# 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 of every action in every state; and
  - the costs of every action (in every state).
- We will also assume that it can build the whole search tree.
- This is just what we did for previous searches.
- We further assume that the agent can recognise the goal state and knows that $h(goal)$ is 0.
- We then set $h(n) = 0$ for all $n$ and run an A* search.
- This won't do much for the agent the first time—it is just uniform cost search.



- However, we *update* $h(n)$ as we do the 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:

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

  where $c(n_i, n_j)$ is the cost of moving from $n_i$ to $n_j$.
- With this update, 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.

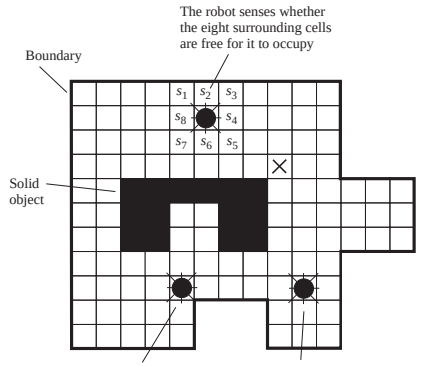
## Learning without a model of action

- 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.
- To do this, the agent has to know what the outcome of all its possible actions are in every state in which it might carry them out.
- This isn't necessarily very realistic.
- Can we make it more realistic?
- Can we do the same kind of learning as we looked at above, but where the agent doesn't have to know what the result of each action is before it carries it out?



- The answer is/are "yes".
- What we have the agent do is to actually carry out the operations to see what happens.
- So rather than say "ah, I'm in state $s_n$, if I do action $a_1$ I will get to $s_m$ and if I do $a_2$ I will get to $s_p$", the agent in $s_n$ picks one of $a_1$ or $a_2$, carries it out, and sees where it ends up.
- This is rather similar to the way in which we learn how to do unfamiliar things.
- Over a number of runs, the agent figures out which moves in which states are good, and concentrates on using those.
- (To do this, the agent will have to build a model of the state space in its "head").










- 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:

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

  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 use the approac we have been discussing.
- 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.
- Here are two ways to do this.



- In the first approach, we use $h(n)$ as above, and search until we find a goal using A*.
- Once we have found a goal, $n_g$, we can set $h(n_g)$ to 0, and back up the real cost of the path we have found to the goal.
- Having done this we have the correct heuristic value of every state on the path to the goal.
- We can use these values as "training examples".
- We can adjust the values of the $w_i$ that determine our $h(n)$ using gradient descent, exactly as when training a TLU.
- To get good values for the $w_i$, we need to repeat this over several searches.

- The second approach is as follows.
- After expanding $n_i$ to $\delta(n_i)$ we adjust the weights $w_i$ so that:

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

- which we can rewrite as:

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

- In other words, we adjust the $w_i$ after every node is expanded rather than waiting until we find the goal.
- To get good values of $w_i$, we again need to keep learning over several searches.



- Note that in the recursive equation on the previous slide, we modify $h(n_i)$ by adding some proportion (controlled by $\beta$) of the difference between what we thought $h(n_i)$ was before the expansion, and what we think it is after.
- $\beta$ 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 $\beta$ 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* $\pi$ 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 $\pi^*$ 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 $\pi$, 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 $\pi$ 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 $\gamma$ 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 $\pi^*$, then we could easily compute the optimal policy:
$$\pi^*(n_i) = \text{argmax}_a \left[ r(n_i, a) + \gamma V^{\pi^*}(n_j) \right]$$
- 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:
$$\text{argmax}_a \left[ r(n_i, a) + \gamma V(n_j) \right]$$
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.