Cognitive systems - Finale project **Shakediel Hiba**
Prof. Gal Kaminka
TA: Mika Barkan

Introduction

In this project we were asked to create an agent that will be able to solve as many problems as possible, in any given environment, with the constraint for some probability for an action to fail.
Here I shall explain the guidelines I followed when approaching the problem, and how I attempt to solve it.
Following the basic cycle that was shown in class many times:

Algorithm 1 basic agent algorithm

```
1: W is knowledge base
2: g is an unsatisfied goal in W
3: B set of actions available
4: P set of precepts available
5: while g is not satisfied do
6:     CHOOSE()
7:     EXECUTE()
8:     PERCEIVE()
9:     REMEMBER()
10: end while
```

And changing it a little bit, because of how pddlsim works:

Algorithm 2 basic agent algorithm for pddlsim

```
1: W is knowledge base
2: g is an unsatisfied goal in W
3: B set of actions available
4: P set of precepts available
5: while g is not satisfied do
6:     PERCEIVE()
7:     REMEMBER()
8:     CHOOSE()
9:     EXECUTE()
10: end while
```

In the following sections I will show the work I did on 2 of the above stages:
REMEMBER and CHOOSE

Leading Guidelines

During the work process I decided on some core concepts that I will try to implement. Those are:

- If there's only 1 action you may take, take it
- Once there's more than 1 possible action to take, plan ahead
- When planning, search for optimal path under the assumption that all transitions are possible (Using planner given inside pddlsim)
- When planning, always plan towards one specific goal
- When in the process of trying to achieve a goal, don't go back, unless absolutely necessary (No other option to take)
- Allow deviations from the original plan, if needed
- In The process of deciding where to go next, look a few steps ahead and go with what seems more promising
- once far enough for original plan, put it aside and plan again (if needed)
- learn from past runs, record the transitions we've made to learn about the environment

The REMEMBER phase

In the remember stage, I record all the transition that the agent has made in the environment. Meaning, a 3D matrix composing of state–action–next-state to a number of occurrences of the tuple.
Also, I'm recording all the rewards (assigning rewards to state-action pairs will be covered in the CHOOSE section) that are accumulated during the current problem solving stage. Meaning, a 2D matrix composing of state–action to a list of rewards recorded in the session.
to sum it up, the REMEMBER phase works as follows:

Algorithm 3 REMEMBER phase

```
1: procedure REMEMBER(previous-state, action, current-state)
2:     compute reward for last action
3:     record transition
4:     record reward
5: end procedure
```

The CHOOSE phase

In the choose stage I try to stick to the main guidelines listed above.
At first, if there's only a single action to take(leading to a state not seen before) in the state we're in, we take it.
If there's more than 1 possible action to take then we run the planner given to us in the pddlsim infrastructure, under the assumption that it is efficient. This planner lets out an "optimal path" from the state we're in to one of the chosen goal state (because of the lack of prior knowledge, i just choose the first uncompleted-goal in the list) with all transitions set to 1.
I use this "optimal path" as a means to weigh the transitions, or in other words - reward them, such that the closer we get to the goal in the plan the more reward we will get. formally, the reward is given by:

$$\frac{indexInPlan + 1}{sizeOfPlan}$$

Once I have a plan ready, instead of following it blindly, i define a look-ahead parameter, which basically computes the local policy for the given situation. In order to achieve this, we expand the weights we assigned to each transition, in a way such that each transition, which is adjacent to a transition in the plan (going out of the same node), will get the original transition weight, but with a discount factor 0.1:

$$offPlanTransitionWeight = inPlanTransitionWeight * 0.1$$

the term above will expand as the expansion in the state graph continues, and we will get the following term for transition weight:

Algorithm 4 transition weight assignment

```
1: procedure WEIGHT-EXPANSION(previous-transition-weight, next-transition-weight)
2:     new-transition-weight = 0.1 * previous-transition-weight
3:     if new-transition-weight $\geq$ next-transition-weight then
4:         next-transition-weight = new-transition-weight
5:     end if
6: end procedure
```

This way, we only update an edge's weight if there's an other path leading to it that brings it closer to the goal.

In the first few times of seeing a transition, we will not want to use the incomplete knowledge we have on it. So, until we have seen a transition enough times, we shall not use the accumulated knowledge and assume transition probability to the next state is 1 and the reward we will get is as the weight.
If a transition leads us to a state we have seen before - we punish the agent, and record the punishment as part of the REMEMBER phase.
Once we have seen a transition enough, we will use the average reward as seen this far and the transition probability as the sum of times getting to a next state divided by the number of tries.

The procedure of computing the next optimal step can be written as:

Algorithm 5 next optimal step

```
procedure CHOOSE-NEXT-ACTION(current-state)
    function GET-MAX-Q-VALUE(current-state, lookahead)
        Let actions be all valid actions for current-state
        for action in action do
            q-val = get-q-value(current-state, action, lookahead)
        end for
        return max(q-val)
    end function
    function GET-Q-VALUE(current-state, possible-action, lookahead)
        if lookahead $\leq 0$ then
            return get-reward(current-state)
        end if
        expected-future-return = gamma * compute-expected-future-return
        return expected-future-return + get-reward(current-state)
    end function
    function COMPUTE-EXPECTED-FUTURE-RETURN(current-state, action, lookahead)
        if transition-count(state) $\leq$ known-threshold then
            transition-probability = 1
        else
            transition-probability = occurrence / tries
        end if
        for next-state in possible-next-states do
            get-max-q-value(next-state, lookahead - 1) * transition-probability
        end for
        return expected-future-return + get-reward(current-state, action)
    end function
end procedure
```

Where gamma is a discount factor for future steps.
We choose the action that will bring us the best q-values

An addition to the above procedure is :
If we have gone far enough from the original plan - we would like to plan again.
But, keep the weights accumulated this far, while trying to achieve the current goal.
And continue with the above search method for next step.
To put it Formally:

Algorithm 6 replaning

```
let t be the previous transition - state action pair
let w be the weight given to t in the expansion process
let threshold be the threshold to replan
if w $\leq$ t then
    plan again if needed
end if
```

The threshold for replaning is set to:

$$threshold = last-in-plan-transition-weight * off-plan-punish-factor^lookahead$$

Implementation Notes

- The punishment for revisiting a state is given by the median of the weights given to a transition thus far in the current session.
- The look-ahead number of steps is given by the term:

  $$min(4, len(plan)/2)$$

  Where 4 and 2 are arbitrarily chosen numbers.
- gamma is set to a constant of 0.8, also chosen arbitrarily.
- Known threshold is set to 100.
- The states are being compressed into a hash for easily comparison and storage
- For every goal that we are planning for, we clear our local history. Meaning, the rewards leading us to the previous goal and the list of visited-states.