COL864 Assignment2

Gobind Singh 2016PH10567

April 2020

1 Domain Representation

The domain consists of objects (mentioned in environment.py), and actions are parametrized on these objects. The objects are classified into categories - canContain (objects which can contain other objects), canSupport (objects which act as surfaces for placing other objects) and canPlaced (objects which can be placed on or in other objects)

The state is represented as nested dictionary: state['grabbed'] - object currently grabbed by the robot
state['fridge'] - fridge state in (Open/Close)
state['cupboard'] - cupboard state in (Open/Close)
state['inside'] - Contains dictionary with objects in canContain as keys with the object inside them in a set as values
state['on'] - Contains dictionary with objects in canContain as keys with the object inside them in a set as values
state['close'] - list of objects close to the robot

The goal is represented as a set of predicates, with three types of predicates - *is, in* and *on*. *Is* predicates specify the state of an object (e.g fridge is closed), *in* predicates specify object a in object b and *on* predicates specify object a on object b.

2 Assumptions

These are some assumptions I have inferred from various interactions with the simulator. Sometimes the assumption might not hold due to randomness of the simulator. However from a semantic level, they should always hold.

- If we come close to an object, we are then close to all objects inside it and objects which contain the original object.
- If we pick and drop an object, we still remain close to it.
- We can pick any object from anywhere as long we are close to it. For eg, if apple is in box, we can move to apple and pick it. For cupboard and fridges, we just need to keep them open.
- Cupboard and fridges have space for objects. For instance if I place apple, banana and orange in fridge one by one, we will be able to do it.

3 Forward Planning

- I have implemented checkAction and changeState functions based on the precondition and effects of actions in the domain.
- I use Depth First Search to search for the plan, using the checkAction and changeState functions
- At each state I generate the applicable actions, and choose one to branch off using a heuristic.
- For generating applicable actions, I used various levels of elimination to remove irrelevant actions. There are three levels of elimination - no elimination, basic elimination (only considers actions containing objects which are relevant to the goal) and aggressive elimination.
- For aggressive elimination, I construct a new action template - *transfer*. TransferAtoB action is a placeholder for a sequence of actions : moveToA,pickA,moveToB,dropB. The aggressive elimination only considers transfer actions, and selects them based on a heuristic.
- The heuristic is derived from a DAG representation of the goal state. If we remove the goal predicates asking for fridge (and cupboard) to be open and close, the goal is just a DAG. Each object has a parent to which it should be assigned (either put it in or on its parent)

- I construct the DAG from the goal, and prioritize those transfers which are between direct children and parents. For eg say the goal is to place apple in tray and tray in fridge. In the DAG the parent of apple will be tray, and the parent of tray will be fridge. The solution of this setting is to have apple on a tray in a fridge. I want to prioritise transferring apple to tray , and tray to fridge rather than directly placing apple in fridge. Also I prioritize actions which start from lower levels. So placing apple in tray will be prioritized over placing tray in fridge.
- If goal-relevant objects are in fridge and cupboard, I use pattern-databases idea. I just add actions for going and opening the fridge and cupboard. This allows all kinds of transfer. As now I can pick any object, I just need to search for various transfers to solve the problem. I run the DFS search, and end when all the *in* and *on* relations are satisfied. Finally I see whether fridge is to be left close or open in the goal, and add that to my plan.
- For ensuring completeness (DFS is not complete), I have added a max depth constraint for the search. From the DAG one infer the upper bound on the number of transfers required to satisfy the goal (it is equal to the number of edges). Thus my algorithm is complete.
- The branching factor when I dont eliminate actions ranges from 20 - 60. After elimination it comes down to about 3-20 depending on the complexity of the goal. When I only consider aggressive elimination, the branching factor is very low (1-5) as only one type of action (transfer) is considered.
- The plan generation time on my machine with all heuristics in place is less than a second for all goals and worlds.

4 Backward Planning

- For backward planning, I represent the goal as a set of constraints. I implement a relevant action checker, which checks whether an action could be the last action in a plan.
- I also implement functions to regress goal backwards, and keep checking whether the initial state satisfies the new constraints.
- For implementing the backward regression of the goal constraints, I changed the *drop* action to a new action template *dropThis(a,b)*. Regressing on *dropThis* is easier, as we know which object was dropped, whereas in drop it could be any object, and one would have to deal with first-order logic methods to build goal constraints. I map the *dropThis* action back to drop finally.
- The algorithm is again depth limited DFS. The max depth is obtained from the same heuristic for the forward planner.
- I also borrow pattern database ideas from the forward planner, removing the fridge and cupboard state predicates initially by opening both of them, and solving the search problem by backward planning, and close them back if needed for the goal.
- The branching factor is very low, in the range of 1-6 in most cases. This is because only relevant actions are considered, and they are very few.
- The plan generation time on my machine is less than a second for all goals and worlds.

5 Forward vs Backward

The plan generation for forward planning with heuristics and backward planning without heuristics is roughly the same, with similar branching factor. This illustrates the power of backward planning, as it works incredibly without any heuristics. However coming up with heuristics for forward planning is easy, and it is plausible that on some domains forward planning with strong heuristics can outshine backward planning.
Forward planning without heuristics has a very high branching factor, and consequently the plan generation time will also rise. Backward planning is more robust to lack of heuristics, however implementing backward planning is difficult, particularly in complex domains.