EECS 492: Introduction to AI
Homework 4 (100 pts)

General Information

Due: 11:59 PM, November 10, 2016
Notes:

- For the written questions, put your answers in a pdf (with your name in the pdf) and submit to the corresponding assignment on Gradescope.
- For the coding questions, submit your code file **firstname_lastname_planner.py** to the corresponding assignment on canvas, and also **append the code** to the end of your pdf.
- Late homework will be penalized 10% per day (each day starts at 11:59 PM on the due day).
- Homework turned in after **three** late days **will not be accepted**.

Note that the problems require that you develop planning graphs. It is absolutely crucial that these graphs are drawn clearly. Otherwise, the graders will not be able to interpret your findings.

Use the following conventions:

- Persistent actions are represented by circles (as seen in the figure below)
- Other actions have a box drawn around them (as seen in the figure below)
- Make sure that the state variables are in one consistent line across the page (as seen in the figure below).
- If a vertical line crosses a horizontal line, use the format shown in the figure below.
- If a vertical line breaks off from a horizontal line, use a black circle (below)
- Draw the planning graphs using the landscape layout. Make sure that the entire planning graph fits on one piece of paper!

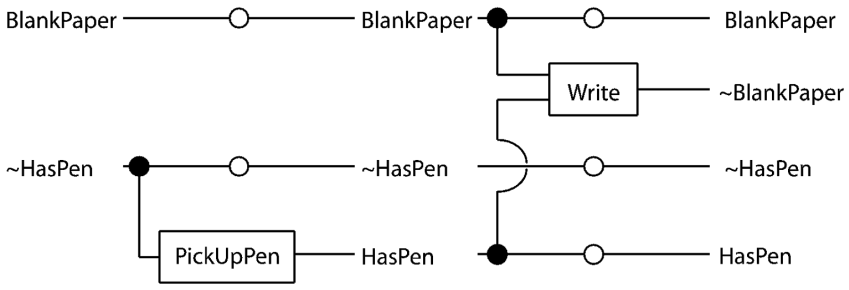


Figure 1: Format for planning graphs

Exercise 1 (10 pts)

The description of the block world can be found in the textbook (Page 370). Use the figure below to describe the actions required to achieve the goal state with preconditions and effects and explain why a *noninterleaved* planner cannot solve this problem. A *noninterleaved* planner is a planner that, when given two subgoals G1 and G2 , produces either a plan for G1 concatenated with a plan for G2 , or vice versa. The initial and goal state are given below with the available actions.

Init *On(C, Table) ^ On(B, Table) ^ On(A, B) ^ Block (A) ^ Block (B) ^ Block (C) ^ Clear (A) ^ Clear (C)*
Goal *On(A, B) ^ On(B, C)*

Action *Move(b, x, y)*
Precond *On(b, x) ^ Clear(b) ^ Clear(y) ^ Block(b) ^ Block(y) ^ (b≠ x) ^ (b≠ y) ^ (x≠ y)*
Effect *On(b, y) ^ Clear(x) ^ ¬On(b, x) ^ ¬Clear(y)*

Action *MoveToTable(b, x)*
Precond *On(b, x) ^ Clear(b) ^ Block(b) ^ (b≠ x),*
Effect *On(b, Table) ^ Clear(x) ^ ¬On(b, x)*

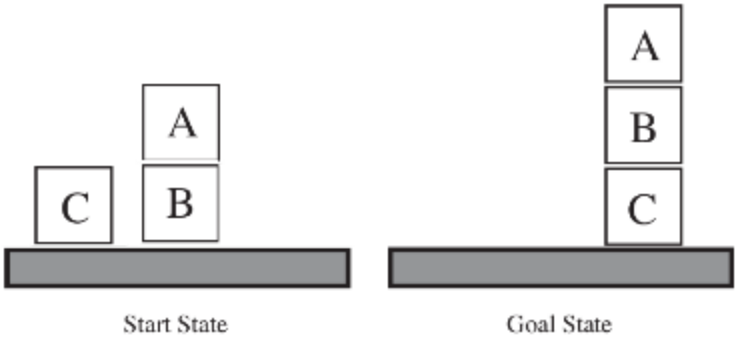


Figure 2: Block world

1. [4 pts] show the preconditions and effects of *MoveToTable(A, B) and Move(B, Table, C)*

2. [6 pts] show why achieving the subgoals *On(A, B)* and *On(B, C)* in order would prevent achieving the goal state.

Exercise 2 (32 pts)

Let's first describe our car-driving environment. We know that to drive a car we must do a few things:

1. The agent must find its keys. Initially it does not have its keys, but it then locates the keys after performing this action.
2. The agent must get into the car. For that he needs keys and after performing this action, he is in the car.
3. The agent must start the car. The car has to have gas, the agent has to have the key and be in the car, after which the engine will be running
4. The agent steps on the gas peddle. The engine must be running. Afterwards, the car is moving.
5. The agent steps on the brake on a moving car. Afterwards, the car will stop moving and engine will stop running and the car will be in the parking lot.

Note: all things in a car in motion (i.e., the agent and the key) will also be in motion.

Consider the following state variables and actions:

- Actions: *FindKeys(FK)*, *GetInCar(GC)*, *StartCar(SC)*, *StepOnGas(SG)*, *StepOnBreak(SB)*
- Literals: *InCar(IC), HasGas(HG), HasKey(HK), EngineRunning(ER), CarMoving(CM), AtParking(P)*

NOTE: The abbreviations mentioned in the brackets for Actions and Schema are not parameters but a convenient abbreviation that could be used for drawing the planning graph.

Initially the car is stationary, agent is not in the car, it doesn't have its keys, but the car has gas.

1. [15 points] We will consider the five actions described above. Describe the action schema for each of the five actions.

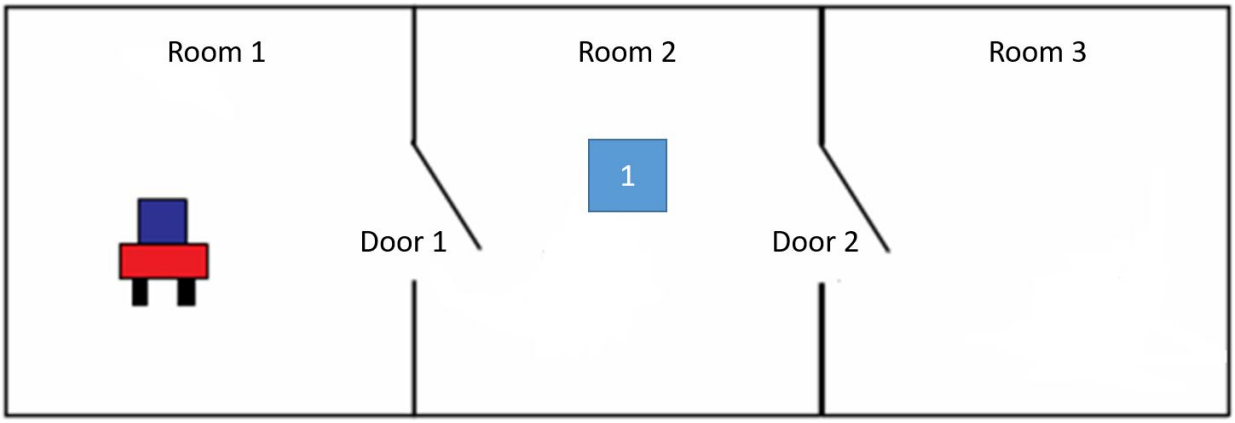
2. [12 points] An agent has to go to the store to get some milk. It has decided to take its car from the car's starting point in the driveway to the grocery store parking lot. Write the planning graph for the transportation component of this problem starting from when the agent is not in the car and ending stopped & engine off at the parking lot. For simplicity's sake, assume that the agent needs to accelerate only once to arrive at the parking lot, the agent doesn't have the keys and car is stationary in the beginning. Draw the planning graph. How many levels of states are needed to obtain the goal state? (Use the abbreviated names for actions and literals to make it look cleaner. Don't show the mutexes.)

3. [4 points] Which actions are mutex with *StepOnBreak(SB)*? (Mention the action with the type of mutex, if there are more than one mutexes between two actions then mention all the types)

4. [1 point] Which literals are mutex with *EngineRunning(ER)*? (Mention the literal with the type of mutex, if there are more than one mutexes between two literals then mention all the types)

Exercise 3 (18 pts)

Consider a world with three rooms. There is a door (Door1) between Room1 and Room2, and another door (Door2) between Room2 and Room3. There is one box in Room2. The robot starts in Room1 and its goal is to get the box into Room3.



Write the action schema for the following actions:

- *GoThru(X,Y)*
- *PushThru(B,X,Y)*

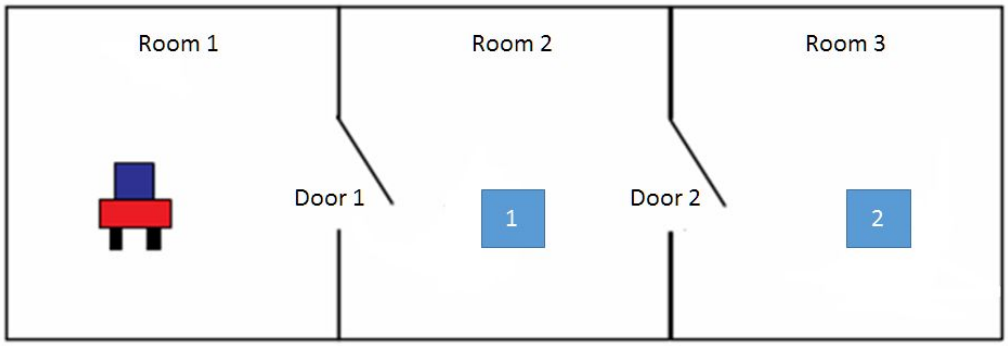
In the schema, B is a box, and X and Y are rooms. Use *In(R,X)* or *In(B,X)* to indicate that the robot (R) or box is in Room X. Also, the robot can move a box to one room at a time and the rooms must be adjacent. If the robot pushes the box, then the robot also moves with the box. The goal room is Room 3.

1. [2 points] Write out the state descriptions for the initial and goal states given three room world.

2. [10 points] Create a planning graph for the problem. Remember to include all actions that are possible at each level, A_i (not just the ones that are on the path to the goal) Stop at the level where the goal state is present. Don't show the mutexes.

3. [5 points] List and categorize the mutexes in the first action level (A0) and the second state level (S1). Represent persistent actions by brackets, such as P[In(R, R1)]. If a mutex is in multiple categories, list all the categories.

4. [1 point] Use a heuristic to estimate the minimum number of actions needed to get to goal state. Let H = num_go_through + num_push_through. Where num_go_through is the minimum number of connecting passages that the robot needs to go through (from its current location) to get to the boxes that are not in the goal room. If all boxes are in the goal room, then H is 0. Num_push_through is the minimum number of pushes the robot needs to perform to get the boxes into the goal room. If all boxes are currently in the goal room, num_push_through is 0.



What is value of H in the given the figure above?

Exercise 4 Forward/Backward Chaining (40 pts)

- Implement forward progression state-space search (Section 10.2.1) and backward relevant-states search (Section 10.2.2) using breadth first search (BFS).

  Each task file is defined as following:
  line 1: initial state
  line 2: goal state
  line 3: action 1 | precond | add_list | del_list
  line 4: action 2 | precond | add_list | del_list
  .

.
.
line N: action X | precond | add_list | del_list

The initial state and goal state are composed of a list of predicates. Predicates are separated by ";", and each predicate is represented as "Object, argument(s)". Arguments are separated by space. For instance, *Cargo, C*; *Airport, JFK; At, C JFK.*

Each action consists of action name/args, precond, add_list and del_list, which are separated by "|". For action name/args, it is represented as "Action, argument(s)". The precond, add_list and del_list follow the same format as predicates. For instance, "*Fly, X Y Z | Cargo, X; Airport, Y; At, X Y | At, X Z | At, X Y*"
Note that we use del_list to indicate negation of preconditions (Page 368).

We provide a utils.py to help you read the task file. After implementing forward progression state-space search and backward relevant-states search, the program will output a list of actions that leads from the initial state to the goal state for each task if a solution is found (terminate after a maximum of 10,000 iterations).

You can run your code by typing
$ python firstname_lastname_planner.py <forward/backward> <task#.txt>

- In the following table, indicate which tasks your code solves for each algorithm (Yes/No)

Task #	Forward	Backward
1		
2		
3		
4		
5		