Exercise Sheet C

Eduard Alcobé, Aakash Maroti, Aaron Verdaguer

Exercise C.1

For this exercise you need to submit one paragraph describing your model, plus the actual pddl domain and instance files.

Monkey and Banana are the constants in the model. Bananas are treated as a single entity here for the sake of simplicity. The monkey can do basic atomic actions like going to a spot, climbing the chair, pushing the chair, grabbing bananas, eating bananas and stepping down from the chair. The predicates include these actions and simple checks like, is the monkey on the floor? Is the monkey at a given location? Does the monkey have bananas? Has the monkey eaten the bananas? And is the monkey on the chair? Each action has preconditions based on the state, and then the states are updated as a result of the action.

Using the above framework we formulate our instance file for the solution. There are a total of 6 possible locations. In the initial state the monkey is at P1 and on the floor. The chair is at P6 and bananas are at P3. The goal state is when Monkey has eaten the bananas.

The domain file for this problem statement. The instance file for this problem statement.

Exercise C.2

Part A

Submit a brief description of the PDDL predicates, objects, initial state, actions and goal of your planning encoding.

General Description: We discuss below all the components of the planning encoding.

- Predicates:
    - **Adjacent** : This denotes when 2 objects are directly adjacent to each other. For example, position v4-v3 is adjacent to v5-v3, v3-v3, v4-v4 and v4-v2.
    - **Adjacent_2** : This is for cases when 2 objects have exactly 1 intermediate step in between them. For example, position v4-v3 is adjacent_2 to v6-v3, v2-v3, v4-v5 and v4-v1.
    - **Use Teleport** : This check makes sure that teleport is used only once.
- Objects:
    - **Player** : The main agent taking all the actions.
    - **Box** : The boxes that need to be moved around are also considered as an object here.
    - **Wall** : The walls limit the movement of the agent and the boxes. They are treated as an object in this formulation.

- Initial State: The initial state is taken as input from the Sokoban problem. The use teleport state is set to false to allow for the use of teleportation in the problem just once.
- Actions:
    - **Move player** : The player is moved from location x to location y, if and only if the player is already in state x, and location y is neither a box nor a wall.
    - **Push box** : Given that the player is at location x and the box is at location y, the player is moved to location y and the box is moved to location z, if and only if, location z is not already a box or a wall, and location y is adjacent to location x and z, and that x and z have exactly 1 intermediate step in between them.
    - **Teleport** : Similar to moving a player, however the location x and y being adjacent to each other is no longer a condition here. This action can only be used if it has not been used yet, once used the check is updated to reflect that it has been used.
- Goal: The goal state is when all the boxes are moved to the storage location.

Part B

For this section you need to submit the source code of your program, along with any static files (such as the domain PDDL file mentioned above) that are necessary to solve arbitrary Sokoban instances.

All the code used in this project can be found here:
https://github.com/aakash94/AutonomousSystems/tree/main/Lab3

The command used to run the code is :
```
python3 sokoban.py -i benchmarks/sasquatch/level1.sok
```

We have 2 approaches here.

1. This is the code presented where every position has been described with just one variable and there are only three possible actions: move-player, push-box and teleport. As it has been mentioned, this is the approach represented in the submitted files, so all results presented in this report have been found following this approach. The files associated with this approach are:
    - sokoban.py
    - domain.pddl
2. In this other approach every position is described with two variables (coordinates x and y), moreover the four possible directions have been distinguished so there are 9 actions (move-player-right, move-player-left,…, push-box-right, …, and teleport. This approach is more time consuming probably because every position is described with two variables. For this reason we haven't worked with it. The files associated with with this approach are:
    - sokoban2
    - domain2.pddl.

Part C

How many of the problems can you solve in each case? How much do the number of steps in the non-optimal solutions differ from the optimal ones?

The command used to run the code is :

```
python3 sokoban.py -i all
```

The table below shows the cost taken by optimal solver and satisfiable solver at different levels. The levels not mentioned in the table were not solved within the 60s. The optimal solver did not find any solution for level 8 and 9 within the time limit. The optimal solver used here is *seq-opt-bjolp* and the satisfiable solver here is *seq-sat-lama-2011.*

Level	Optimal	Satisfiable	Difference
1	103 unit cost	107 unit cost	4 unit cost
2	34 unit cost	42 unit cost	8 unit cost
8		170 unit cost	
9		187 unit cost	
14	45 unit cost	45 unit cost	0 unit cost

Are the Sasquatch problems too easy or too difficult for your compilation-to-PDDL approach?

The problems can be considered difficult for the approach taken, because only 3 out of 50 given problems were solved optimally within the given time limit.

If they are too difficult, what could make them easier to solve?

Increasing the time limit can help us explore more possible solutions. However given the nature of the game, the problem remains a challenging one.
Additionally reducing the complexity of the domain file can also help us find more solutions. We tried 2 different domains, one with a single variable and one with multiple variables depicting the position. We noticed that multiple variables in the domain file significantly slows down the process.