Integradora IV - A brief introduction to Multi-Agent Path Finding Systems and a Proposal For a Simple and Distributed MAPF Algorithm

RAUL F. RODRIGUES and PEDRO G. R. PACHECO
Pontifícia Universidade Católica do Rio Grande do Sul

Multi-Agent Systems is a well-studied field in artificial intelligence, robotics, logistics, and theoretical computer science. We give a brief introduction to the topic, providing basic necessary knowledge on terminology and concepts. Alongside this introductory view, we propose an implementation of a conflict avoidance algorithm that in this paper we called Naive Conflict Avoidance and a web-based simulator that enables the user to create agents and their goals and obstacles. In the simulator another MAPF algorithm can be used called Conflict-Based search, that was implemented in python by another MAPF researcher. Several environment settings are described and tested using the simulator to compare the two algorithms.

Additional Key Words and Phrases: Artificial intelligence, autonomous systems

ACM Reference Format:

Raul F. Rodrigues, Pedro G. R. Pacheco. 2022. A brief introduction to MultiAgent Systems with a focus on MAPF. *ACM Trans. Graph.* XX, X, Article XXX (April 2022), X pages.
DOI: XX

1. INTRODUCTION

Artificial intelligence is one of the most important fields of Computer Science, being the driving force of profound cultural and social changes today[Makridakis 2017]. AI is present today in a multitude of fields and applications like recommendations systems[CHA et al. 2019], self-driving cars[Shalev-Shwartz et al. 2017], human resources management systems[Vardarlier and Zafer 2020], high frequency trading[Arifovic et al. 2019], fiscal fraud detection[Dhieb et al. 2020] and a lot more[Eur 2020].

Multi-Agent Systems is a sub-field of Artificial Intelligence [Wooldridge 2002] that is also growing and appearing more in our lives, with things like self driving cars[Shalev-Shwartz et al. 2017], and in the industry, with robots that manage entire distribution centers[Ma and Koenig 2017], the need for intelligence autonomous systems to interact with one another in the real world will also keep growing.

The use of autonomous drones in industry and other settings in real life has been increasing steadily and is set to continue accelerating in the coming decades. For robots that need to move in the real world, there is a clear necessity for path-finding and collision avoidance with obstacles and other robots or people. The field of Multi-Agent Pathfinding explores these problems, trying to find solutions and algorithms to efficiently and safely navigate agents in difficult environments [Stern et al. 2019b].

In this article, we developed a system to evaluate two different solutions to the problem of solving and avoiding collision in multi-agent pathfinding problems. The first is a very simple, yet effective and scalable algorithm proposed by [Savkin and Huang 2019] we have named Naive Conflict Avoidance. The is a more advanced and well studied conflict avoidance algorithm named Conflict-Based Search first described in [Savkin and Huang 2019]. Both algorithms will be used in an simulator created for the browser that has the capabilities of adding agents and their objectives, and obstacles.

We propose these algorithms as solutions to aerial drones being used in areas of conservation. These autonomous drones must avoid air collisions as they will sometimes fly in more congested zones. We will demonstrate an abstracted simulator with a discrete environment, a flexible number of agents and obstacles. In this environment, agents will have to reach their goals while avoiding other agents and obstacles.

2. DRONES FOR CONSERVATION

The Amazon rain forest today faces a dual threat of stress from Climate Change and deforestation [Malhi et al. 2008]. The Brazilian government has created several permanent conservation areas and nature reserves in the Amazon rain forest area to protect it from mining, logging, and clearing-cut ranching. However, policing these areas is extremely difficult [Fang et al. 2019], due to their size, terrain, and the fact that the government organizations responsible for the maintenance and surveillance of these areas are underfunded and stretched quite thin over all areas.

Several technologies are currently used to monitor permanent areas of conservation (PAC), such as satellite imagery and aerial photography [iba]. Ground sensors powered by solar panels that use satellite communication are also used. Another novel method that is starting to get traction all over the world for the monitoring of conservation areas is the use surveillance drones.

Aerial type autonomous vehicles, or ‘drones’, offer a flexible and affordable solution to specific set of challenges of nature conservation monitoring [Sandbrook 2015]. Today, the use of drones for the conservation of the natural environment is detailed and recorded by the organization `www.conservationdrones.org`, with several projects in Australia, the United States, the United Kingdom, and Switzerland. The projects employ fixed-wing drones (essentially miniature airplanes) and multirotor copters [Con]. The use of drones to monitor very large areas started in the agricultural sector, to analyze crop yields, and to monitor growth problems such as pests [Tripicchio et al. 2015].

One of the activities that aerial drones are suitable for is boundary patrols and collection of evidence of illegal activities [Sandbrook 2015], such as deforestation. Drones can also not only photograph and video to document the infractions but also help ground based local law enforcement catch the perpetrators. The evidence collected in the field can then be used to secure prosecutions.

These projects often employ more than one drone, due to the size of the area that they need to monitor. However sometimes these drones do encounter each other in the field and thus need to avoid conflict with each other. Despite being much more affordable than manned vehicles, these are still quite an expensive solution and the organizations that run them are often run on a quite limited budget. Therefore, reliable and fast collision avoidance is critical for the effective use and success of these monitoring missions.

3. RELATED WORK

In this section, we will present a couple of related articles regarding MAPF and Multi-Agent Systems in general, both as theoretical basis for this article and as good extra reading on the subject. These articles were particularly interesting in their subject related to the goals of this article.

[Ma and Koenig 2017] gives an extremely brief and concise overview of the academic study of Multi-Agent Pathfinding systems. It uses real-world examples to explain certain nuances regarding optimization and complexity class of MAPF algorithms.

It brings the case of an agent dense environment with a exponentially high number vertices, such that it becomes very difficult to find the shortest path quickly, but the authors suggest that there are ways for improving those situations and cites 2 of them. Then it provides two state-of-the-art solutions: the first one is treating all conflicts in a group then repeating the process for subsequent conflicts; the second one is treating each conflict recursively, where the colliding robots are not to traverse the point in which the conflict happened, then repeating the process.

Another piece of work that was investigated points to four solutions to problems of generalizations of real-world scenarios to multi-agent pathfinding systems [Ma et al. 2017]. The proposed scenarios cannot be easily or efficiently modeled by traditional MAPF techniques. The first is about agents working in teams to reach a goal. The second is about agents passing payloads to one another to reach their goals. The third is about agents using human created paths called *highways* to make their behavior more predictable. The fourth one is about agent's planning failing more gracefully.

[Savkin and Huang 2019] proposes several algorithm for both avoiding conflicts in a network of aerial drones and maximizing the coverage area for surveillance and monitoring. The system studied is similar to the one used in this in its characteristics except for the fact that the environment was continuous.

4. MULTI-AGENT SYSTEMS BASIC CONCEPTS

This following section will use as basis the previously mentioned book by Michael Woolridge, *An Introduction to MultiAgent Systems* [Wooldridge 2002], to create an overview of the basics of Multi-Agent Systems.

4.1 Agent

An agent is a computer system located in some environment and capable of autonomous action in this environment to meet its design objectives. Agents gather information about the environment throughout sensor input, and according to those inputs and the inputs from actions of other agents, they take actions to achieve their goals. Thus, these actions can affect and modify the environment. But this ability to *interact* with other agents are one of the defining features of agents in the field of Multi-Agent Systems.

For some applications, agents can have the ability to learn from their experiences, and they also can be a simple control systems, such as a thermostat, that reacts to the change of temperature of the environment and turns on the air conditioner to fit it‘s goal, that is to maintain the temperature at a specific level.

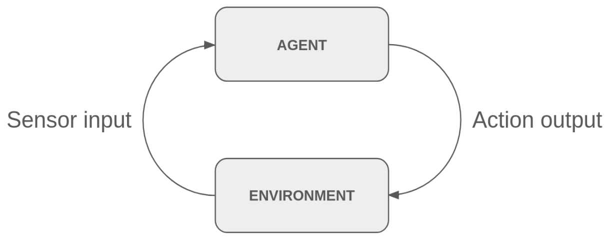

Fig. 1. Agent as described in Wooldridge book.

Agents in Multi-Agent Systems do not have to be intelligent to work effectively in their environment. It is possible to understand the various independent processes running in a Unix-like operating system as multiple agents where the environment is the file-system and main memory and the way to interact with the environment is through and system calls. As shown in Figure 2, each of the processes running concurrently in the system would be seen as agents and would interact with the environment through system calls and I/O calls. Processes in Unix systems are able to communicate with one another either through system files (magic files) or through structures and libraries provided by the operating system; one example would be Linux's *mq_open* [mqo] and which can be used to send messages between processes securely. In this scenario as the system's daemons and cron jobs are viewed as independent agents. It then becomes very clear that most of these agents aren't actually intelligent but the functionality they offer is incredibly important. Often, these daemons even have the word *agent* in their process name, such as *bluetooth-agent* found in many Linux distributions.

4.2 Intelligent Agents

Intelligent Agents are way more complex than a process running in a Unix system. They are categorized by having three main capabilities, listed and suggested by Wooldridge and Jennings, and they are the following: Reactivity, Proactivity, and Social Ability.

Reactivy: This capability expresses that an intelligent agent can perceive the environment and its history, and react to it.

Proactiveness: Proactiveness is the ability of the intelligent agent to take actions in order to carry out its purpose.

Social ability: Interacting with other agents, and also with humans, in order to fulfill its goals, is also a capability of intelligent agents.

Building a purely goal-directed systems is not hard, building purely reactive systems is also not difficult. However, what turns out to be difficult is building a system that achieves an effective balance between goal-directed and reactive behavior.

4.3 Formalization

In this section we are going to formalize the basic concepts of multi-agent systems.

First let's consider an environment *E*, composed by of a finite number of states:

$$E = \{e, e`, ...\}$$

It is not necessary that this environment be discrete, in this case it is just a modelling decision made to ensure that easier to explain this scenario.

An agent is capable of performing a finite number of actions *Ac* that will modify the state of the environment:

$$Ac = \{\alpha, \alpha`, ...\}$$

A run *r*, which is the representation of an agent in an environment, starts with an initial state *e0*, then the agent chooses an action to take and that action modifies the environment, generating another state, then the agent will react to that state, and the process continues as follows:

$$r : e_0 \xrightarrow{\alpha_0} e_1 \xrightarrow{\alpha_1} e_2 \xrightarrow{\alpha_2} e_3 \xrightarrow{\alpha_3} \dots \xrightarrow{\alpha_\mu} e_{\mu-1}$$

R can be defined as the set of all possible runs. R^{Ac} Is the subset of the runs that end with an action. R^E Is the subset of the runs that end with an environment state.

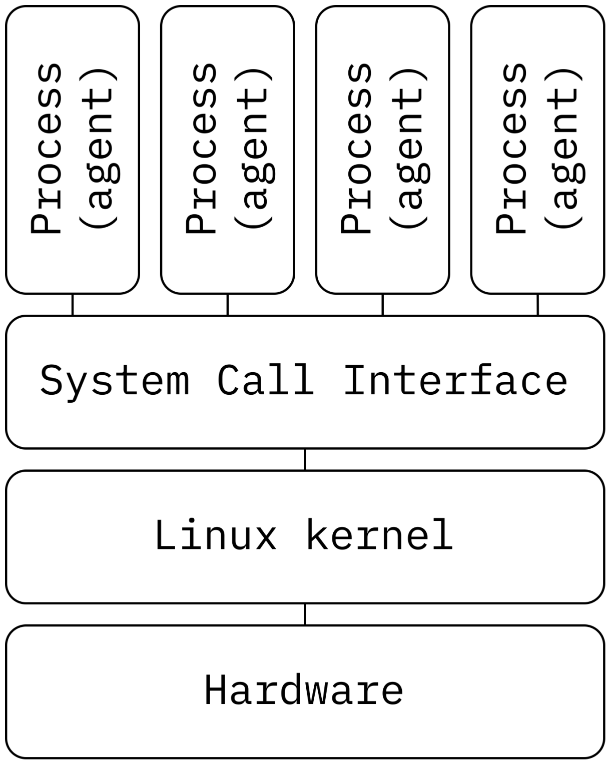

Fig. 2. A simplified view of the architecture of a Linux system

4.4 Environment

The environment is where the agents are situated, it can be a physical environment, or a digital one. They have four basic properties that determine the complexity of the environment and the difficulty of the agents to adapt to the environment, and thus fulfill their goals.

Accessible vs **Not Accessible**: accessibility refers to the level of access of information that the agent has about the environment in which it is located. If the agent has full knowledge about the environment at all times, that environment is classified as accessible. If there are limitations to knowledge, be it in amount or time, the environment is classified as not accessible. Of course, there are degrees to the accessibility of an environment.

Deterministic vs **Non-deterministic**: a deterministic environment is one in which any action has a single guaranteed effect - there is no uncertainty about the state that will result from performing an action. A non-deterministic one is the complete opposite, there may be probabilistic or stochastic variable to the effect of an action made by an agent.

Static vs **Dynamic**: A static environment is one that can be assumed to remain unchanged except by the performance of actions by the agents inside it. However, the dynamic environment is one that has other processes operating on it and hence changes in ways beyond the agents control.

Discrete vs **Continuous**: An environment is discrete if there. are a fixed, finite number of actions and percepts in it. The continuous is the exact opposite.

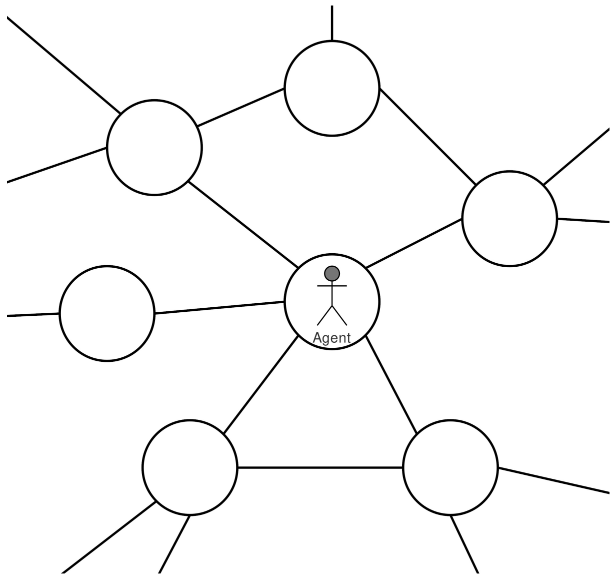

Fig. 3. An agent in a discrete environment, the agent can only move to a finite number of states from the current state.

5. MULTI-AGENT PATHFINDING

Multi-agent Pathfinding is a sub-field of Multi-Agent Systems focused on optimized pathfinding for multiple agents from their current locations to their target locations without colliding with each other [Ma and Koenig 2017]. Being a generalization of the pathfinding and *shortest-path problem*, it inherits the complexity of the pathfinding and adds collision and conflict solving and consensus on top of it. The optimization may focus on path cost of each individual agent or on the total sum of the path costs of all agents.

Being a generalization of pathfinding, many multi-agent pathfinding algorithms are constructed using traditional pathfinding algorithms such as A*, or based on reduction to other well-studied

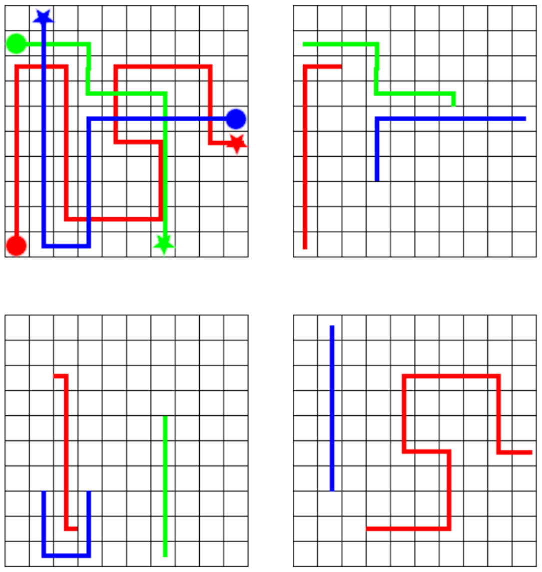

Fig. 4. caption

problems. However, such algorithms are typically incomplete or too costly;

In 2012 *Amazon*, one of the major multinational technology companies today focused on e-Commerce, cloud computing, digital streaming and artificial intelligence, acquired *Kiva Systems*. Founded on 2003, *Kiva Systems* had developed robots used in ware houses and distribution centers to automate the fetching of goods.

5.1 Pathfinding

Pathfinding is the key problem that Multi-Agent Pathfinding tries to tackle. Pathfinding can be generalized as the task of routing the shortest path between two points [Nakagaki et al. 2001]. In Graph Theory, pathfinding is directly mapped to *shortest path problem*, which examines the shortest path between two vertices in a graph [Russell and Norvig 2010]. Under the generalized circumstances that accept negative edge weights of this article and as with most real-world pathfinding problems, *shortest path problem* is an NP-complete problem [Zeitz 2023].

The two most fundamental pathfinding algorithms are the *breadth-first search* and the *depth-first search* algorithms. Algorithms *Dijkstra's* and *A** strategically eliminate possible paths using heuristics and dynamic programming techniques. This article will use the *A** algorithm to find the optimal path for each agent.

The main focus of this article is not which pathfinding algorithm the agents are using or the performance of each individual agent, but how well the agents deal with collisions and conflicts. In this case, all that is necessary is that the algorithm used must be optimal and complete, that is, the algorithm finds the optimal lowest-cost path and finds all possible outcomes of the problem. Both the *A** algorithm and the algorithm used within the conflict-based search collision resolution algorithm are optimal and complete [Sharon et al. 2015a].

5.2 Utility functions

An *utility function* is a function that measures *utility* of a given state. Utility is a numeric value that represents how "good" the state is for the agent to be in. An agent "wants" (here we are using the *intentional stance* discussed earlier) to be in the state with the highest utility. We say that the agent wants to maximize utility in the environment. A utility function can be used in the agent to calculate the utility of it's neighboring states or the states of the entire environment can be preset and the agents are only responsible for navigating the environment.

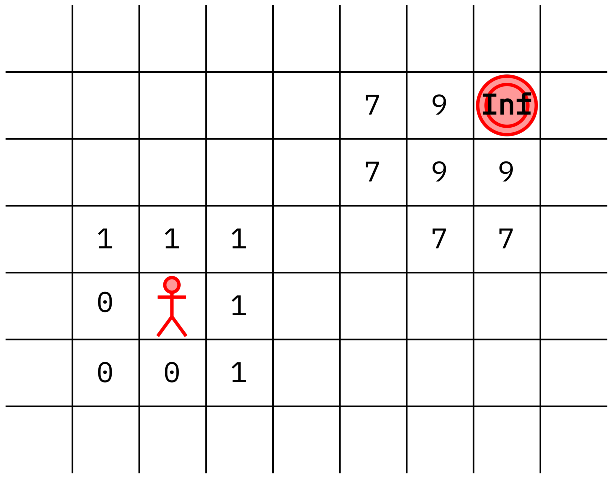

Fig. 5. Diagram showing an agent inside a discrete environment that has had it's states assigned with utility. The utility of the states near he end goal, represented by the red circle, is higher that of states far away. The utility of the end goal is infinite so it can be the highest in the environment.

Figure 5 an agent in a discrete environment with states that have been assigned a utility value. The agent depicted as a stick figure on the bottom left and the states near it show the gradient of utility, with the state nearer to the end goal having a higher utility. The goal, represented by the red circle on the top right, having infinite utility, is just a way of making sure that the agent will not leave the goal after it reaches the goal.

5.3 Conflict

Conflicts are the main problem that Multi-Agent Pathfinding systems try to solve or avoid. A conflict is an event an agent being an obstacle for another agent. There are four main types of conflicts [Stern et al. 2019a]:

(1) **Vertex conflict**: where two agents try to occupy the same space at the same time;

(2) **Edge conflict**: where two agents that are not occupying the same space try to occupy the same the space at the same time. If Vertex conflicts are not allowed, then Edge conflicts are not allowed as well.

(3) **Following conflict**: where at a certain time step an agent occupies a location that was occupied by another agent in the previous step.

(4) **Cycle conflict**: where at least three agents form a Following conflict cycle. Each agent occupies a space that at the previous time step was occupied by another agent. If Following conflict are not allowed, then Cycle conflicts are not allowed.

(5) **Swapping conflict**: where two agents swap spaces in a single time step.

5.4 Centralized vs Decentralized Algorithms

Multi-Agent Pathfinding algorithms can be separated into two distinct groups: centralized and decentralized(or distributed) methods[Sharon et al. 2015b].

5.4.1 *Centralized.* In this category, there is one planner for multiple agents, so this one planner will compute and find the solution for every agent in the same environment, the centralized approach assumes that all knowledge about all agent intents and their work spaces is given to a central server in order to plan optimal paths, with avoidance of collisions, for all agents[Long et al. 2018]. The centralized method also includes the event in which we have a separate CPU for each of the agents, but full knowledge sharing is assumed and a centralized planner controls all agents[Sharon et al. 2015b].

These centralized methods are often difficult to scale to large systems with many agents and their performance can decline when the goals are reassigned too often[Long et al. 2018]. Furthermore, the centralized methods are impossible to apply when multiple agents are deployed in a dynamic environment[Long et al. 2018].

In a real-world scenario, decentralized methods rely mainly on a well-founded communication network between agents and the central server[Long et al. 2018].

5.4.2 *Decentralized.* In a decentralized method, each agent has its own computer power and planner and could apply different communication paradigms from the other agents in the same environment[Long et al. 2018]. In order to achieve their goals, agents communicate with each other to find a conflict-free solution[Ho et al. 2022].

5.4.3 *Differences between them.* Centralized methods are recommended when the environment where the agents are inserted is not dynamic, because the planner would have to recalculate every time the environment suffers any changes, and in a dynamic environment, it is really often.

When thinking about scalability to larger problems, decentralized approaches, which decompose an initial problem into a series of searches, are the best way to deal with these problems. Decomposing the problem into several sub-problems can significantly reduce the computation needed in order to solve the given problem. [Wang et al. 2009].

The centralized approach, which incorporates a global decision maker to plan the paths of all units simultaneously, has prohibitive complexity and cannot be scaled to many units[Wang et al. 2009].

6. ALGORITHMS

6.1 Naive Conflict Avoidance

The Naive Conflict Avoidance was an algorithm proposed by [Savkin and Huang 2019]. This algorithm primarily received information about each agent starting position and goal, about the environment size and all the obstacles contained in that environment.

Every agent receives a priority value. This priority value is unique to all agents in the environment. Then a pathfinding algorithm is used to calculate the a path for each agent in order to its goal. Because the priority value is unique to each agent, deadlock scenarios in which each agent is waiting for another agent to move in a circular fashion do not occur.

In our implementation, the $A*$ pathfinding algorithm was used. The scope of this research takes the internal pathfinding algorithm as a black-box, whose performance or efficiency is not taken into consideration. Our simulator and results do not take into consideration the time taken to calculate the path for each agent, as the focus of this paper is exclusive the conflict and collision solving nature of the algorithms sitting on top of the pathfinding algorithm. The algorithm $A*$ was chosen because it is both optimal and complete[Russell and Norvig 2020]. Any other complete and optimal pathfinding algorithm could have been chosen to create the same comparative results.

Each agent also has a *exclusion area* that represents the sensory ability of the agent. The agent is capable of detecting if there are other agents inside *exclusion area.* Our proposed implementation uses a symmetric area around the agent of size 1. Plus an additional 4 tiles, 2 tiles away from the agent. Because this area is symmetric around the agent, if an agent detects another agent, then that means that the other agent detects the first agent.

Algorithm 1 Naive Conflict Avoidance($agents$, $graph$)

```
1:  $continueFlag \leftarrow true$
2:  $step \leftarrow 0$
3:  while $continueFlag$ do
4:      $continueFlag \leftarrow false$
5:
6:      for each $agent \in agents$ do
7:          if $step < agent.path.length$ then
8:              $continueFlag \leftarrow true$
9:          end if
10:     end for
11:
12:     if !$continueFlag$ then
13:         break                                  ▷ Simulation ends
14:     end if
15:
16:     $agntsWthCollisions \leftarrow \emptyset$
17:     $agntsWthCollisions \leftarrow$ testCollisions($agents, step$)
18:
19:     if $agntsWthCollisions.length \neq 0$ then
20:         for each $agent \in agntsWthCollisions$ do
21:             $stop \leftarrow$ false
22:
23:             $collisions \leftarrow \emptyset$           ▷ collisions are agents
24:             $collisions \leftarrow agent.collisions$
25:             for each $col \in collisions$ do
26:                 if $agent.priority > col.priority$ then
27:                     $stop \leftarrow$ true
28:                 end if
29:             end for
30:
31:             if $stop$ then
32:                 $agent.path \leftarrow$ bubble($agent.path, step$)
33:             else
34:                 $agent.path \leftarrow$ astar($agent, step$)
35:             end if
36:         end for
37:     end if
38:     $step \leftarrow step + 1$
39: end while
40:
41: return $agents$
```

At each step of the simulation, each agents checks if there's another agent inside their exclusion area. If an agent detects an-

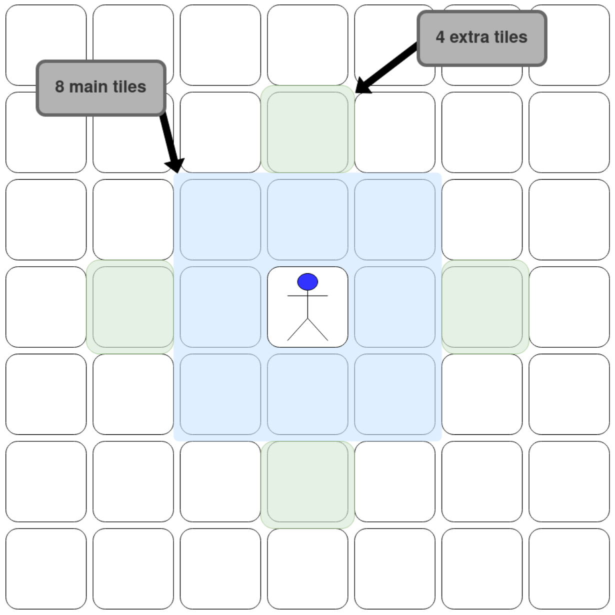

Fig. 6. Diagram showing the detection radius of an agent according to the Naive Collision Avoidance algorithm.

other agent, they share their priorities, and the one with the highest priority stops for one step of the simulation, and the other agent will reroute their path, taking the stopped agent, the highest priority agent, as an obstacle for the pathfinding algorithm. In our implementation this process of stopping the agent was achieved but adding a *bubble* to their pathing step list, as shown in Figure 8.

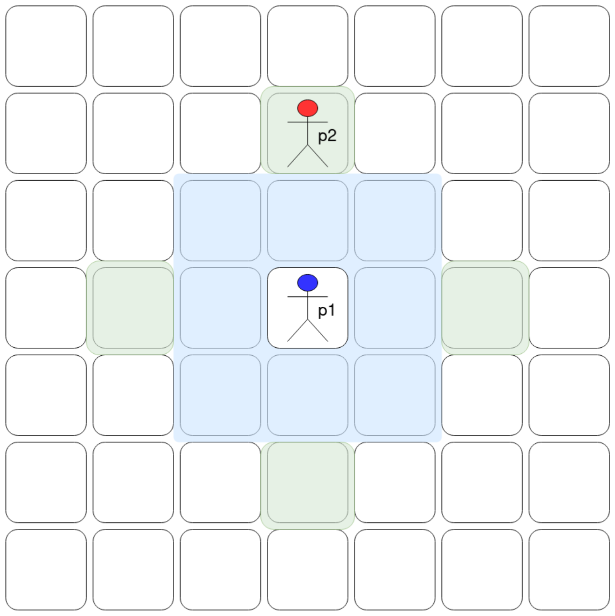

Fig. 7. Diagram showing the detection radius of an agent with another agent inside it.

After the agent reaches their goal, they leave the simulation and are no longer part of any the calculations for any other agent. The

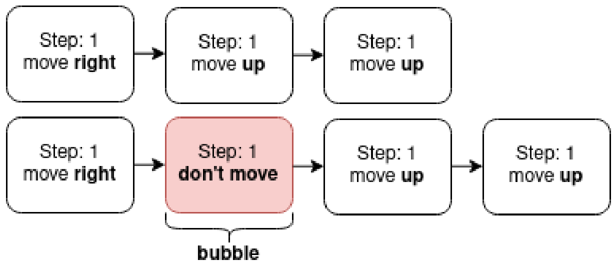

Fig. 8. Diagram showing the method used in order to make an agent stop at a given step of the simulation.

algorithm will continue to run until all agents have no conflicts and achieve their goals.

An important aspect of this method is that it does not deal with actual conflicts but tries to stop them from happening, *avoiding* having to deal with true pathing conflicts.

6.2 Conflict Based Search(CBS)

The Conflict-Based Search algorithm is a centralized MAPF algorithm that uses recursion trees to find a solution before the agents move.

The *MAPF* problem is solved by *CBS* by decomposing it into many single-agent pathfinding problems, so it will reduce a lot of the complexity of each problem, in comparison to the former problem. The *CBS* algorithm is based on tree recursion, so techniques to make tree recursion more efficient, such as alpha-beta pruning, can be used. The algorithm *CBS* can be implemented to be complete and is optimal [Sharon et al. 2015a]. This implementation of *CBS* also uses $A*$ in order to find the paths for each agent.

On the first part of the CBS algorithm, the paths for each agent are computed using a pathing algorithm. On the second part the algorithm goes through every path and finds all the where two agents are the same place at the same step of the simulation. Lastly the algorithm branches and at each branch it adds a constraint where at least one of the conflicting agents cannot be at that point at that step of the simulation.

A conflict is a tuple:

$$(a_i, a_j, vmt)$$

, where both agents ai and aj occupy the vertex v at the same time step t. A consistent solution can be invalid if, despite the fact that the paths are consistent with their individual agent constraints, these paths still have conflicts. A solution is valid if all its paths have no conflicts[Sharon et al. 2015a].

Constraints are tuples:

$$(a_i, v, t)$$

. A consistent path for an agent is a path that satisfies all its constraints[Sharon et al. 2015a].

This process is then repeated until a pathing set is found with the lowest cost that does not have conflicts.

At the high level, CBS searches a tree called the constraint tree (CT). A CT is a binary tree. Each node N in the CT consists of a set of constraints, a solution, and the total cost. A node N in the CT is a goal node when its solution is valid, i.e., the set of paths for all agents has no conflicts.[Sharon et al. 2015a].

In this implementation of the CBS algorithm the only conflicts that were considered prohibitive were the Vertex, Edge and Swapping conflicts.

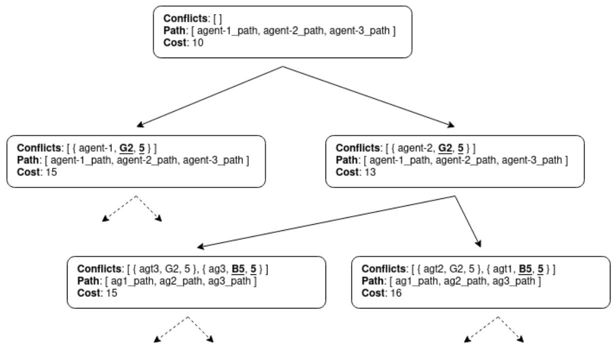
Conflicts: [ ]
Path: [agent-1_path, agent-2_path, agent-3_path]
Cost: 10
Conflicts: [{ agent-1, G2, 5 }]
Path: [agent-1_path, agent-2_path, agent-3_path]
Cost: 15
Conflicts: [{ agent-2, G2, 5 }]
Path: [agent-1_path, agent-2_path, agent-3_path]
Cost: 13
Conflicts: [{ agt3, G2, 5 }, { ag3, B5, 5 }]
Path: [ag1_path, ag2_path, ag3_path]
Cost: 15
Conflicts: [{ agt2, G2, 5 }, { agt1, B5, 5 }]
Path: [ag1_path, ag2_path, ag3_path]
Cost: 16

Fig. 9. A CBS tree representation.

7. DIFFERENCES BETWEEN ALGORITHMS

Both Naive Conflict Avoidance and Conflict Based Search are Multi-Agent Pathfinding algorithms, however, they have many differences between them.

One similarity between them is the fact that both uses the $A*$ algorithm in order to find the best path as possible for each of the agents in the environment. However, the Naive Conflict Avoidance algorithm can only deal with cases where the agents do not need to cooperate with other agents.

The most common problems that the Naive Conflict Avoidance cannot find a solution are the ones where the agents only have a narrow corridor and you can only fit one inside it, this corridor also have spaces where the agents can go to let the others go through the corridor. That case needs cooperation between the agents, where some of them will have to go out their way and wait in these spaces in order for the others to go through

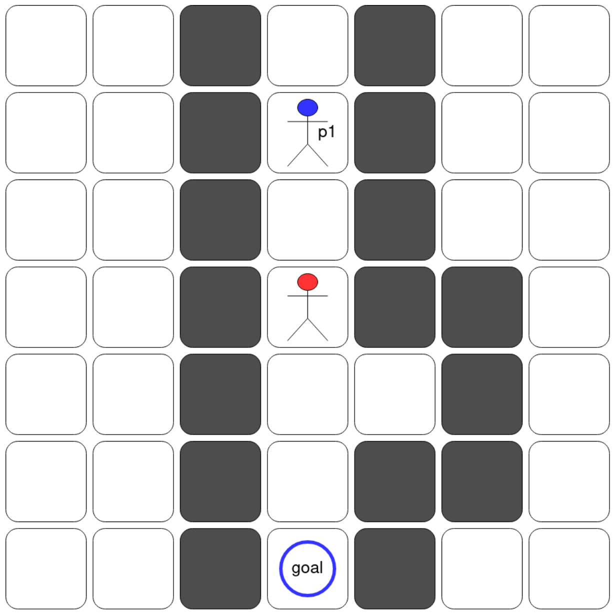

Fig. 10. Example of a narrow corridor environment, where an agent completely blocks that path of another one.

A problem that was encountered while using the *CBS* to find a solution was the time to process the data and find the best alternative. If the problem involved too many agents, and the solution would have to rely on a lot of cooperation between them, the algorithm would take a lot of time and memory to solve it. As Figure 11 shows an example of a problem where it took hours and 10 gigabytes of RAM and *CBS* could not even find the solution, because the algorithm was stopped. In the other hand, the Naive Collision Avoidance ran all the times almost instantly.

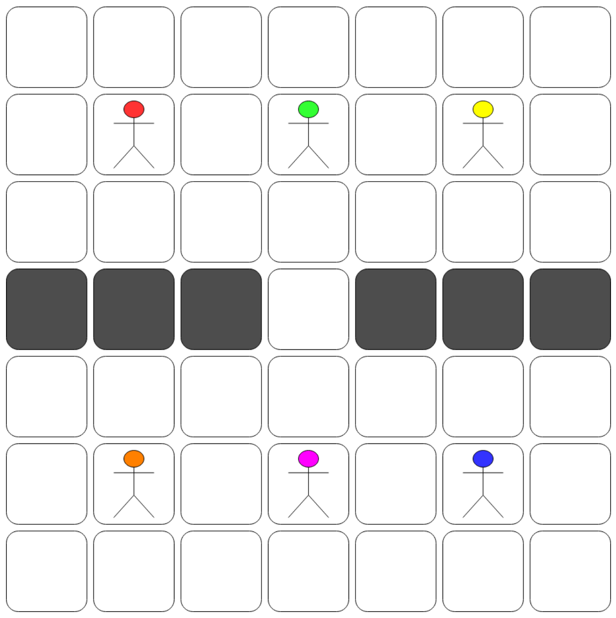

Fig. 11. Example of a bottle neck environment where the agents must cross the wall to their goals on the other side.

Thus, in complex cases where cooperation is not needed, the Naive algorithm can be much faster than the *CBS* algorithm and also reliable.

8. OUR PROPOSED SIMULATOR

With this article, we demonstrate a *MAPF* simulator that enables the creation of a **discrete**, **deterministic** and **static** environment. Two collision avoidance will be implemented, one that will consider the environment as **not accessible** and the agents as decentralized network, the other will be a centralized solution and will consider the environment as **fully accessible**. The first decentralized solution was proposed by [Savkin and Huang 2019] and was not given a name, so this article will call it **Naive Collision Avoidance**. The centralized solution is a well studied algorithm called **Conflict-Based Search**. The user will be able to choose between either algorithm before the start of the simulation.

The system will be implement in Javascript [jav] and it will run on the browser with the help of a Python based back-end. The graphical user interface will use the mouse to shape the environment, and place the agents and their goals. The framework used to create the graphical user interface for the browser will be React [rea].

8.1 The Browser Application

The browser application consisted in a React application[rea], which contained the *Naive Collision* algorithm in a separate file. This file consisted of a couple of functions that would process the information about the simulation and return the correct paths to each agent to follow in order to fulfill their goals and follow the conventions proposed by [Savkin and Huang 2019].

The application could also create a HTTP request to the web server that we created, which also contained the *CBS* algorithm to process the information about the simulation and return the correct paths to each agent to follow in order to fulfill their goals as the response of that request.

8.1.1 *React.* React is a declarative web development framework that facilitates the creation of responsive and dynamic applications for the browser[Banks and Porcello 2017]. Building on top of Node Framework for package management and code infrastructure, it uses Javascript. Our browser application was separated into different React components and constructed on a single *App.js* that is then linked to the *index.html* file.

Using React enabled was to created a game-like interface for the simulator where the user can create obstacles and agents by click on the tiles of the board, the interface responsively updates itself to show the current state if the board. Using *React* to create our front-end also made it easier to program our implementation of **Naive Collision Avoidance** in Javascript and use it natively, without any translation layer, in the back-end processes of the app.

8.2 The Python server

The Python server was built from an already existing application, developed by Ashwin Bose [Bose 2021], which implemented several Multi-Agent Pathfinding methods, centralized or decentralized. We choose to implement the server with the *CBS* implementation.

In order to implement the *CBS* server, we used a Python class called *BaseHTTPRequestHandler* from the *http.server* module, this class has many methods to map the URLs to capture the requests based on the URL and the HTTP method of the request and send them to a specific function matching the specified URL and thus, returning a response.

The server only had one route, which receives a request containing the settings from the Browser Application, like the agents starting point, goal, the environment size and obstacles, then the settings are processed using the Ashwin Bose *CBS* functions and then the server sends the data processed with all the correct paths for each agent as a response for the request.

8.3 Simulation

It is important to note that the system described in this paper is a high-level abstraction of a multi-agent system, as the entire simulator was written for the browser and thus runs on a single thread. In this sense it's not asynchronous nor concurrent, however it does simulate the behavior of a concurrent system. However, to simulate an asynchronous system, it would be necessary to randomize the order the agents take their movements.

9. EXPERIMENTS

To analyze the Naive Conflict Avoidance algorithm, we created a set of test environments to perform a comparative analysis with the CBS algorithm. In our experiments, we created an environment grid of size 10x10, then we separated our experiments by the number of agents. The experiments were divided into 2, 4, and 8 agents in the environment. There is no agreed set of metrics with which to analyze the performance of *MAPF* algorithms, but two popular ones are *makespan* and *flowtime*. At end of the simulation, we took these 2 metrics to compare the performance of both algorithms [Stern et al. 2019a].

Makespan: the number of time steps necessary to make all agents complete their tasks.

Flowtime: the sum of the steps necessary for each agent to complete their goal. This measure is obtained by adding the time steps used by each agent to reach their target location.

This set of environments was chosen by performing a qualitative analysis during the construction of the *NCA* algorithm, trying to understand its fundamental weaknesses and problems. Several generally problematic environments were envisioned, such as bottleneck walls and narrow corridors that could require collaboration between agents to properly navigate them. These hard environments were used as stress tests for this comparative quantitative analysis.

9.1 2 agents experiments

The experiments with two agents were the least complex, as they only involved two agents.

The first experiment, as shown in Figure 12, was without any obstacles, where the agents are in the same column and their goals are behind the other agent.

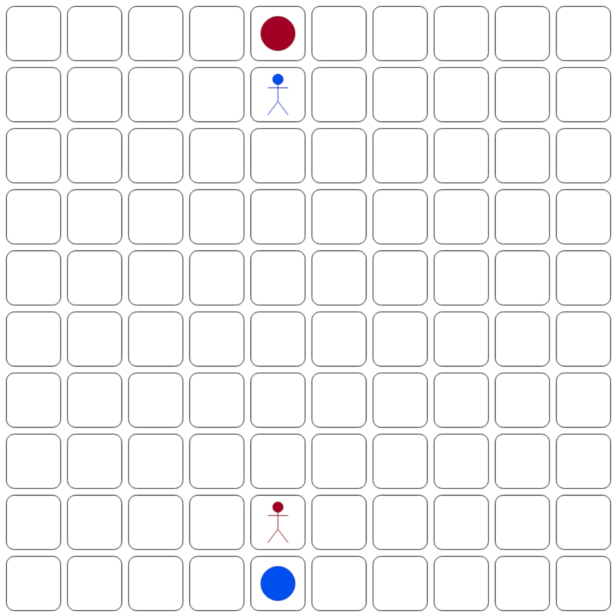

Fig. 12. First 2 agents experiment.

The second experiment was the bottleneck, as shown in Figure 13, where there was a wall dividing the environment, and only one space to cross it, which was the size of an agent. One agent was added in each side of the wall in a way where both of them were at the same distance of the wall.

The final experiment was the cooperation one, as shown in figure 14, where two agents were added in a narrow corridor, one in the start and one in the end, both of the agents goals were in the other side of the corridor. Only one agent could fit in the space of the

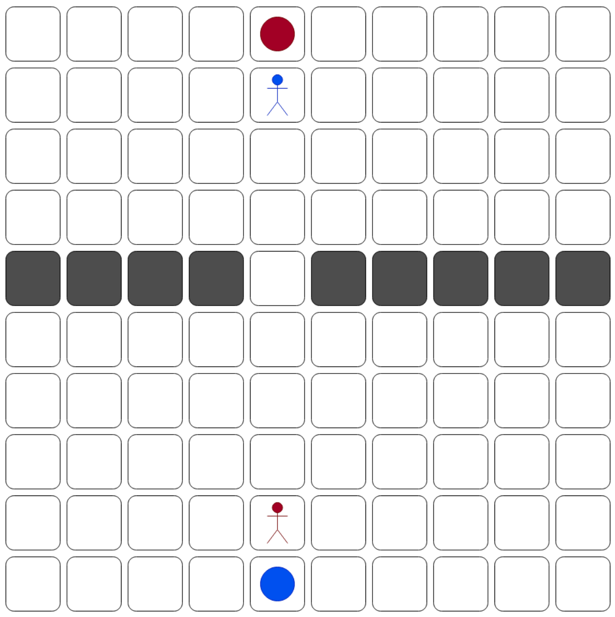
Fig. 13. Bottle neck experiment with 2 agents.

corridor, and in the middle of the corridor, where there was a space connected to it, where only one agent could fit, so, in order for every agent to fulfill their goal, they would have to cooperate, and one of them would have to go to the middle space and stay there while the other agent went to the corridor.

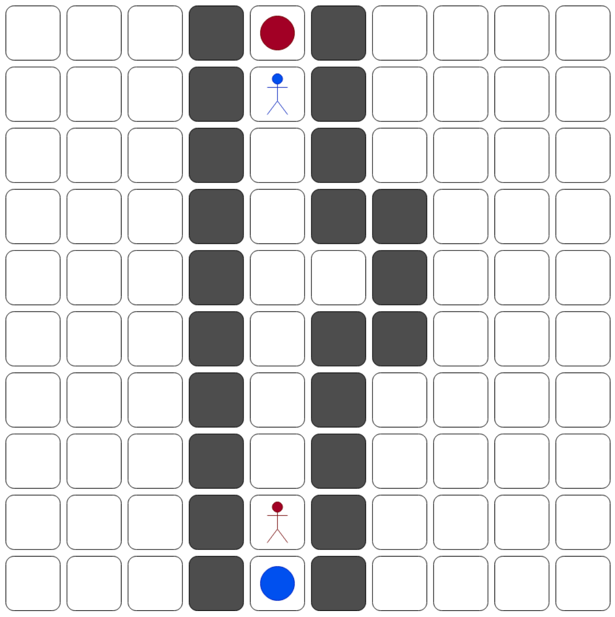
Fig. 14. Narrow corridor experiment with 2 agents.

9.2 4 agents experiments

The first experiment was similar to the first experiment of the two agent experiments, as shown in figure 15, there are two agents in the same row, both of their goals are behind the other agent; however, now there are two more agents in the row below the first, and their goal is behind the other agent in the same row.

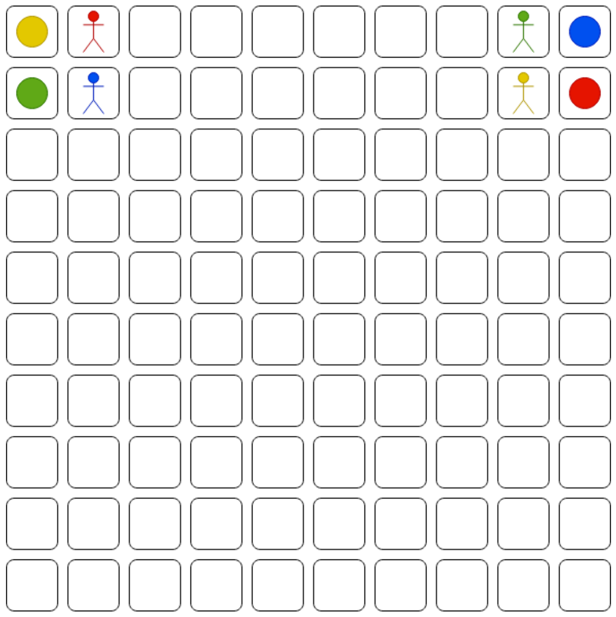
Fig. 15. Simple experiment with 4 agents facing each other.

The second experiment, as shown in Figure 16, was the bottle-neck, where there was a wall that divided the environment, and only one space to cross it, which was the size of an agent. Two agents were added in each side of the wall in a way where always one agent in the top section were at the same distance of the wall as another agent in the bottom section.

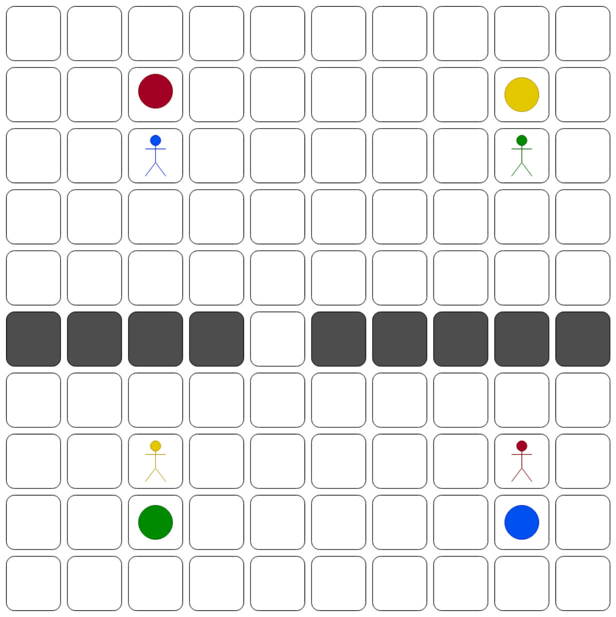
Fig. 16. Bottle neck experiment with 4 agents.

The final experiment, as shown in Figure 17, was one of cooperation, in which obstacles were added so that there were two corridors, both with space for a single agent, crossing each other, as a cross. At each of the four extremes of the cross, there is one agent, and the goal is to go to the other side of the cross. In the middle of the cross was a space that only fitted one agent in order to the agents to cooperate and be able to achieve their goals.

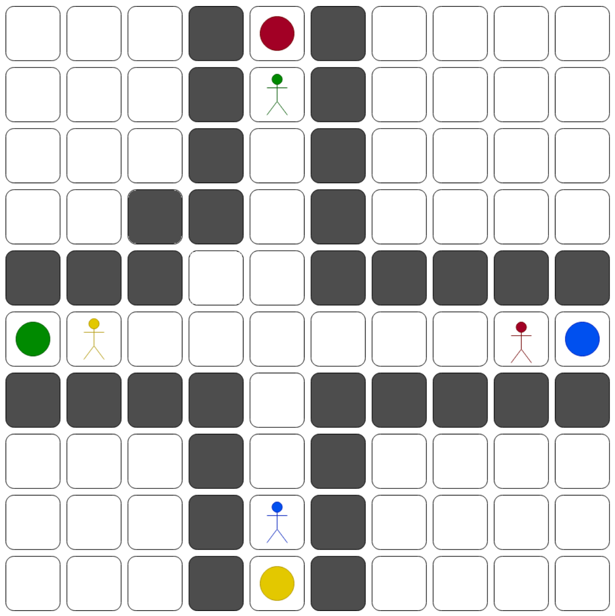
Fig. 17. Cross experiment with 4 agents.

9.3 8 agents experiments

The set of experiments that used eight agents was created to analyze how the algorithms would react to a dense agent environment. Due to the optimal nature of the CBS algorithm, the obstacle complexity of these environments is lower than that of the previous set of experiments. Agent dense environment are one of the predicted fundamental problem of the *NCA* algorithm.

The first experiment, as shown in Figure 18, was four lines of four agents facing each other. The agents must reach the goals that are behind the other line of agents.

The second and final experiment, as shown in Figure 19m is similar to the first one, but there is a wall between the two lines of agents. Agents must negotiate with other agents and avoid the wall between them and their goals.

10. RESULTS

The four metrics collected for each algorithm on each environment setting is the makespan, the flow time, the time to process the answer and if the algorithm was able to process the answer. Additionally there was a constraints on how long the CBS could take to find a solution. Some environment settings proved too complex for the CBS algorithm to find a solution in a reasonable time. Another constraint possible result was that the *NCA* could not successfully navigate the environment.

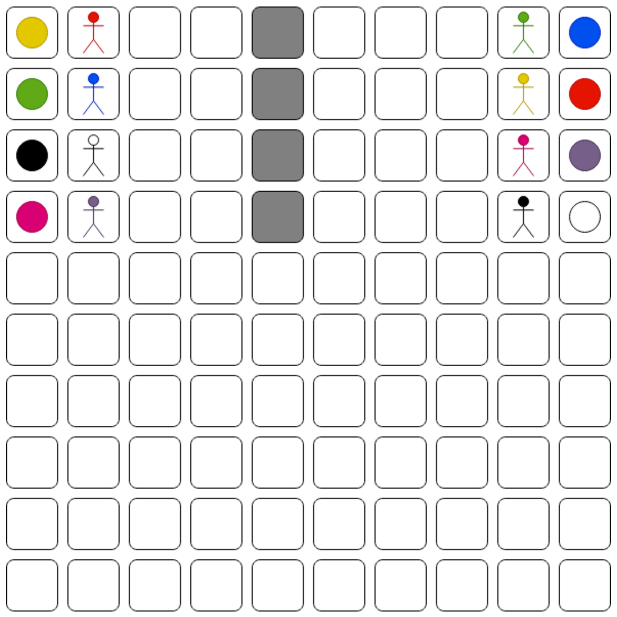
Fig. 18. Experiment with 8 agents facing each other with a wall an obstacle.

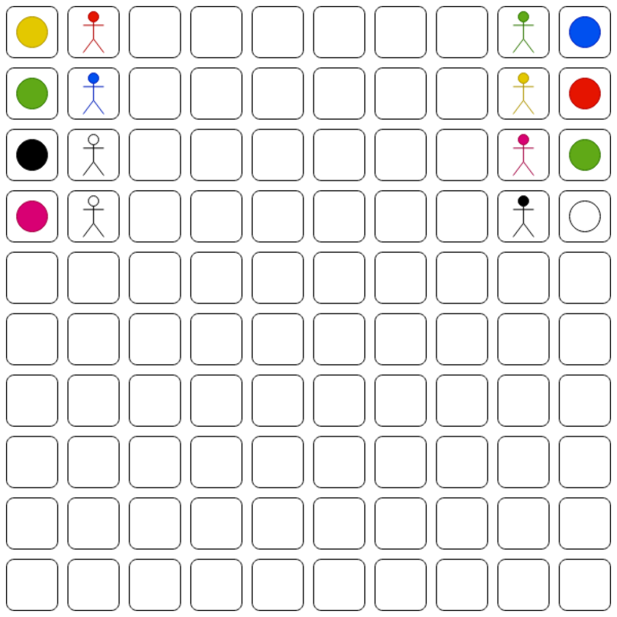
Fig. 19. Experiment with 8 agents facing each

10.1 2 agents results

For the experiments with two agents, four environment settings were used. It is important to add that, in all experiments with two agents, both algorithms processed the solution instantly, so this metric was the same for both algorithms.

The first experiment showed that both algorithms were able to find a solution and the both of them had the same makespan, flow time and time to process, which was 0 seconds.

In the second experiment, only *CBS* was able to find a solution, because in this type of experiment, both agents need to cooperate with each other. The time for *CBS* to process this experiment was about 0 seconds.

The final experiment was similar to the second one, where only *CBS* was able to find a solution, since, as in the second experiment, the agents needed to cooperate in order to find the solution without colliding. The time for *CBS* to process this experiment was about 0 seconds.

10.2 4 agents results

For the experiments with four agents, three experiments were devised.

In the first experiment the makespan of Naive Conflict Avoidance was 26, while *CBS* was 13, which is a great improvement.

In the second experiment, only *CBS* was able to find a solution, since it was a cooperation experiment. The *CBS* time to process the solution was 114 seconds.

In the third experiment, none of the algorithms were able to find a solution. The *CBS* time to process exceeded five minutes, and then the algorithm was stopped.

10.3 8 agents results

The experiments with 8 agents were the hardest for *CBS*. If the environment contained a bottle neck, or too many obstacles, the *CBS* algorithm would run for hours until it found the solution, or the algorithm would consume the maximum RAM the experiments permitted, which was 10 gigabytes.

The Naive Conflict Avoidance algorithm performed well in the performance matter, but as was said before, when cooperation is needed in order to achieve each agent goal, the test would fail and the agents would collide. In the experiments with minimal to none obstacles, the Naive would always perform correctly, but comparing to *CBS*, the simulation would need have a larger makespan in order to finish.

In the first experiment, both algorithms found a solution, the Naiver Conflict Avoidance processed the solution instantly, and the *CBS* took eleven seconds to process the solution. However, the makespan of the Naive Conflict Avoidance was 49, when the *CBS* was 13 ticks, which is a great improvement.

In the second experiment, none of them were able to find a solution, but *CBS* ran for five minutes and then was stopped.

11. DISCUSSION

The experiments showed a clear distinction in the strengths and weaknesses of each algorithm, specially on how each is capable of dealing with complex scenarios and environments. The Naive Conflict Avoidance algorithm, as it was implemented for this article, lacks the ability to recover from a situation where it cannot find a path to its goal. This becomes a problem during bottle necks where other agents can block the only way to reach the goals.

CBS optimal and complete property was also a problem on highly complex environments where finding such optimal pathing for all agents proved too complex. Combined with the fact that it is centralized and must pre-process all paths for all agents, this creates a situation where it may be unfeasible, without further modifications or optimizations, to use it on systems that require fast response for pathing, even if it is not optimal.

12. CONCLUSION

We have created an interactive simulator that enables the user to add agents and obstacles and then choose between two *MAPF* algorithms: Conflict-Based Search, implemented in Python by Ashwin Bose[Bose 2021], and Naive Conflict Avoidance, proposed by [Savkin and Huang 2019] and implemented by us. The simulator is a web application that runs in a browser and is created using the React framework. To understand the characteristics of our the Naive Conflict Avoidance algorithm a comparative analysis was performed using the Conflict-Based Search algorithm. Several environment settings were created to analyze the performance of each algorithm and try to understand the problems and limitations of each one. The fundamental differences in how each algorithm functions, one having a greedy approach and the other is centralized and optimal, appeared in the experiments where in some cases they were not able to find a solution.

13. FUTURE WORK

Firstly, the simulator could have the implementation of other centralized *MAPF* algorithms, such as *SIPP*.

Other improvements could be made as a matter of settings configuration, with a system to save the environment settings and the results. Pre-configured environment settings could also be provided to allow the use to quickly run some tests.

An analytics system could be implemented in order to improve the experiments, this system would inform all the information about the result of the simulation.

Finally, there are several improvements that could be made to the Naive Conflict Avoidance algorithm such as ensuring that even in cases where no paths can be found some measure can be taken so the agent can later progress towards its goal. Currently the only information the agent has about other agents is if they are near, they do not know their heading or if they are in the way of another agent.

REFERENCES

Monitoramento de queimadas — ibama.gov.br. http://www.ibama.gov.br/incendios-florestais/monitoramento-de-queimadas-em-imagens-de-satelites. (????). [Accessed 12-Sep-2022].

mq$_{o}$*pen – openamessagequeue.*. (????). Accessed: 2022-06-12.

(no title) — conservationdrones.org. https://conservationdrones.org/. (????). [Accessed 12-Sep-2022].

React – A JavaScript library for building user interfaces — reactjs.org. https://reactjs.org/. (????). [Accessed 11-Sep-2022].

The JavaScript language — javascript.info. https://javascript.info/js. (????). [Accessed 11-Sep-2022].

2020. What is artificial intelligence and how is it used? | News | European Parliament. (April 2020). https://www.europarl.europa.eu/news/en/headlines/society/20200827STO85804/what-is-artificial-intelligence-and-how-is-it-used

Jasmina Arifovic, Xuezhong He, and Lijian Wei. 2019. High Frequency Trading in FinTech age: AI with Speed. *Available at SSRN 2771153* (2019).

Alex Banks and Eve Porcello. 2017. *Learning React: functional web development with React and Redux.* " O'Reilly Media, Inc.".

Ashwin Bose. 2021. Multi-Agent path planning in Python. https://github.com/atb033/multi_agent_path_planning. (2021).

Namjun CHA, Hosoo CHO, Sangman LEE, and Junseok HWANG. 2019. Effect of AI Recommendation System on the Consumer Preference Structure in e-Commerce: Based on Two types of Preference. In *2019 21st International Conference on Advanced Communication Technology (ICACT)*. 77–80. DOI:http://dx.doi.org/10.23919/ICACT.2019.8701967

Najmeddine Dhieb, Hakim Ghazzai, Hichem Besbes, and Yehia Massoud. 2020. A Secure AI-Driven Architecture for Automated Insurance Systems: Fraud Detection and Risk Measurement. *IEEE Access* 8 (2020), 58546–58558. DOI:http://dx.doi.org/10.1109/ACCESS.2020.2983300

Fei Fang, Milind Tambe, Bistra Dilkina, and Andrew J Plumptre (Eds.). 2019. *Artificial Intelligence and Conservation*. Cambridge University Press, Cambridge, England.

Florence Ho, Rúben Geraldes, Artur Gonçalves, Bastien Rigault, Benjamin Sportich, Daisuke Kubo, Marc Cavazza, and Helmut Prendinger. 2022. Decentralized Multi-Agent Path Finding for UAV Traffic Management. *IEEE Transactions on Intelligent Transportation Systems* 23, 2 (2022), 997–1008. DOI:http://dx.doi.org/10.1109/TITS.2020.3019397

Pinxin Long, Tingxiang Fan, Xinyi Liao, Wenxi Liu, Hao Zhang, and Jia Pan. 2018. Towards Optimally Decentralized Multi-Robot Collision Avoidance via Deep Reinforcement Learning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*. 6252–6259. DOI:http://dx.doi.org/10.1109/ICRA.2018.8461113

Hang Ma and Sven Koenig. 2017. AI Buzzwords Explained: Multi-Agent Path Finding (MAPF). *AI Matters* 3, 3 (oct 2017), 15–19. DOI:http://dx.doi.org/10.1145/3137574.3137579

Hang Ma, Sven Koenig, Nora Ayanian, Liron Cohen, Wolfgang Hoenig, T. K. Satish Kumar, Tansel Uras, Hong Xu, Craig Tovey, and Guni Sharon. 2017. Overview: Generalizations of Multi-Agent Path Finding to Real-World Scenarios. (2017). DOI:http://dx.doi.org/10.48550/ARXIV.1702.05515

Spyros Makridakis. 2017. The forthcoming Artificial Intelligence (AI) revolution: Its impact on society and firms. *Futures* 90 (2017), 46–60. 0016-3287 DOI:http://dx.doi.org/https://doi.org/10.1016/j.futures.2017.03.006

Yadvinder Malhi, J. Timmons Roberts, Richard A. Betts, Timothy J. Killeen, Wenhong Li, and Carlos A. Nobre. 2008. Climate Change, Deforestation, and the Fate of the Amazon. *Science* 319, 5860 (2008), 169–172. DOI:http://dx.doi.org/10.1126/science.1146961

Toshiyuki Nakagaki, Hiroyasu Yamada, and Ágota Tóth. 2001. Path finding by tube morphogenesis in an amoeboid organism. *Biophysical Chemistry* 92, 1 (2001), 47–52. 0301-4622 DOI:http://dx.doi.org/https://doi.org/10.1016/S0301-4622(01)00179-X

Stuart Russell and Peter Norvig. 2010. *Artificial Intelligence: A Modern Approach* (3 ed.). Prentice Hall.

Stuart Russell and Peter Norvig. 2020. *Artificial intelligence* (4 ed.). Pearson, Upper Saddle River, NJ.

Chris Sandbrook. 2015. The social implications of using drones for biodiversity conservation. *Ambio* 44 Suppl 4 (November 2015), 636—647. 0044-7447 DOI:http://dx.doi.org/10.1007/s13280-015-0714-0

Andrey V. Savkin and Hailong Huang. 2019. A Method for Optimized Deployment of a Network of Surveillance Aerial Drones. *IEEE Systems Journal* 13, 4 (Dec. 2019), 4474–4477. DOI:http://dx.doi.org/10.1109/jsyst.2019.2910080

Shai Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. 2017. On a formal model of safe and scalable self-driving cars. *arXiv preprint arXiv:1708.06374* (2017).

Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. 2015a. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence* 219 (2015), 40–66. 0004-3702 DOI:http://dx.doi.org/https://doi.org/10.1016/j.artint.2014.11.006

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

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 Barták. 2019a. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. *CoRR* abs/1906.08291 (2019). http://arxiv.org/abs/1906.08291

Roni Stern, Nathan R Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne T Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, TK Satish Kumar, and others. 2019b. Multi-agent pathfinding: Definitions, variants, and benchmarks. In *Twelfth Annual Symposium on Combinatorial Search*.

Paolo Tripicchio, Massimo Satler, Giacomo Dabisias, Emanuele Ruffaldi, and Carlo Alberto Avizzano. 2015. Towards Smart Farming and Sustainable Agriculture with Drones. In *2015 International Conference on Intelligent Environments*. 140–143. DOI:http://dx.doi.org/10.1109/IE.2015.29

Pelin Vardarlier and Cem Zafer. 2020. *Use of Artificial Intelligence as Business Strategy in Recruitment Process and Social Perspective*. Springer International Publishing, Cham, 355–373. 978-3-030-29739-8 DOI:http://dx.doi.org/10.1007/978-3-030-29739-8_17

Ko-Hsin Wang, Philip Kilby, and Jussi Rintanen. 2009. Bridging the Gap Between Centralised and Decentralised Multi-Agent Pathfinding. (01 2009).

Mike Wooldridge. 2002. *An Introduction to Multi-agent Systems*. John Wiley & Sons, Chichester, England.

Tim Zeitz. 2023. NP-hardness of shortest path problems in networks with non-FIFO time-dependent travel times. *Inform. Process. Lett.* 179 (2023), 106287. 0020-0190 DOI:http://dx.doi.org/https://doi.org/10.1016/j.ipl.2022.106287