Week 1

What is AI?
A system is rational if it does the right thing given what it knows.
Acting rationally (rational agents), acting humanly (turing test), thinking rationally (approach using logics), thinking humanly (cognitive modeling)

Reasoning (thinking) vs Behavior (acting)
Human-like vs Rationality

Rationality - the ability to make the right decision given the available information

Major AI components:
Robotic - manipulating
Machine Learning - adapting and detecting patterns
Automated Reasoning - drawing conclusions
Knowledge Representation - remembering
Computer vision - perceiving
Natural Language Processing - communicating

Rational agent (maximize performance measure):
Search Algorithms - Most intelligent tasks involve searching for a solution in the problem space
Knowledge Representation - Required to store facts & rules about the world
Inference - Reasoning with the knowledge
Planning - Achieve goals by generating sequences of actions based on knowledge about how the world works and how it affects the world
Learning - Acquire new knowledge from the environment and experience
Communication - Understanding speech signals, speaking languages
Perception - Providing information about the environment, recognizing objects

Intelligent agent - perceives the environment using sensors and acts rationally upon the environment using actuators. Mapping percept sequence to rational actions.

A system that is situated, autonomous, adaptive, and sociable. Receive input from the environment and perform actions that change the environment. Acts without external intervention, has control over its actions and internal states.
Reacts flexibly to changes in environment, learns from environment and experience. Able to interact peer-to-peer with others.

Environmental factors:
Fully observable - agent sensor give complete state of environment at a time (chess, board games)
Partially observable - a taxi agent cannot tell whether there is a traffic jam in jl. Sudirman.

Deterministic - if next state is completely determined by the current state
Stochastic - probability

Episodic - if agent experience is divided into atomic episodes, next episode does not depend on actions taken in previous episodes.
Sequential - current decision could affect all future decisions.

Static - environment cannot change while agent is deciding
Dynamic - environment can change while agent is deciding

Discrete - a finite number of distinct states
Continuous - states are not clearly distinguishable

Single agent, competitive agent, cooperative multiagent.

Simple reflex agent - Table lookup of percept-action pairs defining all possible condition-action rules necessary to interact in an environment
Model-based agent - store internal state of the world to remember to use the history
Goal-based agent - choose actions to achieve a goal
Utility-based agent - how to decide which action is best among alternatives
Learning agent - allows the agent to operate in unknown environments and become more competent

Summary

- **Simple Reflex Agent:** Acts only on the current percept using condition–action rules, with no memory.
- **Model-Based Reflex Agent:** Maintains an internal state to handle partially observable environments.
- **Goal-Based Agent:** Chooses actions that achieve specified goals by planning and lookahead.
- **Utility-Based Agent:** Selects actions that maximize overall desirability using a utility function.
- **Learning Agent:** Improves its performance over time by learning from feedback and exploration.

Week 2

State space
V - set of nodes/states
E - set of edges (actions)
Each node has state, parent node, action, path cost, depth

Size of problem is described in terms of nodes in the game tree

Tic-Tac-Toe:	9! nodes
Rubik's Cube:	10^{19} nodes
Reversi/Othello:	10^{58} nodes
Chess:	10^{123} nodes
Go:	10^{360} nodes

Each edge has a fixed positive cost
Goal test is applied to a node's state to determine if it is a goal node
Solution is a sequence of operator associated with a path in a state space from a start node to a goal node
Cost of a solution is the sum of edge costs on the solution path

Initially V={S}, S is start node
Each node in V is checked to see whether it is a goal node, if not it is expanded (continues until goal node is found)

Check for completeness (solution found?) and admissibility (optimal?)
Check for time and space complexity

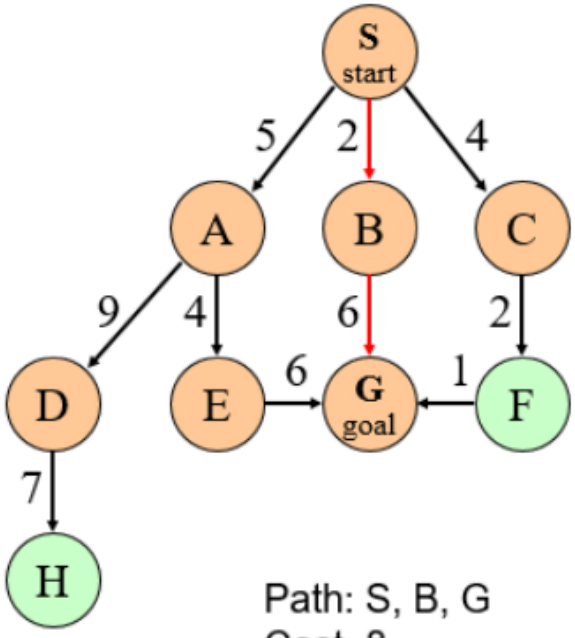
Uniformed Search Strategies (blind search)
- They do not know whether a node is more promising than the others
1. BFS (queue to order nodes)
2. DFS (stack to order nodes)
3. Depth-limited search (DFS with limited depth)
4. UCS (priority queue to order nodes, sorted by path cost)
5. IDS (do DFS to depth 1, if no solution found do DFS to depth 2, repeat until solution found)

BFS Example (8)

generalSearch(problem, queue)

# of nodes tested: 7, expanded: 6

expnd. node	node list
	{S}
S not goal	{A,B,C}
A not goal	{B,C,D,E}
B not goal	{C,D,E,G}
C not goal	{D,E,G,F}
D not goal	{E,G,F,H}
E not goal	{G,F,H,G}
G is goal	{F,H,G}



BFS is complete and optimal (if the path cost is non-decreasing)

Time and space complexity:

$O(|V||E|) = O(b^{d+1}) \approx$ exponential

- d is the depth of the solution
- b is the branching factor at each non-leaf node

For example: $d = 12$, $b = 10$

$10 + 100 + ... + 10^{12} + (10^{13} - 10) = O(10^{13})$

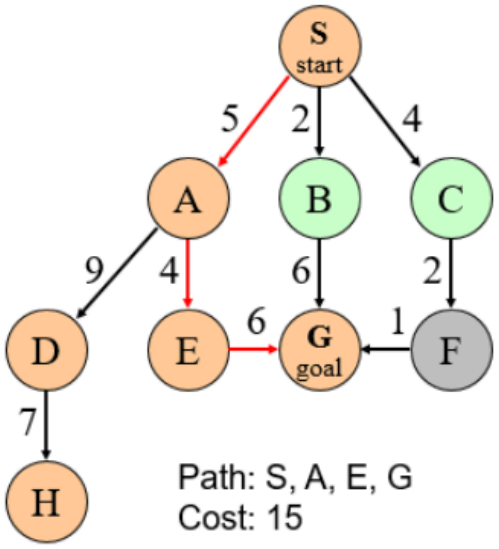
- If BFS expands 10000 nodes/sec and each node uses 1000 bytes of storage, then BFS will take **35 years** to run, and it will use 10 petabytes of memory!

DFS Example (7)

generalSearch(problem, stack)

# of nodes tested: 6, expanded: 4

expnd. node	node list
	{S}
S not goal	{A,B,C}
A not goal	{D,E,B,C}
D not goal	{H,E,B,C}
H not goal	{E,B,C}
E not goal	{G,B,C}
G is goal	{B,C}

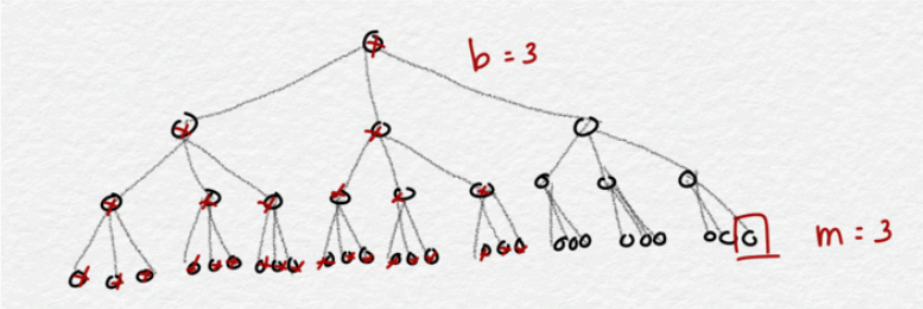


Path: S, A, E, G
Cost: 15

May terminate without a loop detection, not complete (without cycle detection, not optimal)

- Time complexity: O(b^m) ≈ exponential
- Space complexity: O(bm) ≈ linear (less than BFS)
  - m : the maximum depth of the search tree
  - b : the branching factor at each non-leaf node
  - Only store the nodes at the last subtree

chronological backtracking
when search hits a dead end, backs up one level at a time



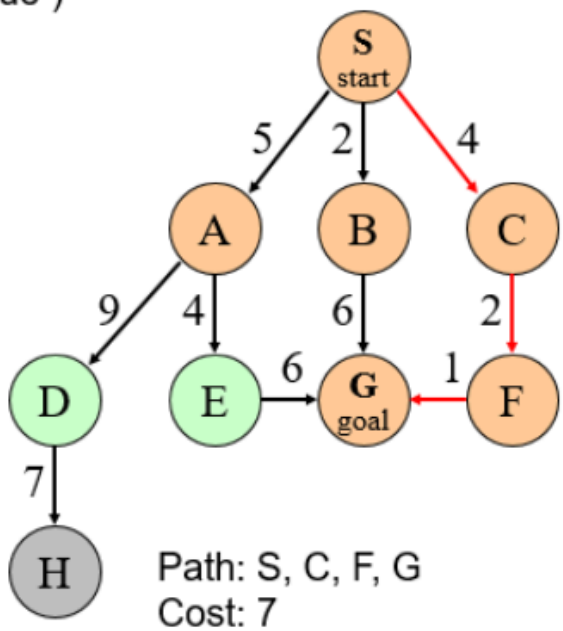
53

UCS Example (7)

generalSearch(problem, priorityQueue)

# of nodes tested: 6, expanded: 5

expnd. node	node list
	{S:0}
S not goal	{B:2,C:4,A:5}
B not goal	{C:4,A:5,G:8}
C not goal	{A:5,F:6,G:8}
A not goal	{F:6,G:8,E:9,D:14}
F not goal	{G:7,G:8,E:9,D:14}
G is goal	{G:8,E:9,D:14}



Complete and Optimal (if branching factor is finite and step costs >= e), basically Dijkstra's Algorithm

- The complexity:
  - is exponential in the optimal solution cost (C^*) divided by the minimum cost of an action (edge): $O(b^{1+\lfloor C^*/e \rfloor})$
  - which is = $O(b^{d+1})$ when all edge costs are equal.

- Time and space complexity: $O(b^d) \approx$ exponential
  - d : the depth of the shallowest solution
  - b : the branching factor at each non-leaf node

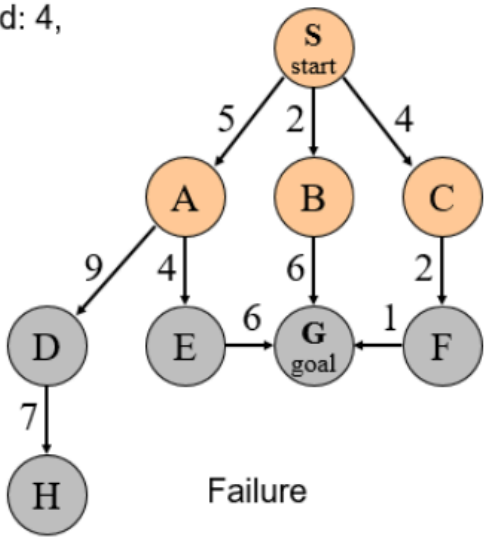
Node expansion is limited to a predetermined depth limit. Terminate with failure if no solution is found/within the depth limit

DLS Example (5)

depthLimitedSearch(problem, stack, limit)

depth: 1, limit: 1, # of nodes tested: 4, expanded: 1

expnd. node	node list
	{S}
S not goal	{A,B,C}
A not goal	{B,C} no expand
B not goal	{C} no expand
C not goal	{} no expand



Completeness and Optimality:

- Not complete if $l < d$.
- Not optimal if $l > d$.

Time Complexity:

- $O(b^l)$

Space Complexity:

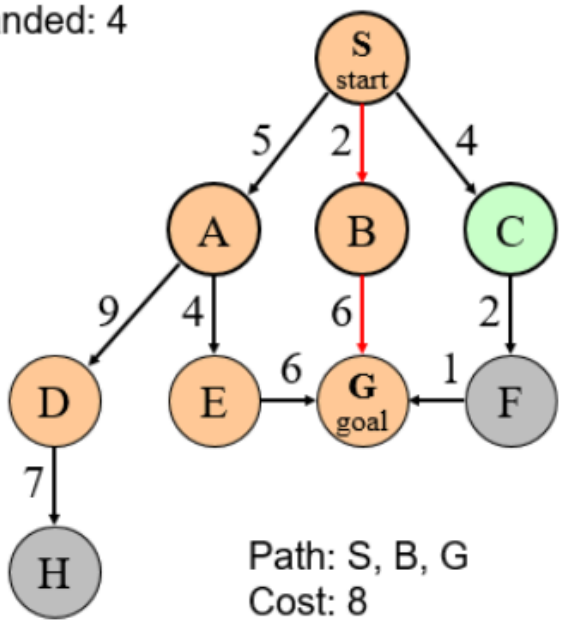
- $O(bl)$

IDS Example (11)

deepeningSearch(problem, stack)

depth: 2, # of nodes tested: 7(3), expanded: 4

expnd. node	node list
	{S}
S not goal	{A,B,C}
A not goal	{B,C} no expand
B not goal	{C} no expand
C not goal	{} no expand
S no test	{A,B,C}
A no test	{D,E,B,C}
D not goal	{E,B,C} no expand
E not goal	{B,C} no expand
B no test	{G,C}
G is goal	{C} no expand



Path: S, B, G
Cost: 8

- Has the advantages of BFS
  - Complete
  - Optimal (if the edges have identical costs)
- Has the advantages of DFS
  - Linear space complexity: O(bd)
  - Finds longer paths more quickly
- Wasteful ?
  - because nodes near the top of the search tree are generated multiple times
- It turns out this is *NOT* very costly
  - For a tree with (nearly) the same branching factor at each level, most of the nodes are in the bottom level
- Worst case time complexity: O(b^d)

- The total number of nodes generated is:

  $N(\text{IDS}) = (d)b + (d\text{-}1)b^2 + \ldots + (1)b^d = O(b^d)$
  - d : the solution's depth
  - b : the branching factor at each non-leaf node
- Compared to BFS:

  $N(\text{BFS}) = b + b^2 + \ldots + b^d + (b^{d+1} - b) = O(b^{d+1})$
- For example: $b = 10$, $d = 5$

  $N(\text{IDS}) = 50 + 400 + 3000 + 20000 + 100000 = 123{,}450$

  $N(\text{BFS}) = 10 + 100 + 1000 + 10000 + 100000 + 999990 = 1{,}111{,}100$
- Notice: BFS generates some nodes at depth d+1, while IDS does not.
- In general, IDS is the *preferred* uninformed search method, when search space is large and d is unknown.

Comparing Uninformed Search Strategies

Criterion	Breadth-First	Depth-First	Depth-Limited	Uniform-Cost	Iterative Deepening
Complete?	Yes*	No	Yes, if l ≥ d	Yes*	Yes
Time complexity	b^{d+1}	b^m	b^l	b^d	b^d
Space complexity	b^{d+1}	bm	bl	b^d	bd
Optimal?	Yes*	No	No	Yes*	Yes*

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

Informed Search - uses domain knowledge to guide selection of the best path
Heuristics is a technique to solve problem faster

A heuristic function, $h(n)$:

- uses domain-specific information in some way
- is computable from the current state description
- estimates the *goodness* of node *n*
- estimates how close to a goal is node *n*
- estimates the minimal cost of the path from node *n* to a goal state

- $h(n) \geq 0$ for all nodes *n*
- $h(n) = 0$ implies that *n* is a goal node
- $h(n) = \infty$ implies that *n* is a dead end from which a goal cannot be reached

- All domain knowledge used in the search is encoded in the heuristic function *h*.

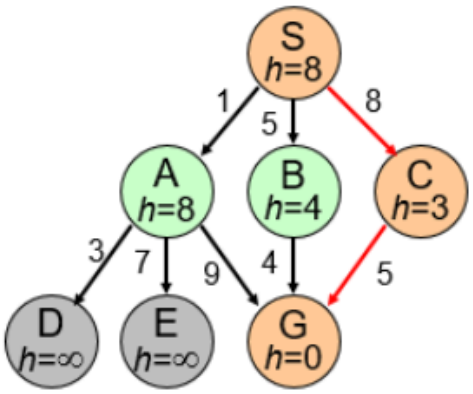
- This is an example of a *weak method* because of the limited way that domain-specific information is used to solve a problem.

Best-First Search: Greedy, A*

Greedy Search Example (4)

$f(n) = h(n)$
# of nodes tested=3, expanded=2

expnd. node	OPEN list
	{S:8}
S not goal	{C:3,B:4,A:8}
C not goal	{G:0,B:4,A:8}
G is goal	{B:4,A:8}

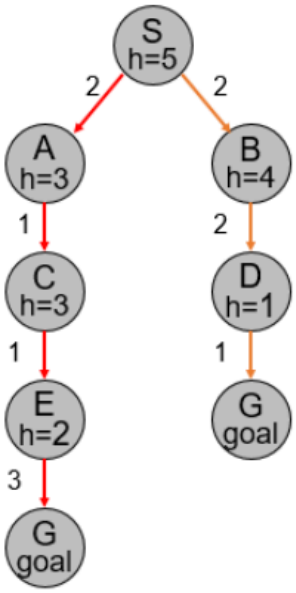


- *Fast but not optimal.*

Path: S, C, G
Cost: 13

Select smallest value

- Not complete
  - If trapped in an infinite path or loops.
- Not optimal/admissible
  - Greedy search finds the left goal (solution cost: 7)
  - Optimal solution is the path to the right goal (solution cost: 5)
- The worst case time complexity = space complexity = $O(b^m)$,
- m: maximum depth of the search space.



NUS
VERSITY
ERNATIONAL

A Search Algorithm

- Use evaluation function $f(n) = g(n) + h(n)$, where
  - $g(n)$ is path cost from start to current node n (as defined in Uniform-Cost search).
  - $h(n)$ is the estimated cost from node n to goal
  - $f(n)$ is the estimated cost of a solution through n to goal
- Nodes on the search frontier (in nodes list) are ranked by the estimated cost of a solution $f(n)$.

NUS
IVERSITY
TERNATIONAL

A* Search Algorithm

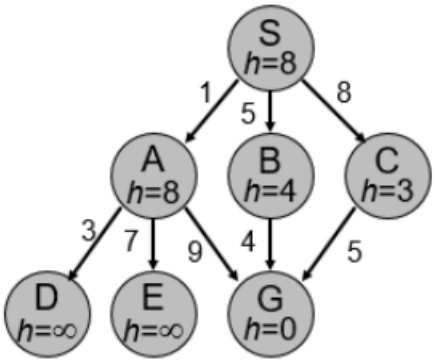
- Uses the same evaluation function as used by Algorithm A search, except adds the constraint that:
  - For all nodes n in the search space, $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost of the minimal cost path from n to a goal.
- When $h(n) \leq h^*(n)$ holds for all n, then ***h* is *admissible***,

  i.e., it never overestimates the cost to the nearest goal.
- An admissible heuristic guarantees that a node on the optimal path can never look so bad to be not considered.
  - E.g. straight-line heuristic.

A* Search Example (5)

NAL

n	$h(n)$	$g(n)$	$f(n)$	$h^*(n)$
S	8	0	8	9
A	8	1	9	9
B	4	5	9	4
C	3	8	11	5
D	∞	4	∞	∞
E	∞	8	∞	∞
G	0	10/9/13	10/9/13	0

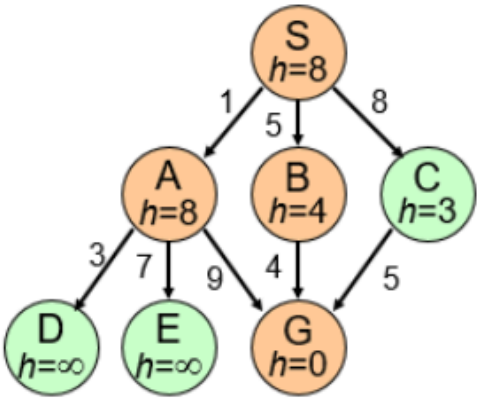


- Since $h(n) \leq h^*(n)$ for all n, *h* is admissible.

A* Search Example (5)

$f(n) = g(n) + h(n)$
# of nodes tested=4, expanded=3

expnd. node	OPEN list
	{S:0+8}
S not goal	{A:9,B:9,C:11}
A not goal	{B:9,G:10,C:11,D:∞,E:∞}
B not goal	{G:9, C:11,D:∞,E:∞}
G is goal	{C:11,D:∞,E:∞}



- *Fast and optimal.*

Path: S, B, G
Cost: 9

Week 3
Partial Searching - seeks a goal node and construct a path from start to goal state (BFS, DFS, UCS, Greedy, A*)
Complete Searching - every node is a solution, can go from one solution to another, can stop anytime and have valid solution, search is about finding better solution. Suitable for problems that do not care about solution path, aim for optimizing an objective function, require exponential time to find optimal solution. Hard problems can be solved in a reasonable time by approximate model or solution.
Approximate model - find exact solution to a simpler version of the problem
Approximate solution - find a non-optimal solution of the original hard problem

Partial Searching

- Finds a path to a goal
- BFS, DFS, UCS, Greedy, etc.

Complete Searching

- Every node is a solution
    - Operator go from one solution to another
    - Can stop anytime and have valid solution
    - Search is now about finding a better solution → Brute-force until optimal solution is found
- Do not care about solution path
- Optimizes objective function
- Examples: TSP, default Minimax algorithm

Traveling Salesman Problem

What is it?

- An example of complete search
- Nodes are cities
- Edges labelled with distance
- Solution → Permutation of cities, always assume tour returns home
- **# of solutions → (n-1)! / 2, where n is the number of cities**

Operators for improving TSP solutions

- Used to create neighbouring solutions —> New solutions obtained from current solutions by making a small, defined change according to the neighbourhood
- Two-swap (common)

    - Swap location of two cities in the tour
    - Example: ABCDE → swap(A, D) yields DBCAE
    - Used to shorten tours by removing crossing edges
- Two-interchange
    - Reverse path between two cities
    - Example: ABCDE → swap(A, D) yields DCBAE
    - Test alternative city orders by swapping their positions

Local Search

About local search

- Searching within a limited or specific part of a data structure
- In other words: only considers nearby solutions in the neighbourhood that can be reached with one application of an operator
- Neighbourhood must be much smaller than the search space → We are not exploring the entire search space unlike global search that explores all possible solutions

Evaluation Function f(n)

- Guides the search
- Tells algorithm how good a given solution/state is → Mapping each state to a numerical value that represents how desirable that state is
- How to know the numerical value to be mapped to each state?
    - Start with finding a goal of what to optimize (Example: minimizing total distance in TSP)
    - Define a metric to measure (Example: total path length in TSP)
- Maximization problems → Higher f(n) = Better state
- Minimization problems → Lower f(n) = Better state

Valley Finding

- Goal is minimizing the value of f(n)
- Valley = Region of a search space when the evaluation function reaches a low
- Works by incrementally moving in the direction that **decreases** the score of the evaluation function

Hill Climbing

- Goal is maximizing the value of f(n)
- Works by incrementally moving in the direction that **increases** the score of the evaluation function
- Very space efficient, also very fast and effective

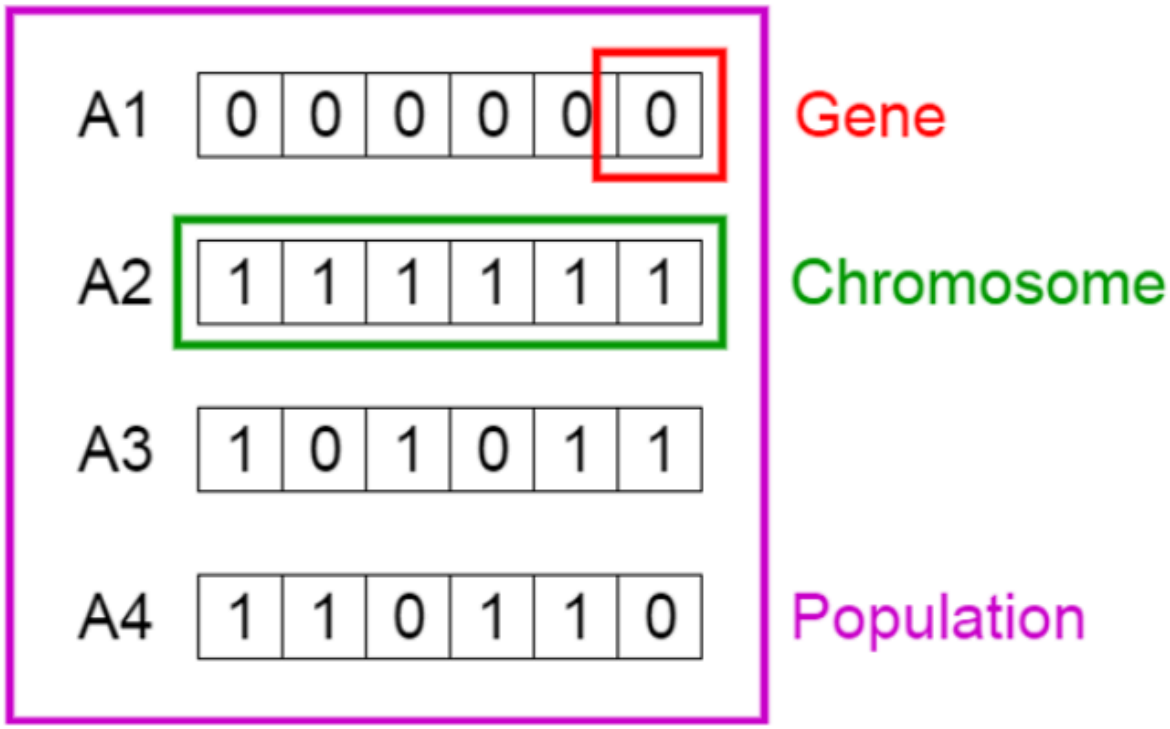
General steps for HC and VF

1. Begin at an initial state
2. Evaluate: Calculate value of current state using evaluation function, f(n)
3. Generate all possible neighbouring states and evaluate their scores
4. Select the best move based on the goal (lower/higher than current state's score)
5. Repeat: the new state becomes the current state
6. Termination: Search stops when no neighbouring states has a lower evaluation score than the current state

Genetic algorithms

Optimization algorithms based on evolution, a parallel search procedure

- Problems are encoded into a representation which allows certain operations to occur
  - Usually use a ***bitstring***
  - The representation is key – needs to be thought out carefully
- An encoded candidate solution is an individual
- Each individual has a fitness which is a numerical value associated with its quality of solution
- A population is a set of individuals
- Populations change over generations by applying strategies to them



BINUS UNIVERSITY INTERNATIONAL

Typical Genetic Algorithm

- Initialize: Population P (size = N) of random individuals (bitstrings)
- Evaluate: for each x ∈ P, compute fitness(x)
- Loop
  - For I = 1 to N
    - Choose 2 parents each with probability proportional to fitness scores (**roulette wheel selection**)
    - **Crossover** the 2 parents to produce a new bitstring (child)
    - With some small probability **mutate** child
    - Add child (and its fitness score) to the population
- Until some child is fit enough or you get bored
- Return the best individual in the population according to fitness function

Selection - select fittest individuals and let them pass their genes.
Cross over - combining part of individuals to create new individual
Mutation - generate desirable features that are not present in the population

Before Mutation

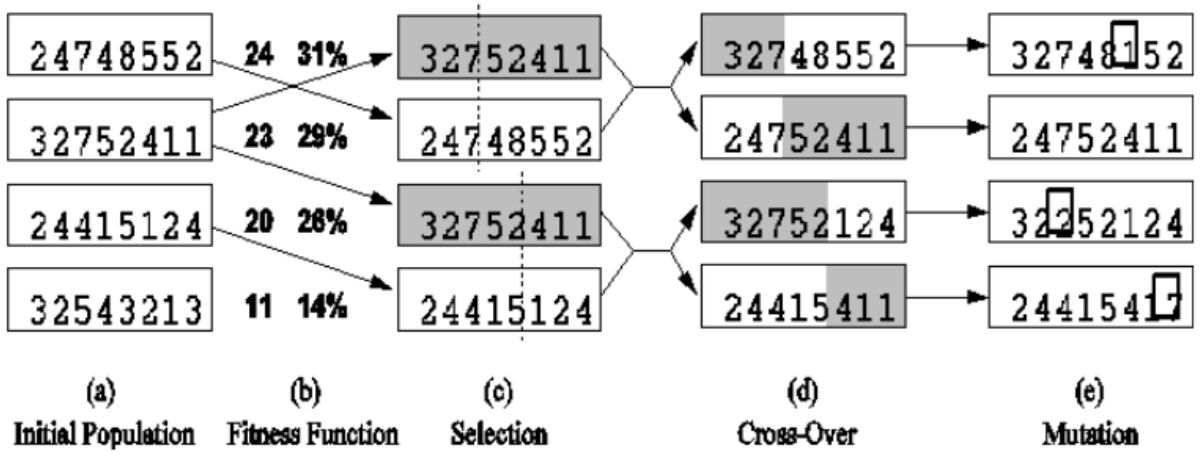
A5	1	1	1	0	0	0

After Mutation

A5	1	1	0	1	1	0

If the population has converged (still similar), algorithm terminates.

- An example of how GA works on the 8-queens problem.



- Why genetic algorithm is a type of search?
  - States: possible solutions
  - Operators: mutation, crossover, selection
  - Parallel search: since several solutions are maintained in parallel
  - Hill-climbing on the fitness function
  - Mutation and crossover allow us to get out of local optima

Week 4

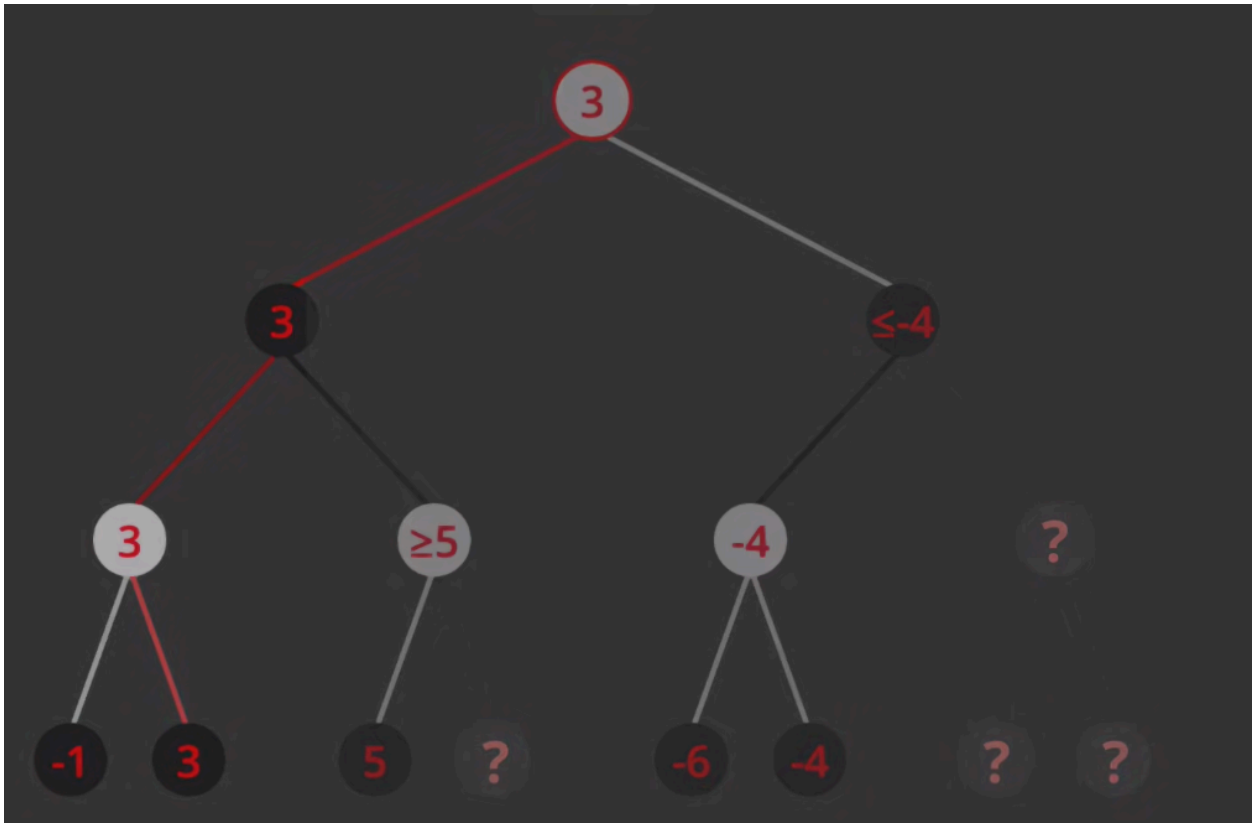
Greedy search with eval function

Always assume the worst case, the opponent chooses the optimal move.

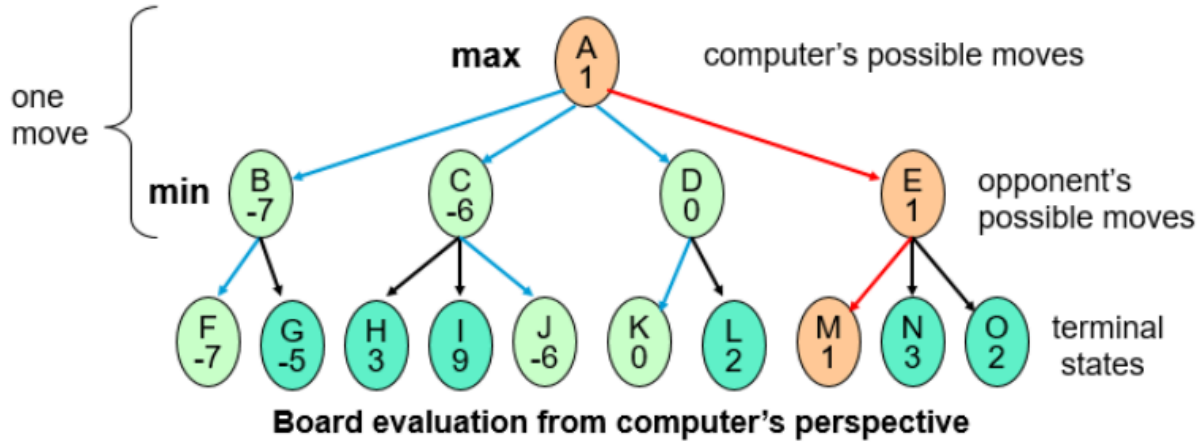
Minimax

Opponent will choose the minimizing move, computer choose best move considering both its move and opponents optimal move

Use minimum when children are opponents. Use maximum when children are computer's move



Minimax Example

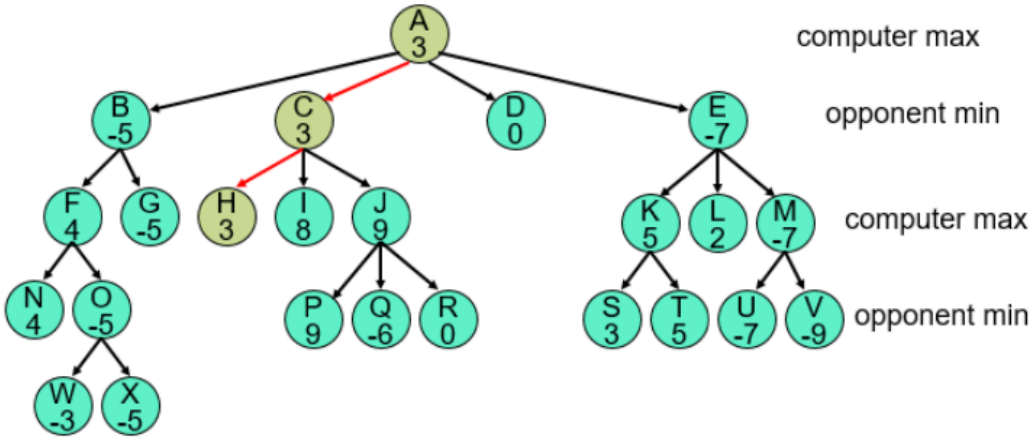


Board evaluation from computer's perspective

Result: choose E

Deeper Game Trees

- The complete minimax values on a tree with more than one move:



BINUS UNIVERSITY INTERNATIONAL

General Minimax Algorithm

For each move by the computer:

1. Perform depth-first search to a terminal state
2. Evaluate each terminal state
3. Propagate upwards the minimax values
   - if opponent's move, propagate up minimum value of children
   - if computer's move, propagate up maximum value of children
4. choose move with the maximum of minimax values of children

BINUS UNIVERSITY INTERNATIONAL

Properties of Minimax Algorithm

- Complete? Yes (if tree is finite)
- Optimal? Yes (against an optimal opponent)
- Space Complexity?
  - Only O(*bm*) nodes need to be kept in memory at any time (depth first exploration)
- Time complexity?
  - Given a search tree with branching factor *b* and maximum depth *m*: $O(b^m)$
- Time complexity is a major problem since computer typically only has a finite amount of time to make a move.

Alpha-beta Pruning

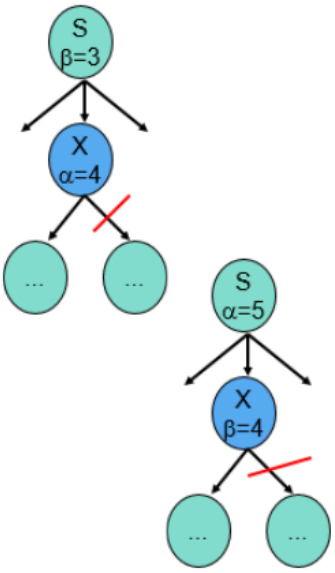
Ignore useless branches.

Runs DFS on the game tree while keeping track of Alpha at maximizing level (highest value) and beta at minimizing level (lowest value).

Alpha-Beta Pruning

Pruning occurs:

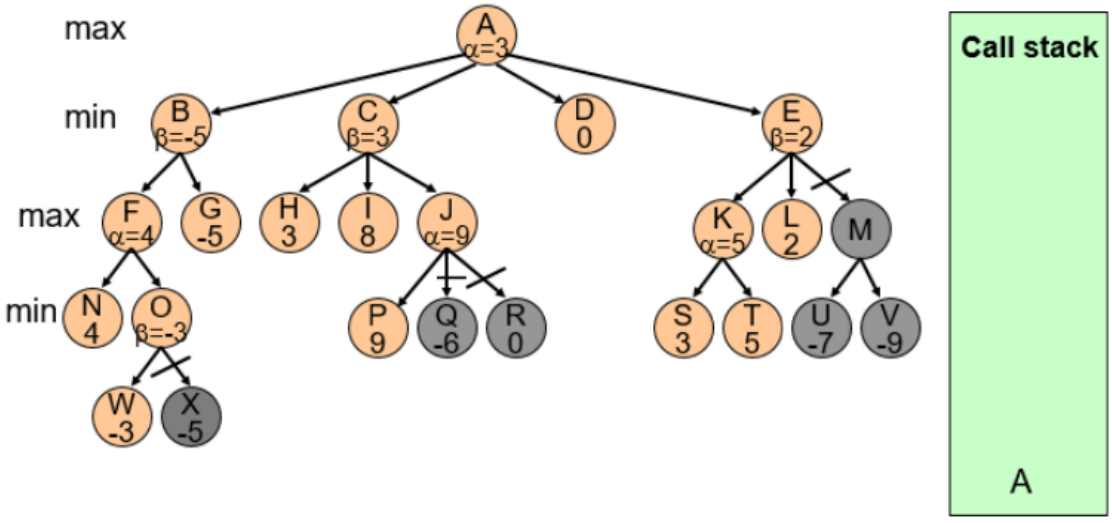
- when maximizing:
  - if ***alpha*** ≥ **parent's *beta***, stop expanding
  - opponent will prevent computer from taking this route
- when minimizing:
  - if ***beta*** ≤ **parent's *alpha***, stop expanding
  - computer should not take this route



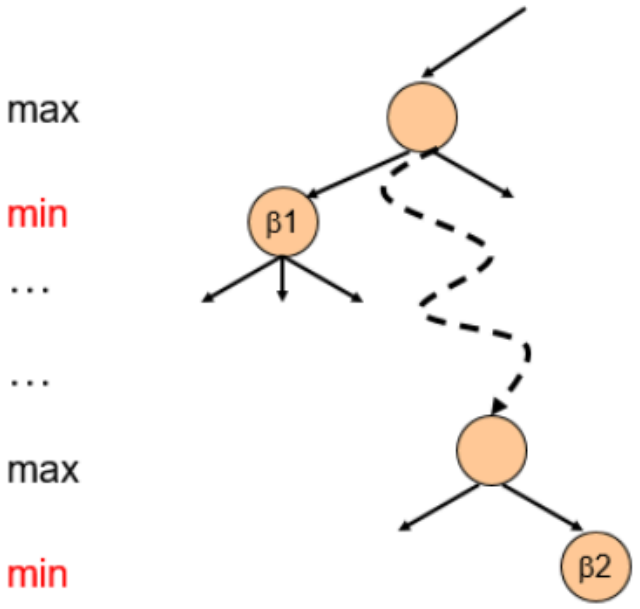
Alpha-Beta Example (32)

NAL

Result: computer chooses move to C



Alpha-Beta Pruning: General Case



If β1 is better than β2 (for max), max will avoid β2 in play.

Effectiveness of Alpha-Beta Search (2)

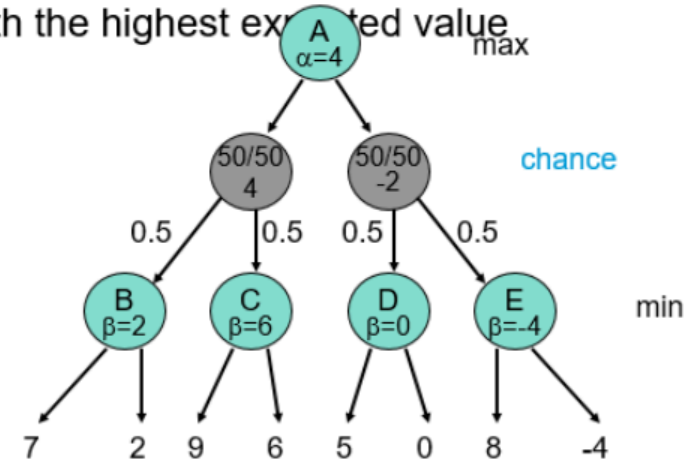
- Pruning does not affect the final result
- Good move ordering improves effectiveness of pruning
- In practice often get $O(b^{(m/2)})$ rather than $O(b^m)$
  - doubles the depth of search
- For example: Chess
  - permits much deeper search for the same time
  - makes computer chess competitive with humans.

Deal with time limit by setting a conservative depth limit so we find a move in time

US
ERSITY
RNATIONAL

Non-Deterministic Games (3)

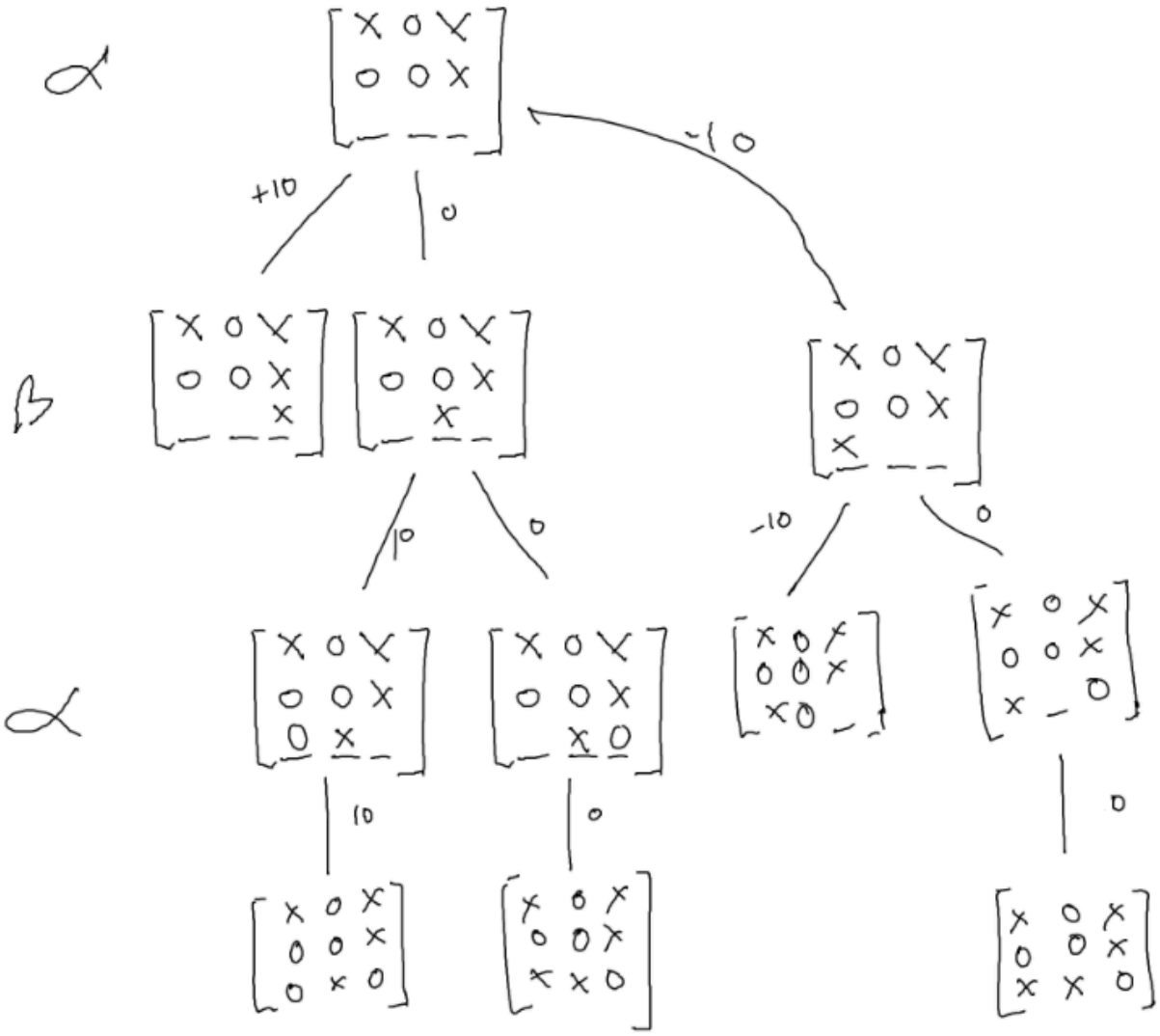
- Weight score by the probabilities that move occurs
- Use **expected-minimax** value for move: **sum** of possible random outcomes
- Choose move with the highest ex[illegible]ted value



INUS
NIVERSITY
TERNATIONAL

Non-Deterministic Games (4)

- The complexity of expected-minimax:
  $O(b^m n^m)$, where **n** is the number of distinct rolls
- Non-determinism increases branching factor
  - 21 possible distinct rolls with 2 dice (since 6-5 is same as 5-6)
- The extra cost of expected-minimax makes it unrealistic to consider looking ahead very far in most games of chance
- Alpha-beta pruning is less effective, why?



Week 6

Bayes Rule

$$P(A,B) = P(A \mid B)\, P(B) = P(B \mid A)\, P(A)$$

$$P(A \mid B) = \frac{P(B \mid A)}{P(B)}\, P(A)$$

Which tells us: how often A happens *given that B happens*, written **P(A|B)**,

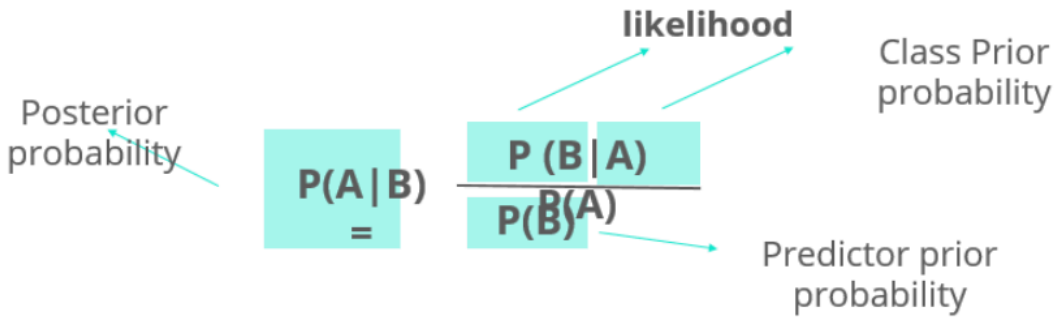
When we know: how often B happens *given that A happens*, written **P(B|A)**

and how likely A is on its own, written **P(A)**

and how likely B is on its own, written **P(B)**

It is very useful when it is hard to get P(A | B) directly, but easier to get the things on the right side

Bayes Rule



- A **prior** probability is an initial probability value originally obtained before any additional information is obtained.
- A **posterior** probability is a probability value that has been revised by using additional information that is later obtained.

Use Bayes rule because often P(A|B) not accessible, only have access to P(B|A)

let S represents the proposition that the patient has a stiff neck, and

let M represents the proposition that the patient has meningitis.

The doctor and patient may like to know P(M | S), but:

- We need to count every single person with stiff neck and meningitis at the same time from the population?
  - Too difficult
- It could change significantly over time given epidemics or other seasonal factors.

- On the other hand, doctors may be able to accumulate statistics that define P(S | M), the probability of people with meningitis have the stiff neck symptoms

- For example,

  if P(M) = 1/50,000, P(S) = 1/20, and P(S | M) = 1/2, then using Bayes' Rule says that:

  - P(M | S) = P(M) * P(S | M) / P(S)

    = (1/50,000) * (1/2) / (1/20)

    = 1/5000

    = 0.0002

BINUS UNIVERSITY INTERNATIONAL

Independence

- Two events are independent if knowledge that one occurred does not change the probability that the other occurred.
- Informally, events are independent if they do not influence one another.

Example

- Toss a coin twice we expect the outcomes of the two tosses to be independent of one another.
- In real experiments this always has to be checked.
- If my coin lands in honey and I don't bother to clean it, then the second toss might be affected by the outcome of the first toss

Naive bayes algorithm - classification technique base on bayes theorem with assumption of independence among predictors. (assume presence of a particular feature in class unrelated to presence of any other)

Naïve Bayes Classifier

For example,

- a fruit may be considered to be an apple if it is red, round, and about 3 inches in diameter.

- Even if these features depend on each other or upon the existence of the other features, all of these properties independently contribute to the probability that this fruit is an apple and that is why it is known as 'Naive'.

Naïve Bayes Classifier

- The generalized Bayes rule as the Naive Bayes Classifier:

$$P(C|E_i) = P(C) * [P(E_1|C)/P(E_1)] * \ldots * [P(E_n|C)/P(E_n)]$$

  - C is the set of possible classifications or classes, e.g. {yes, no} or {+, -}
  - The features/attributes that we choose are E1, E2, ..., En.
  - Then compute: P(C='+' | E1, ..., En) and P(C='-' | E1, ..., En)
    - Assume that E1, ..., En are independent

  - Finally, the class of that input pattern or example is the value for C that gives the maximum probability.

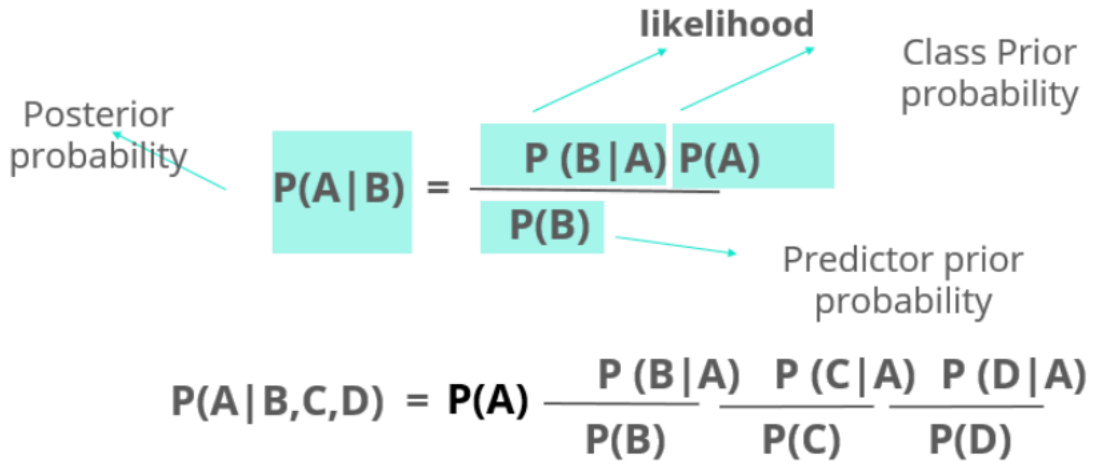
Naïve Bayes Classifier

- Essence of Naïve Bayes, with 1 non-class field, is to calculate this for each class value, given some new instance with fieldval = F:
- P(class = C | Fieldval = F)

- For many fields, our new instance is (e.g.) (F1, F2, ...Fn), and the 'essence of Naive Bayes' is to calculate this for each class:
- P(class = C | F1,F2,F3,...,Fn)

- i.e. What is prob of class C, given all these field vals together?

Naïve Bayes Classifier



$$P(A|B,C,D) = P(A)\,\frac{P(B|A)}{P(B)}\,\frac{P(C|A)}{P(C)}\,\frac{P(D|A)}{P(D)}$$

Remember the **multiplication rule** is a way to find the probability of two events happening at the same time

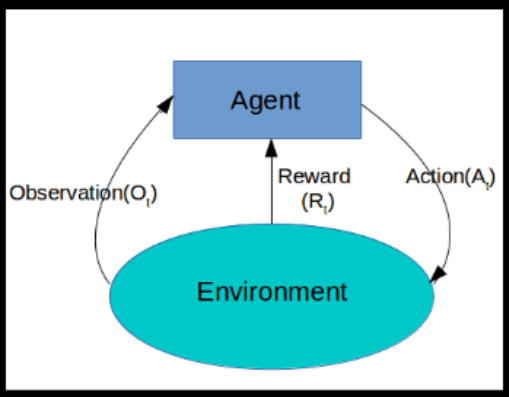
Week 7

Supervised Learning - takes input x to determine output y, outputs can be evaluated immediately.

Reinforcement Learning - Takes current state S to determine action A, predicted action cannot be evaluated directly.

Agent gets positive reinforcement if tasks done well, negative reinforcement if tasks done poorly.

RL Component Definition



Agent: The RL algorithm that learns
Environment: The world through which the agent moves
Action: All the possible steps that the agent can take
State: Current condition returned by the environment

Reactive Agent Algorithm

Repeat:

- s ← sensed state (Accessible or observable state)
- If s is terminal then exit
- a ← choose action (given s)
- Perform a

Approaches

- Learn policy directly– **function** mapping from states to actions
- Learn utility values for states (i.e., the value function)

A flow diagram of model-based RL

Value Function

the ***value*** is the expected long-term return (the sum of all your current and future rewards)

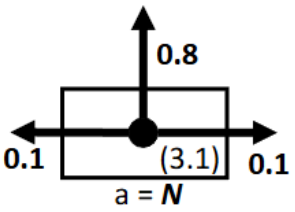
- The agent knows what state it is in
- The agent has a number of actions it can perform in each state.
- Initially, it doesn't know the value of any of the states
- If the outcome of performing an action at a state is **deterministic**, then the agent can update the utility value `V()` of states:
  - `V(oldstate) = reward + V(newstate)`
- The agent learns the utility values of states as it works its way through the state space

Exploration

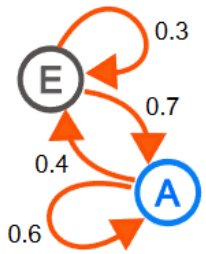
- The agent may occasionally choose to explore suboptimal moves in the hopes of finding better outcomes
  - Only by visiting all the states frequently enough can we guarantee learning the true values of all the states
- A discount factor is often introduced to prevent utility values from diverging and to promote the use of shorter (more efficient) sequences of actions to attain rewards
- The update equation using a discount factor γ is:
  - V(oldstate) = reward + γ * V(newstate)
- Normally, γ is set between 0 and 1

Markov Decision Process (MDP)

MDP is a tuple of $(S, A, \{P_{s,a}\}, \gamma, R)$ where



- S is a set of **states**
- A is a set of **actions**
- $P_{s,a}$ are the **state transition probabilities**
- γ is called the **discount factor**, $\gamma \in [0,1)$
- R is the **reward function**, $R: S \times A \mapsto \mathbb{R}$ sometimes also written as $R: S \mapsto \mathbb{R}$



$$s_0 \xrightarrow[P_{s_0,a_0}]{a_0} s_1 \xrightarrow[P_{s_1,a_1}]{a_1} s_2 \xrightarrow[P_{s_2,a_2}]{a_2} s_3 \xrightarrow[P_{s_3,a_3}]{a_3} \dots$$

Markov Decision Process (MDP)

Find the optimal policy $\boldsymbol{\pi}^*$ by optimizing the value function $\boldsymbol{V^{\pi}}$

$$V^*(s) = \max_{\pi} V^{\pi}(s)$$

$$V^*(s) = R(s) + \max_{a \in A} \gamma \sum_{s' \in S} P_{sa}(s') V^*(s')$$

$$\pi^*(s) = \operatorname*{argmax}_{a \in A} \sum_{s' \in S} P_{s,a}(s') V^*(s')$$

```
Initialize random π
Repeat{
    1. Take action using π to get experience in the MDP
    2. Update estimates of P_{s,a}
    3. Solve Bellman's equation using Value Iteration to get V
    4. Update π(s) := argmax_{a∈A} Σ_{s'} P_{s,a}(s') V(s')
}
```

Step 1: Define the Value Function

$$V(s) = \text{expected total discounted reward starting from } s$$

We want:

- $V(A)$
- $V(E)$

Step 2: Write Bellman Equations

For state A

$$V(A) = R(A) + \gamma\Big[0.6\,V(A) + 0.4\,V(E)\Big]$$

$$V(A) = 5 + 0.9(0.6V(A) + 0.4V(E))$$

For state E

$$V(E) = R(E) + \gamma\Big[0.7\,V(A) + 0.3\,V(E)\Big]$$

$$V(E) = 1 + 0.9(0.7V(A) + 0.3V(E))$$

Step 3: Solve the System

Expand both equations

(1) A:

$$V(A) = 5 + 0.54V(A) + 0.36V(E)$$

$$V(A) - 0.54V(A) = 5 + 0.36V(E)$$

$$0.46V(A) = 5 + 0.36V(E)$$

---

(2) E:

$$V(E) = 1 + 0.63V(A) + 0.27V(E)$$

$$V(E) - 0.27V(E) = 1 + 0.63V(A)$$

$$0.73V(E) = 1 + 0.63V(A)$$

Step 4: Solve simultaneously

From (2):

$$V(E) = \frac{1 + 0.63V(A)}{0.73}$$

Substitute into (1):

$$0.46V(A) = 5 + 0.36\left(\frac{1 + 0.63V(A)}{0.73}\right)$$

$$0.46V(A) = 5 + 0.493 + 0.311V(A)$$

$$0.46V(A) - 0.311V(A) = 5.493$$

$$0.149V(A) = 5.493$$

$$\boxed{V(A) \approx 36.87}$$

Now plug back:

$$V(E) = \frac{1 + 0.63(36.87)}{0.73} \approx \boxed{33.18}$$

Step 5: Interpret the result

- Starting in **A** is better than starting in **E**
- Even though rewards are small, **looping + discounting** makes long-term value large
- This is why γ matters a lot

(Optional) If there were multiple actions

You'd compute:

$$V(s) = \max_a \left[R(s,a) + \gamma \sum_{s'} P(s'|s,a) V(s') \right]$$

And choose the **action with the highest value** → that's the **optimal policy**.

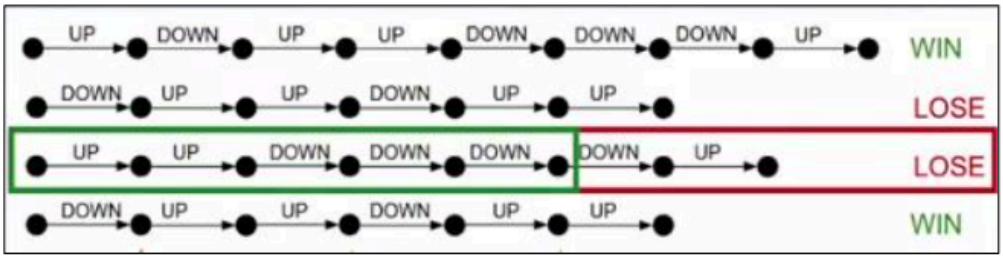
Exam-style summary (memorize this)

> Write Bellman equations using rewards, transition probabilities, and γ, then solve them to find state values.

Markov Decision Process (MDP)

Credit Assignment Problem

- Outcomes of behavior are delayed in time (*sparse reward setting*)
- Discovering which choices are responsible for rewards can present a challenge
- Requires huge training examples (mostly fail in many complex environments)
- **Reward shaping** can be the solution



Value function vs Q-function

The **value function** measure "how good is a state?"

$$V^{\pi}(s) = E\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, \pi\right]$$

The expected cumulative reward from the policy π from state s

The **Q-value function** measure "how good is a state-action pair?"

$$Q^{\pi}(s,a) = E\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$$

The expected cumulative reward from taking action a in state s and the policy π

Markov Decision Process (MDP) Example: Robot in a Grid

Scenario: Imagine a robot navigating a grid world.

- **States (S)**: Each cell in the grid represents a state.
- **Actions (A)**: The robot can move UP, DOWN, LEFT, or RIGHT.
- **State Transition Probabilities (P_sa)**: There's an 80% chance the robot moves in the intended direction, but a 10% chance it veers left or right.
- **Reward Function (R(s, a))**: The robot gets +1 at cell (4,3), -1 at cell (4,2), and -0.04 for each step.
- **Discount Factor (γ)**: Let's say γ = 0.9 to prioritize immediate rewards.

Question: What is the optimal policy π*(s) for the robot to maximize its expected return? What is the optimal value function V*(s)

Use bellman optimality equation

$V^*(s) = \max_{a \in A} \left[R(s,a) + \gamma \sum_{s'} P_{s,a}(s') \cdot V^*(s') \right]$

You'd iteratively calculate the value function for each state, considering the rewards and transition probabilities, until the values converge. The optimal policy would then be the action that maximizes the value function in each state.

Value Function Example

Scenario: Continuing the Robot in a Grid example, let's focus on a single state.

Question: What is the state value $V(s)$ of being in cell (2,2) under a given policy where the robot always moves UP, with a discount factor $\gamma = 0.9$? Assume $R(s,a) = -0.04$ for each step and the value of the cell above (2,2) is $V(s') = 0.5$.

Solution:

- **Update (with discount factor)**: $V(s) = R(s,a) + \gamma \cdot V(s')$
- **Plug in the values**: $V(s) = -0.04 + 0.9 \cdot 0.5 = 0.41$

Q-Function Example

Scenario: Consider a simplified scenario where the robot is in a "Blank" state and can move either "Left" or "Right."

Question: If the robot moves "Right" from the "Blank" state and receives an immediate reward of 1, how does the Q-table update? Assume a learning rate (α) of 0.05 and a discount factor (γ) of 0.99.

Solution:

- **Q-Learning Update Rule**: $Q(s,a) \leftarrow Q(s,a) + \alpha \cdot [r + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a)]$
- **Applying the values**:
  - $Q(\text{Blank}, \text{Right}) = Q(\text{Blank}, \text{Right}) + 0.05 \cdot [1 + 0.99 \cdot \max_{a'} Q(s',a') - Q(\text{Blank}, \text{Right})]$
  - Assuming $\max_{a'} Q(s',a') = 0$ (since we don't have further information), and initially $Q(\text{Blank}, \text{Right}) = 0$:
  - $Q(\text{Blank}, \text{Right}) = 0 + 0.05 \cdot [1 + 0.99 \cdot 0 - 0] = 0.05$

Q-Learning Example

Scenario: A robot is learning to navigate a simple environment.

Given:

- **States**: A, B, C
- **Actions**: UP, DOWN
- **Q-Table**: Initialized to 0 for all state-action pairs.
- **Learning Rate (α)**: 0.1
- **Discount Factor (γ)**: 0.9
- **Exploration Strategy**: ε-greedy with ε = 0.1

Steps:

1. **Initialization**: All Q-values are 0.
2. **Episode 1**:
   - Robot starts in state A.
   - With probability ε = 0.1, the robot explores; otherwise, it exploits.
   - Let's say the robot explores and chooses action UP.
   - The robot moves to state B and receives a reward of 0.
   - **Update Q-value**:
     - $Q(A, UP) = Q(A, UP) + \alpha * [R + \gamma * \max_{a'} Q(B, a') - Q(A, UP)]$
     - $Q(A, UP) = 0 + 0.1 * [0 + 0.9 * 0 - 0] = 0$
3. **Episode 2**:
   - Robot starts in state B.
   - This time, the robot exploits and chooses the action with the highest Q-value. Since both UP and DOWN are 0, it chooses UP.
   - The robot moves to state C and receives a reward of 1.
   - **Update Q-value**:
     - $Q(B, UP) = Q(B, UP) + \alpha * [R + \gamma * \max_{a'} Q(C, a') - Q(B, UP)]$
     - $Q(B, UP) = 0 + 0.1 * [1 + 0.9 * 0 - 0] = 0.1$

1. Markov Decision Process (MDP)

- **Definition:** An MDP is a mathematical framework for modeling decision-making in situations where outcomes are partly random and partly under the control of a decision-maker (the agent).
- **Components:**
  - **States (S):** The possible situations the agent can be in.
  - **Actions (A):** The choices the agent can make.
  - **State Transition Probabilities ($P_{s,a}(s')$):** The probability of moving from state s to state s' after taking action a.
  - **Reward Function ($R(s,a)$ or $R(s,a,s')$):** The reward received after taking action a in state s (and possibly transitioning to state s').
  - **Discount Factor (γ):** A value between 0 and 1 that determines how much the agent cares about future rewards. A higher γ makes the agent focus more on long-term rewards.
- **Example (Robot in a Grid):**
  - **States:** Each cell in the grid.
  - **Actions:** UP, DOWN, LEFT, RIGHT.
  - **Transition Probabilities:** Moving UP has an 80% chance of success, 10% chance of going LEFT, and 10% RIGHT.
  - **Rewards:** +1 for reaching the goal, -1 for hitting a hazard, -0.04 for each step.
  - **Discount Factor:** γ = 0.9 (for example).

2. Policy

- **Definition:** A policy (π) is a strategy that tells the agent what action to take in each state. It can be deterministic (π(s) = a) or stochastic (π(a|s) = P(a|s)).
- **Reactive Agent:** An agent that repeatedly:
  1. Observes the current state.
  2. Chooses an action based on the policy.
  3. Executes the action, receives a reward, and transitions to the next state.
- **Example (CartPole):** A simple policy might be: "If the pole is leaning to the left, move the cart to the left; if it's leaning to the right, move the cart to the right."

3. Value Function

- **Definition:** A value function V(s) estimates the expected long-term return (cumulative reward) starting from a given state s under a specific policy.
- **Update Rule (Deterministic):** $V(s) = R(s) + \gamma \cdot V(s')$
- **Update Rule (with Discount Factor):** $V(s) = R(s) + \gamma \cdot V(s')$
- **Example:** If being in state A gives an immediate reward of 0 and leads to state B, which has a value of 10, then $V(A) = 0 + \gamma \cdot 10$. If $\gamma = 0.9$, then $V(A) = 9$.

4. Q-Learning

- **Definition**: Q-learning is an off-policy RL algorithm that learns the optimal action-value function (Q-function). The Q-function, $Q(s, a)$, represents the expected cumulative reward of taking action a in state s and following the optimal policy thereafter.
- **Update Rule**:
  - $Q(s, a) \leftarrow Q(s, a) + \alpha \cdot [r + \gamma \cdot \max_{a'} Q(s', a') - Q(s, a)]$
  - Where:
    - α is the learning rate.
    - r is the immediate reward.
    - γ is the discount factor.
    - s' is the next state.
    - a' is the action to take in the next state.
- **Example (Q-Table Update)**:

State	Action = Left	Action = Right
Blank	0.05	0.05
...	...	...

5. Exploration vs. Exploitation

- **Exploration**: Trying out different actions to discover potentially better rewards.
- **Exploitation**: Choosing the action that is currently believed to be the best (highest Q-value).
- **ε-Greedy**: A common strategy where, with probability ε, the agent explores a random action; otherwise, it exploits the best-known action.

MDP Components

Component	Description	Example (Robot Grid)
States (S)	Set of possible situations.	Each cell in the grid.
Actions (A)	Set of possible moves the agent can take.	UP, DOWN, LEFT, RIGHT.
Transition Probabilities	Probability of moving from state s to s' after action a.	80% chance of moving in the intended direction, 10% chance veering left/right.
Reward Function	Reward received after taking action a in state s.	+1 at goal, -1 at hazard, -0.04 per step.
Discount Factor (γ)	Determines the importance of future rewards (0 to 1).	0.9 (prioritize immediate rewards slightly).

Q-Learning Update Rule Components

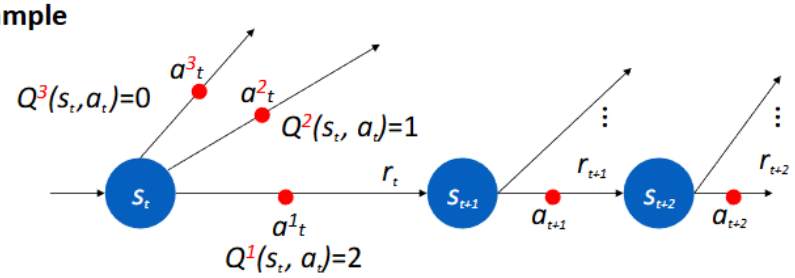
Component	Description
$Q(s, a)$	The current Q-value for taking action a in state s.
α	Learning rate (how much to update the Q-value).
r	Immediate reward received after taking action a in state s.
γ	Discount factor (importance of future rewards).
s'	The next state reached after taking action a in state s.
$\max_{a'} Q(s', a')$	The maximum Q-value achievable from the next state s' (using any action a').

Value function vs Q-function

The Q value when an agent take action a_t in state s_t at time t, the predicted future rewards is defined as

$$Q(s_t, a_t) = E[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \cdots]$$

Example



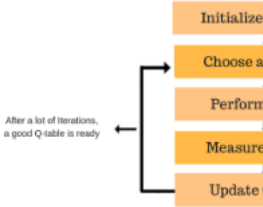
An agent should take action a^1_t because the corresponding Q value $Q^1(s_t,a_t)$ *is the maximum value.*

Q-learning - augments value iteration by maintaining estimated utility value Q(s,a) for every action at every state. Can handle problems with stochastic transitions and rewards without requiring adaptations.

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{current value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{current value}}\Big)}^{\text{temporal difference}}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{new value (temporal difference target)}}$$

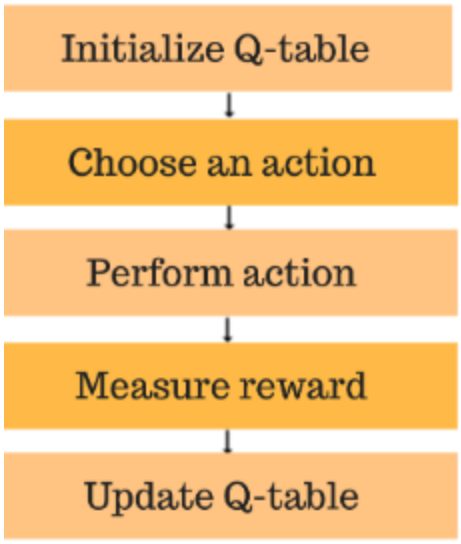
Q-Learning $Q : S \times A \rightarrow \mathbb{R}.$

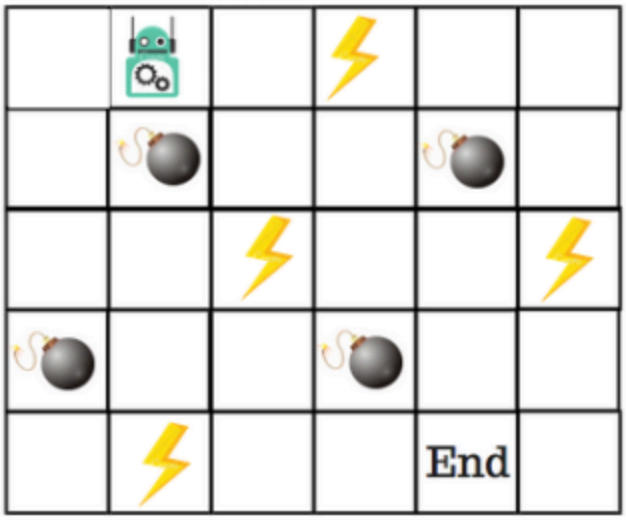


1. Set the gamma parameter, and environment rewards in matrix R
2. Initialize matrix Q to zero
3. Select a random initial state
4. Set initial state = current state
5. Select one among all possible actions for the current state
6. Using this possible action, consider going to the next state
7. Get maximum Q value for this next state based on all possible actions
8. Compute: Q(state, action) = R(state, action) + Gamma * Max[Q(next state, all actions)]
9. Repeat above steps until current state = goal state

Q-Learning algorithm

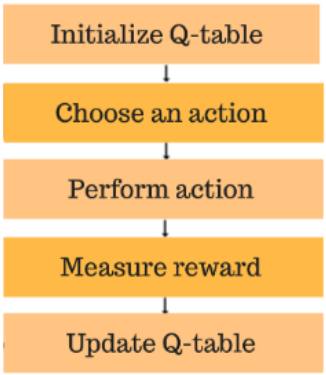
2. Choose an action (Right)
3. Perform action

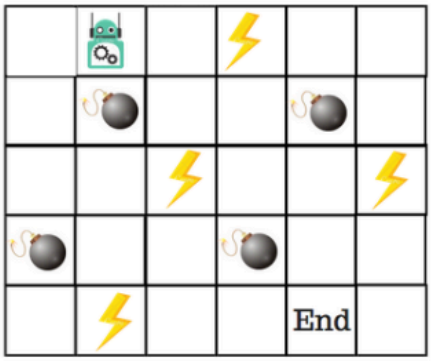




Q-Learning algorithm

4. Measure reward





Check reward with respect to the environment
In this regard, $r = 0$
If we move the robot to the right for 3 times then $r = 1$

Q-Learning algorithm

4. Measure reward

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \left[r + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

Current Q-Value

Learning rate

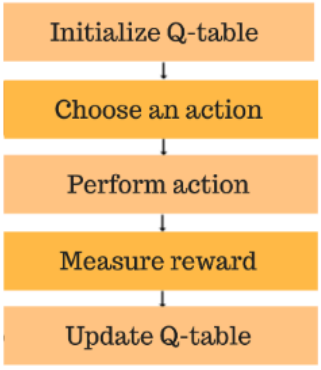
Reward

Discount factor

Maximum expected future reward given the new state and all possible actions at that new state

Q-Learning algorithm

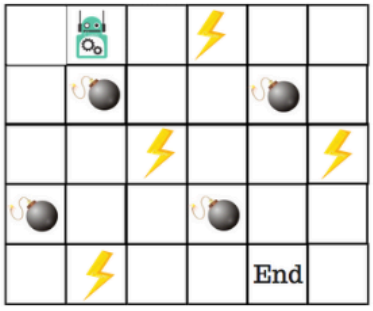
4. Measure reward



$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \left[r + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

Iterative update through the Bellman equation

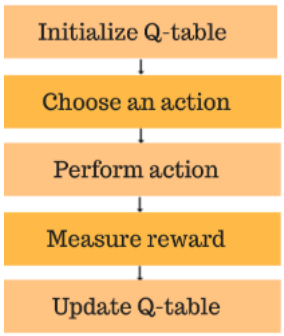
Actions :	↑	→	↓	←
Start	0	0	0	0
Nothing / Blank	0	0	0	0
Power	0	0	0	0
Mines	0	0	0	0
END	0	0	0	0

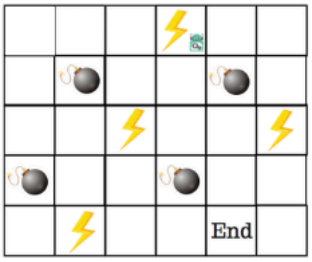


Q-Learning algorithm

4. Measure reward

Example after taking right 3 times





$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha\left[r + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\right]$$

$$Q(Blank, Right) = Q(Blank, Right) + 0.05\,[1 + 0.99 \times 0 - Q(Blank, Right)]$$

$$Q(Blank, Right) = 0 + 0.05\,[1 + 0.99 \times 0 - 0] = 0.05$$

Q-Learning algorithm

$$Q^*(s, a) = \max_\pi E\left[\sum_{t \geq 0} \gamma^t r^t | s_t = s, a_0 = a, \pi\right]$$

$$Q^*(s, a) = E_{\gamma' \sim \varepsilon}\left[\sum_{t \geq 0} \underbrace{r + \gamma \max_{a'} Q^*(s', a')}\ |s, a\right]$$

If the optimal values for the next time step $Q^*(s', a')$ is known

Just need to maximize

$$Q_{i+1}(s, a) = E\left[r + \gamma \max_{a'} Q^*(s', a')\ |s, a\right]$$

The value of Q_i will converge to Q^* as the i goes to **infinity**

Advancement of the algorithm

$$Q_{i+1}(s,a) = E\left[r + \gamma \max_{a'} Q^*(s',a') \,|s,a\right]$$

Not scalable!
Must compute $Q(s,a)$ for every state-action pair.
If state is e.g. current game state pixels, computationally infeasible to compute for entire state space!

Solution: use a function approximator to estimate $Q(s,a)$. E.g. a neural network
Deep Q-learning or Deep Q Network(DQN), if the function approximator is a deep neural network

The Q-value function is modified to $Q(s,a;\theta) \approx Q(s,a)$

Exploration policy

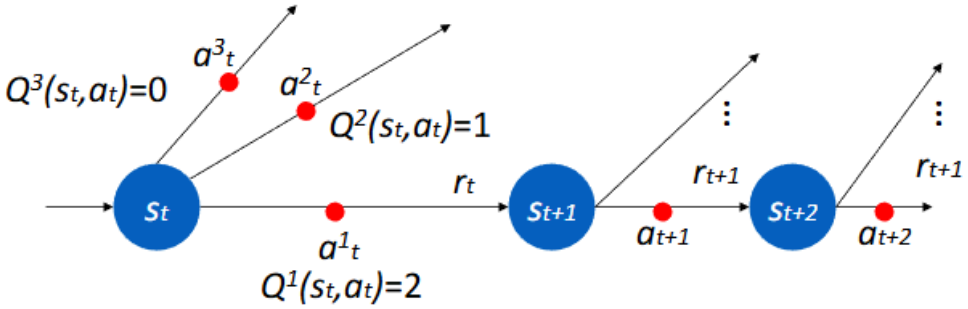
Wacky approach
exploration
act **randomly** in hopes of eventually exploring entire environment

- vs -

Greedy approach
Exploitation
act to maximize utility using current estimate

Exploration policy

- Reasonable balance
  - act more wacky (exploratory) when agent **has little idea** of environment
  - act more greedy when the model is **close to correct**



Normally, we go with $Q^1(s_t,a_t)=2$ because of the highest Q value, with probability ε, the agent explores by choosing a random action (Exploration).

1 MDP Definition (What we're working with)

An **MDP** is defined as:

$$\langle S, A, P, R, \gamma \rangle$$

Symbol	Meaning
S	Set of states
A	Set of actions
(P(s'	s,a))
$R(s, a)$	Reward function
γ	Discount factor

2 Example Problem (We'll Use This Throughout)

Grid World (Classic Exam Example)

States:

- $S = \{A, B, C\}$

Actions:

- $A = \{\text{Left}, \text{Right}\}$

Rewards:

- Reaching **C** gives **+10**
- Every move costs **−1**

Discount factor:

$$\gamma = 0.9$$

Transition Probabilities

From	Action	To	Probability
A	Right	B	1.0
B	Right	C	0.8
B	Right	B	0.2
B	Left	A	1.0
C	Any	C	1.0 (terminal)

3 Return Formula (Long-Term Reward)

Formula:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots$$

Meaning:

- Adds **future rewards**
- Discounted by γ

Example

If rewards over time are:

$$-1, -1, +10$$

$$G_0 = -1 + 0.9(-1) + 0.9^2(10)$$

$$G_0 = -1 - 0.9 + 8.1 = 6.2$$

✅ **Used to evaluate policies**

4️⃣ State Value Function $V(s)$

Formula:

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

Bellman Expectation Equation:

$$V^{\pi}(s) = \sum_{a} \pi(a|s) \sum_{s'} P(s'|s,a)\Big[R(s,a) + \gamma V^{\pi}(s')\Big]$$

Example: Compute $V(B)$ if policy always goes Right

$$V(B) = 0.8(10 + 0.9V(C)) + 0.2(-1 + 0.9V(B))$$

Since **C is terminal**, $V(C) = 0$:

$$V(B) = 0.8(10) + 0.2(-1 + 0.9V(B))$$

$$V(B) = 8 - 0.2 + 0.18V(B)$$

$$0.82V(B) = 7.8 \Rightarrow V(B) = 9.51$$

✅ **This shows how Bellman equations are solved**

5 Action-Value Function $Q(s, a)$

Formula:

$$Q^{\pi}(s,a) = \sum_{s'} P(s'|s,a)\Big[R(s,a) + \gamma V^{\pi}(s')\Big]$$

Example: Compute $Q(B, \text{Right})$

$$\begin{aligned} Q(B,\text{Right}) &= 0.8(10 + 0.9 \cdot 0) + 0.2(-1 + 0.9 \cdot 9.51) \\ &= 8 + 0.2(-1 + 8.56) \\ &= 8 + 1.51 = 9.51 \end{aligned}$$

✅ Used to compare actions

---

6 Bellman Optimality Equation

State Value:

$$V^*(s) = \max_a \sum_{s'} P(s'|s,a)\Big[R(s,a) + \gamma V^*(s')\Big]$$

Action Value:

$$Q^*(s,a) = \sum_{s'} P(s'|s,a)\Big[R(s,a) + \gamma \max_{a'} Q^*(s',a')\Big]$$

Example: Choose Best Action at B

$$Q(B,\text{Right}) = 9.51$$

$$Q(B,\text{Left}) = -1 + 0.9V(A)$$

Since $Q(B,\text{Right}) > Q(B,\text{Left})$

✅ Optimal policy chooses Right

7 Policy Improvement

Formula:

$$\pi'(s) = \arg\max_a Q^\pi(s,a)$$

Meaning:

- Update policy using computed Q-values

Example:

If:

- $Q(B, \text{Right}) > Q(B, \text{Left})$

Then:

$$\pi(B) = \text{Right}$$

---

8 Value Iteration (Algorithmic Use)

Update Rule:

$$V_{k+1}(s) = \max_a \sum_{s'} P(s'|s,a)\left[R(s,a) + \gamma V_k(s')\right]$$

Example (One Iteration)

Initial:

$$V_0(A) = V_0(B) = V_0(C) = 0$$

Update B:

$$V_1(B) = \max(9.51, -1) = 9.51$$

Repeat until convergence.

9 Policy Iteration (Two Steps)

Step 1: Policy Evaluation

$$V^{\pi}(s) = \sum_{a} \pi(a|s) \sum_{s'} P(s'|s,a)[R + \gamma V^{\pi}(s')]$$

Step 2: Policy Improvement

$$\pi'(s) = \arg\max_{a} Q^{\pi}(s,a)$$

Repeat until stable.

---

10 Summary Table (Exam Cheat Sheet)

Concept	Formula Used For
Return G_t	Long-term reward
$V(s)$	Value of state
$Q(s,a)$	Value of action
Bellman Expectation	Policy evaluation
Bellman Optimality	Finding optimal policy
Value Iteration	Direct optimization
Policy Iteration	Evaluate → Improve

RL Algorithms

Value-based

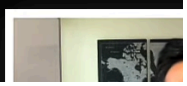
Value function → Optimal Policy

total reward

Q-Learning

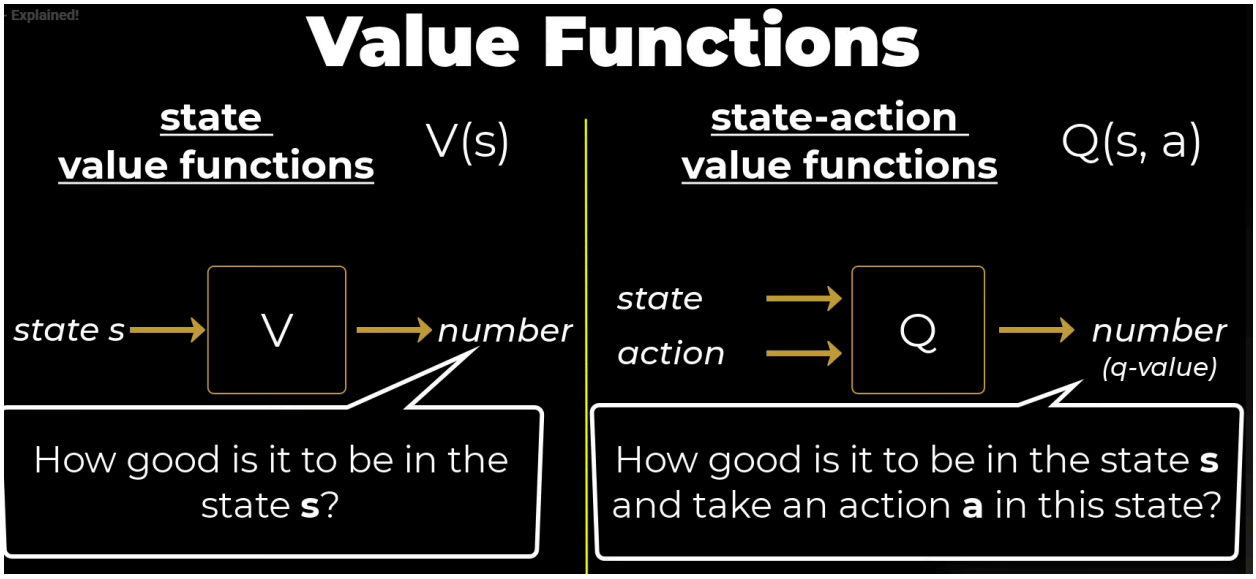
Policy-based

Optimal Policy

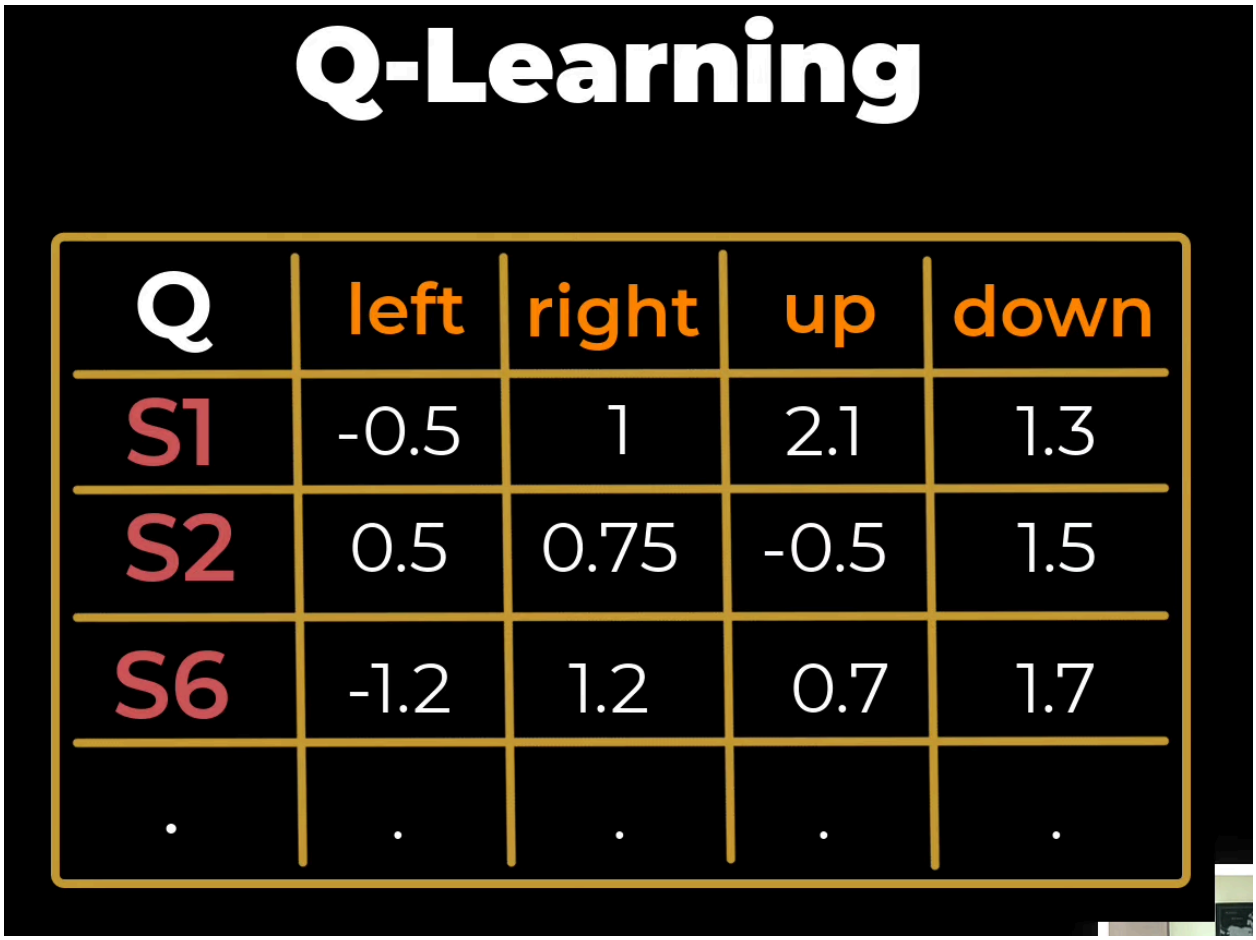


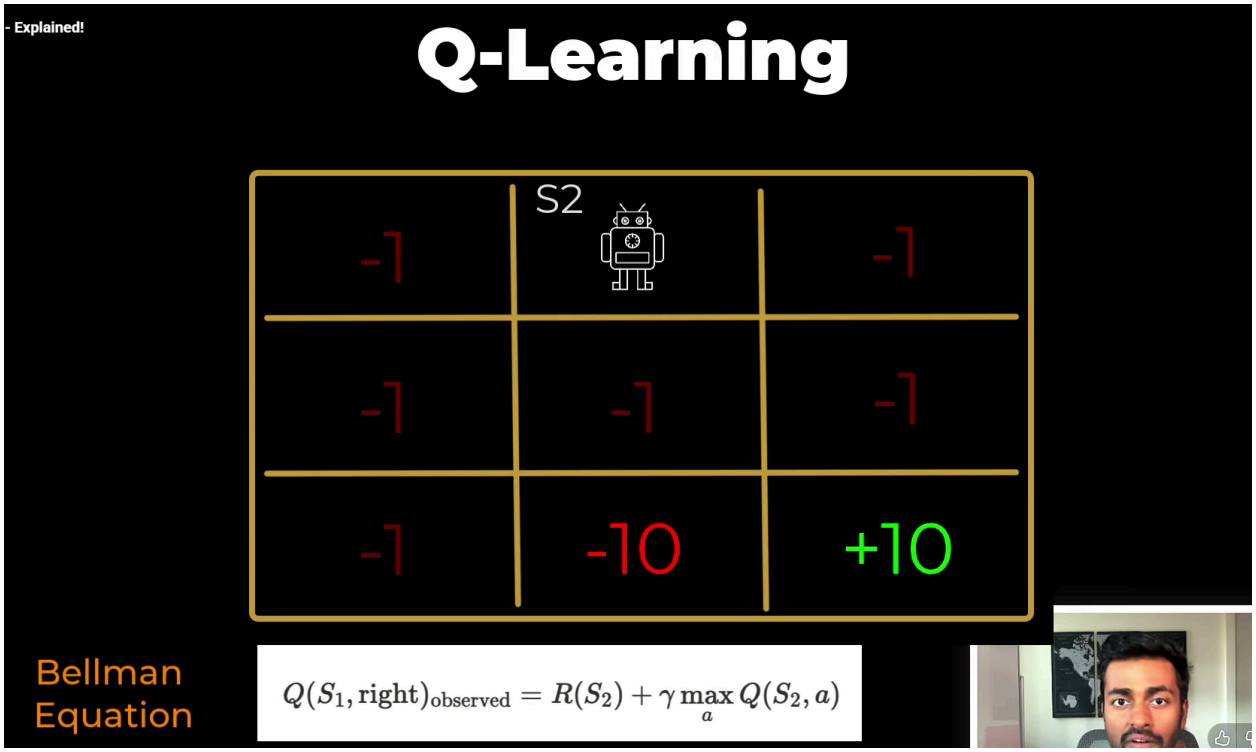
Value-based Methods

Value function → *determines* → Policy → *that maximizes* → total reward



Target policy - optimal policy, action takes depends on behavior policy. The policy the agent is trying to learn





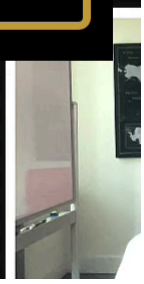
Reward for S2 = -1, discount factor = 0.1, max q value from s2 = 1.5

Q	left	right	up	down
S1	-0.5	1	2.1	1.3
S2	0.5	0.75	-0.5	1.5
S6	-1.2	1.2	0.7	1.7
.	.	.	.	.

an
ion

$$Q(S_1, \text{right})_{\text{observed}} = R(S_2) + \gamma \max_a Q(S_2, a)$$

= -1 + 0.1 * 1.5 **=-0.85**



Temporal Difference Error

$$\text{TD Error} = Q(S_1, \text{right})_{\text{observed}} - Q(S_1, \text{right})_{\text{expected}}$$

= -0.85 - 1 **=-1.85**

Difference timesteps (-0.85 - 1)

Update Rule

$$Q(S_1, \text{right}) = Q(S_1, \text{right}) + \alpha \times \text{TD Error}$$

= 1 + 0.1 * -1.85 **= 0.815**

Learning rate = 0.1

No Q-table? Compute using neural network
Neural network approach

Step 1: Initial network (no knowledge)

At the start, the network is random.

Example (purely from random weights):

```markdown
Qθ(S1) = [ 0.02, -0.11, 0.07, -0.03 ]
         L     R     U     D
```

So:

$$Q(S1, \text{right}) = -0.11$$

No table. No prior knowledge.

Step 2: Forward pass on next state

We move to **S2**, then evaluate it using the **same network**:

```
Qθ(S2) = [ -0.04, 0.01, 0.09, -0.02 ]
```

So:

$$\max_a Q(S2, a) = 0.09$$

⚠ This is **not learned truth** — just the network's current guess.

Step 3: Compute the TD target (this is valid)

$$\text{target} = r + \gamma \max_a Q(S2, a)$$

$$\text{target} = -1 + 0.1 \times 0.09 = -0.991$$

This uses:

- reward from environment ✔
- network prediction ✔
- NO table ✔

Step 4: Compute TD error

$$\delta = \text{target} - Q(S1, \text{right})$$

$$\delta = -0.991 - (-0.11) = -0.881$$

Step 5: Update the network (not a value)

We do NOT write:

```sql
Q(S1, right) = something
```

Instead, we:

- Minimize this loss:

$$L = (Q(S1, \text{right}) - \text{target})^2$$

- Backpropagate
- Update weights

Effect:

> The network is nudged so that
> next time **Q(S1, right)** is **slightly more negative**

---

1 In the parameters (permanently)

- Stored as **weights** θ
- This is the *compressed representation* of all estimates

You cannot read it as a table, but it *implicitly contains* all Q-estimates.

---

2 As a number when you query the model (explicitly)

When you input a state s:

```css
Qθ(s) → [ -2.1, -0.4, -3.0, -2.7 ]
```

Then:

$$\hat{Q}(s, \text{right}) = -0.4$$

📌 **That number is the estimation value**
📌 It only exists **when evaluated**

After training:

```makefile
state = S1
Q0(S1) = [ -2.3, -0.9, -3.1, -2.8 ]
```

Copy code

- Estimated value for `right`:

$$\hat{Q}(S1, \text{right}) = -0.9$$

Bellman Equation

$$Q(S_2, \text{down})_{\text{observed}} = R(S_6) + \gamma \max_a Q(S_6, a)$$

Bellman Equation

$$Q(S_2, \text{down})_{\text{observed}} = R(S_6) + \gamma \max_a Q(S_6, a)$$

= -1 + 0.1 * 1.7 **=-0.83**

Temporal Difference Error

$$\text{TD Error} = Q(S_2, \text{down})_{\text{observed}} - Q(S_2, \text{down})_{\text{expected}}$$

= -0.83 - 1.5 **=-2.33**

Update Rule

$$Q(S_2, \text{down}) = Q(S_2, \text{down}) + \alpha \times \text{TD Error}$$

= 1.5 + 0.1 * (-2.33) **=1.267**

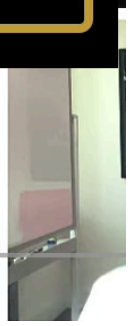
Update table

Q	left	right	up	down
S1	-0.5	0.815	2.1	1.3
S2	0.5	0.75	-0.5	1.267
S6	-1.2	1.2	0.7	1.7
.	.	.	.	.

Q	left	right	up	down
S1	-0.5	0.815	2.1	1.3
S2	0.5	0.75	-0.5	1.5
S6	-1.2	1.2	0.7	1.7
.	.	.	.	.

$$Q(S_1, \text{right}) = Q(S_1, \text{right}) + \alpha \times \text{TD Error}$$

= 1 + 0.1 * -1.85 = 0.815



Update 1 -> 0.815
Week 8