Robin Mehta (robimeht)
Artificial Intelligence Homework 4

Exercise 1:

1. Show the preconditions and effects of MoveToTable(A, B) and Move(B, Table, C)

Action: MoveToTable(A, B)
Preconditions: On(A, B) ^ Clear(A) ^ Block(A) ^ (A ≠ B)
Effect: On(A, Table) ^ Clear(B) ^ ~On(A, B)

Action: Move(B, Table, C)
Preconditions: On(B, Table) ^ Clear(B) ^ Clear(C) ^ Block(C) ^ Block(B) ^ (B ≠ C) ^ (B ≠ Table) ^ (Table ≠ C)
Effects: On(B, C) ^ Clear(Table) ^ ~On(B, Table) ^ ~Clear(C)

2. Show why achieving the subgoals On(A, B) and On(B, C) in order would prevent achieving the goal state.

Subgoal 1: On(A, B)
    Action: Do nothing from init state.
Subgoal 2: On(B, C)
    Action: MoveToTable(A, B)
    Effect: ~On(A, B)
    Action: Move(B, Table, C)
    Effect: ~On(A, B)
Goal state is not achieved because MoveToTable(A, B) is a necessary action for Subgoal 2: [On(B, C)], and results in ~On(A, B), which remains unchanged after Move(B, Table, C) and is a contradiction of the goal state.

Exercise 2:

1. Describe the action schema:

a) FindKeys()
Preconditions: ~HasKey()
Effect: HasKey()

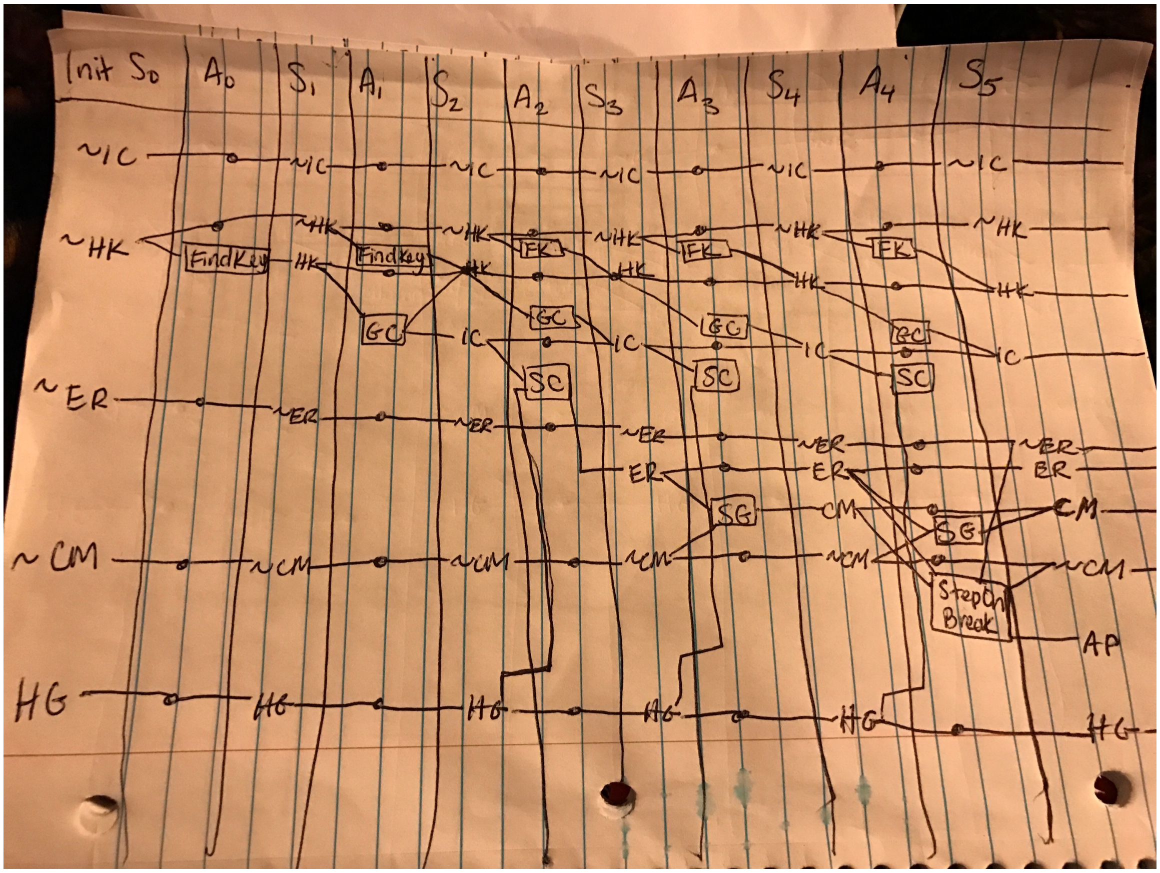
b) GetInCar()
Preconditions: HasKey() ^ ~InCar()
Effects: InCar()

c) StarCar()
Preconditions: HasGas() ^ InCar() ^ HasKey()
Effects: EngineRunn
ing()

d) StepsOnGas()
Preconditions: EngineRunning() ^ ~CarMoving()
Effects: CarMoving()

e) StepsOnBreak()
Preconditions: CarMoving() ^ EngineRunning()
Effects: ~CarMoving() ^ ~EngineRunning() ^ AtParking()

2. Draw the Planning Graph:



3. What actions are mutex with StepOnBreak(SB)?

Inconsistent effects: The effect (~CM) of SB is mutex with the persistent effect of CM. The effect (ER) of SB is mutex with it's effect, ~ER.

Interference: The precondition of SB is CM, which is mutex with the persistent effect of ~CM.

Competing Needs: The precondition of SB is CM, which is in mutex with the precondition of the persistence of ~CM. Also, SG and SB are mutex actions because SG requires ~CM, and SB requires CM.

4. What literals are mutex with EngineRunning(ER)?

Inconsistent support: Persistence of ~ER is mutex with HK, HG, IC, which are preconditions to StartCar, which results in ER.

Exercise 3:

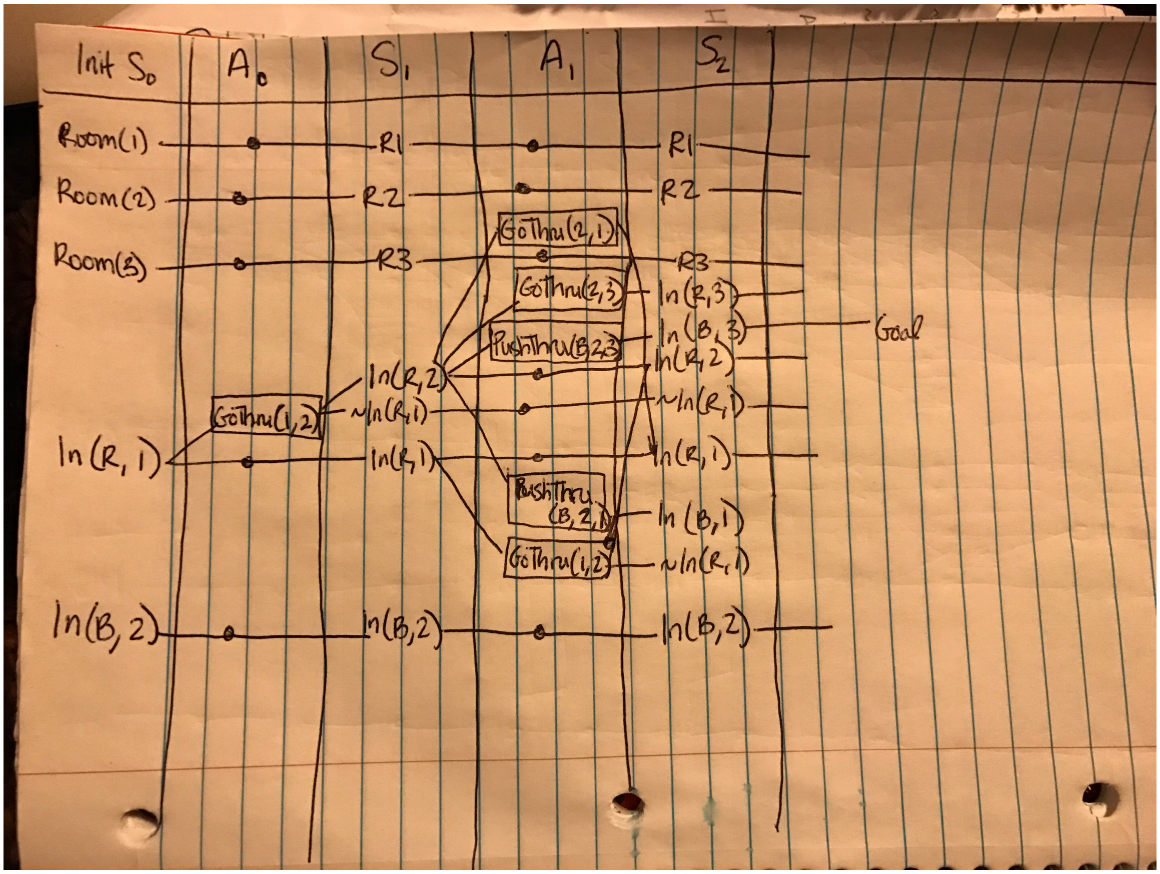
Action: GoThru(X, Y)
Precondition: In(R, X)
Effects: In(R, Y) ^ ~In(R, X)

Action: PushThru(B, X, Y)
Precondition: In(R, X) ^ In(B, X)
Effects: In(R, Y) ^ In(B, Y) ^ ~In(R, X) ^ ~In(B, X)

1. State descriptions for initial and goal states:

Initial State: In(R, 1) ^ In(B, 2) ^ Room(R1) ^ Room(R2) ^ Room(R3) ^ Door(R1, R2) ^ Door(R2, R3) ^ Box(B)
Goal State: In(B, 3)

2. Planning graph:



3:

Mutexes for A0:

Interference mutex, and Inconsistent effects:
The effect of GoThru(1, 2) is ~In(R, 1) which is the negation of the P[In(R, 1)].

Mutexes for S1:

Inconsistent Support:
Literal In(R, 2) mutex with persistent In(R, 1).

4: Heuristic = num_go_thru + num_push_thru.

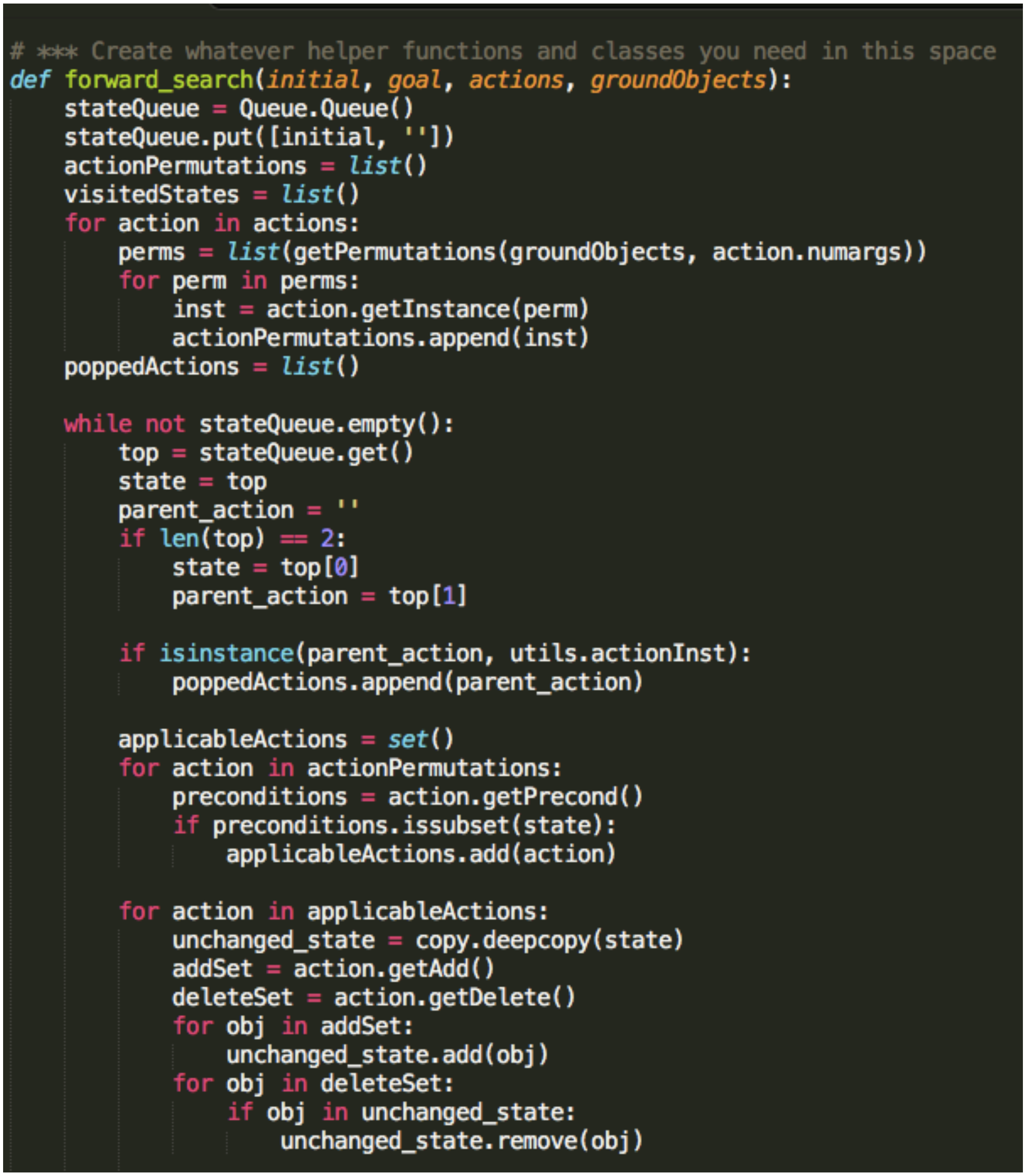
1) Robot goes through Door(1, 2). Num_go_thru = 1
2) Robot pushes box through Door(2, 3)

Heuristic = 1 + 1 = 2

Exercise 4:

Task #	Forward	Backward
1	Yes	Yes
2	Yes	Yes

3	Yes	No
4	Yes	No
5	Yes	Infinite Loops



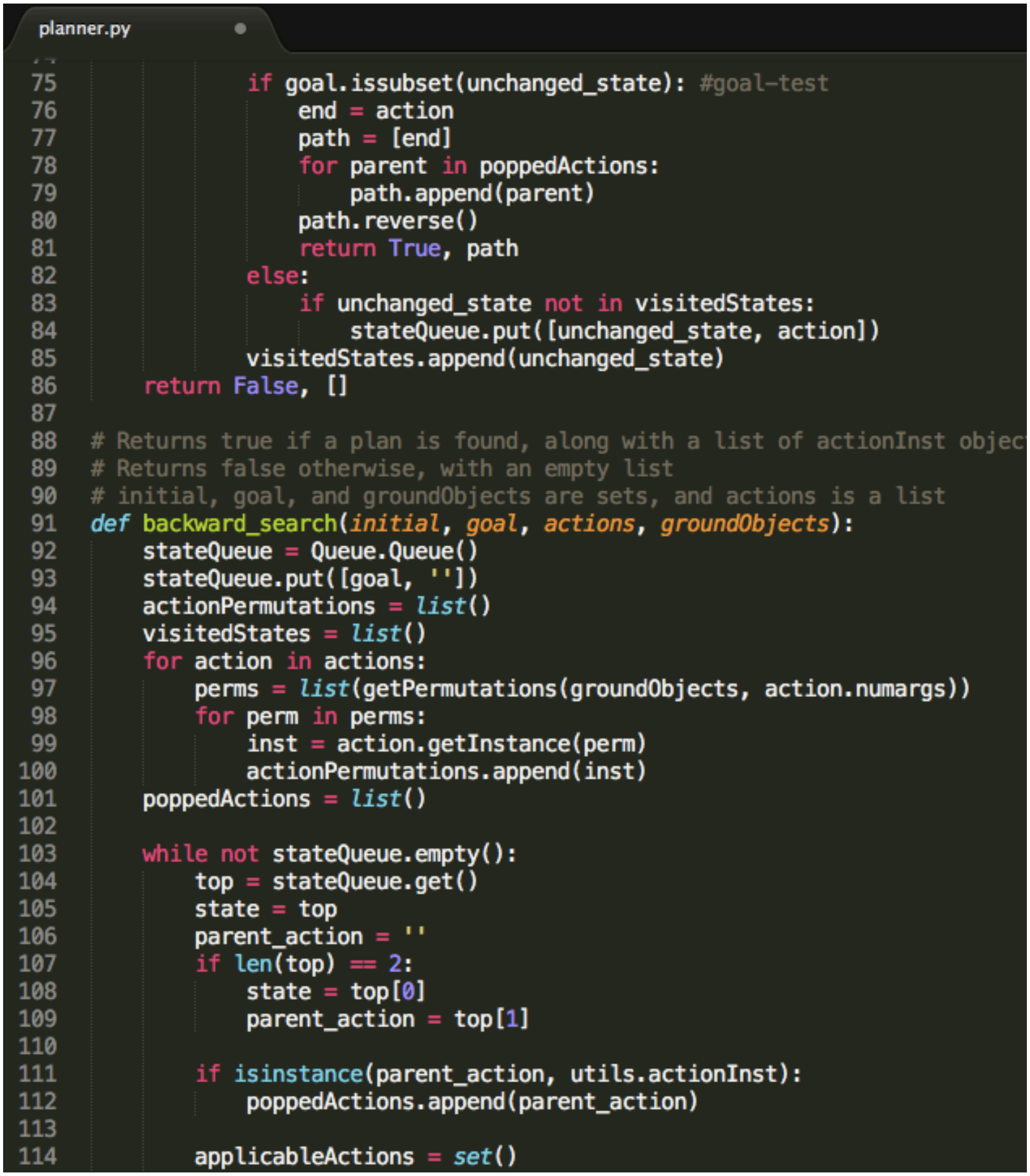
```
# *** Create whatever helper functions and classes you need in this space
def forward_search(initial, goal, actions, groundObjects):
    stateQueue = Queue.Queue()
    stateQueue.put([initial, ''])
    actionPermutations = list()
    visitedStates = list()
    for action in actions:
        perms = list(getPermutations(groundObjects, action.numargs))
        for perm in perms:
            inst = action.getInstance(perm)
            actionPermutations.append(inst)
    poppedActions = list()

    while not stateQueue.empty():
        top = stateQueue.get()
        state = top
        parent_action = ''
        if len(top) == 2:
            state = top[0]
            parent_action = top[1]

        if isinstance(parent_action, utils.actionInst):
            poppedActions.append(parent_action)

        applicableActions = set()
        for action in actionPermutations:
            preconditions = action.getPrecond()
            if preconditions.issubset(state):
                applicableActions.add(action)

        for action in applicableActions:
            unchanged_state = copy.deepcopy(state)
            addSet = action.getAdd()
            deleteSet = action.getDelete()
            for obj in addSet:
                unchanged_state.add(obj)
            for obj in deleteSet:
                if obj in unchanged_state:
                    unchanged_state.remove(obj)
```



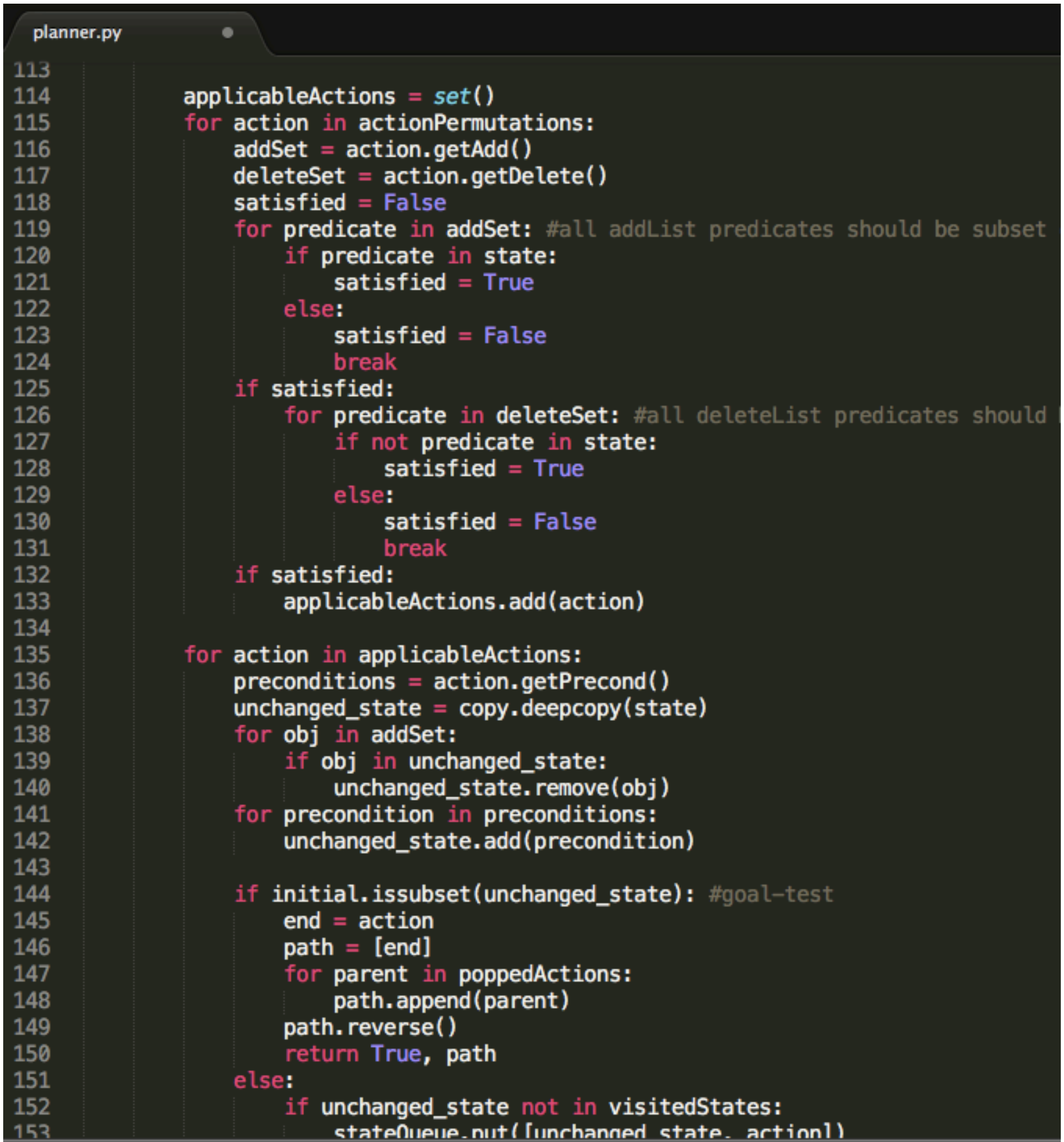
```
            if goal.issubset(unchanged_state): #goal-test
                end = action
                path = [end]
                for parent in poppedActions:
                    path.append(parent)
                path.reverse()
                return True, path
            else:
                if unchanged_state not in visitedStates:
                    stateQueue.put([unchanged_state, action])
            visitedStates.append(unchanged_state)
    return False, []

# Returns true if a plan is found, along with a list of actionInst objec
# Returns false otherwise, with an empty list
# initial, goal, and groundObjects are sets, and actions is a list
def backward_search(initial, goal, actions, groundObjects):
    stateQueue = Queue.Queue()
    stateQueue.put([goal, ''])
    actionPermutations = list()
    visitedStates = list()
    for action in actions:
        perms = list(getPermutations(groundObjects, action.numargs))
        for perm in perms:
            inst = action.getInstance(perm)
            actionPermutations.append(inst)
    poppedActions = list()

    while not stateQueue.empty():
        top = stateQueue.get()
        state = top
        parent_action = ''
        if len(top) == 2:
            state = top[0]
            parent_action = top[1]

        if isinstance(parent_action, utils.actionInst):
            poppedActions.append(parent_action)

        applicableActions = set()
```

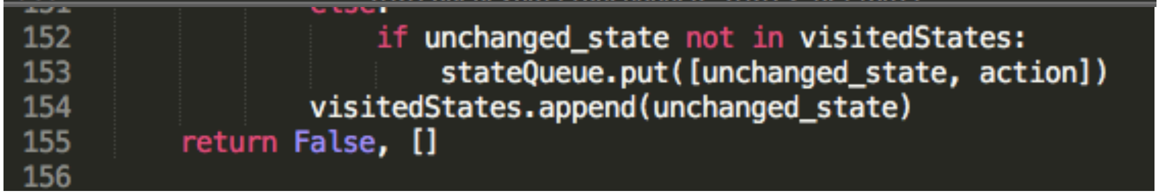


```

        applicableActions = set()
        for action in actionPermutations:
            addSet = action.getAdd()
            deleteSet = action.getDelete()
            satisfied = False
            for predicate in addSet: #all addList predicates should be subset
                if predicate in state:
                    satisfied = True
                else:
                    satisfied = False
                    break
            if satisfied:
                for predicate in deleteSet: #all deleteList predicates should
                    if not predicate in state:
                        satisfied = True
                    else:
                        satisfied = False
                        break
            if satisfied:
                applicableActions.add(action)

        for action in applicableActions:
            preconditions = action.getPrecond()
            unchanged_state = copy.deepcopy(state)
            for obj in addSet:
                if obj in unchanged_state:
                    unchanged_state.remove(obj)
            for precondition in preconditions:
                unchanged_state.add(precondition)

            if initial.issubset(unchanged_state): #goal-test
                end = action
                path = [end]
                for parent in poppedActions:
                    path.append(parent)
                path.reverse()
                return True, path
            else:
                if unchanged_state not in visitedStates:
                    stateQueue.put([unchanged_state, action])
```



```
                if unchanged_state not in visitedStates:
                    stateQueue.put([unchanged_state, action])
```

```
            visitedStates.append(unchanged_state)
    return False, []

```