Engineering Tripos Part IB
Paper 8 — Information Engineering
Part D: Planning and Reinforcement Learning

Glenn Vinnicombe
gv@eng.cam.ac.uk

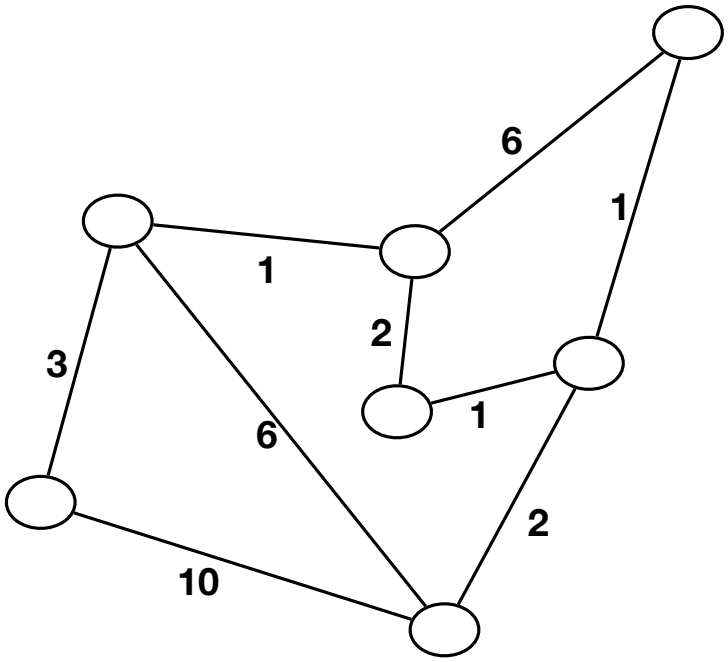
Easter Term 2024

Contents

1 Planning

We start with the simplest planning problem, finding the shortest path on a *graph*.

- Each *node* represents a potential position, or state, of say a robot or a taxi.
- An *edge* between two nodes means that it is possible to move directly from one to the other.
- The *weight* of each edge can represent, for example, the distance, time taken or energy used (or some combination of these) to traverse that edge.



Shortest path problem: Given a *destination* node and an *initial* node, find a path joining them for which the sum of weights is minimal.

To solve this we'll use *Dynamic Programming*, which is a way of breaking down complex problems into simpler subproblems (and very often involves working backwards from the end).

Dynamic Programming is based on Bellman's *Principle of optimality*, which **for the shortest path problem** can be paraphrased as

"If R is a node on the minimal path from P to Q, then that path also contains the minimal paths from P to R and from R to Q"

In particular

$$\text{dist}(u, \text{dest}) = \min_{v \in \text{Neighbours of } u} \text{dist}(u, v) + \text{dist}(v, \text{dest}) \tag{1}$$

This allows us to find all the shortest paths iteratively, working back from the destination, for example:

Once we have a set of distances which satisfy (1) *at all nodes then these must be the shortest distances.*

This informal process, iterating over the edges until the distance estimates stop changing, suffices for hand calculation (and Examples Papers, Exams etc).

1.1 Bellman-Ford Algorithm

To formalise this procedure we can write $\text{dist}_k(u, \text{dest})$ for the shortest distance from u to the destination using no more than k edges. Then clearly

$$\text{dist}_k(u, \text{dest}) = \min_{v \in \text{Neighbours of } u} \text{dist}(u, v) + \text{dist}_{k-1}(v, \text{dest}).$$

and use the following algorithm: [Bellman-Ford]

- Set $\text{dist}_k(\text{dest}, \text{dest}) = 0$ for all k and $\text{dist}_0(u, \text{dest}) = \infty$ when $u \neq \text{dest}$
- For $k = 1$ to $N - 1$:

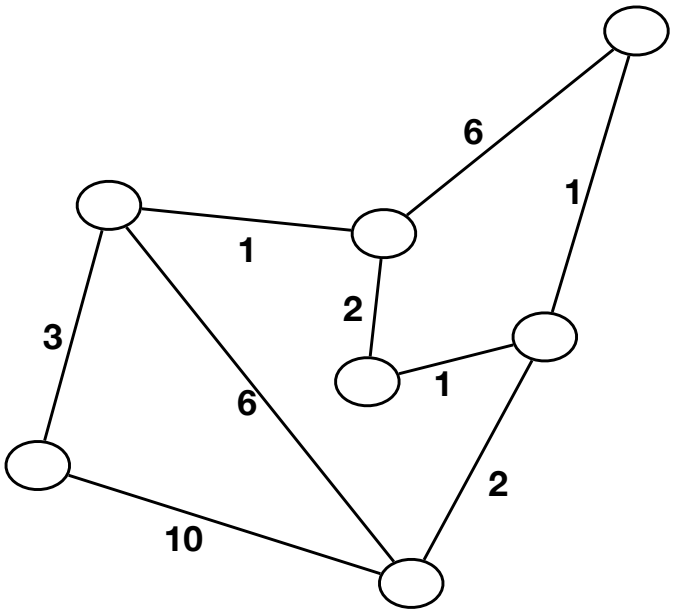
  For each $u \neq \text{dest}$:

  $\text{dist}_k(u, \text{dest}) = \min_{v \in \text{Neighbours of } u} \text{dist}(u, v) + \text{dist}_{k-1}(v, \text{dest})$

where N is the total number of nodes. The longest path possible without cycles is $N - 1$ and so this algorithm is guaranteed to find the shortest paths.

This algorithm also works when some of the lengths (weights) are negative (*as long as there are no negative weight cycles*).

For simplicity we've stated the problem for *undirected graphs* where travel along each edge is allowed in both directions. The algorithm still works for *directed graphs* where the edges have arrows on them (i.e. one way streets - just replace "Neighbours of u" with "nodes with an edge to them from v".

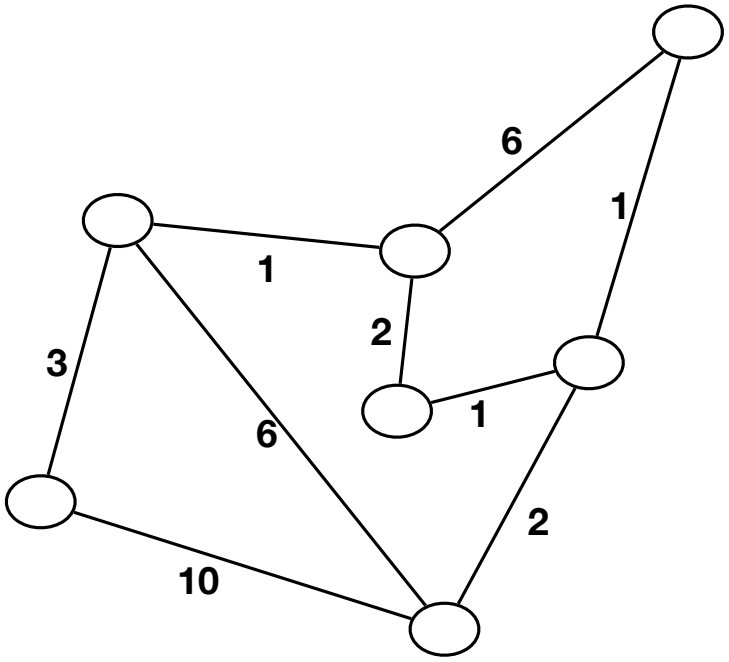


1.2 Dijkstra's Algorithm

If there are no negative weights, then a much faster algorithm is

Dijkstra's Algorithm:

- Mark all nodes as unvisited.
- Assign to every node a tentative distance value: zero for the destination node and infinity for all other nodes.
- Set the destination node to be the *current* node.
- While the current node is not the initial node, repeat the following:
  - Consider all unvisited neighbours of the current node and calculate their distances to the destination node through the current node. Compare this distance to the current assigned tentative distance for that node and assign the smaller one as the new tentative distance.
  - Mark the current node as visited.
  - Set the new current node to be the unvisited node with the smallest tentative distance.



This algorithm works because at each stage the visited set contains the nodes that are closest to the destination and all the shortest paths.

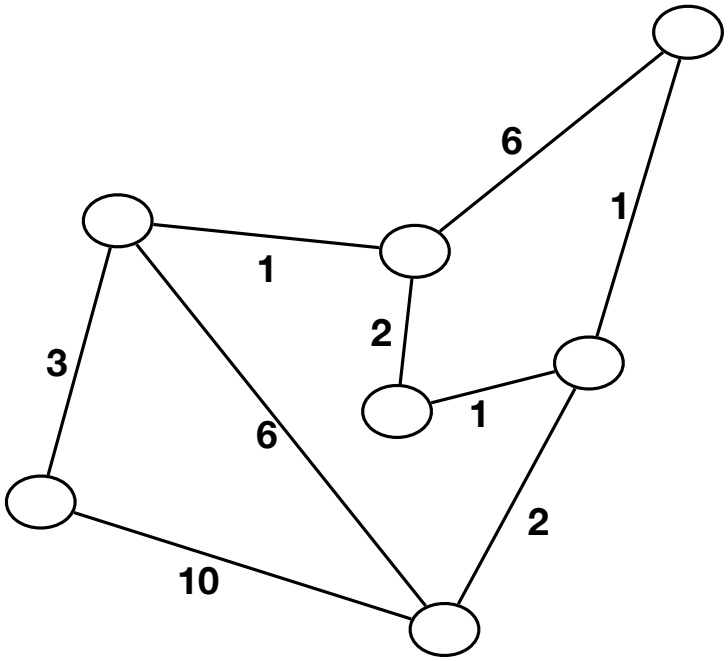
Note that the set of shortest paths form a *tree* (a graph without cycles).

1.3 A* algorithm:

The very widely used A* algorithm is Dijkstra's algorithm plus a heuristic $h(x)$, that has the property that the distance from the *initial* node to x is *at least* $h(x)$

A good choice of heuristic, if the weights correspond to real distances, might be for example distance as the crow flies,

Replace the final step in the loop with "Set the new current node to be the unvisited node x with the smallest *sum of* tentative distance and heuristic $h(x)$."



2 State-Space models and Dynamic programming

2.1 Models

Consider a state space system

$$\underline{s}(t+1) = f\big(\underline{s}(t), \underline{a}(t)\big), \tag{2}$$

where the vector $\underline{s}(t)$ is the *state* of the system at the t'th time step and $\underline{a}(t)$ is the vector of *actions* taken at the t'th time step.

Example: Car Kinematics

Implicitly, there are constraints on the set of actions that can be applied at any state. To keep the notation simple, we regard these constraints as part of the model.

2.2 Optimisation and Value iteration

Let $c\big(\underline{s}(t), \underline{a}(t)\big)$ be the *cost* of applying the action $\underline{a}$ when the current state is $\underline{s}$.

Example: Car Parking

Consider the cost function

$$J(\underbrace{\underline{s}(0)}_{\text{Initial condition}}, \underbrace{\underline{a}(0), \underline{a}(1), ...}_{\text{Actions}}) = \sum_{t=0}^{\infty} \underbrace{c\big(\underline{s}(t), \underline{a}(t)\big)}_{\text{cost}}$$

where $\underline{s}(0)$, $\underline{s}(1)$, ... is the sequence of states generated by $\underline{a}(0)$, $\underline{a}(1)$, ..., according to (2)

We define

$$V(\underline{s}) \triangleq \min_{\underline{a}(0), \underline{a}(1), ...,} J(\underline{s}, \underline{a}(0), \underline{a}(1), ...)$$

that is, the minimal total cost when starting from $\underline{s}(0) = \underline{s}$.

$V(\underline{s})$ is known as the **value function** (or "optimal cost-to-go").

For this minimal cost to be finite there needs to be a *stopping set* X_s

Definition: X_s is a stopping set if for all $\underline{s} \in X_s$ there exists a $\underline{a}$ such that

1. $f(\underline{s}, \underline{a}) \in X_s$
2. $c(\underline{s}, \underline{a}) = 0$

This guarantees a finite cost for all initial states which can be steered to X_s.

This class of problems, where there is a target endpoint X_s but the finishing time is not specified, are known as **Episodic** problems.

Example:

- Note that the shortest path problem is *precisely* a problem of this kind, each node represents a state and an action is a move along an edge, with cost being the weight or length of that edge. The total cost is then the total length.

Just as in the shortest path problem, the value function satisfies the Dynamic Programming (Bellman) equation

$$V(\underline{s}) = \min_{\underline{a}} \big(c(\underline{s}, \underline{a}) + V(f(\underline{s}, \underline{a})) \big)$$

which, as in the shortest path problem, can be solved by **value iteration**

$$V_{k+1}(\underline{s}) = \min_{\underline{a}} \big(c(\underline{s}, \underline{a}) + V_k(f(\underline{s}, \underline{a})) \big)$$

This process is guaranteed to converge for *any* initial guess $V_0(\underline{s})$.

This theorem is the foundation of all optimal control and reinforcement learning.

It is typical to use

$$V_0(\underline{s}) = \begin{cases} 0 & \text{for } \underline{s} \in X_s \\ \infty & \text{otherwise} \end{cases}$$

as in the shortest path problem.

- Note that whenever the number of states and actions are finite, then *any* problem of this kind is a shortest path problem.
- For these problems the setting is normally stochastic, i.e. the resulting $\underline{s}(t+1)$ for a given $\underline{s}(t)$ and $\underline{a}(t)$ is a random variable. $J()$ is then defined as the *expected* cost. This complicates the notation, and requires extra criteria to guarantee convergence, but doesn't change the fundamentals.

- For genuinely infinite horizon problems, with no end in sight, it is often more appropriate to use a *discounted* cost

$$J(\underline{s}(0), \underline{a}(0), \dots) = \sum_{t=0}^{\infty} \lambda^t c\big(\underline{s}(t), \underline{a}(t)\big)$$

  where $\lambda < 1$ is known as a discount factor.

  For such problems the Bellman optimality condition is that

$$V(\underline{s}) = \min_{\underline{a}} \big(c(\underline{s}, a) + \lambda V(f(\underline{s}, \underline{a}))\big)$$

  which can be also solved by value iteration

$$V_{k+1}(\underline{s}) = \min_{\underline{a}} \big(c(\underline{s}, \underline{a}) + \lambda V_k(f(\underline{s}, \underline{a}))\big)$$

- **Rewards:** Instead of a cost $c(\underline{s}, \underline{a})$ we can instead think of maximising a discounted sums of *rewards* $r(\underline{s}, \underline{a})$.

  Then

$$V(\underline{s}) = \max_{\underline{a}} \big(r(\underline{s}, \underline{a}) + \lambda V(f(\underline{s}, \underline{a}))\big)$$

  in the discounted case.

2.3 MPC

One popular approach to planning is to choose actions to maximise the total reward over a horizon of length N, with an approximation to represent the "reward-to-go" at the end, i.e. find

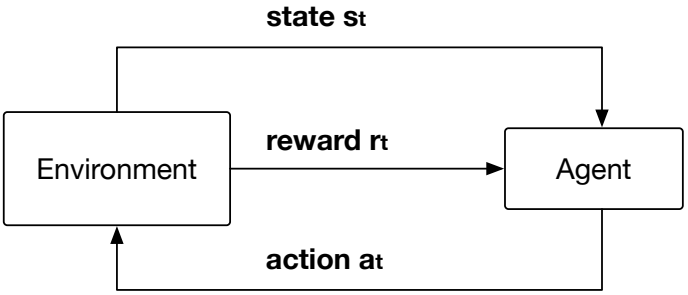
$$\max_{\underline{a}(t), \underline{a}(t+1), \dots, \underline{a}(t+N)} r(t) + \lambda r(t+1) + \lambda^2 r(t+2) + \cdots + \lambda^N r(t+N) + \lambda^{N+1} \tilde{V}(\underline{s}(t+N+1))$$

(subject, as ever, to $\underline{s}(t+1) = f\big(\underline{s}(t), \underline{a}(t)\big)$), where $\tilde{V}()$ is an approximation of the true value function. N is the control horizon, and the larger N is, the less important it is that $\tilde{V}()$ is accurate. The action $\underline{a}(t)$ is applied, $r(t)$ and $\underline{s}(t+1)$ observed, and the process to repeated to choose $\underline{a}(t+1)$.

This technique is variously known as Nonlinear Model Predictive Control or Receding Horizon Control, or various combinations of those words.

3 Reinforcement Learning: Learning from samples

Until now we've assumed knowledge of a model, $\underline{s}(t+1) = f\big(\underline{s}(t), \underline{a}(t)\big)$, reward= $r\big(\underline{s}(t), \underline{a}(t)\big)$. But what if there's no model, all you can do is choose an action and get a reward and be told the next state. How do you optimise your choice of actions to maximise the total reward?



3.1 Exploration vs Exploitation: Multi armed bandits

First we consider the situation where there is no state, i.e. we choose an action $\underline{a}(t)$, get a reward $r(t)$ then choose the next action $\underline{a}(t+1)$ etc, with the aim of maximising the sum of the rewards.

For this problem to be interesting, the reward needs to be random, with a probability distribution that depends on the action.

The multi armed bandit:

There are N different available actions $\{1, 2, \ldots, N\}$.

For each action a, the reward is sampled from the *unknown* probability distribution $p[r|a]$.

Assume that $p[r|a] = 0$ for $r > 1$ (i.e. the maximum reward each step is no greater than one.

At each step you choose an action $a_k \in \{1, 2, \ldots, N\}$

and receive a reward $r_k \in [0, 1] \sim p[r|a]$

The objective is to maximise $\sum_{k=1}^{K} r_k$ (i.e. run for K time steps).

If you knew the distributions, the optimal reward per step would be

$$V^* = \max_a E(r|a)$$

with corresponding optimal action a^*.

The total *regret* (opportunity lost) due to not knowing the distributions is given by

$$L_K = E\left[\sum_{k=1}^{K}(V^* - r_k)\right]$$

Lai and Robbins showed that for large K, and any policy

$$L_K \geq C \log K$$

(and give a formula for C).

Is there an algorithm that gets close to this?

Yes!

3.1.1 UCB1

Let $N_k(a)$ be the total number of times that action a has been chosen so far, and let $W_k(a)$ be the total reward obtained when action a has been chosen.

First take each action once.

Subsequently, choose the action which maximises

$$\frac{W_k(a)}{N_k(a)} + \sqrt{\frac{2\log k}{N_k(a)}}$$

This is called the UCB1 (upper confidence bound 1) algorithm, and for large K it achieves

$$L_K \leq C_1 \log K$$

(This works because, for each arm,

$$Pr\left[E(r|a) > \frac{W_k(a)}{N_k(a)} + \sqrt{\frac{2\log k}{N_k(a)}}\right] < 1/k^4$$

which itself follows from Hoeffding's inequality).

3.1.2 ϵ-greedy

Choose c and let $\epsilon_k = \min(1, c/k)$

At each step k, choose the "best" arm so far with probability $1 - \epsilon_k$ and a random arm with probability ϵ_k.

This algorithm also achieves logarithmic regret for large enough c.

So, for each of these algorithms we obtain, on average, $\max_a E(r, |A) - \text{const} \times \frac{\log(K)}{K}$ per "go".

4 Learning from samples II

4.1 Q-learning

One could potentially learn the value function from samples $\underline{s}(t)$, $\underline{a}(t)$, $r(t)$, $\underline{s}(t+1)$. But this is not straightforward, and even then you need both the model and the reward function to recover the optimal action $\underline{a}^*$, using

$$\underline{a}^*(x) = \arg\max_{\underline{a}} \big(r(\underline{s}, \underline{a}) + V(f(\underline{s}, \underline{a}))\big)$$

Instead the *action-value function* $Q(\underline{s}, \underline{a})$ is used:

Define the action value function $Q(\underline{s}, \underline{a})$ as

$$Q(\underline{s}, \underline{a}) = r(\underline{s}, \underline{a}) + V(f(\underline{s}, \underline{a}))$$

That is, Q is the reward of using action $\underline{a}$ now, and the optimal $\underline{a}$ thereafter.

It follows that

$$V(\underline{s}) = \max_{\underline{a}} Q(\underline{s}, \underline{a})$$

So the Q-function itself satisfies the recursion

$$Q(\underline{s}, \underline{a}) = r(\underline{s}, \underline{a}) + \max_{\underline{a}'} Q(f(\underline{s}, \underline{a}), \underline{a}')$$

The optimal action $\underline{a}^*(\underline{s})$ can now be recovered as

$$\underline{a}^*(\underline{s}) = \max_{\underline{a}} Q(\underline{s}, \underline{a})$$

This is a very clever idea,

In the case of discounted rewards $Q(\underline{s}, \underline{a})$ is defined as

$$Q(\underline{s}, \underline{a}) = r(\underline{s}, \underline{a}) + \lambda V(f(\underline{s}, \underline{a}))$$

and satisfies the recursion

$$Q(\underline{s}, \underline{a}) = r(\underline{s}, \underline{a}) + \lambda \max_{\underline{a}'} Q(f(\underline{s}, \underline{a}), \underline{a}')$$

Q-learning, an algorithm for finding Q, is equally clever:

4.1.1 Q learning

Assume there are finite number of state-action pairs.

First we consider the *Deterministic case:*

Take samples $\underline{s}(i)$, $\underline{a}(i)$, $r(i)$ $\underline{s}(i+1)$ and update

$$Q_{k+1}(\underline{s}(i), \underline{a}(i)) = r(i) + \lambda \max_{\underline{a}'} Q_k(\underline{s}_{i+1}, \underline{a}')$$

(and $Q_{k+1}(\underline{s}, \underline{a}) = Q_k(\underline{s}, \underline{a})$ for all other $\underline{s}, \underline{a}$). Provided each state and action is visited infinitely often then $Q_k \to Q$ as k increases.

This is called a *tabular* method, as it involves maintaining a table of Q-values.

The recursion "joins up" the samples into optimal trajectories.

Note, for stochastic problems the algorithm

$$Q_{k+1}(\underline{s}(i), \underline{a}(i)) = (1 - \alpha_k) Q_k(\underline{s}(i), \underline{a}(i)) + \alpha_k (r(i) + \lambda \max_{\underline{a}'} Q_k(\underline{s}_{i+1}, \underline{a}'))$$

needs to used, where α is known as the learning rate.

To guarantee convergence, α needs to be decreased gradually to zero, but not too gradually, so that $\sum_k \alpha_k^2 < \infty$ but $\sum_k \alpha_k = \infty$.

One possibility is $\alpha_k = 1/k$.

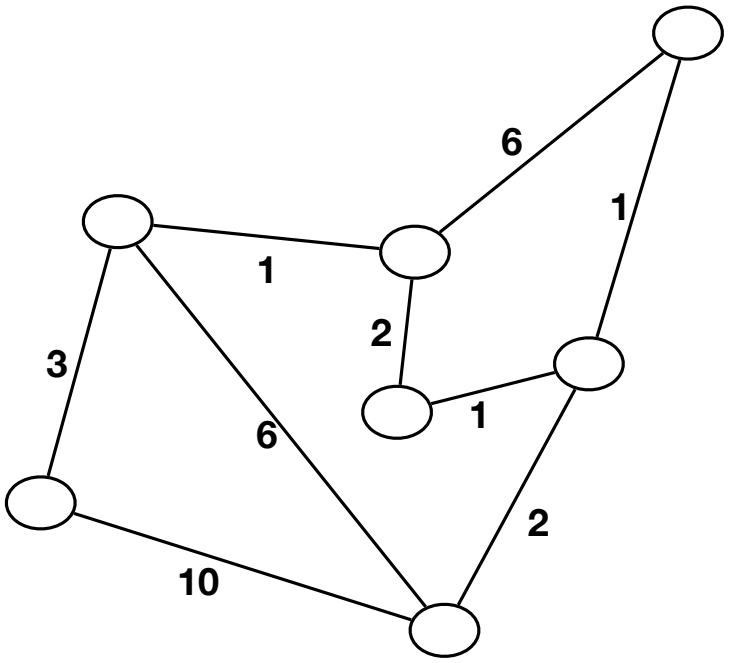
Q-learning is an *off policy* method, ie the experiences need not come from the current policy.

In fact, it is possible to just pick random $\underline{s}$ and $\underline{a}$ pairs, observe $\underline{s}'$ and r, and then apply the update.

More often an ϵ-*greedy* policy is used.

$$\underline{a}(t) = \begin{cases} \arg\max_{\underline{a}} Q(\underline{s}(t), \underline{a}) & \text{with probability } 1 - \epsilon \\ \text{a random exploratory action} & \text{with probability } \epsilon \end{cases}$$

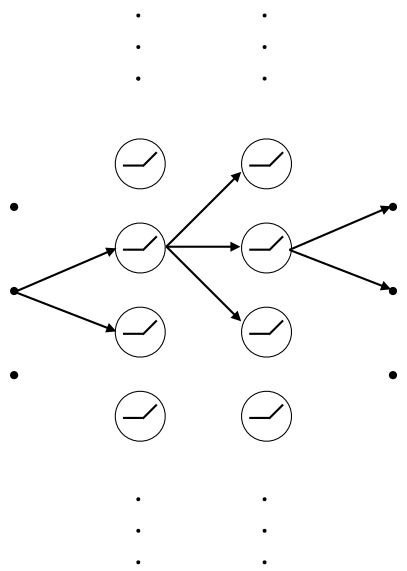
4.1.2 Q-learning for the shortest path problem - a rubbish idea!



4.2 "Deep" Q-learning

For small problems (ie low state dimension, few actions) it is sometimes feasible to discretise (grid) over the states and the actions and use tabular Q-learning directly.

For more complex problems it is necessary to use a functional approximation $Q_\theta(\underline{s}, \underline{a})$, parameterised by the parameters θ. One (popular) possibility is a deep neural net:



In the discounted case θ could then be continuously adjusted to minimise

$$L(\theta) = \mathrm{E}\big(r(t) + \lambda \max_{\underline{a}'} Q_\theta(\underline{s}(t+1), \underline{a}') - Q_\theta(\underline{s}(t), \underline{a}(t))\big)^2$$

using *stochastic gradient descent* (which means that rather than the whole history being used, a selection of the the agents *experiences*, ie the quartets $e_i = (\underline{s}(t), \underline{a}(t), r(t), \underline{s}(t+1))$, are randomly sampled from a *replay buffer*).

$$\overbrace{\underbrace{\underline{s}(0), \underline{a}(0), r(0), \underline{s}(1)}_{e_0}, \underbrace{\underline{s}(1), \underline{a}(1), r(1), \underline{s}(2)}_{e_1}, \ldots, \underbrace{\underline{s}(97), \underline{a}(97), r(97), \underline{s}(98)}_{e_{97}}}^{\text{epsidode 1}}, \overbrace{\underbrace{\underline{s}(0), \underline{a}(0), r(0), \underline{s}(1)}_{e_{98}}, \ldots}^{\text{epsidode 2}}, \text{etc}$$

(Recall that the optimal Q function satisfies

$$r(t) + \lambda \max_{\underline{a}'} Q(\underline{s}(t+1), \underline{a}') - Q(\underline{s}(t), \underline{a}(t)) = 0 \quad)$$

In the stochastic case it has been observed that this approach leads to problems with convergence. Better results have been obtained by setting

$$y(t) = r(t) + \lambda \max_{\underline{a}'} Q_\theta(\underline{s}(t+1), \underline{a}')$$

and then minimising

$$L(\theta) = \mathrm{E}\big(y(t) - Q_\theta(\underline{s}(t), \underline{a}(t))\big)^2$$

The gradient $dL/d\theta$ is calculated by *back propagation* (ie the chain rule),

θ is then updated in the direction $-dL/d\theta$ and process repeated (i.e. a new minibatch of past experiences sampled and the gradient calculated again).

Q-learning is an *off policy* method, ie the experiences need not come from the current policy.

Often an ϵ-*greedy* policy is used.

$$\underline{a}(t) = \begin{cases} \arg\max_{\underline{a}} Q(\underline{s}(t), \underline{a}) & \text{with probability } 1-\epsilon \\ \text{a random exploratory action} & \text{with probability } \epsilon \end{cases}$$

see "Human-level control through deep reinforcement learning" Mnih et al, Nature volume 518, pages 529–533 (26 February 2015)

4.3 Actor-critic methods

For continuous action spaces (i.e. real valued actions, as in most control problems) you could discretise over the actions, but things quickly blow up.

One could put $\underline{a}$ as an input to the neural net, but then it is difficult to find $\max_{\underline{a}} Q(\underline{s}(t), \underline{a})$ (which needs to be evaluated very often).

One possibility is to add a second neural network Π_w to represent the policy (the "actor", with the Q network we have already discussed being the "critic").

As before

$$L_Q(\theta) = \mathrm{E}\left[y(t) - Q_\theta(\underline{s}(t), \underline{a}(t))\right]^2$$

but now, each

$$y(t) = r(t) + \lambda Q_\theta(\underline{s}(t+1), \Pi_w(\underline{s}(t+1)))$$

but we now add a second objective to *maximise*:

$$L_\Pi(w) = \mathrm{E}\left[Q_\theta(\underline{s}(t), \Pi_w(\underline{s}(t)))\right]$$

The algorithm alternates between updating θ, to reduce $L_Q(\theta)$, and updating w to increase $L_\Pi(w)$.

4.4 How AlphaZero chooses its moves

AlphaZero uses a type of Monte Carlo Tree Search (MCTS) to choose it's actions.

(a tree is a graph without cycles)

At each stage of the game, it repeatedly explores the tree from the current position s_k, taking exploratory actions/moves until it finds a position it hasn't evaluated before, which it then evaluates using its neural net to give a value V and stochastic policy p (a vector of probabilities for each possible move).

$$V, p = f_\theta(s)$$

$f_\theta()$ is a convolutional neural net, with weights θ. Each input plane is an 8x8 image showing the position of one kind and colour of piece.

Each edge of the tree corresponds to a position s and an action/move a away from it.

With each edge is stored $Q(s,a)$, the average value of all positions explored below, $N(s,a)$, the number of times this action has been tried and $p(s,a)$.

At each point, the exploratory action taken is the one which maximises

$$Q(s,a) + c.p(s,a)\frac{\sqrt{\sum_{a'} N(s,a')}}{1+N(s,a)}$$

When the time is up (for choosing that move) the preference for moves from the current position s_k is totted up

$$\pi(s_k,a) = \frac{N(s_k,a)}{\sum_{a'} N(s_k,a')}$$

and stored along with the position. The actual move to be played is then selected according to $\pi(s_k,a)$.

4.4.1 Training

When a game is completed, each position is marked with the true result of that game (1 for a win, -1 for a loss, 0 for a draw) and stored in a buffer. The system plays many games against itself, and the periodically the network is trained, by stochastic gradient descent using batches sampled from the buffer, to minimise the *loss* function

$$(Z-V)^2 - \pi.\log(p)$$

and thats it!