

Artificial Intelligence

09.10.2019

Khean Sreythou

Batch 4

Year 2

KIT



Content

Introduction of AI	5
What is AI?	5
Capability to be added to machines to call it as intelligent	6
Goals of AI	6
Intelligent Agent	7
What is Agent?	7
Vacuum Cleaner Agent Example	8
Agent vs. Program	9
Concept of Rationality	9
Nature of Environments	10
File manager agent example	10
Properties of Task Environment	10
Fully observable vs. Partially observable	10
Single agent vs. Multi-agent	10
Deterministic vs. Stochastic	11
Static vs. Dynamic	11
Episodic vs. Sequential	11
Discrete vs. Continuous	11
Known vs. unknown	12
Agent Types	13
Simple Reflex Agents	13
Model Based Reflex Agents	14
Goal Based Agents	15
Utility Based Agents	15
Learning Agents	16
Agent Based Modelling	16
Time & Space Complexity	18
Complexity	18
Space Complexity	18
Time Complexity	18
NP, NP Complete and NP Hard	18
Problem-Solving Agents	19
Problem as a state space search	19
8 Queens Problem	19
Water Pouring Problem	19

Uninformed Search Strategies	21
Tree-Search Algorithms	21
Search Strategies	21
Uninformed Search	22
Breadth-First Search	22
Uniform-Cost Search	23
Depth-First Search	23
Iterative Deepening Search	24
Bi-directional Search	25
Informed (Heuristic) Search Strategies	27
Greedy Best-First Search	28
A* Search	29
Conditions for optimality	30
Memory-bounded Heuristic Search	32
Recursive Best-First Search (RBFS)	32
Local Search Algorithm	34
State-Space Landscape	34
Hill-Climbing Search	34
Problems with Hill-Climbing Search	35
Solutions to Hill-Climbing Search	36
Simulated Annealing	37
Simulated annealing Acceptance Function	38
Algorithm for SA	38
Temperature Initialization	38
Genetic Algorithm	39
Basic Terminology	39
Knapsack Problem	40
Constraint Satisfaction Problems	42
Map Coloring	42
Job-shop Scheduling	44
Logical Agents	45
Knowledge-based Agents	45
Wumpus World	45
Logic	49
Propositional Logic	49
A Simple Knowledge Base: Wumpus World	49
Equivalence, Validity, Satisfiability	50

Inference Rules	51
Use in Wumpus World	51
Define a Proof Problem	52
Proof by Resolution	52
Full Resolution Rule	53
Conjunctive Normal Form	54
Converting a Sentence into CNF	54
Example Proof By Deduction	54
Evaluation of Deductive Inference	55
Resolution	55
Proof Using Resolution	55
Evaluation of Resolution	56
Horn Clauses	57
Reasoning with Horn Clauses	57
Forward Chaining	57
Backward Chaining	59
Forward Chaining vs. Backward Chaining	60
Robinson's Inference Rule	61
Example	61
Scene Interpretation using Predicate Logic	61
Practice	61
Machine Learning	64
Supervised Learning	64
Unsupervised Learning	65
Reinforcement Learning	65
Linear Regression	65
Overfitting and Underfitting	65
Support Vector Machine	66
How does SVM work?	67
Dealing with non-linear and inseparable planes	68
Advantages	68
Disadvantages	68
Naive Baye's Classifier	69
How it works	69
Decision Tree Algorithm	72
What is Decision Tree?	72
How does it work?	73
Advantage	73




Disadvantage	74
K-nearest Neighbor	74
How it works	74
Drawback of KNN	75
K-mean Clustering	75
Reinforcement Learning	75
What is reinforcement learning?	76
Reinforcement vs. Supervised vs. Unsupervised	76
Basic Terminology	76
How it works	76
Deep Learning	78
CNN - Convolutional Neural Network	78
Forward propagation	79
Pooling	79
Activation Function	80
Loss Function	80
Backpropagation	81



Introduction of AI

What is AI?

- **What is Intelligence:** the general mental ability to learn and apply knowledge to manipulate your environment, as well as the ability to reason and have abstract thought.
- **What is Artificial Intelligence:** Artificial Intelligence is a way of making a computer, a computer-controlled robot, or a software think intelligently, in a similar manner the intelligent humans think.
- **Systems that think like humans**
 - Cognitive Modeling Approach
 - “The automation with human thinking”
 - Bellman, 1978
- **Systems that act like humans**
 - Turing Test Approach
 - “The art of creating machines that perform functions that require intelligence when performed by people.”
 - Kurzweil, 1990
- **Systems that think rationally**
 - “Laws of Thought” approach
 - “The study of mental faculties through the use of computational models”
 - Charniak and McDermott
- **Systems that act rationally**

- 
- Rational Agent Approach
 - “The branch of Computer Science that is concerned with the automation of intelligent behavior”
 - Luger and Stubblefield

Capability to be added to machines to call it as intelligent

- **Natural Language Processing:** for communication with human
 - Natural Language Understanding
 - Natural Language Generation
- **Knowledge Representation:** to store information effectively & efficiently.
- **Automated reasoning:** to retrieve & answer questions using the stored information.
- **Machine Learning:** how to build computer systems that adapt and improve with experience.
- **Computer Vision:** is to give computers this powerful facility for understanding their surroundings. **Robotics:** is an electro – mechanical device that can be programmed to perform manual tasks.

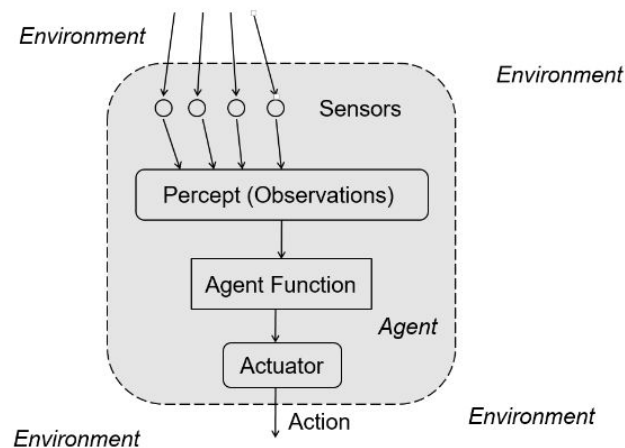
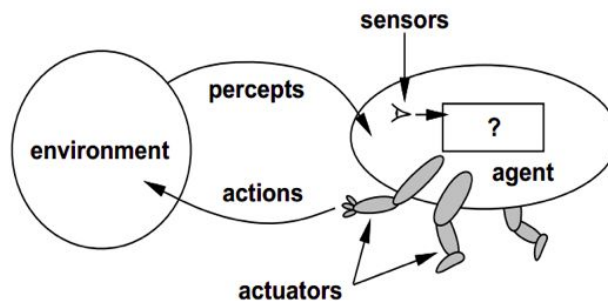
Goals of AI

- To make computers more useful by letting them take over dangerous or tedious tasks from humans.
- To understand human intelligence in Machine

Intelligent Agent

What is Agent?

- An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators.



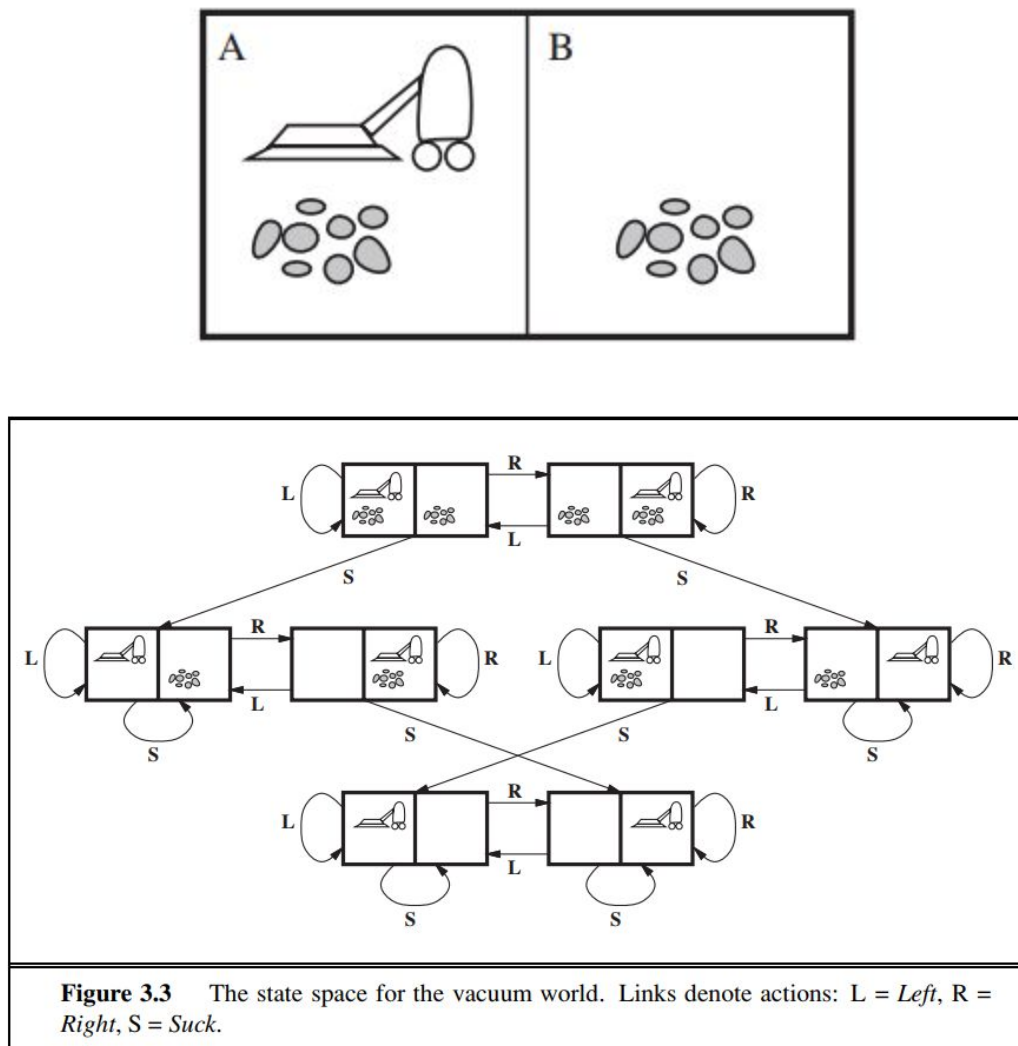
- **Percept:** is the agent's perceptual inputs at any given instant.
- **Agent Function/Agent Program:** is an abstract mathematical description; the agent program is a concrete implementation, running within some physical system.
- The agent function maps from percept histories to actions:

$$f : P^* \rightarrow A$$


- The agent program runs on the physical architecture to produce f .

Vacuum Cleaner Agent Example

- Squares A and B. The vacuum agent perceives which square it is in and whether there is dirt in the square. It can choose to move left, move right, suck up the dirt, or do nothing. One very simple agent function is the following: if the current square is dirty, then suck; otherwise, move to the other square.



- States:** is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt. Thus, there are $2 \times 2^2 = 8$ possible world states. A larger environment with n locations has $n \times 2^n$ states.

- 
- **Initial state:** Any state can be designated as the initial state.
 - **Actions:** Each state has just three actions: Left, Right, and Suck. Larger environments might also include Up and Down.
 - **Transition model:** The actions have their expected effects, except that moving Left in the leftmost square, moving Right in the rightmost square, and Sucking 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.

Agent vs. Program

- **Size:** an agent is usually smaller than a program.
- **Purpose:** an agent has a specific purpose while programs are multi-functional.
- **Persistence:** an agent's life span is not entirely dependent on a user launching and quitting it.
- **Autonomy:** an agent doesn't need the user's input to function.

Concept of Rationality

- For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.
- **Rationality** maximizes expected performance, while **perfection** maximizes actual performance.
- **Information Gathering & Exploration:** Doing actions in order to modify future percepts.
- **Autonomy:** To the extent that an agent relies on the prior knowledge of its designer rather than on its own percepts, we say that the agent lacks autonomy. A rational agent should be autonomous it should learn what it can to compensate for partial or incorrect prior knowledge.

Nature of Environments

Task Environment: PEAS (Performance, Environment, Actuators, Sensors)

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

File manager agent example

- **Sensors:** file explorer
- **Actuators:** commands like WinZip, WinRAR
- **Purpose:** Compress & achieve fields that have not been used in a while
- **Environment:** fully observable (but partially observed), deterministic (strategic), episodic, dynamic, discrete


Properties of Task Environment

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 ⇒ **fully observable**
- **Partially observable** ⇒ because of noisy and inaccurate sensors or because parts of the state are simply missing from the sensor data.
- **Unobservable:** If the agent has no sensors at all then the environment is unobservable.

Single agent vs. Multi-agent

- Single agent: crossword puzzle
- Two agent: chess playing
- Competitive multi agent environment: chess

- 
- Cooperative multi agent environment:
 - Automated taxi driver
 - Avoiding collision

Deterministic vs. Stochastic

- If the next state of the environment is completely determined by the current state and the action executed by the agent, then \Rightarrow **deterministic**; otherwise, \Rightarrow **stochastic**.
- **Uncertain**: if it is not fully observable or not deterministic.

Static vs. Dynamic

- A dynamic environment is always changing overtime. E.g., the number of people in the street.
- **Static environment**: e.g., the destination
- **Semi dynamic**: environment is not changed overtime but the agent's performance score does.

Episodic vs. Sequential

- **Episode** = agent's single pair of perception & action. The quality of the agent's action doesn't depend on other episodes. Every episode is independent of each other. Episodic environment is simpler, the agent doesn't need to think ahead.
- **Sequential**: Current action may affect all future decisions. E.g., Taxi driving and chess.

Discrete vs. Continuous

- If there are a limited number of distinct states, clearly defined percepts and actions, the environment is discrete. E.g., Chess game
- **Continuous**: Taxi driving with different speed and location sweep through a continuous range of values over a period of time

Known vs. unknown

- **Known** : the outcomes for all actions are given.
- **Unknown** : the agent will have to learn how it works in order to make good decisions.
- Known environment \Rightarrow partially observable
- Unknown environment \Rightarrow fully observable

Task Environment	Observable	Agents	Deterministic	Episodic	Static	Discrete
Taxi driving	Partially	Multi	Stochastic	Sequential	Dynamic	Continuous

Structure of Agent

agent = architecture + program

- AI design an agent program that implements the agent function the mapping from percepts to actions.
- Agent program: takes the current percept as input
- Agent function: takes the entire percept history.

Percept sequence	Action
$[A, \textit{Clean}]$	<i>Right</i>
$[A, \textit{Dirty}]$	<i>Suck</i>
$[B, \textit{Clean}]$	<i>Left</i>
$[B, \textit{Dirty}]$	<i>Suck</i>
$[A, \textit{Clean}], [A, \textit{Clean}]$	<i>Right</i>
$[A, \textit{Clean}], [A, \textit{Dirty}]$	<i>Suck</i>
\vdots	\vdots
$[A, \textit{Clean}], [A, \textit{Clean}], [A, \textit{Clean}]$	<i>Right</i>
$[A, \textit{Clean}], [A, \textit{Clean}], [A, \textit{Dirty}]$	<i>Suck</i>
\vdots	\vdots

Issues with table driven approach

- No physical agent in this universe will have the space to store the table
- the designer would not have time to create the table
- no agent could ever learn all the right table entries from its experience
- even if the environment is simple enough to yield a feasible table size, the designer can not fill in the table entries.

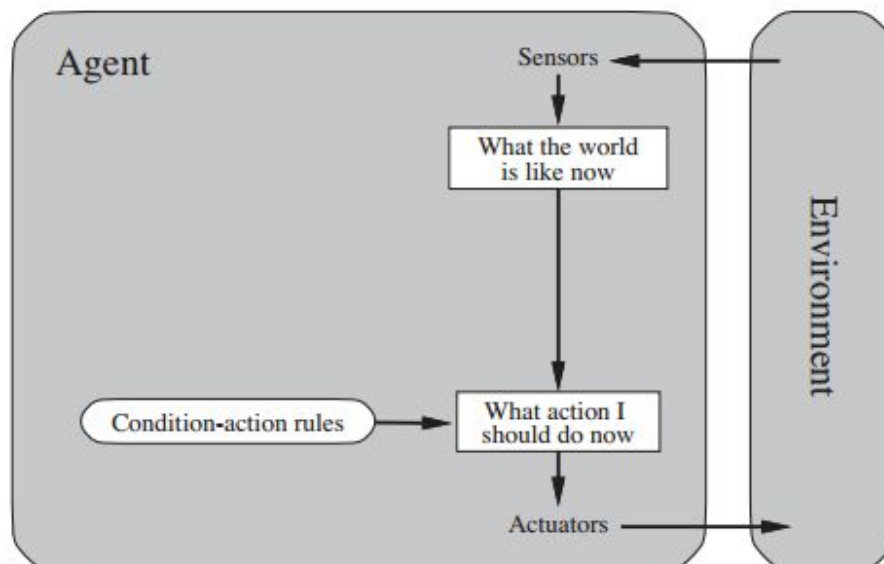
Agent Types

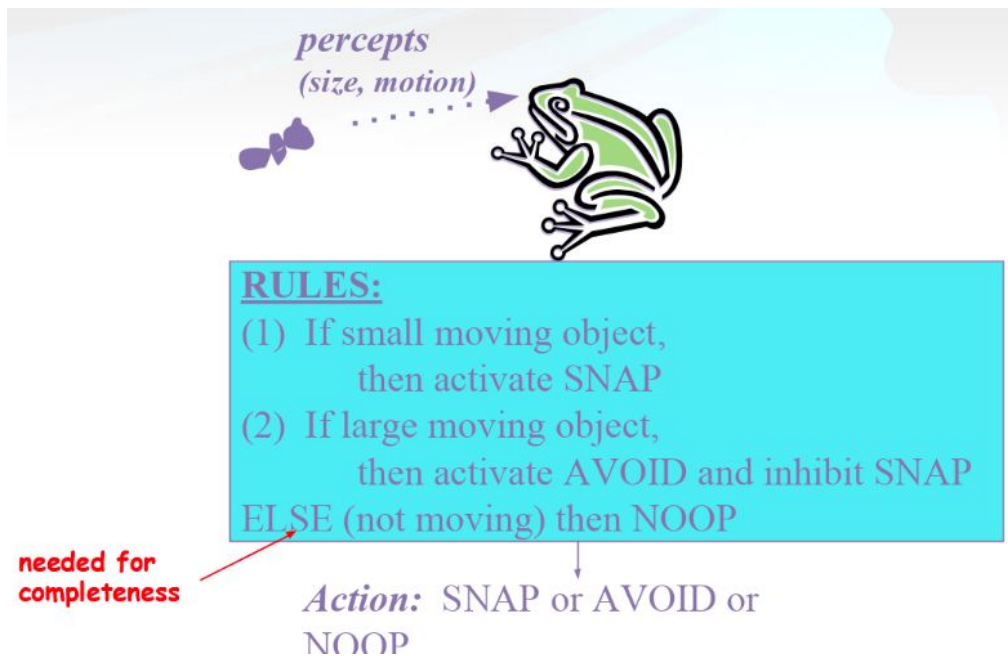
- 4 basic types in order of increasing generality:
 1. Simple reflex agents
 2. Model Based reflex agents
 3. Goal-based agents
 4. Utility-based agents

Simple Reflex Agents

It uses **condition-action rules**:

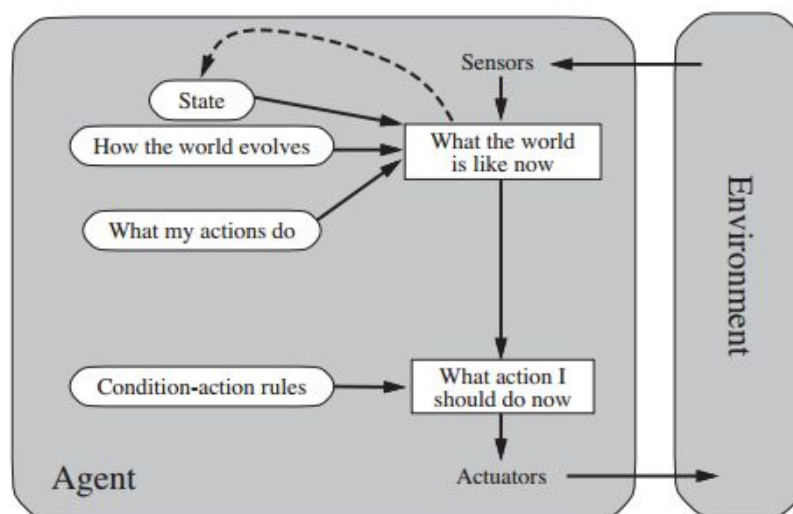
- Rule : “if ... then ...”
- Efficient but have a narrow range of applicability because knowledge sometimes cannot be stated explicitly.
- Work only if the environment is fully observable.





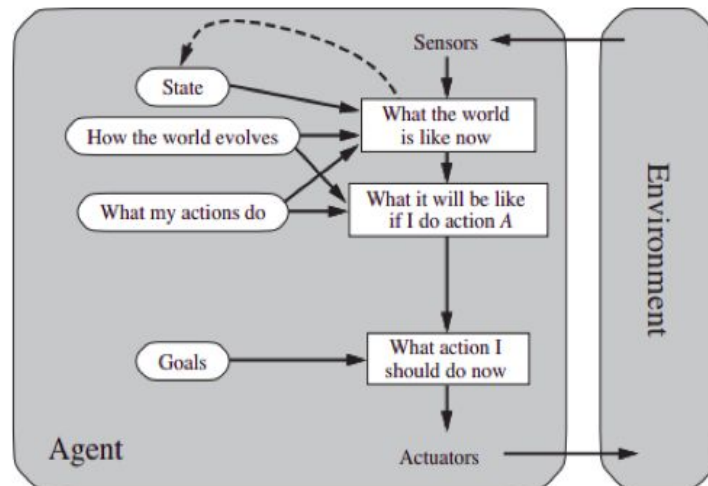
Model Based Reflex Agents

- The agent should maintain some sort of internal state that depends on the percept history and thereby reflects at least some of the unobserved aspects of the current state.
- The agent used the knowledge of **“how the world works”**—whether implemented in simple Boolean circuits or in complete scientific theories—is called **a model-based agent**.



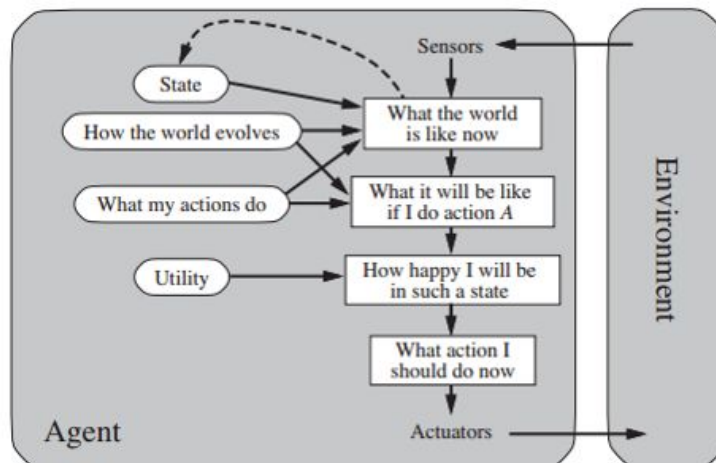
Goal Based Agents

- Less efficient but more flexible
- Search and planning two other subfields in AI to find out the action sequences to achieve its **goal** is **Goal Based Agent**



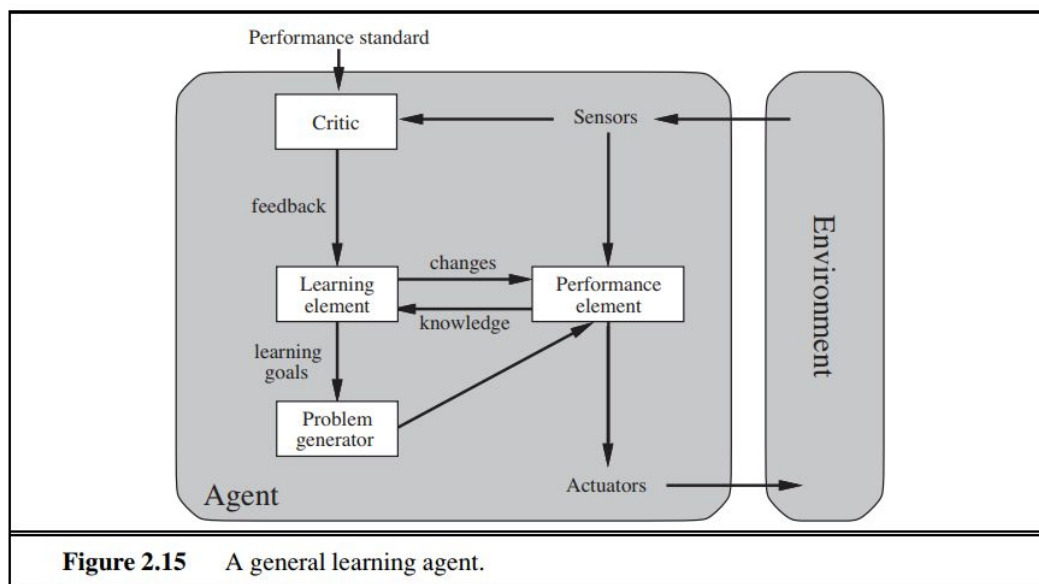
Utility Based Agents

- Goals alone are not enough to generate high-quality behavior.
- Many actions sequences the goals, some are better and some are worse.
- Goal mean **success**, then utility means **the degree of success** (how successful it is)



Learning Agents

- In AI, once an agent is done, we teach it by giving it a set of examples, test it by using another set of example ⇒ **Learning Agent**.
- 4 conceptual components:
 - **Learning element:** making improvement.
 - **Performance element:** selecting external actions.
 - **Critic:** tells the learning element how well the agent is doing with respect to fixed performance standard.
 - **Problem generator:** suggest actions that will lead to new and informative experiences.



Agent Based Modelling

- Simulation modeling technique where a system is modeled as a collection of agents and the relationships between them.
- Agents individually assess its situation in the environment and make decisions on the basis of a set of rules.
- General Characteristics of an agent:



- Autonomy
- Proactiveness
- Reactivity
- Social Ability



Time & Space Complexity

Complexity

- **Complexity** : the running time to execute a process and it depends on space as well as time.
- The focus to determine the cost is done on running time and it depends on:
 - Size of Input Data
 - Hardware
 - Operating system
 - Programming language used

Space Complexity

- **Space complexity** : The amount of computer memory required to solve the given problem of particular size.
 - **Fixed Part** - is needed for instruction space i.e byte code, variable space, constants space etc.
 - **Variable Part** - Instance of input and output data.
- $\text{Space}(S) = \text{Fixed Part} + \text{Variable Part}$

Time Complexity

- **Time complexity** : time required to analyze the given problem of particular size.
 - **Fixed Part** - Compile time
 - **Variable Part** - Run time dependent on problem instance.
- Use a **stopwatch** and time is obtained in seconds or milliseconds.
- **Step Count** - Count number of program steps.

NP, NP Complete and NP Hard

- **P complexity** : represents the set of all decision problems that can be solved in polynomial time.
- **NP complexity** : represents the set of all decision problems for which the instances where the answer is "yes" have proofs that can be verified in polynomial time.
- **NP-Complete** complexity : represents the set of all problems X in NP for which it is possible to reduce any other NP problem Y to X in polynomial time.
- **NP-hard** problems : do not have to be in NP and not have to be decision problems.



Problem-Solving Agents

Problem as a state space search

To build a system to solve a particular problem, we need:

- **Define the problem:** must include precise specification; initial solution & final solution.
- **Analyze the problem:** select the most important features that can have an immense impact.
- **Isolate and represent:** convert these important features into knowledge representation.
- **Problem solving technique(s):** choose the best technique and apply it to particular problem.

8 Queens 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. Different subsets of these are possible depending on where the blank is.
- **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.

Water Pouring Problem

- **Initial state:** The buckets are empty, represented by the tuple (0 0)
- **Goal state:** One of the buckets has two gallons of water in it, represented by either (x2) or (2x)

- **Path cost:** 1 per unit step.

□ Actions and Successor Function

✓ Fill a bucket

$(x\ y) \rightarrow (3\ y)$

$(x\ y) \rightarrow (x\ 4)$

✓ Empty a bucket

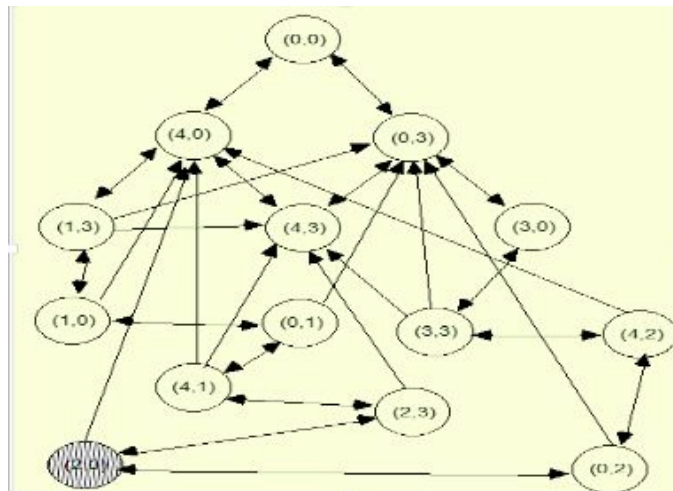
$(x\ y) \rightarrow (0\ y)$

$(x\ y) \rightarrow (x\ 0)$

✓ Pour contents of one bucket into another

$(x\ y) \rightarrow (0\ x+y)$ or $(x+y-4, 4)$

$(x\ y) \rightarrow (x+y\ 0)$ or $(3, x+y-3)$



Uninformed Search Strategies

Tree-Search Algorithms

- Searching for a (shortest / least cost) path to goal state(s).
- Search through the state space.
- We will consider search techniques that use an explicit search tree that is generated by the initial state + successor function.

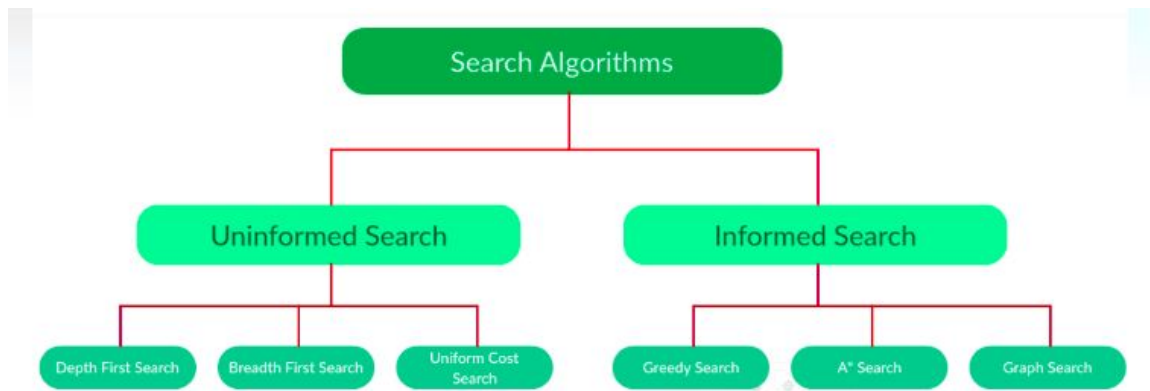
```
initialize (initial node)
Loop
    choose a node for expansion
    according to a strategy
    goal node? → done
    expand node with successor function
```

- **Basic idea:** simulated exploration of state space by generating successors of already-explored states (e.g. expanding states)

Search Strategies

- A search strategy is defined by picking the order of node expansion.
- Strategies are evaluated along the following dimensions:
 - **completeness:** does it always find a solution if one exists?
 - **time complexity:** number of nodes generated
 - **space complexity:** maximum number of nodes in memory
 - **optimality:** does it always find a least-cost solution?
- Time and space complexity are measured in terms of

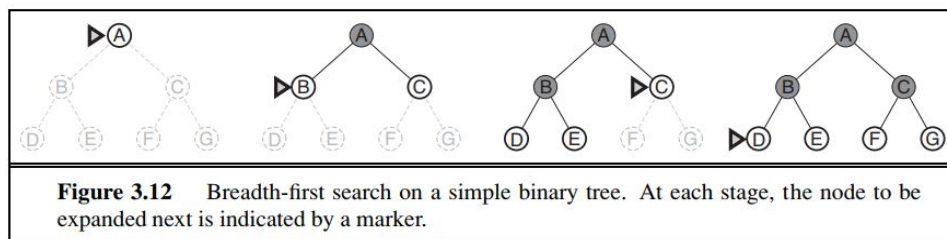
- **b**: maximum branching factor of the search tree
- **d**: depth of the least-cost solution
- **m**: maximum depth of the state space (may be ∞)



Uninformed Search

- Uninformed search strategies use only the information available in the problem definition:
 - Breadth-first search
 - Uniform-cost search
 - Depth-first search
 - Depth-limited search
 - Iterative deepening search
 - Bi-directional search

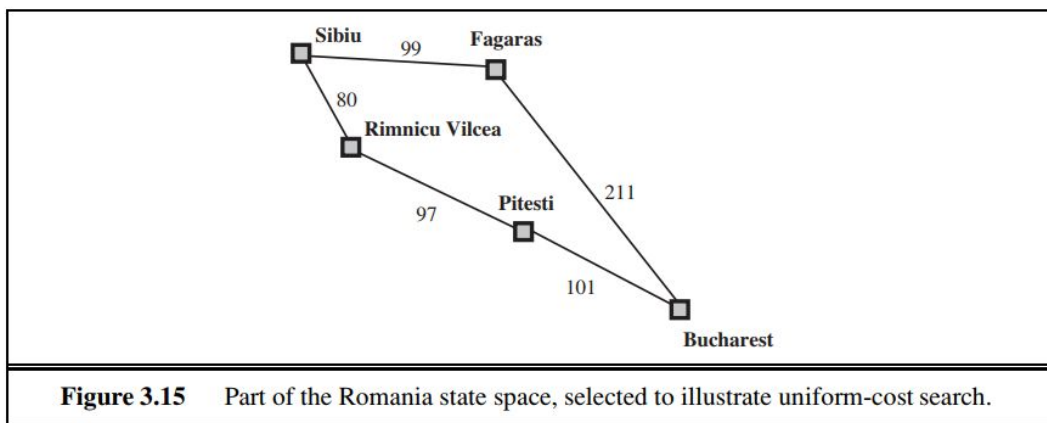
Breadth-First Search



- Expand shallowest unexpanded node.
- Implementation: fringe is a FIFO queue, i.e., new nodes go at end (First In First Out queue.)

Uniform-Cost Search

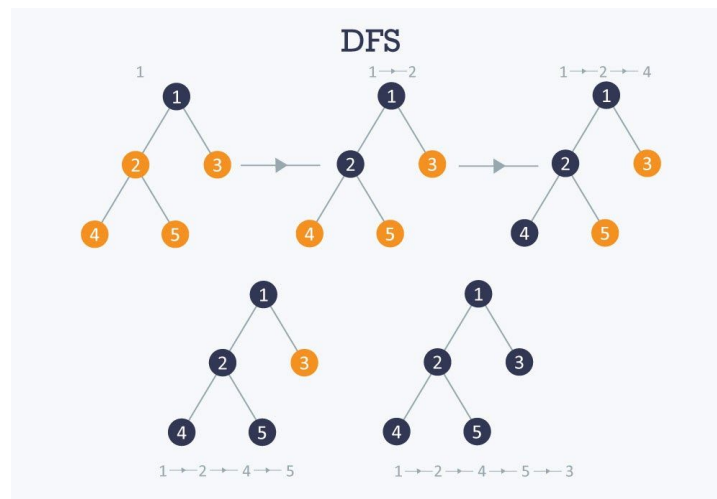
- We factor in the cost of each step (e.g., distance from current state to the neighbors). Assumption: costs are non-negative.
- $g(n)$ = cost so far to reach n
- **Queue** \Rightarrow ordered by cost
- If all the steps are the same \Rightarrow breadth-first search is optimal since it always expands the shallowest (least cost)!
- Uniform-cost search \Rightarrow expand first the nodes with the lowest cost (instead of depth).



- Expand least-cost (of path to) unexpanded node(e.g. useful for finding shortest path on map)
- Implementation: fringe = queue ordered by path cost

Depth-First Search

- “Expand deepest unexpanded node”
- Implementation: fringe = LIFO queue, i.e., put successors at front (“push on stack”) \Rightarrow Last In First Out



Iterative Deepening Search

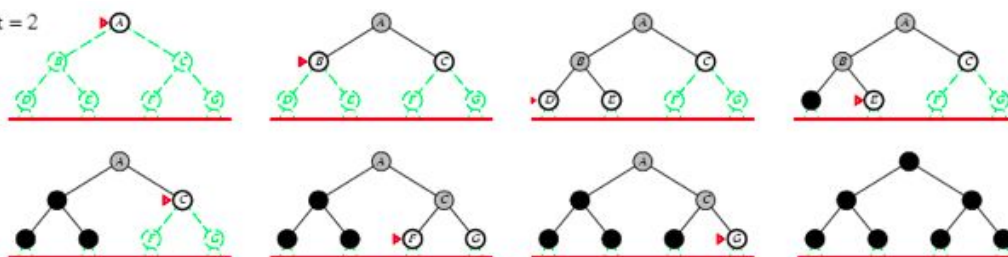
Limit = 0

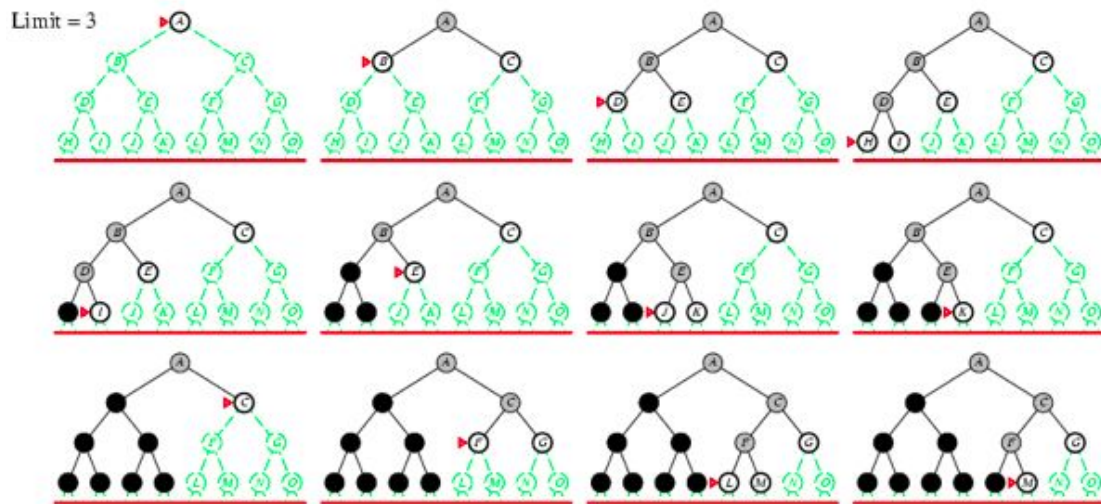


Limit = 1



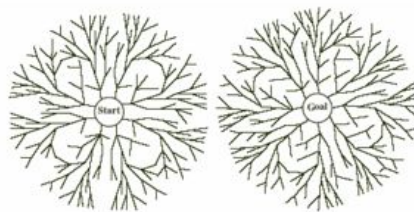
Limit = 2





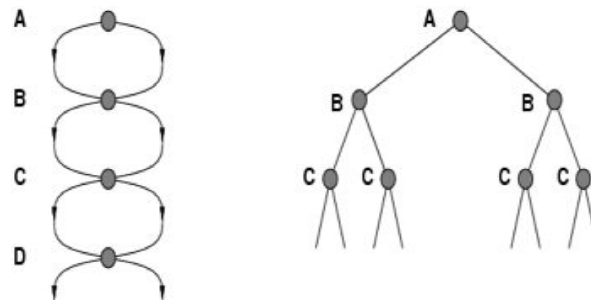
- Combine good memory requirements of depth-first with the completeness of breadth-first when branching factor is finite and is optimal when the path cost is a non-decreasing function of the depth of the node.
- **Iterative deepening search** uses only linear space and not much more time than other uninformed algorithms.
- **Iterative deepening** is the preferred uninformed search method when there is a large search space and the depth of the solution is not known.

Bi-directional Search



- To run 2 simultaneous searches:
 - forward from initial state
 - backward from the goal
- ⇒ Stop when the two searches meet.
- It is implemented by replacing the goal test with a check to see whether the frontiers of 2 searches intersect; if they do, a solution has been found.

- Checking a node for membership in the other search tree can be done in constant time (hash table)
- Key limitations:
 - Space $O(bd/2)$
 - Search backwards can be an issue (e.g., in Chess)
- Problem: lots of states satisfy the goal; don't know which one is relevant.
- Aside: The predecessor of a node should be easily computable (i.e., actions are easily reversible).
- Failure to detect repeated states can turn linear problem into an exponential one!
- Problems in which actions are reversible (e.g., routing problems or sliding-blocks puzzle). Also, in eg Chess; uses hash tables to check for repeated states. Huge tables 100M+ size but very useful.



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	$O(b^{d+1})$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^{d+1})$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes ^c	Yes	No	No	Yes ^c	Yes ^{c,d}

Figure 3.17 Evaluation of 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

- It uses problem-specific knowledge beyond the definition of the problem itself—can find solutions more efficiently than uninformed strategy ⇒ **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 implementation of best-first graph search is identical to that for uniform-cost search except for the use of f instead of g to order the priority queue.
- Most Best First Search algorithm include as a component of f a heuristic function, denoted $h(n)$:
$$h(n) = \text{estimated cost of the cheapest path from the state at node } n \text{ to a goal state.}$$
- **Important Point:**
 - $h(n)$ takes a node as input
 - $g(n)$ depends only on the state at that node
- Example: in Routing Problem, one might estimate the cost of the cheapest path from **Arad to Bucharest** via the straight-line distance from Arad to Bucharest.
- Heuristic are considered to be arbitrary, nonnegative, problem-specific functions, with one constraint: if n is a goal node, then $h(n) = 0$.

Greedy Best-First Search

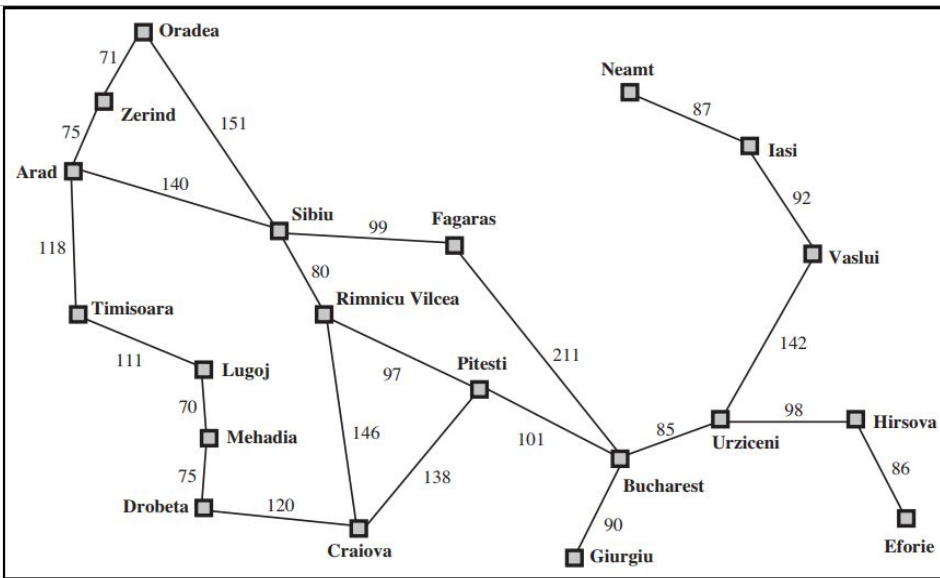


Figure 3.2 A simplified road map of part of Romania.

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

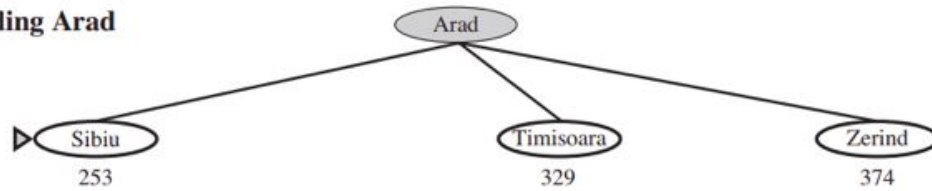
Figure 3.22 Values of h_{SLD} —straight-line distances to Bucharest.

- **Greedy best-first search** tries to expand the node that is closest to the goal that lead to a solution quickly.
- So nodes by using the heuristic function; that is, $f(n) = h(n)$.
- In Routing to Romania problem we can take Straight line distance h_{SLD} in case Bucharest is the goal node.

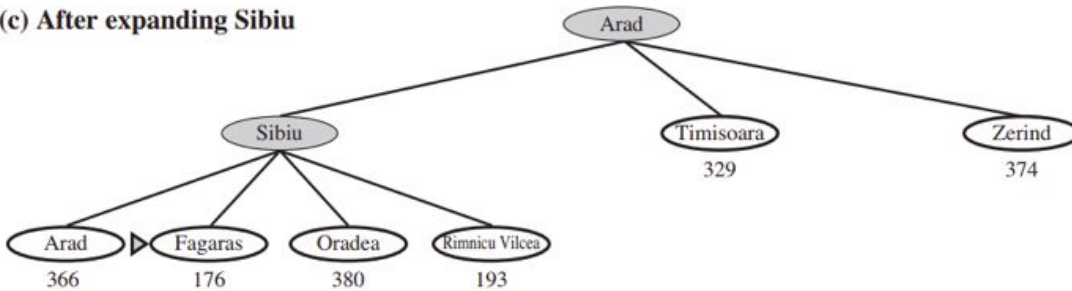
(a) The initial state



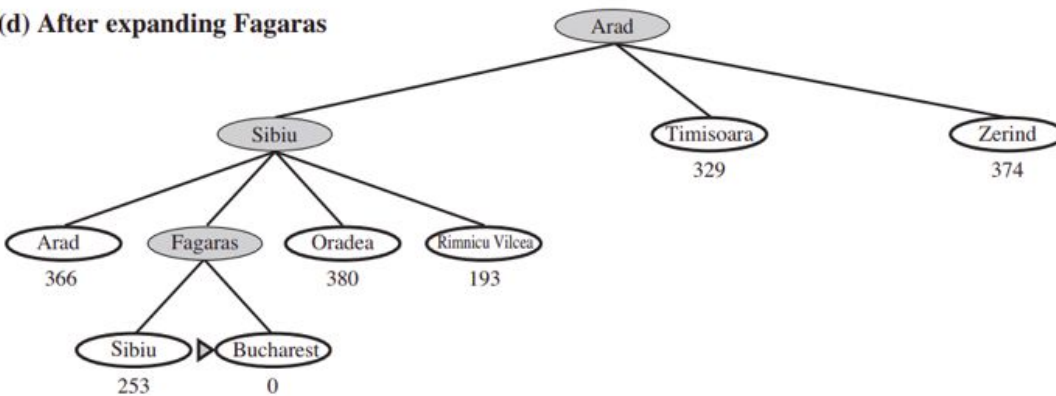
(b) After expanding Arad



(c) After expanding Sibiu



(d) After expanding Fagaras



- Greedy best-first search using h_{SLD} finds a solution without ever expanding a node that is not on the solution path; hence, its search cost is minimal. It is not optimal.
- Greedy algorithm: **at each step it tries to get as close to the goal as it can.**
- Greedy best-first tree search is incomplete even in a finite state space, much like depth-first search.
- Graph search version is complete in finite spaces, but not in infinite ones.
- Worst-case time and space complexity for the tree version is $O(b^m)$, where m is the maximum depth of the search space.

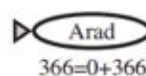
A* Search

- 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: $f(n) = g(n) + h(n)$.
- $g(n)$ gives the path cost from the start node to node n .
- $h(n)$ = the estimated cost of the cheapest path from n to the goal.
- $f(n)$ = estimated cost of the cheapest solution through n .
- Thus, if we are trying to find the cheapest solution, the node with the lowest value of $g(n) + h(n)$.
- If heuristic function $h(n)$ satisfies certain conditions, A* search is both complete and optimal. The algorithm is identical to UNIFORM-COST-SEARCH except that A* uses $g + h$ instead of g .

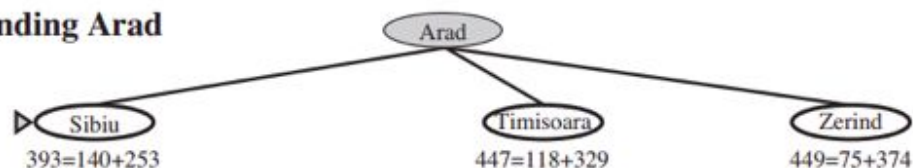
Conditions for optimality

- **First condition : $h(n)$ be an admissible heuristic.** An admissible heuristic is one that **never overestimates** the cost to reach the goal.
- **Second condition : consistency (or monotonicity)** is required only for applications of A* to graph search.
- A* has the following properties: **The tree-search version of A* is optimal if $h(n)$ is admissible, while the graph-search version is optimal if $h(n)$ is consistent.**

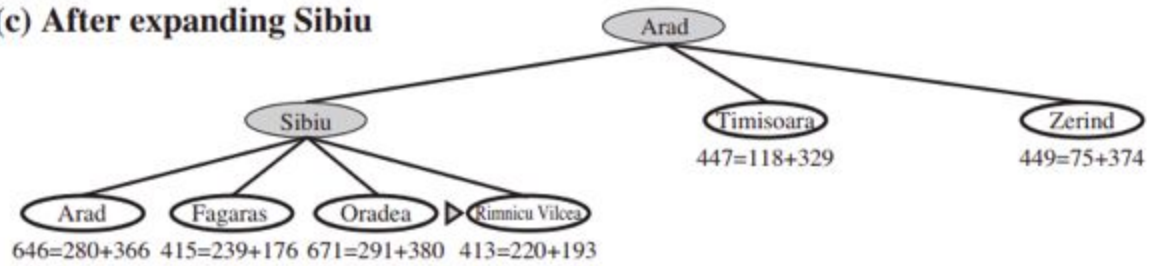
(a) The initial state



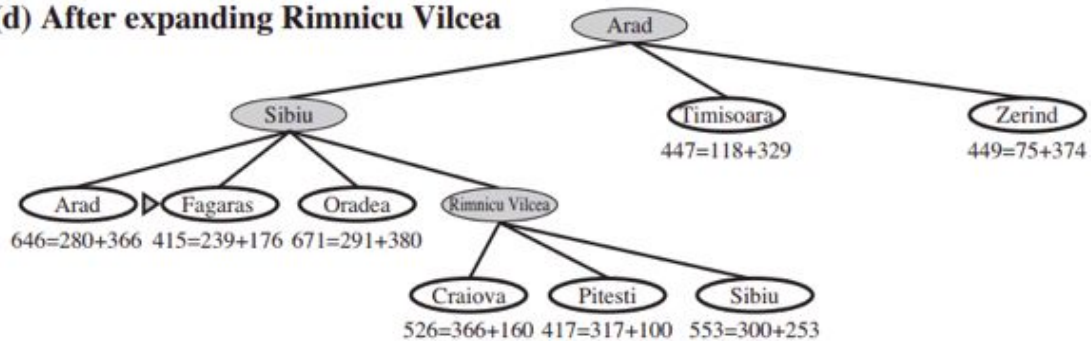
(b) After expanding Arad



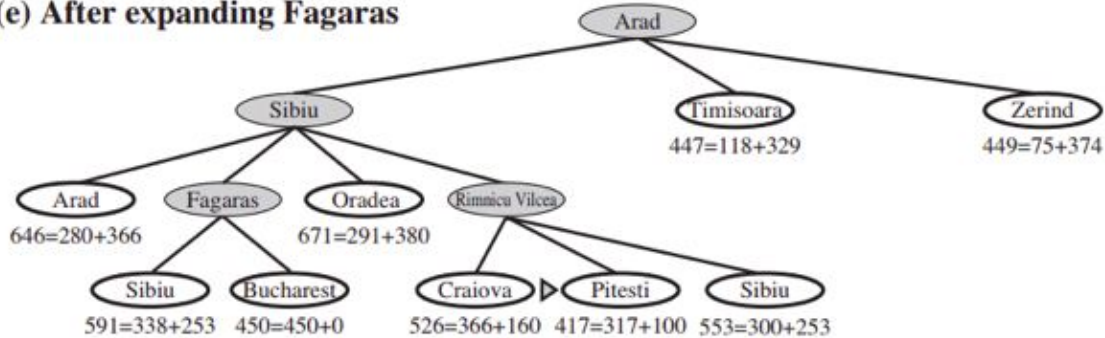
(c) After expanding Sibiu



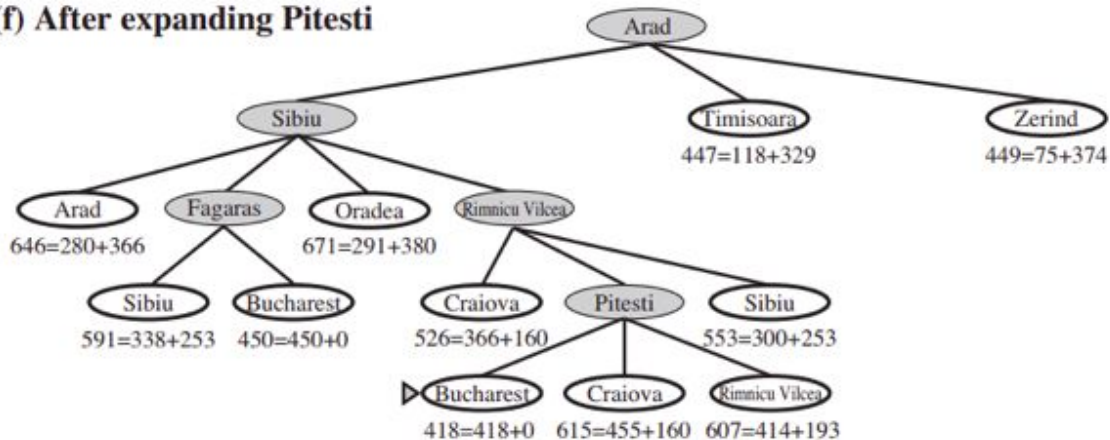
(d) After expanding Rimnicu Vilcea



(e) After expanding Fagaras



(f) After expanding Pitesti



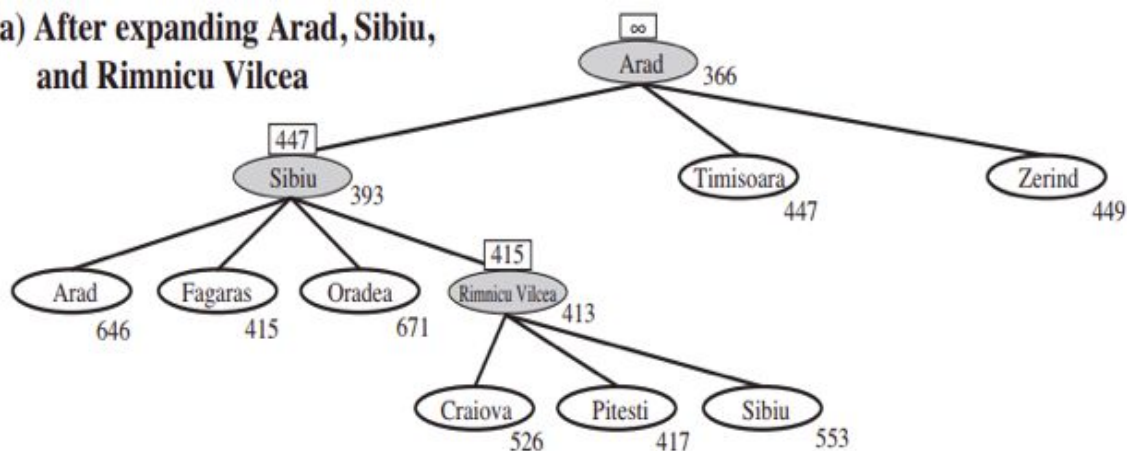
Memory-bounded Heuristic Search

- To reduce memory requirements for A* is to adapt the idea of iterative deepening to the heuristic search context, resulting in the **iterative-deepening A* (IDA*)**.
- The main difference between **IDA*** and **standard iterative deepening** is that the cutoff used is the **cost ($g + h$)** rather than the depth; at each iteration, the cutoff value is the **smallest f-cost of any node that exceeded the cutoff on the previous iteration**.

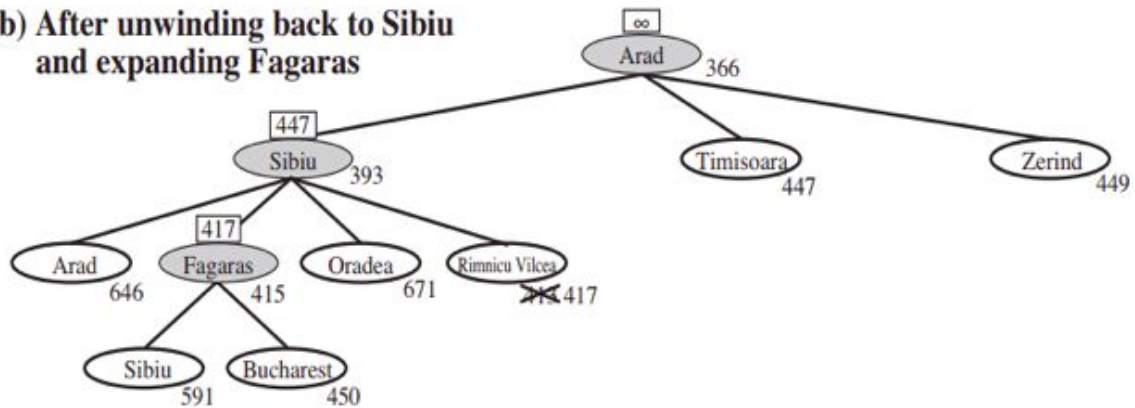
Recursive Best-First Search (RBFS)

- It is a simple recursive algorithm that attempts to mimic the operation of standard best-first search, but using only linear space.
- It uses the f limit variable to keep track of the f-value of the best alternative path available from any ancestor of the current node.
- If the current node exceeds this limit, the recursion unwinds back to the alternative path. As the recursion unwinds, RBFS replaces the f-value of each node along the path BACKED-UP VALUE with a **backed-up value**—the best f-value of its children.

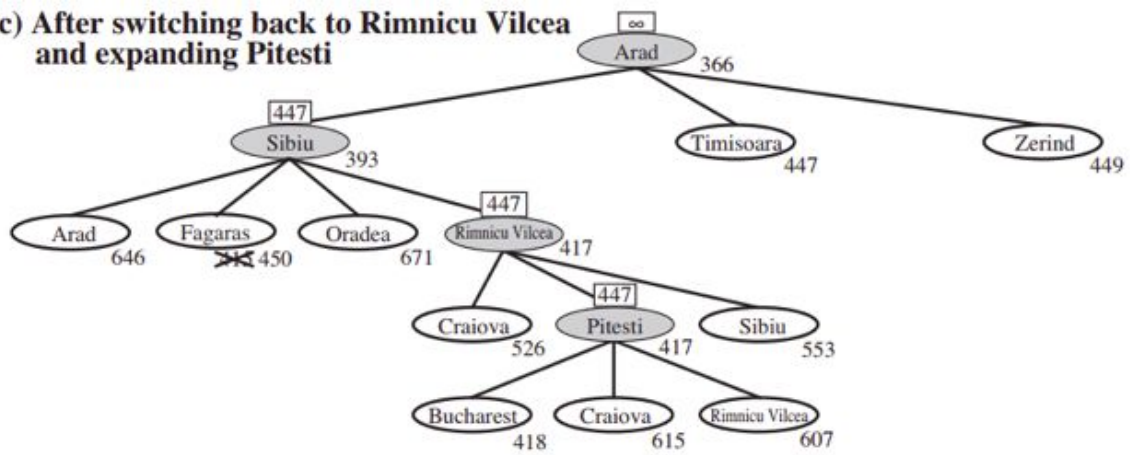
(a) After expanding Arad, Sibiu, and Rimnicu Vilcea



(b) After unwinding back to Sibiu and expanding Fagaras



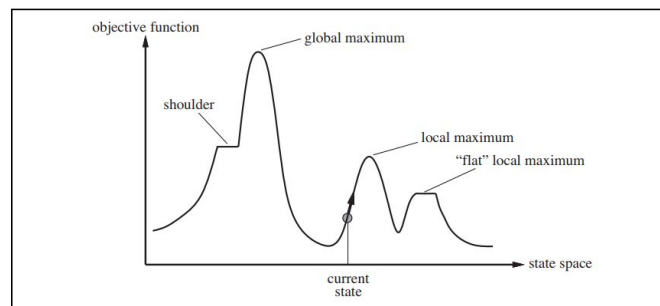
(c) After switching back to Rimnicu Vilcea and expanding Pitesti



Local Search Algorithm

- **Local search algorithms** operate using a single current node (rather than multiple paths) and generally move only to neighbors of that node.
- Although local search algorithms are not systematic, it has **two advantages**:
 - Use less memory — a constant amount;
 - Find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.
- **Local search algorithms** are useful for solving pure optimization problems, in which the aim is to find the best state according to an objective function.

State-Space Landscape

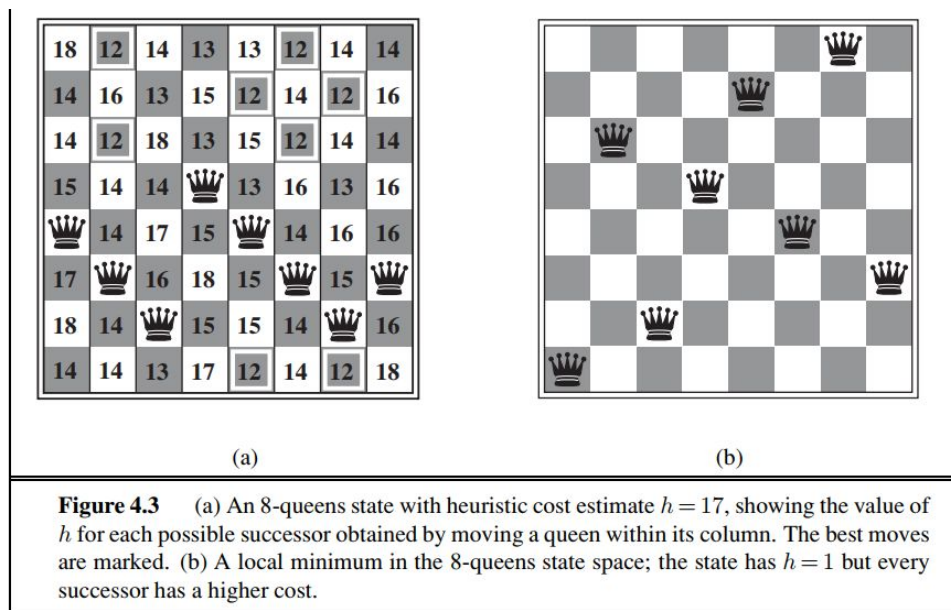


- A landscape has both “location = state” and “elevation = value of the heuristic cost function or objective function”.
- If elevation corresponds to cost \Rightarrow Find global minimum
- If elevation corresponds to an objective function \Rightarrow Find global maximum
- A complete local search algorithm always finds a goal if one exists; an optimal algorithm always finds a global minimum/maximum.

Hill-Climbing Search

- Is a loop that continually moves in the direction of increasing value \Rightarrow **uphill**.
- It terminates when it reaches a “**peak**” where no neighbor has a higher value.
- The algorithm does not maintain a search tree, so the data structure for the current node need only record the state and the value of the objective function.
- Hill climbing does not look ahead beyond the immediate neighbors of the current state.

- Hill-climbing search with Queen's 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 $8 \times 7 = 56$ 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.

Problems with Hill-Climbing Search

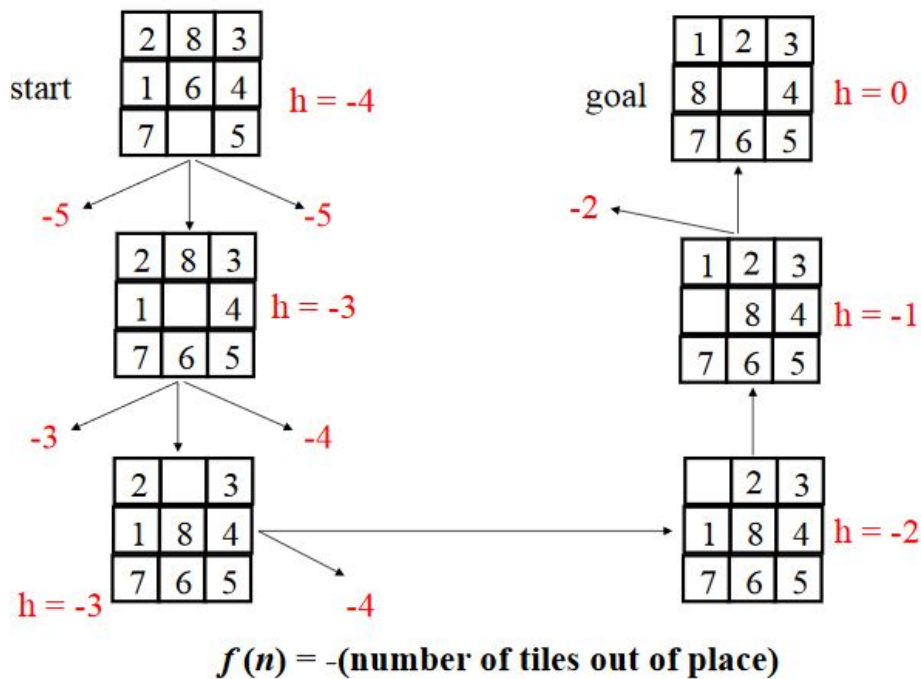
- Hill climbing is called **greedy local search** because it grabs a good neighbor state without thinking ahead about where to go next.
- Hill climbing often gets stuck for the following reasons:
 - Local maxima:** a local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum.
 - Ridges:** Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.

- **Plateau:** 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.

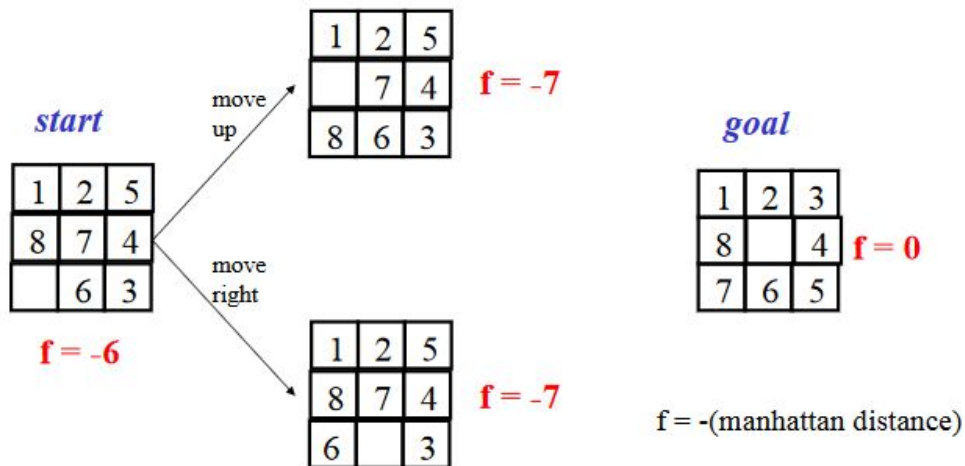
Solutions to Hill-Climbing Search

- Random restart:
 - keep restarting the search from
 - random locations until a goal is found.
- Problem reformulation:
 - reformulate the search
 - space to eliminate these problematic features

Hill climbing example




Example of a local optimum



Simulated Annealing

- The simulated annealing algorithm was originally inspired by the process of annealing in metal work.
- Annealing involves **heating** and **cooling** a material to alter its physical properties due to the changes in its internal structure.
- As the metal cools its new structure becomes fixed, consequently causing the metal to retain its newly obtained properties.
- We keep a temperature variable to simulate this heating process. We initially set it high and then allow it to slowly 'cool' as the algorithm runs.
- While this temperature variable is high the algorithm will be allowed, with more frequency, to accept solutions that are worse than our current solution.
- As the temperature is reduced so is the chance of accepting worse solutions, therefore allowing the algorithm to gradually focus in on an area of the search space in which a close to optimum solution can be found.
- This gradual 'cooling' process is what makes the simulated annealing algorithm remarkably effective at finding a close to optimum solution when dealing with large problems which contain numerous local optimums.

- 
- At each time step, the algorithm randomly selects a solution close to the current one, measures its quality, and then decides to move to it or to stay with the current solution based on probabilities between which it chooses on the basis of the fact that the new solution is better or worse than the current one.

Simulated annealing Acceptance Function

- First we check if the neighbour solution is better than our current solution. If it is, we accept it unconditionally.
 - If however, the neighbour solution isn't better we need to consider a couple of factors.
 - How much worse the neighbour solution is;
 - How high the current 'temperature' of our system is. At high temperatures the system is more likely to accept solutions that are worse.
- $$\exp((\text{solutionEnergy} - \text{neighbourEnergy}) / \text{temperature})$$
- The smaller the change in energy (the quality of the solution), and the higher the temperature, the more likely it is for the algorithm to accept the solution.

Algorithm for SA

- First we need to set the initial temperature and create a random initial solution.
- Then we begin looping until our stop condition is met. Usually either the system has sufficiently cooled, or a good-enough solution has been found.
- From here we select a neighbour by making a small change to our current solution.
- Then we decide whether to move to that neighbour solution.
- Finally, we decrease the temperature and continue looping

Temperature Initialization

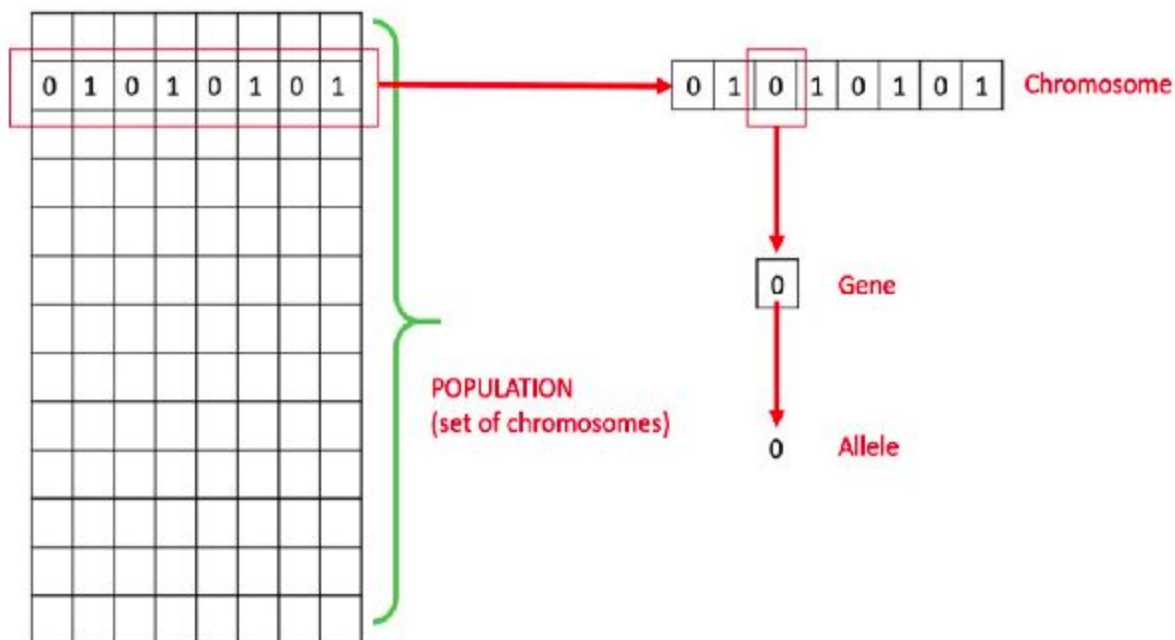
- For better optimization, when initializing the temperature variable we should select a temperature that will initially allow for practically any move against the current solution. This gives the algorithm the ability to better explore the entire search space before cooling and settling in a more focused region.

Genetic Algorithm

- A genetic algorithm is a variant of stochastic beam search in which successor states are generated by combining two parent states.
- GA begin with a set of k randomly generated states, called the population. Each state is represented as a string over a finite alphabet—a string of 0's and 1's.

Basic Terminology

- **Population** : is a subset of all the possible (encoded) solutions to the given program.
- **Chromosomes** : is one such solution to the given problem.
- **Gene** : is one element position of a chromosome.
- **Allele** : is the value a gene takes for a particular chromosome.



- **Genotype** : is the population in the computation space.
- **Phenotype** : is the population in the actual real world solution space.
- **Decoding** : is a process of transforming a solution from the genotype to the phenotype space.

- **Encoding** : is a process of transforming from phenotype to genotype space.
- **Fitness Function** : is a function which takes the solution as input and produces the suitability of the solution as the output.
- **Genetic Operators** : these alter the genetic composition of the offspring, including crossover, mutation, selection, etc.

Knapsack Problem

- You are going on a picnic and have a number of items that you can take along.
- Each item has a weight and a benefit or value.
- There is a capacity limit on the weight you can carry.
- You should carry items with max value.

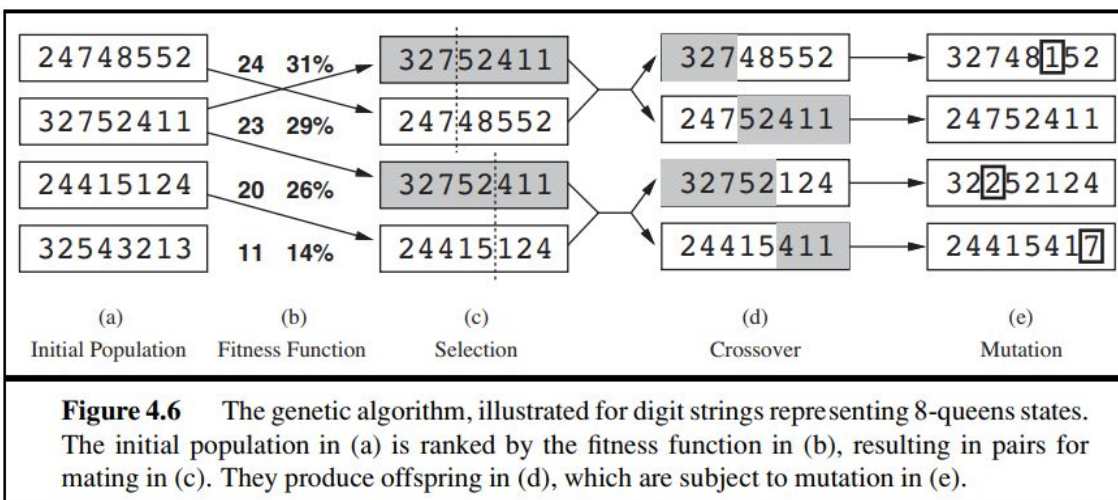
⇒ Item: 1 2 3 4 5 6 7

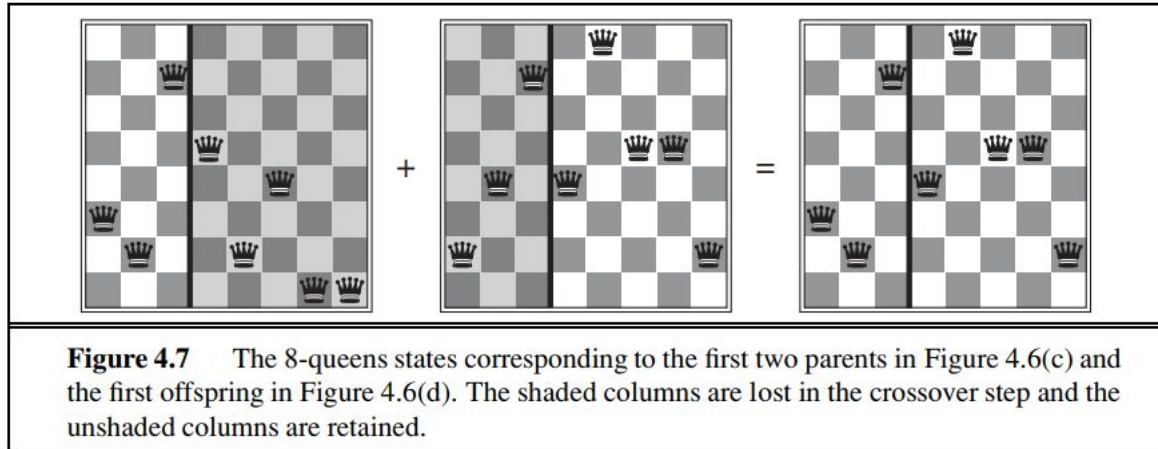
⇒ Benefit: 5 8 3 2 7 9 4

⇒ Weight: 7 8 4 10 4 6 4

⇒ Knapsack holds a maximum of 22 pounds

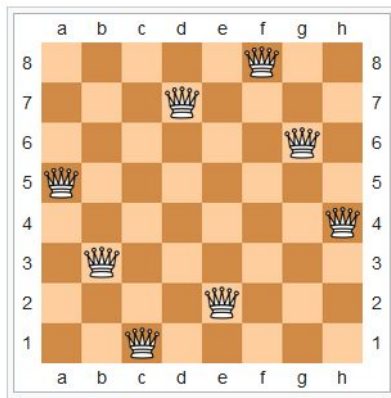
⇒ Fill it to get the maximum benefit





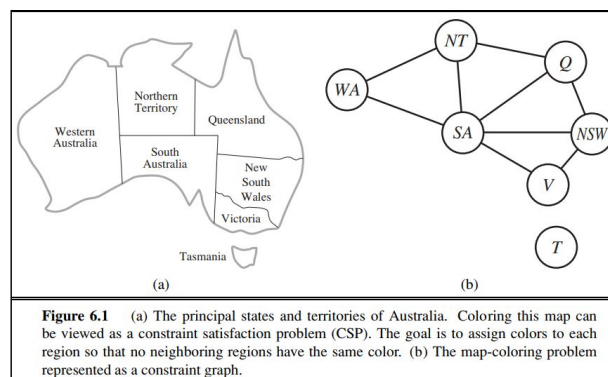
Constraint Satisfaction Problems

- **Constraint satisfaction problem** : A problem is considered to be as solved when each variable has a value that satisfies all the constraints on the variable.
- CSP search algorithms take advantage of the structure of states and use general-purpose to enable the solution of complex problems.
- The main idea is to eliminate large portions of the search space all at once by identifying variable/value combinations that violate the constraints.
- **8-Queens**



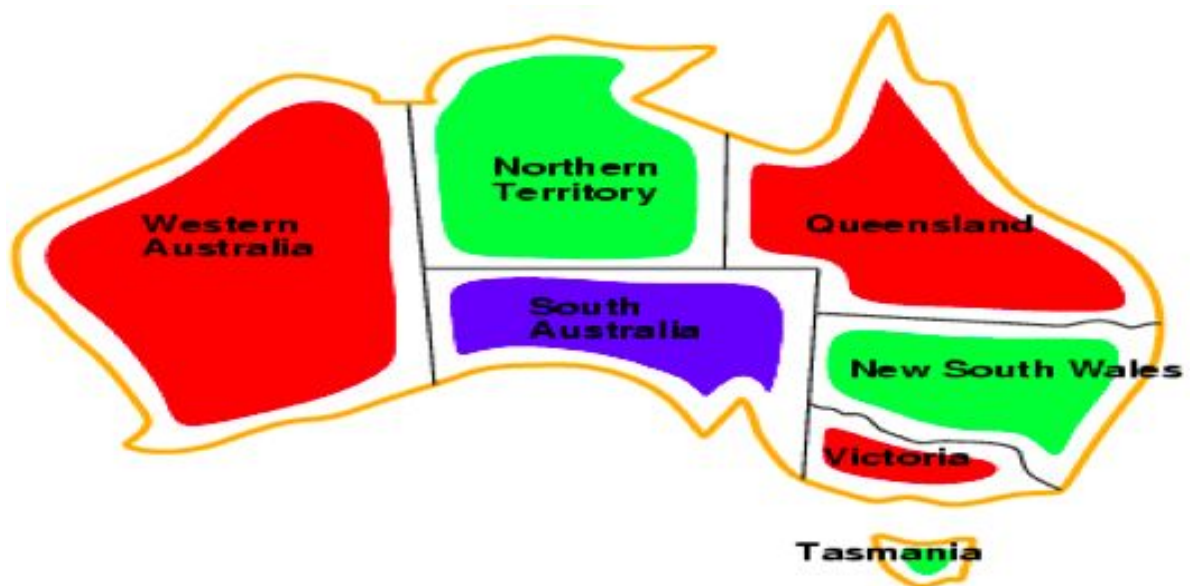
- A constraint satisfaction problem consists of three components, X , D , and C :
 - X is a set of variables, $\{X_1, \dots, X_n\}$.
 - D is a set of domains, $\{D_1, \dots, D_n\}$, one for each variable.
 - C is a set of constraints that specify allowable combinations of values.

Map Coloring



- Task : coloring each region either red, green, or blue in such a way that no neighboring regions have the same color.
- To formulate this as a CSP, we define the variables to be the regions

$$X = \{WA, NT, Q, NSW, V, SA, T\}$$
- The domain of each variable is the set $D_i = \{\text{red}, \text{green}, \text{blue}\}$.
- The constraints require neighboring regions to have distinct colors. Since there are nine places where regions border, there are nine constraints: $C = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V, WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}$.



- Solutions are **complete** and **consistent** assignments
- WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green.
- To solve a CSP, we need to define a state space and the notion of a solution. Each state in a CSP is defined by an assignment of values to some or all of the variables, $\{X_i = v_i, X_j = v_j, \dots\}$.
- A **complete assignment** is one in which every variable is assigned, and a solution to a CSP is a consistent, complete assignment.
- A **partial assignment** is one that assigns values to only some of the variables.

Job-shop Scheduling

- The whole job is composed of tasks, and we can model each task as a variable, where the value of each variable is the time that the task starts, expressed as an integer number of minutes.
- Constraints can assert that one task must occur before another—for example, a wheel must be installed before the hubcap is put on.
- Constraints can also specify that a task takes a certain amount of time to complete. We consider a small part of the car assembly, consisting of 15 tasks: install axles (front and back), affix all four wheels (right and left, front and back), tighten nuts for each wheel, affix hubcaps, and inspect the final assembly.
- $X = \{\text{AxleF}, \text{AxleB}, \text{Wheel RF}, \text{WheelLF}, \text{WheelRB}, \text{WheelLB}, \text{NutsRF}, \text{NutsLF}, \text{NutsRB}, \text{NutsLB}, \text{CapRF}, \text{CapLF}, \text{CapRB}, \text{CapLB}, \text{Inspect}\}$
- Whenever a task $T1$ must occur before task $T2$, and task $T1$ takes duration $d1$ to complete, we add an arithmetic constraint of the form $T1 + d1 \leq T2$.
- The axles have to be in place before the wheels are put on, and it takes 10 minutes to install an axle, so we write
- $\text{AxleF} + 10 \leq \text{WheelRF}; \text{AxleF} + 10 \leq \text{WheelLF};$
- $\text{AxleB} + 10 \leq \text{WheelRB}; \text{AxleB} + 10 \leq \text{WheelLB}$
- For every variable except Inspect we add a constraint of the form $X + dX \leq \text{Inspect}$. Finally, suppose there is a requirement to get the whole assembly done in 30 minutes. We can achieve that by limiting the domain of all variables: $D_i = \{1, 2, 3, \dots, 27\}$

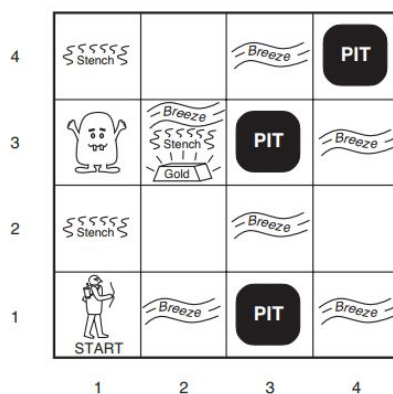
Logical Agents

Knowledge-based Agents

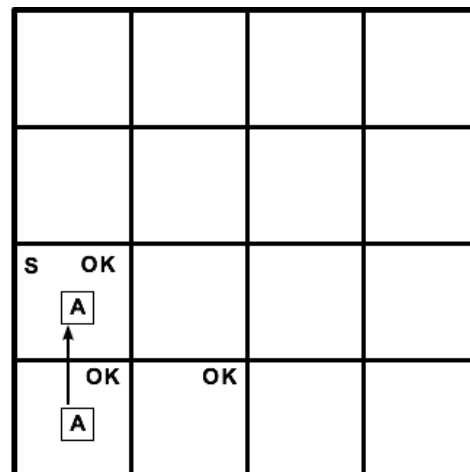
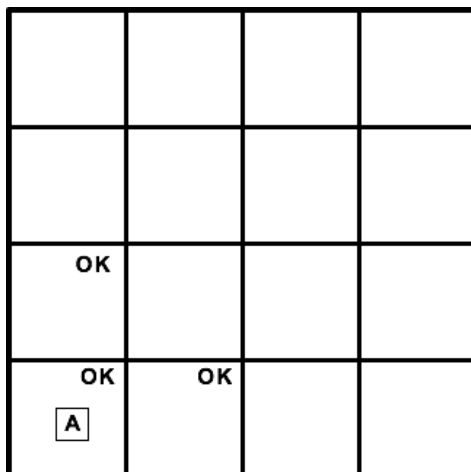
$$KB \vdash_i \alpha$$

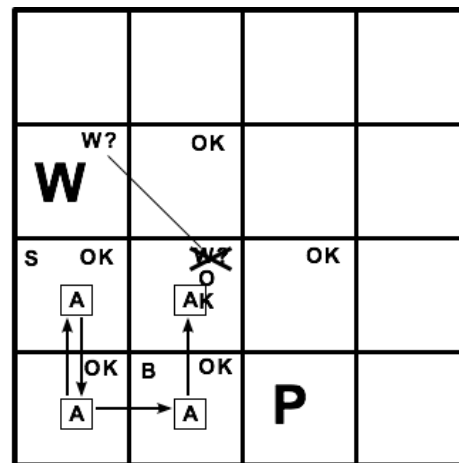
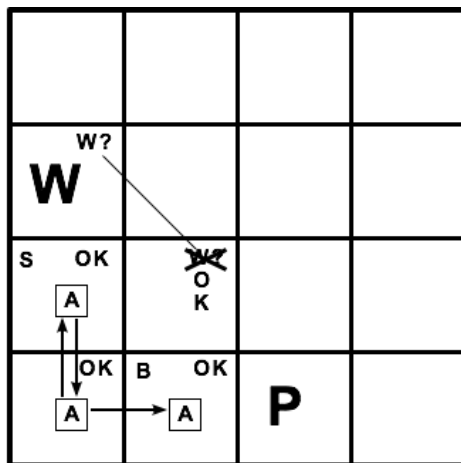
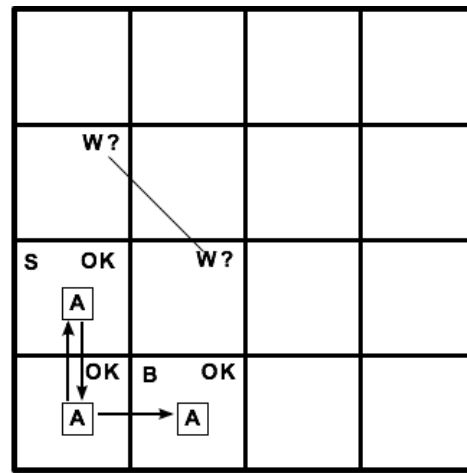
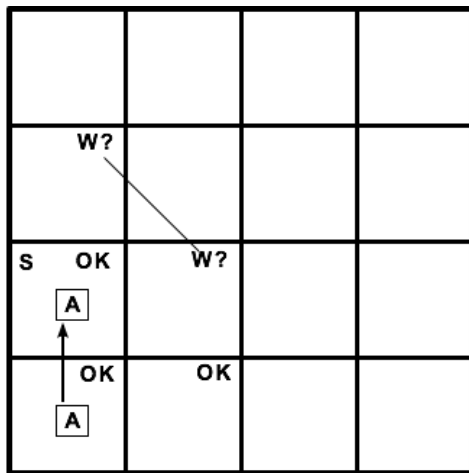
- **Knowledge base:** is a set of sentences. Each sentence is expressed in a language called **Knowledge Representation language** and represents some assertion about the world.
- When the sentence is taken as given without being derived from other sentences is called as **axioms**.
- There must be a way to add new sentences to the knowledge base and a way to query what is known. The standard names for these operations are **tell** and **ask**, respectively. Both operations may involve inference - that is, deriving new sentences from old.
- Each time the agent program is called, it does 3 things:
 1. It **tells** the knowledge base what it perceives.
 2. It **asks** the knowledge base what action it should perform. In the process of answering this query, extensive reasoning may be done about the current state of the world, about the outcomes of possible action sequences.
 3. The agent program **tells** the knowledge base which action was chosen, and the agent executes the action.
- The agent must be able to:
 - Represent states and actions
 - Incorporate new percepts
 - Update internal representations of the world
 - Deduce hidden properties of the world
 - Deduce appropriate actions

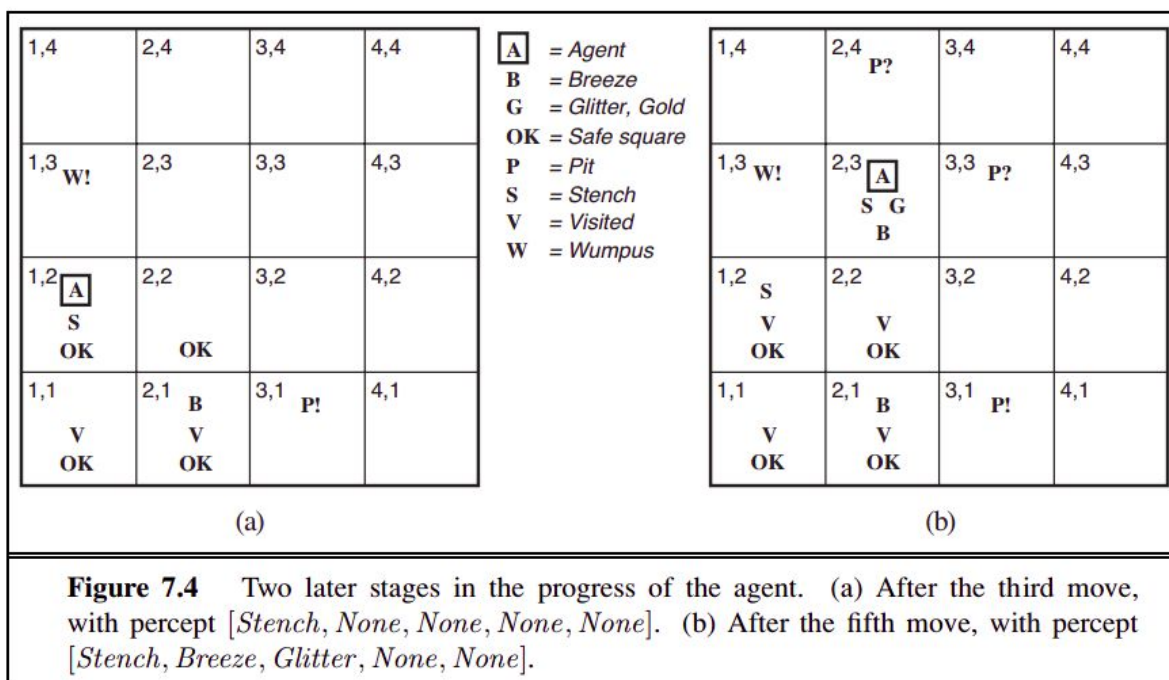
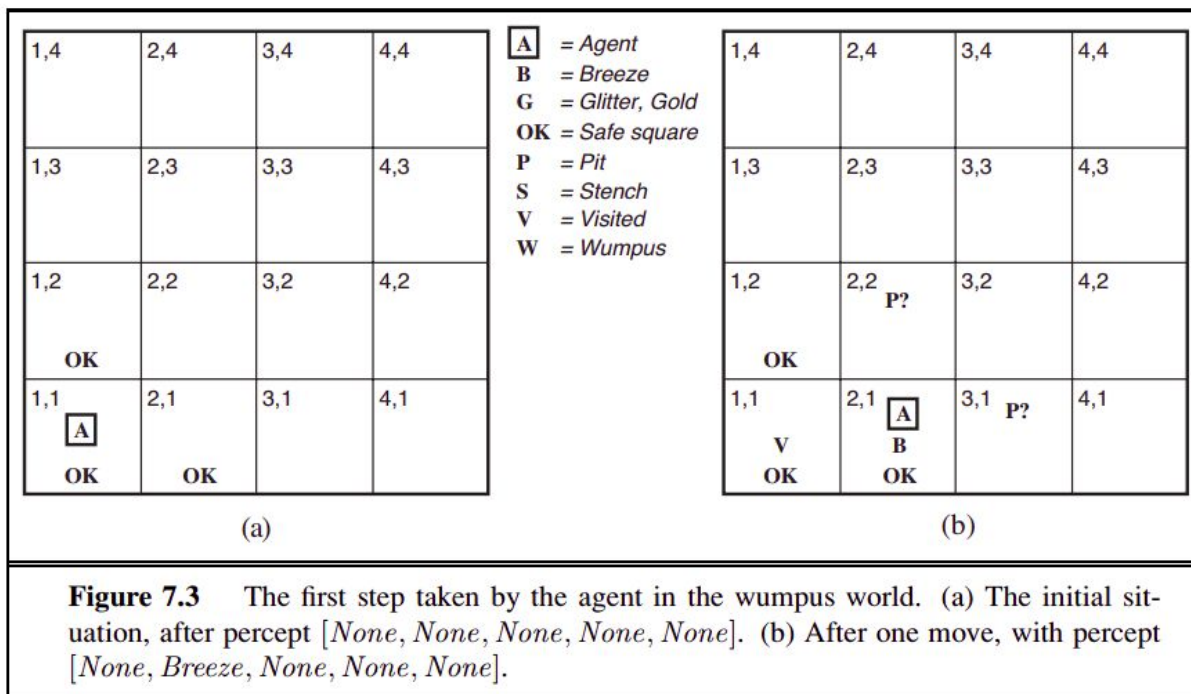
Wumpus World



- Performance Measure: Gold + 1000, Death - 1000, Step - 1, Use arrow - 10
- Environment:
 - Square adjacent to the Wumpus are smelly
 - Squares adjacent to the pit are breezy
 - Glitter if gold is in the same square
 - Shooting kills Wumpus if you are facing it
 - Shooting uses up the only arrow
 - Grabbing picks up the gold if in the same square
 - Releasing drops the gold in the same square
- Actuators: left turn, right turn, forward, grab, release, shoot
- Sensors: Breeze, glitter, smell
- Characterization of Wumpus World:
 - Observable : partial, only local perception
 - Deterministic : Yes, outcomes are specified
 - Episodic : No, sequential at the level of actions
 - Static : Yes, Wumpus and pits do not move
 - Discrete : Yes
 - Single Agent: Yes







Logic

- **Logics** : is information such that conclusions can be drawn
- Syntax defines the sentences in the language
- Semantics define the “meaning” of sentences; i.e., define truth of a sentence in a world
- Ex: the language of arithmetic
 - $x + 2 \geq y$ is a sentence; $x2 + y >$ is not a sentence
 - $x + 2 \geq y$ is true in a world where $x = 7$; $y = 1$
 - $x + 2 \geq y$ is false in a world where $x = 0$; $y = 6$

Propositional Logic

- The syntax of propositional logic : is the allowable sentences. The atomic sentences consist of a single proposition symbol. Each such symbol stands for a proposition that can be true or false.
- There are five connectors in common use:
 - **\neg (NOT) - negation**: if $S1$ is a sentence, then $\neg S1$ is a sentence
 - **\wedge (AND) - Conjunction** : if $S1, S2$ are sentences, then $S1 \wedge S2$ is a sentence
 - **\vee (OR) - Disjunction** : if $S1, S2$ are sentences, then $S1 \vee S2$ is a sentence
 - **\Rightarrow (IMPLIES) - Implication** : if $S1, S2$ are sentences, then $S1 \Rightarrow S2$ is a sentence
 - **\Leftrightarrow (if and only if) - Biconditional** : if $S1, S2$ are sentences, then $S1 \Leftrightarrow S2$ is a sentence

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	false	true	false	false	true	true
false	true	true	false	true	true	false
true	false	false	false	true	false	false
true	true	false	true	true	true	true

A Simple Knowledge Base: Wumpus World

- $P_{x,y}$ is true if there is a pit in $[x, y]$.
- $W_{x,y}$ is true if there is a wumpus in $[x, y]$, dead or alive.

- $B_{x,y}$ is true if the agent perceives a breeze in $[x, y]$.
- $S_{x,y}$ is true if the agent perceives a stench in $[x, y]$
- The sentences we write will suffice to derive $\neg P_{1,2}$ (there is no pit in $[1,2]$)
- Knowledge base Wumpus World:
 - We label each sentence R_i so that we can refer to them:
 - There is no pit in $[1,1]$: $R_1 : \neg P_{1,1}$.
 - A square is breezy if and only if there is a pit in a neighboring square. This has to be stated for each square; for now, we include just the relevant squares:

$$R_2 : B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1}) .$$

$$R_3 : B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1}) .$$
 - The preceding sentences are true in all wumpus worlds. Now we include the breeze precepts for the first two squares visited in the specific world the agent is in.

$$R_4 : \neg B_{1,1}$$

$$R_5 : B_{2,1}$$

Equivalence, Validity, Satisfiability

$(\alpha \wedge \beta) \equiv (\beta \wedge \alpha)$	commutativity of \wedge
$(\alpha \vee \beta) \equiv (\beta \vee \alpha)$	commutativity of \vee
$((\alpha \wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma))$	associativity of \wedge
$((\alpha \vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma))$	associativity of \vee
$\neg(\neg\alpha) \equiv \alpha$	double-negation elimination
$(\alpha \Rightarrow \beta) \equiv (\neg\beta \Rightarrow \neg\alpha)$	contraposition
$(\alpha \Rightarrow \beta) \equiv (\neg\alpha \vee \beta)$	implication elimination
$(\alpha \Leftrightarrow \beta) \equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha))$	biconditional elimination
$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta)$	De Morgan
$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta)$	De Morgan
$(\alpha \wedge (\beta \vee \gamma)) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$	distributivity of \wedge over \vee
$(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$	distributivity of \vee over \wedge

Figure 7.11 Standard logical equivalences. The symbols α , β , and γ stand for arbitrary sentences of propositional logic.

- A sentence is valid if it is true in all models: e.g. True, $A \vee \neg A$, $A \Rightarrow A$, $(A \wedge (A \Rightarrow B)) \Rightarrow B$

- Validity is connected to inference via the Deduction Theorem: $KB \models a$ iff $(KB \Rightarrow a)$ is valid
- A sentence is satisfiable if it is True in some model: e.g. $A \vee B, C$
- A sentence is unsatisfiable if it is True in no models: e.g. $A \wedge \neg A$
- Satisfiability is connected to inference via the following
 - $KB \models a$ iff $(KB \wedge \neg a)$ is unsatisfiable
 - proof by contradiction

Inference Rules

- **Inference Rule** : Patterns of inference that can be applied to derive chains of conclusions that lead to the desired goal.
- **Modus Ponens**
 - Given: $S1 \Rightarrow S2$ and $S1$, derive $S2$
 - For example, if $(WumpusAhead \wedge WumpusAlive) \Rightarrow Shoot$ and $(WumpusAhead \wedge WumpusAlive)$ are given, then $Shoot$ can be inferred.
- **And-Elimination**
 - Given: $S1 \vee S2$, derive $S1$
 - Given: $S1 \vee S2$, derive $S2$
 - $(WumpusAhead \wedge WumpusAlive)$, $WumpusAlive$ can be inferred.
- **DeMorgan's Law**
 - Given: $\neg(A \vee B)$ derive $\neg A \wedge \neg B$
 - Given: $\neg(A \wedge B)$ derive $\neg A \vee \neg B$

Use in Wumpus World

- We start with the knowledge base containing $R1$ through $R5$ and show how to prove $\neg P1,2$, that is, there is no pit in $[1,2]$.

$R1 : \neg P1,1$

$R2 : B1,1 \Leftrightarrow (P1,2 \vee P2,1)$

$R3 : B2,1 \Leftrightarrow (P1,1 \vee P2,2 \vee P3,1)$

$R4 : \neg B1,1$

$R5 : B2,1$

- First, we apply bi-conditional elimination to $R2$ to obtain

$$R6 : (B1,1 \Rightarrow (P1,2 \vee P2,1)) \wedge ((P1,2 \vee P2,1) \Rightarrow B1,1)$$

- Then we apply And-Elimination to R6 to obtain

$$R7 : ((P1,2 \vee P2,1) \Rightarrow B1,1)$$

- Logical equivalence for contrapositives gives

$$R8 : (\neg B1,1 \Rightarrow \neg(P1,2 \vee P2,1))$$

- Now we can apply Modus Ponens with R8 and the percept R4 (i.e., $\neg B1,1$), to obtain

$$R9 : \neg(P1,2 \vee P2,1) .$$

- Finally, we apply De Morgan's rule, giving the conclusion

$$R10 : \neg P1,2 \wedge \neg P2,1 .$$

- That is, neither [1,2] nor [2,1] contains a pit.

Define a Proof Problem

- **INITIAL STATE:** the initial knowledge base.
- **ACTIONS:** the set of actions consists of all the inference rules applied to all the sentences that match the top half of the inference rule.
- **RESULT:** the result of an action is to add the sentence in the bottom of the inference rule.
- **GOAL:** the goal is a state that contains the sentence we are trying to prove.
- Finding a proof can be more efficient because the proof can ignore irrelevant propositions, no matter how many of them there are.

Proof by Resolution

- A single inference rule, resolution, that yields a complete inference algorithm when coupled with any complete search algorithm. a simple version of the resolution rule in the wumpus world.
- We add the following facts to the knowledge base:

$$R11 : \neg B1,2$$

$$R12 : B1,2 \Leftrightarrow (P1,1 \vee P2,2 \vee P1,3)$$

- We can now derive the absence of pits in [2,2] and [1,3] (remember that [1,1] is already known to be pit-less):

$$R13 : \neg P2,2$$

$$R14 : \neg P1,3$$

- We can also apply biconditional elimination to R3, followed by Modus Ponens with R5, to obtain the fact that there is a pit in [1,1], [2,2], or [3,1]:

$$R3 : B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$$

$$R5 : B_{2,1}$$

$$R15 : P_{1,1} \vee P_{2,2} \vee P_{3,1}$$

- Now comes the first application of the resolution rule: the literal $\neg P_{2,2}$ in R13 resolves with the literal $P_{2,2}$ in R15 to give the resolvent

$$R16 : P_{1,1} \vee P_{3,1}$$

- In English; if there's a pit in one of [1,1], [2,2], and [3,1] and it's not in [2,2], then it's in [1,1] or [3,1]
- Similarly, the literal $\neg P_{1,1}$ in R1 resolves with the literal $P_{1,1}$ in R16 to give

$$R17 : P_{3,1}$$

- In English: if there's a pit in [1,1] or [3,1] and it's not in [1,1], then it's in [3,1].
- These last two inference steps are examples of the unit resolution inference rule,

$$\frac{\ell_1 \vee \dots \vee \ell_k, \quad m}{\ell_1 \vee \dots \vee \ell_{i-1} \vee \ell_{i+1} \vee \dots \vee \ell_k},$$

- Thus, the unit resolution rule takes a clause—a disjunction of literals—and a literal and produces a new clause.

$$\frac{\ell_1 \vee \dots \vee \ell_k, \quad m_1 \vee \dots \vee m_n}{\ell_1 \vee \dots \vee \ell_{i-1} \vee \ell_{i+1} \vee \dots \vee \ell_k \vee m_1 \vee \dots \vee m_{j-1} \vee m_{j+1} \vee \dots \vee m_n},$$

$$\frac{P_{1,1} \vee P_{3,1}, \quad \neg P_{1,1} \vee \neg P_{2,2}}{P_{3,1} \vee \neg P_{2,2}}$$

Full Resolution Rule

- The removal of multiple copies of literals is called factoring.
- For example, if we resolve $(A \vee B)$ with $(A \vee \neg B)$, we obtain $(A \vee A)$, which is reduced to just A .
- A resolution-based theorem prover can, for any sentences α and β in propositional logic, decide whether $\alpha \models \beta$

Conjunctive Normal Form

- The resolution rule applies only to clauses so it would seem to be relevant only to knowledge bases and queries consisting of clauses.
- Every sentence of propositional logic is logically equivalent to a conjunctive normal form

Converting a Sentence into CNF

- Converting the sentence $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$ into CNF
 1. Eliminate \Leftrightarrow , replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$
$$(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$$
 2. Eliminate \Rightarrow , replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$:
$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1})$$
 3. CNF requires \neg to appear only in literals, so we “move \neg inwards”
$$\neg(\neg\alpha) \equiv \alpha \text{ (double-negation elimination)}$$
$$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta) \text{ (De Morgan)}$$
$$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta) \text{ (De Morgan)}$$
$$\Rightarrow \text{In the example, we require just one application of the last rule:}$$
$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1})$$
 4. We apply the distributive law, distributing \vee over \wedge wherever possible.
$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1}).$$

Example Proof By Deduction

- Knowledge

S1: $B_{22} \Leftrightarrow (P_{21} \vee P_{23} \vee P_{12} \vee P_{32})$

rule

S2: $\neg B_{22}$

observation

- Inferences

S3: $(B_{22} \Rightarrow (P_{21} \vee P_{23} \vee P_{12} \vee P_{32})) \wedge ((P_{21} \vee P_{23} \vee P_{12} \vee P_{32}) \Rightarrow B_{22})$	[S1, <u>bi elim</u>]
S4: $((P_{21} \vee P_{23} \vee P_{12} \vee P_{32}) \Rightarrow B_{22})$	[S3, <u>and elim</u>]
S5: $(\neg B_{22} \Rightarrow \neg(P_{21} \vee P_{23} \vee P_{12} \vee P_{32}))$	[<u>contrapos</u>]
S6: $\neg(P_{21} \vee P_{23} \vee P_{12} \vee P_{32})$	[S2, S6, MP]
S7: $\neg P_{21} \wedge \neg P_{23} \wedge \neg P_{12} \wedge \neg P_{32}$	[S6, <u>DeMorg</u>]

Evaluation of Deductive Inference

- **Sound:** Yes, because the inference rules themselves are sound. (This can be proven using a truth table argument).
- **Complete**
 - If we allow all possible inference rules, we're searching in an infinite space, hence not complete
 - If we limit inference rules, we run the risk of leaving out the necessary one...
- **Monotonic:** If we have a proof, adding information to the DB will not invalidate the proof

Resolution

- Resolution allows a complete inference mechanism (search-based) using only one rule of inference
- Resolution rule:
 - Given: $P_1 \vee P_2 \vee P_3 \dots \vee P_n$ and $\neg P_1 \vee Q_1 \dots \vee Q_m$
 - Conclude: $P_2 \vee P_3 \dots \vee P_n \vee Q_1 \dots \vee Q_m$
 - Complementary literals P_1 and $\neg P_1$ "cancel out"
- Why it works:
 - Consider 2 cases: P_1 is true, and P_1 is false

Proof Using Resolution

- To prove a fact P, repeatedly apply resolution until either:
 - No new clauses can be added (KB does not entail P)
 - The empty clause is derived (KB does entail P)
- This is proof by contradiction: if we prove that $KB \wedge \neg P$ derives a contradiction (empty clause) and we know KB is true, then $\neg P$ must be false, so P must be true!

- To apply resolution mechanically, facts need to be in **Conjunctive Normal Form** (CNF)
- To carry out the proof, need a search mechanism that will enumerate all possible resolutions.

1. $B_{22} \Leftrightarrow (P_{21} \vee P_{23} \vee P_{12} \vee P_{32})$
2. Eliminate \Leftrightarrow , replacing with two implications
 $(B_{22} \Rightarrow (P_{21} \vee P_{23} \vee P_{12} \vee P_{32})) \wedge ((P_{21} \vee P_{23} \vee P_{12} \vee P_{32}) \Rightarrow B_{22})$
3. Replace implication $(A \Rightarrow B)$ by $\neg A \vee B$
 $(\neg B_{22} \vee (P_{21} \vee P_{23} \vee P_{12} \vee P_{32})) \wedge (\neg(P_{21} \vee P_{23} \vee P_{12} \vee P_{32}) \vee B_{22})$
4. Move \neg "inwards" (unnecessary patens removed)
 $(\neg B_{22} \vee P_{21} \vee P_{23} \vee P_{12} \vee P_{32}) \wedge ((\neg P_{21} \wedge \neg P_{23} \wedge \neg P_{12} \wedge \neg P_{32}) \vee B_{22})$
5. Distributive Law
 $(\neg B_{22} \vee P_{21} \vee P_{23} \vee P_{12} \vee P_{32}) \wedge (\neg P_{21} \vee B_{22}) \wedge (\neg P_{23} \vee B_{22}) \wedge (\neg P_{12} \vee B_{22}) \wedge (\neg P_{32} \vee B_{22})$

(Final result has 5 clauses)

- Given B_{22} and $\neg P_{21}$ and $\neg P_{23}$ and $\neg P_{32}$, prove P_{12}
 $(\neg B_{22} \vee P_{21} \vee P_{23} \vee P_{12} \vee P_{32}) ; \neg P_{12}$
 $(\neg B_{22} \vee P_{21} \vee P_{23} \vee P_{32}) ; \neg P_{21}$
 $(\neg B_{22} \vee P_{23} \vee P_{32}) ; \neg P_{23}$
 $(\neg B_{22} \vee P_{32}) ; \neg P_{32}$
 $(\neg B_{22}) ; B_{22}$
 [empty clause]

Evaluation of Resolution

- Resolution is sound because the resolution rule is true in all cases
- Resolution is complete

- Provided a complete search method is used to find the proof, if proof can be found it will
- Note: you must know what you're trying to prove in order to prove it!
- Resolution is exponential: The number of clauses that we must search grows exponentially...

Horn Clauses

- A Horn Clause is a CNF clause with exactly one positive literal
 - The positive literal is called the head
 - The negative literals are called the body
 - Prolog: head:- body1, body2, body3 ...
 - English: "To prove the head, prove body1, ..."
 - Implication: If (body1, body2 ...) then head
- Horn Clauses form the basis of forward and backward chaining
- The Prolog language is based on Horn Clauses
- Deciding entailment with Horn Clauses is linear in the size of the knowledge base

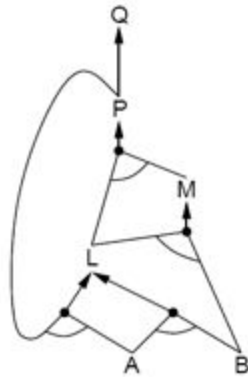
Reasoning with Horn Clauses

- Forward Chaining
 - For each new piece of data, generate all new facts, until the desired fact is generated
 - Data-directed reasoning
- Backward Chaining
 - To prove the goal, find a clause that contains the goal as its head, and prove the body recursively
 - Backtrack when you chose the wrong clause
 - Goal-directed reasoning

Forward Chaining

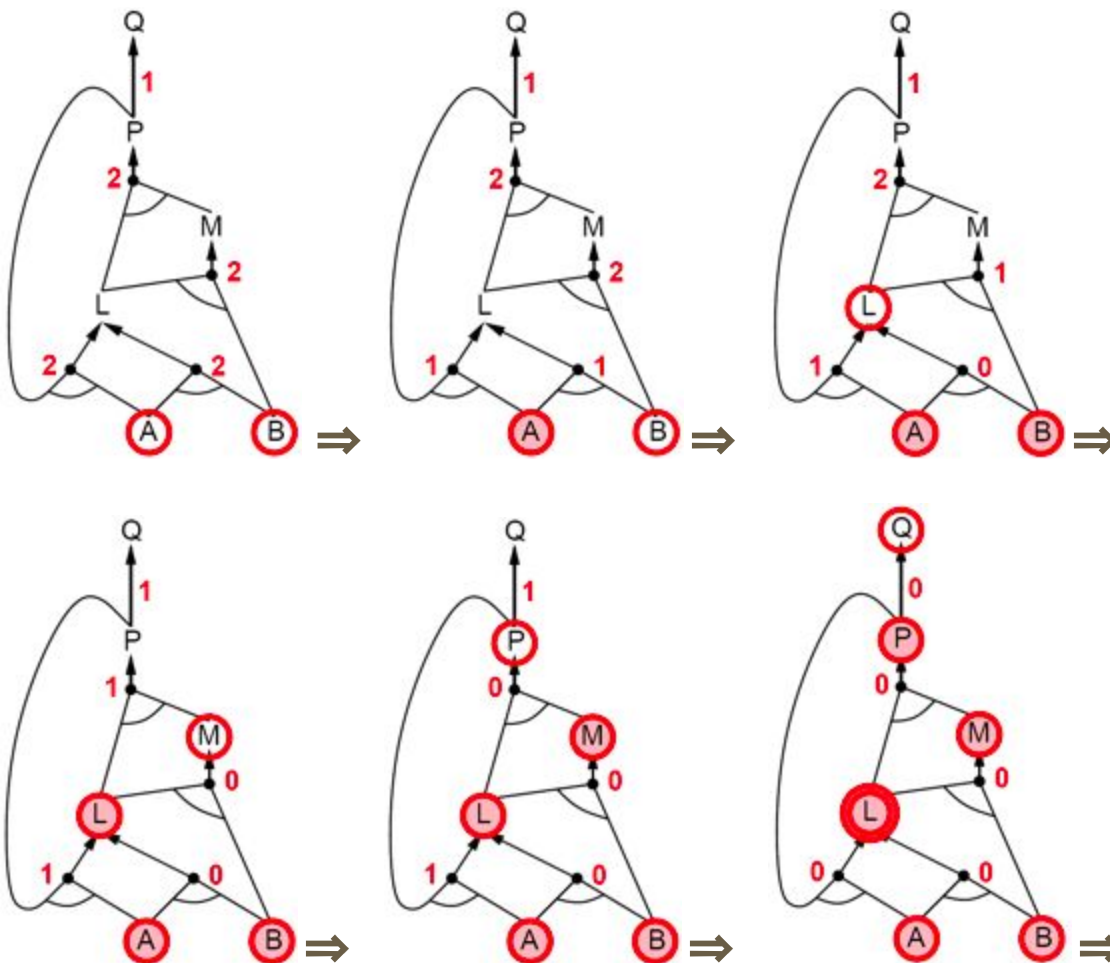
- Fire any rules whose premises are satisfied in the KB
- Add its conclusion to the KB until the query is found

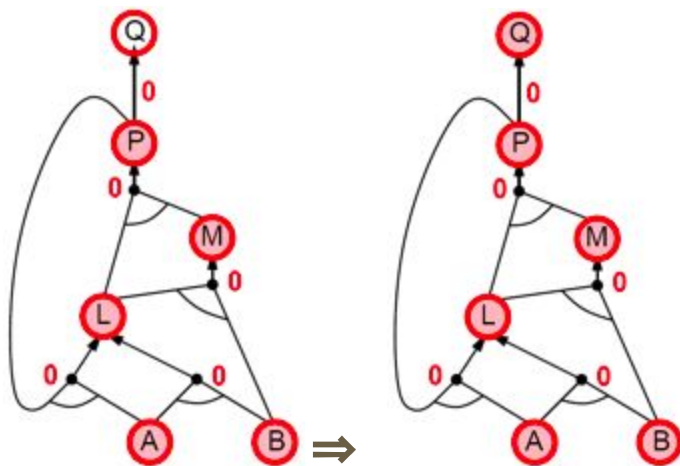
$P \Rightarrow Q$
 $L \wedge M \Rightarrow P$
 $B \wedge L \Rightarrow M$
 $A \wedge P \Rightarrow L$
 $A \wedge B \Rightarrow L$
 A
 B



- AND-OR Graph

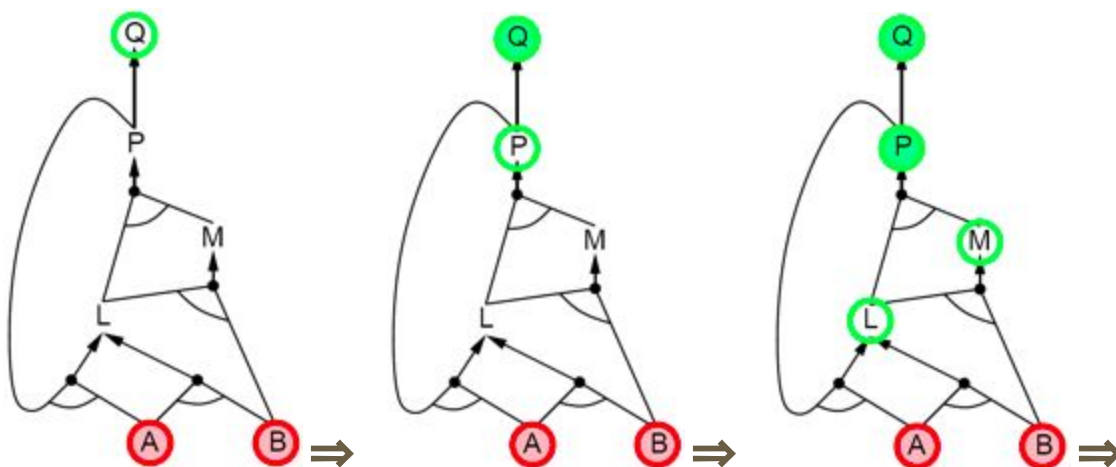
- multiple links joined by an arc indicate conjunction – every link must be proved
- multiple links without an arc indicate disjunction – any link can be proved

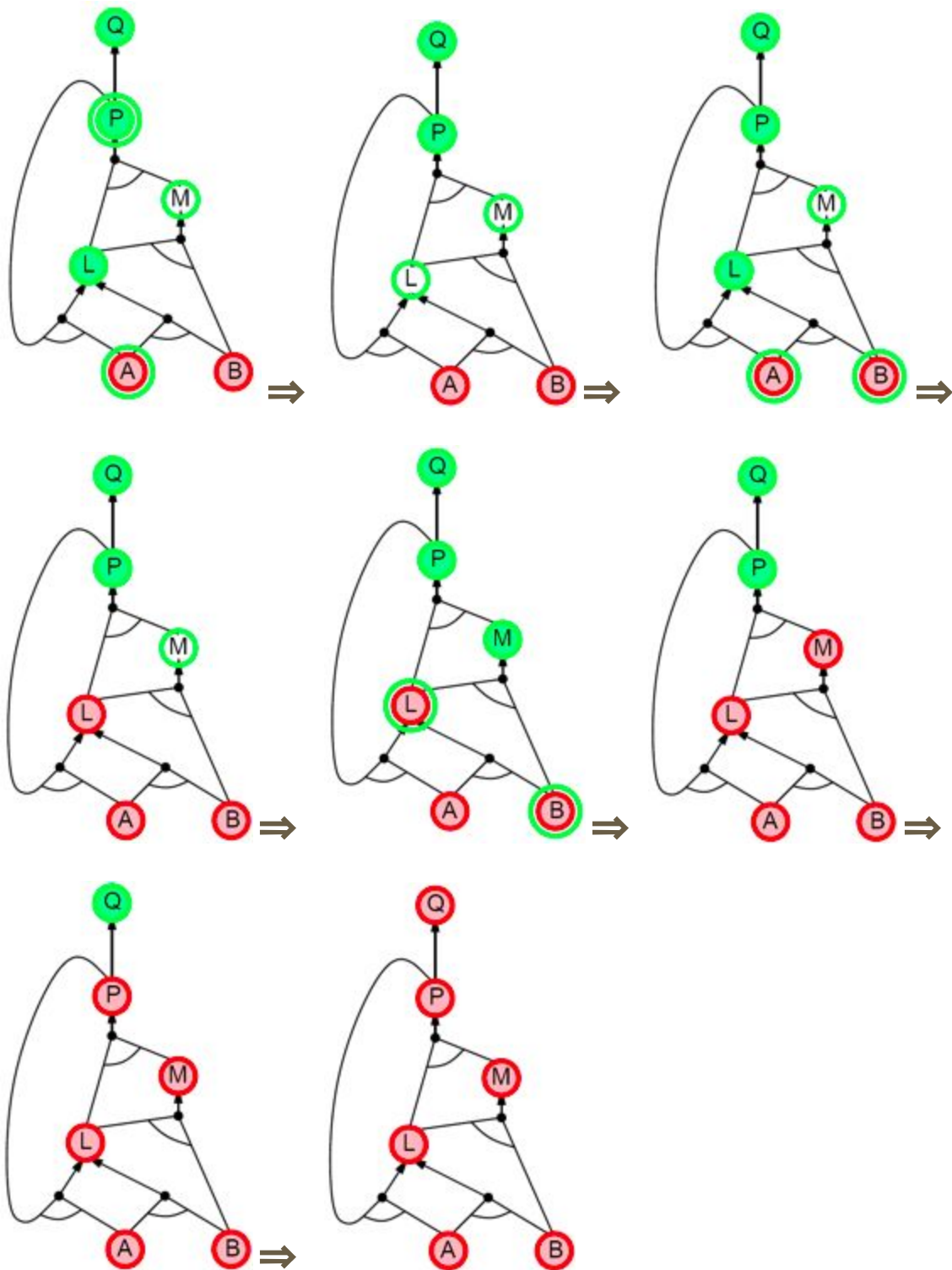




Backward Chaining

- Idea: work backwards from the query q:
 - To prove q by BC,
 - Check if q is known already, or
 - Prove by BC all premises of some rule concluding q
- Avoid loops: Check if new subgoal is already on the goal stack
- Avoid repeated work: check if new subgoal
 - Has already been proved true, or
 - Has already failed





Forward Chaining vs. Backward Chaining

- Forward Chaining is data driven

- Automatic, unconscious processing
 - E.g. object recognition, routine decisions
 - May do lots of work that is irrelevant to the goal
- Backward Chaining is goal driven
 - Appropriate for problem solving
 - E.g. “Where are my keys?”, “How do I start the car?”

Robinson’s Inference Rule

Example

- Let $P = \text{Loves}(X, \text{father-of}(X))$
- $Q1 = \text{Likes}(X, \text{mother-of}(X))$
- $Q2 = \text{Likes}(\text{john}, Y)$
- $R = \text{Hates}(X, Y)$
- After Unifying $Q1$ and $Q2$: $Q = Q1 = Q2 = (\text{john}, \text{mother-of}(\text{john}))$
- Where ‘the substitution S is given by:

$$S = \{\text{john} / X, \text{mother-of}(X) / Y\}$$

$$= \{\text{john}/X \text{ mother-of}(\text{john})/Y\}$$
- The resolvent $(P \vee R)[S]$ is: $(P \vee R)[S] = \text{Loves}(\text{john}, \text{father-of}(\text{john})) \vee \text{hates}(\text{john}, \text{mother-of}(\text{john}))$

Scene Interpretation using Predicate Logic



- Lets use **B** for the property of being a bicycle,
- **M** for the property of being a man, and
- **R** for the relation of riding,

Following formula describes the following salient fact about the situation in the picture:

$$\exists x \exists y : (M(x) \wedge B(y) \wedge R(x, y))$$

Practice

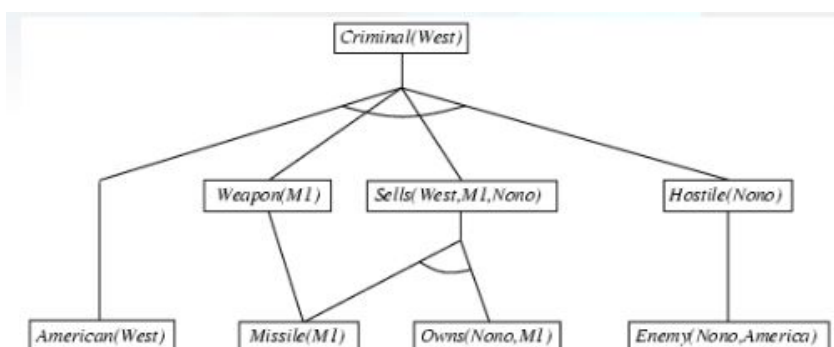
- Proof that West is a criminal

... it is a crime for an American to sell weapons to hostile nations:
 $American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x)$
 Nono ... has some missiles, i.e., $\exists x Owns(Nono, x) \wedge Missile(x)$:

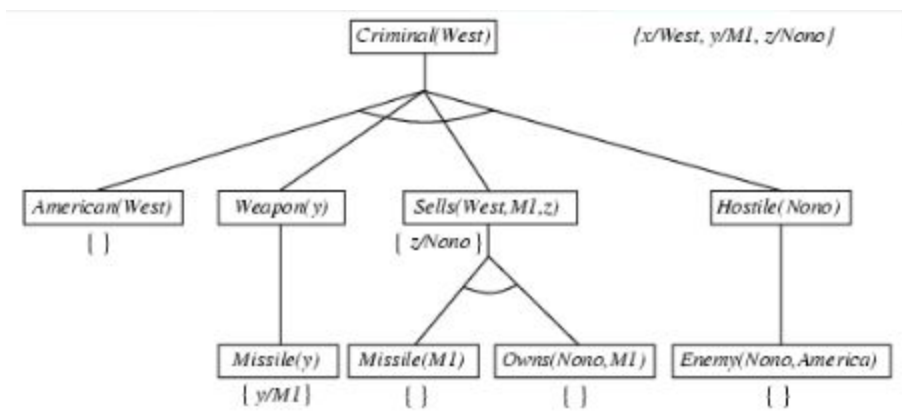
$Owns(Nono, M_1)$ and $Missile(M_1)$
 ... all of its missiles were sold to it by Colonel West
 $Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$
 Missiles are weapons:

$Missile(x) \Rightarrow Weapon(x)$
 An enemy of America counts as "hostile":
 $Enemy(x, America) \Rightarrow Hostile(x)$
 West, who is American ...

$American(West)$
 The country Nono, an enemy of America ...
 $Enemy(Nono, America)$

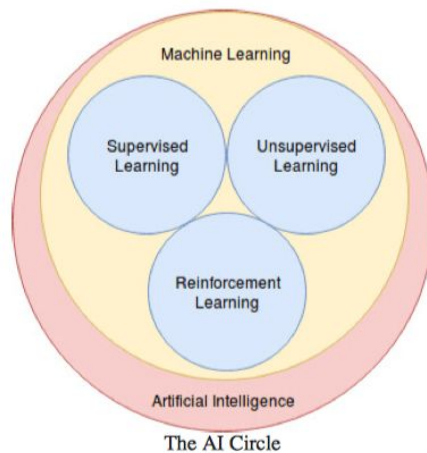


⇒ Forward Chaining



⇒ Backward Chaining

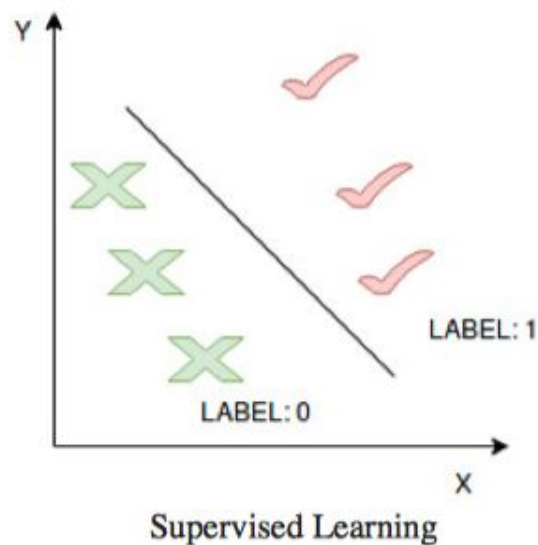
Machine Learning



- Machine Learning : is a way to implement artificial intelligence.
- Machine learning is a branch of computer science in which you devise or study the design of algorithms that can learn.

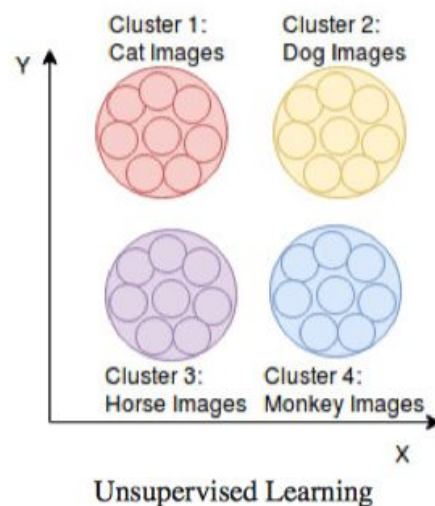
Supervised Learning

- It is a learning technique wherein the whole learning process is governed.
- These learning algorithms main goal is to predict the outcome given a set of training samples along with the training labels, also known as classifying a data point.



Unsupervised Learning

- It can find suitable structure and patterns in the data.
- These consistent patterns become apparent, the similar data points can be clustered together, and different data points will be in different clusters.
- It is mostly used to project high-dimensional data into low-dimension for visualization or analysis purposes.



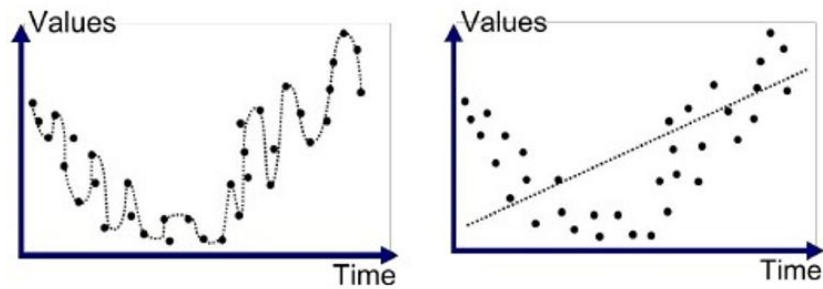
Reinforcement Learning

- It has an agent that learns how to behave in an environment by taking actions and quantifying the results.
- If the agent makes a correct response, it gets a reward point, which boosts up the agent's confidence to take more such actions.

Linear Regression

- **Linear Regression** : is a linear approach to modeling the relationship between y and one or more x .
- x = independent variable or explanatory
- y = dependent variable or scalar response
- **Simple Linear Regression** : the case of one explanatory variable
- **Multiple Linear Regression** : for more than one explanatory variable

Overfitting and Underfitting



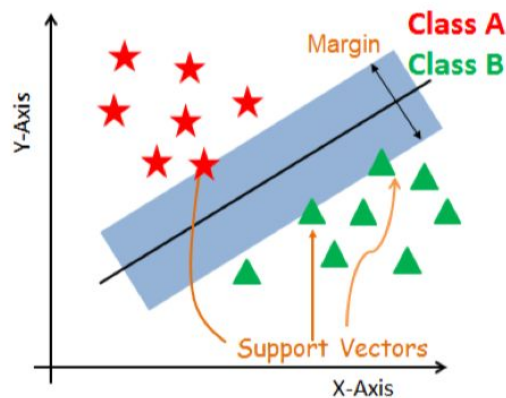
Overfitted

Underfitted

- **Overfitting** : occurs when a statistical model or ML that captures the noise of the data
- **Underfitting** : occurs when a statistical model or ML that cannot capture the underlying trend of the data

Support Vector Machine

- **SVM** is a classification approach, but it can be employed in both types of classification and regression problems.
- It can easily handle multiple continuous and categorical variables.
- **SVM** constructs a hyperplane in multidimensional space to separate different classes.
- **SVM** generates optimal hyperplane in an iterative manner, which is used to minimize an error.
- **Purpose** : to find a maximum marginal hyperplane(MMH) that best divides the dataset into classes.

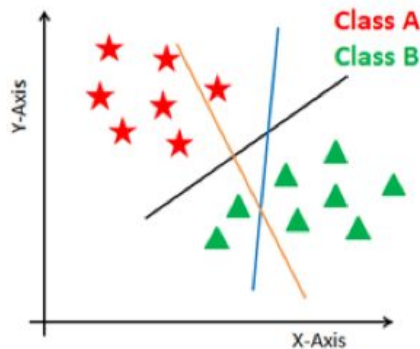


- **Support vectors**
 - are the data points, which are closest to the hyperplane.

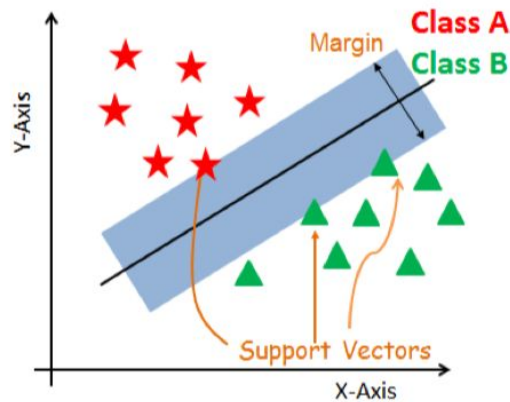
- These points will define the separating line better by calculating margins.
- These points are more relevant to the construction of the classifier.
- **Hyperplane** : is a decision plane which separates between a set of objects having different class memberships.
- **Margin**
 - It is a gap between the two lines on the closest class points.
 - This is calculated as the perpendicular distance from the line to support vectors or closest points.
 - If the margin is larger in between classes, then it is considered a good margin, a smaller margin is a bad margin.

How does SVM work?

- **Objective**
 - To select a hyperplane with the maximum possible margin between support vectors in the given dataset.
 - To segregate the given dataset in the best possible way.
 - SVM searches for the maximum marginal hyperplane in the following steps:
 1. Generate hyperplanes which segregates the classes in the best way.
Here, the blue and orange have higher classification error, but the black is separating the two classes correctly.

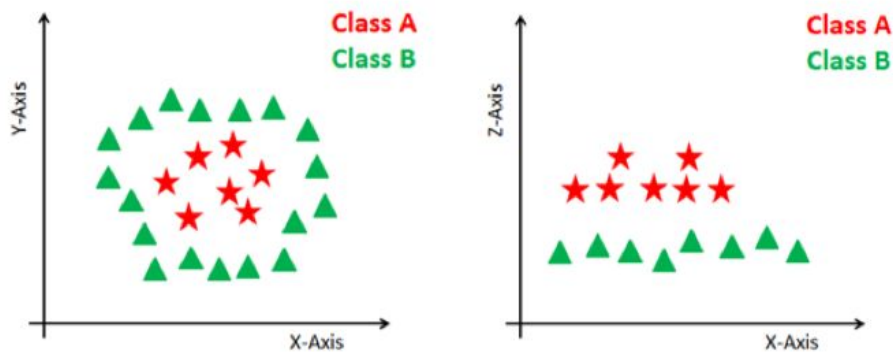


2. Select the right hyperplane with the maximum segregation from either nearest data points as shown in the right-hand side figure.



Dealing with non-linear and inseparable planes

- SVM uses a kernel trick to transform the input space to a higher dimensional space as shown on the right.
- The data points are plotted on the x-axis and z-axis (Z is the squared sum of both x and y: $z=x^2+y^2$).



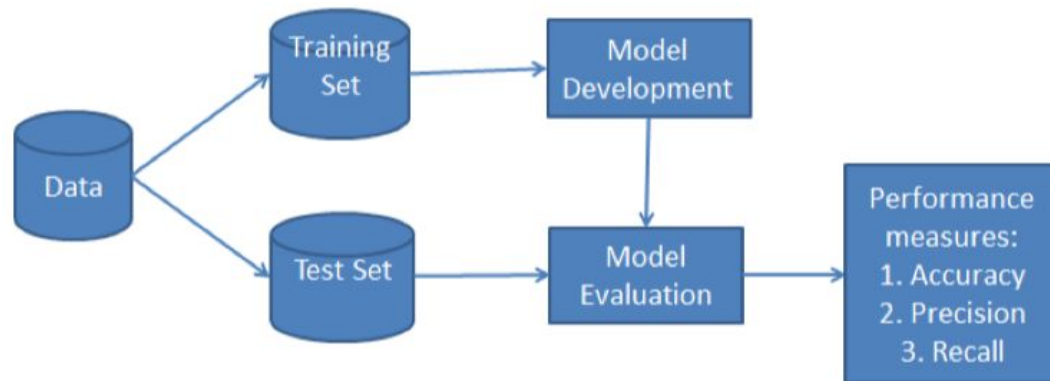
Advantages

- Good accuracy
- Perform faster prediction
- Use less memory because it uses a subset of training points in the decision phase
- Works well with a clear margin of separation and with high dimensional space

Disadvantages

- Not suitable for large datasets because of its high training time
- Takes more time in training
- It works poorly with overlapping classes
- Sensitive to the type of kernel used

Naive Baye's Classifier



- Naive Bayes is a statistical classification technique based on Bayes Theorem.
- It is one of the simplest supervised learning algorithms.
- Naive Bayes classifier is the fast, accurate and reliable algorithm.
- Naive Bayes classifiers have high accuracy and speed on large datasets.
- Naive Bayes classifier assumes that the effect of a particular feature in a class is independent of other features.

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

- $P(h)$: the probability of hypothesis h being true (regardless of the data). This is known as the prior probability of h .
- $P(D)$: the probability of the data (regardless of the hypothesis). This is known as the prior probability.
- $P(h|D)$: the probability of hypothesis h given the data D . This is known as posterior probability.
- $P(D|h)$: the probability of data d given that the hypothesis h was true. This is known as posterior probability.

How it works

1. Step 1: Calculate the prior probability for given class labels
2. Step 2: Find Likelihood probability with each attribute for each class
3. Step 3: Put these values in Bayes Formula and calculate posterior probability.
4. Step 4: See which class has a higher probability, given the input belongs to the higher probability class.

Example: Given an example of weather conditions and playing sports. You need to calculate the probability of playing sports.

Whether	Play
Sunny	No
Sunny	No
Overcast	Yes
Rainy	Yes
Rainy	Yes
Rainy	No
Overcast	Yes
Sunny	No
Sunny	Yes
Rainy	Yes
Sunny	Yes
Overcast	Yes
Overcast	Yes
Rainy	No

Frequency Table		
Whether	No	Yes
Overcast		4
Sunny	2	3
Rainy	3	2
Total	5	9

Likelihood Table 1				
Whether	No	Yes		
Overcast		4	$\approx 4/14$	0.29
Sunny	2	3	$\approx 5/14$	0.36
Rainy	3	2	$\approx 5/14$	0.36
Total	5	9		
	$\approx 5/14$	$\approx 9/14$		
	0.36	0.64		

Likelihood Table 2				
Whether	No	Yes	Posterior Probability for No	Posterior Probability for Yes
Overcast		4	$0/5=0$	$4/9=0.44$
Sunny	2	3	$2/5=0.4$	$3/9=0.33$
Rainy	3	2	$3/5=0.6$	$2/9=0.22$
Total	5	9		

Probability of playing:

$$P(\text{Yes} \mid \text{Overcast}) = P(\text{Overcast} \mid \text{Yes}) P(\text{Yes}) / P(\text{Overcast}) \dots\dots\dots(1)$$

1. Calculate Prior Probabilities: $P(\text{Overcast}) = 4/14 = 0.29$ $P(\text{Yes}) = 9/14 = 0.64$

1. Calculate Posterior Probabilities:

$$P(\text{Overcast} \mid \text{Yes}) = 4/9 = 0.44$$

1. Put Prior and Posterior probabilities in equation (1) $P(\text{Yes} \mid \text{Overcast}) = 0.44 * 0.64 / 0.29 = 0.98$ (Higher)

Probability of not playing:

$$P(\text{No} \mid \text{Overcast}) = P(\text{Overcast} \mid \text{No}) P(\text{No}) / P(\text{Overcast}) \dots\dots\dots(2)$$

1. Calculate Prior Probabilities: $P(\text{Overcast}) = 4/14 = 0.29$ $P(\text{No}) = 5/14 = 0.36$ 1. Calculate Posterior Probabilities:

$$P(\text{Overcast} \mid \text{No}) = 0/9 = 0$$

1. Put Prior and Posterior probabilities in equation (2) $P(\text{No} \mid \text{Overcast}) = 0 * 0.36 / 0.29 = 0$

The probability of a 'Yes' class is higher. So you can determine if the weather is overcast than players will play the sport.

Second Approach (In case of multiple features)

HOW NAIVE BAYES CLASSIFIER WORKS?

Whether	Temperature	Play
Sunny	Hot	No
Sunny	Hot	No
Overcast	Hot	Yes
Rainy	Mild	Yes
Rainy	Cool	Yes
Rainy	Cool	No
Overcast	Cool	Yes
Sunny	Mild	No
Sunny	Cool	Yes
Rainy	Mild	Yes
Sunny	Mild	Yes
Overcast	Mild	Yes
Overcast	Hot	Yes
Rainy	Mild	No

01 CALCULATE PRIOR PROBABILITY FOR GIVEN CLASS LABELS

02 CALCULATE CONDITIONAL PROBABILITY WITH EACH ATTRIBUTE FOR EACH CLASS

03 MULTIPLY SAME CLASS CONDITIONAL PROBABILITY.

04 MULTIPLY PRIOR PROBABILITY WITH STEP 3 PROBABILITY.

05 SEE WHICH CLASS HAS HIGHER PROBABILITY, HIGHER PROBABILITY CLASS BELONGS TO GIVEN INPUT SET STEP.

Probability of playing:

$P(\text{Play} = \text{Yes} \mid \text{Weather} = \text{Overcast}, \text{Temp} = \text{Mild}) = P(\text{Weather} = \text{Overcast}, \text{Temp} = \text{Mild} \mid \text{Play} = \text{Yes})P(\text{Play} = \text{Yes}) \dots\dots\dots(1)$

$P(\text{Weather} = \text{Overcast}, \text{Temp} = \text{Mild} \mid \text{Play} = \text{Yes}) = P(\text{Overcast} \mid \text{Yes}) P(\text{Mild} \mid \text{Yes}) \dots\dots\dots(2)$ 1.

Calculate Prior Probabilities: $P(\text{Yes}) = 9/14 = 0.64$

2. Calculate Posterior Probabilities: $P(\text{Overcast} \mid \text{Yes}) = 4/9 = 0.44$ $P(\text{Mild} \mid \text{Yes}) = 4/9 = 0.44$

3. Put Posterior probabilities in equation (2) $P(\text{Weather} = \text{Overcast}, \text{Temp} = \text{Mild} \mid \text{Play} = \text{Yes}) = 0.44 * 0.44 = 0.1936$ (Higher)

4. Put Prior and Posterior probabilities in equation (1) $P(\text{Play} = \text{Yes} \mid \text{Weather} = \text{Overcast}, \text{Temp} = \text{Mild}) = 0.1936 * 0.64 = 0.124$

Probability of not playing:

$P(\text{Play} = \text{No} \mid \text{Weather} = \text{Overcast}, \text{Temp} = \text{Mild}) = P(\text{Weather} = \text{Overcast}, \text{Temp} = \text{Mild} \mid \text{Play} = \text{No})P(\text{Play} = \text{No}) \dots\dots\dots(3)$

$P(\text{Weather} = \text{Overcast}, \text{Temp} = \text{Mild} \mid \text{Play} = \text{No}) = P(\text{Weather} = \text{Overcast} \mid \text{Play} = \text{No})P(\text{Temp} = \text{Mild} \mid \text{Play} = \text{No}) \dots\dots\dots(4)$

1. Calculate Prior Probabilities: $P(\text{No}) = 5/14 = 0.36$

2. Calculate Posterior Probabilities: $P(\text{Weather} = \text{Overcast} \mid \text{Play} = \text{No}) = 0/9 = 0$ $P(\text{Temp} = \text{Mild} \mid \text{Play} = \text{No}) = 2/5 = 0.4$

3. Put posterior probabilities in equation (4) $P(\text{Weather} = \text{Overcast}, \text{Temp} = \text{Mild} \mid \text{Play} = \text{No}) = 0 * 0.4 = 0$

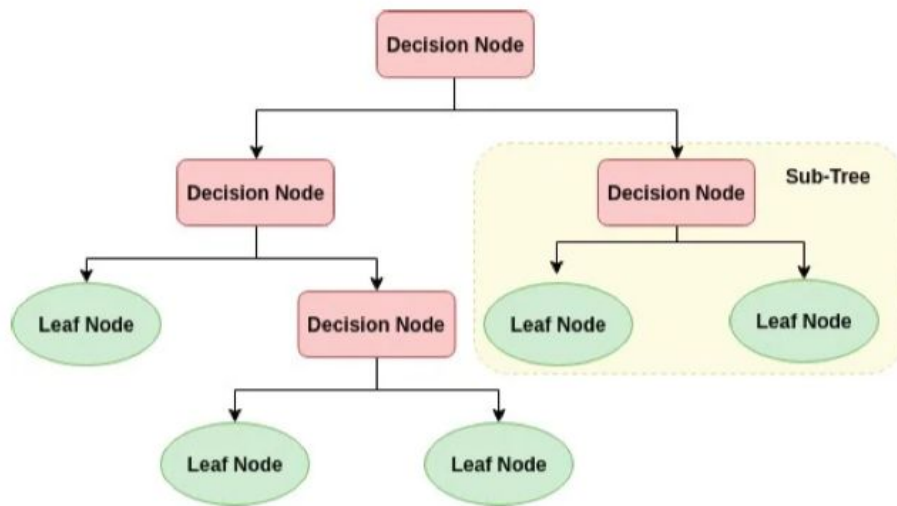
4. Put prior and posterior probabilities in equation (3) $P(\text{Play} = \text{No} \mid \text{Weather} = \text{Overcast}, \text{Temp} = \text{Mild}) = 0 * 0.36 = 0$

The probability of a 'Yes' class is higher. So you can say here that if the weather is overcast then players will play the sport.

Decision Tree Algorithm

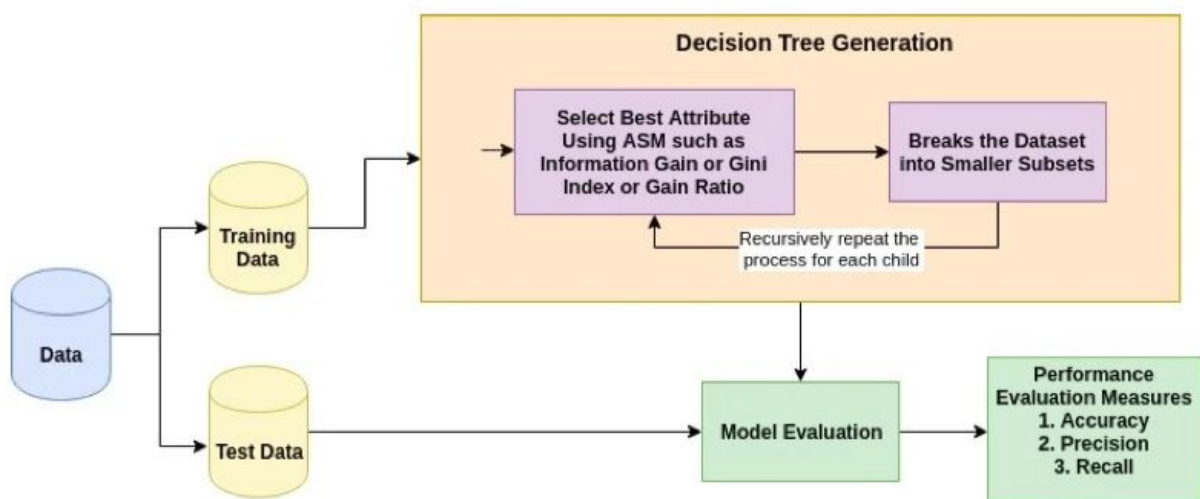
What is Decision Tree?

- Is a flowchart-like tree structure where an internal node represents feature(or attribute), the branch represents a decision rule, and each leaf node represents the outcome.
- Is a white box type of ML algorithm. It shares internal decision-making logic.
- The time complexity of decision trees is a function of the number of records and number of attributes in the given data. Its training time is faster than neural network algorithm.
- Is a distribution-free or non-parametric method, which does not depend upon probability distribution assumptions.
- Can handle high dimensional data with good accuracy.




How does it work?

1. Select the best attribute using Attribute Selection Measures(ASM) to split the records.
2. Make that attribute a decision node and breaks the dataset into smaller subsets.
3. Starts tree building by repeating this process recursively for each child until one of the conditions will match:
 - All the tuples belong to the same attribute value
 - There are no more remaining attributes
 - There are no more instances



Advantage

- 
- Decision trees are easy to interpret and visualize.
 - It can easily capture Non-linear patterns.
 - It requires fewer data preprocessing from the user, for example, there is no need to normalize columns.
 - It can be used for feature engineering such as predicting missing values, suitable for variable selection.
 - The decision tree has no assumptions about distribution because of the non-parametric nature of the algorithm.

Disadvantage

- Sensitive to noisy data. It can overfit noisy data.
- The small variation(or variance) in data can result in different decision tree. This can be reduced by bagging and boosting algorithms.
- Decision trees are biased with imbalance dataset, so it is recommended that balance out the dataset before creating the decision tree.

K-nearest Neighbor

- KNN or k-nearest neighbor algorithm is a supervised learning algorithm
- It is an instance-based machine learning algorithm, where new data points are classified based on stored, labeled instances (data points).
- The k in KNN is a crucial variable or hyperparameter that helps in classifying a data point accurately.
- The k is the number of nearest neighbors you want to take a vote from when classifying a new data point.
- The prediction can be of two types:
 - **Classification** : a class label is assigned to a new data point
 - **Regression** : a value is assigned to the new data point.

How it works

- First, load all the data and initialize the value of k,
- Then, the distance between the stored data points and a new data point that want to classify is calculated using various similarity or distance metrics.
- Next, the distance values are sorted either in descending or ascending order and top or lower k-nearest neighbors are determined.
- The labels of the k-nearest neighbors are gathered, and a majority vote or a weighted vote is used for classifying the new data point. The new data point is assigned a class label based on a certain data point that has the highest score out of all the stored data points.
- Finally, the predicted class for the new instance is returned.

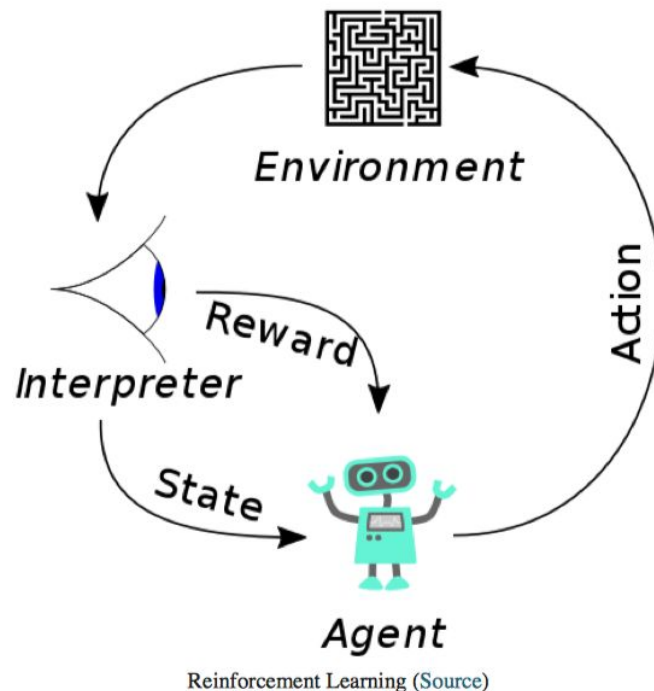
Drawback of KNN

1. Firstly, complexity in searching the nearest neighbor for each new data point.
2. Secondly, determining the value of k sometimes becomes a tedious task.
3. Finally, it is also not clear which type of distance metric one should use while computing the nearest neighbors.

K-mean Clustering

- Clustering is the task of grouping together a set of objects in a way that objects in the same cluster are more similar to each other than to objects in other clusters.
- Similarity is a metric that reflects the strength of relationship between two data objects.
- Clustering is used for exploratory data mining.
- **K-Means** falls under the category of **centroid-based clustering**.
 - A **centroid** is a data point (imaginary or real) at the center of a cluster.
 - In **centroid-based clustering**, **clusters** are represented by a central vector or a centroid.
 - This **centroid** might not necessarily be a member of the dataset.
 - **Centroid-based clustering** is an iterative algorithm in which the notion of similarity is derived by how close a data point is to the centroid of the cluster

Reinforcement Learning



What is reinforcement learning?

- It is a method of machine learning wherein the software agent learns to perform certain actions in an environment which lead it to maximum reward.
- It does so by exploration and exploitation of knowledge it learns by repeated trials of maximizing the reward.

Reinforcement vs. Supervised vs. Unsupervised

- **Supervised learning** : predicts continuous ranged values or discrete labels/classes based on the training it receives from examples with provided labels or values.
- **Unsupervised learning** : tries to club together samples based on their similarity and determine discrete clusters.
- **Reinforcement learning** : is a subset of Unsupervised learning, performs learning very differently. It takes up the method of "cause and effect".

Basic Terminology


- **Agent** : a hypothetical entity which performs actions in an environment to gain some reward. Action (a) : All the possible moves that the agent can take.
- **Environment (e)** : A scenario the agent has to face. State (s): Current situation returned by the environment.
- **Reward (R)** : An immediate return sent back from the environment to evaluate the last action by the agent.
- **Policy (π)** : The strategy that the agent employs to determine next action based on the current state. Value (V): The expected long-term return with discount, as opposed to the short-term reward R. $V_\pi(s)$, is defined as the expected long-term return of the current state under policy π .
- **Q-value or action-value (Q)** : Q-value is similar to Value, except that it takes an extra parameter, the current action a. $Q_\pi(s, a)$ refers to the long-term return of the current state s, taking action a under policy π .

How it works

- **Value Based** : in a value-based reinforcement learning method, you try to maximize a value function $V(s)$. As defined in the terminology previously, $V_\pi(s)$ is the expected long-term return of the current state s under policy π . Thus, $V(s)$ is the value of reward which the agent expects to gain in the future upon starting at that state s.

$$V_\pi(s) = E_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | S_t = s]$$

- **Policy-based** : in a policy-based reinforcement learning method, you try to come up with a policy such that the action performed at each state is optimal to gain



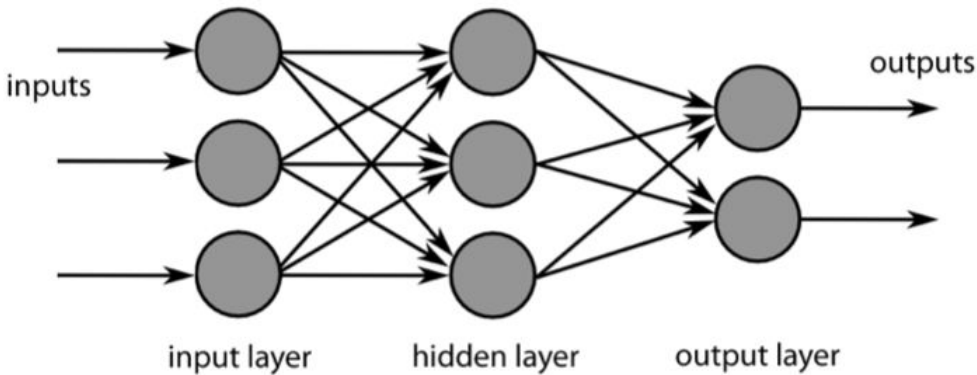
maximum reward in the future. Here, no value function is involved. We know that the policy π determines the next action at any state s . There are two types of policy-based RL methods

- **Deterministic**: at any state s , the same action is produced by the policy π .
- **Stochastic** : each action has a certain probability, given by the equation below -

$$\text{Stochastic Policy} : \pi(a|s) = P[A_t = a|S_t = s]$$

- **Model-Based**: in this type of reinforcement learning, you create a virtual model for each environment, and the agent learns to perform in that specific environment. Since the model differs for each environment, there is no singular solution or algorithm for this type.

Deep Learning



- **Deep learning** : stacking multiple such hidden layers between the input and the output layer.
- **Input layer** : can be pixels of an image or a time series data
- **Hidden layer** : or weights which are learned while the neural network is trained
- **Output layer** : gives a prediction of the input fed into the network.

CNN - Convolutional Neural Network

- It is a main category to do image recognition, image classifications, objects detection, recognition faces etc. in Deep Learning
- CNN image classifications takes an input image, process it and classify it under certain categories (Eg., Dog, Cat, Tiger, Lion). Computers sees an input image as array of pixels and it depends on the image resolution. Based on the image resolution, it will see $h \times w \times d$ (h = Height, w = Width, d = Dimension).
- Technically, deep learning CNN models to train and test, each input image will pass it through a series of convolution layers with filters (Kernels), Pooling, fully connected layers (FC) and apply Softmax function to classify an object with probabilistic values between 0 and 1. The below figure is a complete flow of CNN to process an input image and classifies the objects based on values.

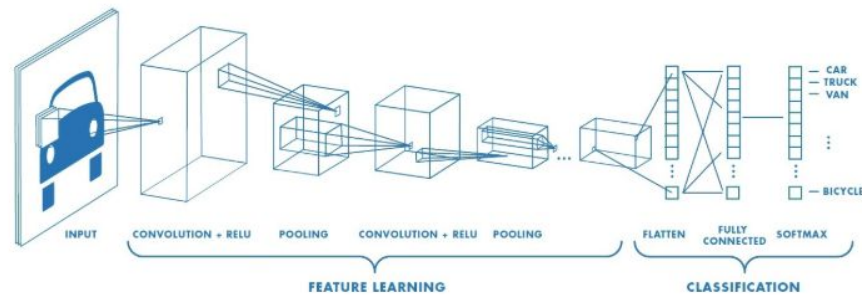


Figure 2 : Neural network with many convolutional layers

Forward propagation

- Multiply - add process
- Dot product
- Forward propagation for one data point at a time
- Output is the prediction for that data point

Pooling

- Pooling layers section would reduce the number of parameters when the images are too large.
- Spatial pooling also called subsampling or downsampling which reduces the dimensionality of each map but retains the important information.
- Spatial pooling can be of different types:
 - Max Pooling
 - Average Pooling
 - Sum Pooling
- Max pooling take the largest element from the rectified feature map. Taking the largest element could also take the average pooling. Sum of all elements in the feature map call as sum pooling.

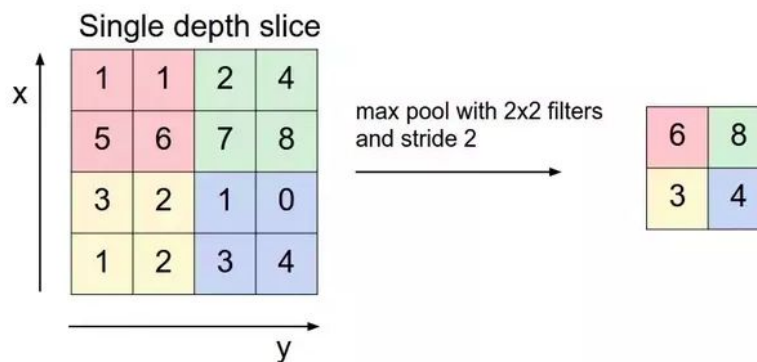
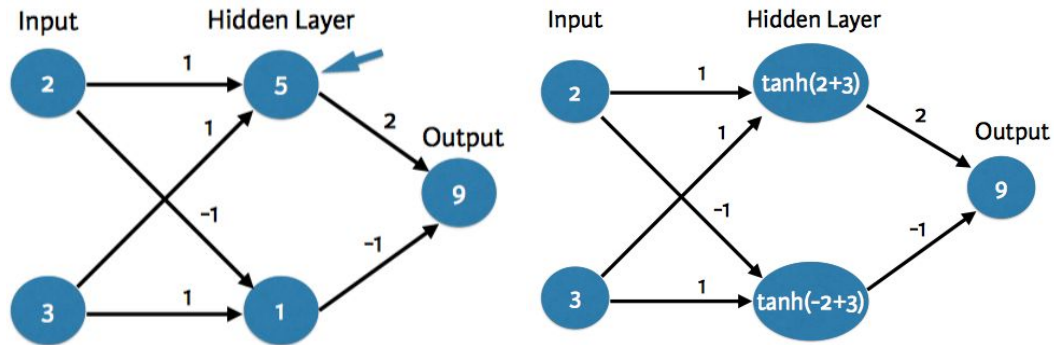


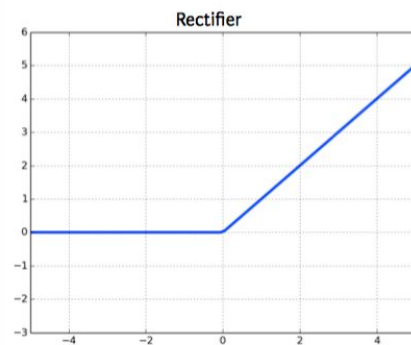
Figure 8 : Max Pooling

Activation Function

- Applied to node inputs to produce node output



ReLU (Rectified Linear Activation)



$$RELU(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

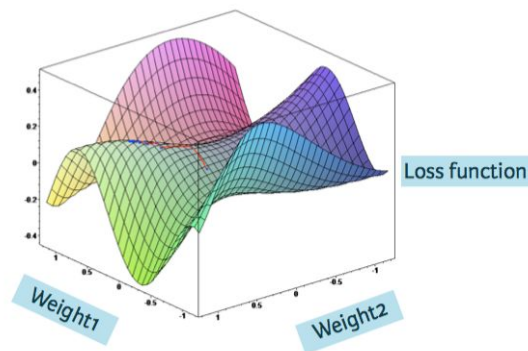
Loss Function

- Aggregates errors in predictions from many data points into single number
- Measure of model's predictive performance
- Lower loss function value means a better model
- Goal: Find the weights that give the lowest value for the loss function
- Gradient descent

Squared error loss function

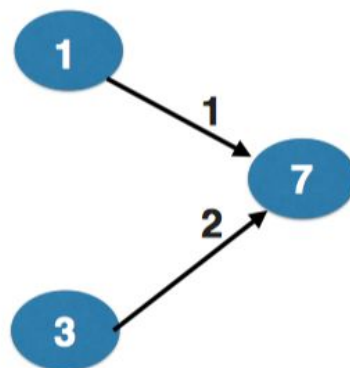
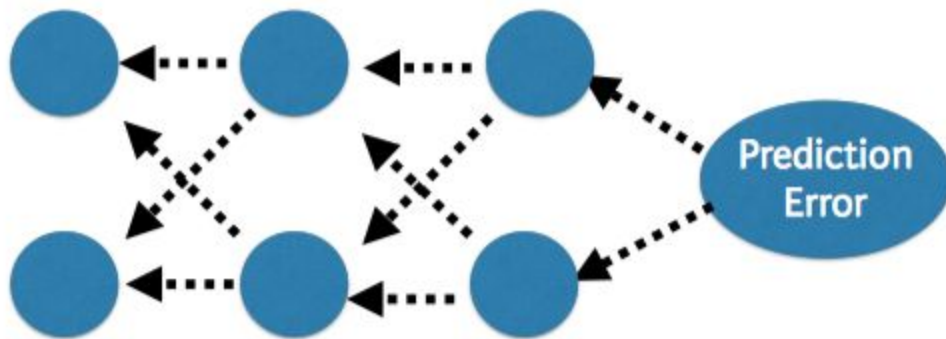
Prediction	Actual	Error	Squared Error
10	20	-10	100
8	3	5	25
6	1	5	25

- Total Squared Error: 150
- Mean Squared Error: 50



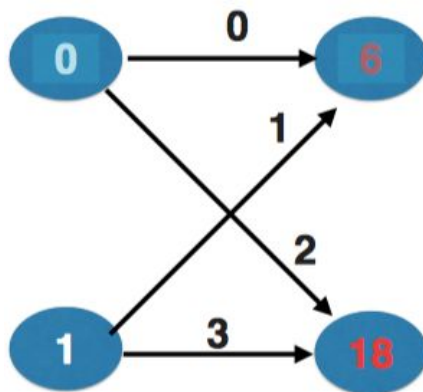
Backpropagation

- Go back one layer at a time
- Gradients for weight is product of:
 1. Node value feeding into that weight
 2. Slope of loss function w.r.t node it feeds into
 3. Slope of activation function at the node it feeds into
- Need to also keep track of the slopes of the loss function w.r.t node values
- Slope of node values are the sum of the slopes for all weights that come out of them



ReLU Activation Function
Actual Target Value = 4
Error = 3

- Top weight's slope = $1 * 6$
- Bottom weight's slope = $3 * 6$



Current Weight Value	Gradient
0	0
1	6
2	0
3	18