Artificial Intelligence

Artificial Intelligence is an artificial machine capable of:

- *Thinking Humanly*
- *Thinking Rationally*
- *Acting Humanly*
- *Acting Rationally*

Thinking ------> Perception ------> Action

Acting Rationally

Artificial Intelligence is a **rational agent**, that perceives the **environment** with **sensors**, and influences the environment with **actuators** in order to maximise some **performance criteria**.

Rational Agent	Anything that makes decisions, typically a person, firm, machine, or software. It attempts to maximise its performance measure on the basis of its percept sequence.
Environment	The configuration or state the agent exists in, it aims to move from a starting state to a goal state.
Sensors	A device which detects or measures a physical property and records, indicates, or otherwise responds to it.
Actuators	A component of a machine that is responsible for moving and controlling a mechanism or system which can interact with the environment.
Performance Criteria	A value to be maximised by the agent to encourage movement towards a given goal
Agent Function / Policy	The mapping from a sequence of observations to actions

Agent Type	**Performance Measure**	**Environment**	**Actuators**	**Sensors**
Medical diagnosis system	Healthy patient, reduced costs	Patient, hospital, staff	Display of questions, tests, diagnoses, treatments, referrals	Keyboard entry of symptoms, findings, patient's answers
Satellite image analysis system	Correct image categorisation	Downlink from orbiting satellite	Display of scene categorisation	Colour pixel arrays
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts: bins	Joined arm and hand	Camera, join and angle sensors

Environment

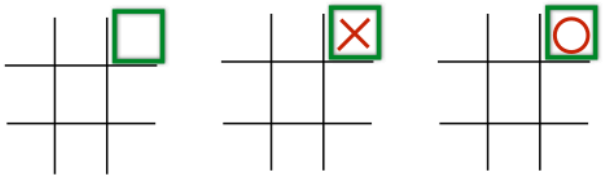
The environment is described by its configuration or state. The agent's Performance Criteria depends on the state - typically the agent will want to change the environment from one state to another.

Example: **Tic-Tac-Toe**

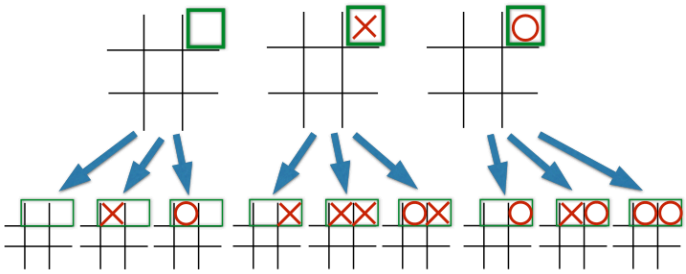
Performance Criteria - reach goal state of three in a row

To compute the number of possible states:

- Each square has 3 possible states



- 2 squares therefore have 3^2 possible states



- Therefore 9 squares have 3^9 possible states (19, 683)
    - However some of the states are invalid
    - You cannot have only 2 X's in a state because it goes against the rules of the game
    - There are 5,478 legal states
- One way to create an agent is to store all 5,478 states and memorise which move to make

For example: in the following state, what move should the agent X make?

O		X
	O	O
X		X

There are 3 possible actions but only one action will lead to a goal state.

- A rational agent would choose to put X in the bottom centre and therefore win

However, in this state what move should the agent X make?

O		X

There are 7 possible actions but none of these will result in an immediate goal state

- We implement a Performance Measure and make the agent judge which action will increase Performance Measure the most.

Observable

Information about the state of the environment available to an agent at a given time

<table>
<tr><th>Fully Observable Environment</th><th>Partially Observable Environment</th></tr>
<tr><td>

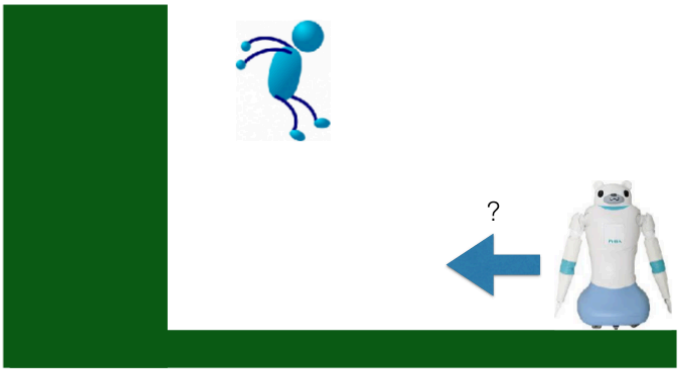
- The state of the environment provides **all information** required for the agent to make a rational decision regarding which action to take

</td><td>

- **Not all information** available to an agent about the state of the environment

</td></tr>
<tr><td colspan="2">

Example: The man is falling. Where should the robot go to catch him?



</td></tr>
<tr><td>

- If we know the position of the man and his velocity, we have enough information to calculate where he will land

$$f\left(s_i\right) \rightarrow a_j$$

</td><td>

- If we know the position and NOT his velocity, we can:
  - Make a sequence of observations of his position
  - Estimate his velocity
  - Make a reasonable position as to where he will land

$$f\left(s_1, s_2, \ldots, s_i\right) \rightarrow a_j$$

</td></tr>
<tr><td colspan="2">

Reasons why environments may be **Partially Observable**:

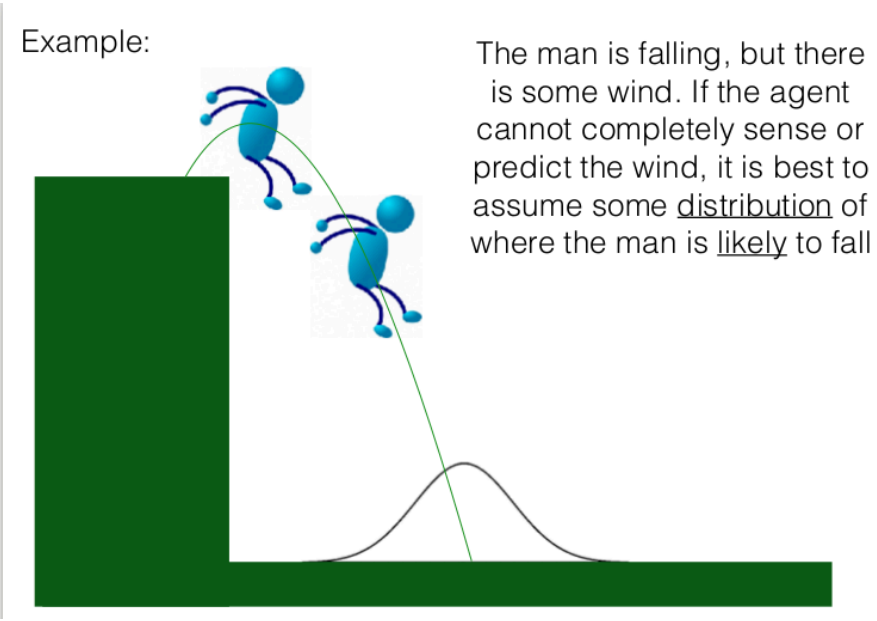
- Inadequate sensing
  - Perception is blocked e.g. by a wall
- Inaccurate or noisy sensing
  - Sensors are wrong
- Sensing or computing is too slow
  - Environment changes too quickly
- Environment is too complex
  - Cannot store all information in a state

</td></tr>
</table>

Deterministic Environment	Stochastic Environment

• The next state is completely determined by the current state and the action executed by the agent
  ○ The agent knows precisely the outcome of each action
	• If the next state has a degree of randomness to its outcome
	• However, an environment may be completely deterministic but only partially observable
	  ○ It would be advantageous for the agent to treat is as stochastic

Example: The man is falling. Where should the robot go to catch him?

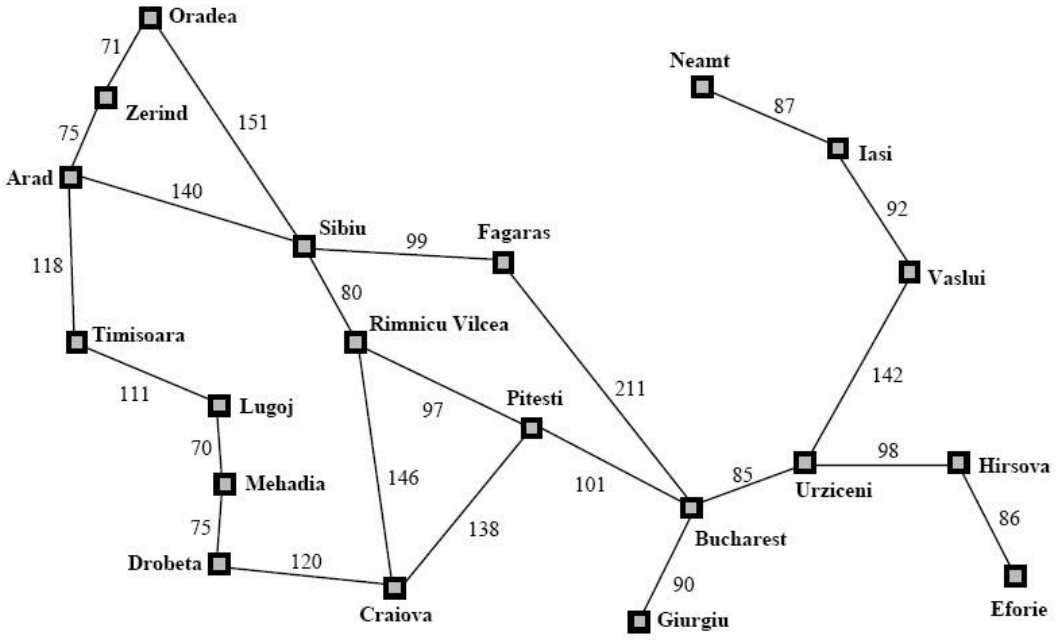


Discrete Environment	Continuous Environment
• There are only a finite number of possible states	• There are an (in theory) infinite number of possible states
Example: • Tic-Tac-Toe • Chess	Example: • Robotics • Stock trading • Medical diagnosis

Algorithms

How does a rational agent decide which action to take?

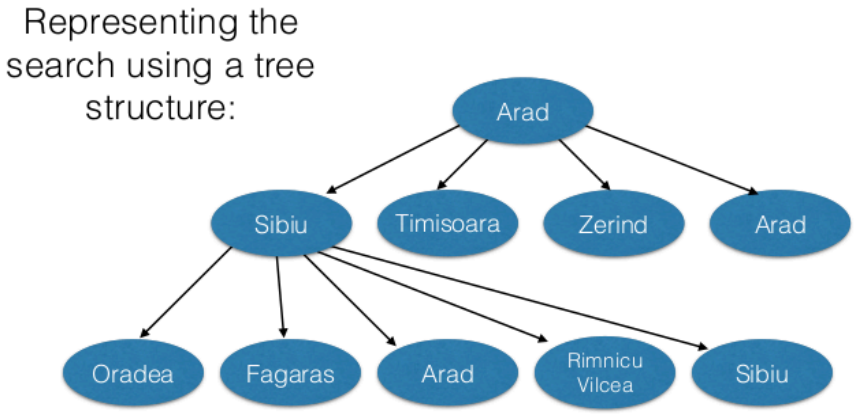
One Method: Search through all available action options and their consequences (predicting what will happen), then choose the action with best possible outcome



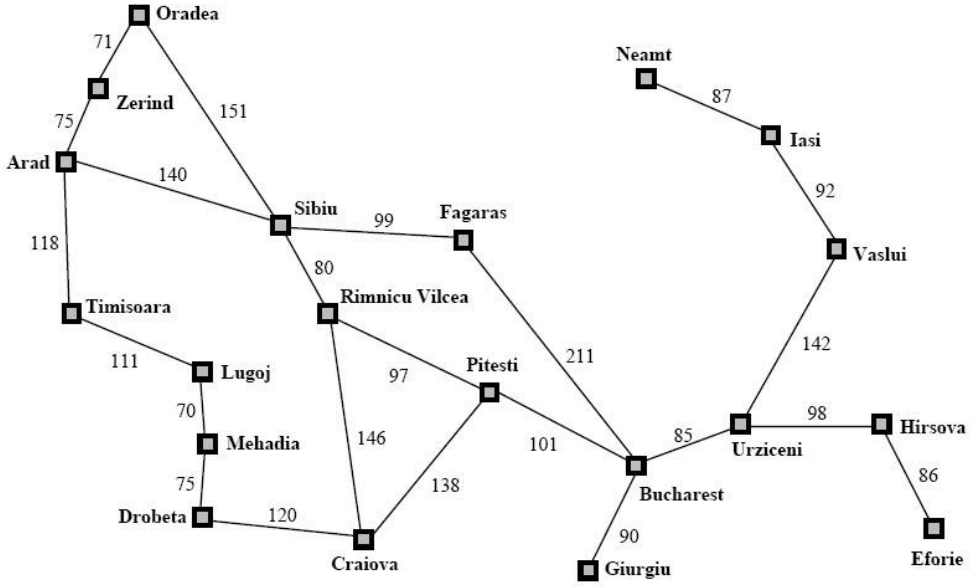
Assumptions: **Discrete** (Finite possibilities), **Fully Observable** (We know the complete state of environment), **Deterministic** (Our actions and our actions alone determine the following state)

*We are in **Arad** and wish to travel to **Bucharest** in the fastest way possible*

1. There are 4 actions available to us: Zerind, Sibiu or Timisoara (or stay in Arad)
2. If we travel to Sibiu there are now 5 actions available to us: (Return to) Arad, Oradea, Fagaras, Rimnicu Vilcea (or stay in Sibiu)



- The search tray may be **infinite** since we could indefinitely return from Arad to Sibiu and back to Arad...etc
    - However, the search tree for Tic-Tac-Toe is finite since the rules of the game do not allow your to return to the previous state
    - Chess may be infinite since the same pieces could move back and forth forever
- To avoid infinite searches, we can store which nodes have already been expanded and then not allow connections to a node in that list.



1. We start in Arad and expand nodes into the **frontier** (nodes available for expansion)
    - List: Arad
    - Frontier: Zerind, Sibiu, Timisoara
2. We expand each node on the frontier again creating a new frontier and store each node we have expanded
    - List: Arad, Zerind, Sibiu, Timisoara
    - Frontier: Oradea, Fagaras, Riminicu Vilcea, Lugoj
3. We expand again - NOTE: We do not allow a connection from the frontier of Oradea to Sibiu because Sibiu is in our List. Oradea becomes a dead end and we do not expand further
    - List: Arad, Zerind, Sibiu, Timisoara, Oradea, Fagaras, Riminicu Vilcea, Lugoj
    - Frontier: Mehadia, Craiova, Pitesti, Bucharest
4. We have found a route to Bucharest: Arad -> Sibiu -> Fagaras -> Bucharest

Effectively, we are using an **algorithm** to find a **solution** to the problem.

- Algorithm: a set of rules for determining a sequence of operations precisely
- Solution: an action sequence the Agent may use to change the environment from initial state to goal state

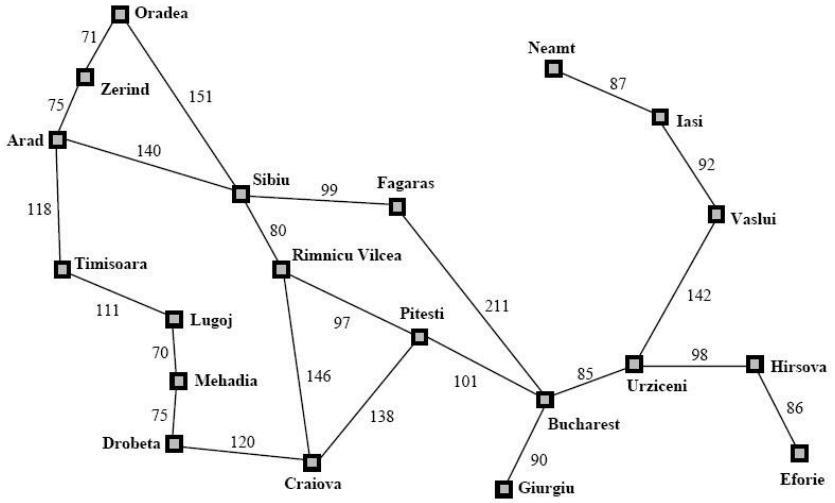
Things to consider when using an Algorithm:

- **Completeness**
    - Is the algorithm guaranteed for find a solution (assuming there is one)
- **Optimality**
    - Is the algorithm guaranteed to find the optimal solution
- **Optimal Solution**
    - The action sequence from initial state to a goal state with lowest possible **cost** (defined by Performance Measure)
- **Time Complexity**
    - How long will it take to find a solution in the worst case?
- **Space Complexity**
    - How much memory is required to find a solution in the worst case?

Breadth-First Search

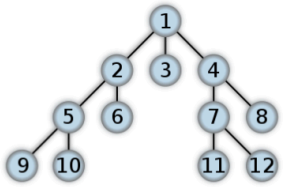
An algorithm for traversing/searching a tree or graph where you start traversing from a selected node and traverse the environment a layer (frontier) at a time.

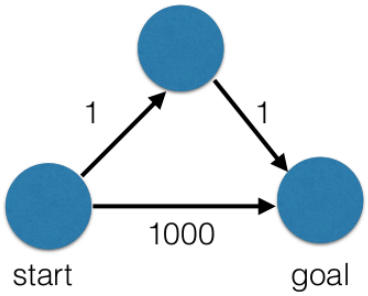
- Current nodes in the frontier are held in a queue and added to a list of vertices once visited
- The value at the front of the queue is always explored first and dequeued once expanded



1. **Start in Arad**
    - Queue: Arad
    - Vertices: Arad
2. **Expand Arad** -> Arad is dequeued and Zerind, Sibiu and Timisoara added to current frontier
    - Queue: Zerind, Sibiu, Timisoara
    - Vertices: Arad, Zerind, Sibiu, Timisoara
3. **Expand Zerind** -> Zerind is dequeued and Oradea is added to current frontier
    - Queue: Sibiu, Timisoara, Oradea
    - Vertices: Arad, Zerind, Sibiu, Timisoara, Oradea
4. **Expand Sibiu** -> Sibiu is dequeued and Fagaras and Rimnicu Vilcea are added to current frontier
    - Queue: Timisoara, Oradea, Fagaras, Rimnicu Vilcea
    - Vertices: Arad, Zerind, Sibiu, Timisoara, Oradea, Fagaras, Rimnicu Vilcea
5. **Expand Timisoara** -> Timisoara is dequeued and Lugoj is added to current frontier
    - Queue: Oradea, Fagaras, Rimnicu Vilcea, Lugoj
    - Vertices: Arad, Zerind, Sibiu, Timisoara, Oradea, Fagaras, Rimnicu Vilcea, Lugoj
6. **Expand Oradea** -> Oradea is dequeued, however all possible expansions are held within vertices so cannot be added to queue
    - Queue: Fagaras, Rimnicu Vilcea, Lugoj
    - Vertices: Arad, Zerind, Sibiu, Timisoara, Oradea, Fagaras, Rimnicu Vilcea, Lugoj
7. **Expand Fagaras** -> Fagaras is dequeued and Bucharest is added to current frontier
    - Queue: Rimnicu Vilcea, Lugoj, Bucharest
    - Vertices: Arad, Zerind, Sibiu, Timisoara, Oradea, Fagaras, Rimnicu Vilcea, Lugoj, Bucharest
8. **Expand Rimnicu Vilcea** -> Rimnicu Vilcea is dequeued and Pitesti and Craiova added to current frontier
    - Queue: Lugoj, Bucharest, Pitesti, Craiova
    - Vertices: Arad, Zerind, Sibiu, Timisoara, Oradea, Fagaras, Rimnicu Vilcea, Lugoj, Bucharest, Pitesti, Craiova
9. **Expand Lugoj** -> Lugoj is dequeued and Mehadia is added to current frontier
    - Queue: Bucharest, Pitesti, Craiova, Mehadia
    - Vertices: Arad, Zerind, Sibiu, Timisoara, Oradea, Fagaras, Rimnicu Vilcea, Lugoj, Bucharest, Pitesti, Craiova, Mehadia
10. **Expand Bucharest** -> DONE!

BFS always chooses the path with least number of steps next:

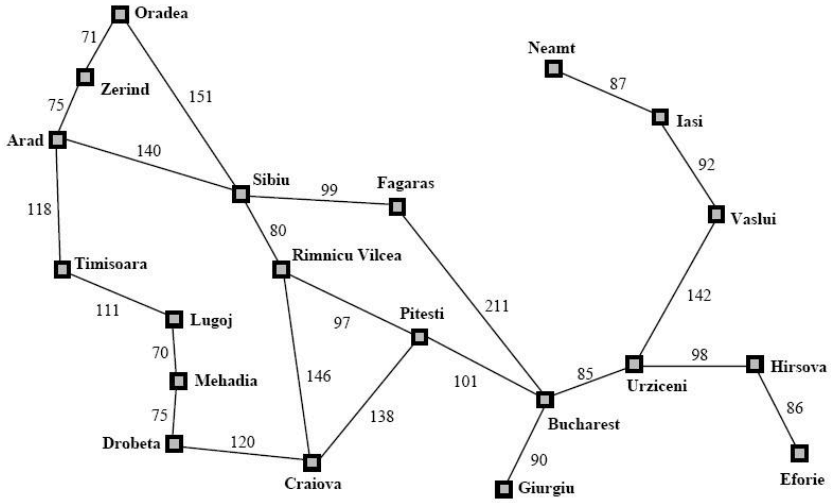


Breadth First Search	
Completeness	BFS is completed (unless graph is infinite). Every vertex will eventually be expanded and the goal will be found.
Optimality	No, BFS will find the path with least number of STEPS which isn't necessarily the optimal  However, in an environment where every step has a uniform cost then BFS would be optimal
Time Complexity	**O(|E|):** In the worst case, BFS will need to traverse every edge in the graph • (|E| =number of edges)
Space Complexity	O(**|V|**): In the worst case, BFS will have to store every vertex (node) in the graph • (|V| = number of nodes)

Depth First Search

An algorithm for traversing/searching a tree or graph where you start traversing from a selected node and expand each 'branch' completely

- Current nodes are held in a stack and added to a list of vertices once visited
- The value at the back of the stack (most recent node) is always expanded first.



1. **Start in Arad**
    - Stack: Arad
    - Vertices: Arad
2. **Expand Arad** -> **Arad** popped off stack, **Sibiu and Timisoara** added to current frontier
    - Stack: Zerind, Sibiu, Timisoara
    - Vertices: Arad, Zerind, Sibiu, Timisoara
3. **Expand Timisoara** -> **Timisoara** popped off stack, **Lugoj** added to current frontier
    - Stack: Zerind, Sibiu, Lugoj
    - Vertices: Arad, Zerind, Sibiu, Timisoara, Lugoj
4. **Expand Lugoj** -> **Lugoj** popped off stack, **Mehadia** added to current frontier
    - Stack: Zerind, Sibiu, Mehadia
    - Vertices: Arad, Zerind, Sibiu, Timisoara, Lugoj, Mehadia
5. **Expand Mehadia** -> **Mehadia** popped off stack, **Drobeta** added to current frontier
    - Stack: Zerind, Sibiu, Drobeta
    - Vertices: Arad, Zerind, Sibiu, Timisoara, Lugoj, Mehadia, Drobeta
6. **Expand Drobeta** -> **Drobeta** popped off stack, **Craiova** added to current frontier
    - Stack: Zerind, Sibiu, Craiova
    - Vertices: Arad, Zerind, Sibiu, Timisoara, Lugoj, Mehadia, Drobeta, Craiova
7. **Expand Craiova** -> **Craiova** popped off stack, **Rimnicu Vilcea and Pitesti** added to current frontier
    - Stack: Zerind, Sibiu, Rimnicu Vilcea, Pitesti
    - Vertices: Arad, Zerind, Sibiu, Timisoara, Lugoj, Mehadia, Drobeta, Craiova, Rimnicu Vilcea, Pitesti
8. **Expand Pitesti** -> **Pitesti** popped off stack, **Bucharest** added to current frontier
    - Stack: Zerind, Sibiu, Rimnicu Vilcea, Bucharest
    - Vertices: Arad, Zerind, Sibiu, Timisoara, Lugoj, Mehadia, Drobeta, Craiova, Rimnicu Vilcea, Pitesti, Bucharest
9. **Expand Bucharest** -> DONE!

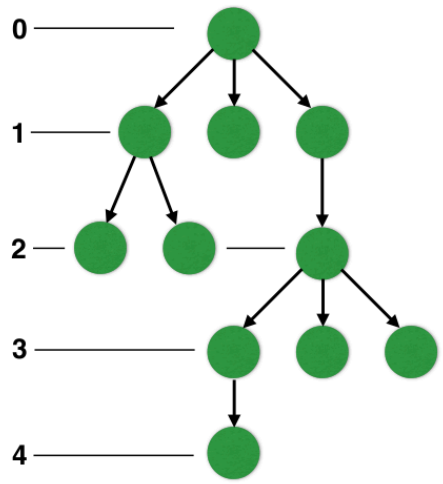
In general, DFS will find a longer solution than BFS

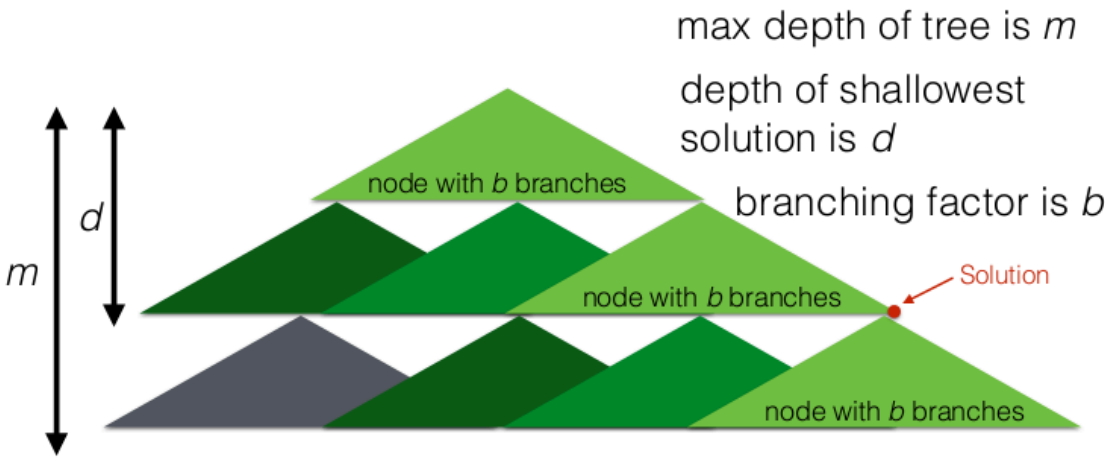
Depth First Search	
Completeness	The way we completed the example is complete. However, in general **DFS is not**! • We must not allow for repeated states, or the algorithm can get stuck in an infinite loop
Optimality	No, algorithm may return unnecessarily long paths
Time Complexity	O(|E|) (without repetition): In the worst case, DFS will traverse every edge in the graph
Space Complexity	(|V|) (without repetition): In the worst case, DFS will have to store every vertex (node) in the graph

Trees

- The depth of a node is the length of the path to the root node.
- The **depth** of a tree is the depth of it's *deepest* node
- The **branching factor** of a tree is the *maximum* number of edges on any one node in the tree

This tree's depth is 4 and it's branching factor is 3





	BFS	DFS
Time	$O(b^d)$	$O(b^m)$
Space	$O(b^d)$	O(bm)

Example: b = 10; m = d = 16; 1000 bytes/node
DFS requires 160 kilobytes, BFS would require 10m terabytes

Uniform Cost Search

Uses a queue like BFS but uses a priority system within the queue, whereby the cumulative cost of the branch is stored in the queue and the lower cost branch is explored as a priority.

1. **Start in Arad**
    - Queue: 0 Arad
    - Vertices: 0 ARad
2. **Expand Arad**
    - Queue:75 Zerind, 140 Sibiu, 118 Timisoara
    - Vertices: 0 Arad, 75 Zerind, 140 Sibiu, 118 Timisoara
3. **Expand Zerind**
    - Queue: 140 Sibiu, 118 Timisoara, 146 Oradea
    - Vertices: 0 Arad, 75 Zerind, 140 Sibiu, 118 Timisoara, 146 Oradea
4. **Expand Timisoara**
    - Queue: 140 Sibiu, 146 Oradea, 229 Lugoj
    - Vertices: 0 Arad, 75 Zerind, 140 Sibiu, 118 Timisoara, 146 Oradea, 229 Lugoj
5. **Expand Sibiu**
    - Queue: 146 Oradea, 229 Lugoj, 239 Fagaras, 220 R. Vilcea
    - Vertices: 0 Arad, 75 Zerind, 140 Sibiu, 118 Timisoara, 146 Oradea, 229 Lugoj, 239 Fagaras, 220 R. Vilcea
6. **Expand Oradea -> returning to Sibiu at 297 rejected since we have 140 Sibiu**
    - Queue: 229 Lugoj, 239 Fagaras, 220 R. Vilcea
    - Vertices: 0 Arad, 75 Zerind, 140 Sibiu, 118 Timisoara, 146 Oradea, 229 Lugoj, 239 Fagaras, 220 R. Vilcea
7. **Expand R. Vilcea**
    - Queue: 229 Lugoj, 239 Fagaras, 317 Pitesti, 336 Craiova
    - Vertices: 0 Arad, 75 Zerind, 140 Sibiu, 118 Timisoara, 146 Oradea, 229 Lugoj, 239 Fagaras, 220 R. Vilcea, 317 Pitesti, 336 Craiova
8. **Expand Lugoj**
    - Queue: 239 Fagaras, 317 Pitesti, 336 Craiova, 299 Mehadia
    - Vertices: 0 Arad, 75 Zerind, 140 Sibiu, 118 Timisoara, 146 Oradea, 229 Lugoj, 239 Fagaras, 220 R. Vilcea, 317 Pitesti, 336 Craiova, 299 Mehadia
9. **Expand Fagaras**
    - Queue: 317 Pitesti, 336 Craiova, 299 Mehadia, 450 Bucharest
    - Vertices: 0 Arad, 75 Zerind, 140 Sibiu, 118 Timisoara, 146 Oradea, 229 Lugoj, 239 Fagaras, 220 R. Vilcea, 317 Pitesti, 336 Craiova, 299 Mehadia, 450 Bucharest
10. **Expand Mehadia**
    - Queue: 317 Pitesti, 336 Craiova, 450 Bucharest, Drobeta 374
    - Vertices: 0 Arad, 75 Zerind, 140 Sibiu, 118 Timisoara, 146 Oradea, 229 Lugoj, 239 Fagaras, 220 R. Vilcea, 317 Pitesti, 336 Craiova, 229 Mehadia, 450 Bucharest, Drobeta 374
11. **Expand Pitesti -> We could consider deleting the 450 Bucharest path at this point**
    - Queue: 336 Craiova, 450 Bucharest, 375 Drobeta, 418 Bucharest
    - Vertices: 0 Arad, 75 Zerind, 140 Sibiu, 118 Timisoara, 146 Oradea, 229 Lugoj, 239 Fagaras, 220 R. Vilcea, 317 Pitesti, 336 Craiova, 229 Mehadia, 450 Bucharest, Drobeta 374, 418 Bucharest
12. **Expand Craiova and Drobet…**
    - Queue: 450 Bucharest, 418 Bucharest
    - Vertices: 0 Arad, 75 Zerind, 140 Sibiu, 118 Timisoara, 146 Oradea, 229 Lugoj, 239 Fagaras, 220 R. Vilcea, 317 Pitesti, 336 Craiova, 229 Mehadia, 450 Bucharest, Drobeta 374, 418 Bucharest
13. **Expand Bucharest 418…..DONE!**

Uniform Cost Search	
Completeness	Yes, unless graph is infinite or there are negative costs
Optimality	Yes, whenever a node is expanded, we know we have found the lowest possible path to that node

Time Complexity	$O(b^{1+C^*e})$
	• b = branching factor
	• d = depth
	• d is approx C*/e
	  ○ C* = optimal cost
	  ○ e = cost of least cost action

Space Complexity	$O(b^{1+C^*e})$

A* Search

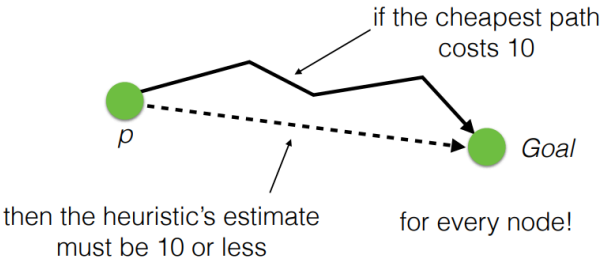
Combining Uniform Cost combined with a Heuristic. E.g. Cost of each node is path cost + penalty for not going East

$$f(p) = \text{PathCost}(p) + h(p)$$

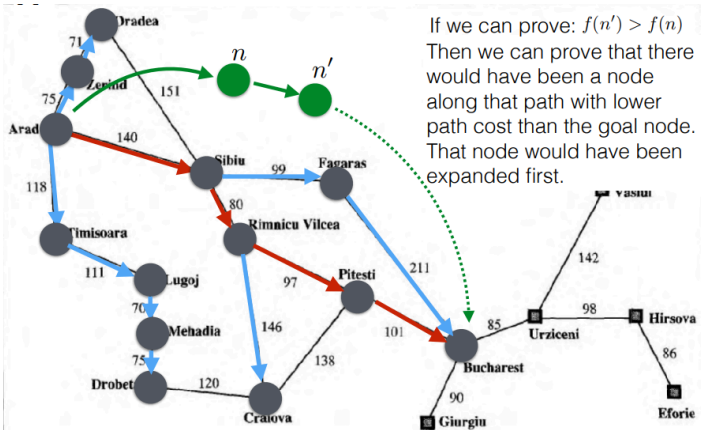
- total estimated cost from start to goal
- actual cost from the start to node *p*
- estimated cost from *p* to the goal

h(*p*) is a *heuristic*, an informed guess, to help us find the goal

A* will be optimal if our search heuristic h(p) is Admissible. It never overestimates the actual cost of the cheapest path from a node to the goal.



<u>Proof of optimality</u>

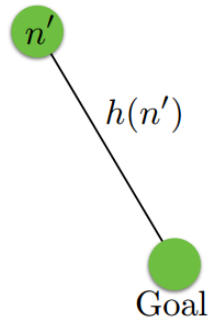


To prove optimality we need to roof that f (n') > f (n) when heuristic is admissible - *that f(n) would be expanded first*

$$f(n') = g(n') + h(n')$$

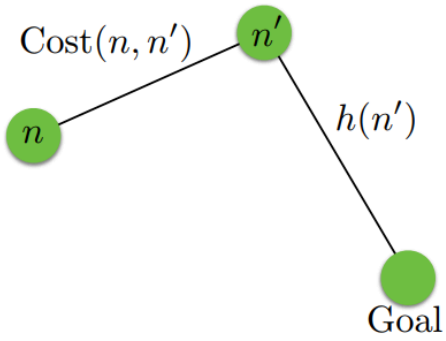
Cost from start to *n'*

Estimate of cost from *n'* to goal

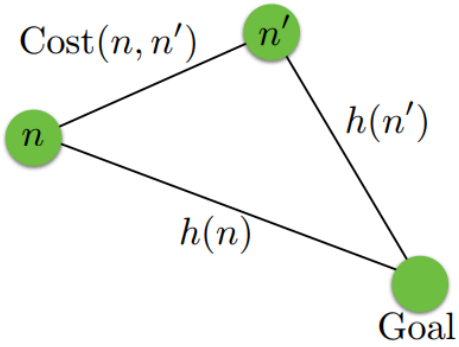


$$f(n') = g(n') + h(n')$$
$$= g(n) + \text{Cost}(n, n') + h(n')$$

Single step cost from *n* to *n'*



$\text{Cost}(n, n') + h(n')$
is more accurate than $h(n)$ as an estimate of the true cost from *n* to Goal. So, $h(n)$ must *underestimate* it



$\text{Cost}(n, n') + h(n') > h(n)$

Triangle inequality

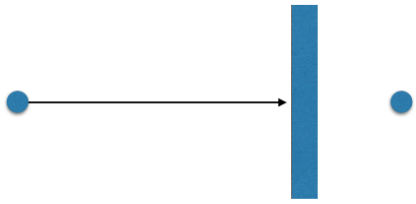
$f(n') > f(n)$

f(n) is less than *f(n')*, so *n* must be expanded first

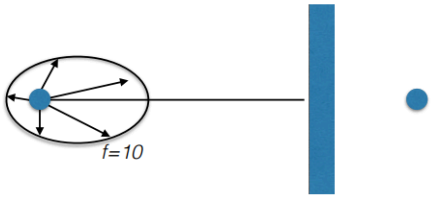
A* is Optimally Efficient

An agent has to search for goal when there is an obstruction the path towards the goal:

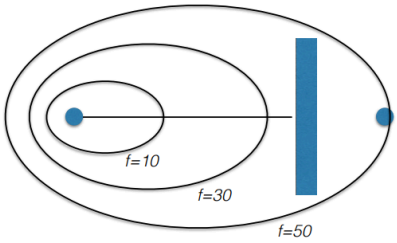
A* will try the straightest possible path first



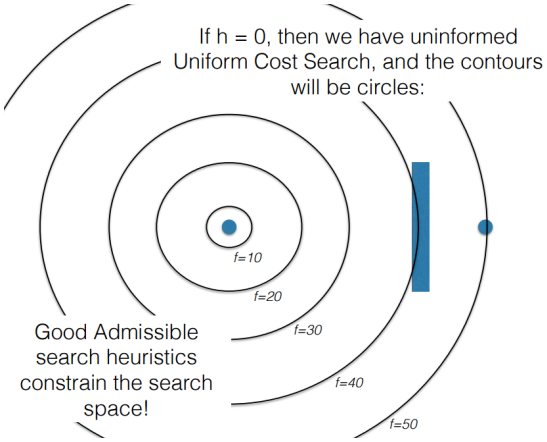
Then will expand its search to nodes that lie within contours of equal cost



Expanding the contour until the goal node is found within the contour of least possible cost



The better the heuristic, the "narrower" the contours will be.



Uniform Cost Search	
Completeness	Yes, so long as cost of nodes are less than or equal to C*
Optimality	Yes (as proved above)
Time Complexity	$O(b^{1+C^*e})$ Worst case, h(p) = 0 and therefore become UCS
Space Complexity	$O(b^{1+C^*e})$

Iterative Improvement Algorithms

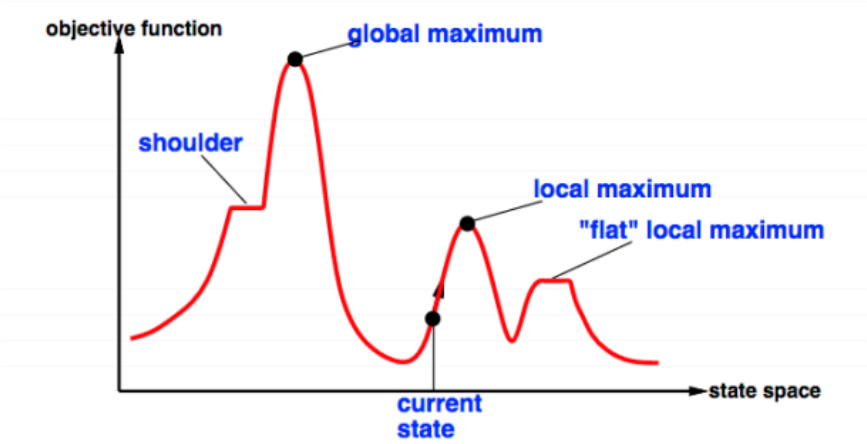
In many problems, the path is irrelevant - the goal state itself is the solution. For example, the Travelling Salesman Problem - finding the shortest route to visit all nodes.

We keep a single current state, and try to gradually improve it.
The goal being to find an optimal configuration.

Example: **Hill-climbing**
"Like climbing Everest in thick fog with amnesia"

- Consider the state space landscape

If the algorithm only climbed when it is possible to do so, it would get stuck at local maxima



Local Search

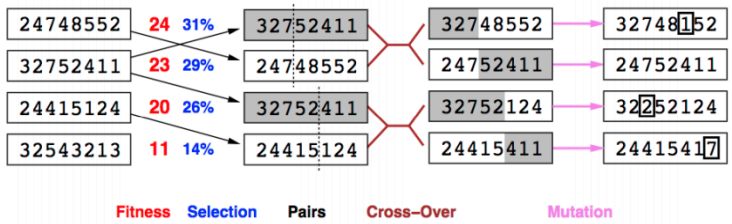
Finds an arbitrary solution to the problem then iteratively and incrementally changes a single element of the solution - e.g. sideways moves

- If the change produces a better solution, an **incremental** change is added to the solution
- The goal is to find an **optimal** solution
- **Simulated Annealing**
  - Escape local maxima by allowing some "bad" moves - ones that do not work towards goal
  - Gradually decrease size and frequency of bad moves
- Another example is Travelling Salesman Problem
  - State-space is set of "complete" configuration - any complete tour
  - We gradually improve it - iterative improvement
  - We find the optimal solution - the fastest route

Local Beam Search
• Keeps 'k' states instead of 1 • Chooses top few of their success • Explores graph by expanding most promising node of a limited set • Uses heuristic which attempts to predict how close a partial solution is to a goal state • Can be expanded into Stochastic Beam Search   ○ Chooses successors randomly, biased towards good ones

Genetic Algorithm

- Mimics Darwinian Evolution
- Variant on Stochastic Beam Search
- Successors are generated as ‘pairs’ which inherit different aspects of their parent solutions



Initialisation	Create initial population of solutions
Evaluation	Each member is evaluated for **‘Fitness’** which is calculated by how well it meets desired requirements
Selection	Individuals with highest fitness are kept and solutions with low fitness are discarded
Crossover	New solutions are created by combining aspects of selected individuals
Mutation	Very small random changes are introduced to solution

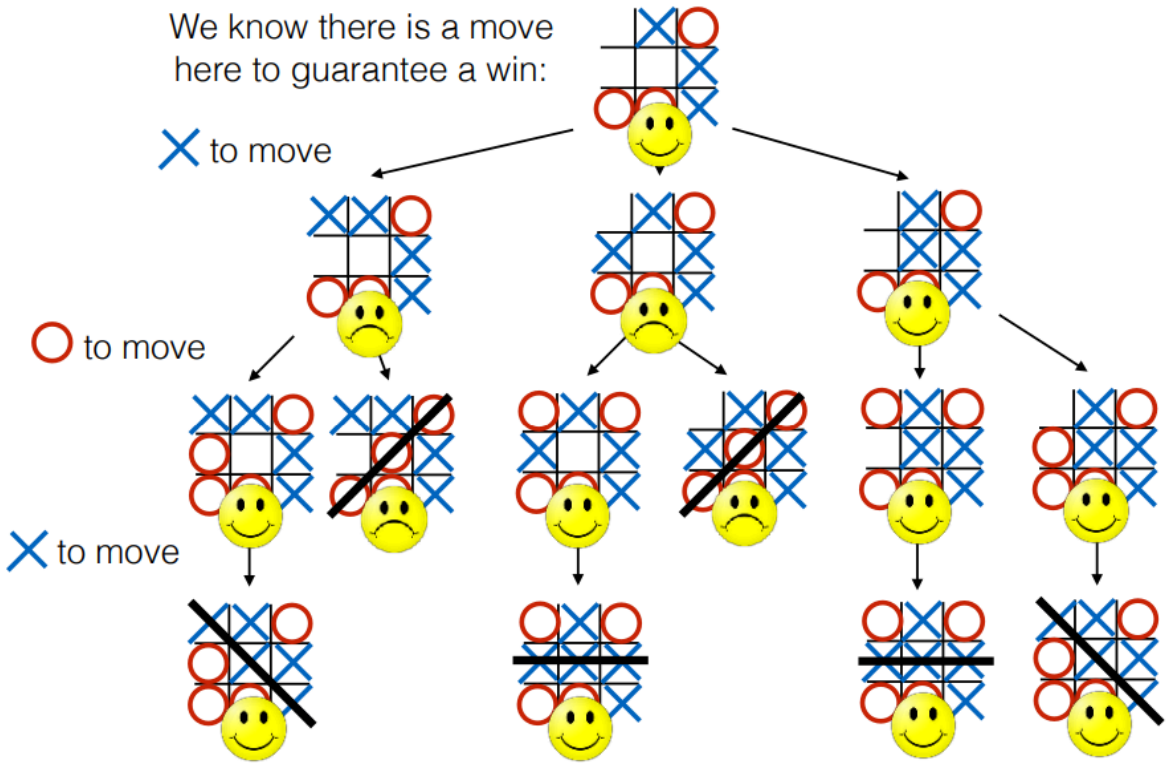
State Space Search

- Successive configurations (states) of an instance are considered with the intention of finding a goal state
- The states form a graph where two states are connected - if there is an operation that can be performed to transform the first state into the second
- Nodes are generated as they are explored and discarded afterwards

Type of Agent (in workbook)

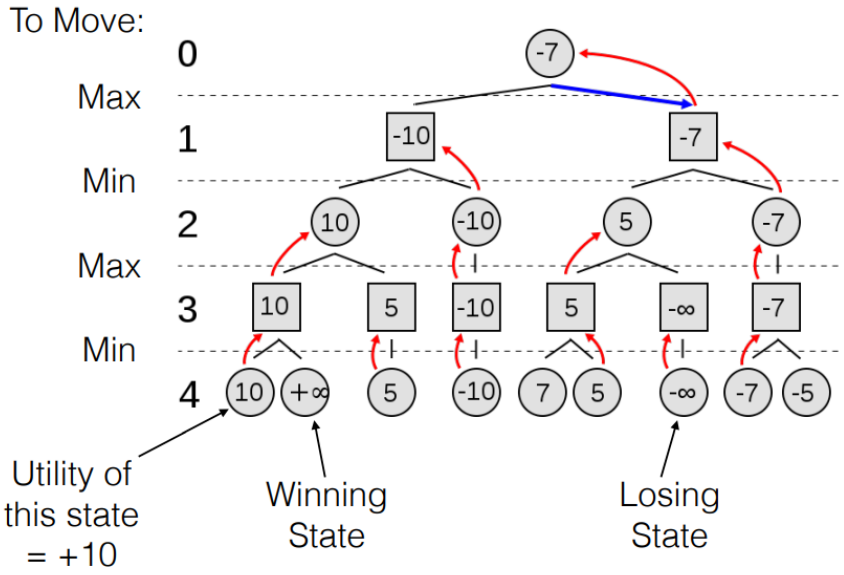
Simple Reflex Agent	
Agent function stored in memory as a "lookup table"	
Advantages	**Disadvantages**
● Extremely fast during execution time -> no computation required	● Cannot change action later ● Cannot adapt to unforeseen circumstances ● What if there are too many states for memory

Competitive Games



- From the first position there are 3 possible options which are simulated
  - There are no wins or losses
- The following possible moves the opposition could take are then simulated
  - In this case, there are two situations in which the opposition wins
- The situations in which the opposition does not win are simulated
  - The agent wins all of these games
- We are now in a situation where we have a number of goal states and a path to get there, however we also have some non-goal states - some of which are connected to the path to the goal states
- Non-complete goal states are considered ‘good’ if they connect directly to a goal states
- However, if a goal state has the possibility of being ‘bad’ then the ‘bad’ outweighs the good

More generally:



Minimax

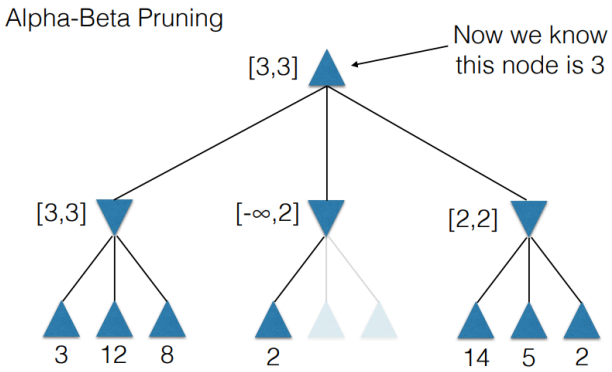
- Decision rule for minimizing possible loss for a worst case scenario
- Behaves like Breadth First Search

Time: O (b^m)
Space: O (bm)

Alpha-Beta Pruning

- Search algorithm that uses minimax but seeks to decrease the number of nodes evaluated by algorithm
- It stops completing evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move

Worst Case - every node visited: O(b^m)
Best Case - Only one opponent move evaluated = O($b^{m/2}$)
In general we say O($b^{3m/4}$)

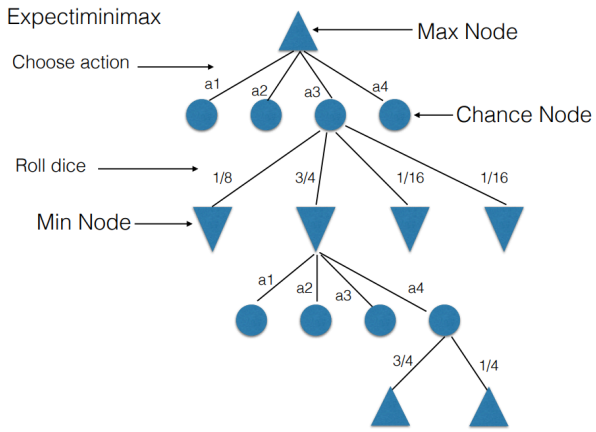


We want to maximise our turn (1st layer) and minimise the opposition's turn (2nd layer) by simulating the effects up to our next turn (3rd layer)

Expectiminimax

- Outcome is based on chance element
- In addition to min & max nodes we introduce chance nodes which take the expected value of a random event occurring
- The utility of a chance node is a weighted average of its children

Time = O($b^m n^m$)
n is number of distinct outcomes of chance



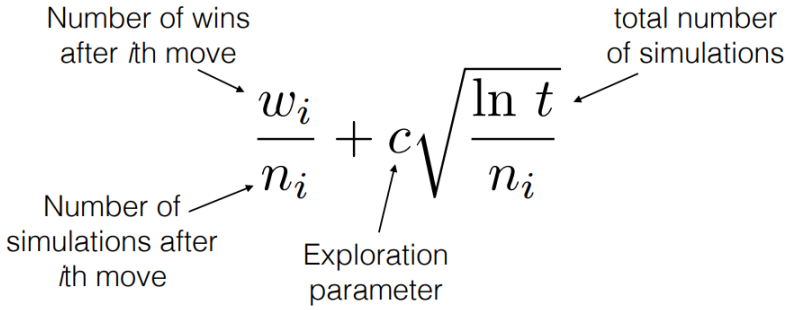
Monte Carlo Tree Search

Selection
- Selects which nodes to explore
  - Uses minimax to a certain depth, with Alpha Beta pruning

Expansion
- If selection does not lead to victory or loss, create more child nodes

Simulation
- Play random moves until completion

Backpropagation
- Use result from simulation to update information in tree



This cost function trades off our desire to exploit the nodes we already know are good vs exploring the tree to find even better nodes

<u>Advantages</u>

- Aheuristic
  - No prior knowledge of the game required
  - Evaluation function can be discovered
- Asymmetric
  - MCTS focuses search on most promising parts of the tree
  - Useful for problems with high branching factors
- Anytime
  - Algorithm can be halted at any point and return current best estimate

	Complexity	
Algorithm	**Time**	**Space**
Depth-First Search	$O(b^d)$	$O(b^d)$
Breadth-First Search	$O(b^m)$	$O(bm)$
Uniform Cost Search	$O(b^{1+c^*/e})$	$O(b^{1+c^*/e})$
A*	$O(b^{1+c^*/e})$	$O(b^{1+c^*/e})$
MiniMax	$O(b^m)$	$O(bm)$
Alpha-Beta Pruning	$O(b^{3m/4})$	$O(bm)$
ExpectiMiniMax	$O(b^d n^m)$	$O(bm)$

b => Branching factor
m => Number of ply
n => Number of distinct outcomes of chance
C* => optimal cost
e => cost of least cost action

Machine Learning

Field of study that gives computers the ability to learn without being explicitly programmed
Instead of coding or developing an algorithm to solve a problem, we use data to have the computer discover a solution for the problem on its own

It performs a function with the data given to it, and gets progressively better at that function.

Supervised Learning	Unsupervised Learning	Reinforcement Learning
Agent learns with a Teacher	Agent must learn without a teacher, tries to find hidden patterns in data	Agent's actions are rewarded or punished and must learn to maximise reward

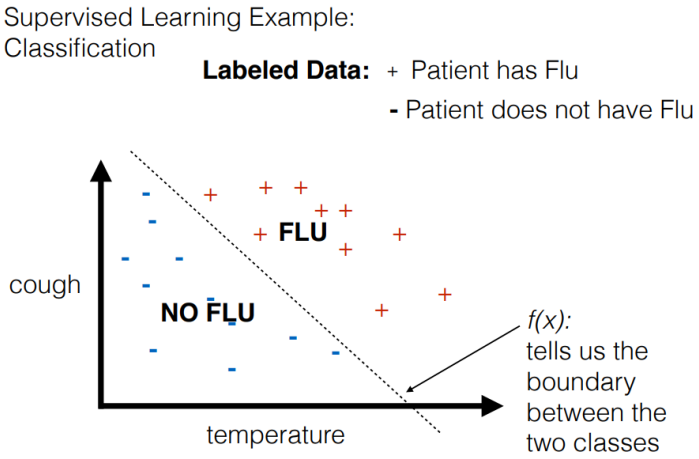
It performs a function with the data given to it, and gets progressively better at that function.

Supervised Learning
Agent is given data which has been labelled by a teacher

(x0, y0), (x1, y1), (x2, y2), …, (xN,yN)
x is data(input)
y is label (output)

We assume that there is some **function** that maps the data (x) to the label (y). If we know what the function is then we can predict the output for a new input. We try to learn f(x) from a **training data set** and evaluate our learning by using a **testing data set**.

Example: Mammals
("cat", TRUE), ("whale", TRUE), ("lizard", FALSE)
(<Honda, 2002, red>, £1200), (<Ford,2007,blue>, £2500)

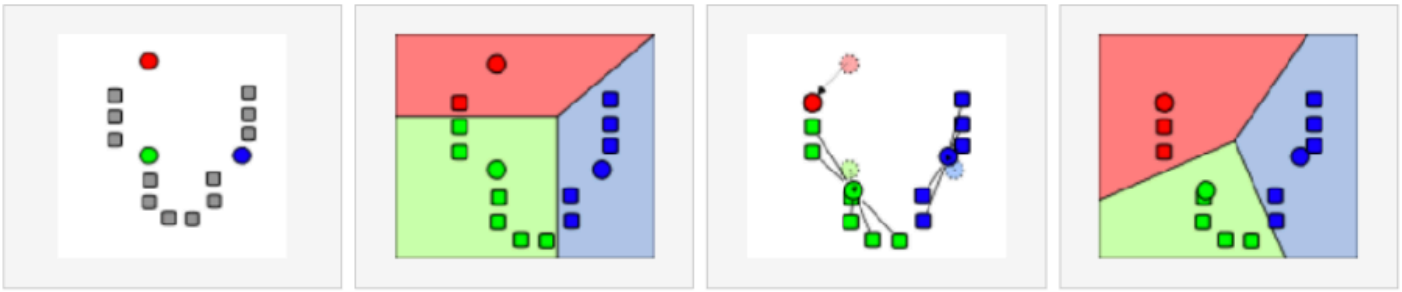


Bias-Variance Tradeoff

Bias: error from modelling assumptions **Variance**: Error from sensitivity to small fluctuations in training data

Unsupervised Learning

Data has no labels. Agent must infer some meaning or structure from the data - computers can often do this better than humans



1. *k* initial "means" (in this case *k*=3) are randomly generated within the data domain (shown in color).

2. *k* clusters are created by associating every observation with the nearest mean. The partitions here represent the Voronoi diagram generated by the means.

3. The centroid of each of the *k* clusters becomes the new mean.

4. Steps 2 and 3 are repeated until convergence has been reached.

Exploration vs Exploitation *Trade off between gaining reward now and searching for more reward in the future*	
Exploration	Exploitation
Tring new things Risky	Going with what you know Safe

Neural Networks

*Computer system modelled on the human brain. Such systems "'learn" (**progressively improve performance** on) tasks by considering examples. They are able to do this **without any prior knowledge** - instead they evolve their own set of characteristics from the material that they process.*

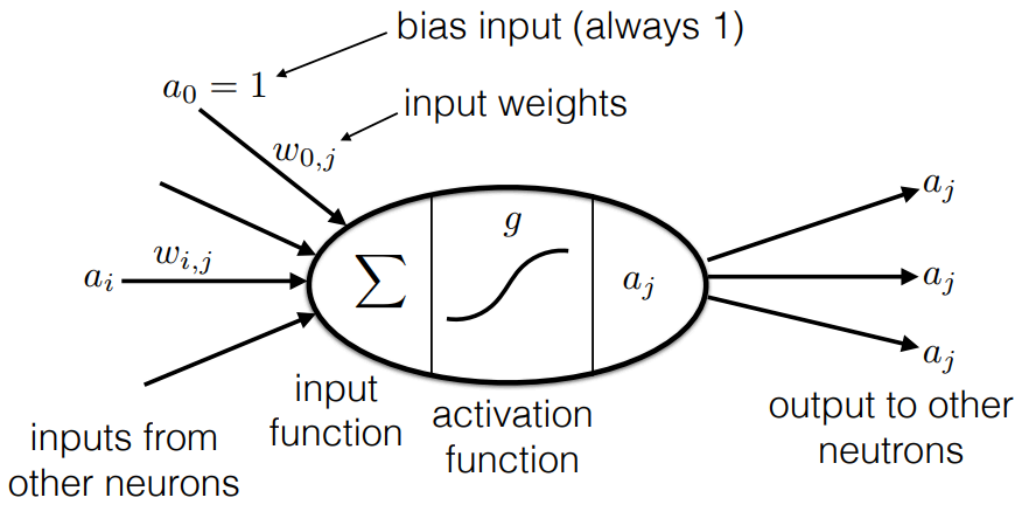
A neural networks is based on a collection of connected nodes - artificial neurons which mirror biological neurons in an animal brain. Each connection - edge, mirrors a biological synapse.

Neuron	Synapse
Something that holds a number between 0 and 1 *A function (takes all the outputs of all the neurons in the previous layer and returns a number)*	A connection between neurons

Neurons and edges have a weight that are adjusted as learning proceeds.A cost function is chosen either because it has desirable properties or because it arises naturally from the problem. The wrights are adjusted during backpropagation. We may not know the cost function ourselves, the program may find a hidden relationship that we didn't previously know was present.

A perception is a neural network composed of one neuron

- We need:
  - Inputs
    - One of which has a bias
  - Weights on bias
    - Represent how much a neuron should take into account a certain input
  - Linear combination of input
  - Gives activation function
    - Check if function g, given the input which is the sum of the weighted inputs is above a specific threshold
    - If so, we send activation to next neuron (or to output)



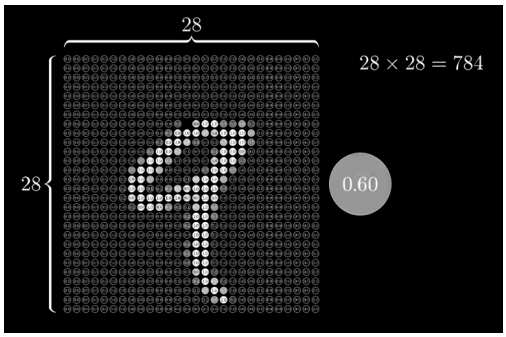
$$a_j = g\left(\sum_{i=0}^{n} w_{i,j} a_i\right)$$

Example: Reading handwritten numbers

Structure

Input Layer
Input image is 28x28 pixels - each pixel corresponds to an individual neuron (**784**)

- Each number holds value corresponding to grayscale value of corresponding pixel - It's activation function
- These take up the **first layer** of the network



Hidden Layers

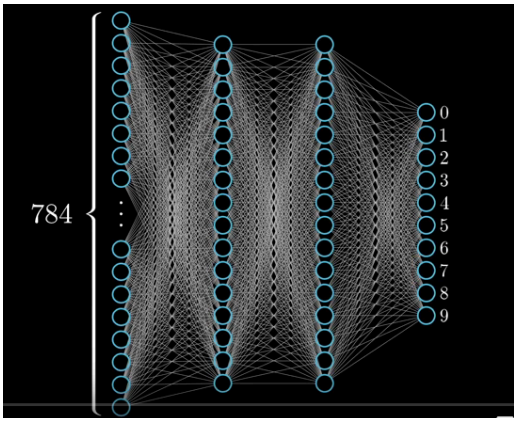
- Can be as many layers as you want
- Can hold as many neurons as you want

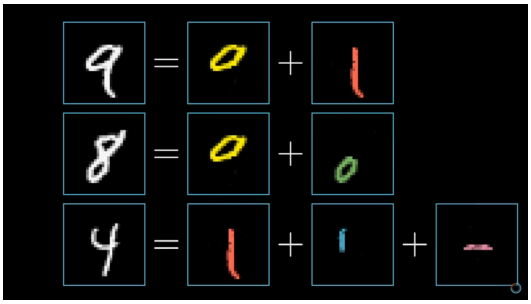
Activations in one layer determine activations in the next.

Computation depends on 'how' activations in one layer bring about activations in the next.

What are we expecting?
When we look at an image, we have priori knowledge: we know that a 9 is a loop on top with a vertical line on the right, we know that an 8 is two loops on top of each other.

- We would hope that each neuron would fire to recognise the patterns.
- The last layer would be a combination of all the various components that can make up a number and the output layer would just put all the combinations together to create the pattern we call each number.
- For example: we would like every time we feed in an image which has a loop on the top that a specific neuron would fire.



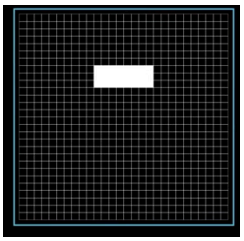


Weights
Example: If we wanted a single neuron in the 2nd layer to recognise the pattern ->. We assign a weight from the first layer to our neuron. We compute the sum of all these weights. In an ideal world all neurons without the pattern in would weigh 0 and all neurons with the pattern would weigh 1.

We want the weight be be between 0 and 1 so we run it through the sigmoid function (very negative inputs become close to 0, very positive inputs become close to 1).

We can also add biases to how high the weighted sum needs to be before the neuron becomes active.

If we have 784 neurons. Each neuron has 784 weights from connections along with the biases. In total, this network has 13,002 weights and biases (knobs and buttons that can be tweaked to change the output).



(w1a1+w2a2+...+wnan) = Weighted Sum

σ(weighted sum) = Sigmoid Function

σ(weighted sum - 10) = Biased

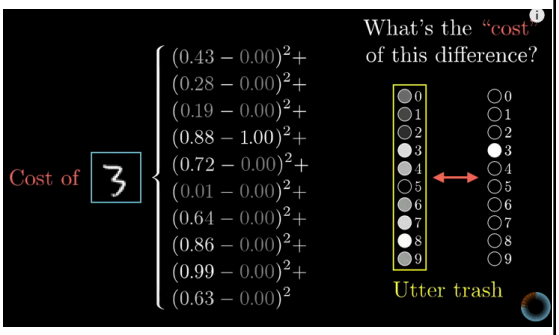
Output Layers
The results we want to achieve:

{0,1,2,3,4,5,6,7,8,9}

We define a cost function

- Give the value that we want as output as 1 and all others to 0
- Add up the squares of the differences between all of the output nodes and the value we have given it
- This value is small when the network confidently classify the image correctly, but is large if it seems like it doesn't know what it's doing

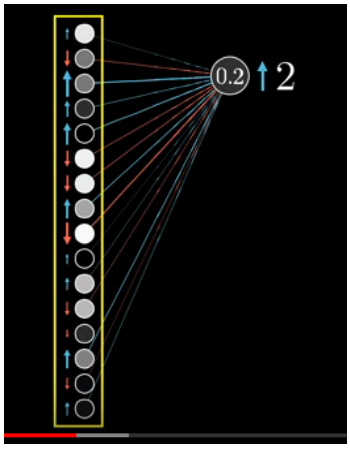
Average all the cost functions of multiple training examples which measures how well the system is performing



Learning
We want to get a lower and lower cost function (better) by incrementally nudging the values of the weights and biases at each neuron. To do this we use backpropagation.

After each instance of training data, we adjust neurons with connections to the given instance of training data and we reduce all others.

We then go back layer-by-layer and adjust the weights and biases. The neurons firing when a certain input is being observed are increased if they are close to 1 and decreased if they are close to 0



Deep Learning

Deep learning (also known as deep structured learning or hierarchical learning) is part of a broader family of machine learning methods based on learning data representations, as opposed to task-specific algorithms. Learning can be supervised, semi-supervised or unsupervised.

Defined by:

- Using a cascade of multiple layers of nonlinear processing units for feature extraction and transformation
    - Each layer uses the output from the previous layer as input
- Learn in supervised and/or unsupervised manners
- Learn multiple levels of representations that correspond to different levels of abstractions; the levels form a hierarchy of concepts

Fully-connected layer (dense): each node is fully connected to all input nodes, each node computes weighted sum of all input nodes. It has one-dimensional structure. It helps to classify input pattern with high-level features extracted by previous layers.

BackPropagation

- Used to train RNNs
- The unfolded network (used during forward pass) is treated as one big feed-forward network
- Accepts the whole time series as input
- The weight updates are computed for each copy in the unfolded network then summed an applied to the RNN weights

Differences between Machine Learning vs Deep Learning
● Machine learning uses algorithms to parse data, learn from that data and make informed decisions based on what it has learned ● Deep learning structures algorithms in layers to create an “artificial neural network” that can learn and make intelligent decisions on its own ● Deep learning is a subfield of machine learning

Example: AlphaGo
Google created a computer program that learned to play the abstract board game called Go, a game known for requiring sharp intellect and intuition. By playing against professional Go players, AlphaGo’s deep learning model learned how to play at a level not seen before in artificial intelligence, and all without being told when it should made a specific move (as it would with a standard machine learning model). It caused quite a stir when AlphaGo defeated multiple world-renowned “masters” of the game; not only could a machine grasp the complex and abstract aspects of the game, it was becoming one of the greatest players of it as well.

Probability Theory

Agents must deal with uncertainty - they cannot precisely predict how the environment will change.

An agent will create a **belief state** - the most likely state of the environment. It then chooses the action that yields the highest **expected utility** averaged over all possible outcomes of the action.

Expected Utility

Represents transactions. If I apply action, I will get some probability. If I want to maximize the utility associated with the state S = 1 - which gives the best response. The outcome is U(s'). I want to compute the likelihood of ending up in this state.

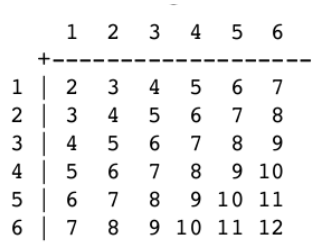
Example:
Should I buy an ice cream for £2?

- 80% chance ice cream is tasty and happiness will increase by 10 — 0.80 x 10 = 8.00
- 19% chance ice cream is minging and I'll lose £2: happiness -5 — 0.19 x -5 = -0.95
- 1% chance I'll get ill: happiness - 100 — 0.01 x -100 = -1.00

Expected Utility = (8.00) + (-0.95) + (-1.00) = **6.05**

Probability Notation						
P(A)	P(A | B)	P(A')	P(A ∧ B)	P(A ∨ B)	P(¬A)	P(A,B)
Probability that A will occur	Probability that A occurs, given that B has occurred	Probability of complement of A	Probability of of A **AND** B	Probability of A **OR** B	Probability that A does not happen	Probability that A and B happen **independently**
	P(A ∧ B) / P (B)	Depends	P(A) * P(B)	P(A) + P(B)	1 - P(A)	P(1 = A ∧ 2 = B)

Probability operations with dice

Two Dice
 
11 possible outcomes:
2,3,4,5,6,7,8,9,10,11,12
	P(2) = 1/36 //only one combination will return a 2
	P(7) = 6/36 = ⅙ //six combinations will return a 7
	
	P(2+3+4+5+6+7+8+9+10+11+12) MUST == 1
	
	P(1,1)
	P(d1 = 1 ∧ d2 = 1) = ⅙ * ⅙ = 1/36

Single Die P(1) = ⅙ P(4) = ⅙ P(2) = ⅙ P(5) = ⅙ P(3) = ⅙ P(6) = ⅙	P(less than four) = P(1) + P(2) + P(3) = 3/6 = ½ P(1 ∨ 2 ∨ 3) = P(1) + P(2) + P(3) = 3/6 = ½

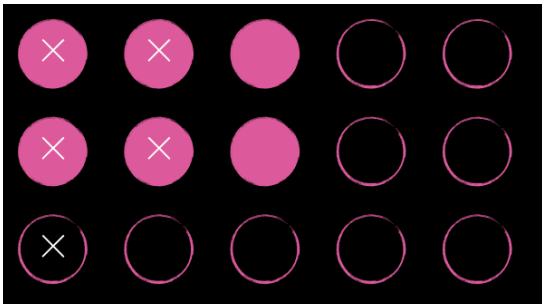
Conditional Probability

Rosie likes 97% of the meals I cook. Matilda likes 3/5ths of the recipes and Molly only likes one meal in 4.
P(no one is happy)?

P(rosie) = 0.97	P(¬rosie) = 0.03
P(matilda) = 0.6	P(¬matilda) = 0.4
P(molly) = 0.25	P(¬molly) = 0.75

P(¬rosie ∧ ¬matilda ∧ ¬molly) = 0.03 * 0.4 * 0.75 = 0.009

15 Lectures; 5 with mistakes; 6 written after a disturbed night



Lectures =

Mistakes =

Disturbed =

P(mistake) = 5/15 = 0.33
P(mistake ∧ disturbed) = 4/15 = 0/27
P(mistake | disturbed = 4/6 = 0.67
P(¬mistake | disturbed) = 1 - P(mistake | disturbed) = 2/6
P(mistake | ¬disturbed) = 1/9

The Product Rule

$$P(a \mid b) = \frac{P(a \wedge b)}{P(b)}$$

$$P(a \wedge b) = P(a \mid b)\, P(b)$$

20% of my recipes contain pasta **and** are liked by Matilda. One quarter of my recipes contain pasta. What is the probability Matilda will like a dish given it contains pasta?
P(m ∧ p) = 0.2
P(p) = 0.25
p(m | p) = P(m ∧ p) / P(p) = 0.2/0.25 = **0.8**

Probability Distribution

The probability of a variable having the given value drawn from a domain

Variable = {value1, value2, … , valueN}

Carbs = {bread,pasta,potato,rice}

P (Carbs = bread) = 0.4
P (Carbs = pasta) = 0.2
P (Carbs = potato) = 0.22
P (Carbs =rice) = 0.18

P(Carbs) = <0.4, 0.2, 0.22, 0.18>

Conditional Probability Distribution

P (x | y) = < P(x0 | y0), … , P(xn | yn) >

P(Matilda | Carbs)
=
<P(m | bread),P(¬m | bread),P(m | pasta),P(¬m | pasta),P(m | potato),P(¬m | potato),P(m | rice),P(¬m | rice)>

Joint Probability Distribution

P (x , y) = < P(x0 ∧ y0), … , P(xn ∧ yn) >

P(Matilda, Carbs)
=
<P(m∧bread),P(¬m∧bread),P(m∧pasta),P(¬m∧pasta),P(m∧potato),P(¬m∧potato),P(m∧rice),P(¬m∧rice>

The Product Rule (Full joint Probability Distribution)

P(Matilda, Carbs) = P(Matilda | Carbs) P(Carbs)

Meal contains pasta
Pasta = {true,false}

Matilda like it
Matilda = {true,false}

	m	**¬m**	
p	p ∧ m	p ∧ ¬m	= P(p)
¬p	¬p ∧ m	¬p ∧ ¬m	= P(¬p)
	= P(m)	= P(¬m)	

P(p) = 0.25
P (m ∧ p) = 0.2
P(m | p) = 0.8

	m	**¬m**	
p	0.2	0.05	0.25
¬p	*0.75 - 0.35 = 0.4*	*7/15 * .75 = 0.35*	**0.75**

Bayes' Rule

$$P(b \mid a) = \frac{P(a \mid b)\, P(b)}{P(a)}$$

==

$$P(cause \mid effect) = \frac{P(effect \mid cause)\, P(cause)}{P(effect)}$$

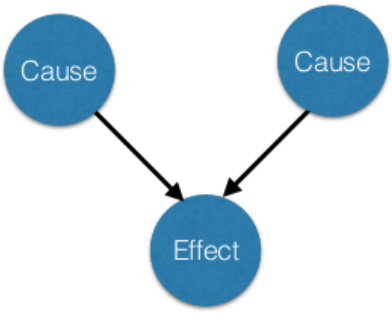
Example:
P(meningitis | stiff neck) = 1/700
Data says: P(meningitis) = 1/50,000 ; P(stiff neck) = 1/100 ; P(stuffneck | meningitis) = 7/10

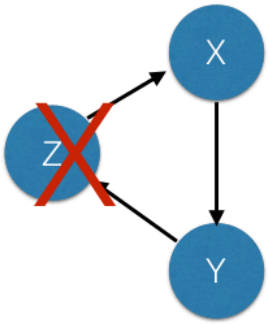
OR

Patient has stiff neck -> what is it the chance of it being meningitis?
Probability of meningitis in population is 1/50000
Probability of a stiff neck is 1/100
Probability of patients with meningitis getting a stiff neck is 7/10
Bayes Rule: (7/10) * (1/50000) / (1/100) = 1/700

Belief Network (Bayesian Network)

The structure of dependence between different variables and the relationship between them





Cycles are not allowed (this would violate the strict cause/effect relationships we define)

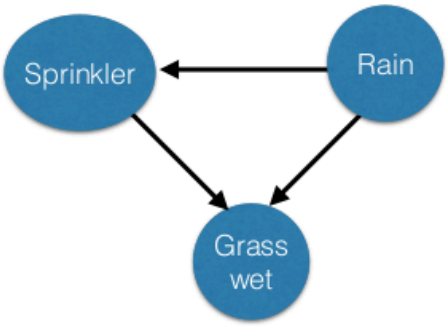
Example

Effect: Grass is wet

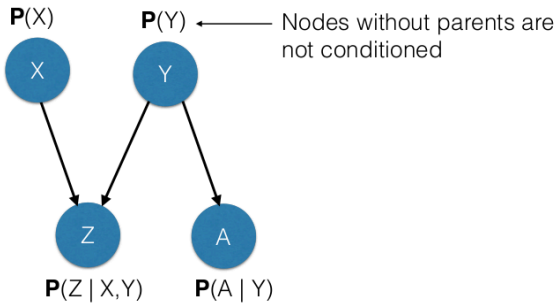
Cause: Sprinkler

Cause: Rain

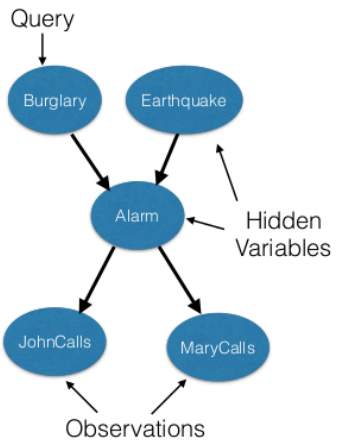
//If it rains, sprinkler will not be on



Each node has a conditional probability distribution dependant on its parents: P(X | Parents(X))



Suppose both Mary and John call, what is the probability of a burglary? $P(b|m,j)$



Query Variables: what we are trying to find out

Observation Variables: what we know

Hidden Variables: what we don't know (and don't care to know)

Markov Chain

*A belief network representing **sequences** of values, or state transitions **over time**. A mathematical system that hop from one state to another. In addition to the state space, a Markov chain tells you the probability for hopping from one state to any other.*

Markov Assumption (first order)

P(Si+1 | S0,...,Si) = P(Si+1|Si)

Si conveys all the information about the history that can affect future states. The future is conditionally independent of the past given the present.

The state at time i+1 only depends on the state at time i

Markov Assumption (second order)

P(Si+1 | S0,...,Si) = P(Si+1 | Si,Si-1)

Si+1 is affected by both the result of Si and Si-1

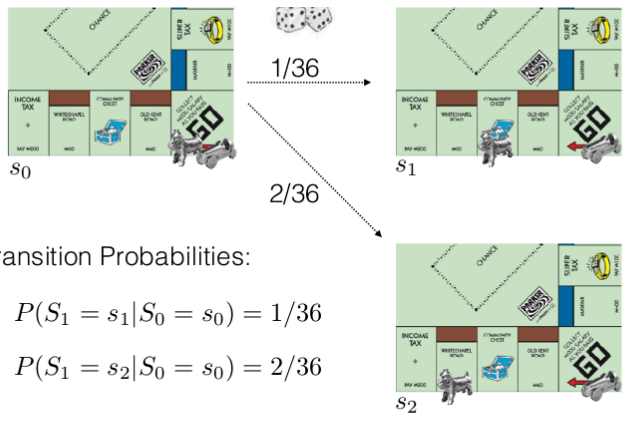
The state at time i+1 depends on both i and i-1

Example: What is the weather tomorrow?
Si = {sunny, rian}

P(S i+1 =sunny | S i =sunny) = 0.9
P(S i+1 =rainy | S i =sunny) = 0.1
P(S i+1 =sunny | S i =rainy) = 0.5
P(S i+1 =rainy | S i =rainy) = 0.5

P(S -> S -> R -> R) = Initial state -> 0.9 -> 0.1 -> 0.5

The state at time i+1 only depends on the state at time i.
It does not matter how you got to i.



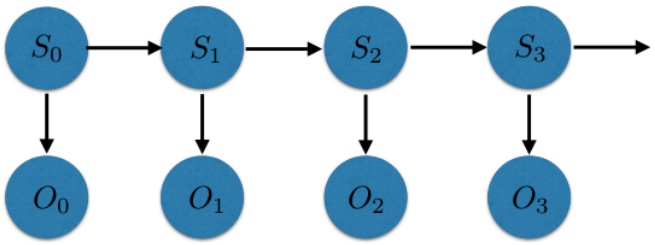
Transition Probabilities:

$$P(S_1 = s_1 | S_0 = s_0) = 1/36$$

$$P(S_1 = s_2 | S_0 = s_0) = 2/36$$

Hidden Markov Model

Markov Model in a partially observable environment



We need:
P(S0) -> initial conditions
P(Si+1 | Si) -> transition probabilities
P(Oi | Si) -> sensor model

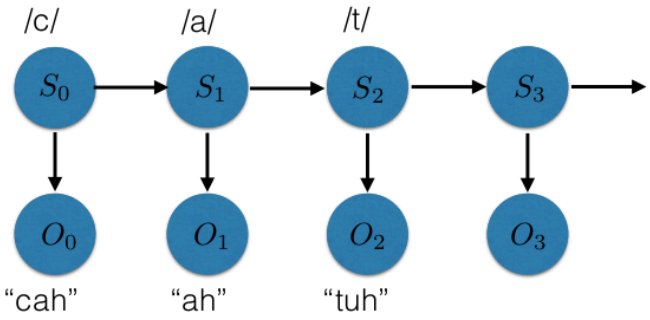
State Estimation

To determine which state we are in, we can use the sequence of past observations
P(Si | O0, … ,Oi)

Questions we can ask:

- **Filtering**
  - What is the current state given all evidence to date?
  - P(Si | O0, … ,Oi)
- **Prediction**
  - What will the state be in the future, given all evidence to date
  - P(Si + k | O0, … ,Oi)
- **Smoothing**
  - What was the state sometime in the past, given all evidence to date?
  - P(Sk | O0, … ,Oi)
- **Most likely explanation**
  - Given a sequence of observations, what is the most likely sequence of states that can explain the observations?

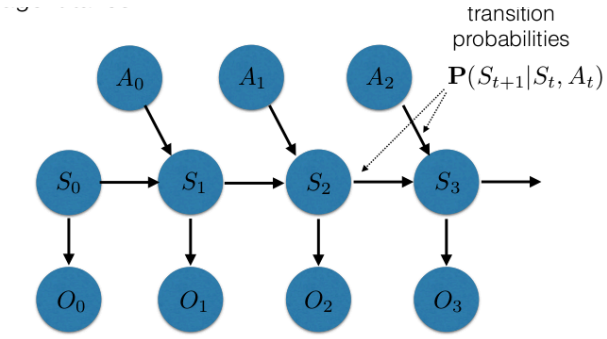
States are *phonemes* (isolated units of sound that make up all words):



Observations are sounds from a microphone

Hidden Markov Model with Actions

Markov Model in a partially observable environment where an agent can influence the environment



State t+1 is dependant on both State t AND Action t

Example: A robot is in a corridor but doesn't know where it is.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

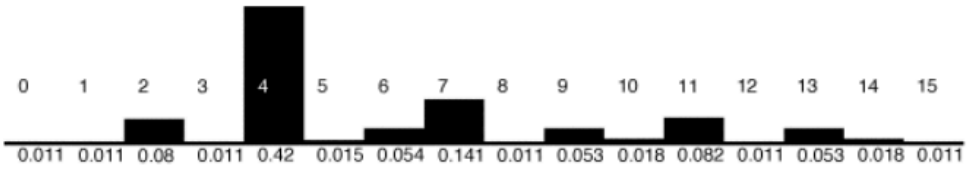
If robots sensors are noisy:

- P(Obs = door | atDoor) = 0.8
- P(Obs = door | notAtDoor) = 0.1

Robot doesn't move very well either:

- P(Loc t+1 = L | Act t =goRight, Loc t =L) = 0.1
- P(Loc t+1 = L+1 | Act t =goRight, Loc t =L) = 0.8
- P(Loc t+1 = L+2 | Act t =goRight, Loc t =L) = 0.074
- P(Loc t+1 = any other L | Act t =goRight, Loc t =L) = 0.002

We can perform a probability distribution over locations, given the action, observation sequence.



How should an agent choose which action to take?

A rational agent tries to **maximise some performance criteria**. However, since we are now considering that actions have **probabilistic outcomes**, the agent should choose the action with the greatest **expected outcome**.

The agent has a Utility Function that depends on the state

U(s)

Example

GOAL U = 100	U = -1	**TRAP** U = -100
U = -1	U = -1	U = -1
U = -1	U = -1	U = -1

We assign goal state a high utility and we may assign other states a small negative utility to encourage the agent to find the goal as fast a possible

The agent has a utility function that depends on the state

- U (s)

Every action gives a probability of transitioning to a certain state:

- P(St +1 | St, At)

If the world is partially observable, the agent needs to perform a sequence of observations

- O0:t = O0, O1, O2, … , Ot

The probability of the next state is dependent on observations

- P (St+1 | O0:t, At)

We do not not know for certain which state a given action will take us, but we can estimate the **Expected Utility** of an action given a sequence of observations:

$$EU(a|o_{0:t}) = \sum_{s'} P(S_{t+1} = s'|o_{0:t}, a)U(s')$$

- and add up all the results
- look at all possible outcomes of that action (all possible states *s'* that the agent may end up in)
- given a sequence of observations and a particular action
- for each outcome, multiply by the utility of that outcome

Calculating EU (fully observable)

$$EU(a|s) = \sum_{s'} P(s'|s, a)U(s')$$

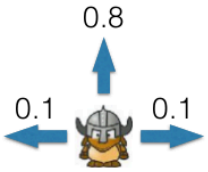
GOAL U = 100	***HERO*** U = -1	**TRAP** U = -100
U = -1	U = -1	U = -1
U = -1	U = -1	U = -1

*Hero is in state S**

What is the expected utility of moving left?
P(St+1 = goal | St = S*, left) = 0.7
P(St+1 = monster | St = S*, left) = 0.1
P(Pt+1 = s* | St = s*, left) = 0.2

EU (left | S*)
=
(0.7 x 100) + (0.1 x -100) + (0.2 x -1)
=
59.8

Calculating Route using EU



80% chance agent moves in chosen direction
10% for either side

The agent wishes to maximise it's expected utility -> find the state sequence to maximise the expected sum of rewards

U([s0, s1, …, sn]) = R(s0) + R(s1) + … + R(sn)

In this instance the agent would go Up, Up, Right, Right, Right.

However if the agent started 2 places to the right, it would still be better to go the long way around rather than risking accidentally bumping into the monster. Although if it accidentally goes up (next to monster) then it is better to keep moving up rather than going back down.

A Policy is more powerful than a Plan

A plan may tell the agent what sequence of actions to take
A policy tells the agent what to do in every possible state

- Obviously, we would like to be able to compute optimal policies for every problem.
- But for large state spaces it is intractable to find the optimal action for *every* state
- If an agent encounters an unknown state, perhaps it can use knowledge from past experiences.
    - For example, perhaps the unknown state is similar enough to a previously encountered state
    - If you've never driven a Ferrari before, you try to apply your experiences from driving a Fiat Punto
- Otherwise you will have to *learn* how to act when in this new state

Instead of summing the rewards of each state

$$U([s_0, s_1, s_2, \ldots]) = R(s_0) + R(s_1) + R(s_2) + \ldots$$

which can possibly be infinite

We can use **Discounted** rewards where we use a discount factor: $0 < \gamma < 1$

$$U([s_0, s_1, s_2, \ldots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots$$

gives the agent some preference for current rewards over future rewards

- Low discount factor => "live in the moment"
- High discount factor => plan for the future

$$U([s_0, s_1, s_2, \ldots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots$$

$$= \sum_{t=1}^{\infty} \gamma^t R(s_t)$$

Finite Horizon	Infinite Horizon
If the agent has a fixed amount of time ● A maximum of N steps to complete a task	There is no time limit
We can get away with simple rewards, as utility will not go to infinity	We need discounted rewards (unless we can guarantee the agent will reach a terminal state)
The optimal policy is nonstationary (time dependant)	For infinite horizon problems, and discounted rewards, the optimal policy is stationary (time independent) and furthermore independent of the starting state

How to calculate U(s)

$$U(s) = R(s) + \gamma \max_a \sum_{s'} P(s'|s,a)U(s')$$

for a given state collect your reward

and assume you will be taking the action that is most likely to move you to a new state with the highest value

Agent will reach the new state *s'* one step in the future, so we should discount its value

But in order to compute *U(s)*, I need to know *U(s)*!
(at least the *U(s)* for every state the agent could possibly transition to from *s*)

IntroA.I.C.Zito

Reinforcement Learning

1. Start with a Fixed Policy
2. Try the Policy -> run a number of trials
3. Improve your model of the environment based on the experience you collect from trials
4. Update your policy and repeat

However, agent may never explore new routes and just improve the same route -> may not be optimal

- To encourage exploration: put artificial rewards in states the agent has not visited often

Model-Free Reinforcement Learning

The agent tries to learn U(s) without learning explicit models of transition probabilities.

Temporal Difference learning rule:
Whenever a transition occurs from s to s', we update U(s):

$$U_{i+1}(s) = U_i(s) + \alpha(R(s) + \gamma U_i(s') - U_i(s))$$

learning rate: a small number (<1) that controls how fast U(s) changes in each iteration

The algorithm converges, when this underlined term is zero: the change of Utility from moving from state *s* to *s'* equals the instant reward *R(s)*

U(s) is slightly updated, based on change in utility to the next state no need to learn transition probabilities!