Linear Planner

Planning and Approximate Reasoning

Universitat Rovira i Virgili

January 9, 2016

Authors:
Kazancli, Erol
Mahyou, Khalid

Contents

1 Problem Description

We have a square building which is composed of 9 offices arranged in a 3 by 3 grid. Some of the offices are dirty, some contain boxes. We have a robot, whose duty is to clean all the offices and move the boxes to their final positions. We have a configuration file containing the information of the initial and the goal state of the building. Initial state includes initial location of the robot, the names of the dirty offices and the location of the boxes. Goal state includes the final positions of the boxes and implicitly all rooms cleaned. There might be at most 8 boxes in the building. The aim is to move the robot from the initial to the goal state efficiently using linear planner.

2 Linear Planner Design

The robot has three actions at its disposal, which it can perform provided the preconditions are met. After these actions are carried out some changes take place in the building state, indicated by the add and delete conditions.

- Clean-office(o): the robot cleans office o.

  Preconditions: robot-location(o), dirty(o), empty(o)

  Add: clean(o)

  Delete: dirty(o)

- Move(o1,o2): the robot moves from o1 to o2.

  Preconditions: robot-location(o1), adjacent(o1,o2)

  Add: robot-location(o2)

  Delete: robot-location(o1)

- Push(o1,o2, b): the robot pushes box b from o1 to o2

  Preconditions: robot-location(o1), box-location(b,o1), adjacent(o1,o2),empty(o2)

  Add: box-location(b,o2), robot-location(o2), empty(o1)

  Delete: empty(o2), box-location(b,o1), robot-location(o1)

To approach the problem we created a class named Building with the following attributes:

```
1 public class Building {
2
3     List<String> boxes = new ArrayList<String>();
4     private List<String> offices = new ArrayList<String>();
5     String robotLocation = "";
6     List<String> dirtyLocations = new ArrayList<String>();
7     Hashtable boxLocations = new Hashtable();
8     String goalRobotLocation = "";
9     Hashtable goalBoxLocations = new Hashtable();
```

We read the initial configuration file and store the values in these attributes. As can be seen above we do not keep empty locations because it can be inferred using boxLocations. We do not keep clean locations either, because it can be inferred from dirtyLocations. Therefore the initial file should not contain Clean or Empty predicates or they will simply be ignored.

To check the fulfilment of the preconditions of a certain action we created a method called checkPrec.

```
1
2  public boolean checkPrec(String action, String arg1, String arg2, String arg3)
3  {
4      if (action.equals("Clean-office")) // arg1 = o1
5      {
6      ...
7      }
8      else if (action.equals("Move")) // arg1 = o1; arg2 = o2
9      {
10     ...
11     }
12     else if (action.equals("Push")) // arg1 = o1; arg2 = o2; arg3 = box
13     {
14     ...
15     }
16 }
```

This method returns true if the preconditions for a given actions are satisfied, using the given action name and the relevant arguments. An example usage would be:
checkPrec(”Move”, ”o3”, ”o4”, ”B”), which would return false because o3 and o4 are not adjacent.

To perform actions after the preconditions are checked, we created a method named performAction.

```
1
2  public void performAction(String action, String arg1, String arg2, String arg3)
3  {
4      if (action.equals("Clean-office")) // arg1 = o1
5      {
6      ...
7      }
8      else if (action.equals("Move")) // arg1 = o1; arg2 = o2
9      {
10     ...
11     }
12     else if (action.equals("Push")) // arg1 = o1; arg2 = o2; arg3 = box
13     {
14     ...
15     }
16 }
```

This method changes the state of the building adding some conditions and deleting others related to the corresponding action with the parameters. An example usage would be:
performAction(”Clean-Office”, ”o5”, ””, ””), which would delete the office o5 from the dirtyLocations list (implicitly adding also the statement of clean(o5))

For the sake of convenience, we also created two very important methods to avoid code repetition: **moveLongDistance(String destination)** and **pushLongDistance(String boxName, String destination)** which perform ”move” and ”push” actions respectively in succession and are used quite frequently since sometimes we repeat the actions ”move” or ”push” until we reach a certain location. These methods include the precondition checks in themselves.

Other important functions used are:

- isAdjacent(off1, off2): This function check if two offices are adjacent
- getDistance(off1, off2): This function gets the distance between two offices

- findXAdjacent(off, x): Returns the adjacent to the right or to the left in the horizontal level. The direction is given by the sign of x.
- findYAdjacent(off, y): Returns the adjacent in the horizontal level. The direction is given by the sign of y.
- findAppropriateBox(String arg1): This function finds an appropriate box to be moved to its goal location which is movable in the goal direction. It gives priority to the boxes with goal positions in the corner.

3 Heuristics Used

Our algorithm is based on a bringing the map closer to its goal state with every new iteration. We fulfil this condition in most of the situations except for deadlocks or sometimes clearing our ways.

We start by selecting a suitable box to be moved. In our trials we realized that if we moved first the boxes whose goal positions are corner offices, we get better results. This is intuitive because corner positions can be easily blocked if the two adjacent offices are occupied. Therefore we give priority to these boxes. The middle office, which is o5 has the least priority since it is in a very key position and should not be blocked permanently. We also take into account the movability of the box, specifically in the direction of its goal. Once we select a suitable box we move to that location. Once arrived, we calculate the horizontal and vertical distances to the goal position and start to push the box in these directions. With every move we bring the box closer to its goal position. If at some point both of these directions towards the goal are occupied, we check if either of the boxes blocking the road can be moved in the opposite directions. If yes, we move them in the opposite directions and continue our way to the goal. This has the probability of distancing the other box away from its goal and is a violation of our heuristic of every time equal or better states but this is affordable, because we gain more than we lose. If none of the blocking boxes can be moved, we stop and start looking for another suitable box. So far we have brought one box to its goal position or at least closer to its goal position and most probably in other steps the way for it will be cleared, as we move other boxes to their goal locations. We keep performing this step, i.e. choosing an appropriate box and moving it as close as possible to its goal, until we moved all boxes to their goal positions or until we can not find a movable (in its goal direction) box. If we can no longer find a suitable box, this means we have reached a local minimum and we need to move to another locality of states to find hopefully the global minimum of distance(between states) which is zero. So we move all the boxes, who has an empty adjacent office in the given direction, randomly in one direction. This does not look like a clever heuristic but we tried other more clever sounding heuristics like clearing a path for one box, or moving only the boxes that has not reached their goals, however moving randomly worked better in our tests. To avoid cycles, each time we changed the direction we moved the boxes. After we move the boxes randomly, we start selecting a suitable candidate again and keep iterating until the goal is reached. If we can escape the locality, we sometimes achieve to reach the goal, if not, an unbreakable cycle starts which means failure. In this case we leave with the latest local minimum found. Here a better heuristic could help break the deadlock better.

Another heuristic we used is as we moved or pushed a box, we clean the offices we are passing if they are dirty. If we are pushing a box and there is a dirty office one step ahead, we first go to that location and clean before we push the box, since the offices cannot be cleaned when there is a box in them. This extra step may be superfluous in cases where there are very few boxes since cleaning while only moving might be enough. These heuristics for cleaning worked quite well because while moving we cover the distances many times and after pushing all the boxes to their goal positions we rarely have a dirty office left. After all the boxes are moved or a fixed number of iterations, we stop pushing and check if any dirty office is left. We clean if any dirty office is left. As a last step we move to the final position.

4 Testing

Take an example of this initial state:

```
Boxes=A,B,C,D,E,F

Offices=o1,o2,o3,o4,o5,o6,o7,o8,o9

InitialState=Dirty(o1); Dirty(o5); Dirty(o8); Dirty(o9); Dirty(o6); Dirty(o2); ...
    Dirty(o4);

Robot-location(o4); Box-location(A,o4); Box-location(B,o6); Box-location(C,o5); ...
    Box-location(D,o1); Box-location(E,o9); Box-location(F,o3);

GoalState = Robot-location(o9); Box-location(A,o3); Box-location(D,o4); ...
    Box-location(B,o7); Box-location(C,o1); Box-location(E,o2); Box-location(F,o5);
```

The resulting plan starts like this:

```
Move(o4,o5)
Move(o5,o2)
Clean-office(o2)
Move(o2,o5)
Push(o5,o2,C)
Move(o2,o1)
Move(o1,o4)
...
```

We can see here that the first box chosen is C which has a goal in the corner position which is compatible with our heuristics. It is pushed towards its goal destination. As can be seen along the way one of the dirty offices is cleaned. Note that the box C is left at o2 before it arrives its destination because the destination is already occupied by D.

And continues as...

```
Move(o4,o5)
Clean-office(o5)
Move(o5,o4)
Push(o4,o5,A)
Move(o5,o4)
Clean-office(o4)
Move(o4,o1)
```

Here we can see that before the box is pushed to the office o5, the room is cleaned first and than the box is pushed. This is because of the fact that a room cannot be cleaned when it is occupied by a box. We can also see that the box A is left on o5 because o2 and o6 occupied for it to reach o3, and these two boxes cannot be moved because the neighbouring offices are also occupied(o5, o1 and o9).

For further details about this example please check the log files.

Take another example with 7 boxes:

```
Boxes=A,B,C,D,E,F,G
Offices=o1,o2,o3,o4,o5,o6,o7,o8,o9
InitialState = Dirty(o1); Dirty(o5); Dirty(o8); Dirty(o9); Dirty(o6); Dirty(o2); ...
    Dirty(o4);
Robot-location(o2); Box-location(A,o7); Box-location(B,o8); Box-location(C,o9); ...
    Box-location(D,o1); Box-location(E,o3); Box-location(F,o2); Box-location(G,o4);
GoalState = Robot-location(o5); Box-location(A,o4); Box-location(D,o2); ...
    Box-location(G,o3); Box-location(B,o7); Box-location(C,o1); ...
    Box-location(E,o8); Box-location(F,o9);
```

```
...
Move(o7,o4)
Push(o4,o5,A)
Push(o5,o4,A)
Move(o4,o5)
Move(o5,o2)
Push(o2,o5,D)
Push(o5,o2,D)
Move(o2,o3)
Move(o3,o6)
Push(o6,o5,G)
Move(o5,o2)
Push(o2,o1,D)
Move(o1,o2)
Move(o2,o5)
Push(o5,o6,G)
Move(o6,o3)
Push(o3,o2,E)
Move(o2,o3)
Move(o3,o6)
Push(o6,o3,G)
...
```

At this part we clearly see several attempts to break a deadlock by our random moving algorithm. Each time a different attempt is undertaken to escape the local minimum by choosing a different box. In the first two attempts we end with the same local minimum. However, this method works in the end with G being moved(in the opposite direction of its goal). We can see that later it is being moved again to o6 and finally to o3, its final destination, which has been vacated by E. Here we see a positive example and this case arrives at its goal in the end. However there are other examples with 7 and 8 boxes where sometimes this heuristic is not enough to break the cycle. You can find negative examples in the log files.

5 Results

Our algorithm and heuristics led to the goal for up to 6 boxes very well. In our tests we have not found a negative result up to(including) 6 boxes. For 7 boxes, we reached our goal in a few cases. For the rest of the 7-box cases and 8-box cases we failed to reach the goal since we came across deadlocks and cycles. Our random moving heuristic worked in some deadlock cases but not for all deadlocks or cycles. We could have come up with a better heuristic for these deadlock positions, moving in a more clever way than randomly but our trials resulted worse in these efforts. Even though our algorithm performed quite well in finding the goal state in most of the cases, we do not think the found solution is the optimum plan, although a very close one. We could use better heuristics in choosing the suitable boxes to be moved, maybe also taking the distances into account. Also solutions to overcome deadlocks might help to find an optimum solution. As for cleaning the dirty offices, we did not have any difficulty with cleaning all the offices even if all the offices are dirty initially.

6 Running the code

In the project provided in a zip file, after extracting in a folder and opening in Eclipse, simply run PlannerFrame.java, which contains the main method. A GUI will pop up, where with the "Choose File" button, the initial configuration file is selected. With the "Run Planner" button, the planner is started and the output files in the same location as the initial configuration file are created. There are three types of output files: Goal, Plan and States. A typical input file name is

testingPos7boxes which has 7 boxes with a positive result with our algorithm. The results can be found in the corresponding output files. We added 4 positive 2 negative examples and the outputs. If a new file is to be tested, we recommend to use one of our example files and change the numbers or add new states if necessary. Please note that we do not use empty or clean predicates, because they are implicit in our implementation. We also added to the zip file some of our test files and the results.