Revision notes - CS4246

Ma Hongqiang

November 8, 2018

Contents

1 Deterministic Planning

1.1 Problem Solving

Definition 1.1 (Problem).
A **problem** is defined by 5 components:

1. **Initial State**
2. Possible **Actions**. ACTION(s) returns set of actions that can be executed in s.
3. **Transition model**. RESULT(s, a) returns state that results from doing a in s.
   The sets of all states reachable from the initial state is called the state space. Transitional model forms a directed graph over all the states.
4. **Goal test**.
5. **Path Cost**. Often it is the sum of the step cost, $c(s, a, s')$.

When the number of states is not too large, we can solve the optimal path by using the UNIFORM-COST-SEARCH.
A^* search is the same as UNIFORM-COST-SEARCH except that it uses $f(n) := g(n) + h(n)$ to sort the priority queue, where

- $g(n)$ is the cost to reach node n
- $h(n)$ is a **heuristic** that estimates the cheapest cost from n to the goal state

Theorem 1.1 (Optimality of A^*).
A^* is optimal under tree search if $h(n)$ is admissble, i.e., h never overestimates the cost to the goal.
A^* is optimal under graph search if $h(n)$ is consistent, i.e.,

$$h(n) \leq c(n, a, n') + h(n')$$

where $c(n, a, n')$ is the cost for going from n to n' using a.

Generally, admissible heuristics comes from a *relaxed* problem, where there are fewer restrictions on the actions.

Definition 1.2 (Domination).
We say h_2 dominates h_1 if $h_2(n) \geq h_1(n)$ for all n in state space.

If h_2 dominates h_1, h_2 will *always* explore fewer nodes with A^*.
Generally, one useful property to identify the heuristic is as below: suppose $S \subseteq \Omega$, and there is $f : \Omega \to \mathbb{R}$. Then, we have

$$\inf f(S) \geq f(\Omega)$$

Here, S is the state of the problem and Ω an relaxed problem, where f is the function that maps states to heurstic's costs.

However, when state space is defined in terms of state variables, i.e. **factored representation**, it is usually exponentially large with respect to the number of variables. Therefore, A^* algorithm will run out of memory as it stores all states. In such cases, **depth first search** should be used, which only requires $O(m)$ memory, where m is the depth of the search.
Other variants of depth-first search includes **iterative deepening search** and **iterative deepening A^***.

1.2 Classical Planning

Classical planning usually uses a language called **Planning Domain Definition Language**(PDDL), in which

- Initial state,
- Actions available in a state,
- Result of applying an action
- Goal test

are defined.

Definition 1.3 (State).
A **state** is a conjunction of fluents.
Fluents are state variables, representing variables that can change through time. In PDDL, fluents are **ground boolean variables**.
A state is then a **conjunction of fluents**.

In description of state, **database semantics** are used where

- Fluents not mentioned are false, because of the closed world assumption.
- Unique name assumption, which means variable of distinct names are distinct.

The state description forbids the below scenarios:

- $\text{At}(x, y)$, as the boolean variable is not grounded
- $\neg$Poor because of the negation
- $\text{At}(\text{Father}(\text{Fred}), \text{Sydney})$ because PDDL does *not* allow functions.

Definition 1.4 (Action Schema, Action).
A set of actions is specified by an **action schema**, which is a **lifted representation**, lifted from propositional logic to a restricted subset of *first order* logic.
A schema consists of

1. an action name,
2. a list of all variables
3. a precondition
4. an effect

By instantiating the variables, a schema can give **ground actions**.
The precondition and effect are conjunctions of literals(positive or negative atmoic sentences). Here, precondition specifies states where the action can be executed: action a is **applicable** in state s if the preconditions are satisfied by s.

A set of action schemas defines a planning domain.

Definition 1.5 (Result).
The **result** of executing action a in state s is a state s' which is a set of fluents formed as follows:

- Start from s
- Remove fluents that appears in teh action's effect as negative literals
- Add fluents that appear in the action's effect as positive literals

$$\textsc{Result}(s, a) = (s - \textsc{Del}(a)) \cup \textsc{Add}(a)$$

Definition 1.6 (Initial State, Goal).
A specific problem is defined by adding an initial state and a goal, where

1. The **initial state** is a conjunction of ground atoms.
2. The **goal state** is conjunction of literal, positive or negative, that may contains variables.

Remark: Variables are treated as existentially quantified.

Solution to a planning problem is essentially a path from initial state to goal. We can find this path from

1. Forward search
2. Backward search

Forward search has the problem of

- large state space, which is exponential to the number of state variables.
- Large action space

but there exists strong domain independent heuristics that can be automatically derived, which makes forward search feasible for many problems.

In contrast, backward search, also called **relevant-state** search, only considers actions that are relevant to the goal, or current state. To formalise this, we distinguish state and description:

Definition 1.7 (State, Description)**.**
In a state, every variable is assigned a value. Therefore, for n ground fluents, there are 2^n ground states.
However, for n ground fluents, there are 3^n descriptions, as each fluent can be positive, negative or not mentioned.
We can use description with omitted fluent to describe a set of states.

In backward search, we regress from a state description to a predecessor state description.

Theorem 1.2 (Regression in PDDL)**.**
Given goal g and action a, regression from g over a gives description g', where

$$g' = (g - \textsc{Add}(a)) \cup \text{Precondition}(a)$$

For backward search, we want actions used for regression to be **relevant**, in the following sense.

Definition 1.8 (Relevant Action)**.**
An action is relevant if

1. At least 1 of the action's effect must unify with an element of the current goal
2. Must not have any effect that negate an element of the current goal

When unifying, we substitute the most general unifier in the regressed state g', to keep the branching factor down without ruling out any solution.

1.2.1 Heuristics for Foward Search

Definition 1.9 (Ignore Pre-condition Heuristic)**.**
Ignore Pre-condition Heuristic drops all pre-conditions.
Since it solves a relaxed problem, it never overestimates, which makes it admissible.
Number of steps is roughly number of unsatisfied goals, except some actions that can satisfy multiple goals, or undo some goals.

If relaxed further to remove all effects except literals in teh goal, problem becomes minimum number of actions, such that the union of effects satisfies the goal, which is known as **set cover problem**, a NP-hard problem. However, it is known that there is a $\log n$-approximation algorithm for set cover.

Definition 1.10 (Ignore Delete List Heuristic).
Ignore Delete List Heuristic assumes goal and pre-condition contains only positive literals. Then this heuristics allows monotonic progress towards goal.
Since it is a relaxed problem, it is admissible. However, the relaxed problem is still NP-hard, and hill climbing gives an approximate solution.

Definition 1.11 (Problem Decomposition).
If we can divide goal g into $g_1, \ldots, g_n$ subgoals, which $a_1, \ldots, a_n$ can be used to solve. If each subproblem uses an admissible heuristic, then **taking max** is admissible.
Furthermore, if we assume **subgoal independence**, then sum of the cost of solving each subgoal

- is optimistic when there is negative interaction
- is pessimistic when there is positive interaction, thus inadmissible.

If the sum is admissible, then it will be a better heuristic than taking max.

1.3 SATPlan

One way to do planning is to transform the planning problem into a **Boolean satisfiability**(SAT) problem, and solve using a SAT solver.
It is known that SAT is NP-complete, but SAT solvers are effective in practice by exploiting various heuristics.

Solving SAT requires finding an assignment to variables that will make a Boolean formula TRUE, or declare no assignment exists.

SAT solvers usually takes input in Conjunctive Normal Form(CNF) which is a conjunction of disjunction of literals. It is known that any boolean formula can be converted into CNF.

Theorem 1.3 (Translate PDDL into SAT).
We need the following steps to translate PDDL to SAT:

1. Propositionalise the actions, by replace each action schema with the set of ground actions by substituting constants for each variable.
2. Define initial state: assert F^0 for every fluent F in initial state and $\neg F^0$ for every fluent not in initial state. [1]
3. Propositionalise the goal: each goal is a conjunction. Therefore, the propositionalised goal should be a disjunction over all possible ground conjunctions obtained by replacing variables with constants.
4. Add **successor-state** axioms. For each fluent F, add axiom of the form

$$F^{t+1} \Leftrightarrow \text{ActionCauses}F^t \vee (F^t \wedge \neg\text{ActionCausesNot}F^t)$$

[1]Superscript denotes the time.

where ActionCasuesF^t is a disjunction of all ground actions that have F in Add list, and ActionCausesNotF^t is a disjunction of all ground actions that have F in Del list. Note, this is a better substitutes to frame axioms, which looks like $A \Rightarrow (F^t \Leftrightarrow F^{t+1})$, for all A and F, as it will have fewer number of total axioms.

5. Add **preconditon** axioms: for each ground action A, add axiom $A^t \Rightarrow \textsc{Pre}(A)^t$. Equivalently speaking, if an action is taken at time t, its preconditions must have been true at time t.

6. Add **action exclusion** axioms: every action is distinct from every other actions, i.e. only one action is allowed at each time step. It is in the form $\neg A_i^t \vee \neg A_j^t$ for all pairs of i, j.
If we want to allow parallel action, add mutual exclusion only if the pair of action really interfere with each other

We have defined the translation process. The general algorithm of SATPlan is defined below:

Theorem 1.4 (SATPlan).

```
function SATPLAN(init,transition,goal,$T_{\max}$) returns solution or failure
  inputs: init,transition,goal, constitute a description of the problem
  $T_{\max}$, an upper limit of plan length

  for $t = 0$ to $T_{\max}$ do
      cnf←TRANSLATE-TO-SAT(init,transition,goal,t)
      model←SAT-SOLVER(cnf)
      if model is not null then
          return EXTRACT-SOLUTION(model)
  return failure
```

2 Decision Theory

Definition 2.1 (Rationality of Agent).
A **rational** agent is the one that chooses the action that maximises its **expected utlity**. Specifically, we want to maximize the expected utility of action a given evidence e

$$EU(a \mid e) := \sum_{s'} P(\textsc{Result}(a) = s' \mid a, e) U(s')$$

where $P(\textsc{Result}(a) = s' \mid a, e)$ is the probability of outcome s' given that action a and evidence e, and $U(s)$ is **utility function** assigning a number to indicate desirability of state s.

2.1 Utility Theory

We take an axiomatic approach to define utility. We first adopt a few axioms that defines preferences between outcomes' behaviours, and utlities arise as the consequence of the preferences.
Here, we assume that agent has the following three types of preferences, namely

- $A \succ B$: agent prefers A over B
- $A \sim B$: agent is indifferent between A and B
- $A \succsim B$: agent prefers A over B or is indifferent

Here A, B may be either states of the world, which is deterministic, or **lottery**, whose outcome is not deterministic.

Definition 2.2 (Lottery)**.**
A **lottery** L with outcome $S_1, \dots, S_n$ that occurs with probabilities $p_1, \dots, p_n$ is denoted as

$$L = [p_1, S_1; p_2, S_2; \dots; p_n, S_n]$$

Theorem 2.1 (Axiom of Utility Theory)**.**
There are 6 axioms that constitutes the **axioms of utility theory**. They are

1. **Orderability**: Given any two lotteries, a rational agent must rate with one of three possibilities:
   (a) Prefer one: $A \succ B$
   (b) Prefer the other: $B \succ A$
   (c) Prefer two equally: $A \sim B$
2. **Transitivity**: If the agent prefers A to B and B to C, then the agent must prefer A to C
$$(A \succ B) \cap (B \succ C) \Rightarrow (A \succ C)$$
3. **Continuity**: If B is between A and C in preference, there must be a p, such that the agent is indifferent to getting A with probability p and C with probability $(1-p)$.
$$A \succ B \succ C \Rightarrow \exists p[p, A; 1-p, C] \sim B$$
4. **Substitutability**: If an agent is indifferent between two lotteries A and B, then the agent is indifferent to two complex lotteries that are the same, except B is substituted for A:
$$A \sim B \Rightarrow [p, A; 1-p, C] \sim [p, B; 1-p, C]$$
The above condition also holds if $\succ$ is substituted for $\sim$.
5. **Monotonicity**: Suppose two lotteries have the same two possible outcomes, A and B. If an agent prefers A to B, the agent must prefer the lottery that has higher probability for A, and vice versa.
$$A \succ B \Rightarrow (p > q) \Leftrightarrow [p, A; 1-p, B] \succ [q, A; 1-q, B]$$
6. **Decomposibility**: Compound lotteries can be reduced to simpler ones using the law of probability.
$$[p, A; 1-p[q, B; 1-q, C]] \sim [p, A; (1-p)q, B; (1-p)(1-q), C]$$

Theorem 2.2 (Existence of Utility Function)**.**
If agent's preferences obey the axioms of utlity, then there exists a function U s.t. $U(A) > U(B)$ iff A is preferable to B and $U(A) = U(B)$ iff agent is indifferent between A and B.

$$U(A) > U(B) \Leftrightarrow A \succ B$$

$$U(A) = U(B) \Leftrightarrow A \sim B$$

Theorem 2.3 (Expected Utility of a Lottery)**.**
The utility of a lottery is the expected value of the utility function

$$U([p_1, S_1; \ldots ; p_n, S_n]) = \sum_i p_i U(S_i)$$

With the above setup, we have the following result:

Theorem 2.4.
An agent can act rationally only by choosing an action that **maximizes the expected utility**. If not, it will violate some of the axioms of utility.

Also, do note that

- an agent's behaviour does not change if the utility function is transformed an affine transformation $U'(S) = aU(S) + b$ with $a > 0$.

- In a deterministic environment, a monotonic increasing transformation also does not change behaviour.
- In a deterministic environment, only the axiom of orderability and transitivity is required, as other axioms all involve lotteries.

2.2 Utility Function

Process of presenting choices to agents and working out the utility function from observed responses is often called **preference elicitation**.
Usually, for **normalised utilities**,we

1. Fix values $u_\perp = 0$ and $u_\top = 1$.
2. For any other prize S, we ask agent to choose between prize and a **standard lottery** $[p, u_\perp; 1 - p, u_\top]$
3. Adjust p until agent is indifferent between S and lottery, which is the utility of S.

However, by Arrow's Impossibility Theorem, utilities of a group cannot be fairly determined.

2.2.1 Utility of Money

An agent ususally prefers money more than less; we say the preference of money is **monotonic preference**.
However, most people are risk averse($U'' < 0$), on positive part of curve; and risk seeking on large negative wealth($U'' > 0$).
To have a clearer representation of the value of a lottery in the eye of some agent, We can measure the value an agent will accept in lieu, which is defined as the **certainty equivalent** of the lottery.
The difference between Expected Monetary Value and its certainty equivalent is the **insurance premium**.
Since utility theory is a normative theory, there can be reallife scenarios where people act irrationally, as people fall into

- **Certainty effect**, where they are strongly attracted to gains that are certain, and choose not to gamble due to possible regrets
- **Ambiguity aversion**, where they prefer known probability to unknown ones
- **Framing Effect**, where exact wording of choices have big impacts on choices
- **Anchoring effect**, where they prefer to make relative utility judgements to absolute ones

2.3 Multiattribute Utility Function

When outcomes are characterised by two or more attributes, we employ multiattribute utility theorey, where we have

- Attribute $\boldsymbol{X} = X1, \ldots, X_n$
- Assignment $\boldsymbol{x} = \langle x_1, \ldots, x_n \rangle$.
- with ordering of values for x_i, and higher value of attribute implying higher utilityu

Definition 2.3 (Strict Dominance)**.**
Option A strictly dominates B if B has lower value on all attributes than A's. If A and B are uncertain, A strictly dominates A if any value of A can strictly dominates all possible values of B.

Definition 2.4 (Stochastic Dominance)**.**
A_1 is stochastically dominating A_2 on X if

$$\forall x, \int_{-\infty}^{x} p_1(x') \,\mathrm{d}\, x' \leq \int_{-\infty}^{x} p_2(x') \,\mathrm{d}\, x'$$

Definition 2.5 (Preference Independent)**.**
An attribute X_1 is **preference independent**(PI) of X_2 if

$$\langle x_1, x_2^0 \rangle \succ \langle x_1', x_2^0 \rangle \Rightarrow \langle x_1, x_2 \rangle \succ \langle x_1', x_2 \rangle \text{ for all } x_2$$

In other words, preference of x_1 does not depend on value of x_2.

We say atrributes A are mutually preference indepdendent(MPI) if any subset $X \subseteq A$ is preference independent of its complement $Y = A \setminus X$.
If A is MPI, there exists an **additive preference function** where there is no uncertainty

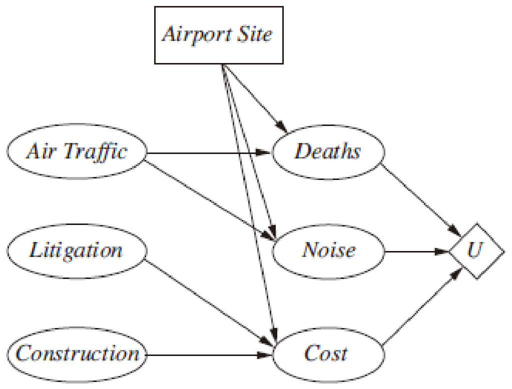
$$V(x_1, \ldots, x_n) = \sum_{i=1}^{n} V_i(x_i)$$

Definition 2.6 (Utility Independent)**.**
We say a set of attributes $\boldsymbol{X}$ is utility independent $\boldsymbol{Y}$ if preferences between lotteries on $\boldsymbol{X}$ are independent of values of $\boldsymbol{Y}$.
A set of attributes is mutually utility independent if each of its subset is utility independent of the remainder.

MUI implies a multiplicative utility function $U = k_1U_1 + k_2U_2 + k_3U_3 + k_1k_2U_1U_2 + k_2k_3U_2U_3 + k_3k_1U_3U_1 + k_1k_2k_3U_1U_2U_3$ for a set of 3 attributes.

2.4 Decision Network

Definition 2.7 (Decision Network)**.**
Decision netowrk combine Bayesian network with additional nodes to denote actions and utilities.



- **Chance nodes**, in oval, represent random variables. It should have a conditional distribution given its parent.
- **Decision Nodes**, in rectangle, represents points where dicision maker has a choice of actions.
- **Utility nodes**, in diamonds, represent the utility function. Its parent are all the variables that directly affect the utility.
  Utility node is associated with **action-utility function**, representing expected utility associated with each action.

In choosing an action, we

- Set evidence variable to the *observed* states
- For each possible value of the decision node
  1. Set decision node to that value
  2. Calculate posterior probability of parents of utility node
  3. Calculate resulting utility of this action
- Return action with highest utility

2.5 Value of Information

Assume exact evidence can be obtained about variable E_j, we can compute the **value of perfect information**(VPI), in three steps:

- Given current evidence $\mathbf{e}$, compute expected utility with current best action α

$$EU(\alpha \mid \mathbf{e}) = \max_a \sum_{s'} P(\text{Result}(a) = s' \mid a, \mathbf{e})U(s')$$

- Value of best new action after $E_j = e_j$ is obtained

$$EU(\alpha_{e_j} \mid \mathbf{e}, e_j) = \max_a \sum_{s'} P(\text{Result}(a) = s' \mid a, \mathbf{e}, e_j)U(s')$$

- Variable E_j can take multiple values e_{jk}, so must average:

$$VPI_{\mathbf{e}}(E_j) = (\sum_k P(E_j = e_{jk} \mid \mathbf{e})EU(\alpha_{e_{jk}} \mid \mathbf{e}, e_{jk})) - EU(\alpha \mid \mathbf{e})$$

We have the following properties on VPI:

- Expected value of information is non-negative:

$$\forall \mathbf{e}, E_j \quad VPI_{\mathbf{e}}(E_j) \geq 0$$

- VPI is not additive

$$VPI_{\mathbf{e}}(E_j, E_k) \neq VPI_{\mathbf{e}}(E_j) + VPI_{\mathbf{e}}(E_k)$$

- VPI is order independent

$$VPI_{\mathbf{e}}(E_j, E_k) = VPI_{\mathbf{e}}(E_j) + VPI_{\mathbf{e},e_j}(E_k) = VPI_{\mathbf{e}}(E_k) + VPI_{\mathbf{e},e_k}(E_j)$$

Generally, as expected value of information is non-negative, agents should gather information before taking actions, if possible.

3 Markov Decision Process

Definition 3.1 (Markov Decision Process)**.**
A markov decision process consists of a tuple (S, A, T, R) where

- S, a set of states, with initial state s_0 specified
- A, a set of actions
- T, a transition model, defined by $P(s' \mid s, a)$.
- R a reward function, in $R(s, a, s')$.

A transition model is a model that describes the probability of reaching state s' if action a is done in state s, i.e. $P(s' \mid s, a)$. **Markovian** assumption is applied here: probability depends only on state s but not history of states earlier.
A reward function is a utility function that depends on *sequence of* states. More details will be provided later.

Definition 3.2 (Policy)**.**
A **policy** $\pi(s)$ is a function from state to action.
Quality of policy is measured by **expected utility of possible environment histories generated by the policy**. Therefore, **optimal policy** π^* is a policy that generates highest expected utility.

Definition 3.3 (Finite Horizon, Infinite Horizon)**.**
In a **finite horizon** problem, there is a fixed time N, after which nothing matters. Therefore, optimal action at a given state can change over time, i.e., optimal policy is **nonstationary**.
In an **infinite horizon** problem, there is no fixed time limit; optimal action depends only on current state, and therefore **stationary**.

There are only two coherent ways to assign utilities:

1. Additive Rewards:
$$U_h([s_0, s_1, \ldots,]) = R(s_0) + R(s_1) + \cdots$$
2. Discounted rewards:
$$U_h([s_0, s_1, \ldots,]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots$$
where γ is a discount factor between 0 and 1.

Suppose we have maximum reward bounded by $\pm R_{\max}$, then the discounted rewards will be bounded by $\frac{R_{\max}}{1-\gamma}$ whereas additive rewards will be unbounded, if the problem has infinite horizon. Therefore, discounted rewards is recommended.

Definition 3.4 (Proper Policy)**.**
In environment with terminal states and policies that are guaranteed to terminate, we can use $\gamma = 1$. A policy that is guaranteed to terminate is called **proper policy**.

Also, infinite sequences can be compared using average reward per unit time.

Definition 3.5 (Expected Utility under discounted rewards)**.**
Expected utility of executing π starting from s is

$$U^{\pi}(s) = E[\sum_{t=0}^{\infty} \gamma^t R(S_t)]$$

with $S_0 = s$, where the expectation is taken with respect to distribution of state sequ3ences determined by π and s.

Therefore, optimal policy π_s^* is given by $\pi_s^* = \arg\max_{\pi} U^{\pi}(s)$.
In case of there are n states and a actions, the number of policies are a^n which is bounded. Therefore, the optimal policy can always be determined.

Definition 3.6 (Value Function)**.**
The utility of a state s under optimal policy $U^{\pi^*}(s)$ is called the **value function**.

Given the value function, the optimal policy can be determined by

$$\pi^*(s) = \arg\max_{a \in A(s)} \sum_{s'} P(s' \mid a, s) U(s')$$

The problem is that U requires information of $\pi^*(s)$ for all s. Therefore, we need techinques to get U.

3.1 Value Iteration

Definition 3.7 (Bellman Equation)**.** *The Bellman equation states, the utility of a state is its immediate reward plus expected utility of next states, given optimal action*

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U(s')$$

Theorem 3.1 (Value Iteration Algorithm)**.**
Value Iteration Algorithm returns a utility function that is ϵ close to the actual utility function.

```
function VALUE-ITERATION($mdp$,$\varepsilon$) returns a utility function
  inputs: $mdp$:=$(S, A, P, R)$ and discount $\gamma$.
    reward $\varepsilon$, the maximum error allowed in utility of any state
  local variables: $U, U'$ vectors of utilities for state in $S$, initially zero
    $\delta$, the maximum change in the utility of any states in an iteration
  repeat
    $U \leftarrow U'$, $\delta \leftarrow 0$
    for each state $s$ in $S$ do
      $U'[s] \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid a, s) U[s']$
      if $|U'[s] - U[s]| > \delta$ then $\delta \leftarrow |U'[s] - U[s]|$.
  until $\delta < \varepsilon(1-\gamma)/\gamma$      return $U$.
```

Essentially, the above algorithm repeatedly apply **Bellman update**

$$U_{t+1}(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U_t(s')$$

Value iteration converges to the value function for discounted problems with $\gamma < 1$. This is because of the **contraction property**

$$\|BU - BU'\| \leq \gamma \|U - U'\|$$

where $\|U\| = \max_s U(s)$ is the max norm.
Repeatedly applying contraction, we have $\|BU_t - U\| \leq \gamma^t \|U_0 - U\|$ for any initial U_0.
If we initialize U_0 to be 0, then $\|U_0 - U\| \leq R_{\max}/(1-\gamma)$, and we need

$$N = \lceil \log(R_{\max}/\varepsilon(1-\gamma)/\log(1/\gamma)) \rceil$$

iterations to get error at most ε.

3.2 Policy Iteration

In the case that utility function do not need to be highly accurate to give correct policy, policy iteration can take advantage.

Definition 3.8 (Policy Iteration)**.**
For **policy iteration**, begin with some initial policy π_0, alternate the following two steps:

- **Policy evaluation**: Given a policy π_i, calculate $U_i = U^{\pi_i}$, the value if π_i is executed, over all the states.
- **Policy improvement**: calculate a new policy π_{i+1} using one step look-ahead based on U_i over all the states.

Terminate when there is no change in policy. Since there is a finite number of policies, it is guaranteed that policy iteration can terminate.

```
function POLICY-ITERATION(mdp) returns a policy
  inputs: mdp, an MDP with state $S$, action $A(s)$ and transition model $P(s' \mid s, a)$.
  local variables: $U$, a vector of utilities for states in $S$, initially 0.
                   $\pi$, a policy vector indexed by states, initially random
  repeat
    $U \leftarrow$ POLICY-EVALUATION($\pi$, $U$, mdp)
    unchanged? $\leftarrow$ true
    for each state $s$ in $S$ do
      if $\max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U[s'] > \sum_{s'} P(s' \mid s, \pi[s]) U[s']$ then do
        $\pi[s] \leftarrow \arg\max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U[s']$
        unchanged? $\leftarrow$ false
  until unchanged?
```

return π.

In the above algorithm, the policy evaluation equations are similar but simpler than Bellman euations. For a state s in iteration i,

$$U_i(s) = R(s) + \gamma \sum_{s'} P(s' \mid s, \pi_i(s))U_i(s')$$

Here, we do not need the max operator.
If we use policy evaluation equations to do iterations, we have

$$U_{t+1}(s) \leftarrow R(s) + \gamma \sum_{s'} P(s' \mid s, a)U_t(s')$$

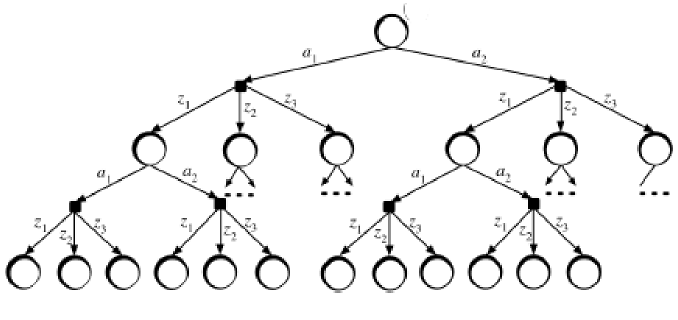
If we do a fixed number, say k iterations before stopping, instead of stopping when convergent, then such algorithm is called **modified policy iteration**.

For n states, the policy evaluation equation can be solved in $O(n^3)$ time. If we speed up by only picking a subset of states to do policy improvement, or for updating in policy evaluation, then it is called as **asynchronous policy iteration**.

3.3 Online Search

The state space grows exponentially with number of variables, and therefore we have curse of dimensionality, when we perform value/policy iteration, since they could possibly iterates through all states. To solve this problem, we may need to use function approximation for the value function, or we do online search with sampling.

Definition 3.9 (Online Search).
The complete online search requires to build a search tree up to a fixed depth D, with root as the current state. Suppose there are $|A|$ actions to take, therefore, root will have $|A|$ children. For each action, there will be a chance/observation node, for each of which there are $|S|$ children. To compute value at the root:

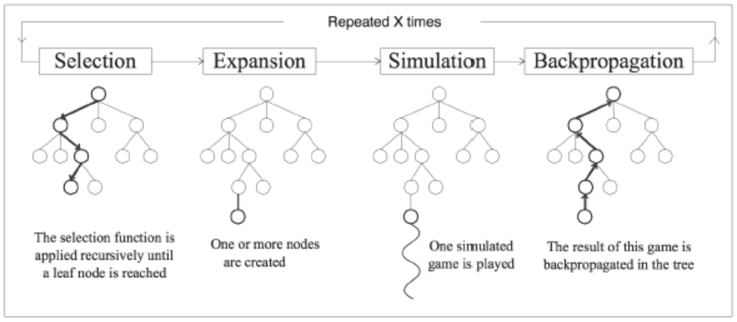


- Initialize leaf with value estimates (or zeroes)
- At observation nodes, compute the expected values of the children

- At action nodes, compute the max of the children

The tree size is $|A|^D|S|^D$. If we use sparse sampling, i.e. we estimate a observation node by only sampling k observations, instead of using all $|S|$ possible observation states, the tree size will be reduced to $|A|^D k^D$. This solves the exponential growth wrt to the number states, but its growth rate is still exponential to the search depth.

Definition 3.10 (Monte Carlo Tree Search).
Monte Carlo Tree Search consists of 4 stages, namely selection, expansion, simulation and backpropagation. MCTS repeatedly run trials from the root, where a trial


Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until a leaf node is reached
One or more nodes are created
One simulated game is played
The result of this game is backpropagated in the tree

- Repeatedly selet node to go to at next level, until
  - target depth reached, or
  - selected node has not been discovered: then create a new one node, and run a simulation using a default policy till required depth
- Back up the outcomes all the way to the root

This is an anytime policy: when time is up, it uses the action that looks best at the root at that time.

For MDP, a tree (actually DAG) node n is associated with a state s. A node n' at the next level is selected by applying an action a to s, then sampling the next state s' according to $p(s' \mid s, a)$. Here, this action a should be selected by balancing exploration with exploitation. The estimated value $\hat{V}(n)$ at a node n is the avergage reutn of all the trials at n. The return $r_t(n)$ of trial t starting from n with state s and next node n' is $R(s) + \gamma r_t(n')$.
The estimated Q-function(action value function) at n, $\hat{Q}(n, a)$ is the average return of all trials at n that starts with action a. Therefore, $\hat{Q}(r, a)$ is used to select the action to take at the root.
All these functions are updated in the back up operation to the root.

Theorem 3.2 (Upper Confidence Tree).
UCT uses the following function to select action to go to at node n:

$$\pi_{UCT}(n) = \arg\max_a (\hat{Q}(n, a) + c\sqrt{\frac{\ln(N(n))}{N(n, a)}}$$

where $N(n)$ is the number of trials through n and $N(n, a)$ is the number of trials through n and a. c is a constant, tuned to do well on the problem.
It is guaranteed that UCT will converge to optimal policy with enough trials($\Omega(\underbrace{\exp(\exp(\cdots exp(}_{D-1}1))))))$.

3.4 POMDP

For Partially Observable Markov Decision Process(POMDP), we no longer observe the state but receive some sensor information that can be used for state estimation.
A POMDP has the same elements as an MDP, state S, actions A, transitions T and reward R.
In addition, it has observation or sensor model defined by $P(e \mid s)$, the probability of perceiving evidence e given s.
Since we do not know actual state of the system, we need to track the **probability distribution** over possible states. We call this **belief state**, or belief. Belief is updated via the following equation:

$$b'(s') = \alpha P(e \mid s') \sum_{s} P(s' \mid s, a) b(s)$$

where $b(s)$ is the current belief, agent does action a, and receives evidence e, and α is the normalising constant so that b' sums to 1. This function is written as $b' = \textsc{Forward}(b, a, e)$.
Note, the belief contains all the information necessary for the agent to act optimally; the optimal action depends only on the agent's current belief. Optimal policy can be described as a mapping $\pi^*(b)$ from belief to action.
The POMDP agent acts as follows:

- Given current belief b, execute action $a = \pi^*(b)$.
- Receive the observation e
- Set the belief to $\textsc{Forward}(b, a, e)$ and repeat.

A POMDP can be viewed as a *belief space* MDP:

- Define reward function in belief space can be defined as

$$\rho(b) = \sum_{s} b(s) R(s)$$

- Can derive $P(b' \mid b, a)$ from underlying POMDP.
- Taken together $P(b' \mid b, a)$ and $\rho(b)$ defines an observable MDP in belief space.
- Optimal policy for this MDP, $\pi^*(b)$ is also an optimal policy for the POMDP.

However, the belief space is continuous, so we have a continuous MDP, and this results that value and policy iteration algorithms described in MDP cannot be used any more.

Theorem 3.3 (Value Iterations for POMDPs)**.**
A policy at belief b_0 is a conditional plan. Multiple conditional plans are possible. We will have an utility $\alpha_p(s)$ by executing a fixed conditional plan p from a state s. Therefore, executing from belief b will have expected utility $\sum_s b(s)\alpha_p(s)$.
The potimal policy at a particular belief b, is to chooser the conditional plan with highest utility

$$U(b) = \max_p b \cdot \alpha_p$$

For finite depth, there is only a finite number of conditional plans. With $|A|$ actions and $|E|$ observations, there are $|A|^{O(|E|)^{d-1}}$ distinct depth-d plans.

The continuous belief space is divided into regions, each corresponding to a conditional plan optimal for that region. Here, $U(b)$ is piecewise linear and convex.
The value iteration formula is

$$\alpha_p(s) = R(s) + \gamma(\sum_{s'} P(s' \mid s, a) \sum_e P(e \mid s')\alpha_{p,e}(s'))$$

This gives rise tothe value iteration algorithm:

Theorem 3.4 (Value Iteration Algorithm)**.**

```
function POMDP-VALUE-ITERATION($pomdp$, $\varepsilon$) returns a utility function
    inputs: $pomdp$, a POMDP with state $S$, actions $A(s)$ and transition model $P(s' \mid s, a)$.
            sensor model $P(e \mid s)$, reward $R(s)$ and discount $\gamma$.
            $\varepsilon$, the maximum error allowed in the utility of any state
    local variables: $U, U'$ sets of plans $p$ with associated utility vectors $\alpha_p$

    repeat
        $U \leftarrow U'$
        $U' \leftarrow$ the set of all plans consisting of an action and, for each possible next percept, a
plan in $U$ with utility vector computed according to equation above
        $U' \leftarrow$REMOVE-DOMINATED-PLANS($U'$)
    until MAX-DIFFERENCE($U, U'$) $< \varepsilon(1-\gamma)/\gamma$
    return $U$
```

Do note that exact POMDP solvers only work for very space problems. Therefore, online search tend to scale better.
Essentially, the execution of POMDP over time can be represented as a **dynamic decision network**(DDN). POMDP solvers need to solve two problems:

- Belief tracking
- Planning problem

Definition 3.11 (POMCP)**.**
Essentially, POMCP is to run UCT on POMDP. To do this, we need to represent beliefs in the nodes.

In POMCP, instead of propagating belief forward in a trial, POMCP samples a state at the root from initiall belief, then runs simulation using the state to generate action-observation history. Therefore, it only needs to sample from $p(s' \mid s, a)$ then $p(e \mid s')$ to generate observation e for the action-observation history.

Another algorithm is DESPOT.

4 Reinforcement Learning

Consider the game where an agent can learn the transition model for its own move and possibly to predict opponent move; but the agent often do not get feedback on whether it has done weel until the end of the game.
The feedback on whether something good or bad happens is called a **reward** or **reinforcement**.
We regard reward as part of the input perception, and the agent must be able to recognize it as reward rather than another input.
In this section, we assume environment is Markov Decision Process. An optimal policy is a policy that maximizes expected total reward.
Reinforcement learning aims to use observed rewards to learn an optimal policy for the environment. Here, we assume agent has no knowledge of the model for environement or reward function.

4.1 Passive Reinforcement Learning

In passive learning, policy π is fixed, so agent always execute $\pi(s)$ in state s. The goal is to learn

- Value function $U^{\pi}(s)$ from observations, where
- **transition model** $P(s' \mid s, a)$ and **reward function** $R(s)$ are unknown.

In fact, if transition and reward functions are known, we can just perform policy evaluation.
In passive reinforcement learning, the agent executes a set of **trials** using π, which can be traced as a sequence of (state, reward) pairs.
The utility, or value of π at a state s is $U^{\pi}(s) = E[\sum_{t=0}^{\infty} \gamma^t R(S_t)]$.

Definition 4.1 (Adaptive Dynamic Programming).
Adaptive Dynamic Programming learns the model, then solves it:

- Learn transition probabilities $P(s' \mid s, a)$;
- Learn reward function $R(s)$
- Calculate value $U^{\pi}(s)$ by solving the Bellman equations.

Agent can use **modified policy iteration** method of doing k iterations of value updates after each change to model.
The algorithm is outlined as below: **function** PASSIVE-ADP-AGENT(*percept*) **returns** an action

```
  inputs: percept, a percept indicating the current state $s'$ and reward signal $r'$
  persistent: $\pi$, a fixed policy
        mdp, an MDP with model $P$, reward $R$, discount $\gamma$
        $U$, a table of utilities , initially empty
        $N_{sa}$, a table of frequencies of state-action pairs, initially zero
        $N_{s'|sa}$, a table of outcome frequencies given state action pairs, initially zero
        $s, a$, the previous state and action, initially null
  **if** $s'$ is new **then** $U[s'] \leftarrow r'$, $R[s'] \leftarrow r'$
  **if** $s$ is not null **then**
    increment $N_{sa}[s,a]$ and $N_{s'|s,a}[s',s,a]$
    **for each** $t$ such that $N_{s'|sa}[t,s,a]$ is nonzero **do**
      $P(t \mid s,a) \leftarrow N_{s'|s,a}[t,s,a]/N_{sa}[s,a]$
  $U \leftarrow$ POLICY-EVALUATION($\pi, U, mdp$)
  **if** $s'$.TERMINAL? **then** $s, a \leftarrow$ null **else** $s, a \leftarrow s', \pi[s']$.
  **return** $a$
```

In a state, the utility or value is the **expected total reawrd** from that state onwards, also called expected reward-to-go or **return**.
In direct utility estimation, often called **Monte Carlo learning**, we treat each trial as providing a sample of this quantit for each state visited.
So we just keep a running average for each state in a table, and in infinitely many trials, sample average will converge to expected value. Esssentially, utility derived from k returns $G_1(s), \ldots, G_k(s)$ is

$$U_k(s) = \frac{1}{k}\sum_{i=1}^{k} G_i(s)$$

We can rearrange the above equation to be

$$U_k(s) = U_{k-1}(s) + \frac{1}{k}(G_k(s) - U_{k-1}(s))$$

where $U_{k-1}(s)$ is the estimate after receiving $k-1$ returns, and we call $G_k(s) - U_{k-1}(s)$ the prediction error for the kth return.
It is clear that Monte Carlo learning is an instance of **supervised learning**, where each example has state as input and observed return as output.
Its advantages are: simple, and each labelled target(return) is an unbiased estimate.
The disadvantages include

- Need to wait till the end of episode before learning can be done
- Variance can be high as a return is a sum of many rewards over the sequence. The varaince can be controlled if we use the discount factor when calculating reward of trials.

4.2 Temporal Difference Learning

Temporal difference learning exploits more of the Bellman equation constraints.
For a transition from state s to s'. TD learning does

$$U^\pi(s) \leftarrow U^\pi(s) + \alpha(R(s) + \gamma U^\pi(s') - U^\pi(s))$$

where α is the **learning rate**.
Here,

- $U^\pi(s)$ increases if $R(s) + \gamma U^\pi(s')$ is larger than $U^\pi(s)$ and decreases it otherwise.
- Therefore, we call $R(s) + \gamma U^\pi(s')$ the TD target and $(R(s) + \gamma U^\pi(s') - U^\pi(s))$ the TD error.
- It converges to expected value if α decreases with number of times the state has been visited. For example, $\alpha(n) = O(1/n)$.

The exact algorithm is below:

```
function PASSIVE-TD-AGENT(percept) returns an action
    inputs: percept, a percept indicating the current state $s'$ and reward signal $r'$.
    persistent: $\pi$, a fixed policy
            $U$, a table of utilities, initially empty
            $N_s$ a table of frequencies of states, initially zerop
            $s, a, r$, the previous state, action, reward, initially null

    if $s'$ is new then $U[s'] \leftarrow r'$
    if $s$ not null then
        increment $N_s[s]$
        $U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma U[s'] - U[s])$
    if $s'$.TERMINAL? then $s, a, r \leftarrow$null else $s, a, r \leftarrow s', \pi[s'], r'$
    return $a$
```

The good thing about Temporal Difference is that

- TD can learn online after each step. MC needs complete episode before learning.
- TD target $R(s) + \gamma U^\pi(s')$ depends only on one measured reward. However, MC target $G(s)$ depends on sum of many rewards. As a result,
  - TD target lower variance, but biased
  - MC target unbiased, but higher variance
- TD usually converges faster than MC in practice

Above is 1-step TD, where we perform one-step look ahead.

Definition 4.2 (n-step TD)**.**
Let $G_{t:t+n} := R_t + \gamma R_{t+1} + \cdots + \gamma^{n-1} R_{t+n-1} + \gamma^n \hat{v}(S_{t+n})$ be the n-step return where $\hat{v}(S_{t+n})$ is the estimated value at state S_{t+n}.
In n-step TD, it sets $G_{t:t+n}$ as the target for update.
When n approaches ∞, we get Monte Carlo learning. Therefore, intermediate values of n for n-step TD tend to work better.

Definition 4.3 (TD(λ))**.**
Instead of using n-step return , one can average over different values of n.
TD(λ) sets the following as the target for update

$$G_t^\lambda = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n}$$

It can be computed efficiently using method called *eligibility traces*.
It converges to TD as $\lambda \to 0$ and converges to MC as $\lambda \to 1$.

The difference between adaptive dynamic programming against TD is

- ADP learns model then solves for value,
- whereas TD, MC does not need a model.
- ADP tend to be more data efficient, however
- TD does not need to compute expectation and does not need to solve system of linear equations, which in turn is computationally more efficient

4.3 Active Reinforcement Learning

Greedy ADP is a greedy agent, but it may not converge at the optimal policy. Therefore, we want to make actions in reinforcement learning not only gain reward but also help to learn a better model. As a result, we need to trade off between

- Exploitation: maximize value as reflected by current estimate
- Exploration: learn more about the model to potentially improve long term well being

One simple method is called **ϵ-greedy exploration**, where it choose a greedy action with probability $1 - \epsilon$ and a random action with probability ϵ.
A scheme for balancing exploration and exploitation must

- Try each action in each state an unbounded number of times, so as to avoid a finite probability of missing an optimal action
- Eventually become greedy so that it is optimal with respect to the true model

Such schemes are greedy in the limit of infinite exploration(GLIE). One simple GLIE scheme is to ϵ-greedy exploration with choice of $\epsilon = \frac{1}{t}$.
However, the shortcoming is, while GLIE based ϵ-greedy eventually converges, it can be slow.
An alternative way is to use greedy action selection with respect to an **optimistic** estimate of the utility $U^+(s)$.
One example of U^+ is to use an exploration function $f(u, n)$ with the following update:

$$U^+(s) \leftarrow R(s) + \gamma \max_a f\left(\sum_{s'} P(s' \mid s, a) U^+(s'), N(s, a)\right)$$

where $N(s, a)$ is the number of times action a has been tried in state s and exploration function $f(u, n)$ *increasing* with u and decreasing with n. An example of $f(u, n)$ can be

$$f(u, n) = \begin{cases} R^+ & \text{if } n < N_e, \text{ a threshold} \\ u & \text{otherwise} \end{cases}$$

where R^+ is an optimistic estimate of best possible reward.

4.3.1 Model Free Learning

Instead of laerning the utility function, we can learn an action-utility function $Q(s, a)$, the value of doing action a in state s. Q-values are related to utility values by

$$U(s) = \max_a Q(s, a)$$

and the optimal action can be derived from arg max. Since an agent that learns a Q-function does not need a model of the form $P(s' \mid s, a)$ for action selection, and thus is called a **model-free** method.
The Q-function similarly satisfies a version of the Bellman equations

$$Q(s, a) = R(s) + \gamma \sum_{s'} P(s' \mid s, a) \max_{a'} Q(s', a')$$

If we use ADP to learn, we can learn $P(s' \mid s, a)$ and then use an iterative method to compute the Q-function, given the estimated model.
Remark: ADP with Q-function still need model for learning, but not to take action. We can discard P if we use MC or TD.

Theorem 4.1 (GLIE ϵ-greedy MC Control)**.**
We sample episode k using current Q-function with ϵ-greedy exploration with $\epsilon = \frac{1}{k}$L $S_1, A_1, R_2, \ldots, S_T$.
For each S_t, A_t in the episode,

$$\begin{aligned} N(S_t, A_t) &\leftarrow N(S_t, A_t) + 1 \\ Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \frac{1}{N(S_t, A_t)(G_t - Q(S_t, A_t))} \end{aligned}$$

Theorem 4.2 (Q-learning)**.**
Using the TD method instead, we can arrive at Q-learning:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

```
function Q-LEARNING-AGENT(percept) returns an action
  inputs: percept, a percept indicating the current state $s'$ and reward signal $r'$
  persistent: $Q$, a table of action values indexed by state and action, initially zero
              $N_{sa}$, a table of frequencies for state-action pairs, initially zero
              $s, a, r$, the previous state, action and reward, initially null

  if TERMINAL?($s$) then $Q[s, None] \leftarrow r'$
  if $s$ is not null then
    increment $N_{sa}[s, a]$
    $Q[s, a] \leftarrow Q[s, a] + \alpha(N_{sa}[s, a])(r + \gamma \max_{a'} Q[s', a'] - Q[s, a])$
  $s, a, r \leftarrow s', \arg\max_{a'} f(Q[s', a'], N_{sa}[s', a']), r'$
  return $a$
```

Theorem 4.3 (SARSA(State-Action-Reward-State-Action))**.**
SARSA is a slight modification from Q-learning with the update rule being

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma Q(s',a') - Q(s,a))$$

where a' is the action actually taken.

The advantage of SARSA is **on-policy**. If agent policy is always exploring, it will learn to take that into account. In contrast, Q-learning is **off-policy**, which works regardless of policy for generating trajectories. Therefore, random policy can be used as input too.
When greedy agent in SARSA takes action with best Q value, the two will be identical.

4.4 Function Approximation

The learning above all require some form of tabular. However, this is not possible if we have many state variables, with respect to which the number of states grows exponentially.
One common solution is to use **function approximation** to represent utility and Q-function. For example, we can represent evaluation function as a linear function of **features**:

$$\hat{U}_\theta(s) = \sum_{i=1}^{n} \theta_i f_i(s)$$

Here, the n parameters in θ is used in the function approximation, and the reinforcement learning agent now learns only θ.
For Monte Carlo learning, we need to obain a set of training samples $((x_i, y_i), u_i)$ where u_i is the measured utility of the jth example. This gives a **supervised learning problem** since it reduces to a linear regression problem.
To minimize the squared error, using online learning, we can, for the jth example, take a step in directino of gradient of

$$E_j(s) := \frac{1}{2}(\hat{U}_\theta(s) - u_j(s))^2$$

For parameter θ_i, we update via

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial E_j(s)}{\partial \theta_i} = \theta_i + \alpha(u_j(s) - \hat{U}_\theta(s)) \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}$$

The same online learning can be applied to temporal difference learning and Q-learning, with the equation adjusted:

- For TD:
$$\theta_i \leftarrow \theta_i + \alpha[R(s) + \gamma \hat{U}_\theta(s') - \hat{U}_\theta(s)] \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}$$
- For Q-learning,
$$\theta_i \leftarrow \theta_i + \alpha[R(s) + \gamma \max_{\alpha'} \hat{Q}_\theta(s',a') - \hat{Q}_\theta(s,a)] \frac{\partial \hat{Q}_\theta(s,a)}{\partial \theta_i}$$

This is called *semi*-gradient methods since target is not true value but also depends on θ.
For passive TD learning, update rule converges to best approximation of **linear function**, whereas for active and non-linear functions, there is no guarantee of convergence.

4.5 Policy Search

A policy π is a mapping from state to action. Assume that the policy is parameterized by some parameters θ. To be useful, the dimension of θ should be much smaller than number of states.
To derive the policy, we can use the Q-function:

$$\pi(s) = \arg\max_a \hat{Q}_\theta(s, a)$$

Policy search then just adjusts θ to improve the policy.
Remark: If Q^* is optimal Q-function, $\frac{Q^*}{10}$ also works optimally. Therefore, policy search is more flexible than Q-learning with functional approximation, as the latter tries to find a value of θ that $\hat{Q}_\theta$ is **close** to Q^*.
The problem is when actions are discrete, the policy will also be continuous as a function of θ, which makes gradient-based search difficult. Therefore, we offen use **stochastic policy** $\pi_\theta(s, a)$ that specifies the probability of selecting action a in state s. For example, the softmax function:

$$P(s, a \mid \pi_\theta) = \frac{e^{\hat{Q}_\theta(s,a)}}{\sum_{a'} e^{\hat{Q}_\theta(s,a')}}$$

If we can specify the policy value, denoted by $\rho(\theta)$, we can try optimizing for the value. One way is to take a step in the direction of the policy gradient vector $\nabla_\theta \rho(\theta)$ if $\rho(\theta)$ is differentiable. Then we update until we reach an local optimum.
In stochastic environment or policy $\pi_\theta(s, a)$, it is possible to obtain an unbiased estimate of gradient at θ,i.e. $\nabla_\theta \rho(\theta)$ directly from results of trials executed at θ.
In the case of single action a from single state s_0, the gradient is

$$\nabla_\theta \rho(\theta) = \nabla_\theta \sum_a \pi_\theta(s_0, a) R(a) = \sum_a (\nabla_\theta \pi_\theta(s, a)) R(a)$$

We then approximate the summation using samples generated from $\pi_\theta(s_0, a)$:

$$\nabla_\theta \rho(\theta) = \sum_a \pi_\theta(s_0, a) \frac{(\nabla_\theta \pi_\theta(s, a)) R(a)}{\pi_\theta(s_0, a)} \approx \frac{1}{N} \sum_{j=1}^{n} \frac{(\nabla_\theta \pi \theta(s_0, a_j)) R(a_j)}{\pi_\theta(s_0, a_j)}$$

For sequential case, this generalizes to

$$\nabla_\theta \rho(\theta) \propto \sum_s p_{\pi_\theta}(s) \sum_a \frac{\pi_\theta(s, a) \nabla_\theta \pi_\theta(s, a) Q_{\pi_\theta}(s, a)}{\pi_\theta(s, a)} \approx \frac{1}{N} \sum_{j=1}^{n} \frac{(\nabla_\theta \pi_\theta(s, a_j)) G_j(s)}{\pi_\theta(s, a_j)}$$

for each state s visited, where a_j is executed in s on jth trials and $G_j(s)$ is the total reward received from state s onwards on jth trial.
We can arrive at REINFORCE algorithm using an online update:

$$\theta_{j+1} = \theta_j + \alpha G_j \frac{\nabla_\theta \pi_\theta(s, a_j)}{\pi_\theta(s, a_j)} = \theta_j + \alpha G_j \nabla_\theta \ln \pi_\theta(s, a_j)$$

Should we want to reduce variance, we can do this transformation $Q_{\pi_\theta}(s, a) \to Q_{\pi_\theta}(s, a) - B(s)$.