PLANNING and APPROXIMATE REASONING

Master in Artificial Intelligence – Practical Exercise 1

Cleaner Robotic Task

Miquel Sugrañes
Hajar Lachheb
Planning and Approximate Reasoning
Professor: Hatem Rashwan
Master in Artificial Intelligence – 2022/2023

Contents

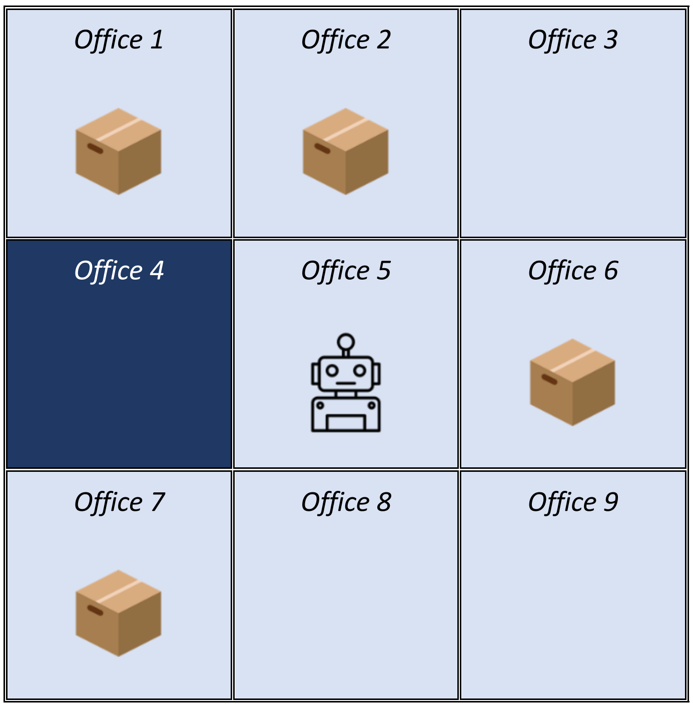
1. Introduction to the problem

The world in this planning exercise represents a floor with 9 offices, organized in a 3x3 matrix. Each office might contain a box, and in addition it might be clean or dirty. In this situation, the task is going to be for a robot to clean each one of the offices but with some restrictions:

- If a dirty office contains a box, the robot must first move this box out of the office to be able to clean it. Only empty offices (without boxes in them) can be cleaned.
- The robot can only move from one office to an adjacent one. This means that it can move horizontally or vertically, and only to office right next to the one it is.
- The robot can move a box to an adjacent office, but only if the office it wants to move the box to is empty (there can only be one box per office at the same time).
- The total number of boxes present in the world must be in the range between 0 and 8.

At the end, all offices must be clean, and the boxes in the position specified in the goal state.

A visual example of the world is shown below, with 4 boxes in offices 1, 2, 6 and 7, and the robot in the office 5. In this case we define office 4 as dirty (dark blue color) and the rest of them clean (light blue color)



2. Analysis of the problem

The problem restrictions require some predicates to make sure we accomplish the task without breaking any rule.

We must check the following:

1) If an office is empty or contains a box.
2) If an office is clean or dirty.
3) If an office is adjacent to another one.
4) The position of the robot (in which office it is).
5) The position of a certain box (in which office a certain box is).

So, this leads us to the following predicates:

1) `(empty ?office)`
2) `(clean ?office)`
3) `(adjacent ?office1 ?office2)`
4) `(at_robot ?office)`
5) `(at_box ?box ?office)`

We could also use a predicate to indicate the dirty state of on office, but in our case, we will do it by defining the negative state of clean.

```
not(clean office) = dirty(office)
```

By doing this we reduce the number of predicates and make the code easier to implement.

Looking at the predicates above, we can see that only two objects are necessary for our PDDL implementation: **office** and **box**. We don't need to define any *robot* or other objects.

Finally, the actions we must implement have to deal with the following needs:

1) The robot needs to move from office to another one.
2) The robot needs to clean an office.
3) The robot needs to move one box from an office to an adjacent one.

Therefore, the actions we must implement are going to be the following:

1) `move(origin, destination)`
2) `clean_office(office)`
3) `push_box(origin, destination, box)`

3. PDDL Implementation

1. *Domain*

The domain file contains the predicates and the actions that we must implement to solve the problem. This predicates and actions will be the ones explained before in the analysis of the problem.

In this part of the document, we will specify how the predicates and actions are implemented in our *pddl* domain file.

- Predicates

The following code shows how are the predicates defined.

```
(:predicates ;todo: define predicates here
    (at_robot ?o) ;Is Robot at the specific office 'office'??
    (at_box ?box ?office); Is a certain box located in the office 'office'?
    (clean ?office); Office 'office' is clean
    (empty ?office); There isn't any box in office 'office'
    (adjacent ?org ?des); Offices 'office1' and 'office2' are adjacent (horizontally
                        ; or vertically)
)
```

- Actions
    - Move action

The first action implemented is going to let the robot move from one office to an other. This action is resumed in the next table:

Name	`move`
Parameters	`?Origin` = origin of the robot at the moment of performing the action. `?Destination` = destination of the robot at the end of performing the action.
Preconditions	`adjacent(?org ?des)` = the origin and destination offices must be adjacent. `at_robot(?org)` = the robot must be in the origin office.
Effect	`at_robot(?des)` = the robot is going be in the destination office. `not(at_robot(?org))` = the robot is no longer going to be in the origin office.

- Clean office

This action is going to let the robot clean an office if it is not. It is necessary for the specific office to not be clean, and the robot must be in that same office in order to clean it.

Name	`clean_office`
Parameters	`?Office` = office to be cleaned.
Preconditions	`empty(?office)` = the office must be empty (without any boxes). `at_robot(?office)` = the robot must be in the office. `not(clean(?office))` = the office must be dirty (not clean).
Effect	`clean(?office)` = the office is going to be clean.

- Push box

This action is going to let the robot push a box from one office to an adjacent one. This must be done if an office has a box in it and it is also dirty, in order to clean it.

Name	`push_box`
Parameters	`?Office1` = the office where the box is initially located. `?Office2` = the office where the box is going to be moved to. `?B` = the box that must be pushed.
Preconditions	`empty(?Office2)` = the box's destination office must be empty. `at_robot(?Office1)` = the robot must be in the office that has a box in it. `at_box(?B ?Office1))` = the box must be in the origin office. `not(empty(?Office1))` = the office initially containing the box is not empty. `adjacent(?Office1 ?Office2))` = the offices must be adjacent.
Effect	`empty(?Office1)` = the office initially containing the box is going to be empty. `not(at_box(?B ?Office1))` = the box is not going to be in the initial office. `at_box(?B ?Office2))` = the box is going to be in the final office. `not(empty(?Office2))` = the final office is no longer going to be empty.

The code for the implementation of this actions is the following:

```
(:action clean_office ; Action to clean an office 'Office'
    :parameters (?Office)
    :precondition (and
        (at_robot ?Office) (empty ?Office) (not(clean ?Office))
    )
    :effect (and
        (clean ?Office)
    )
)

(:action move ; Action for the robot to move from Office 1 to Office 2
    :parameters (?Origin ?Destination)
    :precondition (and
        (adjacent ?Origin ?Destination) (at_robot ?Origin)
    )
    :effect (and
        (at_robot ?Destination) (not(at_robot ?Origin))
    )
)

(:action push_box ; Action to push a box 'B' from Office 1 to Office 2
    :parameters (?Office1 ?Office2 ?B)
    :precondition (and
        (adjacent ?Office1 ?Office2) (empty ?Office2) (not (empty ?Office1))
        (at_robot ?Office1) (at_box ?B ?Office1)
    )
    :effect (and
        (at_box ?B ?Office2) (not (empty ?Office2))
        (not(at_box ?B ?Office1)) (empty ?Office1)
    )
)
```

2. *Problem*

The problem file contains basically three definitions: the objects, the initial state, and the goal state.

- Objects

In our implementation, we defined two different sets of objects: **offices** and **boxes**.

In the world there are always 9 offices, so this is considered by defining nine objects, one for each. The name of each one of these offices objects are set by adding the number of the office to an 'o' character, in example: for the *office 4* the '**o4**' object is defined.

On the other hand, the number of boxes may vary in every test case, so the number of objects representing a box will also be different. Anyway, the name of the boxes objects is set by adding a capital letter in alphabetical order at the end of the word 'box'. In example, if there are two boxes in the world, the first one will be named '**boxA**' and the second one '**boxB**'.

The code for the objects definition of the first test case (see below in part 4 – Test cases) is the following:

```
(:objects
 ; For this example 2 boxes are defined: boxA and boxB,
 ; and also the nine offices: from o1 to o9
     o1 o2 o3 o4 o5 o6 o7 o8 o9
     boxA boxB
 )
```

- Initial state

In this part of the code, we must specify all the predicates that define the initial state of the world. The initial state of the world encompasses the position of each one of the offices and their state (clean or dirty and empty or not), as well as the placement of the boxes and the robot at the beginning.
It is very important in this initial state to define the adjacent relation of the offices, as it is necessary for the different actions implemented. This makes the code a bit long since each office has to be specified as adjacent to each of the corresponding ones.
In this case, the code for the initial state definition of the first test case (see below in part 4 – Test cases) is the following:

```
(:init
        ; In our initial state the Robot will be in the office o4 meanwhile
        ; the boxes will be located in office o6 (boxB) and office o5 (boxA).

        (at_robot o4)
        (at_box boxA o5)
        (at_box boxB o6)
        (clean o1)
        (clean o2)
        (clean o3)
        (clean o4)
        (clean o5)
        (clean o6)
        (clean o7)
        (clean o8)
        (clean o9)
        (empty o1)
        (empty o2)
        (empty o3)
        (empty o4)
        (not(empty o5))
        (not(empty o6))
        (empty o7)
        (empty o8)
        (empty o9)
        (adjacent o1 o2)
        (adjacent o1 o4)
        (adjacent o2 o1)
        (adjacent o2 o3)
        (adjacent o2 o5)
        (adjacent o3 o2)
        (adjacent o3 o6)
        (adjacent o4 o1)
        (adjacent o4 o5)
        (adjacent o4 o7)
        (adjacent o5 o2)
        (adjacent o5 o4)
        (adjacent o5 o6)
        (adjacent o5 o8)
        (adjacent o6 o3)
        (adjacent o6 o5)
        (adjacent o6 o9)
        (adjacent o7 o4)
        (adjacent o7 o8)
        (adjacent o8 o7)
        (adjacent o8 o5)
        (adjacent o8 o9)
        (adjacent o9 o8)
        (adjacent o9 o6)
    )
```

- Goal state

The goal state definition must include, as the initial state definition, all those necessary predicates to describe the final state of each one of the objects of the world that we want to accomplish from our initial state.

In this part of the code, we must define the position of the boxes within the offices, and the state of each one of these offices (clean or dirty and empty or not). The position of the robot, in this case, is not strictly necessary, even it could be defined too.

In this case, the definition of the final state will also vary depending on the number and position of the boxes, the number of dirty offices and the final position of the robot, if defined.

The code for the goal state definition of the first test case (see below in part 4 – Test cases) is the following:

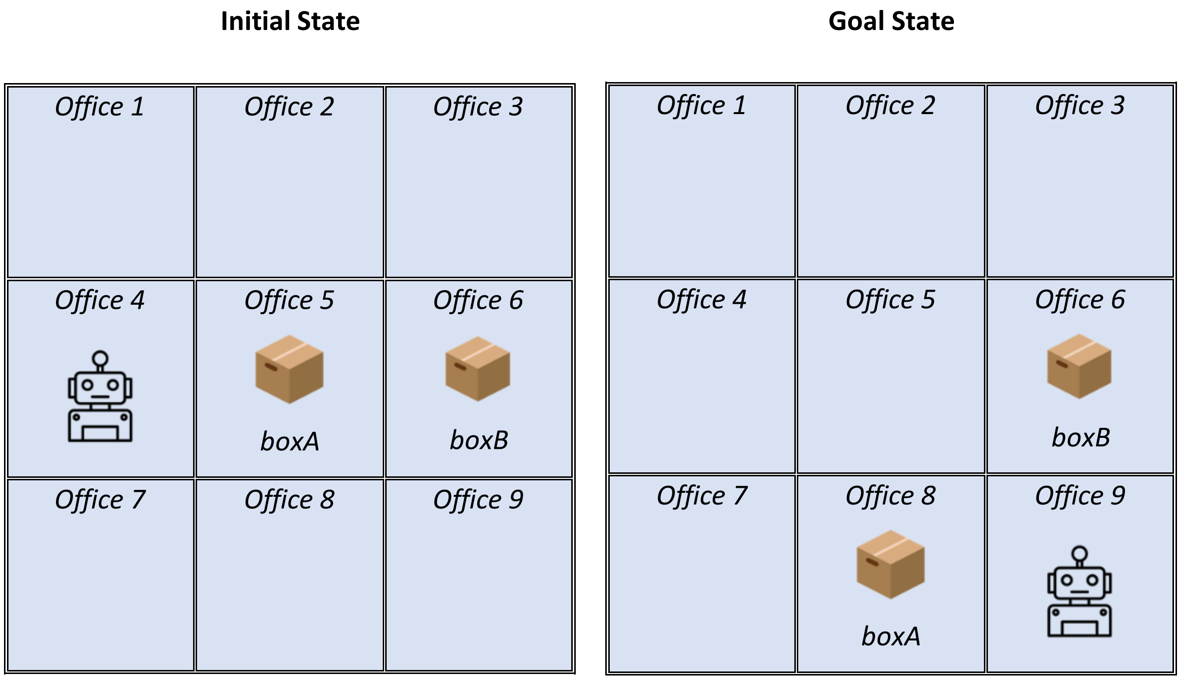
```
(:goal (and
       (at_box boxB o6)
       (at_box boxA o8)
       (at_robot o9)
       (empty o1)
       (empty o2)
       (empty o3)
       (empty o4)
       (empty o5)
       (not(empty o6))
```

```
      (empty o7)
      (not(empty o8))
      (empty o9)
   ))
```

4. Testing cases and results

To test the results, we created 4 different test cases. The first one was already given to us in the explanation of the assignment. The other 3 are variants with different levels of complexity.
For each one of the test cases, the number of nodes generated, the number of nodes expanded, and total time are annotated and analyzed.

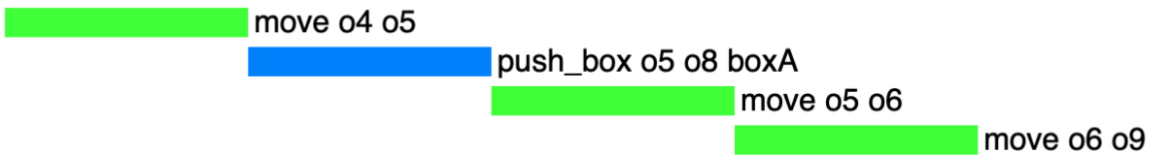
a. Test case 1



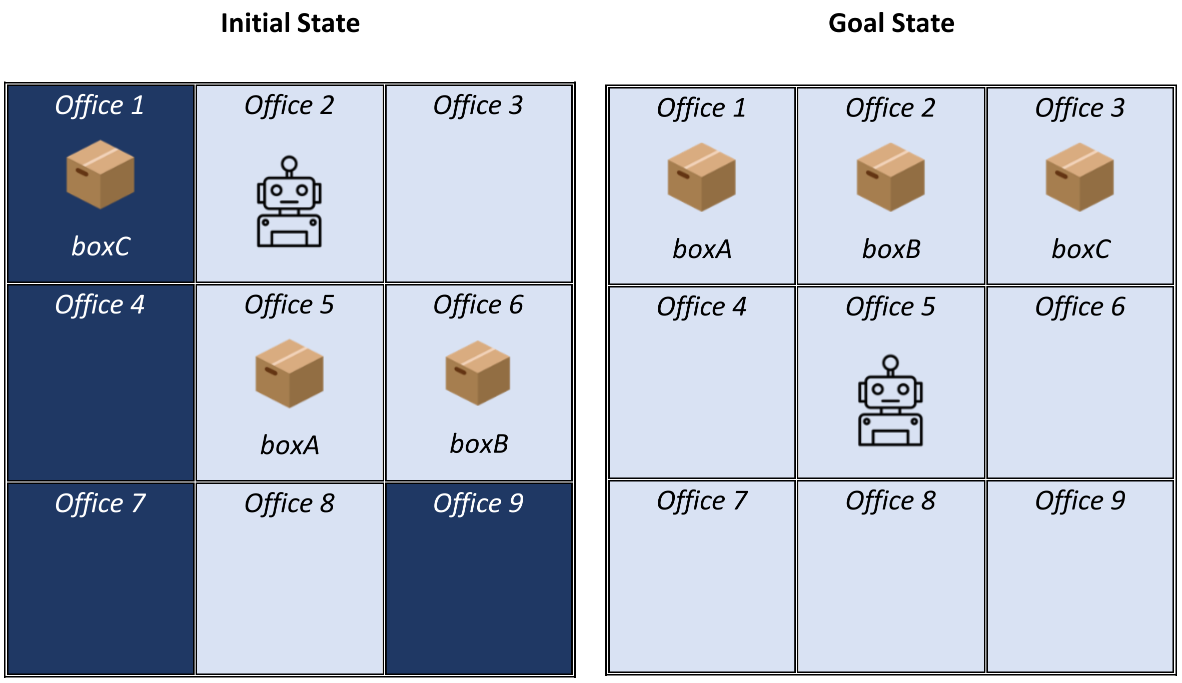
- *Test case metrics*

Nodes generated	Nodes expanded	Total time (s)	N° actions
23	8	-3.8e-10	4

- *Plan report*



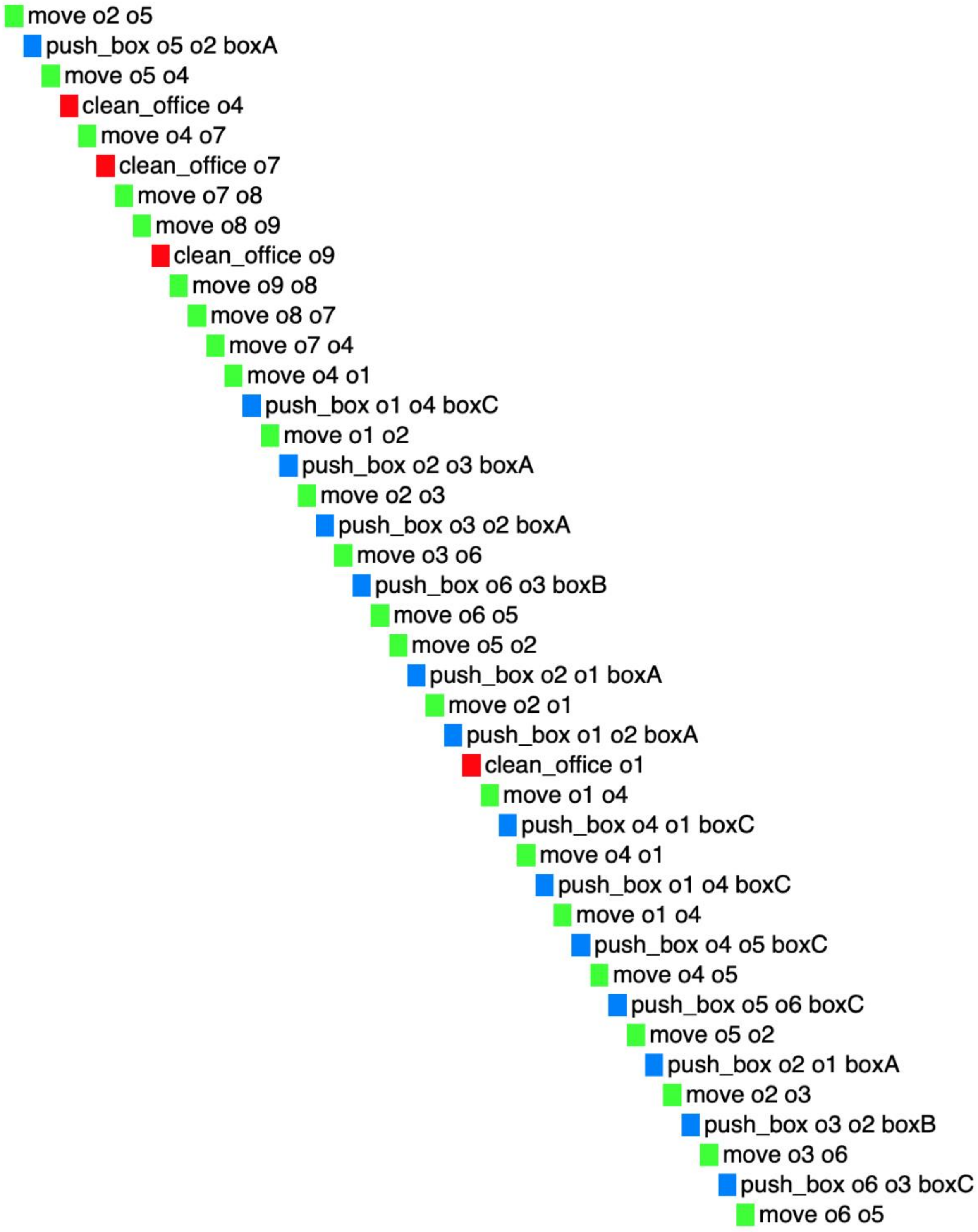
b. Test case 2



- *Test case metrics*

Nodes generated	Nodes expanded	Total time	N° actions
555	474	0.004	41

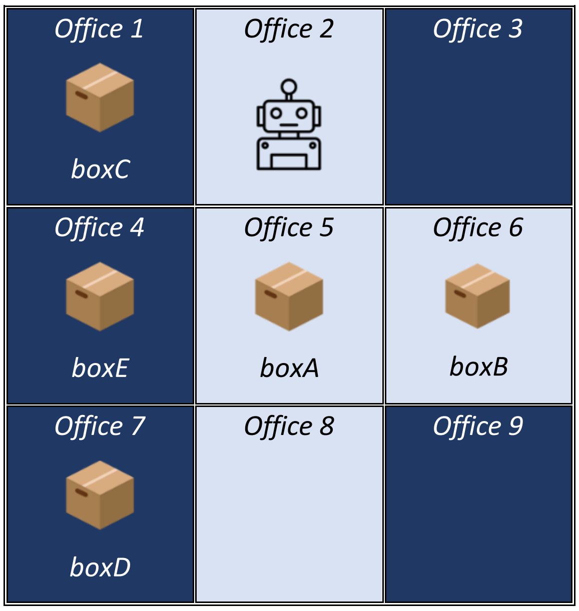
- *Plan report*



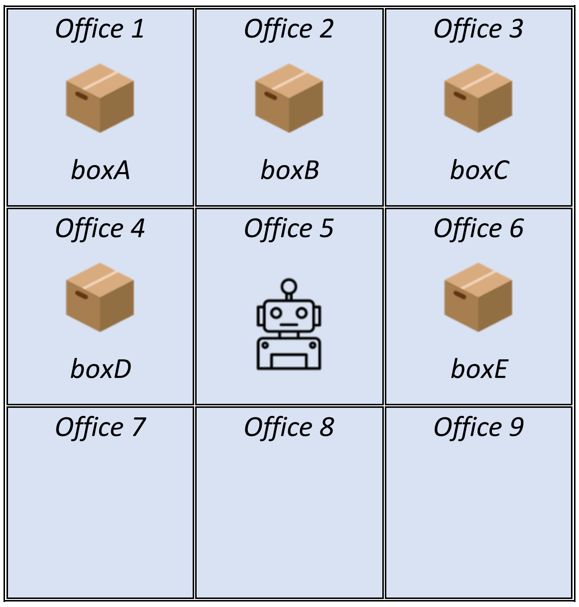
```
🟩 move o2 o5
🟦 push_box o5 o2 boxA
🟩 move o5 o4
🟥 clean_office o4
🟩 move o4 o7
🟥 clean_office o7
🟩 move o7 o8
🟩 move o8 o9
🟥 clean_office o9
🟩 move o9 o8
🟩 move o8 o7
🟩 move o7 o4
🟩 move o4 o1
🟦 push_box o1 o4 boxC
🟩 move o1 o2
🟦 push_box o2 o3 boxA
🟩 move o2 o3
🟦 push_box o3 o2 boxA
🟩 move o3 o6
🟦 push_box o6 o3 boxB
🟩 move o6 o5
🟩 move o5 o2
🟦 push_box o2 o1 boxA
🟩 move o2 o1
🟦 push_box o1 o2 boxA
🟥 clean_office o1
🟩 move o1 o4
🟦 push_box o4 o1 boxC
🟩 move o4 o1
🟦 push_box o1 o4 boxC
🟩 move o1 o4
🟦 push_box o4 o5 boxC
🟩 move o4 o5
🟦 push_box o5 o6 boxC
🟩 move o5 o2
🟦 push_box o2 o1 boxA
🟩 move o2 o3
🟦 push_box o3 o2 boxB
🟩 move o3 o6
🟦 push_box o6 o3 boxC
🟩 move o6 o5
```

c. Test case 3

Initial State



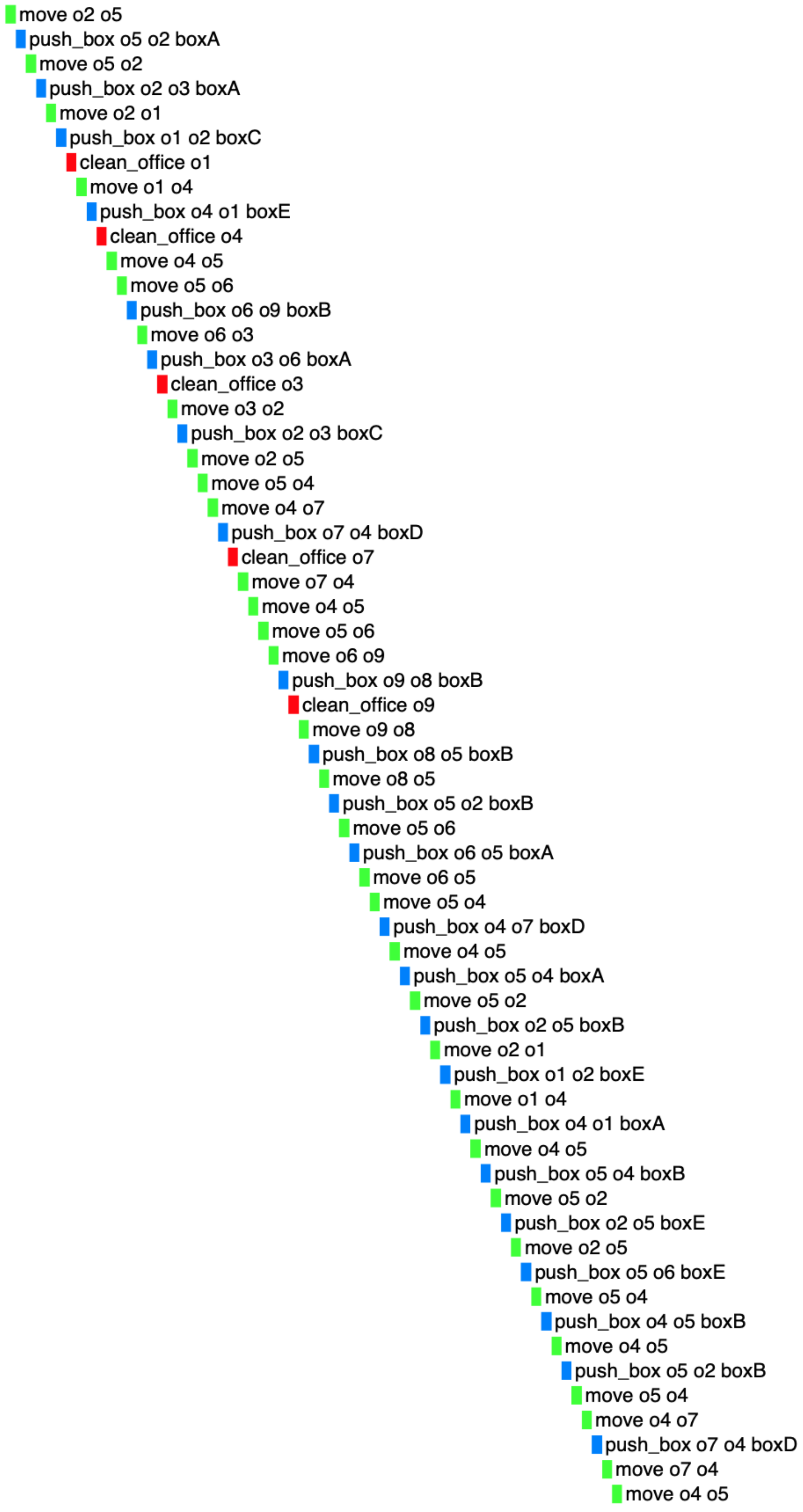
Goal State



- *Test case metrics*

Nodes generated	Nodes expanded	Total time	N° actions
985	260	0.024	61

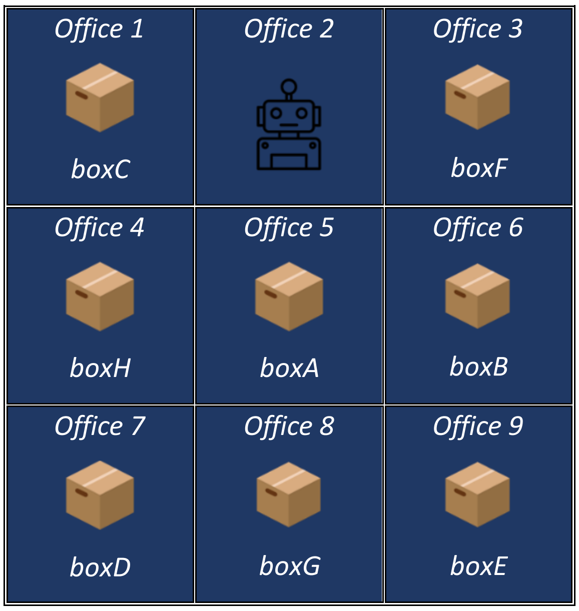
- *Plan report*



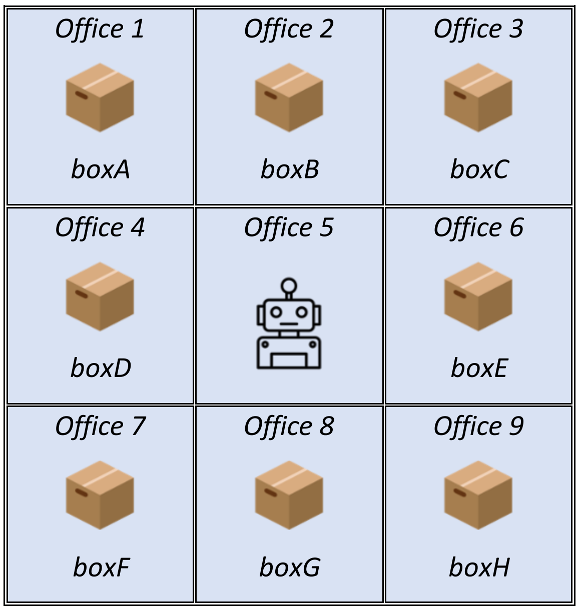
```
🟩 move o2 o5
🟦 push_box o5 o2 boxA
🟩 move o5 o2
🟦 push_box o2 o3 boxA
🟩 move o2 o1
🟦 push_box o1 o2 boxC
🟥 clean_office o1
🟩 move o1 o4
🟦 push_box o4 o1 boxE
🟥 clean_office o4
🟩 move o4 o5
🟩 move o5 o6
🟦 push_box o6 o9 boxB
🟩 move o6 o3
🟦 push_box o3 o6 boxA
🟥 clean_office o3
🟩 move o3 o2
🟦 push_box o2 o3 boxC
🟩 move o2 o5
🟩 move o5 o4
🟩 move o4 o7
🟦 push_box o7 o4 boxD
🟥 clean_office o7
🟩 move o7 o4
🟩 move o4 o5
🟩 move o5 o6
🟩 move o6 o9
🟦 push_box o9 o8 boxB
🟥 clean_office o9
🟩 move o9 o8
🟦 push_box o8 o5 boxB
🟩 move o8 o5
🟦 push_box o5 o2 boxB
🟩 move o5 o6
🟦 push_box o6 o5 boxA
🟩 move o6 o5
🟩 move o5 o4
🟦 push_box o4 o7 boxD
🟩 move o4 o5
🟦 push_box o5 o4 boxA
🟩 move o5 o2
🟦 push_box o2 o5 boxB
🟩 move o2 o1
🟦 push_box o1 o2 boxE
🟩 move o1 o4
🟦 push_box o4 o1 boxA
🟩 move o4 o5
🟦 push_box o5 o4 boxB
🟩 move o5 o2
🟦 push_box o2 o5 boxE
🟩 move o2 o5
🟦 push_box o5 o6 boxE
🟩 move o5 o4
🟦 push_box o4 o5 boxB
🟩 move o4 o5
🟦 push_box o5 o2 boxB
🟩 move o5 o4
🟩 move o4 o7
🟦 push_box o7 o4 boxD
🟩 move o7 o4
🟩 move o4 o5
```

d. Test case 4

Initial State



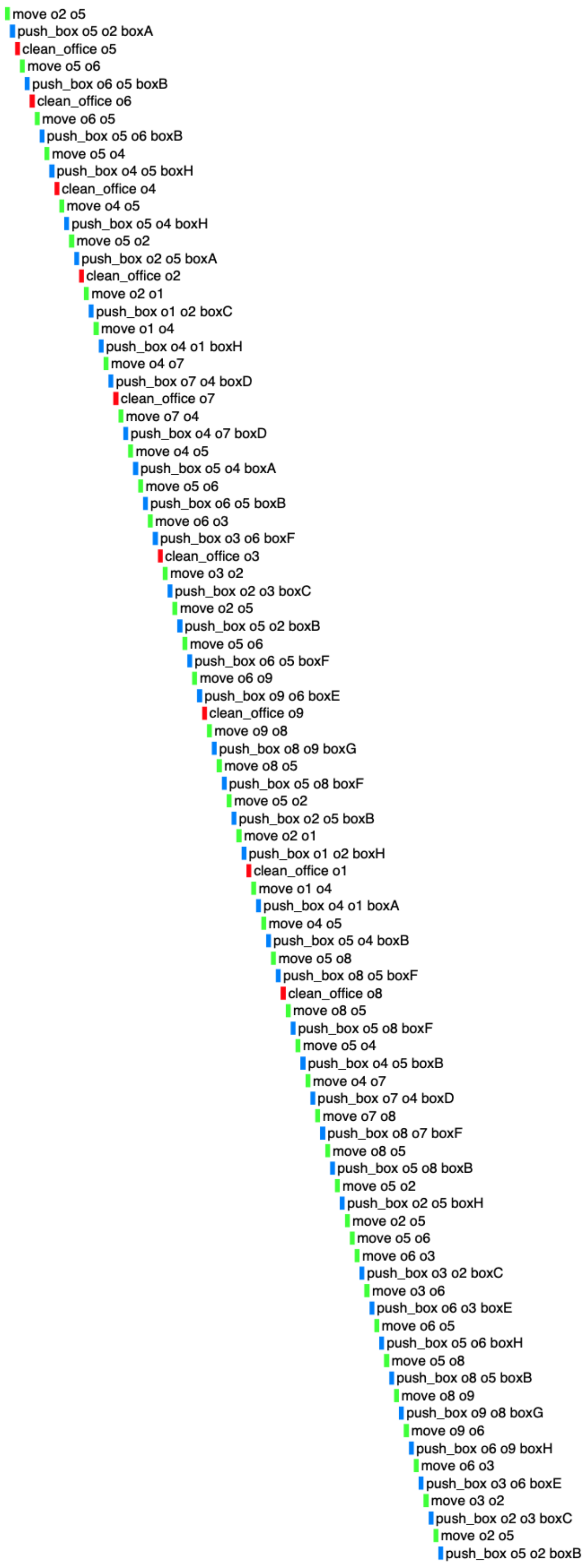
Goal State



- *Test case metrics*

Nodes generated	Nodes expanded	Total time	N° actions
5407	1594	0.212	89

- *Plan report*



```
move o2 o5
push_box o5 o2 boxA
clean_office o5
move o5 o6
push_box o6 o5 boxB
clean_office o6
move o6 o5
push_box o5 o6 boxB
move o5 o4
push_box o4 o5 boxH
clean_office o4
move o4 o5
push_box o5 o4 boxH
move o5 o2
push_box o2 o5 boxA
clean_office o2
move o2 o1
push_box o1 o2 boxC
move o1 o4
push_box o4 o1 boxH
move o4 o7
push_box o7 o4 boxD
clean_office o7
move o7 o4
push_box o4 o7 boxD
move o4 o5
push_box o5 o4 boxA
move o5 o6
push_box o6 o5 boxB
move o6 o3
push_box o3 o6 boxF
clean_office o3
move o3 o2
push_box o2 o3 boxC
move o2 o5
push_box o5 o2 boxB
move o5 o6
push_box o6 o5 boxF
move o6 o9
push_box o9 o6 boxE
clean_office o9
move o9 o8
push_box o8 o9 boxG
move o8 o5
push_box o5 o8 boxF
move o5 o2
push_box o2 o5 boxB
move o2 o1
push_box o1 o2 boxH
clean_office o1
move o1 o4
push_box o4 o1 boxA
move o4 o5
push_box o5 o4 boxB
move o5 o8
push_box o8 o5 boxF
clean_office o8
move o8 o5
push_box o5 o8 boxF
move o5 o4
push_box o4 o5 boxB
move o4 o7
push_box o7 o4 boxD
move o7 o8
push_box o8 o7 boxF
move o8 o5
push_box o5 o8 boxB
move o5 o2
push_box o2 o5 boxH
move o2 o5
move o5 o6
move o6 o3
push_box o3 o2 boxC
move o3 o6
push_box o6 o3 boxE
move o6 o5
push_box o5 o6 boxH
move o5 o8
push_box o8 o5 boxB
move o8 o9
push_box o9 o8 boxG
move o9 o6
push_box o6 o9 boxH
move o6 o3
push_box o3 o6 boxE
move o3 o2
push_box o2 o3 boxC
move o2 o5
push_box o5 o2 boxB
```

5. Conclusions

It is important to emphasize that all the test cases have result with a plan to solve the problem. There is no test case with partial solutions or uncomplete plans, therefore, we could say that our algorithm is complete. Even so, we think that more accurate and efficient plans could be found, as we can see for example in test case 4, where the robot starts in a dirty office, but instead of starting by cleaning it, the robot moves to an adjacent one (office 5) and after pushing the box out of it, he cleans it. Maybe, starting in a dirty office, it could be more efficient to start by cleaning this starting office and then move to the next one.

As we can see from the results metrics, as more difficult the task is, more nodes are generated and expanded. Also, the time to solve the problem increases exponentially. This may be obvious but is important to see how the plan searching behaves, because we might be in the situation that trying to implement a too complex plan consumes all our computational resources, as well as computer memory. So, we think it is relevant to check the metrics and always try to find an easy way of implementing the world and the problem. In our case, the easiest test case is the test case 1, and it increases its difficulty until test case 4.

All in all, this exercise showed us how to implement a problem and a domain in *pddl* and represent a world state to fins a plan to get to the goal. Also, we have been able to see how the searching of a plan behaves in terms of computational cost, even the examples are quite easy for the moment. Realizing all these things gave us a good approach of the planning concept and implementation.