ARTIFICIAL INTELLIGENCE AND ROBOTICS

LM IN ARTICIAL INTELLIGENCE

Assignment 4

SUBMISSION 1

Students :
Álvaro ESTEBAN MUÑOZ

Teacher :
Alessandro SAFFIOTTI
Andrea GOVONI

May 10, 2024

Contents

1 Create an HTN planning domain to navigate in our home environment

1.1 The problem

This task involves the understanding of robot task planning by means of using a given HTN planner, namely Pyhop which is an implementation using the python language instead of PDDL or HDDL. We will have to modify the given implementation in order to allow it to generate plans for any target position from any starting position.

1.2 Objects and Data Structures

We are not going to explain the internal structure of the pyhop planner. However, we need to clarify how the domain of the planner is defined. Namely, the domain is define in the module `htn_domain`. Here we can find the definition of the state with all the possible variables. The given definition of the state includes the following state variables in the form of python dictionaries:

- **pos**: indicates the named position of our robot or objects in the environment.
- **room**: denotes room in which is located our robot or an object.
- **connects**: associates a door with the rooms it connects.

These variables will be modified during the execution of the program in order to keep track of the environment's state so to execute our plan.

Notice inside the same model we also define our plan operators and methods, all of these will allow the robot to construct the plan. The operators given are the following:

- GoTo(target): changes the position of the robot from its current one to the argument passed.
- Cross(door): changes the current room of the robot to the one connected to the door passed as argument.

We are also given the following methods:

- move_in_place(target): returns an empty plan when the robot is already at the desired location.
- move_in_room(target): returns a plan to go to the target when the robot is on the same room as the target.
- move_across_rooms(target): returns a plan to go to the target when the robot is in the adjacent room.

We are also given a help function that helps to find all the adjacent rooms to the target one. This will help the method `move_across_rooms` to find the adjacent rooms. As you can notice from the methods defined above there is no way for the robot to reach a target that is further than the adjacent rooms, this is the main part of our task and we show now what we have done in order to solve this problem.

We show the planning tree in figure [1]. Even thought it works at first instance, as we said before it does not allow the robot to reach targets beyond adjacent rooms. To change this we implemented a new version of the `move_across_rooms` method that searches for the path to the target using the BFS algorithm and then moves to the next closer room on the path to the target. Once this is done, we call `navigate_to` so to make this planification recursive. The new tree can be seen in figure [2]

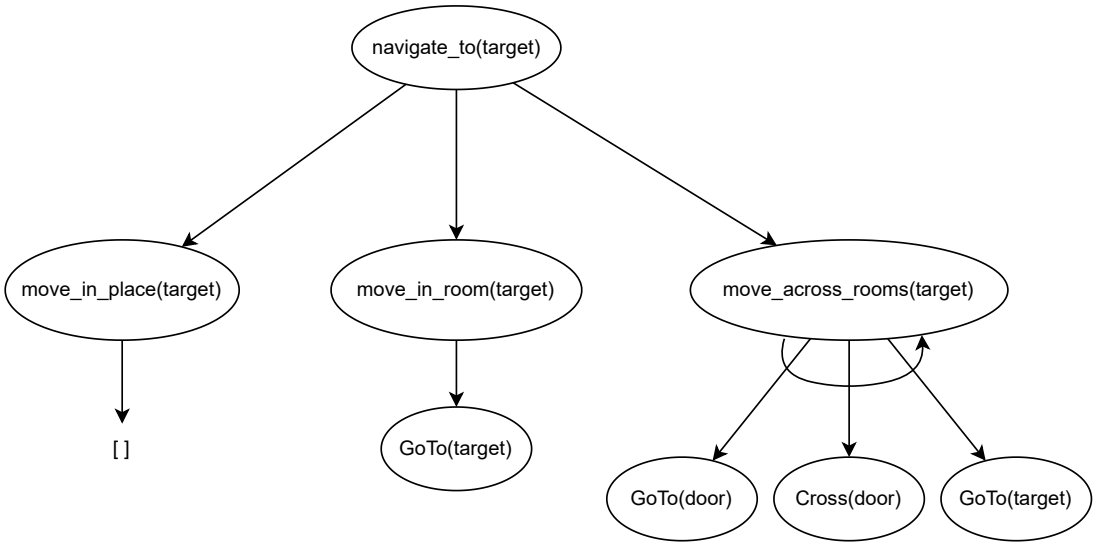

Figure 1: Plan tree

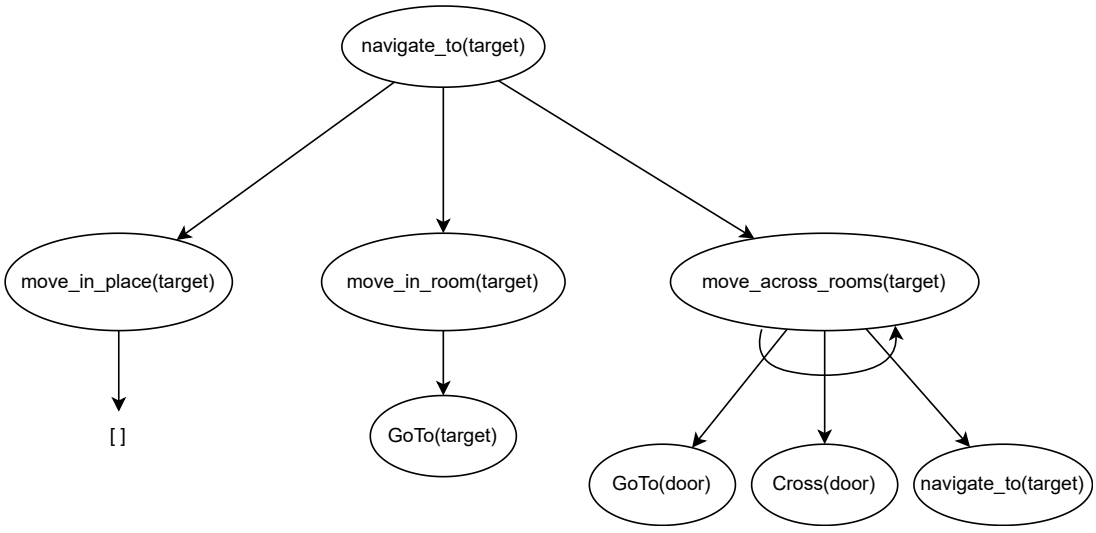

Figure 2: Recursive plan tree

1.3 Results Obtained

With the default implementation of the domain we execute the following plans: `navigate_to(table2` [4], `navigate_to(table3)` [5], `navigate_to(bed1)` [6], `navigate_to(stove)` [7]. If we check the map (figure [3]) we will notice that the last one, `navigate_to(stove)`, is impossible to satisfy since there is no way of reaching a target if it is further than the adjacent rooms.

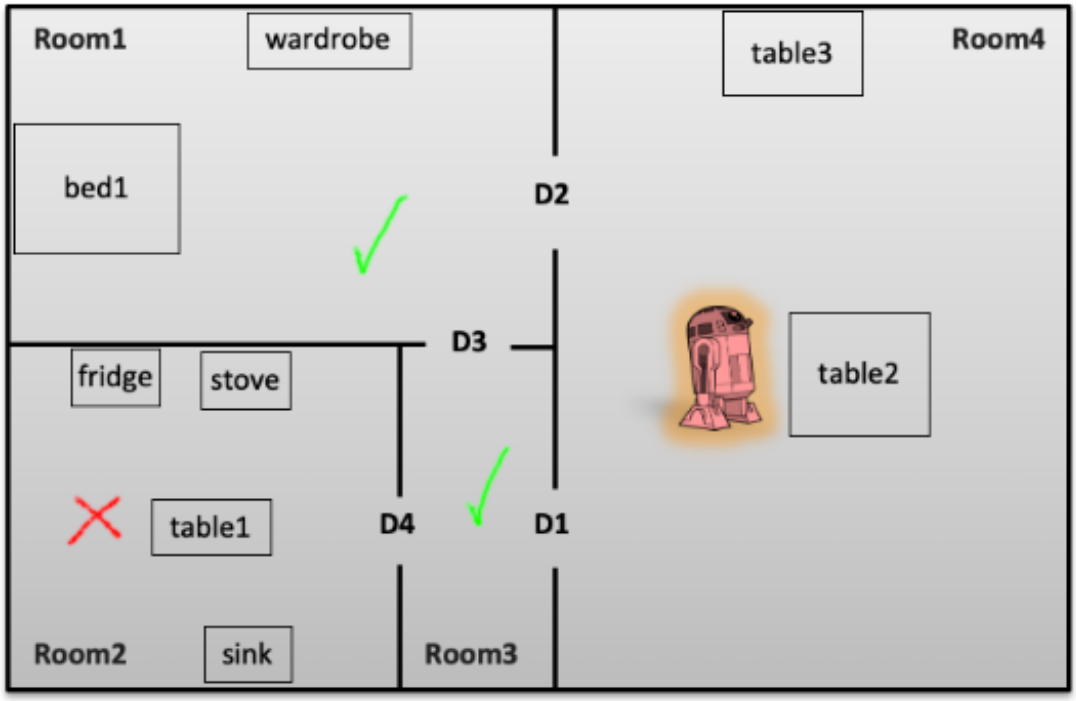

Figure 3: Map of reachable rooms using default implementation

```
Initial state:
    s1.pos = {'me': 'table2'}
    s1.room = {'bed1': 'Room1', 'wardrobe': 'Room1', 'fridge': 'Room2', 'stove':
 'Room2', 'sink': 'Room2', 'table1': 'Room2', 'table2': 'Room4', 'table3': 'Room
4', 'box1': 'Room1', 'box2': 'Room2', 'box3': 'Room3', 'box4': 'Room4', 'me': 'R
oom4'}
    s1.connects = {'D1': ('Room3', 'Room4'), 'D2': ('Room1', 'Room4'), 'D3': ('R
oom1', 'Room3'), 'D4': ('Room2', 'Room3')}
    s1.doors = {'D1': 'Open', 'D2': 'Open', 'D3': 'Open', 'D4': 'Open'}
    s1.holding = {'me': None}
** pyhop, verbose=2: **
   state = s1
   tasks = [('navigate_to', 'table2')]
depth 0 tasks [('navigate_to', 'table2')]
depth 1 tasks []
** result = []


Plan: []
```

Figure 4: Resulting plan for the `navigate_to(table2)` task

```
Initial state:
    s1.pos = {'me': 'table2'}
    s1.room = {'bed1': 'Room1', 'wardrobe': 'Room1', 'fridge': 'Room2', 'stove'
: 'Room2', 'sink': 'Room2', 'table1': 'Room2', 'table2': 'Room4', 'table3': 'Ro
om4', 'box1': 'Room1', 'box2': 'Room2', 'box3': 'Room3', 'box4': 'Room4', 'me':
 'Room4'}
    s1.connects = {'D1': ('Room3', 'Room4'), 'D2': ('Room1', 'Room4'), 'D3': ('
Room1', 'Room3'), 'D4': ('Room2', 'Room3')}
    s1.doors = {'D1': 'Open', 'D2': 'Open', 'D3': 'Open', 'D4': 'Open'}
    s1.holding = {'me': None}
** pyhop, verbose=2: **
   state = s1
   tasks = [('navigate_to', 'table3')]
depth 0 tasks [('navigate_to', 'table3')]
depth 1 tasks [('GoTo', 'table3')]
depth 2 tasks []
** result = [('GoTo', 'table3')]


Plan: [('GoTo', 'table3')]
```

Figure 5: Resulting plan for the `navigate_to(table3)` task

```
    s1.pos = {'me': 'bed1'}
    s1.room = {'bed1': 'Room1', 'wardrobe': 'Room1', 'fridge': 'Room2', 'stove'
: 'Room2', 'sink': 'Room2', 'table1': 'Room2', 'table2': 'Room4', 'table3': 'Ro
om4', 'box1': 'Room1', 'box2': 'Room2', 'box3': 'Room3', 'box4': 'Room4', 'me':
 'Room1'}
    s1.connects = {'D1': ('Room3', 'Room4'), 'D2': ('Room1', 'Room4'), 'D3': ('
Room1', 'Room3'), 'D4': ('Room2', 'Room3')}
    s1.doors = {'D1': 'Open', 'D2': 'Open', 'D3': 'Open', 'D4': 'Open'}
    s1.holding = {'me': None}
depth 4 tasks []
depth 4 returns plan [('GoTo', 'D2'), ('Cross', 'D2'), ('GoTo', 'bed1')]
** result = [('GoTo', 'D2'), ('Cross', 'D2'), ('GoTo', 'bed1')]


Plan: [('GoTo', 'D2'), ('Cross', 'D2'), ('GoTo', 'bed1')]
```

Figure 6: Resulting plan for the `navigate_to(bed1)` task

```
** pyhop, verbose=3: **
   state = s1
   tasks = [('navigate_to', 'stove')]
depth 0 tasks [('navigate_to', 'stove')]
depth 0 method instance ('navigate_to', 'stove')
depth 0 new tasks: False
depth 0 new tasks: False
depth 0 new tasks: False
depth 0 returns failure
** result = False


Plan: False
```

Figure 7: Resulting plan for the `navigate_to(stove)` task

Notice for the new proposed implementation of the `move_across_rooms` method we managed to get a resulting plan (figure [8]) while for the default implementation was not possible (figure [7]).

```
depth 8 returns plan [('GoTo', 'D1'), ('Cross', 'D1'), ('GoTo', 'D4'), ('Cross'
, 'D4'), ('GoTo', 'stove')]
** result = [('GoTo', 'D1'), ('Cross', 'D1'), ('GoTo', 'D4'), ('Cross', 'D4'),
('GoTo', 'stove')]


Plan: [('GoTo', 'D1'), ('Cross', 'D1'), ('GoTo', 'D4'), ('Cross', 'D4'), ('GoTo
', 'stove')]
```

Figure 8: Resulting plan for the new implementation on `navigate_to(stove)` task

1.4 Discussion

As shown on the results on the previous section, we managed to solved the task by generalizing the given methods to reach all the rooms on the full map. This not only solves this task but allows for a possible scaling in case we want to use a completely different map. The robot should be able to find the room in which the target is located and reach that room by recursively applicating the given operators.

2 Extend the previos HTN domain to include the Open and Close actions

2.1 The problem

For this second task we are asked to implement two new operators, `Open(door)` and `Close(door)`. These two new operators will be able to open and close the doors present

in the map and therefore to cross a door we have to make the robot open it first if it is closed.

2.2 Objects and Data structures

Since we have to open and close doors in case they are closed we declare three new methods: `cross_door(door)`, `cross_open_door(door)` and `cross_closed_door(door)`. The first one will be called instead of the `Cross(door)` operator, allowing to branch more our plan into the calling the other two methods depending on the state of the door we want to close. The tree plan is shown on figure [9]

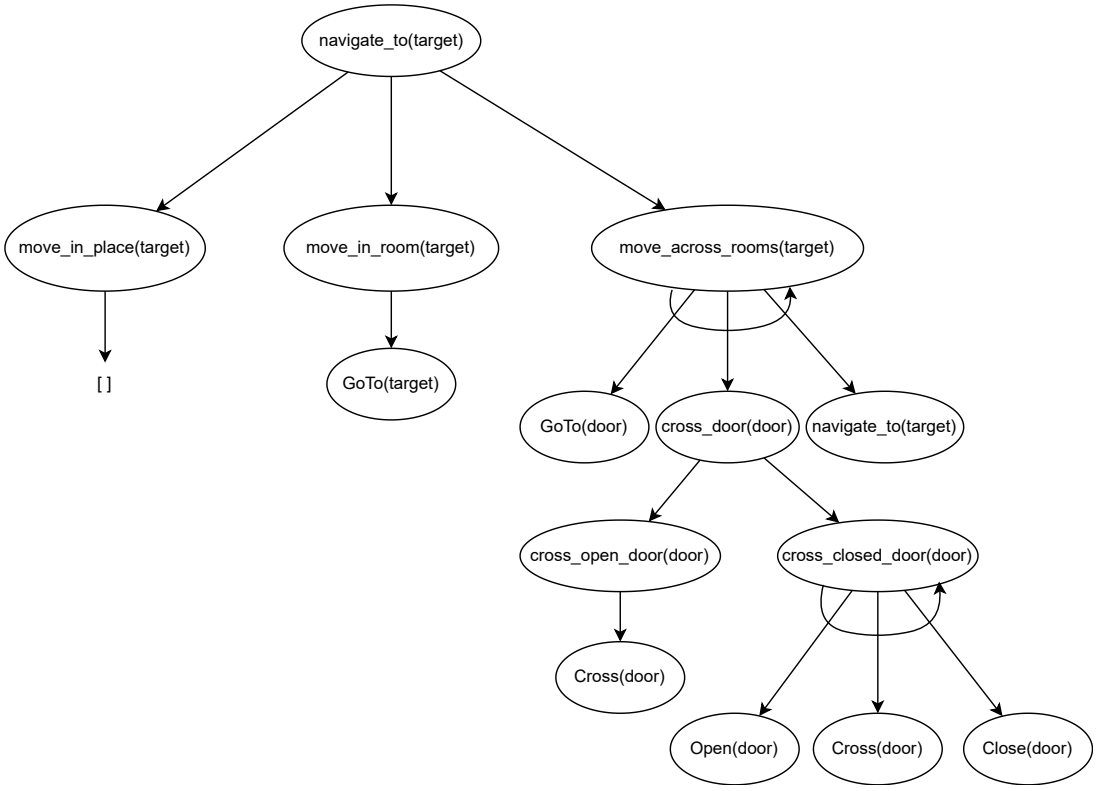

Figure 9: Plan tree with the proposed methods for door opening and closing

2.3 Results Obtained

We execute the same plan as for previous task, `navigate_to(stove)` but this time setting the door `D1` to 'Closed' and check if the plan reaches to create a plan for closing and opening it while just crossing `D4` which remains open.

```
depth 12 tasks []
depth 12 returns plan [('GoTo', 'D1'), ('Open', 'D1'), ('Cross', 'D1'), ('Close
', 'D1'), ('GoTo', 'D4'), ('Cross', 'D4'), ('GoTo', 'stove')]
** result = [('GoTo', 'D1'), ('Open', 'D1'), ('Cross', 'D1'), ('Close', 'D1'),
('GoTo', 'D4'), ('Cross', 'D4'), ('GoTo', 'stove')]


Plan: [('GoTo', 'D1'), ('Open', 'D1'), ('Cross', 'D1'), ('Close', 'D1'), ('GoTo
', 'D4'), ('Cross', 'D4'), ('GoTo', 'stove')]
```

Figure 10: Plan generated for `navigate_to(stove)` after implementing doors

2.4 Discussion

As seen in results section (figure [10]), our planer manages to create a plan where it still goes to stove going through the doors and opening one of them. Manteining the same structure as for the previous task accomplishes with our scaling purpose and therefore we will be able to generate any plan independently of the state of the doors and the shape of the given map.

3 Extend the previous HTN domain to include the PickUp and PutDown actions

3.1 The problem

As for previous problem here we need to define two new operators, this time to allow our robot to `PickUp(box)` and `PutDown(box)` boxes from the environment. Our robot should be able to execute a task like `fetch(box)` which involves going to the target and picking it up.

3.2 Objects and Data structures

Once more, we define our operators, as well as our plan tree (figure [11]). As we can see, the task `fetch(box)` will call a method in charge of returning the `navigate_to(box)` plus the *PickUp(box)* operator on the desired box. This allows the robot to find the box wherever is it and then pick it up.

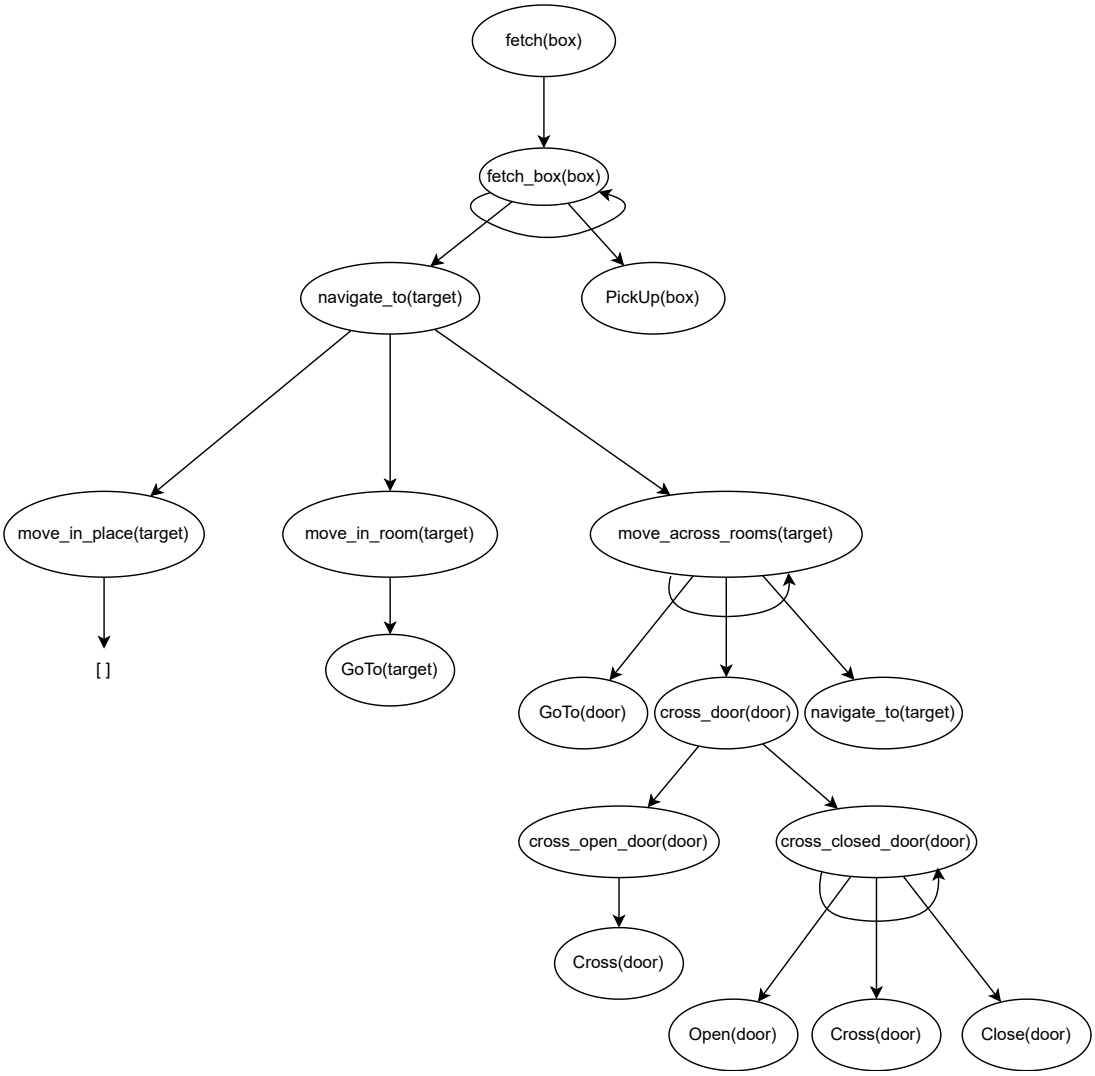

Figure 11: Plan tree for the `fetch(box)` task

3.3 Results obtained

After executing the `fetch(box)` task on `box1`, we obtain the plan show in figure [12]

```
depth 8 tasks []
depth 8 returns plan [('GoTo', 'D2'), ('Cross', 'D2'), ('GoTo', 'box1'), ('PickUp', '
box1')]
** result = [('GoTo', 'D2'), ('Cross', 'D2'), ('GoTo', 'box1'), ('PickUp', 'box1')]


Plan: [('GoTo', 'D2'), ('Cross', 'D2'), ('GoTo', 'box1'), ('PickUp', 'box1')]
```

Figure 12: Generated plan for the `fetch(box)` task on `box1`

3.4 Discussion

Once more, and thanks to previous defined methods and operators, our system is easily scalable to perform the task independent of box's location and doors' state. Notice

we do not need to control if the robot's location is the same as the box since this is a needed condition to return true on the `navigate_to(box)` task.

4 Extend the previous HTN domain to implement a top-level transport task

4.1 The problem

For this task we are required to finish the utility of the `PutDown(box)` operator by implementing a new task that will be in charge of transporting a box from one place to another, we are going to see that our scalable design allows us to make this in quite easy way.

4.2 Objects and Data structures

The only method implementing is the one in charge of sequentially calling the needed tasks for transporting a box to a target. These are `fetch(box)`, `navigate_to(target)` and `PutDown(box)`. We can see the implemented plan tree on figure [13]

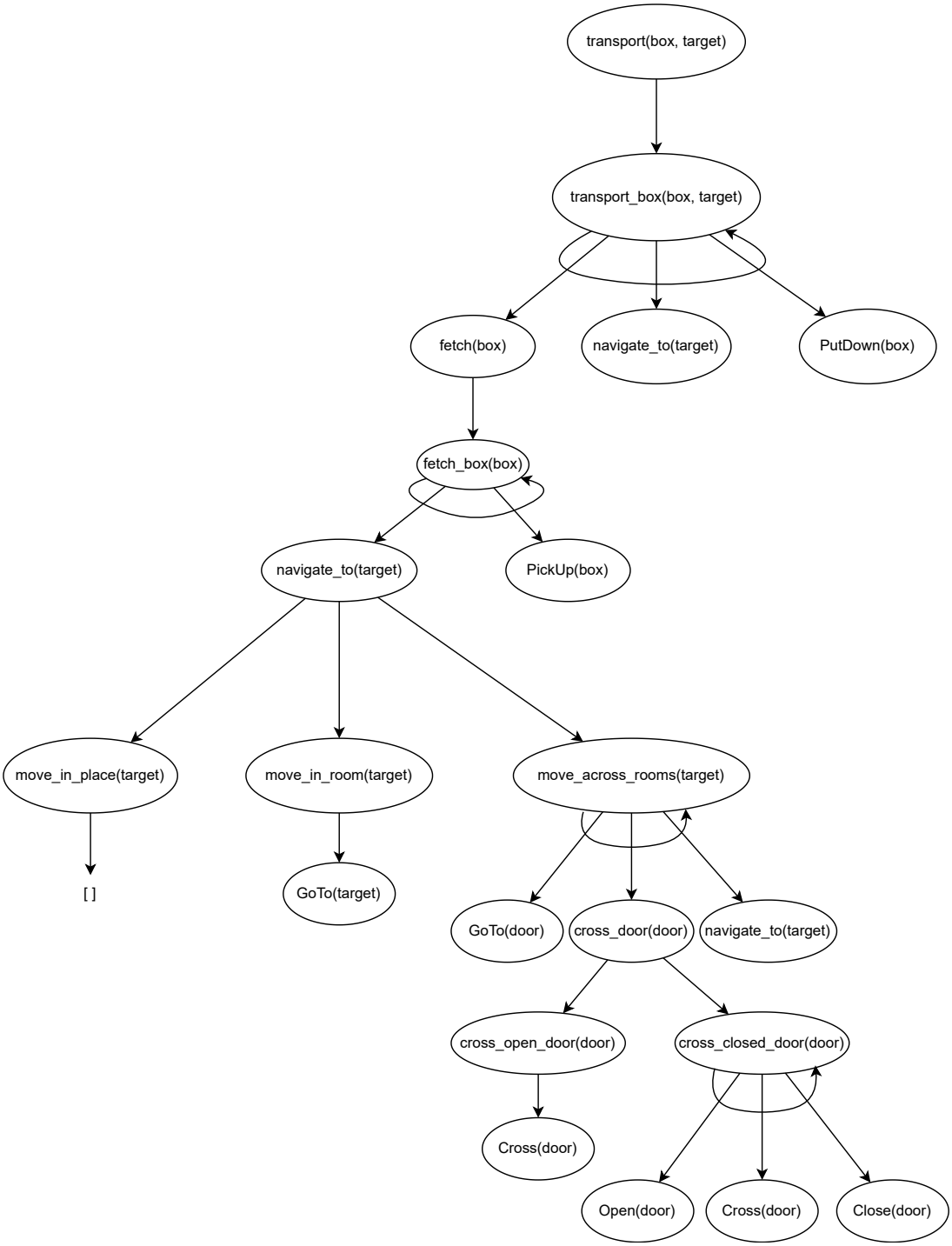

Figure 13: Plan tree for the `transport(box, target)` task.

4.3 Results obtained

The execution of the `transport(box, target)` on box1 and stove as target gave the plan show on figure [14]

```
depth 20 returns plan [('GoTo', 'D2'), ('Cross', 'D2'), ('GoTo', 'box1'), ('PickUp',
'box1'), ('GoTo', 'D3'), ('Cross', 'D3'), ('GoTo', 'D4'), ('Cross', 'D4'), ('GoTo', '
stove'), ('PutDown', 'box1')]
** result = [('GoTo', 'D2'), ('Cross', 'D2'), ('GoTo', 'box1'), ('PickUp', 'box1'), (
'GoTo', 'D3'), ('Cross', 'D3'), ('GoTo', 'D4'), ('Cross', 'D4'), ('GoTo', 'stove'), (
'PutDown', 'box1')]


Plan: [('GoTo', 'D2'), ('Cross', 'D2'), ('GoTo', 'box1'), ('PickUp', 'box1'), ('GoTo'
, 'D3'), ('Cross', 'D3'), ('GoTo', 'D4'), ('Cross', 'D4'), ('GoTo', 'stove'), ('PutDo
wn', 'box1')]
```

Figure 14: Generated plan for `transport(box, target)` on box1 and stove as target

4.4 Discussion

Given the results of the section above we can confirm our methods and operators are not only working good but scaling in a good manner. Every time we want to add operators or tasks that involve in some way the others we can do like we have performed until now, allowing to easily modify the domain of the planner.

5 Make your (simulated) robot execute navigation plans

5.1 The problem

To finish this assignment we are asked to make the robot actually execute navigation plans previously implemented in the given tasks. For simplicity we do not deal with doors and boxes, meaning our unique objective is to make the robot execute the `GoTo(target)` and `Cross(door)` operators.

5.2 Objects and Data structures

We have seen how behaviors are implemented on the previous assignment. Since we already have implemented a `GoToTarget` behavior we will assign it to the `GoTo(target)` operator. Hence, we are left with the implementation of a behavior that will be associated to the `Cross(door)` operator.

Our new behavior will be also called `Cross` and for it we need to define the fuzzy predicates, variables and rules. First of all, the state is defined with just one variable, `calibration` that will be in charge of measuring how centered the robot is with respect to the door. To measure this, we take data from two sonars on each side at 30°, 60°, 330° and 300°, then we take the minimum value on each side and substract them.

$$\text{calibration} = \text{minRSonar} - \text{minLSonar} \tag{5.1}$$

Where `minRSonar` denotes the minimum value of the two sonars on the right side of the robot (300°, 330°) and `minLSonar` denotes the minimum value of the two sonars on the left side of the robot (30°, 60°). This means that if calibration < 0 the robot is uncalibrated to the right and we need to turn left and vice versa.

Our fuzzy predicates are defined as follows:

$$\text{CenterLeft}(\text{calibration}, -1.0, -0.2) \begin{cases} 1 & \text{calibration} < -1.0 \\ 0 & \text{calibration} > -0.2 \\ \frac{-0.2-\text{calibration}}{-0.2+1.0} & \text{otherwise} \end{cases}$$

$$\text{CenterRight}(\text{calibration}, 0.2, 1.0) \begin{cases} 0 & \text{calibration} < 0.2 \\ 1 & \text{calibration} > 1.0 \\ \frac{\text{calibration}-0.2}{1.0-0.2} & \text{otherwise} \end{cases}$$

$$\text{Centered}(\text{calibration}, -0.3, 0.0, 0.3) \begin{cases} 0 & \text{calibration} < -0.3 \vee \text{calibration} > 0.3 \\ \frac{\text{calibration}--0.3}{0.0+0.3} & \text{calibration} > -0.3 \wedge \text{calibration} < 0.0 \\ \frac{0.3-\text{calibration}}{0.3-0.0} & \text{otherwise} \end{cases}$$

Fuzzy variables are kept as for the previous assignment while fuzzy rules are defined as follows:

- Centered $\implies$ Move(Slow)
- CenteredLeft $\implies$ Turn(MLeft)
- CenteredRight $\implies$ Turn(MRight)

5.3 Results obtained

We execute the task `navigate_to(stove)` and the results seem to be quite accurate. The trajectory followed by the robot can be seen on figure [15]. The robot can be seen going to the stove on figure [16].

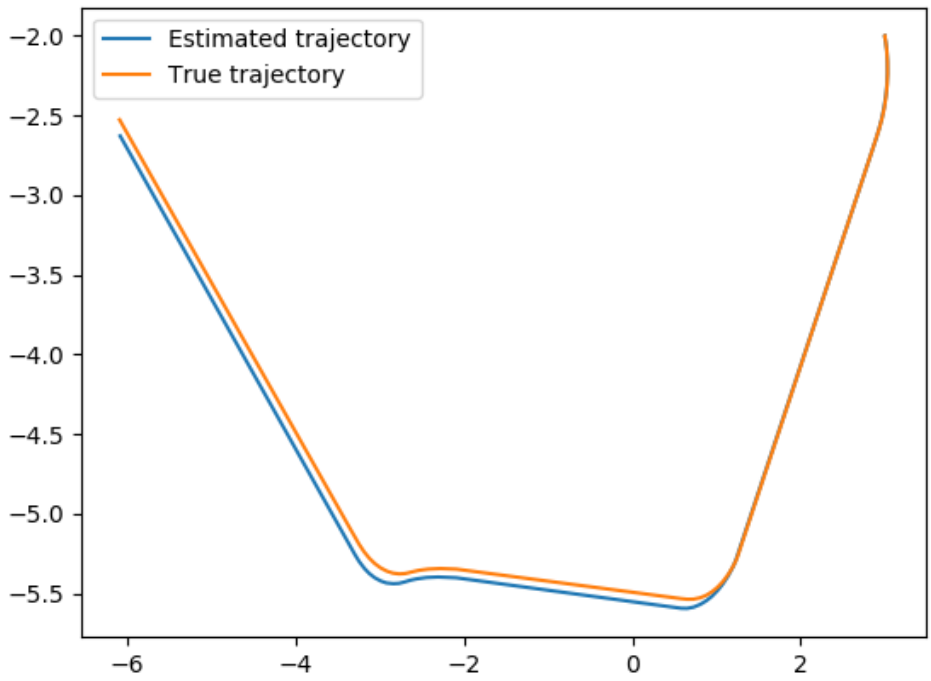

Figure 15: Trajectory followed by the robot for the task `navigate_to(stove)`

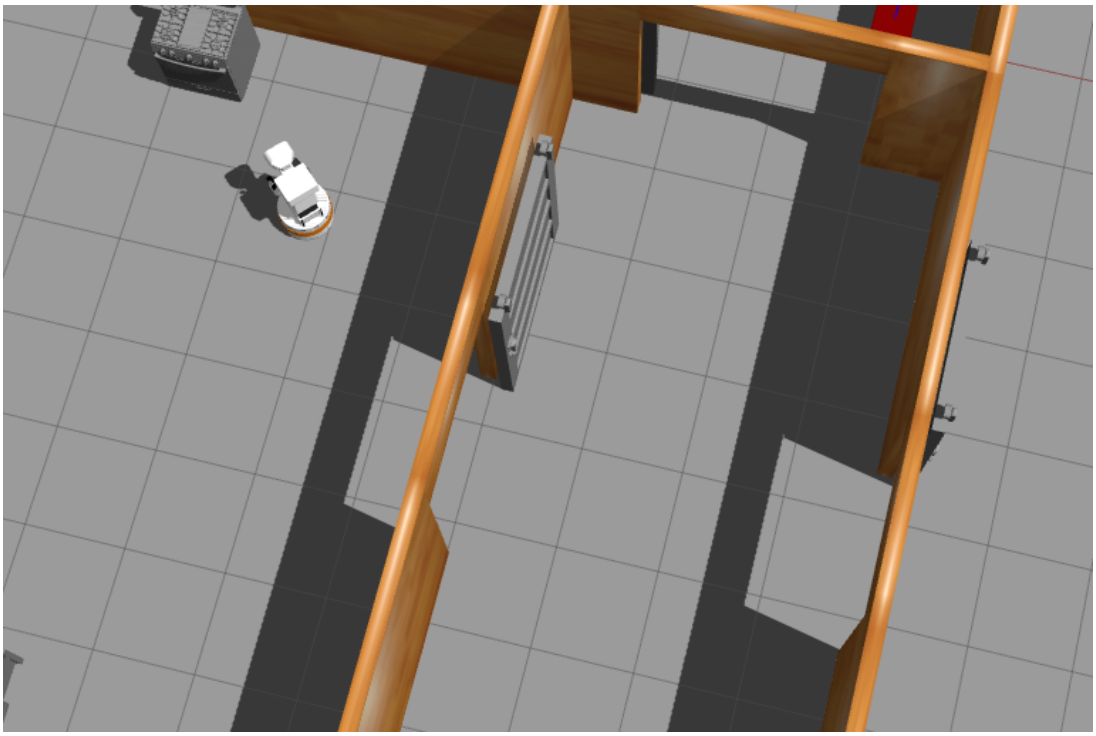

Figure 16: Robot going to its target (stove)

5.4 Discussion

During the whole assignment we have seem planification done correctly can be easily scalable, this added to the implementation of behaviors associated to operators makes the way in which we program a robot much simpler and robust. We do not define each step that needs to be followed by the robot, we just implement behaviors and operators and the robot will robustly act in consequence. Our robot is able to generate a plan and to execute it succesfully.