Artificial Intelligence
DSE 3252

Introduction to AI

ROHINI R RAO & RASHMI MALGHAN
DEPT OF DATA SCIENCE & COMPUTER APPLICATIONS

JANUARY 2024

Thinking Humanly	**Thinking Rationally**
"The exciting new effort to make computers think . . . *machines with minds*, in the full and literal sense." (Haugeland, 1985) "[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning . . ." (Bellman, 1978)	"The study of mental faculties through the use of computational models." (Charniak and McDermott, 1985) "The study of the computations that make it possible to perceive, reason, and act." (Winston, 1992)
Acting Humanly	**Acting Rationally**
"The art of creating machines that perform functions that require intelligence when performed by people." (Kurzweil, 1990) "The study of how to make computers do things at which, at the moment, people are better." (Rich and Knight, 1991)	"Computational Intelligence is the study of the design of intelligent agents." (Poole *et al.*, 1998) "AI . . . is concerned with intelligent behavior in artifacts." (Nilsson, 1998)

Figure 1.1 Some definitions of artificial intelligence, organized into four categories.

Acting humanly: The Turing Test approach (1950)

The computer would need to possess the following capabilities:

- • natural language processing to enable it to communicate successfully in English
- • knowledge representation to store what it knows or hears
- • automated reasoning to use the stored information to answer questions and to draw new conclusions;
- • machine learning to adapt to new circumstances and to detect and extrapolate patterns

To pass the total Turing Test, the computer will need

- • computer vision to perceive objects, and
- • robotics to manipulate objects and move about.

Thinking humanly: The cognitive modeling approach

Determining how humans think through

- Introspection—trying to catch our own thoughts as they go by
- Psychological experiments—observing a person in action
- Brain imaging—observing the brain in action

From the theory of the mind it is possible to express theory as computer program

If the computer program's input–output behavior matches corresponding human behavior, that is evidence that some of the program's mechanisms could also be operating in humans

Cognitive science

- brings together computer models from AI and experimental techniques from psychology to construct precise and testable theories of the human mind

Thinking rationally: The “laws of thought” approach

Greek philosopher **Aristotle** was one of the first to attempt to codify “right thinking,” that

is, irrefutable reasoning processes.

Syllogisms

- patterns for argument structures that always yielded correct conclusions when given correct premises
- example - “Socrates is a man; all men are mortal; therefore, Socrates is mortal.”

These laws of thought were supposed to govern the operation of the mind

their study initiated the field called **LOGIC**

Main obstacles to approach

- it is not easy to take informal knowledge and state it in the formal terms required by logical notation, particularly when the knowledge is less than 100% certain
- there is a big difference between solving a problem “in principle” and solving it in practice

Acting rationally: The rational agent approach

An **Agent** is just something that acts (agent comes from the Latin agere, to do)

Computer agents

- operate autonomously
- perceive their environment
- persist over a prolonged time period
- adapt to change
- create and pursue goals

As opposed to Laws of Thought Approach

- making correct inferences is part of being a rational agent, need to reason logically to the conclusion that a given action will achieve one's goals and then to act on that conclusion
- However there may be a need to act rationally without careful deliberation

Advantages

- More general than "laws of thought"
- Mathematically well defined, so can be used in applications

Foundations of AI

Philosophy

- *Can formal rules be used to draw valid conclusions?*
- *How does the mind arise from a physical brain?*
- *Where does knowledge come from?*
- *How does knowledge lead to action?*

Dualism- Materialism- Naturalism- Utilitiarinism etc.

Principle of Induction – general rules are acquired by exposure to repeated associations between their elements

Mathematics

- *What are the formal rules to draw valid conclusions?*
- *What can be computed?*
- *How do we reason with uncertain information?*

Formal Logic – George Boole proposed propositional or Boolean logic

Probability – generalizing logic to situations with uncertain information

Ronald Fisher is first modern **statistician**

Euclid's **algorithm** for greatest common divisors

Alan Turing tried to characterize functions that are **computable**

Tractability – A problem is intractable if the time required to solve problems grows exponentially with the size of instances

NP-completeness – almost always intractable

Foundations of AI

Economics

- *How should we make decisions so as to maximize payoff?*
- *How should we do this when others may not go along?*
- *How should we do this when the payoff may be far in the future?*

Decision Theory combines probability with utility, provides a framework for individual decisions.
Game Theory – rational agent should adopt policies that are randomized
Operations Research
Markov Decision Process

Neuroscience

- *How do brains process information?*

Study of nervous system , particularly the brain
Measurement of intact brain activity began with invention of **Electroencephalograph** (EEG)
Optogenetics - Single cell electrical recording of neuron activity
Brain-Machine Interface – for both sensing and motor control to restore function to disabled individuals

Foundations of AI

Psychology

- *How do humans and animals think and act?*

Cognitive Psychology views the brain as an information-processing device
Perception involved a form of unconscious logical inference

Knowledge based agent has 3 steps

1. The stimulus must be translated to internal representation
2. The representation is manipulated by cognitive process to derive new internal representations
3. These are in turn retranslated back into action

Human Computer Interaction – idea of intelligent augmentation rather than AI.
Computers should augment human abilities rather than automate away human tasks

Foundations of AI

Computer engineering

• *How can we build an efficient computer?*

Moores law- each generation of computer hardware has brought an increase in speed and capacity and a decrease in price

Quantum Computing

Control theory and cybernetics

• *How can artifacts operate under their own control?*

Modern Control Theory

Viewed purposive behavior as arising from a regulatory mechanism trying to minimize error

Goal is to design systems that minimize cost function

Linguistics

• *How does language relate to thought?*

Computations Linguistics, NLP requires

- Understanding the structure of sentences
- understanding of the subject matter and context

Knowledge Representation is tied to language

Rational Agent

A **rational agent** strives to do the right thing, based on what it can perceive and the action it can perform

Performance measure: An objective criterion for success of agent's behavior

Example : Vacuum Cleaner Robot

Performance Criterion

- How must dust has to be collected?
- How much electricity can be consumed?
- How much time should it take?
- How much noise can be tolerated?

Ideal Rational Agent

For every possible percept sequence, it does whatever action is expected to maximize its performance measure on the basis of evidence perceived so far and built in knowledge

Good Behaviour : The concept of rationality

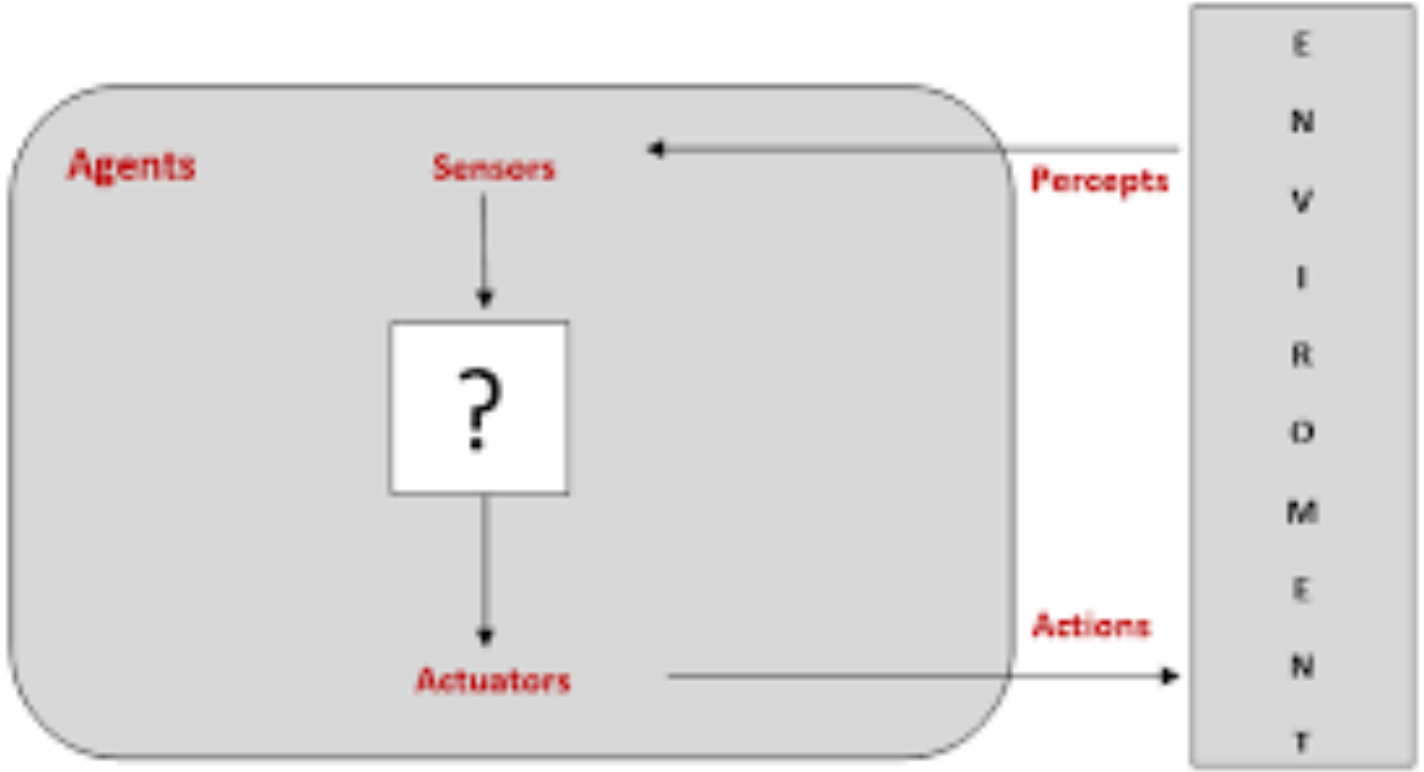
- **Doing the right thing** by considering the consequences of the agent's behavior
- This sequence of actions causes the environment to go through a sequence of states.
- If the sequence is desirable, then the agent has performed well
- This notion of desirability is captured by a performance measure that evaluates any given sequence of environment states

Rationality at any given time depends on :

- • The performance measure that defines the criterion of success.
- • The agent's prior knowledge of the environment.
- • The actions that the agent can perform.
- • The agent's percept sequence to date

As a general rule, it is better to design performance measures according to what one actually wants in the environment, rather than according to how one thinks the agent should behave

Agents interact with environments through sensors and actuators



Agent

- **Percept** to refer to the agent's perceptual inputs at any given instant
- An agent's **percept sequence** is the complete history of everything the agent has ever perceived
- In general, an agent's choice of action at any given instant can depend on the entire percept sequence observed to date but not on anything it hasn't perceived
- An agent's behavior is described by the **agent function** that maps any given percept sequence to an action
- Internally, the agent function for an artificial agent will be implemented by an **agent program**.

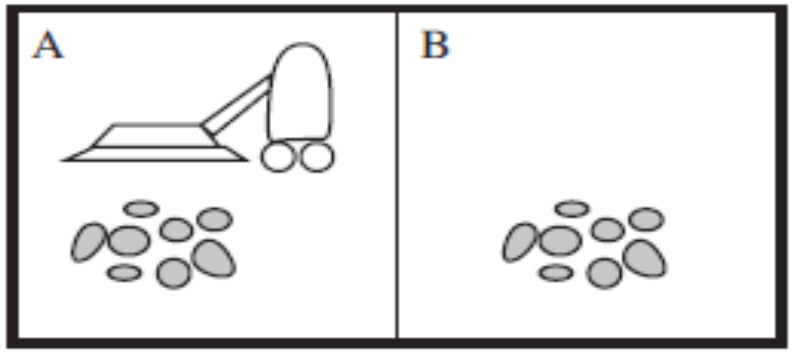


Figure 2.2 A vacuum-cleaner world with just two locations.

Percept sequence	Action
$[A, Clean]$	*Right*
$[A, Dirty]$	*Suck*
$[B, Clean]$	*Left*
$[B, Dirty]$	*Suck*
$[A, Clean], [A, Clean]$	*Right*
$[A, Clean], [A, Dirty]$	*Suck*
⋮	⋮
$[A, Clean], [A, Clean], [A, Clean]$	*Right*
$[A, Clean], [A, Clean], [A, Dirty]$	*Suck*
⋮	⋮

Figure 2.3 Partial tabulation of a simple agent function for the vacuum-cleaner world shown in Figure 2.2.

Specifying the task environment

PEAS

- Consider, e.g., the task of designing an automated taxi driver:

Agent Type	Performance Measure	Environment	Actuators	Sensors
Taxi driver	Safe, fast, legal, comfortable trip, maximize profits	Roads, other traffic, pedestrians, customers	Steering, accelerator, brake, signal, horn, display	Cameras, sonar, speedometer, GPS, odometer, accelerometer, engine sensors, keyboard

Figure 2.4 PEAS description of the task environment for an automated taxi.

Examples of agent types and their PEAS descriptions

Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, reduced costs	Patient, hospital, staff	Display of questions, tests, diagnoses, treatments, referrals	Keyboard entry of symptoms, findings, patient's answers
Satellite image analysis system	Correct image categorization	Downlink from orbiting satellite	Display of scene categorization	Color pixel arrays
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, joint angle sensors
Refinery controller	Purity, yield, safety	Refinery, operators	Valves, pumps, heaters, displays	Temperature, pressure, chemical sensors
Interactive English tutor	Student's score on test	Set of students, testing agency	Display of exercises, suggestions, corrections	Keyboard entry

Figure 2.5 Examples of agent types and their PEAS descriptions.

Properties of task environments

Fully observable vs. partially observable:

- If an agent's sensors give it access to the complete state of the environment at each point in time, then the task environment is fully observable.
- if the sensors detect all aspects that are relevant to the choice of action
- An environment might be partially observable because of noisy and inaccurate sensors or because parts of the state are simply missing from the sensor data
  - for example, a vacuum agent with only a local dirt sensor cannot tell whether there is dirt in other squares
- If the agent has no sensors at all then the environment is unobservable

Single agent vs. multiagent

- Example – Single agent - solving a crossword puzzle , Two agent – playing chess
- An entity becomes an agent when optimizing one agent's performance affects other agent
- Multiagent environment
  - Communication is part of rational behaviour
  - chess is a **competitive**
  - Taxi driving is **cooperative**

Properties of task environments

Deterministic vs. stochastic

- **Deterministic**- If the next state of the environment is completely determined by the current state and the action executed by the agent
- otherwise, it is **Stochastic -** uncertainty about outcomes is quantified in terms of probabilities
- Environment is uncertain if it is not fully observable or not deterministic
- **Nondeterministic** environment - actions are characterized by their possible outcomes, but no probabilities are attached to them

Episodic vs. sequential:

- In an **Episodic** task environment, the agent's experience is divided into atomic episodes.
- In each episode the agent receives a percept and then performs a single action.
- next episode does not depend on the actions taken in previous episodes
  - Example - an agent that is spotting defective parts on an assembly line
- In **sequential** environments, the current decision could affect all future decisions

Properties of Task environments

Static vs. dynamic:

- **Dynamic** If environment can change while an agent is deliberating otherwise, it is static
- Example – Taxi driving is dynamic, playing a puzzle is static

Discrete vs. continuous:

- applies to the state of the environment, to the way time is handled, and to the percepts and actions of the agent
- Example – Chess has finite number of discrete state
- **Taxi driving**
  - is a continuous-state and continuous-time problem: the speed and location of the taxi are a range of continuous values over time.
  - actions are also continuous (steering angles, etc.).

Known vs. unknown:

- agent's (or designer's) state of knowledge about the "laws of physics" of the environment
- In **Known** environment, the outcomes (or outcome probabilities if the environment is stochastic) for all actions are given

Hardest agent is partially observable, multiagent, stochastic, sequential, dynamic, continuous, and unknown

Examples of task environments and their characteristics

Task Environment	Observable	Agents	Deterministic	Episodic	Static	Discrete
Crossword puzzle	Fully	Single	Deterministic	Sequential	Static	Discrete
Chess with a clock	Fully	Multi	Deterministic	Sequential	Semi	Discrete
Poker	Partially	Multi	Stochastic	Sequential	Static	Discrete
Backgammon	Fully	Multi	Stochastic	Sequential	Static	Discrete
Taxi driving	Partially	Multi	Stochastic	Sequential	Dynamic	Continuous
Medical diagnosis	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Image analysis	Fully	Single	Deterministic	Episodic	Semi	Continuous
Part-picking robot	Partially	Single	Stochastic	Episodic	Dynamic	Continuous
Refinery controller	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Interactive English tutor	Partially	Multi	Stochastic	Sequential	Dynamic	Discrete

Figure 2.6 Examples of task environments and their characteristics.

Structure of Agents

- The job of AI is to design an agent program that implements the agent function
- program will run on some sort of computing device with physical sensors and actuators

agent = *architecture* + *program*

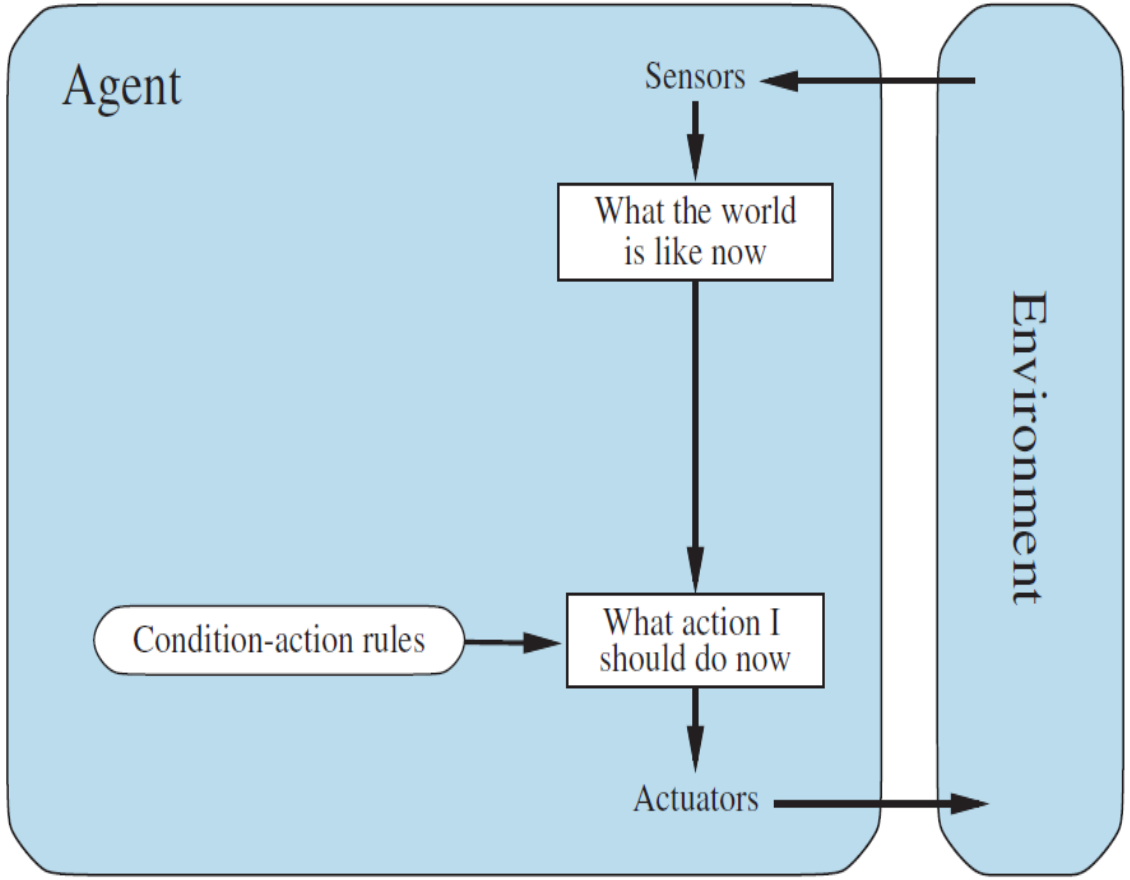
- Agent Programs take the current percept as input from the sensors and return an action to the actuators

```
function TABLE-DRIVEN-AGENT(percept) returns an action
  persistent: percepts, a sequence, initially empty
              table, a table of actions, indexed by percept sequences, initially fully specified

  append percept to the end of percepts
  action ← LOOKUP(percepts, table)
  return action
```

Figure 2.7 The TABLE-DRIVEN-AGENT program is invoked for each new percept and returns an action each time. It retains the complete percept sequence in memory.

P be the set of possible percepts
T be the lifetime of the agent (the total number of percepts it will receive)

Reflex Agents



function SIMPLE-REFLEX-AGENT(*percept*) **returns** an action
  persistent: *rules*, a set of condition–action rules

  state ← INTERPRET-INPUT(*percept*)
  rule ← RULE-MATCH(*state*, *rules*)
  action ← *rule*.ACTION
  return *action*

condition–action rule:
if *car-in-front-is-braking* **then** *initiate-braking*.

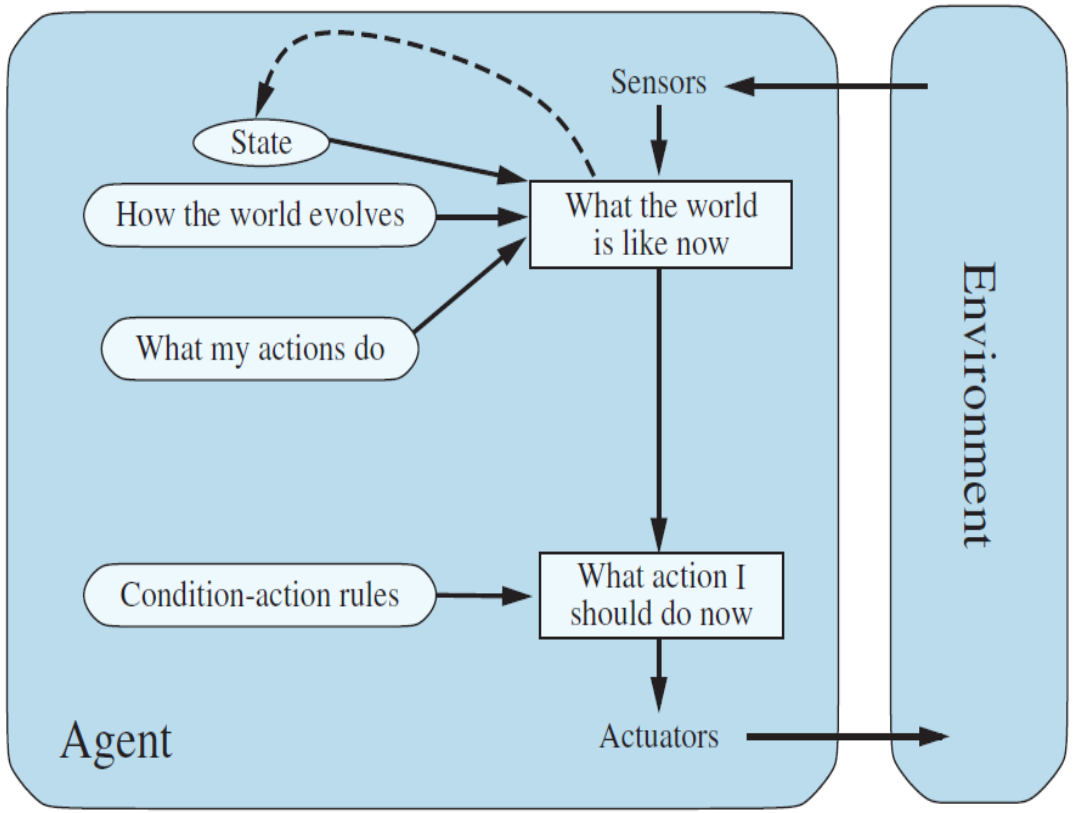
Structure of Agents

Simple reflex agents

- agents select actions on the basis of the current percept, ignoring the rest of the percept history
- Example – Vacuum Agent has location sensor and dirt sensor

```
function REFLEX-VACUUM-AGENT([location,status]) returns an action

    if status = Dirty then return Suck
    else if location = A then return Right
    else if location = B then return Left
```

Model based Reflex Agent

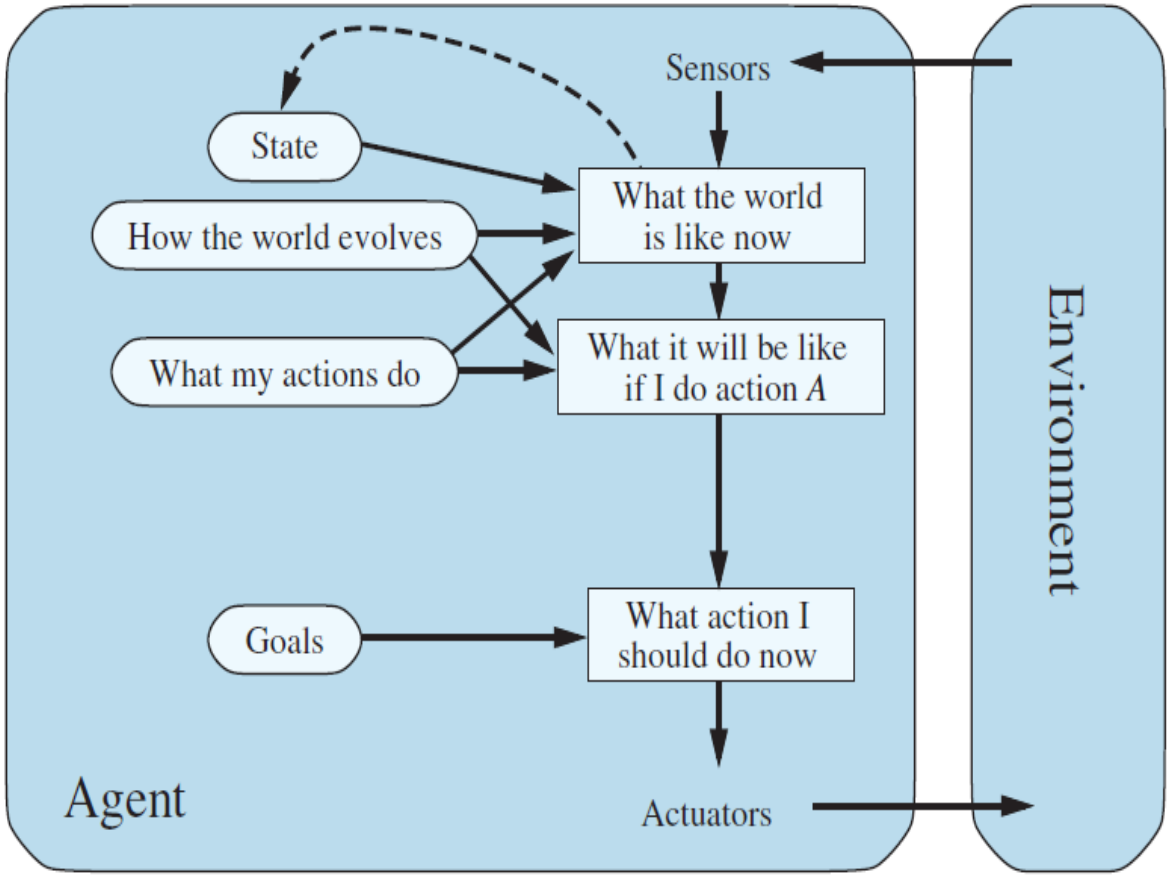


- the agent should maintain some sort of **internal state** that depends on the percept history
- Which reflects at least some of the unobserved aspects of the current state

function MODEL-BASED-REFLEX-AGENT(*percept*) **returns** an action
  persistent: *state*, the agent's current conception of the world state
    transition_model, a description of how the next state depends on the current state and action
    sensor_model, a description of how the current world state is reflected in the agent's percepts
    rules, a set of condition–action rules
    action, the most recent action, initially none

state ← UPDATE-STATE(*state*, *action*, *percept*, *transition_model*, *sensor_model*)
rule ← RULE-MATCH(*state*, *rules*)
action ← *rule*.ACTION
return *action*

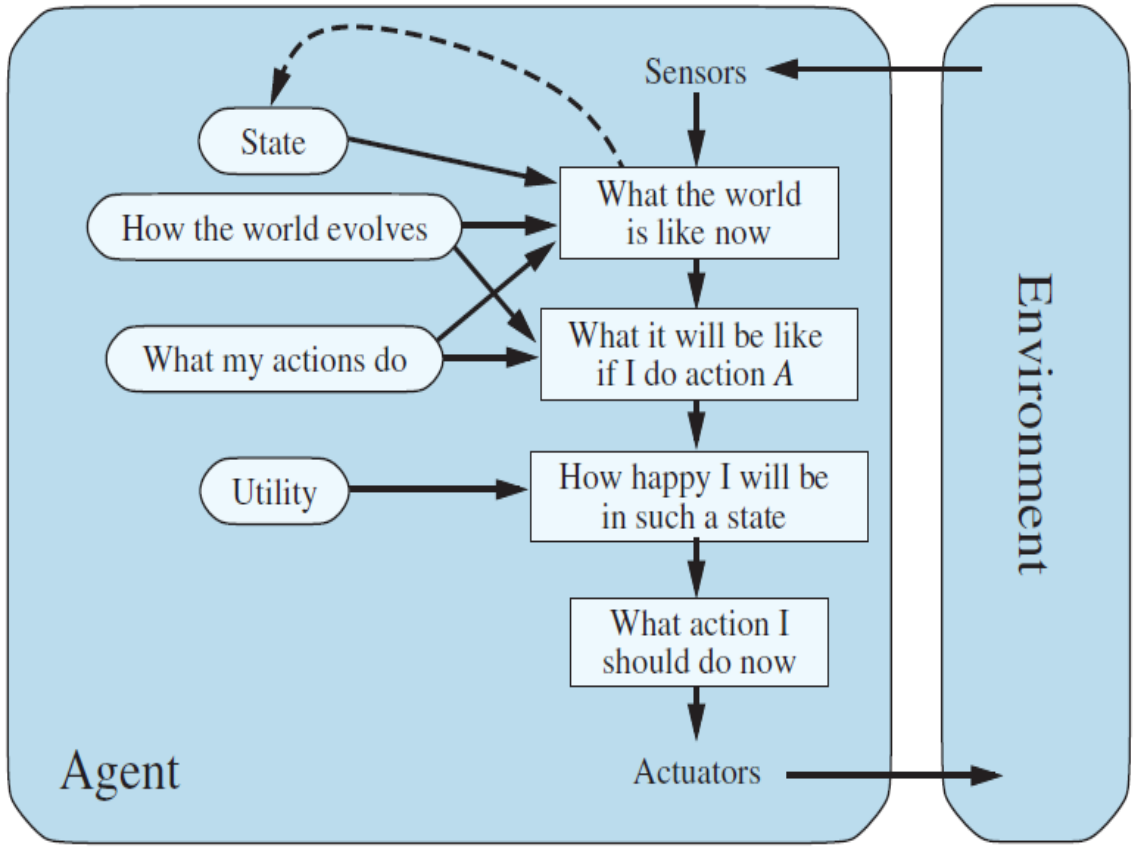
Model based, Goal based agent



The agent program can combine

- The model
- choose actions that achieve the goal
- **Search** and **Planning** are subfields of AI devoted to finding action sequences that achieve the agent's goals

Model based , Utility based agent



Performance measure

- assigns a score to any given sequence of environment states, so it can easily distinguish between good and bad action

An agent's **utility function** is an internalization of the performance measure

- If there are multiple goals, the utility function specifies the appropriate **trade-off**
- when there are several goals that the agent can aim for
  - none of which can be achieved with certainty
  - utility provides a way in which the likelihood of success can be weighed against the importance of the goals

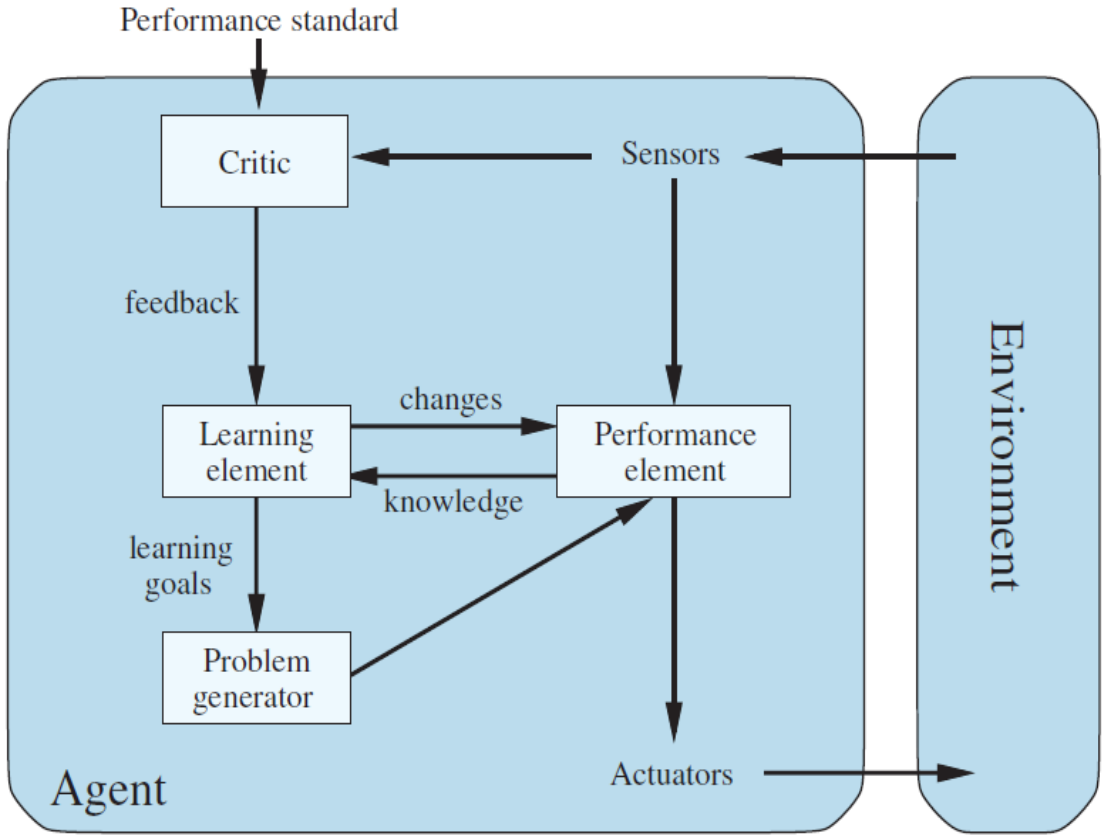
Omniscience

- An omniscient agent knows the actual outcome of its actions and can act accordingly
- but **omniscience** is impossible in reality.
- **Rationality** maximizes expected performance, while perfection maximizes actual performance
- rationality does not require omniscience, then, because the rational choice depends only on the percept sequence to date
- Doing actions in order to modify future percepts—sometimes called **information gathering**
- Example - exploration that must be undertaken by a vacuum-cleaning agent in an initially unknown environment

Learning & Autonomy

- Learn from what it perceives
- The agent's initial configuration could reflect some prior knowledge of the environment, but as the agent gains experience this may be modified and augmented.
- There are extreme cases in which the environment is completely known a priori.
- In such cases, the agent need not perceive or learn; it simply acts correctly
- An agent relies on the prior knowledge of its designer rather than on its own percepts, we say that the agent lacks **autonomy**

General Learning Agent



Learning element which is responsible for making improvements

Performance element, which is responsible for selecting external actions

Critic tells the learning element how well the agent is doing with respect to a fixed performance standard

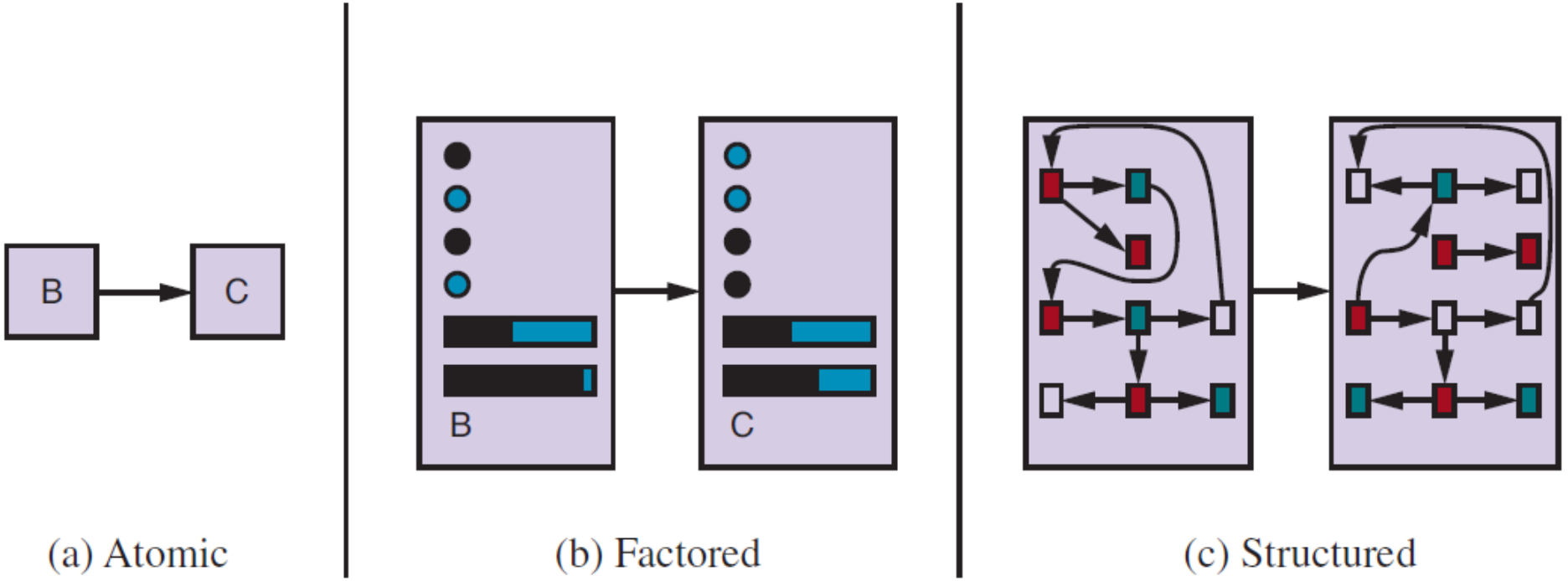
Problem Generator which is responsible for suggesting actions that will lead to new and informative experiences.

Example of Learning Agent

Automated taxi

- **Performance element** consists of whatever collection of knowledge and procedures the taxi has for selecting its driving actions.
- **Critic** observes the world and passes information along to the learning element.
- **Problem generator** might identify certain areas of behavior in need of improvement and suggest experiments
  - Example: Trying out the brakes on different road surfaces under different conditions.

Representing States and Transitions



References

1. Russell S., and Norvig P., Artificial Intelligence A Modern Approach (3e), Pearson 2010

Artificial Intelligence
DSE 3252

Problem Solving

ROHINI R RAO
DEPT OF DATA SCIENCE & COMPUTER APPLICATIONS

JANUARY 2023

Problem Solving Agents

Are goal based agents that use atomic representations

Goal formulation

- First step in problem solving
- based on the current situation and the agent's performance measure
- Environment is represented by states
- Goal state in one in which the goal is satisfied

The agent's task is to find out how to act, now and in the future, so that it reaches a goal state

Problem formulation

- is the process of deciding what actions and states to consider, given a goal

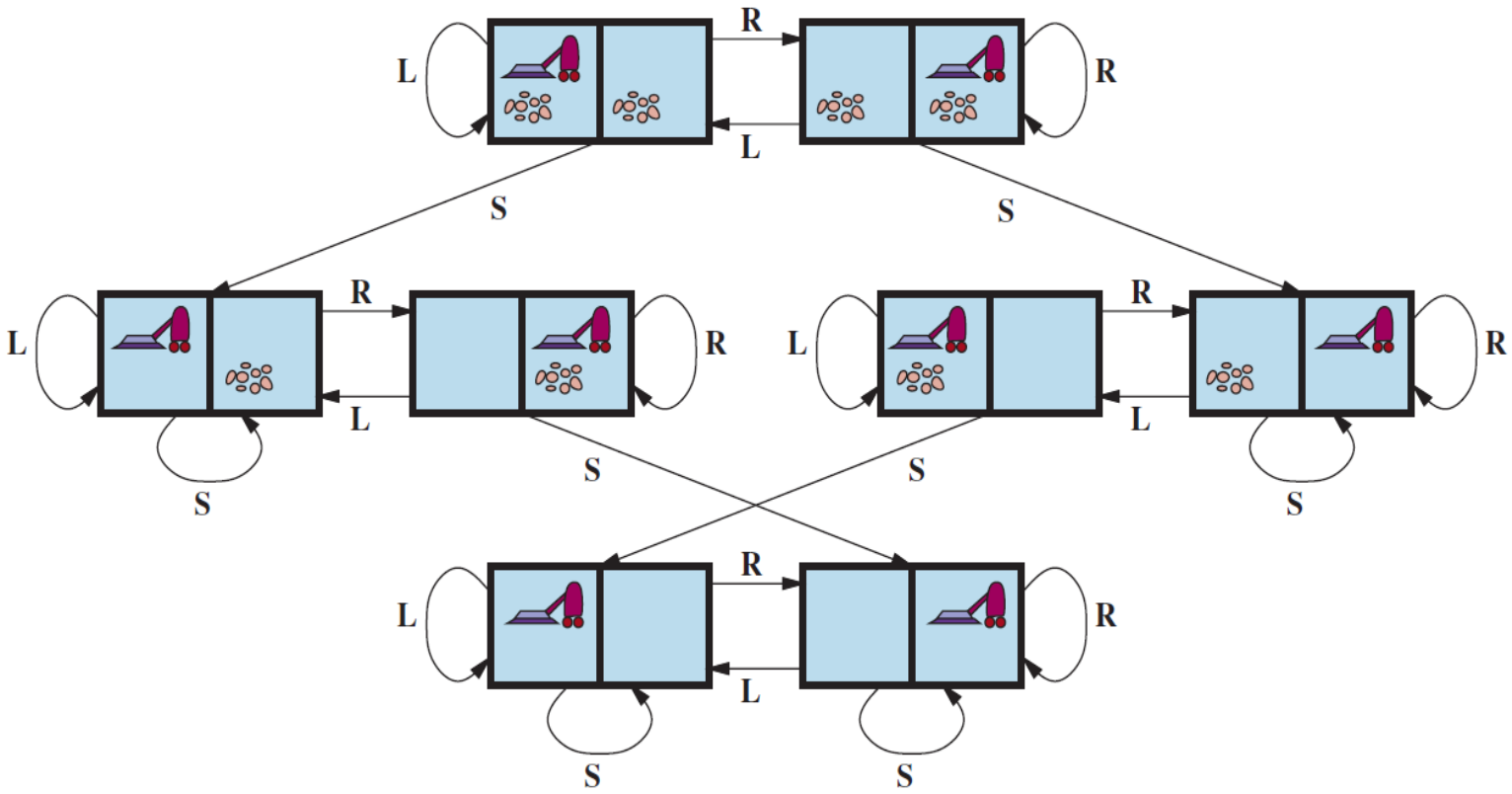
an agent with several immediate options of unknown value can decide what to do by first examining future actions that eventually lead to states of known value

State Space

- forms a **directed network or graph** in which the nodes are states and the links between nodes are actions.
- A **path** in the state space is a sequence of states connected by a sequence of actions.

- A **solution** to a problem is an action sequence that leads from the initial state to a goal state
- Solution quality is measured by the **path cost** function
- an **optimal solution** has the lowest path cost among all solutions

Example 1 – Vacuum Agent State Transition Diagram



Well-defined problems and solutions

A **Problem** can be defined formally:

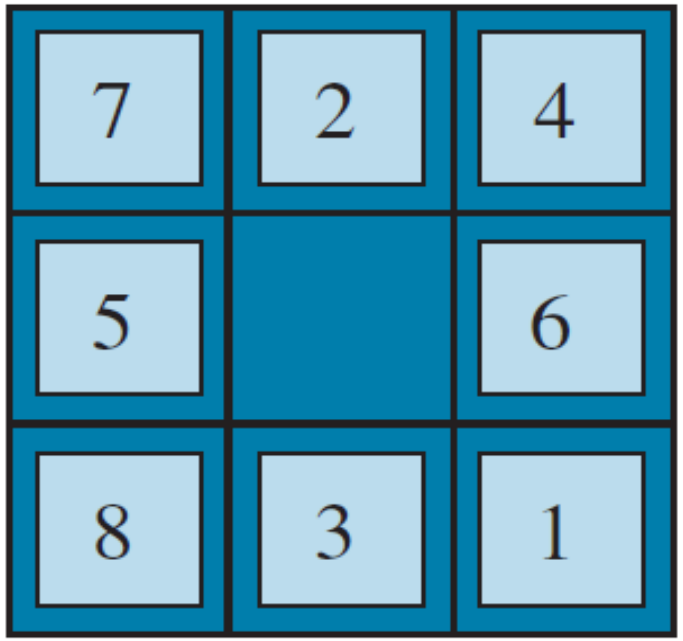
1. The **initial state** that the agent starts in
2. A description of the possible **actions**
3. **Transition model** available to the agent
   - A description of what each action does;
4. The **goal test**, which determines whether a given state is a goal state
5. A **path cost** function that assigns a numeric cost to each path.
   - **Successor** to refer to any state reachable from a given state by a single action

- **State space** of the problem are defined by the initial state, actions, and transition model

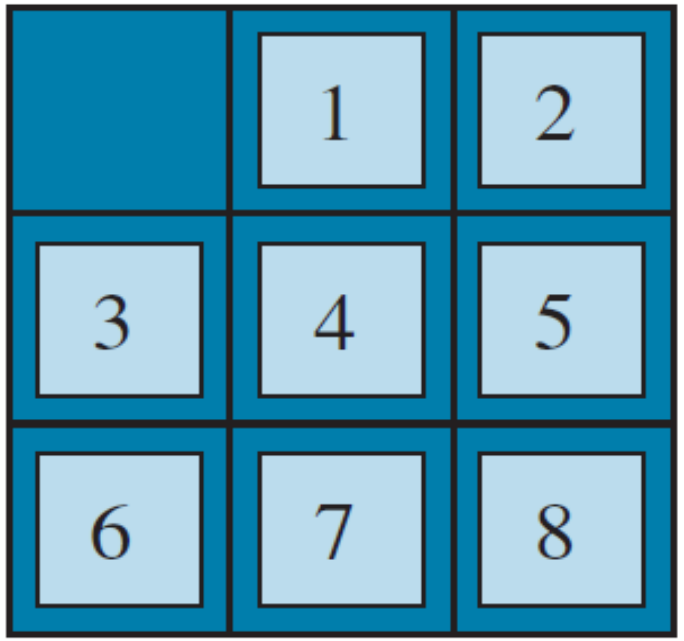
Problem Formulation
Example 1 - Vacuum Agent

- **States:** The state is determined by both the agent location and the dirt locations
  - There are $2 \times 2^2 = 8$ possible world states. A larger environment with n locations has $n \cdot 2^n$ states.
- **Initial state**: Any state can be designated as the initial state.
- **Actions**: each state has just three actions: *Left*, *Right*, and *Suck*
- • **Transition model**: The actions have their expected effects, except that moving *Left* in the leftmost square, moving *Right* in the rightmost square, and *Suck*ing in a clean square have no effect.
- • **Goal test**: This checks whether all the squares are clean.
- • **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.

Problem Formulation
Example 2 – 8 puzzle problem



Start State



Goal State

Problem Formulation
Example 2 – 8 puzzle problem

States: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.

- Initial state: Any state can be designated as the initial state.
- Actions: The simplest formulation defines the actions as movements of the blank space-Left, Right, Up, or Down.
- Transition model: Given a state and action, this returns the resulting state
- Goal test: This checks whether the state matches the goal configuration
- Path cost: Each step costs 1, so the path cost is the number of steps in the path.

“formulate, search, execute”

- The process of looking for a sequence of actions that reaches the goal is called **Search**
  - **Search algorithm** takes a problem as input and returns a **solution** in the form of an action sequence
  - Once a solution is found, the actions are carried out in the **execution** phase
- The possible action sequences starting at the initial state form a **Search Tree:**
  - nodes correspond to states in the state space of the problem.
  - with the initial state NODE at the root
  - the branches are actions
- The set of all leaf nodes available for expansion at any given point is called the **Frontier (Open List)**
- Loopy path
  - result in **repeated state**
  - are a special case of the more general concept of **redundant paths**
- To avoid exploring redundant paths remember where one has been.
- Tree-Search algorithm can be augmented with a data structure called the **explored set (Closed list**), which remembers every expanded node

Partial Search Tree

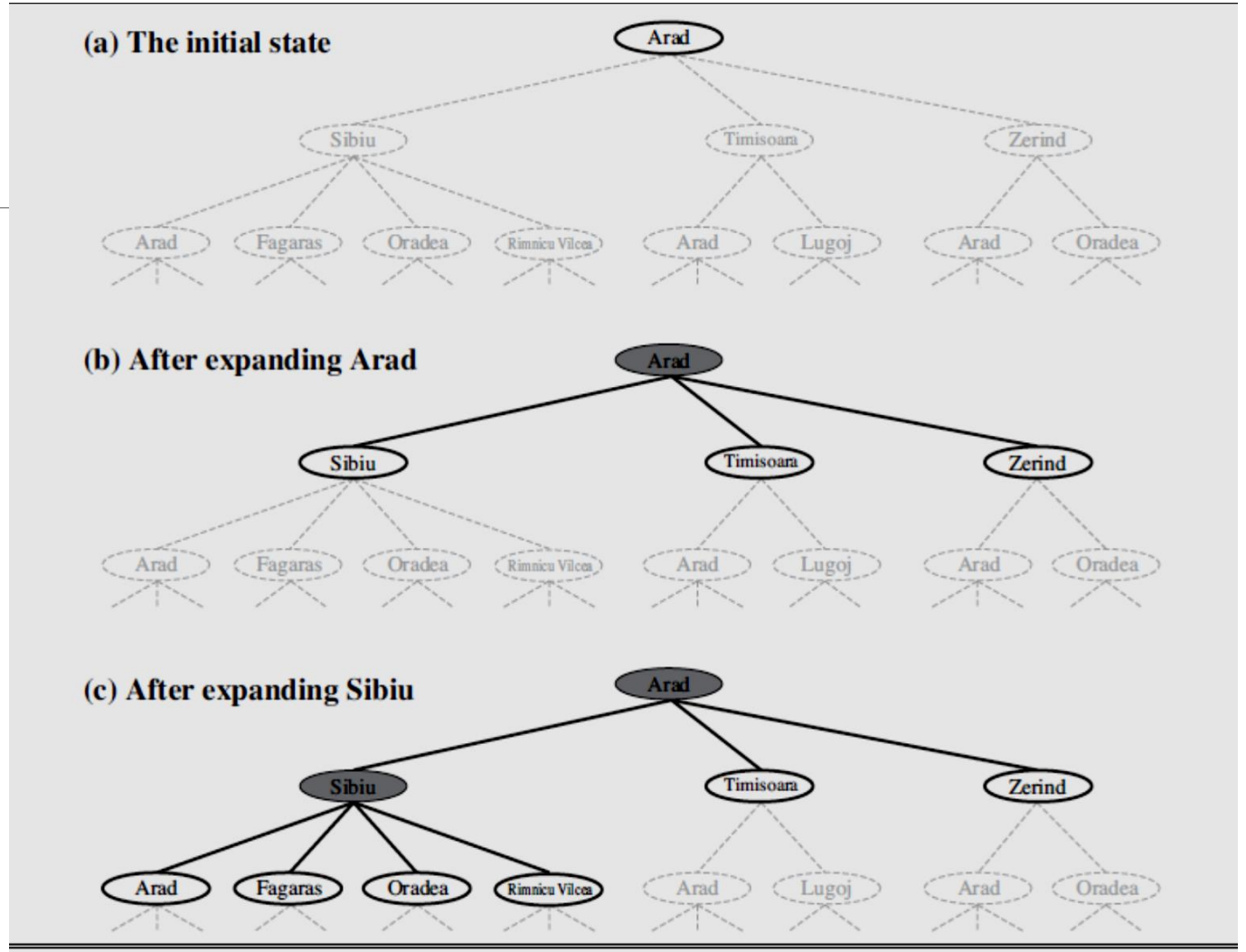


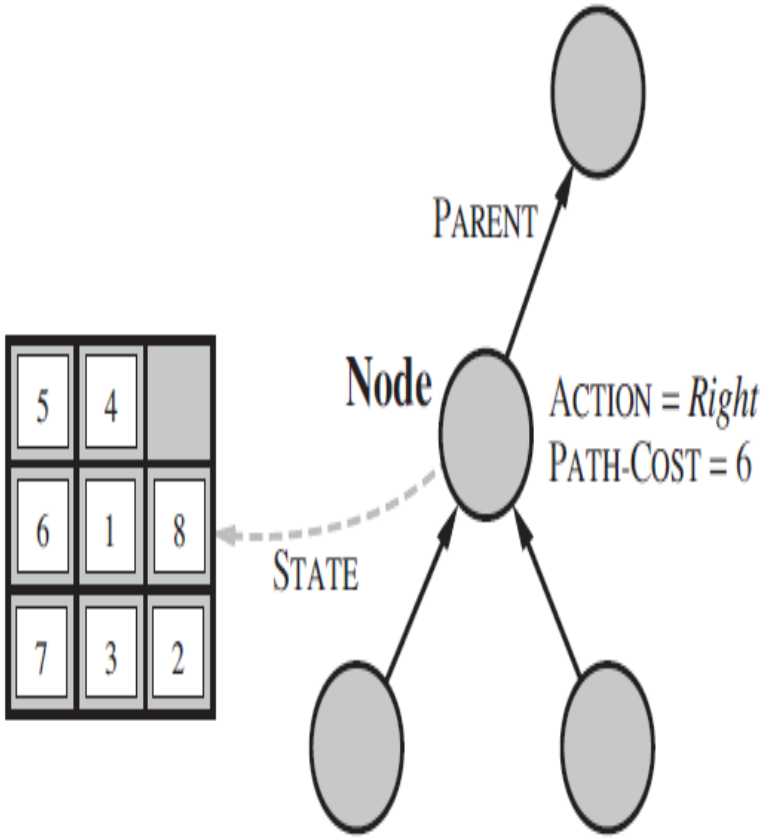
Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

Infrastructure for search algorithms

For each node n of the tree, structure should contains 4 **components**:

- **n.STATE:** the state in the state space to which the node corresponds;
- **n.PARENT**: the node in the search tree that generated this node;
- **n.ACTION**: the action that was applied to the parent to generate the node;
- **n.PATH-COST**: the cost, traditionally denoted by g(n), of the path from the initial state to the node, as indicated by the parent pointers.

function CHILD-NODE(*problem*, *parent*, *action*) **returns** a node
  return a node with
    STATE = *problem*.RESULT(*parent*.STATE, *action*),
    PARENT = *parent*, ACTION = *action*,
    PATH-COST = *parent*.PATH-COST + *problem*.STEP-COST(*parent*.STATE, *action*)



“formulate, search, execute”

- The process of looking for a sequence of actions that reaches the goal is called **Search**
- **Search algorithm** takes a problem as input and returns a **solution** in the form of an action sequence
- Once a solution is found, the actions are carried out in the execution phase

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
      if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

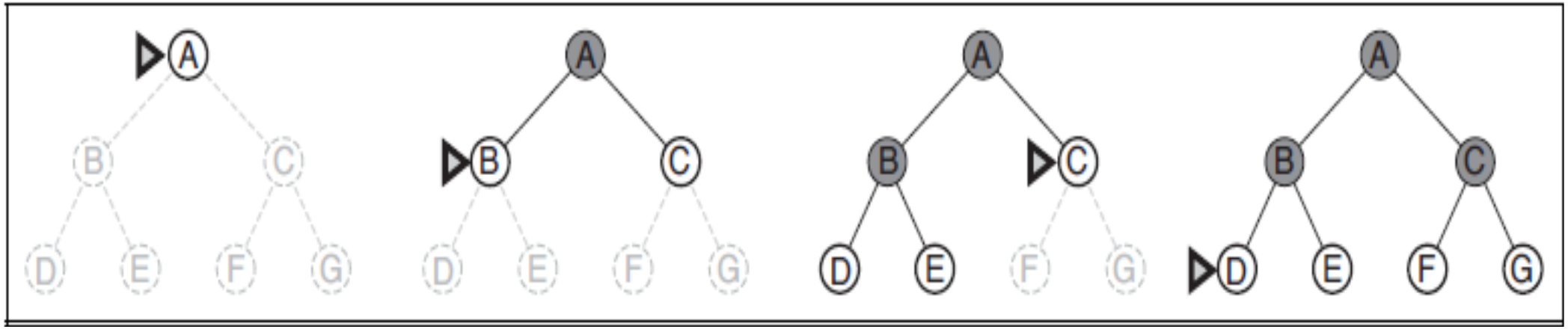
---

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set

Uninformed search strategies
Breadth First Search

Shallowest unexpanded node is chosen for expansion

- the root node is expanded first, then all the successors of the root node are expanded next, then *their* successors, and so on.
- Using a **FIFO queue** for the frontier
- goal test is applied to each node when it is *generated*



Breadth First Search

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)   /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
```

Measuring problem-solving performance Breadth First Search (BFS)

Completeness: Is the algorithm guaranteed to find a solution when there is one?

- Yes, if the shallowest goal node is at some finite depth d & branching factor b is finite.

Optimality: Does the strategy find the optimal solution?

- Yes, if the path cost is a nondecreasing function of the depth of the node.

Time complexity: How long does it take to find a solution?

- The root of the search tree generates b nodes at the first level,, for a total of b^2 at the second level and so on. Suppose that the solution is at depth d.
- Then the total number of nodes generated is $b + b^2 + b^3 \cdot \cdot \cdot + b^d = O(b^d)$

Space complexity: How much memory is needed to perform the search?

- There will be $O(b^{d-1})$ nodes in the explored set and $O(b^d)$ nodes in the frontier

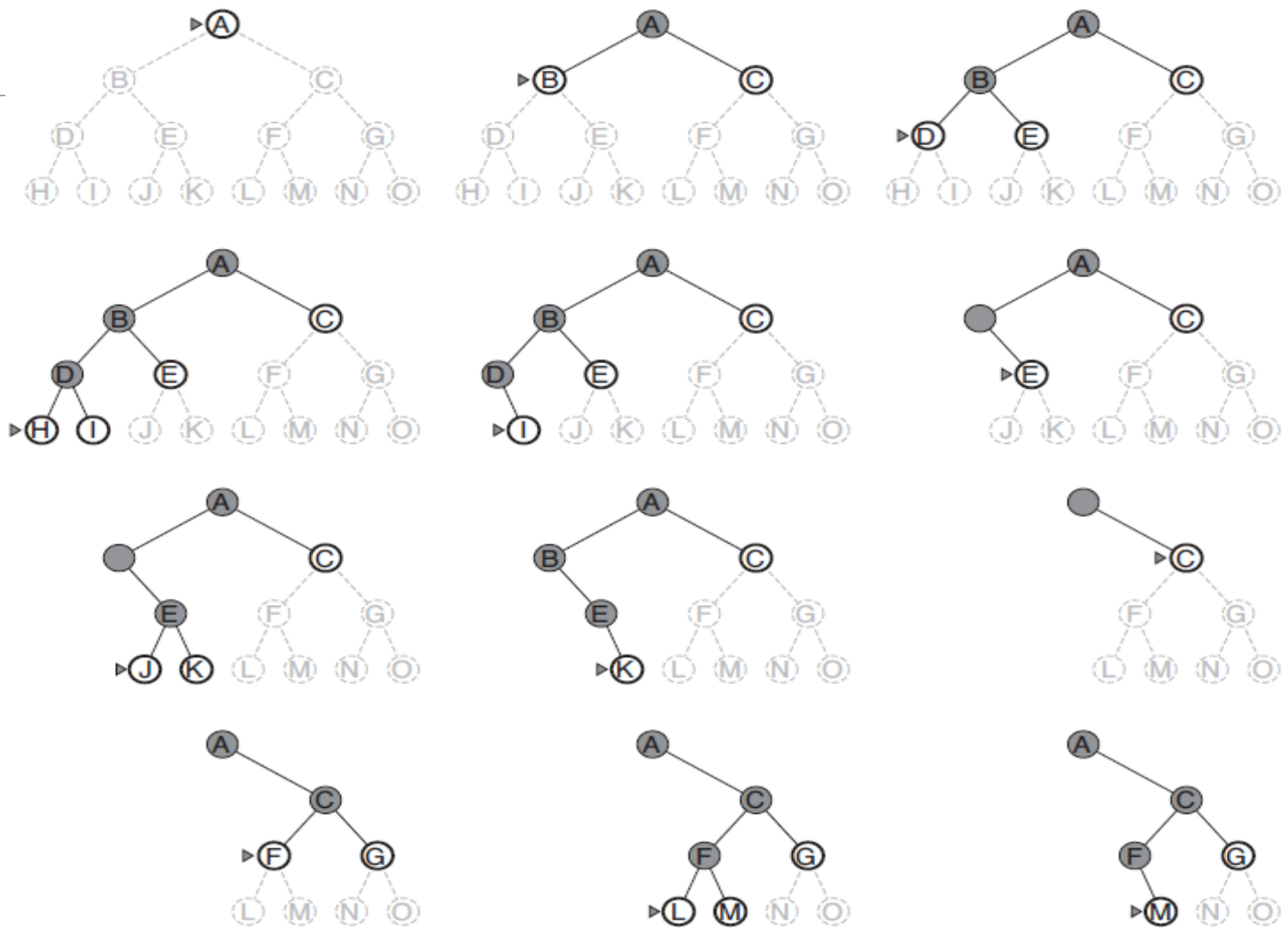
Time and memory requirements for BFS

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Depth First Search

- Instance of graph search which uses LIFO queue
- always expands the *deepest* node in the current frontier of the search tree
- The search proceeds immediately to the **deepest level** of the search tree, where the nodes have no successors.
- As those nodes are expanded, they are dropped from the **frontier**, so then the search "backs up" to the next deepest node that still has unexplored successors.

Depth First Search



Measuring problem-solving performance Depth First Search (BFS)

Completeness: Is the algorithm guaranteed to find a solution when there is one?

- The graph-search version, which avoids repeated states and redundant paths, is complete in finite state spaces
- The tree-search version, on the other hand, is not complete

Optimality: Does the strategy find the optimal solution?

- both versions are nonoptimal

Time complexity: How long does it take to find a solution?

- all of the $O(b^m)$ nodes in the search tree, where m is the maximum depth of any node;

Space complexity: How much memory is needed to perform the search?

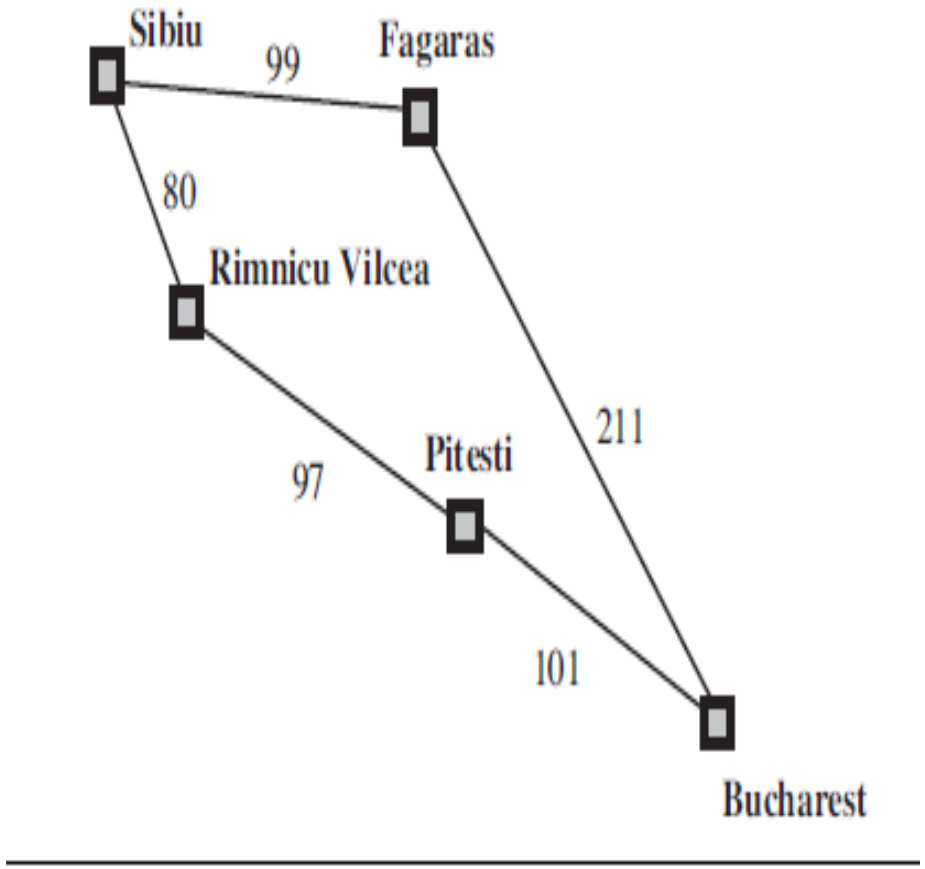
- requires storage of only O(bm) nodes

Uniform Cost Search

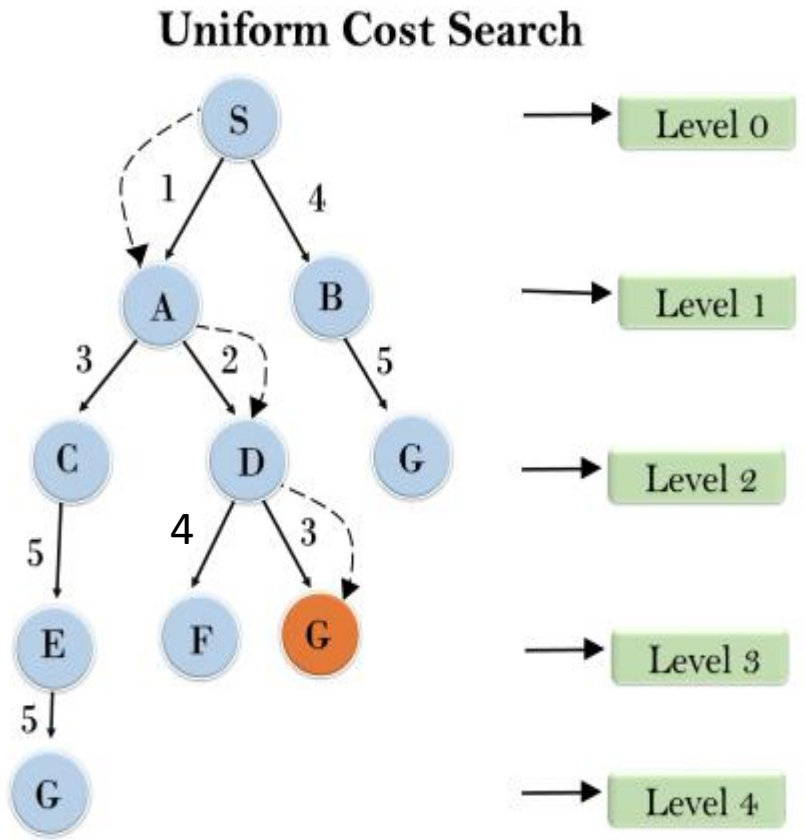
- When all step costs are equal, breadth-first search is optimal because it always expands the shallowest unexpanded node.
- Instead of expanding the shallowest node, uniform-cost search expands the node n with the lowest path cost g(n)
- Store the frontier as a priority queue ordered by g
- Difference from Breadth-first search
  - the goal test is applied to a node when it is selected for expansion rather than when it is first generated
  - a test is added in case a better path is found to a node currently on the frontier

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure

  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
      if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
      add node.STATE to explored
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          if child.STATE is not in explored or frontier then
              frontier ← INSERT(child, frontier)
          else if child.STATE is in frontier with higher PATH-COST then
              replace that frontier node with child
```

Uniform Cost Search



Uniform Cost Search - Example



Measuring problem-solving performance Uniform Cost

Completeness: Is the algorithm guaranteed to find a solution when there is one?

- Completeness is guaranteed provided the cost of every step exceeds some small positive constant €
- May get stuck in an infinite loop if there is a path with an infinite sequence of zero-cost actions

Optimality: Does the strategy find the optimal solution?

- Yes

Time complexity: How long does it take to find a solution?

Space complexity: How much memory is needed to perform the search?

- let C *be the cost of the optimal solution and assume that every action costs at least € Then the algorithm's worst-case time and space complexity is $O(b^{1+\lfloor C^*/\epsilon \rfloor})$

Comparing Uninformed cost strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

Informed (Heuristic) Search Strategies

- The general approach is called **best-first search**.
- Best-first search is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an **evaluation function**, f(n).
- The evaluation function is construed as a cost estimate, so the node with the *lowest* evaluation is expanded first.
- The choice of f determines the search strategy.
- Most best-first algorithms include as a component of f a **heuristic function**, denoted h(n):
- h(n) = estimated cost of the cheapest path from the state at node *n* to a goal state.

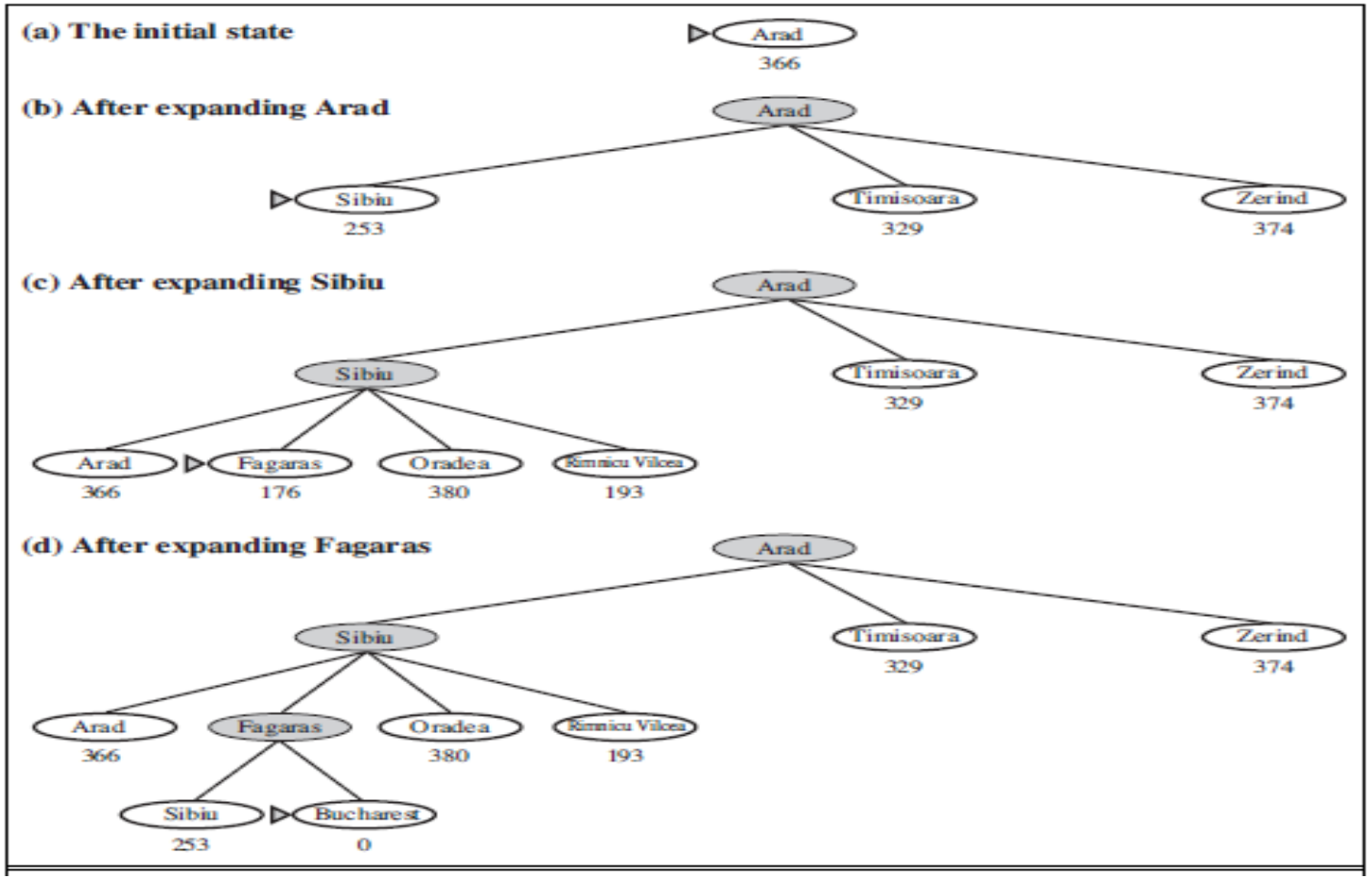
Greedy best-first search

- tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly.
- Thus, it evaluates nodes by using just the heuristic function; that is, f(n) = h(n)

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Figure 3.22 Values of h_{SLD}—straight-line distances to Bucharest.

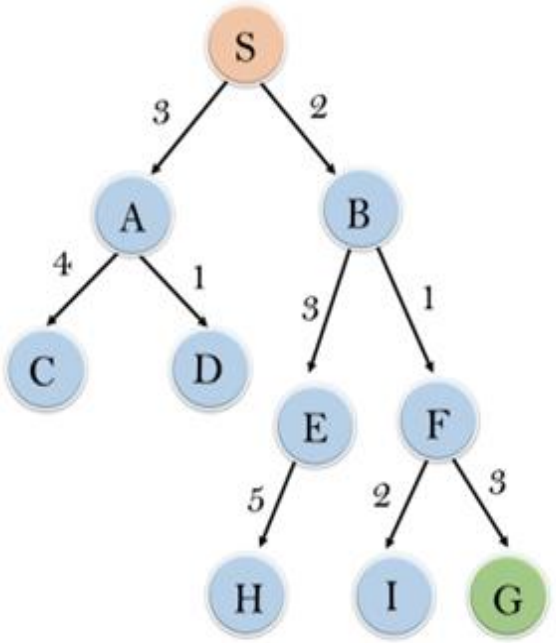
Greedy best-first search



Greedy Best First Algorithm

- **Step 1:** Place the starting node into the OPEN list.
- **Step 2:** If the OPEN list is empty, Stop and return failure.
- **Step 3:** Remove the node n, from the OPEN list which has the lowest value of h(n), and place it in the CLOSED list.
- **Step 4:** Expand the node n, and generate the successors of node n.
- **Step 5:**
  - Check each successor of node n, and find whether any node is a goal node.
  - If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.
- **Step 6:**
  - For each successor node, checks for evaluation function f(n), and then check if the node has been in either OPEN or CLOSED list.
  - If the node has not been in both list, then add it to the OPEN list.
- **Step 7:** Return to Step 2.

Greedy Best First Search - Example



node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0

Measuring problem-solving performance Greedy Best First Search

Completeness: Is the algorithm guaranteed to find a solution when there is one?

- Greedy best-first tree search is incomplete even in a finite state space, much like depth-first search

Optimality: Does the strategy find the optimal solution?

- No

Time complexity: How long does it take to find a solution?

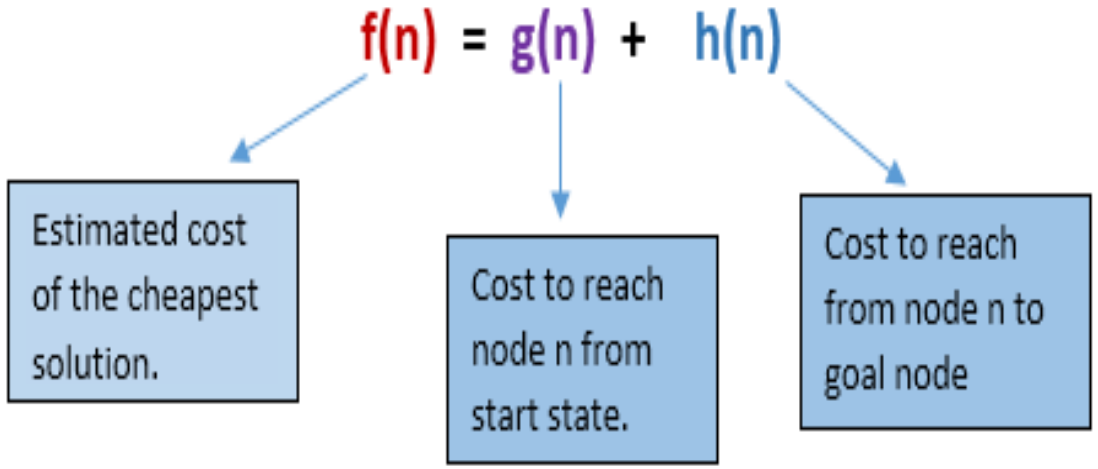
- The worst case time complexity of Greedy best first search is $O(b^m)$.

Space complexity: How much memory is needed to perform the search?

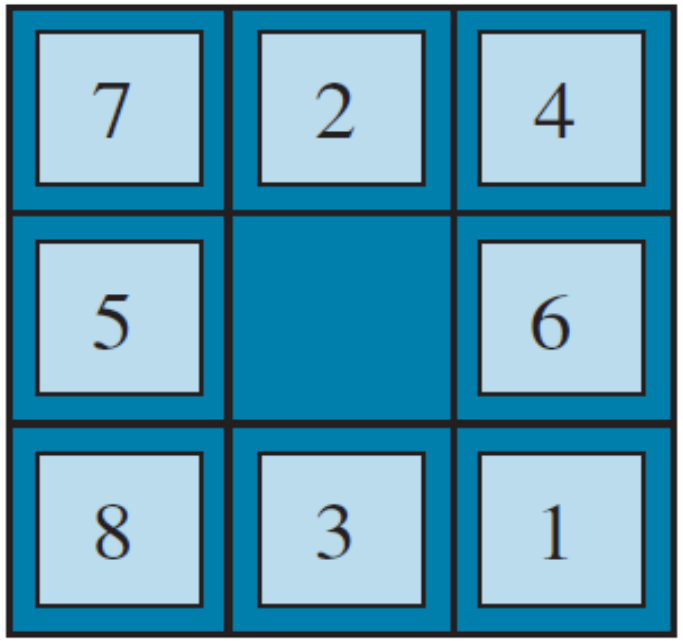
- The worst case space complexity of Greedy best first search is $O(b^m)$. Where, m is the maximum depth of the search space.

A* search: Minimizing the total estimated solution cost

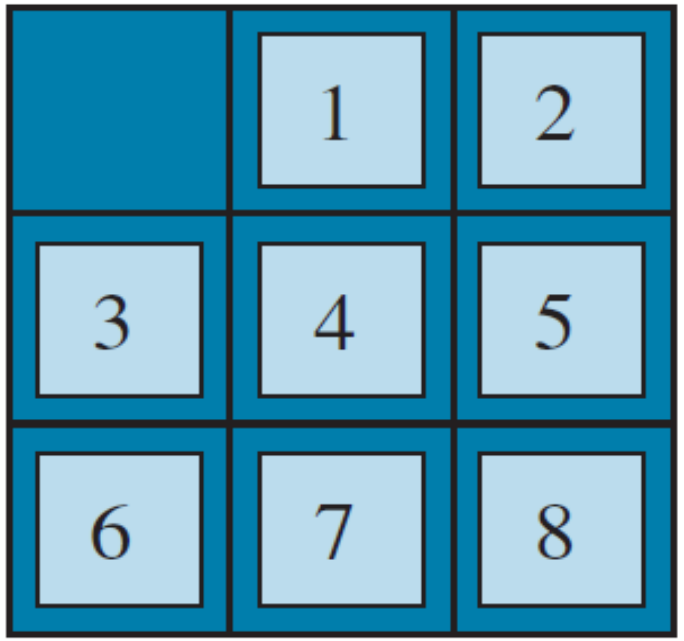
It evaluates nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from the node to the goal:



Heuristic Function
Example 2 – 8 puzzle problem

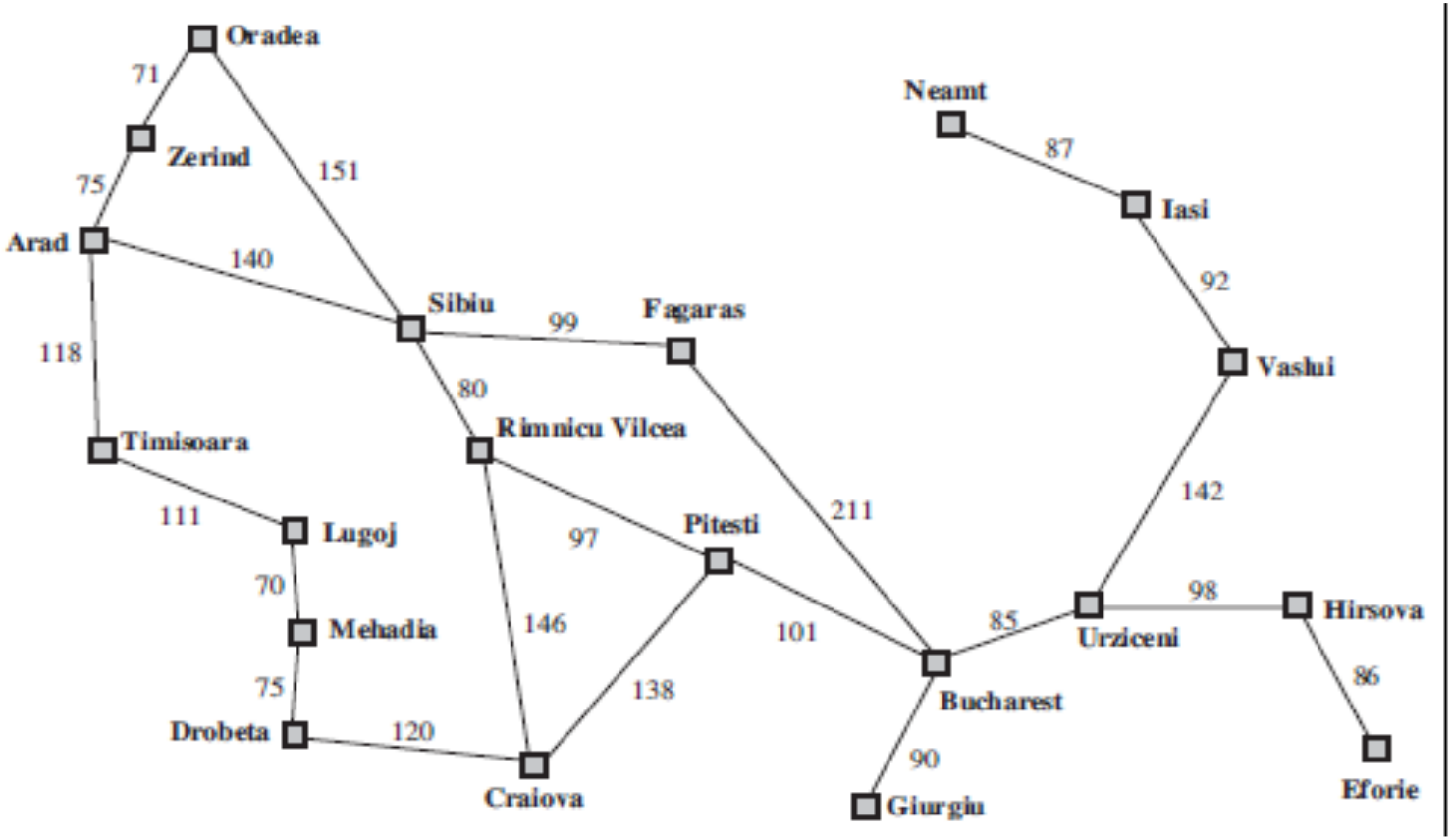


Start State

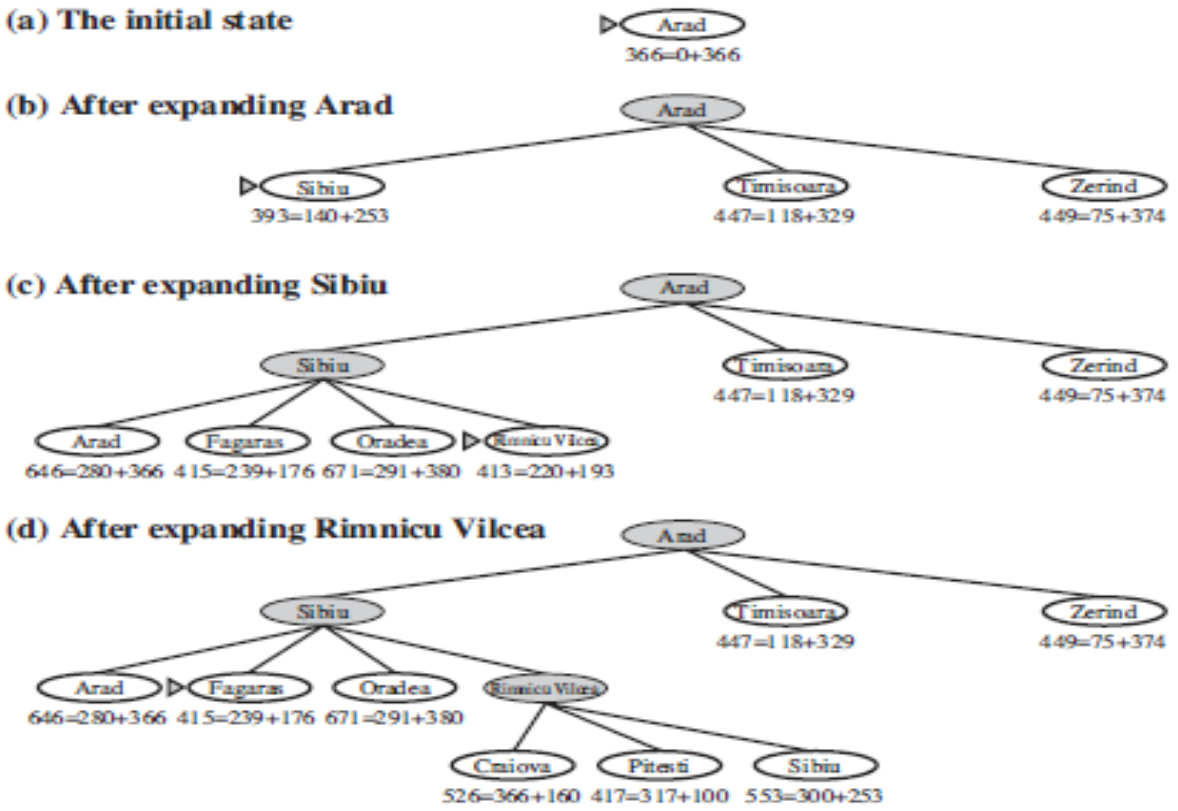


Goal State

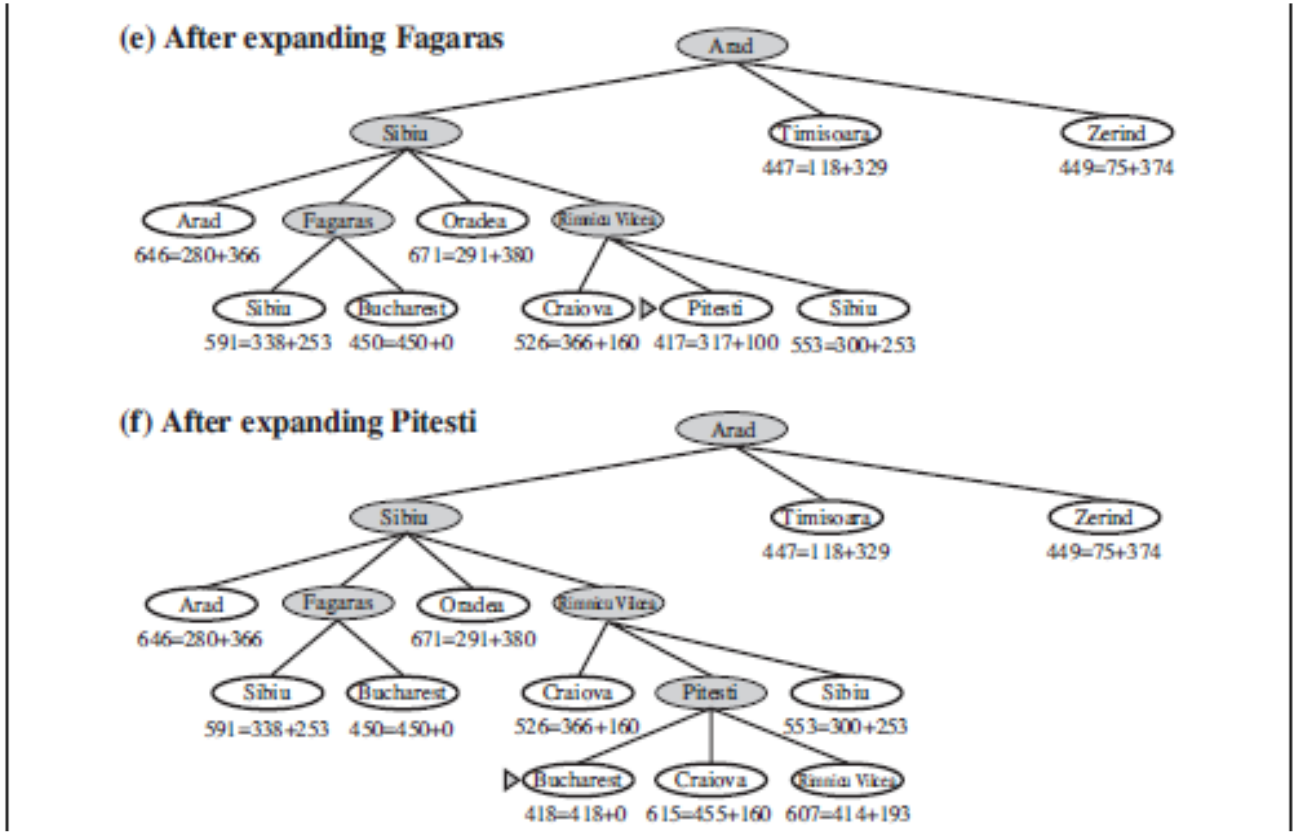
A* - Combines SLD with Path Cost



Stages in A*


(a) The initial state
Arad
366=0+366
(b) After expanding Arad
Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374
(c) After expanding Sibiu
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193
(d) After expanding Rimnicu Vilcea
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253

Stages in A*


(e) After expanding Fagaras
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253
(f) After expanding Pitesti
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
Sibiu
553=300+253
Bucharest
418=418+0
Craiova
615=455+160
Rimnicu Vilcea
607=414+193

Algorithm of A* search:

Step1: Place the starting node in the OPEN list.

Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stops.

Step 3:

- Select the node from the OPEN list which has the smallest value of evaluation function (g+h)
- if node n is goal node then return success and stop

Step 4:

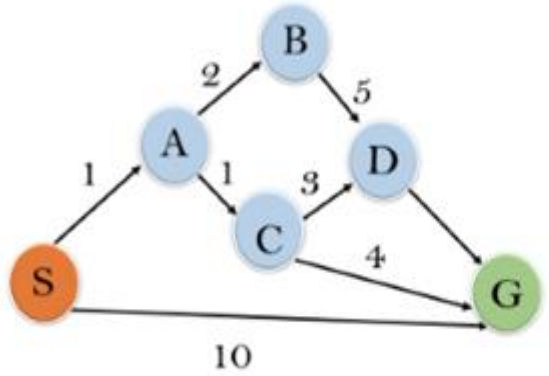
- Expand node n and generate all of its successors, and put n into the closed list.
- For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.

Step 5:

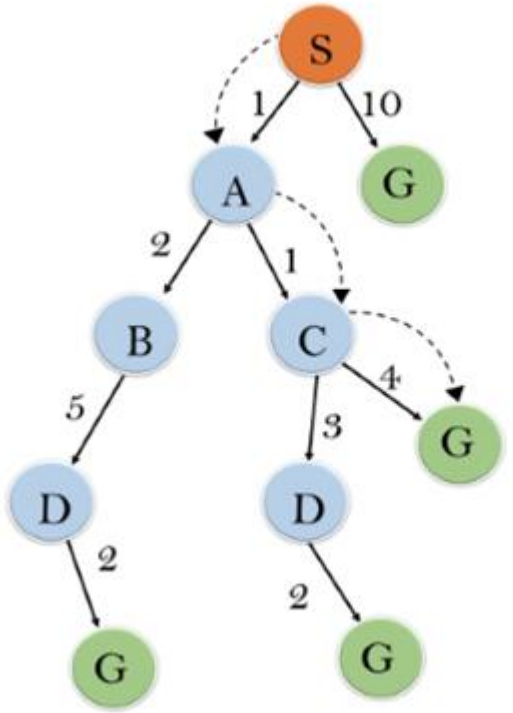
- Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.

Step 6: Return to **Step 2**.

A* Search - Example



State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0



Measuring problem-solving performance
A*

Completeness: Is the algorithm guaranteed to find a solution when there is one?

- Complete as long as , Branching factor is finite & Cost at every action is fixed

Optimality: Does the strategy find the optimal solution?

A* search algorithm is optimal if it follows below two conditions:

- **Admissible:** h(n) should be an admissible heuristic for A* tree search.
  - An admissible heuristic is optimistic in nature, never overestimates the cost to reach the goal
- **Consistency:** for only A* graph-search.
  - A heuristic h(n) is consistent if, for every node n and every successor n' of n generated by any action a, the estimated cost of
  - reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n'

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

Time complexity: How long does it take to find a solution?

- Depends on Heuristic Function and based on depth d of solution is $O(b^d)$.

Space complexity: How much memory is needed to perform the search?

- $O(b^d)$

A* - Features

- A* algorithm returns the path which occurred first, and it does not search for all remaining paths.
- The efficiency of A* algorithm depends on the quality of heuristic.
- A* algorithm expands all nodes which satisfy the condition f(n)

Advantages:

- A* search algorithm is the best algorithm than other search algorithms.
- A* search algorithm is optimal and complete.
- This algorithm can solve very complex problems.

Disadvantages:

- It does not always produce the shortest path as it mostly based on heuristics and approximation.
- A* search algorithm has some complexity issues.
- The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

Adversarial Search

In **multiagent environments**, each agent needs to consider the actions of other agents and how they affect its own welfare

The unpredictability of these other agents can introduce **contingencies** into the agent's problem-solving process

Competitive environments, in which the agents' goals are in conflict, giving rise to **adversarial search** problems—often known as **games**.

Mathematical **game theory**

- a branch of economics
- views any multiagent environment as a game, provided that the impact of each agent on the others is "significant,"
- regardless of whether the agents are cooperative or competitive

In AI, games are

- deterministic, turn-taking, two-player,
- **zero-sum games** of **perfect information** (deterministic, fully observable environments)
- in which two agents act alternately and in which the utility values at the end of the game are always equal and opposite.

this opposition between the agents' utility functions that makes the situation adversarial

Game defined as a kind of search problem

S_0: The initial state, which specifies how the game is set up at the start

PLAYER(s): Defines which player has the move in a state.

ACTIONS(s): Returns the set of legal moves in a state.

RESULT(s, a): The transition model, which defines the result of a move.

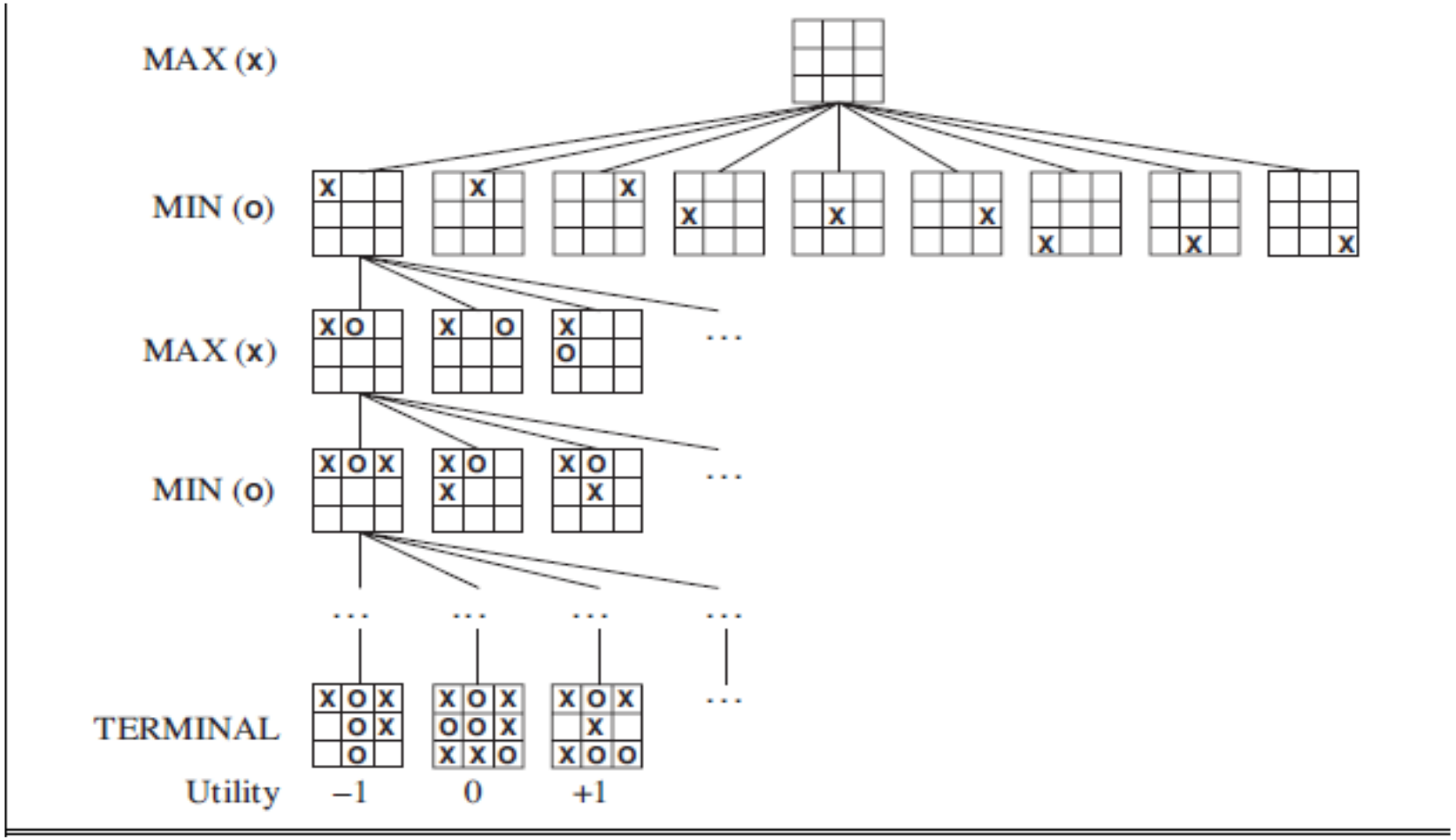
TERMINAL-TEST(s):

- A terminal test, which is true when the game is over and false otherwise. States where the game has ended are called terminal states.

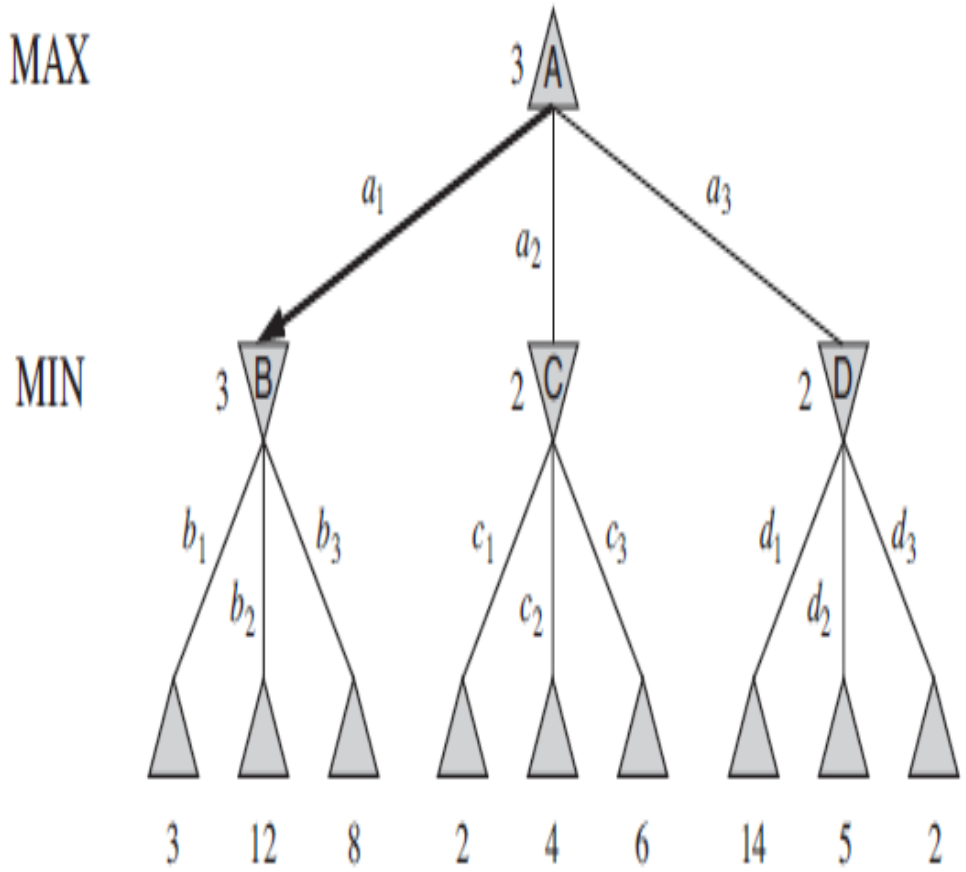
UTILITY(s, p):

- A utility function (also called an objective function or payoff function), defines the final numeric value for a game that ends in terminal state s for a player p.
- In chess, the outcome is a win, loss, or draw, with values +1, 0, or 1/2
- A zero-sum game - where the total payoff to all players is the same for every instance of the game.
- Chess is zero-sum because every game has payoff of either 0 + 1, 1 + 0 or ½ + ½

Partial Game Tree for tic-tac-toe



Optimal Decisions in Games



MAX must find a contingent **strategy**

- which specifies MAX's move in the initial state
- then MAX's moves in the states resulting from every possible response by MIN and so on

Assumption: optimal play for MAX assumes that MIN also plays optimally

Optimum strategy is determined by the minimax value of each node.

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

Minimax Algorithm

computes the minimax decision from the current state.

It uses a simple recursive computation of the minimax values of each successor state

The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are **backed up** through the tree as the recursion unwinds.

performs a complete depth-first exploration of the game tree

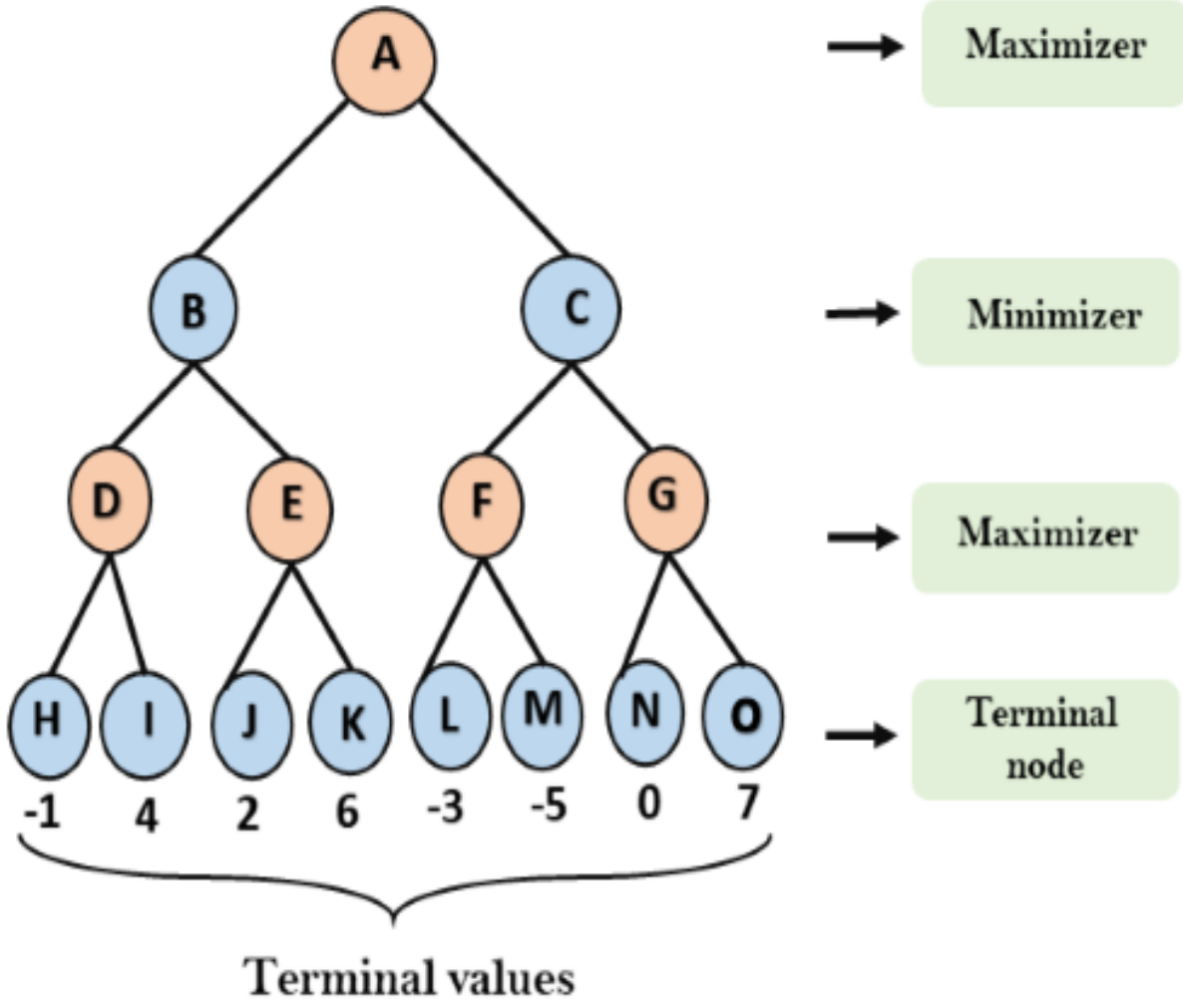
If the maximum depth of the tree is m and there are b legal moves at each point, then

- time complexity of the minimax algorithm is $O(b^m)$
- The space complexity is O(bm) for an algorithm that generates all actions at once, or O(m) for an algorithm that generates actions one at a time

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

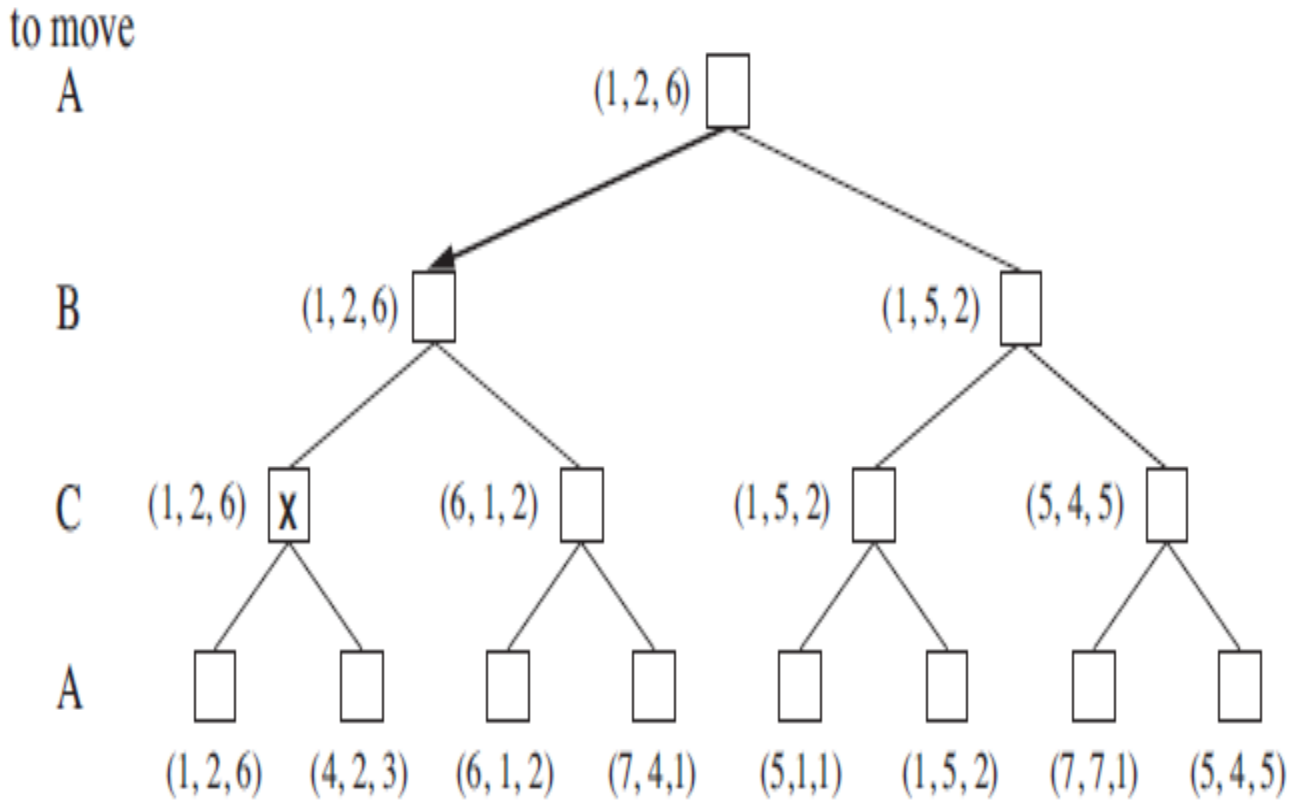
Psuedocode for Mini-Max Algorithm

```
function minimax(node, depth, maximizingPlayer) is
if depth ==0 or node is a terminal node then  return static evaluation of node
 if MaximizingPlayer then      // for Maximizer Player
    maxEval= -infinity
    for each child of node do
        eval= minimax(child, depth-1, false)
        maxEval= max(maxEval,eval)       //gives Maximum of the values
    return maxEval
  else                    // for Minimizer player
   minEval= +infinity
   for each child of node do
       eval= minimax(child, depth-1, true)
       minEval= min(minEval, eval)       //gives minimum of the values
    return minEval
```



- For node D max(-1,- -∞) => max(-1,4)= 4
- For Node E max(2, -∞) => max(2, 6)= 6
- For Node F max(-3, -∞) => max(-3,-5) = -3
- For node G max(0, -∞) = max(0, 7) = 7

3 plies of Game tree with 3 players

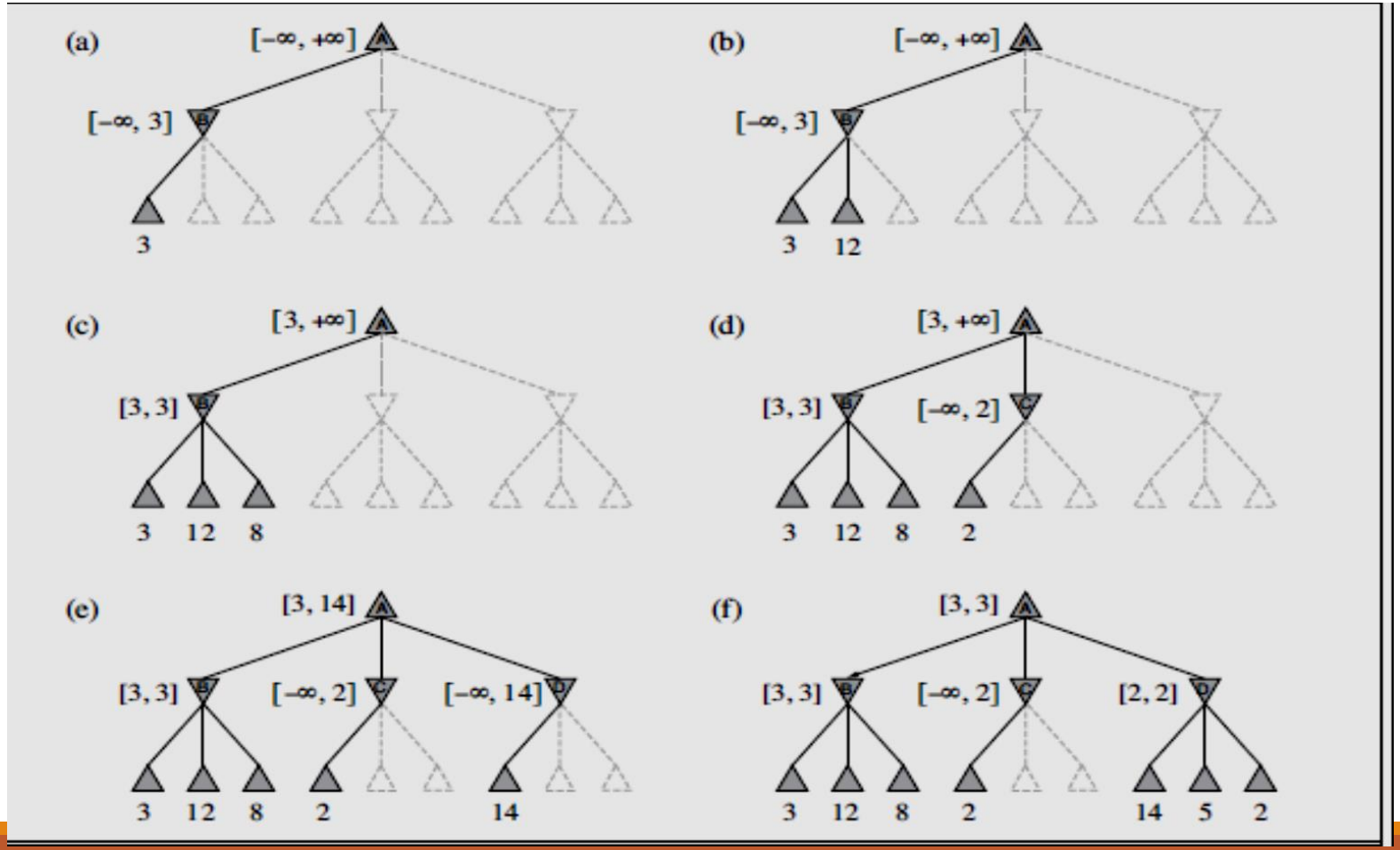


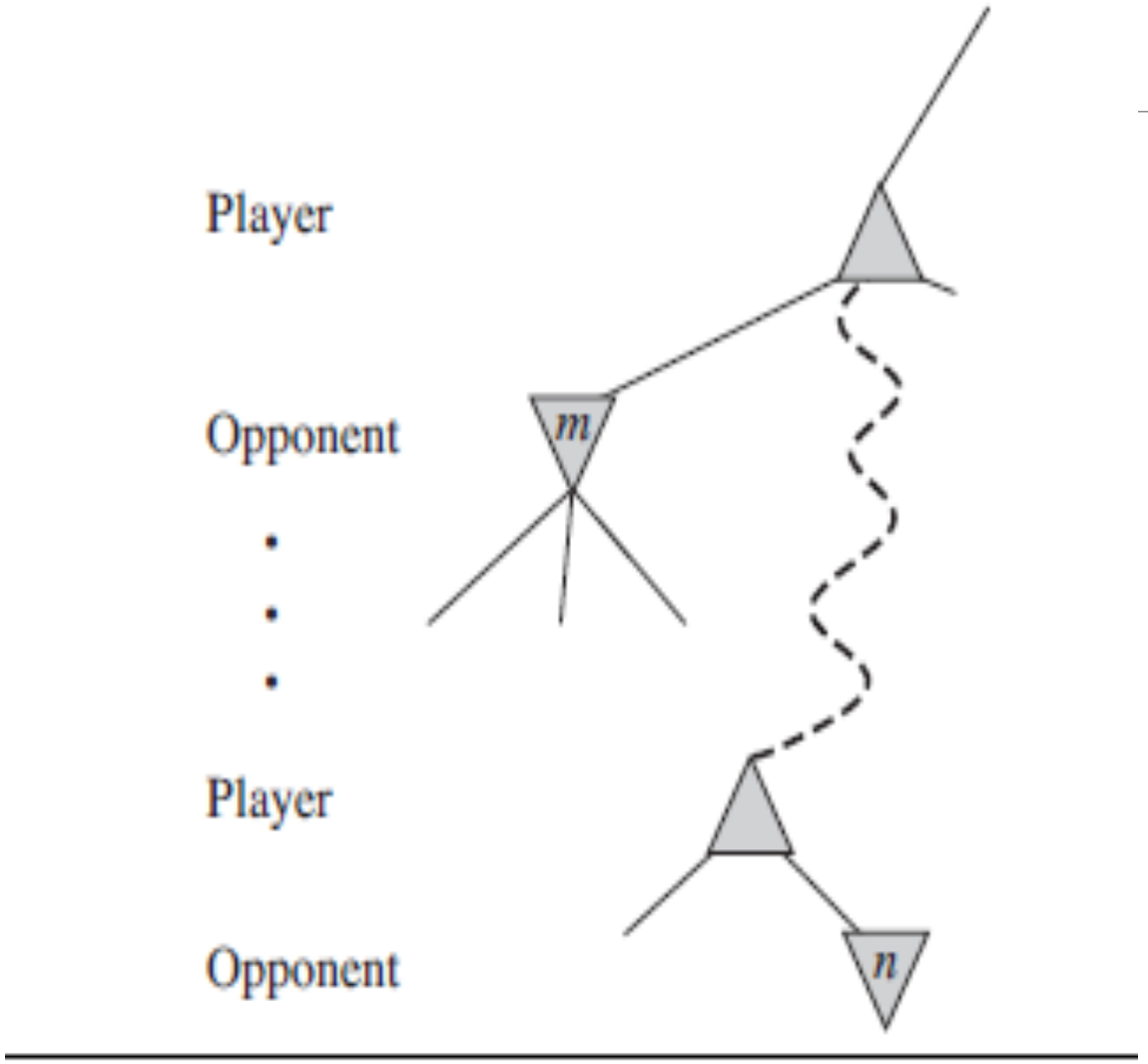
Multiplayer games usually involve **alliances**, whether formal or informal, among the players. Alliances are made and broken as the game proceeds.

Measuring Problem Solving Performance Mini-Max algorithm:

- **Complete-**
  - Yes. It will definitely find a solution (if exist), in the finite search tree.
- **Optimal-**
  - is optimal if both opponents are playing optimally.
- **Time complexity-**
  - As it performs DFS for the game-tree, so the time complexity is **$O(b^m)$**, where b is branching factor of the game-tree, and m is the maximum depth of the tree.
- **Space Complexity**
  - Space complexity of Mini-max algorithm is also similar to DFS which is **O(bm)**.

Stages in the calculation of the optimal decision for the game tree





general case for alpha–beta pruning

If Player has a better choice m either at the parent node of n or at any choice point further up,

then n *will never be reached in actual play.*

So once we have found out enough about n (by examining some of its descendants) to reach this conclusion, we can prune it.

Alpha Beta Pruning

The problem with minimax search is that the number of game states it has to examine is

exponential in the depth of the tree

Alpha–beta pruning

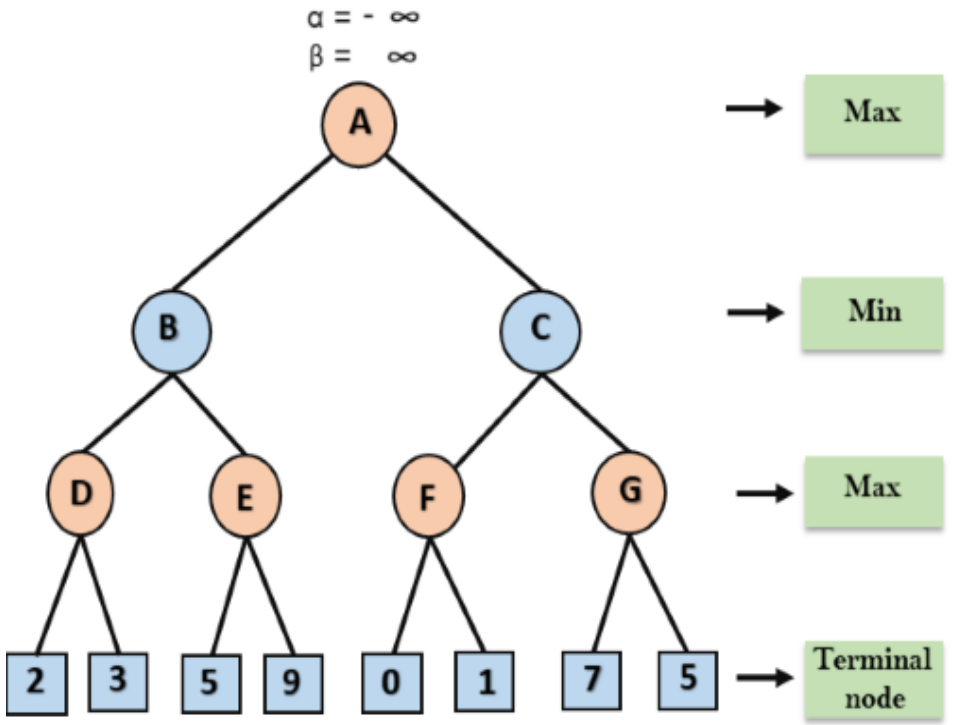
- When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branchs that cannot possibly influence the final decision
- Simplification of Minimax formula

$$\begin{aligned}
\textsc{Minimax}(root) &= \max(\min(3,12,8), \min(2,x,y), \min(14,5,2)) \\
&= \max(3, \min(2,x,y), 2) \\
&= \max(3,z,2) \qquad \text{where } z = \min(2,x,y) \leq 2 \\
&= 3.
\end{aligned}$$

- the value of the root and hence the minimax decision are independent of the values of the pruned leaves x and y.
- Alpha–beta pruning has two parameters that describe bounds on the backed-up values that appear anywhere
- along the path:
- α = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.
- β = the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.

```
function minimax(node, depth, alpha, beta, maximizing player)
if depth ==0 or node is a terminal node then  return static evaluation of node
 if MaximizingPlayer then      // for Maximizer Player
   maxEva= -infinity
   for each child of node do
   eva= minimax(child, depth-1, alpha, beta, False)
  maxEva= max(maxEva, eva)
  alpha= max(alpha, maxEva)
   if beta<=alpha   break
 return maxEva
 else                    // for Minimizer player
   minEva= +infinity
   for each child of node do
   eva= minimax(child, depth-1, alpha, beta, true)
   minEva= min(minEva, eva)
   beta= min(beta, eva)
    if beta<=alpha    break
 return minEva
```

Alpha Beta Pruning



Alpha Beta Search Algorithm

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v
```

```
function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
```

```
function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Improvements on Alpha-Beta Search

Killer Move Heuristics:

- The effectiveness of alpha–beta pruning is highly dependent on the order in which the states are examined
- If we can examine first the successors that are likely to be best, then alpha–beta needs to examine only $O(b^{m/2})$ nodes to pick the best move, instead of $O(b^m)$ for minimax
- **Dynamic move-ordering** – moves that were found to be best in the past.
- Best moves called **killer-moves**

Transposition table

- repeated states occur frequently because of **transpositions**—different permutations of the move sequence that end up in the same position
- The hash table of previously seen positions is maintained in a transposition table
- Minimax & its related algorithms wont work for games like Chess or Go
- Claude Shannon's paper **Programming a Computer for Playing Chess** (1950) proposed
  - **Type A strategy** – (Wide but shallow) Considers all possible moves to a certain depth and then uses heuristic evaluation function to estimate utility of states in that depth
  - **Type B strategy** – (Deep but narrow) ignores moves that look bad and follows promising paths as far as possible

Heuristic Alpha-Beta Tree Search

$$\text{H-MINIMAX}(s,d) = \begin{cases} \text{EVAL}(s) & \text{if CUTOFF-TEST}(s,d) \\ \max_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MIN.} \end{cases}$$

Cutoff test

- should return true if terminal states but otherwise can cutoff search based on depth or any other heuristic

Eval function

- returns an estimate of expected utility of state s to player p
- Should be strongly correlated to actual chances of winning
- Example: Weighted Linear function

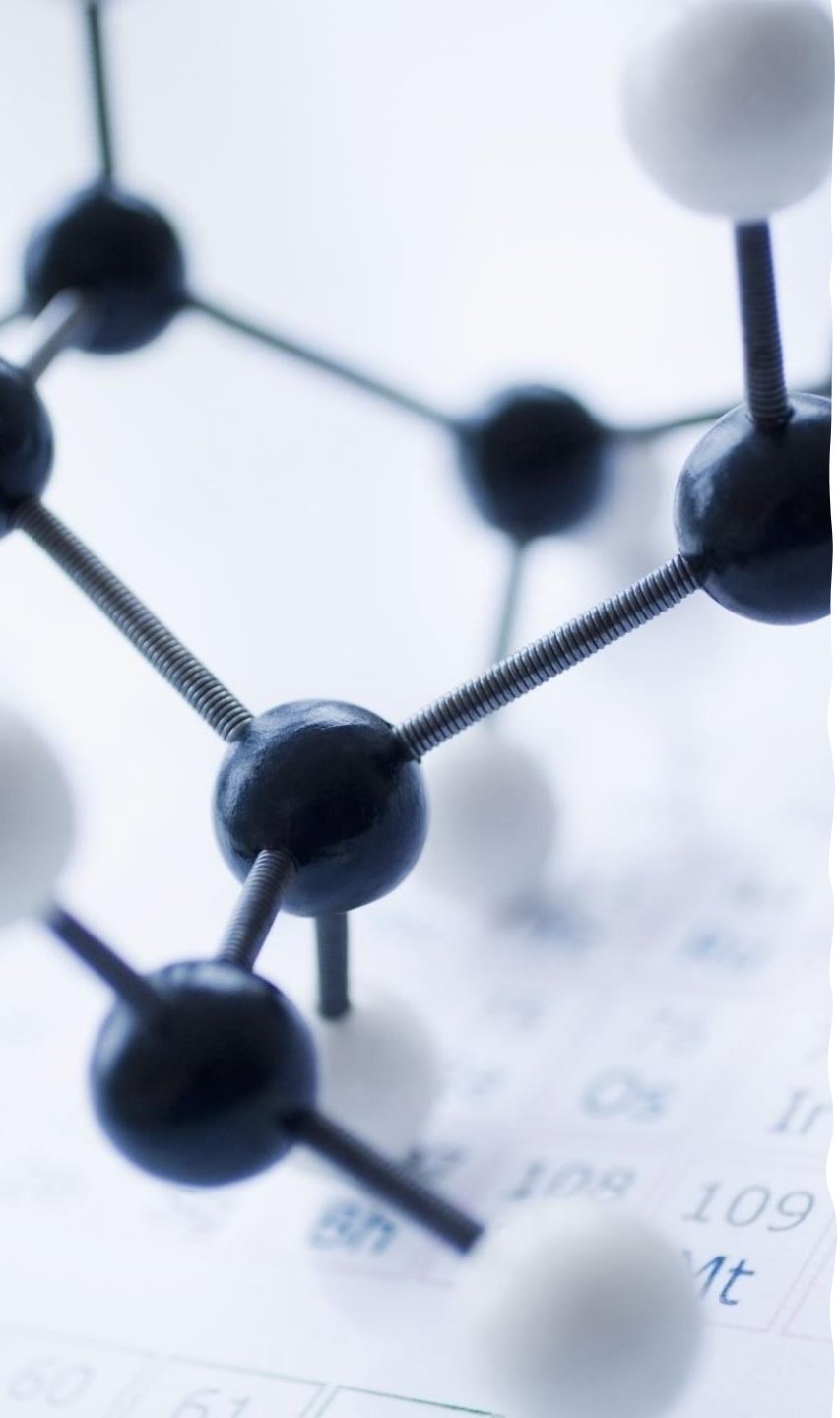
$$\text{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)\,,$$

References

1. Russell S., and Norvig P., Artificial Intelligence A Modern Approach (3e), Pearson 2010

Module 3: Beyond Classical Search
3.1 Local search algorithms and optimization problems

By: Rohini R Rao & Rashmi L Malghan
Dept of Data Science & Computer Applications
January 2024



Local Search Algorithm - Inspired By Evolutionary Biology & Statistic Physics

By: Rohini R Rao & Rashmi L Malghan
Dept of Data Science & Computer Applications
January 2024

Agenda

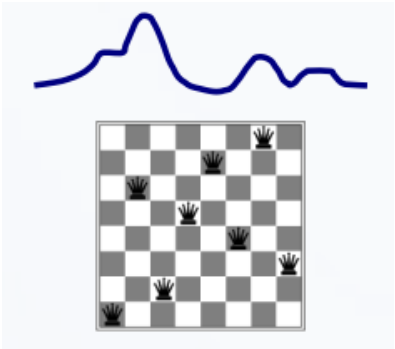
- **Introduction - Hill Climbing**
  - Types of Hill Climbing
  - Example
  - Complexities
  - Applications
- **Simulated Annealing Search**
- **Local Beam Search**
- **Genetic Algorithm**

Iterative Improvement Algorithms

- In the problems we studied so far, the solution is the path.
  - For example, the solution to the 8-puzzle is a series of movements for the "blank tile." The solution to the traveling in Romania problem is a sequence of cities to get to Bucharest.
- In many optimization problems, the path is irrelevant.
  - The goal itself is the solution.
- The state space is set up as a set of "complete" configurations, the optimal configuration is one of them.
- An iterative improvement algorithm keeps a single "current" state and tries to improve it.
- The space complexity is constant

Local Search Algorithms

- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution.
- Examples:
  - to reduce cost, as in cost functions
  - to reduce conflicts, as in n-queens



- The idea: keep a single "current" state, try to improve it according to an objective function.
- Local search algorithms:
  - Use little memory
  - Find reasonable solutions in large infinite spaces

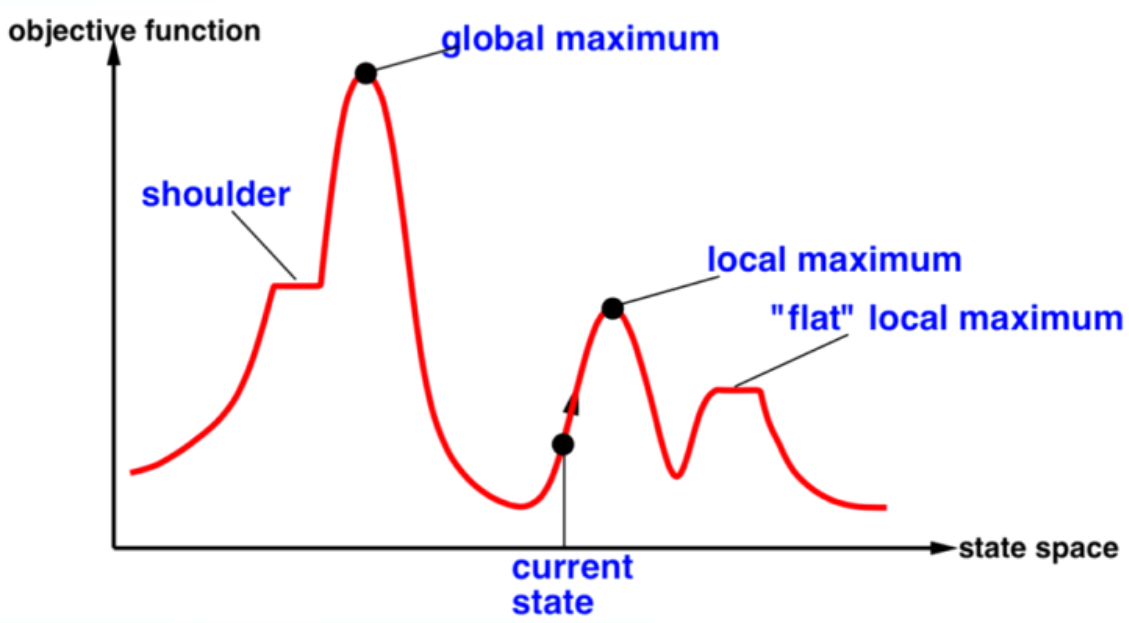
Local Search Algorithms

- Local search can be used on problems that can be formulated as finding a solution maximizing a criterion among a number of candidate solutions.
- Local search algorithms move from solution to solution in the space of candidate solutions (the search space) until a solution deemed optimal is found or a time bound is elapsed.
- For example: The travelling salesman problem, in which a solution is a cycle containing all nodes of the graph and the target is to minimize the total length of the cycle. i.e. a solution can be a cycle and the criterion to maximize is a combination of the number of nodes and the length of the cycle.
- A local search algorithm starts from a candidate solution and then iteratively moves to a neighbor solution

Local Search Algorithms

- Terminate on a time bound or if the situation is not improved after number of steps.
- Local search algorithms are typically incomplete algorithms, as the search may stop even if the best solution found by the algorithm is not optimal

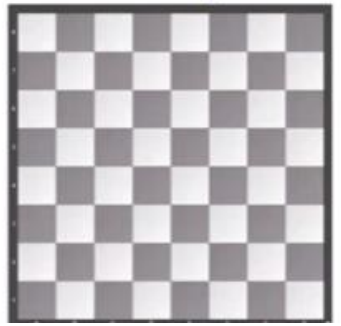
- **Local search** algorithms operate by searching from a start state to neighboring states,
- without keeping track of the paths, nor the set of states that have been reached.
- They are not systematic—
- they might never explore a portion of the search space where a solution actually resides.
- They searches only **the final state**

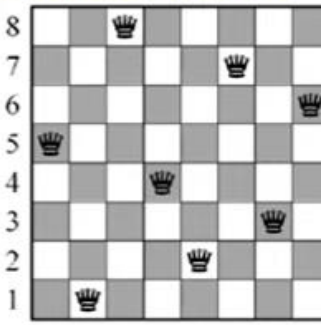


Local Search Algorithm

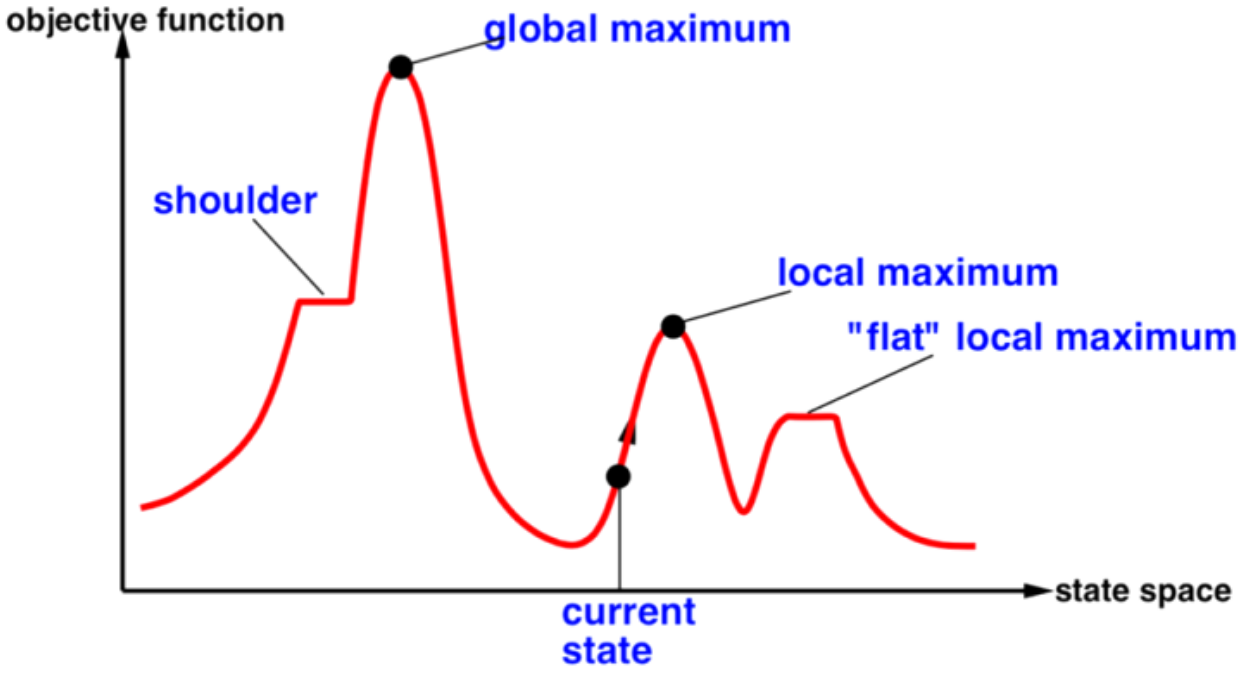
- Its always useful in real time environment.
- Suitable in “continuous environment”

- The Local search algorithm searches only the final state, not the path to get there.
- For example, in the 8-queens problem,
- we care only about finding a valid final configuration of 8 queens (8 queens arranged on chess board, and no queen can attack other queens) and not the path from initial state to final state.
-

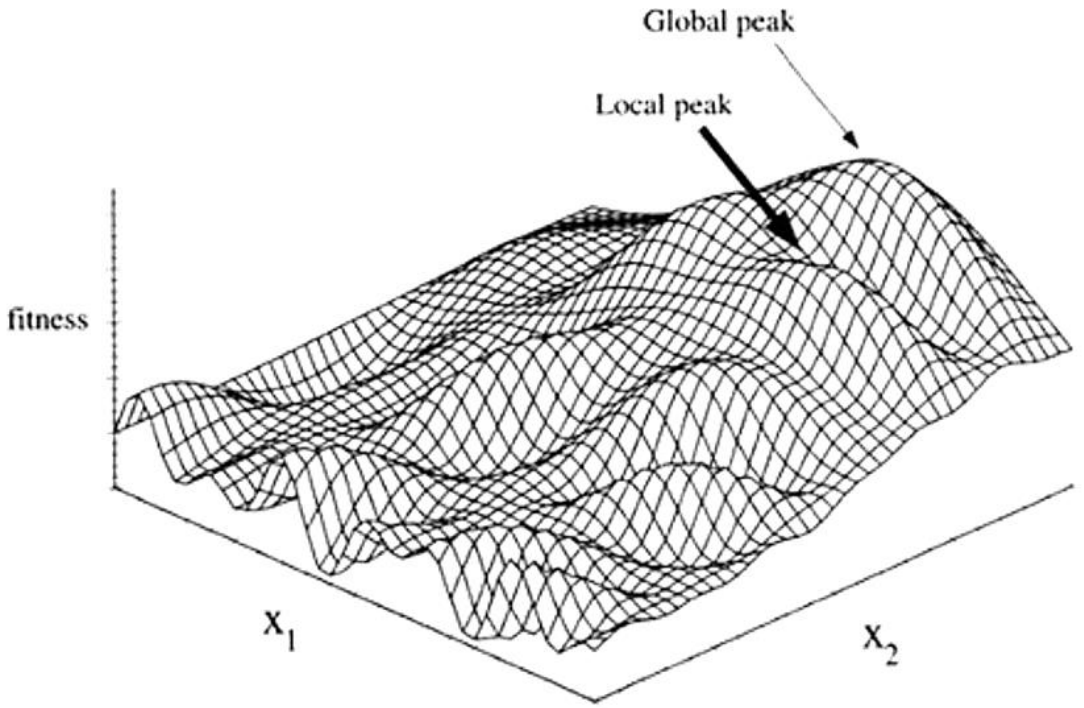




Search Landscape (Two & Three Dimensions)



Two Dimension: A dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow.



Three Dimension

Summary - Local Search And Optimization

- **Local search**
  - Keep track of single current state – Move only to neighboring states – Ignore paths
- **Advantages:**
  - Use very little memory
  - Can often find reasonable solutions in large or infinite (continuous) state spaces.
- **"Pure optimization" problems**
  - All states have an objective function
  - Goal is to find state with max (or min) objective value
  - Does not quite fit into path-cost/goal-state formulation
  - Local search can do quite well on these problems.

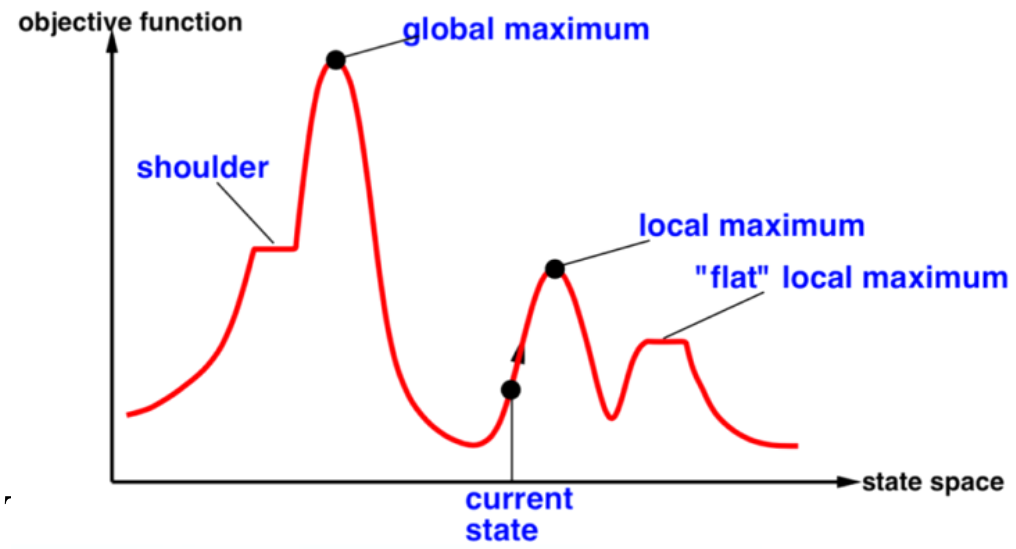
Applications of Local Search Algorithm

- Integrated-circuit design,
- Factory floor layout,
- Job shop scheduling,
- Automatic programming,
- Telecommunications network optimization,
- Crop planning, and
- Portfolio management.

Local Search Algorithms

- Hill Climbing
- Local Beam Search Algorithm
- Evolutionary Algorithm – Genetic Algorithm
- Simulated Annealing

Hill- Climbing Search (Heuristic Search)



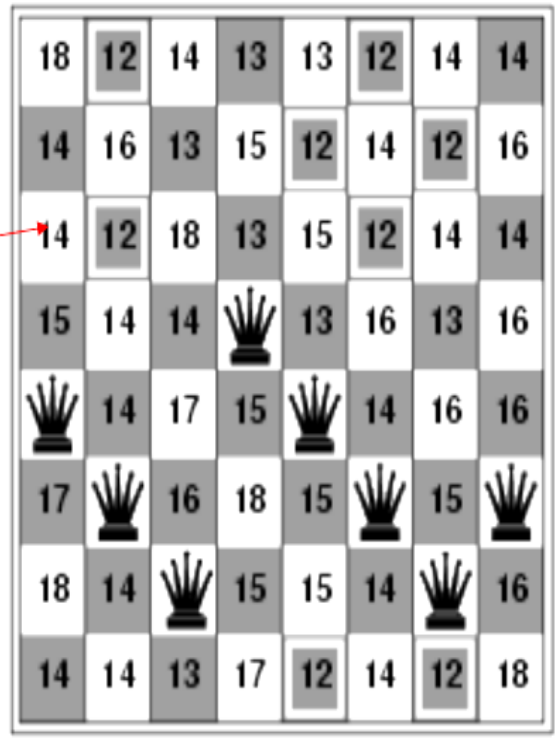
- Continually moves in the direction of increasing value (i.e., uphill). T of the mountain or best solution to the problem.
- Terminates when it reaches a **"peak",** no neighbor has a higher value. It keeps track of one current state and on each iteration moves to the neighboring state with highest value (i.e. It heads in the direction of steepest ascent)
- Only records the state and its objective function value.
- Does not look ahead beyond the immediate.
- Sometimes called Greedy Local Search
- **Problem:** Can get stuck in local maxima,
- Its success depends very much on the shape of the state-space land-scape: if there are few local maxima, random-restart hill climbing will find a "good" solution very quickly

Hill- Climbing Search

```
function HILL-CLIMBING(problem) returns a state that is a local maximum

    current ← MAKE-NODE(problem.INITIAL-STATE)
    loop do
        neighbor ← a highest-valued successor of current
        if neighbor.VALUE ≤ current.VALUE then return current.STATE
        current ← neighbor
```

The hill-climbing search algorithm, which is the **most basic** local search technique. At each step the current node is replaced by the best neighbor; in this version, that means the neighbor with the highest VALUE, but if a heuristic cost estimate h is used, we would find the neighbor with the lowest h.

Example: 8-queens



Each number indicates h if we move a queen in its corresponding column

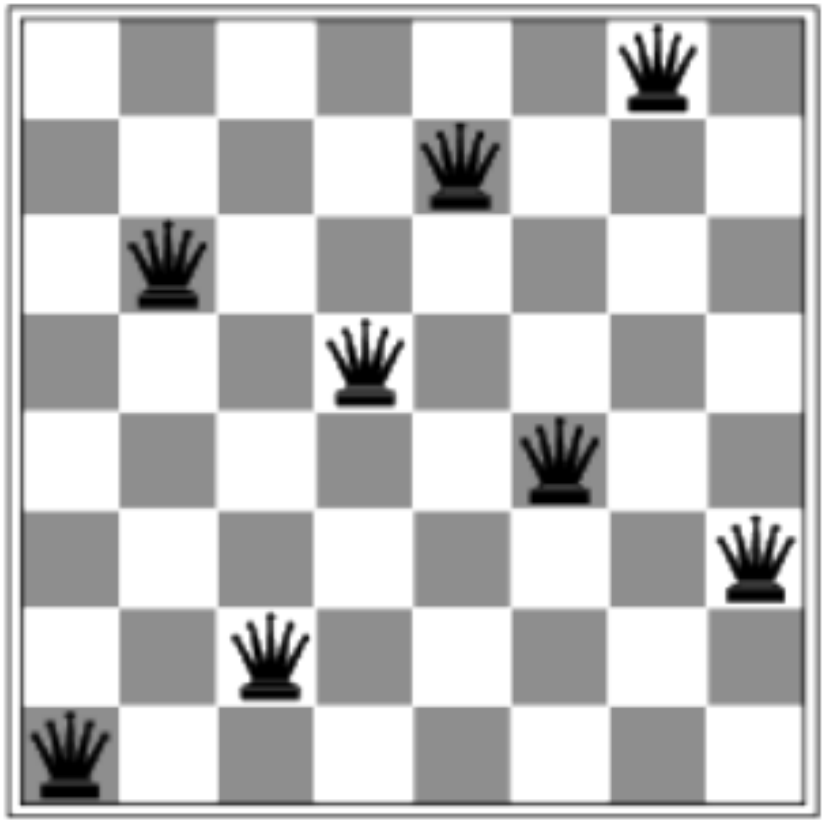
Figure 4.3 (a) An 8-queens state with heuristic cost estimate h = 17, showing the value of h for each possible successor obtained by moving a queen within its column. The best moves are marked.

h = number of pairs of queens that are attacking each other, either directly or indirectly (*h* = 17 for the above state)

- To illustrate hill climbing, we will use the 8-queens problem. Local search algorithms typically use a complete-state formulation, where each state has 8 queens on the board, one per column.
- The successors of a state are all possible states generated by moving a single queen to another square in the **same column** (so each state has 8 × 7 = 56 successors).
- The heuristic cost function **"h"** is the number of pairs of queens that are attacking each other, either directly or indirectly.
- The global minimum of this function is zero, which occurs only at perfect solutions.
- Figure 4.3(a) shows a state with h = 17. The figure also shows the values of all its successors, with the best successors having h = 12. Hill-climbing algorithms typically choose randomly among the set of best successors if there is more than one.

Example: n-queens

- **Figure 4.3 (b)** A local minimum in the 8-queens state space;
- The state has h = 1 but every successor has a higher cost

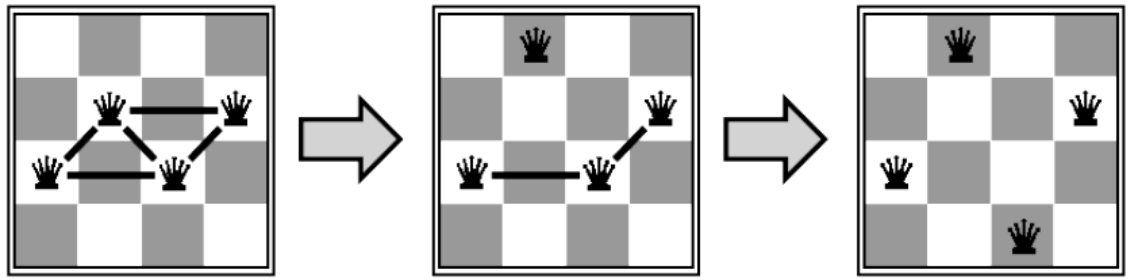


A local minimum with *h* = *1*

Example: n-queens

Put *n* queens on an *n*×*n* board with no two queens on the same row, column, or diagonal.

Move a queen to reduce number of conflicts.



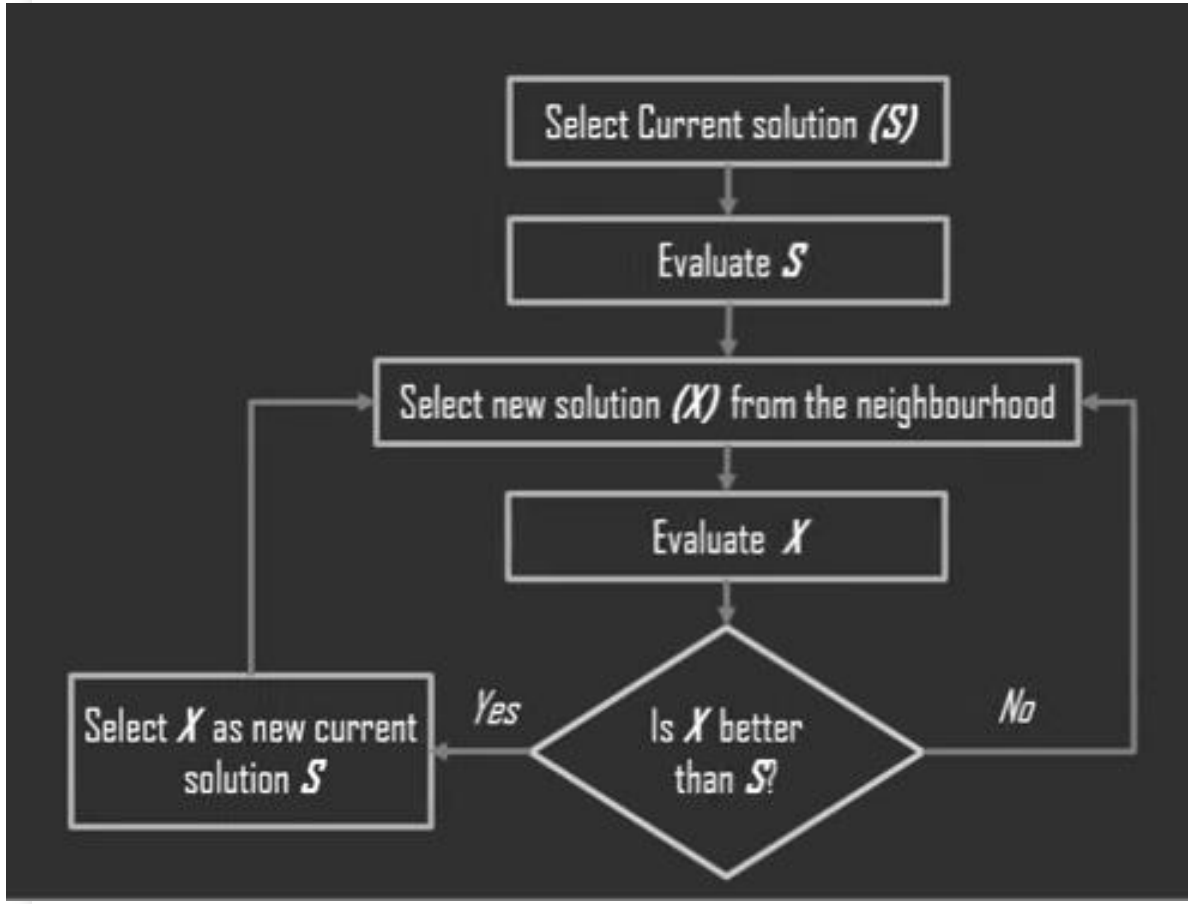
- Hill climbing is sometimes called greedy local search because it grabs a good neighbor state without thinking ahead about where to go next.
- Although greed is considered one of the seven deadly sins, it turns out that greedy algorithms often perform quite well.
- Hill climbing often makes rapid progress toward a solution because it is usually quite easy to improve a bad state.
- For example, from the state in Figure 4.3(a), it takes just five steps to reach the state in Figure 4.3(b), which has h = 1 and is very nearly a solution.

Flowchart: Hill Climbing

- It is variant of "Generate & Test Algorithm"
- With addition of – Greedy Approach

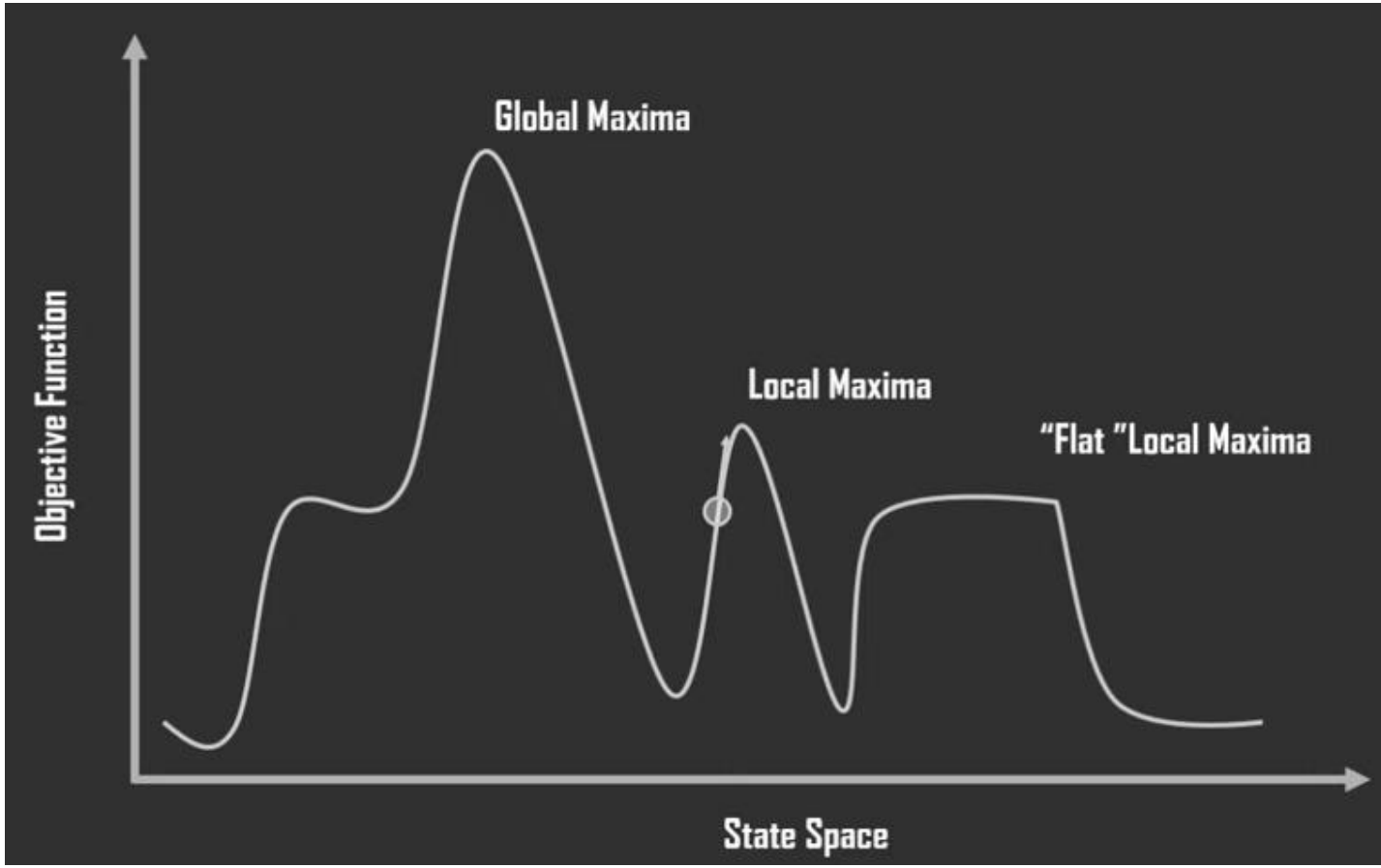
Algorithm

1. Evaluate the initial state.
2. Loop until a solution is found or there are no new operators left to be applied;
   - Select and apply a new operator
   - Evaluate the new state;
     - ✓ **better than current state > new current state**
     - ✓ **Not better > try new operator**

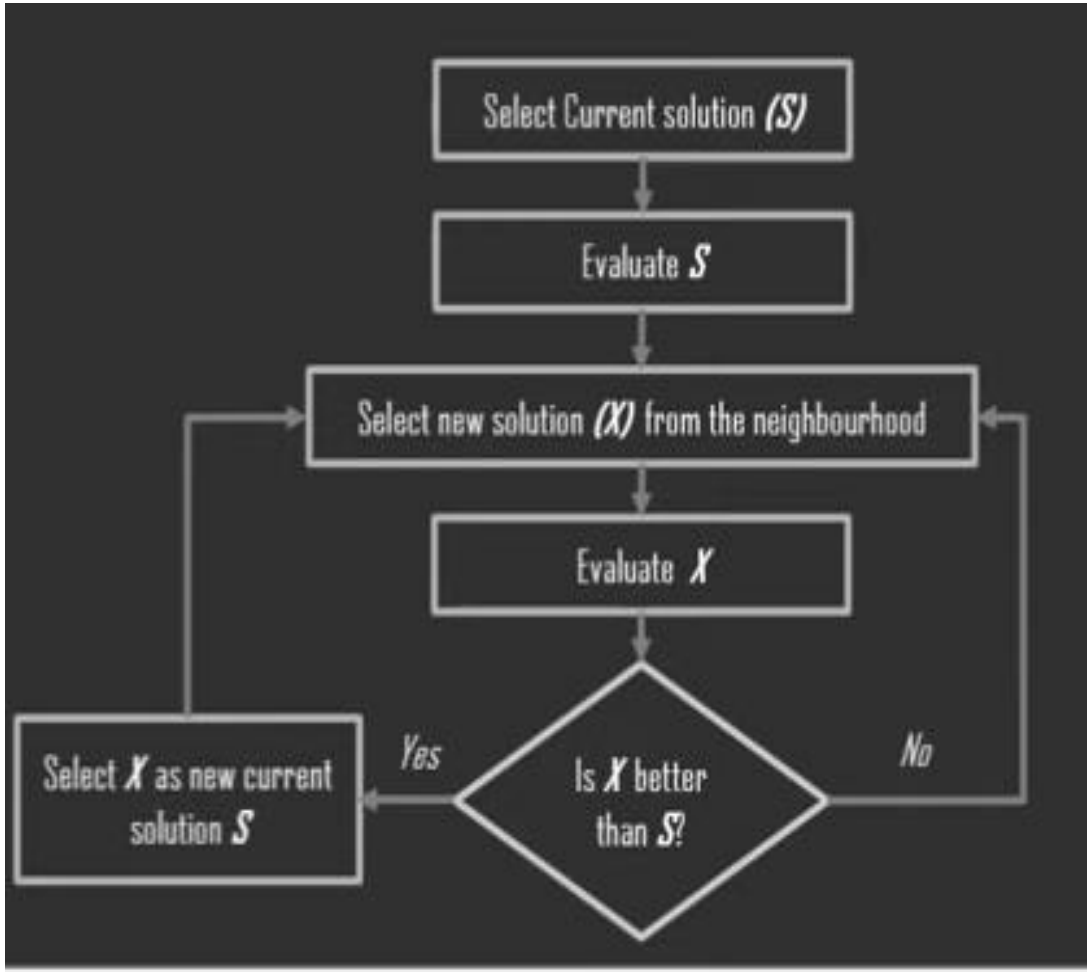


Search Space - Diagram

- Search Space Diagram is having so many regions.
- It is graphical representation of set of states that our search algorithm can reach V/S the value of objective function.
- The objective function - wish to maximize or minimize.
- X- Axis = State/Configuration of algorithm
- Y-Axis = Values of objective function corresponding to specific state
- Solution : The state which has maximum objective function value/Global maximum

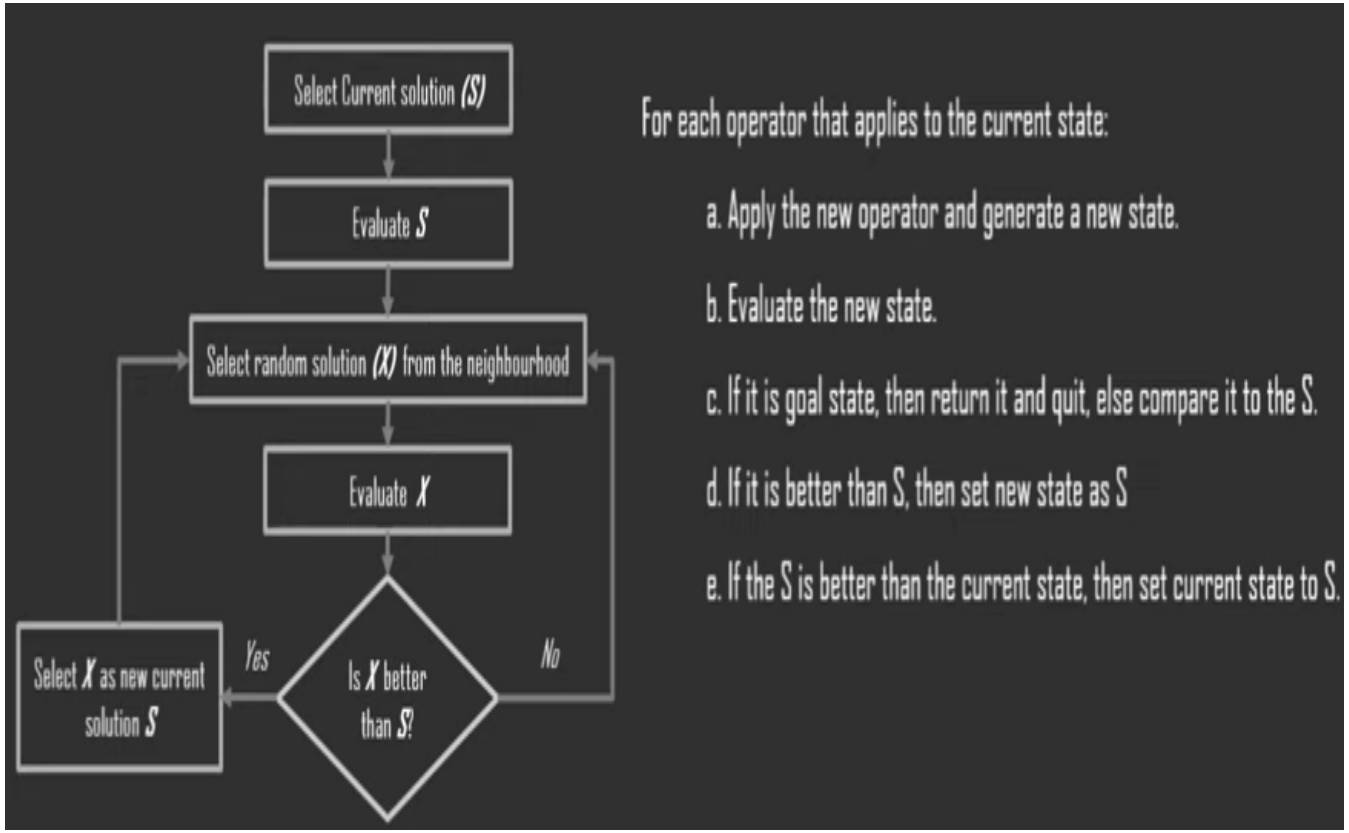


Types of Hill Climbing: 1)Simple Hill Climbing



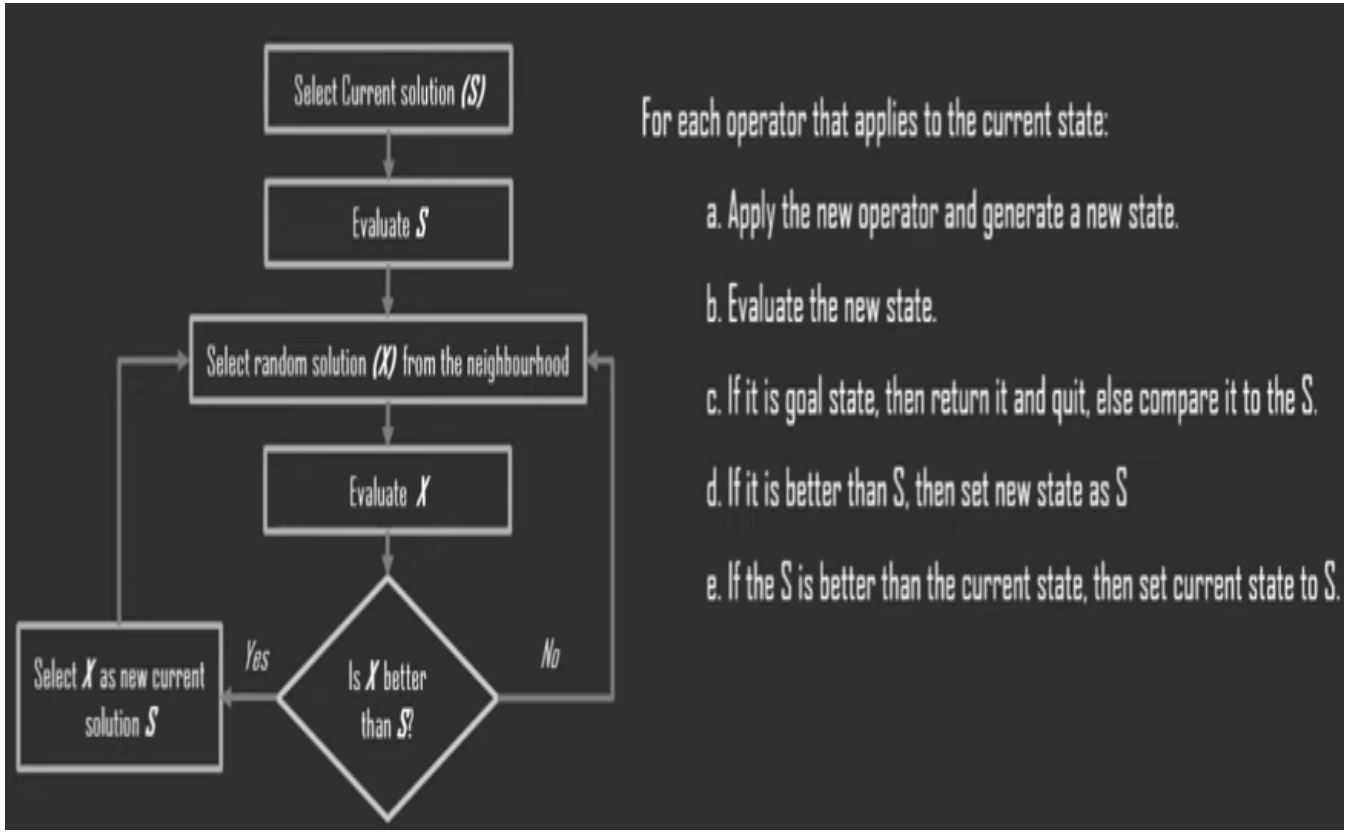
- It is Simplest way to implement hill climbing.
- It only evaluates the neighbor node state at a time.
- Select the first one which optimizes current cost & set it as a current state.
- It checks only 1 successor state, if successor state is better then current state then it moves to next state else remains in same state.
- It is less time consuming
- It gives less optimum solution.
- Solution is not guaranteed

Types of Hill Climbing: 2)Steepest Ascent Hill Climbing



- It is "variation" of simple hill climbing algorithm.
- It examines all the neighboring nodes of current state.
- Selects 1 neighbor node which is closest to the goal state.
- It consumes more time as it searches for "multiple neighbors"
- It gives less optimum solution.
- Solution is not guaranteed

Types of Hill Climbing: 2) Stochastic Hill Climbing



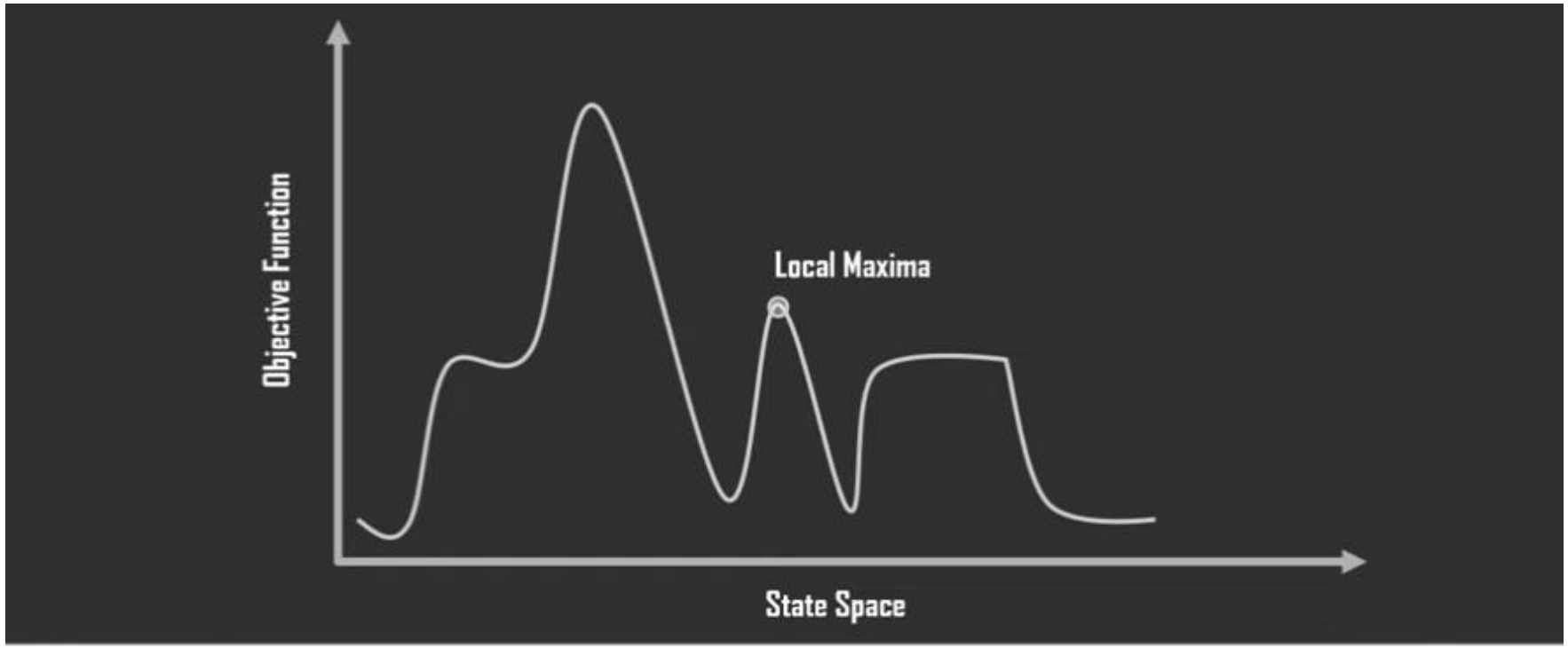
- It doesn't examine all its neighbor before moving.
- It search algorithm - selects one neighbor node – at random.
- Based on that it decides to choose it as current state or examine another state.
- It consumes more time
- Better Solution is guaranteed

Understanding- Hill Climbing

- It gets rids of “Population” & “Crossover”
- It focuses on ease of implementation
- It has faster iterations compared to traditional genetic algorithm
- It is less thorough.

Complexities: 1) Local Maxima

Local maxima: a local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upward toward the peak but will then be stuck with nowhere else to go

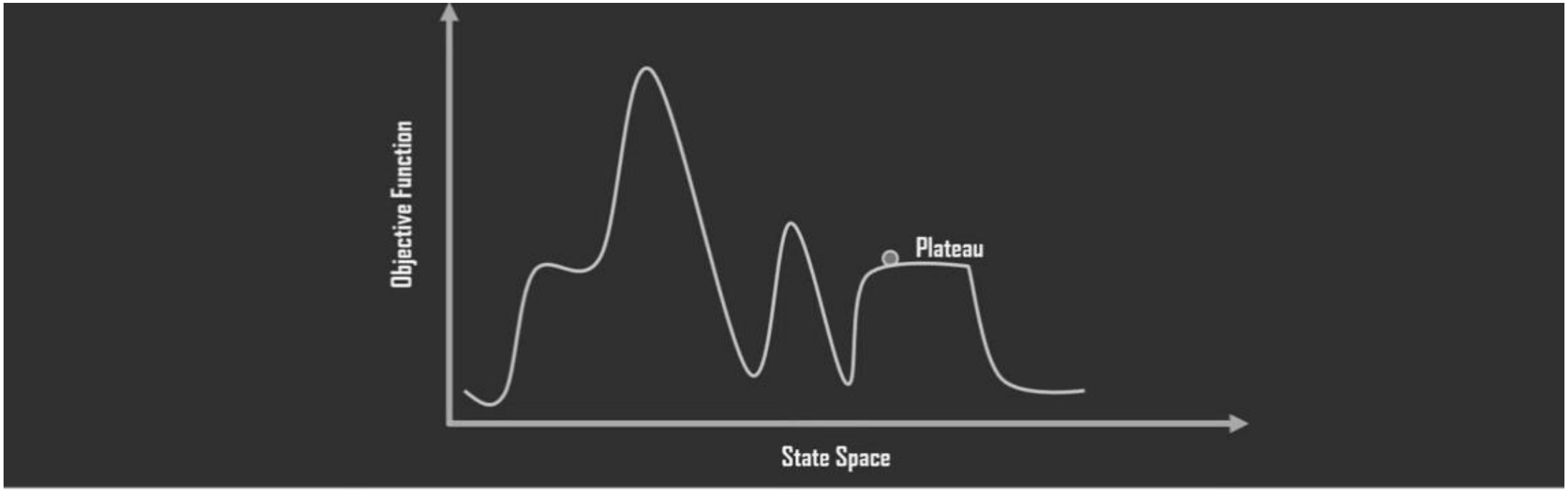


At a local maxima, the process will end even though a better solution may exist.

Utilize backtracking technique to deal with this situation.

2)Reaching Plateau Region

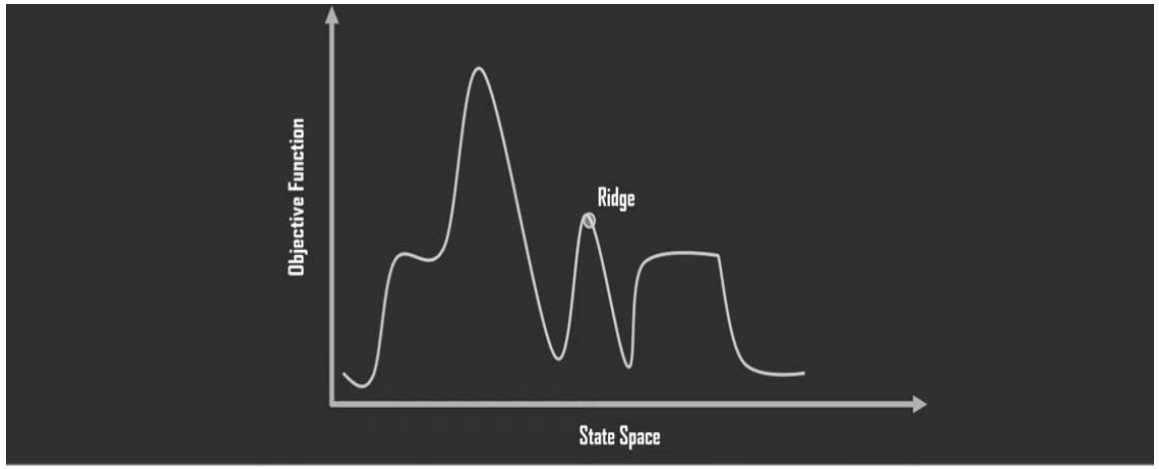
Plateaux: **A** plateau is a flat area of the state-space landscape. It can be a flat local maximum, from which no uphill exit exists, or a shoulder, from which progress is possible.



On plateau all neighbors have same value . Hence, it is not possible to select the best direction.

So, Make a big jump. Randomly select a state far away from the current state. Chances are that we will land at a non-plateau region

3) Ridge

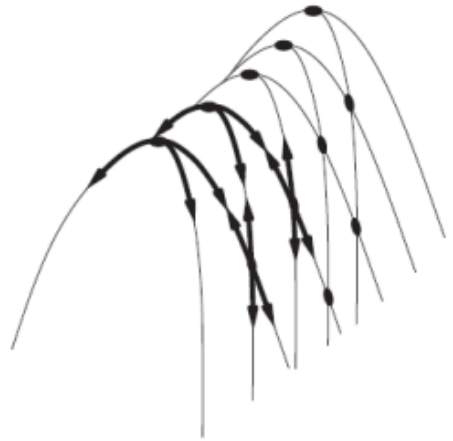


A ridge can look like a peak; hence, algorithm can end.

In this kind of obstacle, use two or more rules before testing. It implies moving in several directions at once.

Difficulties with ridges

The "ridge" creates a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill.

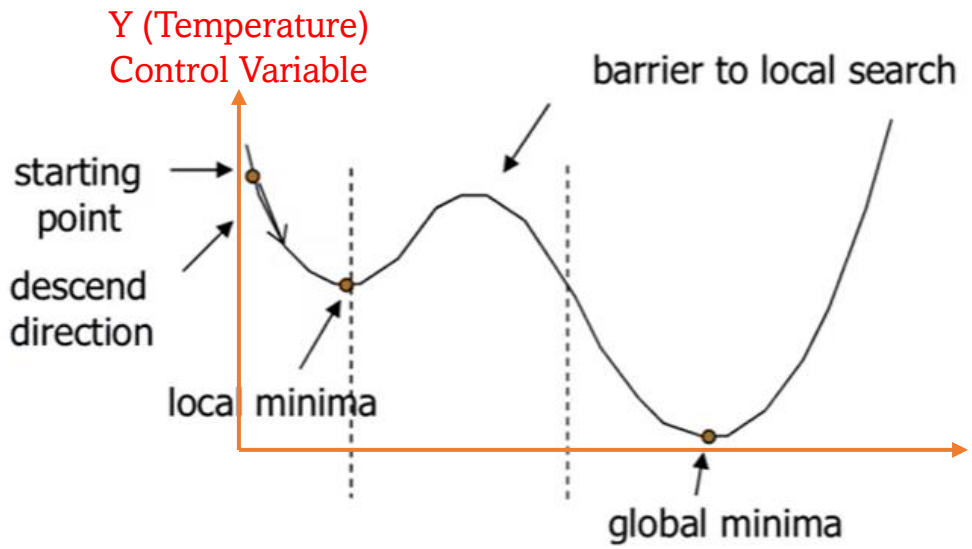


Applications:

Hill Climbing technique can be used to solve many problems, where the current state allows for an accurate evaluation functions, inductive learning methods, robotic coordination problems, etc.

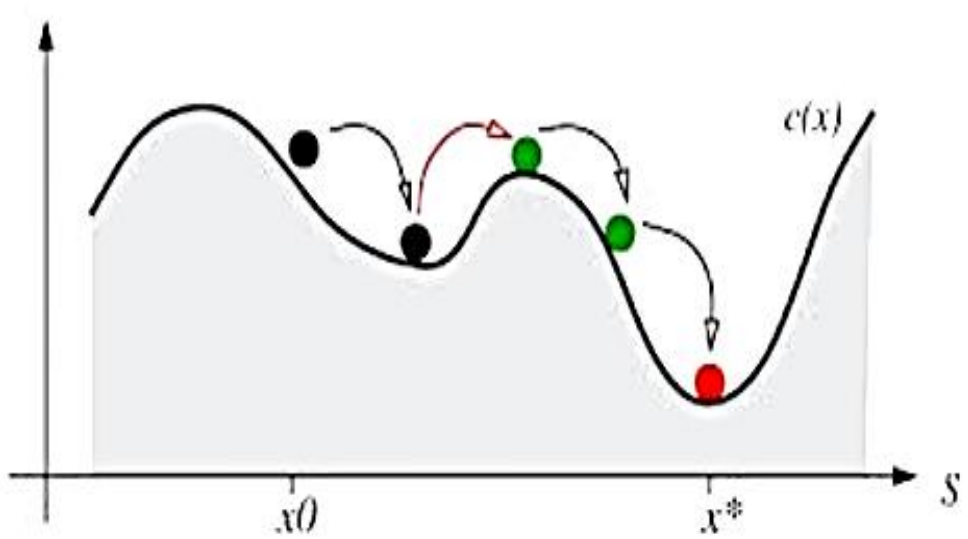
Simulated Annealing

- **Simulated Annealing** is a stochastic global search optimization algorithm and it is modified version of stochastic hill climbing.
- This algorithm appropriate for nonlinear objective functions where other local search algorithms do not operate well.
- **The simulated-annealing solution is to start by shaking hard (i.e., at a high temperature) and**
- **then gradually reduce the intensity of the shaking (i.e., lower the temperature).**
- Simulated Annealing (SA) is very useful for situations where there are a lot of local minima.



Simulated Annealing Search

- To avoid being stuck in a local maxima, it tries randomly (using a probability function) to move to another state, if this new state is better it moves into it, otherwise try another move… and so on.
- Terminates when finding an acceptably good solution in a fixed amount of time, rather than the best possible solution.
- Locating a good approximation to the global minimum of a given function in a large search space.
- Widely used in VLSI layout, airline scheduling, etc.



Properties of Simulated Annealing Search

- The problem with this approach is that the neighbors of a state are not guaranteed to contain any of the existing better solutions which means that failure to find a better solution among them does not guarantee that no better solution exists.
- It will not get stuck to a local optimum.
- If it runs for an infinite amount of time, the global optimum will be found

Resource Used: Mustafa Jarrar: Lecture Notes on Artificial Intelligence Birzeit University, 2020

Simulated Annealing

- Idea: escape local maxima by allowing some "bad" moves bu gradually decrease their size and frequency.
- Devised by Metropolis et al., 1953, for physical process modelling.
- At fixed "temperature" T, state occupation probability reaches Boltzman distribution

$$p(x) = \alpha e^{\frac{E(x)}{kT}}$$

- When T is decreased slowly enough it always reaches the best state x^* because $e^{\frac{E(x^*)}{kT}} / e^{\frac{E(x)}{kT}} = e^{\frac{E(x^*)-E(x)}{kT}} \gg 1$ for small T. (Is this necessarily an interesting guarantee?)
- Widely used in VLSI layout, airline scheduling, etc.

function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state
  inputs: *problem*, a problem
    schedule, a mapping from time to "temperature"

  current ← MAKE-NODE(*problem*.INITIAL-STATE)
  for $t = 1$ **to** ∞ **do**
    $T \leftarrow schedule(t)$
    if $T = 0$ **then return** *current*
    next ← a randomly selected successor of *current*
    $\Delta E \leftarrow next$.VALUE – *current*.VALUE
    if $\Delta E > 0$ **then** *current* ← *next*
    else *current* ← *next* only with probability $e^{\Delta E/T}$

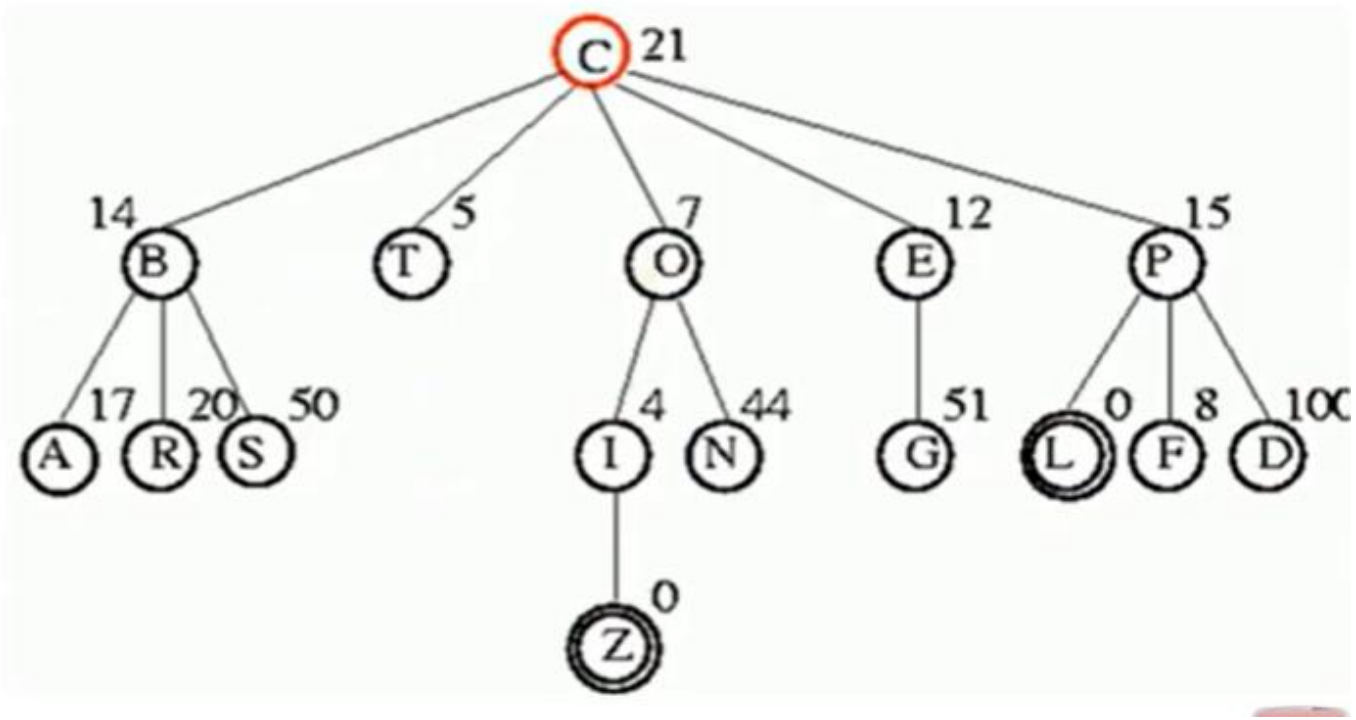
- **Figure 4.5** The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. Downhill moves are accepted readily early in the annealing schedule and then less often as time goes on. The schedule input determines the value of the temperature T as a function of time

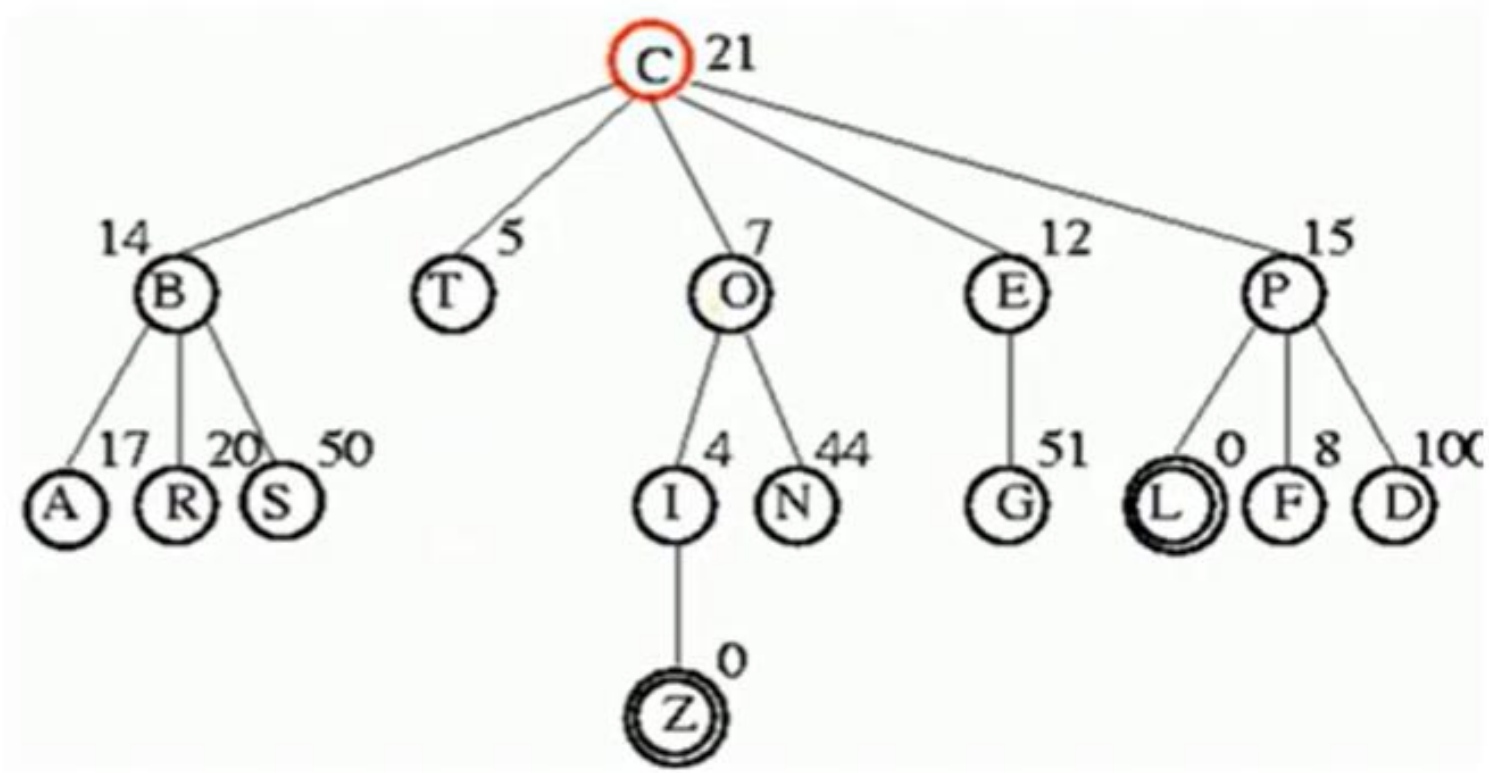
Local Beam Search

- Idea: keep k states instead of 1; choose top k of all their successors
- Not the same as k searches run in parallel! Searches that find good states recruit other searches to join them.
- Problem: quite often, all k states end up on same local hill.
- To improve: choose k successors randomly, biased towards good ones.
- Observe the close analogy to natural selection

Example :

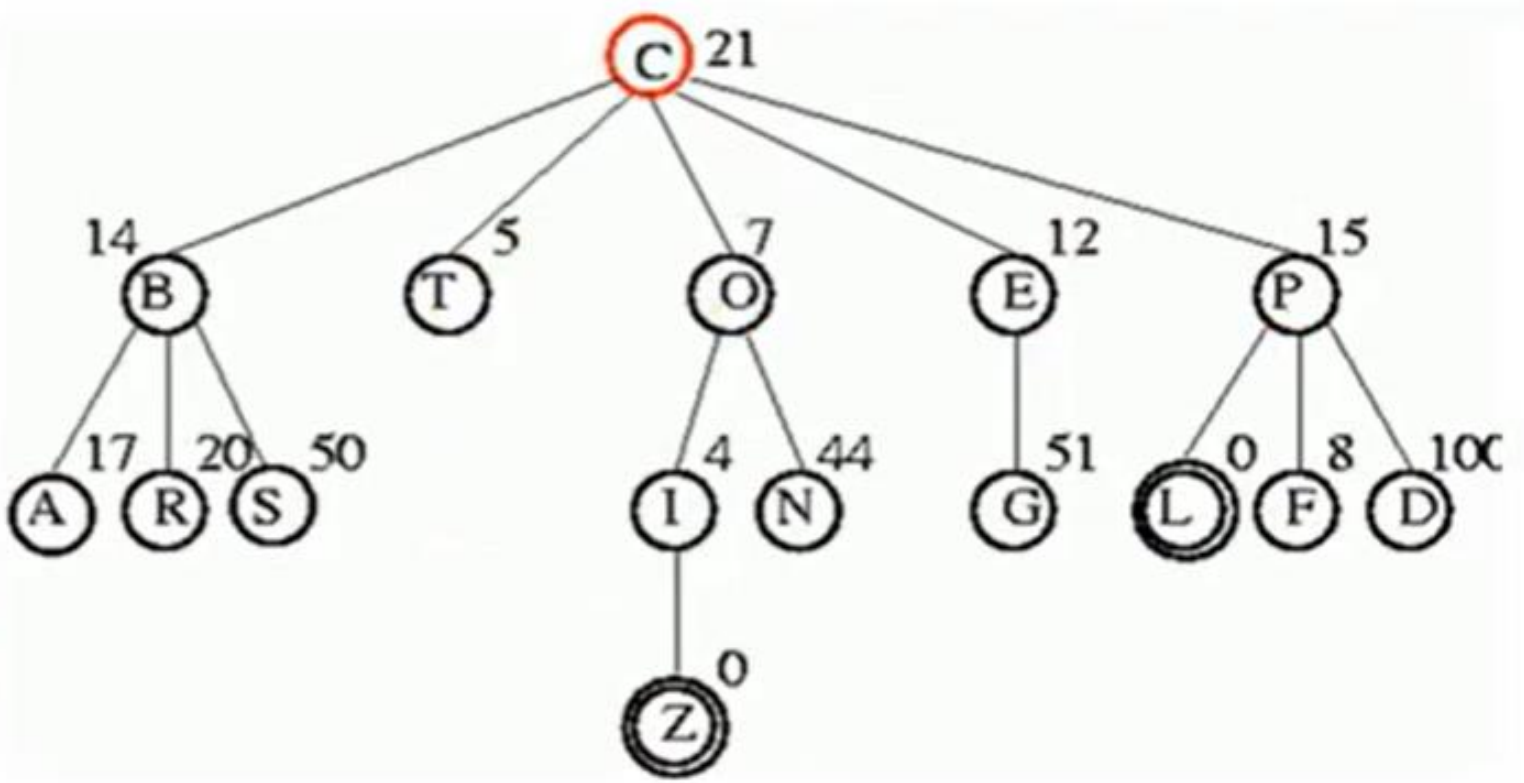
- Start State: C
- Goal State: Z and L
- n = 2 (beam size)
- Iteration 1
- Open List = {c}



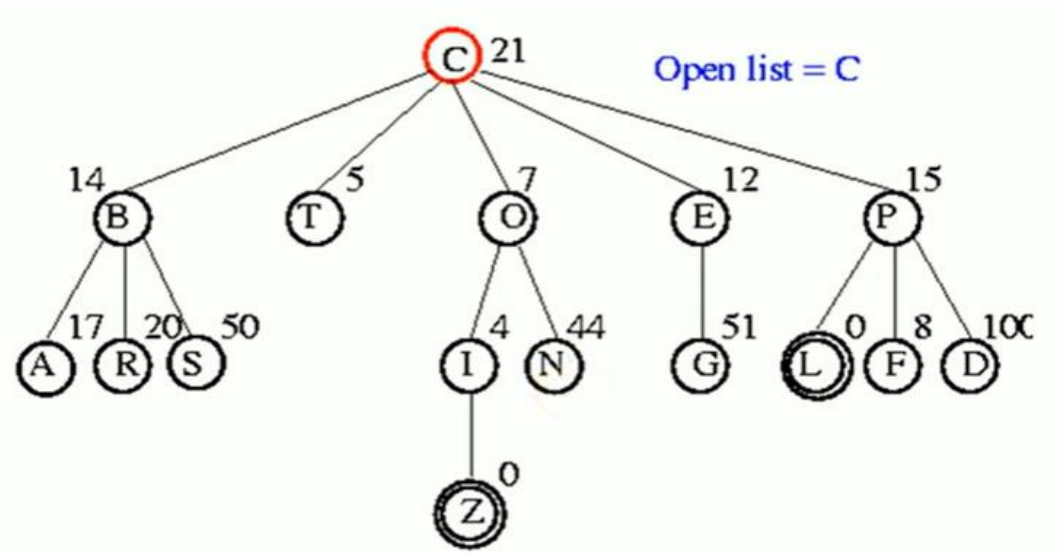


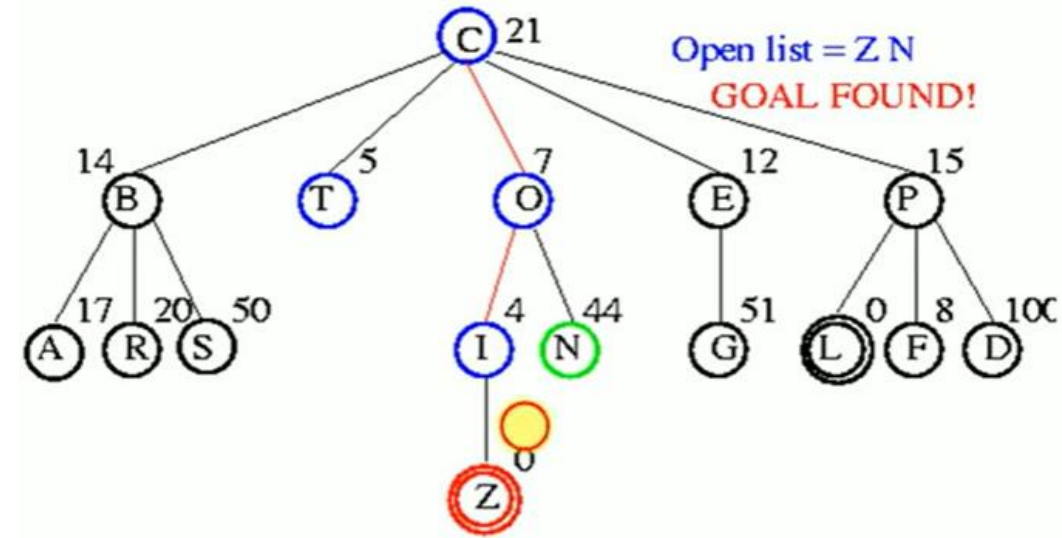
- Iteration 2
- Find successor of C
- = B T O E P
- Remove C from list, now
- Open list = {T,O}
- Iteration 3
- T has no successors, remove T from open list
- Find successor of O, replace O with I and N in open list, then
- Open list = {I, N}

- Open list = {I, N}
- Iteration 4
- Find successor of I
- Z is Goal



Path : C – O – I – Z





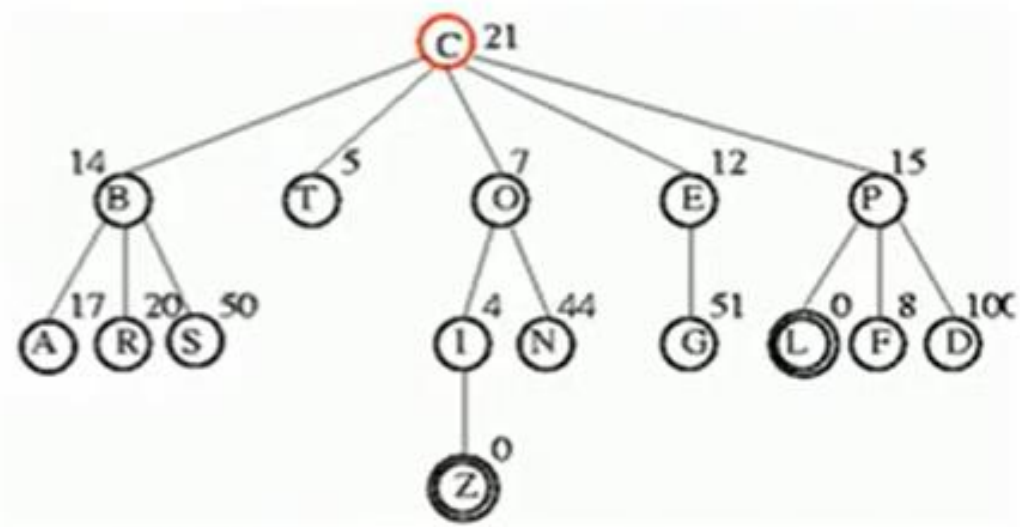
Steps:

- Consider lowest successor (having low estimated value).
- Here T Is considered and Explored.
- Since T cant be explored further (T need to be removed from open lost)
- Open List
- C
- T,O (remove T)
- I N (Remove O)
- Z, N (Explore I)
- Keep only 2 nodes in open list (means value of n is 2)

- Beam search is “intermediate Algorithm” (Between Hill climbing & Best First Search)
- In case of Hill climbing – It wont be able to reach goal state as per this problem. Because it will reach T and Stops there.

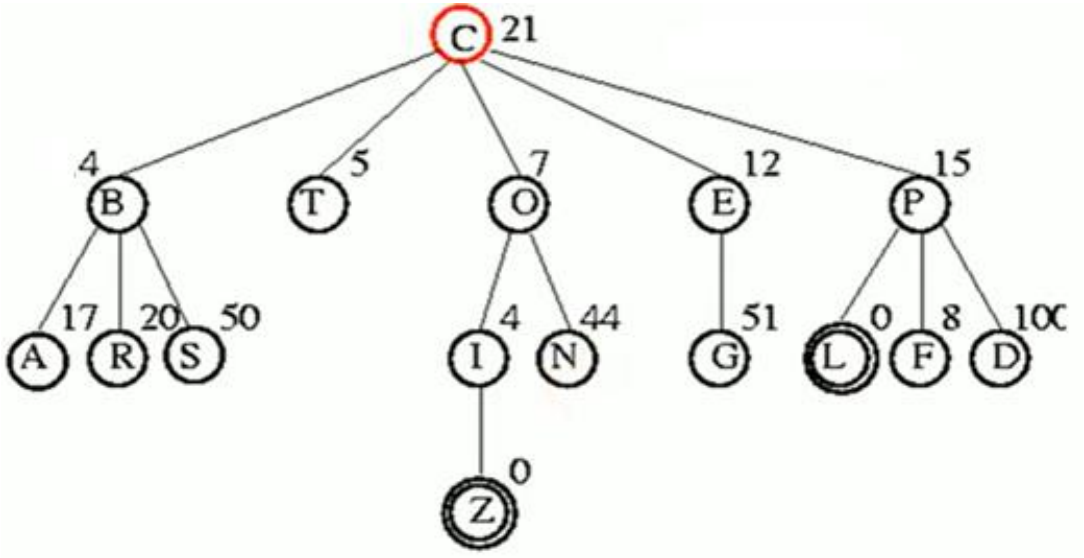
- In Case of BFS:
- Exploring space is Higher (AS it keeps all its successors in open list)

Beam Search Algorithm...



- Beam search algorithm is not complete
- It is not optimal
- The time complexity: The worst-case time = ***O(B*m)***
- The space complexity: The worst-case space complexity = ***O(B*m)***
- ***B*** is the beam width, and m is the maximum depth of any path in the search tree.

Example : Incomplete Solution



- Beam Search is not complete .
  - In this example initially we will select node B & T.
  - T is ignored.
  - B is explored (but we wont reach goal state)

Genetic Algorithm

- Inspired by evolutionary biology and natural selection, such as inheritance. • Evolves toward better solutions. • A successor state is generated by combining two parent states, rather by modifying a single state. • Start with k randomly generated states (population), Each state is an individual
- A state is represented as a string over a finite alphabet (often a string of 0s and 1s) • Evaluation function (fitness function). Higher values for better states. • Produce the next generation of states by selection, crossover, and mutation. • Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population

Genetic Algorithm

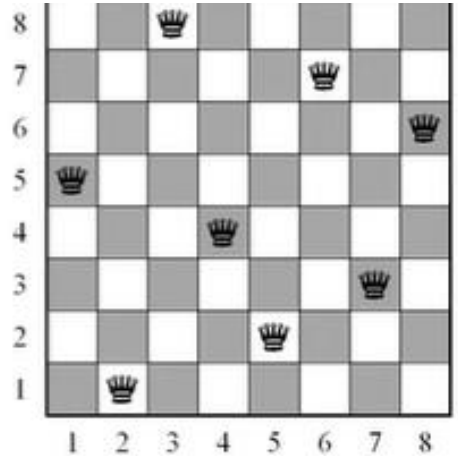
```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
    new_population ← empty set
    for i = 1 to SIZE(population) do
      x ← RANDOM-SELECTION(population, FITNESS-FN)
      y ← RANDOM-SELECTION(population, FITNESS-FN)
      child ← REPRODUCE(x, y)
      if (small random probability) then child ← MUTATE(child)
      add child to new_population
    population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN

function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

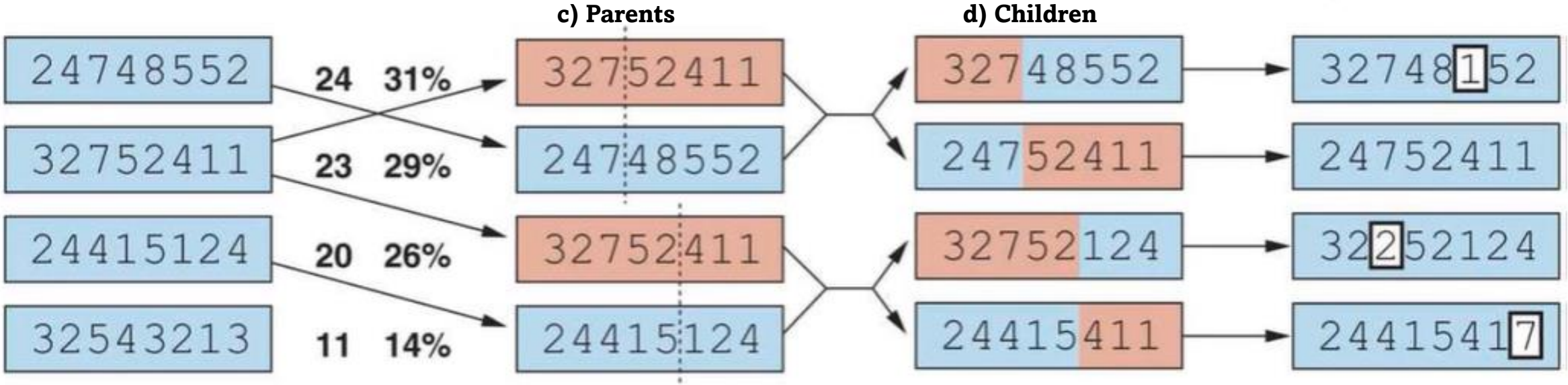
Figure 4.8 A genetic algorithm. The algorithm is the same as the one diagrammed in Figure 4.6, with one variation: in this more popular version, each mating of two parents produces only one offspring, not two.

- Idea: stochastic local beam search + generate successors from pairs of states
- GAs require states encoded as strings.
- Crossover helps iff substrings are meaningful components.
- GAs 6= evolution.
  - e.g., real genes encode replication machinery.

Genetic algorithm for 8 Queens problem

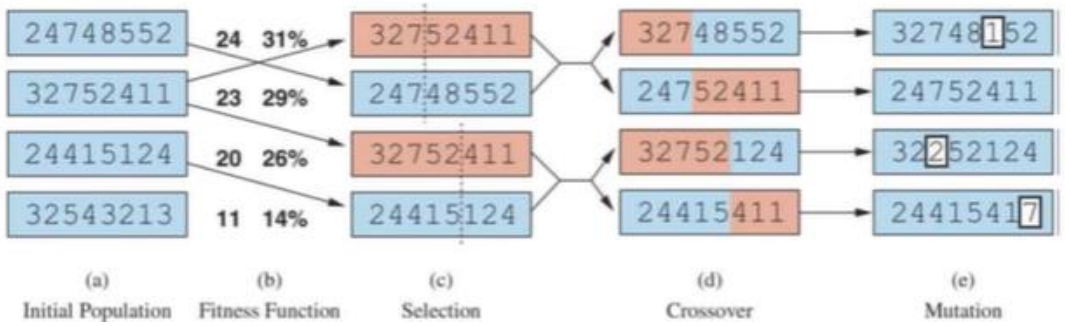
- A genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by a fitness function in (b) resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).





a) Initial Population b) Fitness Function c) Selection d) Crossover e) Mutation

- The 8-queens states corresponding to the first two parents, and the first offspring,
- The green columns are lost in the crossover step and the red columns are retained. row 1 is the bottom row, and 8 is the top row.



= Result
(New population)

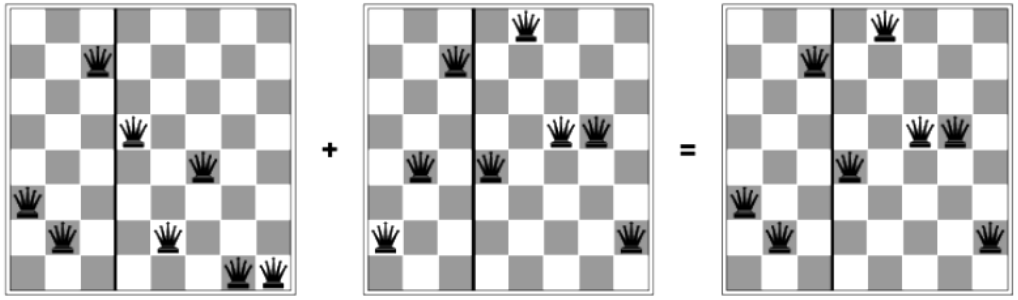
Result

- Consider this “Result” as “New Population”.
- For “New Population” perform 5 steps procedure.
- Iteration- Continues until “Goal State ”is reached.
- Goal state = Placement of all queen in chess board (No queen should attack other queen in single movement).

8 –Queens Problem Solution : In Detail

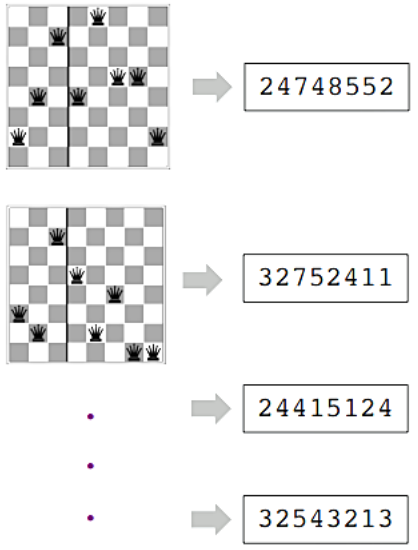
Try to better position the queens using the genetic algorithm.
A better state is generated by combining two parent states.

The good genes (features) of the parents are passed onto the children.



Represent individuals (chromosomes) :

Can be represented by a string digits 1 to 8, that represents the position of the 8 queens in the 8 columns.

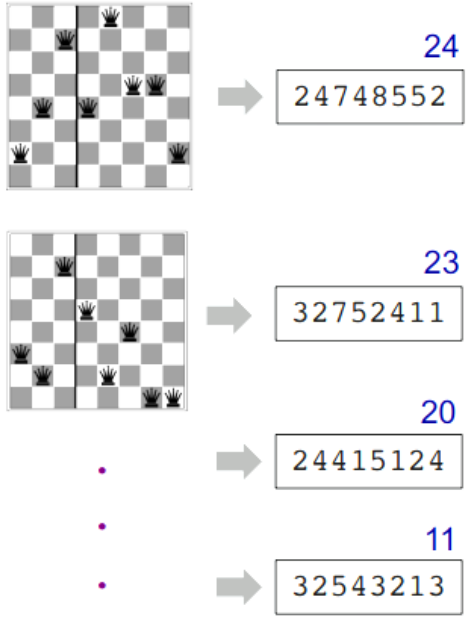


Step 1: Represent Individuals (Chromosomes)

8 –Queens Problem Solution :Step 2

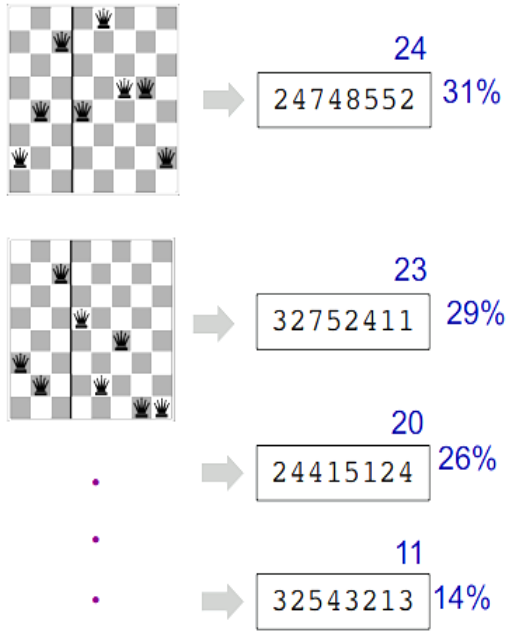
Fitness Function :

Possible fitness function is the number of non-attacking pairs of queens. (min = 0, max = 8 × 7/2 = 28)



Fitness Function :

Calculate the probability of being regenerated in next generation. For example: 24/(24+23+20+11) = 31% , 23/(24+23+20+11) = 29% , etc.

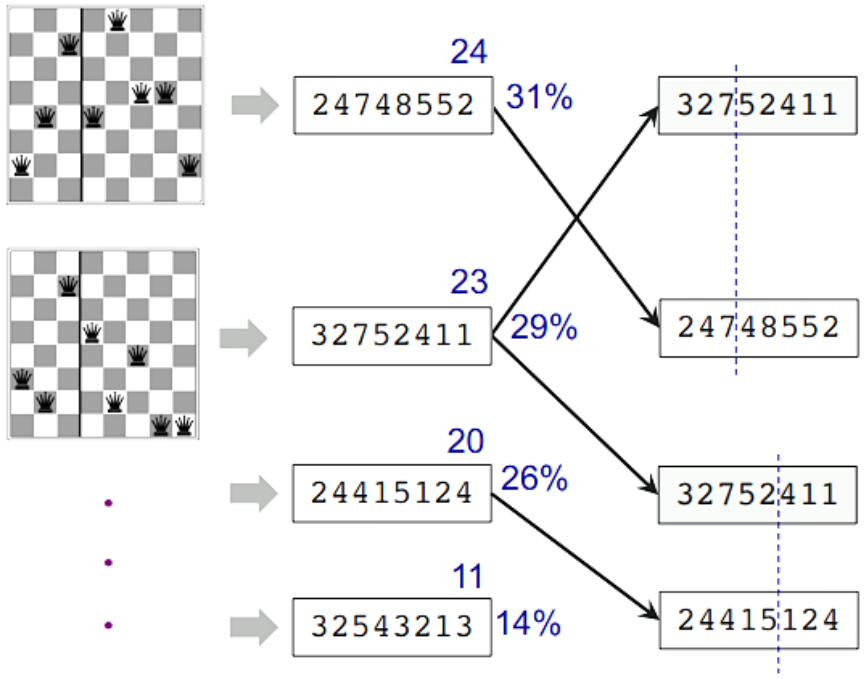


Step 2: Represent Individuals (Chromosomes)

8 –Queens Problem Solution : Step 3 & 4

Selection:

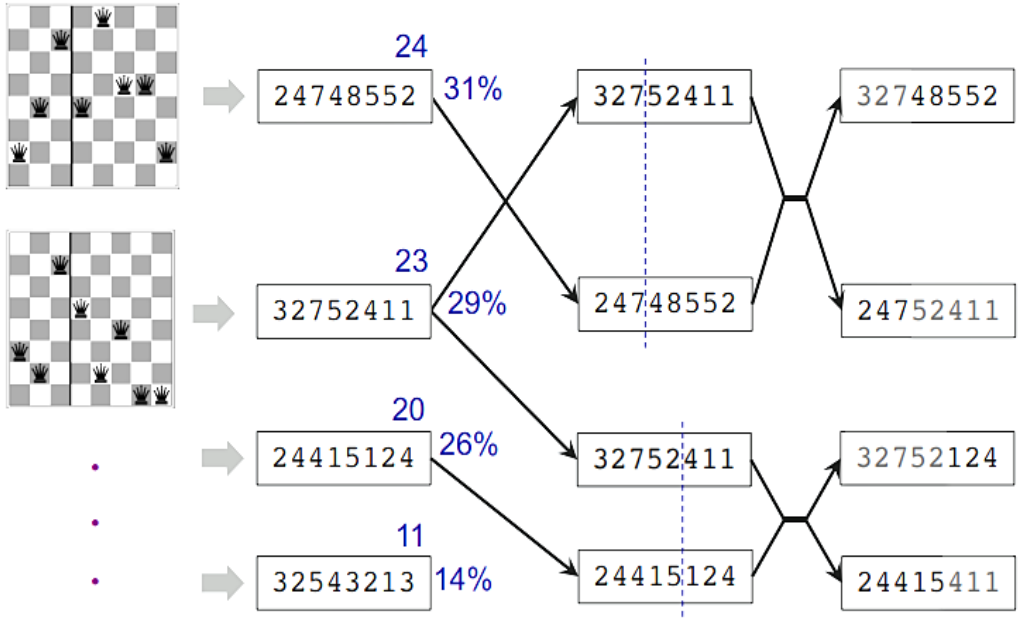
Pairs of individuals are selected at random for reproduction w.r.t. some probabilities. Pick a crossover point per pair.



Step 3: Selection

Crossover

A **crossover** point is chosen randomly in the string. **Offspring** are created by crossing the parents at the crossover point.

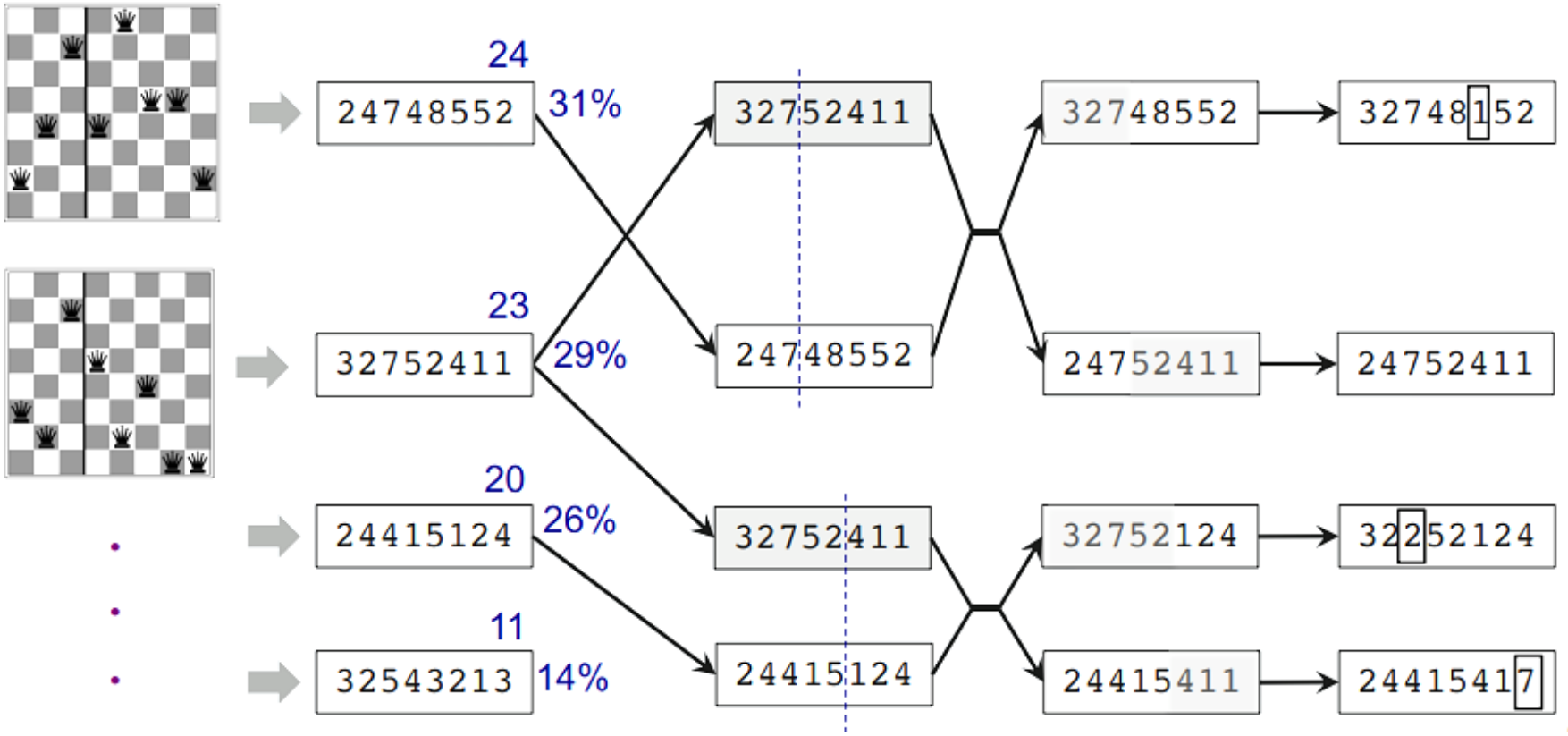


Step 4: Crossover

8 –Queens Problem Solution : Step 5

Mutation

Each element in the string is also subject to some mutation with a small probability.



Step 5: Mutation

Summary

- Hill climbing is a steady monotonous ascent to better nodes.
- Simulated annealing, local beam search, and genetic algorithms are "random" searches with a bias towards better nodes.
- All need very little space which is defined by the population size.
- None guarantees to find the globally optimal solution

References

1. S. Russell and P. Norvig: Artificial Intelligence: A Modern Approach Prentice Hall, 2003, Second Edition.
2. AIMA textbook (3rd edition)
3. AIMA slides (http://aima.cs.berkeley.edu/
4. http://en.wikipedia.org/wiki/SMA*
5. Moonis Ali: Lecture Notes on Artificial Intelligence http://cs.txstate.edu/~ma04/files/CS5346/SMA%20search.pdf
6. Max Welling: Lecture Notes on Artificial Intelligence https://www.ics.uci.edu/~welling/teaching/ICS175winter12/A-starSearch.pdf
7. Kathleen McKeown: Lecture Notes on Artificial Intelligence http://www.cs.columbia.edu/~kathy/cs4701/documents/InformedSearch-AR-print.ppt
8. Franz Kurfess: Lecture Notes on Artificial Intelligence http://users.csc.calpoly.edu/~fkurfess/Courses/Artificial-Intelligence/F09/Slides/3-Search.ppt


Thank you

Artificial Intelligence
DSE 3252

Reinforcement Learning

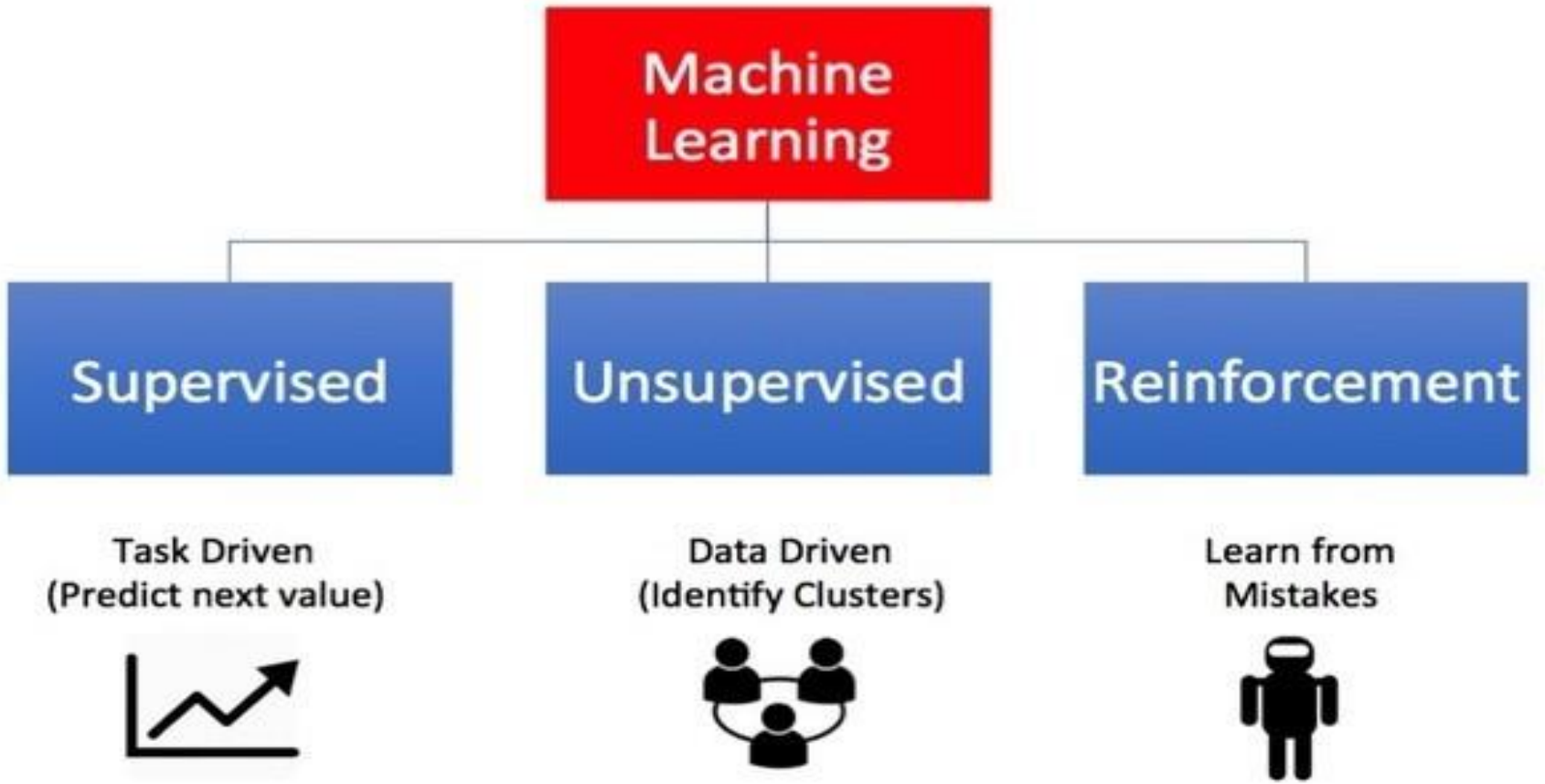
DR.ROHINI R RAO & DR.RASHMI L MALGHAN
DEPT OF DATA SCIENCE & COMPUTER APPLICATIONS

JANUARY 2024

What is Reinforcement Learning

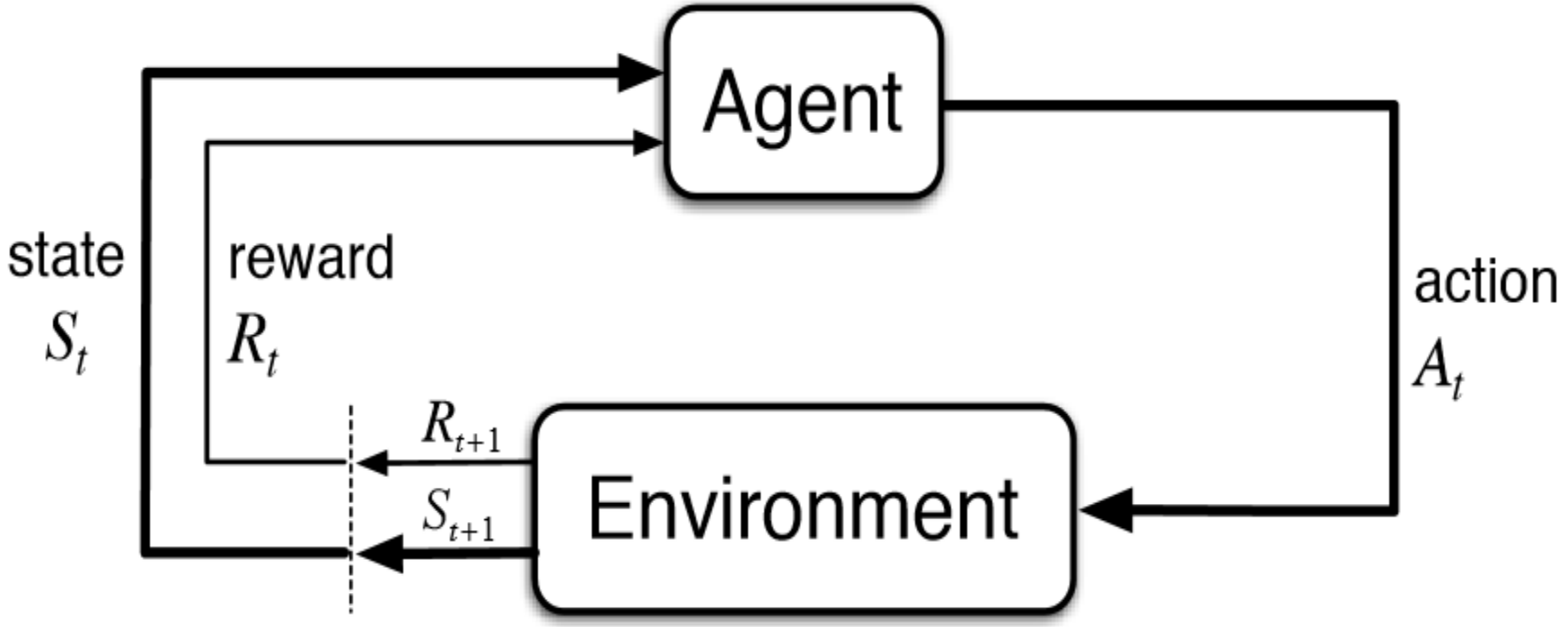
- Reinforcement Learning is a
  - Feedback-based Machine learning technique in which an agent learns to behave in an environment by performing the actions and seeing the results of actions.
  - For each good action, the agent gets positive feedback
  - for each bad action, the agent gets negative feedback or penalty.
- ***"Reinforcement learning is a type of machine learning method where an intelligent agent (computer program) interacts with the environment and learns to act within that."***

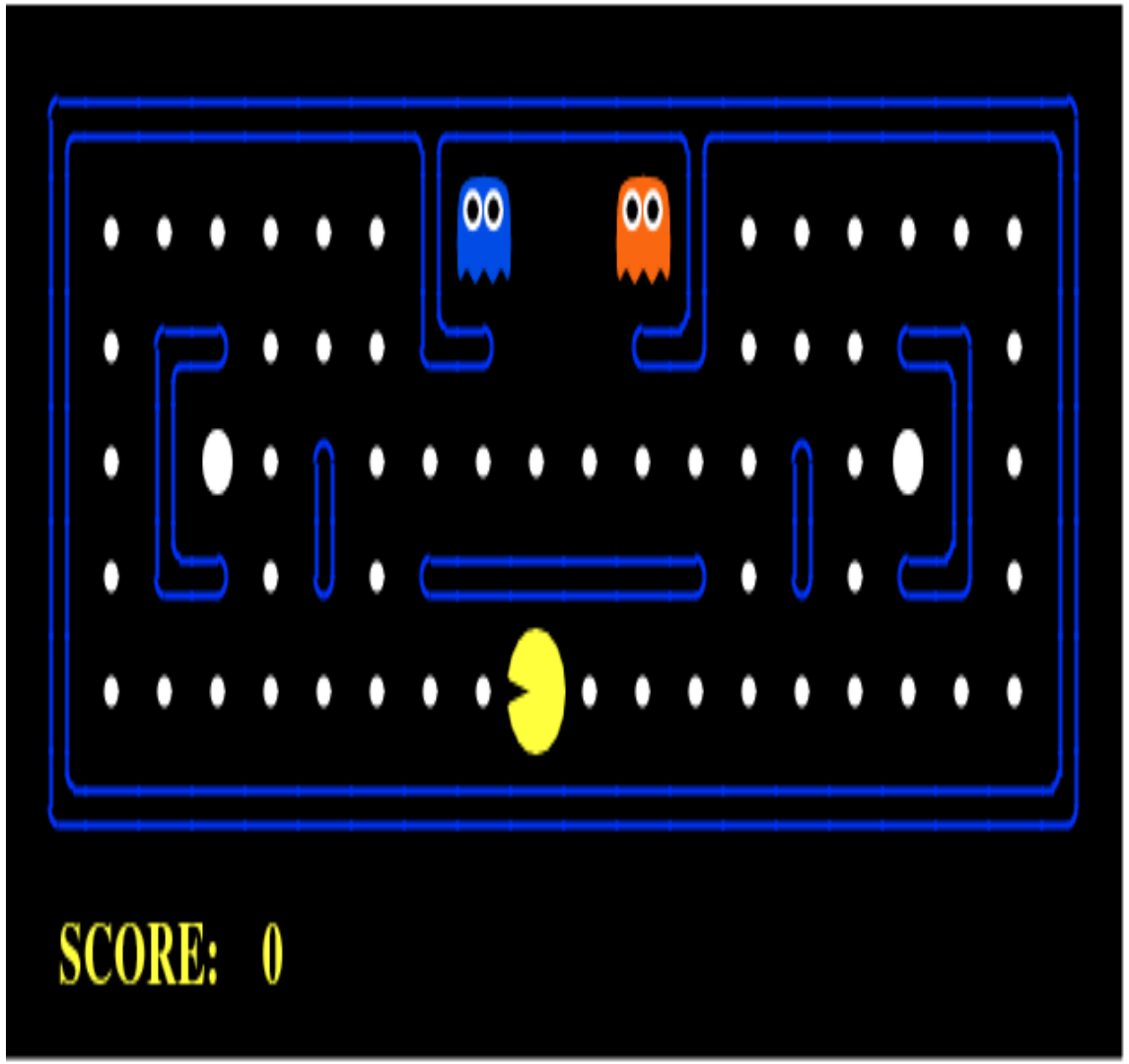
Types of Machine Learning



In Reinforcement learning the goal is to find a suitable action model that would maximize the **total cumulative reward** of the agent

Reinforcement Learning





Problem Formulation in RL

Environment

Physical world in which the agent operates

State

- Current situation of the agent

Reward

- Feedback from the environment

Policy

- Method to map agent's state to actions

Value

- Future reward that an agent would receive by taking an action in a particular state

Markov Decision Process:

The mathematical framework for defining a solution in a reinforcement learning scenario

Can be designed as:

- **Set of states, S**
- **Set of actions, A**
- **Reward function, R**
- **Policy, π**
- **Value, V**
- Set of action (A) has to be taken to transition from our start state to our end state (S).
- Model gets rewards (R) (Positive or Negative) for each action we take
- The set of actions we took defines our policy (π)
- rewards we get in return define our value (V)
- Our task
  - *is to maximize our rewards by choosing the correct policy.*
  - *we have to maximize for all possible values of S for a time t.*

Multi-Armed Bandit Problem

A bandit is defined as someone who steals your money.

One-armed bandit is a simple slot machine wherein you insert a coin into the machine, pull a lever, and get an immediate reward.

A multi-armed bandit

- there are several levers that a gambler can pull, with each lever giving a different return.
- Probability distribution for the reward corresponding to each lever is different and is unknown to the gambler.
- The task is to identify which lever to pull to get maximum reward after a given set of trials.

Multi-Armed Bandit problem (MAB)

- is a special case of **Reinforcement Learning**
- **MAB**
  - collects rewards in an environment by taking some actions after observing some state of the environment.
  - action taken by MAB does not influence the next state of the environment.
  - Therefore, MAB do not model state transitions, credit rewards to past actions, or "plan ahead" to get to reward-rich states.
- Goal of a MAB *agent* is to find a *policy* that collects as much reward as possible.
- **Exploration vs Exploitation Dilemma**
  - Not a good idea to exploit the action that promises the highest reward
  - because then there is a chance that we miss out on better actions if we do not explore enough.

Value of Action

In our k-armed bandit problem, each of the k actions has an expected or mean reward given that that action is selected

- A_t - action selected on time step t
- R_t- corresponding reward as Rt.
- The value of an arbitrary action a, denoted $q_*(a)$
- the expected reward given that a is selected: $q_*(a) = E[R_t \mid A_t=a]$

If you knew the value of each action, then it would be trivial to solve the k-armed bandit

Problem

- $Q_t(a)$ - the estimated value of action a at time step t
- $Q_t(a)$ to be close to $q_*(a)$

Exploitation

- maximize the expected reward on the one step
- maintain estimates of the action values
- Greedy action - at any time step there is at least one action whose estimated value is greatest

Exploration

- exploration may produce the greater total reward in the long run
- greedy action's value is known with certainty, while several other actions are estimated to be nearly as good but with substantial uncertainty

Action Value Methods

Sample Average Method

$$Q_t(a) \doteq \frac{\text{sum of rewards when } a \text{ taken prior to } t}{\text{number of times } a \text{ taken prior to } t} = \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}},$$

where $1_{predicate}$ denotes the random variable that is 1 if predicate is true and 0 if it is not

If the denominator

- is 0 - define $Q_t(a)$ as some default value, such as 0.
- goes to infinity- by the law of large number, Qt(a) converges to $q_*(a)$

Greedy action selection method

$$A_t \doteq \arg\max_a Q_t(a)$$

- argmax_a denotes the action a for which the expression that follows is maximized
- Variation - -greedy - select randomly from among all the actions with equal probability

exploration vs exploitation dilemma

Pure exploitation approach

- select only one slot machine and keep pulling the lever all day long.
- may give you “some” payouts.
- might hit the jackpot (with a probability close to 0.00000….1)

Pure exploration approach

- pull a lever of each & every slot machine
- Get sub-optimal payouts
- May be at least one of them would hit the jackpot.

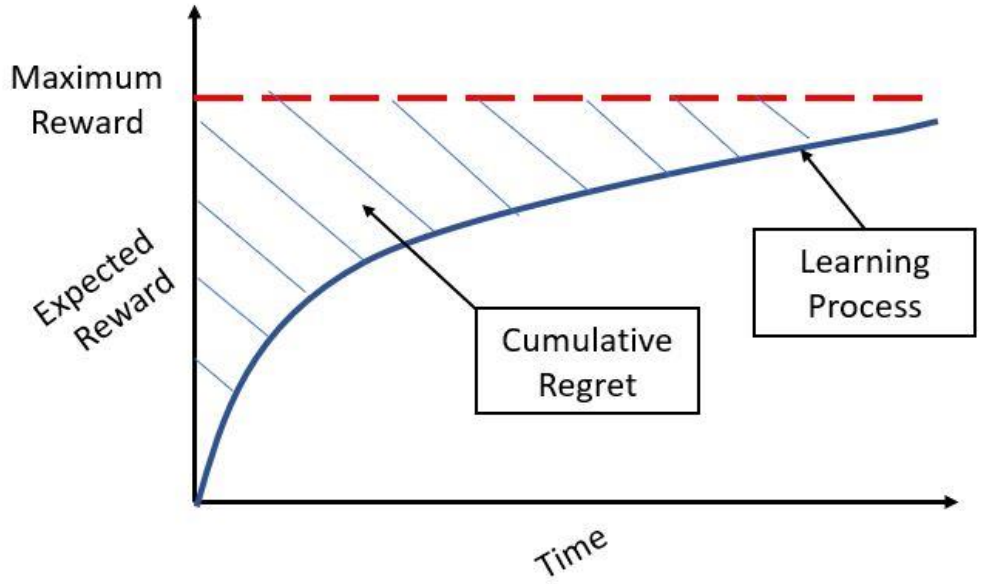
Arm	Reward
1	0
2	0
3	1
4	1
5	0
3	1
3	1
2	0
1	1
4	0
2	0

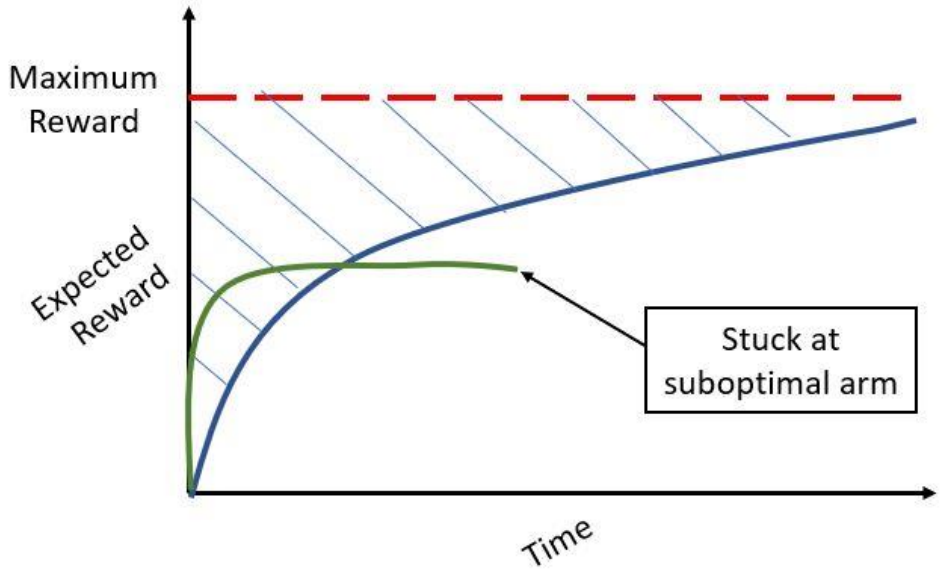
Exploration vs Exploitation trade-off

- To build an optimal policy, the agent faces the dilemma of exploring new states while maximizing its overall reward at the same time.

The best overall strategy may involve short-term sacrifices.

Therefore, the agent should collect enough information to make the best overall decision in the future.





Exploitation vs Exploration

ε -Greedy Method

- Behave greedily most of the time
- A few times with small probability Epsilon, select randomly from among all the actions with equal probability
- independently of the action-value estimates.

Advantages

- as the number of steps increases, every action will be sampled an infinite number of times thus ensuring that all the $Q_t(a)$ converge to $q_*(a)$

Incremental Implementation

$$
\begin{aligned}
Q_{n+1} &= \frac{1}{n}\sum_{i=1}^{n} R_i \\
&= \frac{1}{n}\left(R_n + \sum_{i=1}^{n-1} R_i\right) \\
&= \frac{1}{n}\left(R_n + (n-1)\frac{1}{n-1}\sum_{i=1}^{n-1} R_i\right) \\
&= \frac{1}{n}\Big(R_n + (n-1)Q_n\Big) \\
&= \frac{1}{n}\Big(R_n + nQ_n - Q_n\Big) \\
&= Q_n + \frac{1}{n}\Big[R_n - Q_n\Big],
\end{aligned}
$$

- Error in the estimate – [Target−OldEstimate]
- Target is the n^{th} reward.
- StepSize changes from time step to time step
- In processing the n^{th} reward for action a, the method uses the step-size parameter 1/n
- step size parameter is denoted as $\alpha_t(a)$
- Update Rule

$$NewEstimate \leftarrow OldEstimate + StepSize\Big[Target - OldEstimate\Big]$$

Simple Bandits Algorithm

Initialize, for $a = 1$ to k:

$Q(a) \leftarrow 0$

$N(a) \leftarrow 0$

Loop forever:

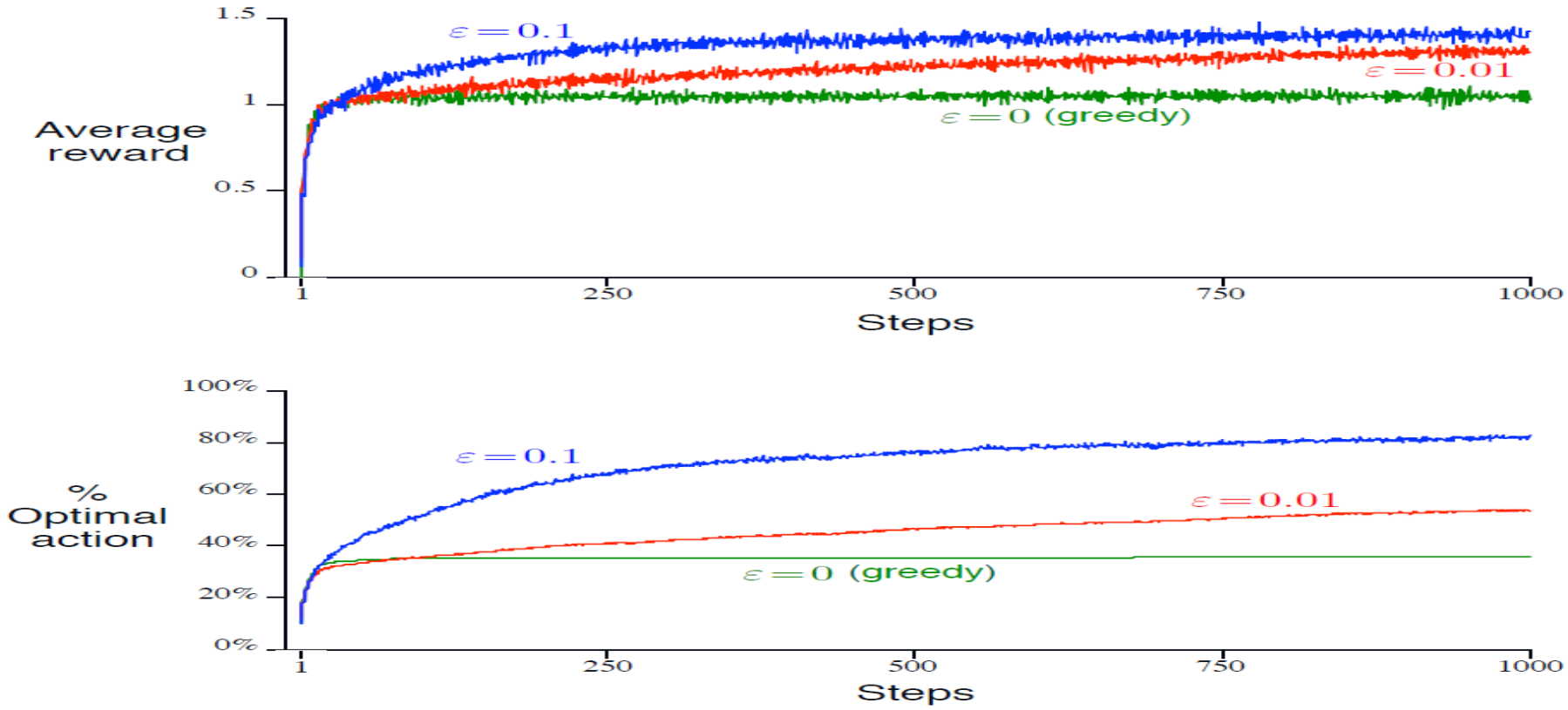
$$A \leftarrow \begin{cases} \text{argmax}_a Q(a) & \text{with probability } 1-\varepsilon \quad \text{(breaking ties randomly)} \\ \text{a random action} & \text{with probability } \varepsilon \end{cases}$$

$R \leftarrow bandit(A)$

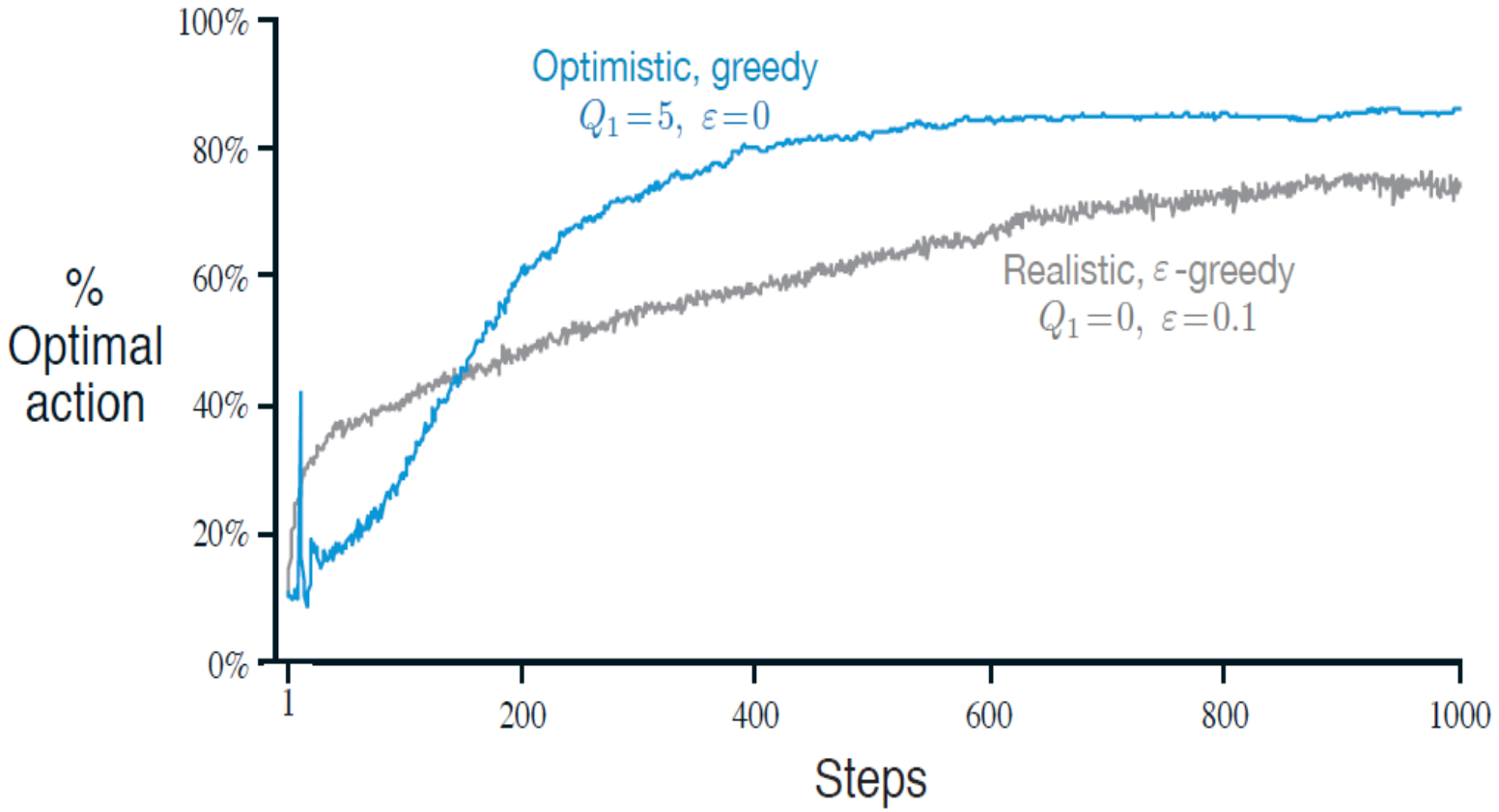
$N(A) \leftarrow N(A) + 1$

$Q(A) \leftarrow Q(A) + \frac{1}{N(A)}\left[R - Q(A)\right]$

Average Performance of ε -greedy



The effect of optimistic initial action values



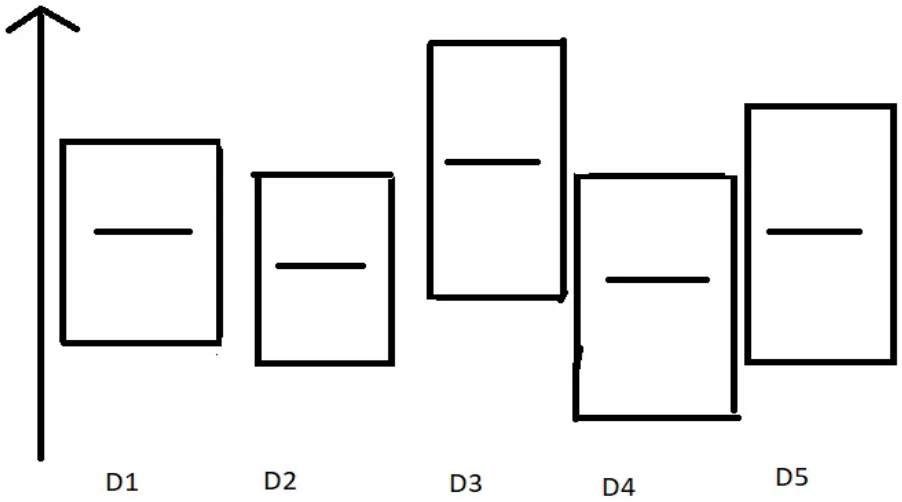
Upper-confidence-bound action selection

Optimism in the face of uncertainty

$$A_t \doteq \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right],$$

where;

- *$Q_t(a)$ is the estimated value of action 'a' at time step 't'.*
- *$N_t(a)$ is the number of times that action 'a' has been selected, prior to time 't'.*
- *'c' is a confidence value that controls the level of exploration.*



UCB Action Selection

$$A_t \doteq \underset{a}{\operatorname{argmax}} \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right],$$

Exploitation:

- $Q_t(a)$ represents the exploitation
- if you don't know which action is best then choose the one that currently looks to be the best

- ***Exploration:***
  - If an action hasn't been tried very often, or not at all, then $N_t(a)$ will be small. Consequently, the uncertainty term will be large, making this action to be selected
  - *As* $N_t(a)$ increments, and the uncertainty term decreases, making it less likely that this action will be selected as a result of exploration
  - it may still be selected as the action with the highest value

As n goes to infinity the exploration term gradually decreases until eventually, actions are selected based only on the exploitation term.

Steps followed in UCB agent

1. *At each round **t**, we compute two numbers for arm A.*
   -> $N_A(t)$ = number of times the arm A was selected up to round t.
   -> $R_A(t)$ = number of rewards of the arm A up to round t.
2. *From these two numbers we have to calculate,*
   a. The average reward of machine m up to round t

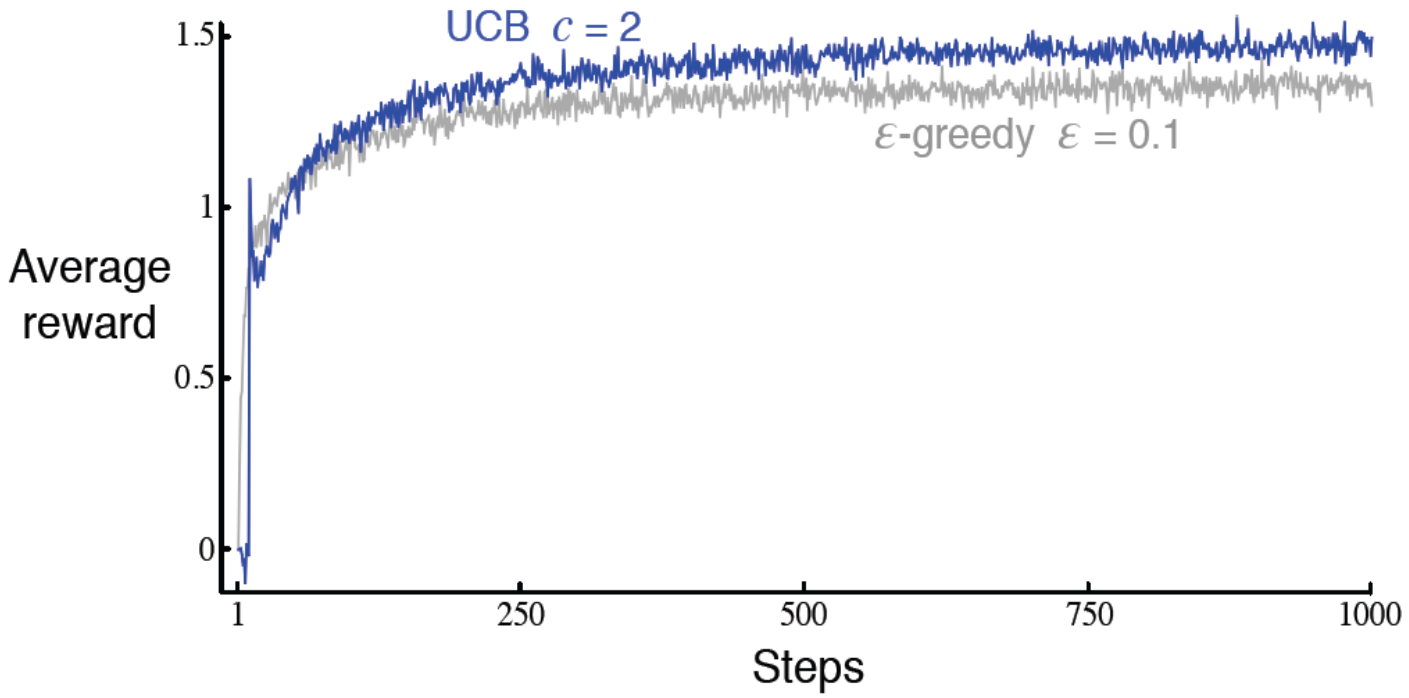
$$r_A(t) = R_A(t) / N_A(t).$$

b. The confidence interval $[r_A(t) — \Delta_A(t), r_A(t)+\Delta_A(t)]$ at round n

with, $\Delta_A(t) => c * sqrt(ln(t) / N_a(t))$

1. *We select the arm A that has the maximum UCB,* $(r_A(t)+\Delta_A(t))$

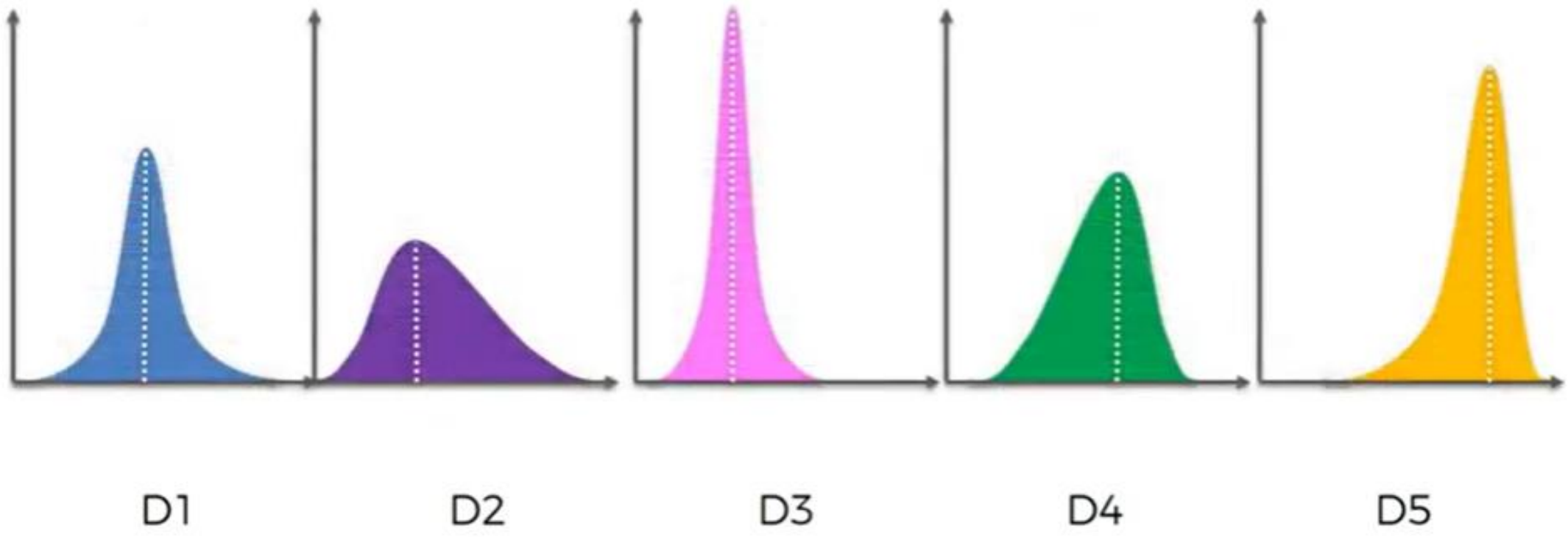
Upper-confidence-bound action selection



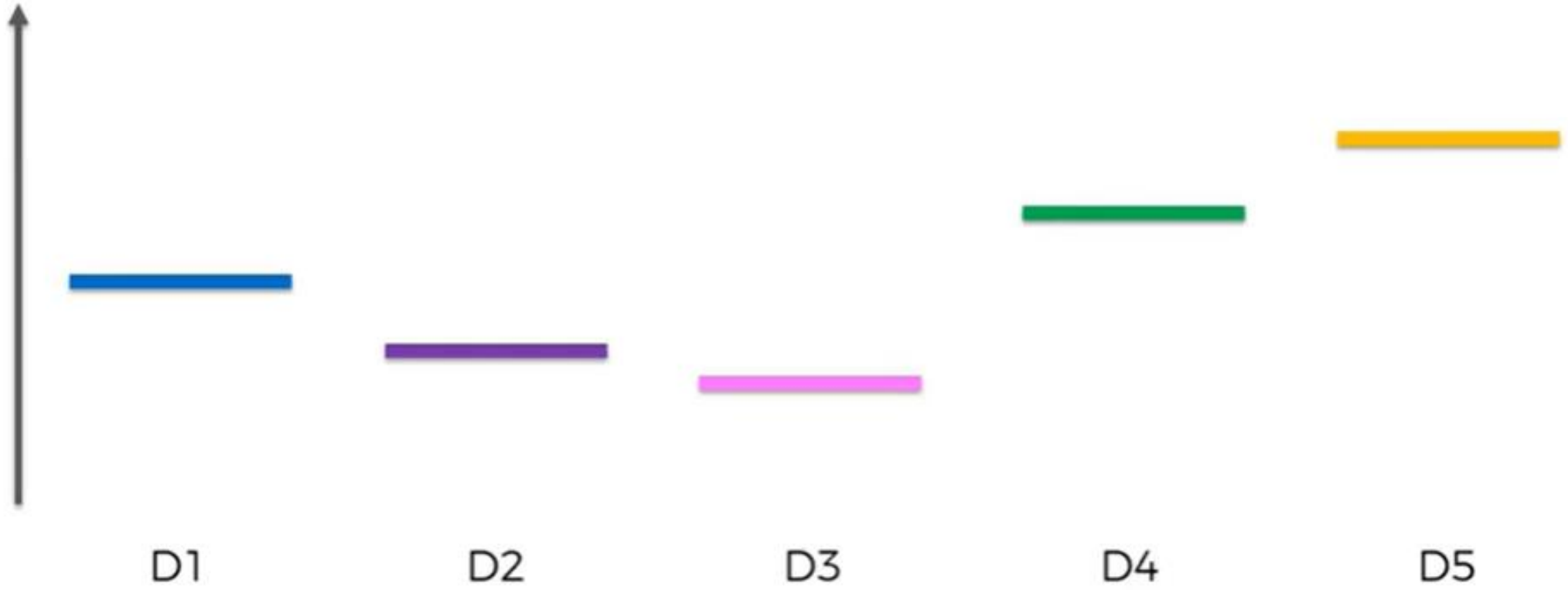
Upper Confidence Bound Algorithm

D1 D2 D3 D4 D5

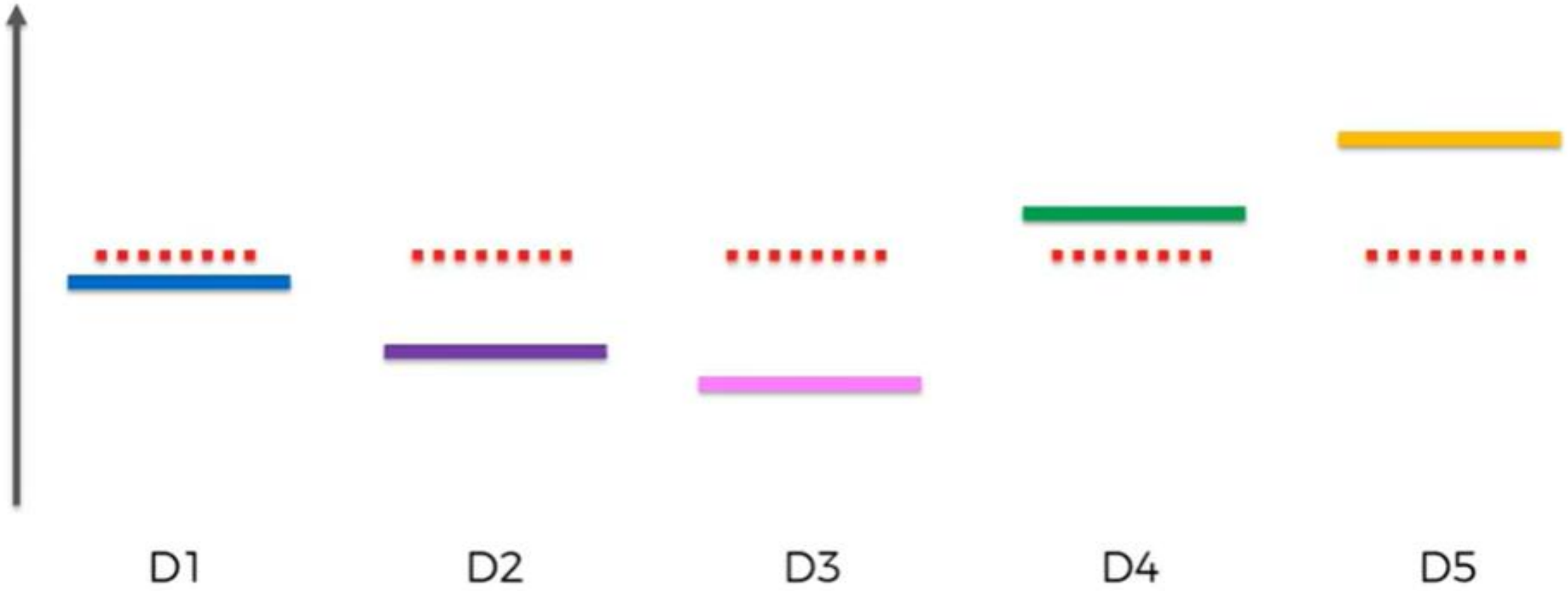
Upper Confidence Bound Algorithm



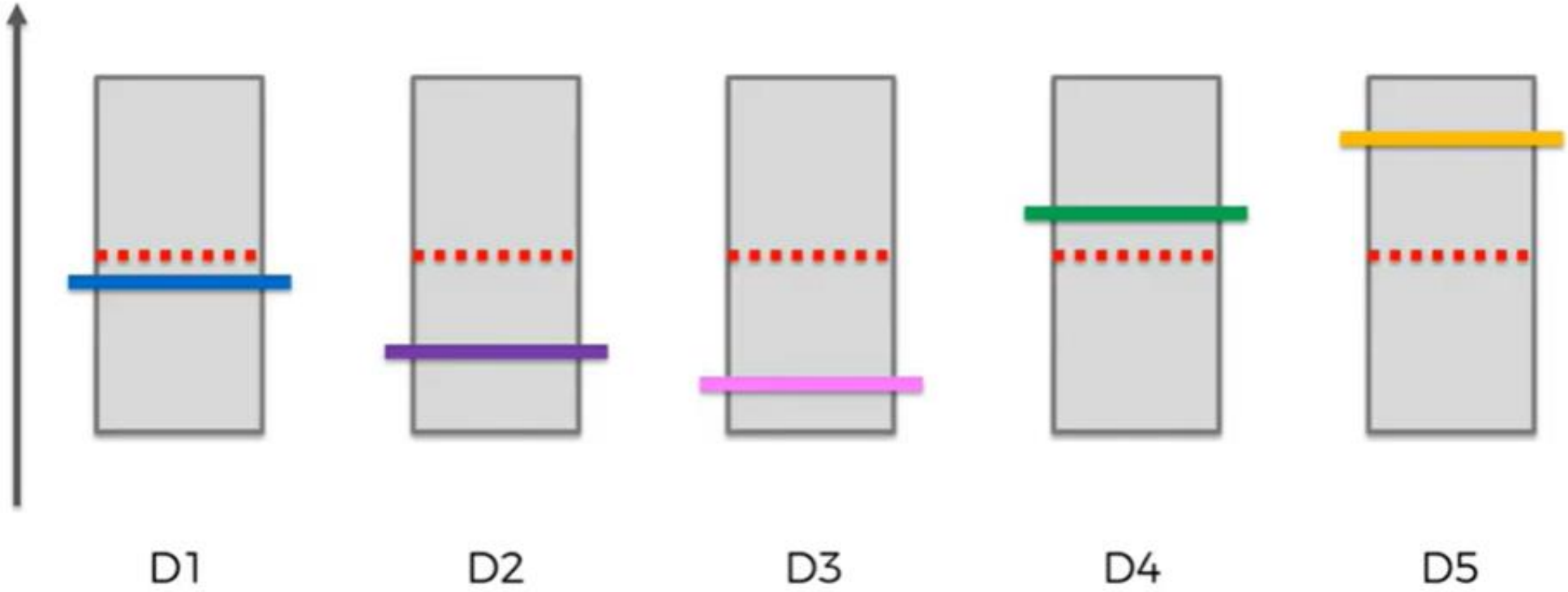
Upper Confidence Bound Algorithm

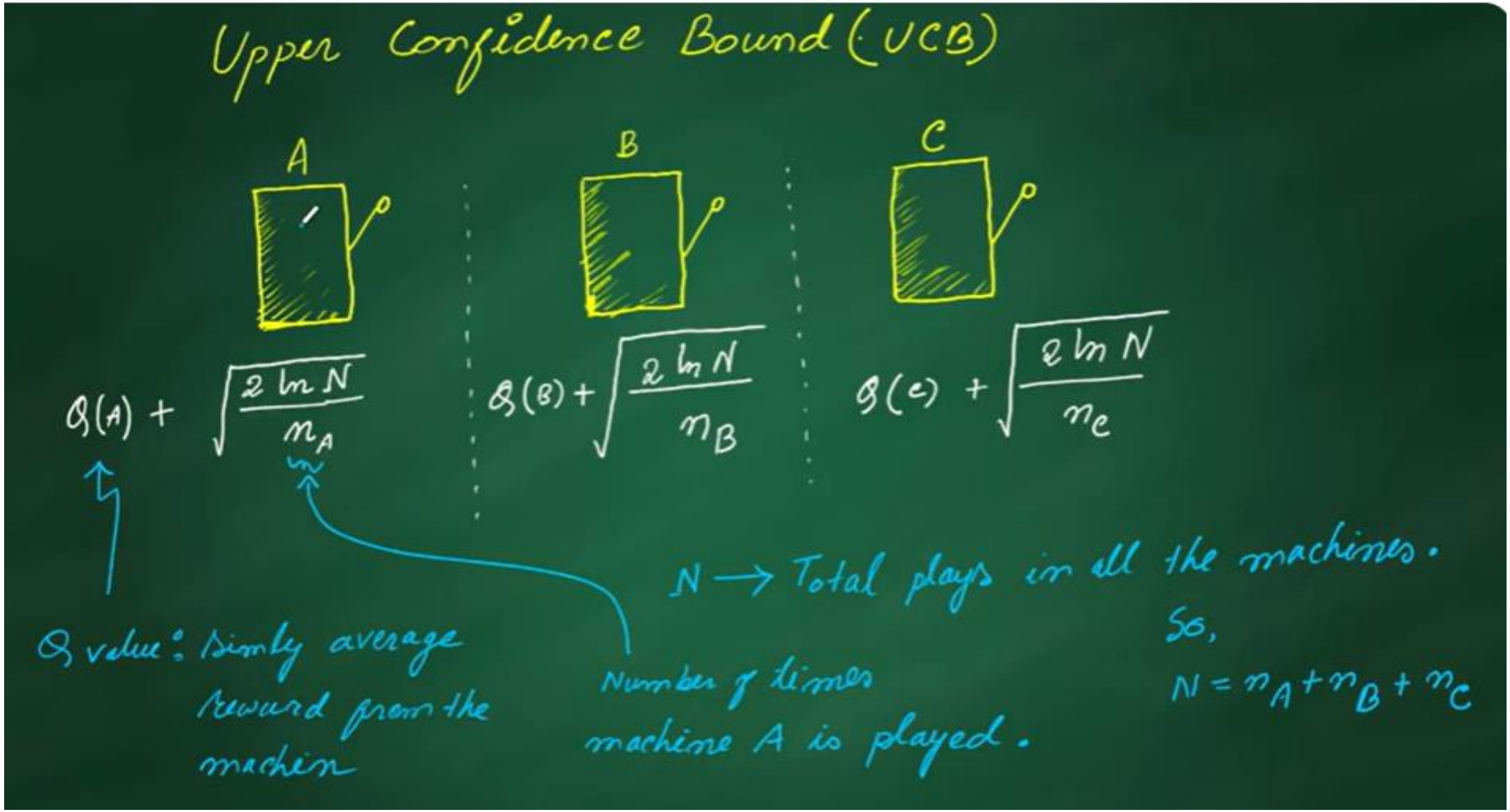


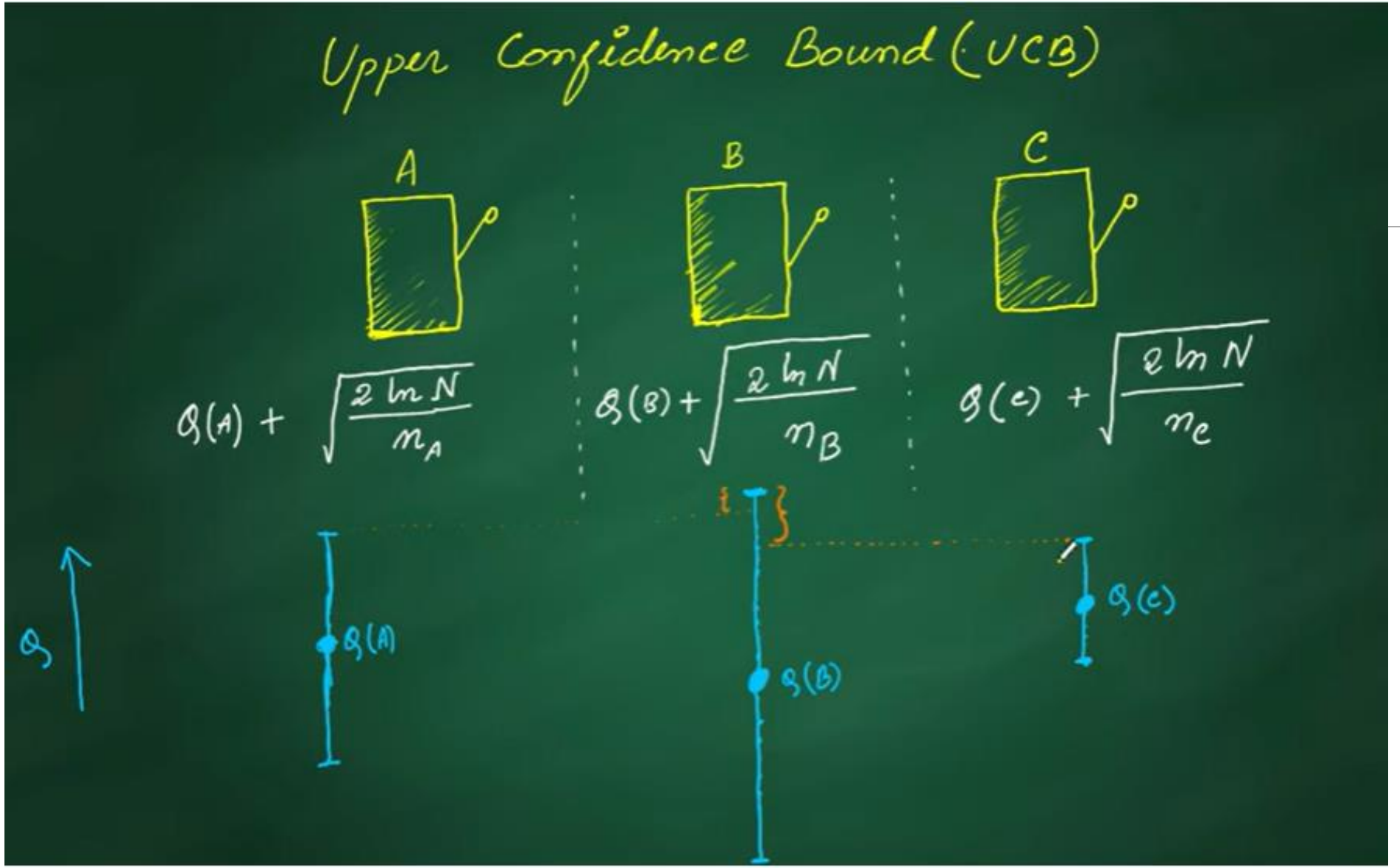
Upper Confidence Bound Algorithm



Upper Confidence Bound Algorithm




Upper Confidence Bound (UCB)
A
B
C
$Q(A) + \sqrt{\frac{2 \ln N}{n_A}}$
$Q(B) + \sqrt{\frac{2 \ln N}{n_B}}$
$Q(C) + \sqrt{\frac{2 \ln N}{n_C}}$
Q value: Simply average reward from the machin
Number of times machine A is played.
N → Total plays in all the machines.
So,
$N = n_A + n_B + n_C$


Upper Confidence Bound (UCB)
A
B
C
$Q(A) + \sqrt{\frac{2 \ln N}{n_A}}$
$Q(B) + \sqrt{\frac{2 \ln N}{n_B}}$
$Q(C) + \sqrt{\frac{2 \ln N}{n_C}}$
Q
Q(A)
Q(B)
Q(C)

Gradient ascent method

- Preference scores $H_t(a)$ towards action a
  - Variables: these are what we keep updating
- Choose actions based on a probability
  - Convert the preference scores to probability distribution using softmax
  - Define the probability of choosing action a as $\pi_t(a) = P[A_t = a] = \frac{e^{H_t(a)}}{\sum_{b=1}^{k} e^{H_t(b)}}$
- Objective function to be maximized: expected reward $\sum_{x=1}^{k} \pi(x)\ q_*(x)$
- But we don't know $q_*(x)$, instead we can draw samples for $x = a$
  - High reward for action $a \rightarrow$ estimate for $q_*(a)$ increases $\rightarrow$ should increase $\pi(a)$
  - Objective function increases $\rightarrow$ parameter $H_t(a)$ increases
  - **Stochastic gradient ascent**

Bandit algorithm as gradient ascent

Our update rule

$$H_{t+1}(a) = H_t(a) + \alpha \frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)}$$

where

$$\mathbb{E}[R_t] = \sum_x \pi_t(x) q_*(x)$$

Derivation of stochastic gradient ascent

- Claim: $\frac{\partial \pi_t(x)}{\partial H_t(a)} = \pi_t(x)(\mathbf{1}_{a=x} - \pi_t(a))$, where $\mathbf{1}_{a=x} = 1$ if $a = x$, 0 otherwise

- Proof: $\frac{\partial \pi_t(x)}{\partial H_t(a)} = \frac{\partial}{\partial H_t(a)} \left[\frac{e^{H_t(x)}}{\sum_{y=1}^{k} e^{H_t(y)}} \right] = \frac{\frac{\partial e^{H_t(x)}}{\partial H_t(a)} \sum_{y=1}^{k} e^{H_t(y)} - e^{H_t(x)} \frac{\partial \sum_{y=1}^{k} e^{H_t(y)}}{\partial H_t(a)}}{\left(\sum_{y=1}^{k} e^{H_t(y)} \right)^2}$

$$= \frac{\mathbf{1}_{a=x} e^{H_t(x)} \sum_{y=1}^{k} e^{H_t(y)} - e^{H_t(x)} e^{H_t(y)}}{\left(\sum_{y=1}^{k} e^{H_t(y)} \right)^2} = \frac{\mathbf{1}_{a=x} e^{H_t(x)}}{\sum_{y=1}^{k} e^{H_t(y)}} - \frac{e^{H_t(x)} e^{H_t(y)}}{\left(\sum_{y=1}^{k} e^{H_t(y)} \right)^2}$$

Gradient Bandit Algorithms

Probability of taking action a at time t

$$\Pr\{A_t = a\} \doteq \frac{e^{H_t(a)}}{\sum_{b=1}^{k} e^{H_t(b)}} \doteq \pi_t(a)$$

The action preferences are updated by

$$H_{t+1}(A_t) \doteq H_t(A_t) + \alpha\big(R_t - \bar{R}_t\big)\big(1 - \pi_t(A_t)\big), \quad \text{and}$$
$$H_{t+1}(a) \doteq H_t(a) - \alpha\big(R_t - \bar{R}_t\big)\pi_t(a), \quad \text{for all } a \neq A_t$$

Initially all action preferences are the same (e.g., $H_1(a) = 0$, for all a)

As Stochastic Gradient Ascent

$$H_{t+1}(a) \doteq H_t(a) + \alpha \frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)}$$

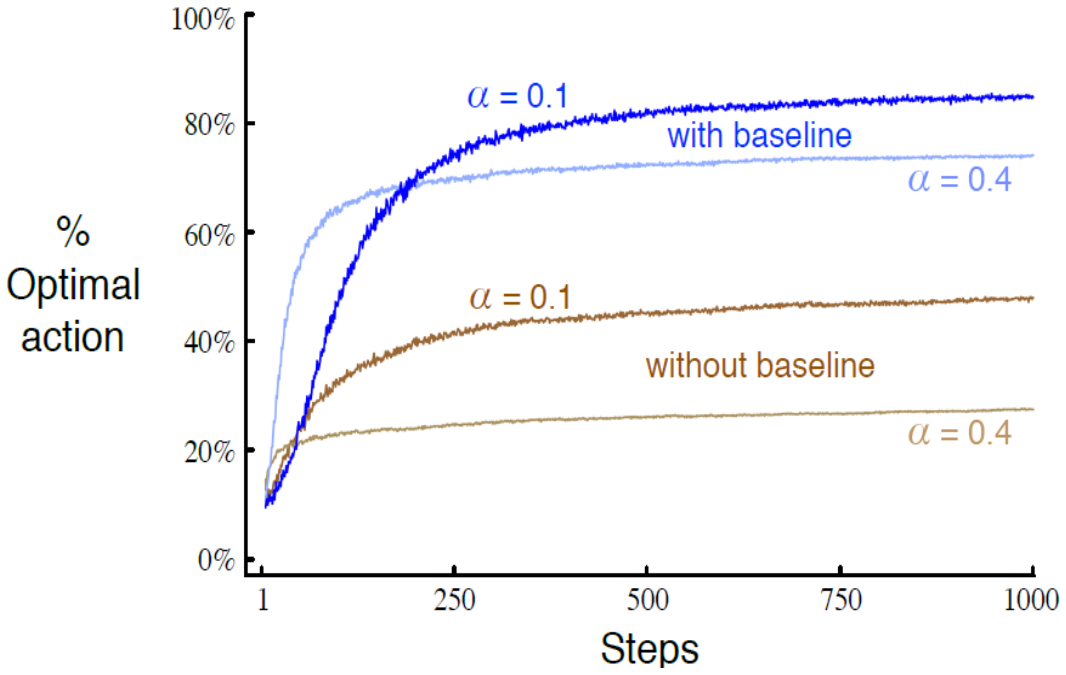
$$\mathbb{E}[R_t] = \sum_x \pi_t(x) q_*(x)$$

Where
α> 0 is step size parameter

$$\bar{R}_t \in \mathbb{R}$$

Is average of all rewards upto and including time t

Gradient Bandit Algorithms



Without Baseline –

$\bar{R}_t$ is set to 0

Ad Optimization

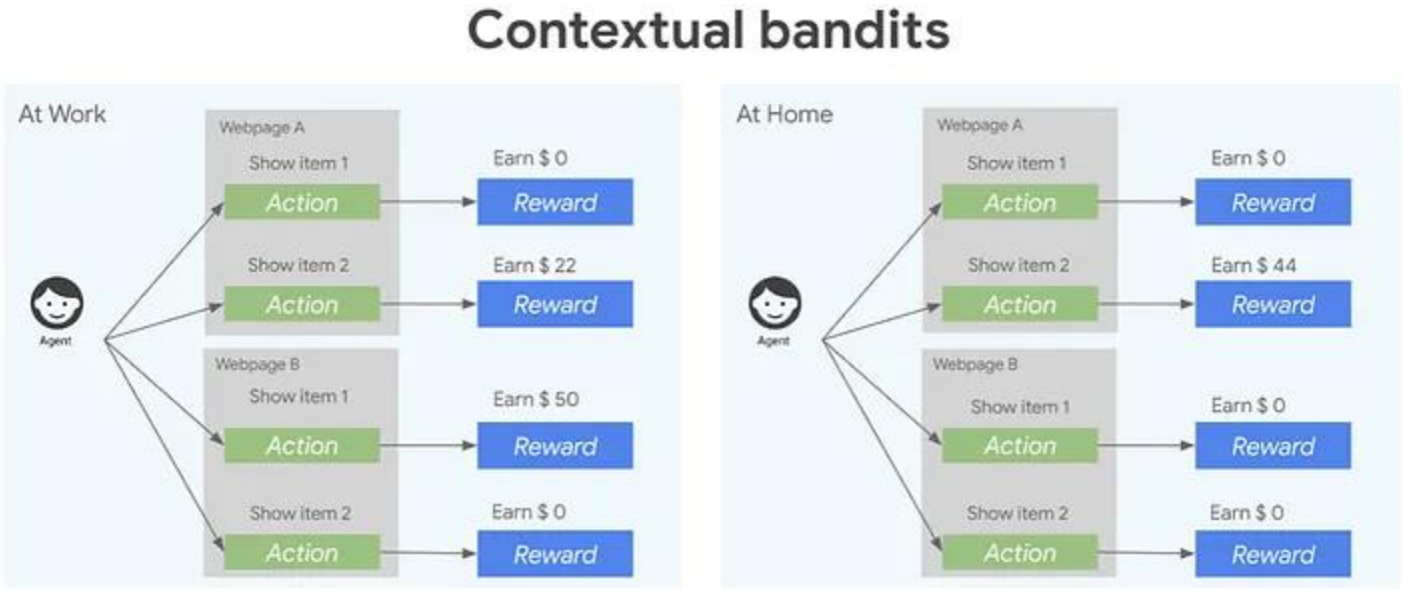
Ad 1	Ad 2	Ad 3	Ad 4	Ad 5	Ad 6	Ad 7	Ad 8	Ad 9	Ad 10
1	0	0	0	1	0	0	0	1	0
0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0

Associative Search (Contextual Bandits)

In k-armed bandit problem each action affects only the immediate reward

Contextual Bandits

- actions are affected by **context, which in turn affects the reward**
- In a general RL task, there is more than one situation/context
- **the goal is to learn a policy**: a mapping from situations/context to the actions that are best in those situations



Multi-armed Bandits Problem

- K actions (feature-free)
- Each action has an average reward (unknown): μ_k
- For t=1,…,T (unknown)
  - Choose an action a_t from {1, …, K} actions
  - Observe a random reward y_t, where y_t is bounded [0,1]
  - $E[y_t] = \mu_{a,t}$: Expected reward of action a_t
- Minimizing Regret: $$R = \sum_{t=1}^{T} [\mu^* - \mu_{a,t}]$$

regret is the difference between the total reward achieved by always selecting the optimal arm and the total reward achieved by the algorithm.

Q. How to choose an action to minimize regret?

Context RL

Assume you wish to show advertisements on a particular website for a user. Now, thinking out loud, you might not show every user the same set of ads. Right? for example, a user searching for shoes should see ads related to footwear or clothing while a person searching for noodles should see ads around edibles, snacks i.e. **we have an added layer of information about the user now,** his/her intentions or we can say the context of his visit to the website. We wish to use this information to show more relatable ads to the particular user. Right?

But in MAB, we can't add context and it considers only actions while estimating the reward. **How to add context to these MABs?**

- Adding the meta information or context becomes from user end becomes very important to remove the junk data.

Feature free bandit setting

Users u_1 with age YOUNG and u_2 with age OLD

u_1

u_2

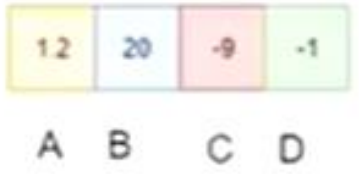
Retirement planning wishes vs. reality

The Player **Wizarding World of Harry Potter ride may conjure a new path for theme park rides**

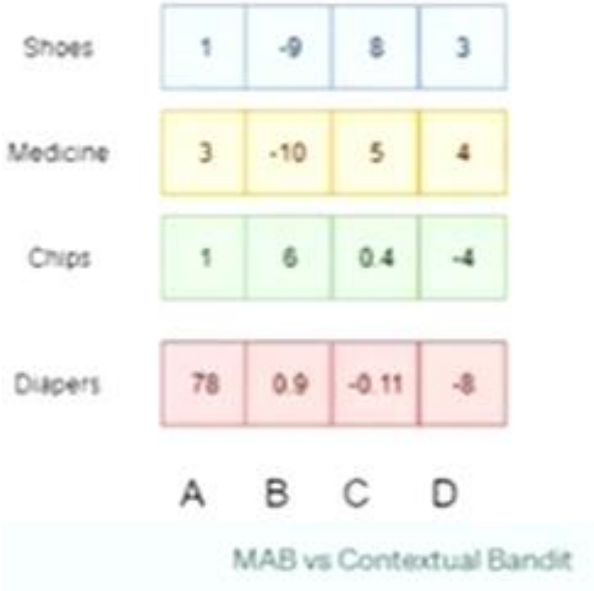
Elon Musk: 198,000 Tesla Model 3 Orders Received in 24 Hours

Not tired yet: Warriors top Spurs for 72nd win, set up date with history

Example: Recommendation System of products:



- All customers it shows the same product recommendation.



- Based on the context model shows the different product recommendation.
- Context yield to higher rewards in real time deployment.
- For shoes: Preference or recommendation will be product “c”
- For medicine: recommendation will be product “C”.

Contextual Bandit Problem

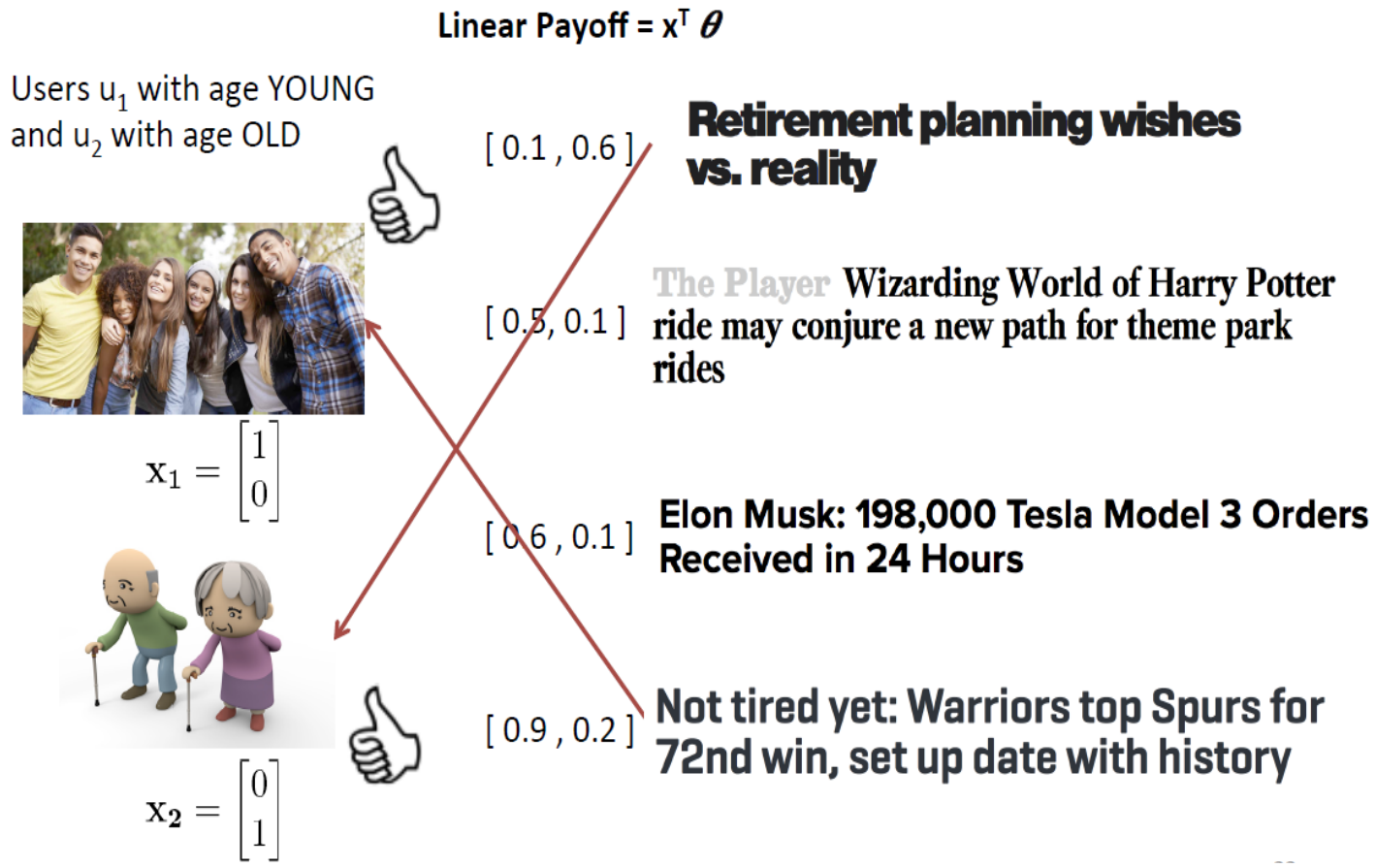
- For t=1,…,T (unknown)
  - User u_t, set A_t of actions (a)
  - Feature vector (context) $\mathbf{x}_{t,a}$: summarizes both user u_t and action a
  - Based on previous results, choose a_t from A_t
  - Receive payoff r_{t,a_t}
  - Improve selection strategy with new observation set $(x_{t,a_t}, a_t, r_{t,a_t})$

$$\mathbf{E}[r_{t,a}|\mathbf{x}_{t,a}] = \mathbf{x}_{t,a}^{\top}\boldsymbol{\theta}_a^*.$$

Minimizing Regret: $R(T) = \mathbf{E}\left[\sum_{t=1}^{T}\left(r_{t,a_t^*} - r_{t,a_t}\right)\right]$

Action with maximum expected payoff at time t

Contextual Linear Bandits setting



Contextual Bandit

For each trail t=1,2,3..., T

1. Observe environment $x_{t,a} \in \mathbb{R}^d$, i.e. user u_t a set of actions $\mathcal{A}_t$ and both their features
2. Choose an arm $a_t \in \mathcal{A}$ based on previous trails an receive payoff r_{t,a_t}.
3. Improve arm selection strategy with new observation $(x_{t,a_t}, a_t, r_{t,a_t})$

Example: News Recommendation

For each time the news page is loaded t=1,2,3..., T

1. Arms or actions are the articles, which can be shown to the user. The environment could be user and article information.
2. If the aricle is clicked $r_{t,a_t} = 1$ otherwise 0.
3. Improve new article selection

Minimize expected regret, i.e

$$R_A(T) = \mathbb{E}\left[\sum_{t=1}^{T} r_{t,a_t^*}\right] - \mathbb{E}\left[\sum_{t=1}^{T} r_{t,a_t}\right]$$

Linear Disjoint Model

- *disjoint* since the parameters are not shared among different arms.
- To solve for the coefficient vector Θ ridge regression is applied to the training data.

$$\mathrm{E}[r_{t,a}|x_{t,a}] = [x_{t,a}]^T \theta_a^*$$

- How to estimate θ_a?
  - Linear regression solution to θ_a is

$$\widehat{\boldsymbol{\theta_a}} = \boldsymbol{argmin_\theta} \sum_{\boldsymbol{m \in D_a}} ([x_{t,a}]^T \theta_a - \boldsymbol{b_a^{(m)}})^2$$

We can get:

$$\widehat{\boldsymbol{\theta_a}} = (\boldsymbol{D_a^T D_a} + \boldsymbol{I_d})^{-1} \boldsymbol{D_a^T b_a}$$

$\boldsymbol{D_a}$ is a m × d matrix of **m** training inputs $[x_{t,a}]$

$\boldsymbol{b_a}$ is a m-dimension vector of responses to $\boldsymbol{a}$(click/no-click)

linUCB Algorithm

- Initialization: $A_a \stackrel{\text{def}}{=} \boldsymbol{D_a^T D_a + I_d}$
  - For each arm a:
    - $A_a = I_d$ //identity matrix d × d
    - $b_a = [0]_d$ //vector of zeros
- Online algorithm:
  - For t=[1:T]:
    - Observe features for all arms $a : x_{t,a} \in R^d$
    - For each arm a :
      - $\theta_a = A_a^{-1} b_a$ //regression coefficients
      - $p_{t,a} = [x_{t,a}]^T \theta_a + \boldsymbol{\alpha} \sqrt{[x_{t,a}]^T A_a^{-1} x_{t,a}}$
    - Choose arm $a_t = argmax_a p_{t,a}$ //choose arm
    - $A_{a_t} = A_{a_t} + x_{t,a_t}[x_{t,a_t}]^T$ //update A for the chosen arm a_t
    - $b_{a_t} = b_{a_t} + r_t\, x_{t,a_t}$ //update b for the chosen arm a_t

Thomson Sampling

Basic Intuition

1. all machines are assumed to have a uniform distribution of the probability of reward
2. For each observation, a new distribution of rewards is generated (exploration)
3. Further observations are used to update the success distributions of rewards
4. After sufficient observations, each slot machine will have a success distribution of rewards (exploitation)

- A simple natural Bayesian heuristic
  - Maintain a belief(distribution) for the unknown parameters
  - Each time, pull arm *a* and observe a reward *r*

- Initialize priors using belief distribution
  - For t=1:T:
    - Sample random variable X from each arm's belief distribution
    - Select the arm with largest X
    - Observe the result of selected arm
    - Update prior belief distribution for selected arm

Example: Web Content Personalization

Vowpal Wabbit

- an interactive ML library and the RL framework for services like Microsoft Personalizer.
- It allows for maximum throughput and lowest latency when making personalization ranks and training the model with all events

Con-Ban Agent performs the following functions:

Some context ‘x’ arrives and is observed by Con-Ban Agent.

Con-Ban Agent chooses an action ‘a’ from a set of actions A, i.e., a ∈ A (A may depend on ‘x’).

Some reward ‘r’ for the chosen ‘a’ is observed by Con-Ban Agent.

For example: **Con-Ban Agent news website**:

- **Decision to optimize:** articles to display to user.
- **Context:** user data (browsing history, location, device, time of day)
- **Actions:** available news articles
- **Reward:** user engagement (click or no click)

Summary of Notation

$\doteq$	equality relationship that is true by definition
$\approx$	approximately equal
$\propto$	proportional to
$\Pr\{X=x\}$	probability that a random variable X takes on the value x
$X \sim p$	random variable X selected from distribution $p(x) \doteq \Pr\{X=x\}$
$\mathbb{E}[X]$	expectation of a random variable X, i.e., $\mathbb{E}[X] \doteq \sum_x p(x)x$
$\arg\max_a f(a)$	a value of a at which $f(a)$ takes its maximal value
$\ln x$	natural logarithm of x
e^x	the base of the natural logarithm, $e \approx 2.71828$, carried to power x; $e^{\ln x} = x$
$\mathbb{R}$	set of real numbers
$f : \mathcal{X} \to \mathcal{Y}$	function f from elements of set $\mathcal{X}$ to elements of set $\mathcal{Y}$
$\leftarrow$	assignment
$(a, b]$	the real interval between a and b including b but not including a
ε	probability of taking a random action in an ε-greedy policy
α, β	step-size parameters
γ	discount-rate parameter
λ	decay-rate parameter for eligibility traces
$\mathbb{1}_{predicate}$	indicator function ($\mathbb{1}_{predicate} \doteq 1$ if the *predicate* is true, else 0)

In a multi-arm bandit problem:

k	number of actions (arms)
t	discrete time step or play number
$q_*(a)$	true value (expected reward) of action a
$Q_t(a)$	estimate at time t of $q_*(a)$
$N_t(a)$	number of times action a has been selected up prior to time t
$H_t(a)$	learned preference for selecting action a at time t
$\pi_t(a)$	probability of selecting action a at time t
$\bar{R}_t$	estimate at time t of the expected reward given π_t

References

1. Richard S Sutton, Andrew G Barto, Reinforcement Learning, second edition, MIT Press

2. Russell S., and Norvig P., Artificial Intelligence A Modern Approach (3e), Pearson 2010

3. https://www.coursera.org/learn/fundamentals-of-reinforcement-learning/home/week/1

Artificial Intelligence
DSE 3252

Reinforcement Learning-1

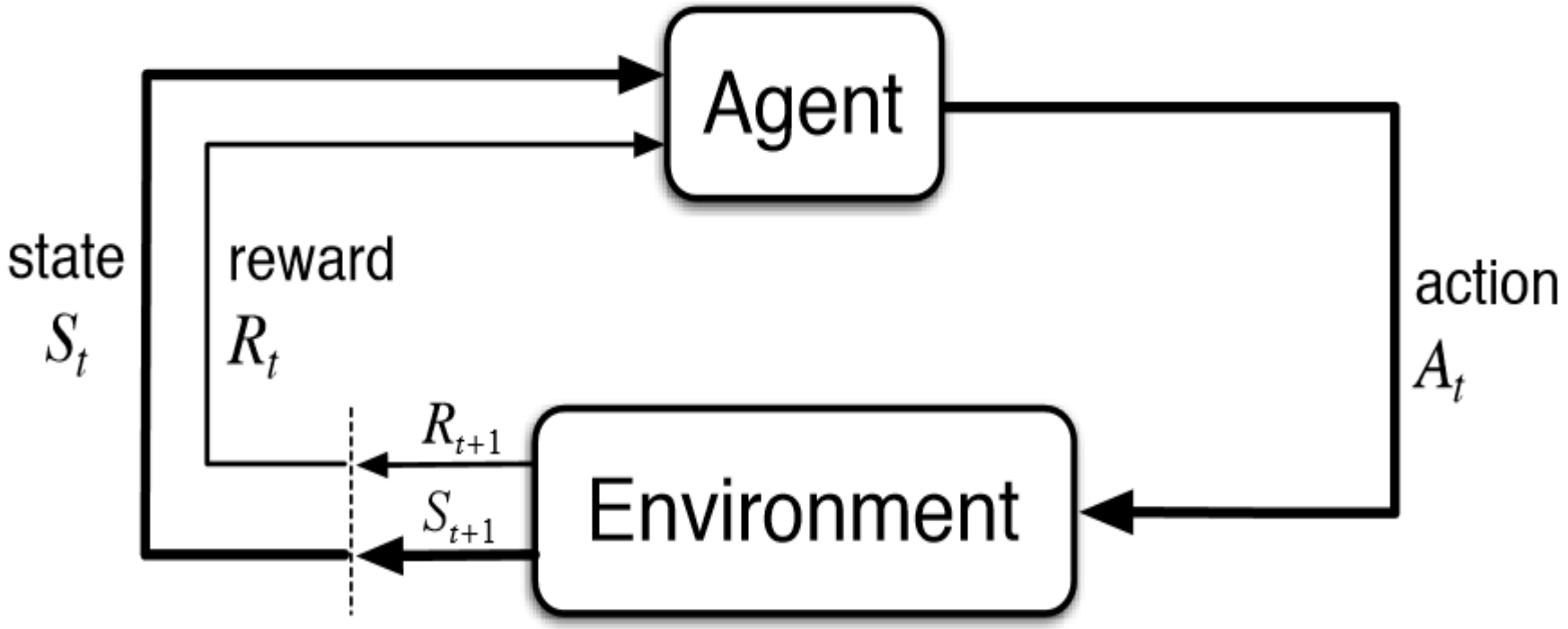
ROHINI R RAO
DEPT OF DATA SCIENCE & COMPUTER APPLICATIONS

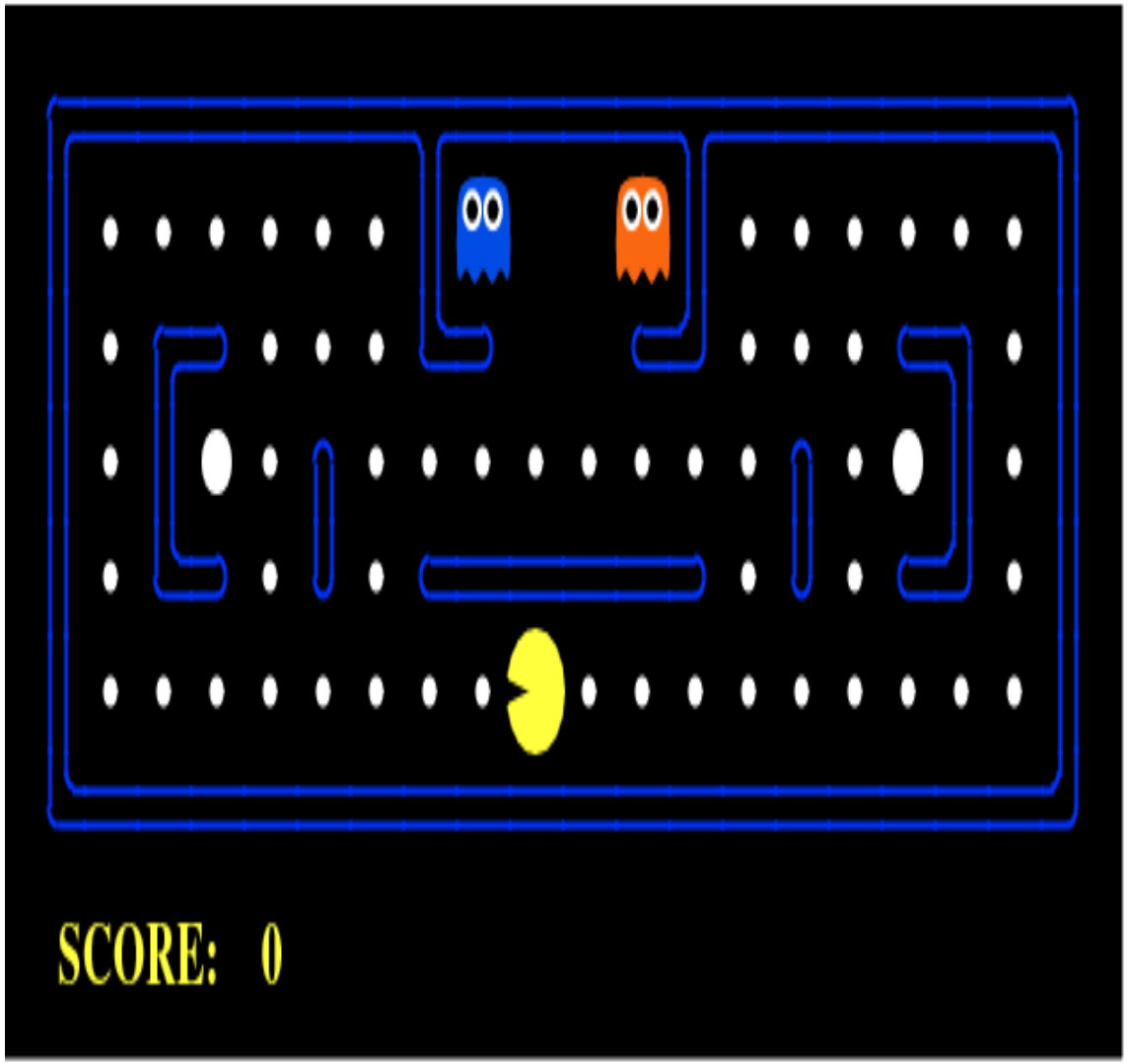
JANUARY 2024

What is Reinforcement Learning

- Reinforcement Learning is a
  - feedback-based Machine learning technique in which an agent learns to behave in an environment by performing the actions and seeing the results of actions.
  - For each good action, the agent gets positive feedback
  - for each bad action, the agent gets negative feedback or penalty.
- ***"Reinforcement learning is a type of machine learning method where an intelligent agent (computer program) interacts with the environment and learns to act within that."***

Reinforcement Learning





Problem Formulation in RL

Environment

Physical world in which the agent operates

State

- Current situation of the agent

Reward

- Feedback from the environment

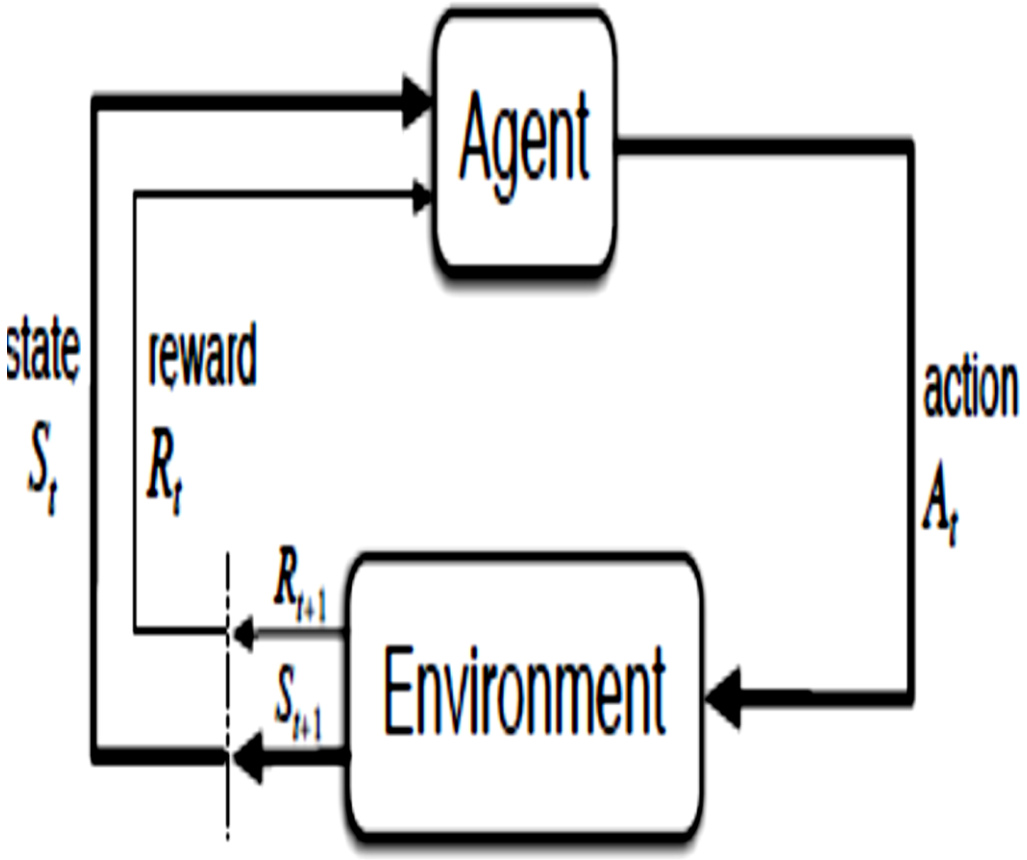
Policy

- Method to map agent's state to actions

Value

- Future reward that an agent would receive by taking an action in a particular state

Markov Decision Process



MDP and Agent give rise to a trajectory

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots$$

Dynamics of MDP

$$p(s', r \mid s, a) \doteq \Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\},$$

for all $s', s \in \mathcal{S}$, $r \in \mathcal{R}$, and $a \in \mathcal{A}(s)$

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a) = 1, \text{ for all } s \in \mathcal{S}, a \in \mathcal{A}(s).$$

MDP as state transition probabilities

$$p(s'|s,a) \doteq \Pr\{S_t=s' \mid S_{t-1}=s, A_{t-1}=a\} = \sum_{r\in\mathcal{R}} p(s',r|s,a). \tag{3.4}$$

We can also compute the expected rewards for state–action pairs as a two-argument function $r : \mathcal{S}\times\mathcal{A}\to\mathbb{R}$:

$$r(s,a) \doteq \mathbb{E}[R_t \mid S_{t-1}=s, A_{t-1}=a] = \sum_{r\in\mathcal{R}} r \sum_{s'\in\mathcal{S}} p(s',r|s,a), \tag{3.5}$$

and the expected rewards for state–action–next-state triples as a three-argument function $r : \mathcal{S}\times\mathcal{A}\times\mathcal{S}\to\mathbb{R}$,

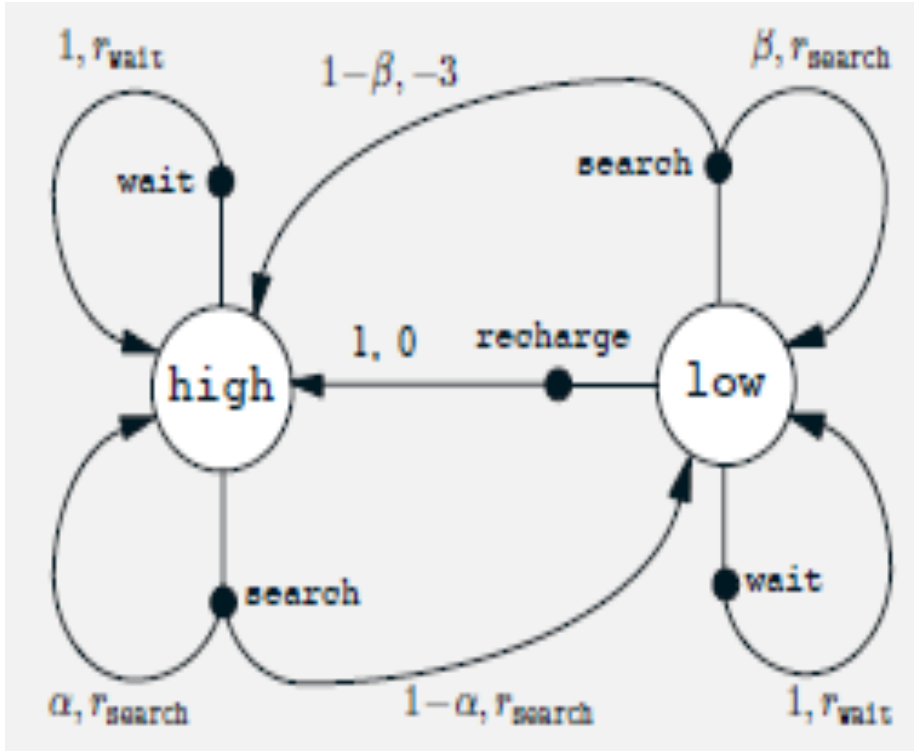
$$r(s,a,s') \doteq \mathbb{E}[R_t \mid S_{t-1}=s, A_{t-1}=a, S_t=s'] = \sum_{r\in\mathcal{R}} r\frac{p(s',r|s,a)}{p(s'|s,a)}. \tag{3.6}$$

Recycling Robot

A mobile robot has the job of collecting empty soda cans in an office environment. It has sensors for detecting cans, and an arm and gripper that can pick them up and place them in an onboard bin; it runs on a rechargeable battery. The robot's control system has components for interpreting sensory information, for navigating, and for controlling the arm and gripper. High-level decisions about how to search for cans are made by a reinforcement learning agent based on the current charge level of the battery. To make a simple example, we assume that only two charge levels can be distinguished, comprising a small state set $\mathcal{S} = \{\texttt{high}, \texttt{low}\}$. In each state, the agent can decide whether to (1) actively `search` for a can for a certain period of time, (2) remain stationary and `wait` for someone to bring it a can, or (3) head back to its home base to `recharge` its battery. When the energy level is `high`, recharging would always be foolish, so we do not include it in the action set for this state. The action sets are then $\mathcal{A}(\texttt{high}) = \{\texttt{search}, \texttt{wait}\}$ and $\mathcal{A}(\texttt{low}) = \{\texttt{search}, \texttt{wait}, \texttt{recharge}\}$.

Example – recycling robot

s	a	s'	$p(s'\|s,a)$	$r(s,a,s')$
high	search	high	α	r_{search}
high	search	low	$1-\alpha$	r_{search}
low	search	high	$1-\beta$	-3
low	search	low	β	r_{search}
high	wait	high	1	r_{wait}
high	wait	low	0	-
low	wait	high	0	-
low	wait	low	1	r_{wait}
low	recharge	high	1	0
low	recharge	low	0	-



Goals and Rewards

Reward hypothesis

- all of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward)

- **Return**

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T,$$

- **Episodes**
  - Each episode ends in a special state called the terminal state
  - followed by a reset to a standard starting state or to a sample from a standard distribution of starting states.
  - next episode begins independently of how the previous one ended.

- Thus the episodes end in the same terminal state, with different rewards for the different outcomes

- **Episodic tasks**
  - distinguish the set of all nonterminal states, denoted S, from the set of all states plus the terminal state, denoted S+.
  - The time of termination, T, is a random variable that normally varies from episode to episode.

- **Continuing Tasks**
  - the agent–environment interaction does not break naturally into identifiable episodes, but goes on continually without limit
  - T = α, and the return, which is what we are trying to maximize, could itself easily be infinite

Discounted Return

Agent chooses A_t to maximize the expected discounted return:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$

where γ is a parameter, $0 \leq \gamma \leq 1$, called the *discount rate*.

Discount rate determines the present value of future rewards

- a reward received k time steps in the future is worth only γ^{k-1} times what it would be worth if it were received immediately
- If $\gamma < 1$, the infinite sum in has a finite value as long as the reward sequence R_k is bounded
- If $\gamma = 0$, the agent is "myopic" , maximizing immediate rewards
- As γ approaches 1, the return objective takes future rewards into account more strongly; the agent becomes more farsighted

Returns at successive time steps

$$
\begin{aligned}
G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \\
&= R_{t+1} + \gamma \left(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots \right) \\
&= R_{t+1} + \gamma G_{t+1}
\end{aligned}
$$

If reward is constant + 1

$$G_t = \sum_{k=0}^{\infty} \gamma^k = \frac{1}{1-\gamma}.$$

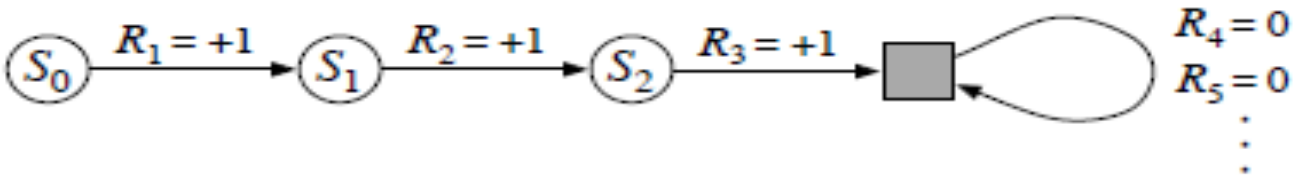
Computation of Expected Return

Consider an MDP process, and compute the expected return G_0, G_1 ….till G_5. Let $\gamma = 0.9$, the following sequence of rewards is received :

R1 = −3, R2 = 4, R3 = 2, R4 = 1, and R5 = -3, with T = 5.

Unified notation for Episodic & Continuous Tasks

State Transition Diagram



$$G_t \doteq \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k,$$

including the possibility that $T = \infty$ or $\gamma = 1$ (but not both).

Policies and Value Functions

Policy

- is a mapping from states to probabilities of selecting each possible action.
- If the agent follows policy π at time t, then π(a|s) is the probability that A_t = a if S_t = s.

Value Function of state s under policy π (state-value function for policy π)

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right], \text{ for all } s \in \mathcal{S},$$

where $\mathbb{E}_\pi[\cdot]$ denotes the expected value of a random variable given that the agent follows policy π, and t is any time step. Note that the value of the terminal state, if any, is always zero. We call the function v_π the *state-value function for policy* π.

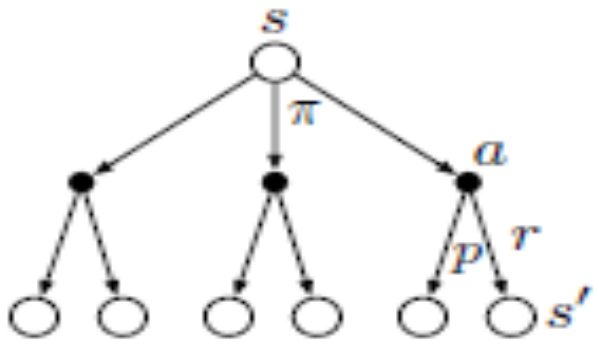
Expected return starting from s, taking the action a, and policy π: (action-value function for policy π)

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right].$$

Bellman Equation v_π

Expresses a relationship between the value of a state and the values of its successor states

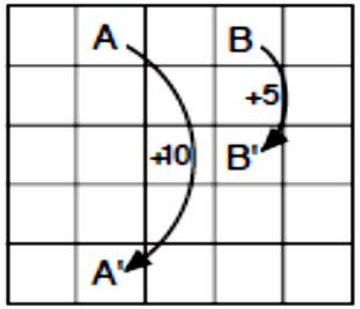
$$
\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r \mid s, a) \Big[r + \gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s'] \Big] \\
&= \sum_a \pi(a|s) \sum_{s', r} p(s', r \mid s, a) \Big[r + \gamma v_\pi(s') \Big], \quad \text{for all } s \in \mathcal{S},
\end{aligned}
$$

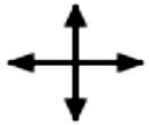


Backup diagram for v_π

Gridworld

of a simple finite MDP. The cells of the grid correspond to the states of the environment. At each cell, four actions are possible: `north`, `south`, `east`, and `west`, which deterministically cause the agent to move one cell in the respective direction on the grid. Actions that would take the agent off the grid leave its location unchanged, but also result in a reward of -1. Other actions result in a reward of 0, except those that move the agent out of the special states A and B. From state A, all four actions yield a reward of $+10$ and take the agent to A'. From state B, all actions yield a reward of $+5$ and take the agent to B'.



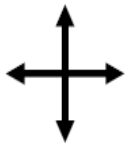


Actions

3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

Figure 3.2: Gridworld example: exceptional reward dynamics (left) and state-value function for the equiprobable random policy (right).

1	2	3	4
5	6 	7	8
9	10	11	12
13	14	15	16

actions 

Reward is -1 for all transition

$$v_1(6) = \sum_{a\in\{u,d,l,r\}} \pi(a|6) \sum_{s',r} p(s',r|6,a)[r + \gamma v_0(s')]$$

$$= \sum_{a\in\{u,d,l,r\}} \underbrace{\pi(a|6)}_{=0.25\ \forall a} \sum_{s'} p(s'|6,a)[\underbrace{r}_{=-1} + \gamma \underbrace{v_0(s')}_{=0\ \forall s'}]$$

$$= 0.25 * \{-p(2|6,u) - p(10|6,d) - p(5|6,l) - p(7|6,r)\}$$

$$= 0.25 * \{-1 - 1 - 1 - 1\}$$

$$= -1$$

$$\Rightarrow v_1(6) = -1$$

Optimal Policies &Optimal Value functions

Optimal State Value Function

$$v_*(s) \doteq \max_\pi v_\pi(s),$$

for all $s \in \mathcal{S}$.

Optimal Action Value Function

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a),$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$.

Thus, we can write q_* in terms of v_* as follows:

$$q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a]$$

Bellman Optimality

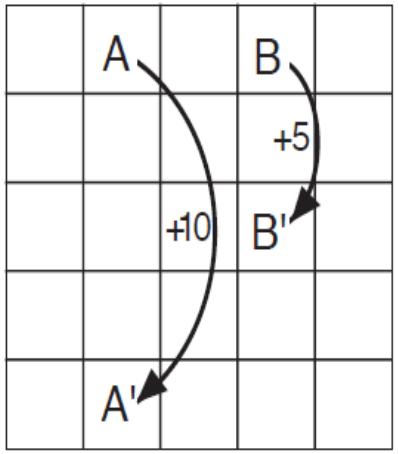
Bellman optimality equation says that the value of each state under an optimal policy must be the return the agent gets when it follows the best action as given by the optimal policy. For optimal policy π^*, the optimal value function is given by:

$$
\begin{aligned}
v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\
&= \max_a \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] \\
&= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\
&= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a]
\end{aligned}
$$

Given a value function q^*, we can recover an optimum policy as follows:

$$
\begin{aligned}
\pi'(s) &\doteq \arg\max_a q_\pi(s,a) \\
&= \arg\max_a \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a]
\end{aligned}
$$

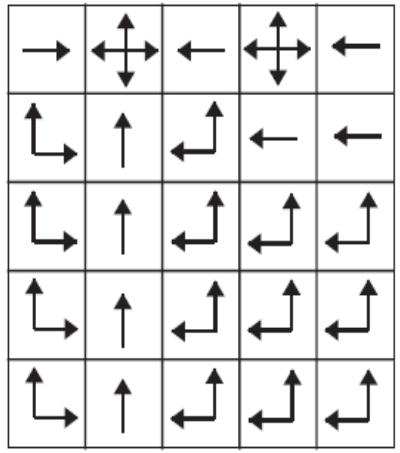
Solving Gridworld



Gridworld

22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

v_*



π_*

Figure 3.5: Optimal solutions to the gridworld example.

Bellman Optimality Equations for recycling robot

example. To make things more compact, we abbreviate the states **high** and **low**, and the actions **search**, **wait**, and **recharge** respectively by **h**, **l**, **s**, **w**, and **re**. Because there are only two states, the Bellman optimality equation consists of two equations. The equation for $v_*(\mathtt{h})$ can be written as follows:

$$\begin{aligned} v_*(\mathtt{h}) &= \max \left\{ \begin{array}{l} p(\mathtt{h}|\mathtt{h},\mathtt{s})[r(\mathtt{h},\mathtt{s},\mathtt{h}) + \gamma v_*(\mathtt{h})] + p(\mathtt{l}|\mathtt{h},\mathtt{s})[r(\mathtt{h},\mathtt{s},\mathtt{l}) + \gamma v_*(\mathtt{l})], \\ p(\mathtt{h}|\mathtt{h},\mathtt{w})[r(\mathtt{h},\mathtt{w},\mathtt{h}) + \gamma v_*(\mathtt{h})] + p(\mathtt{l}|\mathtt{h},\mathtt{w})[r(\mathtt{h},\mathtt{w},\mathtt{l}) + \gamma v_*(\mathtt{l})] \end{array} \right\} \\ &= \max \left\{ \begin{array}{l} \alpha[r_{\mathtt{s}} + \gamma v_*(\mathtt{h})] + (1-\alpha)[r_{\mathtt{s}} + \gamma v_*(\mathtt{l})], \\ 1[r_{\mathtt{w}} + \gamma v_*(\mathtt{h})] + 0[r_{\mathtt{w}} + \gamma v_*(\mathtt{l})] \end{array} \right\} \\ &= \max \left\{ \begin{array}{l} r_{\mathtt{s}} + \gamma[\alpha v_*(\mathtt{h}) + (1-\alpha) v_*(\mathtt{l})], \\ r_{\mathtt{w}} + \gamma v_*(\mathtt{h}) \end{array} \right\}. \end{aligned}$$

Following the same procedure for $v_*(\mathtt{l})$ yields the equation

$$v_*(\mathtt{l}) = \max \left\{ \begin{array}{l} \beta r_{\mathtt{s}} - 3(1-\beta) + \gamma[(1-\beta) v_*(\mathtt{h}) + \beta v_*(\mathtt{l})], \\ r_{\mathtt{w}} + \gamma v_*(\mathtt{l}), \\ \gamma v_*(\mathtt{h}) \end{array} \right\}.$$

For any choice of $r_{\mathtt{s}}$, $r_{\mathtt{w}}$, α, β, and γ, with $0 \leq \gamma < 1$, $0 \leq \alpha, \beta \leq 1$, there is exactly one pair of numbers, $v_*(\mathtt{h})$ and $v_*(\mathtt{l})$, that simultaneously satisfy these two nonlinear equations. ■

Dynamic Programming

In Reinforcement Learning

- **Policy evaluation** refers to determining the value function of a specific policy
- **Control** refers to the task of finding a policy that maximizes reward.

Control is the ultimate goal of reinforcement learning and policy evaluation is usually a necessary step to get there.

Pros & Cons

- Mathematically exact, expressible, and analyzable
- If the problem is relatively small (few states and few actions), DP methods might be the best
- May not be easy to use in continuous actions and states
- To calculate updates environment model is required
- Can get samples from this distribution by having an agent interacting with the environment and collecting experience

Dynamic Programming

To solve a given MDP, the solution must have the components to:

1. Find out how good an arbitrary policy is
2. Find out the optimal policy for the given MDP

Policy Evaluation (Prediction)

$$
\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s',r} p(s', r \mid s, a) \Big[r + \gamma v_\pi(s') \Big],
\end{aligned}
$$

$$
\begin{aligned}
v_{k+1}(s) &\doteq \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s',r} p(s', r \mid s, a) \Big[r + \gamma v_k(s') \Big],
\end{aligned}
$$

Iterative Policy Evaluation, for estimating $V \approx v_\pi$

Input π, the policy to be evaluated

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation

Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\textit{terminal}) = 0$

Loop:

$\Delta \leftarrow 0$

Loop for each $s \in \mathcal{S}$:

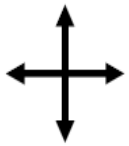
$v \leftarrow V(s)$

$V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

until $\Delta < \theta$

1	2	3	4
5	6 	7	8
9	10	11	12
13	14	15	16

actions 

Reward is -1 for all transition

$$v_1(6) = \sum_{a\in\{u,d,l,r\}} \pi(a|6) \sum_{s',r} p(s',r|6,a)[r + \gamma v_0(s')]$$

$$= \sum_{a\in\{u,d,l,r\}} \underbrace{\pi(a|6)}_{=0.25\ \forall a} \sum_{s'} p(s'|6,a)[\underbrace{r}_{=-1} + \gamma \underbrace{v_0(s')}_{=0\ \forall s'}]$$

$$= 0.25 * \{-p(2|6,u) - p(10|6,d) - p(5|6,l) - p(7|6,r)\}$$

$$= 0.25 * \{-1 - 1 - 1 - 1\}$$

$$= -1$$

$$\Rightarrow v_1(6) = -1$$

$k = 0$

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

$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

$k = 2$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

$k = 3$

0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$\cdots$

$k = 10$

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

$\cdots$

$k = \infty$

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

$\leftarrow v_\pi$

Policy Improvement

for some state s, we want to understand what is the impact of taking an action *a* that does not pertain to policy π.
Let's say we select *a* in *s*, and after that we follow the original policy π.

$$\begin{aligned} q_\pi(s,a) &\doteq \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_\pi(s')\Big]. \end{aligned}$$

If this happens to be greater than the value function $v_\pi(s)$, it implies that the new policy π' would be better to take.
We do this iteratively for all states to find the best policy.

That this is true is a special case of a general result called the *policy improvement theorem*. Let π and π' be any pair of deterministic policies such that, for all $s \in \mathcal{S}$,

$$q_\pi(s, \pi'(s)) \geq v_\pi(s). \tag{4.7}$$

Then the policy π' must be as good as, or better than, π. That is, it must obtain greater or equal expected return from all states $s \in \mathcal{S}$:

$$v_{\pi'}(s) \geq v_\pi(s). \tag{4.8}$$

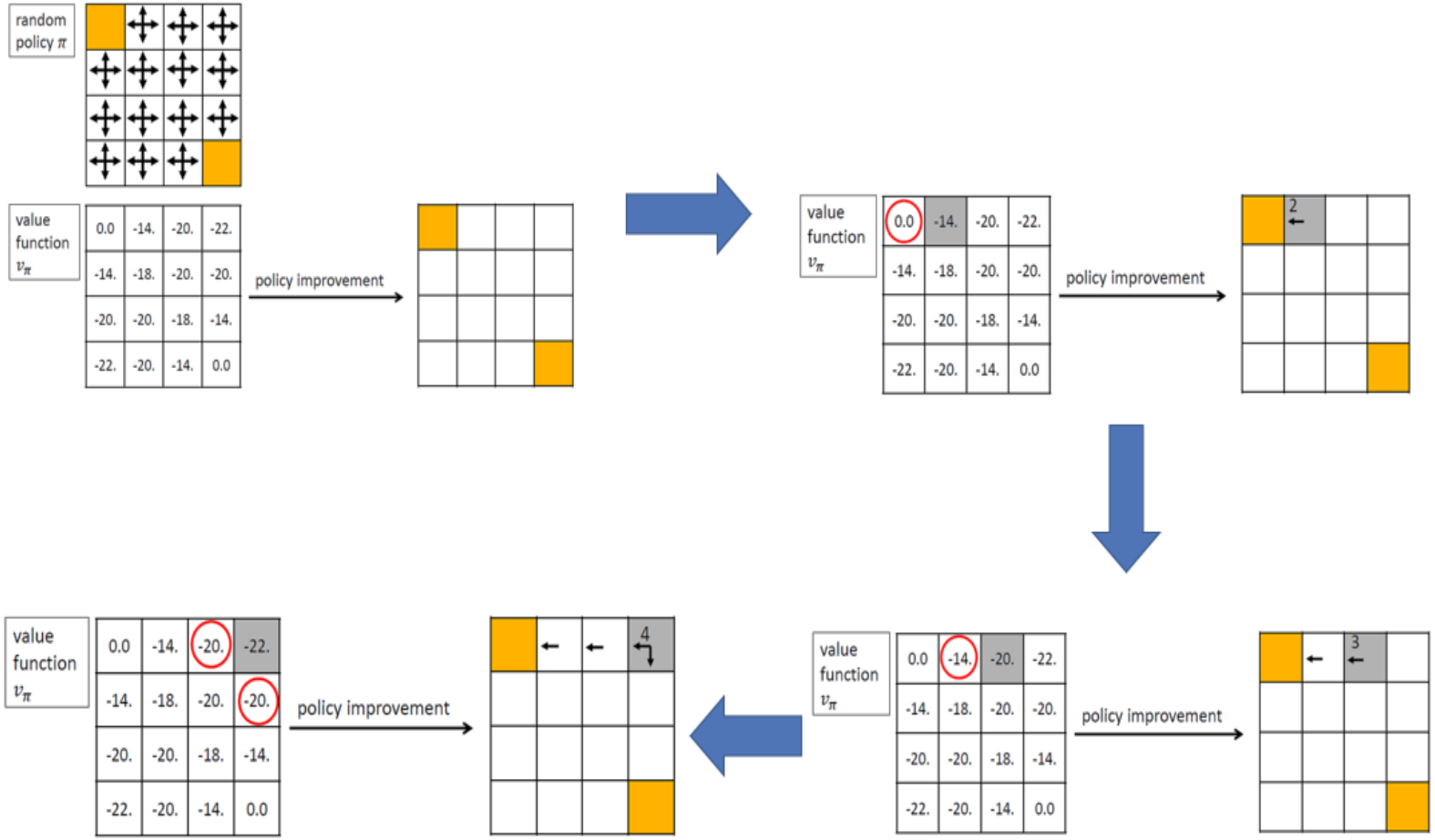
Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
      $\Delta \leftarrow 0$
      Loop for each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s)) \left[r + \gamma V(s') \right]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      old-action $\leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma V(s') \right]$
      If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

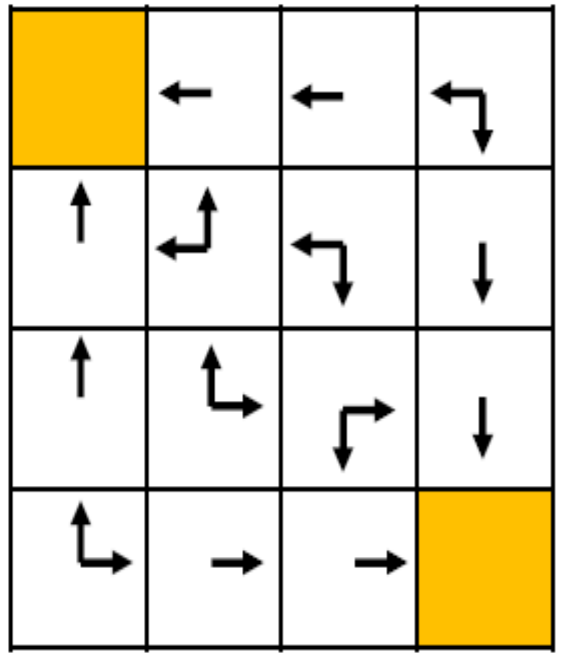
Policy Iteration



Policy Iteration

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

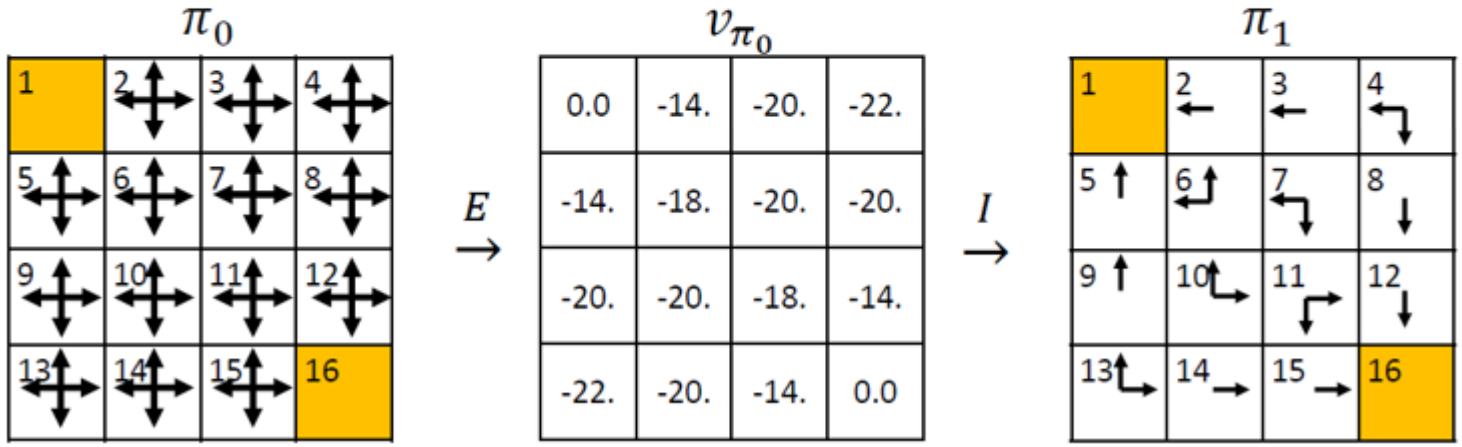
policy improvement



Policy Iteration

Overall, after the policy improvement step using v_π, we get the new policy π':

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \ldots \xrightarrow{I} \pi_* \xrightarrow{E} v_*$$



Value Iteration

This algorithm is called *value iteration*. It can be written as a particularly simple update operation that combines the policy improvement and truncated policy evaluation steps:

$$\begin{aligned} v_{k+1}(s) &\doteq \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma v_k(s')\Big], \end{aligned} \tag{4.10}$$

for all $s \in \mathcal{S}$. For arbitrary v_0, the sequence $\{v_k\}$ can be shown to converge to v_* under the same conditions that guarantee the existence of v_*.

- *does a single iteration of policy evaluation at each step*
- *Then, for each state, it takes the maximum action value to be the estimated state value*

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$

Asynchronous DP

Major drawback is it involves operations over the entire state set of the MDP, that is, they require sweeps of the state set

Asynchronous DP

- are in-place iterative DP algorithms that do not do systematic sweeps of the state set
- Update the values of states in any order whatsoever, using whatever values of other states are available
- The values of some states may be updated several times before the values of others are updated once

Asynchronous algorithms

- make it easier to intermix computation with real-time interaction
- To solve a given MDP, we can run an iterative DP algorithm at the same time that an agent is actually experiencing the MDP.

Efficiency of Dynamic Programming

DP is thought to be of limited applicability because of the curse of dimensionality, the number of states often grows exponentially with the number of state variables.

DP method is

- guaranteed to find an optimal policy in polynomial time even though the total number of (deterministic) policies is k^n
- If n and k denote the number of states and actions, number of computational operations is less than some polynomial function of n and k.

Linear programming methods can also be used to solve MDPs, and in some cases, their worst-case convergence guarantees are better than those of DP methods

References

1. Russell S., and Norvig P., Artificial Intelligence A Modern Approach (3e), Pearson 2010

2. Richard S Sutton, Andrew G Barto, Reinforcement Learning, second edition, MIT Press

3. https://www.coursera.org/learn/fundamentals-of-reinforcement-learning/home/week/1

Artificial Intelligence
DSE 3252

Reinforcement Learning -2

ROHINI R RAO
DEPT OF DATA SCIENCE & COMPUTER APPLICATIONS

JANUARY 2024

Monte Carlo Methods

The term “Monte Carlo” is used for any estimation method whose operation involves a significant random component.

In Reinforcement Learning it is used for methods based on averaging complete returns

- Does not require complete knowledge of the environment
- Does not require prior knowledge of the environment’s dynamics

Although a model is required, the model need only generate sample transitions, not the complete probability distributions of all possible transitions that is required for dynamic programming (DP)

Monte Carlo methods require only experience

- Sample sequences of states, actions, and rewards from actual or simulated interaction with an environment.
- Learning from actual experience, yet can still attain optimal behavior.

Monte Carlo Prediction

Monte Carlo methods can thus be incremental in an episode-by-episode sense, but not in a step-by-step (online) sense

Episodic Tasks :

- To ensure that well-defined returns are available, we assume experience is divided into episodes
- all episodes eventually terminate no matter what actions are selected
- Only on the completion of an episode are value estimates and policies changed

MC Methods

- suppose we wish to estimate $v_{\Pi}(s)$, the value of a state s under policy Π,
- given a set of episodes obtained by following Πand passing through state s.
- Each occurrence of state s in an episode is called a visit to state s.
- States may be visited multiple times in the same episode
- The first time it is visited in an episode is called the first visit to state s.
- **Types of MC methods** are
  - First-visit MC method estimates $v_{\Pi}(s)$, as the average of the returns following first visits to s
  - Every-visit MC method averages the returns following all visits to s.

Monte Carlo Policy Evaluation

Return is the total discounted reward:

Trajectory in episode is

T: *S1, A1, R2, ….Sk ~*

The value function is the expected return

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s]$$

Estimate any expected value simply by adding up samples and dividing by the total number of samples

- i – Episode index
- s – Index of state

$$\bar{V}_\pi(s) = \frac{1}{N} \sum_{i=1}^{N} G_{i,s}$$

First Visit Monte Carlo

Average returns only for the first time s is visited in an episode

1. Initialize the policy, state-value function
2. Start by generating an episode according to the current policy
    1. Keep track of the states encountered through that episode
3. Select a state in 2.1
    1. Add to a list the return received after first occurrence of this state
    2. Average over all returns
    3. Set the value of the state as that computed average
4. Repeat step 3
5. Repeat 2-4 until satisfied

First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated

Initialize:

$V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$

$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:

Append G to $Returns(S_t)$

$V(S_t) \leftarrow \text{average}(Returns(S_t))$

Every visit Monte Carlo

1. Initialize the policy, state-value function
2. Start by generating an episode according to the current policy
    1. Keep track of the states encountered through that episode
3. Select a state in 2.1
    1. Add to a list the return received after every occurrence of this state.
    2. Average over all returns
    3. Set the value of the state as that computed average
4. Repeat step 3
5. Repeat 2-4 until satisfied

Computation of Return

$$A+3 \rightarrow A+2 \rightarrow B-4 \rightarrow A+4 \rightarrow B-3 \rightarrow \text{terminate}$$

$$B-2 \rightarrow A+3 \rightarrow B-3 \rightarrow \text{terminate}$$

First visit	Every visit
V(A) = 1/2(2 + 0) = 1 V(B) = 1/2(-3 + -2) = -5/2	V(A) = 1/4(2 + -1 + 1 + 0) = 1/2 V(B) = 1/4(-3 + -3 + -2 + -3) = -11/4

Incremental Mean & Policy Improvement

We update v(s) incrementally after episodes. For each state St, with return Gt:

$$N(S_t) \leftarrow N(S_t) + 1$$

$$V(S_t) \leftarrow V(S_t) + \frac{1}{N(S_t)}(G_t - V(S_t))$$

In non-stationary problems, it can be useful to track a running mean, i.e., forget old episodes:

$$V(S_t) \leftarrow V(S_t) + \alpha\,(G_t - V(S_t))$$

Policy improvement is done by making the policy greedy with respect to the current value function.

$$\pi(s) \doteq \arg\max_a q(s,a)$$

Monte Carlo Estimation of Action Values

Dynamic Programming (With a model)

- state values alone are sufficient to determine a policy;
- looks ahead one step and chooses whichever action leads to the best combination of reward and next state

Monte Carlo Methods (Without a model)

- however, state values alone are not sufficient
- One must explicitly estimate the value of each action in order for the values to be useful in suggesting a policy.

Thus, primary goal for Monte Carlo methods is to estimate q_*.

To achieve consider the policy evaluation problem for action values.

Policy Evaluation Phase

The policy evaluation problem for action values

- is to estimate $q_{\Pi}(s, a)$, the expected return when starting in state s, taking action a, following policy Π.

Issues include

- Many state–action pairs may never be visited.
- If Π is a deterministic policy, then in following Π one will observe returns only for one of the actions from each state.
- With no returns to average, the Monte Carlo estimates of the other actions will not improve with experience.

For policy evaluation to work for action values, **continual exploratio**n is required.

Exploring Starts Assumption

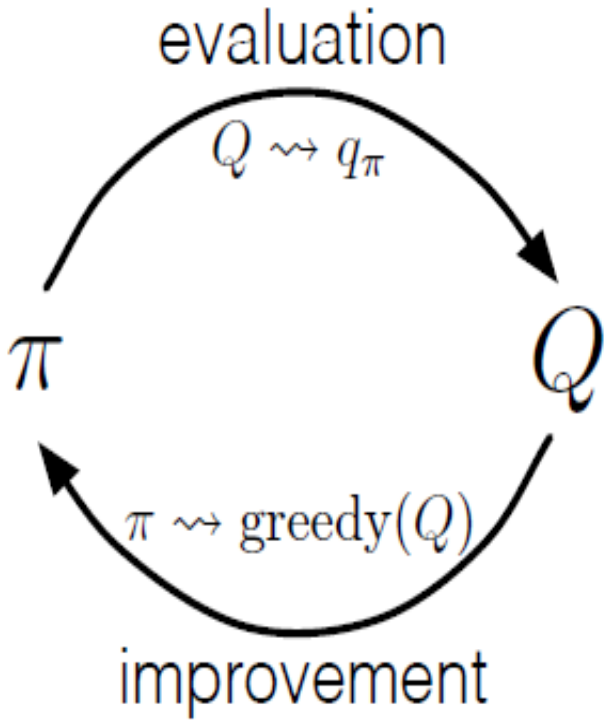
- Specify that the episodes start in a state–action pair, and that every pair has a nonzero probability of being selected as the start.
- This guarantees that all state–action pairs will be visited an infinite number of times in the limit of an infinite number of episodes.

Monte Carlo Control

this method, we perform alternating complete steps of policy evaluation and policy improvement, beginning with an arbitrary policy π_0 and ending with the optimal policy and optimal action-value function:

$$\pi_0 \xrightarrow{\text{E}} q_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} q_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} q_*,$$

where $\xrightarrow{\text{E}}$ denotes a complete policy evaluation and $\xrightarrow{\text{I}}$ denotes a complete policy



Generalised Policy Iteration (GPI)

Monte Carlo Control

- Policy Improvement is a greedy policy with respect to current value function, which is action-value function and therefore no model is required to construct the greedy policy
- For any action-value function q, the corresponding greedy policy is the one that for each state s, deterministically chooses an action with maximal action value:

$$\pi(s) \doteq \arg\max_a q(s,a).$$

$$\begin{aligned} q_{\pi_k}(s, \pi_{k+1}(s)) &= q_{\pi_k}(s, \arg\max_a q_{\pi_k}(s,a)) \\ &= \max_a q_{\pi_k}(s,a) \\ &\geq q_{\pi_k}(s, \pi_k(s)) \\ &\geq v_{\pi_k}(s). \end{aligned}$$

Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

Initialize:

$\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$

$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

$Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Loop forever (for each episode):

Choose $S_0 \in \mathcal{S}$, $A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0

Generate an episode from S_0, A_0, following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$

On Policy vs Off Policy

On-policy method

- attempt to evaluate or improve the same Policy that is used to make decisions.
- For Example:
- *Trying different restaurants in an area to search for the best restaurant.*
- *The agent evaluates the decision as well as improve the decision*

Off-policy method

- uses a behavioral policy to explore the environment and to collect samples generating Agent's behavior
- and a second policy being learned called the target policy which is optimized.
- For Example:
- *Follow Google's recommendations to search for the best restaurant in the area (Behavioral policy)*
- *Agent decides restaurant (Target policy)*

On-Policy can be used for model-based and model-free reinforcement learning

Off-policy is used for model-free reinforcement learning algorithms

Monte Carlo Control with Epsilon-soft

On-policy methods

- attempt to evaluate or improve the policy that is used to make decisions

In on-policy control methods the policy is generally soft

- $\Pi(a|s) > 0$ for all $s \in S$ and all $a \in A(s)$
- but gradually shifted closer and closer to a deterministic optimal policy

ε- greedy policies

- most of the time they choose an action that has maximal estimated action value
- but with probability ε they instead select an action at random

Monte Carlo policy iteration

- alternate between evaluation and improvement on an episode-by-episode basis
- After each episode, the observed returns are used for policy evaluation
- and then the policy is improved at all the states visited in the episode

On-policy first-visit MC control (for ε-soft policies), estimates $\pi \approx \pi_*$

Algorithm parameter: small $\varepsilon > 0$

Initialize:

$\pi \leftarrow$ an arbitrary ε-soft policy

$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

$Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Repeat forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$A^* \leftarrow \text{argmax}_a\, Q(S_t, a)$ (with ties broken arbitrarily)

For all $a \in \mathcal{A}(S_t)$:

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

//Non-greedyaction with the minimal probability of selection

Policy Improvement

That any ε-greedy policy with respect to q_π is an improvement over any ε-soft policy π is assured by the policy improvement theorem. Let π' be the ε-greedy policy. The conditions of the policy improvement theorem apply because for any $s \in \mathcal{S}$:

$$\begin{aligned} q_\pi(s, \pi'(s)) &= \sum_a \pi'(a|s) q_\pi(s,a) \\ &= \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) + (1-\varepsilon) \max_a q_\pi(s,a) \qquad (5.2) \\ &\geq \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) + (1-\varepsilon) \sum_a \frac{\pi(a|s) - \frac{\varepsilon}{|\mathcal{A}(s)|}}{1-\varepsilon} q_\pi(s,a) \end{aligned}$$

(the sum is a weighted average with nonnegative weights summing to 1, and as such it must be less than or equal to the largest number averaged)

$$\begin{aligned} &= \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) - \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) + \sum_a \pi(a|s) q_\pi(s,a) \\ &= v_\pi(s). \end{aligned}$$

Temporal Difference Learning

TD Methods

- can learn directly from raw experience without a model of the environment's dynamics
- update estimates based in part on other learned estimates, without waiting for a final outcome (bootstrap)

A simple every-visit Monte Carlo method suitable for nonstationary environments is

$$V(S_t) \leftarrow V(S_t) + \alpha \Big[G_t - V(S_t) \Big]$$

The simplest TD method makes the update

$$V(S_t) \leftarrow V(S_t) + \alpha \Big[R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \Big]$$

immediately on transition to S_{t+1} and receiving R_{t+1}. In effect, the target for the Monte Carlo update is G_t, whereas the target for the TD update is $R_{t+1} + \gamma V(S_{t+1})$. This TD method is called *TD(0)*, or *one-step TD*, because it is a special case of the TD(λ) and

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated

Algorithm parameter: step size $\alpha \in (0, 1]$

Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\textit{terminal}) = 0$

Loop for each episode:

  Initialize S

  Loop for each step of episode:

    $A \leftarrow$ action given by π for S

    Take action A, observe R, S'

    $V(S) \leftarrow V(S) + \alpha\left[R + \gamma V(S') - V(S)\right]$

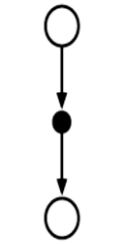
    $S \leftarrow S'$

  until S is terminal

TD combines DP and MC methods

TD combines the sampling of MC with Bootstrapping of DP

$$
\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] .
\end{aligned}
$$



TD(0)

Sample updates

- because they involve looking ahead to a sample successor state (or state–action pair)
- Differ from expected updates of DP methods in that they are based on a single sample successor rather than on a complete distribution of all possible successors.

TD Error

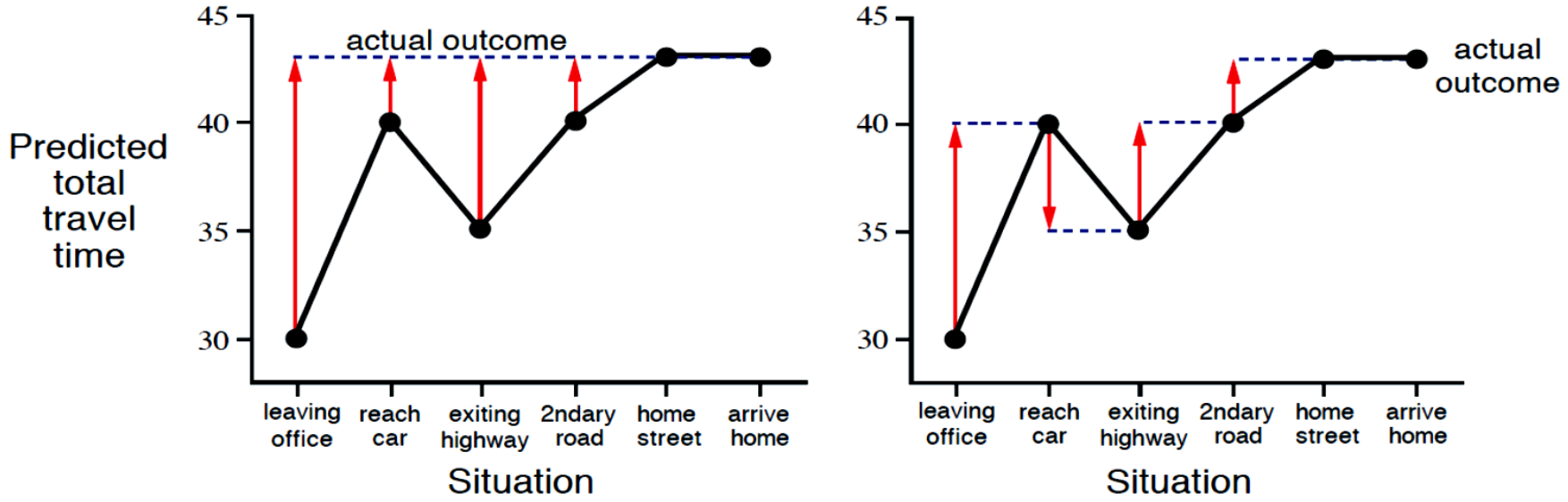
- at each time is the error in the estimate made at that time.
- depends on the next state and next reward, it is not actually available until one time step later

$$\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

State	Elapsed Time (minutes)	Predicted Time to Go	Predicted Total Time
leaving office, friday at 6	0	30	30
reach car, raining	5	35	40
exiting highway	20	15	35
2ndary road, behind truck	30	10	40
entering home street	40	3	43
arrive home	43	0	43

- The rewards is the elapsed time on each leg of the journey
- If not discounting then the return for each state is the actual time to go from that state.
- The value of each state is the predicted time to go

Changes recommended in the driving home example by Monte Carlo methods vs TD methods



Advantages of TD

TD methods better than:

DP methods do not require a model of the environment, of its reward and next-state probability distributions

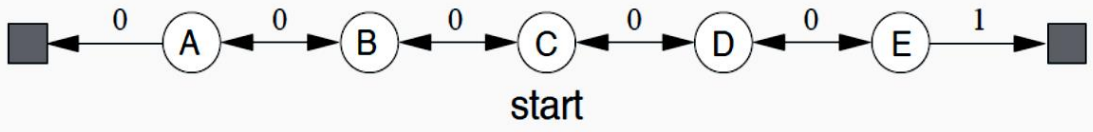
Monte Carlo methods

- as they can be implemented in an online, fully incremental fashion
- Need to wait only 1 time step for updation

both TD and Monte Carlo methods converge asymptotically to the correct predictions

Random Walk

In this example we empirically compare the prediction abilities of TD(0) and constant-α MC when applied to the following Markov reward process:



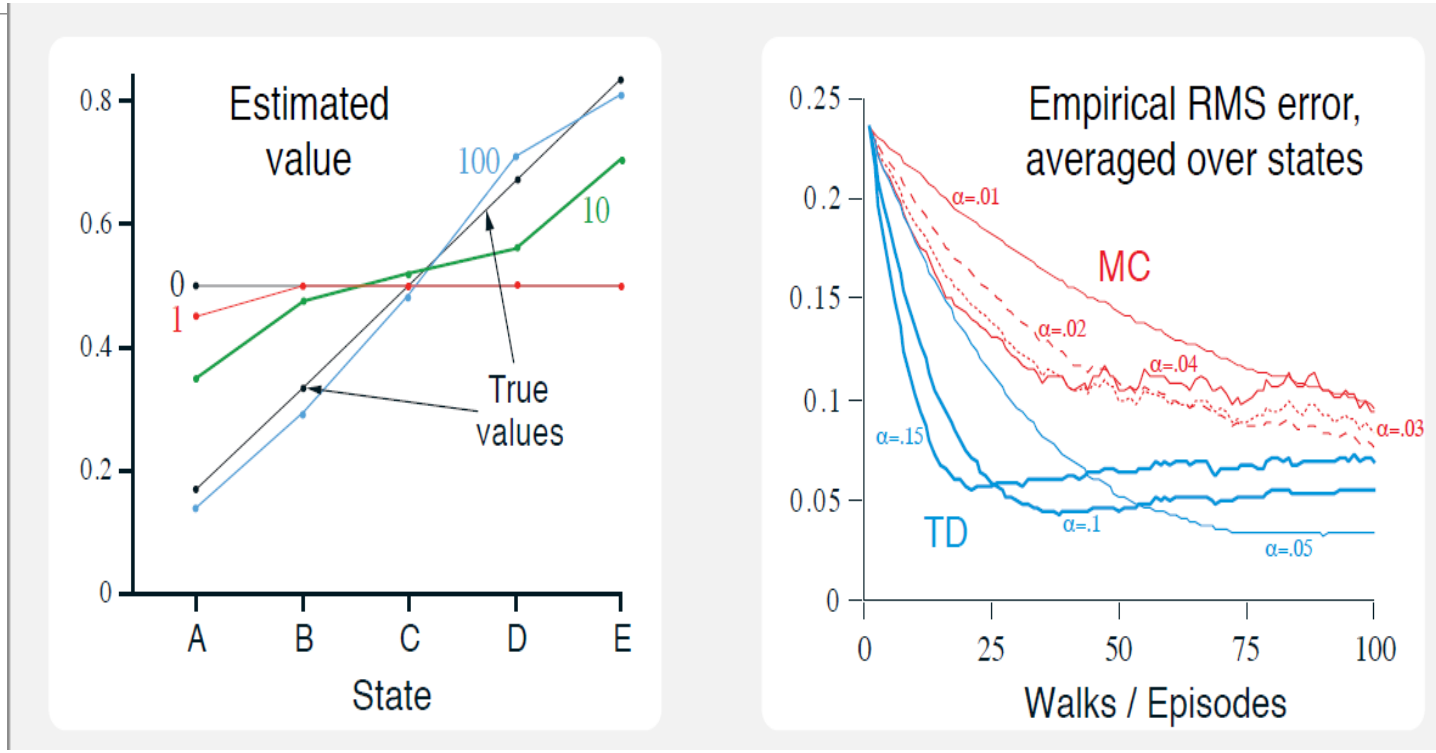
- all episodes start in the center state, C, then proceed either left or right by one state on each step, with equal probability.
- Episodes terminate either on the extreme left or the extreme right.
- When an episode terminates on the right, a reward of +1 occurs
- all other rewards are zero.

For example

Episode : C, 0,B, 0, C, 0,D, 0, E, 1.

- If this task is undiscounted, the true value of each state is the probability of terminating on the right if starting from that state.
- Thus, the true value of the center state is $v_{\pi}(C) = 0.5$.
- The true values of all the states, A through E, are 1/6, 2/6,3/6,4/6,5/6

Random Walk



The performance measure shown is the root mean-squared (RMS) error between the value function learned and the true value function, averaged over the five states, then averaged over 100 runs

Optimality of TD(0)

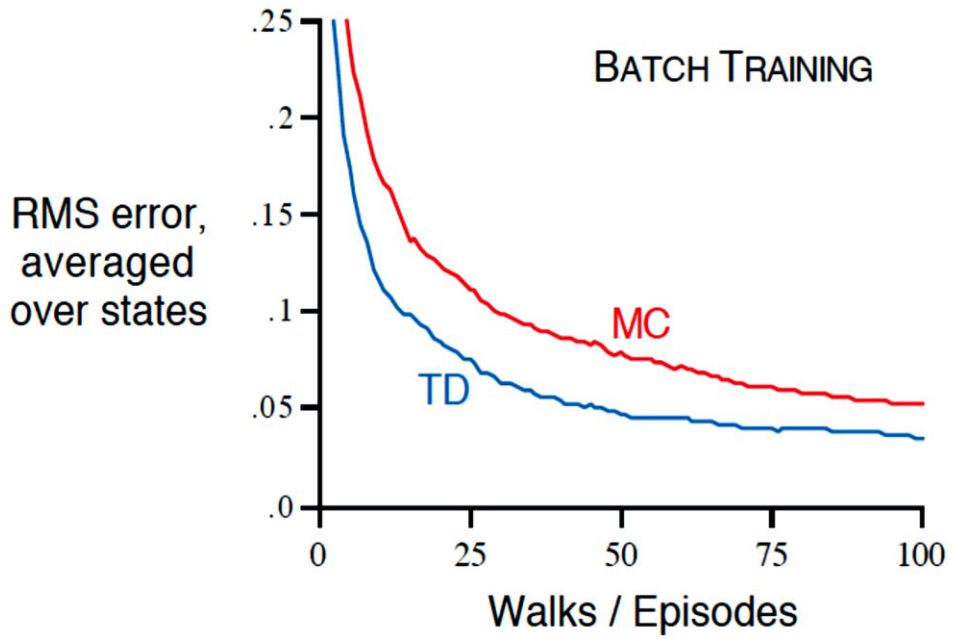
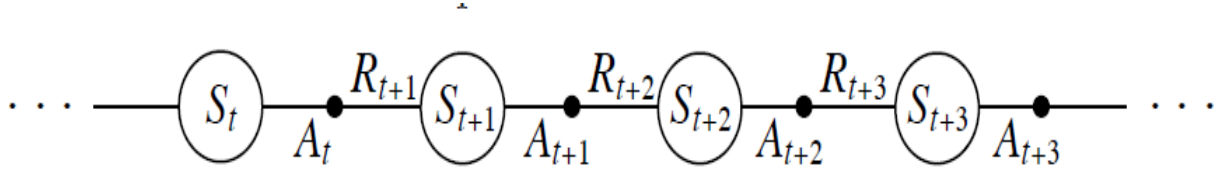


Figure 6.2: Performance of TD(0) and constant-α MC under batch training on the random walk task.

Batch updating

- because updates are made only after processing each complete batch of training data
- Suppose there is available only a finite amount of experience, say 100 time steps
- common approach with incremental learning methods is to present the experience repeatedly until the method converges upon an answer
- Given an approximate value function, V , the increments are computed for every time step t at which a nonterminal state is visited
- but the value function is changed only once, by the sum of all the increments
- Then all the available experience is processed again with the new value function to produce a new overall increment, and so on, until the value function converges

Sarsa: On-policy TD Control



- for an on-policy method we must estimate $q_{\Pi}(s, a)$ for the current behavior policy Π and for all states s and actions a.
- The theorems assuring the convergence of state values under TD(0) also apply to the corresponding algorithm for action values:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\Big].$$

This update is done after every transition from a nonterminal state St.
If S_{t+1} is terminal, then $Q(S_{t+1},A_{t+1})$ is defined as zero.
This rule uses every element of the quintuple of events, $(S_t,A_t,R_{t+1}, S_{t+1},A_{t+1})$, that make up a transition from one state–action pair to the next.
This quintuple gives rise to the name Sarsa for the algorithm

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$

Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:

Initialize S

Choose A from S using policy derived from Q (e.g., ε-greedy)

Loop for each step of episode:

Take action A, observe R, S'

Choose A' from S' using policy derived from Q (e.g., ε-greedy)

$$Q(S, A) \leftarrow Q(S, A) + \alpha\left[R + \gamma Q(S', A') - Q(S, A)\right]$$

$S \leftarrow S'$; $A \leftarrow A'$;

until S is terminal

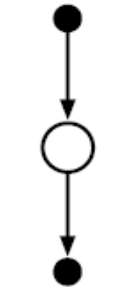
On-policy control algorithm based on Sarsa prediction method

On Policy :

- continually estimate q_{Π} for the behavior policy Π
- and at the same time change Π toward greediness with respect to q_{Π}.

ε-greedy or ε-soft policies

- SARSA converges with probability 1 to an optimal policy
- Action-value function as long as all state–action pairs are visited an infinite number of times
- policy converges in the limit to the greedy policy
- (ε = 1/t)



Sarsa

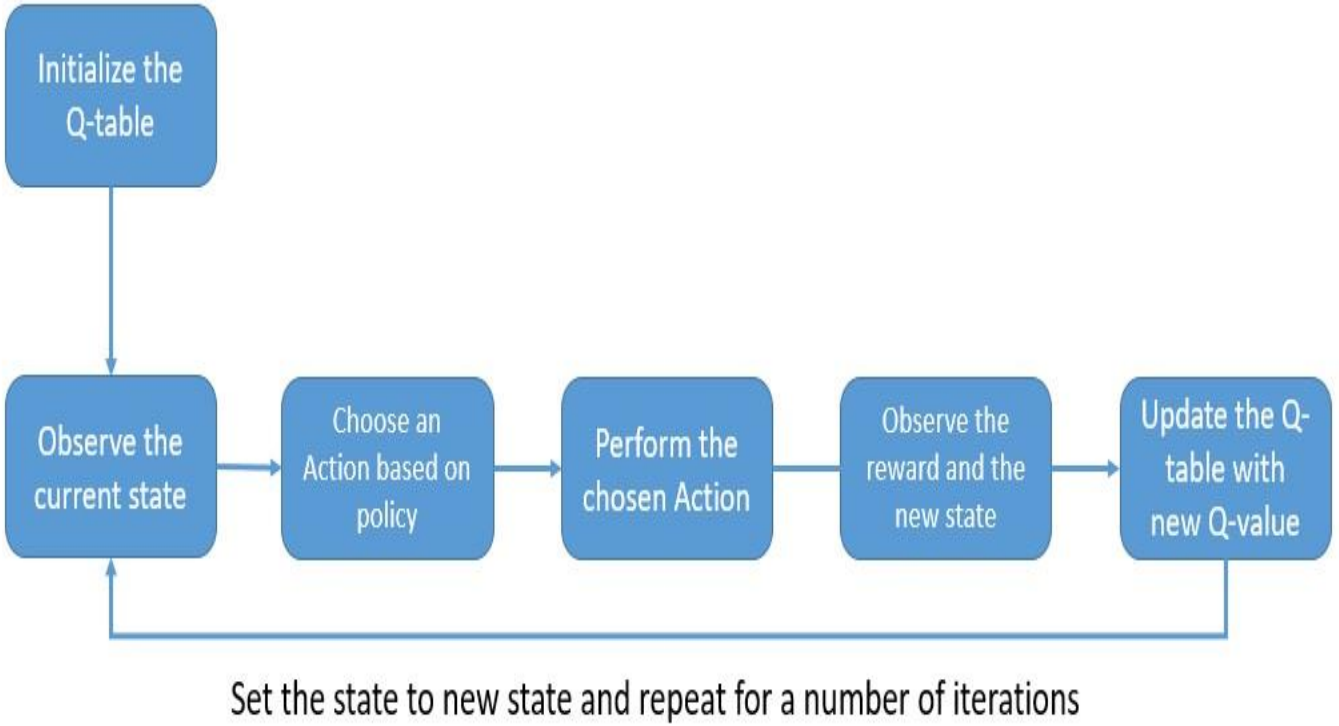
Q-Learning: Off Policy TD Control

One of the early breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as Q-learning (Watkins, 1989)

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right].$$

- The learned action-value function, Q, directly approximates q_*, the optimal action-value function, independent of the policy being followed.
- This dramatically simplifies the analysis of the algorithm and enabled early convergence proofs.
- The policy still has an effect in that it determines which state–action pairs are visited and updated.
- However, all that is required for correct convergence is that all pairs continue to be updated.

Q-Learning Process



- The q-values are stored and updated in a **q-table**
- dimensions matching the number of actions and states in the environment

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$

Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:

  Initialize S

  Loop for each step of episode:

    Choose A from S using policy derived from Q (e.g., ε-greedy)

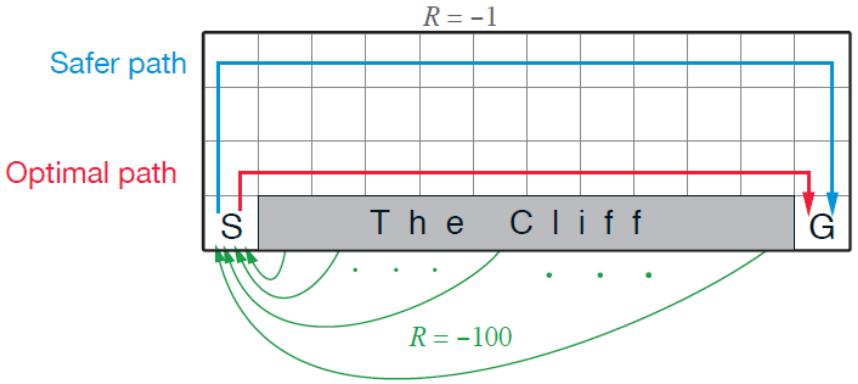
    Take action A, observe R, S'

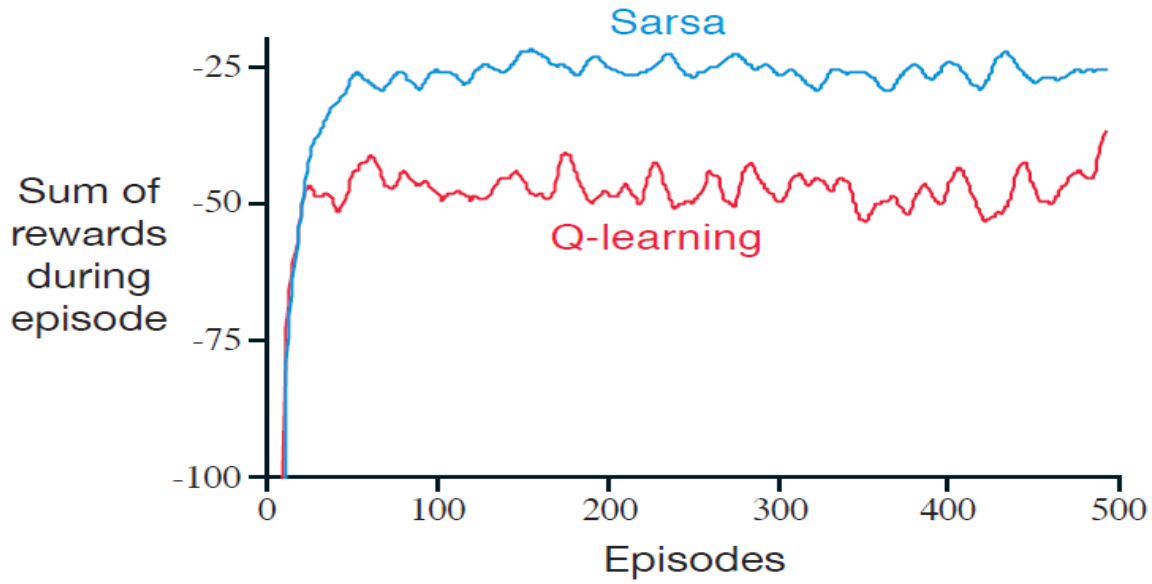
    $Q(S, A) \leftarrow Q(S, A) + \alpha\left[R + \gamma \max_a Q(S', a) - Q(S, A)\right]$

    $S \leftarrow S'$

  until S is terminal

Cliff Walking





Comparison of QL and SARSA

- Both approach work in
  - finite environment
  - or discretized continuous environment
- QL directly learns the optimal policy while SARSA learns a “near” optimal policy.
- QL is a more aggressive agent, while SARSA is more conservative.
  - Example- In Cliff Walking
  - QL will take the shortest path because it is optimal while SARSA will take the longer, safer route
- In practice
  - For fast-iterating environment, QL should be your choice
  - If mistakes are costly , then SARSA is the better option
  - If state space is too large, Use deep q network
    - For example: Ms. Pac-Man has 150 pellets which can be present or absent
    - The number of possible states is for pellets only 2^{150}
    - Add all possible combinations of positions for all ghosts- so very difficult to estimate q-value

Expected SARSA

Instead of the maximum over next state–action pairs it uses the expected value, taking into account how likely each action is under the current policy.

The Update Rule is

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \mathbb{E}_\pi[Q(S_{t+1}, A_{t+1}) \mid S_{t+1}] - Q(S_t, A_t)\Big] \\ &\leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)\Big], \end{aligned}$$

Given the next state, S_{t+1}, this algorithm moves deterministically in the same direction as SARSA moves in expectation

Expected SARSA is more complex computationally than SARSA but, in return, it eliminates the variance due to the random selection of A_{t+1}

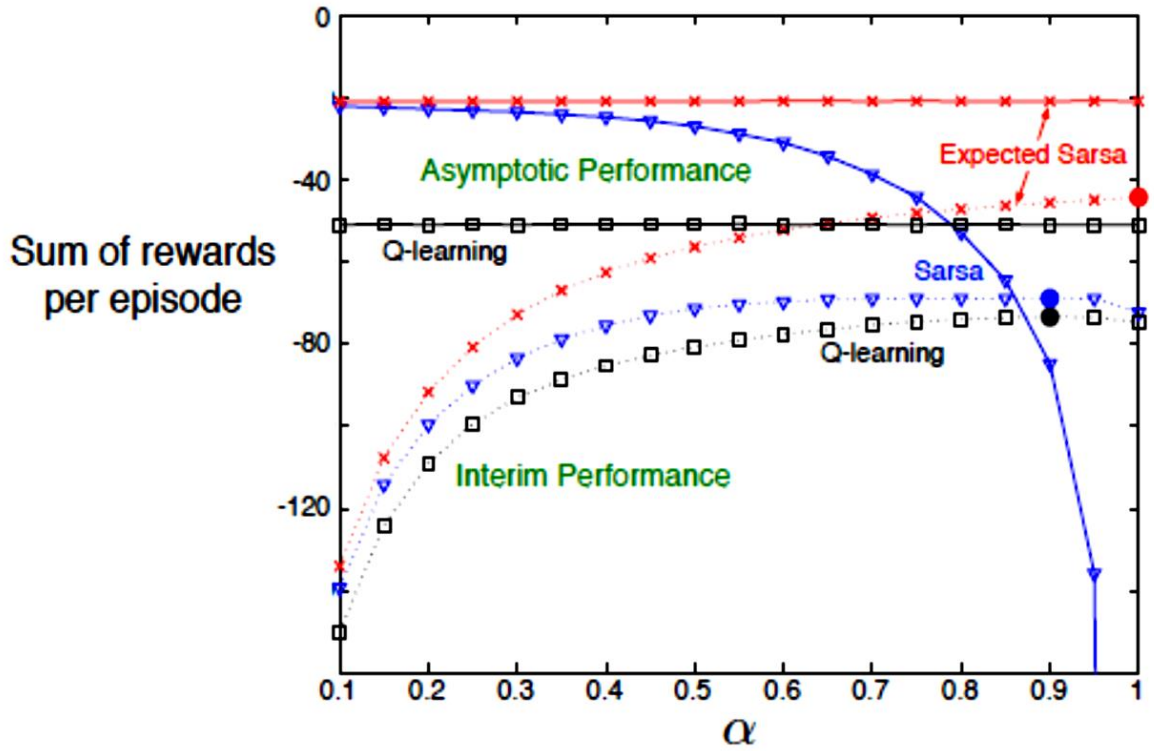
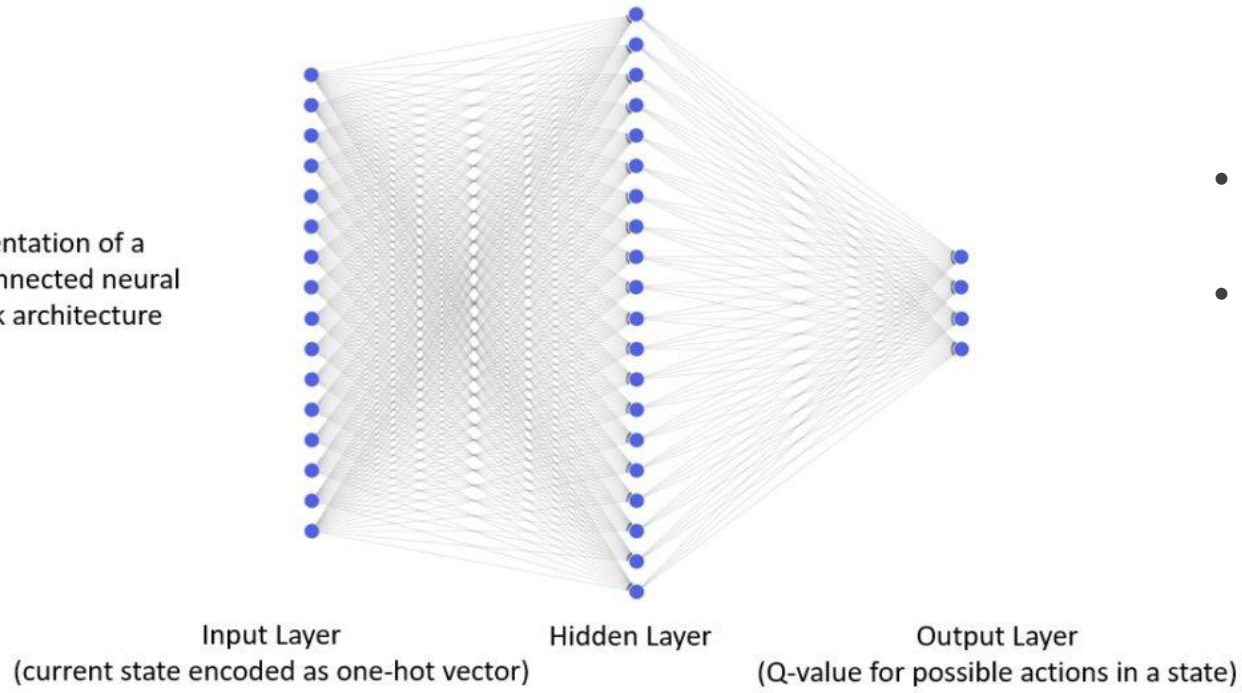


Figure 6.3: Interim and asymptotic performance of TD control methods on the cliff-walking task as a function of α. All algorithms used an ε-greedy policy with $\varepsilon = 0.1$. Asymptotic performance is an average over 100,000 episodes whereas interim performance is an average over the first 100 episodes. These data are averages of over 50,000 and 10 runs for the interim and asymptotic cases respectively. The solid circles mark the best interim performance of each method. Adapted from van Seijen et al. (2009).

Reinforcement Learning with Neural Networks

- The number of actions and states in a real-life environment can be thousands, making it extremely inefficient to manage q-values in a table
- can use neural networks to predict q-values for actions in a given state instead of using a table.

Representation of a fully-connected neural network architecture



- ReLU as the activation function in the hidden layer
- Linear activation function in the output layer

Reinforcement Learning with Neural Networks

The Loss Function and Optimizer

The mean-squared-error loss function measures the square value of the difference between the prediction and the target:

$$\text{loss} = \{(\text{r} + \gamma * max_{a'} Q'(s', a')) - Q(s, a)\}^2$$

take the feedback backward through the network and update the weights

Simplest Solution is Backpropagation with the classical stochastic gradient descent

Reinforcement Learning with Neural Networks

Code Blocks:

```
// Parameters
 discount_factor = 0.95
 eps = 0.5
 eps_decay_factor = 0.999
 num_episodes=500

 //Neural Network
    model = Sequential()
    model.add(InputLayer(batch_input_shape=(1, env.observation_space.n)))
    model.add(Dense(20, activation='relu'))
    model.add(Dense(env.action_space.n, activation='linear'))
    model.compile(loss='mse', optimizer='adam', metrics=['mae’])
```

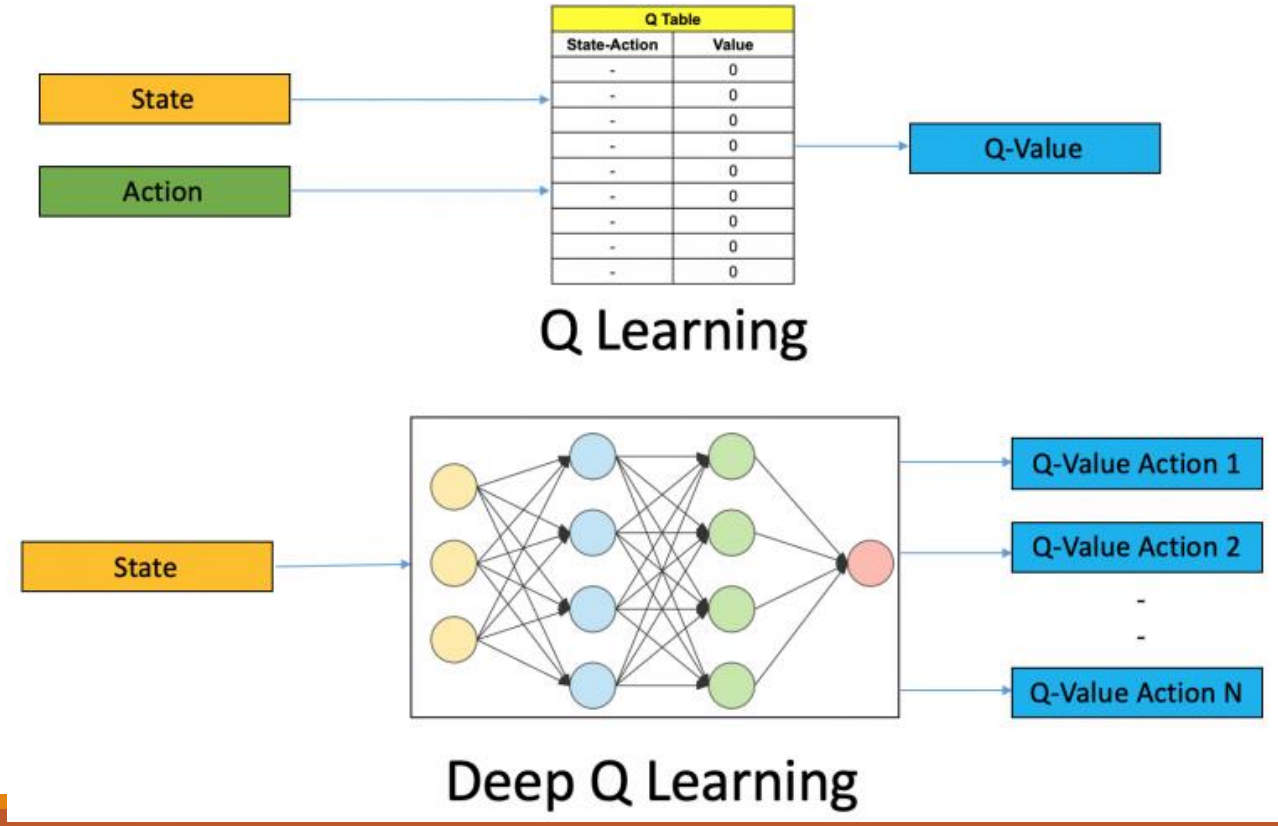
Reinforcement Learning with Neural Networks

```
for i in range(num_episodes):
    state = env.reset()
    eps *= eps_decay_factor
    done = False
    while not done:
        if np.random.random() < eps:     action = np.random.randint(0, env.action_space.n)
        else:
            action = np.argmax(model.predict(np.identity(env.observation_space.n)[state:state + 1]))
        new_state, reward, done, _ = env.step(action)
        target = reward +  discount_factor *
            np.max(model.predict(np.identity(env.observation_space.n)[new_state:new_state + 1]))
        target_vector = model.predict(np.identity(env.observation_space.n)[state:state + 1])[0]
        target_vector[action] = target
        model.fit(np.identity(env.observation_space.n)[state:state + 1],
            target_vec.reshape(-1, env.action_space.n),  epochs=1, verbose=0)
        state = new_state
```

one-hot encoded current state, and the target value converted as a vector to train the model on a single step

Deep Q Learning

- approximate these Q-values with machine learning models such as a neural network
- idea behind DeepMind's algorithm that led to its acquisition by Google for 500 million dollars!



Steps in Deep Q Networks

1. All the past experience is stored by the user in memory
2. The next action is determined by the maximum output of the Q-network
3. The loss function here is mean squared error of the predicted Q-value and the target

   Q-value – Q*

   As per the Q-value update equation derived from the Bellman equation:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

   - green represents the target
   - Since R is the unbiased true reward, the network is going to update its gradient using backpropagation to finally converge

Psuedocode for Deep Q-Learning

Start with $Q_0(s, a)$ for all s, a.

Get initial state s

For k = 1, 2, ... till convergence

Sample action a, get next state s'

If s' is terminal:

$$\text{target} = R(s, a, s')$$

Sample new initial state s'

else:

$$\text{target} = R(s, a, s') + \gamma \max_{a'} Q_k(s', a')$$

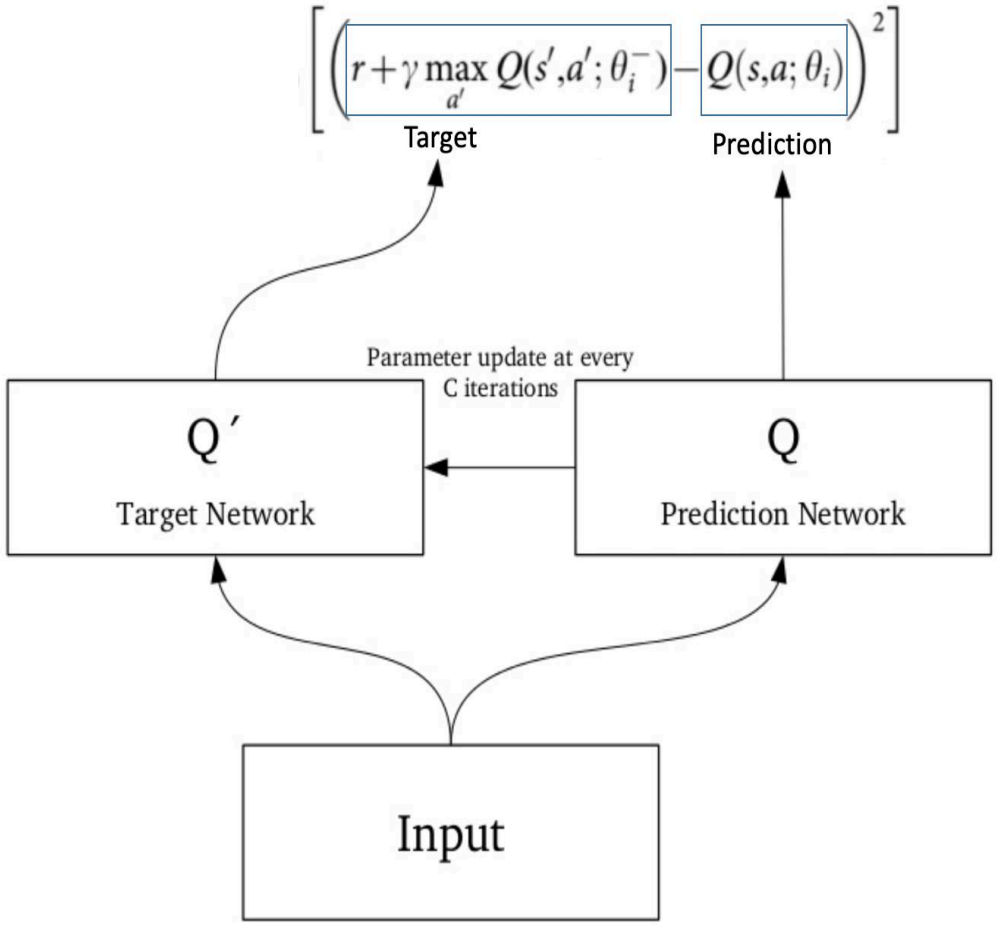
$$\theta_{k+1} \leftarrow \theta_k - \alpha \nabla_\theta \mathbb{E}_{s' \sim P(s'|s,a)} \left[(Q_\theta(s, a) - \text{target}(s'))^2 \right] \Big|_{\theta=\theta_k}$$

$$s \leftarrow s'$$

Chasing a nonstationary target!

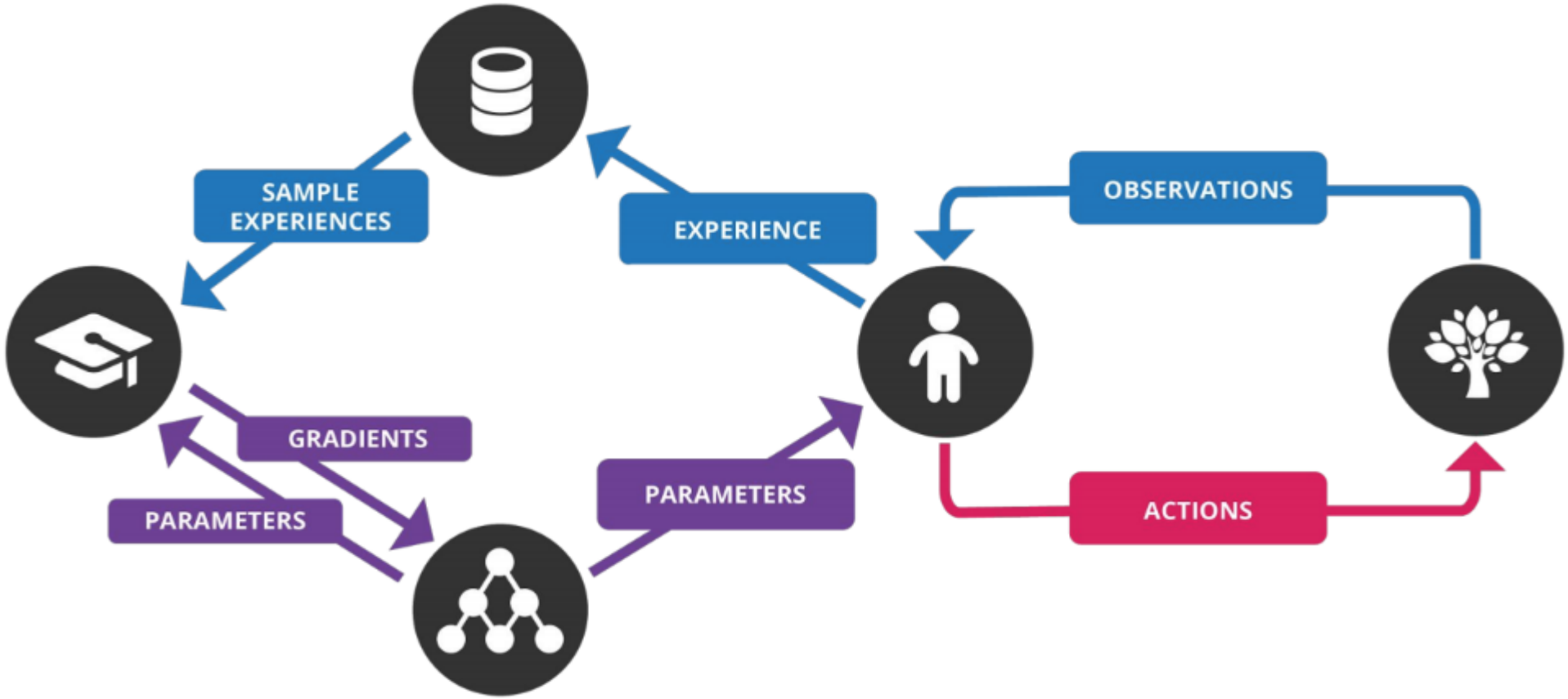
Updates are correlated within a trajectory!

Solution in deep Q-network (DQN)



- Separate network to estimate the target
- **Target network** has the same architecture as the function approximator but with frozen parameters
- For every **C iterations** (a hyperparameter), the parameters from the **prediction network** are copied to the target network.
- This leads to more stable training because it keeps the target function fixed (for a while):
- **Experience Replay**
  - Instead of running Q-learning on state/action pairs as they occur during simulation or the actual experience
  - the system stores the data discovered for [state, action, reward, next_state] – in a large table.

deep Q-network (DQN)



steps involved in deep Q-network (DQN) :

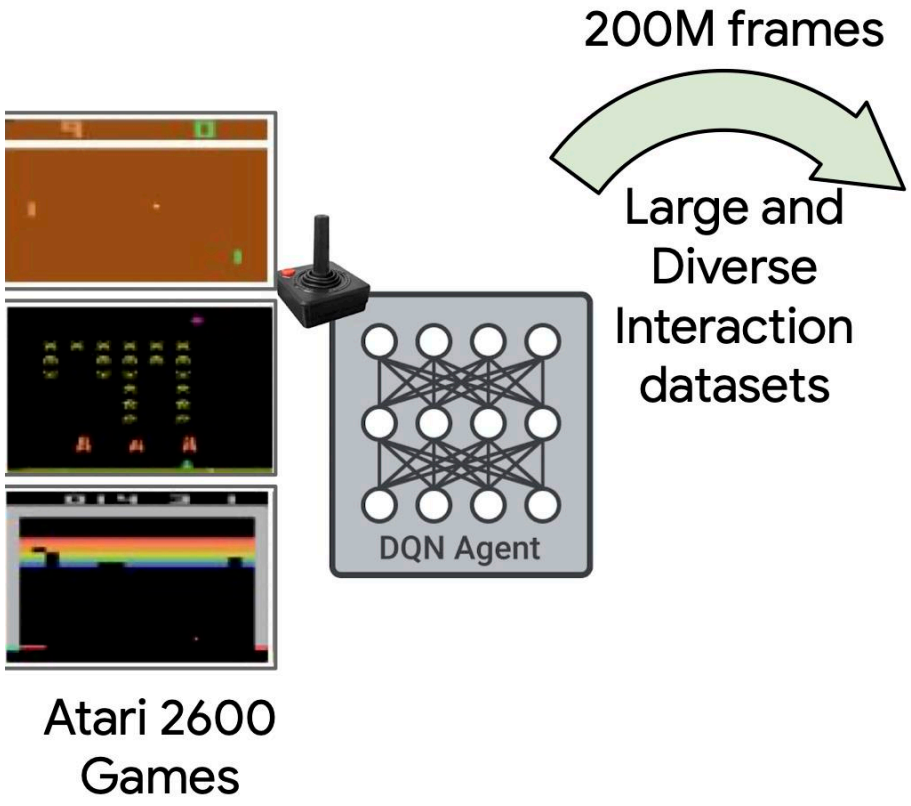
1. Preprocess and feed the game screen (state s) to DQN, which will return the Q-values of all possible actions in the state
2. Select an action using the ε-greedy policy.
3. Perform this action in a state s and move to a new state s’ to receive a reward.
4. This state s’ is the preprocessed image of the next game screen. We store this transition in our replay buffer as <s,a,r,s’>
5. Next, sample some random batches of transitions from the replay buffer and calculate the loss- (squared difference between target Q and predicted Q)
6. Perform gradient descent with respect to our actual network parameters in order to minimize this loss
7. After every C iterations, copy our actual network weights to the target network weights
8. Repeat these steps for M number of episodes

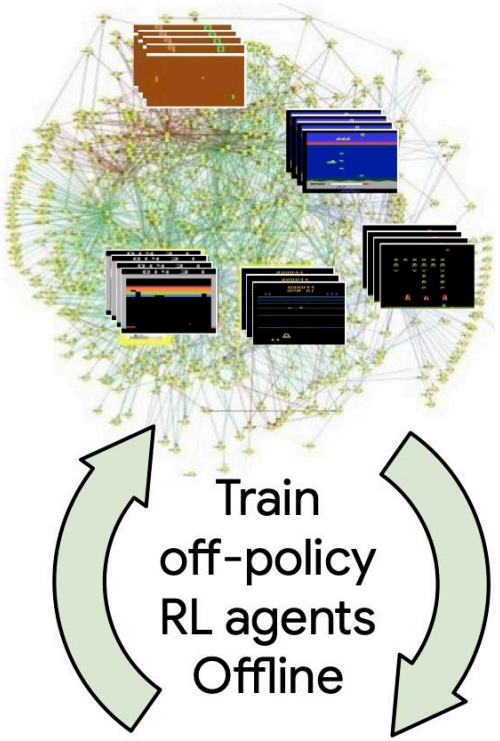
Cartpole & DQN

```
model = Sequential()
model.add(Flatten(input_shape=(1,) + env.observation_space.shape))
model.add(Dense(16))
model.add(Activation('relu'))
model.add(Dense(nb_actions))
model.add(Activation('linear'))
print(model.summary())
```

Cartpole & DQN

```
policy = EpsGreedyQPolicy()

memory = SequentialMemory(limit=50000, window_length=1)

dqn = DQNAgent(model=model, nb_actions=nb_actions, memory=memory,
nb_steps_warmup=10,

target_model_update=1e-2, policy=policy)

dqn.compile(Adam(lr=1e-3), metrics=['mae'])


dqn.fit(env, nb_steps=5000, visualize=True, verbose=2)
```





Discretizing Continuous State Space

In case of Video- Frames are input & CNN is used in the DQN

Cartpole - 5-state discretization of the state space

- S = { hard_lean_left,
- soft_lean_left,
- soft_lean_right,
- hard_lean_right,
- fallen

}

Discretizing Continuous State Space

Rewards

fallen => -1

soft_lean_left => +1

soft_lean_right => +1

hard_lean_left => 0

hard_lean_right => 0

Actions the agent could invoke on the cart in order to balance the pole:

A = { push_left,

push_right

}

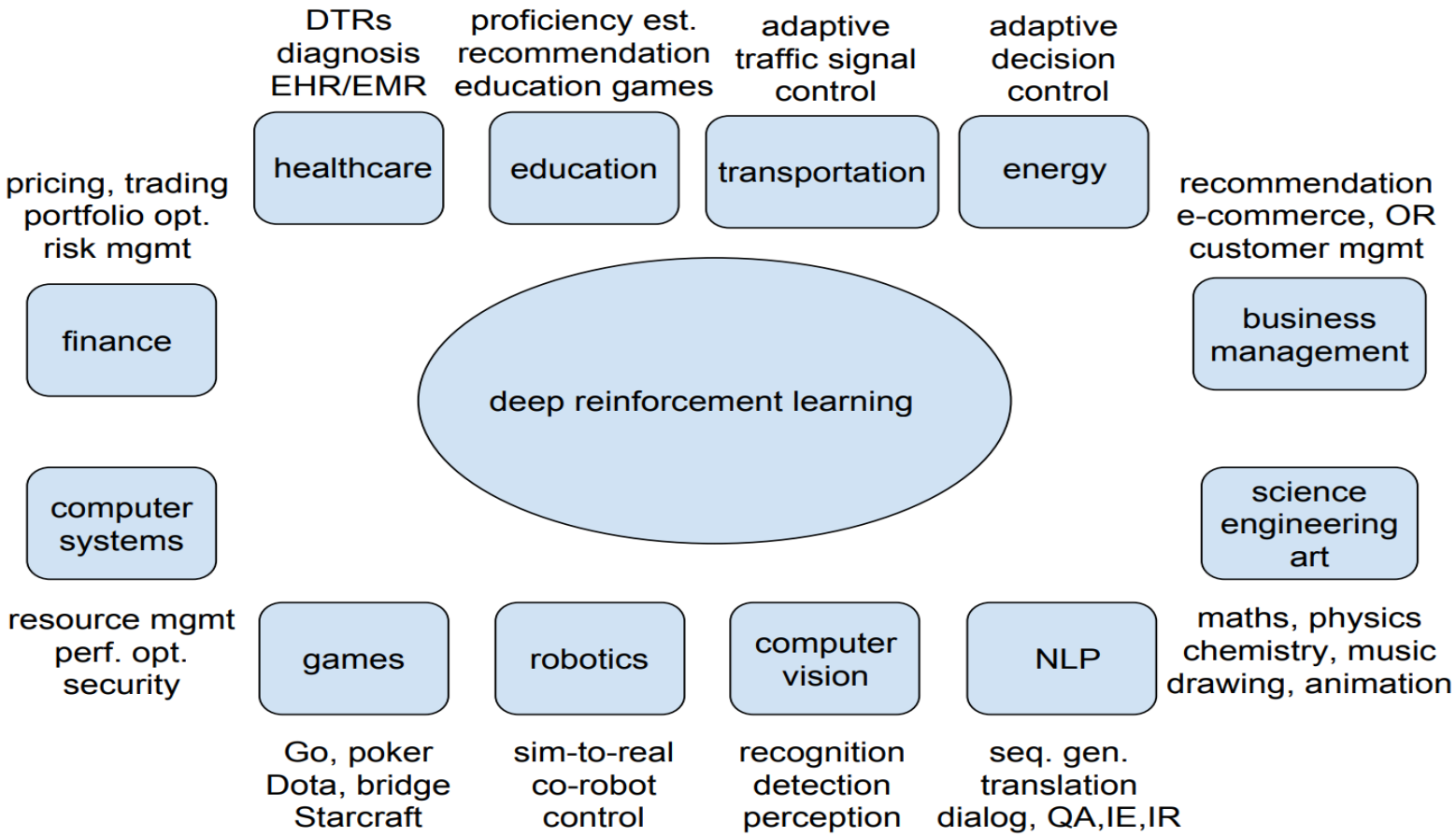
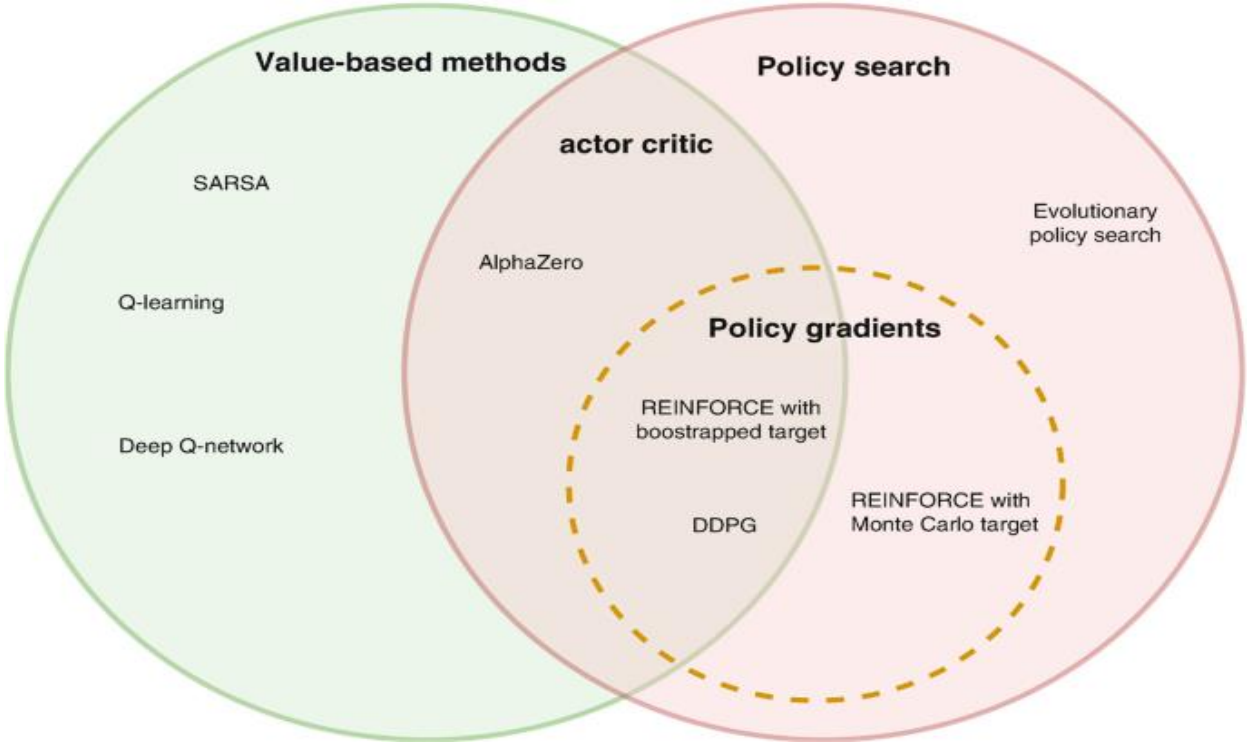


Figure 4: Deep Reinforcement Learning Applications

- Industry Ready Applications
  - Google's Cloud AutoML
  - Facebook's Horizon Platform

Policy vs Value based Reinforcement Learning

- **Policy learning** focuses on directly inferring a policy that maximizes the reward on a specific environment
- **Value learning** tries to quantify the value of every state-action pair
- **Example:** AI agent trying to learn a new chess opening
- Using policy reinforcement learning, the AI agent would try to infer a strategy to develop the pieces in a way that can achieve certain well-known position
- In Value-learning, the AI agent would assign a value to every position and select the moves that score higher.



Deterministic Vs Stochastic Policy

Policy is a set of rules that is referenced whenever we want to know what action to take

Deterministic policy

- offer a state-action mapping $\pi: s \mapsto a$
- Ideally the optimal mapping (that is, if all the Bellman equations are learned to perfection)

Stochastic policy

- conditional probability distribution over the actions in a given state, $\pi: P(a|s)$

Advantages of Stochastic Policy: Use in

1. Continuous Action Spaces
2. Stochastic Environments
3. Multi-agent environments
4. Partially Observable MDPs

For example : In game of rock-paper-scissors use stochastic policy

REINFORCE

Key idea

- **reinforcing good actions:** to push up the probabilities of actions that lead to higher return
- and push down the probabilities of actions that lead to a lower return
- until we get the optimal policy.

The policy gradient method will iteratively amend the policy network weights

- (with smooth updates)
- to make state-action pairs that resulted in positive return more likely
- and make state-action pairs that resulted in negative return less likely

Trajectory $\tau = (s_0, a_0, r_1, s_1, a_1, r_2, s_2, ...a_H, r_{H+1}, s_{H+1})$

Return for a Trajectory $R(\tau) = (G_0, G_1, ..., G_H)$

Total Return or Future Return $G_k \leftarrow \sum_{t=k+1}^{H+1} \gamma^{t-k-1} R_k$

REINFORCE

Policy: A policy is defined as the probability distribution of actions given a state

$$\pi(A_t = a | S_t = s)$$

$$\forall A_t \in \mathcal{A}(s), S_t \in \mathcal{S}$$

- The objective of a Reinforcement Learning agent is to maximize the “expected” reward when following a policy π.
- Define a set of parameters θ to parametrize this policy
- Parameters can be the coefficients of a complex polynomial or the weights and biases of units in a neural network
- If we represent the total reward for a given trajectory τ as r(τ), we arrive at :
- ***Reinforcement Learning Objective****: Maximize the “expected” reward following a parametrized policy*

$$J(\theta) = \mathbb{E}_{\pi}\left[r(\tau)\right]$$

$$\sum_{\tau} \mathbb{P}(\tau;\theta) R(\tau)$$

REINFORCE

- Notation Θ is the policy's parameter vector
- Thus Π(a|s, Θ) = Pr{A_t=a | S_t=s, Θ_t=Θ} for the probability that action a is taken at time t given that the environment is in state s at time t with parameter Θ
- learning the policy parameter based on the gradient of some scalar performance measure J(Θ) with respect to the policy parameter
- These methods seek to maximize performance, so their updates approximate gradient ascent in J:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \widehat{\nabla J(\boldsymbol{\theta}_t)}, \qquad \text{where } \widehat{\nabla J(\boldsymbol{\theta}_t)} \in \mathbb{R}^{d'}$$

is a stochastic estimate whose expectation approximates the gradient of the performance measure with respect to its argument Θ_t

REINFORCE (Monte Carlo Policy Gradients)

Input: a differentiable policy parameterization $\pi(a|s,\theta)$
Algorithm parameter: step size $\alpha > 0$)
Initialize the policy parameter θ at random

(1) Use the policy π_θ to collect a trajectory $\tau = (s_0, a_0, r_1, s_1, a_1, r_2, s_2, ...a_H, r_{H+1}, s_{H+1})$

(2) Estimate the Return for trajectory τ: $R(\tau) = (G_0, G_1, ..., G_H)$
where G_k is the expected return for transition k:

$$G_k \leftarrow \sum_{t=k+1}^{H+1} \gamma^{t-k-1} R_k$$

(3) Use the trajectory τ to estimate the gradient $\nabla_\theta U(\theta)$

$$\nabla_\theta U(\theta) \leftarrow \sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t$$

(4) Update the weights θ of the policy

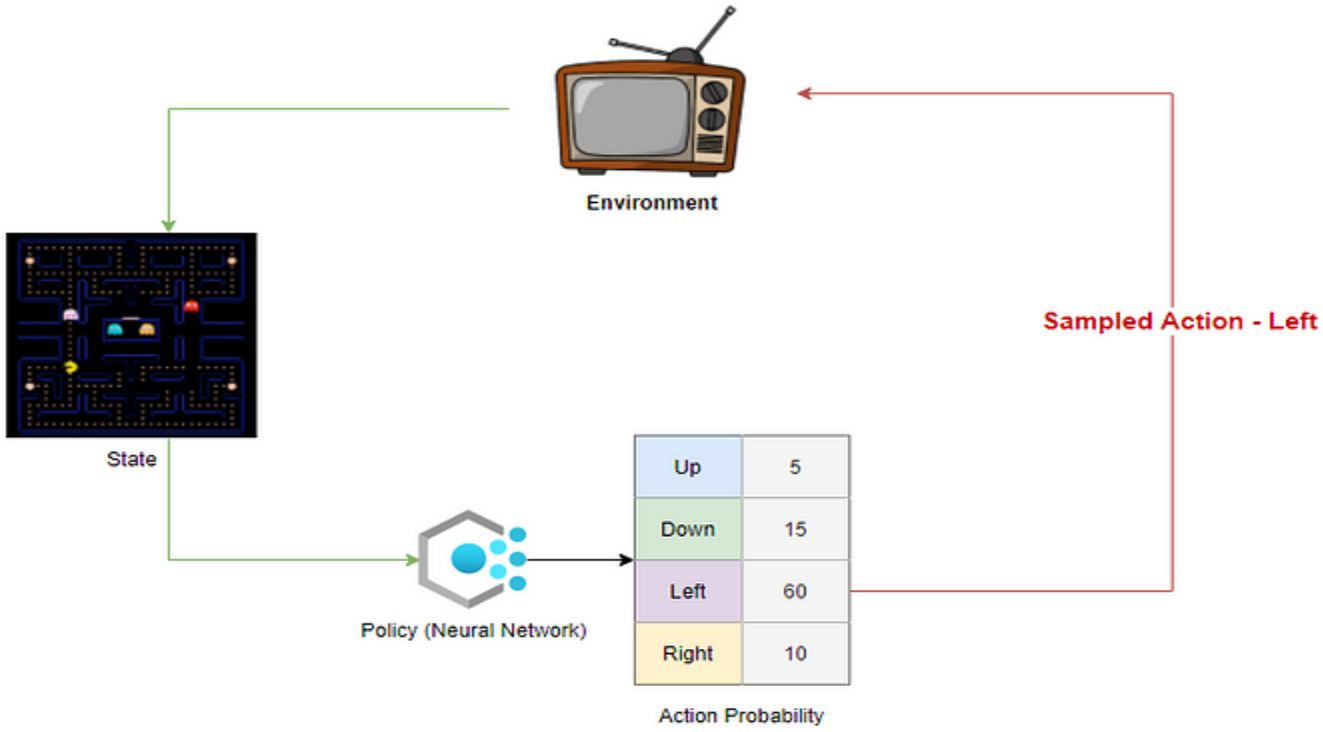
$$\theta \leftarrow \theta + \alpha \nabla_\theta U(\theta)$$

(5) Loop over steps 1-5 until not converged

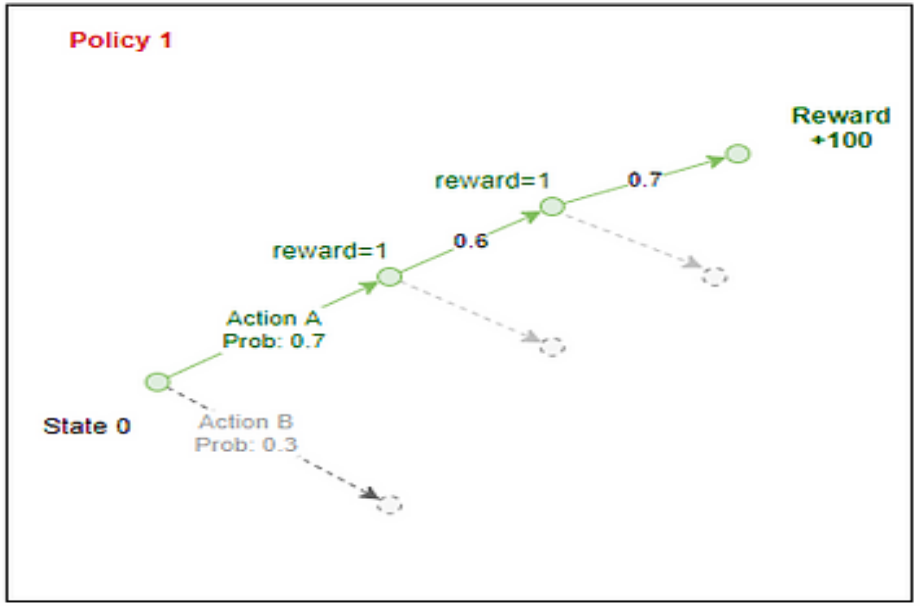
- Sometimes probabilities may be extremely tiny or very close to 1
- instead use a surrogate objective, log p (natural logarithm)
- since the log of probability space ranges from (-∞,0)
- and this makes the log probability easier to compute.

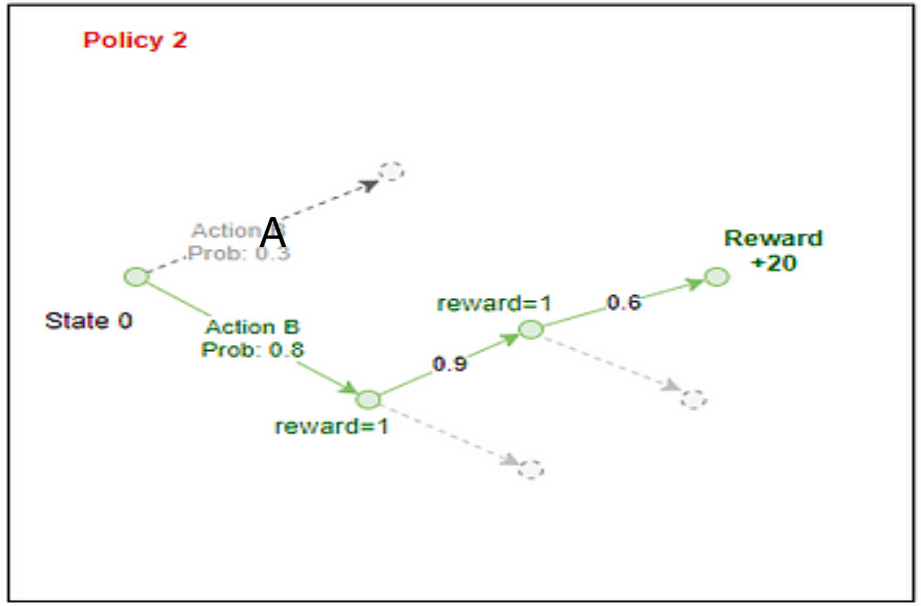
Example

The policy is usually a Neural Network that takes the state as input and generates a probability distribution across action space as output



Example: Policy 1 vs Policy 2 Trajectories



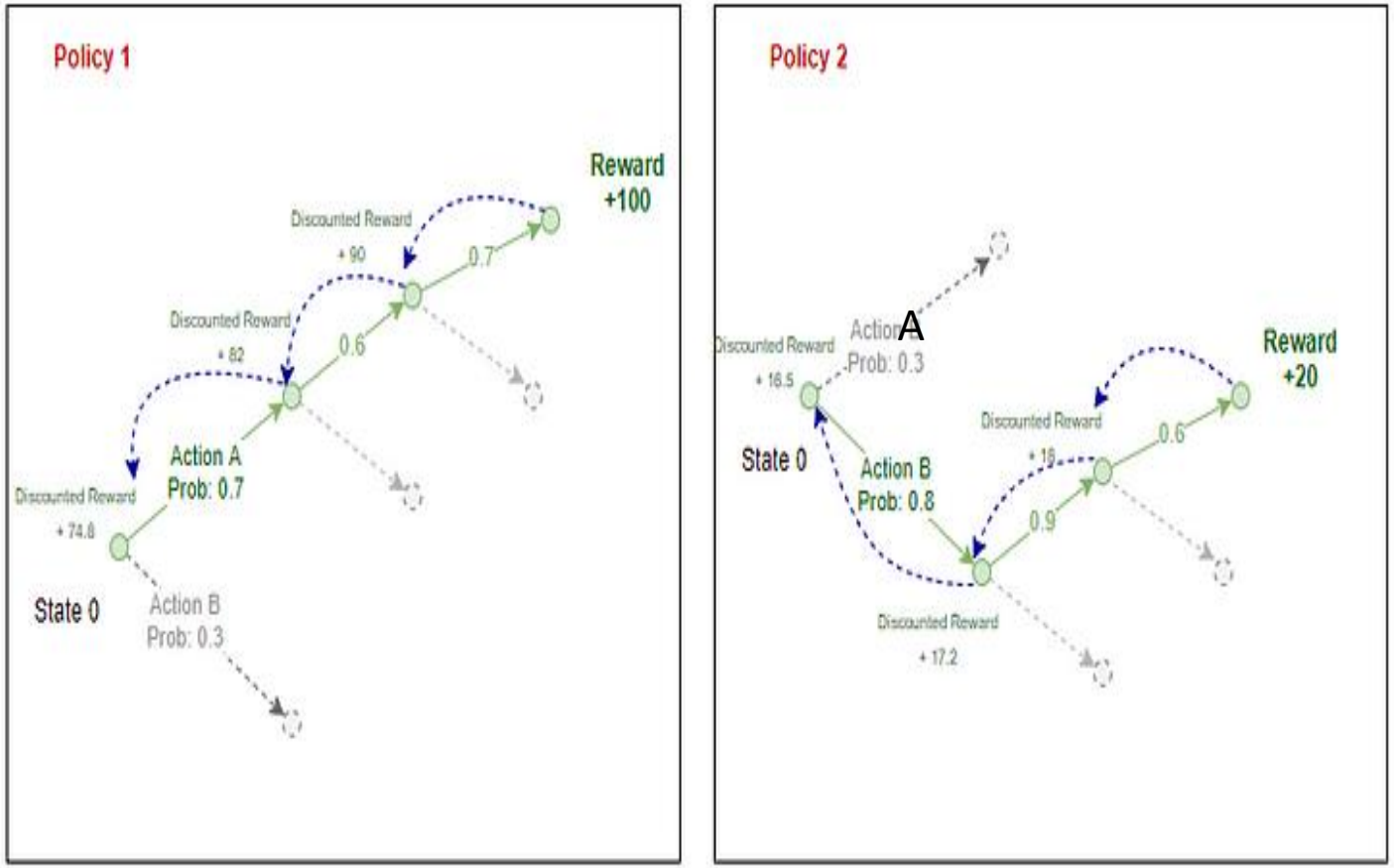


$$\text{Discounted Reward at } t,\ R(t) = r(t) + \gamma * R(t+1) = r(t) + \gamma * r(t+1) + \gamma^2 * r(t+2) + \ldots + \gamma^{(T-t)} * r(T)$$

γ = Discount Factor, usually 0.9
T = Terminal state's time step

$$\text{Expected Reward } G(\theta) = \sum_{i=0}^{n} \text{Probability of action}_k \text{ at state}_i * \text{discounted reward}_{(k,i)}$$

θ = Policy
n = Number of steps in the episode
k = index of action



The steps in the implementation of REINFORCE

1. *Initialize a Random Policy (a NN that takes the state as input and returns the probability of actions)*
2. *Use the policy to play N steps of the game — record action probabilities-from policy, reward-from environment, action — sampled by agent*
3. *Calculate the discounted reward for each step*
4. *Calculate expected reward G*
5. *Adjust weights of Policy (back-propagate error in NN) to increase G*
6. *Repeat from 2*

Disadvantages of REINFORCE

The Policy Gradient is :

$$\nabla_\theta J(\theta) = \mathbb{E}_\tau[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t]$$

- Has high variability in log probabilities and cumulative reward values, because each trajectories during training can deviate from each other at great degrees
- Can result in noisy gradients, and cause unstable learning and/or the policy distribution skewing to a non-optimal direction
- Can have cumulative reward of 0- both “goods” and “bad” actions with will not be learned if the cumulative reward is 0

$$\text{Introducing baseline } b(s):$$

$$\nabla_\theta J(\theta) = \mathbb{E}[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)(G_t - b(s_t))]$$

- will make smaller gradients, and thus smaller and more stable updates.

Common Baseline Functions

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\, G_t\right] \quad \text{REINFORCE}$$

$$= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\; Q^w(s,a)\right] \quad \text{Q Actor-Critic}$$

$$= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\; A^w(s,a)\right] \quad \text{Advantage Actor-Critic}$$

$$= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\; \delta\right] \quad \text{TD Actor-Critic}$$

Actor Critic Methods

Policy gradient in REINFORCE

$$\nabla_\theta J(\theta) = \mathbb{E}_\tau[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t]$$

$$\nabla_\theta J(\theta) = \mathbb{E}_{s_0,a_0,\ldots,s_t,a_t}[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)] \mathbb{E}_{r_{t+1},s_{t+1},\ldots,r_T,s_T}[G_t]$$

$$\mathbb{E}_{r_{t+1},s_{t+1},\ldots,r_T,s_T}[G_t] = Q(s_t, a_t)$$

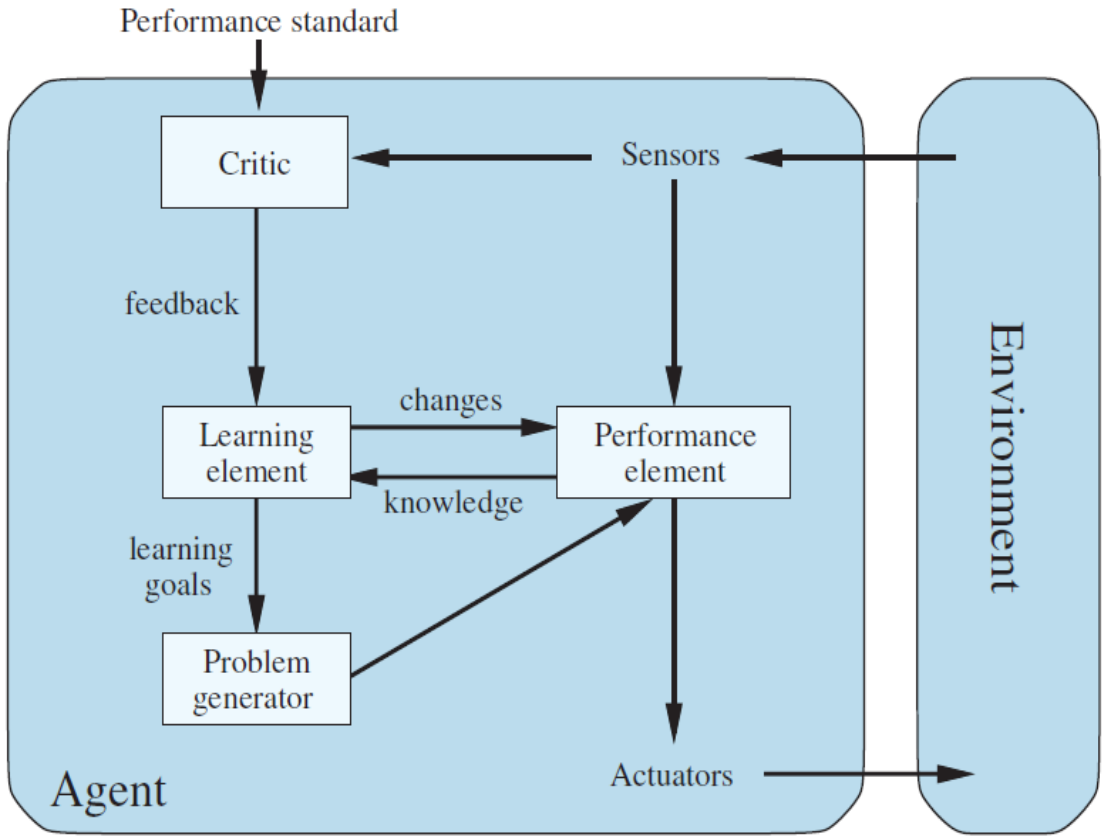
Actor Critic Methods

$$\nabla_\theta J(\theta) = \mathbb{E}_{s_0, a_0, \ldots, s_t, a_t}[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)] Q_w(s_t, a_t)$$

$$= \mathbb{E}_\tau[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) Q_w(s_t, a_t)]$$

Q value can be learned by parameterizing the Q function with a neural network

General Learning Agent

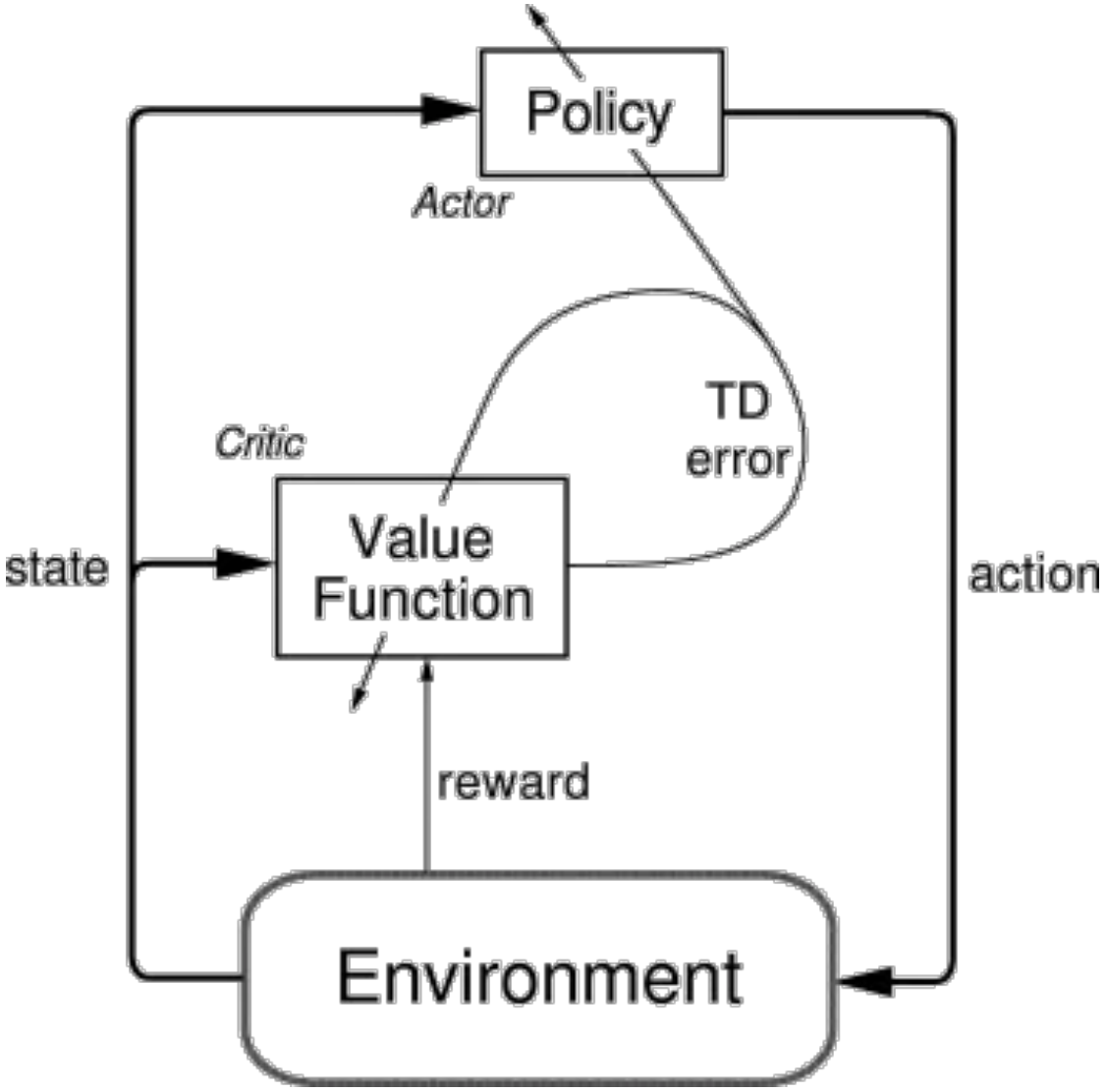


Learning element which is responsible for making improvements

Performance element, which is responsible for selecting external actions

Critic tells the learning element how well the agent is doing with respect to a fixed performance standard

Problem Generator which is responsible for suggesting actions that will lead to new and informative experiences.



Actor Critic Methods

- **The "Critic" estimates the value function**
  - This could be the action-value (the Q value) or state-value (the V value).
- **The "Actor" updates the policy distribution**
  - in the direction suggested by the Critic (such as with policy gradients).
- Both the Critic and Actor functions are parameterized with neural networks

Algorithm 1 Q Actor Critic

Initialize parameters s, θ, w and learning rates α_θ, α_w; sample $a \sim \pi_\theta(a|s)$.

for $t = 1 \ldots T$: **do**

Sample reward $r_t \sim R(s,a)$ and next state $s' \sim P(s'|s,a)$

Then sample the next action $a' \sim \pi_\theta(a'|s')$

Update the policy parameters: $\theta \leftarrow \theta + \alpha_\theta Q_w(s,a) \nabla_\theta \log \pi_\theta(a|s)$; Compute the correction (TD error) for action-value at time t:

$$\delta_t = r_t + \gamma Q_w(s',a') - Q_w(s,a)$$

and use it to update the parameters of Q function:

$$w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s,a)$$

Move to a $\leftarrow a'$ and s $\leftarrow s'$

end for

Advantage Actor Critic (A2C)

Advantage Value- Take a specific action compared to the average, general action at the given state

$$A(s_t, a_t) = Q_w(s_t, a_t) - V_v(s_t)$$

- Using Bellman Equation:

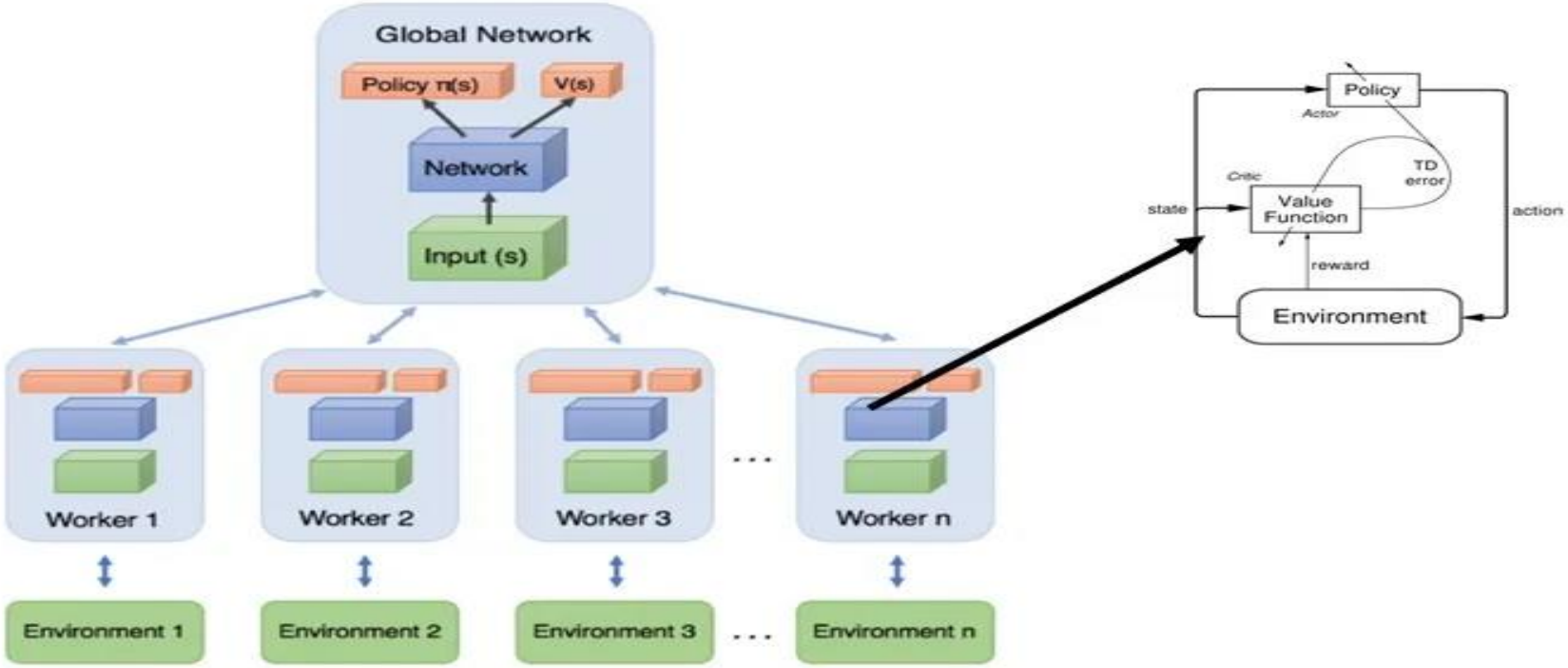
$$Q(s_t, a_t) = \mathbb{E}[r_{t+1} + \gamma V(s_{t+1})]$$

$$A(s_t, a_t) = r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t)$$

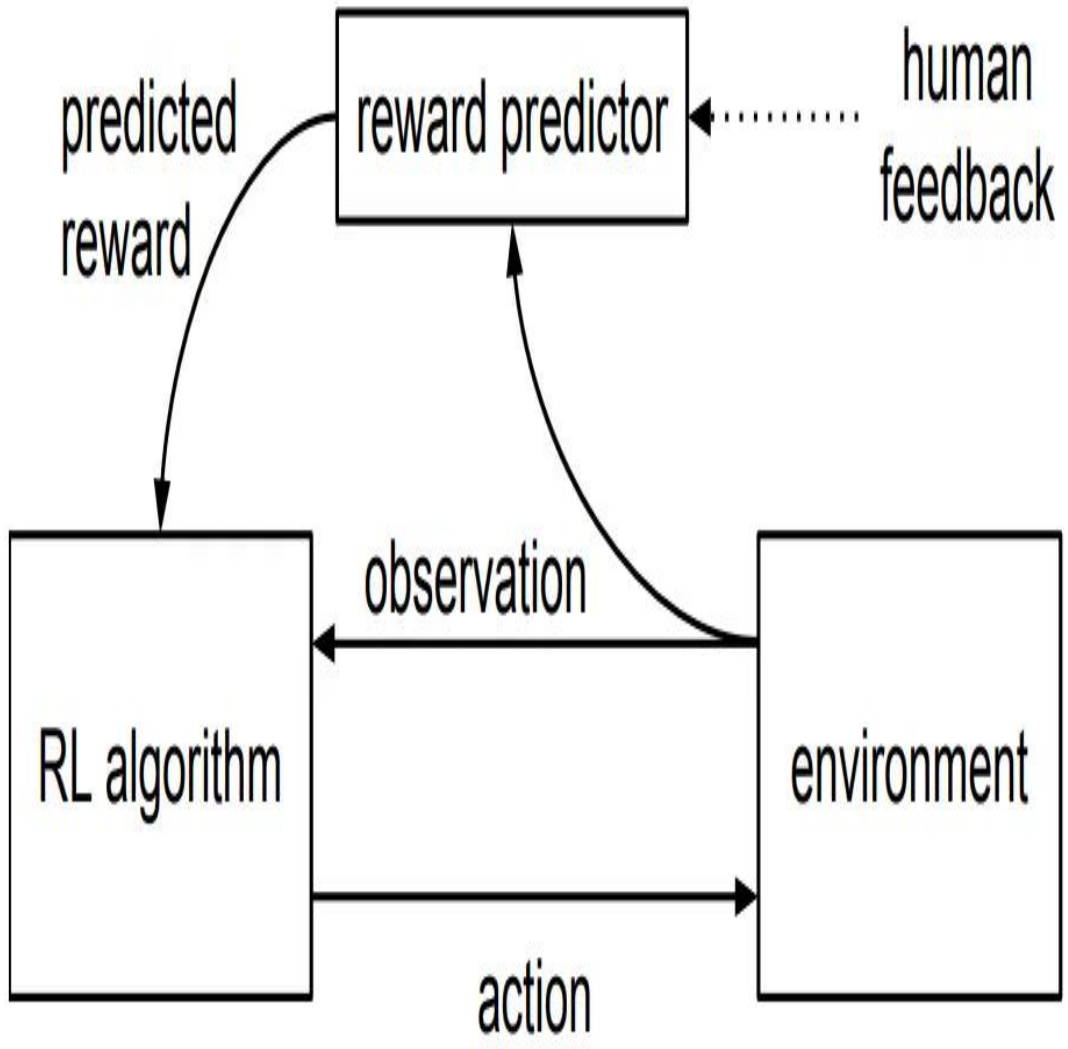
$$\nabla_\theta J(\theta) \sim \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)(r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t))$$

$$= \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) A(s_t, a_t)$$

Asynchronous Advantage Actor Critic (A3C)



- implements parallel training where multiple workers in parallel environments independently update a global value function
- Key benefit of having asynchronous actors is effective and efficient exploration of the state space.

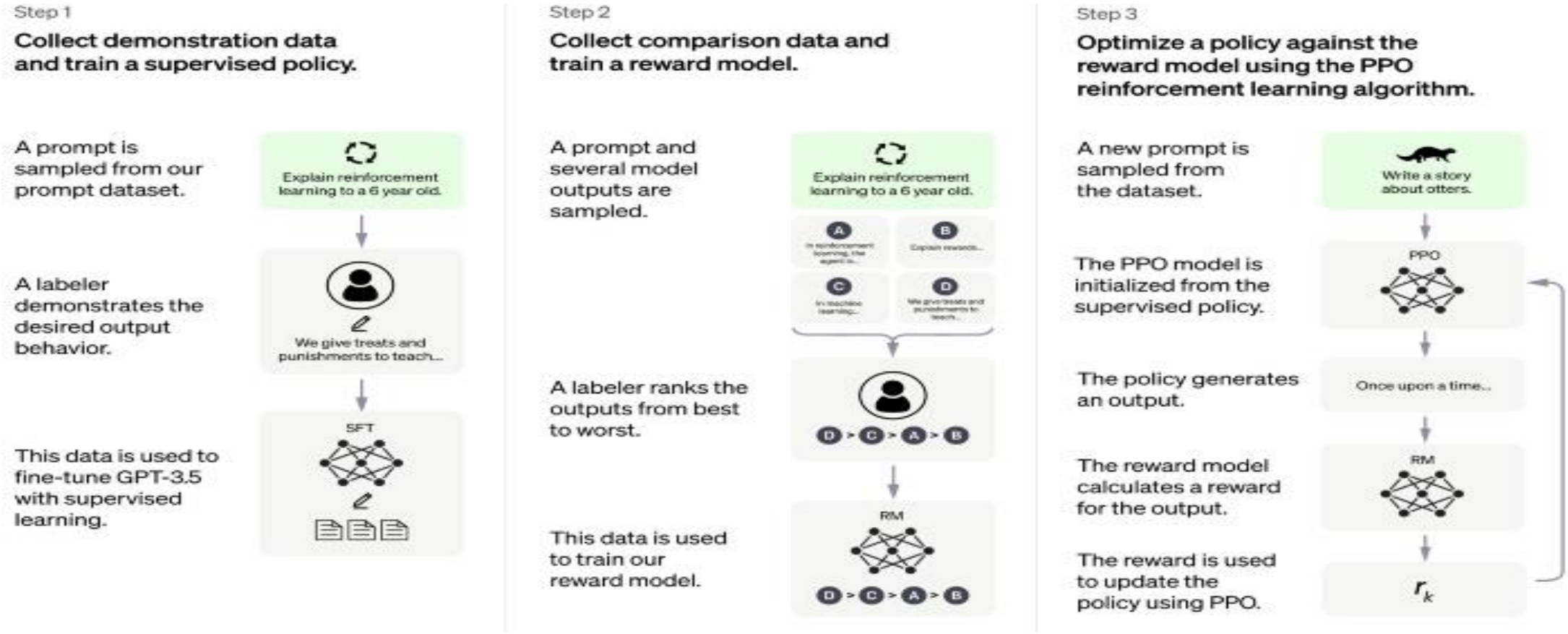


ChatGPT- Reinforcement Learning from Human Feedback

The method overall consists of three distinct steps:

1. **Supervised fine-tuning**
   - A pre-trained language model is fine-tuned on a relatively small amount of demonstration data curated by labelers, to learn a supervised policy (the SFT model) that generates outputs from a selected list of prompts
   - This represents the baseline model.
2. **Mimic human preferences**
   - Labelers are asked to vote on a relatively large number of the SFT model outputs, creating a new dataset consisting of comparison data.
   - A new model is trained on this dataset
   - This is referred to as the reward model (RM).
3. **Proximal Policy Optimization (PPO)**
   - The reward model is used to further fine-tune and improve the SFT model.
   - The outcome of this step is the so-called policy model.

How ChatGPT works



Proximal Policy Optimization (PPO) algorithms are policy gradient methods

References

1. Russell S., and Norvig P., Artificial Intelligence A Modern Approach (3e), Pearson 2010

2. Richard S Sutton, Andrew G Barto, Reinforcement Learning, second edition, MIT Press

3. https://www.coursera.org/learn/fundamentals-of-reinforcement-learning/home/week/1