Applying Abstract Markov Decision Process on Building Domain

Seungchan Kim

May 15th 2018

1 Abstract

Due to its extreme amount of computation, large action-state spaces has been one of the main challenges in robotic planning and learning problems. To solve these problems, robot scientists have suggested hierarchical approaches that decompose a big, complicated task into small, simpler tasks that are computationally more feasible. One of them is Abstract Markov Decision Process (AMDP), which provides the hierarchical frameworks of planning problems and uses abstract states, actions, rewards, transitions, and projection functions. In this project, I aim to extend the use of AMDP framework to new simulative domains - construction and build domains. I suggested new definitions for construction and building domain problems, and tested the effectiveness of AMDP planning method on building domain by comparing with base planner method on the same domain.

2 Introduction

Robots often have to deal with complicated environments and large amount of computations. For example, if a robot's task is to pick up multiple objects from different rooms, the robot has to plan out the policies by comparing future states and optimal actions. The robot has to decide which rooms to navigate first and which objects to pick up, and so on, and if the number of objects and rooms increase, the amount of computation grows combinatorially as well. To tackle these problems, there have been various approaches

that decompose a big, complicated - often unstructured - task into hierarchies of small, simpler tasks that have their own subgoals.

Abstract Markov Decision Process (Gopalan et al. 2017) is an approach that shares the similar objective. It decomposes large planning problems (moving object from one place to another, or cleaning a room with multiple trashes) into Abstract MDP structures. In its hierarchical structure, the root task is divided into smaller problems, each of which has its own subtasks, local rewards and transition functions. At the very base level, the actions are defined as primitive actions; as the level gets higher and more abstract, the actions for the level are defined as non-primitive actions. In the original AMDP paper, this framework was tested on three domains (two simulative environments: taxi problem and cleanup world problem, and one real-robotic setting: cleanup task with turtle-bot).

In this project, I wanted to extend the use of AMDP framework to new domains, which are construction and building domains. For a robot, construction and building problem requires the policy planning for 'assembling objects'. The previous domains that tested AMDP framework (taxi and cleanup) were more about navigation and pickup tasks. Construction and building domains ask the robot questions on how to assemble objects in specific order, or how to put down objects efficiently according to the given blueprint.

In order to test AMDP framework on these problems, I suggested a new simulative environment by giving the definitions of states, actions types, reward and transition functions. With these basic definitions, I also suggested hierarchies of subtasks for construction and building domain problems. Due to the lack of time, I couldn't test AMDP method on construction domain. But instead, I could observe the AMDP planning results for building domain, which is a main part of construction domain. Then I compared the results with base planner called bounded RTDP.

3 Related Work

AMDP framework (Gopalan, et al 2017) is defined with following tuples:

$$< S, A, R, T, \epsilon, F >$$

S is a set of tuples, A is a set of actions, R is reward function (reward when the agent at state s transitions to state s' after executing action a), T is a transition function, and ϵ is a terminal function. F is a projection function, which only exists in AMDP formalism, that maps from the lower-level states to higher-level abstract states.

For each problem, AMDP framework needs hierarchical structures of tasks and subtasks. In AMDP formalism, each subtasks has its own subgoals, as well as local actions, local states, rewards, and transitions functions at multiple levels of hierarchy. In taxi domain, for example, a taxi agent starts at a random position in 5 by 5 space with multiple rooms and walls. It tries to find a passenger in a certain room, navigates through the rooms, picks up a passenger, and transports him to the goal location. The taxi agent decomposes the problem into smaller problems (GET and PUT; level2), which, respectively, are to find a passenger and transport a passenger. These two problems are decomposed into even smaller problems (Pick up, Navigation, Put down; level 1). The very base level actions, also called as primitive actions, are move (west, east, north, south), pickup and drop off.

Inspired by this taxi problem, I defined a construction and building domain. Basically, construction domain is a combination of object-fetching problem and building problem. In the next section, I will define the hierarchies for both construction and building domains, but I will focus more on the building domain.

4 Technical Approach

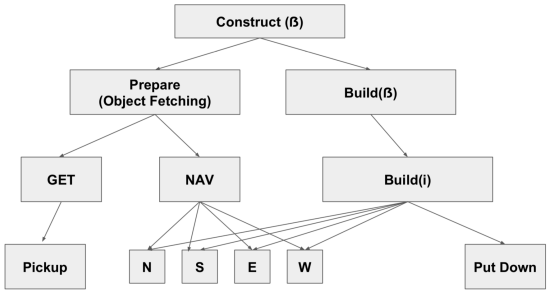

Figure 1: Construction Domain Hierarchies

In construction domain, constructor agent takes a blueprint (β) as an input. The agent divides the task into two separate tasks: first is Prepare (object fetching) and second is Build. To prepare means to find the ingredients needed for the construction, as stated in the blueprint. It searches around the space and navigates through the rooms to find the necessary ingredients (objects). The base level actions should be pickup and move(north, south, east, west).

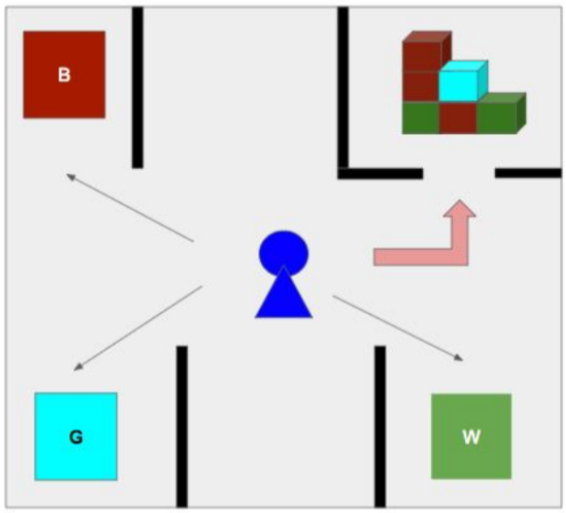

Figure 2: Construction Domain

After prepare task is done (when the agent is ready to start building the structure), the agent starts building task. It decomposes the task into Build(i) subtasks, each of which represents building i-th floor of the structure. For example, if the agent has to build three-story building, it will decomposes the building task into Build(0), Build(1), Build(2) tasks. The very base level primitive actions are move (west, east, south, north) and put down (which is to put the right ingredient block into the right place).

Prepare task (level 2) in the construction domain is very similar to the previous domain (taxi domain). Get and Navigation Tasks at level 1 are similar with the Pickup and Navigation tasks (level 1) in taxi domain. The construction agent - navigating through the rooms, searching ingredients, gathering them, and moving them to the building space - is analogous to the taxi agent navigates to multiple passengers, and transports them to the goal location.

Building task is the main component that makes construction domain more unique (building task is a domain by itself). Let's take a look at the building domain with more details. Figure 3 is a diagram of hierarchy when the robot agent has to build three-story structure. Just like construction domain, the build task is given a

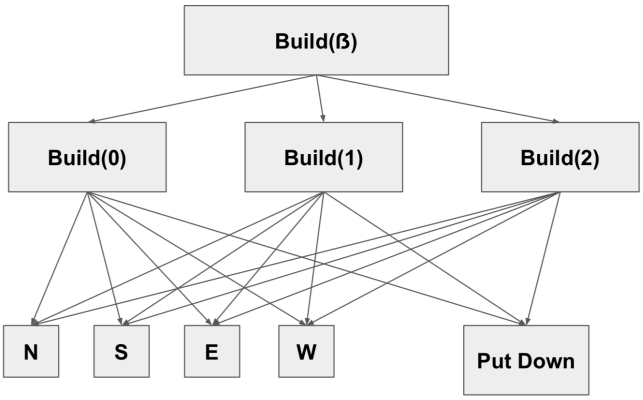

Figure 3: Building Domain Hierarchies

blueprint β to follow. The robot agent has a blueprint, and according to the AMDP hierarchy, the robot starts building the zeroth floor of the building. Then 1st and 2nd floors are completed. For each build(i) action, there are primitive actions move and putdown. The robot agent moves around the space, finds the right position to put objects, and puts down the object.

For the experiment, states and actions are defined as following: a state is defined by map and agent. Map defines the entire space in which robot agent moves around. In our case, we use 3D map (maxX by maxY by maxZ); for convenience, let's assume that we are not building any structures taller than 3-floors. So we define the size of map as N by N by 3, where N is the maximum width of the map(x and y directions) and 3 represents the vertical z directions. Map is composed of each 'position' class, which not only contains the information of x,y,z, coordinate vectors, but also the boolean vector that represents whether or not the certain position is filled with a block. If it is true, a block is already positioned in the x,y,z

coordinate, and if it is false, the position is empty.

Agent is defined with x,y,z coordinate as well as the blueprint given to the agent. In this problem, we do not care about the vertical motion of the robot; hence z is always zero for the agent. The robot agent can freely move around horizontally in 2D space, unless it is blocked by wall or by the structure it is building.

Blueprint is defined as an array of component tuples. Each tuple is composed of the x, y, z coordinates (where the block will be placed at), and the ingredient type (b - brick, w - wood, g - glass) string. For example, the following code defines a sample blueprint

```
public static BuildBlueprint getBluePrint1() {
    String name = "blueprint1";
    BuildComponent c1 = new BuildComponent(3,3,0,"b");
    BuildComponent c2 = new BuildComponent(3,2,0,"b");
    BuildComponent c3 = new BuildComponent(2,3,0,"b");
    BuildComponent c4 = new BuildComponent(3,3,1,"b");
    BuildComponent c5 = new BuildComponent(3,2,1,"b");
    BuildComponent c6 = new BuildComponent(3,3,2,"b");
    BuildComponent[] arr = {c1, c2, c3, c4, c5, c6};
    BuildBlueprint blueprint = new BuildBlueprint(name, arr);
    return blueprint;
}
```

Figure 4: Sample Blueprint

whose name is "blueprint1". There are six components in this blueprint. At zeroth floor, there are three blocks to build ((x=3, y=3), (x=3, y=2), (x=2,y=3)), and at first floor, there are two blocks to build ((x=3,y=3),(x=3,y=2)), and on the second floor, there is one block to build (x=3,y=3)). The ingredient type for all blocks, in this case, is 'b', which represents 'brick'. (The types of ingredients were defined for construction domain, because it would be more relevant with object-fetching task; in building domain, for simplicity, let's just assume that all the ingredient types are the same.)

As for the primitive actions, there are two types - move and putdown. Move actions are north, east, south, west actions, and the robot agent can freely move to the next place, as long as the target position is within the boundary of the map and the agent is not blocked by the structure it is currently building. Putdown actions are limited only when the agent is at the right place; if the

agent is right next to the target position (either x or y coordinates), the agent can possibly put down the object to the position. In other cases, putdown actions are not allowed.

Moreover, there are two cases of illegal actions for putdown. When the agent tries to put down a block that was already put down, or in other words, if it tries to fill in a position coordinate that was already filled in, we define it as a illegal action. Another case of illegal actions is when the agent tries to put down a block to a certain position, while the lower-level position is empty. For example, if the agent tries to put a block at (x=3,y=3,z=1). But if there is no block at (x=3,y=3,z=0), in other words, the lower-level position is empty, then we regard this case as an illegal action.

We give negative rewards (-10) for all illegal actions. For all the actions we give -1 by default, and if the agent completes the task, we give +20 rewards to the agent. For convenience, transition dynamics was set to be deterministic in this project, but we can always set it as non-deterministic function.

5 Evaluation

First, I tested the building domain by counting average number of backup operations. Number of backup operation is a meaningful value to plot, because it contains information of how many Bellman updates are used in planning, and signifies how much time/computation are needed for planning.

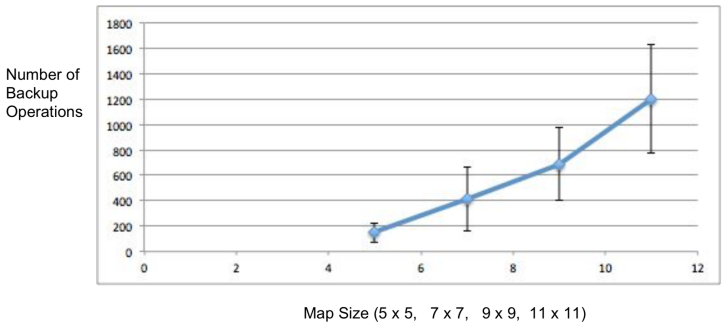

Figure 5: Number of backup operations vs. Map size

Figure 5 shows a graph between number of backup operations vs

map size, when tested with base planner. I tested with four different map sizes (5 by 5, 7 by 7, 9 by 9, and 11 by 11), and set all the other factors equally. (The robot agent starts at random initial point, and the number of blocks in blueprint was 6). As can be seen from the graph, the number of backup operations increased as the map size increased; when map size was 5 by 5, average number of backup operation was less than 200, while the map size was 11 by 11, it increased to over 1200.

Next, I compared the AMDP method and base planner method on building domain. For base planner, I used bounded RTDP method, which is implemented in burlap. Using bounded RTDP, we can similarly count the number of backup operations during the policy planning. In order to see the difference between AMDP planner and base planner, I differentiated the number of blocks used in building the structure; hence three different blueprints (number of blocks = 3, 6, 12) were tested. All the other factors were set equally (size of the map is 9 by 9, etc)

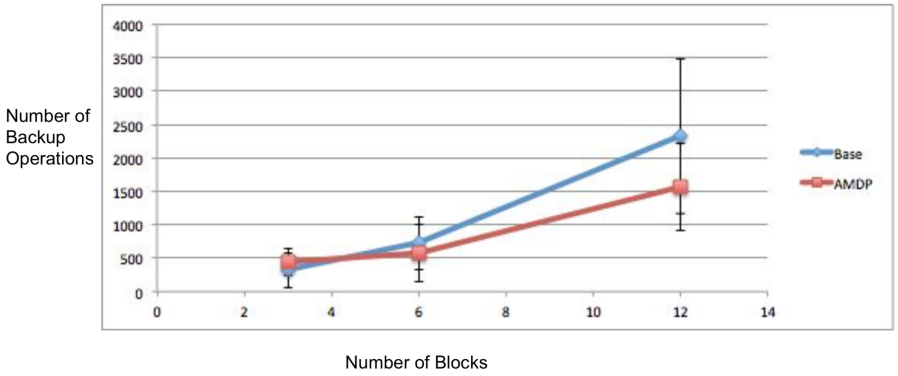

Figure 6: AMDP vs Base Planner

From figure 6, we can compare the number of backup operations of AMDP method and base planner method. As number of blocks (in the blueprint) increased, the number of backup operations increased. When number of blocks was 3, both results for AMDP and base planner methods were around 500 (average number of backup operations of AMDP was slightly higher than that of base planner). However, as number of blocks increased, number of backup operations of AMDP was smaller than that of base planner method;

especially, the gap between AMDP and base planner widened when there are 12 blocks in the blueprint.

As AMDP planning uses hierarchical structures for this domain, it decomposes a entire building task into a network of sub-tasks that has subgoals to complete. In this case, I divided the root task into each of building ith floor of the structure, and it showed that, when the task gets more complicated (number of blocks increased), AMDP planner showed better results than base planner.

6 Conclusion

I extended the use of AMDP framework to new domain: building domain. Building domain is a problem of assembling objects in specific orders or according to given blueprints. I defined states and action types, and also defined the abstraction hierarchies for the problem. Using this hierarchical structure, I implemented AMDP framework to the planning problem.

To compare the effectiveness of AMDP planner with that of base planner method, I counted the average number of backup operations as I tried with different number of blocks in blueprints. AMDP planner showed better results (less backup operations) for structures with more blocks.

For future improvements, I think there are various directions to continue and extend this project. As mentioned earlier, I've only tried to building part of construction domain, and didn't implement prepare (object fetching) part. So it would be meaningful to implement the entire construction domain according to the hierarchy, and check the effectiveness of AMDP planner. Adding some noises to the build domain (considering the case in which the built structure falls or collapses during the construction: how would robot agent react to it?) and counting/comparing the number of illegal actions during AMDP and base plannings would be a minor modification that is worth trying. Comparison with other planning methods (e.g. MAXQ) would also be a meaningful direction to reach.

References

[1] Gopalan, N.; desJardins, M.; Littman, M. L.; MacGlashan, J.; Squire, S.; Tellex, S.; Winder, J.; and Wong, L. L. 2017. *Planning with abstract Markov decision processes.*In International Conference on Automated Planning and Scheduling.

[2] Bakker, B.; Zivkovic, Z.; and Krose, B. 2005. *Hierarchical dynamic programming for robot path planning* In IEEE/RSJ International Conference on Intelligent Robots and Systems

[3] Jong, N., and Stone, P. 2008. *Hierarchical model-based reinforcement learning: R-max+ MAXQ* In International Conference on Machine Learning.

[4] Brown-UMBC Reinforcement Learning and Planning (BURLAP) tutorial
http://burlap.cs.brown.edu/