LEARNING IN STATE SPACE

Overview

- The last few lectures have considered heuristic search.
- Obviously the performance of search techniques depends a lot on the heuristic.
- Sometimes we can work out what good heuristics are from our knowledge of the domain.
- When we can't, we can get an agent to learn the right heuristic.
- This lecture looks at techniques for learning such heuristics
- These are all types of *reinforcement learning*.

Learning heuristics

- We will start by assuming that the agent knows the results and costs of each operation.
- We will also assume that it can build the whole search tree.
- This is just what we did for previous searches.
- We then set $h(n) = 0$ for all n and run an A* search.
- When the agent has expanded node n_i to give a set of children $\delta(n_i)$, it updates its $h(n_i)$ to be:

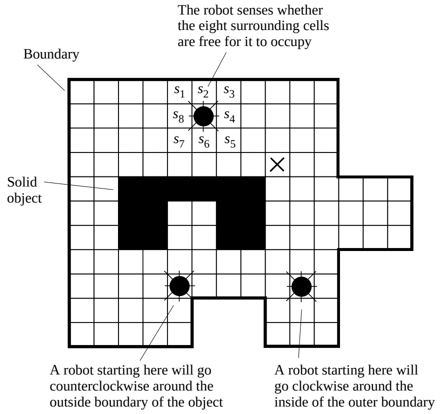
  $$h(n_i) := \min_{nj \in \delta(n_i)} [h(n_j) + c(n_i, n_j)]$$

  where $c(n_i, n_j)$ is the cost of moving from n_i to n_j.
- We further assume that the agent can recognise the goal state and knows that $h(goal)$ is 0.

- This won't do much for the agent the first time–it is just uniform cost search.
- However, subsequent searches will "zoom in" on the right solution faster and faster.
- This happens as the $h_T(n)$ values propagate back from the goal.
- (There are few enough values that these can be stored in a table.)
- Each run propagates the true cost of getting to the goal further back through the search.
- Eventually, the minimal cost path can just be read off the tree.

Learning without a model of action

- As described this kind of search is a ”thought experiment” that an agent carries out.
- In the case of the navigating robot, it is planning its route across the grid.
- Alternatively it would be possible for the agent to actually carry out the operations to see what happens.
- In the case of the robot it could move through the room randomly at first, working out over a number of runs what the outcomes of actions were, and which were most useful at which point.
- (To do this, the agent will have to build a model of the state space in its ”head”).


The robot senses whether the eight surrounding cells are free for it to occupy
Boundary
s_1 s_2 s_3
s_8 s_4
s_7 s_6 s_5
Solid object
A robot starting here will go counterclockwise around the outside boundary of the object
A robot starting here will go clockwise around the inside of the outer boundary

© 1998 Morgan Kaufmann Publishers

- What we assume is that:
  - The agent can distinguish the states it visits (and name them).
  - The agent knows how much actions cost once it has taken them.
- The process starts at the start state s_0.
- The agent then takes an action (maybe at random), and moves to another state. And repeats.
- As it visits each state, it names it and updates the heuristic value of this state as:

  $$h(n_i) := [h(n_j) + c(n_i, n_j)]$$

  where n_i is the node in which an action is taken, n_j is the node the action takes the agent to, and $c(n_i, n_j)$ is the cost of the action.
- $h(n_j)$ is zero if the node hasn’t been reached before.

- Whenever the agent has to choose an action a, it chooses it by:

  $$a = \text{argmin}_a \left[h(\sigma(n_i, a)) + c(n_i, \sigma(n_i, a))\right]$$

  where $\sigma(n_i, a)$ is the state reached from n_i after carrying out a.
- As before, the estimated minimum cost path to the goal is built up over repeated runs.
- However, allowing some randomness in the choice of actions increases the chance that the “estimated minimum cost path” really is the best path.

Learning without a search graph

- For many interesting problems, it is not possible to store all the states/nodes and build the entire search graph.
- Provided we have a model of the effects of actions, we can still search with an evaluation function.
- We start by assembling a heuristic as a linear combination of some set of plausible functions.
- For the 8-puzzle these might be:
  - $W(n)$: number of tiles out of place.
  - $P(n)$: sum of distance each tile is from home.
- Plus any additional functions you can think of.

- Potentially you could consider all the things it is possible to measure.
- Then:

$$h(n) = w_1 W(n) + w_2 P(n) + \ldots$$

- We then learn which weights are best.
- One way to do this is as follows:
- After expanding n_i to $\delta(n_i)$ we adjust the weights so that:

$$h(n_i) := h(n_i) + \beta \left(\min_{n_j \in \delta(n_i)} [h(n_j) + c(n_i, n_j)] - h(n_i) \right)$$

- We modify $h(n_i)$ by adding some proportion of (controlled by β) of the difference between what we thought $h(n_i)$ was before the expansion, and what we think it is after.

- We can rewrite this as:

$$h(n_i) := (1 - \beta) h(n_i) + \beta \min_{n_j \in \delta(n_i)} [h(n_j) + c(n_i, n_j)]$$

- β controls how fast the agent learns—how much weight we give to the new estimate of the heuristic.
- If $\beta = 0$ there is no adjustment.
- If $\beta = 1$, $h(n_i)$ is thrown away immediately.
- Low values of β lead to slow learning, and high values mean that performance is erratic.
- Note that this *temporal difference approach* can also work without a model of the effects of actions (with suitable modification).

Rewards not goals

- For many tasks agents don’t have short term goals, but instead accrue *rewards* over a period of time.
- Instead of a plan, we want a *policy* π which says how the agent should act over time.
- Typically this is expressed as what action should be carried out in a given state.
- We express the reward an agent gets as $r(n_i, a)$, and if doing a in n_i takes the agent to n_j, then:

$$r(n_i, a) = -c(n_i, n_j) + \rho(n_j)$$

  where $\rho(n_j)$ is a reward for being in state n_j.
- We want an optimal policy π^* which maximises the (discounted) reward at every node.

- One way to find the optimum policy is by searching through all possible policies.
- Start with a random policy and calculate its reward.
- Then guess another policy and see if it has a better reward (kind of slow).
- Better would be to tweak the policy by swapping some a in n_i for an a' with a higher $r(n_i, a')$.
- Again there is no guarantee of success.
- But there are better approaches.

- Given a policy π, we can compute the value of each node—the reward the agent will get if it starts at that node and follows the policy.
- If the agent is at n_i and follows π to n_j then the agent will get reward:

  $$V^{\pi}(n_i) = r(n_i, \pi(n_i)) + \gamma V^{\pi}(n_j)$$

  where γ is the discount factor (think of it as the opposite of bank interest).
- The optimum policy then gives us the action that maximises this reward:

  $$V^{\pi^*}(n_i) = \max_a \left[r(n_i, a) + \gamma V^{\pi^*}(n_j)\right]$$

- If we knew what the values of the nodes were under π^*, then we could easily compute the optimal policy:

  $$\pi^*(n_i) = \mathrm{argmax}_a \left[r(n_i, a) + \gamma V^{\pi^*}(n_j)\right]$$

- The problem is that we don't know these values.
- But we can find them out using *value iteration*.
- We start by guessing (randomly is fine) an estimated value $V(n)$ for each node.

- Then when we are at n_i we pick the action to maximise:

  $$\mathrm{argmax}_a \left[r(n_i, a) + \gamma V(n_j)\right]$$

  that is the best thing given what we currently know.
- We then update $V(n_i)$ by:

  $$V(n_i) := (1 - \beta)V(n_i) + \beta \left[r(n_i, a), \gamma V(n_j)\right]$$

- Progressive iterations of this calculation make $V(n)$ a closer and closer approximation to $V^{\pi^*}(n_i)$.
- Intuitively this is because we replace the estimate with the actual reward we get for the next state (and the next state and the next state).

Summary

- This lecture has looked at a number of approaches to learning heuristic functions.
- We started assuming that the agent knew everything but the heuristic, and progressively relaxed assumptions.
- This created a battery of reinforcement learning methods that can be applied in a wide variety of situations.
- These models also tie learning and planning together very closely, and we will revisit them as planning models later in the course.