DECISION-THEORETIC PLANNING

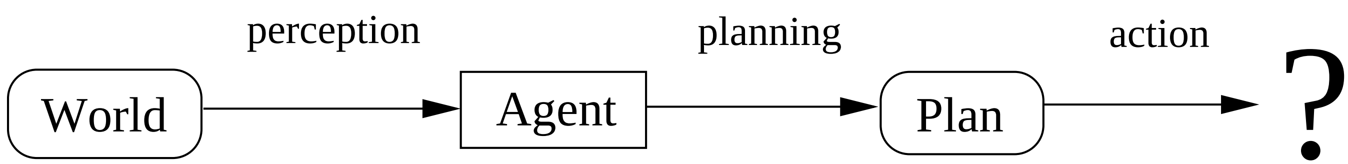
Closed loop planning

- The central question in designing an agent is building it so that it can figure out what to do next.
- That is finding a set of actions which will lead to a goal.
- Previously we studied a traditional approach to planning from AI.
- This was the use of means-ends analysis along with the STRIPS representation.

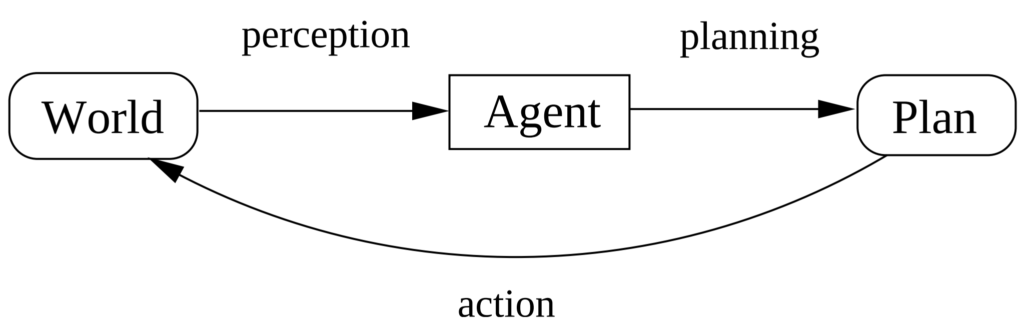
- STRIPS:
  - add condition;
  - delete condition; and
  - precondition.
- Algorithms use:
  - Use precondition to decompose goals;
  - Use add condition to select actions; and
  - Use delete condition to constrain order on actions.

- The main limitations of this approach are:
  - Efficiency (doesn't scale)
  - Robustness
- The second of these is what interests us here.
- The problem is:
  - Plan is linear
  - Planning is separated from acting
  - Actions are non-deterministic
- Though partial-order planning is an improvement on simple means-ends analysis, it still can't cope with non-determinism.

- One way of thinking about this is in terms of *closed loop* planning.
- Classical planning has:



- While close loop planning has actions which are dependent on what is observed in the world:



- Clearly this is the kind of planning that better fits agents.

- Conditional planning is one approach to closed-loop planning.
- Conditional plans are allowed to have branches and loops where control choices depend upon observations.
- For example:
  1. pick up block *A*
  2. while block *A* not held
     pick up block *A*.
  3. if block *C* clear
     put block *A* on block *C*.
  4. else clear block *C*.
- However, the situation gets more complex with unreliable sensors.

- To deal with unreliable sensors we need to bring in decision theory.
- (Just as we did to take account of dice rolls in game playing).
- A problem with using classical decision theory in the context of intelligent agents is that it is a one-shot process.
- The process only takes into account the current state and the one the decision will lead to.
- This is fine if the next state is the goal state.
- In contrast, what we are often interested in is determining a sequence of actions which take us through a series of states, especially when the choice of actions varies from state to state.

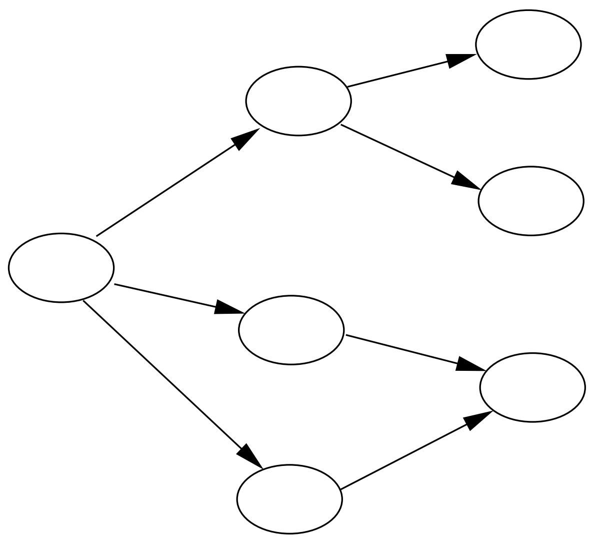
- We do this through the use of *decision theoretic planning* models.
- We will cover two closely related types of these models here:
  - Markov decision processes.
  - Partially observable Markov decision processes.
- Both are close in many ways to the kind of search models we studied earlier.
- The big change is that actions can have more than one outcome.
- So we start by considering planning as search.

Planning as search

- The earliest search models we looked at are a form of planning.
- In the sheep and dogs example, a solution was:
  - A sequence of actions;
  - Which led to a goal
- This is just a plan.
- Adding in a heuristic function gives us an idea of optimality:
- An optimal plan is:
  - A sequence of actions;
  - Which leads to a goal;
  - With minimum cost.

- We can describe a state space search model as:
  - a state space S;
  - an initial state s_0;
  - a set of actions, $A(s) \subseteq A$, applicable in each state $s \in S$;
  - transition function $f(s, a)$ for $s \in S$ and $a \in A$;
  - action costs $c(a, s) > 0$; and
  - a set of goal states $G \subseteq S$

- This gives us a problem space that looks like:



- A solution is a path through this space from initial state to a goal state.

- There are lots of ways of searching this space.
- One simple way is greedy search:
  1. Evaluate each action a which can be performed in the current state:
     $$Q(a, s) = c(a, s) + h(s_a)$$
     where s_a is the next state.
  2. Apply action a that minimises $Q(a, s)$;
  3. If s_a is the goal, exit
     else $s := s_a$, goto 1.
- This just picks the cheapest move at each point.

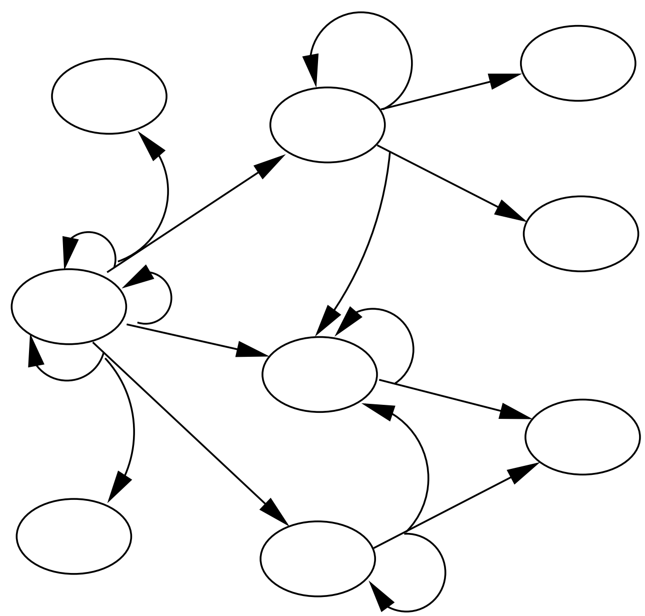
- This is a simple approach that uses little (and constant) memory.
- It can be easily adapted to give a closed-loop version:
  - Instead of s_a being the state we expect to get, make it the one we observe.
- Like any depth first approach, it isn't optimal.
- It might not even find solutions.
- (But we know how to use learning to ensure that it gets better over time).

Markov decision processes

- So far, there is nothing really new here.
- But it is only a small step to a much better representation.
- In a non-deterministic environment, we don't have a simple transition function.
- Instead an action can lead to one of a number of states.
- When we can tell which state we are in, then we have a Markov decision process (MDP)

- An MDP has the following formal model:
  - a state space S;
  - a set of actions, $A(s) \subseteq A$, applicable in each state $s \in S$;
  - transition probabilities $\Pr_a(s'|s)$ for $s, s' \in S$ and $a \in A$;
  - action costs $c(a, s) > 0$; and
  - a set of goal states $G \subseteq S$
- Thus for each state we have a set of actions we can apply, and these take us to other states with some probability.
- We don't know which state we will end up in, but we know which one we are in after the action (we have *full observability*).

- This gives us a problem space that looks like:



- A solution is now choice of action in every possible state that the agent might end up in.

- We can think of this solution as a function π which maps states into applicable actions, $\pi(s_i) = a_i$.
- This function is called a *policy*.
- What a policy allows us to compute is a probability distribution across all the trajectories from a given initial state.
- This is the product of all the transition probabilities, $\Pr_{a_i}(s_{i+1}|s_i)$, along the trajectory.
- Goal states are taken to have no cost, no effects, so that if $s \in G$:
  - $c(a, s) = 0$
  - $\Pr(s|s) = 1$

- We can then calculate the expected cost of a policy starting in state s_0.
- This is just the probability of the policy multiplied by the cost of traversing it:

$$\sum_{i=0}^{\infty} c(\pi(s_i), s_i)$$

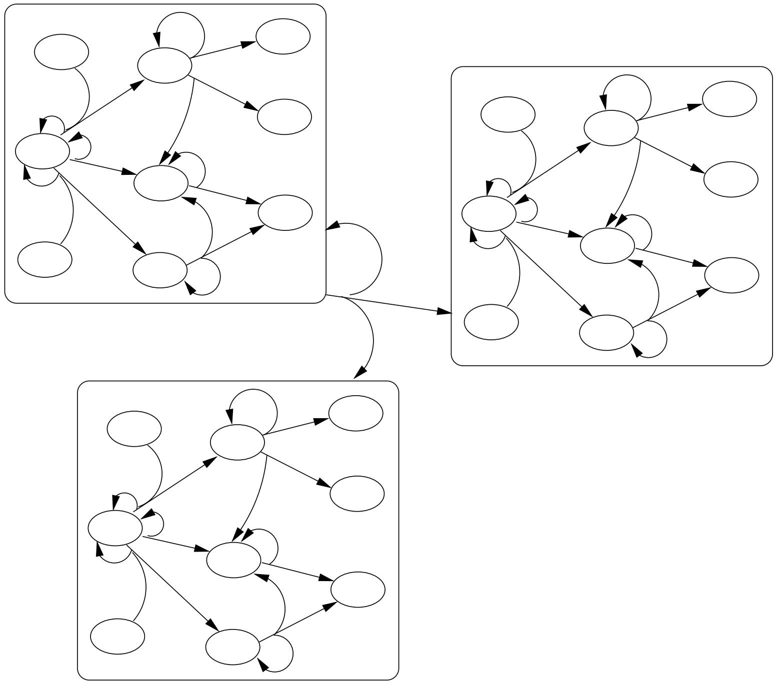
- An optimal policy is then a π^* that has minimum expected cost for all states s.
- As with the search version of the problem, we can solve this by searching, albeit through a much larger space.
- Later we will look at ways to do this search.

Partially observable MDPs

- Full observability is a big assumption (it requires an accessible environment). Much more likely is *partial observability*.
- This means that we don't know what state we are in, but instead we have some set of beliefs about which state we are in.
- We represent these beliefs by a probability distribution over the set of possible states.
- These probabilities are obtained by making observations.
- The effect of observations are modelled as probabilities $\Pr_a(o|s)$, where o are observations.

- Formally a POMDP is:
  - a state space S;
  - a set of actions, $A(s) \subseteq A$, applicable in each state $s \in S$;
  - transition probabilities $\Pr_a(s'|s)$ for $s, s' \in S$ and $a \in A$;
  - action costs $c(a, s) > 0$;
  - a set of goal states, G;
  - an initial belief state b_0;
  - a set of final belief states b_F;
  - observations o after action a with probabilities $\Pr_a(o|s)$

- So we have a situation which looks like:



- This is just an MDP over belief states.

- The goal states of an MDP are just replaced by, for example, states in which we are pretty sure we have reached a goal:

$$\sum_{s \in G} b(s) > 1 - \epsilon$$

- We solve a POMDP by looking for a function which maps belief states into actions, where belief states *b* are probability distributions over the set of states *S*.
- Given a belief state *b*, the effect of carrying out action *a* is:

$$b_a(s) = \sum_{s' \in S} \Pr_a(s|s')b(s')$$

- If we carry out a in b and then observe o, we get to state b_a^o:

$$b_a^o(s) = \frac{\Pr_a(o|s)b_a(s)}{\Sigma_{s' \in S} \Pr_a(o|s')b_a(s')}$$

- The term on the bottom is the probability of observing o after doing a in b.
- Thus actions map between belief states with probability:

$$b_a(o) = \sum_{s' \in S} \Pr_a(o|s')b_a(s')$$

and we want to find a trajectory from b_0 to b_F at minimum cost.

Dynamic programming

- Again we could use greedy search (or any other search technique) to solve POMDPs.
- However, there are more efficient techniques from *dynamic programming* for both MDPs and POMDPs.
- We start from Bellman's *principle of optimality*:

  If a is the best action in s to reach the goal, and s_a is the resulting state, then the optimal cost from s is the optimal cost from s plus the cost of doing a

  $$V^*(s) = \min_{a \in A(s)} [c(a, s) + V^*(s_a)]$$

- This gives us a recursive definition of the optimal cost.

- This can easily be extended to handle MDPs:

$$V^*(s) = \min_{a \in A(s)} [c(a,s) + \sum_{s' \in S} \Pr_a(s'|s) V^*(s')]$$

replacing the cost of the path from s_a with the expected cost across all states that might result from a.

- This search depends upon the heuristic estimate for the expect cost.
- The optimal cost is just $V^*(s)$, so the greedy policy:

$$\pi^*(s) = \arg\text{-}\min_{a \in A(s)} [c(a,s) + \sum_{s' \in S} \Pr_a(s'|s) V^*(s')]$$

is the optimal policy.

- The problem then is to find $V^*(\cdot)$.
- We do this by *value interation*, solving the recursive equation:

$$V^*(s) = \min_{a \in A(s)} [c(a,s) + \sum_{s' \in S} \Pr_a(s'|s) V^*(s')]$$

  for $V^*(\cdot)$ iteratively.
- So:
  - $V_0(s) = 0$;
  - $V_{i+1}(s) = \min_{a \in A(s)}[c(a,s) + \sum_{s' \in S} \Pr_a(s'|s) V_i(s')]$

- Value iteration converges on $V^*(\cdot)$.
- In other words:

$$\lim_{i \to \infty} V_i(s) = V^*(s)$$

- So, if we run the algorithm for long enough, it will give us the optimal value function, and from this we can recover the optimal policy.
- Value iteration can solve MDPs with up to 10^7 states.
- This is enough for many purposes.

- We can combine greedy search with value iteration.
- The algorithm is:
  1. Evaluate each action a applicable in current state s as:

$$Q(s,a) = c(s,a) + \sum_{s' \in S} \Pr_a(s'|s)V_i(s')$$

  2. Apply a that minimises $Q(s,a)$
  3. Update $V(s)$ to $Q(s,a)$.
  4. Observe resulting state s'
  5. Exit if s' is goal, else with $s := s'$ go to 1.

- This process is known as *real-time dynamic programming*.
- $V(s)$ is initialized to $h(s)$
- If h is admissible, and after repeated trials, this greedy policy eventually becomes optimal.
- This is just like the *reinforcement learning* we saw before for learning a heuristic, but adapted for a more realistic environment.
- If h is good, very large problems can be solved this way.

- The same approach can be adopted for POMDPs.
- As we already mentioned, a POMDP is an MDP over belief states:
  - An action a transforms a belief state b into b_a
  - An action a and an observation o map b into b_a^o with probability $b_a(o)$.
- This makes it easy to come up with a RTDP algorithm.

- We have:
  1. Evaluate each action a applicable in current state b as:

$$Q(b,a) = c(b,a) + \sum_{o \in O} b_a(o) V(b_a^o)$$

  2. Apply a that minimises $Q(b,a)$
  3. Update $V(b)$ to $Q(b,a)$.
  4. Observe o
  5. Compute new belief state b_a^o
  6. Exit if b_a^o is final belief state, else with $b := b_a^o$ go to 1.
- POMDPs are much less tractable than MDPs — the state space is way larger.
- Currently POMDPs with ~ 100 states are unsolvable (lots of work on *factoring* state spaces.

Summary

- In this lecture, we have looked at a more sophisticated model of planning than STRIPS.
- Starting from the notion of planning as search, we introduced the Markov decision process representation.
- A solution to an MDP is a *policy*, a choice of what action to take in *every* state.
- We looked at the use of dynamic programming to solve MDPs.