Advanced Topics Data Science and AI
Automated Planning and Acting

Probabilistic Models

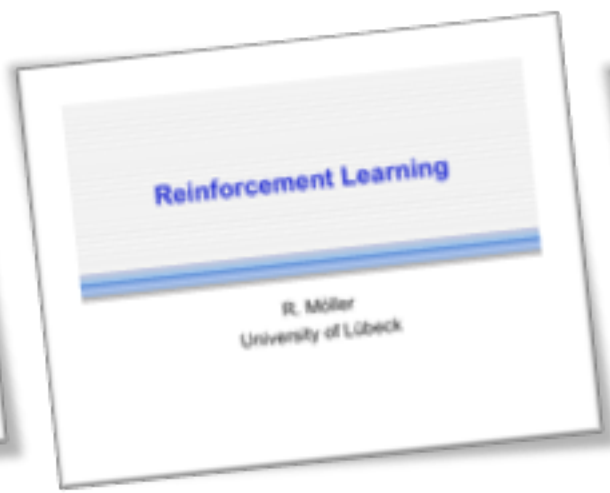
Tanya Braun

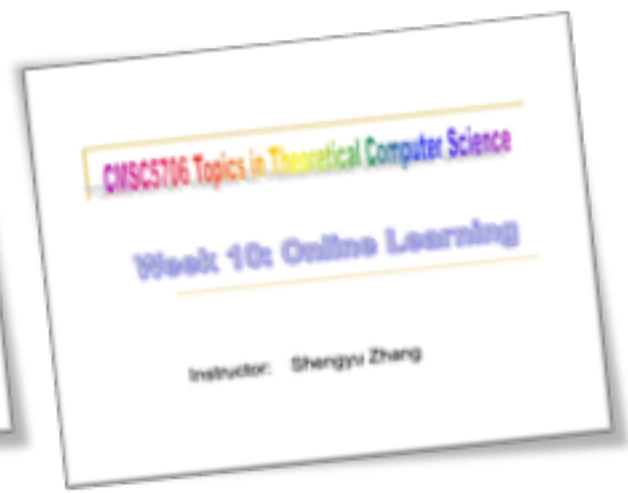
UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR INFORMATIONSSYSTEME

Content

Acknowledgements

- Automated Planning and Acting Chapter 6
- Slides based on material provided by Dana Nau, Ralf Möller, and Shengyu Zhang





Outline

6.2 Stochastic shortest path problems

- Safe/unsafe policies
- Optimality
- Policy iteration, value iteration

6.3 Heuristic search algorithms

- Best-first search
- Determinisation

6.4 Online probabilistic planning

- Lookahead
- Reinforcement learning

Probabilistic Planning Domain

- $\Sigma = (S, A, \gamma, P, cost)$
  - S = set of states
  - A = set of actions
  - $\gamma : S \times A \rightarrow 2^S$ a transition function
  - $P(s' \mid s, a)$ = probability of going to state s' if we perform a in s
    - Require $P(s' \mid s, a) \neq 0$ iff $s' \in \gamma(s, a)$
  - $cost$: $S \times A \rightarrow \mathbb{R}^{>0}$
    - $cost(s, a)$ = cost of action a in state s
    - may omit, default is $cost(s, a) = 1$

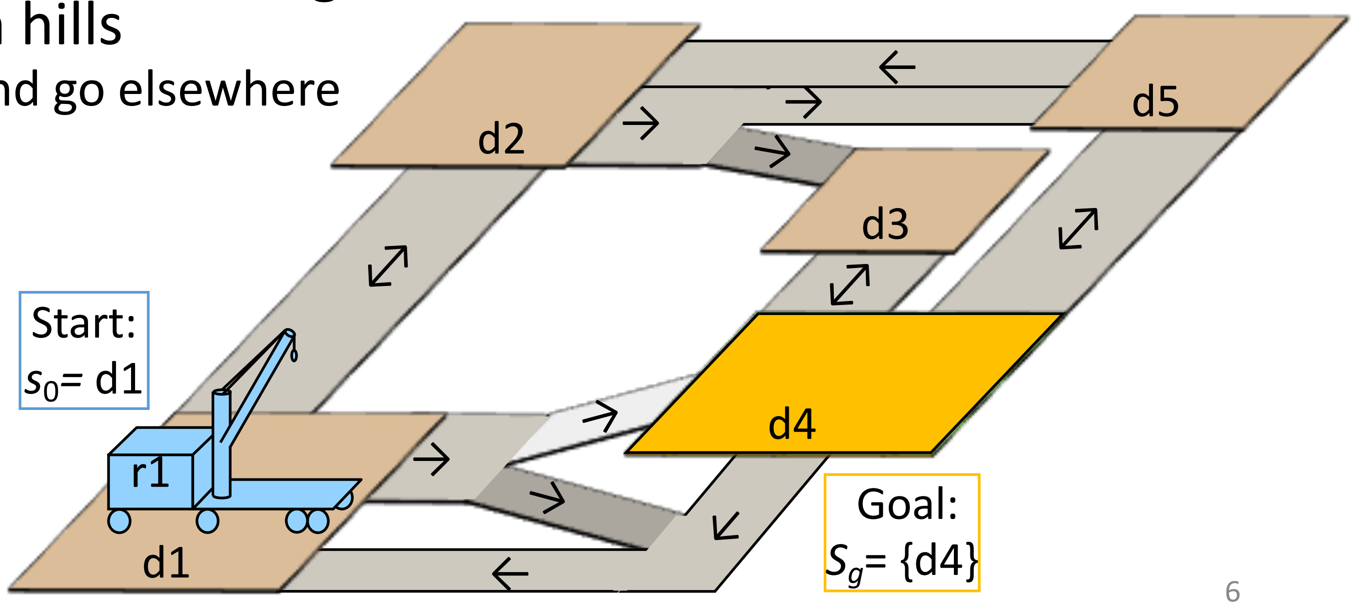
Difference in syntax: MDPs do not have an explicit transition function γ, only a set of applicable actions $A(s)$ per state and the transition model $P(s' \mid s, a)$

Instead of maximising expected utility as before:
Minimise expected cost

Example

- Robot $r1$ starts at $d1$
- Objective: get to $d4$
- Simplified state names: write $\{loc(r1) = d2\}$ as $d2$
- Simplified action names: write $move(r1, d2, d3)$ as $m23$
- $r1$ has unreliable steering, especially on hills
  - May slip and go elsewhere

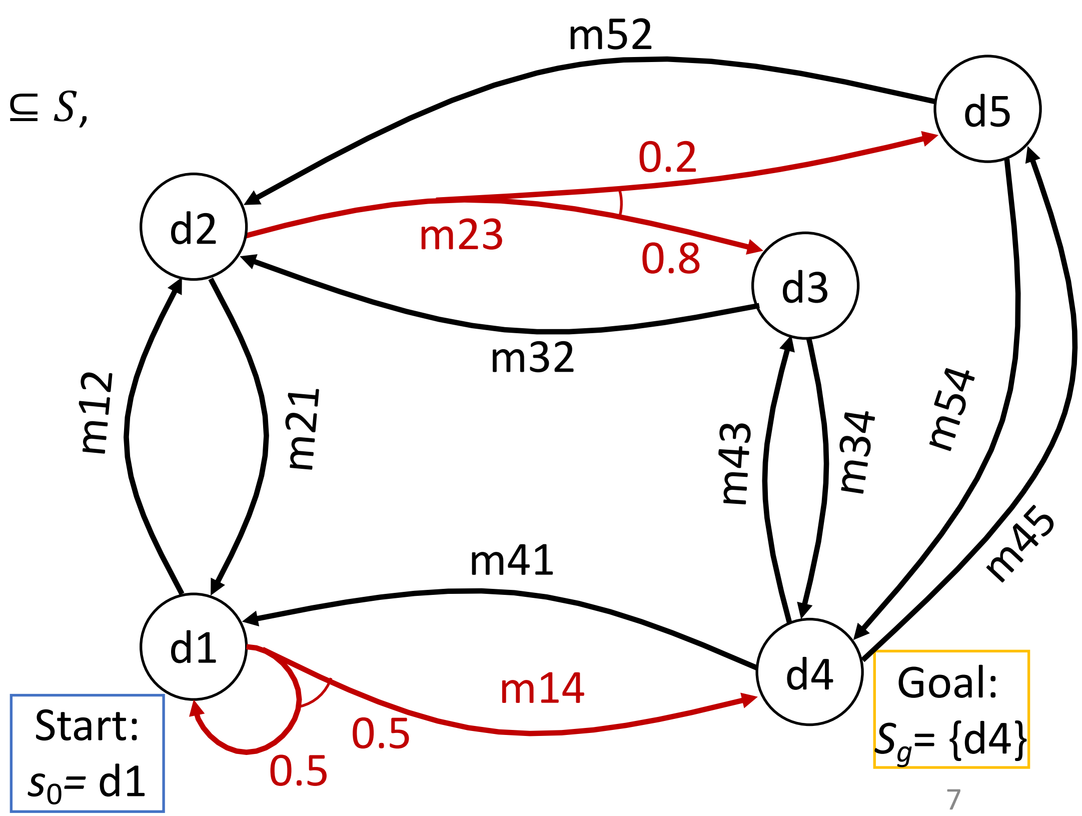
- $m12$: $\mathrm{P}(d2 \mid d1, m12) = 1$
- $m21, m34, m41, m43, m45, m52, m54$: like above
- $m14$: $\mathrm{P}(d4 \mid d1, m14) = 0.5$
  $\mathrm{P}(d1 \mid d1, m14) = 0.5$
- $m23$: $\mathrm{P}(d3 \mid d2, m23) = 0.8$
  $\mathrm{P}(d5 \mid d2, m23) = 0.2$



Policies, Problems, Solutions

- Stochastic shortest path (SSP) problem:
  - a triple (Σ, s_0, S_g)
- Policy:
  - partial function $\pi : S \to A$ s.t.
    - for every $s \in Dom(\pi) \subseteq S$, $\pi(s) \in Applicable(s)$
- Solution for (Σ, s_0, S_g) :
  - a policy π s.t.
    - $s_0 \in Dom(\pi)$ and
    - $\hat{\gamma}(s_0, \pi) \cap S_g \neq \emptyset$

- $m14$: $P(d4 \mid d1, m14) = 0.5$
  $P(d1 \mid d1, m14) = 0.5$
- $m23$: $P(d3 \mid d1, m23) = 0.8$
  $P(d5 \mid d1, m23) = 0.2$



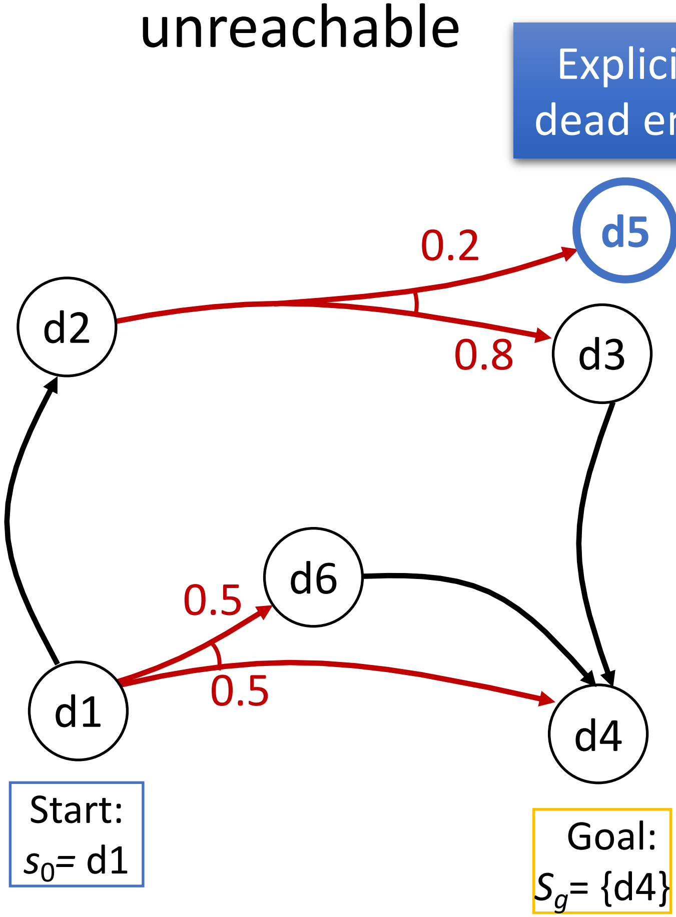
Notation and Terminology

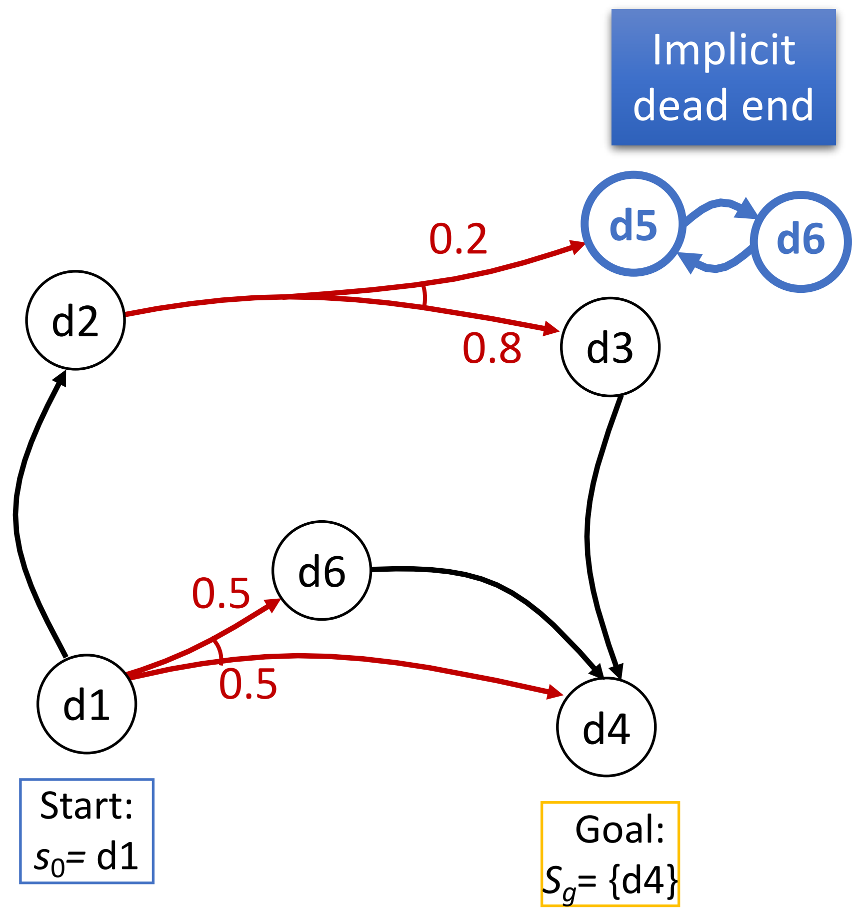
- Transitive closure
  - $\hat{\gamma}(s,\pi)$ = {s and all states reachable from s using π}
- $Graph(s,\pi)$ = rooted graph induced by π at s
  - Nodes: $\hat{\gamma}(s,\pi)$
  - Edges: state transitions
- $leaves(s,\pi) = \hat{\gamma}(s,\pi) \backslash Dom(\pi)$

- A solution policy π is closed if it does not stop at non-goal states unless there is no way to continue
  - for every state $s \in \hat{\gamma}(s,\pi)$, either
    - $s \in Dom(\pi)$ (i.e., π specifies an action at s),
    - $s \in S_g$ (i.e., s is a goal state), or
    - $Applicable(s) = \emptyset$ (i.e., there are no applicable actions at s)

Dead Ends

- Dead end
  - A state or set of states from which the goal is unreachable

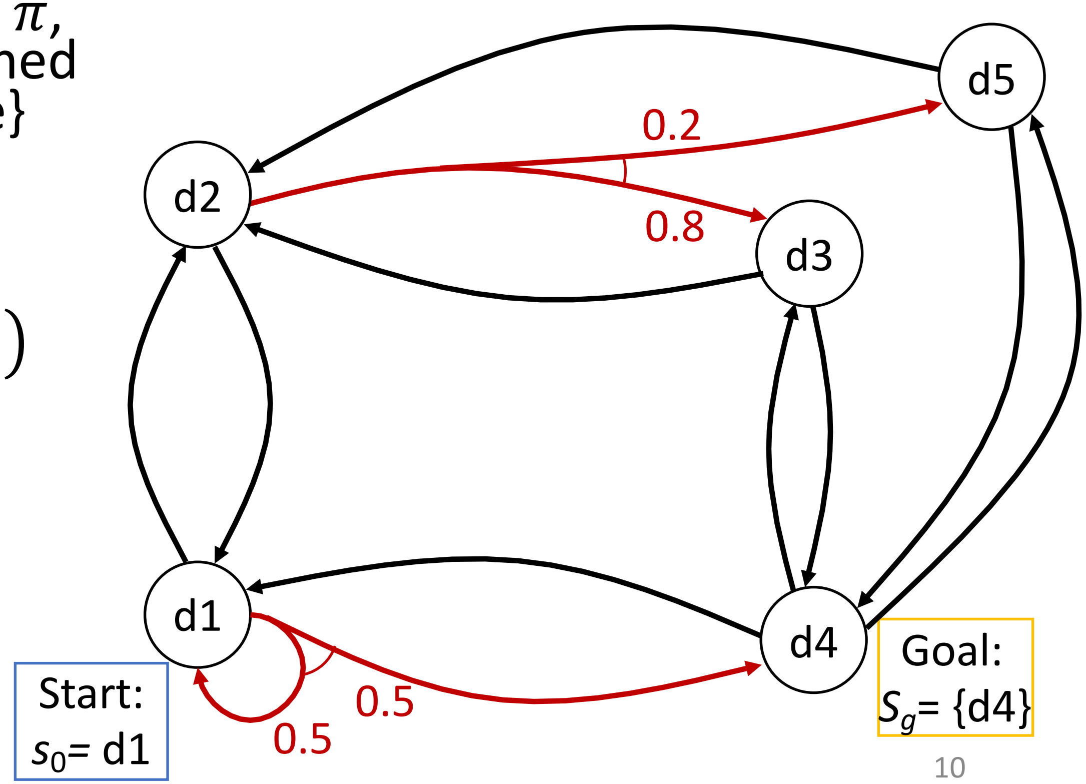




Histories

- History: sequence of states $\sigma = \langle s_0, s_1, s_2, \dots \rangle$
  - May be finite or infinite
    - $\sigma = \langle d1, d2, d3, d4 \rangle$
    - $\sigma = \langle d1, d2, d1, d2, \dots \rangle$
- $H(s, \pi)$ = {all possible histories if we start at s and follow π, stopping if $\pi(s)$ is undefined or if we reach a goal state}
- If $\sigma \in H(s, \pi)$, then
  $$P(\sigma \mid s, \pi) = \prod_i P\big(s_{i+1} \mid s_i, \pi(s_i)\big)$$
  - Thus
    $$\sum_{\sigma \in H(s,\pi)} P(\sigma \mid s, \pi) = 1$$

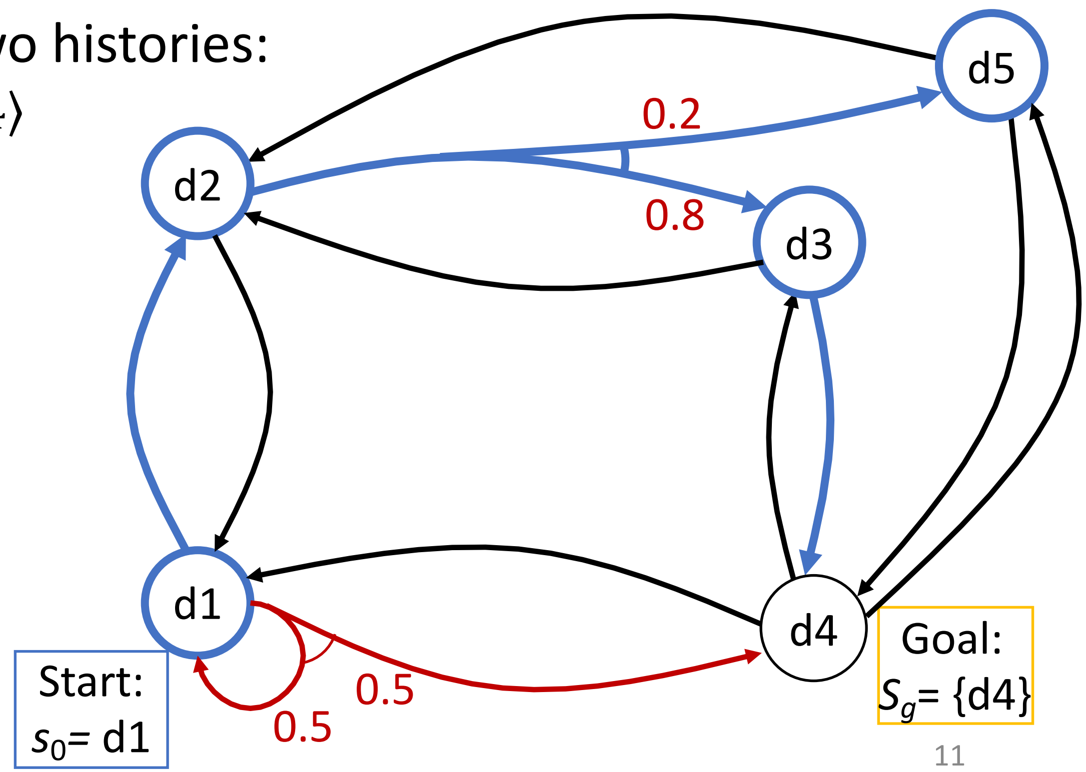
- Probability of reaching a goal:
  $$P\big(S_g \mid s, \pi\big) = \sum_{\substack{\sigma \in H(s,\pi), \\ \sigma \text{ ends at a state in } s_g}} P(\sigma \mid s, \pi)$$



Unsafe Solutions

- Unsafe solution: $0 < P(S_g|s_0, \pi) < 1$

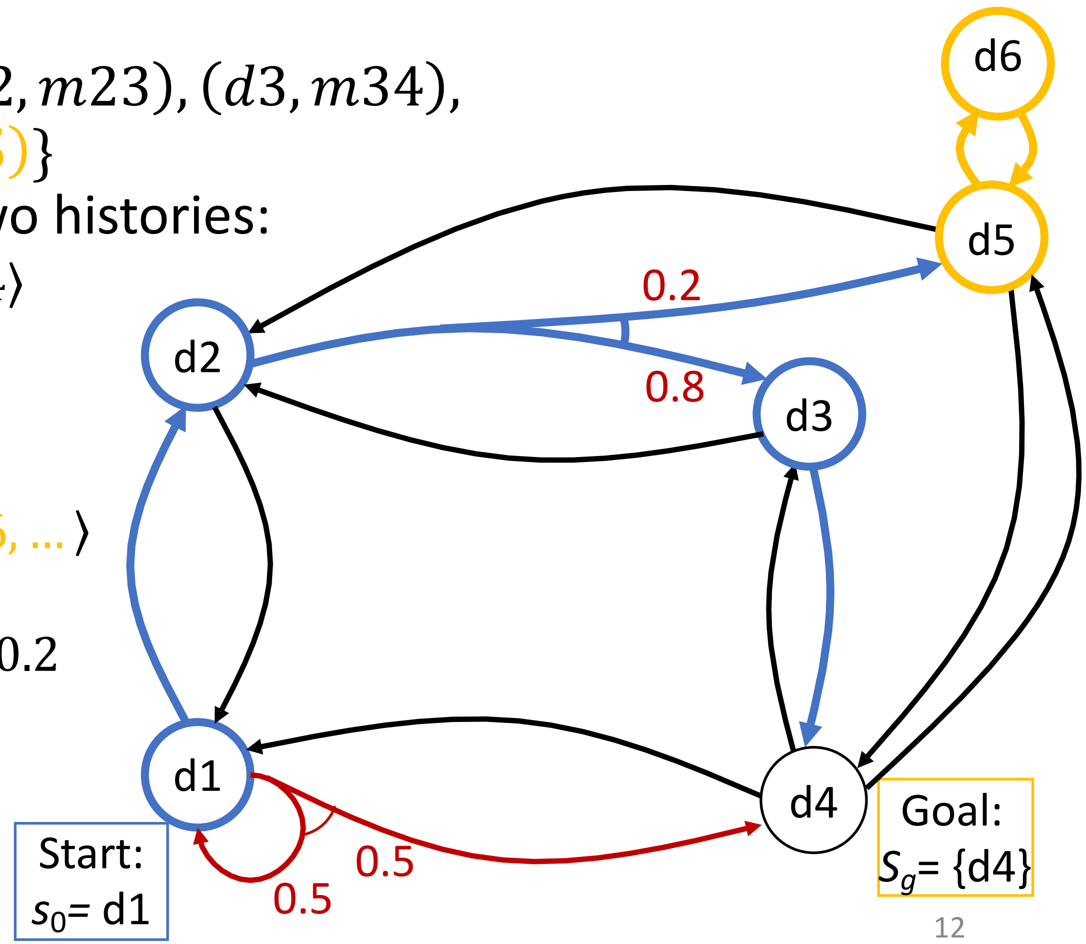
- Example:
  - $\pi_1 = \{(d1, m12), (d2, m23), (d3, m34)\}$
  - $H(s_0, \pi_1)$ contains two histories:
    - $\sigma_1 = \langle d1, d2, d3, d4 \rangle$
    - $P(\sigma_1|s_0, \pi_1)$
      $= 1 \cdot 0.8 \cdot 1 = 0.8$
    - $\sigma_2 = \langle d1, d2, d5 \rangle$
    - $P(\sigma_2|s_0, \pi_1)$
      $= 1 \cdot 0.2 = 0.2$
  - $P(S_g|s_0, \pi_1)$
    $= 0.8$



Unsafe Solutions

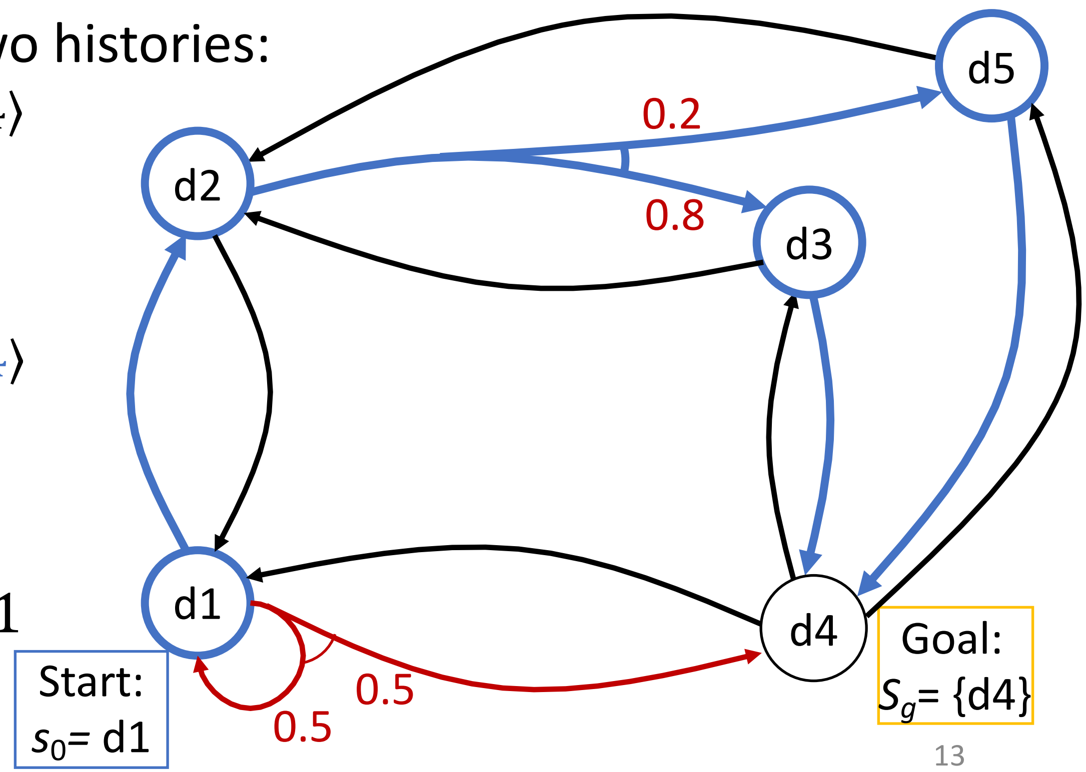
- Unsafe solution: $0 < P\left(S_g \middle| s_0, \pi\right) < 1$

- Example:
  - $\pi_2 = \{(d1, m12), (d2, m23), (d3, m34),$ $(d5, m56), (d6, m65)\}$
  - $H(s_0, \pi_2)$ contains two histories:
    - $\sigma_1 = \langle d1, d2, d3, d4 \rangle$
    - $P(\sigma_1 | s_0, \pi_2)$ $= 1 \cdot 0.8 \cdot 1 = 0.8$
    - $\sigma_3 = \langle d1, d2, d5, d6, \ldots \rangle$
    - $P(\sigma_3 | s_0, \pi_2)$ $= 1 \cdot 0.2 \cdot 1 \cdot \cdots = 0.2$
  - $P\left(S_g \middle| s_0, \pi_2\right)$ $= 0.8$



Safe Solutions

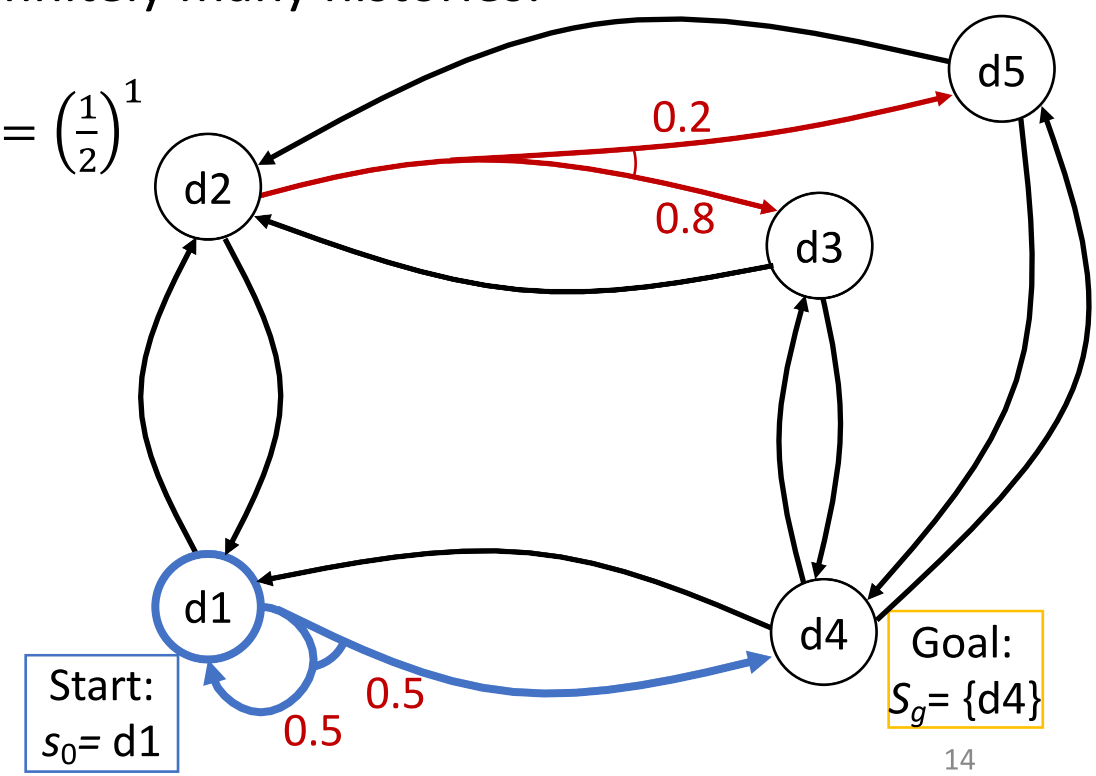
- Safe solution: $P\left(S_g \middle| s_0, \pi\right) = 1$
- An acyclic safe solution:
  - $\pi_3 = \{(d1, m12), (d2, m23), (d3, m34), (d5, m54)\}$
  - $H(s_0, \pi_3)$ contains two histories:
    - $\sigma_1 = \langle d1, d2, d3, d4 \rangle$
    - $P(\sigma_1 | s_0, \pi_3)$
      $= 1 \cdot 0.8 \cdot 1 = 0.8$
    - $\sigma_4 = \langle d1, d2, d5, d4 \rangle$
    - $P(\sigma_4 | s_0, \pi_3)$
      $= 1 \cdot 0.2 \cdot 1 = 0.2$
  - $P\left(S_g \middle| s_0, \pi_3\right)$
    $= 0.8 + 0.2 = 1$



Safe Solutions

- Safe solution: $P(S_g|s_0, \pi) = 1$

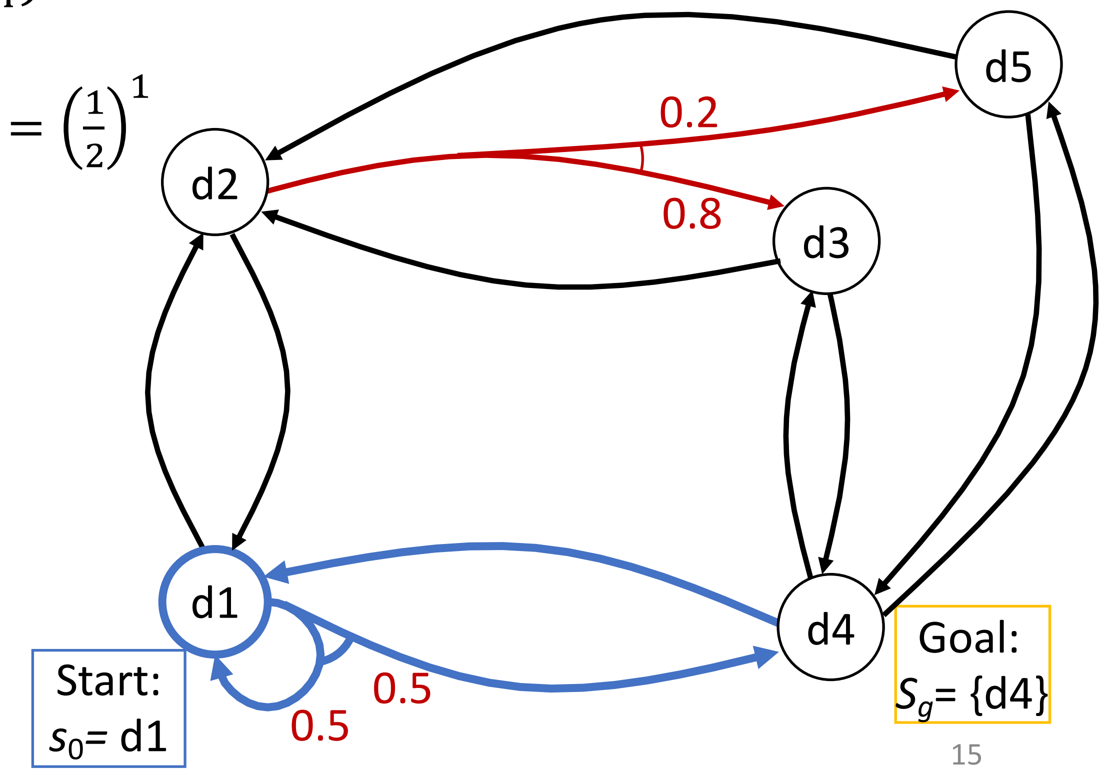
- A cyclic safe solution:
  - $\pi_4 = \{(d1, m14)\}$
  - $H(s_0, \pi_4)$ contains infinitely many histories:
    - $\sigma_5 = \langle d1, d4 \rangle$
    - $P(\sigma_5|s_0, \pi_4) = 0.5 = \left(\frac{1}{2}\right)^1$
    - $\sigma_6 = \langle d1, d1, d4 \rangle$
    - $P(\sigma_6|s_0, \pi_4)$ $= 0.5 \cdot 0.5 = \left(\frac{1}{2}\right)^2$
    - …
  - $P(S_g|s_0, \pi_4)$ $= \frac{1}{2} + \frac{1}{4} + \ \ldots = 1$



Safe Solutions

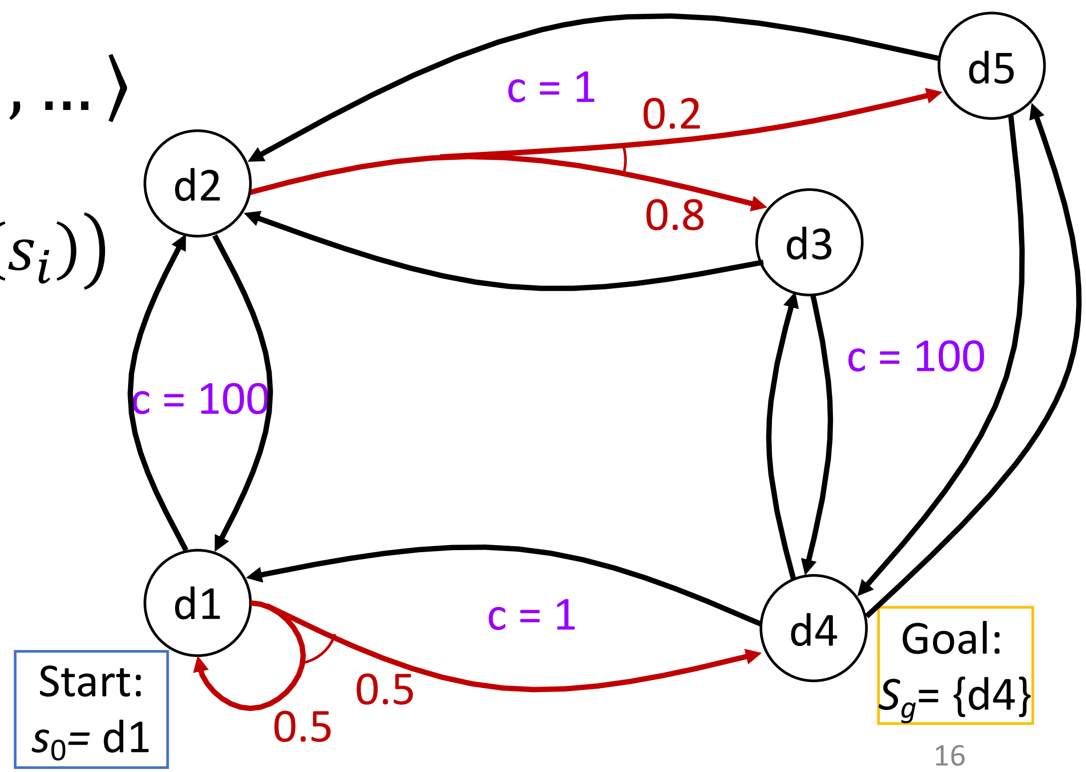
- Safe solution: $P\left(S_g|s_0,\pi\right) = 1$

- Another cyclic safe solution:
  - $\pi_5 = \{(d1, m14), (d4, m41)\}$
  - $H(s_0, \pi_5) = H(s_0, \pi_4)$:
    - $\sigma_5 = \langle d1, d4 \rangle$
    - $P(\sigma_5|s_0,\pi_5) = 0.5 = \left(\frac{1}{2}\right)^1$
    - $\sigma_6 = \langle d1, d1, d4 \rangle$
    - $P(\sigma_6|s_0,\pi_6)$
      $= 0.5 \cdot 0.5 = \left(\frac{1}{2}\right)^2$
    - …
  - $P\left(S_g|s_0,\pi_5\right)$
    $= \frac{1}{2} + \frac{1}{4} + \ \ldots = 1$



Expected Cost

- $cost(s, a)$ = cost of using a in s
  - Example
    - Each ”horizontal” action costs 1
    - Each ”vertical” action costs 100
- Costs of a history
  $$\sigma = \langle s_0, s_1, s_2, \ldots \rangle$$
  - $cost(\sigma \mid s_0, \pi)$
    $$= \sum_{s_i \in \sigma} cost(s_i, \pi(s_i))$$



Expected Cost

- Let π be a safe solution
- At each state $s \in Dom(\pi)$, expected cost of following π to goal:
  - Weighted sum of history costs:

$$V^{\pi}(s) = cost\big(s, \pi(s)\big) + \sum_{\sigma \in H(s,\pi)} P(\sigma \mid s, \pi)\, cost(\sigma \mid s, \pi)$$

  - Recursive formulation

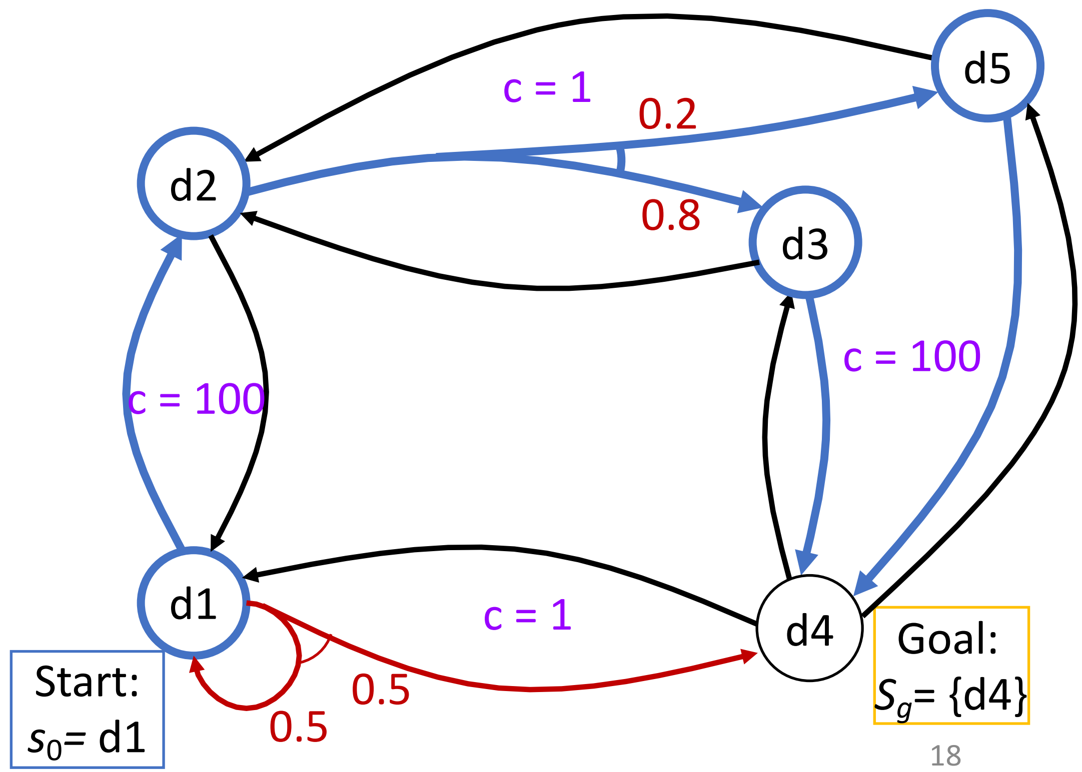
$$V^{\pi}(s) = \begin{cases} 0 & \text{if } s \in S_g \\ cost\big(s, \pi(s)\big) + \sum_{s' \in \gamma(s,\pi(s))} P\big(s' | s, \pi(s)\big) V^{\pi}(s') & \text{otherwise} \end{cases}$$

Compare policy evaluation of the policy iteration algorithm of the previous topic

Example

- $\pi_3 = \{(d1, m12), (d2, m23), (d3, m34), (d5, m54)\}$
- Weighted sum of history cost:
  - $\sigma_1 = \langle d1, d2, d3, d4 \rangle$
  - $\mathrm{P}(\sigma_1 | s_0, \pi_1) = 0.8$
  - $cost(\sigma_1 | s_0, \pi_3)$
    $= 100 + 1 + 100 = 201$
  - $\sigma_4 = \langle d1, d2, d5, d4 \rangle$
  - $\mathrm{P}(\sigma_4 | s_0, \pi_1) = 0.2$
  - $cost(\sigma_4 | s_0, \pi_3)$
    $= 100 + 1 + 100 = 201$
- $V^{\pi_1}(d1)$
  $= 0.8(201) + 0.2(201)$
  $= 201$

- Recursive equation
  - $V^{\pi_1}(d1)$
    $= 100 + V^{\pi_1}(d2)$
    $= 100 + 1 + 0.8V^{\pi_1}(d3) + 0.2V^{\pi_1}(d5)$
    $= 100 + 1 + 0.8(100) + 0.2(100)$
    $= 201$

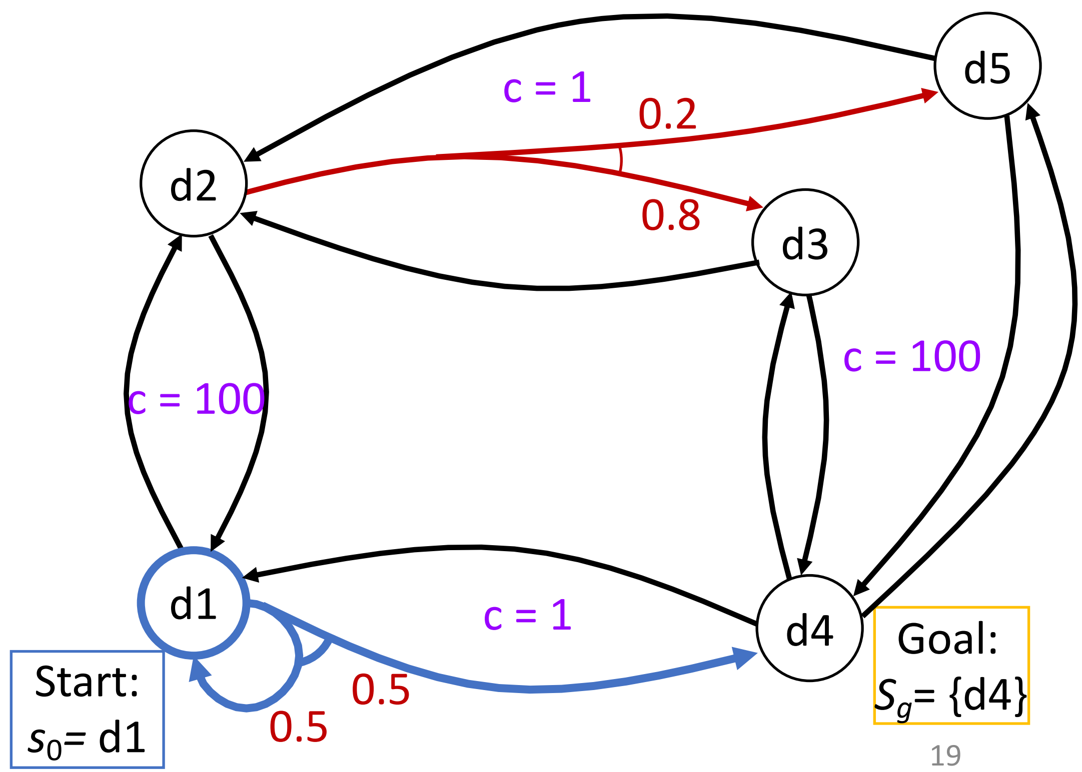


Safe Solutions

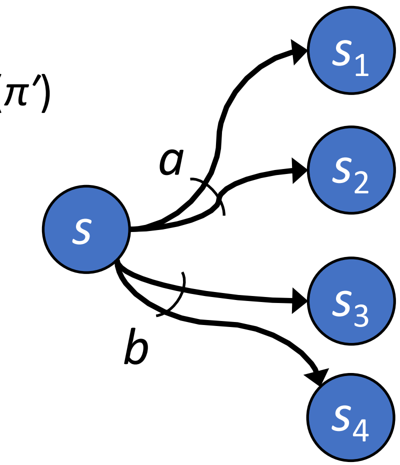
- $\pi_4 = \{(d1, m14)\}$
- Weighted sum of history cost:
  - $\sigma_5 = \langle d1, d4 \rangle$
  - $\mathrm{P}(\sigma_5 | s_0, \pi_5) = \left(\frac{1}{2}\right)^1$
  - $cost(\sigma_5 | s_0, \pi_5) = 1$

  - $\sigma_6 = \langle d1, d1, d4 \rangle$
  - $\mathrm{P}(\sigma_6 | s_0, \pi_6) = \left(\frac{1}{2}\right)^2$
  - $cost(\sigma_6 | s_0, \pi_5) = 2$
  - …
- $V^{\pi_4}(d1)$
  $= \frac{1}{2}(1) + \frac{1}{4}(2) + \ \ldots$
  $= 2$

- Recursive equation
  $V^{\pi_4}(d1)$
  $= 1 + 0.5(0) + 0.5\left(V^{\pi_4}(d1)\right)$
  $\Leftrightarrow 0.5V^{\pi_4}(d1) = 1$
  $\Leftrightarrow \ V^{\pi_4}(d1) = 2$



Planning as Optimisation



- Let π and π' be safe solutions
  - π dominates π' if $V^{\pi}(s) \leq V^{\pi'}(s)$ for every $s \in \mathrm{Dom}(\pi) \cap \mathrm{Dom}(\pi')$
- π is optimal if π dominates *every* safe solution
  - If π and π' are both optimal, then $V^{\pi}(s) = V^{\pi'}(s)$ at every state where they are both defined
- $V^*(s)$ = expected cost of getting to goal using an optimal safe solution
- Recall expected cost of following π to goal starting in s

$$V^{\pi}(s) = \begin{cases} 0 & \text{if } s \in S_g \\ cost(s, \pi(s)) + \sum_{s' \in \gamma(s, \pi(s))} P(s' | s, \pi(s)) V^{\pi}(s') & \text{otherwise} \end{cases}$$

- Optimality principle (Bellman's theorem):

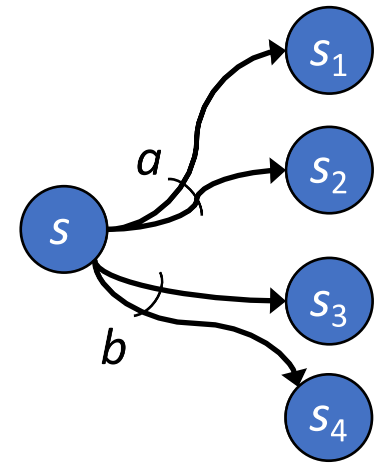
$$V^*(s) = \begin{cases} 0 & \text{if } s \in S_g \\ \min_{a \in Applicable(S)} \left\{ cost(s, \pi(s)) + \sum_{s' \in \gamma(s, \pi(s))} P(s' | s, \pi(s)) V^*(s') \right\} & \text{otherwise} \end{cases}$$

Cost to Go

- Let (Σ, s_0, S_g) be a safe SSP
  - I.e., S_g is reachable from every state
  - Same as safely explorable in non-deterministic models
- Let π be a safe solution that is defined at all non-goal states
  - I.e., $Dom(\pi) = S \setminus S_g$
- Let $a \in Applicable(s)$
- Cost-to-go

$$Q^{\pi}(s,a) = cost(s,a) + \sum_{s' \in \gamma(s,a)} P(s'|s,a)\, V^{\pi}(s')$$

  - Expected cost if we start at s, use a, and use π afterward



- For every $s \in S \setminus S_g$, let

$$\pi'(s) \in \operatorname*{argmin}_{a \in Applicable(s)} Q^{\pi}(s,a)$$

Policy Iteration

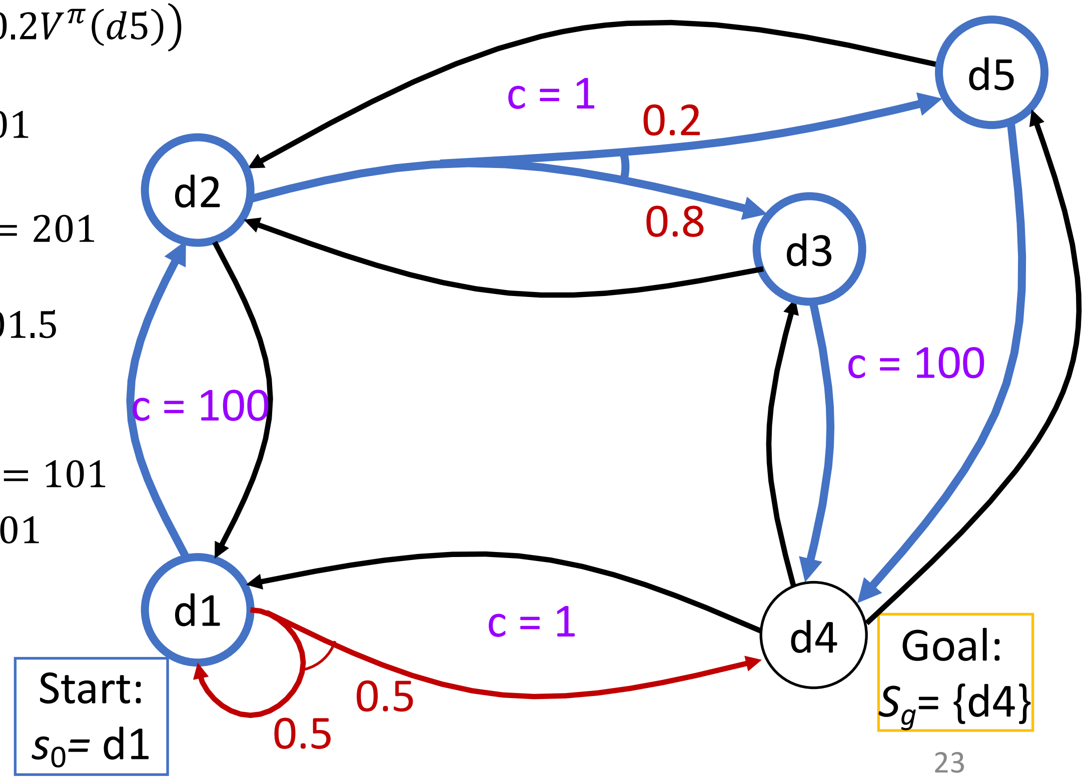
- Converges in a finite number of steps

n equations,
n unknowns,
where $n = |S|$

```
policy-iteration(Σ,s₀,S_g,π₀)
    π ← π₀
    loop
        compute{V^π(s)|s ∈ S}
        for every state s ∈ S \ S_g do
            A ← argmin_{a∈Applicable(s)} Q^π(s,a)
            if π(s) ∈ A then
                π′(s) ← π(s)
            else
                π′(s) ← any action in A
            if π′ = π then
                return π
        π ← π′
```

Example

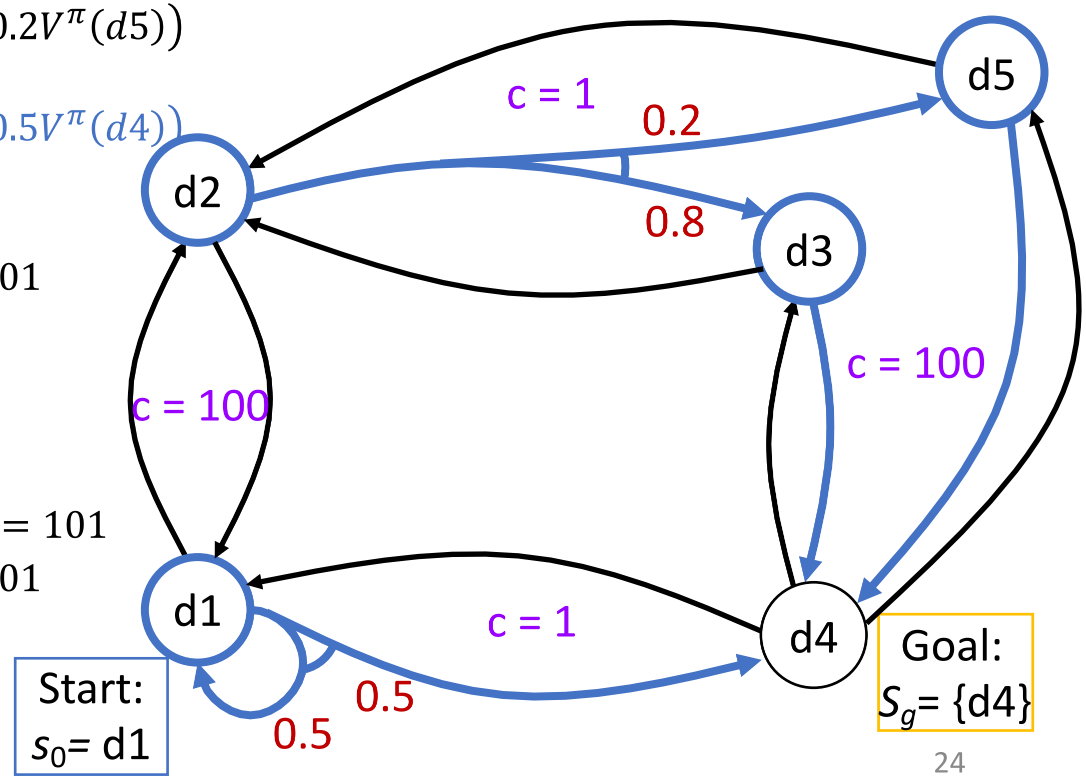
- Start with
  - $\pi = \pi_0 = \{(d1, m12),$
    $(d2, m23), (d3, m34),\ (d5, m54)\}$
- Expected cost
  - $V^\pi(d4) = 0$
  - $V^\pi(d3) = 100 + V^\pi(d4) = 100$
  - $V^\pi(d5) = 100 + V^\pi(d4) = 100$
  - $V^\pi(d2) = 1 + \big(0.8V^\pi(d3) + 0.2V^\pi(d5)\big)$
    $= 101$
  - $V^\pi(d1) = 100 + V^\pi(d2) = 201$
- Cost-to-go
  - $Q(d1, m12) = 100 + 1(101) = 201$
  - $Q(d1, m14)$
    $= 1 + 0.5(201) + 0.5(0) = 101.5$
    - $\text{argmin} = m14$
  - $Q(d2, m23)$
    $= 1 + \big(0.8(100) + 0.2(100)\big) = 101$
  - $Q(d2, m21) = 100 + 201 = 301$
    - $\text{argmin} = m23$
- Cost-to-go continued
  - $Q(d3, m34) = 100 + 0 = 100$
  - $Q(d3, m32) = 1 + 101 = 102$
    - $\text{argmin} = m34$
  - $Q(d5, m54) = 100 + 0 = 100$
  - $Q(d5, m52) = 1 + 101 = 102$
    - $\text{argmin} = m54$



Example

- Continue with
  - $\pi = \{(d1, m14),\ (d2, m23), (d3, m34), (d5, m54)\}$
- Expected cost
  - $V^{\pi}(d4) = 0$
  - $V^{\pi}(d3) = 100 + V^{\pi}(d4) = 100$
  - $V^{\pi}(d5) = 100 + V^{\pi}(d4) = 100$
  - $V^{\pi}(d2) = 1 + \left(0.8V^{\pi}(d3) + 0.2V^{\pi}(d5)\right) = 101$
  - $V^{\pi}(d1) = 1 + \left(0.5V^{\pi}(d1) + 0.5V^{\pi}(d4)\right) = 2$
- Cost-to-go
  - $Q(d1, m12) = 100 + 101 = 201$
  - $Q(d1, m14) = 1 + 0.5(2) + 0.5(0) = 2$
    - $\text{argmin} = m14$
  - $Q(d2, m23) = 1 + \left(0.8(100) + 0.2(100)\right) = 101$
  - $Q(d2, m21) = 100 + 201 = 301$
    - $\text{argmin} = m23$
- Cost-to-go continued
  - $Q(d3, m34) = 100 + 0 = 100$
  - $Q(d3, m32) = 100 + 101 = 201$
    - $\text{argmin} = m34$
  - $Q(d5, m54) = 100 + 0 = 100$
  - $Q(d5, m54) = 100 + 101 = 201$
    - $\text{argmin} = m54$

π unchanged



Value Iteration

- $\eta > 0$:
  - for testing approx. convergence
- V_0 is a heuristic fct. for initial values
  - $V_0(s) = 0 \; \forall s \in S_g$
  - E.g., adapt a heuristic from Ch. 2
- V_i = values computed at i'th iteration
- π_i = plan computed from V_i
- Synchronous version computes V_i and π_i from old V_{i-1} and π_{i-1}
- Asynchronous version updates V and π in place
  - New values available immediately
  - More efficient than synchronous version

```
sync-value-iteration(Σ,s₀,S_g,V₀,η)
    for i = 1,2,… do
        for every state s ∈ S \ S_g do
            for every a ∈ Applicable(s) do
                Q(s,a) ← cost(s,a)+Σ_{s'∈S} P(s'|s,a) V_{i-1}(s')
            V_i(s) ← min_{a∈Applicable(s)} Q(s,a)
            π_i(s) ← argmin_{a∈Applicable(s)} Q(s,a)
            if max_{s∈S}|V_i(s)- V_{i-1}(s)| ≤ η then
                return π_i
```

```
async-value-iteration(Σ,s₀,S_g,V₀,η)
    global π ← ∅
    global V(s) ← V₀(s) ∀ s
    loop
        r ← max_{s∈S\Sg} Bellman-Update(s)
        if r ≤ η then
            return π

Bellman-Update(s)
    v_old ← V(s)
    for every a ∈ Applicable(s) do
        Q(s,a) ← cost(s,a)+Σ_{s'∈S} P(s'|s,a) V(s')
    V(s) ← min_{a∈Applicable(s)} Q(s,a)
    π(s) ← argmin_{a∈Applicable(s)} Q(s,a)
    return |V(s)-v_old|
```

Synchronous $\quad$ Asynchronous

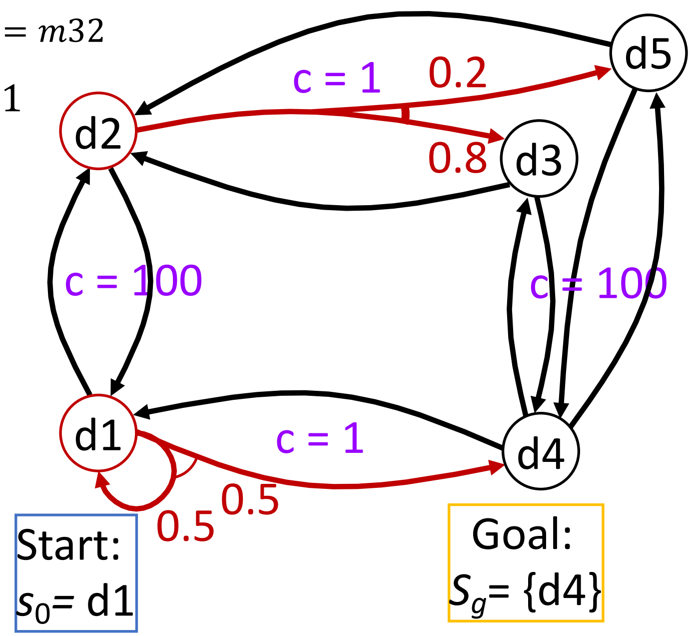
$$\eta = 0.2$$
$$V_0(s) = 0\ \forall s$$

Synchronous

- $Q(d1, m12) = 100 + 0 = 100$
- $Q(d1, m14) = 1 + \big(0.5(0) + 0.5(0)\big) = 1$
  - $V_1(d1) = 1;\ \pi_1(d1) = m14$
- $Q(d2, m21) = 100 + 0 = 100$
- $Q(d2, m23) = \ 1 + \big(0.2(0) + 0.8(0)\big) = 1$
  - $V_1(d2) = 1;\ \pi_1(d2) = m23$
- $Q(d3, m32) = 1 + 0 = 1$
- $Q(d3, m34) = 100 + 0 = 100$
  - $V_1(d3) = 1;\ \pi_1(d3) = m32$
- $Q(d5, m52) = 1 + 0 = 1$
- $Q(d5, m54)$
  $= 100 + 0 = 100$
  - $V_1(d5) = 1;$
    $\pi_1(d5) = m52$
- $r = \max(1 - 0,$
  $1 - 0, 1 - 0, 1 - 0) = 1$

Asynchronous

- $Q(d1, m12) = 100 + 0 = 100$
- $Q(d1, m14) = 1 + \big(0.5(0) + 0.5(0)\big) = 1$
  - $V(d1) = 1;\ \pi(d1) = m14$
- $Q(d2, m21) = 100 + 1 = 101$
- $Q(d2, m23) = 1 + \big(0.2(0) + 0.8(0)\big) = 1$
  - $V(d2) = 1;\ \pi(d2) = m23$
- $Q(d3, m32) = 1 + 1 = 2$
- $Q(d3, m34) = 100 + 0 = 100$
  - $V(d3) = 2;\ \pi(d3) = m32$
- $Q(d5, m52) = 1 + 1 = 2$
- $Q(d5, m54) = 100 + 0 = 100$
  - $V(d5) = 2;\ \pi(d5) = m52$
- $r = \max(1 - 0, 1 - 0,$
  $2 - 0, 2 - 0) = 2$



Synchronous

$\eta = 0.2$

Asynchronous

Synchronous

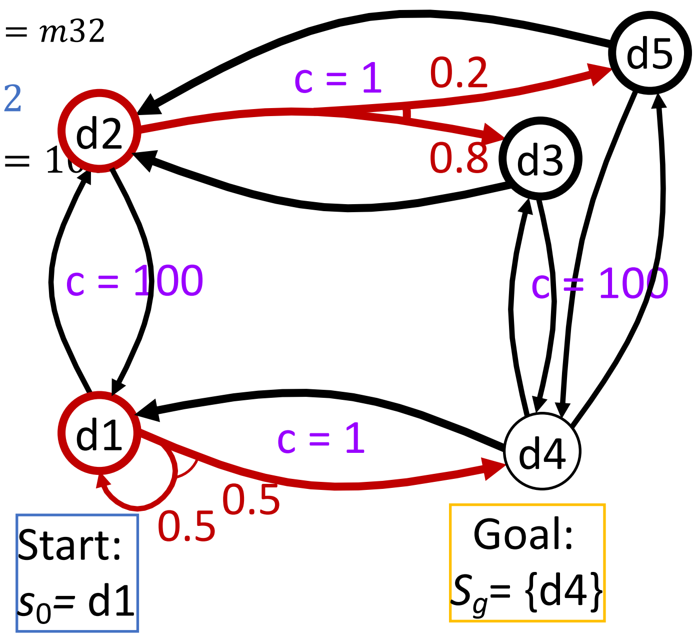
- $Q(d1, m12) = 100 + 1 = 101$
- $Q(d1, m14) = 1 + \big(0.5(1) + 0.5(0)\big) = 1.5$
  - $V_1(d1) = 1.5$; $\pi_1(d1) = m14$
- $Q(d2, m21) = 100 + 1 = 101$
- $Q(d2, m23) = 1 + \big(0.2(1) + 0.8(1)\big) = 2$
  - $V_1(d2) = 2$; $\pi_1(d2) = m23$
- $Q(d3, m32) = 1 + 1 = 2$
- $Q(d3, m34) = 100 + 0 = 100$
  - $V_1(d3) = 2$; $\pi_1(d3) = m32$
- $Q(d5, m52) = 1 + 1 = 2$
- $Q(d5, m54) = 100 + 0 = 10$
  - $V_1(d5) = 1$; $\pi_1(d5) = m52$
- $r = \max(1.5 - 1, 2 - 1, 2 - 1, 2 - 1) = 1$

$V(d1) = 1$
$V(d2) = 1$
$V(d3) = 1$
$V(d5) = 1$

Asynchronous

- $Q(d1, m12) = 100 + 1 = 101$
- $Q(d1, m14) = 1 + \big(0.5(1) + 0.5(0)\big) = 1.5$
  - $V(d1) = 1.5$; $\pi(d1) = m14$
- $Q(d2, m21) = 100 + 1.5 = 101.5$
- $Q(d2, m23) = 1 + \big(0.2(2) + 0.8(2)\big) = 3$
  - $V(d2) = 3$; $\pi(d2) = m23$
- $Q(d3, m32) = 1 + 3 = 4$
- $Q(d3, m34) = 100 + 0 = 100$
  - $V(d3) = 4$; $\pi(d3) = m32$
- $Q(d5, m52) = 1 + 3 = 4$
- $Q(d5, m54) = 100 + 0 = 100$
  - $V(d5) = 4$; $\pi(d5) = m52$
- $r = \max(1.5 - 1, 3 - 1, 4 - 2, 4 - 2) = 2$

$V(d1) = 1$
$V(d2) = 1$
$V(d3) = 2$
$V(d5) = 2$

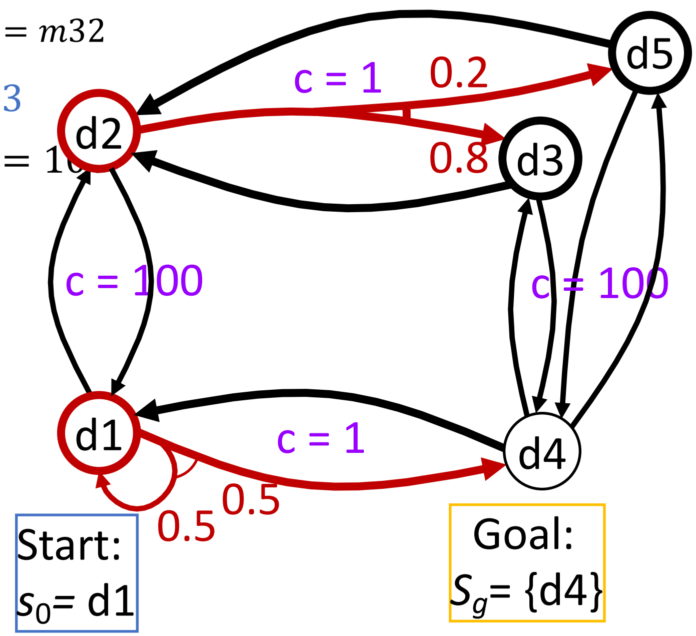


Synchronous $\eta = 0.2$ Asynchronous

Synchronous

- $Q(d1, m12) = 100 + 2 = 102$
- $Q(d1, m14) = 1 + \big(0.5(1.5) + 0.5(0)\big) = 1.75$
  - $V_1(d1) = 1.75$; $\pi_1(d1) = m14$
- $Q(d2, m21) = 100 + 1.5 = 101.5$
- $Q(d2, m23) = 1 + \big(0.2(2) + 0.8(2)\big) = 3$
  - $V_1(d2) = 3$; $\pi_1(d2) = m23$
- $Q(d3, m32) = 1 + 2 = 3$
- $Q(d3, m34) = 100 + 0 = 100$
  - $V_1(d3) = 3$; $\pi_1(d3) = m32$
- $Q(d5, m52) = 1 + 2 = 3$
- $Q(d5, m54) = 100 + 0 = 100$
  - $V_1(d5) = 3$; $\pi_1(d5) = m52$
- $r = \max(1.75 - 1.5, 3 - 2, 3 - 2, 3 - 2) = 1$

$V(d1) = 1.5$
$V(d2) = 2$
$V(d3) = 2$
$V(d5) = 2$



Asynchronous

- $Q(d1, m12) = 100 + 3 = 103$
- $Q(d1, m14) = 1 + \big(0.5(1.5) + 0.5(0)\big) = 1.75$
  - $V(d1) = 1.75$; $\pi(d1) = m14$
- $Q(d2, m21) = 100 + 1.75 = 101.75$
- $Q(d2, m23) = 1 + \big(0.2(4) + 0.8(4)\big) = 5$
  - $V(d2) = 5$; $\pi(d2) = m23$
- $Q(d3, m32) = 1 + 5 = 6$
- $Q(d3, m34) = 100 + 0 = 100$
  - $V(d3) = 6$; $\pi(d3) = m32$
- $Q(d5, m52) = 1 + 5 = 6$
- $Q(d5, m54) = 100 + 0 = 100$
  - $V(d5) = 6$; $\pi(d5) = m52$
- $r = \max(1.75 - 1.5, 5 - 3, 6 - 4, 6 - 4) = 2$

$V(d1) = 1.5$
$V(d2) = 3$
$V(d3) = 4$
$V(d5) = 4$

Synchronous

$\eta = 0.2$

Asynchronous

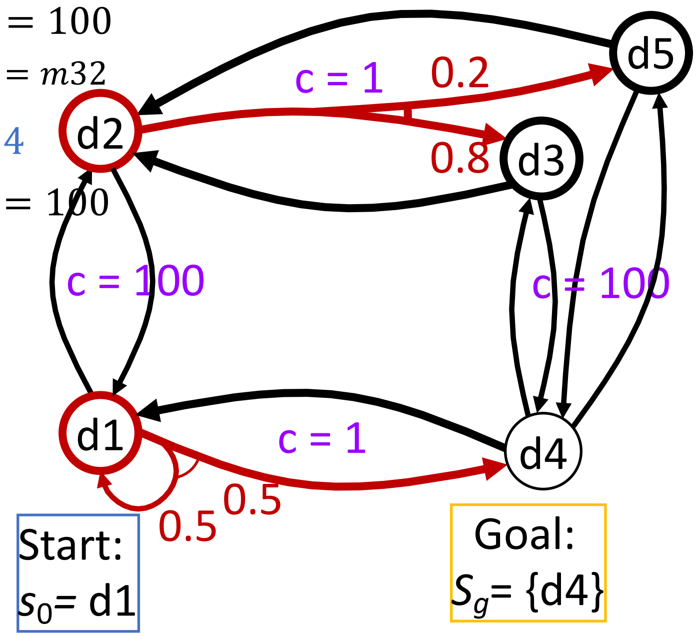
- $Q(d1, m12) = 100 + 3 = 103$
- $Q(d1, m14) = 1 + \big(0.5(1.75) + 0.5(0)\big) = 1.875$
  - $V_1(d1) = 1.875$; $\pi_1(d1) = m14$
- $Q(d2, m21) = 100 + 1.75 = 101.75$
- $Q(d2, m23) = 1 + \big(0.2(3) + 0.8(3)\big) = 4$
  - $V_1(d2) = 4$; $\pi_1(d2) = m23$
- $Q(d3, m32) = 1 + 3 = 4$
- $Q(d3, m34) = 100 + 0 = 100$
  - $V_1(d3) = 4$; $\pi_1(d3) = m32$
- $Q(d5, m52) = 1 + 3 = 4$
- $Q(d5, m54) = 100 + 0 = 100$
  - $V_1(d5) = 4$; $\pi_1(d5) = m52$
- $r = \max(1.875 - 1.75, 4 - 3, 4 - 3, 4 - 3) = 1$

$V(d1) = 1.75$
$V(d2) = 3$
$V(d3) = 3$
$V(d5) = 3$

How long before $r \leq \eta$?
How long, if the "vertical" actions cost 10 instead of 100?

- … $0.5(0)\big) = 1.875$
  - … $m14$
- Q … $+ 1.875 = 101.875$
- $Q(d2$ … $23) = 1 + \big(0.2(6) + 0.8(6)\big) = 7$
  - $V(d2) = 7$; $\pi(d2) = m23$
- $Q(d3, m32) = 1 + 7 = 8$
- $Q(d3, m34) = 100 + 0 = 100$
  - $V(d3) = 8$; $\pi(d3) = m32$
- $Q(d5, m52) = 1 + 7 = 8$
- $Q(d5, m54) = 100 + 0 = 100$
  - $V(d5) = 8$; $\pi(d5) = m52$
- $r = \max(1.875 - 1.75, 7 - 5, 8 - 6, 8 - 6) = 2$

$V(d1) = 1.75$
$V(d2) = 5$
$V(d3) = 6$
$V(d5) = 6$



Discussion

- Policy iteration
  - Computes new π in each iteration; computes V^{π} from π
  - More work per iteration than value iteration
    - Needs to solve a set of simultaneous equations
  - Usually converges in a smaller number of iterations
- Value iteration
  - Computes new V in each iteration; chooses π based on V
  - New V is a revised set of heuristic estimates
    - Not V^{π} for π or any other policy
  - Less work per iteration: does not need to solve a set of equations
  - Usually takes more iterations to converge
- At each iteration, both algorithms need to examine the entire state space
  - Number of iterations polynomial in $|S|$, but $|S|$ may be quite large
- Next: use search techniques to avoid searching the entire space

Summary

- SSPs
- Solutions, closed solutions, histories
- Unsafe solutions, acyclic safe solutions, cyclic safe solutions
- Expected cost, planning as optimization
- Policy iteration
- Value iteration (synchronous, asynchronous)
  - Bellman-update

Outline

6.2 Stochastic shortest path problems

- Safe/unsafe policies
- Optimality
- Policy iteration, value iteration

6.3 Heuristic search algorithms

- Best-first search
- Determinisation

6.4 Online probabilistic planning

- Lookahead
- Reinforcement learning

AO*

- Best-first search for acyclic domains

```
AO*(Σ, s_0, S_g, V_0)
  global π ← ∅; V(s_0) ← V_0(s_0); Envelope ← {s_0}
  while leaves(s_0,π) \ S_g ≠ ∅ do
    select s ∈ leaves(s_0,π) \ S_g
    for all a ∈ Applicable(s) do
      for all s' ∈ γ(s,a) \ Envelope do
        V(s') ← V_0(s')
        Add s' to Envelope
    AO-Update(s)
  return π
```

Requires acyclic Σ

not in book

```
AO-Update(s)
  Z ← {s}   // nodes that need updating
  while Z ≠ ∅ do
    select s ∈ Z s.t. γ̂(s,π(s)) ∩ Z = {s}
    remove s from Z
    Bellman-Update(s)
    Z ← Z ∪ {s' ∈ Envelope | s ∈ γ(s',π)}
```

no π-descendants in Z but s itself
- ensures bottom-up updates

the states “just above” s

```
Bellman-Update(s)
  for every a ∈ Applicable(s) do
    Q(s,a) ← cost(s,a)+Σ_{s'∈S} PR(s'|s,a) V(s')
  V(s) ← min_{a∈Applicable(s)} Q(s,a)
  π(s) ← argmin_{a∈Applicable(s)} Q(s,a)
```

LAO*

- Best-first search for both cyclic and acyclic domains

Σ may be cyclic or acyclic

not in book

all π-ancestors of s in $Envelope$

Asynchronous value iteration, restricted to Z

Different compared to AO*

```
LAO*(Σ, s₀, S_g, V₀)
    global π ← ∅; V(s₀) ← V₀(s₀); Envelope ← {s₀}
    loop
        if leaves(s₀,π) ⊆ S_g ≠ ∅ then
            return π
        select s ∈ leaves(s₀,π) \ S_g
        for all a ∈ Applicable(s) do
            for all s' ∈ γ(s,a) \ Envelope do
                V(s') ← V₀(s')
                Add s' to Envelope
        LAO-Update(s)
    return π
```

```
LAO-Update(s)
    Z ← {s}∪{s' ∈ Envelope | s ∈ γ(s',π)}
    loop
        r ← max_{s∈Z} Bellman-Update(s)
        if leaves(s₀,π) changed or r ≤ η then
            break
```

```
Bellman-Update(s)
    v_old ← V(s)
    for every a ∈ Applicable(s) do
        Q(s,a) ← cost(s,a)+Σ_{s'∈S} PR(s'|s,a) V(s')
    V(s) ← min_{a∈Applicable(s)} Q(s,a)
    π(s) ← argmin_{a∈Applicable(s)} Q(s,a)
    return |V(s)-v_old|
```

LAO* Example

1st iteration of main loop:

- Expand d1: add d2 and d4 to Envelope
- Call LAO-Update(d1)
  - π is empty, so $Z = \{d1\}$

  Iteration 1:
  - $Q(d1, m12) = 100 + 0 = 100$
  - $Q(d1, m14) = 1 + \big(0.5(0) + 0.5(0)\big) = 1$
    - $V(d1) = 1;\ \pi(d1) = m14; r = 1 - 0 = 1$

  Iteration 2:
  - $Q(d1, m12) = 100 + 0 = 100$
  - $Q(d1, m14) = 1 + \big(0.5(1) + 0.5(0)\big) = 1.5$
    - $V(d1) = 1.5;\ \pi(d1) = m14;$
      $r\ =\ 1.5 - 1 = 0.5$

  Iteration 3:
  - $Q(d1, m12) = 100 + 0 = 100$
  - $Q(d1, m14) = 1 + \big(0.5(1.5) + 0.5(0)\big) = 1.75$
    - $V(d1) = 1.75; \pi(d1) = m14;$
      $r = 1.75 - 1.5 = 0.25$

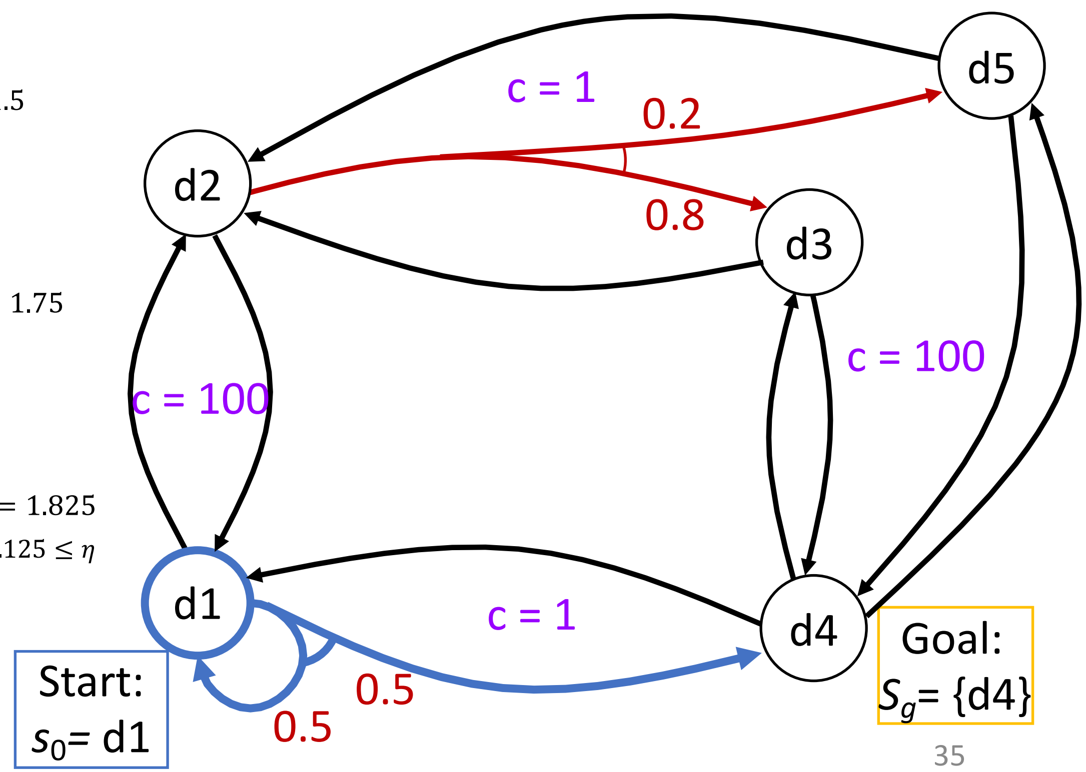
  Iteration 4:
  - $Q(d1, m12) = 100 + 0 = 100$
  - $Q(d1, m14) = 1 + \big(0.5(1.75) + 0.5(0)\big) = 1.825$
    - $V(d1) = 1.825; \pi(d1) = m14; r = 0.125 \leq \eta$

  LAO-Update returns

2nd iteration of main loop:

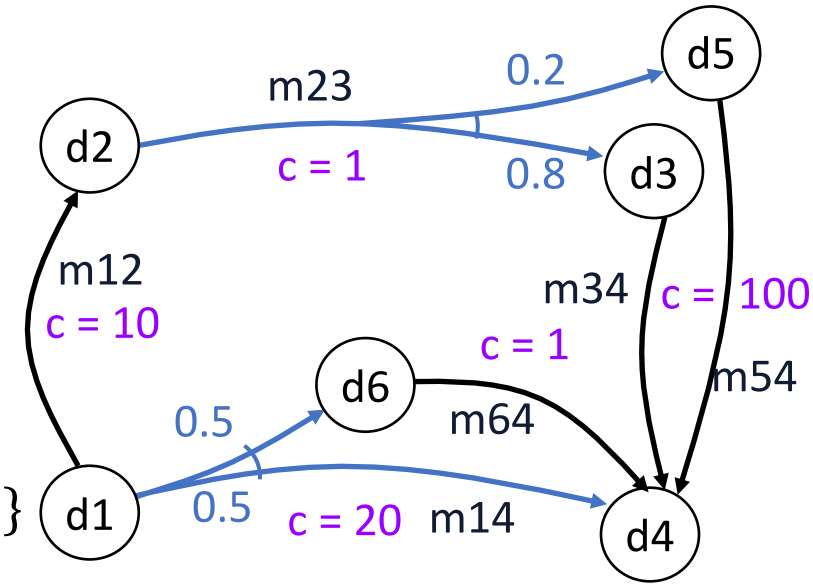
- $leaves(\pi) = \{d4\} \subseteq S_g$
- return π

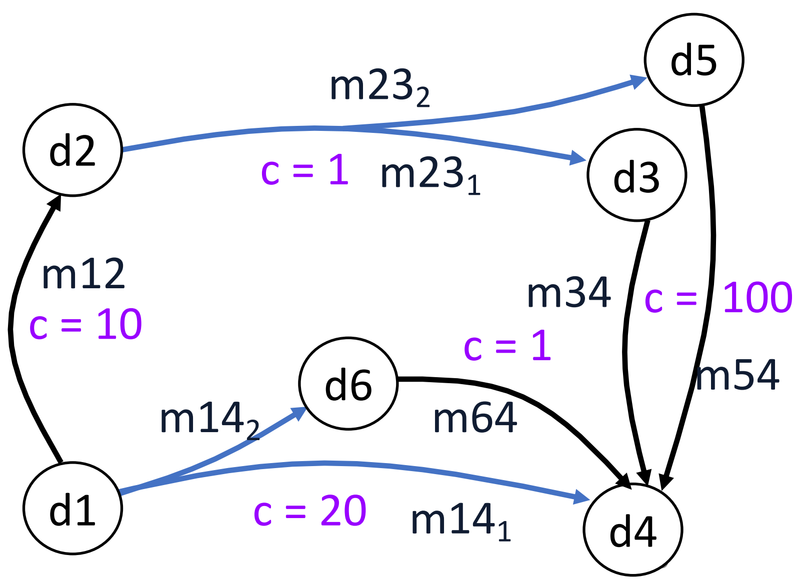
$$\eta = 0.2$$
$$V_0(s) = 0\ \forall s$$



Heuristics through Determinization

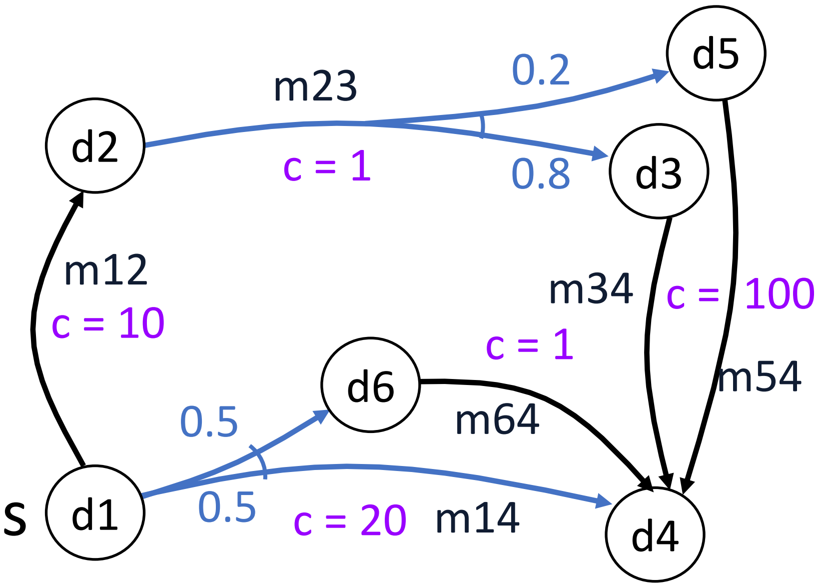
- What to use for V_0?
  - One possibility: classical planner
  - Need to convert nondeterministic actions into something the classical planner can use
- Determinise the actions
  - Suppose $\gamma(s,a) = \{s_1, \dots, s_n\}$
  - $Det(s,a) = \{n \text{ actions } a_1, a_2, \dots, a_n\}$
    - $\gamma_d(s, a_i) = s_i$
    - $cost_d(s, a_i) = cost(s,a)$
- Classical domain $\Sigma_d = (S, A_d, \gamma_d, cost_d)$
  - S = same as in Σ
  - $A_d = \bigcup_{a \in A, s \in S} Det(s,a)$
  - γ_d and $cost_d$ as above

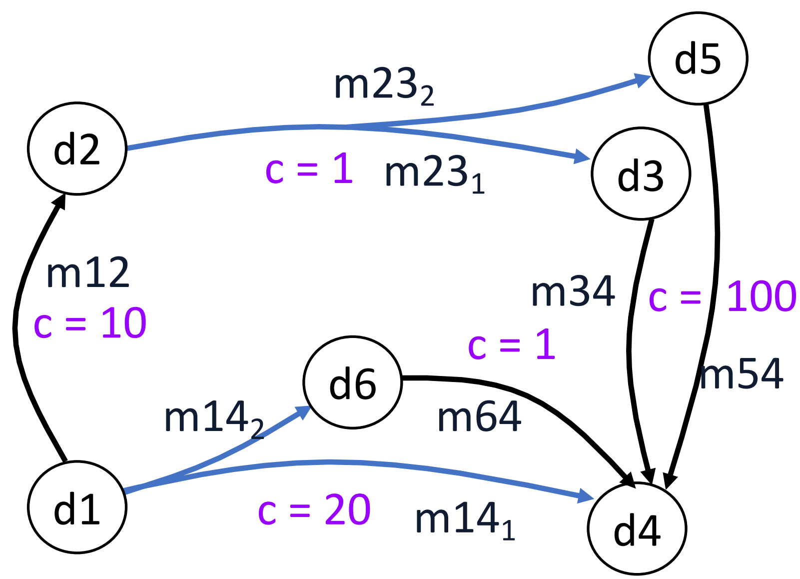




Heuristics through Determinization

- Call classical planner on (Σ_d, s, S_g)
  - Get plan $p = \langle a_1, a_2, \ldots, a_n \rangle$
  - Return
    $$V_0(s) = cost(p) = \sum_{i=1}^{n} cost(a_i)$$
- If the classical planner always returns optimal plans, then V_0 is admissible
- Outline of proof:
  - Let π be a safe solution in Σ
  - Every acyclic execution of π corresponds to a plan p' in Σ_d
    - p' must have cost ≥ $V_0(s)$
    - Otherwise the classical planner would have chosen p' instead of p





Summary

- AO*
  - Acyclic
- LAO*
  - (A)cyclic
- Heuristics through determinisation

Outline

6.2 Stochastic shortest path problems

- Safe/unsafe policies
- Optimality
- Policy iteration, value iteration

6.3 Heuristic search algorithms

- Best-first search
- Determinisation

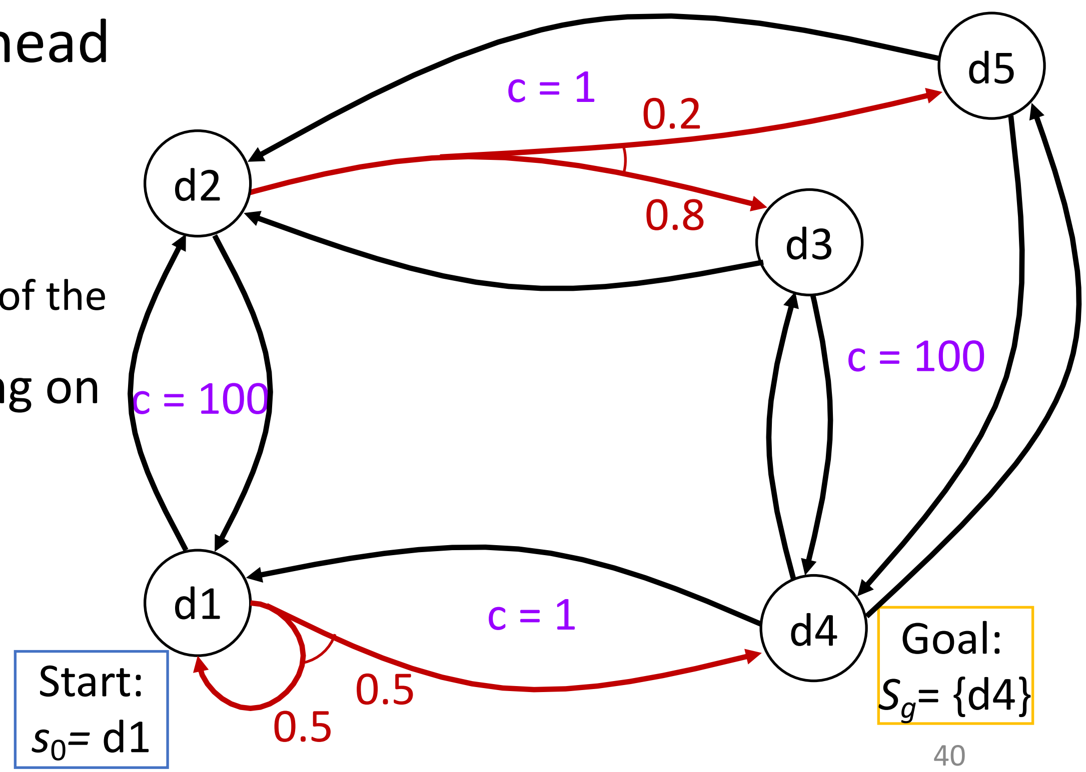
6.4 Online probabilistic planning

- Lookahead
- Reinforcement learning

Planning and Acting

- Same as in Ch. 2, except s instead of ξ
  - Could use $s \leftarrow$ abstraction of ξ as in Ch. 2
- Could also use Run-Lazy-Lookahead or Run-Concurrent-Lookahead
- What to use for Lookahead?
  - AO*, LAO*, …
    - Modify to search part of the space
  - Classical planner running on determinized domain
  - Stochastic sampling algorithms

Run-Lookahead(Σ, s_0, S_g)
    $s \leftarrow s_0$
    while $s \notin S_g$ and $\mathit{Applicable}(s) \neq \emptyset$ **do**
        $a \leftarrow$ Lookahead(s, θ)
        perform action a
        $s \leftarrow$ observe resulting state

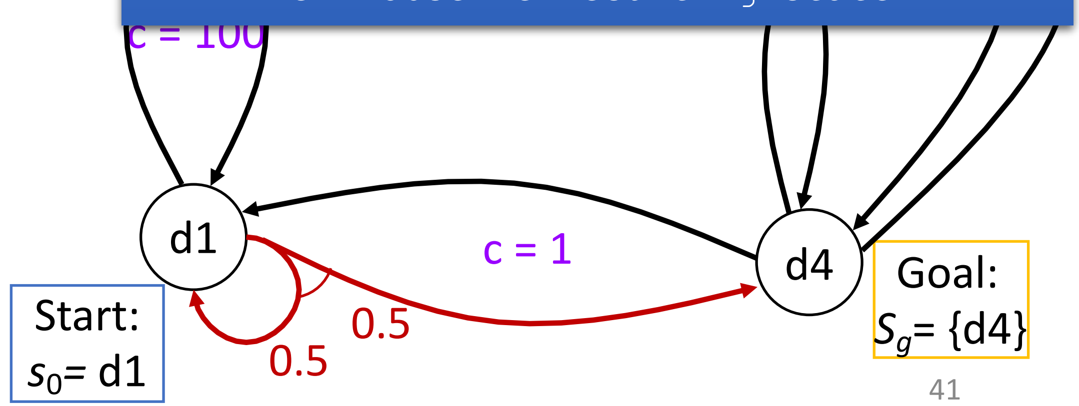


Planning and Acting

- If Lookahead = classical planner on determinized domain
  - ⇒ FS-Replan (Ch. 5)
- Problem: Forward-search may choose a plan that depends on low-probability outcome
- RFF algorithm (see book) attempts to alleviate this

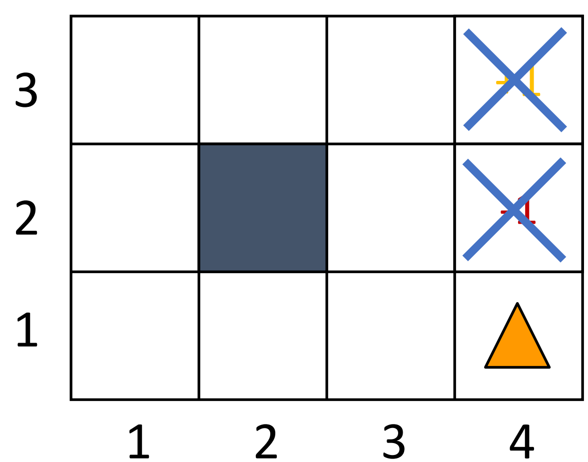
Run-Lookahead(Σ, s_0, S_g)
    $s \leftarrow s_0$
    while $s \notin S_g$ and $Applicable(s) \neq \emptyset$ **do**
        $a \leftarrow$ Lookahead(s, θ)
        perform action a
        $s \leftarrow$ observe resulting state

FS-Replan(Σ, s, S_g)
    $\pi_d \leftarrow \emptyset$
    while $s \notin S_g$ and Applicable(s) $\neq \emptyset$ **do**
        if π_d undefined for s **then**
            $\pi_d \leftarrow$ Forward-Search(Σ_d, s, S_g)
            if π_d = failure **then**
                return failure
        perform action $\pi_d(s)$
        $s \leftarrow$ observe resulting state



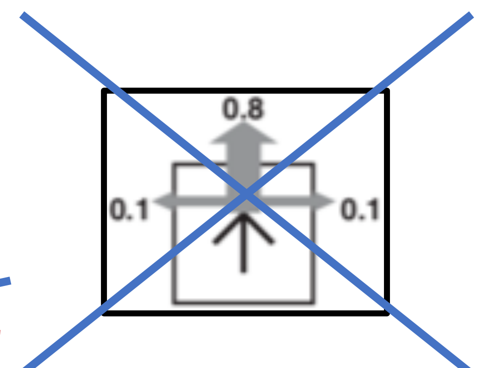
Acting as Reinforcement Learning (RL)

- Agent placed in an environment and must learn to act optimally in it
- Assume that the world behaves like an MDP, except
  - Agent can act but does not know the transition model
  - Agent observes its current state and its reward but does not know the reward function
- Goal: learn an optimal policy



U, D, L, R

~~each move costs 0.04~~



Factors That Make RL Hard

- Actions have non-deterministic <u>effects</u>
  - which are initially <u>unknown</u> and must be learned
- Rewards / punishments can be infrequent
  - Often at the end of long sequences of actions
  - How do we determine what action(s) were really responsible for reward or punishment?
    - Credit assignment problem
  - World is large and complex

Passive vs. Active Learning

- Passive learning
  - Agent acts based on a fixed policy π and tries to learn how good the policy is by observing the world go by
  - Analogous to policy iteration
- Active learning
  - Agent attempts to find an optimal (or at least good) policy by exploring different actions in the world
  - Analogous to solving the underlying MDP

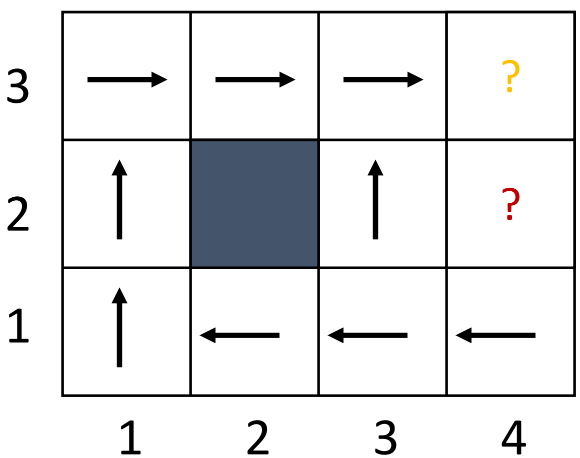
Model-based vs. Model-free RL

- Model-based approach to RL
  - Learn the MDP model ($P(s'|s,a)$ and R), or an approximation of it
  - Use it to find the optimal policy
- Model-free approach to RL
  - Derive the optimal policy without explicitly learning the model

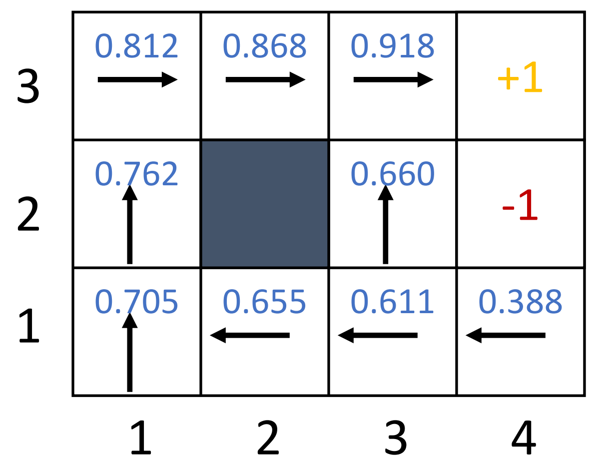
Passive RL

- Suppose we are given a policy
- Want to determine how good it is

- Given π:

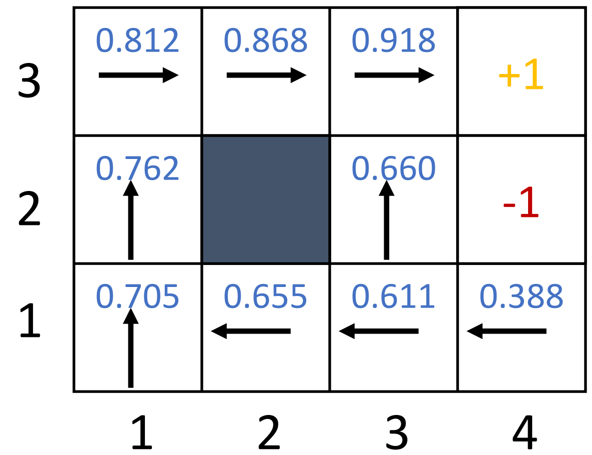


Need to learn $U^{\pi}(s)$:



Passive RL

- Given policy π:
  - Estimate $U^{\pi}(s)$
- Not given
  - Transition model $P(s'|s,a)$
  - Reward function $R(s)$



- Simply follow the policy for many epochs
- Epochs: training sequences

$$(1,1) \rightarrow (1,2) \rightarrow (1,3) \rightarrow (1,2) \rightarrow (1,3) \rightarrow (2,3) \rightarrow (3,3) \rightarrow (3,4) + 1$$
$$(1,1) \rightarrow (1,2) \rightarrow (1,3) \rightarrow (2,3) \rightarrow (3,3) \rightarrow (3,2) \rightarrow (3,3) \rightarrow (3,4) + 1$$
$$(1,1) \rightarrow (2,1) \rightarrow (3,1) \rightarrow (3,2) \rightarrow (4,2) - 1$$

Direct Utility Estimation (DUE)

- Model-free approach
  - Estimate $U^{\pi}(s)$ as average total reward of epochs containing s
    - Calculating from s to end of epoch
- Reward-to-go of a state s
  - The sum of the (discounted) rewards from that state until a terminal state is reached
- Key: use observed reward-to-go of the state as the direct evidence of the actual expected utility of that state

DUE: Example

- Suppose we observe the following trial:
  - $(1,1)_{-0.04} \rightarrow (1,2)_{-0.04} \rightarrow (1,3)_{-0.04} \rightarrow (1,2)_{-0.04} \rightarrow (1,3)_{-0.04} \rightarrow (2,3)_{-0.04} \rightarrow (3,3)_{-0.04} \rightarrow (3,4)_{+1}$
- The total reward starting at $(1,1)$ is 0.72
  - Call this a sample of the observed-reward-to-go for $(1,1)$
- For $(1,2)$, there are two samples for the observed-reward-to-go (assuming $\gamma = 1$)
  1. $(1,2)_{-0.04} \rightarrow (1,3)_{-0.04} \rightarrow (1,2)_{-0.04} \rightarrow (1,3)_{-0.04} \rightarrow (2,3)_{-0.04} \rightarrow (3,3)_{-0.04} \rightarrow (3,4)_{+1}$ [Total: 0.76]
  2. $(1,2)_{-0.04} \rightarrow (1,3)_{-0.04} \rightarrow (2,3)_{-0.04} \rightarrow (3,3)_{-0.04} \rightarrow (3,4)_{+1}$ [Total: 0.84]

DUE: Convergence

- Keep a running average of the observed reward-to-go for each state
  - E.g., for state $(1,2)$, it stores $\frac{(0.76+0.84)}{2} = 0.8$
- As the number of trials goes to infinity, the sample average converges to the true utility

DUE: Problem

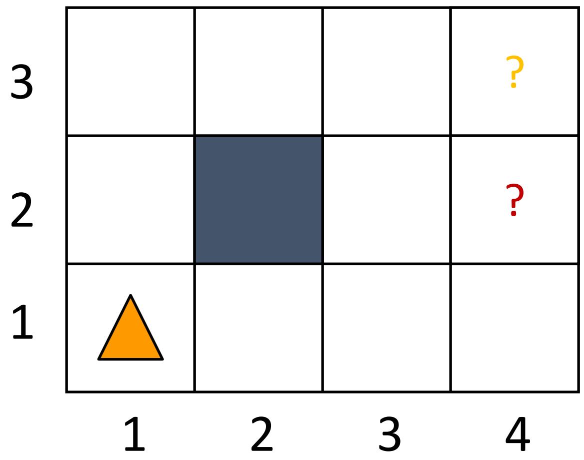
- Big problem: it converges very slowly!
- Why?
  - Does not exploit the fact that utilities of states are not independent
  - Utilities follow the Bellman equation

$$U^{\pi}(s_i) = R(s_i) + \gamma \sum_{s_j} P\big(s_j | \pi(s_i), s_i\big) U^{\pi}\big(s_j\big)$$

Dependence on neighbouring states

DUE: Problem

- Using the dependence to your advantage
  - Suppose you know that state $(3,3)$ has a high utility
  - Suppose you are now at $(3,2)$
  - Bellman equation would be able to tell you that $(3,2)$ is likely to have a high utility because $(3,3)$ is a neighbour
- DUE cannot tell you that until the end of the trial



Adaptive Dynamic Programming (ADP)

- Model-based approach
- Basically learns the transition model $P(s'|s,a)$ and the reward function $R(s)$
  - Takes advantage of constraints in the Bellman equation
- Given policy π:
  - Estimate $U^{\pi}(s)$
  - Based on $P(s'|s,a)$ and $R(s)$, we can perform policy evaluation (part of policy iteration)

ADP: Policy Evaluation

- Policy Iteration:
  - Pick a policy π at random
  - Repeat:
    - Policy evaluation: Compute the utility of each state for π
      - $U_{t+1}(s_i) = R(s_i) + \gamma \sum_{s_j} P\big(s_j|\pi(s_i), s_i\big)U_t\big(s_j\big)$
        - No longer involves a max operation as action is determined by π
    - Policy improvement: Compute the policy π' given U_{t+1}
      - $\pi'(s_i) = \arg\max_a \sum_{s_j} P\big(s_j|\pi(s_i). s_i\big)U_t\big(s_j\big)$
    - If $\pi' = \pi$, then return π

Can be solved in time $O(n^3)$, where n is the number of states

Or solve the set of linear equations:

$$U(s_i) = R(s_i) + \gamma \sum_{s_j} P\big(s_j|\pi(s_i). s_i\big)U\big(s_j\big)$$

(often a sparse system)

ADP: Learn the Model

- Make use of policy evaluation to learn the utilities of states
- To use policy equation

  $$U_{t+1}(s_i) = R(s_i) + \gamma \sum_{s_j} P\big(s_j | \pi(s_i), s_i\big) U_t\big(s_j\big)$$

  agent needs to learn $P(s'|s, a)$ and $R(s)$
- How?

ADP: Learn the Model

- Learning $R(s)$
  - Easy because it is deterministic
  - Whenever you see a new state, store the observed reward value as $R(s)$
- Learning $P(s'|s,a)$
  - Keep track of how often you get to state s' given that you are in state s and do action a
  - E.g., if you are in $s = (1,3)$ and you execute R three times and you end up in $s' = (2,3)$ twice, then $P(s'|\text{R}, s) = \frac{2}{3}$

ADP: Algorithm

Update reward function

Update transition model

```
function passive-ADP-agent(percept)
   returns an action
   input: percept, indicating current state s′, reward r′
   static:
      π, fixed policy
      mdp, MDP with P[s′|s,a], R(s), γ
      U, table of utilities, initially empty
      N_sa, table of freq. for s-a pairs, initially 0
      N_sas′, table of freq. for s-a-s′ triples, initially 0
      s,a, previous state and action, initially null
   if s′ is new then
      U[s′] ← r′
      R[s′] ← r′
   if s is not null then
      increment N_sa[s,a] and N_sas′[s,a,s′]
      for each t s.t. N_sas′[s,a,t] ≠ 0 do
         P[t|s,a] ← N_sas′[s,a,t] / N_sa[s,a]
   U ← Policy-evaluation(π,U,mdp)
   if Terminal?(s′) then
      s,a ← null
   else
      s,a ← s′,π[s′]
   return a
```

ADP: Problem

- Need to solve a system of simultaneous equations – costs $O(n^3)$
  - Very hard to do if you have 10^{50} states like in Backgammon
  - Could make things a little easier with modified policy iteration
- Can we avoid the computational expense of full policy evaluation?

Temporal Difference Learning (TD)

- Instead of calculating the exact utility for a state, can we approximate it and possibly make it less computationally expensive?
- Yes, we can! Using TD:

$$U^{\pi}(s_i) = R(s_i) + \gamma \sum_{s_j} P\big(s_j | \pi(s_i), s_i\big) U^{\pi}\big(s_j\big)$$

  - Instead of doing the sum over all successors, only adjust the utility of the state based on the successor observed in the trial
  - Does not estimate the transition model – model-free

TD: Example

- Suppose you see that $U^{\pi}(1,3) = 0.84$ and $U^{\pi}(2,3) = 0.92$
- If the transition $(1,3) \rightarrow (2,3)$ happens all the time, you would expect to see:

$$U^{\pi}(1,3) = R(1,3) + U^{\pi}(2,3)$$
$$\Rightarrow U^{\pi}(1,3) = -0.04 + U^{\pi}(2,3)$$
$$\Rightarrow U^{\pi}(1,3) = -0.04 + 0.92 = 0.88$$

- Since you observe $U^{\pi}(1,3) = 0.84$ in the first trial and it is a little lower than 0.88, so you might want to "bump" it towards 0.88

Aside: Online Mean Estimation

- Suppose that we want to incrementally compute the mean of a sequence of numbers
  - E.g., to estimate the mean of a random variable from a sequence of samples

$$\hat{X}_{n+1} = \frac{1}{n+1}\sum_{i=1}^{n+1} x_i = \left(\frac{1}{n+1}\sum_{i=1}^{n} x_i\right) + \frac{1}{n+1}x_{n+1} = \left(\frac{n}{n(n+1)}\sum_{i=1}^{n} x_i\right) + \frac{1}{n+1}x_{n+1}$$

average of $n+1$ samples

$$= \left(\frac{n+1-1}{n(n+1)}\sum_{i=1}^{n} x_i\right) + \frac{1}{n+1}x_{n+1} = \left(\frac{n+1}{n(n+1)}\sum_{i=1}^{n} x_i\right) - \left(\frac{1}{n(n+1)}\sum_{i=1}^{n} x_i\right) + \frac{1}{n+1}x_{n+1}$$

$$= \left(\frac{1}{n}\sum_{i=1}^{n} x_i\right) - \left(\frac{1}{(n+1)}\cdot\frac{1}{n}\sum_{i=1}^{n} x_i\right) + \frac{1}{n+1}x_{n+1} = \left(\frac{1}{n}\sum_{i=1}^{n} x_i\right) + \frac{1}{n+1}\left(x_{n+1} - \frac{1}{n}\sum_{i=1}^{n} x_i\right)$$

$$= \hat{X}_n + \frac{1}{n+1}\left(x_{n+1} - \hat{X}_n\right)$$

learning rate

sample $n+1$

- Given a new sample x_{n+1}, the new mean is the old estimate (for n samples) plus the weighted difference between the new sample and old estimate

TD Update

- TD update for transition from s to s'

$$U^{\pi}(s) = U^{\pi}(s) + \alpha\underbrace{\left(R(s) + \gamma U^{\pi}(s') - U^{\pi}(s)\right)}_{\text{new (noisy) sample of utility based on next state}}$$

learning rate

  - Similar to one step of value iteration
  - Equation called backup
- So, the update is maintaining a “mean” of the (noisy) utility samples
- If the learning rate decreases with the number of samples (e.g., $1/n$), then the utility estimates will eventually converge to true values

$$U^{\pi}(s_i) = R(s_i) + \gamma \sum_{s_j} P\left(s_j | \pi(s_i), s_i\right) U^{\pi}\left(s_j\right)$$

TD: Convergence

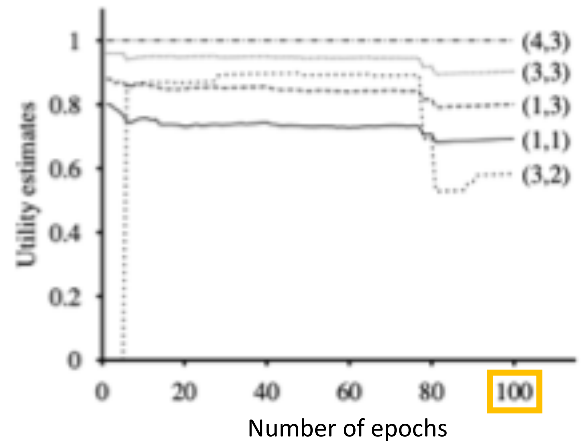
- Since we are using the observed successor s' instead of all the successors, what happens if the transition $s \longrightarrow s'$ is very rare and there is a big jump in utilities from s to s'?
  - How can $U^{\pi}(s)$ converge to the true equilibrium value?
- Answer: The average value of $U^{\pi}(s)$ will converge to the correct value
- This means we need to observe enough trials that have transitions from s to its successors
- Essentially, the effects of the TD backups will be averaged over a large number of transitions
- Rare transitions will be rare in the set of transitions observed

Comparison between ADP and TD

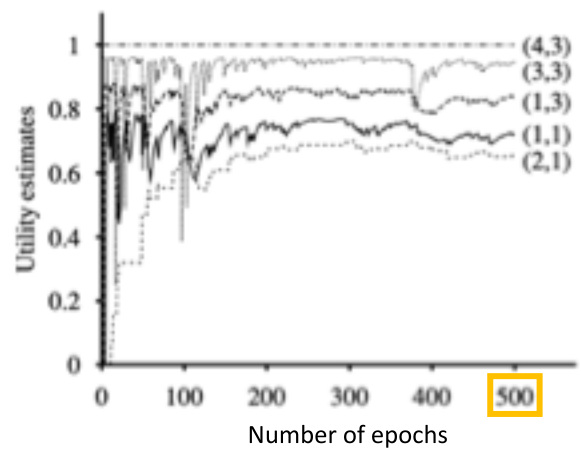
- Advantages of ADP
  - Converges to true utilities faster
  - Utility estimates do not vary as much from the true utilities
- Advantages of TD
  - Simpler, less computation per observation
  - Crude but efficient first approximation to ADP
  - Do not need to build a transition model to perform its updates
    - Important because we can interleave computation with exploration rather than having to wait for the whole model to be built first

ADP and TD

- Utility estimates for 4x3 grid
  - ADP, given optimal policy
    - Notice the large changes occurring around the 78th trial—this is the first time that the agent falls into the −1 terminal state at (4,2)



  - TD
    - More epochs required
    - Faster runtime per epoch



  - Source: AIMA, Russell/Norvig

Overall comparisons

- DUE (model-free)
  - Simple to implement
  - Each update is fast
  - Does not exploit Bellman constraints and converges slowly
- ADP (model-based)
  - Harder to implement
  - Each update is a full policy evaluation (expensive)
  - Fully exploits Bellman constraints
  - Fast convergence (in terms of epochs)
- TD (model-free)
  - Update speed and implementation similar to direct estimation
  - Partially exploits Bellman constraints – adjusts state to "agree" with observed successor
    - Not all possible successors
  - Convergence in between DUE and ADP

Passive Learning: Disadvantage

- Learning $U^{\pi}(s)$ does not lead to an optimal policy, why?
- Models are incomplete/inaccurate
- Agent has only tried limited actions, we cannot gain a good overall understanding of $P(s'|s,a)$
- This is why we need active learning

Goal of Active Learning

- Let us first assume that we still have access to some sequence of trials performed by the agent
  - Agent is not following any specific policy
  - We can assume for now that the sequences should include a thorough exploration of the space
  - We will talk about how to get such sequences later

- The goal is to learn an optimal policy from such sequences
  - Active RL agents
    - Active ADP agent
    - Q-learner (based on TD algorithm)

Active ADP Agent

- Model-based approach
- Using the data from its trials, agent learns a transition model $\hat{T}$ and a reward function $\hat{R}$
- With $\hat{T}(s, a, s')$ and $\hat{R}(s)$, it has an estimate of the underlying MDP
- It can compute the optimal policy by solving the Bellman equations using value or policy iteration

$$U(s) = \hat{R}(s) + \gamma \max_a \sum_{s'} \hat{T}(s, a, s') U(s')$$

- If $\hat{T}$ and $\hat{R}$ are accurate estimations of the underlying MDP model, we can find the optimal policy this way

Issues with ADP Approach

- Need to maintain MDP model
- T can be very large, $O(|S|^2 \cdot |A|)$
- Also, finding the optimal action requires solving the Bellman equation – time consuming
- Can we avoid this large computational complexity both in terms of time and space?

Q-learning

- So far, focus on utilities for states
  - $U(s)$ = utility of state s = expected maximum future rewards
- Alternative: store Q-values
  - $Q(a, s)$ = utility of taking action a at state s
    = expected maximum future reward if action a at state s
- Relationship between $U(s)$ and $Q(a, s)$?

$$\boxed{U(s) = \max_a Q(a, s)}$$

Q-learning can be model-free

- Note that after computing $U(s)$, to obtain the optimal policy, we need to compute
$$\pi(s) = \underset{a}{\operatorname{argmax}} \sum_{s'} T(s, a, s') U(s')$$
  - Requires T, the model of world
  - Even if we use TD learning (model-free), we still need the model to get the optimal policy
- However, if you successfully estimate $Q(a, s)$ for all a and s, we can compute the optimal policy without using the model
$$\pi(s) = \underset{a}{\operatorname{argmax}}\, Q(a, s)$$

Q-learning

- At equilibrium when Q-values are correct, we can write the constraint equation:

$$\underbrace{Q(a,s)}_{\text{Expected value for action-state pair } (a,s)} = \underbrace{R(s)}_{\text{Reward at state } s} + \underbrace{\gamma \sum_{s'} T(s,a,s')\, U(s')}_{\substack{\text{Expected value averaged over all} \\ \text{possible states } s' \text{ that can be reached} \\ \text{from } s \text{ after executing action } a}}$$

Q-learning

- At equilibrium when Q-values are correct, we can write the constraint equation:

$$\underbrace{Q(a,s)}_{\text{Expected value for action-state pair } (a,s)} = \underbrace{R(s)}_{\text{Reward at state } s} + \gamma \underbrace{\sum_{s'} T(s,a,s') \underbrace{\max_{a'} Q(a',s')}_{\text{Best value at the next state = max over all actions in state } s'}}_{\text{Expected value averaged over all possible states } s' \text{ that can be reached from } s \text{ after executing action } a}$$

Q-learning without a Model

- Q-update: after moving from s to state s' using action a

$$\underbrace{Q(a,s)}_{\substack{\text{New estimate} \\ \text{of } Q(a,s)}} \leftarrow \underbrace{Q(a,s)}_{\substack{\text{Old estimate} \\ \text{of } Q(a,s)}} + \underbrace{\alpha}_{\substack{\text{Learning rate} \\ 0 < \alpha < 1}} \left(\underbrace{R(s) + \gamma \max_{a'} Q(a',s') - Q(a,s)}_{\substack{\text{Difference between old} \\ \text{estimate } Q(a,s) \text{ and the} \\ \text{new noisy sample after} \\ \text{taking action } a}} \right)$$

  - TD approach
  - Transition model does not appear anywhere!
  - Once converged, optimal policy can be computed without transition model
    - Completely model-free learning algorithm

Q-learning: Convergence

- Guaranteed to converge to true Q-values given enough exploration
- Very general procedure
  - Because it is model-free
- Converges slower than ADP agent
  - Because it is completely model-free and it does not enforce consistency among values through the model

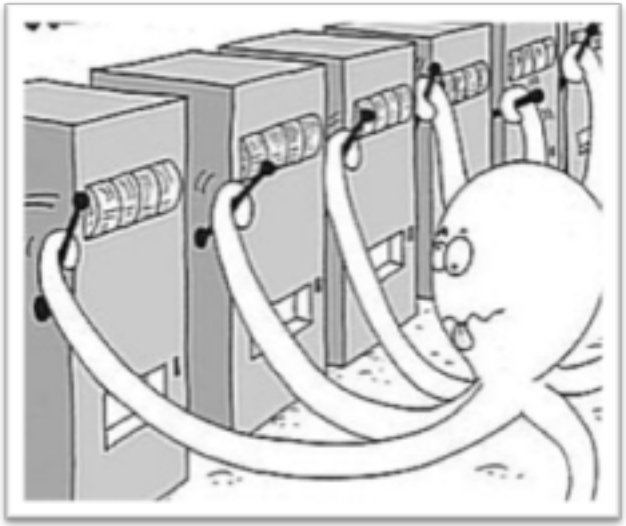
Exploitation vs. Exploration

- Actions are always taken for one of the two following purposes
  - Exploitation: Execute the current optimal policy to get high payoff
  - Exploration: Try new sequences of (possibly random) actions to improve the agent's knowledge of the environment even though current model does not believe they have a high payoff
- Pure exploitation: gets stuck in a rut
- Pure exploration: not much use if you do not put that knowledge into practice

Multi-Arm Bandit Problem

- So far, we assumed that we have a set of epochs of sufficient exploration

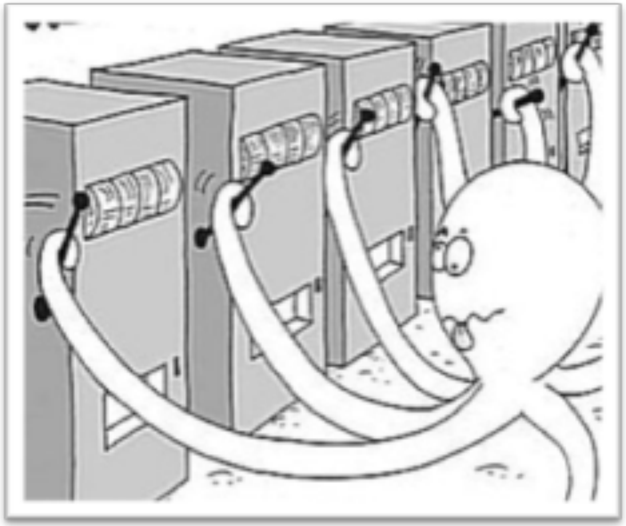
- Multi-arm bandit problem: Statistical model of sequential experiments
  - Name comes from a traditional slot machine (one-armed bandit)
- Question:
  Which machine to play?



Actions

- n arms, each with a fixed but unknown distribution of reward
  - In terms of actions: Multiple actions $a_1, a_2, \dots, a_n$
    - Each a_i provides a reward from an unknown (but stationary) probability distribution p_i
    - Specifically, expectation μ_i of machine i's reward unknown
      - If all μ_i's were known, then the task is easy: just pick $\operatorname{argmax}_i \mu_i$

- With μ_i's unknown, question is which arm to pull



Formal Model

- At each time step $t = 1, 2, \ldots, T$:
  - Each machine i has a random reward $\mathrm{X}_{i,t}$
    - $E[X_{i,t}] = \mu_i$ independent of the past
  - Pick a machine I_t and get reward $X_{I_t,t}$
  - Other machines' rewards hidden
- Over T time steps, we have a total reward of $\sum_{t=1}^{T} X_{I_t,t}$
  - If all μ_i's known, we would have selected $\operatorname{argmax}_i \mu_i$ at each time t
    - Expected total reward $T \cdot \max_i \mu_i$
- Our "regret":

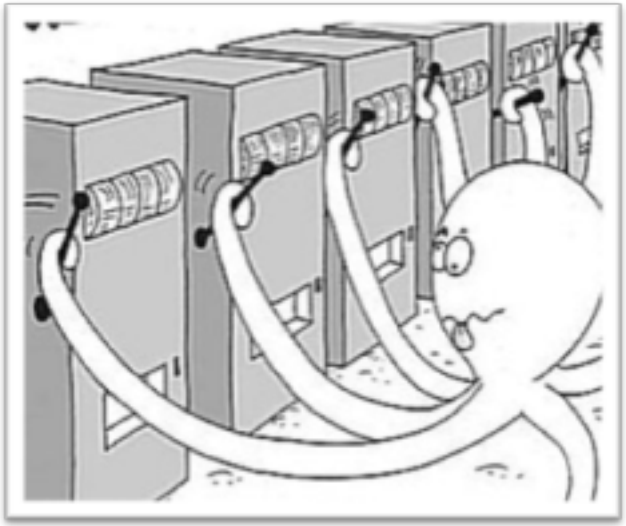
$$T \cdot \max_i \mu_i - \sum_{t=1}^{T} X_{I_t,t}$$

best machine's reward (in expectation)

our reward

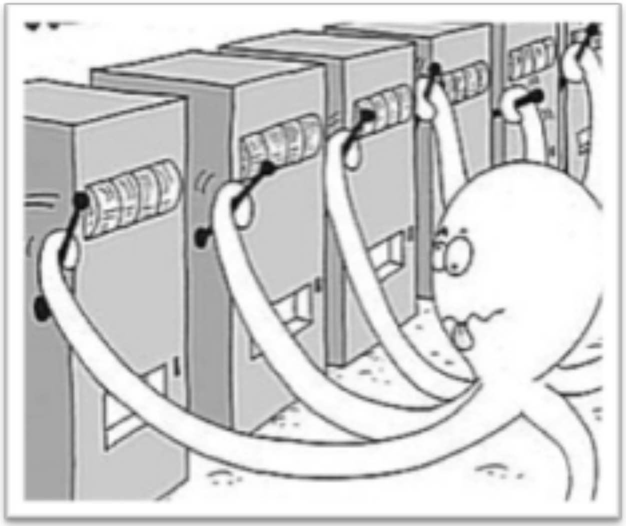
Exploitation vs. Exploration Dilemma

- Exploration: to find the best.
  - Overhead: big loss when trying the bad arms.
- Exploitation: to exploit what we've discovered
  - weakness: there may be better ones that we haven't explored and identified.
- *Question:*
  With a fixed budget,
  how to balance exploration
  and exploitation such that
  the total loss (or regret)
  is small?



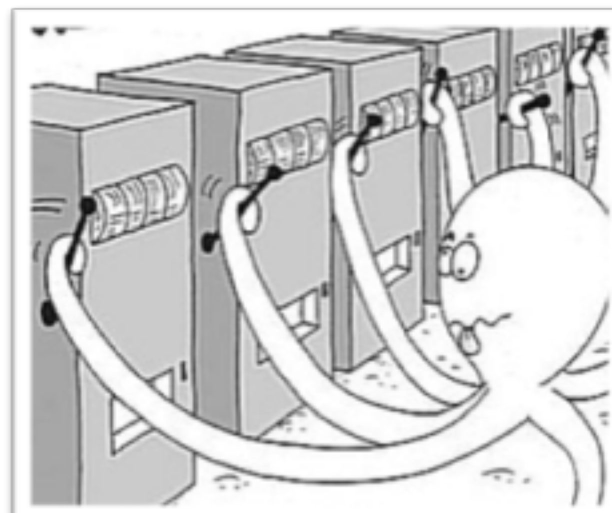
Where does the loss come from?

- If μ_i is small, trying this arm too many times makes a big loss.
  - So we should try it less if we find the previous samples from it are bad
- But how to know whether an arm is good?
- The more we try an arm i, the more information we get about its distribution
  - In particular, the better estimate to its mean μ_i



Where does the loss come from?

- So we want to estimate each μ_i precisely, and at the same time, we do not want to try bad arms too often
  - Two competing tasks
    - Exploration vs. exploitation dilemma
- Rough idea: we try an arm if
  - Either we have not tried it often enough
  - Or our estimate of μ_i so far looks good

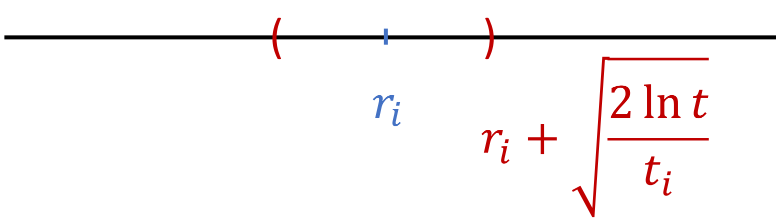


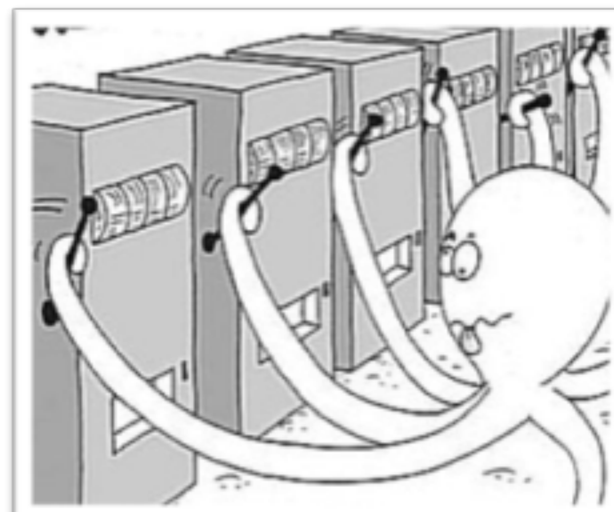
UCB (Upper Confidence Bound) Algorithm

- Assume rewards between 0 and 1
  - If they are not, normalize them

```
UCB(A)
    Try each action a_i once
    loop
        choose an action a_i that has
            the highest value of r_i + √2·ln(t)/t_i
        perform a_i
        update r_i , t_i , t
```

- For each action a_i , let
  - r_i = average reward from a_i
  - t_i = number of times a_i tried
- $t = \sum_i t_i$
- Confidence interval around r_i





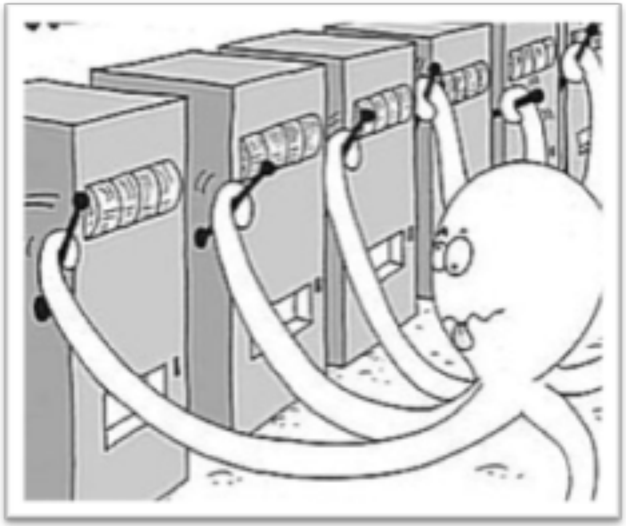
UCB: Performance

- Theorem: If each distribution of reward has support in $[0,1]$, i.e., we have normalised rewards, then the regret of the UCB algorithm is at most

$$O\left(\sum_{i:\mu_i<\mu^*}\frac{\ln T}{\Delta_i}+\sum_{j\in\{1,\dots,n\}}\Delta_j\right)$$

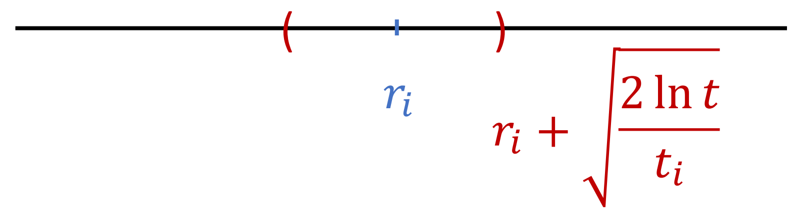
  - $\mu^* = \max_{\mathrm{i}} \mu_i$
  - $\Delta_i = \mu^* - \mu_i$
    - Expected loss of choosing a_i once
  - [without proof]

- Loss grows very slowly with T

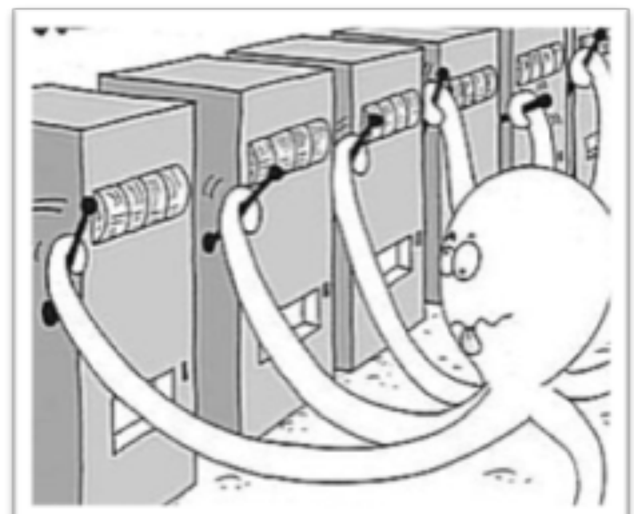


UCB: Performance

- Uses principle of optimism in face of uncertainty
  - We do not have a good estimate $\hat{\mu}_i$ of μ_i before trying it many times
    - We thus give a big confidence interval $[-c_i, c_i]$ for such i
      - $c_i = \sqrt{\frac{2 \ln t}{t_i}}$
    - And select an i with maximum $\mu_i + c_i$

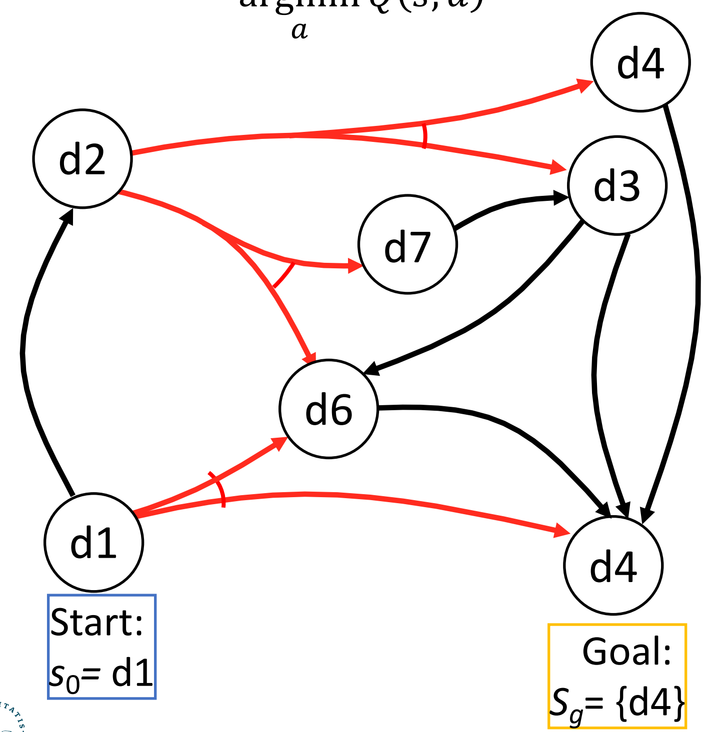


  - If an action has not been tried many times, then the big confidence interval makes it still possible to be tried.
  - I.e., in face of uncertainty (of μ_i), we act optimistically by giving chances to those that have not been tried enough



UCT Algorithm

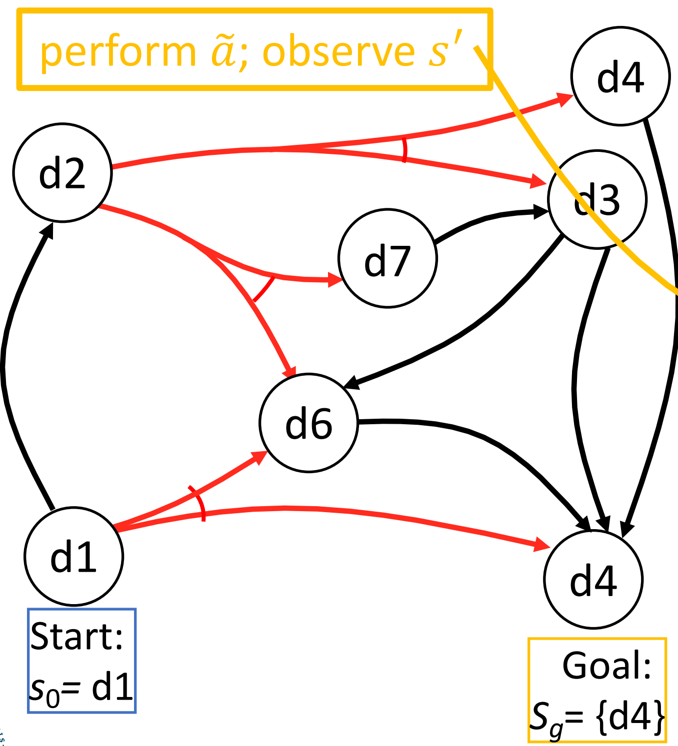
- Recursive UCB computation to compute $Q(s,a)$ for cost
  - Min ops instead of max
  - h horizon (steps into the future)
- Anytime algorithm:
  - Call repeatedly until time runs out
  - Then choose action $\underset{a}{\operatorname{argmin}}\, Q(s,a)$




```
UCT(Σ,s,h)
    if s ∈ S then
        return 0
    if h = 0 then
        return V0(s)
    if s ∉ Envelope then
        add s to Envelope
        n(s) ← 0
        for all a ∈ Applicable(s) do
            Q(s,a) ← 0
            n(s,a) ← 0
    Untried ← {a ∈ Applicable(s)| n(s,a)=0}
    if Untried ≠ Ø then
        ã ← Choose(Untried)
    else
        ã ← argmin_{a∈Applicable(s)}
              {Q(s,a)-C·[log(n(s))/n(s,a)]^½}
    s′ ← Sample(Σ,s,ã)
    cost−rollout ← cost(s,ã) + UCT(s′,h-1)
    Q(s,ã) ← [n(s,ã)·Q(s,ã)+cost-rollout]
               /(1+n(s,ã))
    n(s) ← n(s) + 1
    n(s,ã) ← n(s,ã) + 1
    return cost-rollout
```

UCT as an Acting Procedure

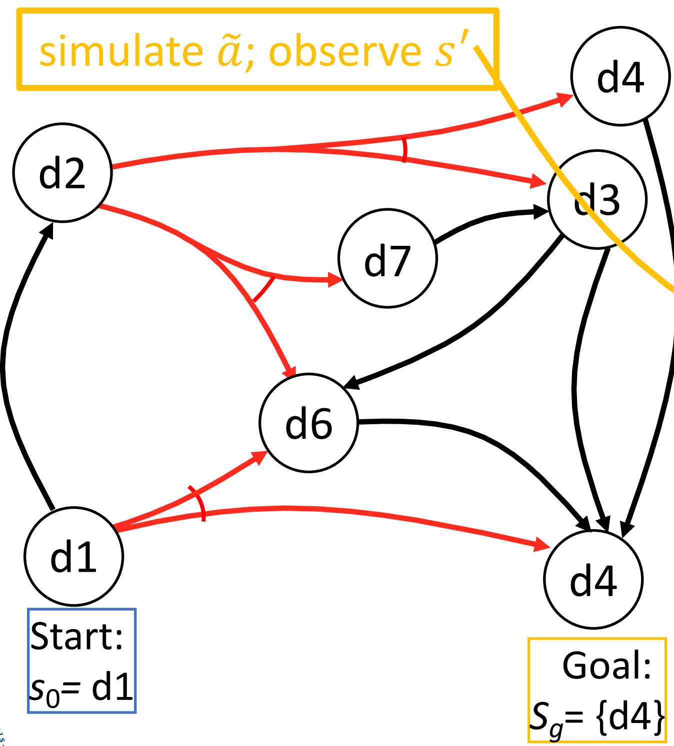
- Suppose probabilities and costs unknown
- Suppose you can restart your actor as many times as you want
- Can modify UCT to be an acting procedure
  - Use it to explore the environment




```
UCT(Σ,s,h)
    if s ∈ S then
        return 0
    if h = 0 then
        return V0(s)
    if s ∉ Envelope then
        add s to Envelope
        n(s) ← 0
        for all a ∈ Applicable(s) do
            Q(s,a) ← 0
            n(s,a) ← 0
    Untried ← {a ∈ Applicable(s)| n(s,a)=0}
    if Untried ≠ ∅ then
        ã ← Choose(Untried)
    else
        ã ← argmin_{a∈Applicable(s)}
            {Q(s,a)-C·[log(n(s))/n(s,a)]^½}
    s′ ← Sample(Σ,s,ã)
    cost-rollout ← cost(s,ã) + UCT(s′,h-1)
    Q(s,ã) ← [n(s,ã)·Q(s,ã)+cost-rollout]
                /(1+n(s,ã))
    n(s) ← n(s) + 1
    n(s,ã) ← n(s,ã) + 1
    return cost-rollout
```

UCT as a Learning Procedure

- Suppose probabilities and costs unknown
  - But you have an accurate simulator for the environment
- Run UCT multiple times in the simulated environment
  - Learn what actions work best




```
UCT(Σ,s,h)
    if s ∈ S then
        return 0
    if h = 0 then
        return V0(s)
    if s ∉ Envelope then
        add s to Envelope
        n(s) ← 0
        for all a ∈ Applicable(s) do
            Q(s,a) ← 0
            n(s,a) ← 0
    Untried ← {a ∈ Applicable(s)| n(s,a)=0}
    if Untried ≠ ∅ then
        ã ← Choose(Untried)
    else
        ã ← argmin_{a∈Applicable(s)}
              {Q(s,a)-C·[log(n(s))/n(s,a)]^½}
    s′ ← Sample(Σ,s,ã)
    cost-rollout ← cost(s,ã) + UCT(s′,h-1)
    Q(s,ã) ← [n(s,ã)·Q(s,ã)+cost-rollout]
                /(1+n(s,ã))
    n(s) ← n(s) + 1
    n(s,ã) ← n(s,ã) + 1
    return cost-rollout
```

UCT in Two-Player Games

- Generate Monte Carlo rollouts using a modified version of UCT
  - Rollout: game is played out to very end by selecting moves at random, result of each playout used to weight nodes in game tree
- Main differences:
  - Instead of choosing actions that minimize accumulated cost, choose actions that maximize payoff at the end of the game
  - UCT for player 1 recursively calls UCT for player 2
    - Choose opponent's action
  - UCT for player 2 recursively calls UCT for player 1
- Produced the first computer programs to play Go well
  - ≈ 2008–2012
- Monte Carlo rollout techniques similar to UCT were used to train AlphaGo

Intermediate Summary

- Run-Lookahead
- Reinforcement learning
  - Passive learning
    - DUE
    - ADP
    - TD
  - Active learning
    - Active ADP
    - Q-learning
  - Multi-armed bandit problem
    - UCB, UCT

Outline per the Book

6.2 Stochastic shortest path problems

- Safe/unsafe policies
- Optimality
- Policy iteration, value iteration

6.3 Heuristic search algorithms

- Best-first search
- Determinisation

6.4 Online probabilistic planning

- Lookahead
- Reinforcement learning

⇒ Next: More on Decision Making