PLANNING and APPROXIMATE REASONING

Master in Artificial Intelligence – Practical Exercise 2

Waiter Robotic Task

Hajar Laccheb
Miquel Sugrañes
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 restaurant with 7 different areas, organized in a 3x2 matrix for the serving areas and an extra room which represents the kitchen (buffet area – BTA). Each serving area might contain a customer, and in addition this customer can be served or not. In this situation, the task is going to be for a robot to serve each one of the customers but with some restrictions:

- In order to serve a client, the robot must have a filled plate. To fill a plate the robot must hold an empty plate and be located in the BTA area.
- The robot can only hold one plate at a time, either if it is empty or filled.
- The robot can only move from one dining area to an adjacent one. The adjacency between areas is defined in figure 1, and the robot can't go from one area to another one if there is a wall in between.
- Only clients who are not served must be served.
- At the end, all clients must be served.

A visual example of the world is shown below, with one client in the PMA area, the robot in the BTA area holding the dish, and the different areas defined. We painted in green the dining areas and in blue the BTA. The robot can move from one area to an adjacent one if they are connected with a slashed line, while we is not allowed to trespass the dark black lines which represents walls (i.e.: we can't move from AMA to PMA). To represent a plate which is filled we will use the chicken icon, while an empty one is represented with the empty plate icon.

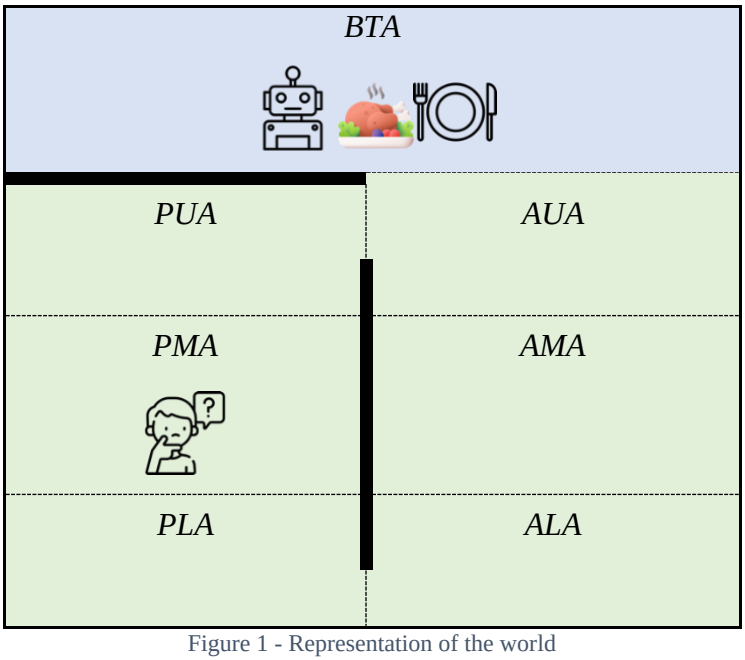


Figure 1 - Representation of the world

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) The location of objects:
    - Location of plate
    - Location of agent (robot)
    - Location of customers (client)
2) If the robot is holding a plate or not.
3) If a plate is filled or not.
4) If a client is served or not.
5) If an area is adjacent to another one (not separated by a wall).

So, this leads us to the following predicates:

1) `(at ?object ?area)`
2) `(hasfood ?plate)`
3) `(served ?customer)`
4) `(adjacent ?area1 ?area2)`
5) `(holding ?waiter ?plate)`
6) `(emptyhanded ?waiter)`

The negations of each predicate will define the inverted state for each, i.e.:

`not(hasfood ?plate)`

will define the state of an empty plate.

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 four objects are necessary for our PDDL implementation: **location(area)**, **plate**, **customer,** and **agent(robot/waiter)**.

Since the robot (the waiter of the restaurant) is always going to be one and the same, we defined a constant named **Agent** that will refer to it. Moreover, the location of the buffet (where the robot can fill the plates) is also defined as a constant for our approach, in this case named **BTA**.

The types necessary for the implementation, thus, are the following: **waiter**, **location** and **customer**.

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

1) The robot needs to move from one area to another one.
2) The robot needs to fill a plate.
3) The robot needs to pick up a plate.
4) The robot needs to serve a customer with a dish.

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

1) `move(origin, destination)`
2) `fill(plate)`
3) `present(plate, customer, location)`
4) `pickup(waiter, plate, location)`

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
    (Adjacent ?x - location ?y - location) ;To define adjacent areas
    (At ?x - object ?y - location) ;To define the location of an object
    (HasFood ?p - plate) ;To define either a plate has food or not
    (Served ?c - customer) ;To define if a customer is served
    (Holding ?a - waiter ?p - plate) ;If the robot is holding a certain plate
    (EmptyHanded ?a - waiter) ;If the robot not holding any plate
)
```

- Actions
    - Move action

The first action implemented is going to let the robot move from one area of the restaurant to an adjacent one. This action is resumed in the next table:

Name	`Move`
Parameters	`?x` = origin of the robot at the moment of performing the action. `?y` = destination of the robot at the end of performing the action.
Preconditions	`Adjacent(?x ?y)` = the origin and destination areas must be adjacent. `At(Agent ?x)` = the robot must be in the origin area.

Effect	`At(Agent ?y)` = the robot is going be in the destination area.
	`not(At(Agent ?x))` = the robot is no longer going to be in the origin area.
	`(forall (?p - plate)`
	`(when (and (Holding Agent ?p))`
	`(and (At ?p ?y) (not (At ?p ?x) ))` = if the robot is holding any plate then the plate won't be in the origin position and it will be in the destination location.

    - Fill action

This action is going to let the robot fill a plate if it is not. It is necessary for the specific plate to not be already filled, and the robot must be in the BTA area in order to fill it.

Name	`Fill`
Parameters	`?p` = plate to be filled.
Preconditions	`At (Agent BTA)` = the robot must be in the BTA area. `not(HasFood(?p)` = the plate must be empty. `Holding(Agent ?p))` = the robot must be holding the plate to be filled.
Effect	`HasFood(?p)` = the plate is going to be filled.

- Present action

This action is going to let the robot serve a customer with a filled plate.

Name	`Present`
Parameters	`?p` = the plate that the robot serves to the client. `?c` = the client who is being served. `?x` = the location where the robot presents the dish to the client.
Preconditions	`At(Agent ?x)` = the agent must be in the location to serve the client. `At(?c ?x)` = the customer must be in the location where to be served. `Holding(Agent ?p)` = the agent must be holding the plate to serve. `HasFood(?p)` = the plate must have food. `not(Served ?c)` = the client must not be already served. `not(EmptyHanded Agent)` = the agent is not empty handed.
Effect	`Served(?c)` = the client is going to be served. `EmptyHanded(Agent)` = the agent is going to be empty handed. `not(Holding(Agent ?p)` = the agent will no longer hold the plate. `At(?p ?x)` = the plate will be in the area where it is served.

- Pick-Up plate action

This action is going to let the robot pick up a plate.

Name	`pickplate`
Parameters	`?p` = the plate to pick up. `?x` = the location where the robot picks up the plate.
Preconditions	`At(Agent ?x)` = the agent must be in the location in which to pick up the client. `At(?p ?x)` = the plate must be in the location in which to be picked up. `(not (exists (?i - item) (Holding Agent ?i)))` = the agent must not be holding any item. `EmptyHanded(Agent)` = the agent must be empty handed.
Effect	`not(EmptyHanded(Agent)` = the agent is no longer going to be empty handed. `Holding(Agent ?p)` = the agent will be holding the plate.

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

```
(:action Move ; Action for the robot to move around the restaurant
        :parameters (?x - location ?y - location)
        :precondition (and
            (At Agent ?x)
            (Adjacent ?x ?y)
            (not (At Agent ?y))
        )
        :effect (and
            (At Agent ?y)
            (not (At Agent ?x))
            (forall (?p - plate)
                (when (and (Holding Agent ?p))
                      (and (At ?p ?y) (not (At ?p ?x)))
                )
            )
        )
    )

(:action Fill ; Action to fill a plate of food
         :parameters (?p - plate)
        :precondition (and
            (At Agent BTA)
            (not (HasFood ?p))
            (Holding Agent ?p)
        )
        :effect (and
            (HasFood ?p)
        )
    )

(:action Present ; Action to serve a plate to a customer
        :parameters (?p - plate ?c - customer ?x - location)
        :precondition (and
            (At Agent ?x)
            (At ?c ?x)
            (Holding Agent ?p)
            (HasFood ?p)
            (not (Served ?c))
            (not (EmptyHanded Agent))
        )
        :effect (and
            (EmptyHanded Agent)
            (Served ?c)
            (not (Holding Agent ?p))
            (At ?p ?x)
        )
    )
```

```
(:action pickplate
        :parameters (?p - plate ?x - location)
        :precondition (and
            (At Agent ?x)
            (At ?p ?x)
            (not (exists (?i - item) (Holding Agent ?i)))
            (EmptyHanded Agent)
        )
        :effect (and
            (Holding Agent ?p)
            (not (EmptyHanded Agent))
        )
    )
```

2. Problem

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

- Objects

In our implementation, we defined different sets of objects: **plates**, **customer**, and **locations**. As we said before, the **Agent** and the **BTA** are already defined constants.

In the world there are always 6 dining areas, so this is considered by defining six objects, one for each. The name of each one of these areas will match with the figure 1, and they will be: **PUA**, **AUA**, **PMA**, **AMA**, **PLA**, **ALA**.

On the other hand, the number of plates may vary in every test case, so the number of objects representing a plate will also be different. Anyway, the name of the plates objects is set by adding a integer (starting with 1) next to the capital letter P. In example, if there are two plates in the world, the first one will be named '**P1** and the second one '**P2**'.

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

```
    (:objects
        P - plate
        C - customer
        PUA AUA PMA AMA PLA ALA - location
    )
```

- 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 restaurant's areas, as well as the placement of the plates, the customers, and the robot at the beginning of the problem.

It is very important in this initial state to define the adjacent relation of the areas, 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
        (At Agent BTA)
        (At P BTA)
        (not (HasFood P))
        (At C PMA)
        (not (Served C))
        (EmptyHanded Agent)
        (Adjacent BTA AUA)
        (Adjacent AUA BTA)
        (Adjacent AUA PUA)
        (Adjacent PUA AUA)
        (Adjacent PUA PMA)
        (Adjacent PMA PUA)
        (Adjacent PMA AMA)
        (Adjacent AMA PMA)
        (Adjacent AMA ALA)
        (Adjacent ALA AMA)
        (Adjacent ALA PLA)
        (Adjacent PLA ALA)
        (Adjacent PLA PMA)
        (Adjacent PMA PLA)
    )
```

- 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 will define the condition that the customers must be served. Moreover, we can specify the location of the agent at the end of the plan.
In this case, the definition of the final state will also vary depending on the number of the clients 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:

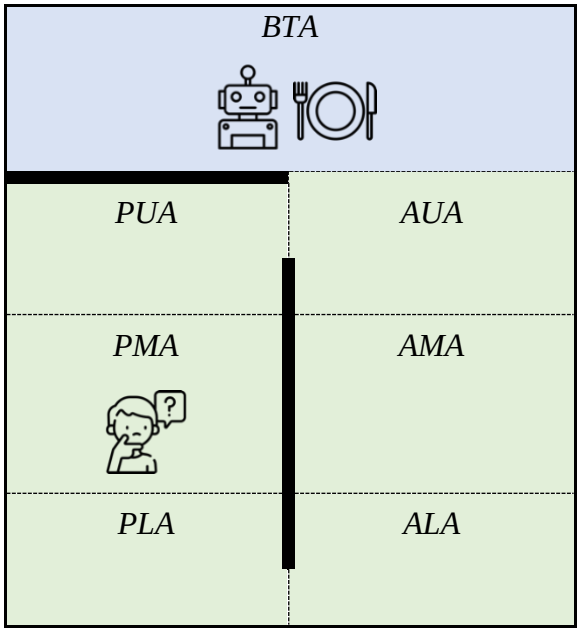
```
(:goal (and
        (Served C)
    ))
```

4. Testing cases and results

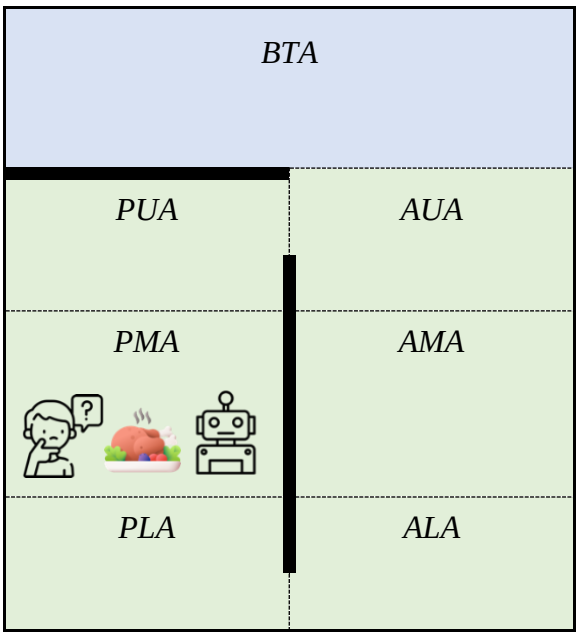
To test the results, we created 4 different test cases. The first one was already given to us in the explanation of the assignment. The other 3 are variants with different levels of complexity. For each one of the test cases, the number of nodes generated, the number of nodes expanded, and total time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

a. Test case 1

Initial State



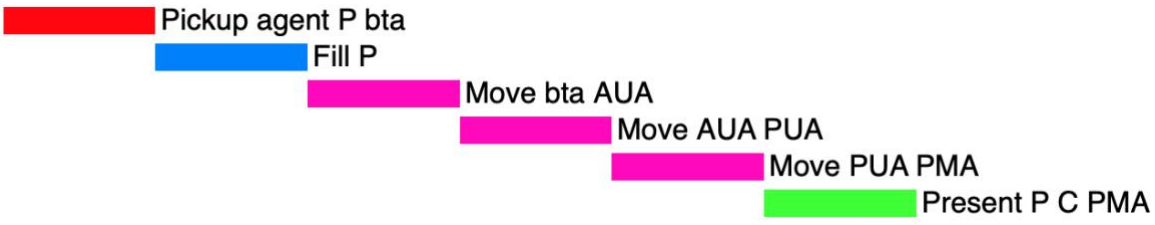
Goal State



i. Test case metrics

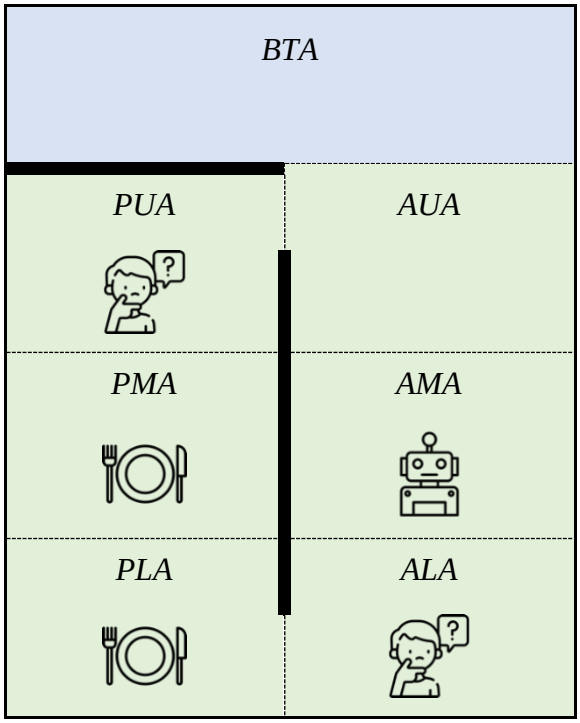
Nodes generated	Nodes expanded	Total time (s)	N° actions
31	22	-1.9e-10	6

ii. Plan report

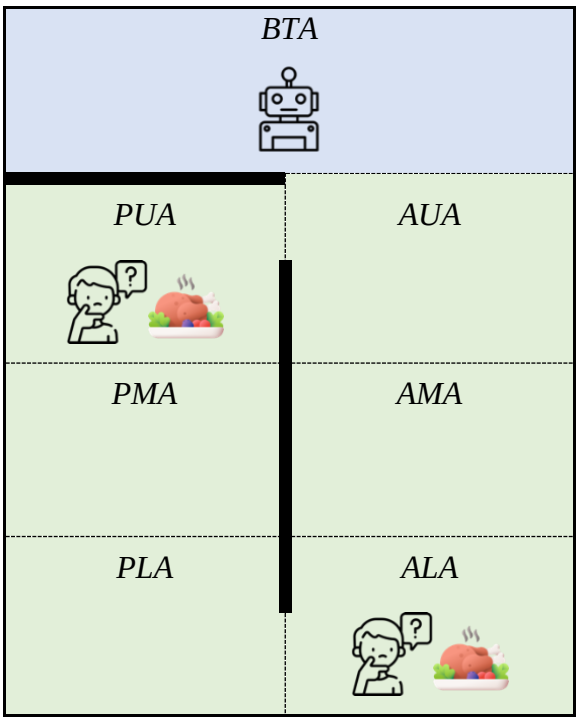


b. Test case 2

Initial State



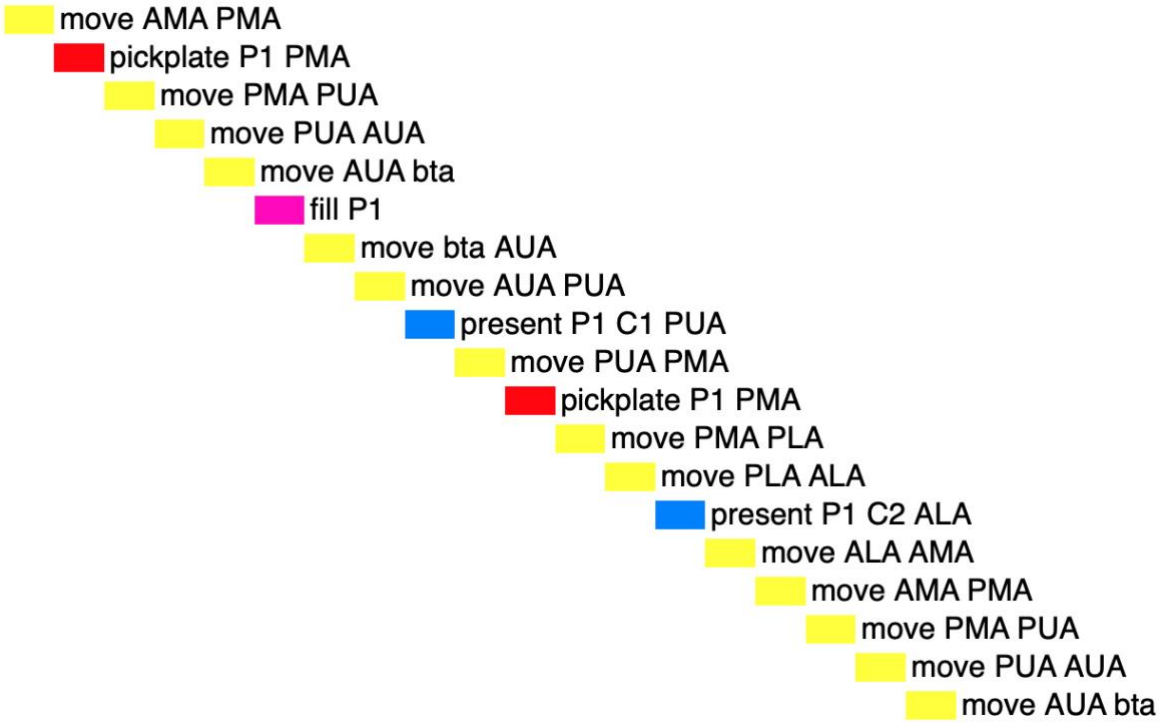
Goal State



i. Test case metrics

Nodes generated	Nodes expanded	Total time	N° actions
61	20	-1.04308e-10	19

ii. Plan report



c. Test case 3

For this last test case we implemented some more actions and predicates in order to increase the difficulty of the task.
To define the world, we added the fact that some plats can fall and be broken on the floor, with all the food also spread on the floor. This definition adds some more restrictions to the problem implemented before:

1) The robot can't move over some broken plate or dropped food.
2) The robot must be in an adjacent area to clean the floor.
3) The robot needs a broom in order to clean the floor.
4) The robot can't hold a plate while holding some trash (broken plates or dropped food).

To do this, we implemented the following actions:

- Put down action

This action is going to let the robot pick up a plate.

Name	`putdown`
Parameters	`?i` = the items the agent has cleaned (trash). `?x` = the location where the robot puts down the items (trash).
Preconditions	`At(Agent ?x)` = the agent must be in the location in which to put down the item. `Holding(Agent ?i)` = the agent must be holding the item. `not(EmptyHanded (Agent))` = the agent must not be empty handed.
Effect	`EmptyHanded(Agent)` = the agent is going to be empty handed. `not(Holding(Agent ?p))` = the agent will be no longer holding the item. `At(?i ?x)` = the item will be in the location where it is put down.

Code	`(:action putdown`
	`:parameters (?i - item ?x - location)`
	`:precondition (and`
	`(Holding Agent ?i)`
	`(At Agent ?x)`
	`(not (EmptyHanded Agent))`
	`)`
	`:effect (and`
	`(not (Holding Agent ?i))`
	`(At ?i ?x)`
	`(EmptyHanded Agent))))`

- Sweep food action

This action is going to let the robot pick up a plate.

Name	`sweepfood`
Parameters	`?df` = the dropped food. `?x` = the location where there is the dropped food. `?y` = the location where the robot is located.
Preconditions	`At(Agent ?y)` = the agent must be in its location. `At(?df ?x)` = the dropped food must be in the 'x' location. `or(Adjacent(?x ?y)(Adjacent(?y ?x))` = the location of the agent and the dropped food must be adjacent. `(exists (?b - broom) (Holding Agent ?b))` = There exists a broom and the agent is holding it. `(not (exists (?p - plate) (Holding Agent ?p)))` = The robot is not holding any plate from the world. `EmptyHanded (Agent)` = the agent must be empty handed.
Effect	`not(EmptyHanded(Agent))` = the agent is no longer going to be empty handed. `not(At(?df ?x))` = the dropped food will no longer be in the 'x' location.

Code	`(:action sweepfood`
	`:parameters (?df - droppedFood ?x - location ?y - location)`
	`:precondition (and`
	`(At ?df ?x) ;If the location x has both, robot sweeps it up in one action`
	`(At Agent ?y)`
	`(or (Adjacent ?x ?y) (Adjacent ?y ?x))`
	`(exists (?b - broom) (Holding Agent ?b))`
	`(not (exists (?p - plate) (Holding Agent ?p)))`
	`(EmptyHanded Agent))`
	`:effect (and`
	`(not (At ?df ?x))`
	`(not (EmptyHanded Agent))))`

- Sweep plate action

This action is going to let the robot pick up a plate.

Name	`sweepplate`
Parameters	`?bp` = the broken plate. `?x` = the location where there is the broken plate. `?y` = the location where the robot is located.
Preconditions	`At(Agent ?y)` = the agent must be in its location. `At(?bp ?x)` = the broken plate must be in the 'x' location. `or(Adjacent(?x ?y)(Adjacent(?y ?x))` = the location of the agent and the broken plate must be adjacent. `(exists (?b - broom) (Holding Agent ?b))` = There exists a broom and the agent is holding it. `(not (exists (?p - plate) (Holding Agent ?p)))` = The robot is not holding any plate from the world. `EmptyHanded(Agent)` = the agent must be empty handed.
Effect	`not(EmptyHanded(Agent))` = the agent is no longer going to be empty handed. `not(At(?bp ?x))` = the broken plate will no longer be in the 'x' location.
Code	`(:action sweepplate :parameters (?bp - brokenPlate ?x - location ?y - location) :precondition (and (At ?bp ?x) ;If the location x has both, robot sweeps it up in one action (At Agent ?y) (or (Adjacent ?x ?y) (Adjacent ?y ?x)) (exists (?b - broom) (Holding Agent ?b)) (not (exists (?p - plate) (Holding Agent ?p))) (EmptyHanded Agent)) :effect (and (not (At ?bp ?x)) (not (EmptyHanded Agent))))`

- Pick up broom action

This action is going to let the robot pick up a plate.

Name	`pickbroom`
Parameters	`?b` = the broom to pick up. `?x` = the location where the broom is at.
Preconditions	`At(Agent ?x)` = the agent must be in the 'x' location. `At(?b ?x)` = the broom must be in the 'x' location. `(not (exists (?i - item) (Holding Agent ?i)))` = The robot is not holding any item from the world.
Effect	`Holding(Agent ?b)` = the agent will be holding a broom.

Code	`(:action pickbroom`
	`:parameters (?b - broom ?x - location)`
	`:precondition (and`
	`(At agent ?x)`
	`(At ?b ?x)`
	`(not (exists (?i - item) (Holding Agent ?i))))`
	`:effect (and`
	`(Holding Agent ?b)))`

For the different actions implemented we had to update the types and predicates from the domain file. The new types are: **plate broom (item)**, **brokenPlate**, **droppedFood**.
The updated predicate is the Holding one:

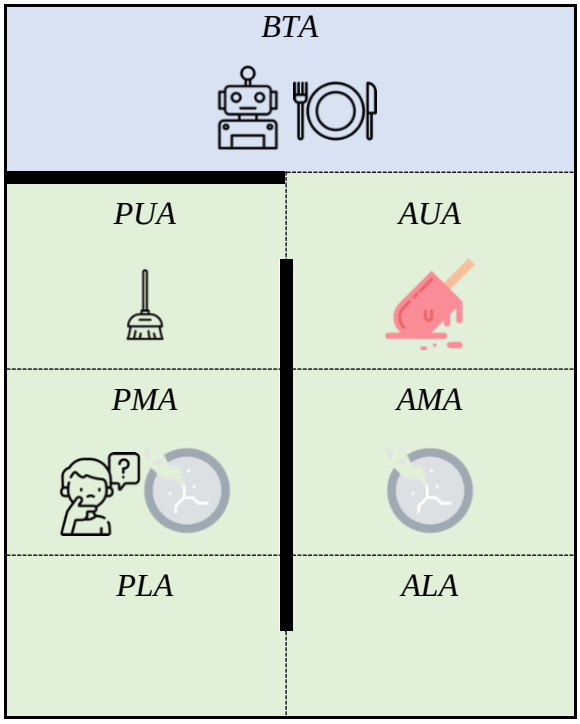
```
(Holding ?a - waiter ?i - item)
```

With this we can determine if the robot is holding any item from the world.
Moreover, we have also updated the move action, since more restrictions have been added. This new move action which considers the new restrictions (such as if there is any broken plate or dropped food in the floor of an area) is the one implemented with the following code:

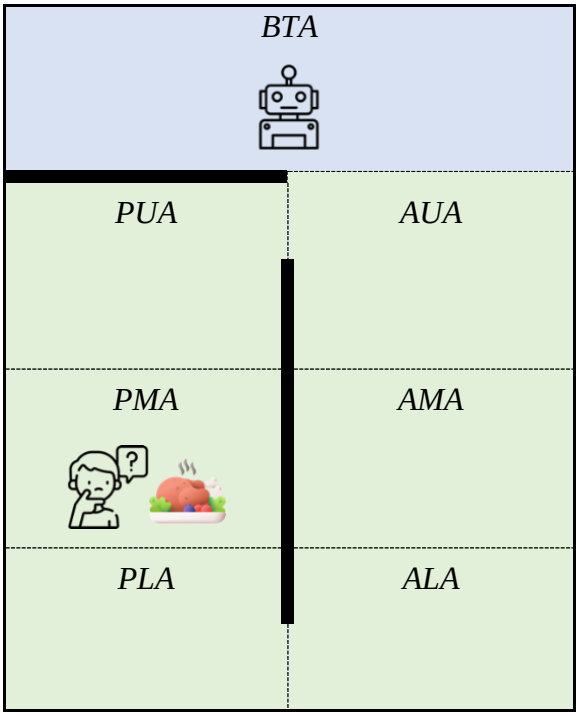
```
(:action move
        :parameters (?x - location ?y - location)
        :precondition (and
            (At Agent ?x)
            (not (At Agent ?y))
            (or (Adjacent ?x ?y) (Adjacent ?y ?x))
            (not (exists (?bp - brokenPlate) (At ?bp ?y))) ; Agent cannot
move if it contains a broken plate)
        :effect (and
            (At Agent ?y)
            (not (At Agent ?x))))
```

Now, for the testing the initial and goal states are going to be:

Initial State



Goal State



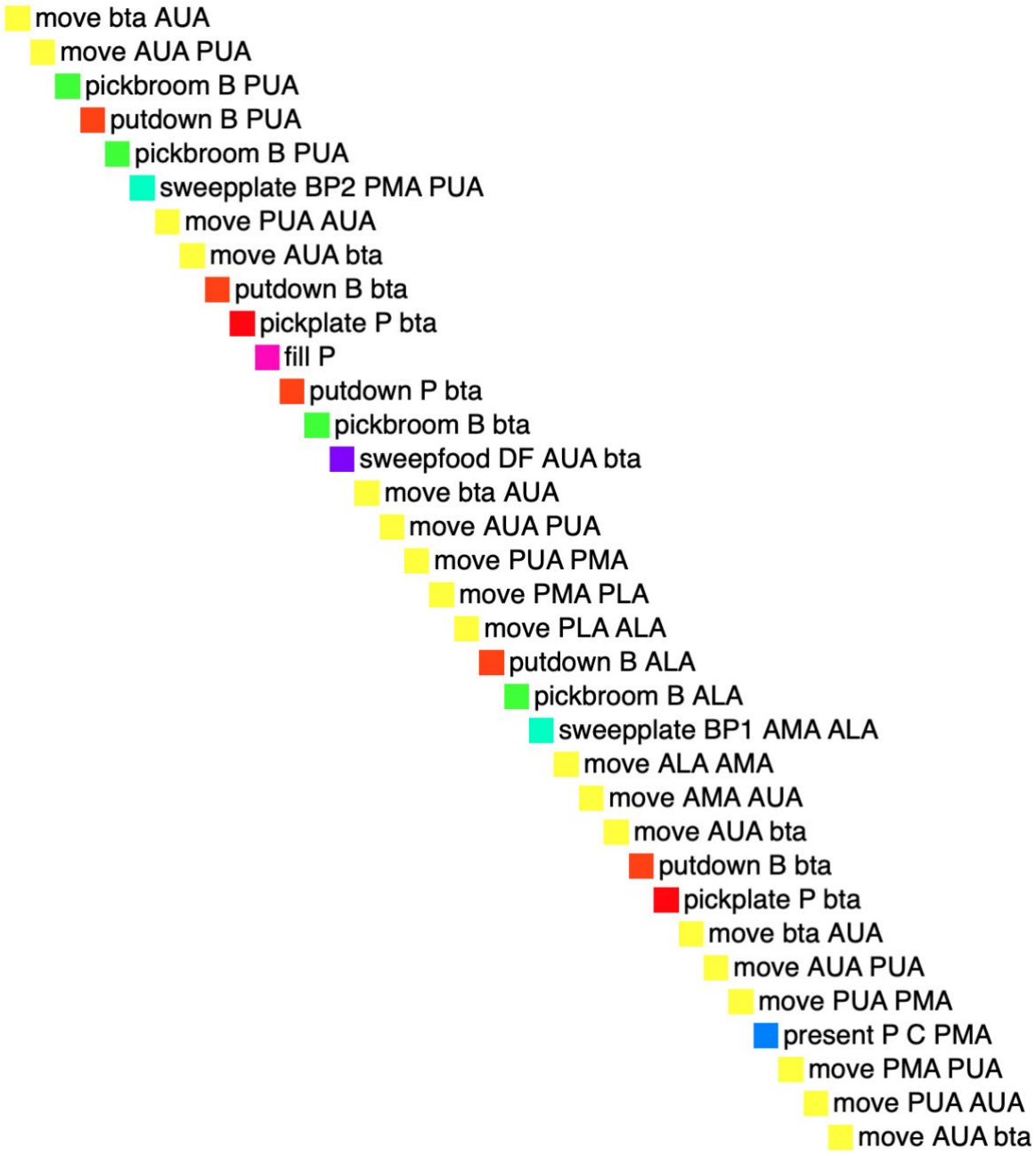
Where we can see two broken plates, one in PMA and one in AMA, some dropped food in AUA, a broom in PUA and the agent and an empty plate in BTA.

The goal is to have all the areas cleaned, that is, without any broken plates or dropped food, as well as the customer served and the robot in the BTA area.

i. Test case metrics

Nodes generated	Nodes expanded	Total time	N° actions
115	43	0.004	34

ii. Plan report



5. Testing cases and results

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 3 where the agent puts down and picks up several times the broom instead of starting to clean the dirty areas from the very beginning when picking up the broom for the first time or the following ones.

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 we thought the most difficult one would be the test case 3 as we implemented more restrictions to the robot and more predicates and actions in the domain, and analyzing the results obtained, we can see that the number of nodes generated and expanded are higher than the ones in the rest of test, as well as the number of actions and the total time. So, our hypothesis has been confirmed, and the test case 3 is actually the most difficult one for the planner to solve. Even so, the plan is found in a short time and without a high number of nodes generated and expanded, which means that even is the most difficult of the three implemented, it is not a very difficult problem for the planner, and it isn't computationally expensive.