2011] , Sajid et al. [2012] , Wiktor et al. [2014] , De Wilde et al. [2013]).
- *Hybrid Algorithms*: these algorithms contain both movement rules and massive search.

#### MAPF partition: agents' interaction

- *Cooperative Pathfinding*: here, each agent has full knowledge of all other agents and their planned routes; cooperative agents will follow paths that are planned for them or obey constraints on their movements; in particular, agents may be willing to take longer routes to their goals to avoid colliding with other agents (Silver [2005] , Wang et al. [2009]).
- *Non-Cooperative Pathfinding* (also known as *Self-Interested Pathfinding*): agents do not know others' plans; they only try to minimize their costs (e.g., human drivers' behavior). A self-interested agent might follow a path that causes other agents to delay or even not find their destinations. Formally, a self-interested agent in MAPF always chooses to follow the path with the best individual social welfare (Bnaya et al. [2013]).
- *Hybrid Pathfinding* (also known as *Agent-Centered Search*): here, agents do not have access to global information about the world, but just some local window or radius around each agent (Koenig [1996] , Jansen and Sturtevant [2008] , Koenig [2001]).
- *Antagonist Pathfinding*: every agent tries to reach its goal while preventing the others from reaching theirs.

### 1.1.5 MAPF Solvers: Concepts

Many ideas can be exploited to develop decentralized methods. Some fundamental underlying concepts are:

- Priority;

- Configuration Space-Time;
- Abstraction;
- Heuristics;
- Hierarchy.

### Priority

One well-known idea to solve MAPF problems is to assign *priorities* to agents. Planning for multi-agent systems in which we assign priorities is called ***prioritized planning***. In this setting, the problem to avoid collisions between two agents falls to the agent with lower priority (Stern [2019]).

There are many techniques to assign priorities, mainly divided into:

- fixed priority values assignment;
- varying priority values assignment.

Moreover, the priority assignment may be random or based on further knowledge of the problem.

The first developed methods using priority considered a fixed scheme (Warren [1990]). Examples of randomly generated priorities are in Bnaya et al. [2013]. Further works are, e.g., Ryan [2008b] , Ryan [2008a] , Čáp et al. [2012] , Čáp et al. [2015] , Velagapudi et al. [2010].
While prioritized planning is fast in simple grids, it is highly sensitive to the priority scheme. Fixed priority solvers may not solve specific classes of problems because we may choose a path for one agent that impedes finding a solution for another successively processed. This is why dynamically changing priority schemes were introduced.

One of the first to use dynamically changing schemes was Bennewitz et al. [2001], while more recently Wang et al. [2009] , Chen et al. [2009] , Regele and Levi [2006]. Dynamically changing priorities can also be assigned based on heuristics [1.1.5]. An example is Wang et al. [2008], in which cycling conflicts are avoided by letting the agent with a higher *node density heuristic* value pass (i.e., the agent with a higher number of computed paths that pass through the node). Silver [2005] presents some algorithms with fixed priority assignments, then slightly modifies them, obtaining the same effects of dynamically changing priorities.

The computational efficiency and simplicity of prioritized planning algorithms is the main reason for their extensive use.

### Configuration Space-Time

The concept of priority is strictly connected to that of *Configuration Space-Time*. Indeed, to find a plan for the agent with the $i^{th}$ priority may require an agent to wait in its cell. Thus this becomes the shortest path problem in a time-expansion graph (Stern [2019]).

When talking about configuration, we generally refer to motion planning: a *configuration* specifies the agent's location, orientation, velocity, etc., and an edge between two configurations represents kinematic motion. In *classical MAPF*, agents are assumed to occupy one vertex, with no volume, and no shape, and move at a constant speed [1.2.4]. In this specific case, we talk about *space-time reservation table*.
A ***space-time reservation table*** is a three-dimensional table indexed by both $(x, y)$ location in the world and a time $t$. The value of an entry $(x, y, t)$, if it exists, is the unit that will be occupying the given cell then (Sturtevant and Buro [2006]).

The first paper presenting the idea of configuration space-time is Erdmann and Lozano-Perez [1987]. More recently, the first to apply the idea to grid maps was Silver [2005]. Other examples are Regele and Levi [2006] and Sturtevant and Buro [2006]. Ma et al. [2019] explore the space of all possible partial priority orderings as part of a novel systematic and conflict-driven combinatorial search framework to guarantee completeness.

### Abstraction

The use of space-time reservation tables is linked to the concept of *abstraction*. ***Abstraction*** is the process of replacing one initial state space, i.e. the *map*, by another the map, called *search graph* or *abstract space*, that is easier to search (Holte et al. [1996]). So an abstraction can be seen as the process of reducing the resolution of the map while maintaining connectivity information in the abstraction.

Abstraction models can be subdivided into *grid-based models* and *graph models* [1.1.2]. Using a grid map is the most popular approach to abstract a real-world navigation map into a search space. The map is discretized into a grid of atomic locations called ***tiles***. Then we define one node for each accessible tile and undirected edges between adjacent nodes so that it is possible to move from one node to the other.

Space-time reservation tables are one form of state space abstraction. Particular mention deserves *WHCA** (Silver [2005]) that adds an intermediate layer of abstraction equivalent to the base level state space for $w$ steps and the abstract level state space for the remainder of the search. There are many other examples of abstraction. Wang et al. [2008] abstracts the grid map into a *flow-annotated search graph*. Ryan [2008a] exploits specific subgraph structures of the map to abstract the search. Wagner and Choset [2011] , Sharon et al. [2013] , Sharon et al. [2015] , Barer et al. [2014] add one further abstraction by building *search trees*. Henkel and Toussaint [2020] presents a novel method to abstract

a map into a *directed roadmap graph* that allows for collision avoidance in multi-robot navigation.

### Heuristic

In the last decades of the 20th century, many researchers have investigated abstraction as a means of automatically creating *admissible heuristics* for *single-agent heuristic search*. In pathfinding, a ***heuristic function*** is a function telling us how close we are to the goal. A heuristic is ***admissible*** if it never overestimates the cost of getting from a state to the goal in the starting space. Examples of heuristics are the *Manhattan Distance*, the *True Distance Heuristic*, the *node density*, or the *Sum of Individual Costs (SIC)*. By exploiting this function, it is possible to speed up search [5].

The best example of a heuristic search is A*, which requires the heuristic to be admissible. Other *single-agent heuristic search* algorithms are presented in Edelkamp and Eckerle [1997] and Hernández and Meseguer [2005]. MAPF solvers examples are Standley [2010] , Silver [2005]. The MAPF solver *WHCA** employs a heuristic search in a space-time domain based on *HCA** and is limited to a fixed depth. *HCA** uses an algorithm called *RRA** to reuse search data when computing the heuristic distance, reducing the computational cost. Heuristic functions can be adapted for specific purpose too, generally to assign priorities (Silver [2005], Wang et al. [2009]). To solve conflicts, some methods add random noise to the heuristic function (Silver [2005]). Wang et al. [2008] modifies the heuristic to favor straighter paths in a flow-annotated search graph. Other examples of heuristics are connected to the rule-based algorithms (Luna and Bekris [2011] , Sajid et al. [2012] , Wiktor et al. [2014] , De Wilde et al. [2013]).

### Hierarchy

The concept of *hierarchy* is strictly related to abstraction. By ***hierarchy***, we mean the hierarchy of abstractions of the search space to ease the search mainly for large problem spaces. More in detail, hierarchical approaches have two main phases:

1 . building the hierarchical search framework;

2 . use of this framework for pathfinding.

The first step in building the framework for hierarchical search is to define a topological abstraction of the maze. Once the abstract graph has been constructed and the intra-edge distances (i.e., the distance between clusters) computed, the grid is ready to use a hierarchical search. During the second phase, we compute an *abstract path*. Then, for each node in the abstracted graph, we compute the path inside of it. Finally, we can convert the abstract path into a sequence of moves on the original grid. This final part is called *path refinement*. *Path smoothing* can be used to improve the quality of the path-refinement solution.

The hierarchy can be extended to several levels, transforming the abstract graph into a *multi-level graph*. So each level in the hierarchy is a more abstract map version. In a multi-level graph, nodes and edges have labels showing their level in the abstraction hierarchy. We perform pathfinding using a combination of small searches in the graph at various abstraction levels (Botea et al. [2004]).

Up to 2004, the hierarchy was mainly a two-level hierarchy. One well-known example of a two-level hierarchy is that of Sharon et al. [2013]: a specific tree structure abstraction is considered for search, then nodes are checked to be solutions at the lower level. Examples of a two-level hierarchy are Čáp et al. [2012] and its variants presented in Barer et al. [2014]. Holte et al. [1996] noted the choice of hierarchy is critical, and a too large hierarchy could perform worse than simple ones. In Silver [2005], *WHCA** adds one intermediate level of abstraction to *HCA**, by considering a window of size $w$ in which to apply cooperation between agents. Multi-level hierarchical pathfinding are shown in Sturtevant and Buro [2005], Botea et al. [2004], Sturtevant and Buro [2006], Ryan [2008a], Surynek [2009].

## 1.2 MAPF: Formal Definition, Properties, and Variants

Researchers in Theoretical Computer Science, Artificial Intelligence, and Robotics have studied multi-agent pathfinding under slightly different names.
The literature is full of many kinds of different definitions, assumptions, and notations.

In the following, we will mainly refer to Ma and Koenig [2017] and Stern et al. [2019] for formally describing MAPF, its elements, and its variants. The authors introduced a unified terminology to *"navigate through and understand existing literature and to establish appropriate baselines for comparison"*.

### 1.2.1 Formal Definition

*MAPF* is an example of a multi-agent planning problem in which the task is to plan paths for a group of agents to a set of target destinations. The key constraint is that the agents must follow these paths concurrently without colliding with each other (Silver [2005]).

Let us focus on the ***Classical (Offline) MAPF*** problem.

**Definition 1** *The main inputs to a **classical MAPF** problem are:*

- *an undirected graph $G = (V, E)$;*
- *a set of $k$ agents (e.g., robots, cars, characters, ...);*
- *a function $s : [1, \ldots, k] \to V$, mapping an agent to a source vertex;*
- *a function $t : [1, \ldots, k] \to V$, mapping an agent to a target vertex;*

*Time is discretized. In every time step, each agent is located in a single graph vertex and can perform a single action.*

*An* ***action*** *is a function $a : V \to V$ such that $a(v) = v'$ means that if an agent is at node $v$ and performs $a$, then it will be in node $v'$ in the next time step. There are two types of actions:* ***wait*** *(the agent stays in its current node another step) and* ***move*** *(the agent moves from its current node $v$ to an adjacent one in the graph, $v'$).*

*Generally, for a sequence of actions $\pi = (a_1, \ldots, a_n)$ and an agent $i$, the location of the agent after executing the first $x$ actions in $\pi$, starting from its source $s(i)$, is denoted by $\pi_i[x]$. A sequence of actions $\pi$ is a* ***single-agent plan*** *for agent $i$ if and only if executing this sequence of actions in $s(i)$ results in being at $t(i)$. A* ***solution*** *is a set of $k$ single-agent plans, one for each agent.*

### 1.2.2 Objective functions

Researchers developed many objective functions to evaluate the MAPF solution. The most common ones are *makespan* and *sum of costs*.

- ***Makespan.*** It is the number of time steps required for all the agents to reach their target. For a MAPF solution $\pi = \{\pi_1, \ldots, \pi_k\}$, the makespan of $\pi$ is defined as $max_{1 \leq i \leq k} |\pi_i|$.
- ***Sum of costs.*** This objective function is given by the sum of time steps required for every agent to reach its target. The sum of costs of $\pi$ is defined as $\sum_{1 \leq i \leq k} |\pi_i|$ and is also known as ***flowtime***.

These are not the only possible objective functions. Others could be, for example, the total non-waiting actions required to reach the target, sometimes referred to as the ***sum-of-fuel***, or the maximum number of agents reaching their targets within a given time, i.e., ***deadline***.

### 1.2.3 Characteristics

#### MAPF Solvers

MAPF algorithms for finding valid solutions can be partitioned into three classes: *optimal solvers, sub-optimal solvers,* and *bounded sub-optimal solvers* (Barer et al. [2014]).

- ***Optimal Solvers***. When they find a solution, these methods bring an optimal one, i.e., no other solutions are better than this.
- ***Unbounded Suboptimal Solvers***. Many existing MAPF solvers aim at finding a solution fast while allowing the returned solution to be suboptimal. Most suboptimal MAPF solvers are *unbounded*, i.e., they do not provide any guarantee on the quality of the returned path.
- ***Bounded Subptimal Solvers.*** A *bounded suboptimal* search algorithm accepts a parameter $w$ (sometimes known as $1+\epsilon$) and returns a solution that is guaranteed to be less than or equal to $w \times C^*$, where $C^*$ is the cost of the optimal solution. Bounded

suboptimal search algorithms provide a middle ground between optimal algorithms and unbounded suboptimal algorithms. Setting different values of $w$ allows the user to control the trade-off between runtime and solution quality.

A fundamental property MAPF solvers might or might not have is the *completeness*. A MAPF solver is ***complete*** if it always finds a solution to the problem when it exists.

### Conflicts

The goal of MAPF solvers is to find a solution without collisions. To achieve this, the notion of ***conflicts*** during planning is necessary.
Let $\pi_i$ and $\pi_j$ be a pair of single-agent plans.

- ***Vertex conflict.*** A *vertex conflict* between two single-agent plans occurs if and only if according to these plans the agents are planned to occupy the same node at the same time step. Formally, there is a *vertex conflict* between $\pi_i$ and $\pi_j$ if and only if there exists a time step $x$ such that $\pi_i[x] = \pi_j[x]$.
- ***Edge conflict.*** An *edge conflict* between two single-agent plans occurs if and only if according to these plans the agents are planned to traverse the same edge at the same time step in the same direction. Formally, an *edge conflict* between $\pi_i$ and $\pi_j$ occurs if and only if there exists a time step $x$ such that $\pi_i[x] = \pi_j[x]$ and $\pi_i[x+1] = \pi_j[x+1]$.
- ***Following conflict.*** A *following conflict* between two single-agent plans occurs if and only if one agent is planned to occupy a vertex that was occupied by another agent in the previous time step. Formally, a *following conflict* between $\pi_i$ and $\pi_j$ occurs if and only if there exists a time step $x$ such that $\pi_i[x+1] = \pi_j[x]$ (this definition may be used to avoid problems in case of default/failure inside the system).
- ***Cycle conflict.*** A *cycle conflict* between a set of single-agent plans occurs if and only if the same time step every agent moves to a vertex that was previously occupied by another agent, forming a "rotating cycle" pattern. Formally, A *cycle conflict* between a set of single-agent plans $\pi_i, \pi_{i+1}, \ldots, \pi_j$ occurs if and only if there exists a time step $x$ in which $\pi_i[x+1] = \pi_{i+1}[x]$ and $\pi_{i+1}[x+1] = \pi_{i+2}[x] \ldots$ and $\pi_{j-1}[x+1] = \pi_j[x]$ and $\pi_j[x+1] = \pi_i[x]$.
- ***Swapping conflict.*** A *swapping conflict* between two single-agent plans occurs if and only if the agents are planned to swap locations in a single time step. Formally, a *swapping conflict* between $\pi_i$ and $\pi_j$ occurs if and only if there exists a time step $x$ such that $\pi_i[x+1] = \pi_j[x]$ and $\pi_j[x+1] = \pi_i[x]$. This conflict is sometimes called *edge conflict*. Note, a swapping conflict is a cycle conflict for two agents.

From the definitions above, we have the following implications:

$$edge\ conflict \Rightarrow vertex\ conflict$$

$$cycle\ conflict \Rightarrow following\ conflict$$

$$swapping\ conflict \Rightarrow following\ conflict$$
$$swapping\ conflict \Rightarrow cycle\ conflict$$

Hence:

$$forbidden\ vertex\ conflict \Rightarrow forbidden\ edge\ conflict$$
$$forbidden\ following\ conflict \Rightarrow forbidden\ cycle\ conflict$$
$$forbidden\ following\ conflict \Rightarrow forbidden\ swapping\ conflict$$
$$forbidden\ cycle\ conflict \Rightarrow forbidden\ swapping\ conflict$$

To properly define a classical MAPF problem, one needs to specify which types of conflicts are allowed in a solution.

#### Behavior at target

In a classical MAPF solution, the agents may reach their target at different time steps, so we have to define how an agent behaves in the time steps after it has reached its final node but before everyone has reached its target. The two common assumptions for how agents behave at their targets are *stay at target* and *disappear at target*.

- ***Stay at target.*** An agent stays at its target until all agents have reached their targets. This waiting agent will cause a vertex conflict with any plan that passes through its target after it has reached it.
- ***Disappear at target.*** When an agent reaches its target, it immediately disappears. So, it won't collide with anyone once arrived.

Note, if the agent-at-target behavior is *stay at target* and the objective function is *sum of costs*, then one needs to specify how staying at a target affects the sum of costs. The common assumption is that an agent staying at its target counts as a wait action unless it is not planning to move away from this location.

Intuitively, the easiest hypothesis to avoid future obstacles is agents disappear at the target.

### 1.2.4 Variants

#### MAPF on Weighted Graphs: Different moves duration and costs

In the classical MAPF setting, we assume each action takes exactly one time step. In more complex MAPF literature motion models, different actions may have *different duration*. In this latter case, the graph representing the potential locations that agents may occupy becomes a weighted graph where every edge's weight represents the time needed to traverse this edge (or the cost to use that edge).

The types of weighted graphs used in research include:

- ***MAPF in*** $2^k$***-neighbor grids.*** Such maps are a restricted form of weighted graphs in which every vertex represents a cell in a two-dimensional grid. The move actions of an agent in a cell are all its $2^k$ neighboring cells, where $k$ is a parameter.

- ***MAPF in Euclidean Space.*** This is a generalization of MAPF in which every node represents a Euclidean point $(x, y)$ and the edges represent allowed move actions.

### From Pathfinding to Motion Planning: Closing the gap with more realistic settings

In classical MAPF, agents are assumed to occupy one vertex, with no volume or shape, and move at a constant speed. On the other hand, motion planning algorithms directly consider these properties. There, an agent belongs at each time step to a *configuration* instead of only a vertex, where a ***configuration*** specifies the agent location, orientation, velocity, etc., and an edge between configurations represent kinematic motion.

Several MAPF variants are steps towards closing this gap between classical MAPF and motion planning. In ***MAPF with large agents***, agents have a specific geometric shape and volume. In ***MAPF with kinematic constraints***, we consider kinematic constraints over agents' actions.

### MAPF with more tasks

While in the classical MAPF each agent has the only task of reaching its target cell, several extensions consider agents with more than one target.

- ***Anonymous MAPF.*** In this case, the objective is to move the agents to a set of target vertices, but it does not matter which agent reaches which target. In other words, every agent can be assigned to any target (but necessarily a one-to-one mapping).
- ***Colored MAPF.*** Here, agents are grouped into teams with a set of targets. This is a generalization of the *anonymous MAPF*. One can generalize colored MAPF even further, assigning a target and an agent to multiple teams.
- ***Online MAPF.*** In *online MAPF* (also called ***Lifelong MAPF***), a *sequence* of MAPF problems is solved on the same graph. *Online MAPF* can be classified as follows:
  - ***Warehouse model.*** A fixed set of agents solves a MAPF problem, but *after* an agent finds a target, it may be tasked to go to a *different* target, taking inspiration from MAPF for autonomous warehouses.
  - ***Intersection model.*** Here, new agents may appear *while* the others are following their plan, and each agent has the *only* task to reach its target. This setting takes inspiration from autonomous vehicles entering and exiting intersections.

# Chapter 2

# Introduction to the Specific Problem

In the previous chapter, we presented the general MAPF framework, starting from the historical background, presenting its main features, and arriving at variants of the classical MAPF. Let us now focus on the specific problem we tackle in this document.

*This work is part of a larger project on decentralized solutions for large-scale path planning problems.* In particular, It has the final objective to develop a solution to the classic MAPF problem (Ma and Koenig [2017], Stern et al. [2019]) with the following characteristics:

- large grid [1] with ≈ 15,000 cells;
- a small number of obstacles ( ≈ 14,000 free cells );
- no narrow passages;
- 75,000 agents moving on the grid daily with 7,000 at peak hour;
- online setting, with new agents appearing any time and having to reach their targets;
- agents disappearing at target, with some potential queue formation;
- goal: maximize daily throughput.

This request refers to the problem of automation inside a large warehouse. A *physical* grid of ***tiles*** composes the floor. Over this grid, apart from fixed obstacles, tiles are either free or occupied by at most one *skid*. A ***skid*** (also called ***cart agent***) is a squared mobile device that moves from tile to tile, carrying specific objects, like boxes, from an origin

[1] Although this work focuses on grid environments, the presented algorithms apply equally to more general pathfinding domains. Any discontinuous environment can be used, as long as every agent's route is planned with discrete motion elements.

to the corresponding destination tile. These skids correspond to the mobile agents of the classical MAPF setting, but in this case, we have a new kind of agent, the *tile*.

The two types of intelligent agents, i.e., skids and tiles, communicate with each other to accomplish the MAPF goal. [2] Specifically, a generic cart agent stays on a tile agent (so the latter is *occupied*). It builds the shortest path considering the grid as empty except for the fixed obstacles. Then, it starts moving on that path, one step at a time. Before moving to a new tile, it checks its availability: if the tile is available, i.e., not occupied, it moves there (so the new tile becomes *busy* and the first one *available*), otherwise the cart agent has to proceed differently (waiting without changing cell, moving to another cell, or appropriately moving to the interested cell). This method is useful when dealing with the *following conflicts* [1.2.3].

Once the skid reaches its goal, we assume it exits the system (*disappear at target*). In more realistic situations, the mobile agent could come back to the origin, to carry new uploaded boxes to new destinations. Intuitively, this last kind of problem is similar to a *lifelong MAPF problem* [1.2.4], in which each cart agent completes a task (in our case, to go from the origin to the destination), and then another one (to come back from the destination to the origin).

The goal is to maximize the number of agents reaching their destination in one day, i.e., the *daily throughput*.

We include the possibility of agents to appear any random time in the grid (*online setting* [1.2.4]). For this reason, while implementing our algorithm, we must avoid online skids being inserted upon occupied cells [6.1.1].
While developing our initial algorithm, we first opted for an offline setting then we extended it to the case of online agents.

Many types of conflicts may arise during the solving phase [1.2.3]. The majority of the literature articles consider *only vertex and edge conflicts*.
Thanks to the specific problem framework and the implementation technique [3.1], *vertex conflicts never happen*.

As the reader may imagine, we should consider many more aspects to apply a MAPF search algorithm to reality. For example, in physical applications, failures could arise. One should model the possibility that some hardware component breaks, not enabling the physical passage of the skid from one to another tile. We could use a warning function to disable that specific neighborhood temporarily.
Problems like this are out of the objectives of this document. We assume no robot motion failure will occur.

[2] We use interchangeably the terms skids, mobile agents, cart agents, and the terms tile, node, and cell.

# Chapter 3

# Implementation & Analysis

## 3.1 Coding

A crucial part of this work consists of the *code implementation* of the developed MAPF algorithms. While the developed algorithms are decentralized, the implementation of the multi-agent system is centralized on an Intel Core i3 PC. So necessarily, the code manages the agents sequentially, whereas they are processed contemporarily in warehouse applications. This is not a restriction, as long as we guarantee that the ordering of how agents are processed doesn't negatively influence the results [6.1.1].

*Object Oriented Programming* [1] is fundamental to simulate multi-agent systems with hundreds or thousands of mobile agents. We use `Python` programming language, version 3.9 (python) for coding, being profitable both in terms of power and computation speed.

It is possible to simulate and solve MAPF problems by considering every skid and tile as belonging to a specific data structure.

Every skid is sequentially processed as follows. It checks whether the tile of interest is available (in real applications, communication protocols are needed). If so, it occupies the cell. Otherwise, it proceeds differently. In this way the only possible type of conflict is the edge collision; there is no possibility of *vertex collisions* [1.2.3]. Despite this, we are not limiting the set of possible applications. Indeed, from a practical point of view, it cannot happen that two skids make availability requests simultaneously for the same tile. In other words, tiles dominate the movement of skids.

To guarantee uniform speed for the skids, we impose in the codes that no agent can make more than one action. Without this, the algorithm would produce very different results.

So we conclude that considering tiles as agents has the major advantage of avoiding vertex conflict. Nevertheless, while tiles avoid physical cell sharing, this doesn't preclude

[1] ***Object Oriented Programming (OOP)*** is a programming paradigm that relies on the concept of "objects" which can contain data and code (https://en.wikipedia.org/wiki/Object-oriented_programming).

deadlocks formation around cells. For this reason, in this work, one of the primary objectives is to reduce congested areas.

## 3.2 Analysis & Visualization

To ***compare*** different algorithms, with different grids and scenarios, we measure some specific variables as outcomes of the run code [7]. We summarize and ***analyze*** these results using the `MATLAB` software (MathWorks). Further statistical aspects are considered with `R` too (Foundation). All these charts are enhanced using `PowerPoint` tool from the Microsoft Office package. An example is shown in Figure 8.1.

To ***visualize*** the simulated scenarios, we use specific `Python` libraries. For computational reasons, we check the performance on simple small grids with few agents. This helps us not only to evaluate the correctness of the implemented algorithm but also to understand specific behaviors.

# Chapter 4

# Benchmarks

## 4.1 Benchmark Grids and Scenarios

To simulate multi-agent systems, we consider specific real and synthetic domains developed by the MAPF researchers and uploaded on the *MAPF community website* (Koenig [2022], Stern et al. [2019]). The database contains more than 30 benchmark grids (both real and synthetic), described as text files, every one containing as many rows and columns as the dimension of the grid. This text contains two types of characters:

- '.': this character indicates a grid's node free of fixed obstacles;
- 'T' / '@': this means that a specific cell contains a fixed obstacle; mobile agents cannot move there.

For each of these benchmark grids, there are 25 ($\times$ 2) benchmark scenario sets. Each benchmark scenario file contains a list of agents' start/goal locations, with a maximum length of 1000 randomly generated start-goal couples. The idea is to consider problems where we add one agent at a time until an algorithm cannot solve a problem in a given time/memory limit.

In this document, we focus on *3 benchmark grids*:

- $48 \times 48$ *empty grid*. This domain doesn't have any fixed obstacle, so the number of free cells is $48 \times 48 = 2304$ (without considering skids); recalling [2], to simulate a 50% of free cells occupied by mobile agents, we should consider approximately 1200 agents. This is what we do in [7].
- $256 \times 256$ *Boston grid* (Figure 4.1). The number of free cells is 47768. To simulate a 50% of occupied cells, we should consider close to 23400 skids. For computational reasons, we limit ourselves to a maximum of 1200 agents in the domain.

- $128 \times 128$ *Maze Grid* [1] (Figure 4.2). The number of free cells is 14818. To simulate the 50% of occupied cells, we should consider close to 7500 skids. Again, for computational reasons, we limit ourselves to a maximum of 1200 agents.

Figure 4.1: Boston Benchmark Grid.

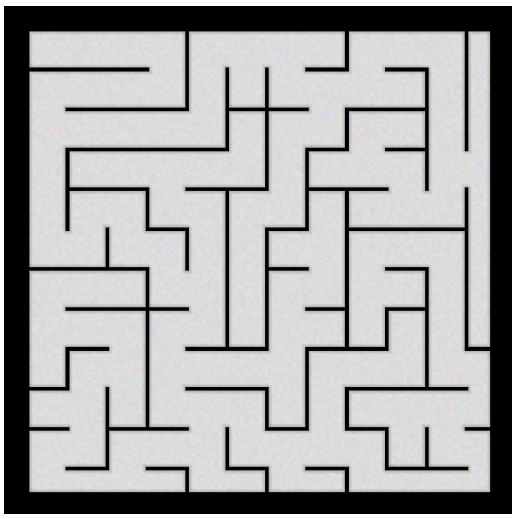
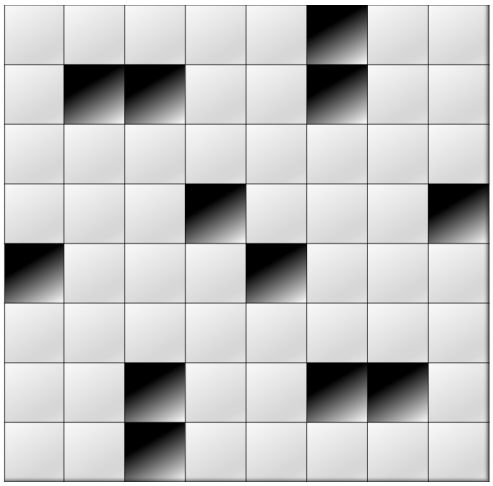


Figure 4.2: Maze Benchmark Grid.

[1] When we apply our algorithms to these more complex grids, we have additive information on their properties, specifically, the length of the paths and the presence of narrow passages.

Moreover, we build two specific small benchmark grids (and the corresponding benchmark scenarios) to be used for checking the correctness of our algorithms (Figures 4.3a, 4.3b, 4.4).



(a)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $S_4$ | $G_{14}$ | $G_{12}$ | $S_{15}$ | $G_5$ | | $G_9$ | $S_8$ |
| $G_{22}$ | | | $S_{20}$ | $S_{21}$ | | $G_1$ | $G_{17}$ |
| $S_{17}$ | $S_{24}$ | $S_{23}$ | $G_{25}$ | $G_4$ | $S_{11}$ | $S_{22}$ | $G_{26}$ |
| $G_{21}$ | $S_{10}$ | $S_{26}$ | | $S_7$ | $G_{23}$ | $G_{20}$ | |
| | $G_{19}$ | $G_{13}$ | $S_2$ | | $S_{25}$ | $S_3$ | $S_{13}$ |
| $G_3$ | $G_{18}$ | $S_{14}$ | $G_{11}$ | $S_5$ | $G_{24}$ | $G_{16}$ | $S_{19}$ |
| $G_6$ | $S_1$ | | $S_{12}$ | $S_{18}$ | | | $S_6$ |
| $S_9$ | $G_{10}$ | | $G_2$ | $S_{16}$ | $G_{15}$ | $G_8$ | $G_7$ |

(b)

Figure 4.3: Custom Benchmark Grid. Agents must navigate from $S_i$ to $G_i$.

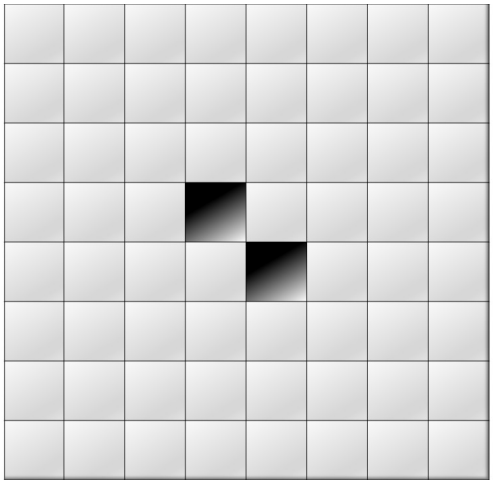


Figure 4.4: Second Custom Benchmark Grid. Agents must navigate from $S_i$ to $G_i$.

## 4.2 Benchmark Algorithms (from the literature)

As presented in [1.1], the literature is full of decentralized MAPF algorithms. Silver [2005] can be considered as the masterpiece of decoupled MAPF approaches. Its three decentralized MAPF algorithms contain innovative ideas of cooperation between agents, starting a new era in which agents don't focus on themselves anymore, but navigate taking care of the others' actions, either completely from the start to the end of their journey or partially. Even today, the most advanced methods are based on these ideas.

We initially focus on a self-interested policy, where agents don't share their world's knowledge with any other [1.1.4]. Then we improve this basic approach, introducing some form of cooperation by letting agents share some properties but still trying to solve conflicts locally, not in advance. In [8.2.2], we compare our best version with the three Silver's "benchmark" algorithms, showing cooperation outperforms selfishness, although sometimes the latter may find solutions where the former doesn't.

In the following, we give a full description of these algorithms, respectively, CA*, HCA*, and WHCA*, reviewed with the following point:

- method;
- year;
- solution;
- approach;
- motivation;

- brief description;
- hierarchy;
- abstraction;
- conflicts;
- priority;
- heuristic;
- behavior at target;
- pros and cons.

Before this, we recall the algorithm LRA* that Silver defined as *"the current video-games industry standard"* in 2005 and that the cooperative approaches outperform.
Finally, we specify which will be considered in [8.2.2].

### 4.2.1 LRA* (Local Repair A*)

- **Method.** *Decentralized method* for MAPF.
- **Year.** Before 2005.
- **Solution.** Suboptimal solution.
- **Approach.** Offline approach.
- **Motivation.** This algorithm was the video-games-industry standard in 2005. It is the decoupled adaptation of the A* algorithm.
- **Brief description.** Each agent searches for a route to the destination using the A* algorithm, ignoring all other agents except its current neighbors. The agents then start following their routes, until a collision is imminent. Whenever an agent is about to move into an occupied position it instead recalculates the remainder of its route. Cycles are possible, so it is usual to try and add some modifications to escape such problems.
- **Hierarchy.** No hierarchy is considered in this algorithm.
- **Abstraction.** Grid maps are the most popular approach to abstract a real-world navigation map into a search space. The map is discretized into a grid of atomic locations called tiles. Then, the standard way to build a search graph from a grid map is to define one node for each accessible tile; we define undirected edges between adjacent nodes, so moving between two adjacent locations is allowed in both directions. We discretize the virtual environment into a grid of tiles. Then easily we obtain a graph structure.

- **Conflicts.** Whenever an agent is about to move into an occupied position it instead recalculates the remainder of its route. Cycles are possible, so it is usual to try and add some modifications to escape such problems. One possibility is to increase an agent's agitation level every time it is forced to reroute. Agents will hopefully escape from the problematic area, as they behave increasingly randomly.
- **Priority.** Priority is not taken into account in this algorithm.
- **Heuristic.** This search method is substantially an adaptation of A*. So the heuristic is any admissible heuristic used in A*. Since increasing agents' agitation levels helps escape cycles problems, it modifies this heuristic. In particular, random noise is added to the A* heuristic, proportionally to the agitation level.
- **Behavior at target.** Once the destination is reached by the agent, it doesn't move anymore. Consequently, it may block off parts of the map to other agents.
- **Pros & Cons.** LRA* is known to have many drawbacks when applied to complex environments. If bottlenecks occur in crowded regions, they may take arbitrarily long to be solved. While caught in a bottleneck, agents constantly reroute in an attempt to escape, requiring a full recomputation of the A* search almost every turn. This leads to visually unintelligent behavior. Each agent makes a change in route independently, leading to cycles in which the same location may be visited by agents repeatedly in a loop.

### 4.2.2 CA* (Cooperative A*)

- **Method.** *Decentralized method* for MAPF.
- **Year.** 2005.
- **Solution.** Suboptimal solution.
- **Approach.** Offline approach.
- **Motivation.** This algorithm is developed to overcome the drawbacks of Local Repair A* by using cooperative search.
- **Brief description.** The task is decoupled into a series of single-agent pathfindings. Each search is performed in three-dimensional space-time taking into account the planned routes of other agents. After calculating each agent's route, the states along the path are marked on a reservation table and are considered impassable (for precisely the duration of the intersection) and avoided during searches by subsequent agents.
- **Hierarchy.** No hierarchy is considered.
- **Abstraction.** Each single-agent search is performed in three-dimensional space-time taking into account the planned routes of other agents. After calculating each agent's route, the states along the path are marked on a reservation table and are considered

impassable and avoided during searches by subsequent agents. The reservation table represents the agent's shared knowledge about each other's planned routes. It is a sparse data structure marking off regions of space-time.

- **Conflicts.** This method avoids conflicts, but it may be so restrictive that any decoupled greedy algorithm that pre-calculates the optimal path will not be able to solve some classes of problems, as explained in the following.
- **Priority.** Any decoupled greedy algorithm that pre-calculates the optimal path will not be able to solve some classes of problems. This can happen when a greedy solution for one agent prevents any solution for another agent. In general, such algorithms are sensitive to the ordering of the agents, requiring sensible priorities to be selected for good performance.
- **Heuristic.** CA* uses heuristics to search the single-agent path. Any admissible heuristic can be used, such as Manhattan distance (but we should consider better heuristics to help reduce the computation).
- **Behavior at target.** Once the destination is reached by the agent, it doesn't move anymore. Consequently, it may block off parts of the map to other agents.
- **Pros & Cons.** One drawback of this algorithm is the heuristic choice (as previously stated, we should consider better heuristics to help reduce the computation since the Manhattan distance gives a poor performance in more challenging environments). Even more important, there are some classes of problems that Cooperative A* cannot solve.

### 4.2.3 HCA* (Hierarchical Cooperative A*)

- **Method.** *Decentralized method* for MAPF.
- **Year.** 2005.
- **Solution.** Suboptimal solution.
- **Approach.** Offline approach.
- **Motivation.** A drawback of CA* is the choice of an appropriate admissible heuristic. In particular, we should consider a heuristics better than Manhattan distance to help reduce the computation. One method for improving a heuristic based on abstractions of the state space is to use Hierarchical A* (abstract distances are computed on-demand, which is more appropriate in a dynamic context), the basis for Hierarchical Cooperative A* (HCA*).
- **Brief description.** HCA* is just like CA* with a more sophisticated heuristic, the true distance heuristic, using RRA* (Reverse Resumable A*) to calculate the abstract distance on-demand. In particular, RRA* executes a modified A* algorithm in a reverse direction.

- **Hierarchy.** A hierarchy refers to a series of abstractions of the state space, each more general than those previous, and is not restricted to spatial hierarchy. HCA* uses a simple hierarchy containing a single domain abstraction, ignoring both the time dimension and the reservation table.
- **Abstraction.** Recall abstraction works by replacing one state space with another, the 'abstract' space, that is easier to search. The abstraction is a simple 2-dimensional map with all agents removed. Abstract distances can thus be viewed as perfect estimates of the distance to the destination, ignoring any potential interactions with other agents.
- **Conflicts.** Similarly to CA*, this method avoids conflicts, but it may be so restrictive that any decoupled greedy algorithm that pre-calculates the optimal path will not be able to solve some classes of problems.
- **Priority.** Similar to the priority of CA*.
- **Heuristic.** Heuristics are used both in the abstraction phase and when using RRA* to reuse search data. In one case, the heuristic is the true distance, while in the other it is the Manhattan distance heuristic.
- **Behavior at target.** Once the destination is reached by the agent, it doesn't move anymore. Consequently, it may block off parts of the map to other agents.
- **Pros & Cons.** This method overcomes CA*'s heuristic problems by computing abstract distances on-demand, via Hierarchical A*. One of the issues with Hierarchical A* is how to best reuse search data in the abstract domain and this is solved considering RRA* search in the abstract domain.

### 4.2.4 WHCA* (Windowed HCA*)

- **Method.** *Decentralized method* for MAPF.
- **Year.** 2005.
- **Solution.** Suboptimal solution.
- **Approach.** Offline approach.
- **Motivation.** CA* and HCA* present three main issues:
  - when an agent sits at its destination, it may block off parts of the map to other agents;
  - sensitivity to agents ordering;
  - high computational cost.

  WHCA* solves these problems by *windowing the search*.

- **Brief description.** Windowing the search means cooperative search is limited to a fixed depth specified by the current window. Each agent searches for a partial route to its destination and then follows it. At regular intervals, the window is shifted forward and the algorithm computes a new partial route. To ensure the agent heads in the correct direction, only the cooperative search depth is limited to a fixed depth, while the abstract search is executed to full depth. So the search is reduced to a $w$-step window using the abstract distance heuristic introduced for HCA*. Finally, we can reuse the RRA* search results for each consecutive window.

- **Hierarchy.** We said HCA* uses a simple hierarchy containing a single domain abstraction, ignoring the time dimension and the reservation table. In WHCA*, a window of size $w$ is like an intermediate abstraction. Hence we substantially have two levels of abstraction, an intermediate and a complete one.

- **Abstraction.** A window of size $w$ is like an intermediate abstraction, equivalent to the base level state space for $w$ steps and the abstract level state space for the remainder of the search. In other words, other agents are only considered for $w$ steps (via the reservation table) while ignored for the remainder of the search.

- **Conflicts.** With respect to previous algorithms, WHCA* avoids possible conflicts of agents blocking the motion of others. Cycles are indeed possible using this algorithm. To solve this, the agent's agitation level increases every time it must reroute, as previously explained.

- **Priority.** With this algorithm, it is possible to plan paths for groups of agents at different time steps. So the priority is fixed (similarly to CA* and HCA*) for agents inside the same group but changes between groups. This brings the same advantages as dynamic priorities.

- **Heuristic.** This algorithm uses heuristics in different moments. The search is reduced to a $w$-step window using the abstract distance heuristic introduced for HCA*. As for HCA*, WHCA* can use the Manhattan distance heuristic in the RRA* search phase. Finally, the distance heuristic is modified when cycles are formed: random noise is added to the distance heuristic in proportion to the agitation level, hoping the increasingly random movement will help agents escape from the problematic area.

- **Behavior at target.** The windowed search can continue once the agent has reached its destination. The agent's goal is no longer to reach the destination, but to complete the window via a terminal edge.

- **Pros & Cons.** There are at least three issues with previous algorithms. One issue is how they terminate once the agents reach their destination. If an agent sits on its destination, it may block off parts of the map to other agents. Ideally, agents should continue to cooperate after reaching their goals so that an agent can move off its destination and allow others to pass. A second issue is sensitivity to agent order. Although it is sometimes possible to prioritize agents globally, a more robust solution

is to dynamically vary the agents' order. A third issue is that the previous algorithms must calculate a complete route to the destination in a large three-dimensional state space. By windowing the search, WHCA* mainly solves these problems. With respect to centralized approaches, this method can significantly improve scalability and speed. However, such a method is incomplete. It provides no guarantees for the total running time and is unable to a priori tell whether it would succeed in finding a solution to a given instance.

### 4.2.5 Implementation: Drawbacks

Our algorithm's optimized version is compared with CA* and HCA*. We also implemented a simplified version of WHCA* [2], where the search is windowed, but we don't partition agents in sequentially processed groups, which instead would be useful in real-time scenarios while interleaving planning and execution, e.g., in interactive computer games. Our WHCA* simplified version hardly finds solutions even for few agents.

Finally, we implemented a Multi-agent A* version to check the difference between centralized and decentralized algorithms. Despite the correctness of the code, the disadvantages of centralized pathfinding in terms of time and space computational complexity are obvious, even for few agents on a small empty grid.

In conclusion, we proceed with the more constraining but still powerful (decentralized) cooperative pathfinding versions, i.e., CA* and HCA*.

[2] Silver showed many steps to improve HCA*, up to a complete version of WHCA* (Silver [2006]).

# Chapter 5

# Single-Agent Search Methods on Graphs

This chapter reviews the main shortest path methods for single agents (Dijkstra's and A* search), starting from the basic graph search techniques. We integrate their description with figures showing how they work on simple grids. Because of its advantages over the other algorithms, we use A* as the search function for developing our MAPF solvers.

## 5.1 Introduction to the Shortest Path Search

The ***shortest path search*** between two nodes is a fundamental operation on graph structures [1].

**Definition 2** *In the general problem setting, a* ***weighted graph*** *is a pair* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$*, where* $\mathcal{V}$ *is the set of vertices, and* $\mathcal{E}$ *is the set of edges (i.e., links connecting pairs of vertices), with real numbers assigned to edges (the weight of an edge).*
*An* ***unweighted graph*** *is a weighted graph with unitary edges weights.*
*A* ***path*** *is a sequence of vertices* $(v_1, \ldots, v_n)$ *such that* $v_i$ *is adjacent to* $v_{i+1}$ *for* $1 \leq i < n$*.*
*The* ***cost of a path*** *is given by the sum of all its edges' weights.*
*The* ***shortest path problem*** *is the problem of finding the path with the minimal cost.*

We may tackle this problem differently, depending on the specific objective we have:

- *all-pairs shortest path*;
- *single-pair shortest path*;
- *single-source shortest path*;
- *single-destination shortest path.*

[1] https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)

Specifically, if the goal is to find the shortest path between *all* node pairs (***all-pairs shortest path problem***), more efficient methods are different than those used when we are interested in finding the shortest path from one *single* node (***single-source shortest path problem***). The two other types of shortest path problems are the ***single-pair shortest path problem***, the problem of finding the shortest path connecting *two* generic nodes, and the ***single-destination shortest path problem***, where we look for the shortest path from *all* vertices to a *single* destination node [2].
Moreover, the best algorithm choice depends on the edges values properties: some algorithms are capable of working on graphs with just *positive* edges weights (e.g., ***Dijkstra's Algorithm***), while others are suitable even for problems on graphs with *null or negative* weights (e.g., ***Bellman-Ford Algorithm***) (Crescenzi et al. [2012]).

We further subdivide in *informed* and *uninformed* algorithms [3].

- ***Informed search*** (or ***heuristic search***) algorithms exploit a *"knowledge function"* (***heuristic function***) to find efficiently one or more solutions to a given search problem. The *heuristic function* is used to establish the best search order (i.e., priorities) to reduce the overall cost [4].
- ***Uninformed search*** (or ***blind search***) algorithms explore all possible states without exploiting any other information to solve the problem. Generally, it is inefficient since, by discarding any further knowledge, its computational cost explodes with the number of states [5].

The two main advantages of heuristic search over blind search are reduced computational cost and the avoidance of a computational explosion while searching.
The price to pay is the lack of completeness: uninformed search techniques bring always optimal solutions (when they exist), but informed searches don't.

For our analyses, we focus on single-source shortest paths between grid nodes. Here, the weights between edges are always positive. First, we analyze uninformed methods, graphically showing their drawbacks, followed by examples of heuristic search techniques (Stout [1997]).

[2] https://en.wikipedia.org/wiki/Shortest_path_problem

[3] https://www.geeksforgeeks.org/difference-between-informed-and-uninformed-search-in-ai/

[4] https://www.okpedia.it/ricerca_informata

[5] https://www.okpedia.it/ricerca_non_informata

## 5.2 Breadth-First Search (for the shortest path)

Considering a start node, this method iteratively extracts its neighbors. They are put into the ***frontier of the search***. Then neighbors of neighbors are extracted and added to the frontier too, which tends to enlarge. This loop is the basis for all search algorithms on graphs, including the state-of-the-art A* method.

Since Breadth-First Search is used for many graph problems, the loop doesn't build the paths by itself. It simply tells how to visit everything on the map. With a data structure (typically a dictionary) containing each node's parents (i.e., those nodes that have been visited at the previous time step), we reconstruct the shortest path, going backward from the goal to the start node.

In Figure 5.1 we show an example of how this specific search method explores a generic $8 \times 8$ empty grid, starting from a specific location (node $S$ in position $(x, y) = (6, 1)$). Every single loop expands the frontier of dashed light-blue nodes in all directions by adding a node's neighbors. Each figure's window shows the expansion up to a certain depth.

The search stops after extracting all the network nodes, even without specific information about the destination. Despite this, if the goal location is known, we could speed up the process by introducing the possibility of an ***early exit***: the search terminates whenever the destination node is reached. The *early exit* still guarantees an optimal solution because all the eventually unexplored nodes won't belong to any such optimal path.

This method suffers from two obvious problems.

1 . We need an improved method that saves the cumulative cost since in generic maps the steps may not be equal.

2 . It searches in all directions instead of directing its focus toward the goal.

The Breadth-First method is an uninformed search algorithm.

## 5.3 Bidirectional Breadth-First Search (for the shortest path)

This specific enhancement speeds up the process by a factor of 2 by running Breadth-First Search contemporarily both from the start and the goal location. The search stops if the two frontiers have a shared node (early exit setting) or when all the network nodes have been reached.

Figure 5.2 presents the first steps of the Bidirectional Breadth-First Search on a $8 \times 8$ empty grid starting from node $S$ (position $(x, y) = (6, 1)$) and heading to node $G$ (position $(x, y) = (0, 3)$). Compared with Figure 5.1, now the grid has approximately double-covered nodes for each maximal depth.

The drawbacks of Bidirectional Breadth-First Search are the same as the previous algorithm.
Bidirectional Breadth-First method is an uninformed search algorithm.

## 5.4 Depth-First Search

Depth-First Search (DFS) is the complement to Breadth-First Search: while the latter visits all a node's ***siblings*** (nodes at the same depth with the same father) *before any children*, the Depth-First method's idea is to visit all node's *descendant before any of its siblings*. In other words, in Breadth-First Search, siblings are visited before children, while in Depth-First Search, siblings are visited after children. A fundamental parameter of Depth-First Search is the ***depth limit***.

As intuition suggests, BFS is more suitable for searching vertices closer to the given starting node, while DFS is better when goal nodes are away from the origin. In this last case, BFS's time and space consumption are higher than in DFS.

Table 5.1 shows a direct comparison between depth-first search in terms of time and space complexity, completeness, and optimality.

| | BFS | DFS |
|---|---|---|
| Time complexity | $O(n+m)$ | $O(n+m)$ |
| Space complexity | $O(n+m)$ | $O(m)$ |
| Completeness | Yes | No |
| Optimality | Yes | No |

Table 5.1: Comparison between Breadth-First Search (BFS) and Depth-First Search (DFS) as graph and tree search algorithms. $n$ stays for the number of nodes, while $m$ is the number of edges. Time and space complexity are considered. BFS is always complete, but there may be situations in which DFS doesn't find a solution even if one exists. Finally, different from BFS, DFS is a suboptimal solver.

In Figure 5.3 we show the functioning of Depth-First Search applied to the usual $8 \times 8$ empty grid starting from a source node $S$ in position $(x, y) = (6, 1)$. Assuming a counterclockwise ordering of the neighbors to be visited as that in the top-left part of the algorithm, it is quite understandable the evolution of the search on this simple grid: the search will proceed on the right until the border is reached, then it will proceed upwards, and so on.

To avoid infinite loops, every time a node is expanded, we check whether the neighbor we are considering was already visited. If so, we compare the associated costs as described in Dijkstra's Algorithm [5.6]. The idea of signing nodes and comparing costs lets us understand if a specific node has to be revisited or not. This helps avoid infinite loops, not only in DFS but in all graph search methods.

As previously done, we could explore all the grid, or include an early exit check: if the extracted node coincides with the goal node, then the algorithm stops. As shown in Table

5.1, the problem with considering an early exit is that DFS could extract a sub-optimal solution. To understand this, consider Figure 5.3 again. If the goal node were in position $(x, y) = (0, 3)$, the algorithm with the early exit technique would return a longer path, going all along the grid border.

Drawbacks of BFS are included in those of DFS too.
Depth-First Search is an uninformed search method.

## 5.5 Iterative-Deepening Depth-First Search

This is an enhancement of DFS. The *depth limit* choice is quite important, but not easy to make. If we impose a too small depth, we might not reach the goal node. On the other hand, a deep search could be costly or even unnecessary.
The remedy imposed by Iterative-Deepening Depth-First Search consists in applying DFS consecutively, each time increasing the depth parameter until the goal is reached.

The idea of this algorithm is shown in Figure 5.4. As we can see, the search begins at small depths and increases over time.

The behavior is similar both to DFS and BFS. So, we conclude this is a compromise between the two techniques, proceeding in-depth and amplitude at the same time.
This is an uninformed search algorithm.

## 5.6 Dijkstra's Algorithm

In many problems, we should consider *maps with varying costs* for passing from a cell to an adjacent one. Contrary to Breadth-First and Bidirectional Breadth-First methods, this is a classical approach for traversing graphs with weighted edges.

First, the set of unsigned vertices equals the set of nodes minus the starting vertex. During the *initialization step*, we look for all the followers of the starting vertex. The shortest path between the starting node and its successors is the edge weight. The starting node is the predecessor of these vertices. These values are saved into a *priority queue* with the corresponding nodes, defining the *frontier*. After this, the algorithm passes to the next step, the iterated one.

The *iterated step* consists in extracting the node with higher priority from the frontier, i.e., the one with a smaller cost from the origin vertex. We consider its neighbors, but before adding them to the frontier (as done in Breadth-First Search), Dijkstra's algorithm assigns a *priority value* for each.

The ***priority value*** is the cost of the shortest path from the origin vertex to that considered node. To compute the cost of the path, we proceed as follows. We compare the priority value assigned to the neighbor to the cost of the path passing through the predecessor of the considered neighbor. If the second priority is smaller than the previously assigned value, we update the cost. If the neighbor doesn't have an assigned priority yet, i.e, it is unsigned, then we assign the cost of the path passing through its predecessor.

**Algorithm 1** Dijkstra's Search Code.

```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = dict()
came_from[start] = None
while not frontier.empty() do
    current = frontier.get()
    if current == goal then
        break
    end if
    for next in graph.neighbors(current) do
        if next not in came_from then
            priority = heuristic(goal, next)
            frontier.put(next, priority)
            came_from[next] = current
        end if
    end for
end while
```

Figure 5.5 shows Dijkstra's Search on an $8 \times 8$ empty grid with movement costs: for easier comprehension, we assigned green/red values on the left-upper part of each cell;

the movement cost between two adjacent nodes is given by the maximum between the two corresponding values. At the same time, on the right-bottom corner of the vertex, we have the cumulative smallest cost required to move from the origin to that specific position. This last value is used by the algorithm to understand which frontier node to expand first.

This figure is split into *two parts*, to compare the behavior on two grids with slightly *different cell values*. This is an example where *different cell values imply different shortest paths*. In this way, we see both the search process and the different consequences for different grids. More specifically, while computing the shortest path, higher cell values in positions $(3, 4)$ ad $(4, 4)$ bring higher priority values for the corresponding nodes. Nodes in positions $(3, 1)$ and $(4, 3)$ will be expanded first, opening new positions that will bring the shortest path (Figure 5.5d).

As previously done, we could explore all the grid, or include an *early exit check*: if the extracted node coincides with the goal node, then the algorithm stops. Even though this method solves the problem of searching on graphs with varying edge costs, the other drawback remains: we are ignoring the direction along which to easily find the solution. This is an uninformed algorithm.
Code 5.6 shows the coded algorithm [6].

[6] Credits to: https://www.redblobgames.com/pathfinding/a-star/introduction.html

## 5.7 Best-First Search

Both Breadth-First Search and Dijkstra's Search have the drawback of a frontier expanding in all directions. This is good if we search shortest paths to many or even to all the graph's nodes. On the other hand, it might be unnecessary if we know the goal position. A powerful method exploiting this knowledge is the *informed* algorithm of Best-First Search.

The idea is the same as before: to expand the frontier towards the destination in the best possible way. Like in Dijkstra's algorithm, we use a priority queue to decide which frontier node to expand first. But now we assign a higher priority to nodes we *estimate* to be closer to the goal vertex, instead of considering the length from the source.

Figure 5.6 shows the functioning of this method on an $8 \times 8$ empty grid with movement costs: we assigned green/red values on the left-upper part of each cell; the movement cost between two adjacent nodes is given by the maximum between the two corresponding values. On the right-bottom corner of frontier vertices, we have the estimated cost required to move from that specific position to the goal location. This last value is used to choose which frontier node to expand first. In the case of many frontier nodes with the highest priority, the choice is random.

This is an example of *heuristic* search, where we define an *heuristic function* to have an estimate of the distance to the goal location. Closer nodes are extracted first. Although it is fast, generally this search method doesn't find optimal solutions. The reason why this happens is the use of the heuristic function: it just gives an estimate of the distance to the goal, but in some cases, this could lead to the passage through costly areas (like in our example). In Figure 5.6 the *Manhattan distance* was used as heuristic function [7].

The heuristic choice guaranteeing optimality is the ***true distance heuristic***. It substantially calculates the shortest path between the goal and the analyzed node. But this means an optimal search method should be used for Best-First Search to be optimal too. In conclusion, an optimal algorithm cannot be based just on an estimate.

[7] The ***Manhattan distance*** between two grid nodes is the sum of their projections on the x and y axes.

## 5.8 A* Algorithm

The best two algorithms so far are Dijkstra's and Best-First Search. Dijkstra's algorithm produces optimal paths with a costly technique since the frontier expands in all directions. Best-First Search is a heuristic search method that exploits a heuristic function to estimate the distance from the goal location. It guarantees a higher speed in finding the solution but brings a solution that is not necessarily optimal.

A* algorithm merges the advantages of both these two methods, guaranteeing an *optimal solution in reduced time*. The *priority value* equals the sum of the distance of the node from the starting position with the estimated distance between the node and its destination (based on the heuristic function definition). The vertex with a smaller sum (i.e., an estimate of the path value passing from that node) is the first to be analyzed.
Let us see more in detail.

The A* algorithm maintains two sets, the *OPEN* list, and the *CLOSED* list. The ***OPEN*** list contains those nodes that need to be examined, while the ***CLOSED*** list keeps track of those that have already been examined. Initially, the OPEN list contains just the initial node, and the CLOSED list is empty. Each node $n$ in the graph maintains the following additional information:

- $g(n)$: the cost of getting from the initial node to $n$;
- $h(n)$: the estimate, according to the heuristic function, of the cost of getting from $n$ to the goal node;
- $f(n) = g(n) + h(n)$: intuitively, this is the estimate of the best solution that goes through $n$.

A* has a *main loop* that repeatedly gets the node, $n$, with the lowest $f(n)$ value from the OPEN list (in other words, the node that we think is the most likely to be part of the optimal path).
If $n$ is the goal node, then we are done, and we return the solution by backtracking from $n$. Otherwise, we remove $n$ from the OPEN list and add it to the CLOSED list. Next, we generate all the possible successor nodes of $n$.
For each successor node $n'$, if it is already in the CLOSED list and the copy there has an equal or lower $f$ estimate, then we can safely discard the newly generated $n'$ and move on (we can do this since a copy with a better estimate on the CLOSED list means we have already looked at it, and the new copy won't do any better). Similarly, if $n'$ is already in the OPEN list and the copy there has an equal or lower $f$ estimate, we can discard the newly generated $n'$ and move on (we're going to be looking at a better version of $n'$ later, so no need to keep this one round). If no better version of $n'$ exists in either the CLOSED or OPEN lists, we remove the worst copies from the two lists and set $n$ as the parent of $n'$. We also have to calculate the cost estimates for $n'$. Lastly, we add $n'$ to the OPEN list and return to the beginning of the main loop.

A* is a best-first search algorithm where the merit of a node $n$, $f(n)$, is the sum of the actual cost of reaching that node from the initial state, $g(n)$, and the estimated cost of reaching the goal state from that node, $h(n)$.
Code 5.8 shows the coded algorithm [8].

**Algorithm 2** $A^*$ Search Code.

```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = dict()
cost_so_far = dict()
came_from[start] = None
cost_so_far[start] = 0
while not frontier.empty() do
    current = frontier.get()
    if current == goal then
        break
    end if
    for next in graph.neighbors(current) do
        new_cost = cost_so_far[current] + graph.cost(current, next)
        if next not in cost_so_far or new_cost < cost_so_far[next] then
            cost_so_far[next] = new_cost
            priority = new_cost + heuristic(goal, next)
            frontier.put(next, priority)
            came_from[next] = current
        end if
    end for
end while
```

Figure 5.7 shows the functioning of this method on an $8 \times 8$ empty grid with movement costs: we assigned green/red values on the left-upper part of each cell; the movement cost between two adjacent nodes is given by the maximum between the two corresponding values. On the right-bottom corner of frontier vertices, we read the estimated cost required to move from that specific position to the goal location. This last value is used to choose which frontier node to expand first. In the case of many frontier nodes with the highest priority, the choice is random. We specify each node's parent with black arrows. These arrows represent the pointer to the parent and are saved in the specific data structure for computing the path once the goal is reached.
Figure 5.7b shows the situation in which the parent node is changed for the vertex in position $(x, y) = (4, 4)$. This is useful to understand that sometimes the process could extract again some nodes to update this information.

[8] Credits to: https://www.redblobgames.com/pathfinding/a-star/introduction.html

A* has the property that it will always find an optimal solution to a problem if the heuristic function never overestimates the actual solution cost. Its major drawback is that it requires exponential space in practice. Depending on the situation, A* is preferred to Dijkstra and vice versa. Indeed, A* exploits the heuristic to move along the most promising direction, while Dijkstra's search explores uniformly every direction. This last approach is useful when we are looking for alternative optimal paths.
A* is an example of a heuristic search.

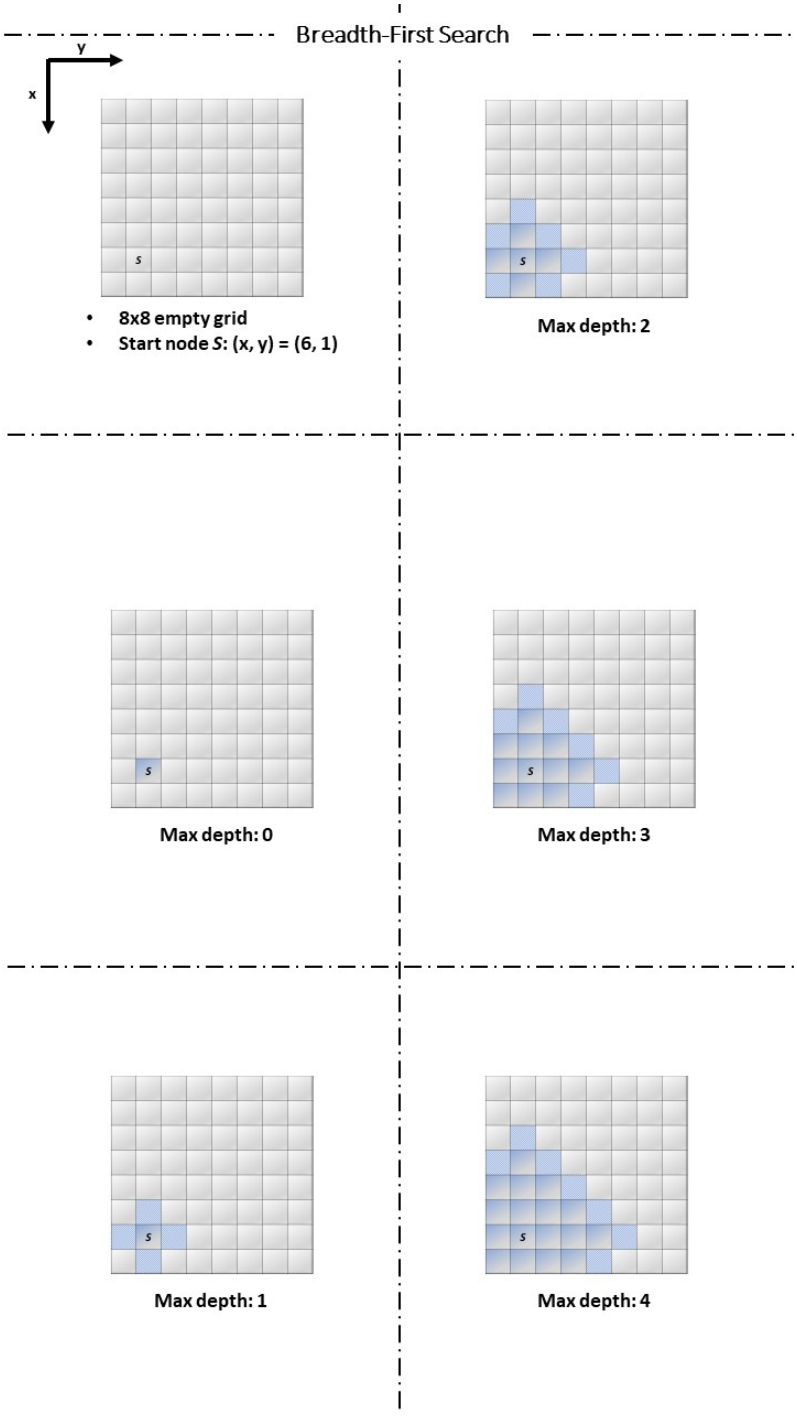



Figure 5.1: Breadth-First Search example. The grid is the $8 \times 8$ empty square (16 free tiles). The search starts from node $S$ in position $(x, y) = (6, 1)$. The expanding frontier is made by dashed light-blue nodes. Already visited nodes are represented in shadowed blue color.

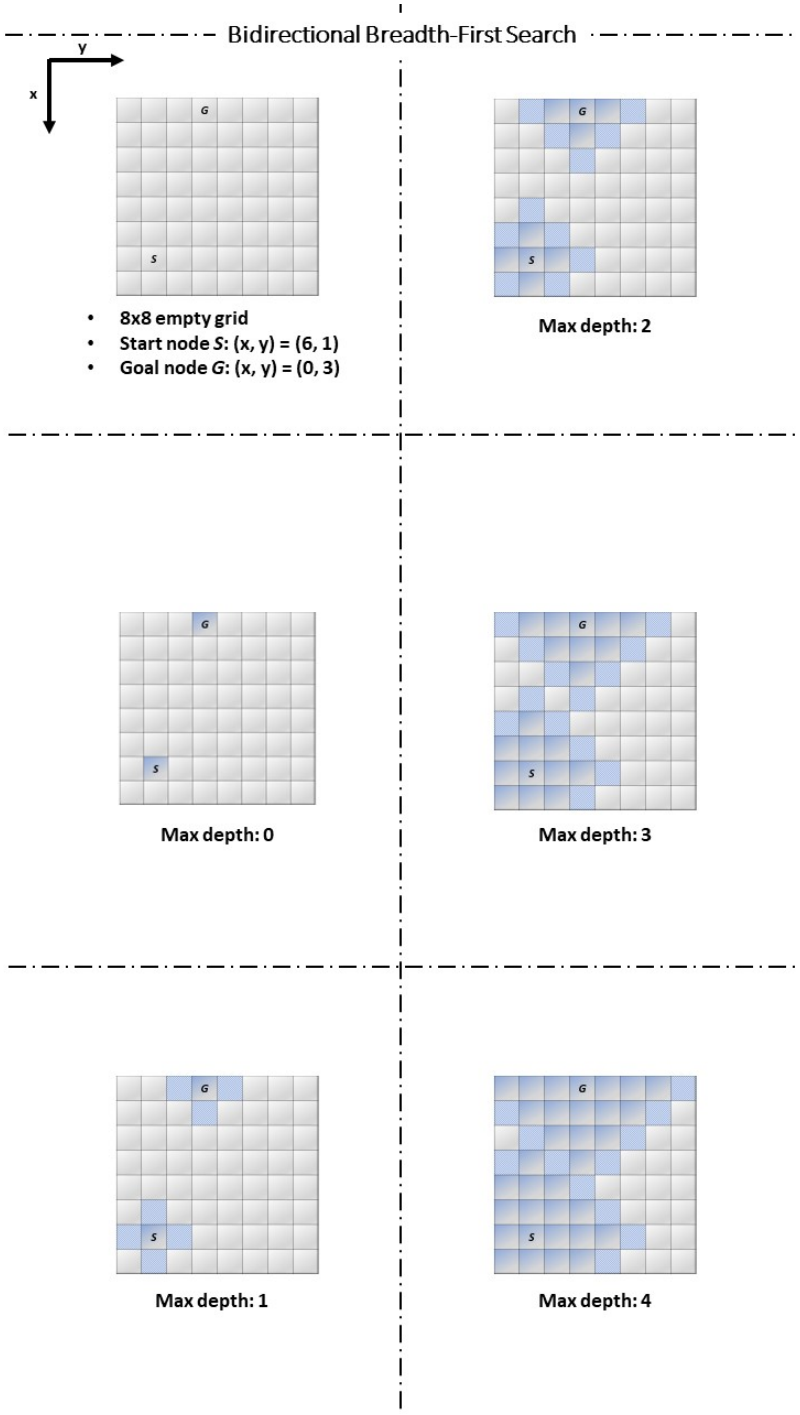



Figure 5.2: Bidirectional Breadth-First Search example. The grid is the $8 \times 8$ empty square (16 free tiles). The search starts both from node $S$ in position $(x, y) = (6, 1)$ and the goal $G$ in $(x, y) = (0, 3)$. The expanding frontier is made by dashed light-blue nodes. Already visited nodes are represented in shadowed blue color.

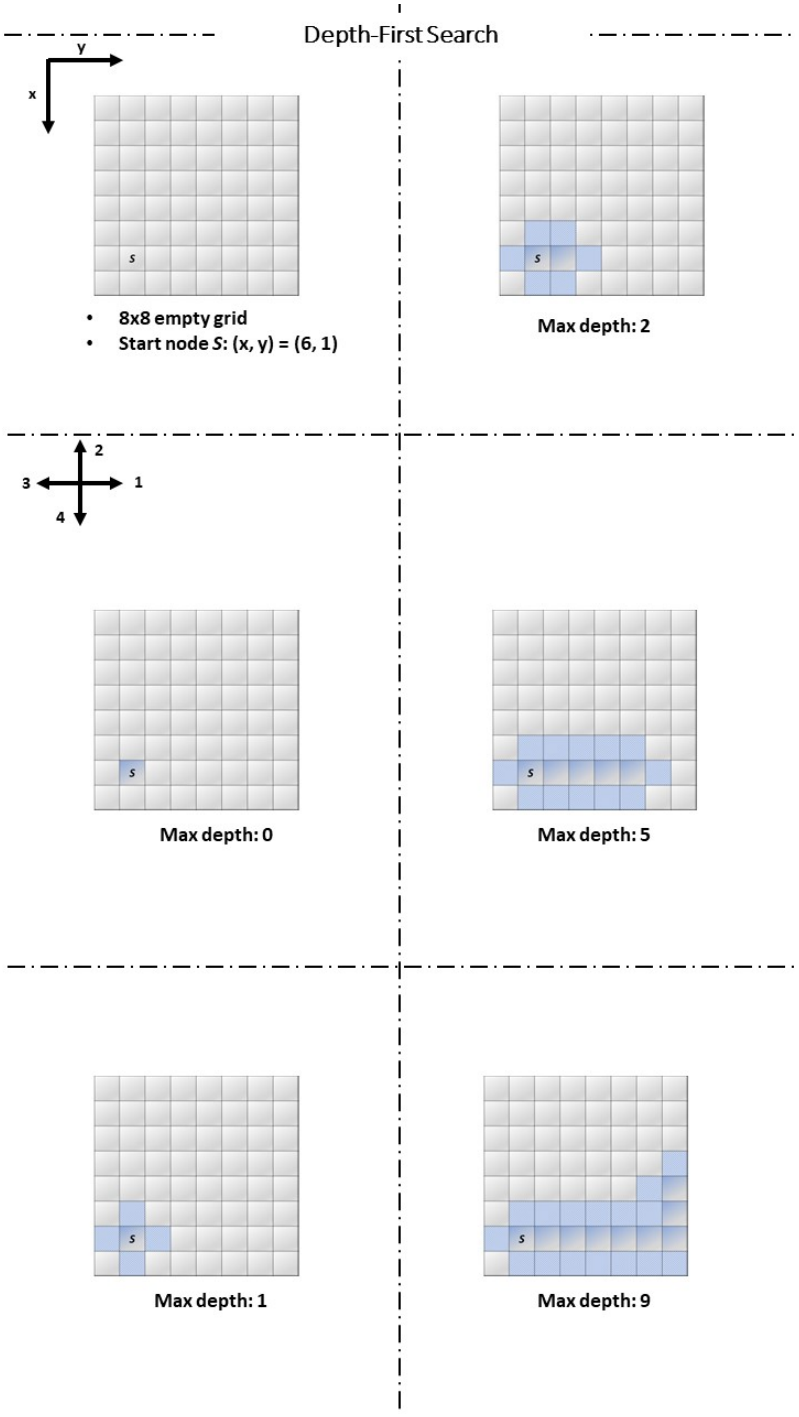
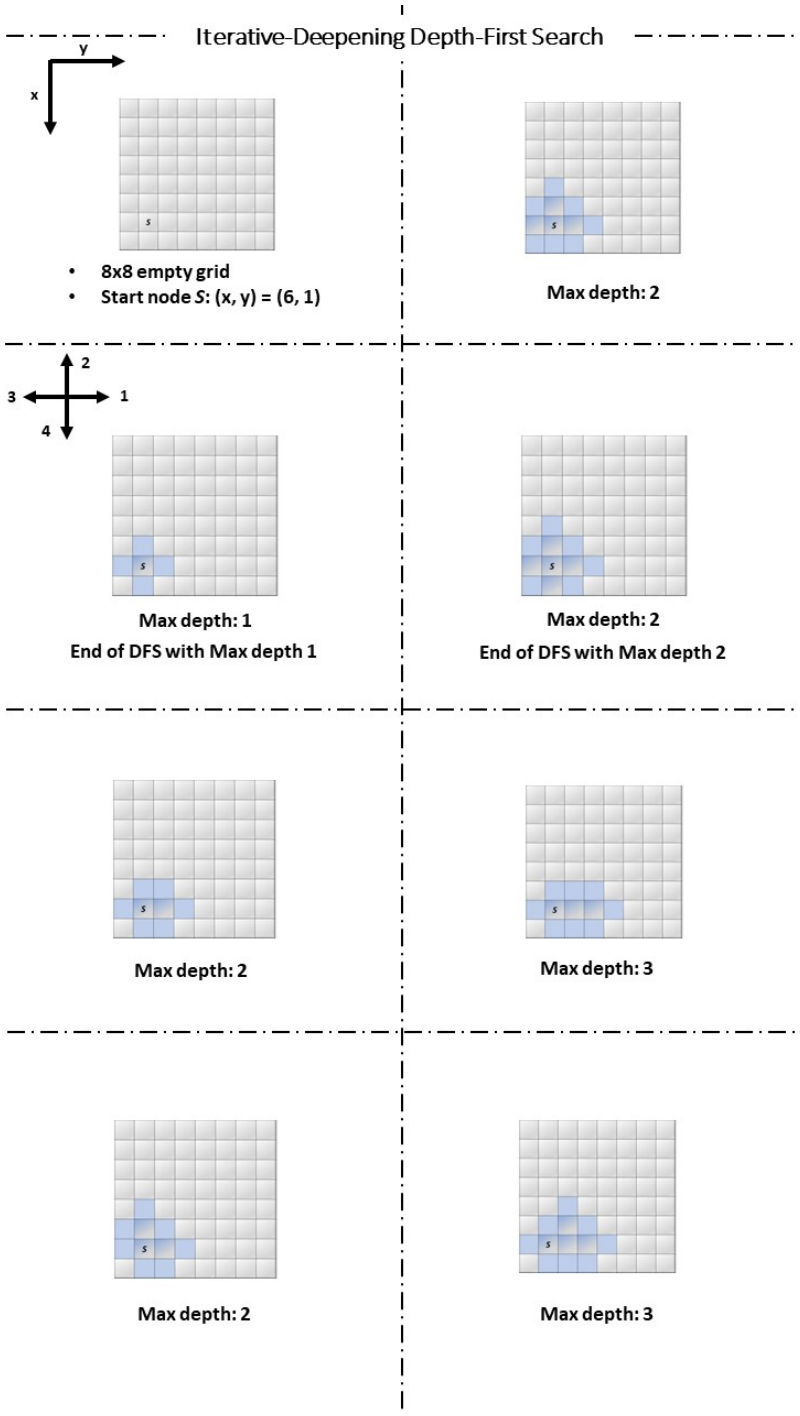
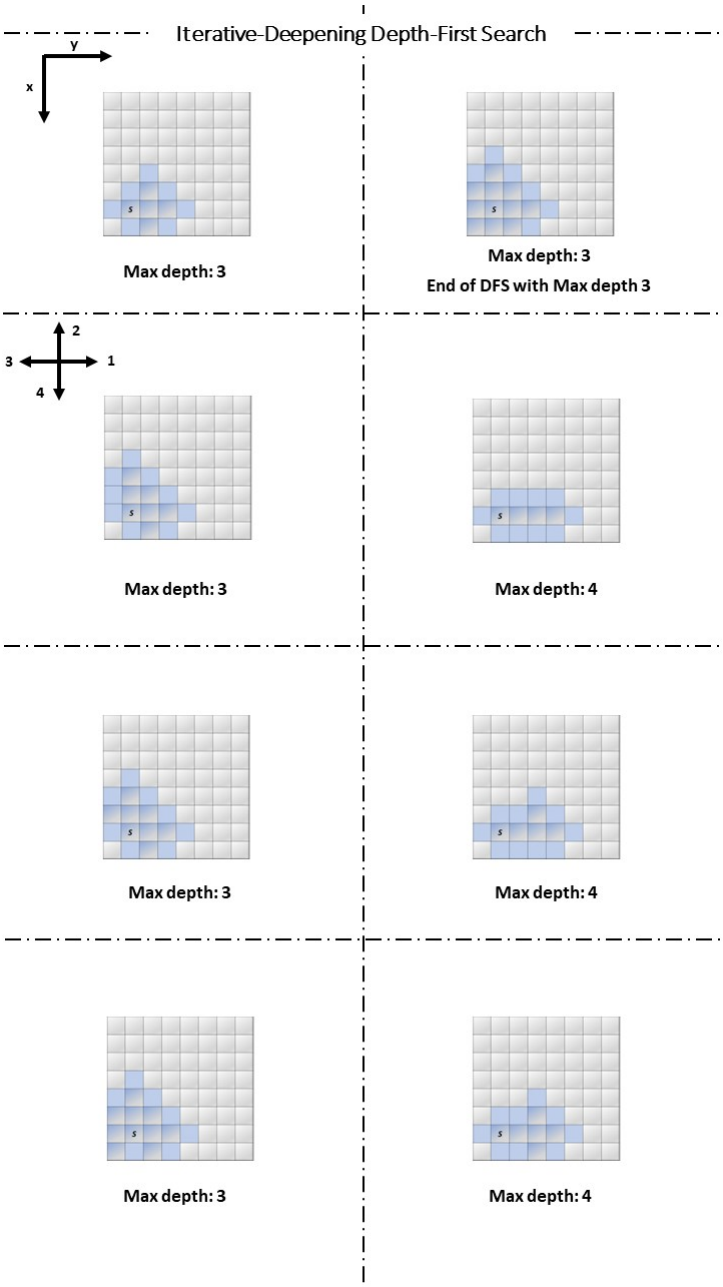



Figure 5.3: Depth-First Search example. The grid is the $8 \times 8$ empty square (16 free tiles). The search starts from node $S$ in position $(x, y) = (6, 1)$. The expanding frontier is made by dashed light-blue nodes. Already visited nodes are represented in shadowed blue color. The order neighbors are visited is counterclockwise.

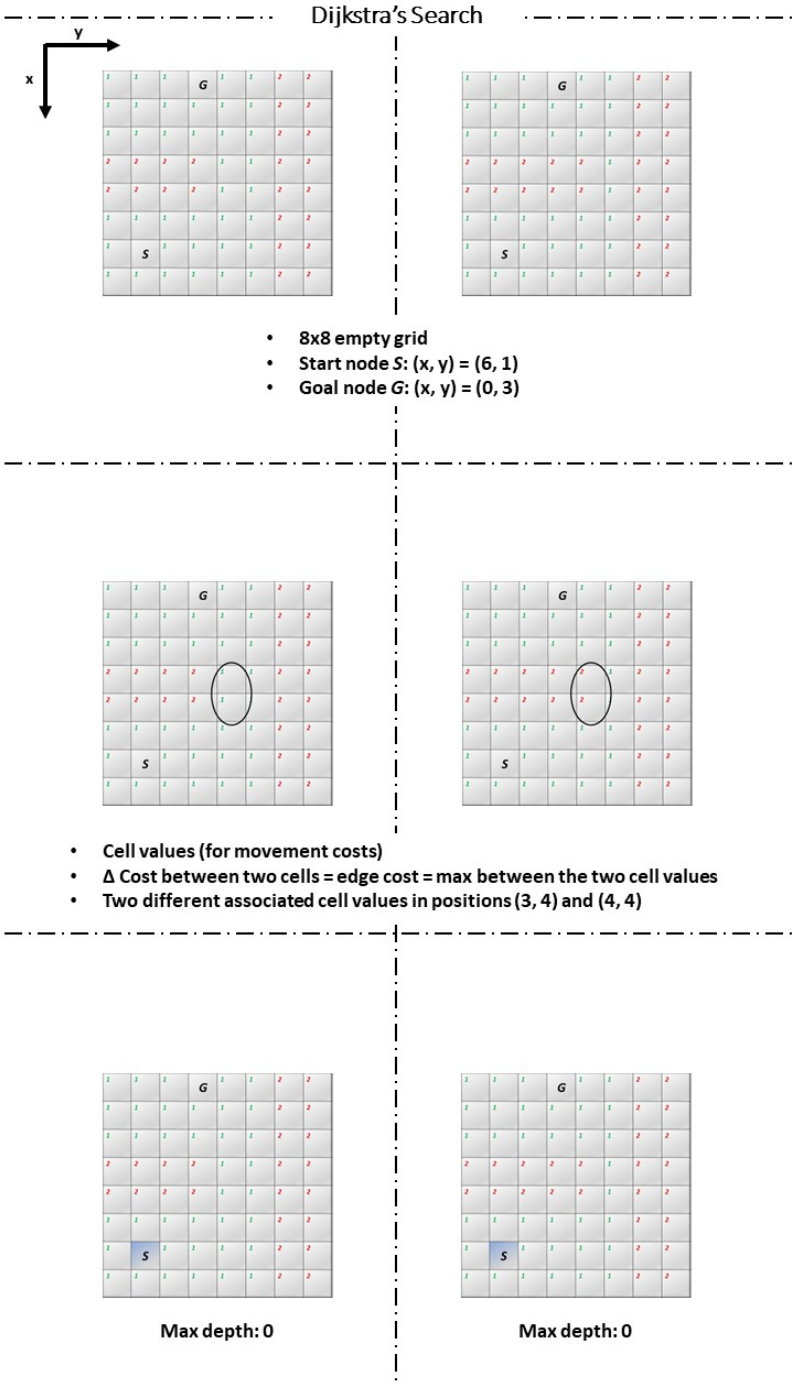

Iterative-Deepening Depth-First Search
y
x
s
8x8 empty grid
Start node S: (x, y) = (6, 1)
Max depth: 2
2
3
1
4
Max depth: 1
End of DFS with Max depth 1
Max depth: 2
End of DFS with Max depth 2
Max depth: 2
Max depth: 3
Max depth: 2
Max depth: 3

(a)

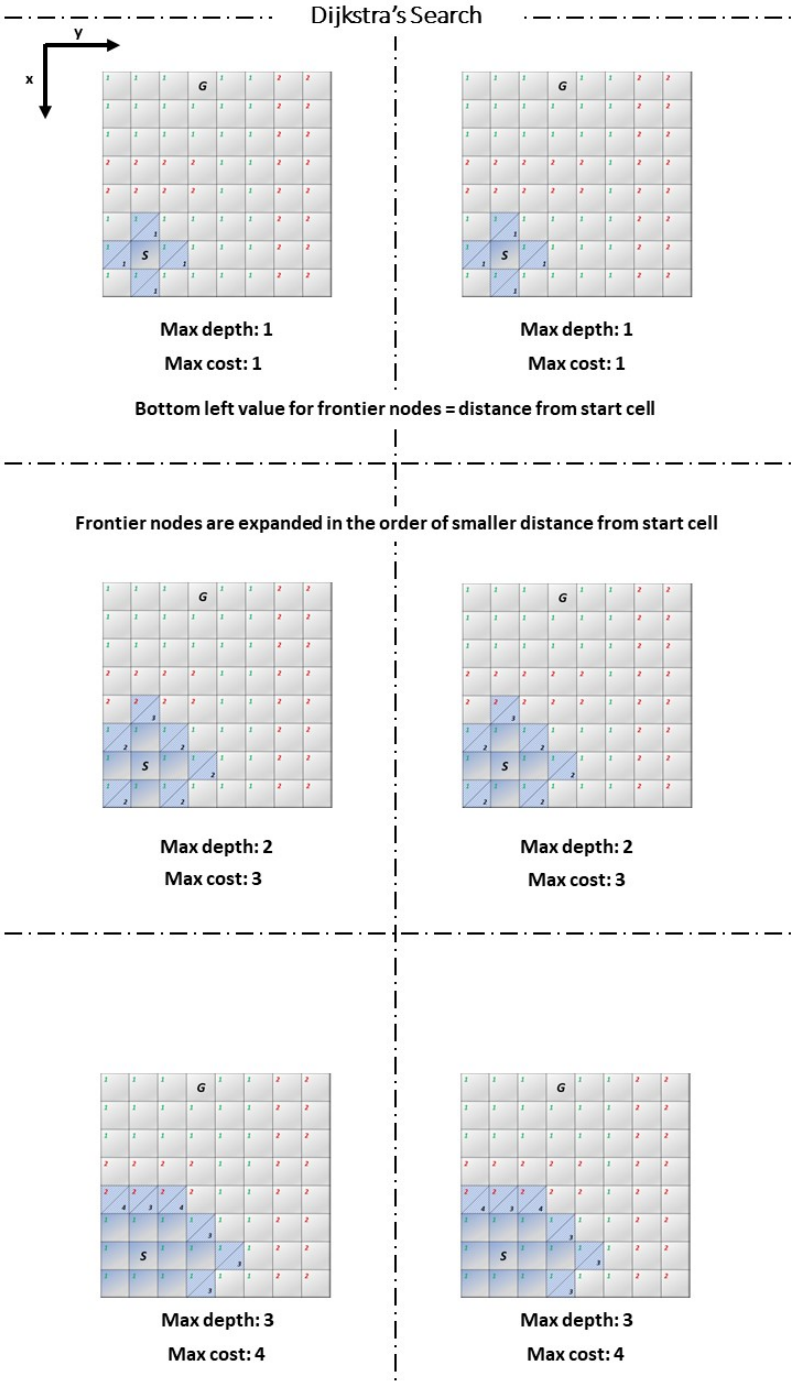



(b)

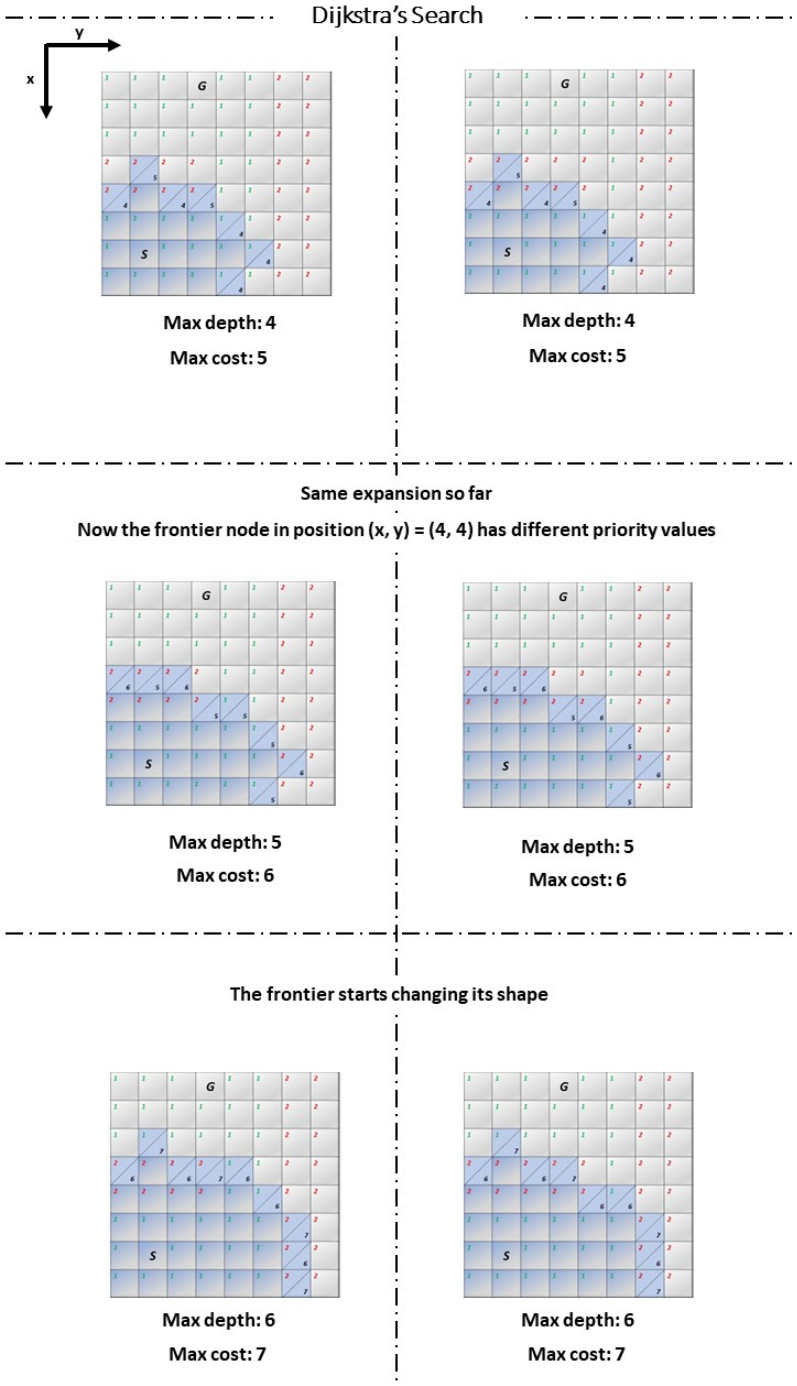
Figure 5.4: Iterative-Deepening Depth-First Search example. The grid is the $8 \times 8$ empty square (16 free tiles). The search starts from node $S$ in position $(x, y) = (6, 1)$. The expanding frontier is made by dashed light-blue nodes. Already visited nodes are represented in shadowed blue color. The order neighbors are visited is counterclockwise.


Dijkstra's Search
y
x
G
S
• 8x8 empty grid
• Start node S: (x, y) = (6, 1)
• Goal node G: (x, y) = (0, 3)
• Cell values (for movement costs)
• Δ Cost between two cells = edge cost = max between the two cell values
• Two different associated cell values in positions (3, 4) and (4, 4)
Max depth: 0
Max depth: 0

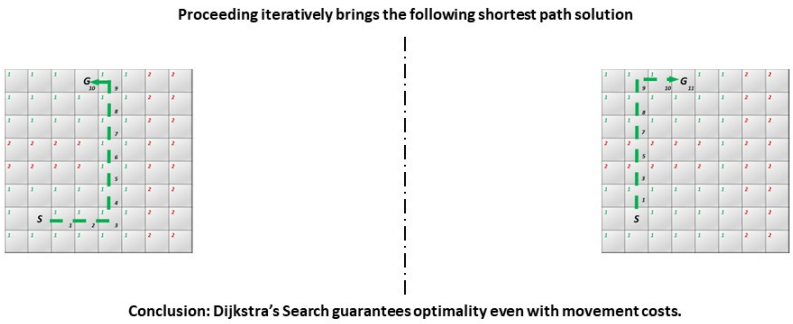
(a)


Dijkstra's Search
y
x
G
S
Max depth: 1
Max cost: 1
Max depth: 1
Max cost: 1
Bottom left value for frontier nodes = distance from start cell
Frontier nodes are expanded in the order of smaller distance from start cell
Max depth: 2
Max cost: 3
Max depth: 2
Max cost: 3
Max depth: 3
Max cost: 4
Max depth: 3
Max cost: 4

(b)

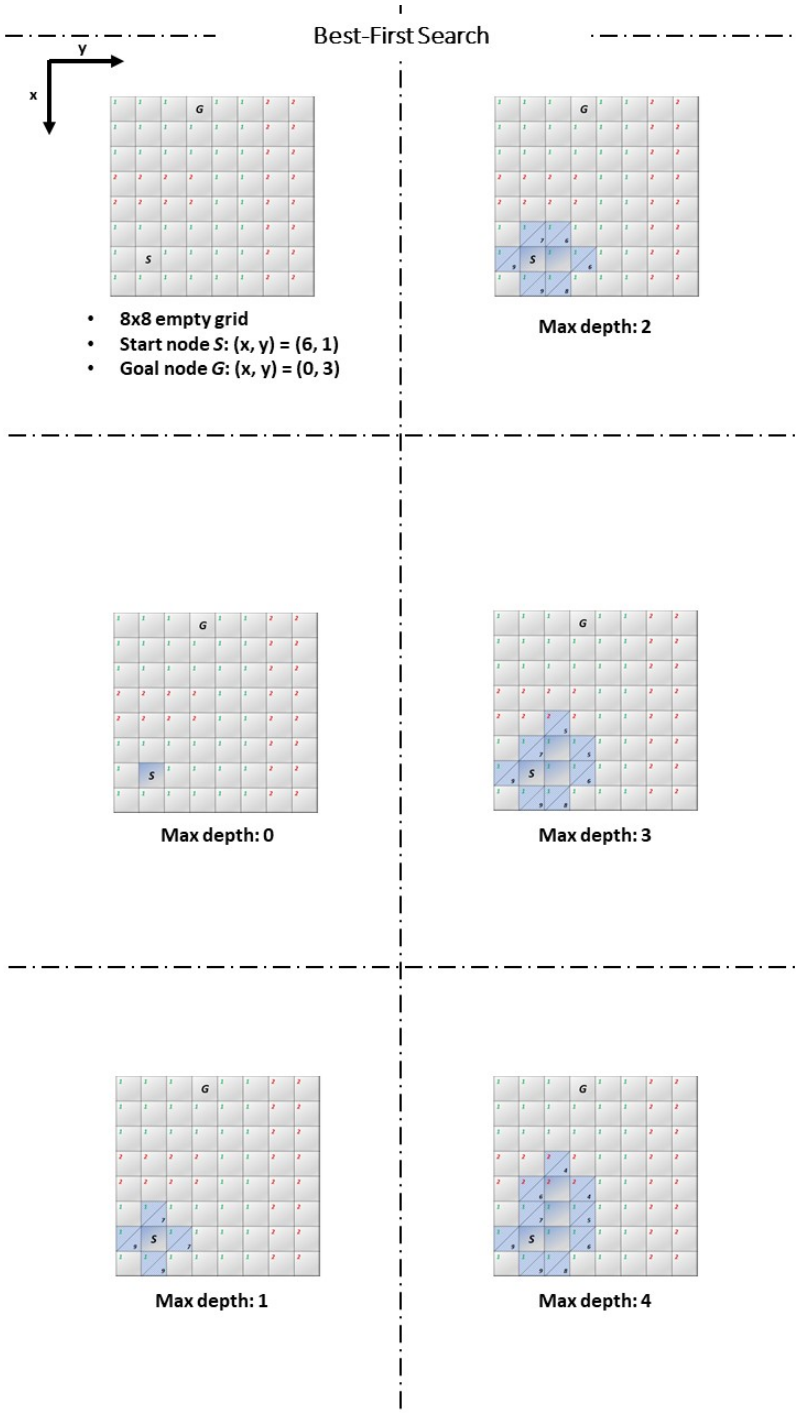

Dijkstra's Search
y
x
G
S
Max depth: 4
Max cost: 5
Max depth: 4
Max cost: 5
Same expansion so far
Now the frontier node in position (x, y) = (4, 4) has different priority values
Max depth: 5
Max cost: 6
Max depth: 5
Max cost: 6
The frontier starts changing its shape
Max depth: 6
Max cost: 7
Max depth: 6
Max cost: 7

(c)

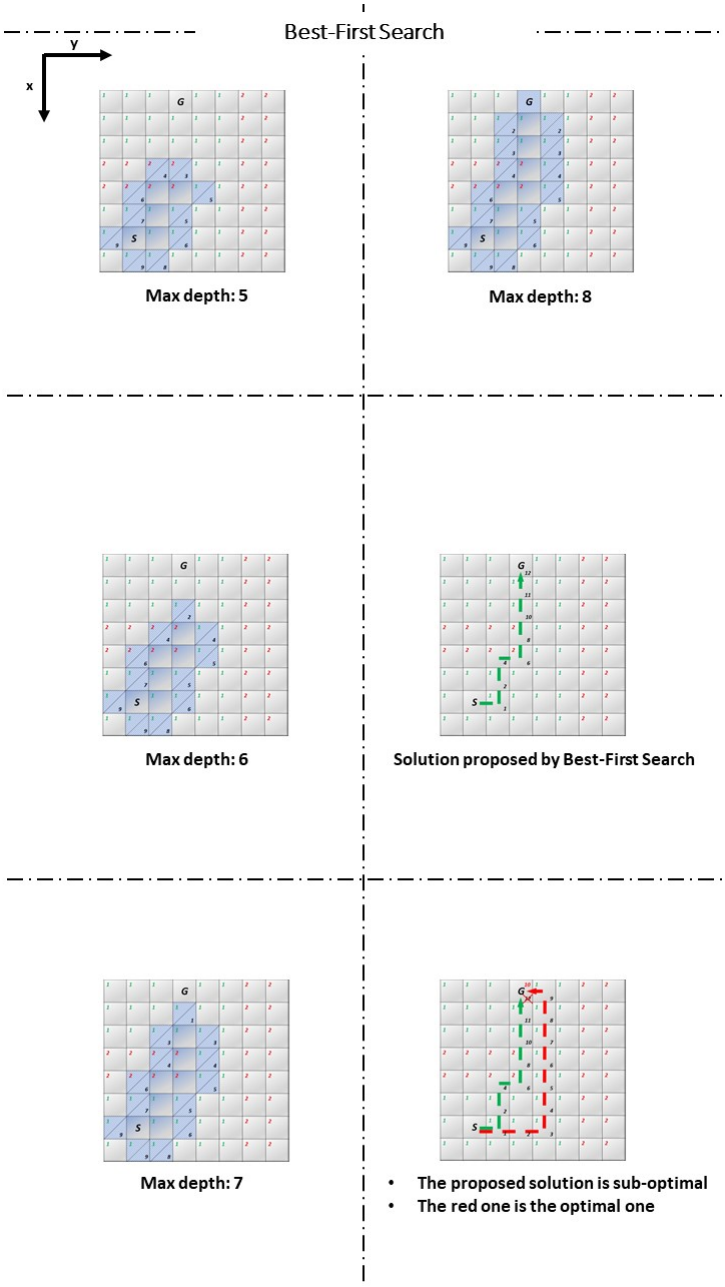



(d)

Figure 5.5: Dijkstra's Search example. The grid is the $8 \times 8$ empty square (16 free tiles). It is not homogeneous. Cell values are specified with green/red values on the left-upper part of each cell. On the right-bottom corner of each vertex, we have the cumulative smallest cost required to move from the origin to that specific node. The search starts from node $S$ in position $(x, y) = (6, 1)$, while the goal $G$ is in position $(x, y) = (0, 3)$. The expanding frontier is made by dashed light-blue nodes. Already visited nodes are represented in shadowed blue color. The green dashed line is the built shortest path.

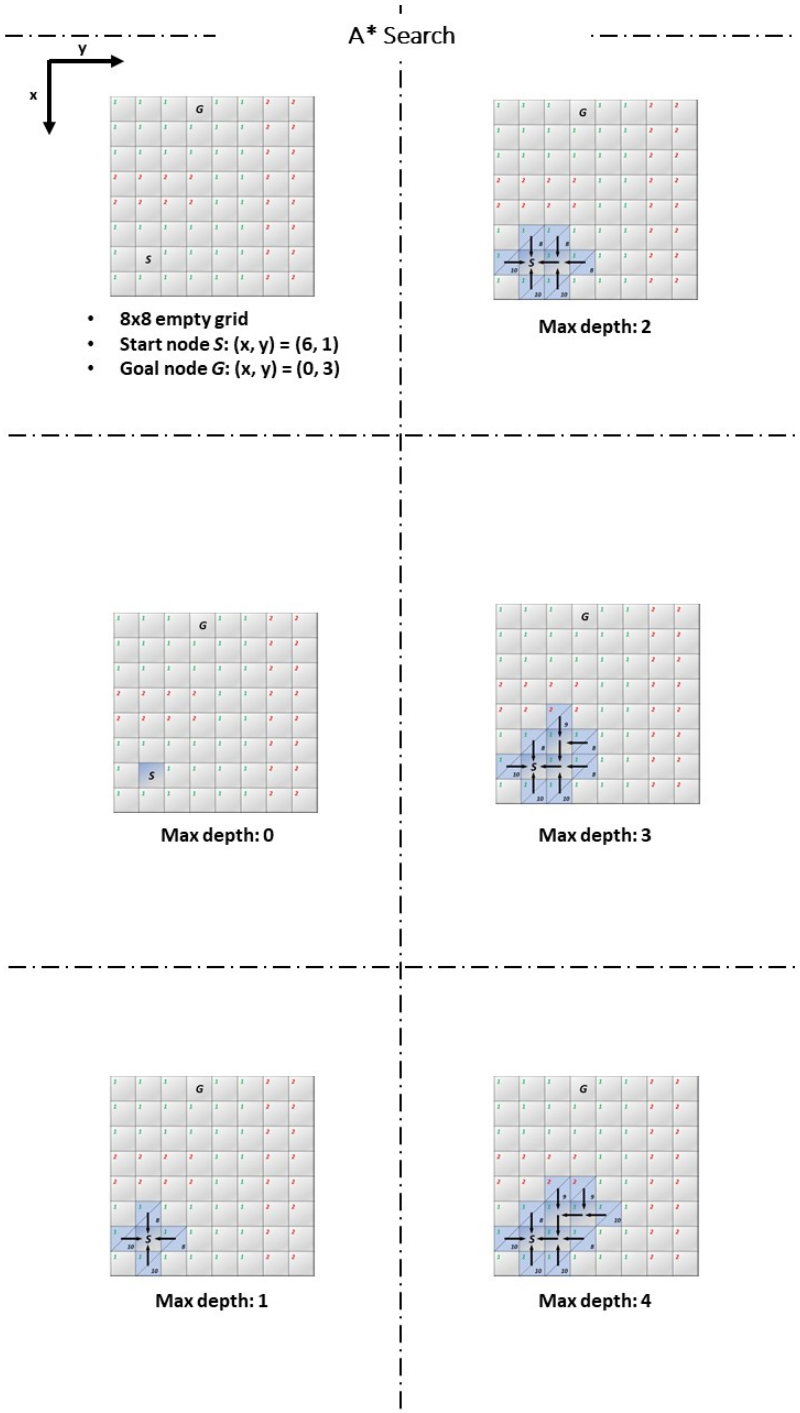

Best-First Search
y
x
G
S
8x8 empty grid
Start node S: (x, y) = (6, 1)
Goal node G: (x, y) = (0, 3)
Max depth: 2
Max depth: 0
Max depth: 3
Max depth: 1
Max depth: 4

(a)

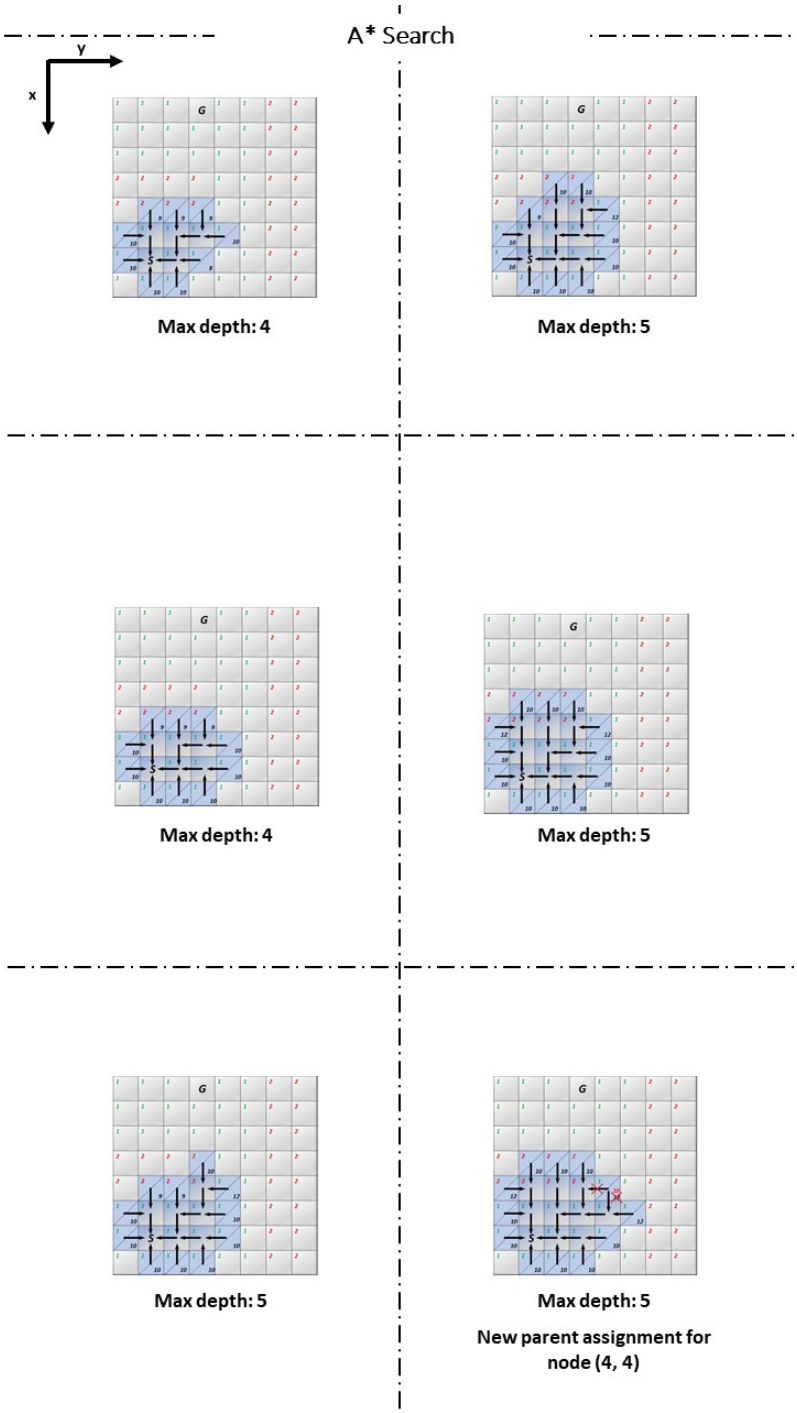



(b)

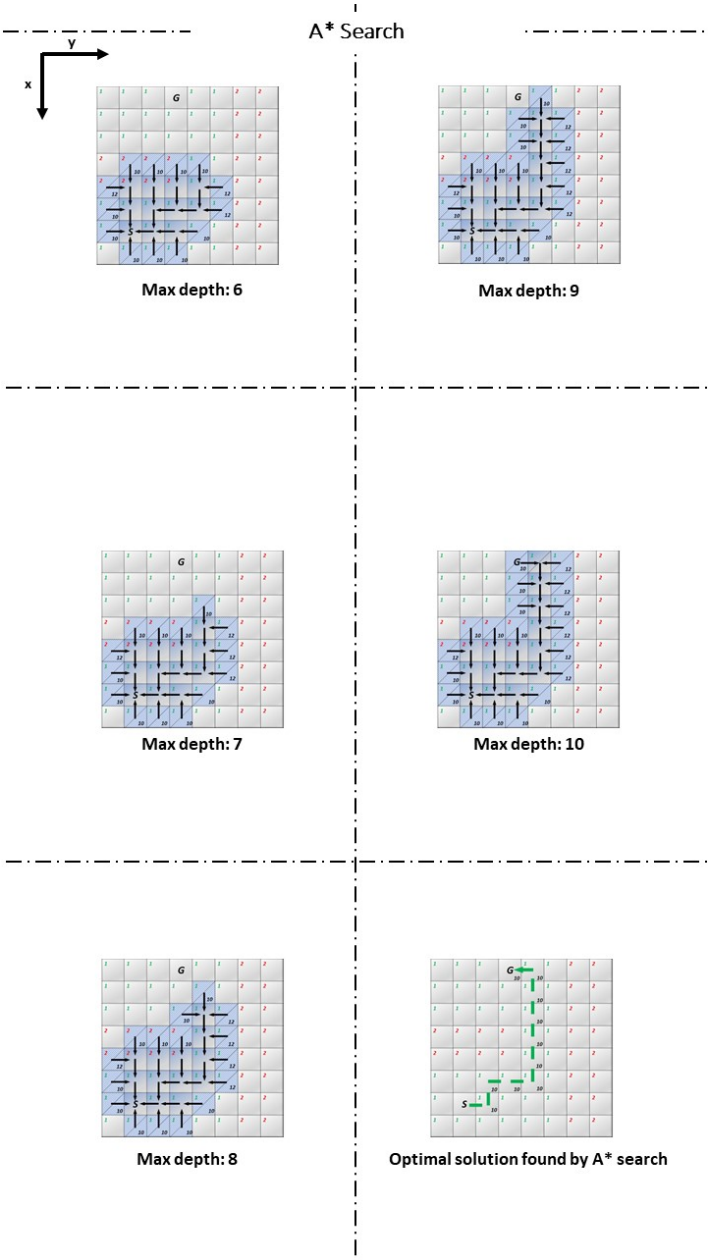
Figure 5.6: Best-First Search example. The grid is the $8 \times 8$ empty square (16 free tiles) with movement costs. Cell values are specified with green/red values on the left-upper part of each cell. On the right-bottom corner of each vertex, we have the estimated distance to the goal. The search starts from node $S$ in position $(x, y) = (6, 1)$ and goes to the destination node $G$ in position $(x, y) = (0,3)$. The expanding frontier is made by dashed light-blue nodes. Already visited nodes are represented in shadowed blue color. The computed path is shown in green dashed line, but the true shortest path is the red one.


A* Search
y
x
G
S
8x8 empty grid
Start node S: (x, y) = (6, 1)
Goal node G: (x, y) = (0, 3)
Max depth: 2
Max depth: 0
Max depth: 3
Max depth: 1
Max depth: 4

(a)


A* Search
y
x
G
S
Max depth: 4
Max depth: 5
Max depth: 4
Max depth: 5
Max depth: 5
Max depth: 5
New parent assignment for node (4, 4)

(b)




(c)

Figure 5.7: A* Search example. The grid is the $8 \times 8$ empty square (16 free tiles) with movement costs. Cell values are specified with green/red values on the left-upper part of each cell. On the right-bottom corner of each vertex, we have the estimated distance to the goal. The search starts from node $S$ in position $(x, y) = (6, 1)$ and goes to the destination node $G$ in position $(x, y) = (0, 3)$. The expanding frontier is made by dashed light-blue nodes. Already visited nodes are represented in shadowed blue color. The computed path is shown in green dashed line. The black arrows represent the pointer to the node's parent. This is useful when finally computing the shortest path.

# Part II

# Developed Algorithms

# Chapter 6

# Decentralized Algorithms

## 6.1 Basic Model Introduction

We distinguish three main elements to develop an algorithm capable of solving the problem of Online MAPF:

- choice of a specific conflict avoidance technique;
- choice of a way to insert new agents in the system without occupying already taken tiles;
- choice of agents' ordering (only during implementation, while in practice, they are processed contemporarily).

Note, it is reasonable to consider the third point for a decentralized/decoupled approach because the algorithm is implemented on a single PC [3], necessarily managing every agent sequentially. This is not a restriction while simulating real multi-agent systems, as long as we guarantee the order avoiding loop pathological agents' behaviors [6.1.1].

Next, we describe all the three above-mentioned steps for our simplest method.

### 6.1.1 Basic Search

#### Conflict Avoidance

The first step we consider for solving our MAPF problem is to choose an appropriate collision avoidance technique. There are many possibilities in the literature.

In our basic search algorithm, we develop a simple version, with a ***random-style collision avoidance***. In few words, when possible conflicts occur, we *randomly* choose one skid. This skid frees its tile to let the other pass on it [1].

[1] We define as ***possible conflicts*** those situations in which two skids are close to generating a conflict, i.e., whenever an agent plans in vain to move to an occupied cell.

More specifically, given two agents, A and B, close to colliding in the next time step, we randomly choose one of them, e.g., agent A; it tries to *move away* to let the other agent pass. If agent A has some available free tile in its neighborhood (i.e., the agent is not blocked), it can move there and let agent B occupy the tile. On the other hand, if A is blocked, then both agents wait. In the case of highly dense areas, this could bring scenarios in which many agents are fixed in their positions for many time steps. In such cases, this problem is solved if we order skids differently at each time step.

Consider the swap conflict avoidance example in Figure 6.1. We just described the right-hand side scenario. While following their shortest path to the corresponding destinations, agents A and B incur into adjacent tiles at time step $T$. To avoid a collision at time $T+1$, our algorithm chooses randomly which agent has to free the tile to let the other skid pass (in other words, which agent will occupy the other agent's cell). At the bottom on the left of the image, agent A passes, while B moves away. B chooses randomly its new position (the left cell in this case). Analogous reasoning is done on the right-hand side.

This is a basic suboptimal strategy since conflicting agents have to modify their path, augmenting the time required to reach their goal. Moreover, the randomness introduces *"unintelligent" skids movements*.

For experimental analysis, the reader has to know that in our developed algorithms, apart from when specifically imposed, every agent replans at each time step. We will see in [6.2.2] that changing this improves the computational cost.

### Adding new online agents to the system

As presented in [1.2.4], MAPF is partitioned into Offline and Online versions. In our setting, we assume new agents may come to the system anytime. This is called *MAPF intersection model* setting.

To simulate randomly arriving skids, we implemented a specific section: when all the agents have chosen the action for a specific time step, we either put or not a new agent in the system. Substantially, a new agent enters the system with a probability extracted from a Bernoulli distribution. Before being physically put into the system, we must check the starting tile is actually unoccupied. From a practical point of view, in this way, we exclude situations where a skid is added to an existing one. If the cell is free, we insert the cart agent, and the new tile is set as occupied, whereas in the other case, the agent is added to a *"waiting" queue*.

Let us focus on a scenario without this strategy (Figure 6.2). Suppose agent C is directly associated with a tile, already occupied by another agent, say A (time $T+1$ in Figure 6.2a). Consider the case where A is processed before C. If A leaves the cell, it will be set as unoccupied. Now, another skid, B, could be processed, after A, before C. B might move to the tile that has just been left by A since it is set to unoccupied. But this would be an error because that cell is occupied by C (time $T+5$ in Figure 6.2b). It is clearly the best choice for agent C to wait to enter the system until the supposed starting

cell is free, to avoid such conflicts.

This check is useful when the grid has many moving agents, with highly congested areas. As soon as a free tile is assigned to the new online agent, it computes its path and starts moving on the grid.

The implementation of this part doesn't affect negatively the search performance. For this reason, during the experimental analysis, we mainly focus on the offline setting. All the developed variant search methods are capable of working both offline and online. The comparison is easier in the offline version. Anyway, we present results with online skids too.

### Agents random ordering

If our search method was just based on random-style collision avoidance (plus the online setting), even in small empty grids, we would have problems when the number of agents increases. This is not caused by the choice of an offline or an online setting, but by the combination of this specific collision avoidance and the sequentiality of the agents during the process.

Suppose we have a highly congested area and consider two adjacent agents, A and B; A has to change its tile but is blocked, while B wants to move to A's tile. A coded loop would imply a fixed ordering while processing agents. If agent A is blocked and waits, the only thing B can do is to wait too. Hence, if A is processed before B, then B is fixed as long as A doesn't move. Even worse, if agent C wants to move to B's position, the queue of waiting agents would be longer (Figure 6.3). Clearly, in real-world applications, this dependency doesn't exist, and, as a consequence, neither these pathologies.

Agents' random ordering at each time step is used to reduce, or even remove, these loops. Before being processed during a specific time step, we randomly permute the ordering of the agents. Again, as done for collision avoidance, we are introducing randomness into our system to remove unwanted situations (these pathological situations may be interpreted as local minima of our MAPF problem).

In conclusion, the permutation of the agents is fundamental to removing pathological situations in which some loops prevent us from finding a solution to MAPF.

### Pseudo Code

Here we show the Pseudo Code 6.1.1 of the described basic search, integrated with Table 6.1, summarizing its main properties.

**Algorithm 3** Basic Search Pseudo Code

```
build the set of obstacles (e.g. walls);
while skids are moving to their destinations do
    if new online skid agent is added to the skids' set then
        if possible, add the mobile agent to the corresponding starting cell;
    end if
    shuffle the agents' order;
    for skid agent = 1, ..., k do
        if the agent has not been considered yet then
            if agent position and goal coincide then
                remove the agent (disappear at target);
            else   (agent not at target → planning)
                compute individual shortest path avoiding walls and consider the next possible position;
                if the next possible tile is occupied then
                    if the other agent has not been considered yet then
                        if possible, one of the two agents is randomly chosen and occupies the other cell (and previous tile becomes blank);
                        if agent position and goal coincide then
                            remove the agent (disappear at target);
                        end if
                    else
                        wait;
                    end if
                else
                    cart agent occupies the available tile and previous tile becomes blank;
                    if agent position and goal coincide then
                        remove the agent (disappear at target);
                    end if
                end if
            end if
        end if
    end for
end while
```

(Note the shortest path is built using A* algorithm)

| File | Generic benchmark file |
|---|---|
| Approach | Online |
| Solution | Sub-Optimal |
| Conflict Avoidance | Random-Style |
| Permutation | Yes |
| Topology Constraints | No |
| Notes | - |

Table 6.1: Basic Search properties summary.

## 6.2 Basic Search Variants

We have shown a simple search to solve Online MAPF instances, *basic search*. Now, we aim to apply some alternative or integrative techniques to this method, obtaining new algorithms. In [7.3] we deeply focus on their comparison. Here, we limit ourselves to giving a detailed description.

The main ideas behind our variants are summarized in:

- intelligent movements exploiting traffic-inspired human interaction and specific rules;
- computational complexity reduction;
- congestion awareness;
- conflict avoidance techniques used to encourage the motion of skids closer/further to their destination;

### 6.2.1 Basic Search + Optimized Pre-Conflict Avoidance

#### Model Description

Many papers in the literature propose some solving techniques directly from the road traffic world. Inspired by agents' movements in road networks, we modified the way skids behave when approaching conflicts, obtaining more intelligent movements, by introducing some traffic rules.

- If the cell the skid plans to occupy is at the moment occupied for a *small amount of time*, then the agent waits. The idea is taken from how drivers behave when road bottlenecks are forming: every driver waits for the others to move on. This waiting time doesn't have to be much if the traffic doesn't flow badly.
- If the cell is occupied for a considerable amount of time, but *different* agents have been occupying it, the skid waits. Consider an intersection between two roads, one highly congested and the other free. A driver on the second road trying to pass the intersection observes a queue of moving cars and waits until they are all passed. This consciousness is fundamental in general to solve vertex conflicts.

- If the cell is occupied for a *significant amount of time* by the *same skid*, then the other waiting agent applies some conflict avoidance technique to move towards its goal. From a road traffic perspective, a driver's motion could be blocked for example by some works on the road. In some way, the car needs to move away.

For this purpose, we introduce a data structure to save the skids *id*s associated with the tile over the last few events (arrivals, departures), not time steps, on each tile. A node updates its movements every time an agent enters or leaves. If nothing happens, the tile doesn't modify anything. This *cell memory* isn't linked with the time steps. In fact, on a specific time step, one tile may modify its history many times (for example, when one agent leaves and another enters)[2].

Hopefully, the agents' motion is clever, at the cost of a higher time spent waiting, increasing the overall sum of costs and makespan.

It is interesting to compare *basic search* and *basic search with optimized pre-conflict avoidance* in terms of the number of avoided conflicts [7.2.2]. We should observe a higher number of avoided conflicts in the second case because an agent close to a possible conflict checks the tile's history and waits for a not negligible amount of time before trying to solve this possible collision. Every time step where the agent is still waiting, the number of possible collisions increments. Hence, while in the basic search we solve the possible conflict immediately, in the optimized approach we correctly count many times some possible conflicts between the same two agents.

### Pseudo Code

In Pseudo Code 6.2.1 and Table 6.2 we show the pseudo code and the summarizing table. (Note the shortest path is built using A* algorithm)

| | |
|---|---|
| File | Generic benchmark file |
| Approach | Online |
| Solution | Sub-Optimal |
| Conflict Avoidance | Random-Style |
| Permutation | Yes |
| Topology Constraints | No |
| Notes | Variant of Basic Search with optimized agents' behavior before conflict avoidance |

Table 6.2: Basic Search + Optimized Pre-Conflict Avoidance properties summary.

[2]This doesn't prevent the algorithm from being a decoupled MAPF solver, since agents are always on their own while planning and moving. The difference is that we assign some common knowledge.

**Algorithm 4** Pseudo Code: Basic Search + Optimized Pre-Conflict Avoidance

```
build the set of obstacles (e.g. walls);
while skids are moving to their destinations do
    if new online skid agent is added to the skids' set then
        if possible, add the mobile agent to the corresponding starting cell;
    end if
    shuffle the agents' order;
    for skid agent = 1, ..., k do
        if the agent has not been considered yet then
            if agent position and goal coincide then
                remove the agent (disappear at target);
            else  (agent not at target → planning)
                compute individual shortest path avoiding walls and consider the next possible position;
                if the next possible tile is occupied then
                    if it is occupied for a small amount of time then
                        wait;
                    else
                        if there is a sequence of moving objects then
                            wait;
                        else
                            if the other agent has not been considered yet then
                                if possible, one of the two agents is randomly chosen and occupies the other cell (and previous tile becomes blank);
                                if agent position and goal coincide then
                                    remove the agent (disappear at target);
                                end if
                            else
                                wait;
                            end if
                        end if
                    end if
                else
                    cart agent occupies the available tile and previous tile becomes blank;
                    if agent position and goal coincide then
                        remove the agent (disappear at target);
                    end if
                end if
            end if
        end if
    end for
end while
```

### 6.2.2 Basic Search + Computational Optimization on Search Calls

#### Model Description

A key element influencing the computational complexity in MAPF solvers is the number of search method calls.

Here, we improve the basic algorithm by reducing the number of search method calls. With basic search we compute the shortest path *at each time step* for each skid, while now the shortest path is computed *only* at the beginning, to let the process start and whenever a possible conflict is solved. Let us see more in detail.

In basic search, every single mobile agent uses the shortest path to decide which is the best neighbor cell to move on. If it is already occupied, then the conflict avoidance technique is taken into account. *Independently* on the occurrence of possible conflicts, in the following time step the same agent would recall the search method to compute the shortest path and determine the best position in its neighborhood for that step, disregarding previous information.
Now, instead, we reduce the calls, speeding up the algorithm, thanks to a data structure memorizing the computed shortest path. The skid checks the next position from this data structure (generally a *list of nodes*). Whenever that position is free, it moves there. This nodes' list is modified only after possible collisions when the agent moves away from its position.

The number of search method calls reduction is impressive as we can see from the experiments in [7.2.3], implying a reduced total CPU time too.

#### Pseudo Code

Comparing Pseudo Code 6.1.1 and Pseudo Code 6.2.2, the difference is clear. In pseudo code 6.1.1, for each time step, for each skid, we compute the shortest path even before deciding whether to move or wait; now we compute only when strictly needed, otherwise, we exploit the path calculated at the previous time step.

In Table 6.3 we consider the summary of this method. Apart from the use of the search function, basic search and this variant are identical.

(Note the shortest path is built using A* algorithm)

**Algorithm 5** Pseudo Code: Basic Search + Computational Optimization on Search Calls

```
build the set of obstacles (e.g. walls);
while skids are moving to their destinations do
    if new online skid agent is added to the skids' set then
        if possible, add the mobile agent to the corresponding starting cell;
    end if
    shuffle the agents' order;
    for skid agent = 1, ..., k do
        if the agent has not been considered yet then
            if agent position and goal coincide then
                remove the agent (disappear at target);
            else  (agent not at target → planning)
                consider the next possible position (from the pre-computed shortest path);
                if the next possible tile is occupied then
                    if the other agent has not been considered yet then
                        if possible, one of the two agents is randomly chosen and occupies the other cell (and previous tile becomes blank);
                        compute the new shortest path (for the involved skids);
                        if agent position and goal coincide then
                            remove the agent (disappear at target);
                        end if
                    else
                        wait;
                    end if
                else
                    cart agent occupies the available tile and previous tile becomes blank;
                    if agent position and goal coincide then
                        remove the agent (disappear at target);
                    end if
                end if
            end if
        end if
    end for
end while
```

| | |
|---|---|
| File | Generic benchmark file |
| Approach | Online |
| Solution | Sub-Optimal |
| Conflict Avoidance | Random-Style |
| Permutation | Yes |
| Topology Constraints | No |
| Notes | Variant of Basic Search with reduced number of search method calls |

Table 6.3: Properties summary of Basic Search + Computational Optimization on Search Calls.

### 6.2.3 Basic Search + Traffic Directions

#### Model Description

As we have done previously for basic search with optimized pre-conflict avoidance [6.2.1], again we take inspiration from the road traffic to build a new version of our algorithm.

The main idea is to impose some agents' motion restrictions on specific grid areas, like one-way streets in road networks. In this work, we study the effect of restricting horizontal movements: for some grid rows, cart agents can move only in the positive direction, while for other rows they move oppositely, in the negative direction. No constraint is applied for up and down movements. Around fixed obstacles, we impose no restrictions (apart from the motion on the obstacle cell). In other words, we modified the grid's structure in terms of admissible movements.

To better understand this, we plot and show in Figure 6.4 the resulting grid, starting from our custom map (Figure 4.4). Cyan rows correspond to those rows in which it is *NOT* possible to move *left*, while in fuchsia rows oppositely *right* moves are *NOT* possible. Tiles around fixed obstacles are yellow-colored: agents can move in any direction except those pointing to the fixed obstacle.

From the implementation point of view, this implies just some modification in the definition of nodes' neighborhoods.

These constraints can be modified for the specific grid and application. In Figure 6.4 we impose the verse to alternate at each row. But other choices are possible, also in the vertical direction. Figures 6.5 and 6.6 show the grid for two different constraints on the admissible movements. Figure 6.5 represents our custom grid when we impose that couples of rows admit the same movements. Instead, Figure 6.6 shows the grid when groups of four rows have the same movement constraints.

We expect a more clever motion of the skids, but constraining the topology of this specific graph is not good from the overall cost point of view. Notice the similarity and the differences between *basic search + optimized pre-conflict avoidance* and *basic search + traffic directions*.

- Both the algorithms aim at improving the intelligence of skids while moving in crowded systems, at the cost of a higher computational cost and sum of time steps.
- In the first case we consider a restriction of the agents' actions, while in the second method the restriction is imposed on the topology itself.

In general, by imposing restrictions on systems, some problems arise. Agents find more difficulties in moving towards their goals, raising the maximum number of time steps and the time cost. In [8.2.1] we check this.

In [1.2.4], we said in realistic situations, such as fulfillment centers, agents should be capable of going from their origin to many destinations, such as charging areas or new shelves. A mobile-robots automated warehouse is generally an extended platform with different locations for different jobs. Some areas are specific for *pick* & *place*, while somewhere else these vehicles are charged. In that case, it is useful to introduce movement constraints to separate those robots heading to different areas.

### Pseudo Code

The pseudo code of the method is shown in 6.2.3, while the summary table is in 6.4.
(Note the shortest path is built using A* algorithm)

| | |
|---|---|
| File | Generic benchmark file |
| Approach | Online |
| Solution | Sub-Optimal |
| Conflict Avoidance | Random-Style |
| Permutation | Yes |
| Topology Constraints | Some cells restrict the possible movements |
| Notes | Variant of Basic Search with traffic directions |

Table 6.4: Properties summary of Basic Search + Traffic Directions.

**Algorithm 6** Pseudo Code: Basic Search + Traffic Directions

```
build the set of obstacles (e.g. walls);
IF DIRECTION CONSTRAINTS ARE IMPOSED, BUILD APPROPRIATELY THE RESTRICTIONS ON THE GRID;
while skids are moving to their destinations do
    if new online skid agent is added to the skids' set then
        if possible, add the mobile agent to the corresponding starting cell;
    end if
    shuffle the agents' order;
    for skid agent = 1, ..., k do
        if the agent has not been considered yet then
            if agent position and goal coincide then
                remove the agent (disappear at target);
            else  (agent not at target → planning)
                compute individual shortest path (considering direction constraints) avoiding walls and consider the next possible position;
                if the next possible tile is occupied then
                    if the other agent has not been considered yet then
                        if possible, one of the two agents is randomly chosen and occupies the other cell (with no restriction on direction) (and previous tile becomes blank);
                        if agent position and goal coincide then
                            remove the agent (disappear at target);
                        end if
                    else
                        wait;
                    end if
                else
                    cart agent occupies the available tile and previous tile becomes blank;
                    if agent position and goal coincide then
                        remove the agent (disappear at target);
                    end if
                end if
            end if
        end if
    end for
end while
```

### 6.2.4 Basic Search + Congestion Awareness

#### Model Description

When developing a MAPF solver we have to pay particular attention to the possible formation of highly populated areas. In too highly congested areas, the search may bring deadlocks and no progress.

In this enhancement of our basic method, agents try to pass through less uncongested zones. We talk about ***congestion awareness***. To model congestion, we memorize the areas where every skid's path passes and let agents know this when computing their shortest path. The grid nodes are grouped in ***macro cells***, and the corresponding grid of macro cells is called ***macro grid***. Each macro cell has a specific positive cost. Starting from a homogeneous macro grid, every mobile agent computes its shortest path using the costs of the macro grid. Then it updates the costs of the macro cells: for those macro cells through which the path passes, we add a $+1$, representing the marginal congestion contribution. Other agents then try to follow alternative routes along which the overall cost is smaller. If we proceed in this way for all the processes, the values of the macro cells will explode. To avoid this, every time an agent is processed again, it first removes its previously inflicted costs on the macro cells by applying a $-1$ and recomputes its path, adding $+1$ in the corresponding macro cells. If the collision avoidance technique causes a skid movement, then we don't modify the macro grid.

In some sense, *skids look for a trade-off between maximizing the distance from congested areas and minimizing the total travel time and cost.*

With congestion awareness, skids tend to visit less *"expensive"*, i.e., less crowded areas. Agents tend to encounter each other less than in the case without congestion awareness. Hence, by comparing this enhancement with the basic search [7.2.5], we expect that for the same number of agents, this second version gives a smaller number of conflicts. Moreover, the number of time steps should be lower since, when agents deal with possible conflicts, they unavoidably add time steps to reach their destination.

### Pseudo Code

Here we show the pseudo code 6.2.4 of this variant and its summary in Table 6.5.

**Algorithm 7** Pseudo Code: Basic Search + Congestion Awareness

```
build the set of obstacles (e.g. walls);
while skids are moving to their destinations do
    if new online skid agent is added to the skids' set then
        if possible, add the mobile agent to the corresponding starting cell;
    end if
    shuffle the agents' order;
    for skid agent = 1, ..., k do
        if the agent has not been considered yet then
            if agent position and goal coincide then
                remove the agent (disappear at target);
                modify the grid weights;
            else   (agent not at target → planning)
                modify the grid weights, compute individual shortest path avoiding walls and consider the next possible position;
                if the next possible tile is occupied then
                    if the other agent has not been considered yet then
                        if possible, one of the two agents is randomly chosen and occupies the other cell (and previous tile becomes blank);
                        if agent position and goal coincide then
                            remove the agent (disappear at target);
                            modify the grid weights;
                        end if
                    else
                        wait;
                    end if
                else
                    cart agent occupies the available tile and previous tile becomes blank;
                    if agent position and goal coincide then
                        remove the agent (disappear at target);
                        modify the grid weights;
                    end if
                end if
            end if
        end if
    end for
end while
```

(Note the shortest path is built using A* algorithm)

| File | Generic benchmark file |
|---|---|
| Approach | Online |
| Solution | Sub-Optimal |
| Conflict Avoidance | Random-Style |
| Permutation | Yes |
| Topology Constraints | No |
| Notes | Congestion awareness through a macro grid |

Table 6.5: Properties summary of Basic Search + Congestion Awareness.

### 6.2.5 Basic Search + New Conflict Avoidance based on Agents' Distance from Origin/Destination (first version: Destination)

#### Model Description

As previously observed, there are many ways to manage possible conflicts, not necessarily based on *randomness*. We consider two different versions of the basic search where the random-style collision avoidance is substituted by another one considering the *estimated distance* (*closeness* in one version, *remoteness* in the other) from the destination of agents. For all previously developed algorithms, conflicts are solved by *randomly* choosing which one of the two agents to pass. No attention is given to the position of the agent on its path. Now, we remove some randomness by imposing the closest or furthest agent to pass, depending on the version. In the *first version*, we suppose the closest to be the chosen one. In some sense, this implies the agents tend to become even *more selfish while arriving close to their destination*. If the distance is the same, we proceed randomly. In the *second version*, the furthest will be selected. In this case, the agents tend to become *more generous while approaching their target node*. It is interesting to compare the two performances [7.2.6].

With respect to the *basic search*, the difference in the code is now we have a *deterministic* check for choosing the passing agent, instead of a *random* one. Even though this only implies a small difference from the implementation point of view, during experimental analysis we confirm that by removing randomness, we are substantially *constraining* the agents' motion, easily bringing undesirable results.

These presented conflict avoidance techniques work well in the case of *swapping collisions*. But generally, this is not always the case. There may be situations where one of the two involved agents has no intention to occupy the other cell. Hence, if randomly chosen, the agent would occupy the other tile even though it is not in his path. In this sense, guaranteeing one agent passes doesn't imply it gets closer to its goal. The first version favors the agents closer to their destination. This may be useful if we consider disappearing at target since substantially we are pushing agents to conclude their path.

### Pseudo Code

In 6.2.5 and 6.2.5, we show the pseudo code for the first and the second version respectively, while the summaries are in Tables 6.6 and 6.7.

**Algorithm 8** Pseudo Code: Basic Search + New Conflict Avoidance(1)

```
build the set of obstacles (e.g. walls);
while skids are moving to their destinations do
    if new online skid agent is added to the skids' set then
        if possible, add the mobile agent to the corresponding starting cell;
    end if
    shuffle the agents' order;
    for skid agent = 1, ..., k do
        if the agent has not been considered yet then
            if agent position and goal coincide then
                remove the agent (disappear at target);
            else   (agent not at target → planning)
                compute individual shortest path avoiding walls and consider the next possible position;
                if the next possible tile is occupied then
                    if the other agent has not been considered yet then
                        if possible, between the two agents, the one CLOSER to its destination occupies the other cell (and previous tile becomes blank); if the distance is the same, proceed randomly;
                        if agent position and goal coincide then
                            remove the agent (disappear at target);
                        end if
                    else
                        wait;
                    end if
                else
                    cart agent occupies the available tile and previous tile becomes blank;
                    if agent position and goal coincide then
                        remove the agent (disappear at target);
                    end if
                end if
            end if
        end if
    end for
end while
```

(Note the shortest path is built using A* algorithm)

**Algorithm 9** Pseudo Code: Basic Search + New Conflict Avoidance(2)

```
build the set of obstacles (e.g. walls);
while skids are moving to their destinations do
    if new online skid agent is added to the skids' set then
        if possible, add the mobile agent to the corresponding starting cell;
    end if
    shuffle the agents' order;
    for skid agent = 1, ..., k do
        if the agent has not been considered yet then
            if agent position and goal coincide then
                remove the agent (disappear at target);
            else  (agent not at target → planning)
                compute individual shortest path avoiding walls and consider the next possible position;
                if the next possible tile is occupied then
                    if the other agent has not been considered yet then
                        if possible, between the two agents, the one FURTHER to its destination occupies the other cell (and previous tile becomes blank); if the distance is the same, proceed randomly;
                        if agent position and goal coincide then
                            remove the agent (disappear at target);
                        end if
                    else
                        wait;
                    end if
                else
                    cart agent occupies the available tile and previous tile becomes blank;
                    if agent position and goal coincide then
                        remove the agent (disappear at target);
                    end if
                end if
            end if
        end if
    end for
end while
```

| File | Generic benchmark file |
|---|---|
| Approach | Online |
| Solution | Sub-Optimal |
| Conflict Avoidance | Deterministic based on destination distance |
| Permutation | Yes |
| Topology Constraints | No |
| Notes | Deterministic collision avoidance |

Table 6.6: Properties summary of Basic Search + new conflict avoidance based on agent's distance from destination.

| File | Generic benchmark file |
|---|---|
| Approach | Online |
| Solution | Sub-Optimal |
| Conflict Avoidance | Deterministic based on origin distance |
| Permutation | Yes |
| Topology Constraints | No |
| Notes | Deterministic collision avoidance |

Table 6.7: Properties summary of Basic Search + new conflict avoidance based on agent's distance from the origin.

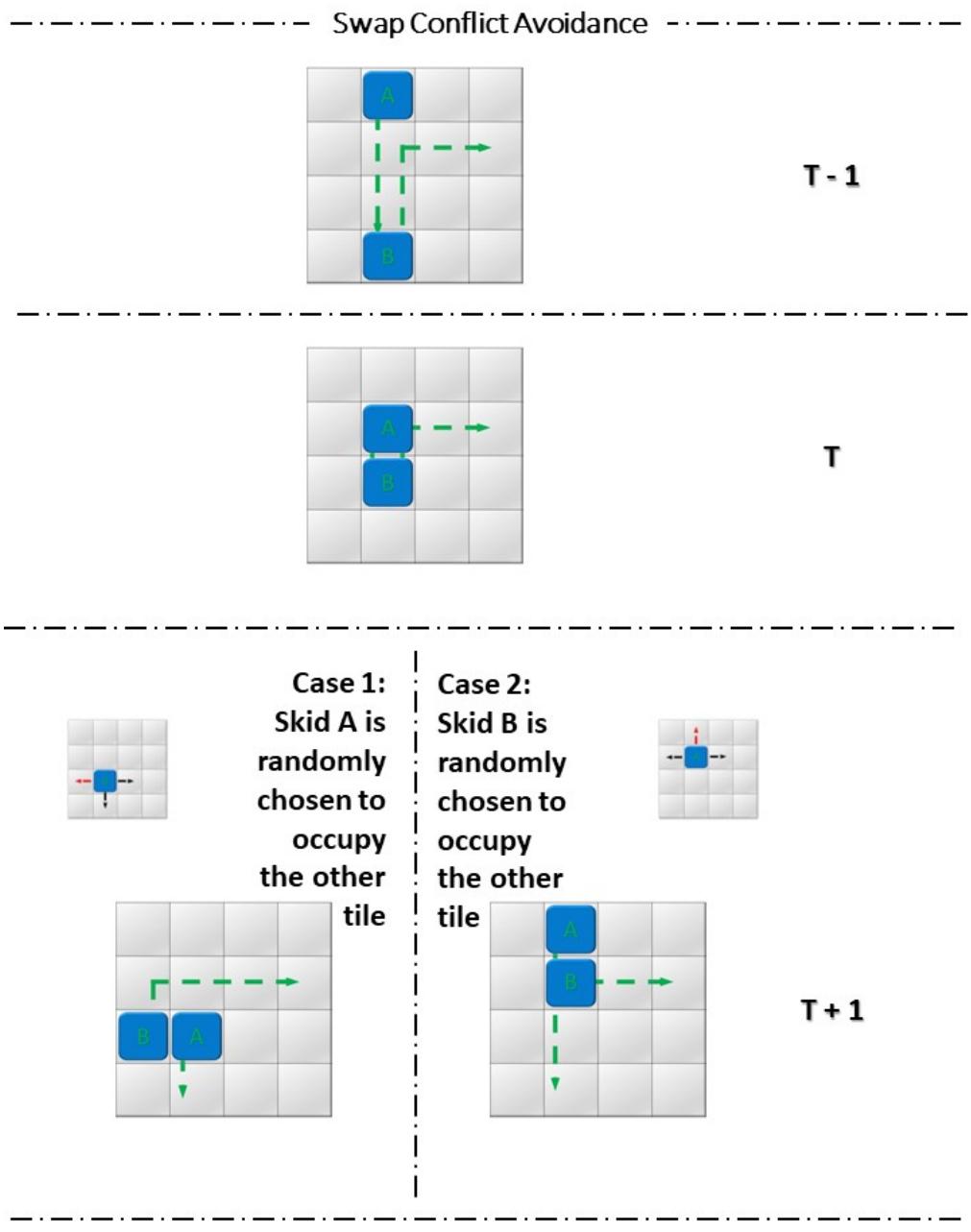
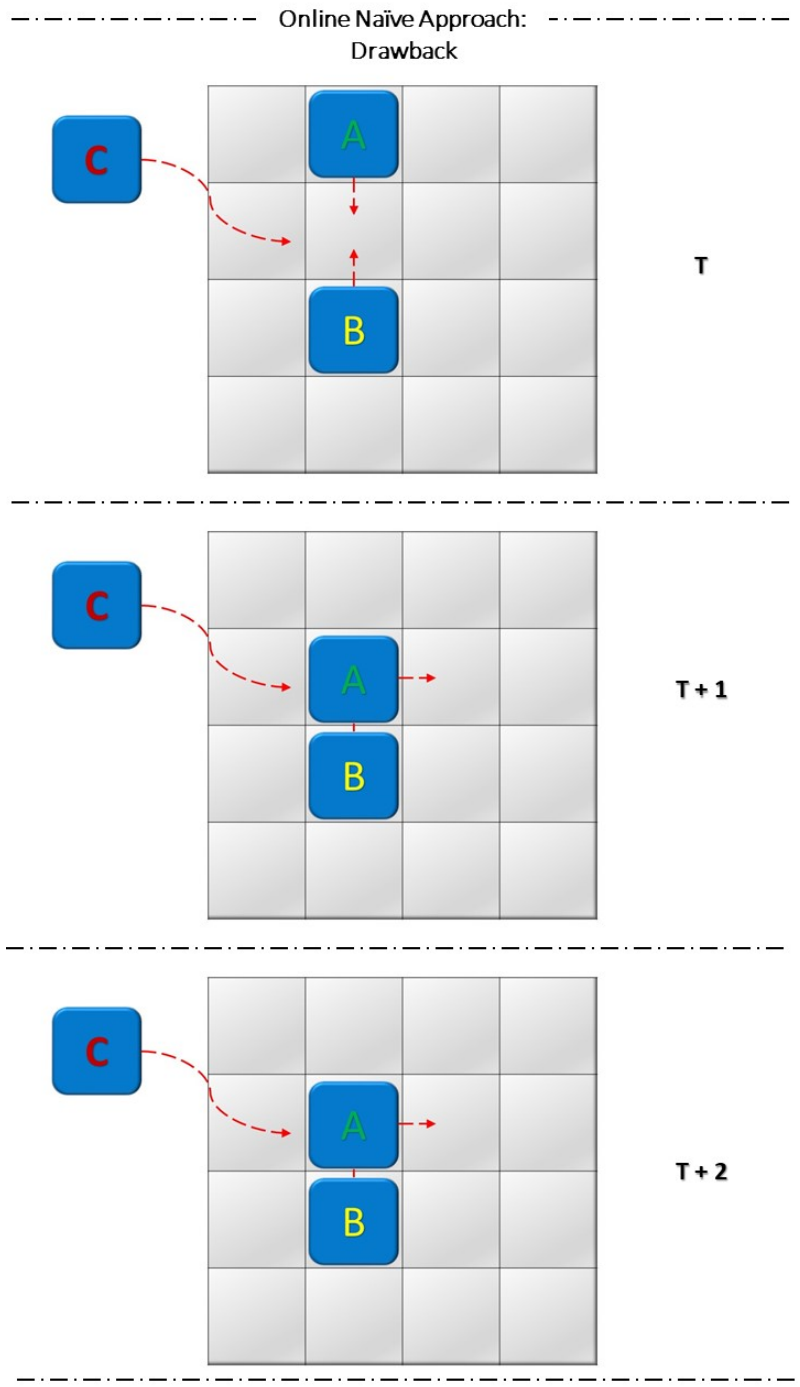
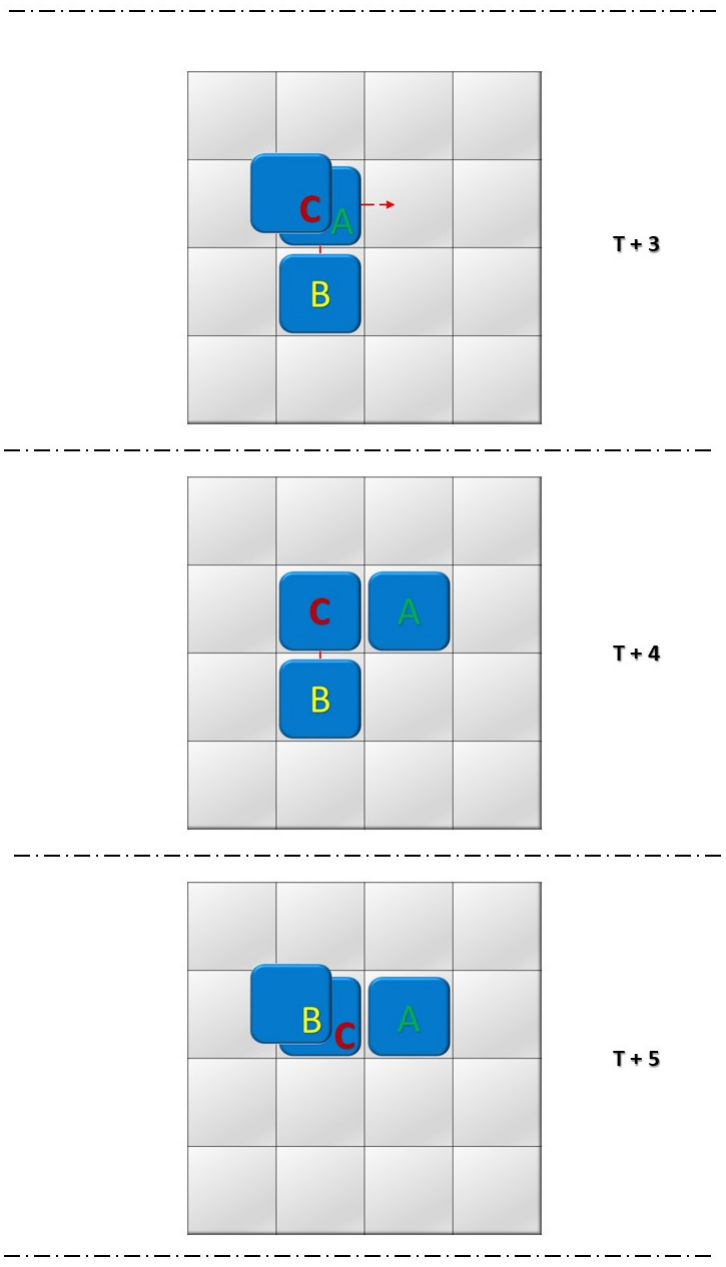



Figure 6.1: Swap conflict avoidance example. The grid is the $4 \times 4$ square (16 tiles). Skids are the blue squares (A and B). Individual planned shortest paths are indicated with green arrows. Black arrows are for the possible moves the agent has to apply to let the other pass; the red one is randomly chosen.




(a)




(b)

Figure 6.2: Online Naïve Approach: Drawback. The grid is the $4 \times 4$ square (16 tiles). Skids are the blue squares (A, B, C). Individual planned shortest paths are indicated with red arrows.

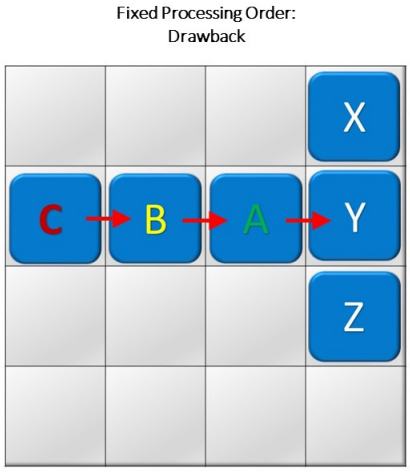



Figure 6.3: Fixed Processing Order: Drawback. The grid is the $4 \times 4$ square (16 tiles). Skids are the blue squares (A, B, C, X, Y, Z). X, Y, Z are blocked for some reason. In order, A, B, C want to move (individual planned shortest paths are indicated with red arrows). Due to the fixed processing order, they are blocked consequently.

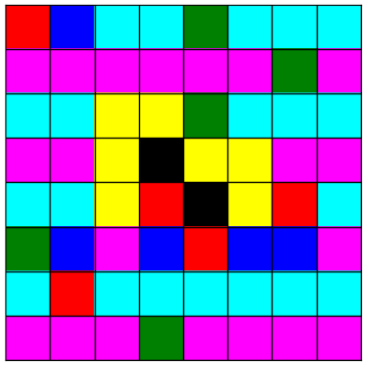


Figure 6.4: Traffic directions on Custom Benchmark Grid. Cyan rows correspond to those rows in which it is *NOT* possible to move *left*, while in fuchsia rows oppositely *right* moves are *NOT* possible. Tiles around fixed obstacles (black) are yellow-colored: agents can move in any direction except those pointing to the fixed obstacle. Red cells and green cells indicate, respectively, start and goal nodes. Blue cells correspond to skids' positions.

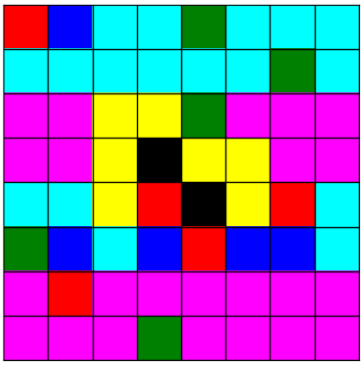


Figure 6.5: Traffic directions on Custom Benchmark Grid. Cyan rows correspond to those rows in which it is *NOT* possible to move *left*, while in fuchsia rows oppositely *right* moves are *NOT* possible. Tiles around fixed obstacles (black) are yellow-colored: agents can move in any direction except those pointing to the fixed obstacle. Red cells and green cells indicate, respectively, start and goal nodes. Blue cells correspond to skids' positions.

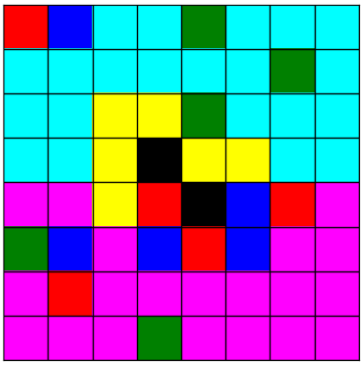


Figure 6.6: Traffic directions on Custom Benchmark Grid. Cyan rows correspond to those rows in which it is *NOT* possible to move *left*, while in fuchsia rows oppositely *right* moves are *NOT* possible. Tiles around fixed obstacles (black) are yellow-colored: agents can move in any direction except those pointing to the fixed obstacle. Red cells and green cells indicate, respectively, start and goal nodes. Blue cells correspond to skids' positions.

# Chapter 7

# Experimental Analysis

Previously, we described the evolution of MAPF solvers and their main properties. Then we presented our developed algorithms for solving MAPF from a self-interested or slightly cooperative point of view.

In the following, we analyze in detail the simulation results. For every algorithm, we study the performance on a $48 \times 48$ empty grid map. Then we check if patterns repeat on different benchmark grids [4.1]. All of this is done exploiting graphical representations using `MATLAB`, `R`, and `PowerPoint`, where we define the performance in terms of:

- number of avoided conflicts;
- number of A* search method calls;
- sum of time steps / sum of costs;
- maximum number of required steps / maximum cost;
- Speed and time consumption.

After this, we compare the algorithms, determining effects, pros, and cons. We combine the best techniques into an improved version of the basic search.

## 7.1 Performance measures

### 7.1.1 Number of avoided conflicts

It is clear now that the first objective of a multi-agent pathfinding algorithm is to find a way to bring each agent to its destination without interfering with the other's path. But if our method routes skids in a way that significantly reduces the number of possible conflicts, then we have two main advantages:

- the total computation required to escape conflicting situations is dramatically reduced;
- consequently, even a low-quality conflict avoidance policy would be usable.

We exploit this note when considering congestion awareness.
For these reasons, during our experimental analysis, we pay particular attention to the number of conflicts avoided by the algorithm.

Oppositely to the self-interested setting, with cooperation, there are no techniques to deal with possible conflicts. The cooperative approach itself avoids conflicts; this is not done locally. For this reason, in cooperative pathfinding, we don't consider this variable [8.2.2].

### 7.1.2 A* search method calls

Even before considering a collision avoidance technique, we need a way to compute the individual shortest paths. As previously said [5], there are many possible search algorithms for this purpose. We choose the A* search method since it gives optimal individual paths with reduced time and memory consumption [5.8].

To speed up the search, we also want the algorithm to make a small amount of A* calls. If we think about applying A* initially for every skid and then each time we have to replan, we would like to have a few conflicting situations. On the other, if we apply A* at each step, independently of whether a collision was just avoided or not, we hope the total number of steps is as small as possible.
In other words, the number of search method calls is strictly related both to the number of avoided conflicts and the sum of timesteps.

To conclude, we must pay attention to the number of A* calls.

### 7.1.3 Sum of Time steps - Sum of Costs

This is an interesting objective function in MAPF problems, describing how many overall actions are required for the system to conclude the task.

In practical situations, the *sum of time steps*, or *sum of costs* [1.2.2], is used to check the efficiency of the search method. If every action, *move* or *wait*, has an assigned cost, e.g., in terms of energy, oil, or money, we would like to reduce it as much as possible.

Interestingly, the sum of time steps gives an *insight into the behavior of skids in the system* too. Of course, MAPF *visualization* is the best way to understand how objects are moving in the system, but the sum of time steps is useful for this too.
A small sum of costs is a symbol of neat movements, while if agents have aesthetically unpleasant or inefficient behaviors, this causes an increment in the total sum. To define our agents as autonomous, we want them to apply reasonable movements, contributing to a small sum of time steps [1].
In the algorithms where agents are self-interested, we expect the local repair (i.e., the collision avoidance techniques) will cause these kinds of unwanted behaviors. Intuitively, with higher cooperation between skids, their behavior gets better.

We are using the sum of time steps and the sum of costs interchangeably, but they generally are different quantities. For grids with *unitary* edge cost, these two quantities coincide. On the other hand, to represent more real-world environments, we may consider *non-homogeneous* grids and operations too, implying a difference both quantitatively and qualitatively. Next, we will sometimes impose varying costs on the grid and observe the difference between the over-mentioned quantities.

### 7.1.4 Maximum individual number of steps - Maximum individual cost

The other most used objective function for MAPF problems is the *makespan*, i.e., the maximum number of time steps required for all agents to reach their target.

Similarly to the sum of costs, we may want to model real situations where agents have limited capacity, for example, in terms of energy or mechanical consumption. Consider a multi-robot system. If each electric robot has 8 hours of autonomy, quite realistically, we don't want them to turn off in the middle of the grid.

The makespan objective function is not related to the sum of costs. There may be situations where the search algorithm produces an elevated sum of costs but with a small makespan, or, conversely, the sum can be small but one agent has a high cost to reach its destination. So, generally, we don't expect any specific connection between the two evolutions.

As noticed, the sum of time steps and the sum of costs are different for a non-homogeneous grid. This applies to the makespan too. When analyzing the concept of congestion awareness, by including some varying weights on edges, we will present this difference.

[1] Concerning the behavioral interpretation, the *sum of fuel* (non-waiting actions required to reach the target) objective function is even better.

### 7.1.5 CPU time consumption and connected aspects

Every program we run requires a series of operations to be executed sequentially. A program is an ordered sequence of instructions, each coded as a binary sequence. A ***CPU*** is a device implemented in a microprocessor, used to execute one instruction at a time. Its fundamental operation is the *execution of an instruction*.
*Three phases* are involved in the execution of an instruction, and they are repeatedly executed over time: *extraction* (from the memory), *decoding*, and *execution* of the instruction. If we are asking to compute a huge amount of operations, the algorithm runs in an elevated amount of time. Despite this, if our CPU is slow in the execution, even with few operations there might be a high time consumption.

Our search algorithms are run on a PC with an Intel CORE i3 microprocessor. A stronger microprocessor would use less CPU time.

In our work, we want these algorithms to find a solution in a small amount of time. So, we consider the time required to run the code, even though we know it is specific to the microprocessor we are using, and it depends on whether the PC is executing other jobs too. For this reason, results are not as well reading as for the other variables, but we still find trends, like when a method or a grid brings better or worse solutions. Sometimes, we also analyze the percentage of skids that arrived at the destination before a given time, specifically 5 and 10 seconds. This improves our understanding of the algorithm's performance.

## 7.2 Experimental analysis

### 7.2.1 Experimental analysis: Basic Search

#### Basic Search on 48x48 empty grid

The first variable we analyze is the *number of avoided conflicts* (Figure 7.1). Intuitively, while increasing the number of agents, each skid has a higher probability of meeting other agents during its path. Since the number of agents increases, we expect the number of possible conflicts to grow exponentially.
The experimental results confirm our intuition.

With up to 200 agents, the variable is always lower than 1000 (5x factor). With a double number of agents, the possible collisions quadruplicate (10x factor). 800 skids imply up to 30000 potential conflicts (more than 35x factor).

Let us now consider the *number of search method calls* for the $48 \times 48$ empty grid. Figure 7.1a confirms a connection between the number of conflicting situations and the sum of individual time steps. The growth is exponential, with a higher variability on the right-hand side of the chart. We can even notice similar spikes while increasing the agents in the system.

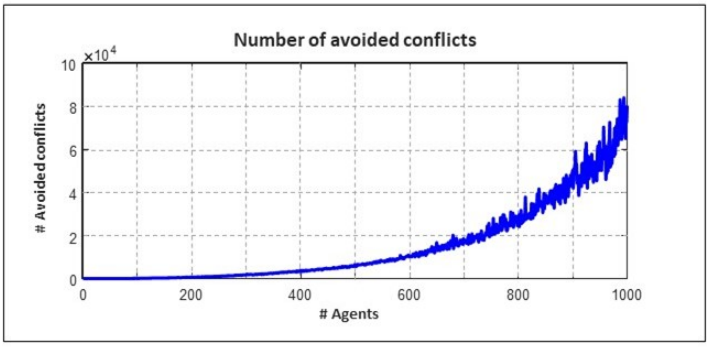
By looking at the $y$-axis values, we find the following relation:

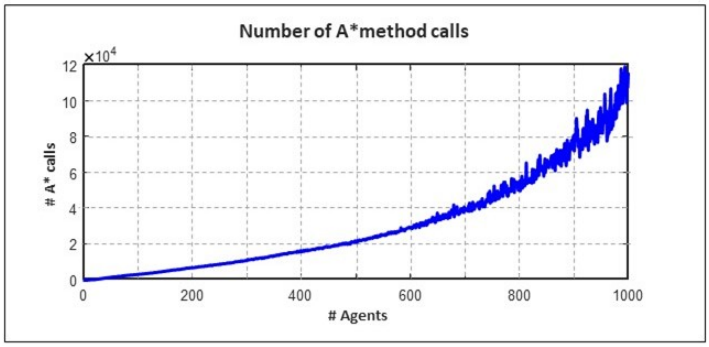
$$sum\ of\ steps\ \approx\ \#\ A^*\ search\ calls\ +\ \#\ avoided\ conflicts$$

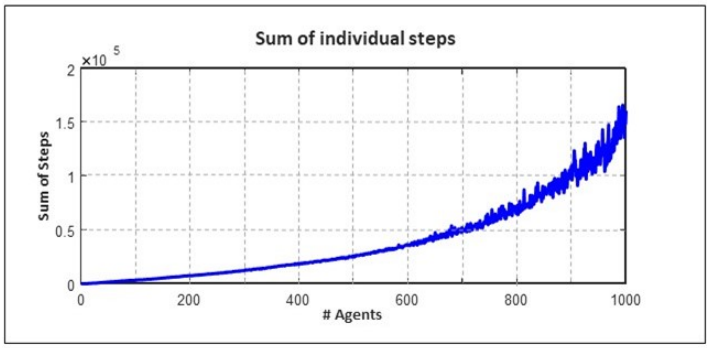
Since the code processes sequentially every agent, whenever a possible conflict could happen, generally, but not necessarily, one of the two agents has not been considered yet. Once agents escape the possible collision, the unconsidered agent is marked as considered, it doesn't recompute its path and is not analyzed until the next step. So, during that step, we count two individual actions, one single A* search call, and one avoided conflict. In this case, the relation holds as equality.
On the other hand, if both agents have been marked as considered, there are two A* function calls, one occurred possible collision, but just $2 \geq 3$ individual actions. Hence, the relationship is just an approximation.

We have noticed the *sum of time steps* is strictly related to the number of search method calls.
While increasing the number of agents, the number of paths increases too. Moreover, the higher the number of paths, the higher the number of possible conflicts. To deal with conflicts, further time steps are needed. This explains the exponential behavior of the sum of time steps (Figure 7.1b).

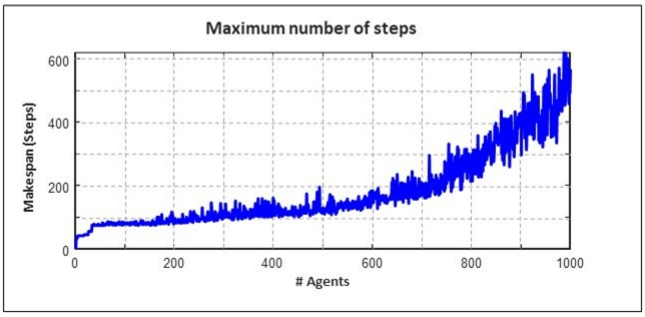
If we impose movement costs on the grid, we should substitute the sum of time steps with the sum of costs. The approximate relation obtained previously doesn't hold anymore. Indeed, movements are associated with weights.

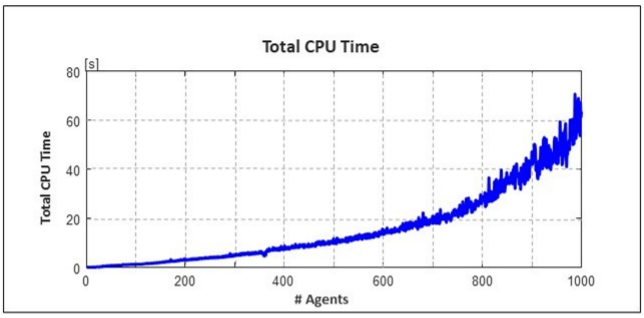

Number of avoided conflicts
×10^4
# Avoided conflicts
# Agents
0
2
4
6
8
10
0
200
400
600
800
1000


Number of A*method calls
×10^4
# A* calls
# Agents
0
2
4
6
8
10
12
0
200
400
600
800
1000


Sum of individual steps
×10^5
Sum of Steps
# Agents
0
0.5
1
1.5
2
0
200
400
600
800
1000

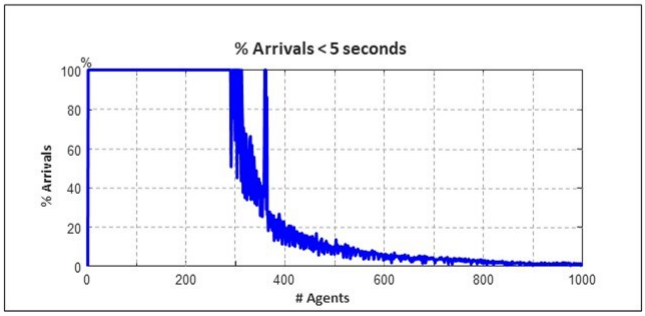
(a)

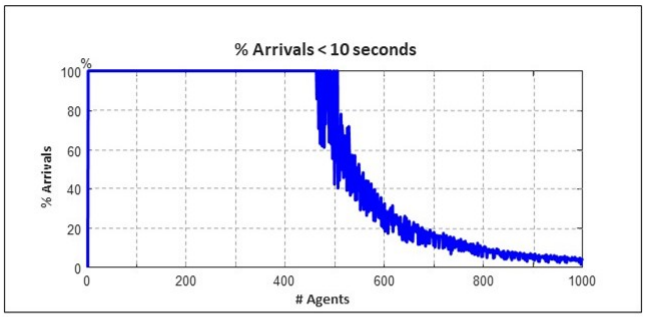



(b)

The *CPU time* required to solve the problem is approximately linear in the first part and almost exponential on the right-hand side (Figure 7.1b). Before 600 skids, we wait no more than 15 seconds, linearly in the number of agents, meaning the algorithm is very good when the grid is sparsely populated. This reminds the advantage of a decentralized




(c)

Figure 7.1: Basic Search performance. We consider the $48 \times 48$ empty grid. On the $x$-axis, we have the number of agents (from 1 to 1000), while on the vertical axis we show 7 different performance measures.

approach over centralized ones: the latter is proven to require an exponential computational cost.
In the variant shown in [6.2.2], agents only replan when conflicts are avoided. The CPU time is expected to increase less rapidly. Moreover, we don't expect any influence neither in the number of avoided conflicts nor in the sum of time steps.

The drawback of this algorithm is agents are completely *self-interested* [1.1.4]. Using local collision avoidance implies agents apply aesthetically unpleasant moves if they are in crowded areas. Hence, it is more difficult for them to reach their destinations [7.3]. The computational cost chart confirms this: the evolution is almost exponential when going from 600, to 800, up to 1000 agents.

To understand deeper the time consumption of the *basic search*, we can refer to the last two charts of Figure 7.1, where we control the percentage of skids that finished their job before the thresholds of 5 and 10 seconds. Starting from 500 agents, the number of agents that arrived before 10 seconds falls from 100% to $< 10\%$ for 800 agents. From 300 to 500 skids, more than half the number of skids use more than 5 and less than 10 seconds to reach the corresponding goal, on average.

Interestingly, note the high variability of the arrivals percentage under 5 seconds when the number of skids is around 300 and that of arrivals under 10 seconds when we consider around 500 agents, respectively. The variability is due to two elements:

- trivially, when we have few agents, changing the $y$-axis value of only one of them may highly change the mean; increasing the number of individuals reduces this effect;
- randomly permuting agents at each loop brings different behaviors even when adding one single agent to the system [7.3].

The first contribution tends to decrease and asymptotically disappear when augmenting the agents, whereas the second contribution persists.

Concerning the *maximum number of time steps*, we have a less clear exponential growth, because added agents may have the origin and destination very close or far from each other. It's the randomness in the position of the origin and the goal that causes this spiky chart. Furthermore, the random permutation of agents' order at each loop plays a key role. It is interesting to compare the makespan and the sum of time steps for different seeds, as described below [7.2.1].

### Basic Search on 48x48 empty grid: Comparison between performances using different seeds

We previously described the advantages of permuting the agents while performing the basic search. We introduced the ordering to solve pathological loops like those where agents are blocked by others already processed (Figure 6.3). Consider such a situation. Randomness guarantees in the future the blocked skid will be processed first (i.e., its *priority* is higher than that of the other involved skids) and will push away some agent from its immediately planned next step [6.1.1].

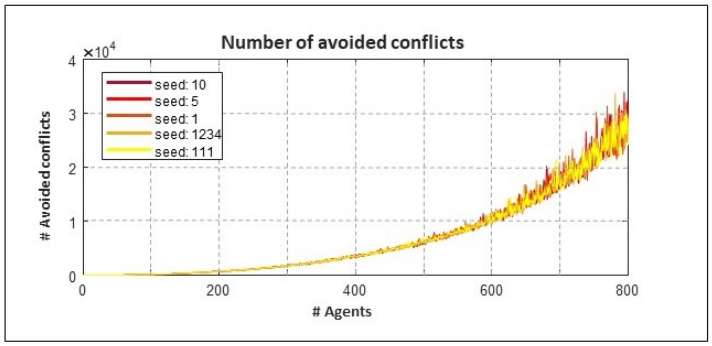
We want to understand if the permutation influences the number of avoided conflicts and how much. For this, we compare the results of 5 *basic search*'s runs, each with a different *seed* used to guide the generation of *pseudo-random permutations*: 10, 5, 1, 1234, 111. Different random permutation seeds modify the results (not all the *"spikes"* coincide), but if the trend is quite the same, we can proceed with one specific seed.

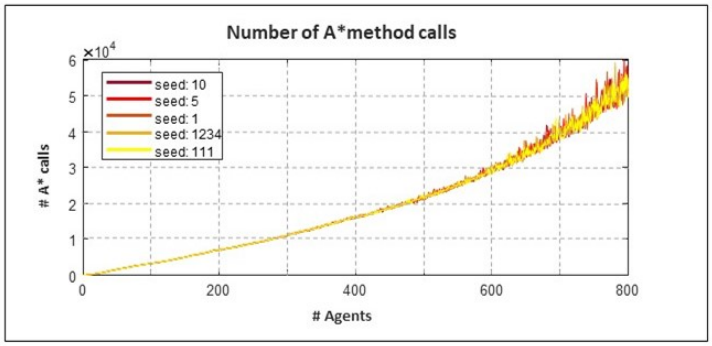
From Figure 7.2, we notice 4 out of 5 curves almost coincide (apart from the percentage of arrivals, which may depend on the jobs the PC is concurrently doing). Slight differences are observable when the number of agents exceeds 600, but the overall trend is the same. When we have few agents, the number of possible conflicts and pathological loops (deadlocks) are reduced. By adding skids, paths tend to intersect each other with an exponential increase both in the number of possible collisions and in the number of deadlocks. A good or bad ordering may reduce or enlarge this behavior, respectively. Hence, when we increase the skids' cardinality, we observe more differences between runs with different seeds.

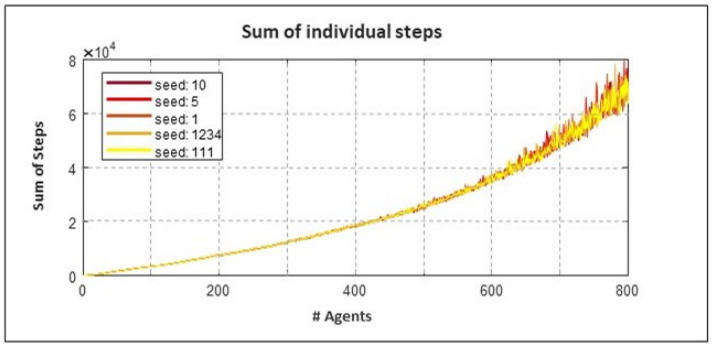
Particular mention deserves the makespan values. From searches with different seeds, by increasing the number of agents, we notice a similar sum of costs but very different makespan values. Two considerations come from this.

- Permuting agents to simulate agents' speed in deciding what to do is useful for some skids, not others. Indeed, some agents may occupy cells that help reach their goal, whereas others don't route where they would. While the overall system has the same behavior, different permutations bring different makespans. For this reason, the sum of time steps chart shows similar values, but the maximum of the required time steps is highly variable.
- In this setting, the sum of time steps outperforms the makespan since the latter could be misleading.

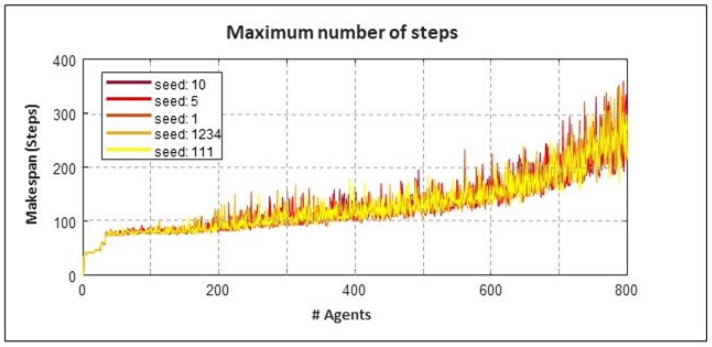
Some permutations are better than others, depending on the specific skids' scenarios. Nevertheless, our code implementation requires many loops so many different permutations are used during the process. A bad ordering of the agents will negatively influence our search for at most one step. The same interpretation works for good skid ordering. Consequently, the permutation is itself very important in MAPF, as many articles show. By applying different orderings at each iteration, the positive or negative effect is restricted to a small portion of the process. That is the advantage of randomly assigning priorities at each step, and explains why the differences are small.

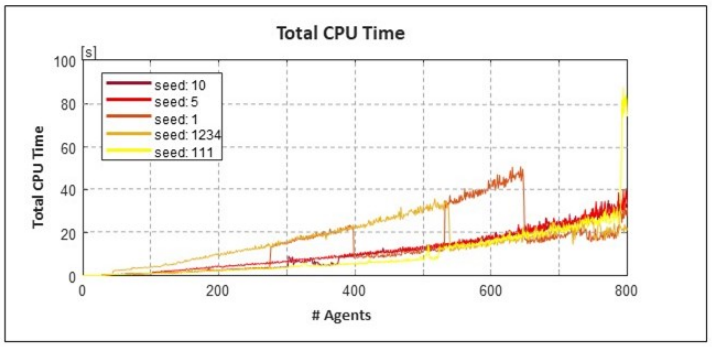

Number of avoided conflicts
×10^4
# Avoided conflicts
seed: 10
seed: 5
seed: 1
seed: 1234
seed: 111
0
1
2
3
4
0
200
400
600
800
# Agents


Number of A*method calls
×10^4
# A* calls
seed: 10
seed: 5
seed: 1
seed: 1234
seed: 111
0
1
2
3
4
5
6
0
200
400
600
800
# Agents

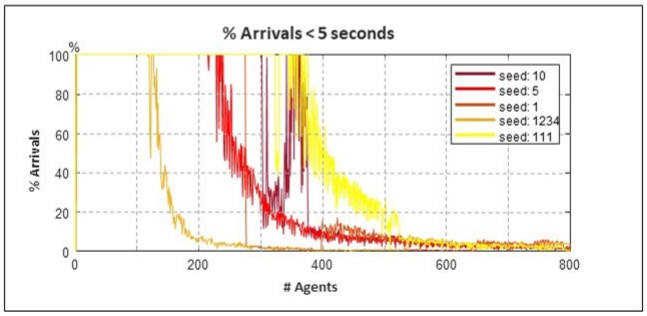

Sum of individual steps
×10^4
Sum of Steps
seed: 10
seed: 5
seed: 1
seed: 1234
seed: 111
0
2
4
6
8
0
200
400
600
800
# Agents

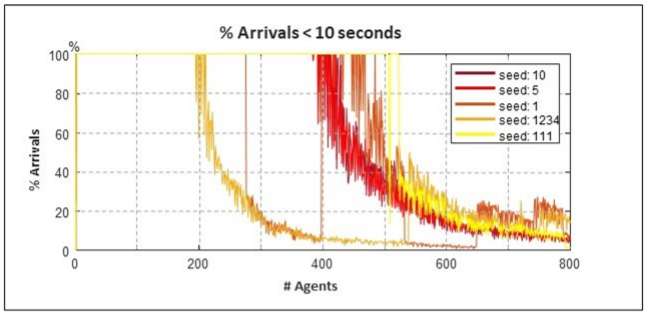
(a)


Maximum number of steps
Makespan (Steps)
# Agents
seed: 10
seed: 5
seed: 1
seed: 1234
seed: 111
0
100
200
300
400
0
200
400
600
800


Total CPU Time
[s]
Total CPU Time
# Agents
seed: 10
seed: 5
seed: 1
seed: 1234
seed: 111
0
20
40
60
80
100
0
200
400
600
800

(b)




(c)

Figure 7.2: Comparison of the basic algorithm with 5 different seeds. We consider the $48 \times 48$ empty grid. On the $x$-axis, we have the number of agents (from 1 to 800), while on the $y$-axis we show 7 different performance measures.

Since all the other algorithms exploit this permutation idea, we can proceed with our analysis considering a specific seed (e.g., 10).

Figure 7.3 shows the same results, but now the number of skids goes from 50 to 800, with stepsize equal to 50. This is useful when considering more complex algorithms for time-computational reasons.

### Basic Search on other benchmark grids

Finally, we deeper study the *basic search* on other grids. Figures 7.4 and 7.5 show the performance charts obtained applying this method to two other benchmark grids, Boston grid (Figure 4.1) and Maze Grid (Figure 4.2), respectively.

Concerning the Boston grid, the first 3 charts (Figure 7.4a) show an approximately linear growth, especially for the number of A* method calls and the sum of individual steps. This is different from the case of the empty grid, where we have seen a clear exponential trend.

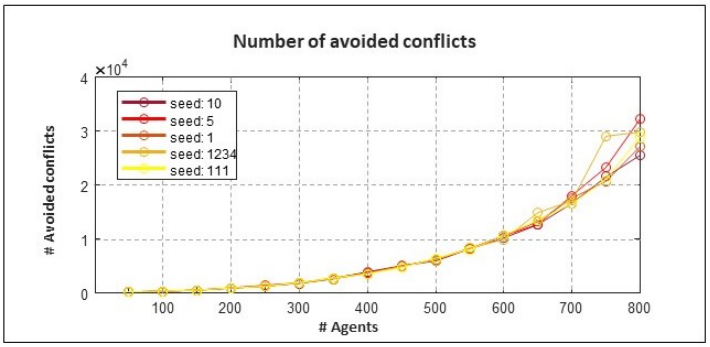
The absence of proportionality is confirmed by Table 7.1.

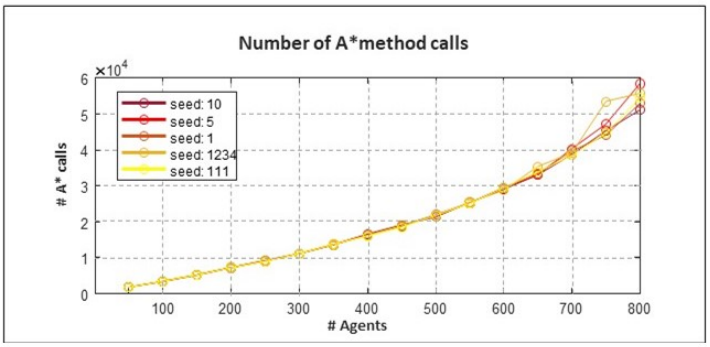
| *Basic Search* | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Avoided conflicts | | | Search method calls | | | Sum of time steps | | |
| # Agents | Empty Grid | Boston Grid | Ratio | Empty Grid | Boston Grid | Ratio | Empty Grid | Boston Grid | Ratio |
| 100 | 157 | 563 | 3.59 | 3337 | 20150 | 6.04 | 3458 | 20632 | 5.97 |
| 200 | 731 | 2129 | 2.91 | 7094 | 39897 | 5.62 | 7611 | 41529 | 5.46 |
| 300 | 1669 | 5671 | 3.40 | 11066 | 62861 | 5.68 | 12228 | 66807 | 5.46 |
| 400 | 3767 | 11842 | 3.14 | 16502 | 88844 | 5.38 | 18994 | 96571 | 5.08 |
| 500 | 5834 | 36442 | 6.25 | 21308 | 137228 | 6.44 | 25043 | 158067 | 6.31 |
| 600 | 10139 | 34873 | 3.44 | 29019 | 152864 | 5.27 | 35439 | 173874 | 4.91 |
| 700 | 16624 | 59921 | 3.60 | 38739 | 200816 | 5.18 | 48742 | 235258 | 4.83 |
| 800 | 25590 | 72233 | 2.82 | 51194 | 232706 | 4.55 | 66246 | 274188 | 4.14 |
| 900 | 38440 | 119490 | 3.11 | 67686 | 306552 | 4.53 | 89789 | 372284 | 4.15 |

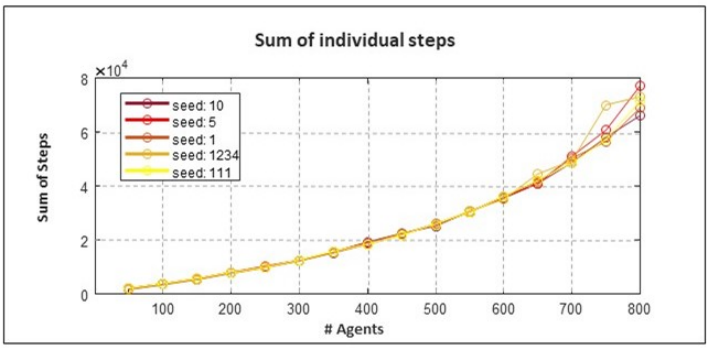
Table 7.1: Comparison of the *Basic search* performance between the empty and the Boston grid. Three variables are analyzed: number of avoided conflicts, number of A* calls, and the sum of individual time steps. Ratio columns represent the ratio between the values for the same variable but different grids.

As we can see from the ratios between the values for different grids, they are not preserved.

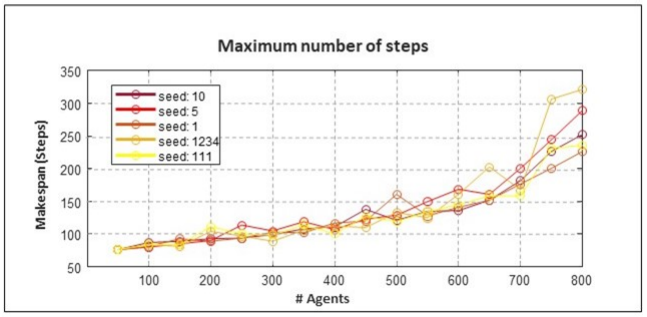
Furthermore, they are unexpectedly different concerning different variables. We would expect similar ratios between different grids when changing the analyzed variable because of the approximate relation shown above. To understand this, we recall the relation above is just an approximation, so there may be some cases where it is not true. In particular, when the grid is sparsely populated, i.e., the number of agents is reduced and there are many empty areas, the number of avoided conflicts is too small for the relation to hold. A higher number of skids is needed.

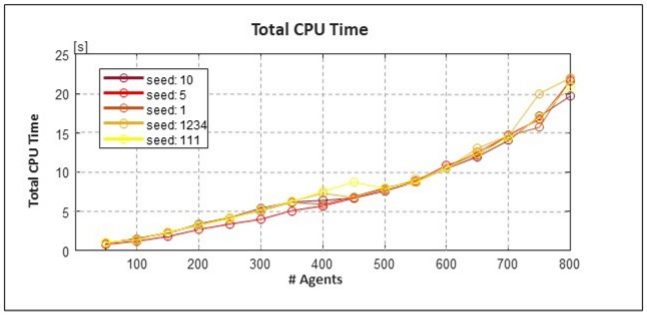

Number of avoided conflicts
×10^4
# Avoided conflicts
# Agents
seed: 10
seed: 5
seed: 1
seed: 1234
seed: 111


Number of A*method calls
×10^4
# A* calls
# Agents
seed: 10
seed: 5
seed: 1
seed: 1234
seed: 111


Sum of individual steps
×10^4
Sum of Steps
# Agents
seed: 10
seed: 5
seed: 1
seed: 1234
seed: 111

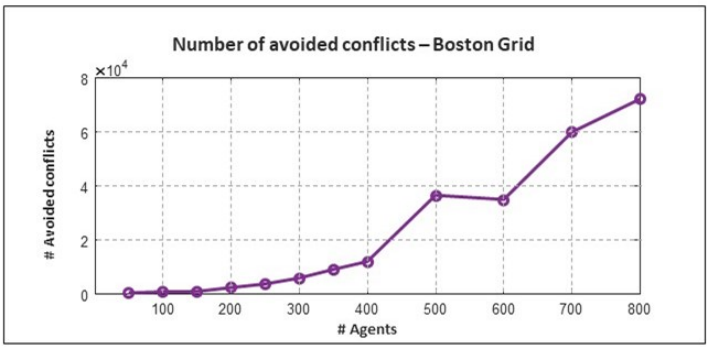
(a)

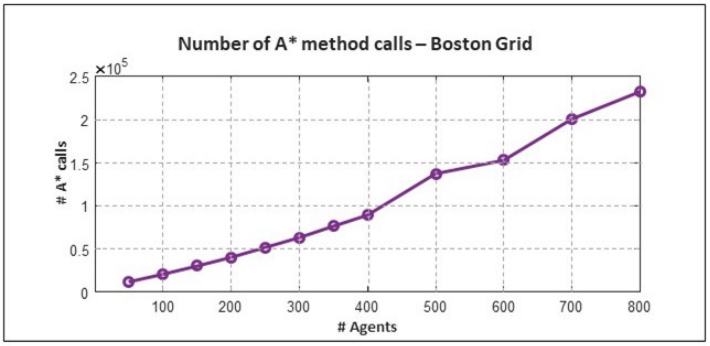


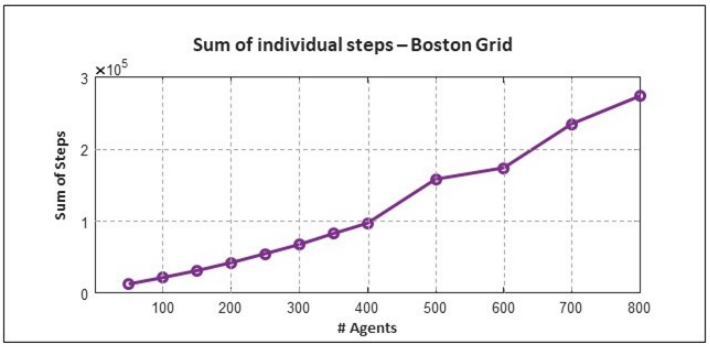




(b)

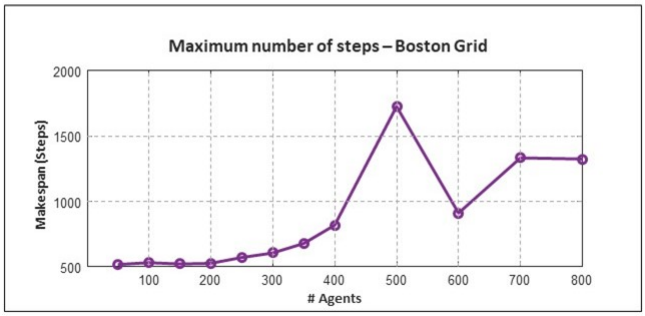
Figure 7.3: Comparison of the basic algorithm with 5 different seeds. We consider the $48 \times 48$ empty grid. On the $x$-axis, we have the number of agents (from 50 to 800 with stepsize 50), while on the $y$-axis we show 5 different performance measures.

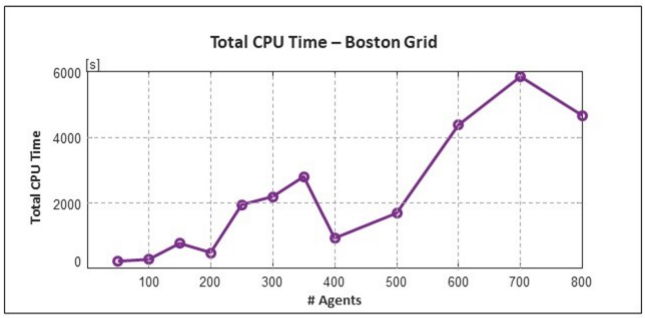

Number of avoided conflicts – Boston Grid
×10^4
# Avoided conflicts
0
2
4
6
8
100
200
300
400
500
600
700
800
# Agents


Number of A* method calls – Boston Grid
×10^5
# A* calls
0
0.5
1
1.5
2
2.5
100
200
300
400
500
600
700
800
# Agents


Sum of individual steps – Boston Grid
×10^5
Sum of Steps
0
1
2
3
100
200
300
400
500
600
700
800
# Agents

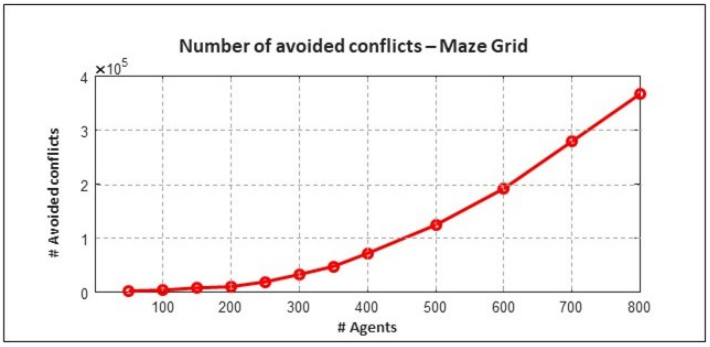
(a)

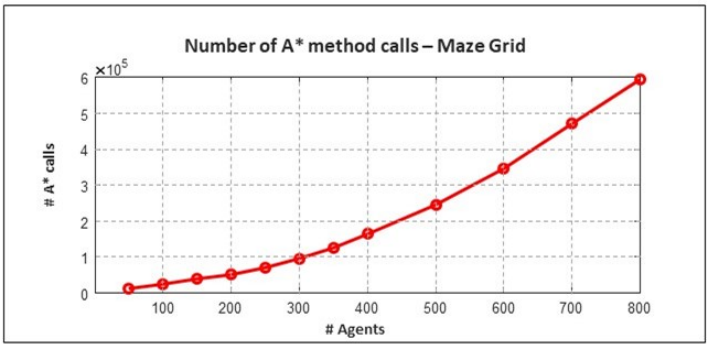


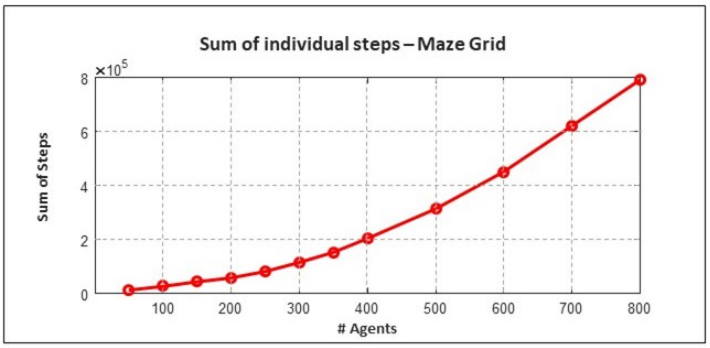




(b)

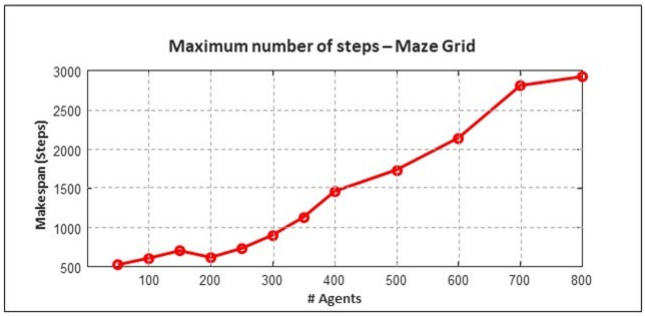
Figure 7.4: Basic Search performance. We consider the $256 \times 256$ Boston grid. On the $x$-axis, we have the number of agents (from 50 to 800 with stepsize 50), while on the $y$-axis we show 5 different performance measures.

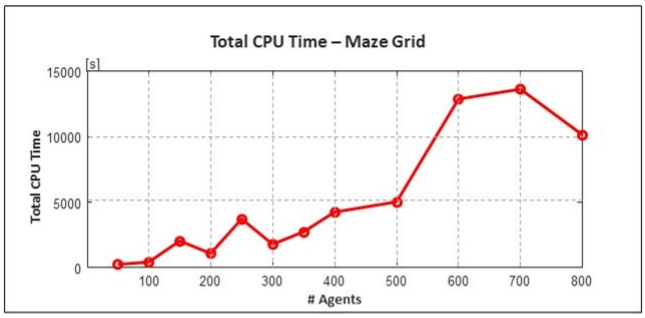

Number of avoided conflicts – Maze Grid
×10^5
# Avoided conflicts
0
1
2
3
4
100
200
300
400
500
600
700
800
# Agents


Number of A* method calls – Maze Grid
×10^5
# A* calls
0
1
2
3
4
5
6
100
200
300
400
500
600
700
800
# Agents


Sum of individual steps – Maze Grid
×10^5
Sum of Steps
0
2
4
6
8
100
200
300
400
500
600
700
800
# Agents

(a)




(b)

Figure 7.5: Basic Search performance. We consider the $128 \times 128$ Maze grid. On the $x$-axis, we have the number of agents (from 50 to 800 with stepsize 50), while on the $y$-axis we show 5 different performance measures.

This is what happens with the Boston grid: even though the number of considered agents is the same as the empty grid, it is so small for the dimension of the grid that the number of avoided conflicts, hence the number of A* method calls, are too small and the approximation becomes false.

As we may imagine, the same considerations hold for the $128 \times 128$ maze grid (Figure 7.5), where again there are no trends on the 3 main variables, as shown in Table 7.2.

| *Basic Search* | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Avoided conflicts | | | Search method calls | | | Sum of time steps | | |
| # Agents | Empty Grid | Maze Grid | Ratio | Empty Grid | Maze Grid | Ratio | Empty Grid | Maze Grid | Ratio |
| 100 | 157 | 2519 | 16.04 | 3337 | 23646 | 7.09 | 3458 | 25469 | 7.37 |
| 200 | 731 | 9213 | 12.60 | 7094 | 50714 | 7.15 | 7611 | 56899 | 7.48 |
| 300 | 1669 | 31690 | 18.99 | 11066 | 95788 | 8.66 | 12228 | 114375 | 9.35 |
| 400 | 3767 | 70788 | 18.79 | 16502 | 163572 | 9.91 | 18994 | 203255 | 10.70 |
| 500 | 5834 | 123665 | 21.20 | 21308 | 245970 | 11.54 | 25043 | 313448 | 12.51 |
| 600 | 10139 | 191916 | 18.93 | 29019 | 347358 | 11.97 | 35439 | 450923 | 12.72 |
| 700 | 16624 | 280001 | 16.84 | 38739 | 472553 | 12.20 | 48742 | 622067 | 12.76 |
| 800 | 25590 | 368354 | 14.39 | 51194 | 596554 | 11.65 | 66246 | 792514 | 11.96 |
| 900 | 38440 | 541385 | 14.08 | 67686 | 822474 | 12.15 | 89789 | 1108206 | 12.34 |

Table 7.2: Comparison of the *Basic search* performance between the empty and the maze grid. Three variables are analyzed: number of avoided conflicts, number of A* calls, and the sum of individual time steps. Ratio columns represent the ratio between the values for the same variable but different grids.

Let us see more in detail the concept of skids' percentage inside the grid.

To see a crowded situation proportional to that of the empty grid's case with 800 agents, we should have Boston and Maze grids with respectively:

$$x/47768 = 35\% \rightarrow x = 16719$$

and

$$x/14818 = 35\% \rightarrow x = 5186$$

having $800/(48 \times 48) = 35\%$ occupied-by-agents cells in the empty grid [2].
For computational reasons, we analyzed up to 1200 agents in the online approach [8.2.1]. This is approximately equal to the percentage required for the specific problem [2].

In conclusion, it is not possible to find clear proportional trends between complex big grids and the simple $48 \times 48$ domain. For more similar results, we should consider the same percentage of occupied cells, instead of the same number of agents.

For all the presented graphs, notice the variability increases while increasing the number of agents. We give an explanation of this in [7.3].

[2] 47768 is the number of obstacle-free cells on the Boston map. In the maze, it is equal to 14818.

### 7.2.2 Experimental analysis: Basic Search + Optimized Pre-Conflict Avoidance (Variant 1)

#### Variant 1 on 48x48 empty grid

By exploiting the drivers' behavior on congested roads, we see agents waiting some time steps before dealing with the possible collisions. So the same conflict is correctly counted many times [6.2.1]. We can easily explain the *makespan* chart in Figure 7.6b, where the maximum number of steps for the optimized algorithm is always higher than the simple one.

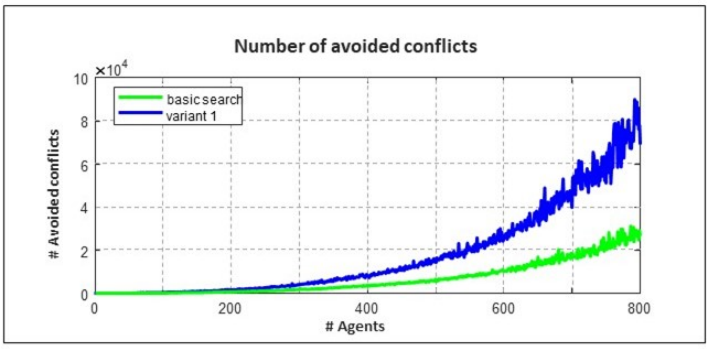
The *"waiting effect"* is not restricted to the maximal length of the path, but to all agents. From this, the *sum of individual steps* is higher too, and the corresponding graph is correct (Figure 7.9a). The exponential behavior is justified by the exponential behavior of the basic search.

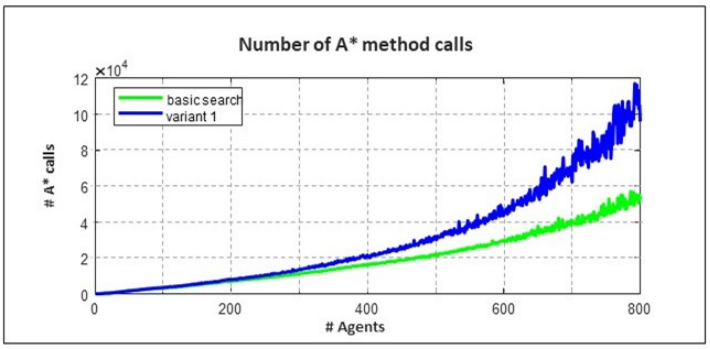
We set the *memory of the tiles* for at most three previous events. Hence, as said in [6.2.1], we generally expect a 3x factor growth of the *number of avoided conflicts*.
The run code confirms this (Figure 7.9a).
In Table 7.3 we show 8 equidistant values both from the *basic search* and this optimized variant. The corresponding mean value ratio is 2.33. Interestingly, this proportionality is also shown in the high variability of the values for more than 600 skids too: the values range triplicates.

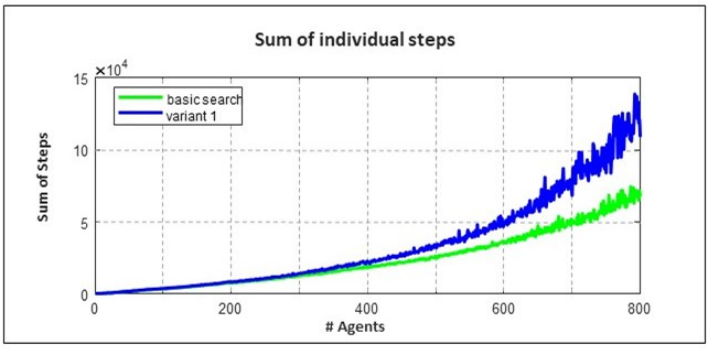
| Empty Grid | | | |
|---|---|---|---|
| | Avoided conflicts | | |
| # Agents | Basic Search | Variant 1 | Ratio |
| 100 | 167 | 388 | 2.32 |
| 200 | 718 | 1483 | 2.07 |
| 300 | 1756 | 4004 | 2.28 |
| 400 | 3412 | 8090 | 2.37 |
| 500 | 6400 | 14505 | 2.27 |
| 600 | 10651 | 25124 | 2.36 |
| 700 | 16670 | 39920 | 2.39 |
| 800 | 26547 | 69156 | 2.61 |

Table 7.3: Comparison between *basic search* and *variant 1* for 8 different values of the number of conflicts. Data come from the empty grid scenario. Ratio columns represent the values ratio between the *basic search* and the *variant 1*.

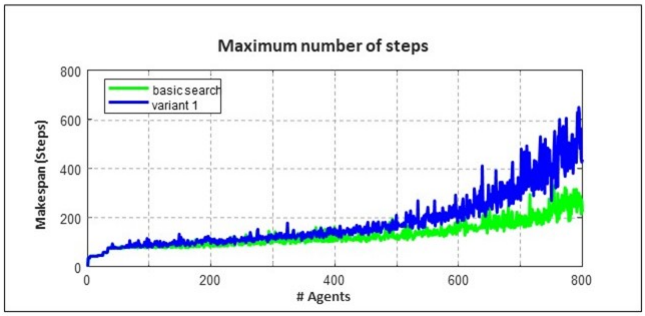
The number of avoided conflicts influences the previously described variables, so we focus more on this by presenting the *linear regression* obtained by the *least squares method* (Figure 7.7).

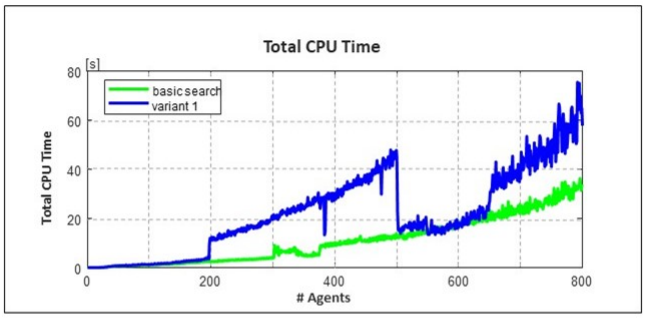

Number of avoided conflicts
×10^4
10
8
6
4
2
0
basic search
variant 1
# Avoided conflicts
0
200
400
600
800
# Agents


Number of A* method calls
×10^4
12
10
8
6
4
2
0
basic search
variant 1
# A* calls
0
200
400
600
800
# Agents


Sum of individual steps
×10^4
15
10
5
0
basic search
variant 1
Sum of Steps
0
200
400
600
800
# Agents

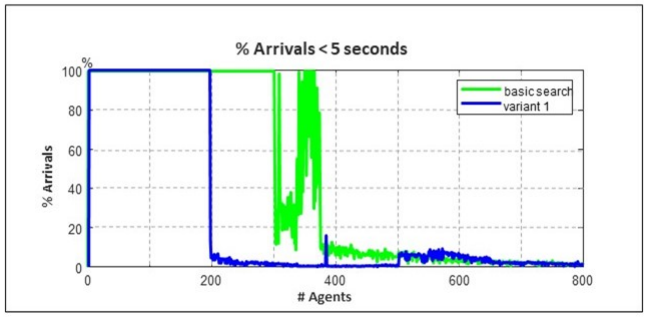
(a)

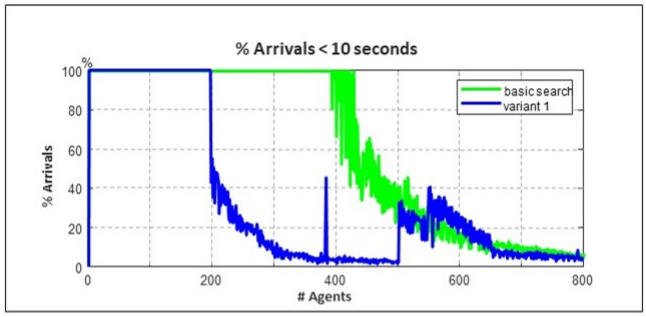



(b)

We analyzed the relation between the *basic search* and the *variant 1* with a linear approximation, where on the $x$-axis we have the # of possible conflicts for the first method while on the $y$-axis we analyze the same variable but for the optimized version.




(c)

Figure 7.6: Comparison between *basic search* and *variant 1*. We consider the $48 \times 48$ empty grid. On the $x$-axis, we have the number of agents (from 1 to 800), while on the $y$-axis we show 7 different performance measures.

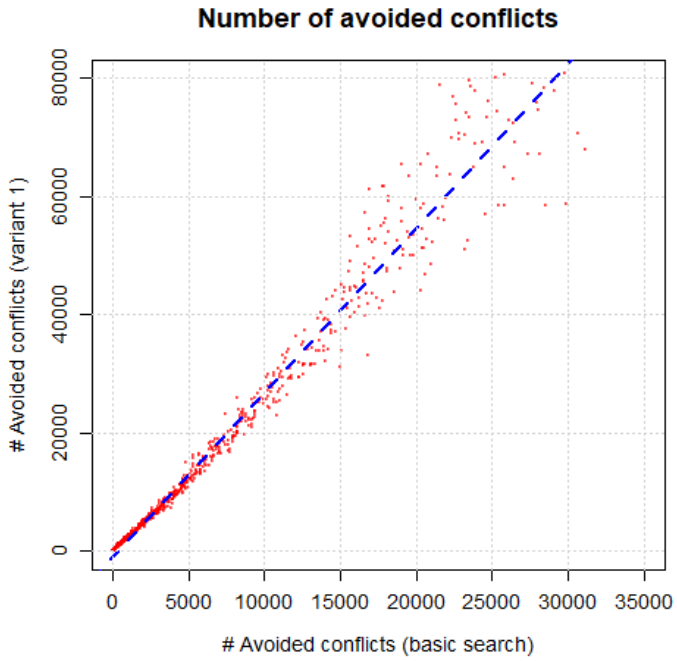



Figure 7.7: Comparison between *basic search* and *variant 1* in terms of the number of avoided conflicts. For the same number of agents, on the $x$-axis we present the value obtained when using the *basic method*, whereas on the $y$-axis we give the *variant 1*'s value. The regression line is shown in blue color.

The $\beta$ coefficient (*angular coefficient*) is 2.78 [3].
Note $2.78 \neq 3$, because many times while waiting before acting, the possible conflict is solved differently. This proves the utility of this variant. The more the average is below 3, the more useful is the method.

There are two main *drawbacks* of *variant 1*.

- Like for the *basic method*, the path is replanned each time, implying an unnecessary huge amount of *search function calls*, even more than in the simple case.

[3] We stress the fact that linear regression comes from the ***least squares optimization problem***. For this latter method, we don't make any statistical assumption on the residual distribution, whereas, for the linear regression, that is a particular case of the Statistics' *Linear Model*, we assume $\sim \mathcal{N}(0, \sigma^2)$. In our case, data spread out while increasing the number of agents (variance increases $\Rightarrow$ heteroskedasticity), against this assumption. To make a statistical inference, we should appropriately modify our model.

- Our main objective function, i.e., the sum of time steps, is too high with respect to the *basic search*.

These drawbacks reflect on the *time consumption*. The total required CPU time is higher than the basic algorithm, and the arrivals' percentage dramatically falls.

The percentages in the last two charts get together when reaching 800 skids because both algorithms require more than 10 seconds for each agent to reach its destination.

### Variant 1 on other benchmark grids

All the five main variables show higher values than in the basic algorithm also when we consider the Boston's and Maze grids (Figures 7.8 and 7.9).

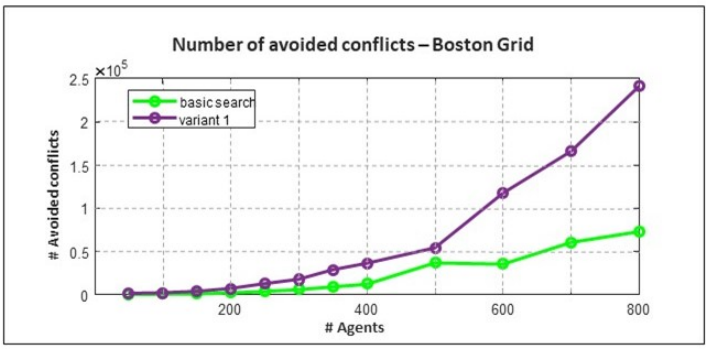
We remark that the percentage of occupied cells is much smaller than for the $48 \times 48$ empty grid, given the same number of mobile agents. While 800 skids equal the 35% of the empty grid, when taking into account Boston's grid the percentage falls to 2%, and it is 5% for the Maze grid. Hence, it has no sense to compare the values for these grids.

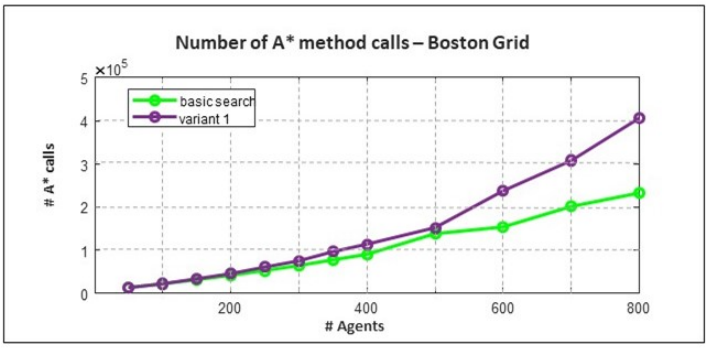
Apart from this, even in these two more complex grids, we observe a proportionality between the *basic search* and the *optimized variant*.
The ratio values are summarized in Table 7.4.

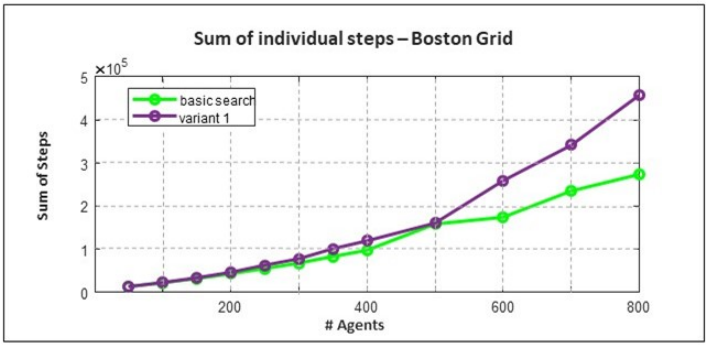
| Avoided conflicts | | | | | | |
|---|---|---|---|---|---|---|
| | Boston Grid | | | Maze Grid | | |
| # Agents | Basic Search | Variant 1 | Ratio | Basic Search | Variant 1 | Ratio |
| 100 | 563 | 1342 | 2.38 | 2519 | 8989 | 3.57 |
| 200 | 2129 | 6370 | 2.99 | 9213 | 49539 | 5.38 |
| 300 | 5671 | 16898 | 2.98 | 31690 | 107319 | 3.39 |
| 400 | 11842 | 35442 | 2.99 | 70788 | 208257 | 2.94 |
| 500 | 36442 | 53405 | 1.47 | 123665 | 332372 | 2.69 |
| 600 | 34873 | 116974 | 3.35 | 191916 | 498253 | 2.60 |
| 700 | 59921 | 165815 | 2.77 | 280001 | 835232 | 2.98 |
| 800 | 72233 | 241596 | 3.34 | 368354 | 1151288 | 3.13 |
| 900 | 119490 | 340312 | 2.85 | 541385 | 1351034 | 2.50 |

Table 7.4: Comparison between *basic search* and *variant 1* for 9 different values of the number of conflicts. Data come from the Boston grid and the Maze grid scenarios. Ratio columns represent the values ratio between the *basic search* and the *variant 1*.

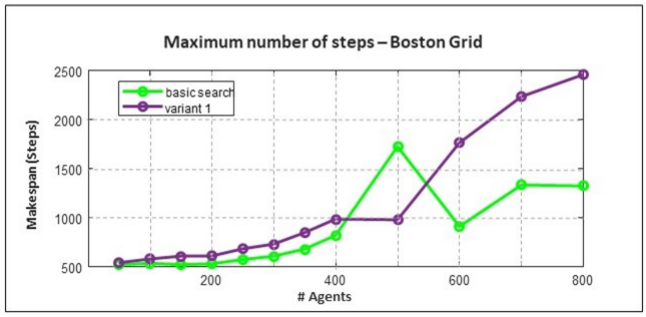
When we have more than 600 agents for all three methods, adopting this variant technique is costly, especially from the point of view of time consumption.

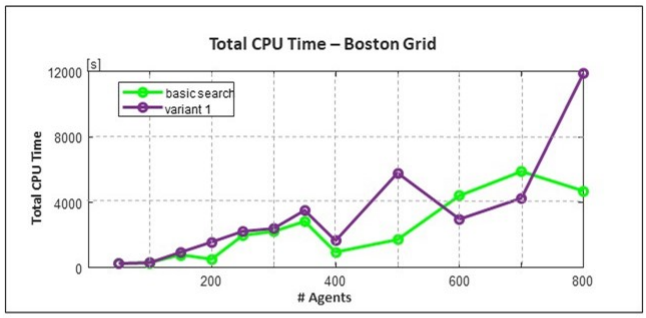

Number of avoided conflicts – Boston Grid
×10^5
2.5
2
1.5
1
0.5
0
# Avoided conflicts
basic search
variant 1
200
400
600
800
# Agents


Number of A* method calls – Boston Grid
×10^5
5
4
3
2
1
0
# A* calls
basic search
variant 1
200
400
600
800
# Agents


Sum of individual steps – Boston Grid
×10^5
5
4
3
2
1
0
Sum of Steps
basic search
variant 1
200
400
600
800
# Agents

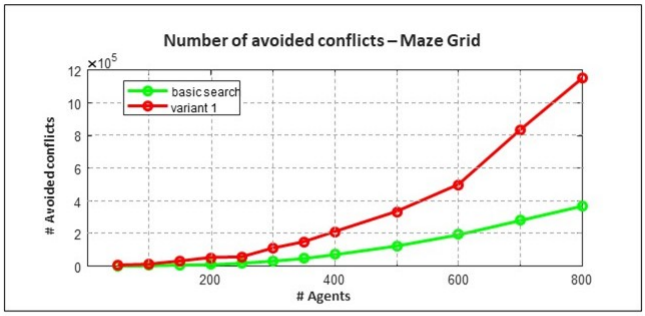
(a)

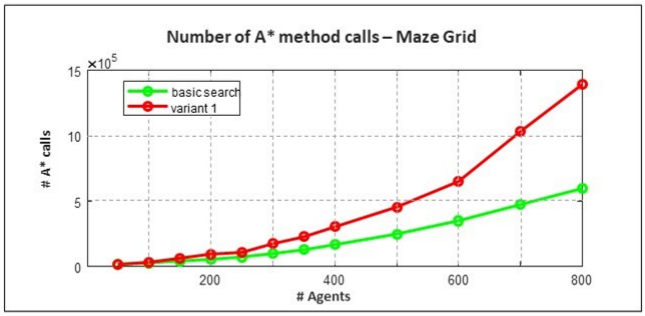


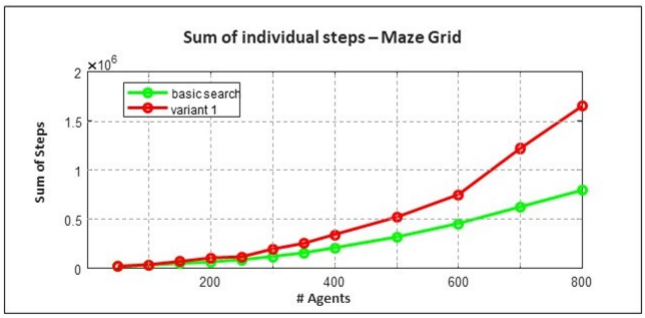




(b)

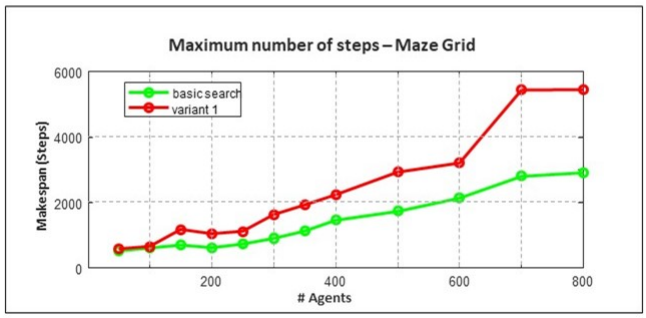
Figure 7.8: Comparison between *basic search* and *variant 1*. We consider the $256 \times 256$ Boston grid. On the $x$-axis, we have the number of agents (from 50 to 800, with stepsize 50), while on the $y$-axis we show 5 different performance measures.

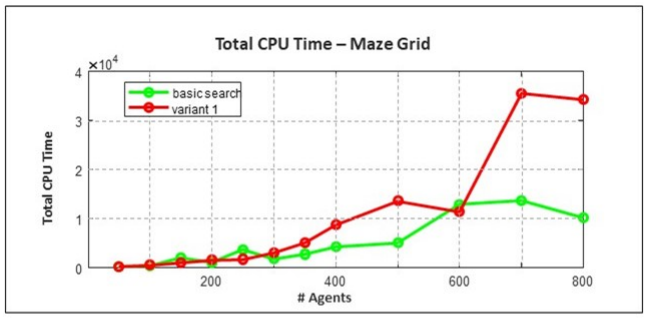

Number of avoided conflicts – Maze Grid
×10^5
12
10
8
6
4
2
0
# Avoided conflicts
basic search
variant 1
200
400
600
800
# Agents


Number of A* method calls – Maze Grid
×10^5
15
10
5
0
# A* calls
basic search
variant 1
200
400
600
800
# Agents


Sum of individual steps – Maze Grid
×10^6
2
1.5
1
0.5
0
Sum of Steps
basic search
variant 1
200
400
600
800
# Agents

(a)




(b)

Figure 7.9: Comparison between *basic search* and *variant 1*. We consider the $128 \times 128$ Maze grid. On the $x$-axis, we have the number of agents (from 50 to 800, with stepsize 50), while on the $y$-axis we show 5 different performance measures.

### 7.2.3 Experimental analysis: Basic Search + Computational Optimization on Search Calls (Variant 2)

#### Variant 2 on 48x48 empty grid

The only fundamental difference between the *basic search* and the version with the computational optimization on the search method calls (*variant 2*) is in the latter case, we replan only when strictly needed, instead of doing so every time we consider an agent.

This version optimizes the computational point of view. There is no improvement in the system.

The *number of avoided conflicts*, the *sum of time steps*, and the *makespan* are the same as the *basic search*. This is shown correctly in Figure 7.10.

The *number of A* method calls* and the *total CPU time* (Figures 7.10a and 7.10b) show the positive effect of the new approach. The number of search function calls is highly optimized, speeding up the process too.

Table 7.5 shows the number of A* calls and the CPU time required by the algorithm to find a suboptimal solution to the MAPF problem on the $48 \times 48$ empty grid. At the same time, we show the percentage value compared to the *basic search*. Observe the massive improvement when the grid contains few agents. The positive effect tends to decrease (but never disappears) when increasing the number of skids. Indeed, as the agents increase, many more possible conflicts occur, meaning much more time replanning is needed.

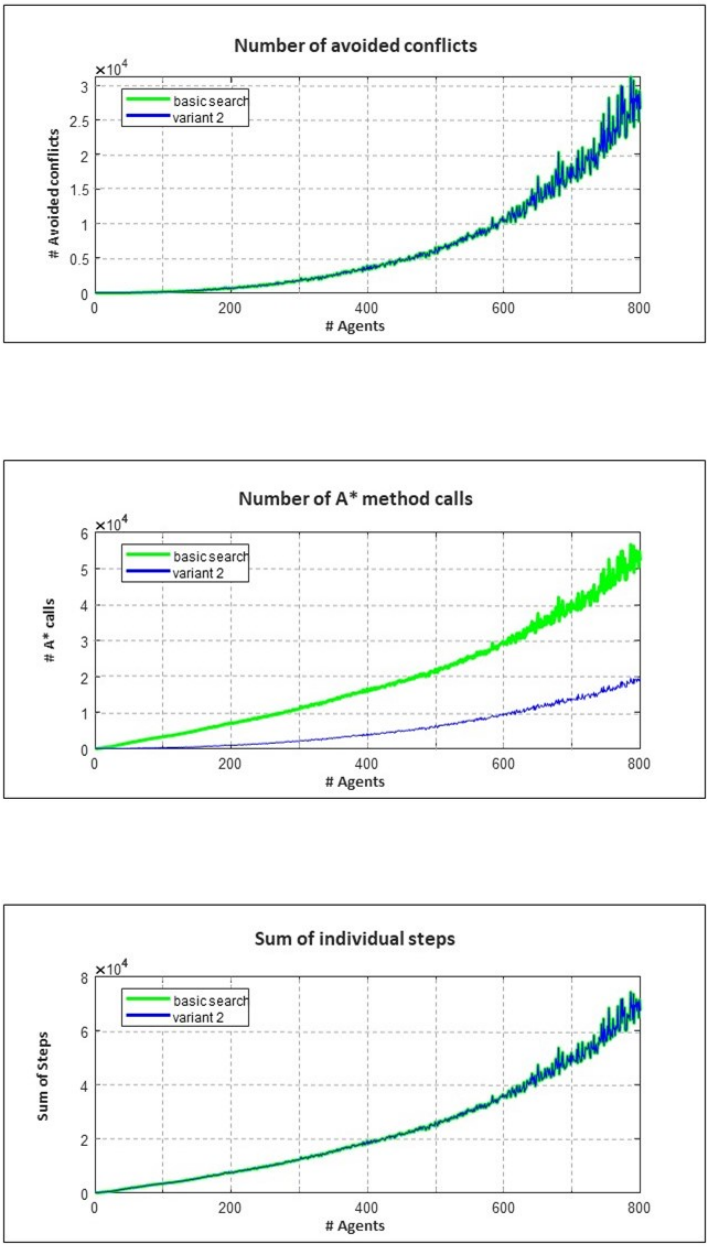
#### Variant 2 on other benchmark grids

The same advantages are shown on the more complex Boston grid and the Maze (Figures 7.11 and 7.12, respectively).

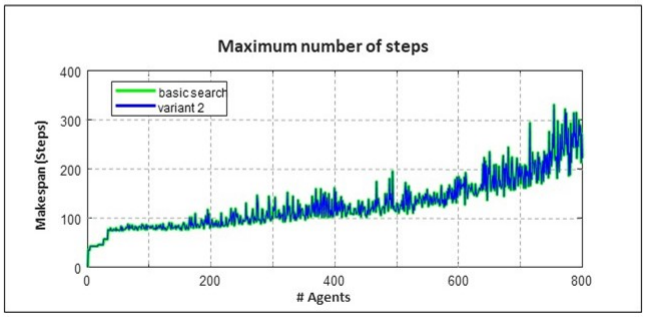
Note the exponential trend in the A* calls variable growth almost disappears. Only when surpassing 600 skids we notice a slight curve. This is reasonable.
When we have few agents, the number of possible collisions is so small that we replan no more than ten times for each agent. This is especially observable for the two bigger grids. In highly crowded grids, replanning is frequent, but the effect of this optimized computation is always positive because the steps with no collisions increase too.

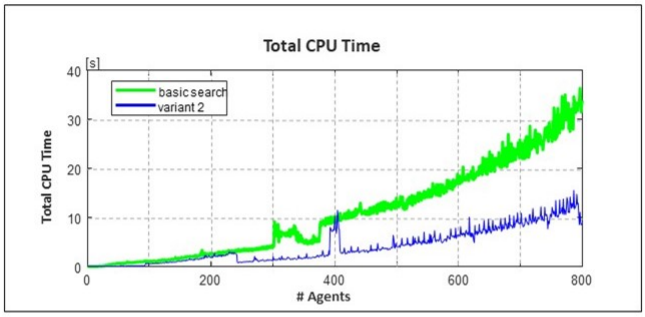
*This idea is very efficient and we could use it for any variant.*

Interestingly, depending on the choice of the algorithm, we may be interested in different objective functions. If we think about applying A* initially for every skid and then each time we have to replan, we would like to have few conflicting situations. On the other hand, if we apply A* at each step, we hope the total number of steps is as small as possible.


Number of avoided conflicts
×10^4
3
2.5
2
1.5
1
0.5
0
# Avoided conflicts
basic search
variant 2
0
200
400
600
800
# Agents
Number of A* method calls
×10^4
6
5
4
3
2
1
0
# A* calls
basic search
variant 2
0
200
400
600
800
# Agents
Sum of individual steps
×10^4
8
6
4
2
0
Sum of Steps
basic search
variant 2
0
200
400
600
800
# Agents

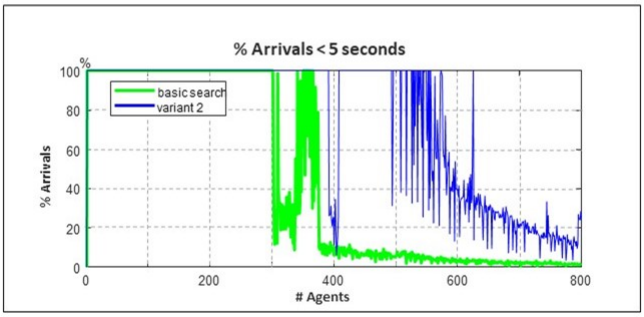
(a)

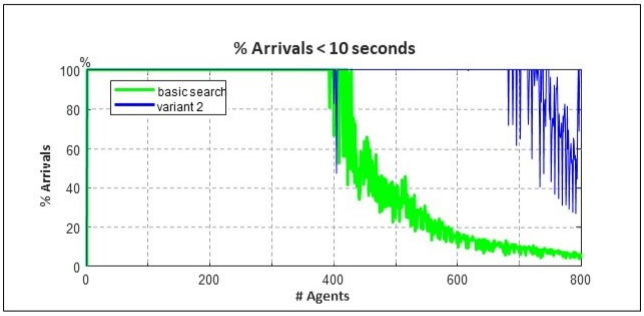



(b)

To conclude, this variant improves the basic method, bringing the number of A* method calls and the total CPU time to grow *linearly* with the number of agents, which is one of the main goals of decentralized algorithms.




(c)

Figure 7.10: Comparison between *basic search* and *variant 2*. We consider the $48 \times 48$ empty grid. On the $x$-axis, we have the number of agents (from 1 to 800), while on the $y$-axis we show 7 different performance measures.

| Avoided conflicts | | | |
|---|---|---|---|
| | Empty Grid | | |
| # Agents | Basic Search | Variant 2 | Percentage |
| 100 | 3337 | 285 | 8.5% |
| 200 | 7094 | 959 | 13.5% |
| 300 | 11066 | 2029 | 18.3% |
| 400 | 16502 | 4072 | 24.7% |
| 500 | 21308 | 5841 | 27.4% |
| 600 | 29019 | 9395 | 32.4% |
| 700 | 38739 | 13584 | 35.1% |
| 800 | 51194 | 18842 | 36.8% |

| CPU Time [s] | | | |
|---|---|---|---|
| | Empty Grid | | |
| # Agents | Basic Search | Variant 2 | Percentage |
| 100 | 1.4 | 0.2 | 14.3% |
| 200 | 3.3 | 0.7 | 21.2% |
| 300 | 5.3 | 1.0 | 18.9% |
| 400 | 6.3 | 2.5 | 39.7% |
| 500 | 7.5 | 2.8 | 37.3% |
| 600 | 10.4 | 4.7 | 45.2% |
| 700 | 14.0 | 6.6 | 47.1% |
| 800 | 19.7 | 9.2 | 46.7% |

Table 7.5: Comparison between *basic search* and *variant 2* for 8 different values of the number of conflicts and the time consumption. Data come from the empty grid scenario. The last column represents the ratio (in percentage) between the *variant 2* and the *basic search*.

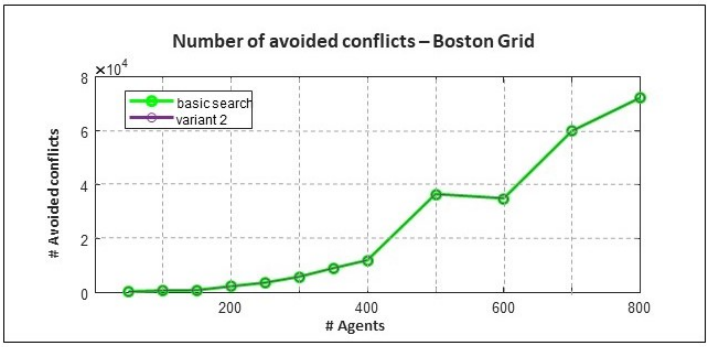
### 7.2.4 Experimental analysis: Basic Search + Traffic Directions (Variant 3)

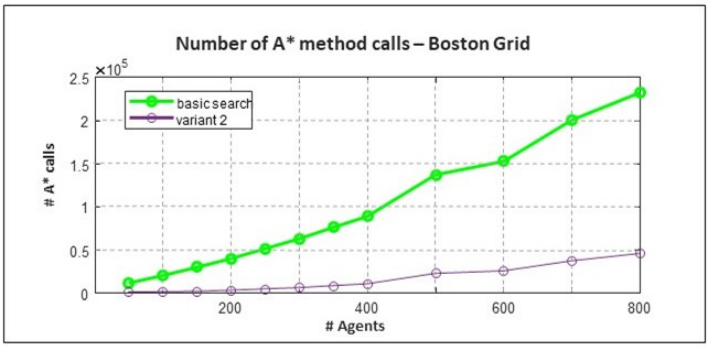
#### Variant 3 on 48x48 empty grid

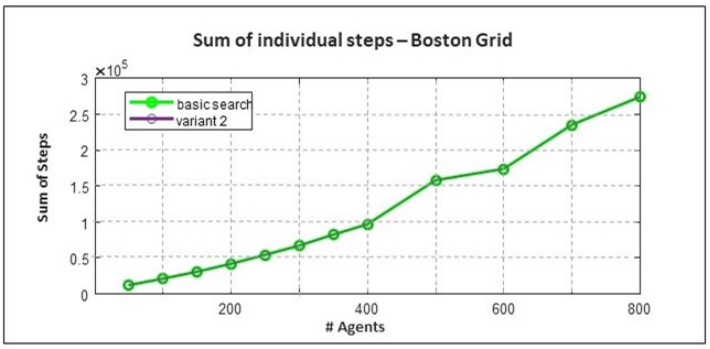
Inspired by the functioning of road networks, we modify the grid's structure by specifying some *"admissible" movements*. From the implementation point of view, this implies just some modification when defining nodes' neighborhoods.

This change requires some a priori knowledge of the specific problem, i.e., the structure of the grid, the fixed obstacles' positions, and the start and goal positions [6.2.3].

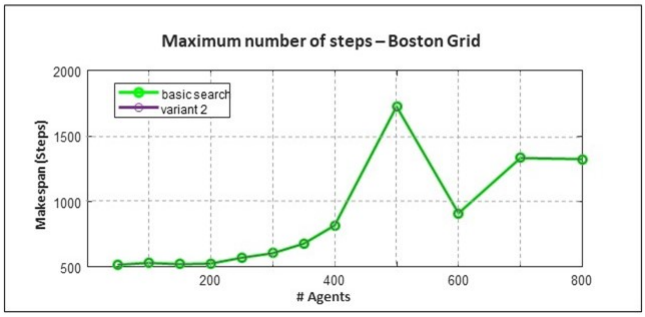
In this work, we focus on algorithms that can be applied to different kinds of grids without exploiting their specific properties. Moreover, the scenarios we used are randomly generated [4.1]. We don't deal with agents going from the same side to the opposite one or with paths showing some patterns that could be exploited.

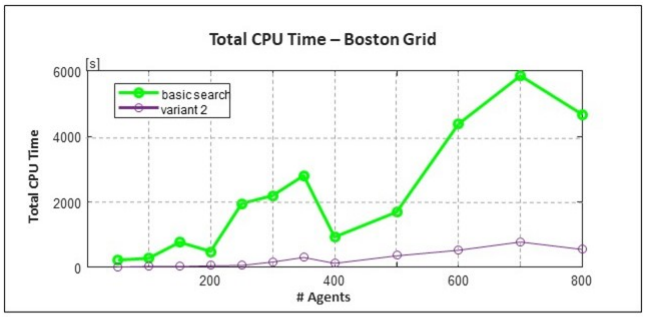

Number of avoided conflicts – Boston Grid
×10^4
8
6
4
2
0
# Avoided conflicts
basic search
variant 2
200
400
600
800
# Agents


Number of A* method calls – Boston Grid
×10^5
2.5
2
1.5
1
0.5
0
# A* calls
basic search
variant 2
200
400
600
800
# Agents


Sum of individual steps – Boston Grid
×10^5
3
2.5
2
1.5
1
0.5
0
Sum of Steps
basic search
variant 2
200
400
600
800
# Agents

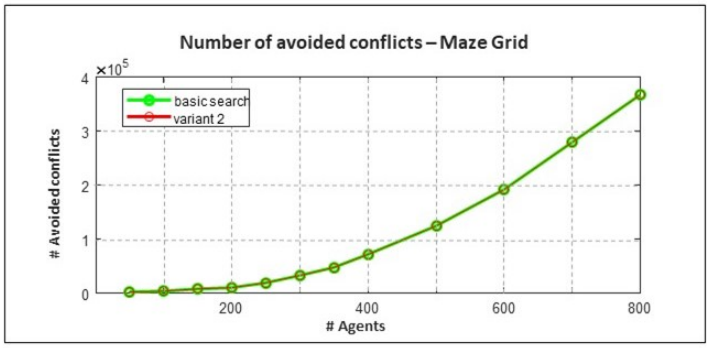
(a)

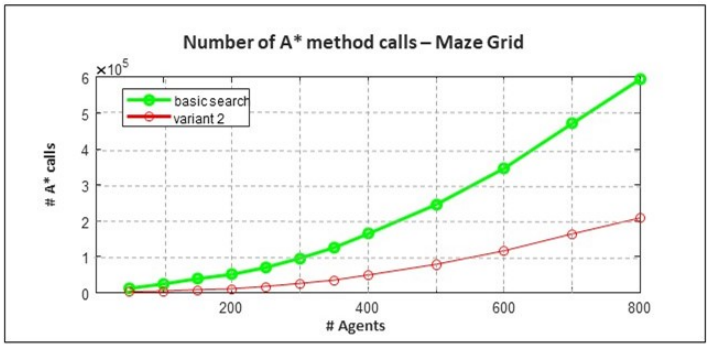


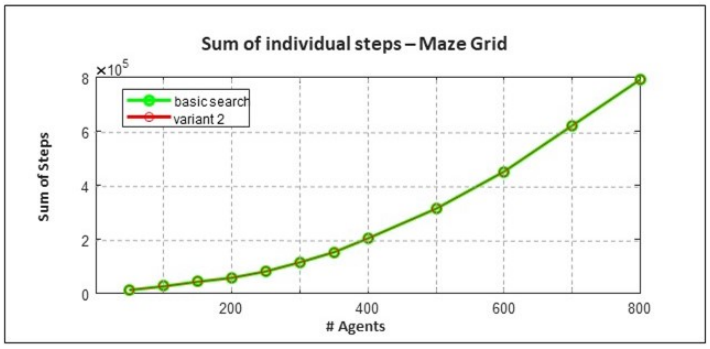




(b)

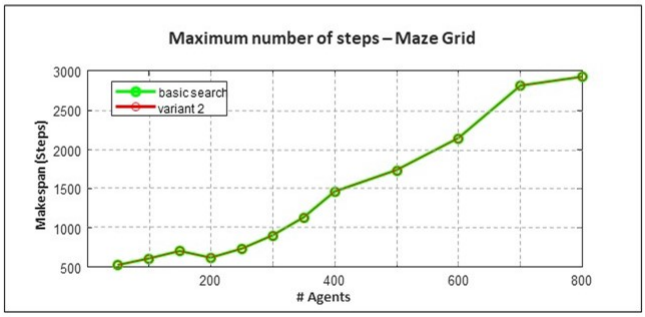
Figure 7.11: Comparison between *basic search* and *variant 2*. We consider the $256 \times 256$ Boston grid. On the $x$-axis, we have the number of agents (from 50 to 800, with stepsize 50), while on the $y$-axis we show 5 different performance measures.

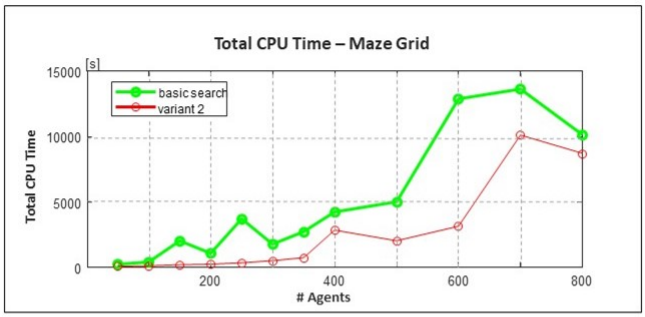

Number of avoided conflicts – Maze Grid
×10^5
basic search
variant 2
# Avoided conflicts
# Agents
0
1
2
3
4
200
400
600
800


Number of A* method calls – Maze Grid
×10^5
basic search
variant 2
# A* calls
# Agents
0
1
2
3
4
5
6
200
400
600
800


Sum of individual steps – Maze Grid
×10^5
basic search
variant 2
Sum of Steps
# Agents
0
2
4
6
8
200
400
600
800

(a)




(b)

Figure 7.12: Comparison between *basic search* and *variant 2*. We consider the $128 \times 128$ Maze grid. On the $x$-axis, we have the number of agents (from 50 to 800, with stepsize 50), while on the $y$-axis we show 5 different performance measures.

Mainly, we want to understand if these *constraints* improve the algorithm's performance on generic grids with generic scenarios too.

We consider first the case in which agents have restrictions on the horizontal direction: alternating on the vertical axis, they cannot move to the left or right (Figure 6.4). In this way, we improve the basic method by a small percentage (Figure 7.13). With less than 600 agents on the domain, the two performances almost coincide concerning *# avoided conflicts*, *# A* method calls*, and the *sum of time steps*. Hence, we can say that with congested areas (35% of the total obstacles-free cells are occupied by mobile agents), it is useful to impose some constraints to move skids more cleverly, in particular with a smaller sum of individual path lengths. This is confirmed by looking at the chart of the *maximum number of steps* (Figure 7.13b). Finally, the *computational cost* is smaller than the first algorithm.

So it seems that by constraining the system, we speed up the process. To better check on this, we constrain the structure even more and look at the results.

#### Variant 3 on 48x48 empty grid: Comparison between performances using different topological constraints

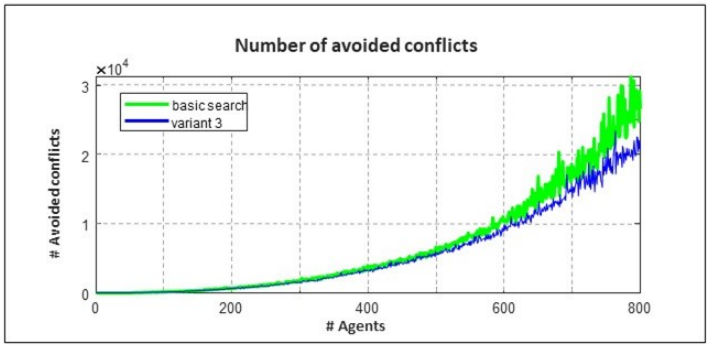
So far, we focused on the case where we alternate rows. Two alternatives we developed consider coupled rows and rows grouped in 4 (Figures 6.5, 6.6, respectively). Anyway, we can modify these constraints for the specific grid and application.

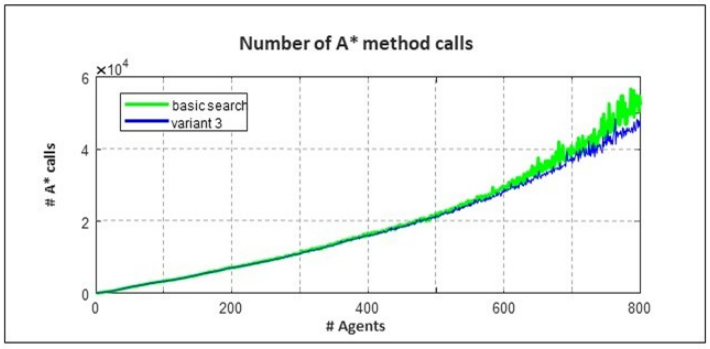
Interestingly, changing the constraints on the generic empty grid doesn't improve the performance of our method. Figure 7.14 shows almost coinciding lines for every variable. The only difference regards the makespan chart (Figure 7.14b), where some agents may require different path lengths depending on the structure we impose.

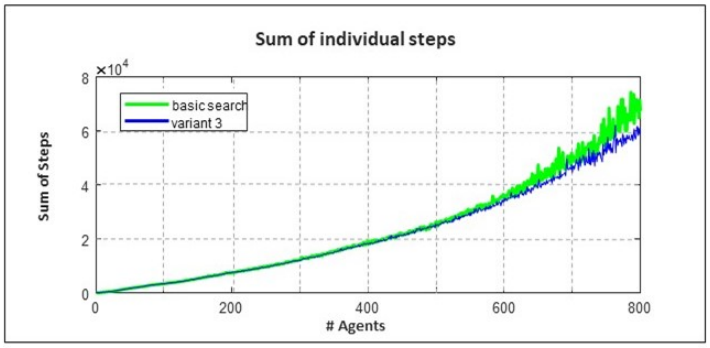
From this, we understand that it is not possible to improve the *basic search*'s performance by just imposing some general constraints on the domain. We need specific *information* about the grid and the agents' starting and final positions. Without this further knowledge, our variant is not as effective as it could be potentially. For this reason, we discard this possible improvement. We remind the reader that our attention is to methods' application to generic grids and scenarios, aiming to improve them.

#### Variant 3 on other benchmark grids

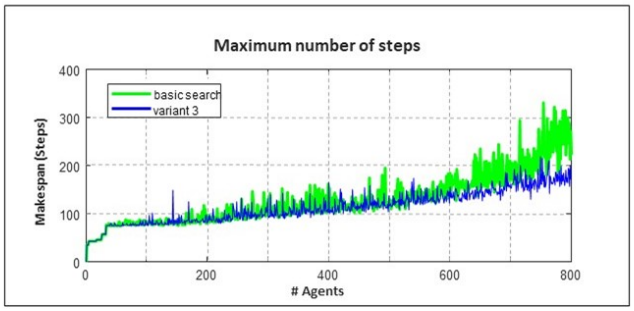
When applying these constraints to more complex domains, the performance tends to decrease. As we can see from Figures 7.15 and 7.16, when the number of agents is reduced, for what we said above, the two lines are very close to each other. When we increase them, the basic method is preferable.

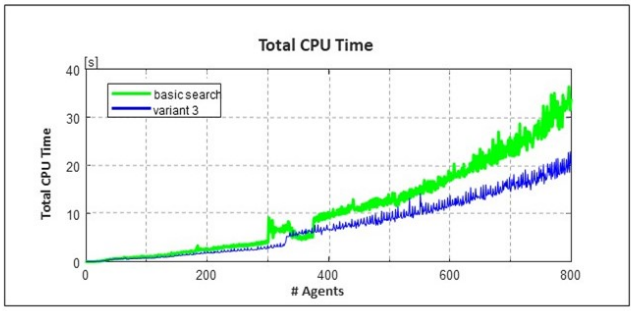

Number of avoided conflicts
×10^4
3
2
1
0
# Avoided conflicts
basic search
variant 3
0
200
400
600
800
# Agents


Number of A* method calls
×10^4
6
4
2
0
# A* calls
basic search
variant 3
0
200
400
600
800
# Agents


Sum of individual steps
×10^4
8
6
4
2
0
Sum of Steps
basic search
variant 3
0
200
400
600
800
# Agents

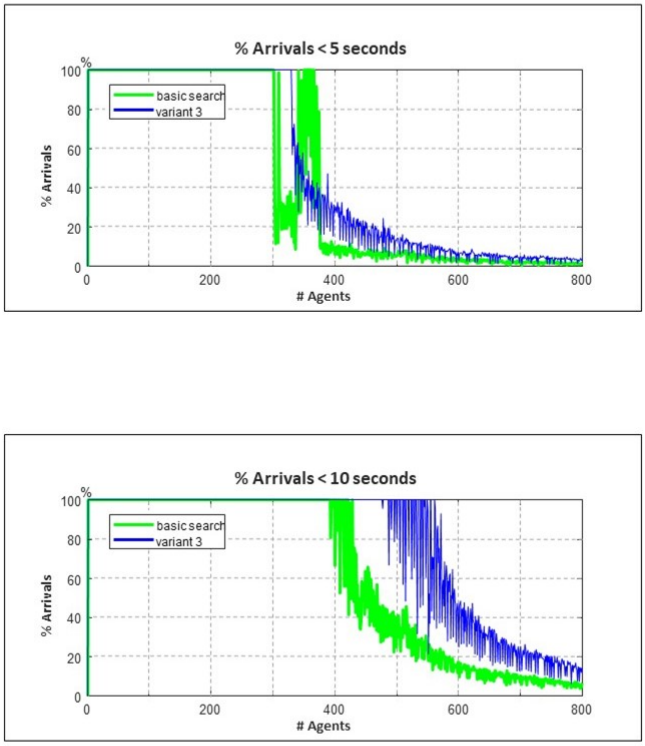
(a)



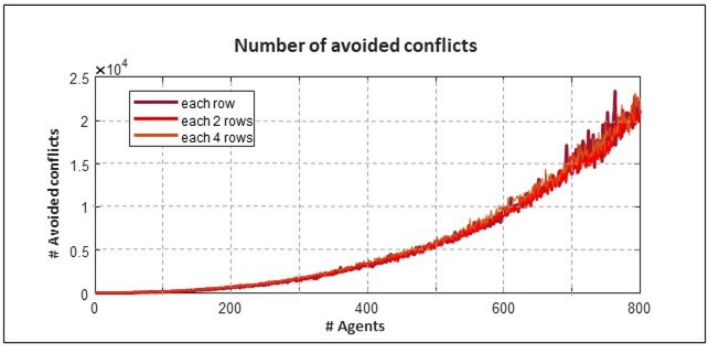

(b)

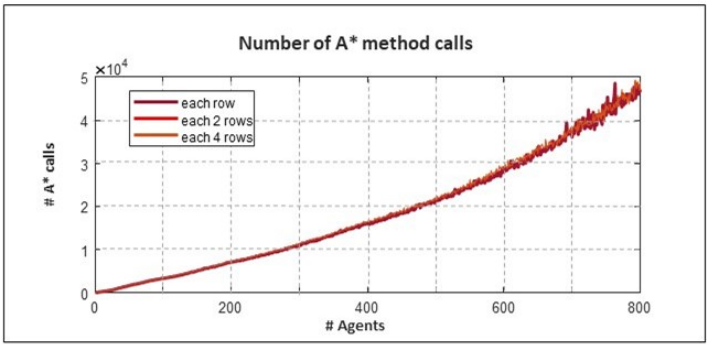
By looking at the performance for the Maze grid (Figure 7.16), the tendency is clear: for a few agents, i.e., up to 400 skids, the performance is the same. When the number of cart agents is over 600, the *basic search* outperforms the *variant 3*, especially for the sum of individual steps. The relation in terms of time used varies with the number of agents. This confirms the high dependency on the grid's structure and the agents' paths.

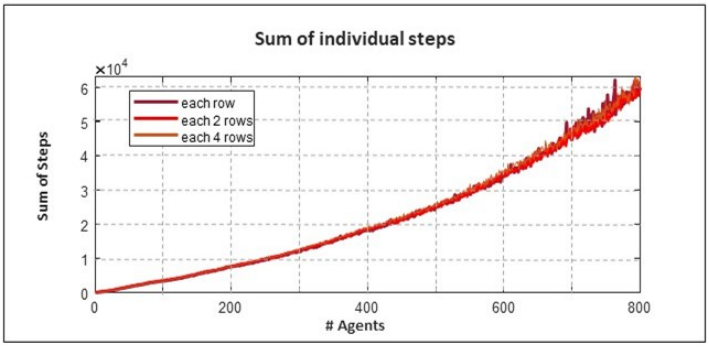



(c)

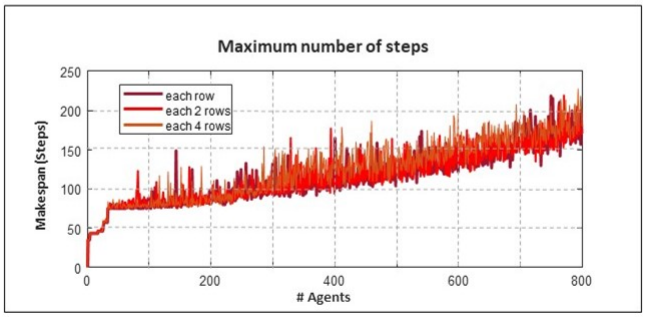
Figure 7.13: Comparison between *basic search* and *variant 3*. We consider the $48 \times 48$ empty grid. On the $x$-axis, we have the number of agents (from 1 to 800), while on the $y$-axis we show 7 different performance measures.

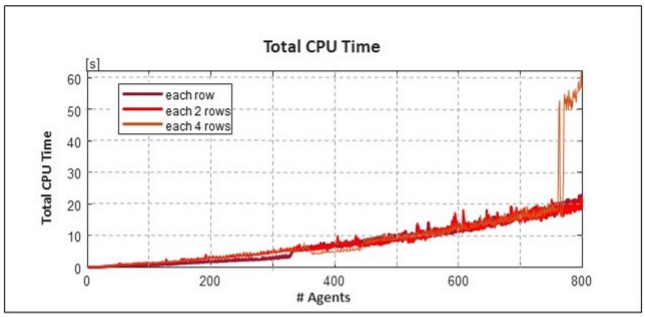

Number of avoided conflicts
×10^4
2.5
2
1.5
1
0.5
0
# Avoided conflicts
each row
each 2 rows
each 4 rows
0
200
400
600
800
# Agents


Number of A* method calls
×10^4
5
4
3
2
1
0
# A* calls
each row
each 2 rows
each 4 rows
0
200
400
600
800
# Agents


Sum of individual steps
×10^4
6
5
4
3
2
1
0
Sum of Steps
each row
each 2 rows
each 4 rows
0
200
400
600
800
# Agents

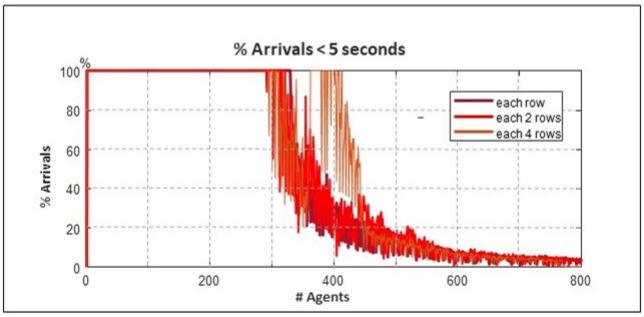
(a)

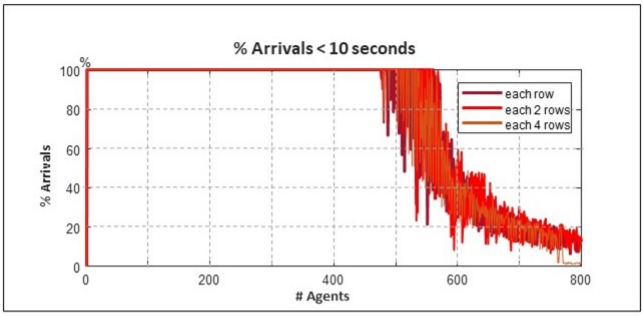



(b)

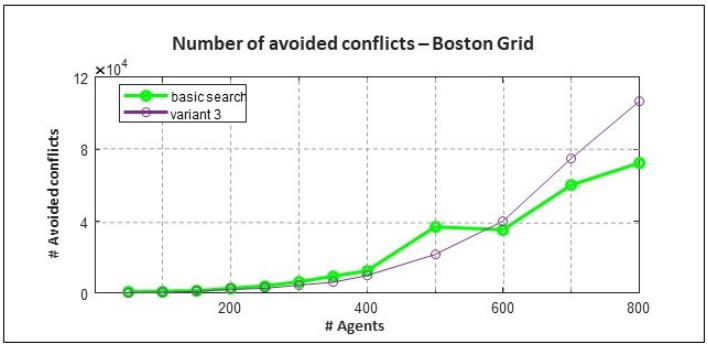
In *conclusion*, we discard this possible improvement because our objective is to analyze powerful methods for generic grids and scenarios.

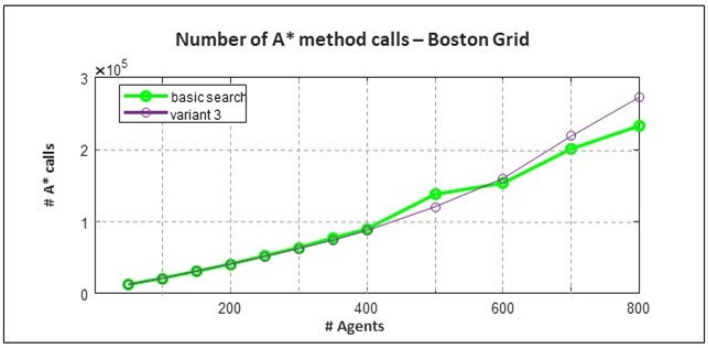


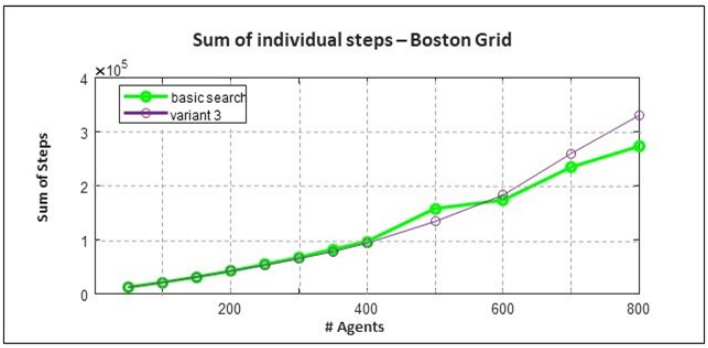




(c)

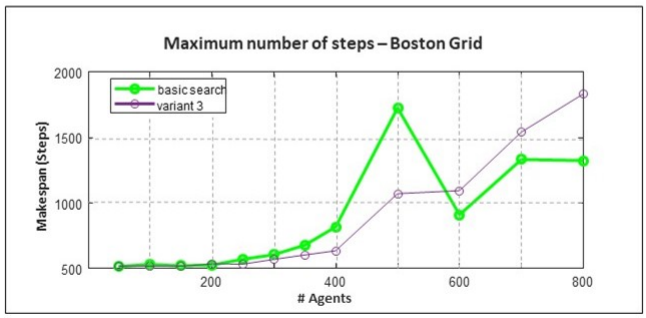
Figure 7.14: Comparison between *variant 3* and two other versions with different topological constraints. We consider the $48 \times 48$ empty grid. On the $x$-axis, we have the number of agents (from 1 to 800), while on the $y$-axis we show 7 different performance measures.

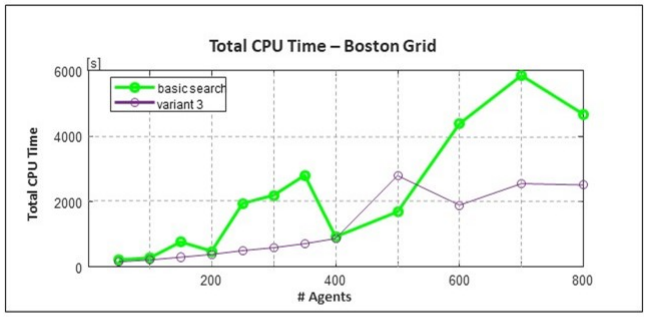

Number of avoided conflicts – Boston Grid
×10^4
12
8
4
0
# Avoided conflicts
basic search
variant 3
200
400
600
800
# Agents


Number of A* method calls – Boston Grid
×10^5
3
2
1
0
# A* calls
basic search
variant 3
200
400
600
800
# Agents


Sum of individual steps – Boston Grid
×10^5
4
3
2
1
0
Sum of Steps
basic search
variant 3
200
400
600
800
# Agents

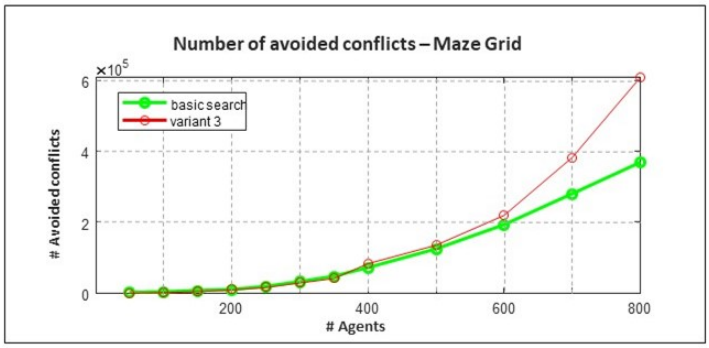
(a)

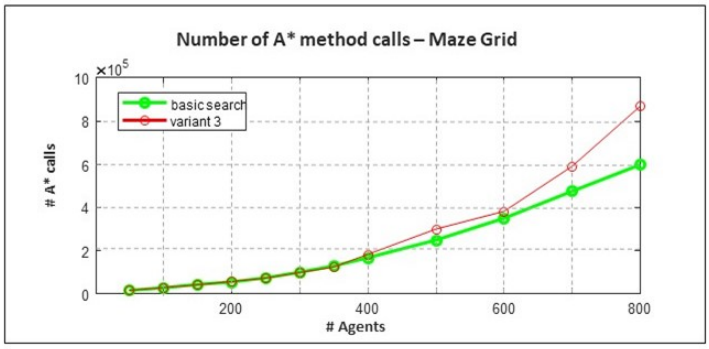


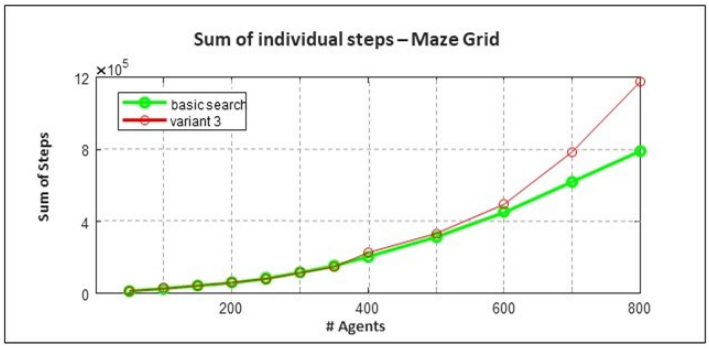




(b)

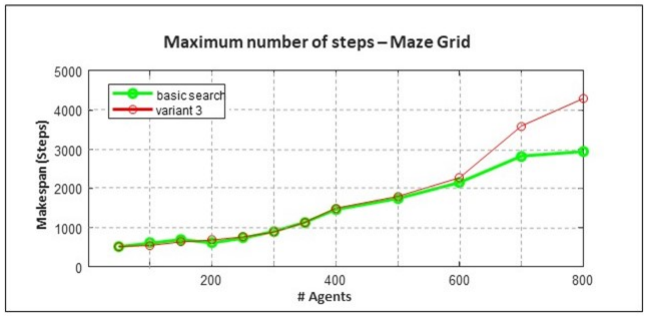
Figure 7.15: Comparison between *basic search* and *variant 3*. We consider the $256 \times 256$ Boston grid. On the $x$-axis, we have the number of agents (from 50 to 800, with stepsize 50), while on the $y$-axis we show 5 different performance measures.

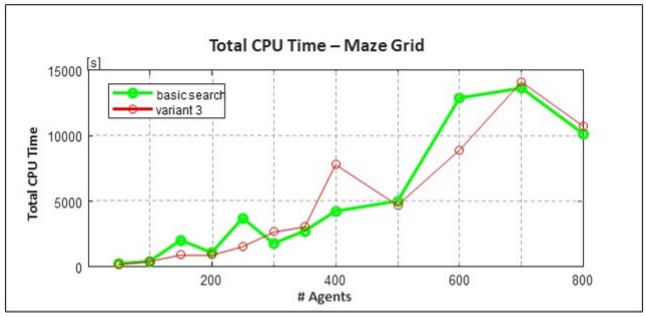

Number of avoided conflicts – Maze Grid
×10^5
6
4
2
0
# Avoided conflicts
basic search
variant 3
200
400
600
800
# Agents


Number of A* method calls – Maze Grid
×10^5
10
8
6
4
2
0
# A* calls
basic search
variant 3
200
400
600
800
# Agents


Sum of individual steps – Maze Grid
×10^5
12
8
4
0
Sum of Steps
basic search
variant 3
200
400
600
800
# Agents

(a)




(b)

Figure 7.16: Comparison between *basic search* and *variant 3*. We consider the $128 \times 128$ Maze grid. On the $x$-axis, we have the number of agents (from 50 to 800, with stepsize 50), while on the $y$-axis we show 5 different performance measures.

### 7.2.5 Experimental analysis: Basic Search + Congestion Awareness (Variant 4)

**Variant 4 on 48x48 empty grid**

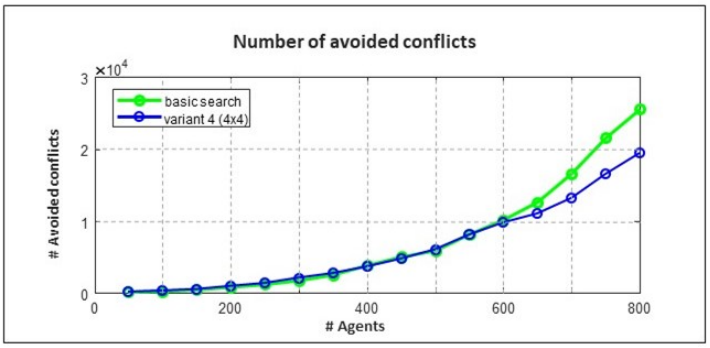
We impose costs on the grid's locations to let agents pass through less uncongested areas. As we expected, simulations show a reduction in the number of conflicts but a higher computational cost.

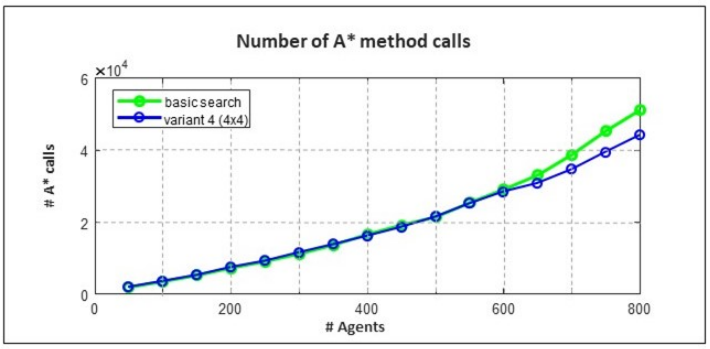
We first consider the comparison with the basic search and the optimized version where we use a $4 \times 4$ *macro grid* [6.2.4] (Figure 7.17). For a small number of agents on the $48 \times 48$ empty grid, the difference between the simple approach and the case with congestion awareness is negligible. The explanation is that with few skids, the number of conflicting paths is reduced. The same holds for the number of conflicts and the sum of time steps.
While incrementing the number of skids, we have the formation of congested areas and many more conflicts. *Congestion awareness pushes agents to isolated locations* (thanks to weights on edges). So skids try to maximize their distance between each other, and at the same time, they minimize their path to the final vertex. By looking for less crowded areas, the collisions risk is highly reduced. This justifies why, by increasing the number of agents, the *amount of possible conflicts* for the *variant 4* is smaller than for the *basic search*.

Due to the higher # avoided conflicts, by proceeding without congestion awareness the solution has a higher sum of individual steps too. Every time two agents are about to collide, they change the node, augmenting their total path length. So, regarding the *sum of individual steps*, the difference between the two algorithms depends on the different numbers of avoided collisions.
Consequently, also the *number of A* calls* changes after 600 agents.

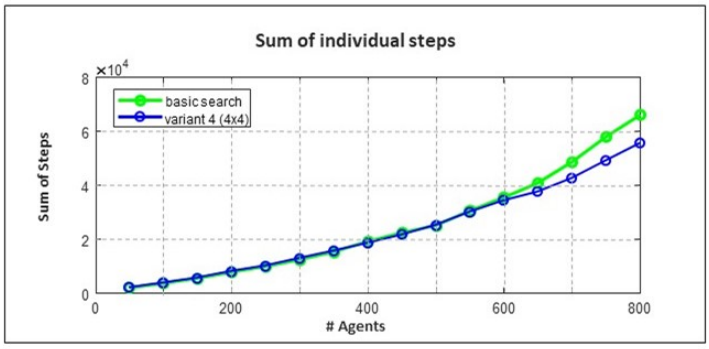
In conclusion, despite the higher time consumption, applying congestion awareness improves the performance of the first basic technique [4].

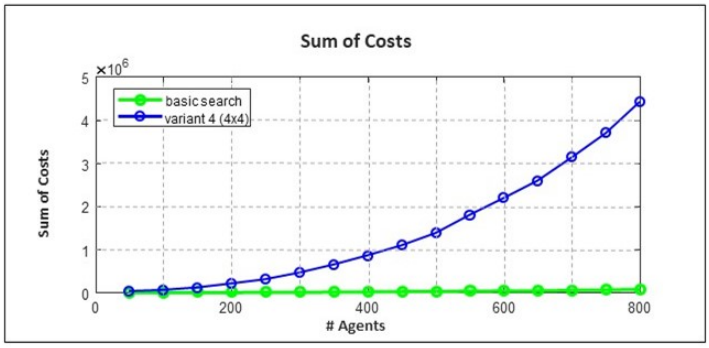
[4] In Figures 7.17b and 7.17c, we present the evolution of the *sum of costs* and the *maximum cost*, just for let the reader understand the difference between the sum of costs and the sum of time steps. Anyway, none of the benchmark grids is weighted. The costs we impose do not exist, they are part of the congestion awareness framework.


Number of avoided conflicts
×10^4
3
2
1
0
basic search
variant 4 (4x4)
# Avoided conflicts
0
200
400
600
800
# Agents

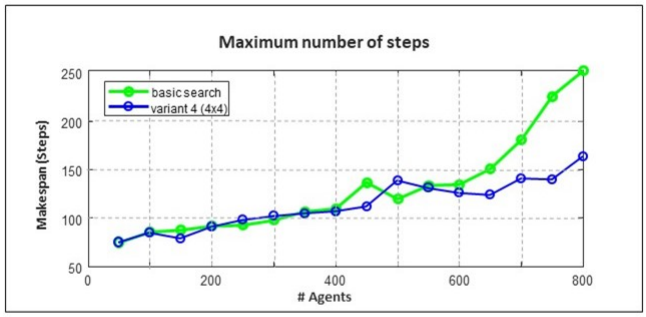

Number of A* method calls
×10^4
6
4
2
0
basic search
variant 4 (4x4)
# A* calls
0
200
400
600
800
# Agents

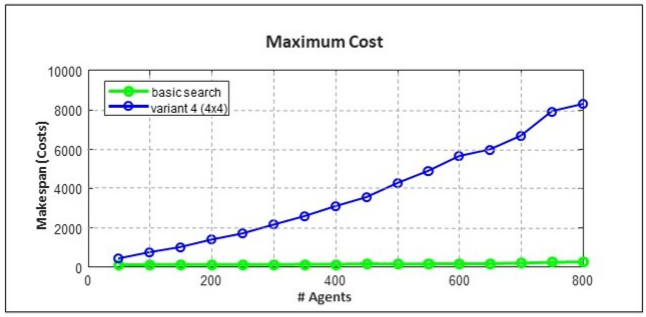
(a)

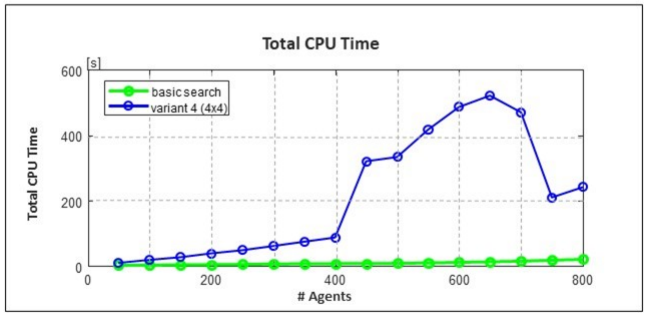

Sum of individual steps
×10^4
basic search
variant 4 (4x4)
Sum of Steps
# Agents
0
2
4
6
8
0
200
400
600
800


Sum of Costs
×10^6
basic search
variant 4 (4x4)
Sum of Costs
# Agents
0
1
2
3
4
5
0
200
400
600
800

(b)




(c)

Figure 7.17: Comparison between *basic search* and *variant 4* with Macro Grid size $4 \times 4$. We consider the $48 \times 48$ empty grid. On the $x$-axis, we have the number of agents (from 50 to 800, with stepsize 50), while on the $y$-axis we show 7 different performance measures.

### Variant 4 on 48x48 empty grid: Comparison between performances using different macro grid dimensions

We focus now on the problem of studying the effect of the macro grid dimension. In Figure 7.18, on 7 different variables, we compare the same optimized version for different macro grids dimensions: $4 \times 4$, $8 \times 8$, $16 \times 16$, $24 \times 24$, $48 \times 48$.

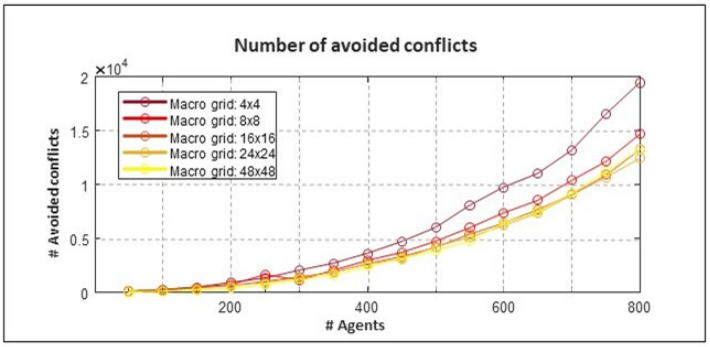
Increasing the macro grid dimension improves the results, especially the *number of avoided conflicts*. Nevertheless, the gap between two *"consecutive"* lines gets smaller. There is not much improvement when we pass from $24 \times 24$ to $48 \times 48$. In other words, the performance is similar if we consider a macro cell as a $2 \times 2$ or a $1 \times 1$ square (i.e., a single node).
To let the reader capture this aspect of the different dimensions, we show in Table 7.6 some values for each type of macro grid, giving the percentage of improvement on other versions.

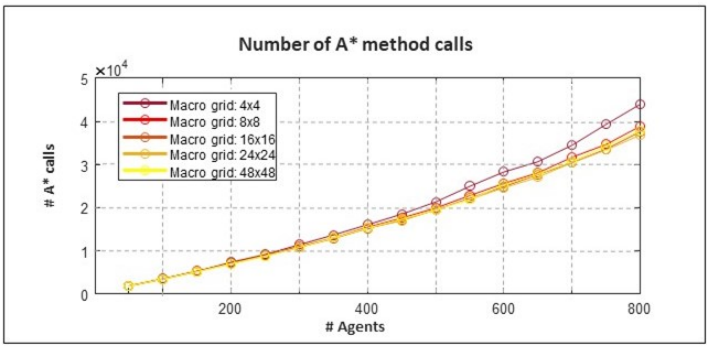
| Avoided conflicts | | | | | | |
|---|---|---|---|---|---|---|
| | Empty Grid | | | | | |
| # Agents | No Congestion Awareness | Congestion Awareness Macro Grid: 4x4 | Congestion Awareness Macro Grid: 8x8 | Congestion Awareness Macro Grid: 16x16 | Congestion Awareness Macro Grid: 24x24 | Congestion Awareness Macro Grid: 48x48 |
| | Value | % | % | % | % | % |
| 100 | 157 | 134.4% | 87.9% | 79.6% | 58.6% | 43.9% |
| 200 | 731 | 118.6% | 87.8% | 77.0% | 76.1% | 51.2% |
| 300 | 1669 | 129.7% | 64.8% | 82.0% | 75.7% | 71.4% |
| 400 | 3767 | 95.2% | 77.2% | 69.2% | 68.8% | 63.7% |
| 500 | 5834 | 102.5% | 80.5% | 71.1% | 71.9% | 66.2% |
| 600 | 10139 | 96.1% | 72.7% | 63.3% | 61.1% | 63.6% |
| 700 | 16624 | 79.4% | 62.7% | 54.7% | 54.4% | 55.4% |
| 800 | 25590 | 76.2% | 57.7% | 52.1% | 48.8% | 51.9% |
| 900 | 38440 | 68.9% | 48.4% | 44.9% | 46.0% | 45.7% |

Table 7.6: Comparison between *basic search* and *variant 4* for 9 different values of the number of conflicts. Data come from the empty grid scenarios. Percentage columns represent the values ratio (%) between the *variant 4* and the *basic search*. 5 different cases macro grid dimensions are considered.

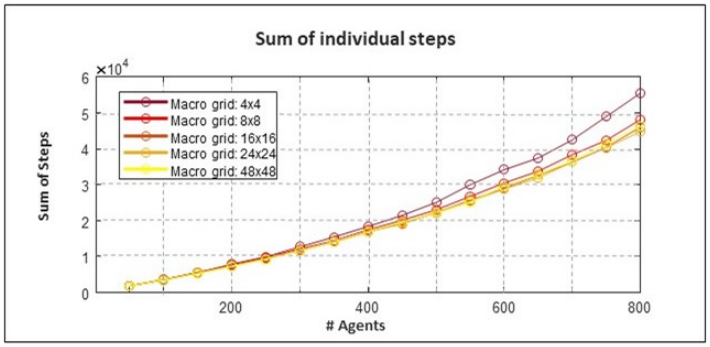
Comparing the charts of # avoided conflicts (Figure 7.18a) and of the *sum of individual steps* (Figure 7.17b), we observe a smaller improvement in the latter case. This is correct because the price we pay when avoiding congested areas is a longer path.

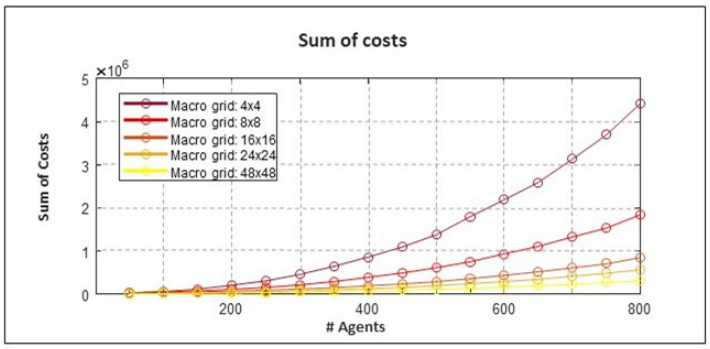
Interestingly, we can consider the *basic search* as a particular case of the *variant 4*, where the macro grid has dimension $1 \times 1$. Increasing the weights at the same time for every cell doesn't change the performance. This is another proof for choosing macro grids with a higher number of nodes.


Number of avoided conflicts
×10^4
2
1.5
1
0.5
0
# Avoided conflicts
Macro grid: 4x4
Macro grid: 8x8
Macro grid: 16x16
Macro grid: 24x24
Macro grid: 48x48
200
400
600
800
# Agents

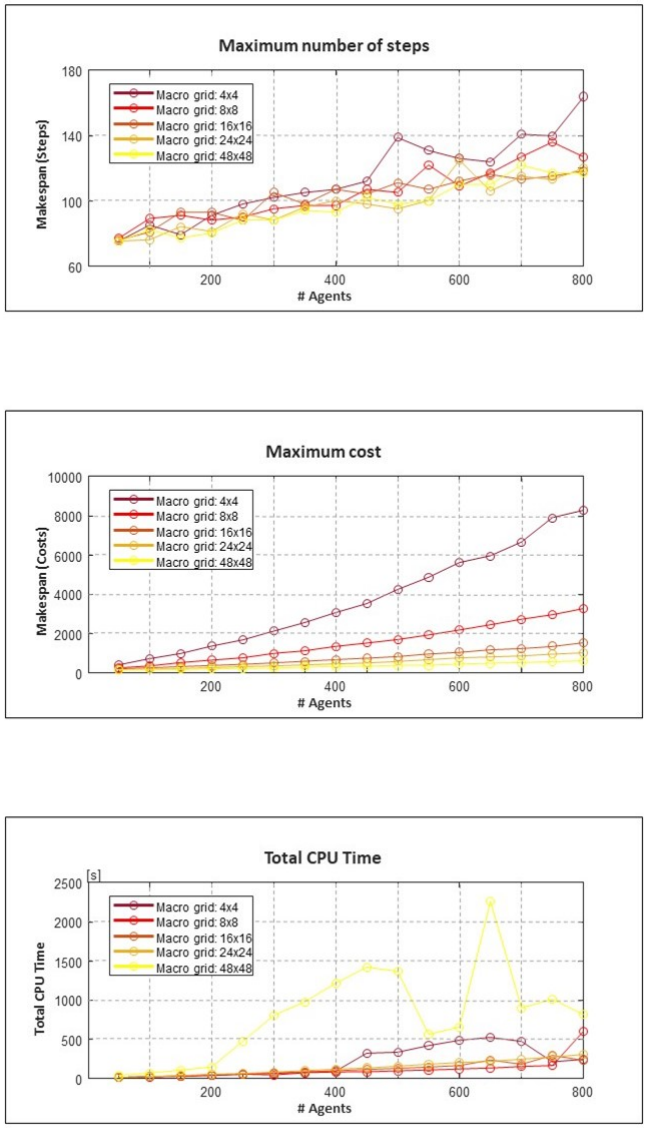

Number of A* method calls
×10^4
5
4
3
2
1
0
# A* calls
Macro grid: 4x4
Macro grid: 8x8
Macro grid: 16x16
Macro grid: 24x24
Macro grid: 48x48
200
400
600
800
# Agents

(a)


Sum of individual steps
×10^4
Sum of Steps
# Agents
Macro grid: 4x4
Macro grid: 8x8
Macro grid: 16x16
Macro grid: 24x24
Macro grid: 48x48


Sum of costs
×10^6
Sum of Costs
# Agents
Macro grid: 4x4
Macro grid: 8x8
Macro grid: 16x16
Macro grid: 24x24
Macro grid: 48x48

(b)




(c)

Figure 7.18: Performance of *variant 4* with different Macro Grid sizes. We consider the $48 \times 48$ empty grid. On the $x$-axis, we have the number of agents (from 50 to 800, with stepsize 50), while on the $y$-axis we show 7 different performance measures.

In *conclusion*, by increasing the dimension of the macro grid abstraction, we have better performances that tend to coincide when this dimension is close to the grid's dimension.

The problem of congestion awareness is the amount of time needed to update the macro grid. This cost is higher with the dimension of the macro grid. Complex benchmark grids require a too high amount of time for our analyses. Hence we don't take them into account. As we will see in [8.1.1], with appropriate improvements to the algorithm's speed, we can apply congestion awareness to such kinds of grids too.

### 7.2.6 Experimental analysis: Basic Search + New Conflict Avoidance based on Agents' Distance from Origin/Destination (Variant 5)

#### Variant 5 on 48x48 empty grid

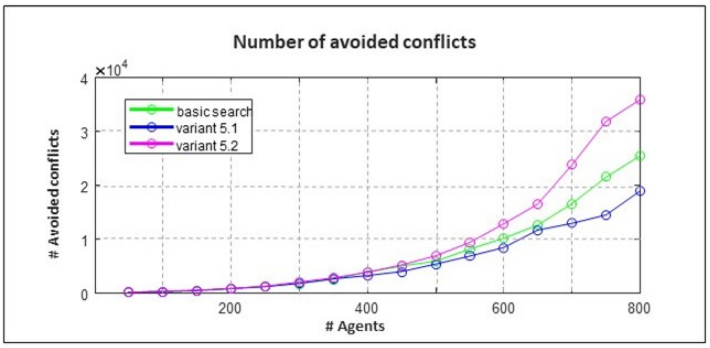
Now we analyze the performance when we use another collision avoidance technique. *Variant 5* is subdivided into two specific approaches:

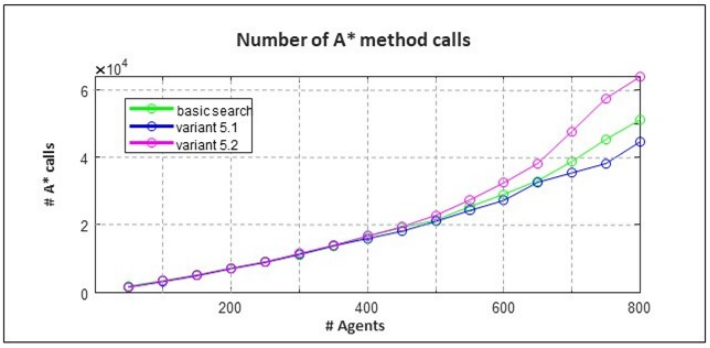
- *Variant 5.1*: when two agents are about to collide, the one *CLOSER* to its destination has the priority to occupy the other cell that the other agent necessarily has to leave.
- *Variant 5.2*: when two agents are about to collide, the one *FURTHER* from its destination has the priority to occupy the other cell that the other agent necessarily has to leave.

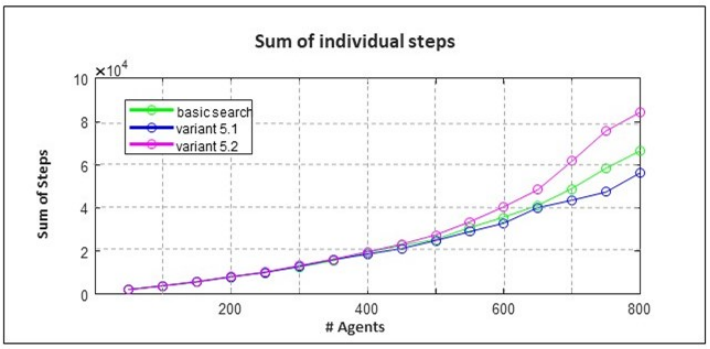
We implement a *deterministic check* to choose the passing agent, instead of a random one, by imposing the closest or furthest agent to pass. This simple difference from the code programming point of view translates into huge differences in terms of performance. By *removing randomness* we are *constraining* the agents' motion, often bringing undesirable results [8.2.1].

Figure 7.19 shows the performance on the $48 \times 48$ empty grid for agents going from 50 to 800, incrementing by 50, but simulations showed that even with this grid the skids may get stuck in deadlocks. We might expect the two alternative policies to show almost identical results because depending on their distance from the goal, the same agent may be favored or not. Potentially, every agent can be the leader during a possible conflict. As a consequence, we might conclude the number of avoided conflicts doesn't depend on the collision avoidance technique we used. Interestingly, the performance charts confirm the opposite: methods' performance is not always balanced.

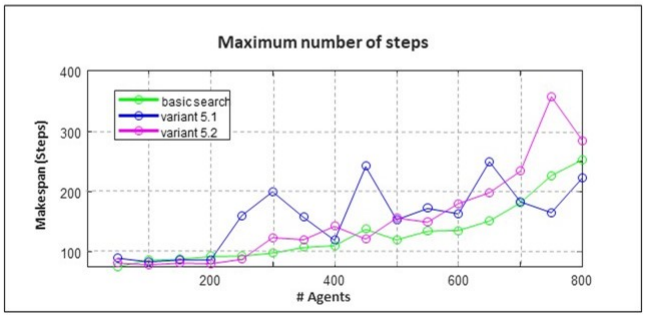
Concerning the *number of avoided conflicts* and *A* method calls*, and the *sum of individual time steps*, results are similar when considering less than 600 agents. After this, *variant 5.1* outperforms both *basic search* and *variant 5.2*, that is the worst.

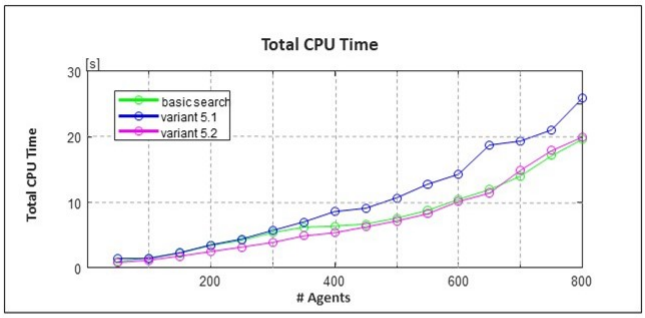

Number of avoided conflicts
×10^4
4
3
2
1
0
# Avoided conflicts
basic search
variant 5.1
variant 5.2
200
400
600
800
# Agents


Number of A* method calls
×10^4
6
4
2
0
# A* calls
basic search
variant 5.1
variant 5.2
200
400
600
800
# Agents


Sum of individual steps
×10^4
10
8
6
4
2
0
Sum of Steps
basic search
variant 5.1
variant 5.2
200
400
600
800
# Agents

(a)




(b)

Figure 7.19: Comparison between *basic search*, *variant 5.1*, and *variant 5.2*. We consider the $48 \times 48$ empty grid. On the $x$-axis, we have the number of agents (from 50 to 800, with stepsize 50), while on the $y$-axis we show 5 different performance measures.

On the other hand, from the perspective of the *CPU time consumption*, *variant 5.1* is slightly worse than the other two algorithms. Likely, encouraging those agents closer to their goals in solving conflicts pushes them towards their final nodes. Oppositely, agents extend their paths with the second policy, requiring more time steps to conclude their job.

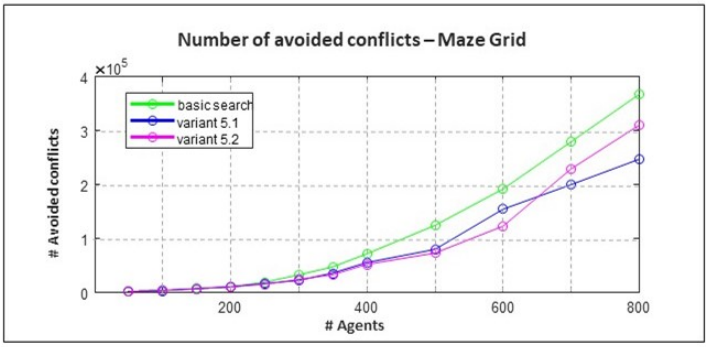
*Variant 5.2* has worse outcomes than the basic randomized conflict avoidance technique. Hence, we prefer not to use this version of *variant 5*.

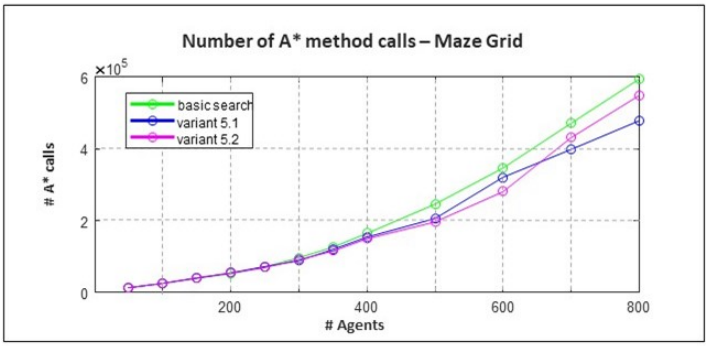
### Variant 5 on other benchmark grids

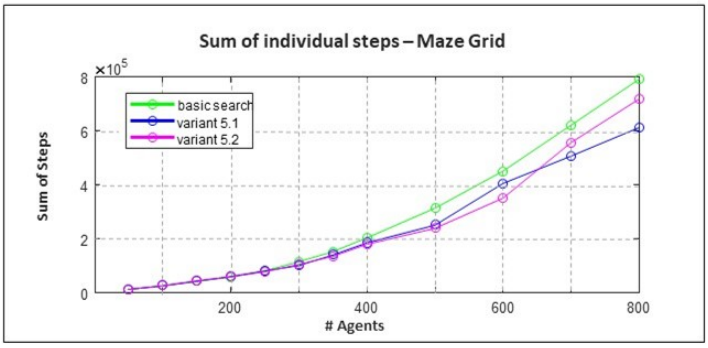
We want to confirm the advantage of using *variant 5.1* over *variant 5.2*, also in the case of the Boston grid and the maze domain. Boston's scenario is too constrained to be solved with these deterministic approaches. Even with 150 agents ($\approx 0.3\%$ of the grid's free cells), there are unresolved loops. Luckily, the Maze grid is simpler, and we show this in Figure 7.20.

Oppositely to the empty grid, for the Maze grid's scenario, the *basic search* is the worst method regarding avoided conflicts, A* calls, and the sum of steps. The two versions of *variant 5* present similar results, and only when passing from 600 to 800 agents the first type outperforms the second one. Hopefully, the gap will increase even more with the number of agents.

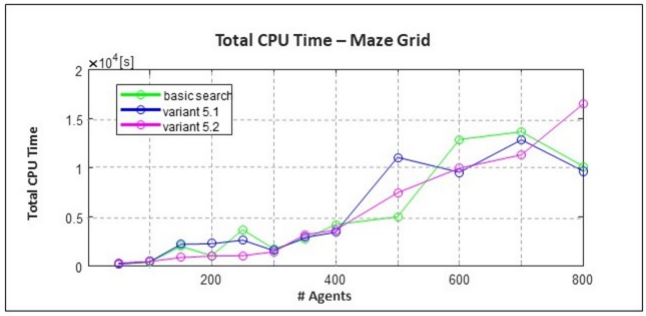
In *conclusion*, *variant 5.1* is the best method between these 3 ones. But we cannot consider it the best technique because pathological situations may form not only for complex grids but also for the simplest empty environment.


Number of avoided conflicts – Maze Grid
×10^5
# Avoided conflicts
# Agents
basic search
variant 5.1
variant 5.2


Number of A* method calls – Maze Grid
×10^5
# A* calls
# Agents
basic search
variant 5.1
variant 5.2


Sum of individual steps – Maze Grid
×10^5
Sum of Steps
# Agents
basic search
variant 5.1
variant 5.2

(a)


(b)

Figure 7.20: Comparison between *basic search*, *variant 5.1*, and *variant 5.2*. We consider the $128 \times 128$ Maze grid. On the $x$-axis, we have the number of agents (from 50 to 800, with stepsize 50), while on the $y$-axis we show 5 different performance measures.

## 7.3 Comparison & Conclusion

We analyzed the performances of our basic search method and 5 variants on different benchmark grids in terms of experimental number of avoided conflicts, number of A* search method calls, sum of time steps, maximum number of required steps, speed, and time consumption.

These variables are highly correlated with each other and guarantee the full comprehension of the algorithms' behavior while augmenting the number of agents in the system. In Figures 7.21 and 7.22 we show the *Correlation Matrix* for our variables concerning the *basic search* data, using both the *Pearson correlation* and the *Spearman correlation*. [5]

In particular, the *number of avoided conflicts* is a fundamental variable linked to all the others.

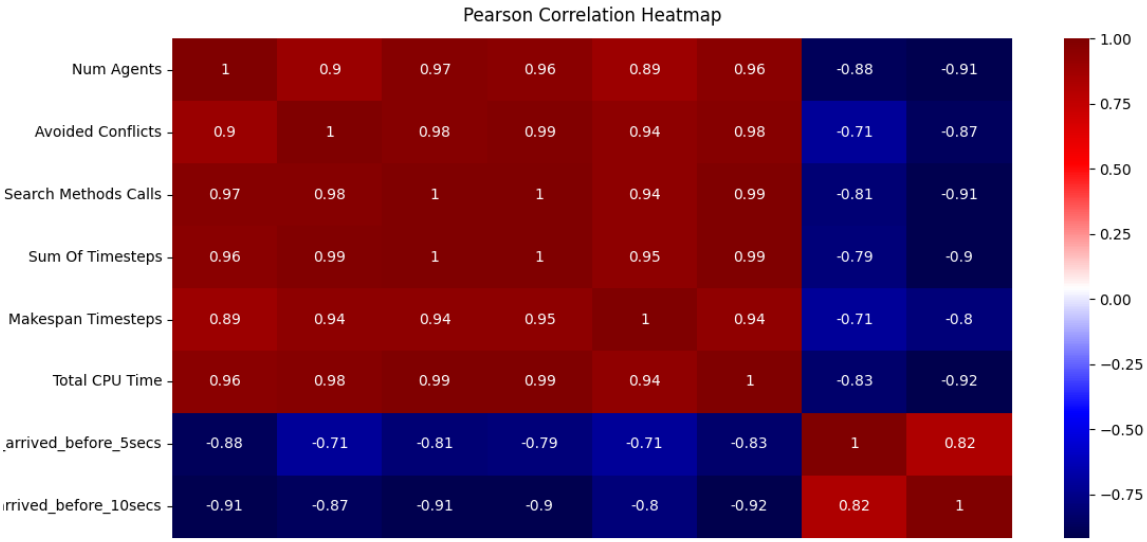



Figure 7.21: Pearson Correlation Matrix for the *basic search* data.

[5] In Statistics, the ***Pearson correlation coefficient*** is a measure of *linear correlation* between two sets of data. The ***Spearman correlation coefficient*** is a measure of correlation that assesses how well the relationship between two variables can be described using a *monotonic function*. A Spearman correlation of 1 results when the two variables being compared are monotonically related, even if their relationship is not linear (https://en.wikipedia.org/wiki/Pearson_correlation_coefficient)(https://en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient).

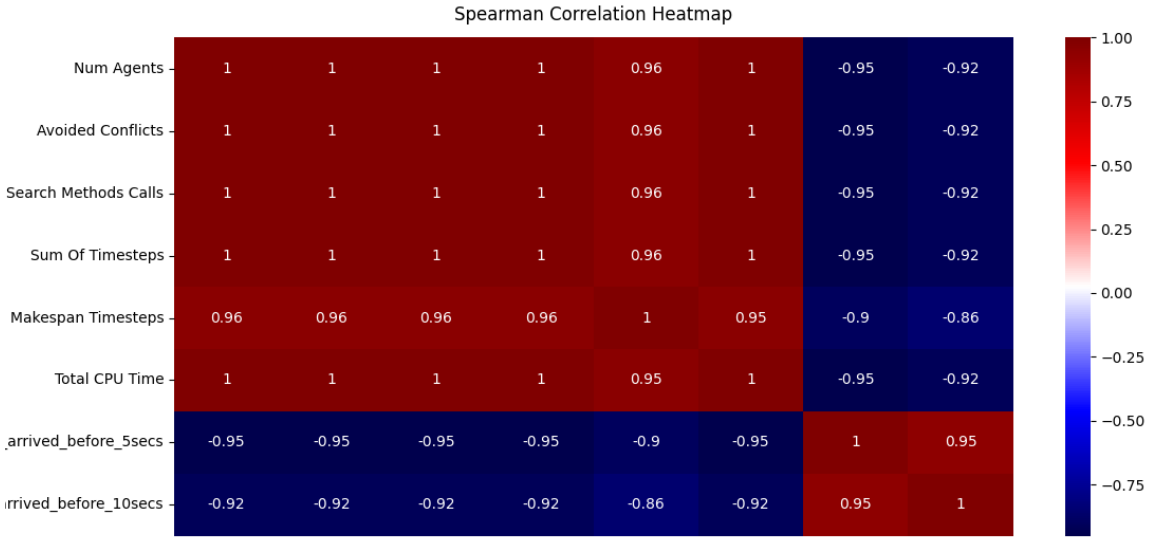



Figure 7.22: Spearman Correlation Matrix for the *basic search* data.

We summarize the experimental results.

- *Number of avoided conflicts.* The optimized pre-conflict avoidance technique, i.e., *variant 1* [6.2.1] has the worst effect on the number of avoided conflicts since by waiting for conflicts to be resolved an agent is implicitly risking to collide with others. The best results are obtained using the traffic directions on the grid (*variant 3*, [6.2.3]), congestion awareness (*variant 4*, [6.2.4]), and favouring agents closer to their goals (*variant 5.1*, [6.2.5]). They are better than *basic search* and the optimized version on the number of search method calls, i.e., *variant 2* [6.2.2]. Finally, favoring agents if they are further from the destination (*variant 5.2*) is worse because closer agents tend to remain in the system, causing new possible conflicts.

- *Number of A* method calls.* The problem of *variant 1* reflects on the number of A* search method calls, pretty much higher than any other developed algorithm. As for the case of number of avoided conflicts, *variant 3*, *variant 4*, and *variant 5.1* are better, but now the best results are obtained with *variant 2*, which was developed to optimize this variable as much as possible. Finally, *variant 5.2* is worse than *basic search*, but better than *variant 1.*

- *Sum of individual steps.* Trends are always the same: favoring agents closer to their goals, using traffic directions, or having congestion awareness bring the best results.

- *Maximum number of steps.* This variable depends on the agents' characteristics, i.e.,

start and goal locations, even more than the algorithms. Despite this, *variant 4* using congestion awareness brings the best results.

- *Total CPU time*. While its values depend on the CPU and the jobs the PC is doing, it helps to understand which are the fastest and slowest algorithms. In the empty grid, the worst results regard congestion awareness.

As a whole, we observe a similar evolution between the number of avoided conflicts, the number of A* search method calls, and the sum of individual steps. Indeed, there is a pattern between them, as we observe in Figure 7.23.
The other two variables have different behavior, but this is reasonable. The makespan depends primarily on the distance origin-destination. The CPU Time, on the other hand, depends on the microprocessor type.

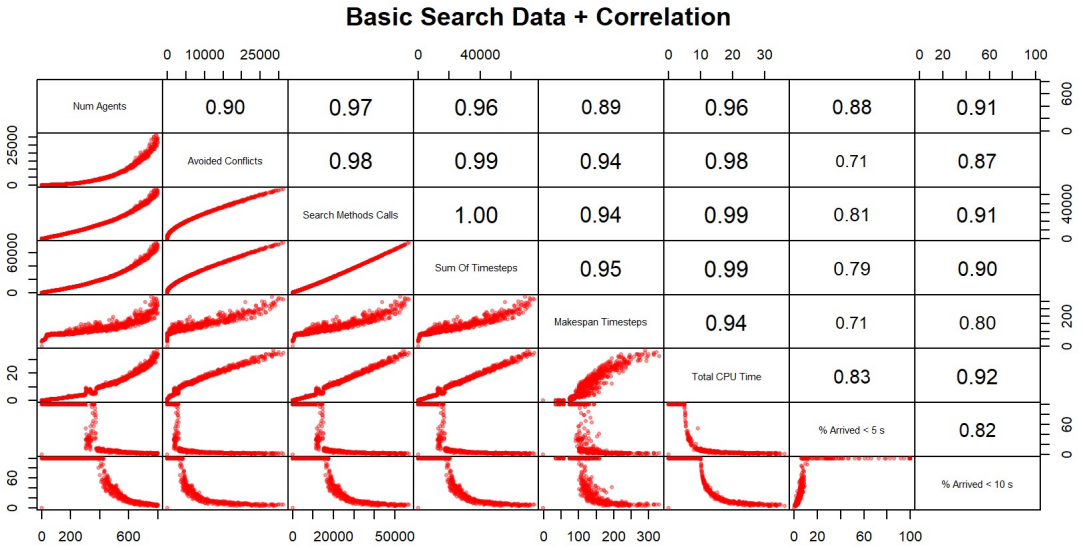



Figure 7.23: $48 \times 48$ empty grid basic search data. On the lower left half different variables are visually compared. On the upper right part variables are compared measuring the Pearson Correlation.

The different performances are clear if we consider more than 600 agents in the system. We can explain this as follows.
Considering the empty grid with less than approximately 600 agents, i.e., less than 25% of free cells are occupied by skids, the domain is sufficiently sparsely populated so that the number of avoided conflicts is small, irrespective of the efficiency of the considered algorithm. This changes when congested areas increase, especially when the percentage of skids in the system gets around $40 - 50\%$ (the maximum percentage we analyzed for the empty grid).

The percentage of the cells occupied by the agents influences also the variables' variance. In particular, all graphical representations showed a higher variability while increasing the number of agents. This variability seems to increase even more after reaching 600 agents. Let us make an example considering the *basic search* results [7.2.1]. In Figure 7.24, we consider the evolution of the number of avoided conflicts, which we said is one of the most important variables.

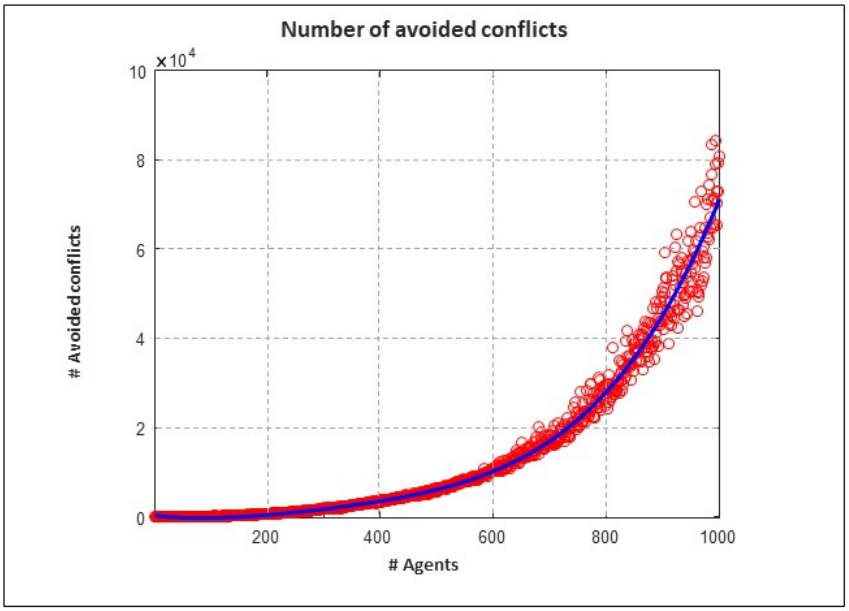



Figure 7.24: Number of avoided conflicts for the *basic search*. Data are shown in red circles. The least-squares polynomial (degree 4) is shown in blue. The number of agents in the system is shown on the $x$-axis, while on the $y$-axis the number of avoided conflicts.

The standard deviation from the least-squares polynomial of degree 4 is shown in Figure 7.25. Here, the variability increases with the higher number of agents.

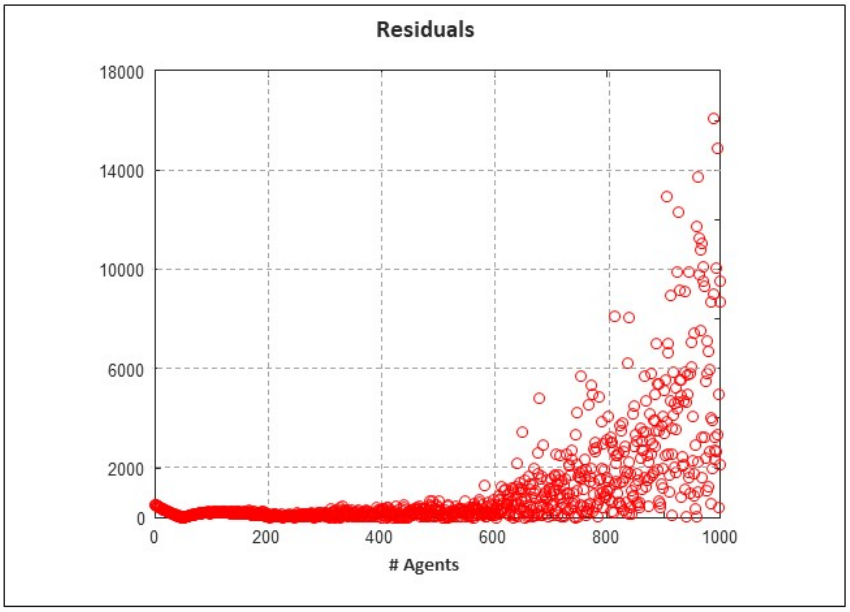



Figure 7.25: Residuals of *basic search* number of avoided conflicts' data used to fit the least-squares polynomial (degree 4). Data are shown in red circles. The number of agents in the system is shown on the $x$-axis, while on the $y$-axis the residual values.

The high variability is caused by the *randomness* in the origin-destination distance applied to the *random* permutation of the agents at each step of the process. When passing from $n > 0$ to $n+1$ agents, we include a path whose length is short or long, depending on the origin-destination distance. Since benchmark scenarios contain randomly generated agents, agent $n+1$ may have a short path connecting its origin to the destination, while agent $n+2$ could require a higher number of time steps. Interestingly, there is no dependency here on the number of agents. For this reason, even on the left-hand side of the variables charts, the spiky trend occurs, but with a smaller order of magnitude. When there are many agents, dense areas and possible conflicts are frequent. Due to the local conflict avoidance, agents' overall path is quite longer than with few skids. The presence of a new agent in a crowded area could influence positively or negatively the path length of the involved skids, bringing them to shorter or longer paths.

This explains the higher deviations observed in Figure 7.25. Even though the overall trend is the same, we observe a dependency on the seed too, as from Figure 7.2, where different seeds bring different spikes.

The experimental comparison confirmed what we observed while describing our models.

- *variant 1* has two main drawbacks. First, as for the basic search, the path is re-planned each time, implying an unnecessary huge amount of search function calls. Second, the sum of time steps is too high for the basic search. These drawbacks reflect on the time consumption.
- The optimization on the number of search method calls, i.e., *variant 2*, is useful to reduce A* function calls, but apart from this, the other variables remain unchanged. It should be appropriately combined with other variants.
- It is not possible to improve the *basic search*'s performance by just imposing some general constraints on the domain, as we have done for *variant 3*. We need specific information about the grid and the agents' starting and final positions. Without this further knowledge, our variant is not as effective as it could be potentially. Our attention is to methods' application to generic grids and scenarios.
- Congestion awareness (*variant 4*) pushes agents to isolated locations. So, by looking for less crowded areas, the collisions' risk is highly reduced. This justifies why, by increasing the number of agents, for the *variant 4* there are fewer avoided conflicts than for the basic search. Congestion awareness is useful for those algorithms where we may incur pathological situations of deadlocks, with crowded areas occupied by agents that cannot move away. Indeed, the advantage of congestion awareness is it reduces the number of possible conflicts. This also implies a reduction in the number of A* calls and the sum of individual steps. Augmenting the macro grid dimension helps in general. When this is almost equal to the original grid dimension, also smaller grids have good performances. The only drawback of congestion awareness is the amount of time needed to update the macro grid but with appropriate improvements to the algorithm's speed [8.1.1], we can apply congestion awareness even to complex benchmark domains.
- The major drawback of *variant 5* is it may bring deadlocks formation (a chain/group of agents not leaving a crowded area) not only for complex grids but also for the simple empty environment. They fail where other methods don't.

It is interesting to analyze the effect of *combining congestion awareness and # search calls optimization* since taken individually, they both enhance the basic version, the first one in terms of the sum of time steps, the other one in terms of the number of A* calls. Note the basic search is equivalent to its *variant 4* where we consider a $1 \times 1$ macro grid dimension.
We will check the quality of the resulting method in the next chapter [8.2.1].

# Chapter 8

# Final Optimized Algorithm and Conclusions

## 8.1 Final Optimized Algorithm

### 8.1.1 Model Description

We develop a final optimized version of the basic search, exploiting both the computational optimization on search calls and the congestion awareness.

- *Computational optimization on the number of search method calls.* The shortest path is computed only at the beginning to let the process start and whenever a possible conflict is solved. We have seen that for *basic search* the number of search function calls is highly optimized [7.2.3].
- *Congestion awareness.* Agents try to pass through less uncongested areas. We memorize the areas where every skid's path passes and let agents know this when computing their shortest path. In particular, the grid nodes are grouped in *macro cells*, each one with a specific positive cost. Every mobile agent computes its shortest path using the costs of the *macro grid*. Then it updates the costs of the macro cells. Other agents then try to follow alternative paths with smaller costs [6.2.4].

Every agent chooses the path whose overall cost on the macro grid is minimized. Then, it keeps moving until the destination is reached or a possible collision occurs. In the latter case, the possible conflict is solved with the *random-style collision avoidance* approach [6.1.1], and the path is replanned. For each replanning, the macro grid is modified too.

### 8.1.2 Pseudo Code

The pseudo code of the method is shown in 8.1.2, while the summary is in Table 8.1.

**Algorithm 10** Pseudo Code: Basic Search + Congestion Awareness + Computation Optimization on Search Calls

```
build the set of obstacles (e.g. walls);
while skids are moving to their destinations do
    if new online skid agent is added to the skids' set then
        if possible, add the mobile agent to the corresponding starting cell;
    end if
    shuffle the agents' order;
    for skid agent = 1, ..., k do
        if the agent has not been considered yet then
            if agent position and goal coincide then
                remove the agent (disappear at target);
                modify the grid weights;
            else  (agent not at target → planning)
                consider the next possible position (from the pre-computed shortest path);
                if the next possible tile is occupied then
                    if the other agent has not been considered yet then
                        if possible, one of the two agents is randomly chosen and occupies the other cell (and previous tile becomes blank);
                        modify the grid weights, compute individual shortest path;
                        if agent position and goal coincide then
                            remove the agent (disappear at target);
                            modify the grid weights;
                        end if
                    else
                        wait;
                    end if
                else
                    cart agent occupies the available tile and previous tile becomes blank;
                    if agent position and goal coincide then
                        remove the agent (disappear at target);
                        modify the grid weights;
                    end if
                end if
            end if
        end if
    end for
end while
```

(Note the shortest path is built using A* algorithm)

| File | Generic benchmark file |
|---|---|
| Approach | Online |
| Solution | Sub-Optimal |
| Conflict Avoidance | Random-Style |
| Permutation | Yes |
| Topology Constraints | No |
| Notes | Congestion awareness through a macro grid + reduced number of search method calls |

Table 8.1: Properties summary of Final Optimized Version (Basic Search + Congestion Awareness + Computational Optimization on Search Calls).

## 8.2 Comparison

### 8.2.1 Comparison with previous algorithms

Figure 8.1 shows the comparison between all the developed algorithms. The last two lines on the legend correspond to the final optimized version with macro grid dimension $48 \times 48$ and $24 \times 24$, respectively.

This new version outperforms both the simple variant that optimizes the number of search calls, *variant 2* and the version that uses congestion awareness, *variant 4*.

Thanks to the congestion awareness, the *number of avoided conflicts* is reduced, with results better than for traffic (*variant 3*), congestion awareness (*variant 4*), and optimized collision avoidance (*variant 5.1*) approaches, that we said were better than the *basic search* from the point of view of # avoided collisions.

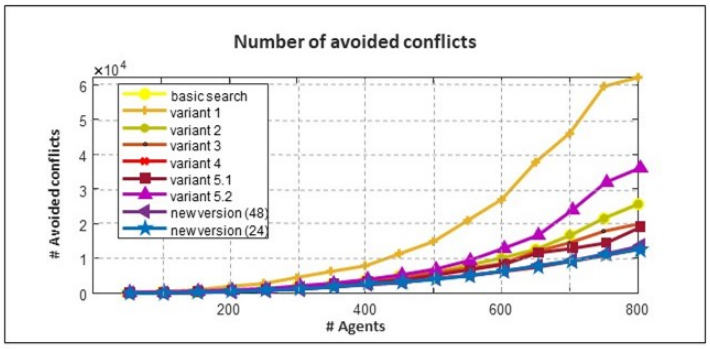
Concerning the *number of search calls*, congestion awareness and optimized computation of the search method calls together give better results than if taken singularly. The chart shows an improvement of more than three times the *basic search*.

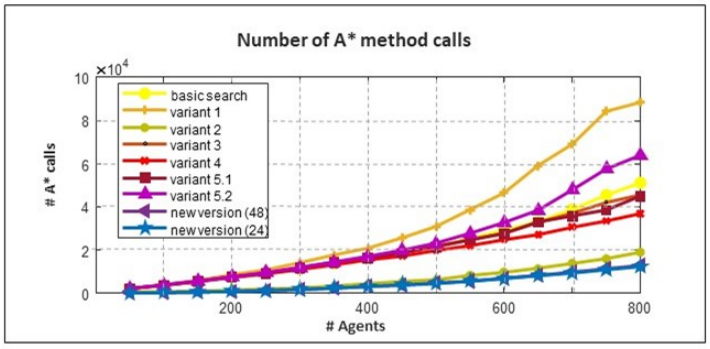
The *sum of individual steps* is consequently the best one with almost linear growth.

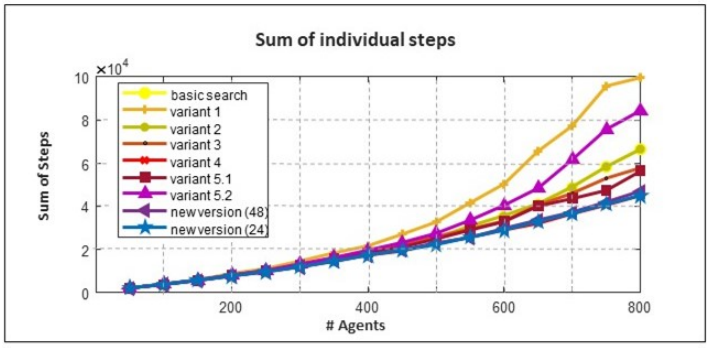
Interestingly, even the *maximum number of individual steps* to reach the destination is the smallest. This is due to congestion awareness, which guarantees agents don't have to replan many times and then can follow mainly their shortest path.

Finally, the only drawback of this new version is the quite high *CPU time consumption*. Despite this, we would always prefer this optimized version to the previous ones because of the advantages for the other variables, especially the sum of time steps.

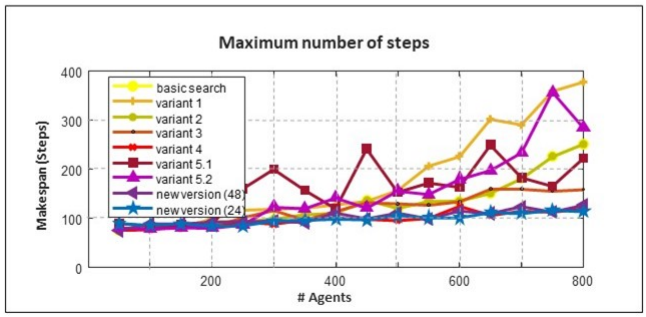
The basic search with optimized number of A* calls, i.e., *variant 2*, is equivalent to this optimized version with $1 \times 1$ macro grid dimension.

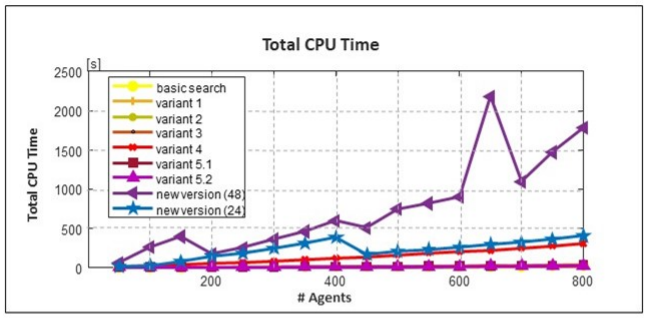

Number of avoided conflicts
×10^4
# Avoided conflicts
# Agents
basic search
variant 1
variant 2
variant 3
variant 4
variant 5.1
variant 5.2
new version (48)
new version (24)


Number of A* method calls
×10^4
# A* calls
# Agents
basic search
variant 1
variant 2
variant 3
variant 4
variant 5.1
variant 5.2
new version (48)
new version (24)


Sum of individual steps
×10^4
Sum of Steps
# Agents
basic search
variant 1
variant 2
variant 3
variant 4
variant 5.1
variant 5.2
new version (48)
new version (24)

(a)




(b)

Figure 8.1: Comparison between all the developed algorithms. We consider the $48 \times 48$ empty grid. On the $x$-axis, we have the number of agents (from 50 to 800, with stepsize 50), while on the $y$-axis we show 5 different performance measures.

In general, changing the macro grid size brings different results. During the experimental phase with congestion awareness, we observed an improvement when enlarging the macro grid dimension. When this dimension is very high, results are so close that smaller macro grids could outperform larger ones. A $24 \times 24$ macro grid is better than the $48 \times 48$ version, both from the number of avoided conflicts, the number of search method calls, and the sum of individual steps.

From this comparison we notice that assigning ***freedom*** or ***constraints*** to the motion of the skids brings very different results. In particular, by constraining their possible actions, the performance gets worse, and agents find more difficulties while moving towards their goals, raising the maximum number of time steps and the time cost.
As said in [6.2.3], *variant 1* and *variant 3* constrain the movements: in the first case we consider a restriction of the agents' actions, while in the second method the restriction is imposed on the topology itself. Moreover, the two different versions of *variant 5* reduce the set of possible actions for conflict avoidance, whereas, in *basic search*, randomness helps to avoid pathological deadlocks. Concerning the variants with congestion awareness, we are not reducing the set of possible paths. Indeed, its main objective is to choose the best path given the information about the others. There are no constraints.

We can conclude that we need some way to assign the agents / the grid's structure more freedom to improve performances. In the following, we describe the goodness of randomness from another point of view.

### Mathematical Optimization and Randomness

MAPF is strongly related to ***Mathematical Optimization***. Some researchers have tackled multi-agent path planning problems from this point of view [1.1], e.g. exploiting linear and quadratic programs, in which one possible objective function could be the total sum of time steps. These specific centralized methods have the problem of being computationally unfeasible with many agents. Still, the idea of minimizing the objective function is very interesting.

We know centralized algorithms bring optimal solutions (*global optimum*), while generally, suboptimality holds for decentralized ones. This means that, even though they find a solution to the MAPF problem, the agents' paths correspond just to a *local optimum*.

In Mathematical Optimization, both to approximate the objective function and to find its global minima, it is important to find a *trade-off* between two important operations, namely *exploration* and *exploitation*.

- ***Exploration***. This operation corresponds to evaluating candidate solutions that are not neighbors to the current solution (or solutions).
- ***Exploitation***. It corresponds to when a search is done in the neighborhood of the current solution (or solutions).

By just applying exploitation, the search process might get stuck into local minima. The

solution is to consider exploration too: generally, the search space is explored by randomly selecting points in areas different from those of the current solution. Consequently, randomness helps search processes to pass from local minima to global ones.

*When considering decentralized approaches, we want to find suboptimal solutions as close as possible to optimal ones. In terms of local and global minima, this means requiring our process to reach a solution that is as close as possible to the global minimum.*
This justifies the use of randomness in our developed algorithms.

### Online setting

As said while introducing the specific problem [2], we intend to model scenarios where agents enter the system at any time. All the previously presented algorithms are built for this purpose. Now, we summarize the results of our experiments when considering online agents too (previous results focus on the offline setting).

The maximum value of agents that we consider during these simulations is $900 + 300 = 1200$. 900 skids start at the beginning of the search process (i.e., offline skids), and the other 300 are added randomly over time [1]. We choose this setting because our final goal is to reproduce warehouses with more than half of the free cells occupied by mobile agents. With a total amount of 1200 agents at peak hour, the $48 \times 48$ empty grid satisfies this condition. Moreover, adding further agents in case of more complex grid scenarios would be too costly from the CPU time point of view.

In Table 8.2 we summarize the performances of all the developed algorithms concerning the main studied variables, using the benchmark $48 \times 48$ empty grid.

Offline results are preserved in the online setting: except for the time consumption, the best algorithm is the one combining the basic search with congestion awareness and computational optimization on the number of search method calls, with a macro grid of size $24 \times 24$.

The same results hold for the more complex benchmark grids, as we can see from Tables 8.3 and 8.4.

In *conclusion*, our final optimized version is appropriate also to deal with new skids entering the system at any time.

[1] Note, in real applications, agents are not added randomly. This is a simpler assumption we make, without loss of generality.

### 8.2.2 Comparison with benchmark algorithms

As we said in [4.2], Silver [2005] is considered the masterpiece of decoupled MAPF approaches. The ideas of cooperation behind its three developed decentralized algorithms set a new era in which agents don't focus on themselves anymore (as for LRA* [4.2.1]) but navigate taking care of the other's actions, either completely, from the start to the end of their journey, or partially.

Our algorithm's optimized version is compared with CA* [4.2.2] and HCA* [4.2.3], showing **cooperation outperforms self-interest**, although sometimes these algorithms don't find any existing solution. Indeed, we also implemented a simplified version of WHCA* [4.2.4], but this is likely not to find solutions even for few hundreds of agents.

Furthermore, we implemented a Multi-Agent A* version to check the difference between centralized and decentralized algorithms. Despite the correctness of the code, the disadvantages of centralized pathfinding in terms of time and space computational complexity arise even for few agents on a small empty grid.

In Figure 8.2, we show the algorithms' performances in terms of the sum of time steps, makespan, and CPU time. We compare the *basic search*, the two *final optimized variants*, *CA** and *HCA**.

The two benchmark algorithms have similar outcomes. That is reasonable since HCA* differs from CA* only for the heuristic choice. For each plotted variable, CA* and HCA* outperform our selfish algorithms.

CA* and HCA* need up to half the *sum of time steps* required by our best self-interested approaches. We notice both literature cooperative and our developed self-interested algorithms show linear growth with the number of agents. This proves the quality of our optimized version since it gives results similar to cooperative methods, thanks to the idea of congestion awareness.

The *makespan* is approximately constant for the implemented benchmarks, while our best algorithms slightly increase.

Finally, concerning *time consumption*, cooperation gives dramatically fast results, for two reasons:

- the number of A* calls is the same as the number of agents, which is sharply smaller than for our versions;
- our final optimized versions exploit the concept of the macro grid, building and updating it for every single action;
- from the implementation point of view, our methods are more complex.

Nevertheless, we stop at 800 agents since both Silver's algorithms fail at finding a solution with a higher number of skids, while our methods succeed.

Interestingly, our *basic search* is similar to LRA*. The underlying idea of locally repairing conflicts is preserved in all our optimized versions.

We conclude the reasons why we have the worst results are connected to those of LRA*. As noticed by Silver, if bottlenecks occur in crowded areas, they may take arbitrarily long to be solved, because agents constantly reroute trying to escape, requiring a full recomputation of the A* search almost every turn. This leads to unintelligent behavior [2]. Each agent makes a path change independently, forming cycles where the same location may be visited by agents repeatedly in a loop.
Despite this, our algorithm outperforms LRA* for two reasons:

- by using the *random-style* collision avoidance, there are more ways for an agent to escape conflicts (involved agents are free to choose who will pass and who will move away);
- *congestion awareness* helps to choose less crowded areas, so some form of world knowledge is used to improve the performance.

Cooperation is the key to planning paths with a highly reduced amount of A* search method calls, finding solutions faster, with fewer replanning phases than the case of self-interested approaches. This implies a lower sum of time steps and makespan. Collisions are avoided intrinsically in advance since agents don't act individually anymore.

Still, cooperation constrains agents' motion. Skids must consider the other's positions and paths. From a practical point of view, this would require constant communication, sometimes prohibitively expensive.

[2] Aesthetically unpleasant or inefficient behaviors must be removed as much as possible. One could think about improving the self-interested approach, but what has an impact on the efficiency of the movements is the use of cooperation.

## 8.3 Conclusions

Starting from a purely self-interested basic search with *random-style* collision avoidance, we improved the performance by introducing ways for agents' cooperation, specifically using *congestion awareness* to push agents into less crowded regions. Since it doesn't constrain skids' motion, our final optimized algorithm outperforms all the previously developed ones. Agents have a high level of freedom.

Passing to the comparison with some of the most important algorithms in the literature, we notice *cooperation outperforms self-interest*, at the cost of higher required communication and higher risk of not finding any solution. As said in [4.2], *"globally cooperative"* approaches, such as CA* and HCA*, may be unnecessary in practical cases with rising problems in case of agents staying at target and with a too high dependency on agents' ordering. Introduced with WHCA*, *"local cooperation"* solves these difficulties: for every agent, we consider a cooperation window that moves accordingly to the agents' position, iteratively up to the corresponding destination. With this last windowed method, agents both *cooperate* and have *freedom*. Nevertheless, because of the elevated number of replanning phases, cooperation may still cause no solution.

Coming back to our *final optimized algorithm*, we expect that by reducing the cooperation window results will increase. Indeed, planning too far into the future is not necessary and sometimes unuseful. We talk about these aspects in the final part of our work.

| Online setting: 900 offline + 300 online | | | | |
|---|---|---|---|---|
| | Empty Grid | | | |
| Algorithm | # Avoided Conflicts | # A* Calls | Sum of Time Steps | CPU Time [s] |
| *basic search* | 46483 | 85956 | 112497 | 31.5 |
| *variant 1* | 219691 | 263606 | 327311 | 107.6 |
| *variant 2* | 46483 | 27994 | 112497 | 17.9 |
| *variant 3.1* | 31296 | 71139 | 89687 | 20.5 |
| *variant 3.2* | 31636 | 71083 | 89957 | 27.7 |
| *variant 3.3* | 33553 | 73416 | 93117 | 29.9 |
| *variant 4* *Macro Grid* $4 \times 4$ | 30289 | 68404 | 85918 | 393.8 |
| *variant 4* *Macro Grid* $8 \times 8$ | 20258 | 57493 | 69950 | 279.7 |
| *variant 4* *Macro Grid* $16 \times 16$ | 18989 | 56236 | 68187 | 353.6 |
| *variant 4* *Macro Grid* $24 \times 24$ | 18942 | 55969 | 67890 | 491.1 |
| *variant 4* *Macro Grid* $48 \times 48$ | 19735 | 57482 | 69763 | 2500.4 |
| *variant 5.1* | 30872 | 70657 | 88879 | 116.0 |
| *variant 5.2* | 155490 | 205215 | 288271 | 87.9 |
| *final optimized variant* *Macro Grid* $48 \times 48$ | 20012 | 18386 | 70003 | 3996.9 |
| *final optimized variant* *Macro Grid* $24 \times 24$ | 18672 | 17457 | 67659 | 612.1 |

Table 8.2: Online setting: 900 offline agents + 300 online agents. Comparison between all the developed algorithms in terms of number of avoided conflicts, number of A* calls, sum of time steps, and CPU time. The benchmark grid is the $48 \times 48$ empty grid. For each column, the best value is marked in red color.

| Online setting: 900 offline + 300 online | | | | |
|---|---|---|---|---|
| | Boston Grid | | | |
| Algorithm | # Avoided Conflicts | # A* Calls | Sum of Time Steps | CPU Time [s] |
| *basic search* | 161382 | 409499 | 498966 | 6382.9 |
| *variant 1* | 521362 | 780741 | 898927 | 11611.8 |
| *variant 2* | 161382 | 88782 | 498966 | 4034.5 |
| *variant 3.1* | 219161 | 477236 | 593086 | 5543.0 |
| *variant 3.2* | 203667 | 458956 | 569072 | 7312.7 |
| *variant 3.3* | 167437 | 419326 | 511173 | 5006.4 |
| *variant 4* *Macro Grid* $8 \times 8$ | Too high computational cost | | | |
| *variant 5.1* | Too highly congested areas | | | |
| *variant 5.2* | Too highly congested areas | | | |
| *final optimized variant* *Macro Grid* $8 \times 8$ | 77298 | 51126 | 373428 | 13812.4 |

Table 8.3: Online setting: 900 offline agents + 300 online agents. Comparison between all the developed algorithms in terms of number of avoided conflicts, number of A* calls, sum of time steps, and CPU time. The benchmark grid is the $256 \times 256$ Boston grid. For each column, the best value is marked in red color.

| Online setting: 900 offline + 300 online | | | | |
|---|---|---|---|---|
| | Maze Grid | | | |
| Algorithm | # Avoided Conflicts | # A* Calls | Sum of Time Steps | CPU Time [s] |
| *basic search* | 968054 | 1380718 | 1889630 | 28563.6 |
| *variant 1* | Too high computational cost | | | |
| *variant 2* | 968054 | 482778 | 1889630 | 15925.6 |
| *variant 3.1* | 1486609 | 1947279 | 2691015 | 45795.7 |
| *variant 3.2* | 1162736 | 1604380 | 2218179 | 26800.0 |
| *variant 3.3* | 662472 | 1013948 | 1373763 | 21258.0 |
| *variant 4* *Macro Grid* $8 \times 8$ | Too high computational cost | | | |
| *variant 5.1* | 765167 | 1181417 | 1588768 | 39313.9 |
| *variant 5.2* | Too highly congested areas | | | |
| *final optimized variant* *Macro Grid* $8 \times 8$ | 406671 | 224938 | 928605 | 20100.0 |

Table 8.4: Online setting: 900 offline agents + 300 online agents. Comparison between all the developed algorithms in terms of number of avoided conflicts, number of A* calls, sum of time steps, and CPU time. The benchmark grid is the $128 \times 128$ Maze grid. For each column, the best value is marked in red color.

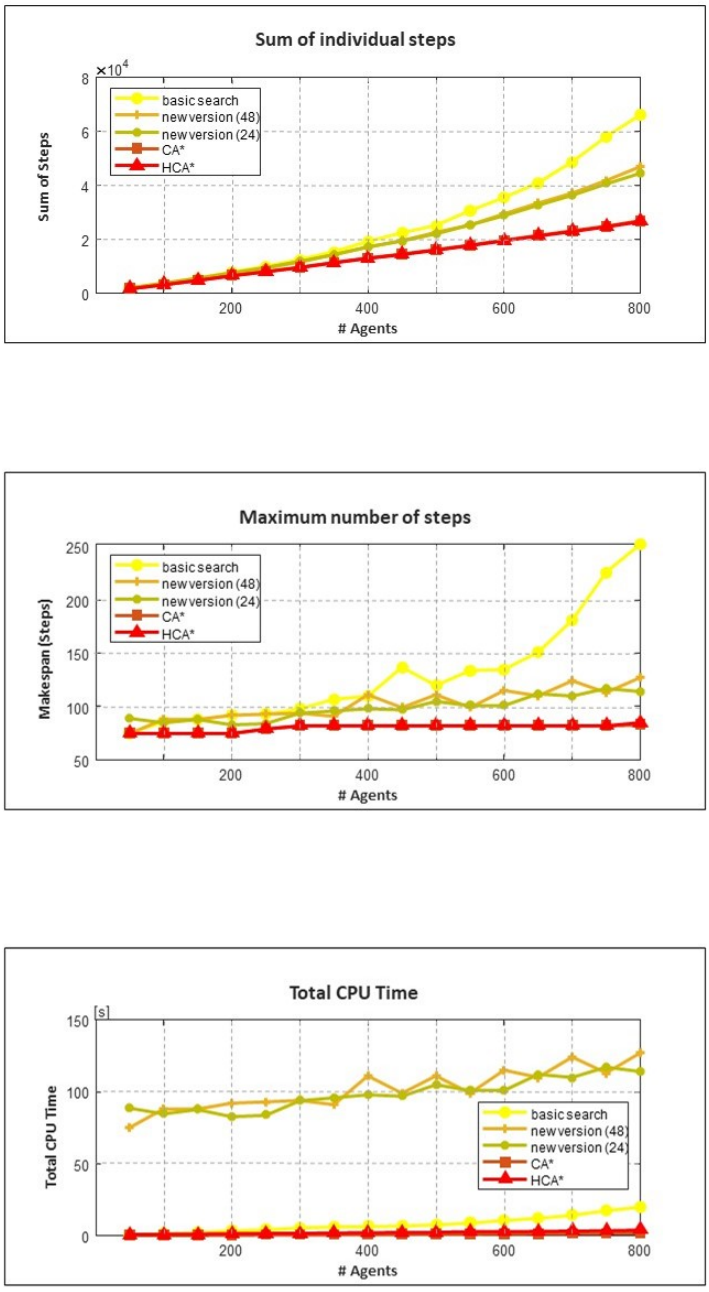



Figure 8.2: Comparison between *basic* and *optimized developed selfish algorithms* and *literature cooperative benchmarks*. We consider the $48 \times 48$ empty grid. On the $x$-axis, we have the number of agents (from 50 to 800, with stepsize 50), while on the $y$-axis we show 3 different performance measures.

# Part III

# Links to Congestion Games and Network Traffic Flow

In this part, we recall the main concepts of Network Traffic Flow and Game Theory, showing the connections to MAPF problems [3].

Specifically, MAPF can be interpreted as a Network Flow problem. For this reason, we can study our problem from two different points of view, each one with a specific objective.

- *Traffic Flow Optimization.* As said before in this document, decentralized algorithms have different advantages over centralized ones. But they have the great drawback of generally being suboptimal. By using the concepts of User Optimum Traffic Assignment (UO-TAP) and System Optimum Traffic Assignment (SO-TAP), it is possible to quantify how much a decoupled algorithm's solution is suboptimal, compared to a centralized one, from the point of view of the generated traffic.

- *Game Theory.* We know Network Traffic Flow models are examples of congestion games. Hence, we can apply the concepts of Nash Equilibrium and Best Response to understand which are the best actions to be applied by the agents close to possible conflicts, and how to push in the best way agents in less congested areas. The main problem with traffic networks is there are many agents in the system, so a game theoretic analysis would generally be more complex.

This establishes a deep connection with Graphs Theory. We intend to use these concepts in the near future, to understand, describe, and implement smart solutions to the MAPF problem.

[3] Theoretical elements are mainly taken from University "Lecture notes on Network Dynamics", by Giacomo Como and Fabio Fagnani.

# Chapter 9

# Network Traffic Flow

In this chapter, we first review the main aspects of Network Flow Optimization. Then, we show that MAPF can be treated as an optimal transport problem. For this reason, we can quantify the inefficiency of decentralized approaches over centralized ones in terms of Price of Anarchy from the Traffic Assignment Problem.

The idea of network traffic flow is also linked to Games Theory, as described in the next chapter.

## 9.1 Network Flow Optimization

Network flow optimization is useful to study optimal traffic assignments in traffic networks. To study network flow optimization, we consider a larger notion of network structure than that of *weighted directed graph*. First, we give the definition of *weighted directed graph*, then we present the broader concept of *directed multigraph*.

**Definition 3** *A **directed weighted graph** is a triple $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{W})$ where:*

- *$\mathcal{V}$ is the countable set of **nodes**, also called **vertices**;*
- *$\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the set of **links** (also called **edges**);*
- *$\mathcal{W} \in \mathcal{R}_+^{\mathcal{V}\times\mathcal{V}}$ is the **weight matrix**, with $\mathcal{W}_{ij} > 0$ if and only if $(i, j)$ is a link.*

*We denote with $n = |\mathcal{V}|$ the **order** of the graph. We define links $(i, i)$ as **self-loops**.*

**Definition 4** *A **directed multigraph** is a quadruple $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \theta, \kappa)$, where:*

- *$\mathcal{V}$ is the set of nodes;*
- *$\mathcal{E}$ is the set of links;*
- *$\theta : \mathcal{E} \to \mathcal{V}$ and $\kappa : \mathcal{E} \to \mathcal{V}$ are two functions such that, if $e \in \mathcal{E}$, $\theta(e)$ and $\kappa(e)$ are two distinct nodes representing the start and the termination of link $e$.*

*Two links $e_1$, $e_2 \in \mathcal{E}$ such that $\theta(e_1) = \theta(e_2)$ and $\kappa(e_1) = \kappa(e_2)$ are called* ***parallel****, whereas if $\theta(e_1) = \kappa(e_2)$ and $\kappa(e_1) = \theta(e_2)$, they are called* ***opposite****.*

To every unweighted directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a multigraph can be associated by simply considering $\tilde{\mathcal{G}} = (\mathcal{V}, \mathcal{E}, \theta, \kappa)$, with the same set of vertices and links of the original $\mathcal{G}$ and start and terminal functions defined by $\theta(i, j) = i$, $\kappa(i, j) = j$.

Two important matrices in the context of network connectivity are the *node-link incidence matrix* and the *link-path incidence matrix*.

**Definition 5** *A* ***node-link incidence matrix*** *$B \in \mathcal{R}^{\mathcal{V}\times\mathcal{E}}$ is defined as*

$$B_{\theta(e)e} = +1, \quad B_{\kappa(e)e} = -1, \quad B_{ke} = 0 \ \forall k \in \mathcal{V}\backslash\{\theta(e), \kappa(e)\},$$

*for every link $e \in \mathcal{E}$ that is not a self-loop (i.e., $\theta(e) \neq \kappa(e)$), and $B_{ie}$ for all $i$ in $\mathcal{V}$ if $e$ is a self-loop.*

**Definition 6** *For two nodes $o \neq d$ in $V$ such that $d$ id reachable from $o$, the* ***link-path incidence matrix*** *$A^{(o,d)}$ in $\{0,1\}^{\mathcal{E}\times\Gamma_{(o,d)}}$ is defined as*

$$A^{(o,d)}_{e\gamma} = \begin{cases} 1 & if \quad e \text{ is along } \gamma \\ 0 & if \quad e \text{ is not along } \gamma. \end{cases}$$

In the following example we show how the concepts of node-link and link-path incidence matrix apply to the network in Figure 9.1.[1]

**Example 1** *Given the graph in Figure 9.1, the link-path incidence matrix $A$ and the node-link incidence matrix $B$ are:*

$$A = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

$$B = \begin{bmatrix} +1 & +1 & 0 & 0 & 0 \\ -1 & 0 & +1 & +1 & 0 \\ 0 & -1 & -1 & 0 & +1 \\ 0 & 0 & 0 & -1 & -1 \end{bmatrix}$$

[1] Image Credits to: Como, Fagnani.

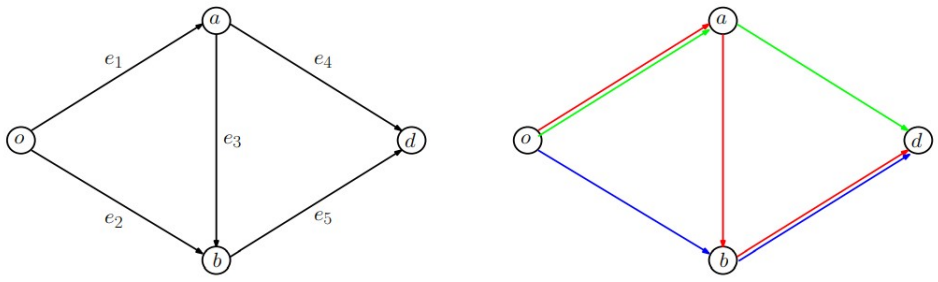



Figure 9.1: On the left, a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, s, t)$ with node set $\mathcal{V} = \{o, a, b, d\}$ and link set $\mathcal{E} = \{e_1, e_2, e_3, e_4, e_5\}$. On the right, the three distinct $o - d$ paths are colored: $\gamma^{(1)} = (o, a, d)$ (green); $\gamma^{(2)} = (o, a, b, d)$ (red); $\gamma^{(3)} = (o, b, d)$ (blue).

Let us now introduce the notion of *network flow*. Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, s, t)$, an ***exogenous net flow*** on $\mathcal{G}$ is a vector $\nu \in \mathcal{R}^{\mathcal{V}}$ satisfying the constraint

$$\sum_i \nu_i = 0 \tag{9.1}$$

The positive part $[\nu_i]_+ = max\{0, \nu_i\}$ and the negative part $[\nu_i]_- = max\{0, -\nu_i\}$ of the net flow in a node $i$ are respectively the ***exogenous inflow*** in and the ***external outflow*** from $i$. The constraint 9.1 is then equivalent to requiring that the total exogenous inflow matches the total external outflow.
We consider

$$\chi = \frac{1}{2}\sum_i |\nu_i| = \sum_i |\nu_i|_+ = \sum_i |\nu_i|_- \tag{9.2}$$

as the ***throughput***, i.e., *the total flow that goes through the network.*
Nodes $i$ such that $\nu_i > 0$ are generally called ***sources***, ***origins***, or ***generators***, while nodes $i$ such that $\nu_i < 0$ are called ***sinks***, ***destinations***, or ***loads***.

**Definition 7** *Given a graph $\mathcal{G}$ and a vector of exogenous net flows $\nu \in \mathcal{R}^{\mathcal{V}}$ satisfying 9.1, a **network flow** is a nonnegative vector $f \in \mathcal{R}_+^{\mathcal{E}}$ whose entries $f_e$ satisfy the flow balance equations*

$$\nu_i + \sum_{e \in \mathcal{E} | \kappa(e) = i} f_e = \sum_{e \in \mathcal{E} | \theta(e) = i} f_e, \quad i \in \mathcal{V} \tag{9.3}$$

The term $f_e$ represents the flow on the link $e \in \mathcal{E}$. Hence, the equation above states that the total inflow in a node $i$, resulting from both possible exogenous inflow $[\nu_i]_+$ and flows $f_e$ from incoming links $e$, equals the total outflow from $i$, resulting from both possible

external outflow $[\nu_i]_-$ and flows $f_e$ towards outgoing links $e$. These *flow balance equations* can be rewritten more compactly in terms of the *node-link incidence matrix* as

$$Bf = \nu \tag{9.4}$$

The previous definition can then be rewritten as follows.

**Definition 8** *Given a vector $\nu$ in $\mathcal{R}^{\mathcal{V}}$ of exogenous net flows, a **network flow** on $\mathcal{G}$ is a vector $f \in \mathcal{R}^{\mathcal{E}}$ that satisfies the nonnegativity and conservation of mass constraints:*

$$f \geq 0, \quad Bf = \nu \tag{9.5}$$

*We define **o-d flows** the network flows with a single origin $o$ and a single destination $d$, i.e., nonnegative vectors $f \in \mathcal{R}^{\mathcal{E}}_+$ such that*

$$Bf = \chi(\delta^{(o)} - \delta^{(d)}) \tag{9.6}$$

*for some throughput value $\chi$.*

In network flow optimization, we generally consider the problem of selecting a network flow $f^*$ from the set of vectors satisfying Definition [8] that minimizes a separable convex cost function.

Let us define the ***link cost function*** $\psi_e(f_e)$, $e \in \mathcal{E}$, describing the cost $\psi_e(f_e)$ incurred when a flow $f_e \geq 0$ passes through link $e$.
We make the following assumption on such cost functions.

**Assumption 1** *For every link $e$ in $\mathcal{E}$, the **cost function***

$$\psi_e(f_e) : [0, +\infty) \to [0, +\infty]$$

*is such that $\psi_e(0) = 0$, non-decreasing, continuously differentiable and convex in the internal $[0, c_e)$ where*

$$c_e = sup\{x \geq 0 : \psi_e(x) < +\infty\}$$

*is referred to as the **flow capacity** of link $e$.*

Requiring $\psi_e(0) = 0$ means to require no cost for sending no flow. Non-decreasing $\psi_e(x)$ means the more the flow the higher the cost incurred. Allowing $\psi_e(f_e)$ to possibly take value $+\infty$ allows to consider the case of infinite link flow capacity ($c_e < +\infty$), or elevated link flow ($x \geq c_e$). Convexity of the cost function $\psi_e(x)$ means that its derivative $\psi'_e(f_e)$ is non-decreasing in $x$. Since this derivative can be considered as the infinitesimal change of the cost incurred when an infinitesimal unit of flow is added to the quantity $x$ already in the link, this is quite a natural assumption. Finally, differentiability is mainly required to simplify practical scenarios. Consequently:

**Definition 9** *A **convex separable network flow optimization problem** is defined as*

$$M(\nu) \quad = \inf_{f \in \mathcal{R}_+^{\mathcal{E}} \; ; \; Bf = \nu} \quad \sum_{e \in \mathcal{E}} \psi_e(f_e) \tag{9.7}$$

When $M(\nu) = +\infty$, the network flow optimization problem is said to be ***unfeasible***, while ***feasible*** otherwise.

It can be shown that ***the optimal transport problem is a generalization of the shortest path problem*** with:

- linear costs,
- arbitrary net-flow $\nu$ in $\mathcal{R}^{\mathcal{V}}$,

where linear costs are as follows. Let each link $e$ in $\mathcal{E}$ be assigned a positive weight $l_e$ representing its physical length; given the edge flow $f_e$, the linear cost function is

$$\psi_e(f_e) = l_e f_e. \tag{9.8}$$

***MAPF can be treated as an optimal transport problem, but now the cost function is not linear.*** In our MAPF setting, the length $l_e$ is the time needed to move from one cell to its neighbor; since we are using time steps, this length is 1. The flow corresponds to the number of agents [2]. With linear cost functions, we are implicitly considering that the congestion does not affect the length/cost to move from one node to the next one. Even though the time needed to move from one cell to its neighbor will always be one time step, with linear costs we are disregarding the fact that when possible conflicts occur, an agent has to wait for the other to pass. This augments the overall cost on that edge.

In the next section, we establish a connection between transportation networks' traffic assignment and optimal transport problems. We will understand the correct cost function choice boils down to deciding whether the approach is centralized or decentralized (SO-TAP vs UO-TAP).

[2] The *flow capacity* of link $e$ is not of great concern; it can be set by simply following its definition or directly to $+\infty$.

## 9.2 Optimal Traffic Assignments in Transportation Networks

Now we focus on the optimal traffic assignment problem in transportation networks. In general, here links $e$ in $\mathcal{E}$ represent portions of roads and nodes represent junctions.
In our setting, the grid structure is trivially translated into graph where nodes corresponds to cells, and unitary edges connect adjacent cells.

In general, the cost function is defined in terms of a delay function $\tau_e(f_e)$ that returns the delay encountered by any user traversing a link as a function of the flow on that link. We assume these delay functions satisfy the following assumption.

**Assumption 2** *For every link $e$ in $\mathcal{E}$, the* ***delay function***

$$\tau_e : \mathcal{R}_+ \to \mathcal{R}_+ \cup \{+\infty\}$$

*is a non-decreasing, twice continuously differentiable in the interval $[0, c_e)$, and such that $\tau_e(f_e) = +\infty$ for $f_e > c_e$ and*

$$2\tau_e'(f_e) + f_e\tau_e''(f_e) \geq 0, \quad 0 \leq f_e < c_e. \tag{9.9}$$

Similarly to the first assumption, [9.1], for the cost function, this last assumption, [9.2], allows us to treat both the case of finite and infinite link flow capacities. It is quite natural to assume that the delay is non-decreasing in the flow, since congestion can only slow down traffic, never speed it up. The assumption of twice continuous differentiability is used for simplicity. The inequality relation above immediately follows if, e.g., the delay functions are convex.

### 9.2.1 System Optimum Traffic Assignment

**Definition 10** *Let a delay function $\tau_e$ satisfying the assumption [9.2] be assigned to every link of a multigraph describing the topology of a transportation network. The* ***System Optimum Traffic Assignment Problem (SO-TAP)*** *consists in a network flow optimization problem with link cost functions*

$$\psi_e(f_e) = f_e \cdot \tau_e(f_e), \quad f_e \geq 0, \quad e \in \mathcal{E}. \tag{9.10}$$

The interpretation of such link cost functions is that if the flow on link $e$ is $f_e$, then the delay is $\tau_e(f_e)$, and the cost represents the product of the delay times the flow, i.e., the total delay. Furthermore, with this definition of the link cost function, the first assumption [9.1] holds too.

In our specific case, we choose a *linear delay function*. To understand this, let us consider the conflict situation. A possible collision between two agents is solved by letting one of them pass. In the case of more agents, we observe the formation of a queue. In queueing theory, we know the delay of arrivals to a server is proportional to the number of agents on the buffer. Hence, in the case of $n$ skids, the delay is proportional to $n$. Note, the time needed to move from one cell to its neighbor is always one time step. The amount of time the agent has to wait before moving impacts the cost function $\psi_e$.

### 9.2.2 User Optimum Traffic Assignment

Let us assume each link $e$ in $\mathcal{E}$ is equipped with delay functions $\tau_e(f_e)$ returning the delay experienced by any user traversing link $e$ when the total flow through it is $f_e$. Such delay functions are assumed to be continuous and non-decreasing. Then:

**Definition 11** *The **User Optimum Traffic Assignment Problem (UO-TAP)** consists in a network flow optimization problem with link cost functions*

$$\psi_e(f_e) = \int_0^{f_e} \tau_e(s)ds, \quad f_e \geq 0, \quad e \in \mathcal{E}. \tag{9.11}$$

Monotonicity and convexity of these link cost functions follow from the fact that the delay functions are nonnegative-values and non-decreasing. The other two conditions for *SO-TAP* are not needed in this case.

***The idea of UO-TAP is to model flows resulting not from a centralized optimization, but rather as the outcome of selfish behaviors of drivers.***

*We assume drivers choose their route so as to minimized the delay they experience along it.* This is formalized by the notion of *Wardrop equilibrium.*

#### Wardrop equilibrium

Given a couple origin-destination ($o \neq d$) in $\mathcal{V}$ such that $d$ is reachable from $o$, let $\Gamma_{o,d}$ be the set of all $o-d$ paths, while the link-path incidence matrix is denoted as $A^{(o,d)}$. Let us consider a nonnegative vector $z$ in $\mathcal{R}^{\Gamma_{o,d}}$ such that $1'z = \chi$, whose entries $z_p$ represent the aggregate flow along the $o-d$ path $p$ and recall that, by the Flow Decomposition Theorem [3],

$$f = A^{(o,d)}z \tag{9.12}$$

satisfies

$$f \geq 0, \quad Bf = BA^{(o,d)}z = \chi(\delta^{(o)} - \delta^{(d)}), \tag{9.13}$$

[3]The ***Flow Decomposition Theorem*** states that every assignment of flows to both $o-d$ paths and cycles in the multigraph induces a *unique* network flow $f$ on the links and that, conversely, for every network flow $f$ on the links, there exists a generally *non-unique* assignment of flows to both $o-d$ paths and cycles in the multigraph that induces $f$.

i.e., it is a network flow from $o$ to $d$ of throughput $\chi$.
Then, we can define the *Wardrop equilibrium* as follows.

**Definition 12** *For a given throughput $\chi > 0$, a **Wardrop equilibrium** is a flow vector*

$$f = A^{(o,d)} z$$

*where $z$ in $\mathcal{R}^{\Gamma_{o,d}}$ is such that $z \geq 0$, $1'z = \chi$, and for every path $p$ in $\Gamma_{o,d}$*

$$z_p > 0 \Longrightarrow \sum_{e \in \mathcal{E}} A_{ep}^{(o,d)} \tau_e(f_e^{(0)}) \leq \sum_{e \in \mathcal{E}} A_{eq}^{(o,d)} \tau_e(f_e^{(0)}) \quad \forall q \in \Gamma_{o,d} \tag{9.14}$$

So, it is a network flow vector that is associated with an $o-d$ path distribution $z$ such that if some drivers choose $p$ as their route from $o$ to $d$, then the total delay associated with this path cannot be worse than the total delay corresponding to any other $o-d$ path. ***A Wardrop equilibrium should be interpreted as the result of a rational, and selfish behavior of drivers that want to minimize their own delay: none of them would choose a suboptimal route if a better one is available.***

The potential fall in efficiency from social to selfish equilibria is an example of *price of anarchy* [4].

**Definition 13** *The **Price of Anarchy (PoA)** associated to a Wardrop equilibrium $f^{(0)}$ is*

$$PoA(0) = \frac{\sum_{e \in \mathcal{E}} f_e^{(0)} \tau_e(f_e^{(0)})}{\min\limits_{f \in \mathcal{R}_+^{\mathcal{E}} \ ; \ Bf = \chi(\delta^{(o)} - \delta^{(d)})} \sum_{e \in \mathcal{E}} f_e \tau_e(f_e)} \tag{9.15}$$

*i.e., it is the ratio between the total cost at the Wardrop equilibrium and the minimum possible total cost.*

### 9.2.3 Price of Anarchy for Decentralized MAPF Algorithms

The concept of Price of Anarchy is particularly important if we want to compare decentralized with centralized methods in terms of their respective objective functions for the total travel time (sum of individual time steps). Despite this, it is not immediate to translate the concepts above to the MAPF framework.

Even though we easily define MAPF as an optimal transport problem, we should tackle properly some difficulties concerning the optimal traffic assignment. In particular, we should redefine the concept of Wardrop equilibrium. As said above, a *Wardrop equilibrium* is a network flow vector that is associated with an $o-d$ path distribution $z$ such that if

[4] In general, the **Price of Anarchy (PoA)** is a concept in economics and game theory that measures how the efficiency of a system degrades due to selfish behavior of its agents (https://en.wikipedia.org/wiki/Price_of_anarchy).

some drivers choose $p$ as their route from $o$ to $d$, then the total delay associated with this path cannot be worse than the total delay corresponding to any other $o - d$ path.

In MAPF, there is not one single origin and destination. The edge flow vector $f^{(0)} = A^{(o,d)}z$ can be adapted to our scenario of many start and goal positions. The Flow Decomposition Theorem states that flow assignments to paths (and cycles) induce a unique link flow vector and, conversely, for every link flow vector there exist generally many flow assignments to $o - d$ paths (and cycles). Intuitively, this still holds in our setting. The number of rows of matrix $A$ is always equal to the number of edges, while the number of columns corresponds to the number of paths connecting every couple origin-destination.

Once we adapted the definition of network flow $f$ on the links, we have to check if the conditions in the Wardrop equilibrium definition are satisfied. Again, intuitively, we expect the conditions to hold if we extend matrix $A$. Formal proof may require more attention.

Moreover, when defining a Wardrop equilibrium, we are implicitly making 3 assumptions:

- users are subject to the same costs;
- users have access to the delay functions;
- stationarity setting.

The third condition is the most problematic. The number of agents is finite, so traffic delays are not fixed. Even with an elevated number of agents, practically it is difficult for skids to move along the same paths as flows on traffic networks do: the number of mobile robots changes during the day inside a warehouse, and also the task.
Sometimes, though, it is not so unrealistic as a situation. There may be warehouses where mobile agents tend to navigate to specific areas, doing specific jobs, continuously. In this case, stationarity is reasonable too. Let us assume:

- full knowledge of the grid's congestion, with consequent knowledge of flow times and delay functions;
- stationarity, so that delay functions do not change over time.

Once we deal with these observations, we can refer to MAPF as a UO-TAP.

On the other hand, without loss of generality, MAPF has connections also to the SO-TAP when we assume the centralized optimizer knows the grid's structure and the agents' characteristics.

At this point, the *price of anarchy* is equivalent to the ratio between the objective function value obtained with a decentralized method and the objective function value for a centralized algorithm.
We could use it to measure how the efficiency of a system reduces due to the selfish behavior of its agents. The higher the price of anarchy, the more inefficient the considered decentralized method.
The efficiency is maximal when $PoA = 1$. To reach this value, either we modify the grid's structure, removing some nodes and edges, or we impose appropriate costs on specific agents. Moreover, by looking at the objective functions' definitions, we observe that UO-TAP and SO-TAP coincide when the delay function is constant.

To conclude, ***by using the concept of the price of anarchy, it is possible to quantify how much a decoupled algorithm's solution is suboptimal, compared to a centralized one***.

# Chapter 10

# Game Theory

We have seen in the previous chapter that MAPF can be interpreted as a network flow problem. Such models, precisely network traffic flow models, are examples of *congestion games*. Hence, we can study MAPF in the context of game theory. In particular, we will consider the concepts of *Best Response* and *Nash Equilibria* to understand which are the best actions to be applied by the agents to avoid conflicts and how to be pushed in less congested areas while navigating.

First, we present the fundamental elements of classical game theory. Then, we link to MAPF problems. We study the Nash equilibria in MAPF both statically to decide how to best escape possible collisions and dynamically, inspired by the concept of *Best Response Dynamics*, to push the agents in uncongested areas.

## 10.1 Game Theory: Overview

We consider games in strategic form [1]. There is a finite set of ***players*** $\mathcal{V}$ and a set of ***actions*** $\mathcal{A}$. The assignment of an action to each player is described by a vector $x \in \mathcal{A}^{\mathcal{V}}$ that is called an ***action configuration*** (or ***action profile***). We denote by

$$\mathcal{X} = \mathcal{A}^{\mathcal{V}}$$

the ***configuration space***. Each player $i \in \mathcal{V}$ is equipped with a ***utility*** function, or ***reward***, or ***payoff*** function

$$u_i \ : \ \chi \ \rightarrow \ \mathcal{R}$$

that associates with every action configuration $x$ in $\mathcal{X}$ the utility $u_i(x)$ that player $i$ gets when each player $j$ is playing action $x_j \in \mathcal{A}$. We indicate with

$$x_{-i} = x_{|\mathcal{V}\setminus\{i\}}$$

[1] The ***strategic form*** is the most used form, in game theory; it is generally more convenient for mathematical analysis and is usable for both discrete and continuous strategy sets. In the literature, games are typically presented in a more general setting, allowing for the possibility that players have different actions sets $\mathcal{A}_i$ for $i \in \mathcal{V}$(Lasaulce and Tembine [2011]).

the vector obtained from the action profile $x$ by removing its $i$-th entry. The utility can then be rewritten in the form

$$u_i(x_i, x_{-i}) = u_i(x)$$

to stress that player $i$ chooses to play action $x_i$, and the rest of the players choose to play $x_{-i}$. We refer to the triple $(\mathcal{V}, \mathcal{A}, \{u_i\}_{i\in\mathcal{V}})$ as a ***(strategic form) game***.

### 10.1.1 Best response & Nash equilibrium

Every player $i$ is to be interpreted as a rational agent choosing its action $x_i$ from the action set $\mathcal{A}$ so as to maximize its own utility $u_i(x_i, x_{-i})$. This utility depends not only on player $i$'s action, but also on the actions of the other players, $x_{-i}$, so it is natural to introduce the *best response (BR)* function:

**Definition 14** *The **Best Response (BR)** function for agent $i$ when all the other agents' actions are $x_{-i}$ is defined as*

$$\mathcal{B}_i(x_{-i}) = argmax_{x_i\in\mathcal{A}} u_i(x_i, x_{-i}). \tag{10.1}$$

Assuming that player $i$ knows what the rest of the players' actions are and that these are not changing, choosing an action in $\mathcal{B}_i(x_{-i})$ is its rational choice as it makes its utility as large as possible. In other words, *it is the best action to respond to other agents.*

**Definition 15** *A **(pure strategy) Nash equilibrium (NE)** for the game $(\mathcal{V}, \mathcal{A}, \{u_i\}_{i\in\mathcal{V}})$ is an action configuration $x^* \in \chi$ such that*

$$x_i^* \in \mathcal{B}_i(x_{-i}^*), \quad i \in \mathcal{V}. \tag{10.2}$$

The interpretation of a Nash equilibrium is the following. It is an action configuration such that no player has any incentive to unilaterally deviate from its current action, as its utility with the current action is the best possible given the current actions chosen by the other players. In other words, *a Nash equilibrium is an action configuration in which each agent is playing its best response.* We denote by $\mathcal{N}$ the set of NE of a game.

**Example 2** *(Prisoner's dilemma)*
*"Two members of a criminal gang are arrested and imprisoned. Each prisoner is in solitary confinement with no means of speaking to or exchanging messages with the other. The prosecutors do not have enough evidence to sentence them on the principal charge, but have evidence to convict them to $-b$ years on prison on a lesser charge. Simultaneously, the prosecutors offer each prisoner a bargain. Each prisoner is given the opportunity either to confess the other by testifying that he committed the main crime (action $-1$), or to remain silent (action $+1$). If one prisoner betrays the other and the other stays silent, the betrayer is freed of both the minor and major charges (corresponding to a payoff $d$) and the one who remain silent get sentenced to $-c$ years in prison. If both prisoners betray each other, they both get sentenced to $-a$ years in prison each."*

*The prisoner's dilemma is a symmetric* $2 \times 2$*-game* [2] *with payoff matrix as in Figure 10.1 whose entries satisfy*

$$c < a < b < d$$

*(e.g.,* $a = 15$*,* $b = -5$*,* $c = -30$*,* $d = 0$*).*

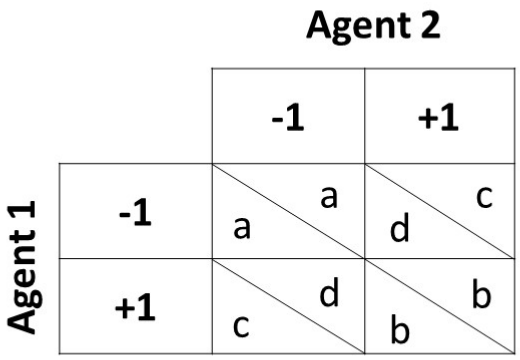



Figure 10.1: Payoff matrix of a symmetrix two-player game with action set $\mathcal{A} = \{-1, +1\}$.

*By looking at Figure 10.1, we find the prisoner's dilemma admits a unique (pure strategy) Nash equilibrium* $(-1, -1)$*, i.e., where the prisoners betray each other. Observe that, on the other hand,* ***if the prisoners could coordinate and remain silent*** *(so that the action profile is* $(+1, +1)$*) then they would both get a better payoff* $b > a$ *than the one they get at the Nash equilibrium* $(-1, -1)$*.*

[2] $2 \times 2$***-games*** *refer to two-player games with binary action space, i.e.,* $|\mathcal{A}| = 2$*. A two-player game with two utility functions* $u_i(r, s)$*, for* $i = 1,2$*, with* $r$ *the action played by player* $i$ *and* $s$ *the action played by his opponent, is* ***symmetric*** *if* $u_1(r, s) = u_2(r, s) = \varphi(r, s),\ r, s \in \mathcal{A}$*.*

### Congestion games

***Congestion games*** are an important family of games, with many applications, included network traffic flow models [9]. In congestion games, the number of players is finite but arbitrary. The actions chosen by the players are to be interpreted as (subsets of) *shared resources*, and the utility associated with the action only depends on the total number of players using the same resource(s). In most applications, by choosing a resource, every player decreases the utility (equivalently, increases the cost) of all players choosing the same action.

The following example adapts well to the case when resources are edges in a graph, and the action is a path between two nodes.

**Example 3** *Consider a set of players $\mathcal{V}$, a finite set of resources $\mathcal{E}$ (e.g., links in a transportation network $\mathcal{G} = (\mathcal{V}, \mathcal{E})$) and congestion costs $d_e : \mathcal{R}_+ \to \mathcal{R}_+$, where $d_e(f_e)$ is the congestion cost associated with resource $e$ when $f_e$ players use it. Every action $a \in \mathcal{A}$ is a (non-empty) subset of $\mathcal{E}$ (e.g., a given $o-d$ path in $\mathcal{G}$). The link-path incidence matrix [9.1] takes the name of resource-strategy incidence matrix $A \in \mathcal{R}^{\mathcal{E}\times\mathcal{A}}$ has entries $A_{ea} = 1$ if resource $e$ is used by action $a$ and $A_{ea} = 0$ otherwise. Given any configuration $x \in \mathcal{X}$ and any resource $a \in \mathcal{A}$, we define*

$$n_a(x) = |\{i \in \mathcal{V} : x_i = a\}|$$

*as the number of players choosing resource $a$ in configuration $x$. The vector $f = An(x)$ has dimension equal to the number of resources $\mathcal{E}$ and entries $f_e = \sum_{a\in\mathcal{A}} A_{ea} n_a(x)$ corresponding to the number of players using the different resources. Finally, the utility of every player $i \in \mathcal{I}$ is*

$$u_i(x_i, x_{-i}) = -\sum_{e\in\mathcal{E}} A_{ex_i} d_e(f_e) = -\sum_{e\in\mathcal{E}} A_{ex_i} d_e((An(x))_e).$$

### Network games

An important class of games is the one of the so-called *network games*. These are games where the players are represented as nodes of a graph $\mathcal{V} = (\mathcal{V}, \mathcal{E})$ and their utilities depend only on their own and their out-neighbors actions [3].

**Definition 16** *A **network game over a graph $\mathcal{G}$ ($\mathcal{G}$-game)** is any triple $(\mathcal{V}, \mathcal{A}, \{u_i\}_{i\in\mathcal{V}})$ whose utilities functions satisfy the following property: for any player $i \in \mathcal{V}$ and configurations $x, y \in \mathcal{A}^{\mathcal{V}}$ such that $x_j = y_j$ foe every $j \in N_i \cup \{i\}$ it holds*

$$u_i(x) = u_i(y)$$

Note that every game is a network game with respect to the *complete graph*.

[3] Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{W})$, the ***out-neighborhood*** and the ***in-neighborhood*** of a node $i \in \mathcal{V}$ are, respectively, the sets $\mathcal{N}_i^+ = \{j \in \mathcal{V} | (i, j) \in \mathcal{E}\}$, $\mathcal{N}_i^- = \{j \in \mathcal{V} | j \in \mathcal{V} | (j, i) \in \mathcal{E}\}$. Their nodes are respectively referred to as **out-neighbors** and **in-neighbors**

### Pairwise graphical game

Let us consider an indirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, W)$ with no self-loops and assume that, for every two neighboring nodes $i$, $j$, it is defined a two-player game. The corresponding utilities of $i$ and $j$ are, respectively, denoted by $\varphi^{(i,j)} : \mathcal{A} \times \mathcal{A} \to \mathcal{R}$ and $\varphi^{(j,i)} : \mathcal{A} \times \mathcal{A} \to \mathcal{R}$ and are called ***interaction utilities***. According to the usual terminology used in two-player games $\varphi^{(i,j)}(a, b)$ is the utility obtained by $i$ when it plays action $a$ and its opponent $j$ plays the action $b$. The utility, of every player $i \in \mathcal{V}$ is then simply set to the weighted sum of the utilities of the various two-player games that $i$ is playing with its neighbors:

$$u_i(x) = \sum_j W_{ij} \varphi^{(i,j)}(x_i, x_j)$$

The choice of having the graph undirected is solely for the sake of interpreting the interaction as a two-player game. The definition above adapts to any directed graph as long as we have the interaction utility $\varphi^{(i,j)}$ defined on each edge $(i, j) \in \mathcal{E}$.

In other words, a *pairwise graphical game is a network game where every agent is playing a two-player game with its neighbors.*

## 10.1.2 Best response dynamics

The *best response dynamics* is an example of game-theoretic learning process. We focus on the asynchronous best response dynamics, where players in a strategic form game get randomly activated one at a time and switch to a best response action.

### Best response dynamics

**Definition 17** *Consider a strategic-form game $(\mathcal{V}, \mathcal{A}, \{u_i\}_{i \in \mathcal{V}})$. The **continuous-time asynchronous best response dynamics** is a Markov chain $X(t)$ with state space $\mathcal{X} = \mathcal{A}^{\mathcal{V}}$, where every player $i \in \mathcal{V}$ is equipped with an independent rate-1 Poisson clock. When its clock ticks at time $t$, player $i$ updates its action to some $y_i$ chosen from the action set $\mathcal{A}$ with conditional probability distribution that is uniform over the best response set*

$$\mathcal{B}_i(X_{-i}(t)) = argmax_{x_i \in \mathcal{A}} \{u_i(x_i, X_{-i}(t))\}.$$

Hence, the continuous-time asynchronous best response dynamics is a continuous-time Markov chain [4] $X(t)$ with state space coinciding with the configuration space $\chi$ of the game and transition rate matrix $\Lambda$ as follows: $\Lambda_{xy} = 0$ for every two configurations $x, y \in \chi$ that differ in more than one entry, and

$$\Lambda_{xy} = \begin{cases} |\mathcal{B}_i(x_{-i})|^{-1} & if\ y_i \in \mathcal{B}_i(x_{-i}) \\ 0 & if\ y_i \notin \mathcal{B}_i(x_{-i}) \end{cases}$$

[4] A ***Markov chain*** is a stochastic model describing a sequence of possible events in which the probability of each event depends only on the state attained in the previous event (https://en.wikipedia.org/wiki/Markov_chain).

for every two configurations $x, y \in \mathcal{X}$ differing in entry $i$ only, i.e., such that $x_i \neq y_i$ and $x_{-i} = y_{-i}$.

Regarding the convergence of BR-dynamics, a theoretical results states that for every distribution of the initial configuration $X(0)$ on $\mathcal{X}$, there exists a random time $T \geq 0$, finite with probability 1, such that $X(t) \in \mathcal{N}$ for every $t \geq T$, where $\mathcal{N} \subseteq \mathcal{X}$ is the set of Nash equilibria.

## 10.2 Best Response for Decentralized MAPF Algorithms

In the case of shared knowledge of the world or a portion, when an agent chooses the path to follow, its choice depends on other agents' actions. This is somehow connected to the concept of best response introduced in [10.1.1].

More in detail, the best response could be used either for the action to avoid conflicts or for the path choice in general.

- *Best response action* [10.2.1]. When possible collisions occur (especially for selfish agents), the involved agents act somehow to solve the conflict. To choose the individual best action, given the others', we need the concept of *Nash equilibria*.
- *Best response path* [10.2.2]. When the agents know the others' characteristics, specifically their position or their planned path, choosing the best path may depend on the plans of the others. The higher the knowledge, the more informed is the algorithm when selecting the best path. We must also consider that if we use this knowledge to constrain an agent's possible actions, we might threaten the solver, up to not finding any solution.

### 10.2.1 Best response action

Here, we consider the use of the best response and Nash equilibria [10.1.1] to avoid conflicts between agents. In our situation, a Nash equilibrium corresponds to the action configuration where each entry is the best choice an agent has to make to get closer to its target while avoiding conflicts.

Before starting, we make the following observation. The possible actions are 5: *move up, move right, move down, move left, wait*. Recalling the definition of Nash equilibria, every skid tries to move toward its goal. When not possible, it is forced to wait and this becomes the best response. By not moving, the agent is substantially a fixed obstacle and doesn't influence the other agent's choice.
So when no move actions are possible or when there are no Nash equilibria where agents move toward their goals, there are surely Nash equilibria in which one, many, or even all the agents wait. This means our system doesn't improve on that step, and we get stuck on local minima [8.2.1]. We will remark this problem next.

Now, we intend to apply Game Theory to understand the actions that neighbor agents have to make to avoid imminent collisions. By *"neighbors"* we mean those agents inside a *local grid area*. In the following analysis, the neighborhood is the $3 \times 3$ squared area whose center is the agent of interest (Figure 10.2). Since this square is centered in the agent's position, it moves accordingly.

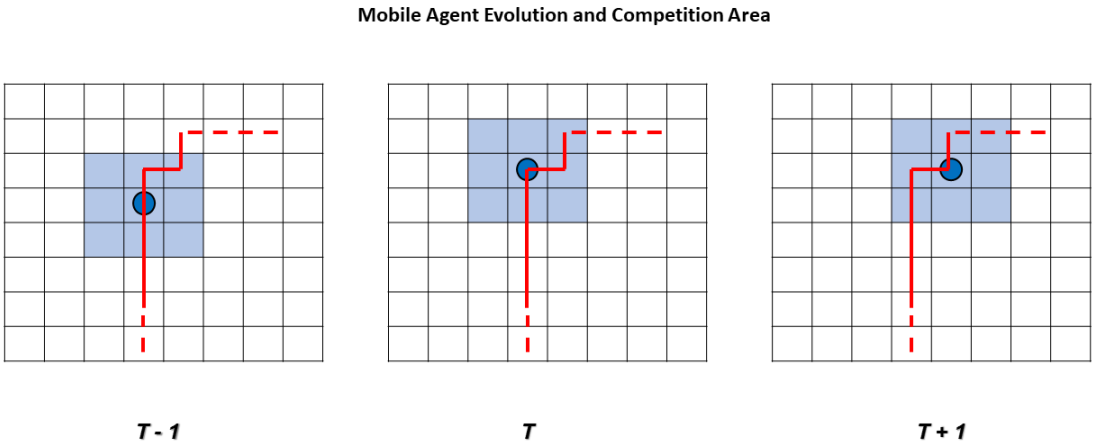



Figure 10.2: Time evolution of the agent's position (blue circle) inside the grid. the neighborhood area is a light blue $3 \times 3$ square. The planned path is shown in red color.

Following the approach described below, inductively, starting from some examples and thanks to the definition of an appropriate *utility function*, we find that at least one *Nash equilibrium* always exists.

For simplicity, we will consider an empty domain. The procedure easily extends to the case of a generic grid. In this setting, an action could either *move* or *wait* and costs 1. Considering the choice of the path as action might correspond to infinitely many possible actions for every agent. To solve imminent conflicts, instead, the number of actions not only is finite but restricted to the 5 above mentioned, summarized in Figure 10.3. This favors the choice of a small competition area.

Figure 10.3: Set of actions: up movement, right movement, down movement, left movement, wait.

The considered utility function is the following:

$$u(x_i, x_{-i}) = \begin{cases} u(x_i) & if\ x_i\ and\ x_{-i}\ don't\ bring\ collisions\ with\ i \\ -\infty & otherwise. \end{cases} \tag{10.3}$$

where $u(x_i)$ = - (cell movement cost + shortest path length from the new position). Since the movement costs are all set to 1, changing path requires *at least* an additive cost of 2, except when the new position is still on a shortest path whose length is equal to the previous one [5].

In the following, we consider some scenarios occurring with 2 agents. Then we analyze an extension to 3 agents. Finally, we obtain generalized observations on this technique.

### Case: 2 agents

**Example 1.** The first example is shown in Figure 10.4.
The table with agents' utilities is shown in Figure 10.5. By applying the definitions of best response and Nash equilibria, we conclude the Nash equilibria are the green-colored configurations (3). These three equilibria correspond to the situations where each agent navigates toward its goal without colliding with any other. Of course, if possible, players play actions that bring them closer to the final position without colliding.

**Example 2.** In Figure 10.6 we have the second example.
Comparing A and B utilities (Figure 10.7), we find there are two Nash equilibria. One corresponds to the unique configuration where each agent moves toward its goal; the other corresponds to B moving along its path and A waiting.

Oppositely to Example 1, there may be equilibria where some agents wait.

**Example 3.** Here (Figure 10.8), although agents could proceed on a single direction to reach their goals as fast as possible, but they would replan immediately or they would collide, since the directions are orthogonal. Similarly to the second NE of Example 2, the

[5] The utility function defined above doesn't coincide with a pairwise graphical game utility function. The game we are considering is with two or more players, depending on how many agents are inside the $3 \times 3$ grid, but we don't consider pairwise games.

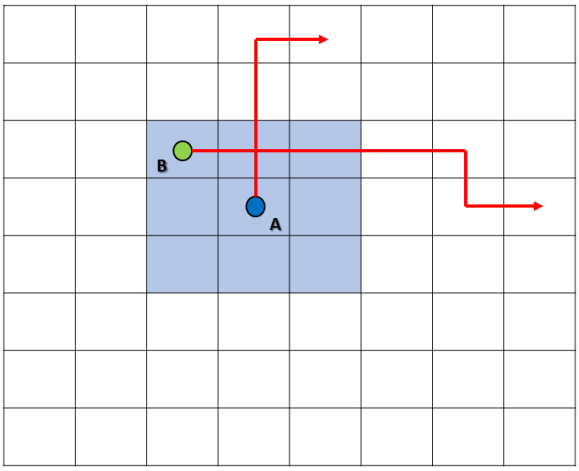



Figure 10.4: Example 1. Path planned by the agents in the competition area.

| B / A | ↑ | → | ↓ | ← | • |
|---|---|---|---|---|---|
| ↑ | -8 / -4 | -∞ / -∞ | -6 / -4 | -8 / -4 | -7 / -4 |
| → | -8 / -4 | -6 / -4 | -6 / -4 | -8 / -4 | -7 / -4 |
| ↓ | -8 / -6 | -6 / -6 | -6 / -6 | -8 / -6 | -7 / -6 |
| ← | -8 / -6 | -6 / -6 | -∞ / -∞ | -8 / -6 | -7 / -6 |
| • | -8 / -5 | -6 / -5 | -6 / -5 | -8 / -5 | -7 / -5 |

Figure 10.5: Example 1. Utility values table. Each cell is divided into agent A utility and agent B utility, depending on the chosen action. Green-colored cells refer to Nash equilibria.

optimal strategy is to let one of the players proceed and, while the other waits (Figure 10.9). There are two Nash equilibria. If it is not possible to wait, intuition suggests that

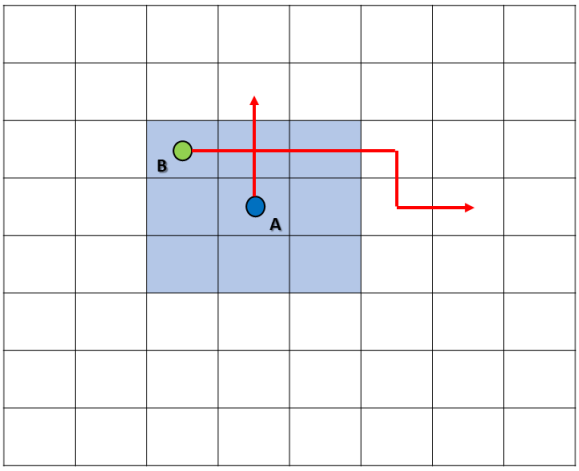



Figure 10.6: Example 2. Path planned by the agents in the competition area.

| B / A | ↑ | → | ↓ | ← | • |
|---|---|---|---|---|---|
| ↑ | -7 / -2 | - ∞ / - ∞ | -5 / -2 | -7 / -2 | -6 / -2 |
| → | -7 / -4 | -5 / -4 | -5 / -4 | -7 / -4 | -6 / -4 |
| ↓ | -7 / -4 | -5 / -4 | -5 / -4 | -7 / -4 | -6 / -4 |
| ← | -7 / -4 | -5 / -4 | - ∞ / - ∞ | -7 / -4 | -6 / -4 |
| • | -7 / -3 | -5 / -3 | -5 / -3 | -7 / -3 | -6 / -3 |

Figure 10.7: Example 2. Utility values table. Each cell is divided in agent A utility and agent B utility, depending on the chosen action. Green-colored cells refer to Nash equilibria.

one of the two agents departs from its shortest path, freeing the position for the other skid.

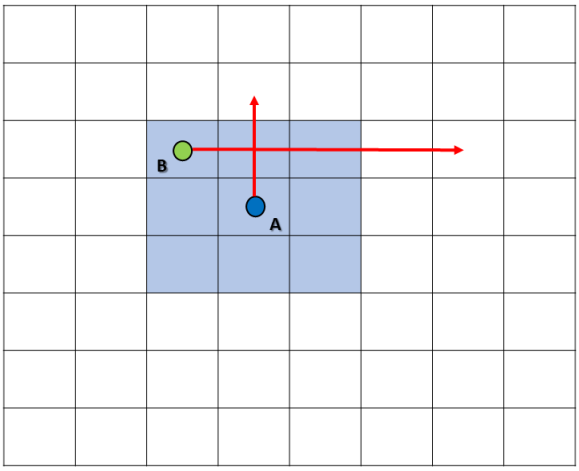



Figure 10.8: Example 3. Path planned by the agents in the competition area.

| B / A | ↑ | → | ↓ | ← | • |
|---|---|---|---|---|---|
| ↑ | -6 / -2 | - ∞ / - ∞ | -6 / -2 | -6 / -2 | -5 / -2 |
| → | -6 / -4 | -4 / -4 | -6 / -4 | -6 / -4 | -5 / -4 |
| ↓ | -6 / -4 | -4 / -4 | -6 / -4 | -6 / -4 | -5 / -4 |
| ← | -6 / -4 | -4 / -4 | - ∞ / - ∞ | -6 / -4 | -5 / -4 |
| • | -6 / -3 | -4 / -3 | -6 / -3 | -6 / -3 | -5 / -3 |

Figure 10.9: Example 3. Utility values table. Each cell is divided in agent A utility and agent B utility, depending on the chosen action. Green-colored cells refer to Nash equilibria.

**Example 4.** The two agents should move straight on the same direction, but with opposite verse. Figure 10.10 represents the situation. As observed before, in case of

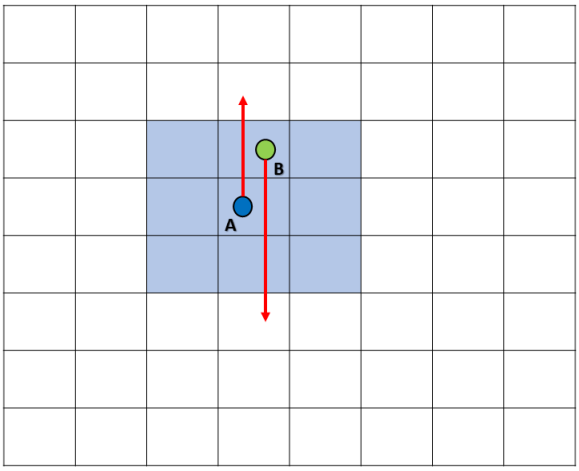



Figure 10.10: Example 4. Path planned by the agents in the competition area.

impossibility to improve the current position, either the skid waits, or it chooses a longer path. From the utility values table (Figure 10.11), unless they both wait, one agent proceeds on the way, while the other frees the tile, and viceversa.

**Example 5.** Before moving to the case of 3 agents in the competition area, we analyze a simple case with obstacles, to show the procedure is identical. We refer to Figure 10.12 for this. Once the utility values are computed (Figure 10.13), we find 3 Nash equilibria, all with both agents getting closer to the their destination.

### Case: 2 agents. Conclusion

Starting from these specific examples, we conclude there exists always at least one Nash equilibrium. 4 scenarios of possible collision could happen:

- Scenario 1. Non-adjacent agents. To reach their goal they need to occupy tiles that are shifted with respect to the agents' actual position, both in terms of $x$-axis and $y$-axis (Examples 1, 5). Easily, we verify the two agents can move toward their goals without colliding. Nash equilibria correspond to those configurations where agents follow their shortest path (or an alternative) and don't collide.
- Scenario 2. Non-adjacent agents. One agent's goal corresponds to those in scenario 1, while the remaining player just needs to move in a unique direction (Example 2). One possible Nash equilibrium is made by the agent constrained to the unique direction to proceed along its path, while the other changes direction, either following its shortest path or an alternative one (still shortest).

| B / A | ↑ | → | ↓ | ← | • |
|---|---|---|---|---|---|
| ↑ | -5 / -2 | -5 / -2 | -∞ / -∞ | -5 / -2 | -∞ / -∞ |
| → | -5 / -4 | -5 / -4 | -3 / -4 | -5 / -4 | -4 / -4 |
| ↓ | -5 / -4 | -5 / -4 | -3 / -4 | -5 / -4 | -4 / -4 |
| ← | -5 / -4 | -5 / -4 | -3 / -4 | -5 / -4 | -4 / -4 |
| • | -5 / -3 | -5 / -3 | -∞ / -∞ | -5 / -3 | -4 / -3 |

Figure 10.11: Example 4. Utility values table. Each cell is divided in agent A utility and agent B utility, depending on the chosen action. Green-colored cells refer to Nash equilibria.

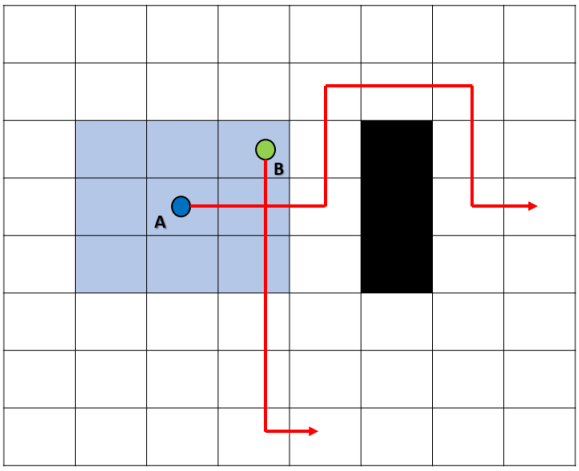



Figure 10.12: Example 5. Path planned by the agents in the competition area.

- Scenario 3. Non-adjacent agents. To reach their destination, they must navigate in one direction. If the direction is the same, the Nash equilibrium corresponds to both

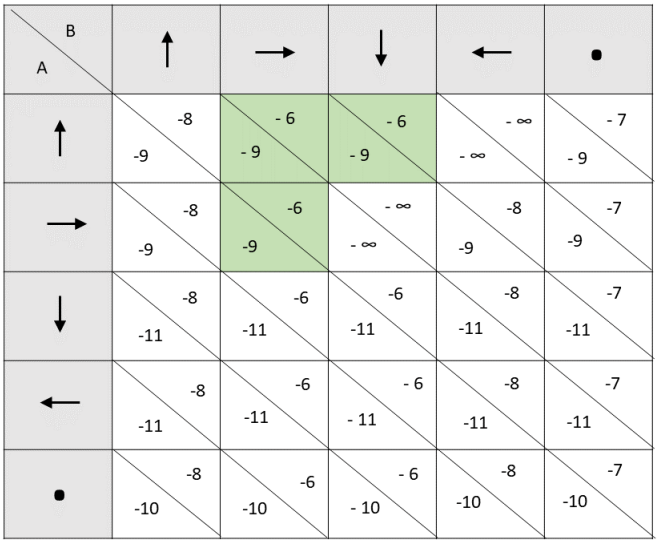



Figure 10.13: Example 5. Utility values table. Each cell is divided in agent A's utility and agent B's utility, depending on the chosen action. Green-colored cells refer to Nash equilibria.

the agents moving in that direction. In the case of orthogonal directions (Example 3), one agent waits, and the other passes. This is reasonable. When one agent passes near the other, the second one could move in some direction or wait; by waiting, a single cost unit is added to its overall path cost; if it moves to another cell, then at least two cost units are added.

- Scenario 4. Adjacent agents (Example 4). If directions are orthogonal, or equal with opposite verse, the NE with all players follow their respective shortest paths exists. If this is not the case, it is possible for one agent to shift to let the other pass.

In conclusion, there is always at least one Nash equilibrium, connected to the shortest path from the current position to the corresponding goal.

### Case: 3 agents

So far, our attention was for 2 agents. We want to understand if NE always exist even for 3 agents, and to characterize them.

**Example 1.** Consider the example in Figure 10.14.
To avoid a heavy use of complex figures of the utility functions, we directly present the results.
In this case, there are 3 Nash equilibria:

- agents A and C wait and B moves along its minimal length path;

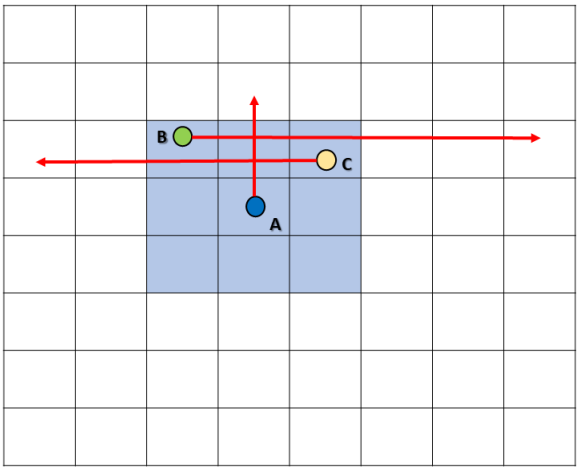



Figure 10.14: Example 6. Path planned by the agents in the competition area.

- agents A and B wait and C moves;
- agents B and C wait and A moves.

This confirms what we noticed for the 2-agents case: when it is not possible to "improve" the current position for all the skids contemporarily, the goal is to do so for as many agents as possible, and the remaining agents wait. Instead, if there is no possible improvement for any skid, some take longer paths.

**Example 2.** We make similar considerations for the example in Figure 10.15.
Here, all the agents move toward their target without colliding. When this is not possible, A and B keep moving, while C waits, being already close to its goal.
So, in this example the NE are:

- A moves right, B moves up, C waits;
- A and B move right, C waits;
- A moves up, B and C move down.

Note that the configuration (A moves up, B moves down, C waits) is not a Nash equilibrium, since C could improve its utility by going down.

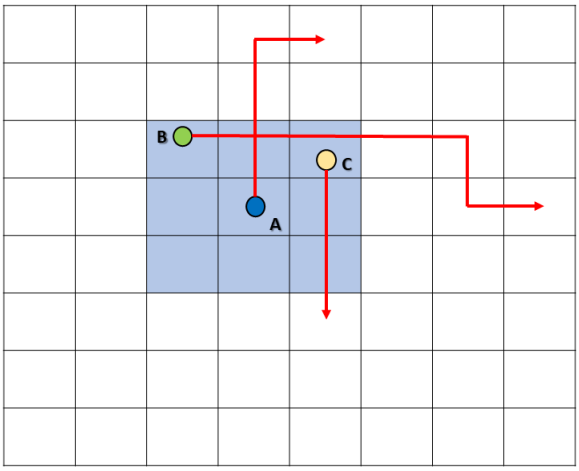



Figure 10.15: Example 7. Path planned by the agents in the competition area.

**Example 3.** Consider Figure 10.16. In this case there are many NE. Nevertheless, they

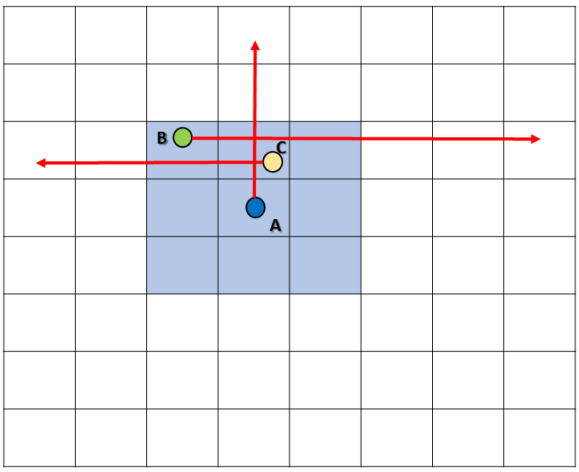



Figure 10.16: Example 8. Path planned by the agents in the competition area.

all reflect our previous observations. There are cases where everyone improves, or someone improves and others wait, or even cases where some agents choose longer paths.

More in detail:

- A and B move up, C moves left;
- A moves up, B moves down, C moves left;
- A moves up, B and C move left;
- A and C move up, B waits;
- A moves up, B waits, C moves right;
- A and B move right, C moves down;
- A and C move down, B moves right;
- A moves left, B moves right, C moves down;
- A waits, B and C move right;
- A waits, B moves right, C moves up;
- A, B, and C wait,

Note the drawback of this approach: the set of NE contains the configuration in which all agents wait. This solves conflicts for this time step but brings the same scenario to the next step.

### Case: 3 agents. Conclusion

As noticed, such as in the case of two agents, these are the possible situations: when it is not possible to "improve" the current position for all the skids contemporarily, the goal is to do so for as many agents as possible and the remaining agents wait. If there isn't an improvement possibility for any skid, some take longer paths.

### General cases and extension

We can easily extend the previous results on the case of a generic number of agents (at most 9, since the competition grid is the $3 \times 3$ square).

Let us consider the case of highly crowded areas. Suppose all the competition area's cells are occupied by skids or fixed obstacles, e.g., a narrow passage. As seen above, a trivial Nash equilibrium has all the agents waiting. To solve this limiting situation, we need some cooperation between robots, not this selfish approach. Only in this way we can escape such bottlenecks.

**Conclusion**

By introducing a *competition area*, we built a utility function with 5 possible players' actions. Irrespective of the various scenarios, the presence of fixed obstacles, and the number of agents, it is always possible to find the best actions to bring skids closer to their goals, without colliding. This is done using the concept of *Nash equilibrium*.

Despite this, by considering this approach to bottlenecks, we confirm again that our *random-style* collision avoidance technique is not perfect and should be improved.
The reason is that specifying one single procedure for every possible collision is restrictive, there are many scenarios where the best action is different. No algorithm could tackle all these possible scenarios. It would be excessive even from the implementation point of view.
***Applying Game theory and Nash equilibria for collision avoidance helps to understand that locally solving conflicts is not the best choice to reduce the sum of time steps***. Indeed, not only the agents don't choose the best action, but they may choose actions that bring to possible collisions the next time step. ***In conclusion, cooperation is the key to improving the performance of our algorithms (and our experimental analysis showed this too)***.

In this section, we are analyzing Nash equilibria considering no communication between agents. But it is exactly communication and cooperation that helps to improve results from the system point of view, as we know both from game theory and network traffic flow concepts.

### 10.2.2 Best response path

We have seen that congestion awareness helps to improve the performance of a basic self-interested approach. In our code's implementation, we exploit the idea of modeling congestion with macro grids, to let agents know the others (in particular, others' paths and positions). By combining this idea with the optimized path computation, we built our final optimized MAPF solver [8.1.1], whose solution is better than all previous methods [8.2.1].

We could improve this optimized variant. Indeed, considering the possible congested areas, each agent is substantially making its choice in response to the other agents' actions. This is very similar to the idea behind the game-theoretic learning process, specifically the Best Response Dynamics [10.1.2].
In the best response dynamics agents make choices based on their neighborhood, while in our final variant the skids choose a path depending on all the other agents. For this reason, we may consider all the agents to be neighbors, by modeling the *"interaction graph"* as a complete graph. This corresponds to saying that here the resource is the grid itself, and since all agents share this domain, they have an (abstract) link connecting them.

In other words, even ***the optimized variant of the basic search is connected to the idea of the best response dynamics***, selecting at every step each agent's best response (i.e., an action/path), to respond to the others.

Clearly, we must notice that, in a game theoretic setting, it is usually our intention to comprehend the properties of best response dynamics convergence. When applying game-theoretic learning processes to MAPF, it is difficult to talk about convergence, mainly for two reasons:

- Agents reach their goals at different time steps. Once arrived to their destination, every agent disappears. This is similar to say that the interaction graph of the agents loses some nodes during the process. We may solve this by considering agent to stay at target and applying the same *wait* action. In this way, the neighborhood (i.e., all agents, the interaction graph is complete) doesn't change.

- Even though agents stay at target, at some point every one reaches the corresponding goal. We should consider this moment as the "convergence" of the algorithm, or simply an "early stopping criterion"?

As for the discussion on Wardrop equilibrium [9.2.3], also for game theory we understand it is not easy to model agents' pathfinding as a learning process.

We still take inspiration from the notion of best response dynamics, trying to establish a way to push agents toward their goal in the best possible way.

We thought about three possible methods to improve our algorithm with a game-theoretic learning process.

- Starting from the final optimized search method, suppose at each step agents can choose between the shortest path without any congestion information (but adding a penalty every time a possible collision occurs) and the shortest path with congestion awareness.

- Similarly to the best response dynamics, agents could define a path taking into account the closest agents up to a certain distance, after which the grid is considered empty. In other words, the agent considers neighbors' actions to choose the best response.

- The two previous ideas are put together: an agent can choose between the shortest path without considering any information about congestion, and the shortest path with congestion awareness, where congestion awareness is restricted to a local window.

One of our future objectives is to implement and compare these algorithms (11). Before seeing more in detail each of these possible methods, we remind the reader that in the literature there are already some results on games theory applied to MAPF, but often leaving some not answered questions. Only during the last years, it seems the researcher community got full comprehension of the main aspects (Jordán et al. [2019]). Interestingly, one possible idea to apply best response dynamics to MAPF is to extract one basic solution and then apply, out of the MAPF framework, this theory.

### Game-theoretic learning process for MAPF: version 1

Starting from the final optimized search method, we give more freedom to the agent when choosing its path.

Suppose agents can regularly choose between the shortest path without considering any information on congestion (but adding a penalty every time a possible collision occurs) and the shortest path with congestion awareness. If the agent follows the shortest path, then it is not taking into account the macro grid weights since they bring too long conflict-free paths. Instead, if the best path is the shortest path with congestion awareness (weighted-shortest path), the unweighted shortest path would pass through a too highly crowded region. Consequently, the agent would have to solve locally too many collisions, bringing an elevated cost.

To compare the shortest and weighted-shortest paths, we consider their costs as follows.

- Shortest path with a local repair. We compute the shortest path on the unweighted grid and add a penalty every time this computed path intersects another agent's one.
- Weighted-shortest path. As already noticed before, we simply compute the shortest path with cell weights taken from the computed macro grid.

Note that in both cases we need to know the positions and paths of all the grid agents, as in the original optimized version.

We could decide if replanning each time a collision occurs or at regular time intervals.

Concerning the final optimized variant, we expect better results for the following reason. Sometimes it might be better to proceed through crowded areas and solve possible conflicts than choose too long conflict-free paths.

### Game-theoretic learning process for MAPF: version 2

We said the final optimized version of our MAPF solver has similar ideas to the *best response dynamics*. Specifically, given a neighborhood, that in this case is the full grid, the agent chooses the best path, once he knows those of all the other agents.
We try to give the agent more freedom. Differently from the original optimized version, now the agent's choice doesn't depend on all the grid's agents, but just on those in a specific local window. Note that by reducing the window in which considering the paths of neighbors, the agent has more freedom [8.3].

The idea is always similar to the one presented for the optimized version, but now we are not considering a *complete "interaction graph"*. In particular, agents could define a path taking into account the closest agents up to a certain distance, after which the grid is considered empty. In other words, the agent considers neighbors' actions to choose the best response.

Let us justify the correctness of the windowed knowledge. From the application point of view, it is not important to solve collisions that might happen far in the future. It requires higher communication between mobile agents and more constrained movements, two non-negligible problems. On the other hand, we mean to solve those conflicts close to the agent from a space-time perspective. It finds justification also in ordinary life.

Despite the similarity with the optimized version [8.1.1] in choosing the shortest path on a weighted grid, now we consider congestion differently. Instead of assigning weight to nodes based on the paths passing through them, each cell (or macro cell) of the local window has a cost proportional to the number of agents in its neighborhood. As for congestion awareness, the cost to pass from one node to its neighbor is the maximal cell cost. When building its path, an agent chooses the one with the smallest cost, i.e., the one that is both shortest and that avoids as much as possible those skids in the neighborhood.

By windowing the world's knowledge, we expect better results. Indeed, in the original optimized version, macro cells have an associate cell cost equal to the number of paths passing through them, irrespective of when this passage will happen. A generic skid tries to avoid potentially crowded areas thanks to the information summarized in the macro grid but while the agent's motion is time-dependent, this information is time-independent. Consequently, an agent might avoid some areas now, even though they will be crowded in the future. By modifying the definition of the macro grid and considering it locally, we solve this problem. One further aspect to take care of is if and when to replan. It is necessary to replan the path the world's knowledge is restricted to a local window. Once this window is passed, some further knowledge is needed. Otherwise, the process would go on as in the basic search. So after a specific number of steps, e.g., the time steps needed to leave the window, or a portion, the agent must replan with the same technique.

The main drawback with this version is to compute these local weights many times during the search.

Interestingly, the idea of windowing the shared knowledge of the agents is similar to that introduced by Silver when developing Cooperative Pathfinding algorithms [4.2]. For this reason, it would be interesting to apply the ideas of best response dynamics even for WHCA*. More in general, sharing knowledge with a portion of the system is the idea behind Hybrid Pathfinding (also known as Agent-Centered Search) [1.1.4].

#### Game-theoretic learning process for MAPF: version 3

The two previous ideas don't exclude each other. They can be combined to hopefully outperform them singularly.

An agent may choose between the shortest path without considering any information about congestion and the shortest path with congestion awareness, where congestion awareness is restricted to a local window and interpreted as in the second version above (each cell (or macro cell) of the local window has a cost proportional to the number of

agents in its neighborhood).

As in the first version, there would be two possible path choices.

- Shortest path with a local repair. We compute the shortest path on the unweighted grid and we add a penalty every time this computed path intersects another agent's one.
- Weighted-shortest path (with local window). We choose the weighted shortest path where nodes are weighted in the local window and have unitary costs outside.

Again, after a specific number of steps, the agent must replan, with the same technique.

We expect the same advantages as previous versions. Conflicts, and consequently the number of A* search calls, are highly reduced in the local window. At the same time, we don't have to mistakenly take into account too far agents' movements. Every agent has more freedom when choosing the best action but still uses cooperation to improve its performance. Finally, the drawback of version 2 repeats.

# Chapter 11

# Conclusion and Future Work

Multi-Agent Pathfinding (MAPF) started being studied during the last decades of the 20th century with practical solutions after 2000. Today's developed algorithms are mainly based on those ideas of cooperation presented during the first years of the 21st century.

Thanks to technological progress, MAPF started being applied widely over the last decade. One of the most interesting applications regards warehouse automation in Industry 4.0.

Research is still proceeding on the enhancement of these ideas. It seems to flourish the connection between path planning and AI data-driven models such as Machine Learning, Deep Learning, or Reinforcement Learning. Moreover, new variants of the problem need proper attention, such as Online Multi-Agent Pathfinding or Task Assignment and Pathfinding (TAPF).

In this work, we focused on decentralized grid-based solvers. Precisely, we divided the document into three parts:

- Introduction;
- Developed Algorithms;
- Links to Congestion Games and Network Traffic Flow.

The objective of the introduction is to let the reader get comfortable with the concepts of Multi-Agent Pathfinding, starting from a historical introduction, with an overview of the main ideas and algorithms, then going deeper into the formalization of our specific MAPF problem.
Our work is integrated with `Python` codes for system simulation and performance analysis. Data from simulations are visualized and analyzed with `MATLAB`, `R` software, and `PowerPoint`. To simulate multi-agent systems we used both custom and benchmark grids. For benchmark grids and scenarios, we referred to three real/synthetic domains from the community website.
To conclude this part, we made a complete review of the single-agent search methods on graphs, as they are the basis for planning in multi-agent systems.

The second part relates to the development, implementation, and comparison of decentralized algorithms. They have the same basic idea of selfish agents planning individually their path without taking care of anyone else until a possible collision occurs.
Different versions focus on different aspects of multi-agent systems, taking inspiration from the approaches available in the literature or exploiting new ideas: computational optimization, intelligent movements, and congestion awareness.

During the experimental phase, we compare our developed versions in terms of the number of avoided conflicts and search method function calls, the sum of time steps, maximal path length, and time consumption. These variables are highly correlated with each other and guarantee the full comprehension of the algorithm behaviors while augmenting the number of agents in the system.

Those methods with the best results are combined into a *final optimized version* that outperforms all previous algorithms.

We made interesting observations about the use of randomness in our algorithm. Specifically, oppositely to constraining the domain and/or the agents' motion, the *random-style* collision avoidance technique corresponds to assigning agents a higher level of freedom when moving into the system. The advantage is that the agents find a suboptimal path in the majority of the cases. On the other hand, its drawback is that when possible collisions occur, guaranteeing one agent to pass doesn't imply it gets closer to its goal. This could bring inefficient behaviors.

One of the possible future works is to modify the chosen approach to deal with collisions more efficiently while still maintaining selfish behavior.

Furthermore, experimental analysis was useful to observe that *cooperation* between agents improves the final performance by replacing the *selfish* behavior with some form of global or local cooperation. We compared our final optimized version with some of the best-known literature algorithms. Even though the major drawback of such cooperative methods is the lack of completeness, in general, they bring better results, dramatically reducing the number of replanning phases and search method calls and avoiding unintelligent behaviors.

In the third part, our goal is to establish connections between MAPF and the concepts of Network Traffic Flow and Game Theory. Only in the last years has some advanced comprehension been obtained in the literature. We intend to use these concepts in the near future, to understand, describe, and implement smart solutions to the MAPF problem.

Concerning Network Traffic Flow, it would be our future intention to formally adopt the definition of Wardrop equilibrium to MAPF, to use the linked idea of Price of Anarchy to quantify the performance of decentralized algorithms over centralized ones.

From the Game Theory point of view, we first translated our problem into a congestion game. Then, we used the static idea of Nash equilibria to prove the inefficiency of a selfish approach, due to the elevated number of different occurring scenarios for choosing the best conflict avoidance. Dynamically, we introduced the connection between MAPF and the Best Response Dynamics. As for the Wardrop equilibrium, we aim at formalizing this. Inspired by the concepts behind best response dynamics, we finally present three possible improvements to our final optimized solver. We intend to implement and compare them.

# Bibliography


Max Barer, Guni Sharon, Roni Stern, and Ariel Felner. Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. In *SOCS*, 2014. URL https://www.aaai.org/ocs/index.php/SOCS/SOCS14/paper/viewFile/8911/8875.

Maren Bennewitz, Wolfram Burgard, and Sebastian Thrun. Optimizing schedules for prioritized path planning of multi-robot systems. In *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No. 01CH37164)*, volume 1, pages 271–276. IEEE, 2001. URL https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.28.9997&rep=rep1&type=pdf.

Zahy Bnaya, Roni Stern, Ariel Felner, Roie Zivan, and Steven Okamoto. Multi-agent path finding for self interested agents. In *International Symposium on Combinatorial Search*, volume 4, 2013. URL https://www.bgu.ac.il/~zivanr/files/MAPF_SOCS2013.pdf.

Adi Botea, Martin Müller, and Jonathan Schaeffer. Near optimal hierarchical path-finding. *J. Game Dev.*, 1(1):1–30, 2004. URL https://webdocs.cs.ualberta.ca/~mmueller/ps/hpastar.pdf.

Y. Uny Cao, Alex S. Fukunaga, and Andrew Kahng. Cooperative mobile robotics: Antecedents and directions. *Autonomous Robots*, 4:7–27, 1997. doi: 10.1023/A:1008855018923. URL https://doi.org/10.1023/A:1008855018923.

Michal Čáp, Peter Novak, Jiří Vokřínek, and Michal Pěchouček. Asynchronous decentralized algorithm for space-time cooperative pathfinding. *arXiv preprint arXiv:1210.6855*, 2012. URL https://arxiv.org/pdf/1210.6855.pdf.

Michal Čáp, Peter Novák, Alexander Kleiner, and Martin Selecký. Prioritized planning algorithms for trajectory coordination of multiple mobile robots. *IEEE transactions on automation science and engineering*, 12(3):835–849, 2015. URL https://arxiv.org/pdf/1409.2399.pdf.

Kuang-Yuan Chen, Peter A Lindsay, Peter J Robinson, and Hussein A Abbass. A hierarchical conflict resolution method for multi-agent path planning. In *2009 IEEE Congress on Evolutionary Computation*, pages 1169–1176. IEEE, 2009. URL https://ieeexplore.ieee.org/document/4983078.

Crescenzi, Gambosi, Grossi, and Rossi. *Strutture di dati e algoritmi*. Pearson, 2012.

Boris De Wilde, Adriaan W Ter Mors, and Cees Witteveen. Push and rotate: cooperative multi-agent path planning. In *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, pages 87–94, 2013. URL https://www.ifaamas.org/Proceedings/aamas2013/docs/p87.pdf.

E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. doi: 10.1007/BF01386390. URL https://doi.org/10.1007/BF01386390.

Stefan Edelkamp and Jiirgen Eckerle. New strategies in learning real time heuristic search. In *Proc. AAAI Workshop on On-Line Search*, pages 30–35, 1997. URL https://www.aaai.org/Papers/Workshops/1997/WS-97-10/WS97-10-005.pdf.

Michael Erdmann and Tomas Lozano-Perez. On multiple moving objects. *Algorithmica*, 2 (1):477–521, 1987. URL https://dspace.mit.edu/bitstream/handle/1721.1/5602/AIM-883.pdf?sequence=2&isAllowed=y.

Ariel Felner, Roni Stern, S. E. Shimony, Eli Boyarski, Meir Goldenberg, Guni Sharon, Nathan R Sturtevant, Glenn Wagner, and Pavel Surynek. Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges. In *SOCS*, 2017. URL https://ojs.aaai.org/index.php/SOCS/article/view/18423.

The R Foundation. What is r? URL https://www.r-project.org/about.html.

Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. doi: 10.1109/TSSC.1968.300136. URL https://ieeexplore.ieee.org/document/4082128.

Christian Henkel and Marc Toussaint. Optimized directed roadmap graph for multi-agent path finding using stochastic gradient descent. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pages 776–783, 2020. URL https://arxiv.org/pdf/2003.12924.pdf.

Carlos Hernández and Pedro Meseguer. Lrta*(k). In *IJCAI*, 2005. URL https://www.ijcai.org/Proceedings/05/Papers/0764.pdf.

Robert Holte, M. Perez, Robert Zimmer, and Alan MacDonald. Hierarchical a*: Searching abstraction hierarchies efficiently. pages 530–535, 01 1996. URL https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.42.1704&rep=rep1&type=pdf.

Hopcroft, Schwartz, and Sharir. On the complexity of motion planning for multiple independent objects; pspace- hardness of the "warehouseman's problem.". *The International Journal of Robotics Research.*, 3:76–88, 1984. doi: 10.1177/027836498400300405. URL https://journals.sagepub.com/doi/10.1177/027836498400300405.

Renee Jansen and Nathan Sturtevant. A new approach to cooperative pathfinding. In *Proceedings of the 7th international joint conference on Autonomous agents and multi-agent systems-Volume 3*, pages 1401–1404. Citeseer, 2008. URL https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.161.1251&rep=rep1&type=pdf.

Jaume Jordán, Javier Bajo, Vicent Botti, and Vicente Julian. An abstract framework for non-cooperative multi-agent planning. *Applied Sciences*, 9(23):5180, 2019. URL https://www.mdpi.com/2076-3417/9/23/5180.

Sven Koenig. Agent-centered search: Situated search with small look-ahead. 1996. URL https://www.aaai.org/Library/AAAI/1996/aaai96-217.php.

Sven Koenig. Agent-centered search. *AI Magazine*, 22(4):109, Dec. 2001. doi: 10.1609/aimag.v22i4.1596. URL https://ojs.aaai.org/index.php/aimagazine/article/view/1596.

Sven Koenig. Benchmarks. http://mapf.info/index.php/Main/Benchmarks, 2022.

Samson Lasaulce and Hamidou Tembine. Chapter 1 - a very short tour of game theory. In Samson Lasaulce and Hamidou Tembine, editors, *Game Theory and Learning for Wireless Networks*, pages 3–40. Academic Press, Oxford, 2011. ISBN 978-0-12-384698-3. doi: https://doi.org/10.1016/B978-0-12-384698-3.00001-3. URL https://www.sciencedirect.com/science/article/pii/B9780123846983000013.

Ryan Luna and Kostas E. Bekris. Push and swap: Fast cooperative path-finding with completeness guarantees. In *IJCAI*, 2011. URL https://people.cs.rutgers.edu/~kb572/pubs/push_and_swap_ijcai.pdf.

Hang Ma and Sven Koenig. Ai buzzwords explained: Multi-agent path finding (mapf). *AI Matters*, 3(3):15–19, oct 2017. doi: 10.1145/3137574.3137579. URL https://doi.org/10.1145/3137574.3137579.

Hang Ma, Daniel Harabor, Peter J. Stuckey, Jiaoyang Li, and Sven Koenig. Searching with consistent prioritization for multi-agent path finding. AAAI'19/IAAI'19/EAAI'19. AAAI Press, 2019. ISBN 978-1-57735-809-1. doi: 10.1609/aaai.v33i01.33017643. URL https://doi.org/10.1609/aaai.v33i01.33017643.

MathWorks. Matlab documentation center. URL http://www.mathworks.it/it/help/matlab/.

python. About python programming language. URL https://www.python.org/about/.

Ralf Regele and Paul Levi. Cooperative multi-robot path planning by heuristic priority adjustment. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5954–5959. IEEE, 2006. URL https://ieeexplore.ieee.org/document/4058417.

Malcolm Ryan. Constraint-based multi-agent path planning. In Wayne Wobcke and Mengjie Zhang, editors, *AI 2008: Advances in Artificial Intelligence*, pages 116–127, Berlin, Heidelberg, 2008a. Springer Berlin Heidelberg. URL https://link.springer.com/chapter/10.1007/978-3-540-89378-3_12.

Malcolm RK Ryan. Exploiting subgraph structure in multi-robot path planning. *Journal of Artificial Intelligence Research*, 31:497–542, 2008b. URL https://www.aaai.org/Papers/JAIR/Vol31/JAIR-3115.pdf.

Qandeel Sajid, Ryan Luna, and Kostas Bekris. Multi-agent pathfinding with simultaneous execution of single-agent primitives. In *International symposium on combinatorial search*, volume 3, 2012. URL https://ojs.aaai.org/index.php/SOCS/article/view/18243.

Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial intelligence*, 195:470–495, 2013. URL https://www.ijcai.org/Proceedings/11/Papers/117.pdf.

Guni Sharon, Roni Stern, Ariel Felner, and Nathan R Sturtevant. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219:40–66, 2015. URL https://www.sciencedirect.com/science/article/pii/S0004370214001386.

David Silver. Cooperative pathfinding. In R. Michael Young and John E. Laird, editors, *Proceedings of the First Artificial Intelligence and Interactive Digital Entertainment Conference, June 1-5, 2005, Marina del Rey, California, USA*, pages 117–122. AAAI Press, 2005. URL https://www.aaai.org/Papers/AIIDE/2005/AIIDE05-020.pdf.

David Silver. Cooperative pathfinding, 2006. URL https://www.davidsilver.uk/wp-content/uploads/2020/03/coop-path-AIWisdom.pdf.

Trevor Standley. Finding optimal solutions to cooperative pathfinding problems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 24, pages 173–178, 2010. URL https://www.cs.huji.ac.il/~jeff/aaai10/02/AAAI10-039.pdf.

Roni Stern. Multi-agent path finding–an overview. *Artificial Intelligence*, pages 96–115, 2019. URL https://www.researchgate.net/profile/Roni-Stern-3/publication/336611576_Multi-Agent_Path_Finding_-_An_Overview/links/62532fe0b0cee02d69613905/Multi-Agent-Path-Finding-An-Overview.pdf.

Roni Stern, Nathan R. Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne T. Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, T. K. Satish Kumar, Eli Boyarski, and Roman Bartak. Multi-agent pathfinding: Definitions, variants, and benchmarks. *Symposium on Combinatorial Search (SoCS)*, pages 151–158, 2019. URL http://mapf.info/index.php/Main/Tutorials?action=download&upname=socs19a.pdf.

Bryan Stout. Smart moves: Intelligent pathfinding. *Published in Game Developer Magazine*, 1997. URL https://fmfi-uk.hq.sk/Informatika/Uvod%20Do%20Umelej%20Inteligencie/clanky/smartmov.pdf.

Nathan Sturtevant and Michael Buro. Partial pathfinding using map abstraction and refinement. In *AAAI*, volume 5, pages 1392–1397, 2005. URL https://www.aaai.org/Papers/AAAI/2005/AAAI05-221.pdf.

Nathan Sturtevant and Michael Buro. Improving collaborative pathfinding using map abstraction. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 2, pages 80–85, 2006. URL https://www.aaai.org/Papers/AIIDE/2006/AIIDE06-017.pdf.

P. Surynek. An optimization variant of multi-robot path planning is intractable. *Proceedings of the AAAI Conference on Artificial Intelligence*, 24:1261–1263, 2010. URL https://doi.org/10.1609/aaai.v24i1.7767.

Pavel Surynek. An application of pebble motion on graphs to abstract multi-robot path planning. In *2009 21st IEEE International Conference on Tools with Artificial Intelligence*, pages 151–158, 2009. doi: 10.1109/ICTAI.2009.62. URL https://ieeexplore.ieee.org/document/5367139.

Jur van den Berg, Jack Snoeyink, Ming Lin, and Dinesh Manocha. Centralized path planning for multiple robots: Optimal decoupling into sequential plans. 06 2009. doi: 10.15607/RSS.2009.V.018. URL http://www.roboticsproceedings.org/rss05/p18.pdf.

Prasanna Velagapudi, Katia Sycara, and Paul Scerri. Decentralized prioritized planning in large multirobot teams. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4603–4609. IEEE, 2010. URL http://www.cs.cmu.edu/~pscerri/papers/VelagapudiIROS10.pdf.

Glenn Wagner and Howie Choset. M*: A complete multirobot path planning algorithm with performance bounds. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3260–3267, 2011. doi: 10.1109/IROS.2011.6095022. URL https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.221.1909&rep=rep1&type=pdf.

Ko-Hsin Cindy Wang, Adi Botea, et al. Fast and memory-efficient multi-agent pathfinding. In *ICAPS*, pages 380–387, 2008. URL https://www.aaai.org/Papers/ICAPS/2008/ICAPS08-047.pdf.

Ko-Hsin Cindy Wang, Adi Botea, et al. Tractable multi-agent path planning on grid maps. In *IJCAI*, volume 9, pages 1870–1875. Pasadena, California, 2009. URL http://users.cecs.anu.edu.au/~cwang/ijcai09-paper.pdf.

Charles W Warren. Multiple robot path coordination using artificial potential fields. In *Proceedings., IEEE International Conference on Robotics and Automation*, pages 500–505. IEEE, 1990. URL https://ieeexplore.ieee.org/document/126028.

Adam Wiktor, Dexter Scobee, Sean Messenger, and Christopher Clark. Decentralized and complete multi-robot motion planning in confined spaces. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1168–1175, 2014. doi: 10.1109/IROS.2014.6942705. URL https://ieeexplore.ieee.org/document/6942705.

Alexander Zelinsky. A mobile robot navigation exploration algorithm. *IEEE Transactions of Robotics and Automation*, 8(6):707–717, 1992. URL https://ieeexplore.ieee.org/document/182671.