Alessandro Lombardi
Fiorenzo Parascandolo

SAT and CP-based approaches for Multi-Agent Path Finding

Course on Combinatorial Decision Making and Optimization

University of Bologna
Academic Year 2019–2020

Abstract

Multi-Agent Path Finding (MAPF) is a problem with practical implications in several fields: from robotics and self-driving cars to transportation and logistics. The task is to find non-conflicting paths for a set of agents given their starting positions and destinations, usually minimizing a cost function. There are many variations on the classical problem and many approaches have been proposed. In this work we will focus on SAT and CP-based approaches following the paper of R. Barták, J. Švancara and M. Vlk, "A Scheduling-Based Approach to Multi-Agent Path Finding with Weighted and Capacitated Arcs", published in Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems [3]. As the authors suggest, this type of problem lends itself particularly well to be formalized using a compact set of constraints and we found interesting to developed as a project for our Combinatorial Decision Making and Optimization course.

Introduction

A formal and brief overview

Agents move in a grid world under the assumptions of uniform duration of actions given the discretization of time in time steps. The grid can be easily represented as a **directed graph** so that each agent is in a node and can move through the outgoing arcs and it is not possible for two agents to be at the same node at the same time. Formally, an instance of MAPF can be defined as ordered 4-tuple $(G, A, origin, destination)$ where $G = (V, E)$ is a directed graph and A is a set of agents. Functional symbols $origin$: A→V and $destination$: A→V describe respectively origin and destination nodes of an agent. For each agent $a \in A$, we denote by $origin(a) \in V$ its starting node and by $destination(a) \in V$ its destination node. The word **conflict** is used to denote any illegal situation on the plan and it is generally in the center of many MAPF Solvers and techniques. MAPF solution is called **valid** if and only if there is no conflict between any two single-agent plans. In the current implementation the main assumptions are:

- Two agents cannot be found at the same node at the same time, **node collision**.
- Two agents cannot exchange their positions, **edge collision**.
- The moves of the agents are **discrete** and **synchronous**.

The task is to return a set of actions for each agent, that respects the mentioned constraints and moves each agent to its goal minimizing a cumulative cost function. The literature presents two well-known cost functions:

- **Makespan**. It is the total time until the last agent reaches its destination (the maximum of the individual costs). The solutions proposed in this work are **makespan optimal**.
- **Sum-of-costs**. It is the summation over all agents of the steps required to reach their destinations. It represents an upper bound of the makespan and could be seen as the sum of individual costs.

In conclusion, the literature proposes two main families of MAPF solvers:

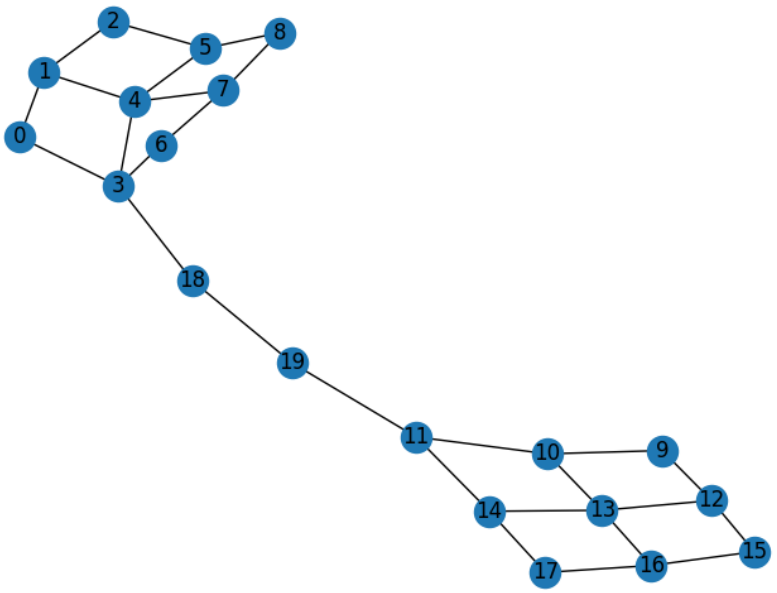
- **Reduction-based solvers**. This type of approach is based on solvers that reduce the problem to a known one, for example SAT or Integer linear programming, and it is particularly efficient in the case of unit cost per move. This work follows this approach.
- **Search-based solvers**. In this case the problem can be formalized as a search in a global search space, for example some variants of A*.

Our representation

Our implementation uses two Python APIs: **Z3Py** and **Docplex**. For this reason, we are able to use a single encoding of the input data. As we are interested in the case of unitary arcs we will represent our graph as a list of sets of integers, or in details, as a list of neighbors. The vertices are named by integer values, the same used as indexes in the

graph representation. Consequently, each element of the list is a set containing the indexes of the neighboring nodes. Agents are similarly named by indexes of a list of 2-tuples. Each tuple contains origin position and destination of a specific agent. To easily manipulate and deploy environments we use the powerful **NetworkX** library.

```
ENVIRONMENT:
0: {0, 1, 3}
1: {0, 1, 2, 4}
2: {1, 2, 5}
3: {0, 3, 4, 6, 18}
4: {1, 3, 4, 5, 7}
5: {8, 2, 4, 5}
6: {3, 6, 7}
7: {8, 4, 6, 7}
8: {8, 5, 7}
9: {9, 10, 12}
10: {9, 10, 11, 13}
11: {19, 10, 11, 14}
12: {9, 12, 13, 15}
13: {10, 12, 13, 14, 16}
14: {17, 11, 13, 14}
15: {16, 12, 15}
16: {16, 17, 13, 15}
17: {16, 17, 14}
18: {19, 18, 3}
19: {11, 18, 19}
AGENTS:
[(14, 19), (17, 1)]
```



1 SMT-based approach

In the original paper the authors introduced a SAT-based approach while this implementation replaces it with a SMT-based one. Satisfiability Modulo Theories (SMT) solvers takes systems in arbitrary format (first-order logic), while SAT solvers are limited to Boolean equations and variables, nevertheless they still mantain the speed and automation of today's Boolean engines. The authors of the paper developed their solution using the Picat language, we use the Z3 Solver's Python API: Z3Py.

> *"Satisfiability Modulo Theories (SMT) problem is a decision problem for logical formulas with respect to combinations of background theories such as arithmetic, bit-vectors, arrays, and uninterpreted functions. Z3 is an efficient SMT solver with specialized algorithms for solving background theories."[6]*

The most difficult part is certainly represented by the constraint modeling and their adaptation for the Z3 Theorem Prover. From the original work, we define the following variables as **Z3 Functions**, or more formally **Formulas**, in the file solvers/model_smt:

- $\forall x \in V, \forall a \in A, t \in 0, ..., T : At(x, a, t)$ meaning that agent a is at node x at time step t.
- $\forall (x, y) \in E, \forall a \in A, t \in 0, ..., T-1 : Pass(x, y, a, t)$ meaning that agent a goes through arc (x, y) at time step t.

```
at_ = Function('at', IntSort(), IntSort(), IntSort(), BoolSort())
pass_ = Function('pass', IntSort(), IntSort(), IntSort(), IntSort(),
BoolSort())
```

Variables types in Z3 are called also **sorts**. Standard well-known sorts are *Bool*, *Int* and *Real* but programmers may also define their own types. The definition of Z3 Functions specifies the types of the arguments and as the type of the returned value, in this case both *at_* and *pass_* are Boolean Functions and can be assigned to either true or false. We define as Z3 Integer Functions the following constant representations:

- $\forall a \in A : origin(a)$ which returns the initial position of the agent a.
- $\forall a \in A : dest(a)$ which returns the goal position of the agent a.

```
orig_ = Function('orig', IntSort(), IntSort())
dest_ = Function('dest', IntSort(), IntSort())
```

The model is completed introducing the constraints on the variables:

All agents must be in their initial position at time $t = 0$:

$$\forall a \in A : At(origin(a), a, 0) = 1 \tag{1.1}$$

All agents must be in their goal position at time $t = T$:

$$\forall a \in A : At(dest(a), a, T) = 1 \tag{1.2}$$

All agents can be in one node in every moment:

$$\forall a \in A, \forall t \in \{0, \ldots, T\} : \sum_{x \in V} At(x, a, t) \leq 1 \tag{1.3}$$

Every vertex must be occupied at most by an agent in every moment:

$$\forall x \in V, \forall t \in \{0, \ldots, T\} : \sum_{a \in A} At(x, a, t) \leq 1 \tag{1.4}$$

If an agent is in a node it needs to leave by one of the outgoing arcs:

$$\begin{aligned} \forall x \in V, \forall a \in A, \forall t \in 0, \ldots, T-1 : At(x, a, t) \\ \Rightarrow \sum_{(x,y) \in E} Pass(x, y, a, t) = 1 \end{aligned} \tag{1.5}$$

If an agent is using an arc, it must arrive at the corresponding node in the next time step

$$\begin{aligned} \forall (x, y) \in E, \forall a \in A, \forall t \in \{0, \ldots, T-1\} : Pass(x, y, a, t) \\ \Rightarrow At(y, a, t+1) \end{aligned} \tag{1.6}$$

Two agents cannot exchange their positions:

$$\begin{aligned} \forall (x, y) \in E, x \neq y, \forall t \in \{0, \ldots, T-1\} : \\ \sum_{a \in A} Pass(x, y, a, t) + Pass(y, x, a, t) \leq 1 \end{aligned} \tag{1.7}$$

The correct movements inside the graph are guided by the constraints 1.5-1.7, nevertheless it is not necessary to specify that each agent must make one and only on movement at a given instant t because the spurious movements introduced by the constraint 1.6 will never be performed thanks to 1.5. In particular, a valid solution may contain some true *pass_* that do not actually belong to the plan because they are simply the result of some unassignments. For example in a path-like graph of length 3 where an agent must move from one extreme to the node at the center, there might be a true *pass_* from the other extreme to the center because both 1.5 and 1.6 are respected, but thanks to 1.5 the agent will be only *at_* the two correct nodes.

The Z3 implementation is divided in three parts. The details of the first one have been described before during variable and constant definitions. The second one defines intermediate variables useful to the definitions of the constraints in the third and last section. In particular, the second part defines the summations present in the formal formulations of the original work, making easier to define the involved constraints. For example the following code uses the Z3 function *If* to map true variables to 1 and false variables to 0 cycling all combinations and summing over a certain "dimension" (for simplicity it is usually the inner one):

```
sum3_tmp = [[[If(at_(vertex, agent, time), 1, 0)
                        for vertex in range(edges_len)]
                      for time in range(makespan + 1)]
                    for agent in range(agents_len)]

sum3 = [[sum(sum3_tmp[agent][time])
         for time in range(makespan + 1)]
        for agent in range(agents_len)]

sum3 = [element for sublist in sum3
            for element in sublist]
```

The resulting list is finally flatten and then used in the constraint 1.3 in the following way:

```
s.add([vertex <= 1 for vertex in sum3])
```

The other constraints which do not use summations are solved using the Z3 function *ForAll* like the 1.6

```
s.add([ForAll([x, y, a, t],
                  Implies(
                      And(x >= 0, x < edges_len, y >= 0, y < edges_len,
                          t >= 0, t <= makespan - 1, arc_(x, y),
                          pass_(x, y, a, t)),
                      at_(y, a, t + 1)
                  ))])
```

Universal and existential quantifiers bind variables to the scope of the quantified formula. In our case the variables x, y, a and t are Z3 Integer variables, which are bound independently to each *ForAll* expression. Because a is used in the other *ForAll* style constraints it has been similarly bound to a specific interval of integer values outside the shown piece of code. *arc_* is a Boolean Z3 Function used to represent the presence of an arc between a couple of nodes.

To find an optimal makespan the makespan is increased until a satisfiable formula is generated.

2 CP-based approach

We now follow the paper showing how to formalize a MAPF problem as a scheduling problem. Similarly to the authors we implement the model using the Python API of IBM ILOG CPLEX Studio: the library Docplex. For this reason, the formal description uses some functions belonging to the grammar of the aforementioned IBM solver. IBM ILOG CPLEX allows the definition of activities with interval variables. The presence of an agent in a node can be seen as the activity of occupying that node for some amount of time and can be formulated in CPLEX as an interval variables that begins and finishes in a certain moment, represented by the predicates *StartOf* and *EndOf*. The difference between the end time and the start time of the activity can be set using the predicate *LengthOf*. A great advantage of using interval variables is represented by their optionality, to check whether a certain activity is present or absent in the resulting schedule it is possible to use the predicate *PresenceOf*. In order to allow each agent to be able to visit the same node several times the authors developed a multi-layer model based on the copy of the original graph with some additional arcs that allow the transition between couples of graphs. Let l be the number of layers, namely the maximum number of times each agent can visit the same node. In the original paper the layers belong to the set $1, \ldots, l$, we decided to choose a zero-indexed approach such that our set of layers is $0, \ldots, l-1$. $\forall a \in A, \forall x \in V, \forall k \in 0, ..., l-1$ we considered the following optional activities:

- $N[x, a, k]$ corresponds to the time of an agent a spent at node x when the activity starts at layer k. The start and end of this activity can span all over the available time, the length must be found.
- $N^{in}[x, a, k]$ describes the time spent in the incoming arc at layer k.
- $N^{out}[x, a, k]$ describes the time spent in the outgoing arc at layer k.
- $A[x, y, a, k]$ where $k \in 0, ..., l-1$ which corresponds to transiting an agent a from a node x to the node y at layer k. The start and end of this activity can span all over the available time, the length is set to time required to traverse the arc.
- $A[x, x, a, k]$ where $k \in 0, ..., l-2$ which corresponds to transiting an agent a from a node x to the node x at layer k. The start and end of this activity can span all over the available time, the length is set to zero because arcs between layers are not real.

We declare the described interval variables inside recursive Python data structures to easy access them using their parametric arguments as indexes. For example N is built using recursive lists representing in the following order the vertex, agent and layer dimensions, follows that the Pythonic *N[x][a][k]* is equivalent to the formal $N[x, a, k]$. On the other hand, we handle differently A, because using a list of list is correct until the wanted operations are limited on cycling for each vertex its neighbors and access all of them using the index, which in this case is not the real identifier of the neighbor. For example the nodes 5 and 6 are neighbors and *A[4][0][a][k]* represents a movement between these two nodes, there is no way to refer the index 0 to the node 6, it could be for example another neighbor of 5, like the node 3. In particular the constraint 2.7 requires to use the neighbors of each node as origins for the A interval variables. To overcome this problem, we use a dictionary to explicitly access for an origin in A (still represented as an index of the outermost list) its neighbors and their identifiers (keys of the dictionary). Recalling the previous example *A[4][6][a][k]* surely represents $A[5, 6, a, k]$. Probably substituting also the outer list with

a dictionary would lead to a more clear model, but we decide to keep it for legacy with the other interval variables. In order to simplify the implementation of some constraints we created two different interval variables to represent the following transitions $A[x, y, a, k]$ and $A[x, x, a, k]$. Transitions from and to the same node are used by the solver to increase the layer (2.11) and to allow multiple visits to the same node by the same agent. We represent these transitions defining the interval variables *A_equal*, which are indexed using only three values as the destination of the transition is always the origin itself and would be redundant. Differently from A interval variables which may have different lengths depending on the arc, *A_equal* variables have always length 0. The following pieces of code are from the file solvers/model_cp.py.

```
N = [[[interval_var(start=(0, upper_bound),
                    end=(0, upper_bound),
                    name="N_%s_%s_%s" % (vertex, agent, layer),
                    optional=True)
       for layer in range(num_layers)]
      for agent in range(agents_len)]
     for vertex in range(edges_len)]

A = [dict((neighbor, [[interval_var(start=(0, upper_bound),
                                    end=(0, upper_bound),
                                    length=1,
                                    name="A_%s_%s_%s_%s" %
                                     (vertex, neighbor, agent, layer),
                                    optional=True)
                       for layer in range(num_layers)]
                      for agent in range(agents_len)])
           for neighbor in edges[vertex] if vertex != neighbor)
     for vertex in range(edges_len)]


A_equal = [[[interval_var(start=(0, upper_bound),
                          end=(0, upper_bound),
                          length=0,
                          name="Ae_%s_%s_%s_%s" %
                           (vertex, vertex, agent, layer),
                          optional=True)
             for layer in range(num_layers - 1)]
            for agent in range(agents_len)]
           for vertex in range(edges_len)]
```

Also the variable indicating the final makespan must be declared

```
makespan = integer_var(0, upper_bound, name="MKSP")
```

We now introduce the constraints.

$$PresenceOf(N[orig(a), a, 0]) = 1 \tag{2.1}$$

$$PresenceOf(N[dest(a), a, l-1]) = 1 \tag{2.2}$$

$$PresenceOf(N^{in}[orig(a), a, 0]) = 0 \tag{2.3}$$

$$PresenceOf(N^{out}[dest(a), a, l-1]) = 0 \tag{2.4}$$

As already mentioned the formal representation is very similar to the implementation, for example the following constraints are almost identical to the formal definitions.

```
[model.add(presence_of(N[pair[0]][a][0]) == 1)
 for a, pair in enumerate(agents)]
[model.add(presence_of(N[pair[1]][a][num_layers - 1]) == 1)
 for a, pair in enumerate(agents)]
[model.add(presence_of(Nin[pair[0]][a][0]) == 0)
 for a, pair in enumerate(agents)]
[model.add(presence_of(Nout[pair[1]][a][num_layers - 1]) == 0)
 for a, pair in enumerate(agents)]
```

For this reason we decide to not comment all the constraints. Constraints from 2.1 to 2.4 are used to constraint the optionality of some specific interval variables like those representing the presence of an agent at the beginning in its starting position or at the end in its final position. The constraints 2.5 and 2.6 bonds the variables *Nin* or *Nout* with N to represent that in a specific layer an agent which traverses the incoming or outgoing arcs of a vertex (excluding respectively origin and destination), has to occupy the vertex. Timings between *Nin* or *Nout* and N are dealt in the constraints 2.15 and 2.16.

$$\begin{aligned} &\forall x \in V, \forall k \in \{0, \dots, l-1\}, x \neq orig(a) \vee k \neq 0 : \\ &PresenceOf(N[x,a,k]) \iff PresenceOf(N^{in}[x,a,k]) \end{aligned} \tag{2.5}$$

$$\begin{aligned} &\forall x \in V, \forall k \in \{0, \dots, l-1\}, x \neq dest(a) \vee k \neq l-1 : \\ &PresenceOf(N[x,a,k]) \iff PresenceOf(N^{out}[x,a,k]) \end{aligned} \tag{2.6}$$

$$\begin{aligned} &\forall x \in V, \forall k \in \{0, \dots, l-1\} : \\ &Alternative(N^{in}[x,a,k], \{A[x,x,a,k-1]\} \cup \bigcup_{(y,x) \in E} A[y,x,a,k]) \end{aligned} \tag{2.7}$$

An interesting and already mentioned constraint is the 2.7. Similarly to 2.8 it defines the correct cardinality in which interval variables *Nin*, A and *A_equal* must be bound in the solution. The *alternative* function creates an alternative constraint between the passed interval variable and the set of interval variables passed as the second argument. If the interval variable is present, then one and only one of the intervals in the set will be selected by the alternative constraint to be present. In particular, for each activity in the arcs incoming on a specific vertex, for an agent on a given layer there must be at most one traverse: from the same vertex between different layers (only if the given layer is not the first), or from a neighbor vertex occurring in the same layer. To avoid indexes out of bound exceptions a control must be performed on the layer resulting on the possibility of adding a different constraint to the model.

```
[model.add(
alternative(Nin[vertex][agent][layer],
            [A_equal[vertex][agent][layer - 1]] +
            [A[neighbor][vertex][agent][layer]
             for neighbor in edges[vertex].difference({vertex})]
            if layer > 0 else
            [A[neighbor][vertex][agent][layer]
```

```
            for neighbor in edges[vertex].difference({vertex})]
          ))
for layer in range(num_layers)
for agent in range(agents_len)
for vertex in range(edges_len)]
```

$$\forall x \in V, \forall k \in \{0, \ldots, l-1\} : \\ Alternative(N^{out}[x,a,k], \{A[x,x,a,k]\} \cup \bigcup_{(x,y)\in E} A[x,y,a,k]) \tag{2.8}$$

$$\forall (x,y) \in E, \forall k \in \{0, \ldots, l-1\} : \\ PresenceOf(A[x,y,a,k]) \Rightarrow PresenceOf(N^{in}[y,a,k]) \tag{2.9}$$

$$\forall (x,y) \in E, \forall k \in \{0, \ldots, l-1\} : \\ PresenceOf(A[x,y,a,k]) \Rightarrow PresenceOf(N^{out}[x,a,k]) \tag{2.10}$$

$$\forall x \in V, \forall k \in \{0, \ldots, l-2\} : \\ PresenceOf(A[x,x,a,k]) \Rightarrow PresenceOf(N^{in}[x,a,k+1]) \tag{2.11}$$

$$\forall x \in V, \forall k \in \{0, \ldots, l-2\} : \\ PresenceOf(A[x,x,a,k]) \Rightarrow PresenceOf(N^{out}[x,a,k]) \tag{2.12}$$

Links between *A*/*A_equal* and *Nin*/*Nout* interval variables are specified in the constraints 2.9-2.12. In the following implementation is easy to observe the difference between *A* and *A_equal* and how is useful to divide into two interval variables the activity representing the transitions.

```
# 9
[model.add(if_then(presence_of(A[vertex][neighbor][agent][layer]),
                   presence_of(Nin[neighbor][agent][layer])))
 for vertex in range(edges_len)
 for neighbor in edges[vertex].difference({vertex})
 for agent in range(agents_len)
 for layer in range(num_layers)]

# 11
[model.add(if_then(presence_of(A_equal[vertex][agent][layer]),
                   presence_of(Nin[vertex][agent][layer + 1])))
 for vertex in range(edges_len)
 for agent in range(agents_len)
 for layer in range(num_layers - 1)]
```

$$StartOf(N[orig(a), a, 0]) = 0 \tag{2.13}$$

$$EndOf(N[dest(a), a, l-1]) = MKSP \tag{2.14}$$

$$\begin{aligned} \forall x \in V, \forall k \in \{0, \ldots, l-1\}, x \neq orig(a) \vee k \neq 0 : \\ StartOf(N[x, a, k]) = EndOf(N^{in}[x, a, k]) \end{aligned} \tag{2.15}$$

$$\begin{aligned} \forall x \in V, \forall k \in \{0, \ldots, l-1\}, x \neq dest(a) \vee k \neq l-1 : \\ EndOf(N[x, a, k]) = StartOf(N^{out}[x, a, k]) \end{aligned} \tag{2.16}$$

$$NoOverlap(\bigcup_{a \in A, k \in \{1,\ldots,l-1\}} \forall x \in V : N[x, a, k]) \tag{2.17}$$

The constraint 2.17 represents another interesting case. The *no_overlap* function constrains a set of interval variables to not overlap with each other. By default (like in the 2.18's implementation) it can receive a list of interval variables and it will constraint them to not overlap and to be at least distant one to another (the difference between the start of one and end of another, where the former succeeds the latter) zero. In order to modify this behavior we pass a *sequence_var* containing the interval variables and we create a **transition matrix** which represents for each combination of variables the minimum distance they must maintain.

```
tm_size = num_layers * agents_len
tm = transition_matrix(tm_size)

for i in range(tm_size):
    for j in range(tm_size):
        if j not in range((i // num_layers) * num_layers, (i // num_layers)
        * num_layers + num_layers):
            tm.set_value(i, j, 1)

[model.add(no_overlap(sequence_var([N[vertex][agent][layer]
                                    for agent in range(agents_len)
                                    for layer in range(num_layers)]), tm))
 for vertex in range(edges_len)]
```

For a specific vertex (notice the outer loop on vertices) the same agent can occupy the same node in different layers with a distance between each possible pair at minimum 0, because traversing layers is not a real movement and occurs immediately. On the other hand, between different agents in the same or different layers, the distance between possible pairs must be at least 1.

$$NoOverlap(\bigcup_{a \in A, k \in \{1,\ldots,l-1\}} \forall (x, y) \in E : \{A[x, y, a, k], A[y, x, a, k]\}) \tag{2.18}$$

Finally a minimization constraint is added for the makespan variable.

```
model.add(model.minimize(makespan))
```

3 Results

This section is dedicated to the description and analysis of the results we obtained from the tests. The aim is to compare the two approaches that we have previously described to solve the MAPF problem, in particular, we used the following performance parameters:

time_solving (both Z3 and CPLEX): measured in seconds.

memory_usage (both Z3 and CPLEX): memory occupied in MB.

n_conflicts (only Z3): indicates assignments that created contradictions. If the formula can be satisfied but the number of conflicts is high means that the solver tried a lot of assignments therefore it didn't proceed in a goal-oriented way.

n_decision (only Z3): number of decisions.

Let's recall that for the SMT-based approach we used Z3 while we used IBM CP Optimizer for the scheduling approach. The latter provides the possibility to perform a parallel search, in this regard we have used 8 workers (default value).

3.1 Implementation

We have used the Python package NetworkX for the creation, manipulation of complex networks. This library provides also many standard graph algorithms, like finding an agent's shortest path, which was used in the CP-based approach to compute the optimal number of layers and in the SMT-based approach to compute a lower bound. For this purpose we have implemented the solving MAPF algorithm described by the authors with a little variation: we used an heuristic upper bound on the makespan rather than computing it with the **Push and Swap (PaS)** algorithm and we also used it as upper bound of the number of layers in the search cycle of the first feasible solution. This is because PaS, in addition to the upper bound computation, performs a first check on the existence of a feasible solution to the problem. Since we started from an heuristic upper bound, we moved the control of the existence of a solution introducing, in fact, an upper bound also for the number of layers. Instead, the SMT-based approach cycle the makespan starting with the makespan equal to the maximum shortest path until a feasible solution is found. In this way is sure that the first feasible solution found is also the best. Therefore we carried out some tests to compare the performance of these two approaches according to the parameters described above. Finally, a test can be briefly described like this:

1. Graph generation with NetworkX.
2. Agents generation. The number of agents may change according to a certain criterion (for example proportional to the number of nodes in the graph) while departure and arrival nodes may be set randomly or manually for each agent.
3. Execution of the resolution algorithms for both approaches described above.
4. Storage of performance parameters of the two algorithms executed with the optimal parameters.
5. Depending on the environment, increasing the size of the graph up to a predetermined maximum size.

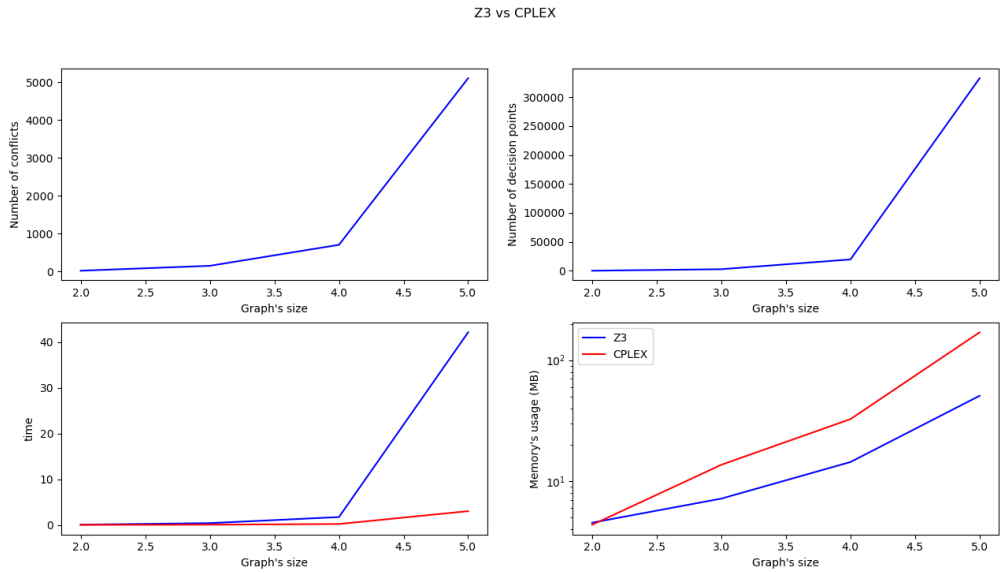
6. Plot of the results using console and Matplotlib to show the static graph or an animation of the agents moving.

From an implementation point of view we have developed an interface that would allow us to perform various tests in a flexible way, in particular allowing us to choose the type of graph, its size and the number of agents.

3.2 Testing

3.2.1 Extensive testing

The first test was performed on a two-dimensional grid generated by NetworkX by varying the size from *MIN_SIZE=2* to *MAX_SIZE=5* with *seed=42*. Also the number of agents varies with the size of the graph, in particular following the relationship $n\_agents = SIZE_i$ where $SIZE_i$ is the size of the graph in step i. This is to evaluate the performance of the two approaches to increasing the the complexity of the problem which, in general, is proportional to the size of the graph and the number of agents. The plot shown in the figure describe the trend of the parameters described above when the size of the graph changes. The first consideration is that *time_solving* and *memory_usage* increase as the size of the graph, as is easy to expect, however, implementation with CPLEX is faster than Z3 but uses more memory. The reason may depend on the is that the former performs a parallel search. Another observation that can be made is that the *n_conflicts* and *n_decision* grow as the size of the graph increases and therefore of the memory required for the search space as we expected.



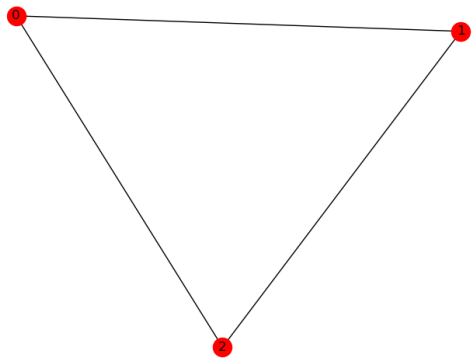
3.2.2 Synchronization and collision avoidance

We also carried out a test in order to quickly and easily verify the correctness of the implemented constraints. In this regard, we considered a very border line case: a graph with 3 nodes and 3 agents such that each of these must reach a position which is occupied by another, therefore the only satisfiable solution is that in which there is a synchronous movement of all which we recall being a hypothesis of MAPF.

```
ENVIRONMENT:
0: {0, 1, 2}
1: {0, 1, 2}
2: {0, 1, 2}
AGENTS:
[(0, 1), (1, 2), (2, 0)]
```

As we can see from the output both Z3 and CPLEX have found the same and only solution to the problem, also avoiding the swap between two agents and the case in which multiple agents are in the same node, which are not possible according to MAPF hypothesis.

```
================================================
Z3
Makespan 1: sat
Agent 0:        0       1
Agent 1:        1       2
Agent 2:        2       0
------------------------------------------------
================================================
CPLEX
Solution with makespan 1:
Agent 0:        0       1
Agent 1:        1       2
Agent 2:        2       0
```



3.2.3 Warehouse

This is a classical MAPF environment, inspired by real-world autonomous warehouse applications, like those deployed in Amazon depots. The environment represents the generic structure of a warehouse where shelves are positioned parallel one to another forming usually tight corridors where only robot can navigate, making not easy the coordination between the different agents.

```
AGENTS:
[(51, 23), (2, 33), (36, 21), (3, 48), (29, 27)]
================================================================
Step 1) Searching for optimal number of layers
================================================================
Solution with makespan 10:
Agent 0:        51      41      42      43      34      29      24      18
      17      16      23
Agent 1:        2       3       13      14      22      27      32      39
      40      41      33
Agent 2:        36      35      45      46      47      48      38      37
      31      26      21
Agent 3:        3       4       5       15      14      22      27      32
      39      38      48
Agent 4:        29      24      18      17      16      6       5       15
      14      22      27
================================================================
Step 2) Solving with 4 layers
================================================================
Solution with makespan 9:
Agent 0:        51      41      33      28      23      16      17      16
      23      23
Agent 1:        2       3       4       14      15      15      16      23
      28      33
```

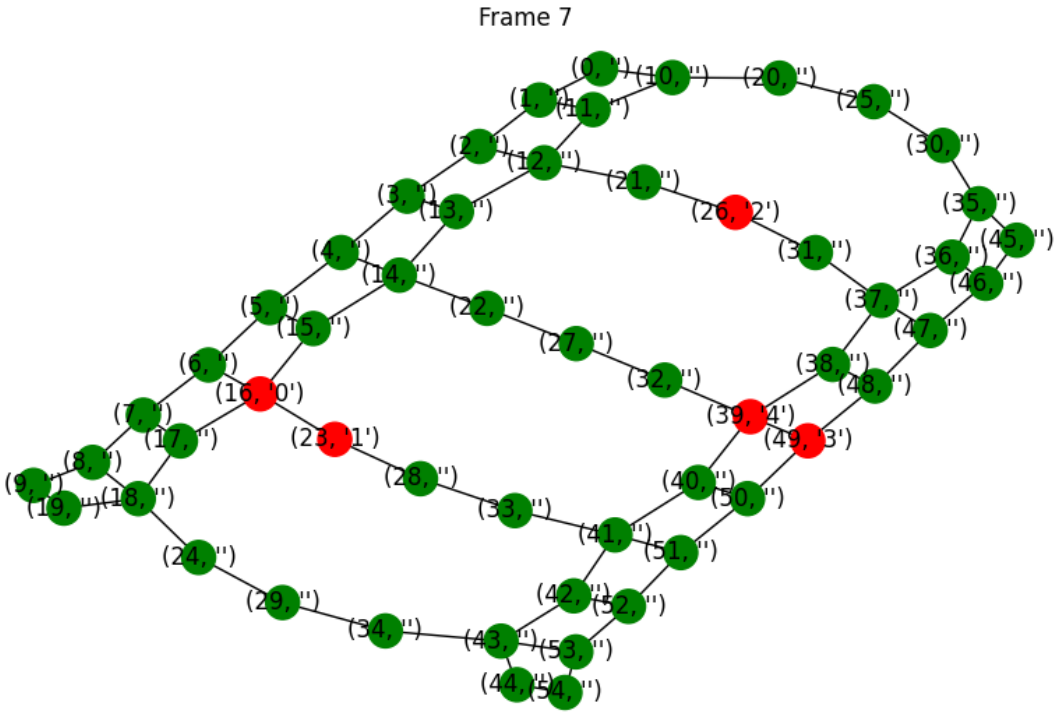


Figure 1: A frame of the animation representing 5 agents moving in a 10×7 warehouse with 4 shelves and corridors of width 1.

Agent 2:	36	37	31	26	21	26	31	26
21	21							
Agent 3:	3	4	14	22	27	32	39	49
48	48							
Agent 4:	29	34	43	42	41	40	40	39
32	27							

3.2.4 Dungeon

Dungeons represent specific environments that are widely used in video games, in particular in the rogue-like genre. Dungeons are usually represented as a set of rooms linked by corridors such that each room is accessible from only one or more corridors. In real world scenarios they can be useful to simulate indoor environments such as houses and offices. In this regard, exploiting NetworkX, we have developed a function for automatic and pseudo-random generation of dungeons environments specifying:

room_num the number of rooms

room_size_min minimum room size (side length)

room_size_max maximum room size (side length)

corridor_length_min minimum corridor length

corridor_length_max maximum corridor length

We performed tests on this type of graph starting from these hypotheses:

- The number of agents must be equal to the number of rooms.
- There must be only one agent in each room.
- Each agent's goal is in another room

We consider this test a very interesting case for the MAPF problem because the agents have to coordinate to use smartly the corridors which can be traversed at most by one agent at a time.

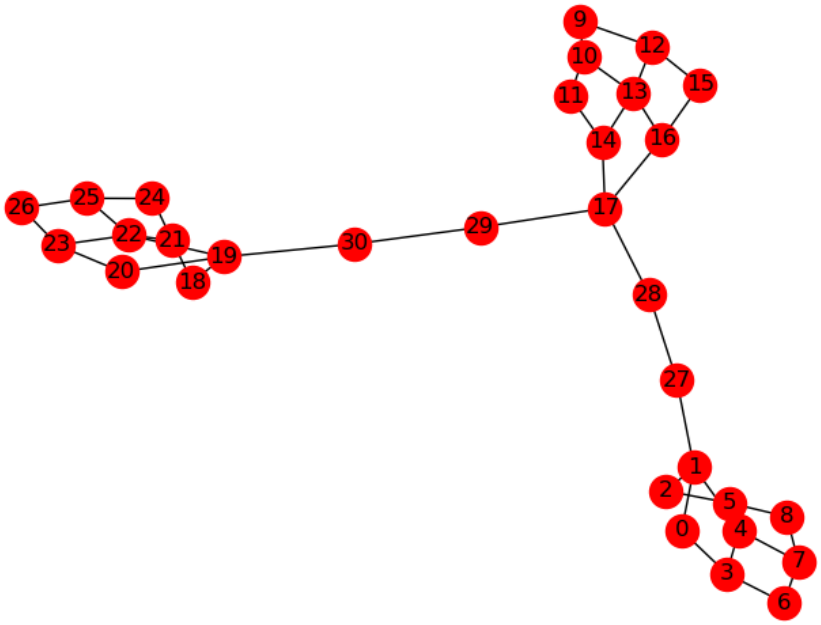


Figure 2: An example of indoor environment.

```
AGENTS:
[(4, 13), (13, 22), (22, 4)]
==========================================================
Z3
==========================================================
Makespan 8: unsat
Makespan 9: unsat
Makespan 10: sat
Agent 0:        4       1       27      28      17      14      14      17
      17      14      13
Agent 1:        13      10      9       10      13      16      17      29
      30      19      22
Agent 2:        22      22      19      30      29      17      28      28
      27      1       4
==========================================================
CPLEX
==========================================================

Step 1) Searching for optimal number of layers

==========================================================
Solution with makespan 10:
Agent 0:        4       1       27      28      17      16      13      13
      13      13      13
```

```
Agent 1:          13          14          14          14          14          14          17          29
      30          19          22
Agent 2:          22          19          30          29          29          17          28          27
      1           4           4
================================================================

Step 2) Solving with 3 layers

================================================================
Solution with makespan 10:
Agent 0:          4           1           27          28          17          14          11          10
      13          13          13
Agent 1:          13          12          9           10          13          16          17          29
      30          19          22
Agent 2:          22          19          30          29          29          17          28          27
      1           4           4
```

References

[1] Docplex documentation. `https://ibmdecisionoptimization.github.io/docplex-doc`.

[2] Z3 python guide. `https://ericpony.github.io/z3py-tutorial/guide-examples.htm`.

[3] Barták, Roman and Švancara, Ji?í and Vlk, Marek. A scheduling-based approach to multi-agent path finding with weighted and capacitated arcs. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '18, page 748–756, Richland, SC, 2018. International Foundation for Autonomous Agents and Multiagent Systems.

[4] Sven Koenig. Learn all about multi-agent path finding (mapf). `http://mapf.info`.

[5] Sven Koenig. Multi-agent path finding. `http://idm-lab.org/slides/mapf-tutorial.pdf`.

[6] Nikolaj Bjørner, Leonardo de Moura, Lev Nachmanson, and Christoph Wintersteiger. Programming z3. `https://theory.stanford.edu/~nikolaj/programmingz3.html`.