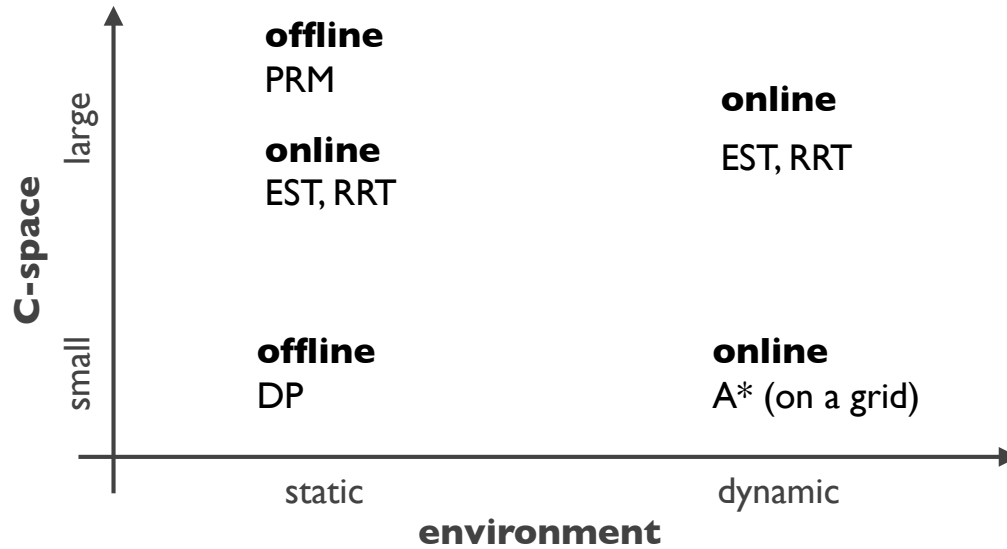


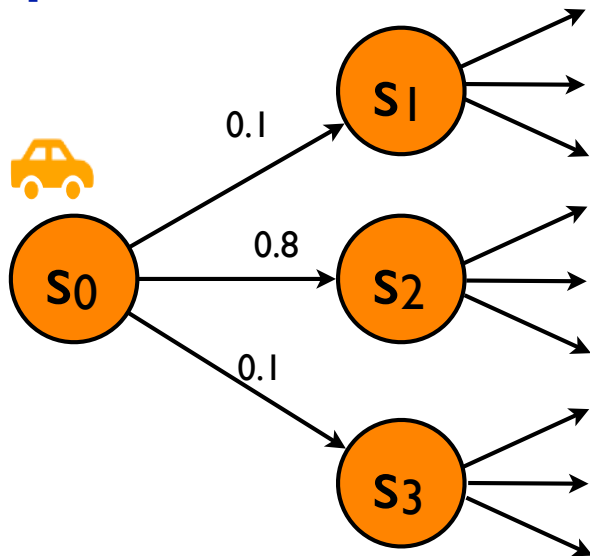
Last lecture ...



Markov chain. A (discrete) Markov chain consists of two basic elements.

- S is a set of states.
Example. A state s may represent the current robot vehicle position.
- $T(s, s') = p(s' | s)$ is a probabilistic state-transition function, which accounts for uncertainty by specifying a probability distribution for the new state s' at time $t + 1$ if the robot is in state s at time t .

Example.

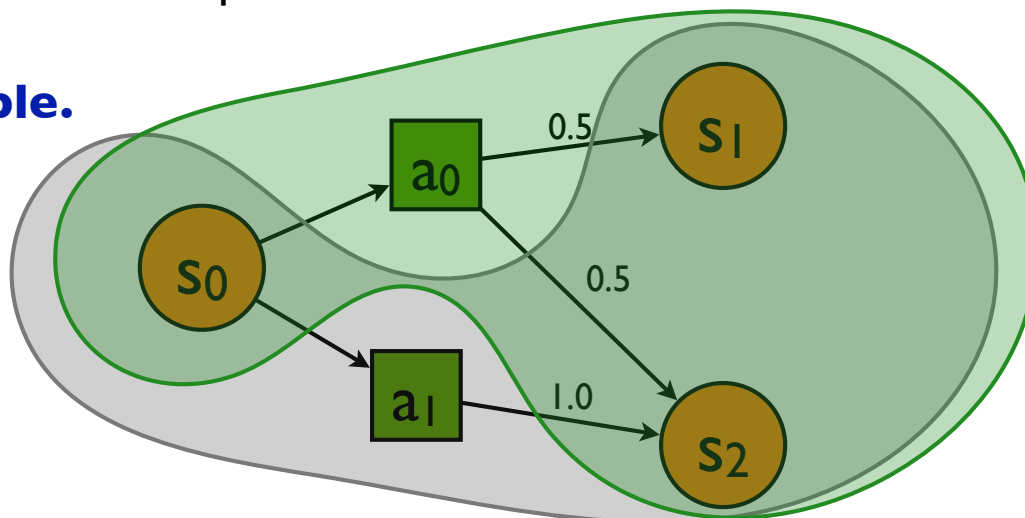


	S0	S1	S2	S3
S0	0	0.1	0.8	0.1
S1	...			
S2	...			
S3	...			

Markov decision process. A (discrete) Markov decision process (MDP) consists of four basic elements.

- S is a set of states.
Example. A state s may represent the current robot vehicle position.
- A is a set of actions.
Example. An action a may represent the speed, acceleration, or steering command for the robot vehicle.
- $T(s, a, s') = p(s' | s, a)$ is a probabilistic state-transition function, which accounts for action uncertainty by prescribing a probability distribution for the new state s' at time $t + 1$ if the robot takes action a in state s at time t .
- $R(s, a)$ is a reward function, which gives the robot a real-valued reward if the robot takes action a in state s . It allows us to prescribe desirable robot behavior.

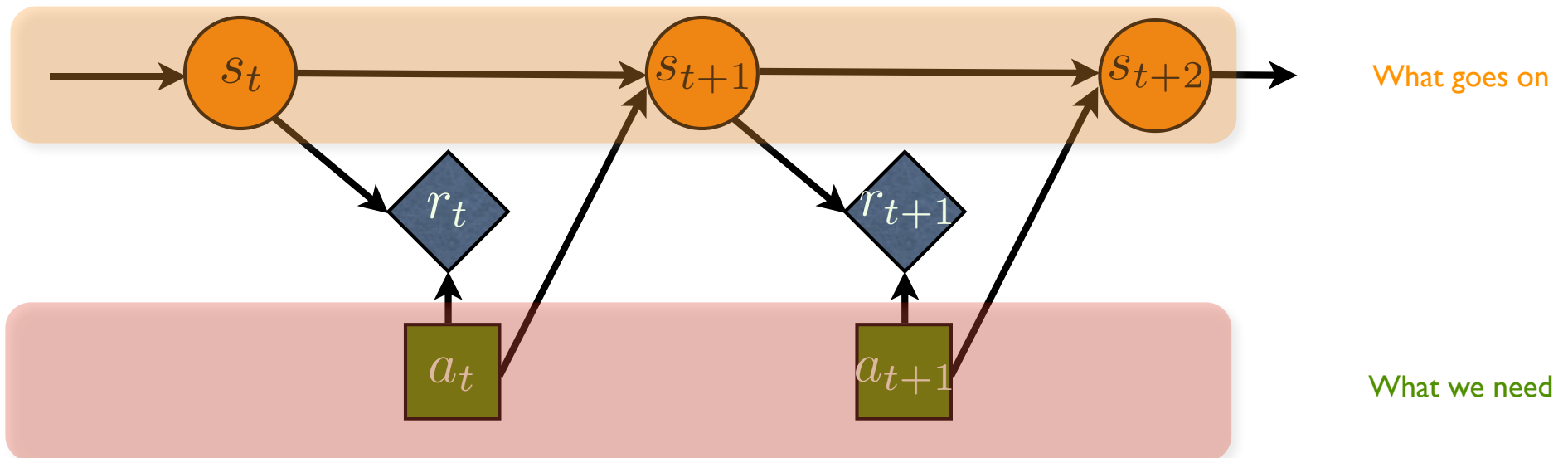
Example.



The MDP considers **action** uncertainty, but still assumes perfect **sensing**.

Finite-horizon MDP planning. Find a sequence of actions $(a_0, a_1, \dots, a_{N-1})$ that maximizes the **value** (expected total reward):

$$V = \mathbb{E} \left[\sum_{t=0}^{N-1} R(s_t, a_t) \right]$$



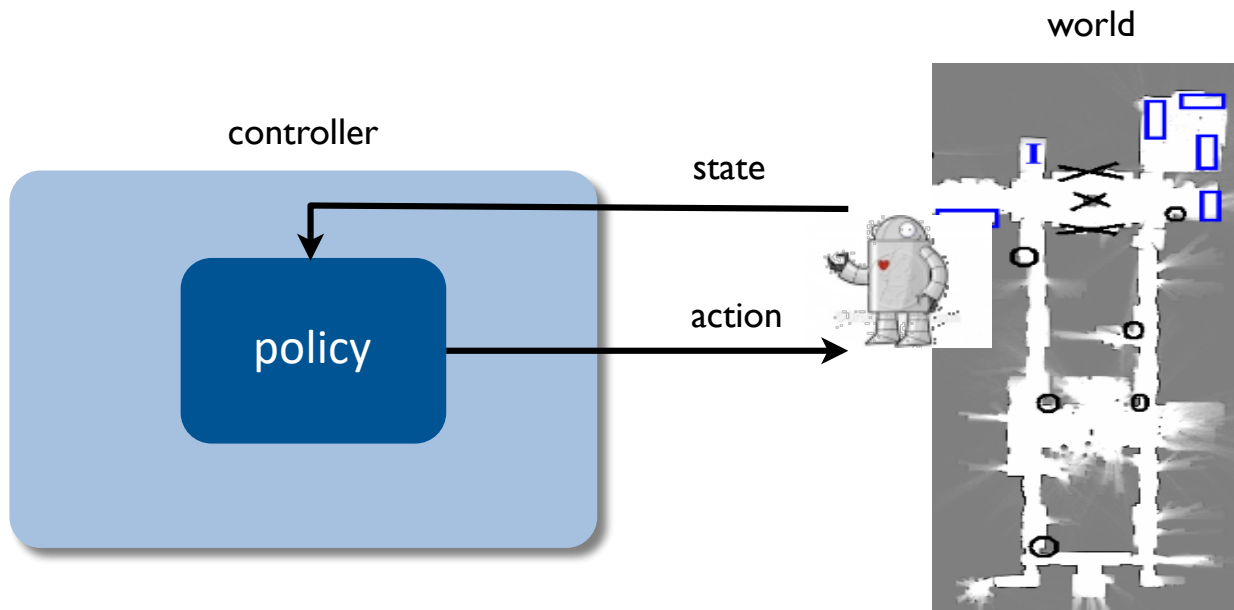
How do we represent the plan?

Open-loop plan. An **open-loop plan** specifies an action at $a_t = \pi(t)$ at each time t . However, the end state of an action is uncertain and is only known after the action is taken. It would seem beneficial to incorporate this information in choosing the action.

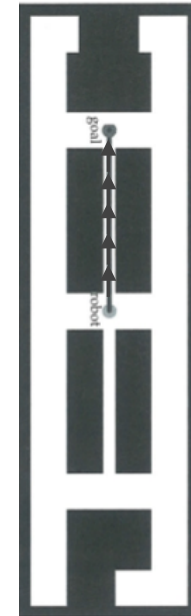
Closed-loop plan. A **closed-loop (feedback) plan** specifies an action at $a_t = \pi(s_t)$ based on the current state s_t at time t .

A closed-loop plan is also called a **policy**.

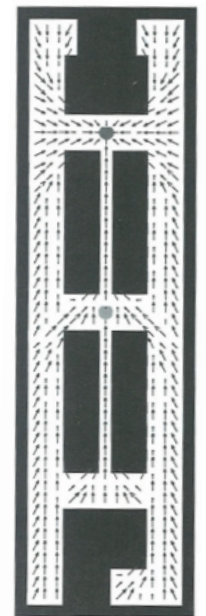
To execute a closed-loop policy,



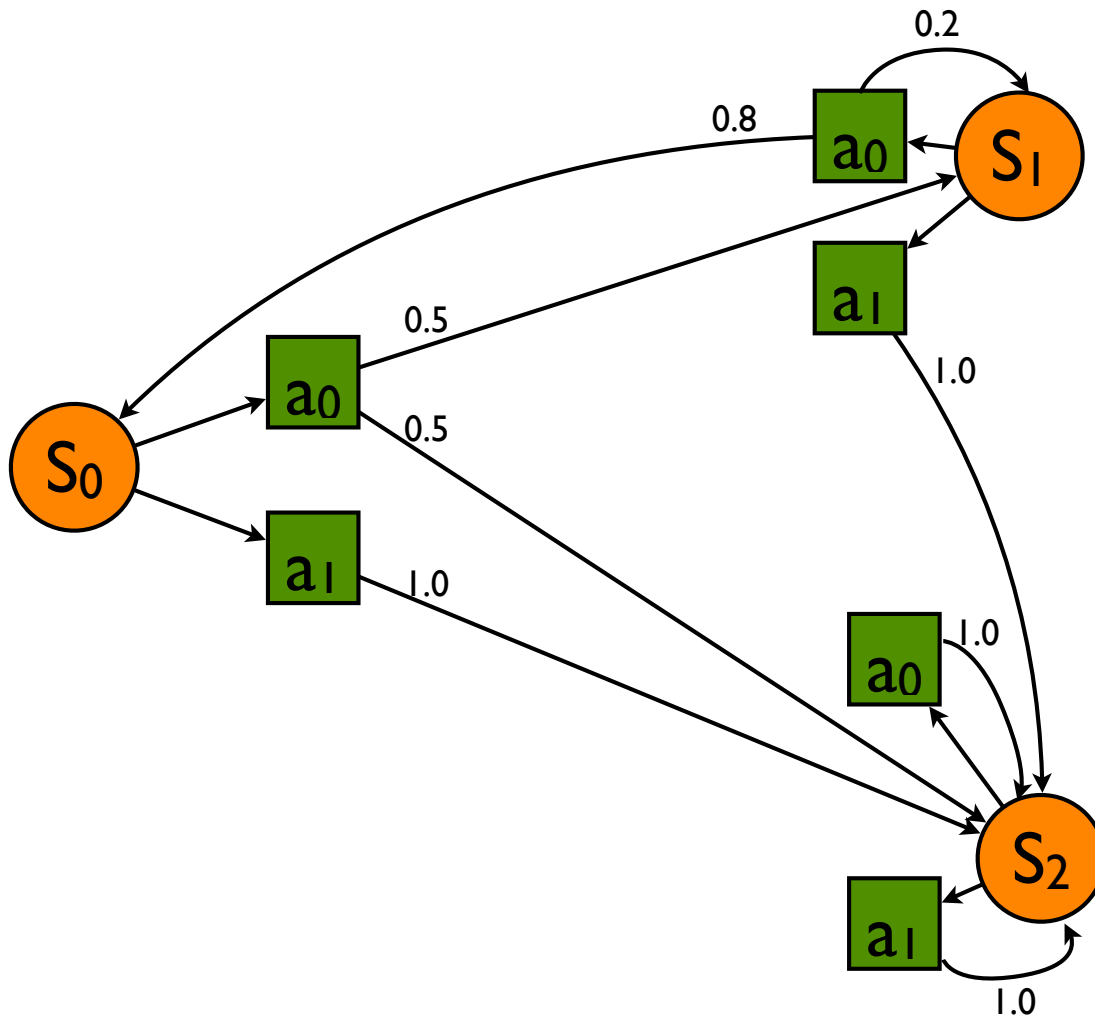
open-loop



closed-loop



Question. s_2 is an absorbing state. Do you see why?



At S_2 , we always go back to S_2 regardless of the action.

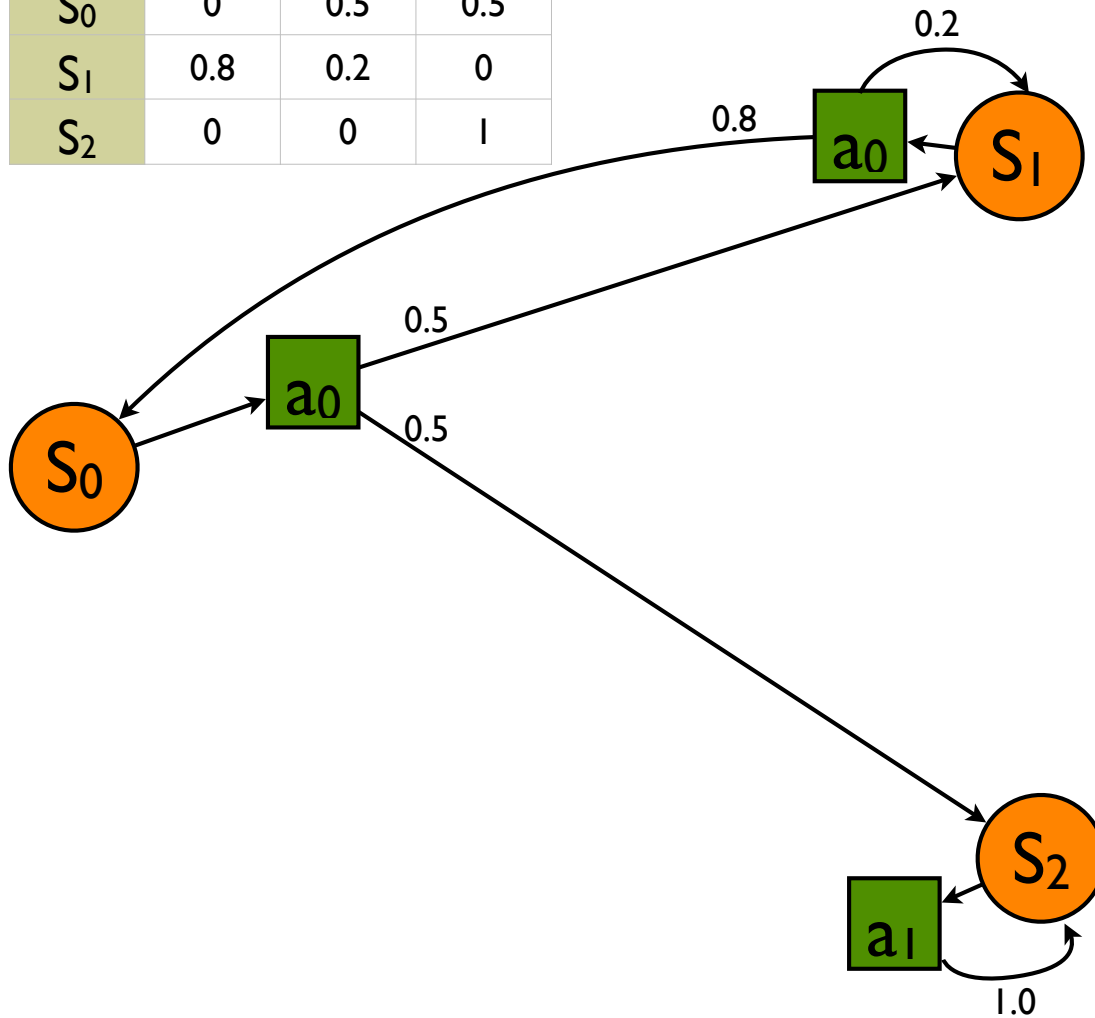
At S_1 , we go to S_2 if we take action a_1 .

At S_0 , we go to S_2 if we take action a_1 .

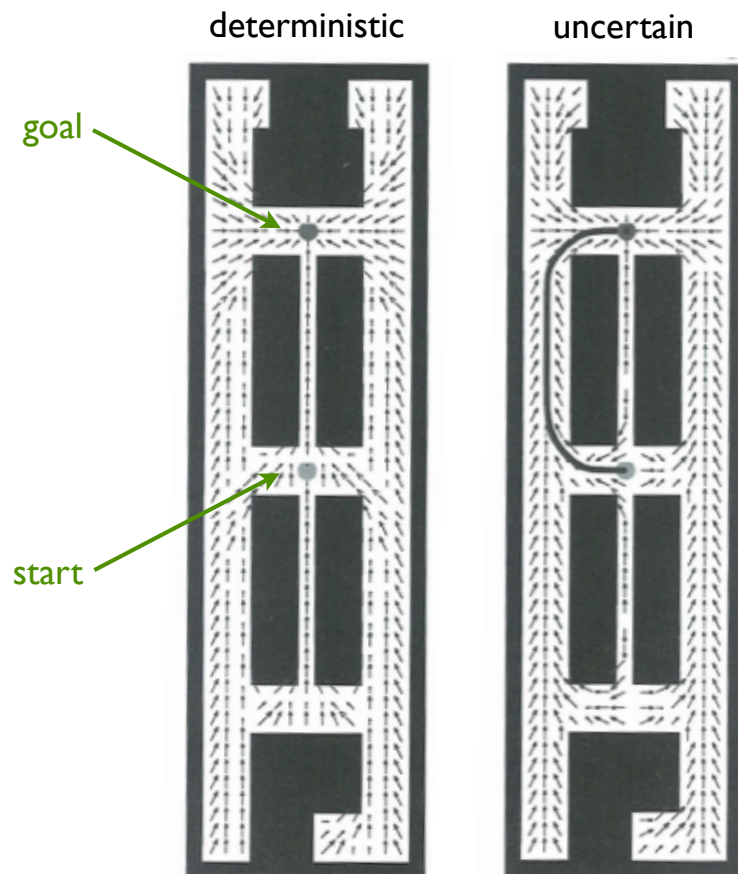
Is it possible for us to shuttle between S_0 and S_1 by always taking action a_0 ? The probability of looping once is $0.8 * 0.5 = 0.4$. To loop n times, the probability is 0.4^n , which approaches 0, as n increases.

Given a policy $\pi(s)$, the MDP becomes a Markov chain with transition matrix

	S_0	S_1	S_2
S_0	0	0.5	0.5
S_1	0.8	0.2	0
S_2	0	0	1



Question. The two environments are similar. However, one has deterministic dynamics: each move achieves its intended location exactly. The other has stochastic dynamics: a move achieves its intended location with probability 0.9 and deviates to a nearby position with total probability 0.1. What are the differences between the resulting policies and why?



The robot may run into the walls in the narrow corridor and incur penalties, because of uncertainty in robot control. Near the entrance of a narrow corridor, it is safer to back out and take an alternative, slightly longer route, rather than “force” the way through.

Value iteration. Value iteration is based on the idea of dynamic programming.

- Initialize

$$V_1(s) = \max_{a \in A} R(s, a)$$

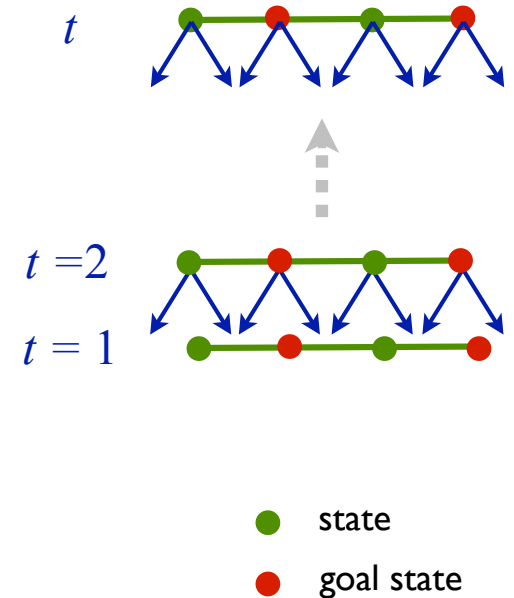
- If there are t time steps to go, recurse

$$V_t(s) = \max_{a \in A} \left\{ R(s, a) + \sum_{s' \in S} T(s, a, s') V_{t-1}(s') \right\}$$

action uncertainty

for $t = 2, 3, \dots, N$.

Value iteration mirrors the dynamic programming algorithm for the shortest path, except for the change required to account for action uncertainty. If all actions are deterministic, value iteration basically reduces to the dynamic programming algorithm for the shortest path.



We can also simplify the initialization step and write down the algorithm more elegantly.

- Initialize

$$V_0(s) = 0$$

- If there are t time steps to go, recurse

$$V_t(s) = \max_{a \in A} \left\{ R(s, a) + \sum_{s' \in S} T(s, a, s') V_{t-1}(s') \right\}$$

for $t = 1, 2, 3, \dots, N$.

Computational efficiency. Each iteration of value iteration takes time $O(|S|^2 |A|)$. For N time steps, the running time of value iteration is $O(|S|^2 |A| N)$ in the worst case.

Question. Can value iteration work for a high-dimensional state space?

Plan execution. We represent a compute policy $\pi_t(s)$ as a table indexed by s and t . The table stores the result of value iteration:

$$\pi_t(s) = \operatorname{argmax}_{a \in A} \left\{ R(s, a) + \sum_{s' \in S} T(s, a, s') V_{t-1}(s') \right\}$$

During the execution, look up the action from the table for state s at time t .

Infinite-horizon MDP planning. Some tasks end naturally by reaching a goal or a time limit. Others may continue forever, e.g., following and monitoring a moving target. Finite-horizon MDP planning is insufficient if the number of time steps N goes to infinity, because

$$V = \mathbb{E} \left[\sum_{t=0}^{\infty} R(S_t, a_t) \right]$$

becomes unbounded. To handle the infinite-horizon planning problem, we introduce the discount factor γ ,

$$V = \mathbb{E} \left[\sum_{t=0}^{\infty} \gamma^t R(S_t, a_t) \right]$$

The discount factor serves multiple purposes:

- γ models the fact that an immediate reward is better than a future reward.
- γ ensures that the value is always finite, provided that $R(s, a)$ is bounded.

Setting $\gamma = 1$, we get back the finite-horizon planning problem, if the maximum number of steps is finite.

Bellman's principle of optimality.

- The optimal value function and the optimal policy are stationary and do not depend on the time t explicitly.
- The optimal value function $V^*(s)$ for an infinite-horizon MDP must satisfy

$$V^*(s) = \max_{a \in A} \left\{ R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') V^*(s') \right\}$$

To compute an optimal policy, we apply the same value iteration algorithm, but change the termination criterion.

- Initialize

$$V_0(s) = 0$$

- If there are t time steps to go, recurse

$$V_t(s) = \max_{a \in A} \left\{ R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') V_{t-1}(s') \right\}$$

until $\|V_t - V_{t-1}\|_\infty \leq \epsilon$, where $\|F(s)\|_\infty = \max_{s \in S} |F(s)|$ is the maximum norm.

One can show that if $\|V_{t+1} - V_t\|_\infty \leq \epsilon$, then

$$\|V_t - V^*\|_\infty \leq \frac{\epsilon}{1 - \gamma}$$

So value iteration computes a near-optimal policy.

Plan execution. Let $\hat{\pi}(s)$ be the policy obtained from value iteration and $\hat{V}(s)$ be the corresponding value function.

- Lookup execution. Store $\hat{\pi}(s)$ as a table indexed by s . Look up the action at state s using $\hat{\pi}(s)$.
- Lookahead execution. Store $\hat{V}(s)$ as a table indexed by s . Perform one-step lookahead and take the action

$$a_t = \arg \max_{a \in A} \left\{ R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') \hat{V}(s') \right\} \text{ one more iteration wrt to the current state } s$$

Question. $R(s, a)$ specifies a state-dependent action reward. How do we specify the following?

- state-independent action reward
- state reward (rather than action reward)
- action reward that depends on both the start and end states.

The first two, $R(a)$ and $R(s)$ are special cases of $R(s, a)$.

Suppose that $R(s, a, s')$ depends on both the start and end states. Then $R(s, a) = \sum_{s' \in S} T(s, a, s')$ $R(s, a, s')$.

Example.

- The state space is a 10x10 grid.
- states* • Gray cells represent obstacles.
- The initial state is the top-left cell.
- rewards* • A reward of 1 is collected when reaching the bottom right cell.
- The discount factor is 0.9.
- actions* • Each cell has 4 neighbors. At each non-obstacle cell, the robot attempts to move to a neighboring cell. The move is successful with probability 3/4. Otherwise the robot moves to a different neighboring cell with equal probabilities.
- The agent always has the option to stay put, which succeeds with certainty.

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0

Example.

Initialization

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

Iteration 1

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0.75	0
0	0	0	0	0	0	0	0.75	1	0
0	0	0	0	0	0	0	0	0	0

Iteration 2

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0.51	0
0	0	0	0	0	0	0	0.56	1.43	0
0	0	0	0	0	0	0	1.43	1.9	0
0	0	0	0	0	0	0	0	0	0

$V_0(s) = 0$ for all s .

$V_1(s)$

* For move-right to reach s' , the value is $0.75 \times 1 + 0 = 0.75$
 immediate reward $0.75 \times 1 = 0.75$
 future reward $0.9 \times 0 = 0$

* For all other actions, the value is $0 + 0 = 0$.

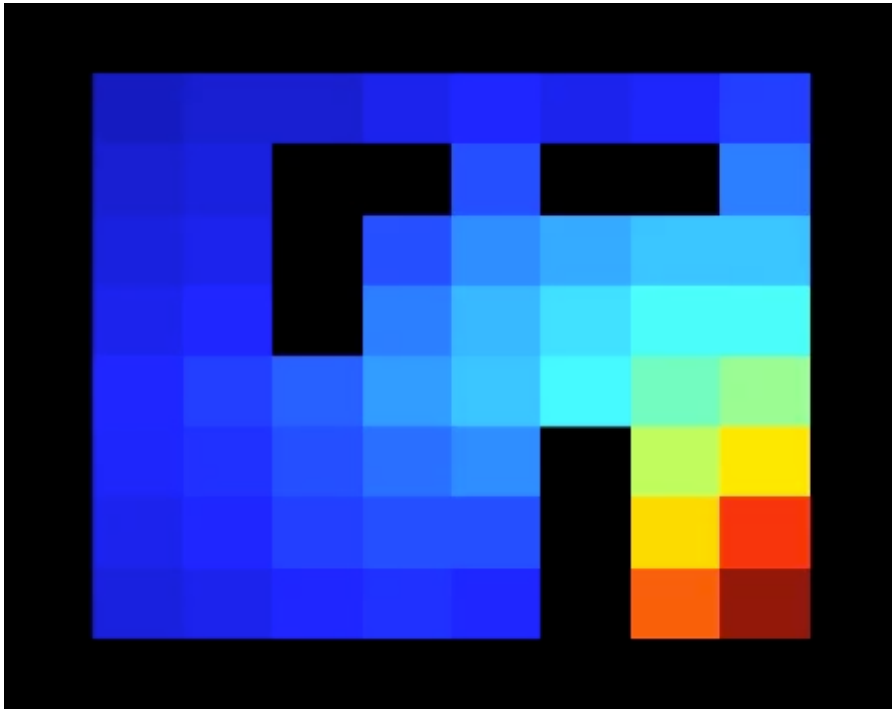
So $V_1(s') = 0.75$.

$V_2(s)$

* For move-right at s' , the value is 1.425
 immediate reward 0.75×1
 future reward $0.9 \times (0.75 \times 1 + 1/12 \times 0 + 1/12 \times 0 + 1/12 \times 0)$
 $= 0.675$

* For move-down, the value is 0.5625
 immediate reward 0
 future reward $0.9 \times (0.75 \times 0.75 + 1/12 \times 0.75 + 1/12 \times 0 + 1/12 \times 0) = 0.5625$

Example.



Example.

Iteration 50

0	0	0	0	0	0	0	0	0	0
0	0.44	0.54	0.59	0.82	1.15	0.85	1.09	1.52	0
0	0.59	0.69	0	0	1.52	0	0	2.13	0
0	0.75	0.90	0	0	2.12	2.55	2.98	3.00	0
0	0.95	1.18	0	2.00	2.70	3.22	3.80	3.88	0
0	1.20	1.55	1.87	2.41	2.92	3.51	4.52	5.00	0
0	1.15	1.47	1.74	2.05	2.25	0	5.34	6.47	0
0	0.99	1.26	1.49	1.72	1.74	0	6.69	8.44	0
0	0.74	0.99	1.17	1.34	1.27	0	7.96	9.94	0
0	0	0	0	0	0	0	0	0	0

Question. Given the matrix above, how do we find the best action at each state?

Question. What is the exact value $V^*(\text{goal})$?

The table gives the estimated value function. We can apply lookahead execution.

$$1 + 0.9 + 0.9^2 + \dots = 10$$

One intuitive way of thinking about the value iteration is to sweep over all states systematically and apply the backup operation to update $V(s)$ at every state s :

$$V_t(s) = \max_{a \in A} \left\{ R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') V_{t-1}(s') \right\}$$

One can show that if every state s is updated infinite number of times, then $V_t(s)$ converges to $V^*(s)$ regardless of the order of backup operations.

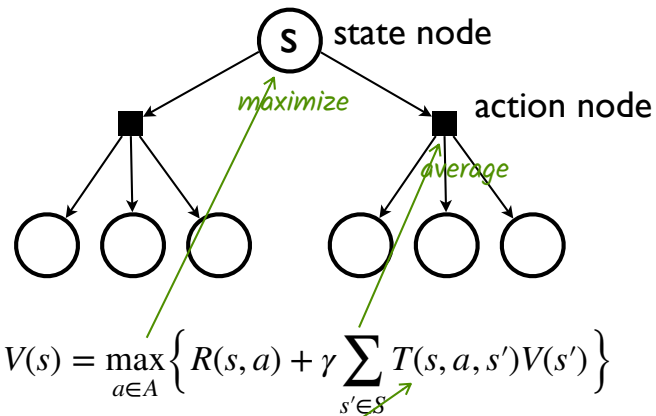
Is it really necessary to apply backup at every state s ? In the example, the first two iterations of value iteration show that the value of most states do not change. They remain 0 for a considerable number of iterations. If the a state changes in the value, its neighboring states may change their values in the next iteration. The change “propagates”.

How do we choose the states to update? How do we order them?

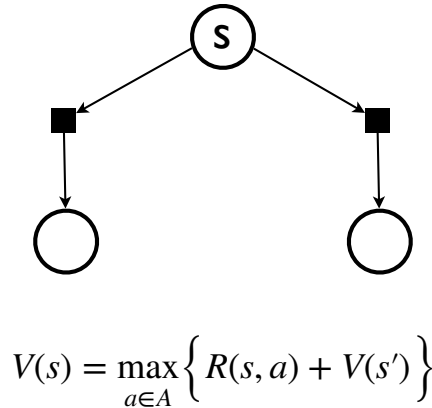
Forward search. Perform heuristic tree search, starting from an initial state S_0 . Initialize $V(s) = R_{\max}/1 - \gamma$, which is optimistic. Back up at every node encountered:

$$V(s) = R_{\max}(1 + \gamma + \gamma^2 + \dots)$$

probabilistic transition

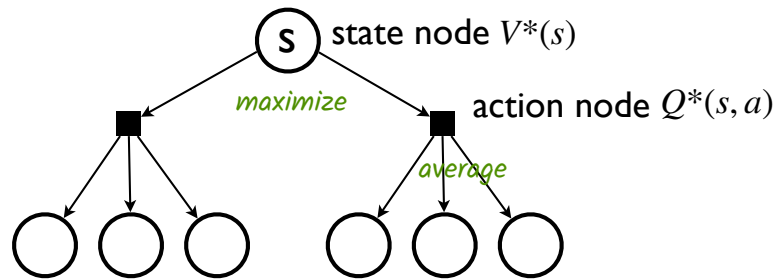


deterministic transition

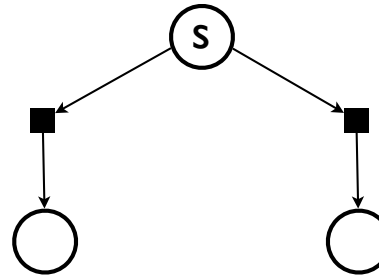


Forward search. Perform heuristic tree search, starting from an initial state S_0 . Initialize $V(s) = R_{\max}/1 - \gamma$, which is optimistic. Back up at every node encountered:

probabilistic transition



deterministic transition



$$V^*(s) = \max_{a \in A} Q^*(s, a),$$

$$Q^*(s, a) = \sum_{s' \in S} T(s, a, s') (R(s, a) + \gamma V^*(s'))$$

backup $V(s) = \max_{a \in A} \left\{ R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') V(s') \right\}$

$$V(s) = \max_{a \in A} \left\{ R(s, a) + V(s') \right\}$$

Define the **state value function** $V^\pi(s)$ for a policy $\pi(s)$ as

$$V^\pi(s) = E[R(s_1, a_1) + \gamma R(s_2, a_2) + \dots \mid s_1 = s, \pi]$$

where each action is chosen according to π and the expectation is taken with respect to the distribution of uncertain action consequences. Similarly, define the **state-action value function**, also called the Q-function, as

$$Q^\pi(s, a) = E[R(s_1, a_1) + \gamma R(s_2, a_2) + \dots \mid s_1 = s, a_1 = a, \pi]$$

Let $V^*(s)$ and $Q^*(s)$ be the optimal state-value function and the optimal action-value function over all policies. Then,

$V^*(s)$ and $Q^*(s, a)$ are closely related, and we can obtain one from the other. Further,

$$Q^*(s, a) = \sum_{s' \in S} T(s, a, s') \left(R(s, a) + \gamma \max_{a \in A} Q^*(s', a) \right)$$

Compare with

$$V(s) = \max_{a \in A} \left\{ R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') V(s') \right\}$$

Monte Carlo tree search (MCTS).

- Search heuristic
- Improved initialization
- Sampled backup

Search heuristic. For the search heuristic, store $Q(s, a)$ at each action node. Store also $N(s, a)$, which counts the number of time that the action node has been visited during the search. At a state node s , store $N(s) = \sum N(s, a)$, which sums over all child action nodes of s .

Start at the current state. At a state node, choose the action node (s, a) that maximizes the **upper confidence bound (UCB)**:

$$Q(s, a) + \beta \sqrt{\frac{\log N(s)}{N(s, a)}}$$

Exploitation
favor action nodes that provide high values.

Exploration
favor action nodes that are less explored.

β
a constant that tunes the trade-off between exploration and exploitation.

At an action node (s, a) , sample a state node s' according to the distribution $p(s' | s, a) = T(s, a, s')$. Repeat and traverse a path $(s_1, a_1, s_2, a_2, \dots)$ down until the search reaches a leaf node. We expand the leaf node and add its children to the tree.

Initialization. For a newly-added action node (s, a) , what is the initial value $Q(s, a)$? A simple choice is the optimistic upper bound:

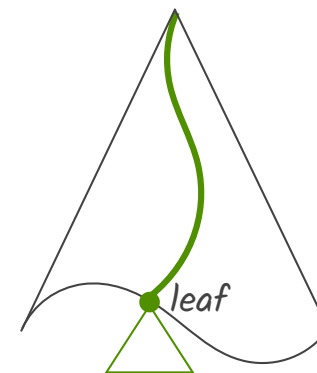
$$Q(s, a) = R(s, a) + \frac{\gamma R_{\max}}{1 - \gamma}$$

Maybe we can do better by handcrafting a policy π and use $Q^\pi(s, a)$. Often, we have intuition about a reasonable, but not necessarily optimal policy. How do we calculate $Q^\pi(s, a)$?

- Solve equations.
- Alternatively, perform Monte Carlo simulation, sometimes called **roll-outs** in this context.

$Q^\pi(s, a)$ is the expected total reward given $s_1 = s$ and $a_1 = a$. To simulate,

- $s_1 = s, a_1 = a$,
- Simulate a_1 in s_1 and obtain s_2 .
- $a_2 = \pi(s_2)$



Here we perform maximization. So an optimistic bound is an overestimate.

Solve a set of simultaneous equations:
For each (s, a) pair,

$$Q^\pi(s, a) = \sum_{s' \in S} T(s, a, s') (R(s, a) + Q^\pi(s', \pi(s')))$$

Both T and R are given. For each state-action pair (s, a) , $Q(s, a)$ is an unknown.

- Simulate a_2 in s_2 and obtain s_3 .
- $a_3 = \pi(s_3)$
- ...

for a chosen constant number of steps. Repeat and get N trajectories of the form $(s_1, a_1, r_1, s_2, a_2, r_2, \dots)$. Average the total rewards over all simulated trajectories.

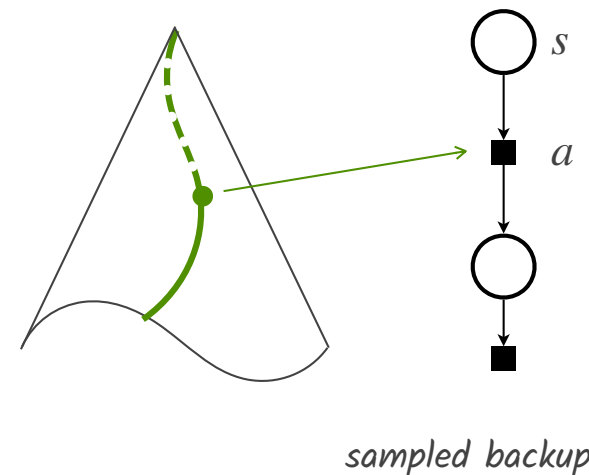
Sampled backup. To update the value of $Q(s, a)$, let q the total reward at (s, a) resulting from the latest tree traversal.

$$Q(s, a) \leftarrow \frac{N(s, a)}{N(s, a) + 1} Q(s, a) + \frac{1}{N(s, a) + 1} q$$

update the average

It is called the sampled backup, because every update is based on one additional traversal.

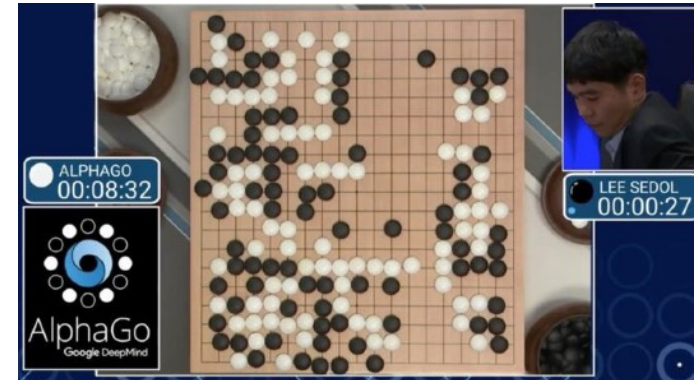
MTCS puts the three ideas together. It repeatedly traverses the tree from the root to a leaf, under the guidance of the UCB bound. It expands the leaf node. It then backtracks the traversal path and updates $Q(s, a)$ at every action node along the way.



Intuitively MTCS tries out many policies by performing MC simulations. It orders the trials under the guidance of UCB. It then chooses the best action according to the estimated $Q(s, a)$.

Does MCTS work?

- Yes. It is the underlying search method employed in AlphaGo. It is among the most successful for planning in very large state spaces.
- No. It converges to the optimal action, but the worst-case running time is a tower of exponentials with respect to the depth of the search tree.





The MDP model consists of four elements: state space S , action space A , a probabilistic state-transition function $T(s, a, s')$, and a reward function $R(s, a)$. What if we do not know $T(s, a, s')$ or $R(s, a)$?

MCTS does not use $T(s, a, s')$ or $R(s, a)$ explicitly, but relies on a “black-box” simulator to provide **many** simulation trajectories during the planning.

If we do not have a simulator, Use real-world data directly. However, large amounts of data may not be easily accessible.

Reinforcement learning. Each experience is a trajectory $\xi = (s_1, a_1, r_1, s_2, a_2, r_2, \dots)$. Assume an underlying MDP model unknown a priori. Find an optimal policy $\pi(s)$ from a set of experiences.

indirect  • **Model-based** approach. Use tuples (s_t, a_t, s_{t+1}) to learn $T(s, a, s')$. Use tuples (s_t, a_t, r_t) to learn $R(s, a)$. Apply a planning algorithm to compute $\pi(s)$.

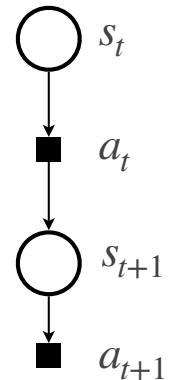
direct  • **Model-free** approach. Learn $\pi(s)$ or $Q(s, a)$ directly.

The word “model” is used a lot in the literature, but its precise technical meaning is not well-defined and may be impossible to define in general. Here it refers to T and R .

SARSA. Given $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$, update $Q(s, a)$:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(\underbrace{r_t + \gamma Q(s_{t+1}, a_{t+1})}_{\text{combine new data } r_t \text{ and old } Q \text{ value to form new estimate of } Q(s_t, a_t)} - \underbrace{Q(s_t, a_t)}_{\text{difference between new and old } Q} \right)$$

learning rate



To acquire “useful” experiences, we need to choose the actions a_1, a_2, \dots judiciously. In the forward search algorithm, the heuristic plays exactly the same role.

Learn the Q-function.

- Use the policy associated with the current Q :

exploitation $a_t = \arg \max_{a \in A} Q(s_t, a)$

This may be a self-fulfilling prophecy. It ignores alternatives that are very different, just like the greedy heuristic.

- Choose an action a_t uniformly at random. This considers all policies equally, but is clearly inefficient.

exploration

Use the ϵ -greedy mixture strategy: apply the exploitation strategy with probability $1 - \epsilon$ and the exploration strategy with probability ϵ .

Input:

S: state space

A: action space

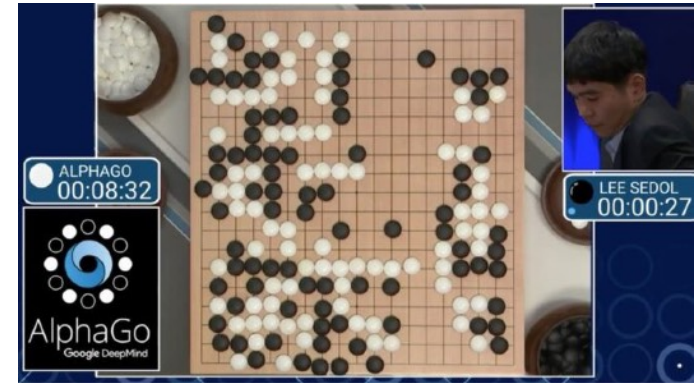
Output:

$Q(s,a)$: action-value function

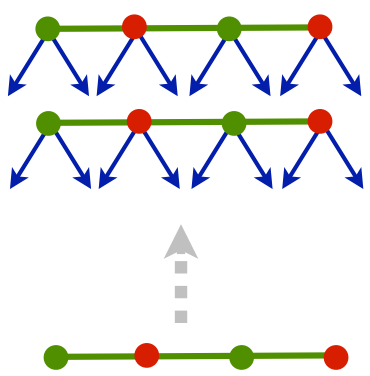
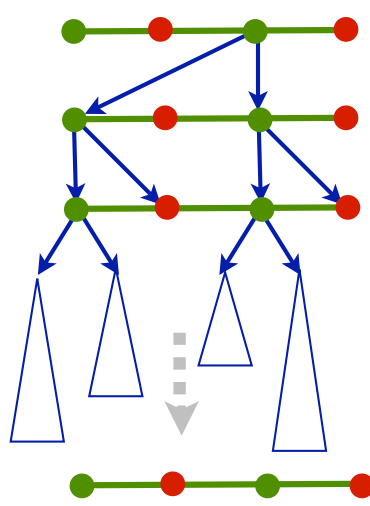
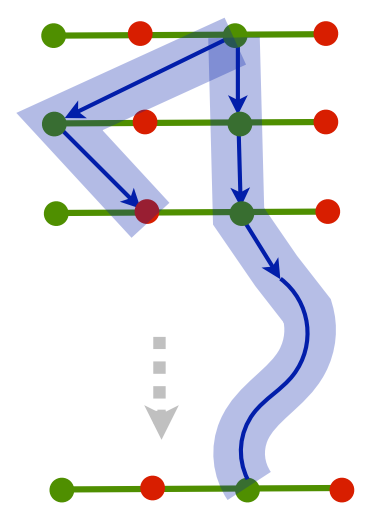
SARSA

```
1: Initialize  $Q(s,a) = 0$  for all  $s \in S$  and  $a \in A$ .
2: repeat M times
3:   Choose s randomly.
4:   repeat N times
5:     Take action a at state s according to the  $\epsilon$ -greedy
       policy derived from  $Q(s,a)$ , and receive r, s'.
6:     Let a' be the action at s' according to the  $\epsilon$ -greedy
       policy derived from  $Q(s,a)$ .
7:      $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma Q(s',a') - Q(s,a))$ .
8:      $s \leftarrow s'$ .
9:      $a \leftarrow a'$ .
12:   if s is goal then exit.
13: return Q
```

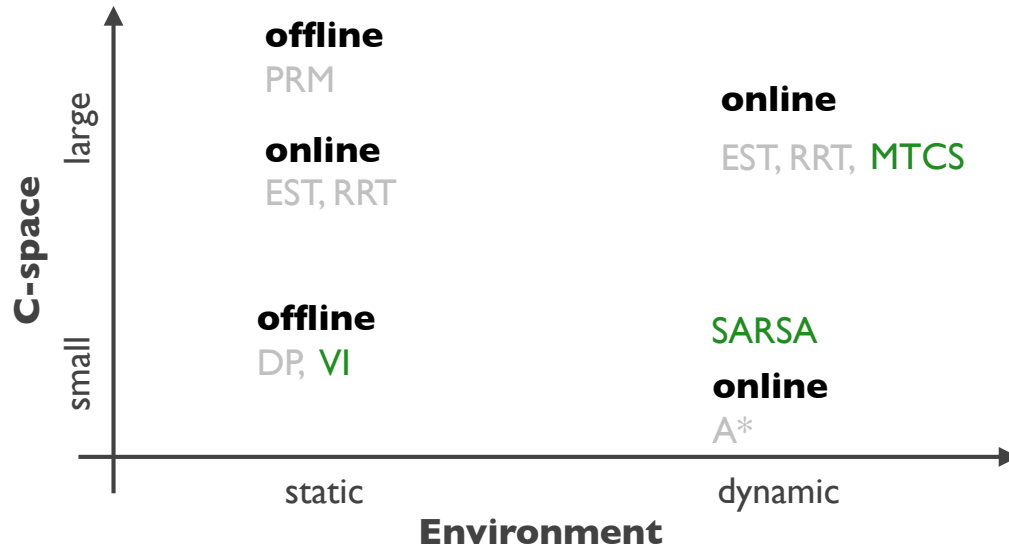
Question. Would you use the model-based or the model-free approach to reinforcement learning for Go? Why?



Summary.

	dynamic programming value iteration	tree search MCTS	reinforcement learning SARSA
	 <p>● state ● goal state</p>		
Input	S, A, T, R	S, A T, R or simulator	S, A simulated or real experiences
Output	$V(s)$ for all $s \in S$	$Q(s, a)$ at a fixed initial state s	$Q(s, a)$ over an initial state distribution over S
Search	exhaustive enumeration	UCB	ϵ -greedy
Backup	full distributional backup	sampled backup	sampled backup

Summary.



Required readings.

Supplementary readings.

- [Sutton & Barto] Sect 4.4-4.5, 8.9-8.11

Key concepts.

- Value iteration
- Monte Carlo tree search
- SARSA
- Exploration and exploitation