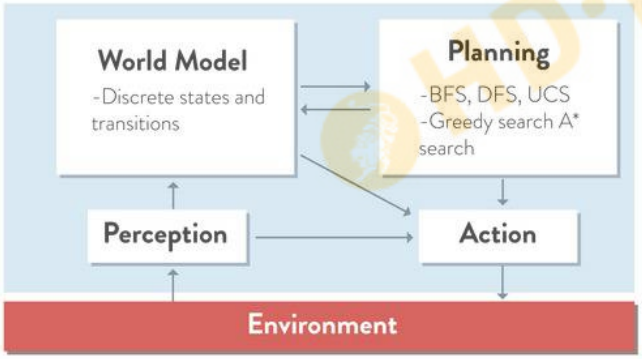
Week 3

Path Search Agents



- **Reactive and Model-Based Agents** choose their actions based only on what they currently perceive, or have perceived in the past.
- A **Planning Agent** is different because it can use Search techniques to plan several steps ahead in order to achieve its goal(s).

Path Search Strategies

- Path search strategies can be classified as either Uniformed or Informed.
- **Uninformed Search Strategies** can only distinguish goal states from non-goal states.
    - Strategies are distinguished by the order in which the nodes are expanded.
    - Eg. Breadth-First, uniform cost, depth first, iterative deepening, bidirectional
- **Informed Search Strategies** use heuristics to try to get "closer" to the goal.
    - More efficient than uninformed
    - Systemically generates new states, and tests them against the goal
    - Eg. Greedy, A*

Problem solving

Formulating the Problem

The first step in solving a path search problem is to decide on a World Model which captures only those aspects of the domain that are relevant for solving the problem, and "abstracts away" any extraneous details.

Single-State Task Specification

For the tasks in week 3 and 4, an appropriate World Model is a State Space, with a set of states, a set of allowable transitions between states, an initial state, one or more goal states, and, in some cases, a cost associated with each transition.
A solution is a sequence of transitions that will take us from the initial state to a goal state.

For example, the Romania Task can be specified as follows.

- state space = set of cities on the map (ie. other cities)
- initial state = "at Arad"
- actions (or operators) are transitions between directly connected cities, e.g. Arad → Zerind, Arad → Sibiu, etc.
- Goal test. In this case, there is only one goal specified ("at Bucharest").
- path cost = total driving distance along the roads in the path

Abstracting the Problem

The state space needs to be abstracted to lower the complexity for problem solving. Let's look at the terminology associated with problem abstraction.

- (Abstract) State: A set of real states.
- (Abstract) Action: A complex combination of real actions.
    - e.g. "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.
- For guaranteed realizability, any real state "in Arad" must get you to some real state "in Zerind"
- (Abstract) Solution: A set of real paths that are solutions in the real world.

Some example problems include:

Toy Problems: Toy problems have a concise, exact description.
Real World Problems: These do not have a single agreed description

Robotic Assembly

- Path search can also be applied to real-world problems like robotic assembly, where the states and transitions are continuous and the goal is determined by specifying a list of conditions that need to be satisfied.
- States: The real-valued coordinates of robot joint angles and parts of the object to be assembled.
- Operators: The continuous motions of joint angles.
- Goal Test: The complete assembly of object.
- Path Cost: The time to execute.

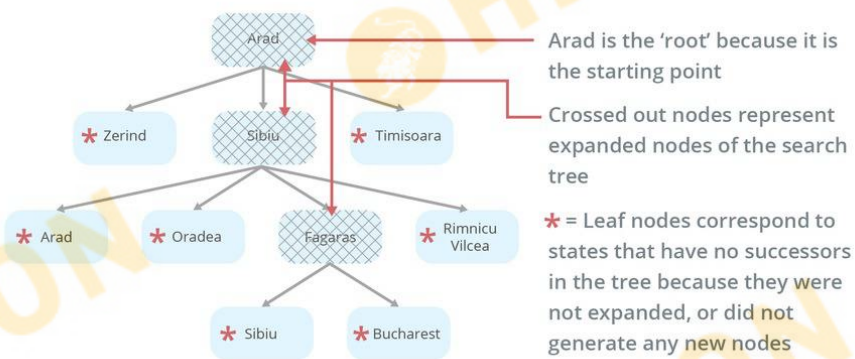
Path Search Algorithms

- The path search algorithms discussed follow the same basic pattern of tree search.
- At each step in the search, **one of the leaf nodes** in the search tree is **expanded**, and **all possible transitions** from that node are **explored**, in order to generate new leaves for the tree which are children of the expanded node.
- The search mechanism maintains a priority queue of leaf nodes which have been generated but not yet expanded.
    - The nodes in this priority queue are sometimes called the "Frontier".
- The various **algorithms differ only in the way they choose the next node for expansion**, from the nodes in this priority queue.

The structure of the algorithms is as follows:

1. Start with a priority queue consisting of just the initial state.
2. Choose a state from the queue of states which have been generated but not yet expanded.
3. Check if the selected state is a Goal State. If it is, STOP (solution has been found).
4. Otherwise, expand the chosen state by applying all possible transitions and generating all its children.
5. If the queue is empty, Stop (no solution exists).
6. Otherwise, go back to Step 2

Eg.



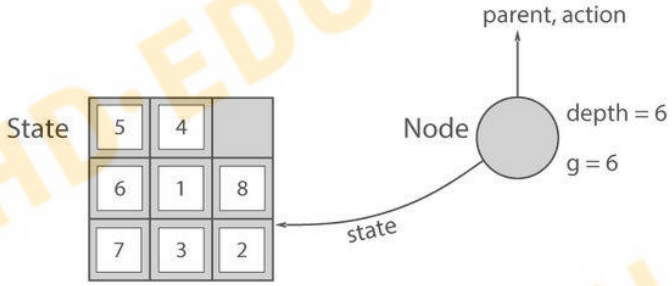
Data Structures for a Node

- Nodes are not the same as states. Some states may appear multiple times in the tree, while others might not (yet) occur at all.
- We can implement data structures that are specific to a particular search problem. (For example, if the state space is a 2-dimensional grid, it makes sense for the nodes to also be stored in a 2-dimensional array).
- However, If the search space has a specific structure, one possibility is to have a node data structure with five components:
    1. Corresponding state
    2. Parent node: the node which generated the current node.
    3. Operator that was applied to generate the current node.
    4. Depth: number of nodes from the root to the current node.
    5. Path cost (from the root to the current node)

States vs. Nodes

A State - (A representation of) a physical configuration
A Node - A data structure constituting part of a search tree includes parent, children, depth, path cost g(x)
States DO NOT have parents, children, depth, or path cost!

- Note: Two different nodes can contain the same state.



Data Structures for Search Trees

- **Frontier**: A collection of nodes waiting to be expanded.
- It can be implemented as a priority queue with the following operations:
    - MAKE-QUEUE (ITEMS) creates queue with given items. Boolean
    - EMPTY (QUEUE) returns TRUE if no items in queue.
    - REMOVE-FRONT (QUEUE) removes the item at the front of the queue and returns it.
    - QUEUEING-FUNCTION (ITEMS, QUEUE) inserts new items into the queue.

Comparing Search Strategies

- Strategies are defined picking the order of nodes expansion.
- Strategies are evaluated along the following dimensions:
    - **Completeness**
        - Does it always find a solution if one exists?
    - **Time Complexity**
        - The number of nodes generated/expanded
    - **Space Complexity**
        - The maximum number of nodes in memory
    - **Optimality**
        - Does it always find a least-cost solution?
- Time complexity and space complexity are measured in terms of:
    - b – **maximum branching factor** of the search tree
    - d – **depth** of the least-cost solution (shortest path)
    - m – **maximum depth** of the state space (this may be ∞, eg numbers of cities in a graph)

Algorithm Comparison

- There are two approaches to comparing algorithms:
- **Benchmarking**: Run both algorithms on a computer and measure the speed of each.
- **Analysis of Algorithms**: Perform a mathematical analysis of the algorithms.

Benchmarking

- Run two algorithms on a computer and measure speed.
- Depends on implementation, compiler, computer, data, network ...
- Measuring time
- Processor cycles
- Counting operations
- Statistical comparison, confidence intervals

Analysis of Algorithms

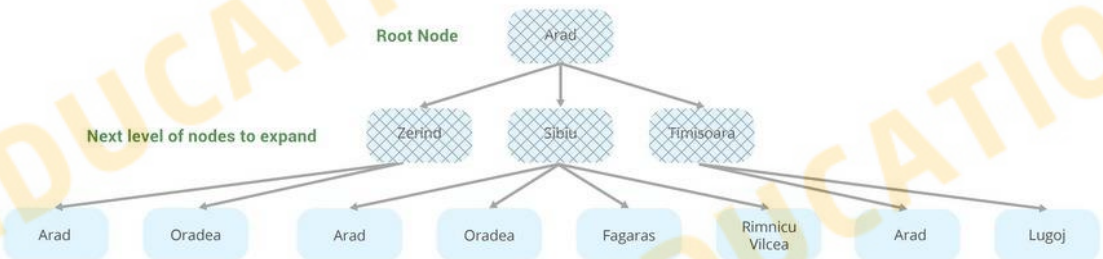
- T(n) is O(f(n)) means $\exists n_0, k : \forall n > n_0 T(n) \leq kf(n)$
    - There exists number n0 and k such that when n > n0, T(n) <= kf(n)

- n = input size
- T(n) = total number of step of the algorithm

- Independent of the implementation, compiler, ...
- **Asymptotic analysis**: For large n, an O(n) algorithm is better than an $O(n^2)$algorithm.
- O() abstracts over constant factors
  - e.g. $T(100n + 1000)$ is better than $T(n^2+1)$ only for $n > 10$.
- O() notation is a good compromise between precision and ease of analysis.

Breadth-First Search

- All nodes are expanded at a given depth in the tree before any nodes at the next level are expanded; so it prioritizes breadth over depth, hence the name 'breadth first'.
- It is very systematic and it finds the shallowest goal first.
- Its sequence is:
  1. Expand the root node first
  2. expand all nodes generated by the root node
  3. expand all nodes generated by those nodes, and so on.



- Implementation: **Queueing Function** = put newly generated successor nodes at end of queue
- Best used on graphs where branching factor is low
  - Searches equally 'everywhere', so explores every possible move
- Discard repeated nodes (note how arad appears multiple times when its already the start point) - Use dijkstra's algorithm

Properties of BFS:

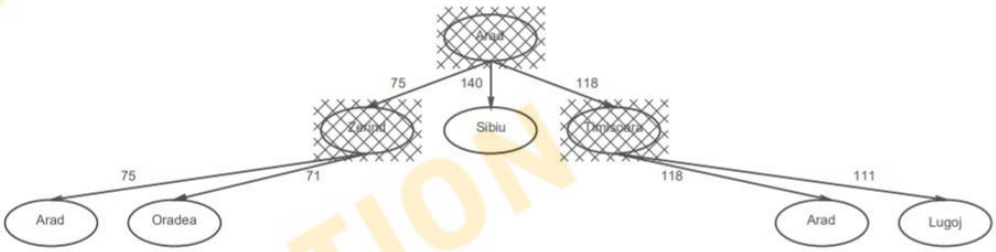
- **Completeness** (Guaranteed to find a path to the goal):
  - Yes. The shallowest goal must be at some fixed depth d . It will be found before any deeper nodes are expanded (assuming b is finite).
- **Time Complexity:**
  - The algorithm will expand the root node, then at most b nodes at depth 1, then at most b^2 nodes at depth 2, b^3 nodes at depth 3, etc. (where b is the branching factor). In the worst case, the goal node could be the last node generated at depth d. If we check each node at the time it is generated (to see if it is a goal state), the maximum number of nodes generated (assuming $b > 1$) would be:

    $$1 + b + b^2 + \ldots + b^d = \frac{(b^{d+1} - 1)}{b - 1} = O(b^d)$$

    Note: In some formulations of BFS, the test for a goal state only occurs when a node is expanded, rather than when it is generated. In that case, almost all the nodes at depth d+1 would be generated before we discover that the goal occurred at depth d, so the complexity would be $O(b^{d+1})$.
- **Space Complexity:**
  - All generated nodes are kept in memory, so it's $O(b^d)$
  - If the algorithm continues and the goal node is expanded then it'll be $O(b^{d+1})$
  - Weakness with BFS, memory intensive
- **Optimality** (Does it find the shortest path):
  - Yes, as long as all actions have the same cost

Uniform Cost Search

- We always **expand the node whose total path length from the start node is shortest**.
    - For each node, it keeps a track of what the path distance is from the start state to that node, and expands the node with the shortest path length
- This reduces to Breadth First Search when all actions have the same cost.
- UCS finds the cheapest goal, provided path cost is monotonically increasing along each path (i.e. no negative-cost steps).
    - Ie. BFS finds shortest number of steps (lower amount of cities), but UCS will find the shortest distance



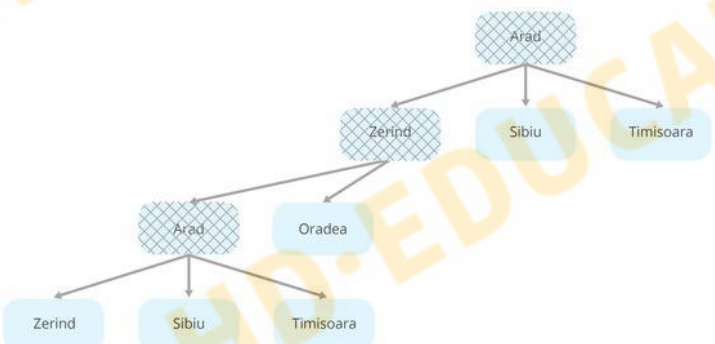
- Implementation: **Queueing Function** = insert nodes in order of increasing path cost.
- Note (for students familiar with advanced data structures): When the number of nodes is large, a more efficient implementation may be to store the nodes in a heap rather than a list. In that case, the nodes would not literally be stored in increasing order while they are in the priority queue; but, when they are removed from the queue, they are guaranteed to come out in the correct order.

Properties of Uniform-Cost Search

- **Completeness** (Guaranteed to find a path to the goal):
    - Yes, if b is finite and step cost $\geq \varepsilon$ with $\varepsilon > 0$
- **Time Complexity:**
    - $O(b^{\lceil C^* / \varepsilon \rceil})$ where C^* = cost of optimal solution, and assume every action costs at least ε
- **Space Complexity:**
    - $(b^{\lceil C^* / \varepsilon \rceil})$ ($b^{\lceil C^* / \varepsilon \rceil} = b^d$ if all step costs are equal)
- **Optimality** (Does it find the shortest path):
    - Yes

Depth-First Search

- Newly generated nodes are added to the front of the queue (thus making it act as a stack).
- Need to be careful about removing repeated nodes, due to infinite loops
- Always expand the deepest unexpanded node
    - Eg. explore the graph as if we were physically moving around in Romania, backtracking when we reach a dead-end.



Implementation:
QUEUEING FUNCTION = insert newly generated states at the front of the queue. It can alternatively be implemented by recursive function calls.

Properties of Depth-First Search

- **Completeness** (Guaranteed to find a path to the goal):
  - No. It can fail in infinite-depth spaces, or spaces with loops. If we modify the algorithm to avoid repeated states along a path, it becomes complete in finite spaces.
- **Time Complexity:**
  - O(b^m) (terrible if m is much larger than d but if solutions are dense, may be much faster than breadth-first)
- **Space Complexity:**
  - O(bm), i.e. linear space!
  - Don't need to keep all the nodes/paths in memory, only the one being currently explored and it's children and paths followed
- **Optimality** (Does it find the shortest path):
  - No, can find suboptimal solutions first

Depth-Limited Search

- Depth-Limited Search is a variant which expands nodes like Depth-First Search but imposes a cutoff on the maximum depth of the path.
- The problem with Depth-Limited Search is: How do you pick a good limit?
  - Iterative Deepening Search

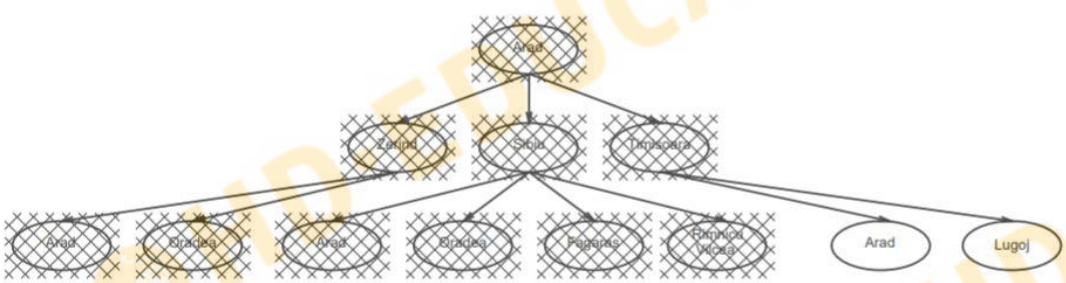
Properties of Depth-Limited Search

- **Completeness** (Guaranteed to find a path to the goal):
  - Yes, no infinite loops
- **Time Complexity:**
  - O(bk) where k is the depth limit
- **Space Complexity:**
  - O(bk), linear space
- **Optimality** (Does it find the shortest path):
  - No, can find suboptimal solutions first

Iterative Deepening Search

- Breadth-First Search (BFS) is complete and optimal, but uses a lot of memory.
- Depth-First Search (DFS) is neither complete nor optimal, but has the advantages of using very little memory.
- Does a series of depth limited searches
- Tries to combine the benefits of depth-first (low memory) and breadth-first (optimal and complete) by doing a series of depth-limited searches to depth 1, 2, 3, etc
  - Early states will be expanded multiple times, but that might not matter too much because most of the nodes are near the leaves.
- The nodes at depth d only get generated once, only previous depths get generated multiple times

Here is an example of IDS with branching factor b = 2, depth d = 3.



Properties of Iterative Deepening Search

- **Completeness** (Guaranteed to find a path to the goal):
  - Yes
- **Time Complexity:**
  - Nodes at the bottom level are expanded once, nodes at the next level twice, and so on:
  - depth-limited: $1 + b^1 + b^2 + ... + b^{d-1} + b^d = O(b^d)$
  - iterative deepening: $(d + 1)b^0 + db^1 + (d - 1)b^2 + ... + 2 \cdot b^{d-1} + 1 \cdot b^d = O(b^d)$
  - example b = 10, d = 5 :
    - depth-limited: 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111
    - iterative-deepening: 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456
    - The root node is generated 6 times, the 10 nodes at depth 1 are generated 5 times, the 100 nodes at depth 2 are generated 4 times, etc., until the 100000 nodes at depth 5 are generated once each.
    - only about 11% more nodes (for b = 10) while IDS only stores 50 nodes vs 100k for BFS
- **Space Complexity:**
  - O(bd), linear space
  - Previously explored nodes are deleted, only the current path is saved in memory
- **Optimality** (Does it find the shortest path):
  - Yes, if step costs are identical

Week 4

Search Strategies

- Recall that all the search algorithms we consider have the same basic structure:
    1. Start with a priority queue consisting of just the initial state.
    2. Choose a state from the queue of states which have been generated but not yet expanded.
    3. Check if the selected state is a Goal State. If it is, STOP (solution has been found).
    4. Otherwise, expand the chosen state by applying all possible transitions and generating all its children.
    5. If the queue is empty, Stop (no solution exists).
    6. Otherwise, go back to Step 2
- Search strategies are distinguished by the order in which new nodes are added to the queue of nodes awaiting expansion.

Best-First Search

- BFS and DFS treat all new nodes the same way:
    - BFS add all new nodes to the back of the queue
    - DFS add all new nodes to the front of the queue
- Best-First Search is a general term for more sophisticated algorithms which use an evaluation function to try to guess which would be the best node to expand next.
- Normally, there is a function f(n) which assigns a number to every previously generated node n. The node n chosen to be expanded next is the one with the smallest value of f(n).
    - TL;DR Best First Search uses an evaluation function f() to order the nodes in the queue; we have seen one example of this:
        - UCS f(n) = cost g(n) of path from root to node n
- This method is "Seemingly" Best-First Search, because we don't know for sure that this node is the best one, we are just choosing what seems to be the best node, based on the information currently available
- Informed or Heuristic search strategies rely on a heuristic function h(n) which estimates the cost of getting to the goal from node n
    - Greedy Search f(n) = estimate h(n) of cost from node n to goal
    - A∗ Search f(n) = g(n) + h(n) (UCS and greedy combination)

Bidirectional Search

- Idea: Search both forward from the initial state and backward from the goal, and stop when the two searches meet in the middle.
- We need an efficient way to check if a new node already appears in the other half of the search. The complexity analysis assumes this can be done in constant time, using a Hash Table.
- Assume branching factor = b in both directions and that there is a solution at depth = d. Then bidirectional search finds a solution in $O(2b^{d/2}) = O(b^{d/2})$ time steps.

Issues with BDS

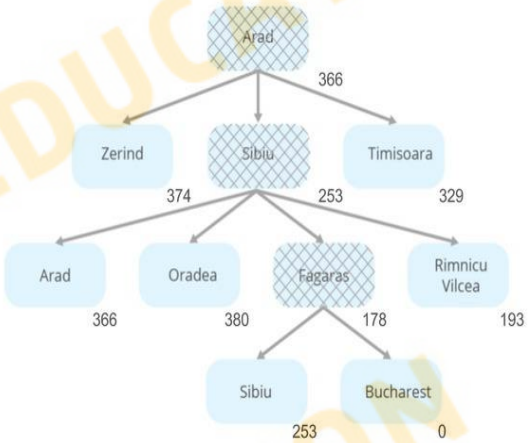
- Searching backwards means generating predecessors starting from the goal, which may be difficult
- there can be several goals – e.g. chekmate positions in chess
- space complexity: $O(b^{d/2})$ because the nodes of at least one half must be kept in memory.

Greedy Search

A Best-First Search that selects the next node for expansion using the heuristic function for its evaluation: f(n) = h(n)

- Assume h(n) = 0 if n is a goal state,
- Ie, greedy search minimises the estimated cost to the goal and expands which ever node is estimated to be closest to the goal

- Greedy because it only considers the next best move for its current situation without worrying about overall solution, ie. estimates shortest path to goal from current node



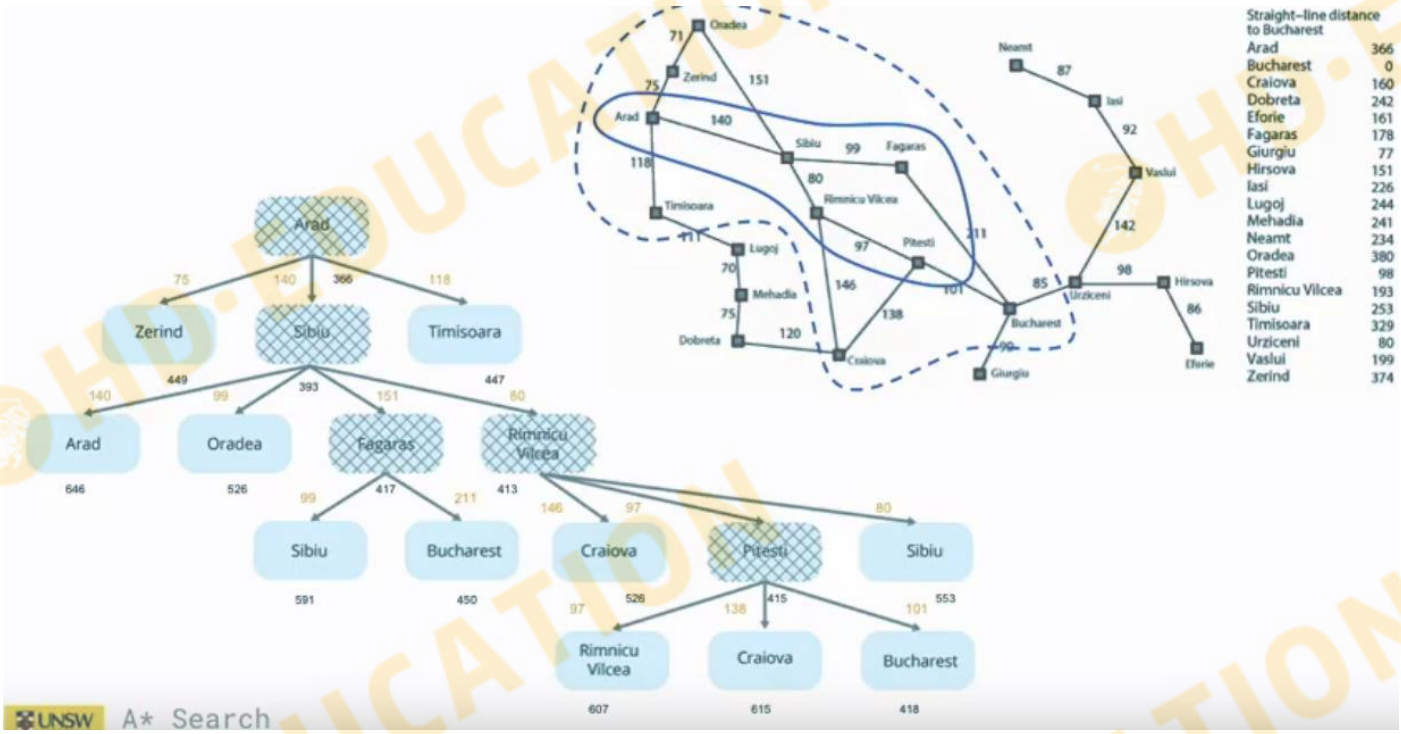
- Euclidean distance as a heuristic Romania example
  - hSLD(n) = straight-line distance between n and the goal location (Bucharest).
  - Assume that roads typically tend to approximate the direct connection between two cities.
  - Need to know the map coordinates of the cities:
    - (Sibiux −Bucharestx)2 + (Sibiuy −Bucharesty) 2

Properties of Greedy Search

- **Completeness** (Guaranteed to find a path to the goal):
  - No. Can get stuck in loops.
  - Complete in finite space with repeated state checking
- **Time Complexity:**
  - O(b^m) where m is the maximum depth in the search space
- **Space Complexity:**
  - O(b^m),retains all nodes in memory
- **Optimality** (Does it find the shortest path):
  - No.
- Greedy search has the same deficits as DFS. Although, a good heuristic can reduce time and memory significantly

A* Search

- A* Search is a combination of Greedy Search and Uniform Cost Search.
- A ∗ Search uses evaluation function f(n) = g(n) + h(n)
  - g(n) = cost from initial node to node n
  - h(n) = estimated cost of cheapest path from n to goal
  - f(n) = estimated total cost of cheapest solution through node n
- Greedy Search minimizes h(n)
  - efficient but not optimal or complete
- Uniform Cost Search minimizes g(n)
  - optimal and complete but not efficient
- A* Search minimizes f(n) = g(n) + h(n)
  - idea: preserve efficiency of Greedy Search but avoid expanding paths that are already expensive
- A ∗ Search is optimal and complete provided h() is admissible in the sense that it never overestimates the cost to reach the goal.



For the above example,

- Sibiru to Bucharest's straight line distance is shortest compared to Zerind or Timisoara. We add '140' to that distance to make 393 in total. (Sibiu = 253+140, Zerind = 374+75, Timisoara = 329+118).
- Then, from Sibiru it picks Rimnicu Vilcea as Rimnicu's distance of 193 + 80 (dist to Sibiu) + 140 (dist from Sibiu to start) is smaller than Fagaras at 413, which is 178 + 99 + 140 (417)
- Then pitesti is chosen as Ptesti is 98 from goal + 96 (Rimnicu) + 80 + 140 which is 415 vs Craiova (526) or Sibiu (553)
- Then, Bucharest has a path length of 0, meaning that it's value of 418 is the entire path length (101 + 97 + 80 + 140)
- However, Faragaras at the Sibiu choice has a smaller path distance at 417 so we go back there to see if there's a smaller path to the goal there.
- When expanded, the path to Bucharest from Fagaras is at 450 which is more expensive. As all the other estimates are smaller than 418, the algorithm can stop and return the path to the goal.

Heuristics

- Heuristic h() is called admissible if $\forall n\ h(n) \leq h*(n)$ where $h*(n)$ is true cost from n to goal
- If h is admissible then f(n) never overestimates the actual cost of the best solution through n.
- Example: hSLD() is admissible because the shortest path between any two points is a line.
- Theorem: A * Search is optimal if h() is admissible

Properties of A* Search

- **Completeness** (Guaranteed to find a path to the goal):
  - Yes, unless there are infinitely many nodes with f<= cost of solution
- **Time Complexity:**
  - O(b^m) (terrible if m is much larger than d but if solutions are dense, may be much faster than breadth-first)
- **Space Complexity:**
  - Keeps all nodes in memory
- **Optimality** (Does it find the shortest path):
  - Yes, if h() is admissible.

Iterative deepening A* search

- Iterative Deepening A ∗ is a low-memory variant of A ∗ which performs a series of depth-first searches, but cuts off each search when the sum f() = g() + h() exceeds some pre-defined threshold.
- The threshold is steadily increased with each successive search.
- IDA∗ is asymptotically as efficient as A∗ for domains where the number of states grows exponentially.

Heuristic Function

There is a whole family of Best First Search algorithms with different evaluation functions f(), that have a heuristic function h() .

- h() = estimated cost of the cheapest path from current node n to goal node.
- Heuristic functions are problem specific functions that provide an estimate of solution cost.

Dominance

If h2(n) > h1(n) for all n (both admissible) then h2 dominates h1 and is better for search. So the aim is to make the heuristic as large as possible, but without exceeding h*().

How to Find Heuristic Functions ?

Admissible heuristics can often be derived from the exact solution cost of a simplified or “relaxed” version of the problem. (i.e. with some of the constraints weakened or removed)

Week 5

Types of Games

Discrete Games

- Fully observable, deterministic (Chess, Checkers, Go, Othello)
- Fully observable, stochastic (Backgammon, Monopoly)
- Partially observable (Poker, Scrabble)

Continuous, Embodied Games

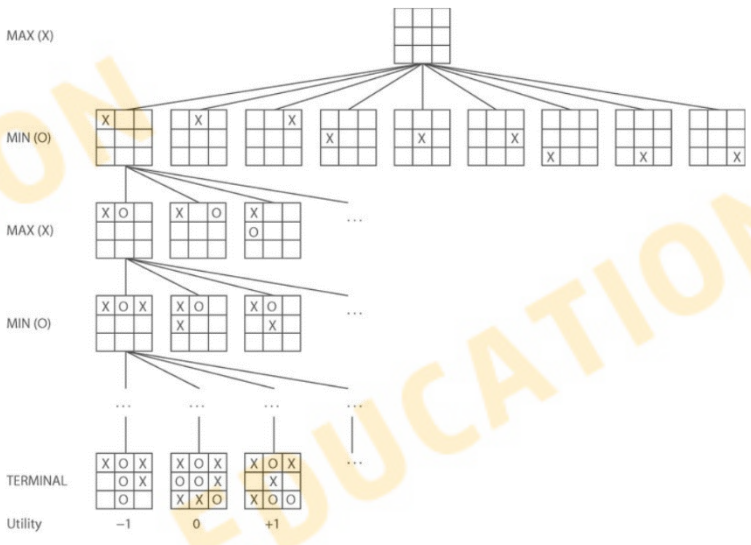
- Robocup Soccer
- Pool (Snooker

Game Trees and Search

The path search problems we looked at in the previous modules, although complex in their own way, were single-agent tasks where the agent can make whatever "moves" it wants without interference from another agent.

When playing a game, you have to deal with an unpredictable opponent. So the solution is not a single "plan" but rather a strategy which will enable you to respond to every opponent reply.

Additionally, you have a limited time in which to complete your move. So there is a tradeoff between speed of computation and accuracy/quality of your strategy.

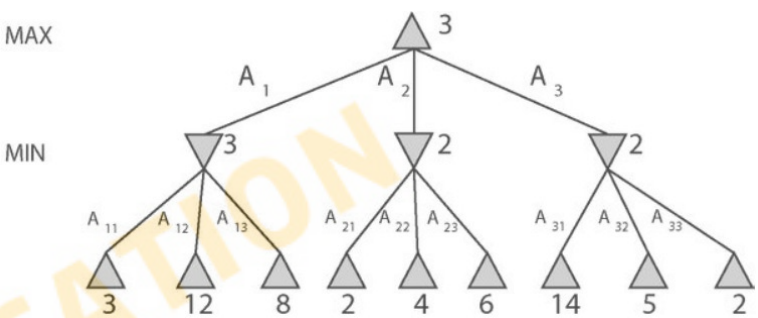


Game Tree Example

Minimax Search

Minimax is when one player tries to maximise the minimum score, and another player tries to minimise the maximum score.

Walkthrough example :



- Let's begin by considering the following very simple game, with only two moves.
- At the first move, you make a choice of A1, A2 or A3.
- At the second move, I make a choice between three options, depending on your choice.
- Afterwards, I have to pay you the number of dollars specified at the corresponding leaf of the tree.
- If you were to choose A1, which choice should I make? What about A2 and A3?
- The same principles can be be applied to trees of greater depth and arbitrary shape. One player is trying to minimize the ultimate payoff; the other player is trying to maximize it.

Minimax Algorithm

Minimax can be implemented in a way that has the same basic structure as Depth First Search.

At each leaf node, it returns either a terminal value (if the game has ended) or a heuristic value.

At each interior node, it recursively computes the values of all the child nodes, and returns either the minimum or maximum of these values.

Minimax and Negamax

The above formulation of Minimax assumes that all nodes are evaluated with respect to a fixed player (e.g. White in Chess).

If we instead assume that each node is evaluated with respect to the player whose turn it is to move, we get a simpler formulation known as Negamax.

Properties of Minimax

- **Completeness** (Guaranteed to find a path to the goal):
  - Only if tree is finite (some games have specific rules to avoid repeated states).
- **Time Complexity:**
  - $O(b^m)$ where b is the branching factor and m is the maximum number of moves in the game.
- **Space Complexity:**
  - O(bm), because we use Depth-First exploration.
- **Optimality** (Does it find the shortest path):
  - Yes, against an Optimal opponent. Against a non-optimal opponent, we could do better by exploiting the weaknesses of the opponent (or the possibility of them making mistakes)

Reducing the Search Effort

For Chess, $b \approx 35$, $m \approx 100$, so searching the full tree is completely infeasible. There are however, two ways we can reduce the search cost:

- Don't search to a final position. Instead, use heuristic evaluation at the leaves.
- Use α-β pruning to trim branches that don't need to be evaluated.

Heuristic Evaluation for Chess

Let's suppose we search a fixed number of moves ahead in a Chess game. When we get to a leaf in the search tree, how can we come up with a "heuristic evaluation" for this position?

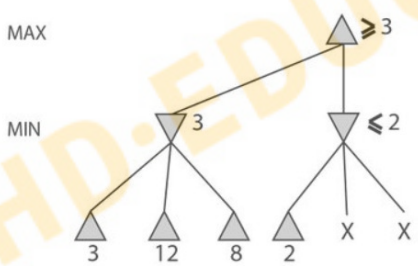
- Material
  - The simplest kind of evaluation is based on Material, i.e. which pieces are still on the board. It is often assumed that if a Pawn is worth 1 point, then a Knight or Bishop is worth 3 points, a Rook is worth 5 points, and a Queen is worth 9 points.
- Position
  - If we want to get more sophisticated, we can add a small adjustment depending on which square a piece is occupying.
- Interaction
  - We can also include additional features to indicate a particular piece is attacking or defending another piece.

The final evaluation then becomes a linear combination of a large number of features (typically, about 2000). But, the evaluation is very fast because only a small number of features are non-zero for any particular board state. For example, the Bishops can only occupy one or two of the 64 squares at a time, so we just need to look these values up in a lookup table and add them to the total, etc.

The value of individual features can be determined by reinforcement learning

Alpha-Beta Pruning Example

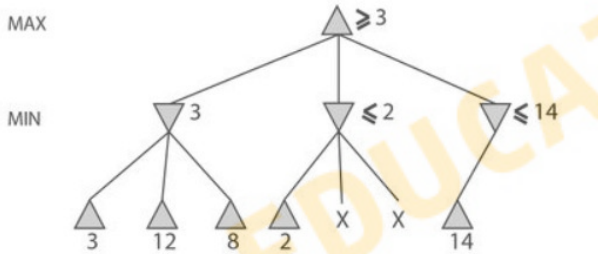
Let's consider again the simple two-move game we looked at last time. Once we know the information shown in this diagram, do we really need to evaluate the leaf positions marked as "X" ?



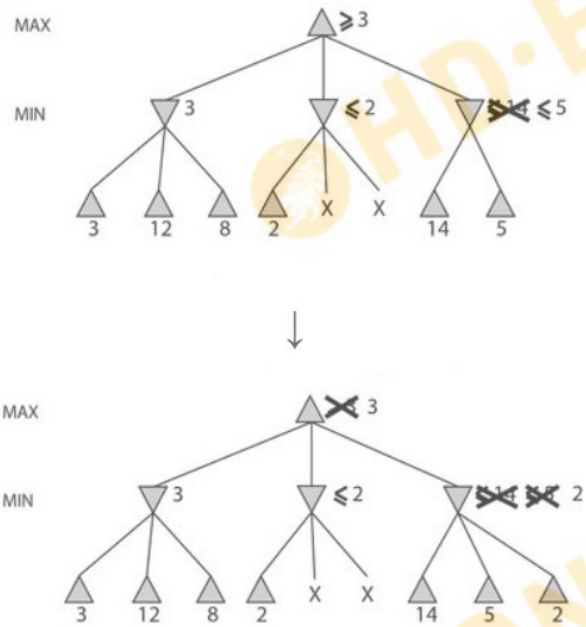
From the given information, we can already deduce the following:

- If we choose the left branch (and keep making optimal choices afterwards), we are guaranteed to get at least $3.
- If we choose the middle branch, we may have to accept only $2.

In other words, we have seen enough to know that the middle branch is not the best choice. We do not need to evaluate the nodes mark "X". These nodes can instead be pruned off, and we move on to explore another branch of the tree.



At this point, we are still not sure whether the right branch is a good choice or not, so we need to keep expanding its children.



In general, we are more likely to be able to "prune" in this way if the best move, or a very good move (in this case the "2") is examined early. In this example, if the "2" in the right branch had been examined first, we would have been able to do some pruning there as well.

Alpha-Beta Search Algorithm

These ideas can be exploited at every level of the tree by using the alpha-beta search algorithm. This algorithm is guaranteed to give the same answer as Minimax search, but the code runs much faster. At each node of the tree, we keep track of two values α and β such that the current node will only be relevant if its evaluation is between α and β.

```
function alphabeta( node, depth, α, β )
    if node is a terminal node or depth == 0
        return heuristic value of node
    if we are to play at node
        foreach child of node
            let α = max( α, alphabeta( child, depth-1, α, β ))
            if α ≥ β
                return α
        return α
    else // opponent is to play at node
        foreach child of node
            let β = min( β, alphabeta( child, depth-1, α, β ))
            if β ≤ α
                return β
        return β
```

Negamax Formulation of α-β Search

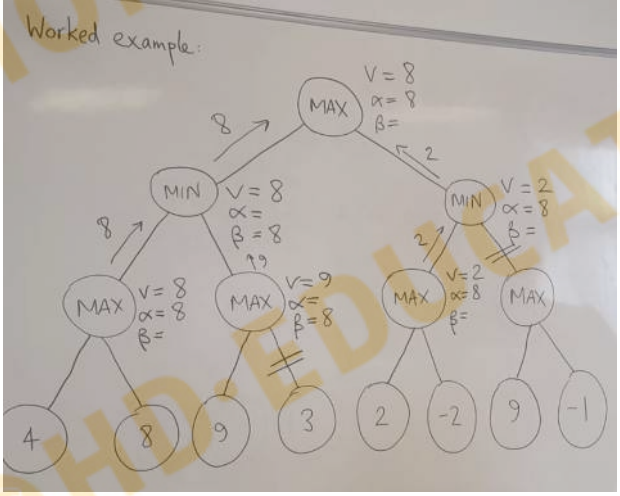
```
function minimax( node, depth )
    return alphabeta( node, depth, -∞, ∞ )
```

```
function alphabeta( node, depth, α, β )
    if node is terminal or depth == 0
        return heuristic value of node
        // from perspective of player whose turn it is to move
    foreach child of node
        let α = max( α, -alphabeta( child, depth-1, -β, -α ))
        if α ≥ β
            return α
    return α
```

When we trace through the alpha-beta search algorithm by hand, it is much easier to use the above version. But, when we write it into a computer program, it is better to use the Negamax implementation. Notice how the recursive call swaps both the sign and the order of α and β.

Why is it called α-β ?

- α is the best value already explored option along to the root for maximiser (maximise the score)
- β is the best value already explored option along to the root for minimiser (minimise the score)



- We assume the opponent is always playing the optimal move
- If we find a move whose value exceeds α, pass this new value up the tree.
- If the current node value exceeds β, it is "too good to be true", so we "prune off" the remaining children

Properties of Alpha-Beta Search

Alpha-Beta pruning is guaranteed to give the same result as minimax, but speeds up the computation substantially. Good move ordering improves effectiveness of pruning.

With "perfect ordering," time complexity = $O(b^{m/2})$

To prove that a "bad" move is bad, we only need to consider one (good) reply, but to prove that a "good" move is good, we need to consider all replies. This means alpha-beta can search twice as deep as plain Minimax. An increase in search depth from 6 to 12 could change a very weak player into a quite strong one.

Games of Chance

Stochastic Games are games which include an element of chance. In order to deal with "chance" events such as the rolling of the dice, we add a new type of node into our game tree search – known as a Chance node, and develop a generalisation of the Minimax algorithm, known as Expectimax.

Expectimax

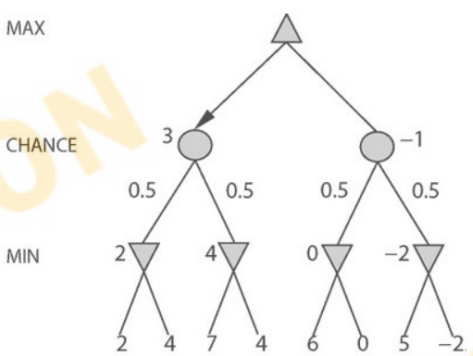
In order to understand Expectimax, let's consider the following very simple game of chance.

First, You make a choice between the left or right branch of the tree.

Then, a coin is tossed, and the game proceeds to either the left or right sub-branch, depending on the outcome of the coin toss.

Finally, I have the choice to move either left or right, and the leaf specifies the number of dollars that I have to pay to You.

At the lowest level of the tree we have Min nodes, because I will always want to pay the smallest amount possible and will therefore choose the branch with the Minimum value.



However, at the Chance nodes, the outcome is not chosen by either player but instead determined randomly. We deal with this by averaging the values of the child nodes, in order to compute the average or "Expected" value of the money that You will receive, once the game has reached that particular node. For the left branch, this value is 0.5x2 + 0.5x4 = 3. For the right branch, it is 0.5x0 + 0.5x(-2) = -1, meaning that, on average, You would be paying Me one dollar.

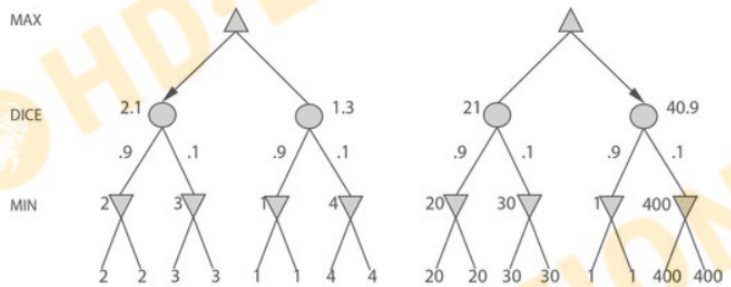
At the top level, You will choose the left branch, because you would prefer to receive $3 rather than pay $1 (on average).

The Importance of Exact Values

For Minimax search, the exact numerical values of the leaves do not matter but only their ordering – i.e. whether one position is ranked higher or lower than another.

For example, in the game above, what matters is that the two positions marked "2" are equally good, while the positions marked "1" and "4" are worse and better, respectively. If the numbers 1, 2 and 4 are replaced by 1, 20 and 400, the algorithm would run the same way and the same move would be chosen.

For Expectimax, on the other hand, Exact Values Do Matter! For example, in these two game trees, the ordering of the position evaluations is preserved, but the choice of move is different.



What does this mean in practice? For deterministic games like Chess, the board evaluation often consists of a linear combination of board features, i.e. $s = \sum_{i=1}^{N} w_i f_i$

where w_i is a parameter and f_i = 0 or 1depending on whether the i^{th} feature is present in that particular board state. The resulting value s could be positive or negative, and may typically range from, say -100 to 100.

For stochastic games like Backgammon, or when we are trying to learn the parameters through machine learning (discussed in Week 12), we typically want a board evaluation which will give us a value between 0 and 1, that can be interpreted as the probability of winning the game from this position. In practice, this is usually achieved by taking the linear function above and composing it with a (monotonic) sigmoidal function

$$z = \frac{1}{1 + \exp(-s)}$$

For Minimax search, it doesn't matter whether we use s or z for our evaluation, because the choice of move will be the same. But, for Chance nodes, and for machine learning, we need to use z instead of s.

Partially Observable Games

Card games are partially observable, because (some of) the opponents' cards are unknown. This makes the problem very difficult, because some information is known to one player but not to another, so typically we can calculate a probability for each possible deal.

The idea is to compute the minimax value of each action in each deal, then choose the action with highest expected value over all deals. One program that we can use to illustrate this strategy is GIB. It is a very strong bridge program, and it approximates this idea by:

- Generating 100 deals consistent with bidding information.
- Picking the action that wins most tricks on average.

Week 6

Types of Learning

- **Supervised Learning:** Agent is presented with examples of inputs and their target outputs
- **Reinforcement Learning:** Agent is not presented with target outputs, but is given a reward signal, which it aims to maximize.
- **Unsupervised Learning**: Agent is only presented with the inputs themselves, and aims to find structure in these inputs

Supervised Learning

Has a training set and a test set, each consisting of a set of items. For each item, a number of input attributes and a target value are specified.

- The aim is to predict the target value, based on the input attributes.
- Agent is presented with the input and target output for each item in the training set; it must then predict the output for each item in the test set

Various learning paradigms are available:

- Decision Tree
- Neural Network
- Support Vector Machine, etc.

Issues with Supervised Learning

- Framework (decision tree, neural network, SVM, etc.)
- Representation (of inputs and outputs)
- Pre-processing / post-processing
- Training method (perceptron learning, backpropagation, etc.)
- Generalization (avoid over-fitting) – Program can become too rigid, and not know how to deal with new data that it hasn't seen in it's training data.
- Evaluation (separate training and testing sets)

Ockham's Razor

"The most likely hypothesis is the simplest one consistent with the data."

Since there can be noise in the measurements, in practice we need to make a trade-off between simplicity of the hypothesis and how well it fits the data.

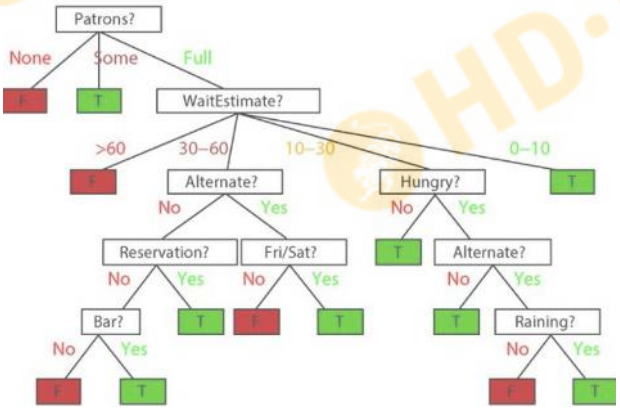
However, if we know that the data is extremely accurate then we may choose to fit the line exactly (use the 4^{th} order polynomial instead of parabola)

Decision Trees

Given a data set, we can create a decision tree to predict an outcome.

For example,

	Alt	Bar	F/S	Hun	Pat	Price	Rain	Res	Type	Est	Wait?
X_1	T	F	F	T	Some	$$$	F	T	French	0–10	T
X_2	T	F	F	T	Full	$	F	F	Thai	30–60	F
X_3	F	T	F	F	Some	$	F	F	Burger	0–10	T
X_4	T	F	T	T	Full	$	F	F	Thai	10–30	T
X_5	T	F	T	F	Full	$$$	F	T	French	>60	F
X_6	F	T	F	T	Some	$$	T	T	Italian	0–10	T
X_7	F	T	F	F	None	$	T	F	Burger	0–10	F
X_8	F	F	F	T	Some	$$	T	T	Thai	0–10	T
X_9	F	T	T	F	Full	$	T	F	Burger	>60	F
X_{10}	T	T	T	T	Full	$$$	F	T	Italian	10–30	F
X_{11}	F	F	F	F	None	$	F	F	Thai	0–10	F
X_{12}	T	T	T	T	Full	$	F	F	Burger	30–60	T



Generalization

- If the training data is not inconsistent, we can split the attributes in any order and still produce a tree that correctly classifies all examples in the training set.
- However, we really want a tree which is likely to generalize to correctly classify the (unseen) examples in the test set.
- In view of Ochkam's Razor, we prefer a simpler hypothesis, i.e. a smaller tree.
- We can chose attributes that give us more information to create smaller trees
- This notion of "informativeness" can be quantified using the mathematical concept of "entropy".
    - Eg. Patrons is a "more informative" attribute than Type, because it splits the examples more nearly into sets that are "all positive" or "all negative".
- A parsimonious tree can be built by minimizing the entropy at each step.

Entropy

Entropy is a measure of how much information we gain when the target attribute is revealed to us.

- In other words, it is not a measure of how much we know, but of how much we don't know.
- If the prior probabilities of the n target attribute values are $p_1 \ldots p_n$ then the entropy is

$$H(<p_1 \ldots p_n>) = \sum_{i=1}^{n} -p_i \log_2 p_i$$

Getting back to decision trees, suppose we have p positive and n negative examples at a node.

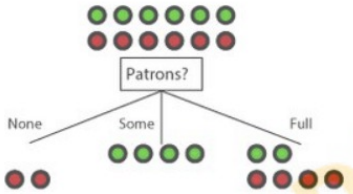
The $H(<\frac{p}{(p+n)}, \frac{n}{(p+n)}>)$ bits are needed to classify a new example.

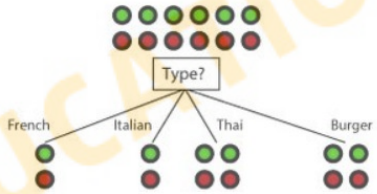
e.g. for 12 restaurant examples, p=n=6 so we need 1 bit.

- An attribute splits the examples E into subsets E_i, each of which needs less information to complete the classification.
- Let E_i have p_i positive and n_i negative examples
- The $H(<\frac{p_i}{(p_i+n_i)}, \frac{n_i}{(p_i+n_i)}>)$ bits needed to classify a new example
- So the expected number of bits per example over all branches is

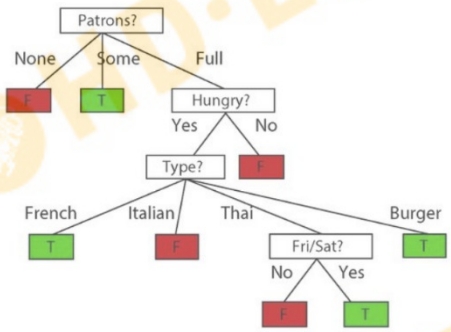
$$\sum_i \frac{p_i+n_i}{p+n} H$$ ¿¿

- For Patrons, this is 0.459 bits, for Type this is (still) 1 bit.
- Therefore, Patrons is a better choice.





$$\text{For Patrons, Entropy} = \frac{1}{6}(0)+\frac{1}{3}(0)+\frac{1}{2}\left[-\frac{1}{3}\log(\frac{1}{3})-\frac{2}{3}\log(\frac{2}{3})\right]$$
$$= 0+0+\frac{1}{2}\left[\frac{1}{3}(1.585)+\frac{2}{3}(0.585)\right] = 0.459$$
$$\text{For Type, Entropy} = \frac{1}{6}(1)+\frac{1}{6}(1)+\frac{1}{3}(1)+\frac{1}{3}(1) = 1$$



Continuing with this process, we get a much smaller tree than if we had chosen the attributes randomly.

Induced Tree

According to Ockham's Razor, we may wish to prune off branches that don't provide much benefit in classifying the items (to make smaller trees)

- When a node becomes a leaf, all items will be assigned to the majority class at that node. We can estimate the error rate on the (unseen) test items using the **Laplace error.**

Laplace error: $E = 1 - \frac{n+1}{(N+k)}$

N = total number of (training) items at the node

n = number of (training) items in the majority class

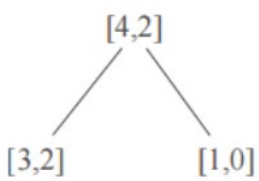
k = number of classes

- If the **average** Laplace error of the children exceeds that of the parent node, we prune off the children.
- That is, we calculate the Laplace error of all the children, average them, and compare them to the parent

Initially only the leaves in the tree are considered for pruning. But, if leaves are pruned, the parent node becomes a leaf and we can continue the process, until no further pruning is indicated.

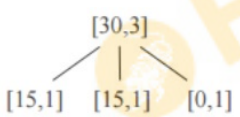
Minimal Error Pruning

Example: Should the children of this node be pruned or not?



- Left child has class frequencies [3,2] = $E = 1 - \frac{n+1}{(N+k)} = 1 - \frac{3+1}{5+2} = 0.429$
- Right child has E = 0.333
- Parent node has E = 0.375
- Average for Left and Right child is $E = \frac{5}{6}(0.429)+\frac{1}{6}(0.333) = 0.413$
- Since 0.413 > 0.375, children should be pruned.

Example 2. Should the children of this node be pruned or not?



- Left and Middle child have class frequencies [15,1] = $E = 1 - \frac{n+1}{(N+k)} = 1 - \frac{15+1}{16+2} = 0.1111$
- Right child has E = 0.333
- Parent node has $E = \frac{4}{35} = 0.114$

- Average for Left, Middle and Right child is $E = \frac{16}{33}(0.111) - \frac{16}{33}(0.111) + \frac{1}{33}(0.333) = 0.118$
- Since 0.118 > 0.114, children should be pruned.

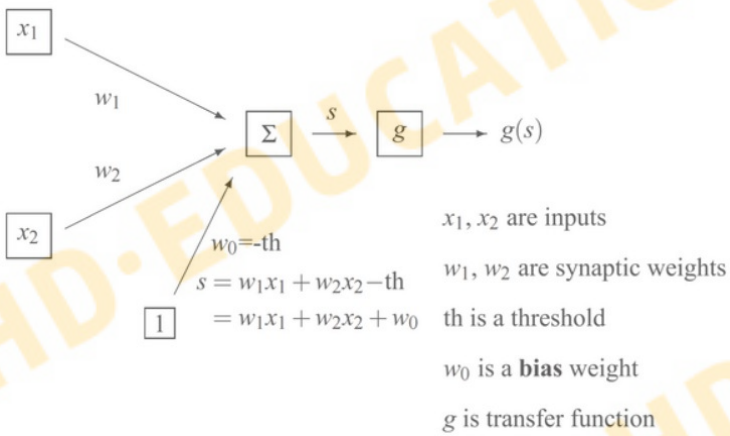
Artificial Neural Networks

(Artificial) Neural Networks are made up of nodes which have the following:

- inputs edges, each with some weight
  - Weights can be positive or negative and may change over time (learning).
  - The input function is the weighted sum of the activation levels of inputs.
- one or more outputs edges
- an activation level (a function of the inputs)
  - The activation level is a non-linear transfer function of this (linear) input: $activation_i = g(s_i) = g$¿

Some nodes are inputs (sensing), some are outputs (action)

McCulloch & Pitts Model of a Single Neuron



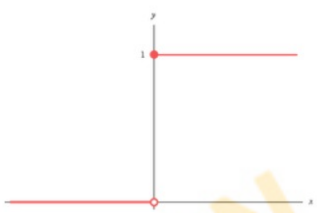
Consider an artificial neuron with n inputs $x_1 \ldots x_n$ and one output. The neuron computes a weighted sum of the inputs, $w_1x_1+\ldots+w_nx_n$. If this weighted sum is greater than a certain threshold th, the output of the neuron is 1; otherwise the output is 0.

As a mathematical convenience, rather than subtracting the threshold th , we instead add a bias w_0, and then apply a transfer function g() So the output of the neuron can be written as g(s), where

$S = w_1x_1+\ldots+w_nx_n + w_0$

Transfer Function

Originally, a (discontinuous) step function was used for the transfer function:

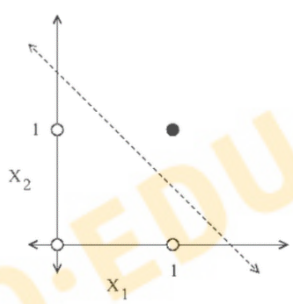


(Later, other transfer functions were introduced, which are continuous and smooth)

Example: the AND function

As an example, consider the AND function of two variables X_1 and X_2, where FALSE is represented by 0 and TRUE by 1.

X_1	X_2	$X_1 \wedge X_2$
0	0	0
0	1	0
1	0	0
1	1	1



We can plot these four point on a two dimensional graph, and draw a line separating the positive from the negative point(s)

The slope of this line is -1 and the y-intercept is 1.5, so the equation of the line is $X_2 = -X_1 + 1.5$, i.e is $X_1 + X_2 - 1.5 = 0$. . However, we need to replace this equation with an inequality, i.e. $X_1 + X_2 - 1.5 >$ or < 0.

We can check which way the inequality goes by examining the positive point, where $X_1 = 1$ and $X_2 = 1$, . This point satisfies $X_1 + X_2 - 1.5 > 0$, so the weights of the network must be $w_1 = 1$, $w_2 = 1$, $w_0 = 1.5$, .

Note that, in the case of the perceptron network, multiplying all of the weights by a fixed positive constant will not change the function that is computed. For example, $w_1 = 10$, $w_2 = 10$, $w_0 = 15$ would also compute the AND function. (This will not be the case when we move to continuous transfer functions in the next section.)

Linear Separability

A single perceptron can only compute Linearly Separable functions.

- It must be possible to draw a straight line (or, if there are n inputs, an (n-1)-dimensional hyperplane) in such a way that all the positive points lie on one side, and all the negative points lie on the other side.

Examples include:

AND $w_1 = w_2 = 1.0, \quad w_0 = -1.5$

OR $w_1 = w_2 = 1.0, \quad w_0 = -0.5$

NOR $w_1 = w_2 = -1.0, \quad w_0 = 0.5$

The Need for Training - The above approach works fine if there are a small number of inputs. We need another method to train a larger network to learn a new function to automatically determine weights for the network, based on a set of training data.

Perceptron Learning

Perceptron Learning Rule

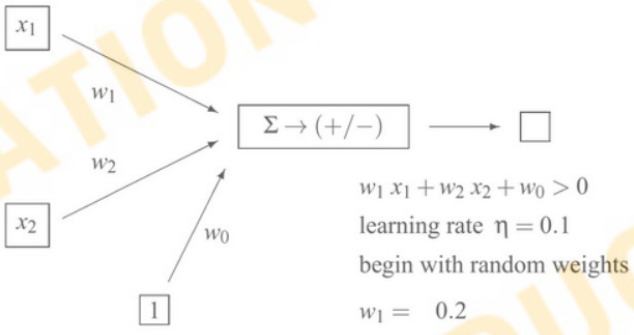
The idea is to adjust the weights as each input is presented.

- Choose r > 0 (the learning rate)
- Recall: $s = w_1x_1+...+w_nx_n + w_0$
- If g(s) = 0, but should be 1
  $W_k = w_k + rx_k$
  $W_0 = w_0 + r$
  So $s = s + r(1 + \sum_k x_k^2$ ¿¿
- If g(s) = 1 but should be 0
  $W_k = w_k - rx_k$
  $W_0 = w_0 - r$
  So $s = s + r(1 + \sum_k x_k^2$ ¿¿

Otherwise, weights are unchanged.

Keep cycling through the training items, until all of them are successfully learned.

Example



$w_1 x_1 + w_2 x_2 + w_0 > 0$

learning rate $\eta = 0.1$

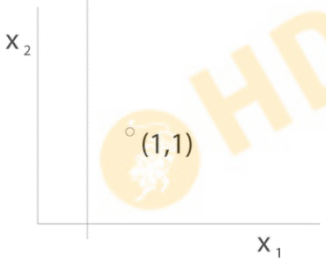
begin with random weights

$w_1 = 0.2$

$w_2 = 0.0$

$w_0 = -0.1$

1.



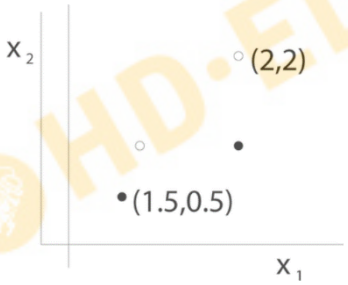
$0.2x_1 + 0\ x_2 - 0.1 > 0$
$W_1 = w_1 - rx_1 = 0.1$
$W_2 = w_2 = rx_2 = -0.1$
$W_0 = w_0 - r = -0.2$

2.

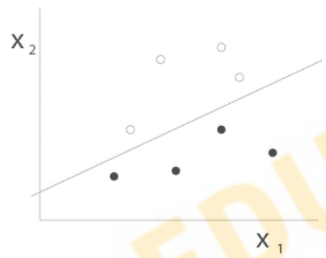
$0.1x_1 - 0.1x_2 - 0.2 > 0$

$W_1 = w_1 - rx_1 = 0.3$

$W_2 = w_2 = rx_2 = 0.0$
$W_0 = w_0 - r = -0.1$



3.
$0.3x_1 + 0\ x_2 - 0.1 > 0$
3rd point is correctly classified, no change
4th point:
$W_1 = w_1 - rx_1 = 0.1$
$W_2 = w_2 = rx_2 = -0.2$
$W_0 = w_0 - r = -0.2$



4.

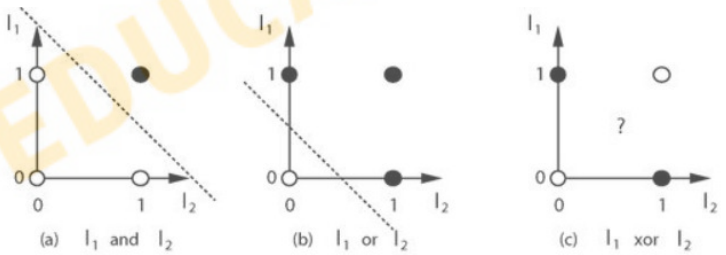
Eventually, all the data will be correctly classified (provided that it is linearly separable).

Week 7

Limitations of Perceptrons

Problem: many useful functions are not linearly separable (e.g. XOR)

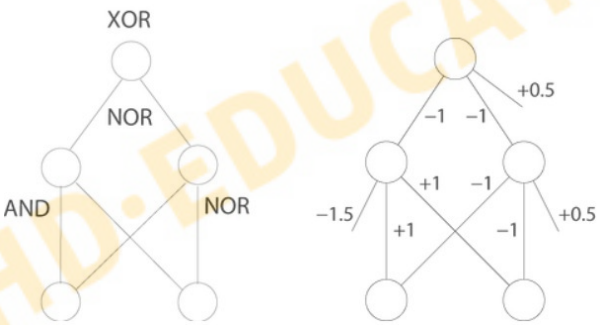
I_1	I_2	I_1 XOR I_2
0	0	0
0	1	1
1	0	1
1	1	0



Possible solution: Rewrite a given logical expression in terms of simpler functions such as AND, OR and NOR, which can each be implemented by a single neuron. Then build a network of neurons to implement this combination.

For example, x1 XOR x2 can be written as: (x1 AND x2) NOR (x1 NOR x2)

Multi-Layer Neural Networks



Backpropogation

Neural Network Training as Cost Minimization

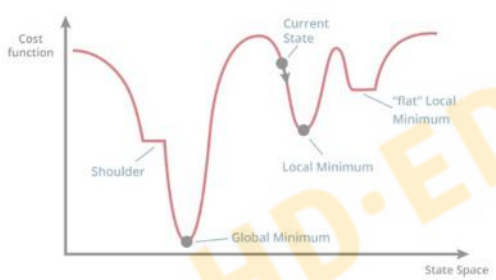
Imagine we have a training set consisting of input patterns (x_1,x_2), each with a corresponding target output t. When the same input (x_1,x_2) is fed to the network, it produces output z .

Our aim is to make the actual output z , for each item in the training set, as close as possible to the target value for that item. We can do this by minimising the Error or "Cost" function:

$$E=\frac{1}{2}\sum(z-t)^2$$ (The factor of 1/2 is included just for mathematical convenience.)

- If we think of E as height, this defines an error landscape in the weight space.
- When formulated this way, the problem becomes very similar to the Constraint Satisfaction Problems we explored in Week 6, using Local Search.

Local Search in Weight Space

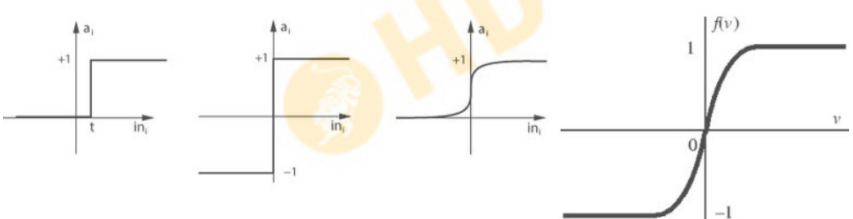


The problem is that, if we use the step function as our transfer function, the landscape will not be smooth but will instead consist almost entirely of flat local regions and "shoulders", with occasional discontinuous jumps.

For the perceptron this didn't matter, because it only had one layer. But, for networks with two or more layers, it becomes a big problem.

Key Idea

The key idea is to replace the (discontinuous) step function with a differentiable function, such as:



- Sigmoid **g(s)** = $\frac{1}{1+e^{-2s}}$
- hyperbolic tangent **g(s)** = tanh(s) = $\frac{\left(e^{s}-e^{-s}\right)}{e^{s}+e^{-s}}=2\left(\frac{1}{1+e^{-2s}}\right)-1$

Gradient Descent

The error function E is defined to be (half) the sum over all input patterns of the square of the difference between actual output and target output.

$$E=\frac{1}{2}\sum\left(z-t\right)^{2}$$

The aim is to find a set of weights for which E is very low. If the functions involved are smooth, we can use multi-variable calculus to adjust the weights in such a way as to take us in the steepest downhill direction.

$$w=w-r\frac{dE}{dw}$$

The parameter r is called the learning rate.

Chain Rule

If,

y = y(u) and
u = u(x),

Then,

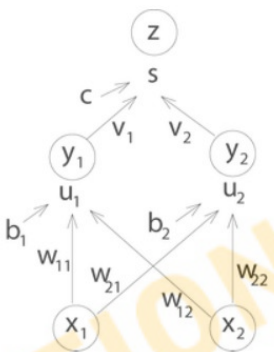
$$\frac{dy}{dx}=\frac{\frac{dy}{du}*du}{dx}$$

This principle can be used to compute the partial derivatives in an efficient and localized manner. Note that the transfer function must be differentiable (usually sigmoid, or tanh).

Note:

if $z(s)=\frac{1}{1+e^{-s}}$, $z'(s)=z(1-z)$.

Forward Pass



$$\begin{aligned} u_1 &= b_1 + w_{11}x_1 + w_{12}x_2 \\ y_1 &= g(u_1) \\ s &= c + v_1y_1 + v_2y_2 \\ z &= g(s) \\ E &= \frac{1}{2}\sum(z-t)^2 \end{aligned}$$

Backpropagation

Partial Derivatives

$$\begin{aligned} \frac{\partial E}{\partial z} &= z-t \\ \frac{dz}{ds} &= g'(s)=z(1-z) \\ \frac{\partial s}{\partial y_1} &= v_1 \\ \frac{dy_1}{du_1} &= y_1(1-y_1) \end{aligned}$$

Useful Notation

$$\delta_{\text{out}}=\frac{\partial E}{\partial s} \quad \delta_1=\frac{\partial E}{\partial u_1} \quad \delta_2=\frac{\partial E}{\partial u_2}$$

$$\begin{aligned} \delta_{\text{out}} &= (z-t)\,z\,(1-z) \\ \frac{\partial E}{\partial v_1} &= \delta_{\text{out}}\,y_1 \\ \delta_1 &= \delta_{\text{out}}\,v_1\,y_1\,(1-y_1) \\ \frac{\partial E}{\partial w_{11}} &= \delta_1\,x_1 \end{aligned}$$

Training Tips

- re-scale inputs and outputs to be in the range 0 to 1 or −1 to 1
- initialize weights to very small random values
- on-line or batch learning
- three different ways to prevent overfitting:
  - limit the number of hidden nodes or connections

    - limit the training time, using a validation set
    - weight decay
- adjust learning rate (and momentum) to suit the particular task

Applications of Neural Networks

- Autonomous Driving
- Game Playing
- Credit Card Fraud Detection
- Handwriting Recognition
- Financial Prediction

Supervised Learning

We have a training set and a test set, each consisting of a set of examples. For each example, a number of input attributes and a target attribute are specified.

- The aim is to predict the target attribute, based on the input attributes.
- Various learning paradigms are available, such as Decision Trees, Neural Networks, and others.

Learning of Actions

Supervised Learning can also be used to learn Actions, if we construct a training set of situation-action pairs (this is called Behavioural Cloning).

However, there are many applications for which it is difficult, inappropriate, or even impossible to provide a "training set".

- Optimal control - mobile robots, pole balancing, flying a helicopter
- Resource allocation - job shop scheduling, mobile phone channel allocation
- Mix of allocation and control -elevator control, backgammon

Reinforcement Learning Framework

- Learn from experience, reward based learning
- Can learn the best things to do without knowing anything about the environment that is in, by just interacting with the environment
- An agent interacts with its environment.
- There is a set S of states and a set A of actions.
- At each time step t, the agent is in some state s_t. It must choose an action a_t whereupon it goes into state s_{t+1} $=n(s_t, a_t)$ and receives reward $r(s_t,a_t)$.
- The reward function is user defined
- In general, r() and n() can be multi-valued, with a random element
- The aim is to find an optimal *policy* π: S-> A which will maximize the cumulative reward.

Models of Optimality

Question: Is a fast nickel worth a slow dime?

Is it better to get a +1 reward immediately or a +10 reward 5 turns later?

A finite horizon reward $\sum_{i=0}^{h} r_{t+i}$

An infinite discounted reward $\sum_{i=0}^{\infty} \gamma^i r_{t+i}, \quad 0 \le \gamma < 1$

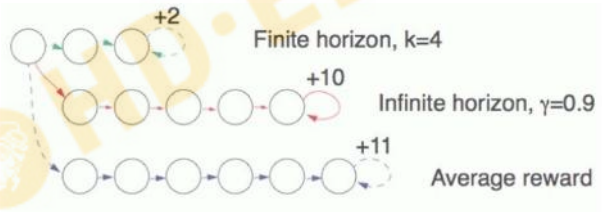
An average reward $\lim_{h\to\infty} \frac{1}{h} \sum_{i=0}^{h-1} r_{t+i}$

Key Takeaways:

- Best reward also deals with perception of future/utility cost – if you will be rich in the future, you will take money now. If you think you'll be poor in the future, you'll wait for the money.
- Computationally, a finite horizon reward is simple – sum up all rewards from a set number of moves
- An infinite discounted reward is easier for proving theorems – idea is get reward now and it's worth x amount, but a reward in the future is worth slightly less. Eg. Reward 1 is now, reward 0.9 is 1 step later, reward 0.81 is 2 steps later. Values rewards in the future, but values rewards now more

- An average reward is hard to deal with because one cannot sensibly choose between a small reward soon and a large reward very far into the future. Idea – care about every reward in the future, only works in games that are endless/non episodic like chess or balance a pole forever

Comparing Models of Optimality

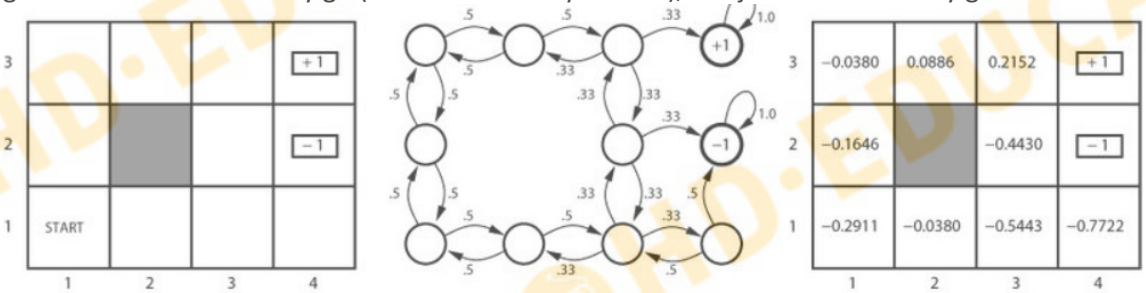


The choice of action depends on our model of optimality.

- If we are using a finite horizon model with , the top action would be preferred as the agent only looks ahead 4 steps and does not see the +10.
- If we are using an infinite horizon model with discount factor 0.9, the second action would be preferred. This is as even though the 6th step gives us a reward of 11, with the discount it will only cost 9.9 compared to the 10 so we pick the 5th one
- If we are using an average reward model, the third action would be preferred. If we look ahead infinitely, the first few steps are irrelevant so we pick which ever gives us the highest reward.

Value Function

For each state s let V*(s) be the maximum discounted reward obtainable from s. The V* function is the optimal value function, that is the best reward they can get in their state. In the example case below as the agent does not get a choice in where they go (all determined by chance), it is just the value they get.



Learning the Value Function can help you to determine the optimal strategy.

Environment Types

Environments can be:

- Passive and Stochastic (the previous example) – where the game state doesn't change, no active action taken
- Active and Deterministic (chess) – where agent can take action
- Active and Stochastic (backgammon) – take action based on chance (eg. Poker)

Exploration / Exploitation Trade-off

Most of the time we should choose what we think is the best action. Eg. Go to an ok old place, or a potentially great or terrible new place

However, to ensure convergence to the optimal strategy, we must occasionally choose something different from our preferred action. For instance, you may choose a random action 5% of the time, or use a Bolzmann distribution to choose the next action:

$$P(a) = \frac{e^{\frac{V(a)}{T}}}{\sum_{b \in A} e^{\frac{v(b)}{T}}}$$

Delayed Reinforcement

The general situation can be thought of as multiple rooms, each with its own slot machines. When we pull a lever, we get a reward and we are also

transported to another room. The agent does not know in advance which lever leads to which room, or what rewards are available.

For the example to the side, the optimal solution is to get to 2 and loop itself. However, the agent does not know that and must make decisions with no information (which is done by Temporal Difference Learning)

Temporal Difference Learning

Temporal Difference Learning is alternately known as either Adaptive Heuristic Critic or the Widrow-Hoff Rule.

TD(0) can be written out as: $$\hat{V}(s) \leftarrow \hat{V}(s) + \eta\,[\, r(s,a) + \gamma\, \hat{V}(\delta(s,a)) - \hat{V}(s)\,]$$

Where η represents the learning rate.

It works with the estimated value function V which is an estimate of how good it is to be in state s . Then, the delta function δ(s,a) returns the next state if you take action a, which is then run through V again to determine how good it is to be in the next state if you take action a. The brackets is the bellman equation – if I have my value and state now, I can approximate and update my current value by using the value of the next state I have stored. So instead of to run through the entire enviornment, can just say that since the value function is accurate, then the value of the current state is the current reward plus the discounted the value of the next state. Even if the value function isn't accurate, it can converge into an optimal value anyway if you run it long enough.

The (discounted) value of the next state, plus the immediate reward, is used as the target value for the current state. A more sophisticated version, called TD(n) , uses a weighted average of future states.

Q-Learning

Q learning is a way of very quickly learning optimal actions stably, with conversion guarantees. Q learning says 'give me a state and action and I will tell you how good that pair of state and action is'. That is, predict the best action in state s in order to maximise a cumulative reward

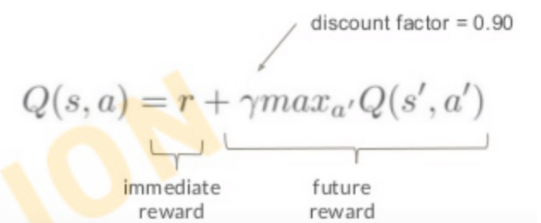
For each s, let V*(s) be the maximum discounted reward obtainable from s, and let Q(s,a) be the discounted reward available by first doing action a and then acting optimally.

Then the optimal policy is $\pi^*(s) = \text{argmax}_a Q(s,a)$

where $Q(s,a) = r(s,a) + \gamma V^*(\delta(s,a))$

then $V^*(s) = \max_a Q(s,a)$

so $Q(s,a) = r(s,a) + \gamma \max_b Q(\delta(s,a), b)$

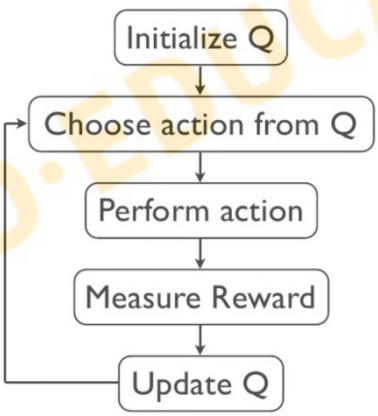


which allows us to iteratively approximate Q by $\hat{Q}(s,a) \leftarrow r + \gamma \max_b \hat{Q}(\delta(s,a), b)$

Theoretical Results

Theorem: Q-learning will eventually converge to the optimal policy, for any deterministic Markov decision process, assuming an appropriately randomised strategy. (Watkins & Dayan 1992)

Theorem: TD-learning will also converge, with probability 1. (Sutton 1988, Dayan 1992, Dayan & Sejnowski 1994)



Limitations of Theoretical Results

1. **Delayed Reinforcement**
   Any reward resulting from an action may not be received until several time steps later. This also slows down the learning.
2. **Search space must be finite**
   Convergence is slow if the search space is large.
   It relies on visiting every state infinitely often.
3. **For "real world" problems, we can't rely on a lookup table**
   Some kind of generalisation (e.g. TD-Gammon) is required.

Week 8

Constraint Satisfaction Problems (CSPs)

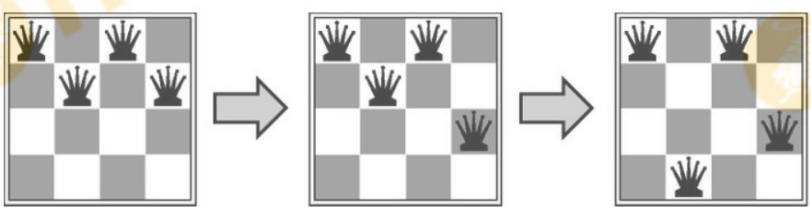
CSPs are defined by a set of variables X_i, each with a domain D_i of possible values and a set of constraints C_i.
To solve this, we need to find an assignment of the variables X_i from the domains D_i such that none of the constraints C_i are violated.

Eg. Colouring in each state Australia with adjacent states of different colours

- Variables: WA, NT, Q, NSW, V, SA, T
- Domains: red, green, blue
- Constraints: Adjacent regions must have different colours

Eg 2. N Queens puzzle – Put n Queens on an n by n Chess board so that no two Queens are in the same row, column, or diagonal.

- Variables: Q1,Q2,Q3,Q4
- Domains: 1,2,3,4
- Constraints:
  Qi != Qj (not in same row)
  |Qi-Qj| != |i-j|(now in same diagonal



Cryptarithmetric

Cryptarithmetric is a type of mathematical puzzle where the numbers have been replaced with letters, or other symbols.

Eg 4. Sudoku

- Variables: 9x9 = 81 numbers
- Domains: 1,2,3,4,5,6,7,8,9
- Constraints: No digit repeated in a row, column or (3x3) box.

Varieties of Constraints
Different types of constraints are appropriate for different types of problems.

- **<u>Unary constraints</u>**
  Involve a single variable
  M ≠ 0
- **<u>Binary constraints</u>**
  Involve pairs of variables
  SA ≠ WA
- **<u>Higher-order constraints</u>**
  Involve 3 or more variables
  Y = D+E or Y = D+E −10
- **<u>Inequality constraints on Continuous variables</u>**
  These are appropriate for factory scheduling,
  e.g. EndJob1 + 5 ≤ StartJob

- **Soft constraints**
  Sometimes, certain conditions are considered desirable but not essential,

e.g. An 11:00am lecture is better than an 8:00am lecture.

Backtracking Search
Constraint Satisfaction Problems can be solved by assigning values to variables one at a time, in different combinations. Whenever a constraint is violated, we back track tot the most recently assigned variable and assign it a new value.

This can be achieved by a Depth First Search on a special kind of state space, where states are defined by the values assigned so far:

- **Initial state**: The empty assignment.
- **Successor function**: Assign a value to an unassigned variable that does not conflict with previously assigned values of other variables. (If no legal values remain, the successor function fails.)
- **Goal test**: All variables have been assigned a value, and no constraints have been violated.

The difference between path search and constraint satisfaction:
- **Path Search**: knowing the final state of a path is easy, getting to the goal state is hard,
- **CSP**: the difficult part is knowing what the final state is, and getting to the final state is easy.
  Eg. Rubiks cube vs n-Queens

Backtracking Search
The search space for a Constraint Satisfaction Problems has specific properties, making it quite different from the traditional Path Search Problems.
For example:
- If there are n variables, then every solution will occur at exactly depth n
- Furthermore, variable assignments are commutative in the sense that the order in which variables are assigned does not change the final result.
  - Eg. If we assign WA = RED and then NT = GREEN, the state is the same as if we assigned NT = GREEN then WA = RED.

For this reason, we can speed up the search by choosing to assign the variables in a different order in different parts of the search tree, and by choosing the order in which the different values for a particular variable are explored.

General-Purpose Heuristics
Certain general-purpose heuristics can be used to greatly speed up Backtracking Search. We need to consider:
- Which variable should be assigned next?
- In what order should its values be tried?
- Can we detect inevitable failure early?

Minimum Remaining Values
Minimum Remaining Values (MRV): Is when we choose the variable with the fewest remaining legal moves to work on next.

Eg. After colouring WA Red, we choose either NT or SA next because these have two remaining values while the other states have three.

After colouring NT Green, we choose SA next because it has only one remaining value.

Degree Heuristic
Degree Heuristic: Choose the variable with the most constraints on remaining variables.
- Used as a *tie-breaker* among variables with the same minimum MRV value.

Eg. SA is the best region to color first, because it interacts with 5 other variables.

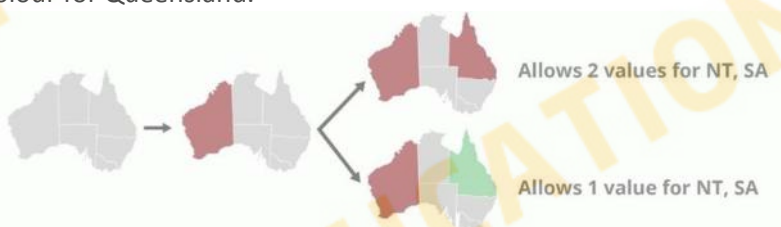
After colouring SA Blue, it is best to choose NT, Q or NSW because these each interact with two other variables.

After colouring SA, QLD is the best choice because it has only on remaining value, and it borders on NSW.

Least Constraining Value
Given a variable, choose the least constraining value: the one that rules out the fewest number of values in the remaining variables.

Eg. Suppose we colour WA Red and then assign a colour for Queensland.

If Q in Red, then two values remain for NT and SA. If we colour it Green or Blue, only one value will remain for NT, SA. Therefore, Red is preferred.

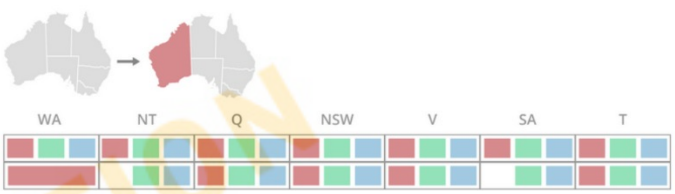


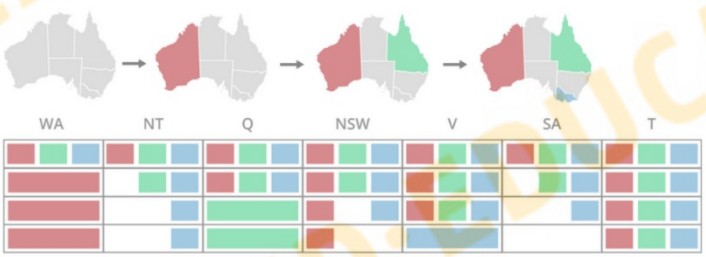
Forward Checking
Idea: Keep track of remaining legal values for unassigned variables. When any variable has no legal values remaining, prune off that part of the search tree, and backtrack.

Eg.

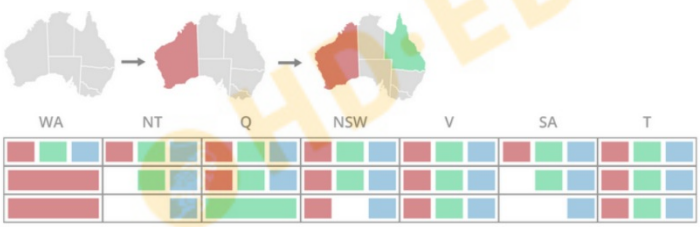
Initially all values are the same

If WA is Red, Red is not available for NT or SA.





After colouring Q Green, Green is not available for NT, NSW or SA.

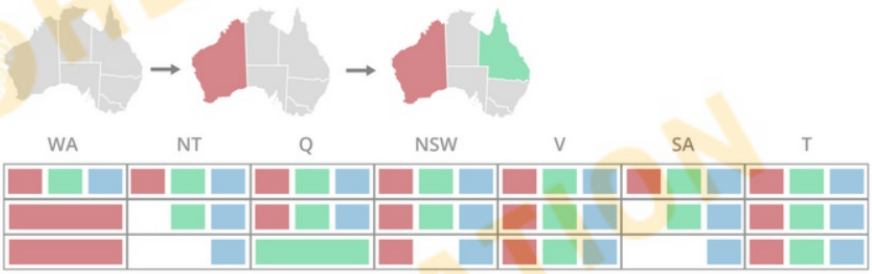


After colouring Vic Blue, Blue is not available for NSW or SA. This means that SA has no values remaining, so we terminate this part of the search.

Constraint Propagation

Forward checking spreads information from assigned to unassigned variables, but doesn't provide early detection for all failures:

Eg. For this state

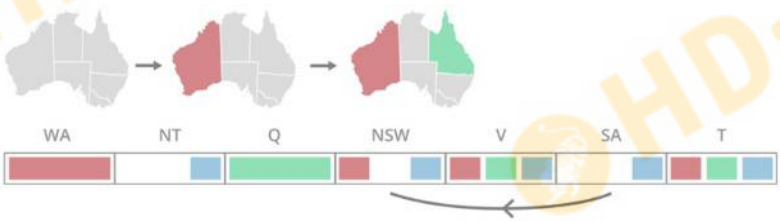


- Blue is the only option remaining for NT and SA. But, they border each other, so they can't both be Blue!
- This kind of reasoning can be implemented by Constraint Propagation, using Arc Consistency.

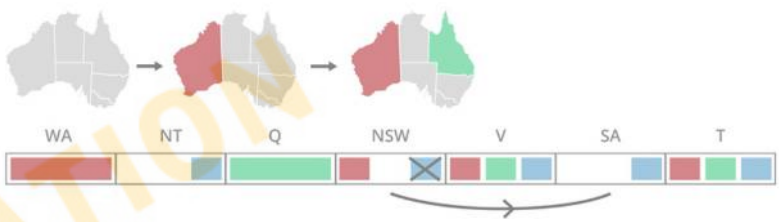
Arc Consistency

We look at all interacting variables to make sure they are consistent.

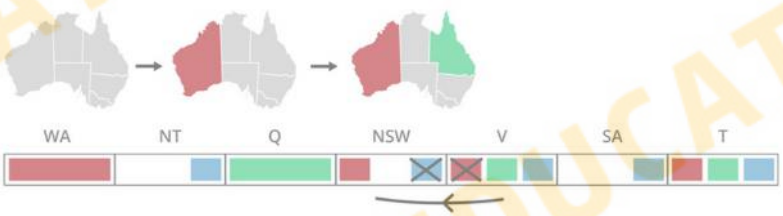
- X → Y is consistent if for every remaining value for X there is some remaining value for Y that is compatible with it.



This is ok, because Blue for SA is compatible with Red for NSW.



The other way around, Blue for NSW is not compatible with any value for SA, so we cross it out.



If X loses a value, neighbors of X need to be rechecked. In this case, Red for Victoria is not compatible with any remaining value for NSW, so we cross that out.

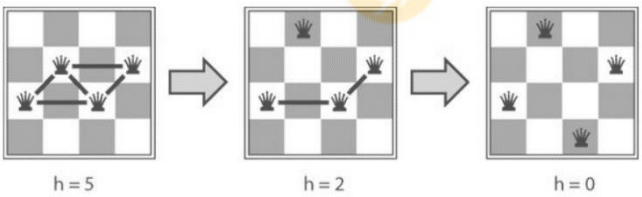
Blue for SA is not compatible with any remaining value for NT, so we cross it out, leaving SA with no legal values.

- Arc Consistency detects failure earlier than forward checking.
- For some problems, it can speed up the search enormously.

- For others, it may slow the search due to computational overheads. For small problems, it's best to leave this out as any time saved from not calculating additional states is cancelled by the computational time

Local Search

"Iterative Improvement" or Local Search is another class of algorithms for solving CSP's. These algorithms assign all variables randomly in the beginning (and violates several constraints), and then change one variable at a time, trying to reduce the number of violations at each step.

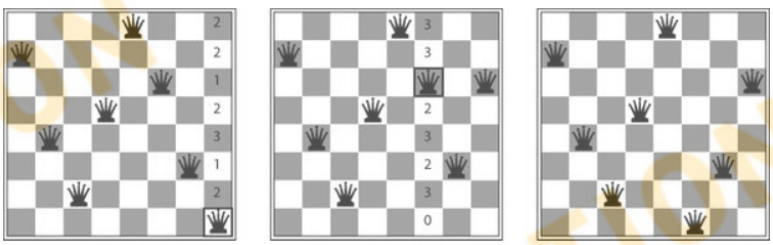


Hill-Climbing by Min-Conflicts

Variable selection: randomly select any conflicted variable

Value selection by min-conflicts heuristic

- Choose the value that violates the fewest constraints

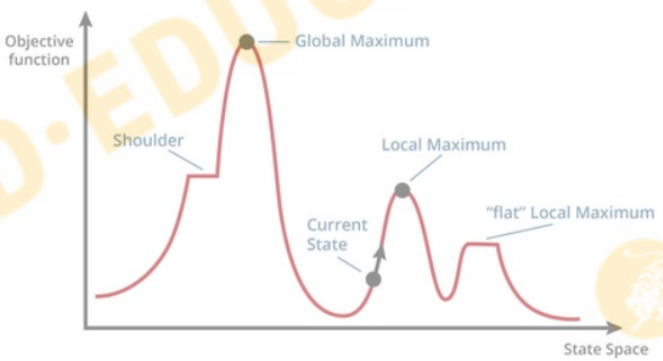


Flat regions and local optima

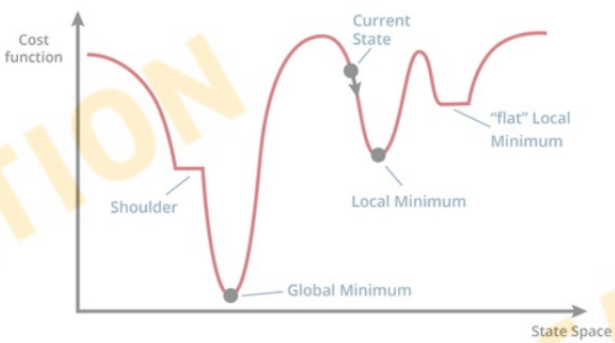
Sometimes it is necessary to go sideways or backwards in order to make progress towards the actual solution.

The term Hill Climbing suggests that we are trying to climb up to the highest hill.

- For minimizing the number of conflicts, it is more natural to think of starting at the top of a ridge and climbing down into the valleys, trying to reach the lowest point.



Hill Climbing may sometimes get stuck in a **Local Minimum** –



where constraints are still violated, but a change to any one variable would result in even more constraints being violated.

- One way to deal with local minima is *Random Re-Start Hill Climbing*. If we detect that we are stuck in a local minimum, we start again from a random initial state.

Hill Climbing for n-Queens from a random starting point will reach a global minimum about 15% of the time, and get stuck in a local minimum the other 85% of the time. After a few random re-starts, it can solve n-Queens in almost constant time for arbitrary n with high probability

Simulated Annealing

Simulated annealing is a method for finding a 'good' solution to an optimization problem.

- Stochastic hill climbing based on difference between the number of constraints violated in the previous state (h0) and new state (h1).
- If h1 < h0, definitely make the change
- Otherwise, make the change with probability $e^{-(h1-h0)/T}$ where T is a "temperature" parameter.
- Reduces to ordinary hill climbing when T = 0
- Becomes totally random search as T = infinite
- Sometimes, we gradually decrease the value of T during the search

Ising Model of Ferromagnetism

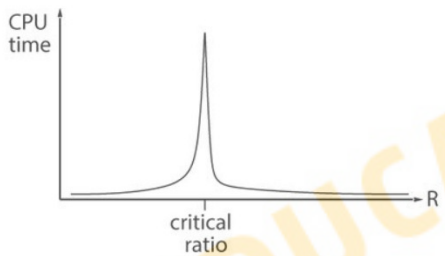
Simulated Annealing is inspired by the Ising Model of Ferromagnetism. An iron bar contains charged ions in a regular lattice, whose spins can be either up or down. If the spins line up with each other, it becomes a magnet.

This can be seen as a CSP where the number of violated constraints is the number of pairs of neighbouring ions with opposite spins. When the temperature is high, the ions flip their spins randomly. As the temperature is lowered, the probability of an ion flipping against its neighbours becomes less and less (given by the Boltzmann equation).

Phase transition in CSP's

Regardless of whether we use backtracking or local search to solve them, randomly-generated CSP's tend to be easy if there are very few or very many constraints. They become extra hard in a narrow range of the ratio:

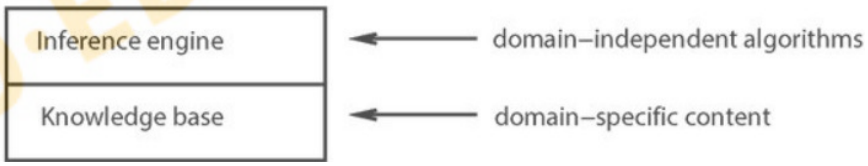
$$R = \frac{number\ of\ constraints}{number\ of\ variables}$$



Week 9

Knowledge Bases

- A Knowledge Base is a set of sentences in a formal language.
- It takes a Declarative approach to building an agent (or other system):
- Tell the system what it needs to know, then it can Ask itself what to do.
- Answers should follow from the knowledge base.



Knowledge Based Agent

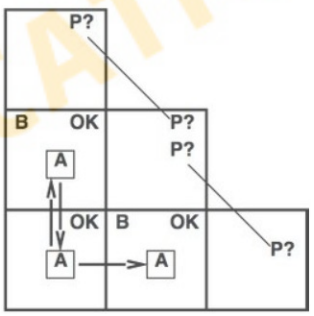
The agent must be able to:

- represent states, actions, etc.
- incorporate new percepts
- update internal representations of the world
- deduce hidden properties of the world
- determine appropriate actions

Reasoning about Actions

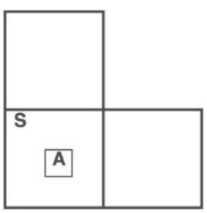
Let's consider this situation.

- If there is a Breeze in (1,2) and (2,1), then there are no safe actions.
- Assuming that pits are uniformly distributed, a pit is more likely in (2,2) than in (3,1).
- How much more likely?



Reasoning about Future States

- If there is a Smell in (1,1), there is no safe square to move into.
- However, we can also use logic to reason about future states.
- In this case, we can use a strategy of coercion:
- Shoot straight ahead
    - Wumpus was there ⇒ dead ⇒ safe
    - Wumpus wasn't there ⇒ safe



Logic In General

- Logics are formal languages for representing information such that conclusions can be drawn.
- Syntax defines what sentences are allowed in the language.
- Semantics define the "meaning" of sentences; i.e. define whether a sentence is true or false in a world.

For example, in the language of arithmetic:

- $x + 2 \geq y$ is a well-formed sentence; $x2 + y >$ is not a well-formed sentence
- $x + 2 \geq y$ is true if the number x +2 is no less than the number y
- $x + 2 \geq y$ is true in a world where $x = 7, y = 1$
- $x + 2 \geq y$ is false in a world where $x = 0, y = 6$

Entailment

Entailment means that one thing follows from another: **KB |= a**

Knowledge base KB entails sentence a if and only if a is true in all worlds where KB is true.

- e.g. the KB containing "the Moon is full" and "the tide is high" entails "Either the Moon is full or the tide is high".
- e.g. $x + y = 4$ entails $4 = x + y$

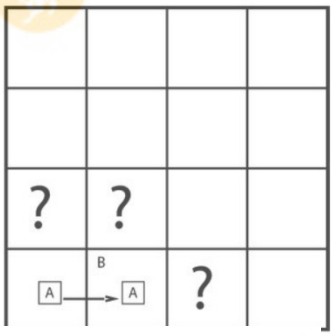
Entailment is a relationship between sentences (i.e. syntax) that is based on semantics.

Models

- Logicians typically think in terms of models, which are formally structured worlds with respect to which truth can be evaluated.
- We say *m* is a model of a sentence *α* if α is true in *m*
- M(a) is the set of all models of *α*
- Then KB|= a if and only if M(KB) ∈ M(a)

Entailment in the Wumpus World

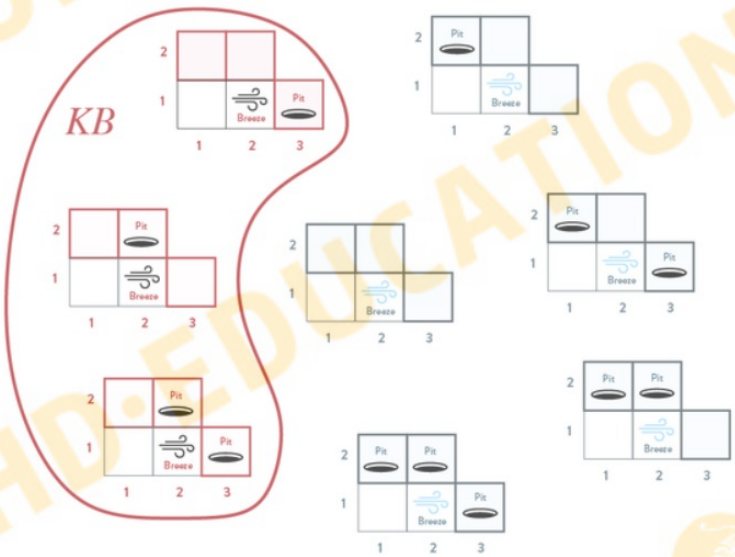
Consider that you are starting an episode of Wumpus World. You do not detect anything in the bottom left square (1,1) so you move right to square (2,1), where you detect a Breeze.



- We know that the Wumpus is not in those squares: (There is no smell)
- We cannot yet determine whether or not the gold is there: (You'd have to be in the square itself to detect the glitter.)
- Therefore we consider only the pits.

Wumpus Combinations

In each question mark, either there is a pit, or there isn't one. Each one is a Boolean choice, so this results in 2 x 2 x 2 = 8 possible combinations:



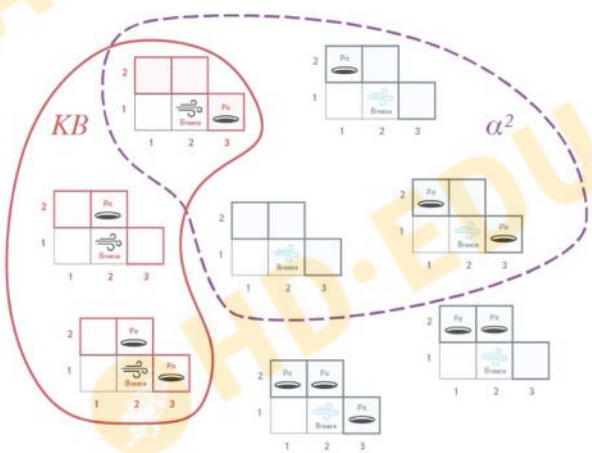
Wumpus Models

The Agent has observed that there is a Breeze in Square (2,1) but No Breeze in Square (1,1). These observations, along with the Wumpus World rules, form its Knowledge Base. Only 3 of the 8 combinations are consistent with the KB. We refer to these three combinations as models of the KB.

Models and Entailment

Consider the proposition a_1= "[1,2] is safe". Note that the models in the KB are a subset of the models of a_1. Therefore, every combination which satisfies the KB also satisfies a_1 . We say that the KB entails a_1, written KB |= a_1.

Now consider the proposition a_2 "[2,2] is safe". In this case, the models in the KB are not a subset of the models of a_2. There are some models of the KB that are not models of a_2. We therefore say that the KB does not entail a_2, written KB !| = a_2.



Syntax

Propositional logic is the simplest logic. It illustrates basic ideas. The proposition symbols P1, P2 etc. are sentences.

- If S is a sentence, ~S is a sentence (negation)
- If S_1 and S_2 are sentences, S_1 ^ S_2 is a sentence (conjunction)
- If S_1 and S_2 are sentences, S_1 v S_2 is a sentence (disjunction)
- If S_1 and S_2 are sentences, S_1 -> S_2 is a sentence (implication)
- If S_1 and S_2 are sentences, S_1 <-> S_2 is a sentence (biconditional)

Semantics

Each model specifies TRUE / FALSE for each proposition symbol. For example, if there are Pits in (1,2) and (2,2) but not (3,1) we would have the following assignments:

$P_{1,2}$ = TRUE $P_{2,2}$ = TRUE $P_{3,1}$ = FALSE

Propositional Logic: Semantics

Rules for evaluating truth with respect to a model m :

¬ S	is TRUE iff	S	is FALSE		
S_1 ^ S_2	is TRUE iff	S_1	is TRUE and	S_2	is TRUE
S_1 v S_2	is TRUE iff	S_1	is TRUE or	S_2	is TRUE
S_1 -> S_2	is TRUE iff	S_1	is FALSE or	S_2	is TRUE
S_1 -> S_2	is FALSE iff	S_1	is TRUE and	S_2	is FALSE
S_1 <-> S_2	is TRUE iff	S_1	is TRUE and	S_2	is TRUE

Simple recursive process evaluates an arbitrary sentence, e.g.

~$P_{1,2}$ ^ ($P_{2,2}$ v $P_{3,1}$) = TRUE ^ (FALSE V TRUE) = TRUE ^ TRUE = TRUE

Truth Tables for Connectives

For example:

- Let P be the proposition "Fred is served alcohol".
- Let Q be the proposition "Fred is over 18 years old".
- Then, P-> is the statement:

"If Fred is served alcohol, then he must be over 18."

P	Q	¬ P	P ∧ Q	P ∨ Q	P ⇒ Q
F	F	T	F	F	T
F	T	T	F	T	T
T	F	F	F	T	F
T	T	F	T	T	T

- However, being served alcohol does not somehow make Fred magically become over 18. Instead, this should be thought of as a kind of rule or regulation. This regulation would only be violated in the case where Fred is served alcohol but is not over 18, i.e. when P is TRUE but Q is FALSE.
- Note also that P-> Q is equivalent to ~Q->~P. This is known as the contrapositive statement:
  - "If Fred is not over 18, then he must not be served alcohol."
- However, it is not the same as Q->P , which is known as the converse statement.

Wumpus World Sentences

- Let $P_{i,j}$ be true if there is a Pit in [i,j] .
- Let $B_{i,j}$ be true if there is a Breeze in [i,j].

So we get sentences like: ~$P_{1,1}$, ~$B_{1,1}$, $B_{2,1}$

"Pits cause breezes in adjacent squares" $B_{1,1}$ <-> ($P_{1,2}$ v $P_{2,1}$), $B_{2,1}$ <-> ($P_{1,1}$ v $P_{2,2}$ v $P_{3,1}$)

"A square is breezy if and only if there is an adjacent pit":

Logical Equivalence and Inference Rules

Two sentences are logically equivalent if they are true in same models:

α ≡ β if and only if α |= β and β |= α

Logical Equivalence Rules:

commutativity:	$p \wedge q \Leftrightarrow q \wedge p$	$p \vee q \Leftrightarrow q \vee p$
associativity:	$p \wedge (q \wedge r) \Leftrightarrow (p \wedge q) \wedge r$	$p \vee (q \vee r) \Leftrightarrow (p \vee q) \vee r$
distributivity:	$p \wedge (q \vee r) \Leftrightarrow (p \wedge q) \vee (p \wedge r)$	$p \vee (q \wedge r) \Leftrightarrow (p \vee q) \wedge (p \vee r)$
implication:	$(p \Rightarrow q) \Leftrightarrow (\neg p \vee q)$	
idempotent:	$p \wedge p \Leftrightarrow p$	$p \vee p \Leftrightarrow p$
double negation:	$\neg\neg p \Leftrightarrow p$	
contradiction:	$p \wedge \neg p \Leftrightarrow \text{FALSE}$	
excluded middle:		$p \vee \neg p \Leftrightarrow \text{TRUE}$
de Morgan:	$\neg(p \wedge q) \Leftrightarrow (\neg p \vee \neg q)$	$\neg(p \vee q) \Leftrightarrow (\neg p \wedge \neg q)$

Inference Rules:

generalization:	$p \Rightarrow p \vee q$
specialization:	$p \wedge q \Rightarrow p$

Validity and Satisfiability

- A sentence is valid if it is true in all models,

- e.g. TRUE, A v ~A, A->A, (A ^ (A-> B)) -> B
- Validity is connected to inference via the Deduction Theorem:
  - KB |= a if and only if (KB->a) is valid
- A sentence is satisfiable if it is true in some model(s)
  - e.g. (AvB) ^ C
  - The models which satisfy this sentence are: {A,C}, {B,C}, {A,B,C}
- A sentence is unsatisfiable if it is true in no models
  - e.g. A v ~A

Satisfiability is connected to inference via the following:

- KB |= a if and only if (KB ^ ~a) is unsatisfiable
- i.e. prove a by reductio ad absurdum.

Inference

When the total number of combinations is small, we can check all of them one by one, as in the previous examples. However, as the number of variables increases, the number of combinations grows exponentially, so we need to have a more efficient method of determining which sentences are entailed by the KB. Such a method is sometimes referred to as a logical procedure.

- KB|-$_i$ a means that sentence a can be derived from KB by logical procedure i
- **Soundness:** logical procedure i is sound if whenever KB|-$_i$, it is also true that KB |= a
- **Completeness:** procedure i is complete if whenever KB |= a, it is also true that KB|-$_i$
- In general, a huge number of sentences are entailed by the KB, and we need to do some clever "searching" in order to derive the particular a that we are looking for. (This procedure has been likened to finding a needle in a haystack).
- We will explore one logical procedure – known as Resolution – in detail, and briefly discuss a couple of others.

Converting to Conjunctive Normal Form

In order to apply Resolution, we must first convert the KB into Conjunctive Normal Form (CNF).

- This means that the KB is a conjunction of clauses, and each clause is a disjunction of (possibly negated) literals. e.g. (A v ~B) ^ (B v ~C v ~D)
- Conjunctive normal form does not have nested brackets, and inside the brackets there cannot be any 'ands' (^). All brackets must be joined together with ands

Any KB can be converted to CNF by the following steps:

Original KB:

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

Step 1: Eliminate $\Leftrightarrow$, replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$

$$(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$$

Step 2: Eliminate $\Rightarrow$, replacing $\alpha \Rightarrow \beta$ with $\neg \alpha \vee \beta$

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg (P_{1,2} \vee P_{2,1}) \vee B_{1,1})$$

Step 3: Move $\neg$ inwards using de Morgan's rules and double-negation:

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1})$$

Step 4: Apply distributivity law ($\wedge$ over $\vee$) and flatten:

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1})$$

Resolution

Once the KB has been converted into Conjunctive Normal Form, we can derive new sentences by repeatedly applying the Resolution Inference Rule:

Suppose we have two disjunctive clauses l_1 v … v l_k and m_1 v …. V m_n

Suppose further that these clauses contain literals l_i and m_j which are complementary, i.e. l_i = P and m_j = ~P, or l_i = ~P and m_j = P , for some P.

We can then derive a new clause by eliminating l_i, m_j and combining all the other literals, i.e.

$$\frac{l_1 \vee \ldots \vee l_i \vee \ldots \vee l_k, \quad m_1 \vee \ldots \vee m_j \vee \ldots \vee m_n}{l_1 \vee \ldots \vee l_{i-1} \vee l_{i+1} \vee \ldots \vee l_k \vee m_1 \vee \ldots \vee m_{j-1} \vee m_{j+1} \vee \ldots \vee m_n}$$

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

For example:

Resolution is sound and complete for Propositional Logic.

Wumpus World Example

Use resolution to prove that if there is a Breeze in (1,1), there must also be a Breeze in (2,2), i.e. prove from the following (which has already been converted to CNF, as above) :

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1}) \wedge (\neg P_{1,2} \vee B_{2,2}) \wedge (\neg P_{2,1} \vee B_{2,2})$$

Answer: We look for pairs of clauses in the KB which contain complementary literals, and combine them with resolution to derive new clauses:

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

The last clause is equivalent to $\sim B_{1,1} \vee B_{2,2}$

Note that each resolution adds a new clause to the KB. We can continue to use the old clauses as well as the new ones.

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

Proof Methods

We now have two methods for checking whether or not a particular clause is a logical consequence of the sentences in a KB.

The first method is called Truth Table Enumeration, where we simply list all possible combinations of truth values, identify which of them are models of the KB, and check each model to see whether or not is TRUE in that model. The time taken for this method is proportional to 2^n where n is the number of distinct symbols in the KB.

Resolution gives us an alternative method which is generally somewhat faster than Truth Table Enumeration:

1. convert the KB into Conjunctive Normal Form,
2. add the negative of the clause you are trying to prove,
3. continually apply a series of resolutions until either
   a) you derive the empty clause, or
   b) there are no more pairs of clauses to which resolution can be applied
   c) You should work through the Activity at the end of this page to see how this process works in detail.

Although faster that Truth Table Enumeration, Resolution is still a relatively slow method. It can be compared to the Uninformed Path Search methods. It is sound and complete, but it generates new clauses indiscriminately, and can still take an exponentially long time as the number of variables increases.

There are other, more efficient, proof methods, which take account of the similarity or "proximity" between the current clause and the "goal" clause a. These methods can thus be thought of in analogy with the Informed Path Search methods.

Horn Clauses

Model Checking can be done more efficiently if the clauses in the KB all happen to be in a special form – for example they may all be Horn Clauses. This means that each clause is written as an implication involving only positive literals, in the form:

(conjunction of symbols) ⇒ symbol

For example: C ∧ (B ⇒ A) ∧ (C ∧ D ⇒ B)

Deduction with Horn Clauses can be done by Modus Ponens:

$$\frac{p_1, \ldots, p_n, \quad p_1 \wedge \ldots \wedge p_n \Rightarrow q}{q}$$

Efficient Proof Methods, using Horn Clauses, can generally be divided into Forward Chaining and Backward Chaining.

Forward Chaining

Look for a rule $p_1 \wedge .. \wedge p_n \rightarrow q$ such that all the clauses on the left hand side are already in the KB. Apply this rule, and add q to the KB.

- Repeat this process until the goal clause a has been derived (or we run out of rules to apply).

Backward Chaining

Backward Chaining instead maintains a list of subgoals that it is trying to prove. Initially, this list consists of the ultimate goal a.

Choose a clause q from the list of subgoals.

- check if q is known already
- otherwise, find a rule with q on the right side and add clauses from the left side of this rule as new subgoals
- check to make sure each new subgoal is not on the list already, and has not already been proved, or failed.

Forward 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 get into a PhD program?

Satisfiability as Constraint Satisfaction

Consider the following problem:

You are given a KB, perhaps in a special form such as 3-CNF (this means Conjunctive Normal Form, with at most three literals in each clause). Does there exist any assignment of truth values to the symbols which will make all of the clauses in the KB TRUE?

For example, is there an assignment of truth values to A,B,C,D,E which will make the following TRUE?

$$(\neg D \vee \neg B \vee C) \wedge (B \vee \neg A \vee \neg C) \wedge (\neg C \vee \neg B \vee E) \wedge (E \vee \neg D \vee B) \wedge (B \vee E \vee \neg C$$

This provides a classic example of a Constraint Satisfaction Problem, and it can be solved by methods such as Hill Climbing or Simulated Annealing.

If the KB is randomly generated, the difficulty of finding a solution generally depends on the ratio m/n of the number of clauses m divided by the number of distinct symbols n . For example, suppose n= 50 and the KB is in 3-CNF. The problem becomes especially difficult when the ratio m/n is approximately 4.3. If the ratio is smaller than this threshold, the problem generally becomes easy because it is under-constrained. If the ratio is larger, the problem again becomes easy, this time because it is over-constrained.

Other Constraint Satisfaction Problems like n-Queens are sometimes converted to 3-CNF, as a way of measuring whether they are under- or over-constrained.

Limitations of Propositional Logic

"A square is breezy if and only if there is an adjacent pit."

This statement must be converted into a separate sentence for each square:

$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$, $B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$, etc.

What we really want is a way to express such a statement in one sentence for all squares, e.g.

$$\text{Breezy}(i,j) \Leftrightarrow (\text{Pit}(i-1,j) \vee \text{Pit}(i+1,j) \vee \text{Pit}(i,j-1) \vee \text{Pit}(i,j+1)$$

First-Order Logic will allow us to do this. It extends Propositional Logic by introducing variables, predicates, functions and quantifiers.

Syntax of First Order Logic

- Objects: people, houses, numbers, theories, colors, football games, wars, centuries ...
- Predicates: red, round, bogus, prime, multistoried, ... brother of, bigger than, inside, part of, has color, occurred after, owns, comes between, ...
- Functions: father of, best friend, third inning of, one more than, ...
- Constants Gold, Wumpus, [1,2], [3,1], etc.
- Predicates Adjacent(), Smell(), Breeze(), At()
- Functions Result(), Mother()
- Variables x,y,a,t
- Connectives ^ V ~ -> <->
- Equality =

- Quantifiers ∃ ∀

Sentences

Atomic sentence = predication($term_1$,…,$term_n$) or $term_1$ = $term_2$
Term = function($term_1$,…,$term_n$) or a constant or variable

- Eg. At(Agent, [1,1], S0)
- Holding(Gold, S5)

Complex sentences are made from atomic sentences using connectives

$$\neg S,\ S_1 \wedge S_2,\ S_1 \vee S_2,\ S_1 \Rightarrow S_2,\ S_1 \Leftrightarrow S_2$$

Eg. Pit(x) ^ Adjacent(x,y)

Universal Quantification

$$\begin{aligned}&(\text{Glitter}([1,1]) \Rightarrow \text{At}(Gold,[1,1]))\\ \wedge\ &(\text{Glitter}([1,2]) \Rightarrow \text{At}(Gold,[1,2]))\\ \wedge\ &(\text{Glitter}([1,3]) \Rightarrow \text{At}(Gold,[1,3]))\\ \wedge\ &\ldots\end{aligned}$$

- ∀ {Variables}{Sentence}
  - Where there's glitter, there's gold: ∀x Glitter(x) -> At(Gold,x) =
- ∀xP is equivalent to the conjunction of instantiations of P
- Typically, -> is the main connective with ∀
- Common mistake: using ^ as the main connective with ∀
  - Eg. ∀x Glitter(x) ^ At(Gold,x) means "There is Glitter everywhere and Gold everywhere."

Existential Quantification

$$\begin{aligned}&(\text{Sheep}(Dolly) \wedge \text{Black}(Dolly))\\ \vee\ &(\text{Sheep}(Lassie) \wedge \text{Black}(Lassie))\\ \vee\ &(\text{Sheep}(Skippy) \wedge \text{Black}(Skippy))\\ \vee\ &\ldots\end{aligned}$$

- ∃{Variables}{Sentence}
- **Some sheep are black:** ∃x Sheep(x) ^ Black(x) =
- ∃xP is equivalent to the disjunction of instantiations of P
- **Typically ^ is the main connective with ∃**
- **Common mistake: using -> as the main connective with ∃**
  - **Eg.** ∃x Sheep(x) -> Black(x) is true if there is anyone who is not at sheep

Properties of Quantifiers

- ∀x∀y is the same as ∀y∀x
- ∃x∃y is the same as ∃y∃x
- ∃x∀y is not the same as ∀y∃x

∃x∀y Loves(x,y) = "There is a person who loves everyone in the world"
∀y∃x Loves (x, y) = "Everyone in the world is loved by at least one person"

Quantifier duality: each can be expressed using the other

$$\forall x\ \text{Likes}(x, IceCream) \Leftrightarrow \neg\exists x\ \neg\text{Likes}(x, IceCream)$$

$$\exists x\ \text{Likes}(x, Broccoli) \Leftrightarrow \neg\forall x\ \neg\text{Likes}(x, Broccoli)$$

Deducing Hidden Properties

Properties of locations:

$$\forall x,t\ \text{At}(Agent,x,t) \wedge \text{Smell}(t) \Rightarrow \text{Smelly}(x)$$
$$\forall x,t\ \text{At}(Agent,x,t) \wedge \text{Breeze}(t) \Rightarrow \text{Breezy}(x)$$
$$\forall x,t\ \text{At}(Agent,x,t) \wedge \text{Glitter}(t) \Rightarrow \text{AtGold}(x)$$

Squares are breezy near a pit:

- Causal rule – infer effect from cause : $\forall x,y\ \text{Pit}(x) \wedge \text{Adjacent}(x,y) \Rightarrow \text{Breezy}(y)$
- Diagnostic rule – infer cause from effect: $\forall y\ \text{Breezy}(y) \Rightarrow \exists x\ \text{Pit}(x) \wedge \text{Adjacent}(x,y)$

Definition for the Breezy predicate (combines Causal and Diagnostic): $\forall y\ \text{Breezy}(y) \Leftrightarrow \exists x\ \text{Pit}(x) \wedge \text{Adjacent}(x,y)$

Keeping Track of Change

Facts hold only in certain situations, not universally eg. Holding(Gold, Now) rather than Holding(Gold)

Situation calculus is one way to represent change:

- Adds a situation argument to each non-universal predicate

- e.g. Now denotes a situation in Holding(Gold,Now)

Situations are connected by the Result() function.

Result(a,s) is the situation that results from doing action a in state s .

Describing Actions

We can plan a series of actions in a logical domain in a manner analogous to the Path Search algorithms discussed in Weeks 3 and 4. But, instead of the successor state being explicitly specified, we instead need to deduce what will be true and false in the state resulting from the previous state and action:

"Effect" axiom – describe changes due to action $\forall s\ \text{AtGold}(s) \Rightarrow \text{Holding}(Gold, \text{Result}(Grab, s))$

Planning actions in the real world raises a number of potential problems.

- **Frame problem**: Some facts will change as a result of an action, but many more will stay as they were. For example, grabbing the gold will not change the fact that we are holding the arrow. We could add an explicit rule for this: $\forall s\ \text{Holding}(Arrow, s) \Rightarrow \text{Holding}(Arrow, \text{Result}(Grab, s))$
  However, adding too many of these frame axioms can make the process unmanageable. For example, if a cup is red, and you turn it upside down, it is still red. But, if a cup is full of water, and you turn it upside down, it is no longer full of water. Large-scale expert systems of the 1980's often failed because of their inability to encode this kind of "commonsense" reasoning in explicit rules.
- **Qualification problem**: Normally, we expect actions to have a certain effect. But, in the real world there could be endless caveats. What happens if the gold is slippery, or nailed down, or too heavy, or you can't reach it, etc.
- **Ramification problem**: Real actions have many secondary consequences. Every time we take a step, our clothes and shoes get a little more worn out. When we are driving, the rubber on our tyres gets gradually worn down. Most of the time we ignore these tiny effects. But eventually we will need to replace our tyres in order to avoid having an accident.

In general, we assume that a fact is true if a rule tells us that an action made it true, or if it was true before and no action made it false.

Searching for a Situation

Initial condition in KB:

At(Agent, [1,1], S_0)

At(Gold, [1,2], S_0)

Query: $\text{Ask}(\text{KB}, \exists s\ \text{Holding}(Gold, s))$ i.e., in what situation will I be holding the gold?

Answer: $s = \text{Result}(Grab, \text{Result}(Forward, S_0))$ i.e., go forward and then grab the gold

This assumes that the agent is interested in plans starting at S_0 and that S_0 is the only situation described in the KB.

Searching for a Plan of Actions

- We can represent a plan as a sequence of actions $[a_1, a_2, \ldots, a_n]$
- PlanResult(p,s) is the result of executing plan p, starting from state s.
- Then the query $\text{Ask}(\text{KB}, \exists p\ \text{Holding}(Gold, \text{PlanResult}(p, S_0)))$ has the solution $p = [Forward, Grab]$.

We can define PlanResult recursively in terms of Result:

$\forall s\ \text{PlanResult}([], s) = s$

$\forall a, p, s\ \text{PlanResult}([a \mid p], s) = \text{PlanResult}(p, \text{Result}(a, s))$

Planning systems are special-purpose reasoners designed to do this type of inference more efficiently than a general-purpose reasoner.

Week 10

Uncertainty

There are many situations where an agent has to choose an action based on incomplete information.

This is true, for example if the environment is stochastic. In the game of Backgammon, we do not know what the next dice roll will be.

Another reason is partial observability. Some aspects of the environment may be hidden from the agent altogether. In addition, robot sensors are often noisy in the sense that the quantities observed by the robot differ to some extent from their "true" values (due to factors which the agent is unable to observe).

Planning under Uncertainty

When planning in an uncertain environment, our choice of action may depend on the level of risk we are willing to take.

- In order to understand this, let's consider the common problem of trying to determine how to get to the airport on time for a flight.
- Let action A_t = leave for airport t minutes before flight. Will A_t get me there on time?

What are some of the problems?

- Partial observability, noisy sensors
- Uncertainty in action outcomes (flat tyre, etc.)
- Immense complexity of modelling and predicting traffic

Hence a purely logical approach either (a) risks falsehood or (b) leads to conclusions that are too weak for decision making.

a) "A_{45} will get me there on time", or
b) "A_{30} will get me there on time if there's no accident on the bridge and it doesn't rain and my tyres remain intact, etc, etc." (A_{1440} might be safe but I'd have to stay overnight in the airport ...)

Methods for handling Uncertainty

1. **Default or Nonmonotonic Logic:**
   - Assume that my car does not have a flat tire, or any other problems.
   - Assume that A45 works unless contradicted by evidence.
   - Issues:
     - Which assumptions are reasonable?
     - How do you handle contradiction?
2. **Probability:**
   - Given the available evidence, A_{30} will get me there on time with probability 0.04

Probability

Probabilistic assertions summarise the effects of:
Laziness: failure to enumerate exceptions, qualifications, etc.
Ignorance: lack of relevant facts, initial conditions, etc.

Subjective or Bayesian probability:

Probabilities relate propositions to one's own state of knowledge e.g. $P(A_{30}|\text{no reported accidents}) = 0.06$

These are NOT claims of a "probabilistic tendency" in the current situation (but might be learned from past experience of similar situations)

Probabilities of propositions change with new evidence: e.g. $P(A_{30}|\text{no reported accidents, 5 a.m.}) = 0.15$
(Analogous to logical entailment status $KB \models \alpha$, not absolute truth)

Making Decisions Under Uncertainty

Suppose I believe the following:

P(A30 gets me there on time|...) = 0.04
P(A90 gets me there on time|...) = 0.70
P(A120 gets me there on time|...) = 0.95
P(A1440 gets me there on time|...) = 0.9999

The actions you choose depend upon your preferences. For instance, is eating a great meal at the airport more important than catching your flight?

Utility theory - used to represent and infer preferences

Decision theory = utility theory + probability theory

This is part of the reason why manned space missions are so much more expensive than un-manned missions. If we want to reduce the probability of failure from, say, 0.05 to 0.01, it might cost us billions of additional dollars.

Probability Basics

- Begin with a set Ω – the sample space (e.g. 6 possible rolls of a die)
- ω ∈ Ω is a sample point/possible world/atomic event
- A probability space or probability model is a sample space with an assignment P(ω) for every ω ∈ Ω s.t.
- 0 ≤ P(ω) ≤ 1
    - Eg. P(1) = P(2) = P(3) = P(4) = P(5) = P(6) = 1/6
- An event A is any subset of Ω P(A) = $\sum_{\{\omega \in A\}} P(\omega)$
    - Eg. P(die roll <4) = P(1) + P(2) + P(3) = 1/6 + 1/6 + 1/6 = ½

Random Variables

A random variable (r.v.) is a function from sample points to some range (e.g. the Reals or Booleans)

For example, Odd(3) = true.

P induces a probability distribution for any r.v. X:

$$P(X = x_i) = \sum_{\{\omega : X(\omega) = x_i\}} P(\omega)$$

Eg. $P(\texttt{Odd} = \texttt{true}) = P(1) + P(3) + P(5) = \frac{1}{6} + \frac{1}{6} + \frac{1}{6} = \frac{1}{2}$

Propositions

Think of a proposition as the event (set of sample points) where the proposition is true

Given Boolean random variables A and B:

$$event\ \ a = set\ \ of\ \ sample\ \ points\ \ where\ \ A(\omega) = \texttt{true}$$

$$event\ \ \neg a = set\ \ of\ \ sample\ \ points\ \ where\ \ A(\omega) = \texttt{false}$$

$$event\ \ a \wedge b = points\ \ where\ \ A(\omega) = \texttt{true}\ \ and\ \ B(\omega) = \texttt{true}$$

With Boolean variables, sample point = propositional logic model e.g., A = true, B = false, or a ^ ~b

Proposition = disjunction of atomic events in which it is true

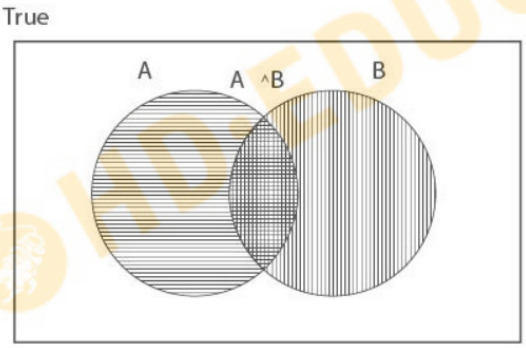
$$(a \vee b) \equiv (\neg a \wedge b) \vee (a \wedge \neg b) \vee (a \wedge b)$$

e.g., $\rightarrow P(a \vee b) = P(\neg a \wedge b) + P(a \wedge \neg b) + P(a \wedge b)$

So Why Use Probability?

$$P(a \vee b) = P(a) + P(b) - P(a \wedge b)$$

The definitions imply that certain logically related events must have related probabilities.



An agent who bets according to probabilities that violate these axioms can be forced to bet so as to lose money regardless of outcome.

Syntax for Propositions

Propositional or Boolean random variables

e.g., Cavity (do I have a cavity in my tooth, which needs a filling from the dentist?)

- "Cavity = true" is a proposition, also written cavity
- Discrete random variables (finite or infinite)

e.g., Weather is one of (sunny, rain, cloudy, snow)

- "Weather = rain" is a proposition
- Values must be exhaustive and mutually exclusive
- Continuous random variables (bounded or unbounded)

e.g. Temp = 21.6; also allow, e.g. Temp < 22.0
Arbitrary Boolean combinations of basic propositions

Prior Probability
Prior or unconditional probabilities of propositions

e.g. P(Cavity = true) = 0.2, and P(Weather = sunny) = 0.72
- correspond to belief prior to arrival of any (new) evidence.

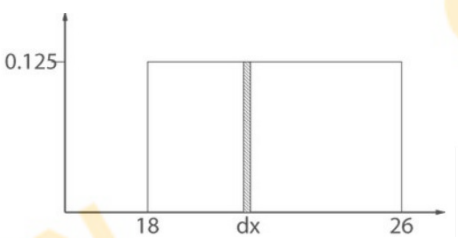
Probability distribution gives values for all possible assignments:
P(weather) = <0.72, 0.1, 0.008, 0.1> (normalized, i.e., sums to 1)

Joint Probability
Joint probability distribution for a set of r.v.'s gives the probability of every atomic event on those r.v's (i.e., every sample point)

Weather =	*sunny*	*rain*	*cloudy*	*snow*
Cavity = `true`	0.144	0.02	0.016	0.02
Cavity = `false`	0.576	0.08	0.064	0.08

P(Weather, Cavity) is a 4 ×2 matrix of values:
Every question about a domain can be answered by the joint distribution because every event is a sum of sample points.

Probability for Continuous Variables
Express distribution as a parameterized function.



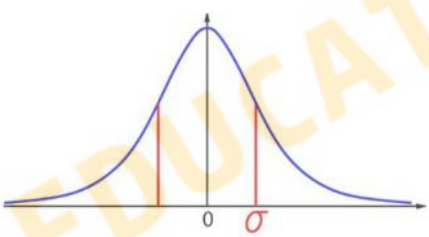
e.g P(X = x) = U[18,26](x) = uniform density between 18 and 26

Here P is a density; integrates to 1.
P(X=20.5) = 0.125 really means

$$\lim_{dx \to 0} P(20.5 \leq X \leq 20.5 + dx)/dx = 0.125$$

Gaussian Density

$$P(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu)^2/2\sigma^2}$$



Probabilistic Agents
We consider an Agent whose World Model consists not of a set of facts, but rather a set of probabilities of certain facts being true, or certain random variables taking particular values.

When the Agent makes an observation, it may update its World Model by adjusting these probabilities, based on what it has observed.

Example: Tooth Decay
Let's assume you live in a place where people eat a lot of sugar and don't take good care of their teeth, so that 20% of the population at any given time have a cavity in one of their teeth which needs a filling from the dentist.

If you consider yourself to be a typical person with a typical diet, you would estimate your own probability of having a cavity as 20%. We write this as P(cavity) = 0.2

Now suppose you wake up one morning with a toothache. Suddenly, you will think it is much more likely that you have a cavity, perhaps as high as 60%. We say that the conditional probability of cavity, given toothache, is , written as follows: P(cavity | toothache) = 0.6

Now suppose that you don't have a toothache, but you just go to the dentist for a regular checkup. The dentist will use a small hook-shaped instrument called a probe, and check whether this probe can catch on the back of your tooth. If it does catch, this information will increase the probability that you have a cavity.

Joint Probability Distribution
We assume there is some underlying joint probability distribution over the three random variables Toothache, Cavity and Catch, which we can write in the form of a table:
Note that the sum of all the entries in the table is 1.0 .

	toothache		¬ toothache	
	catch	¬ catch	catch	¬ catch
cavity	.108	.012	.072	.008
¬ cavity	.016	.064	.144	.576

For any proposition , we can calculate the probability of ϕ by summing the atomic events where ϕ is true:

$$P(\phi) = \sum_{\omega:\omega \models \phi} P(\omega)$$

Given the above joint probability distribution, the probability of a random person having a tooth ache is the sum of all the columns in the 'toothache' section. The probability of toothache v cavity is all the columns in the toothache section and all the rows in the cavity section.

Conditional Probability Defined
If we consider two random variables a and b, with P(b) != 0 , then the conditional probability of a given b, is

$$P(a \mid b) = \frac{P(a \wedge b)}{P(b)} \quad , \text{or} \quad P(a \wedge b) = P(a \mid b)P(b) = P(b \mid a)P(a)$$

When an agent is considering a sequence of random variables at successive time steps, they can be chained together by applying this rule repeatedly:

$$P(X_n, \ldots, X_1) = P(X_n \mid X_{n-1}, \ldots, X_1)P(X_{n-1}, \ldots, X_1)$$

$$= P(X_n \mid X_{n-1}, \ldots, X_1)P(X_{n-1} \mid X_{n-2}, \ldots, X_1) = \ldots = \prod_{i=1}^{n} P(X_i \mid X_{i-1}, \ldots, X_1)$$

Eg. the probability of not having a cavity, if you have a toothache, is P(~cavity| toothache)

	toothache	
	catch	¬ catch
cavity	.108	.012
¬ cavity	.016	.064

Which is $\frac{P(\ cavity^{toothache})}{P(toothache)} = \frac{0.016+0.064}{0.108+0.012+0.016+0.064} = 0.4$

Independent Variables

Let's consider the joint probability distribution for Cavity and Weather.

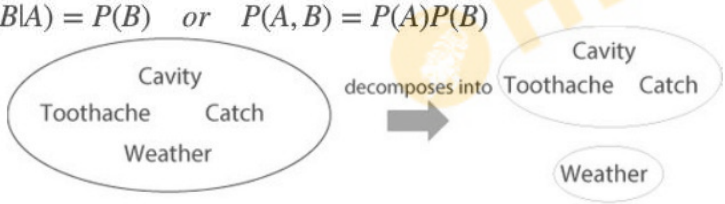
Weather =	*sunny*	*rain*	*cloudy*	*snow*
Cavity = `true`	0.144	0.02	0.016	0.02
Cavity = `false`	0.576	0.08	0.064	0.08

Notice that:

$$P(cavity \mid Weather = sunny) = \frac{0.144}{0.144 + 0.576} = 0.2 = P(cavity)$$

In other words, learning that the Weather is sunny has no effect on the probability of having a cavity (and the same for rain, cloudy and snow). We say that Cavity and Weather are independent variables.

$$P(A|B) = P(A) \quad or \quad P(B|A) = P(B) \quad or \quad P(A, B) = P(A)P(B)$$



Independence

A and B are independent iff

If the variables were not independent, the agent would need to store 32 items in its probability table. But, since Weather is independent from the other variables, the agent only needs to store two smaller probability tables, with a total of 8+4=12 items.

$$P(\text{Toothache}, \text{Catch}, \text{Cavity}, \text{Weather}) = P(\text{Toothache}, \text{Catch}, \text{Cavity})P(\text{Weather})$$

(Note: the number of free parameters is slightly less, because the values in each table must sum to 1).

Conditional Independence

	toothache		¬ toothache	
	catch	¬ catch	catch	¬ catch
cavity	.108	.012	.072	.008
¬ cavity	.016	.064	.144	.576

The variables Toothache, Cavity and Catch are not independent. But, they do exhibit a more subtle relationship known as conditional independence.

If you have a cavity, the probability that the probe will catch is 0.9, no matter whether you have a toothache or not. If you don't have a cavity, the probability that the probe will catch is 0.2, regardless of whether you have a toothache. In other words, $P(\text{Catch} \mid \text{Toothache}, \text{Cavity}) = P(\text{Catch} \mid \text{Cavity})$

We say that Catch is conditionally independent of Toothache given Cavity.

This conditional independence reduces the number of free parameters from 7 down to 5.

For larger problems with many variables, deducing this kind of conditional independence among the variables can reduce the number of free parameters substantially, and allow the Agent to maintain a simpler World Model.

Equivalent Statements:

$$P(\text{Toothache} \mid \text{Catch}, \text{Cavity}) = P(\text{Toothache} \mid \text{Cavity})$$

$$P(\text{Toothache}, \text{Catch} \mid \text{Cavity}) = P(\text{Toothache} \mid \text{Cavity})P(\text{Catch} \mid \text{Cavity})$$

Bayes' Rule

The formula for conditional probability can be manipulated to find a relationship when the two variables are swapped:

$$P(a \wedge b) = P(a \mid b)P(b) = P(b \mid a)P(a) \quad \rightarrow \text{Bayes' rule} \quad P(a \mid b) = \frac{P(b \mid a)P(a)}{P(b)}$$

This is often useful for assessing the probability of an underlying cause after an effect has been observed:

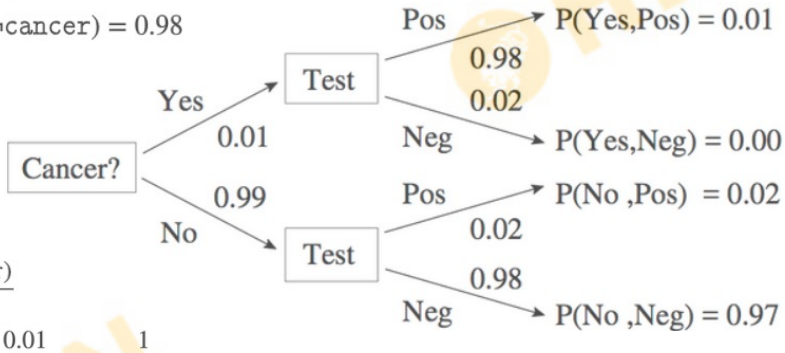
$$P(\text{Cause} \mid \text{Effect}) = \frac{P(\text{Effect} \mid \text{Cause})P(\text{Cause})}{P(\text{Effect})}$$

Example: Medical Diagnosis

Question: Suppose we have a 98% accurate test for a type of cancer which occurs in 1% of patients. If a patient tests positive, what is the probability that they have the cancer?

Answer: There are two random variables: Cancer (true or false) and Test (positive or negative). The probability P(cancer) = 0.01 is called a prior, because it represents our estimate of the probability before we have done the test (or made some other observation). We interpret the statement that the test is 98% accurate to mean:

$$P(\text{positive} \mid \text{cancer}) = 0.98, \quad \text{and} \quad P(\text{negative} \mid \neg\text{cancer}) = 0.98$$



Using Bayes' Rule:

$$P(\text{cancer} \mid \text{positive}) = \frac{P(\text{positive} \mid \text{cancer})P(\text{cancer})}{P(\text{positive})}$$

$$= \frac{0.98 * 0.01}{0.98 * 0.01 + 0.2 * 0.99} = \frac{0.01}{0.01 + 0.02} = \frac{1}{3}$$

Even though the test is 98% accurate, the chance of having the cancer is still relatively low, once we have taken into account the effect of the prior.

Bayes' Rule and Conditional Independence

$$P(\text{cavity}, \text{toothache}, \text{catch}) = P(\text{toothache} \mid \text{catch}, \text{cavity})\, P(\text{catch} \mid \text{cavity}) P(\text{cavity})$$

$$= P(\text{toothache} \mid \text{cavity})\, P(\text{catch} \mid \text{cavity}) P(\text{cavity})$$

This is an example of a naive Bayes model:

$$P(\text{Cause}, \text{Effect}_1, \ldots, \text{Effect}_n) = P(\text{Cause}) \prod_i P(\text{Effect}_i \mid \text{Cause})$$

Total number of parameters is linear in n

Specifying the Probability Model

We will use $B_{i,j}$ to indicate a Breeze in square (i,j) and $P_{i,j}$ to indicate a Pit in square (i,j).

We use known to represent what we know, i.e. $B_{1,2} \wedge B_{2,1} \wedge \neg B_{1,1} \wedge \neg \text{Pit}_{1,2} \wedge \neg \text{Pit}_{2,1} \wedge \neg \text{Pit}_{1,1}$

We use unknown to represent the joint probability of Pits in all the other squares, i.e. $P(\text{Unknown}) = P(\text{Pit}_{1,4}, \ldots, \text{Pit}_{4,1})$

We divide Unknown into Fringe and Other, where P(Fringe) = P($\text{Pit}_{1,3}$, $\text{Pit}_{2,2}$, $\text{Pit}_{3,1}$) and Other is all the other variables. We then have:

$$P(\text{Pit}_{1,3} \mid \text{known}) = \sum_{\text{unknown}} P(\text{Pit}_{1,3}, \text{unknown} \mid \text{known})$$

$$= \sum_{\text{fringe}} \sum_{\text{other}} P(\text{Pit}_{1,3}, \text{fringe}, \text{other} \mid \text{known})$$

$$= \sum_{\text{fringe}} \sum_{\text{other}} P(\text{Pit}_{1,3} \mid \text{fringe}, \text{other}, \text{known})\, P(\text{fringe}, \text{other} \mid \text{known})$$

$$= \sum_{\text{fringe}} P(\text{Pit}_{1,3} \mid \text{fringe}) \sum_{\text{other}} P(\text{fringe}, \text{other} \mid \text{known})$$

$$= \sum_{\text{fringe}} P(\text{Pit}_{1,3} \mid \text{fringe}) \sum_{\text{other}} P(\text{known} \mid \text{fringe}, \text{other}) P(\text{fringe}, \text{other}) / P(\text{known})$$

Note that we have used Bayes' Rule, along with the fact that $P_{1,3}$ is independent of other, given fringe.

Fringe Models

The set of models for fringe which are compatible with the known facts are as follows:

If we denote this set by F, then we can restrict the sum to fringe $\in F$, because P(known | fringe, other) = 0 outside of F. So the conditional probability reduces to:

$$\sum_{\text{fringe}\in F} P(\text{Pit}_{1,3}\,|\,\text{fringe}) \sum_{\text{other}} P(\text{known}\,|\,\text{fringe},\text{other})P(\text{fringe},\text{other})/P(\text{known})$$

Note also that

$$P(\text{known}) = \sum_{\text{fringe}\in F} \sum_{\text{other}} P(\text{known}\,|\,\text{fringe},\text{other})P(\text{fringe},\text{other})$$

This means that other and fringe become independent, and known becomes independent of other, given fringe.

$$P(\text{known}\,|\,\text{fringe},\text{other}) = P(\text{known}\,|\,\text{fringe}) = 1, \quad \text{for fringe} \in F, \text{ so}$$

$$P(\text{known}) = \sum_{\text{fringe}\in F} P(\text{fringe}) = (0.2)^3 + (0.2)^2(0.8) + (0.2)^2(0.8) + (0.2)^2(0.8) + (0.2)(0.8)^2$$

$$= 0.008 + 0.032 + 0.032 + 0.032 + 0.128 = 0.232$$

The numerator includes only those models for which is true, i.e.

$$P(\text{Pit}_{1,3}\,|\,\text{known}) = \frac{0.008 + 0.032 + 0.032}{0.232} = \frac{9}{29} \simeq 0.310$$

In a similar way,

$$P(\text{Pit}_{2,2}\,|\,\text{known}) = \frac{0.008 + 0.032 + 0.032 + 0.128}{0.232} = \frac{25}{29} \simeq 0.862$$

Week 2

Rational Agent Model

Example: finding the right temperature when taking a shower.

The task (the problem) is that the water isn't at a comfortable temperature and the Rational Agent is acting towards the goal of achieving the optimal shower temperature.

- **Agent**: Robot
- **Actions**: Adjusting the hot water tap, or both taps
- **Percepts**: Water temperature
- **Performance** measure: Keeping the temperature close to a desired value
- **Environment**: The shower - taps, pipes, nozzle
- **Actuators**: Limbs & hands

- **Sensors**: Temperature sensors

The robot senses the temperature and may adjust the taps to increase or decrease the water temperature until it reaches a suitable or optimal value.

The PEAS Model

Tasks are the problems to which rational agents are the solutions. In order to define a task, we need to specify four things:

- **Performance Measure**
  - How can we tell if an agent is doing a good or bad job? What are the qualities of performance that we would like to measure?
- **Environment**
  - What are the components/attributes of the environment (which are relevant to the agent)?
- **Actuators**
  - What are the outputs that enable action upon an environment?
- **Sensors**
  - What are the inputs that sense and provide data for the agent?

PEAS Model of Wumpus World

Performance Measure

- +1000 for returning with gold
- -1000 if you die
- -1 for each action
- -10 for shooting the arrow

Environment

- 4x4 Grid, Wumpus, Gold, Pits, Smells, Breezes.
- squares adjacent to a Pit are Breezy
- squares adjacent to the Wumpus are Smelly
- Shoot toward the Wumpus will kill it (uses up the arrow)
- Grab will pick up the gold (if in the same square)

Actuators

- Left, Right, Forward, Grab, Shoot

Sensors

- Breeze, Glitter, Stench

Simulated vs. Situated or Embodied

Simulated: A separate program is used to simulate an environment, feed percepts to agents, evaluate performance, etc.

Situated: The agent acts directly on the actual environment.

Embodied: The agent has a physical body in the world.

- Chess is Simulated; Robocup is both Situated and Embodied.

Static vs. Dynamic

Static: The environment does not change while the agent is thinking.

Dynamic: The environment may change while the agent is thinking.

- Chess is Static; Robocup is Dynamic.
- For example, in Robocup, if the ball is in front of you but you take too long to act, another player may come in and kick it away. In Chess, the environment does not change when it is your turn.
- Note: In a multi-player game, a static environment will obviously change when the opponent moves, but it cannot change while it is “our turn”.

Discrete vs. Continuous

Discrete: There are only a finite (or countable) number of discrete percepts/actions.

Continuous: States, percepts or actions can vary continuously.

- Chess is Discrete; Robocup is Continuous.
- For example, in Robocup, the position of the ball can vary continuously, but in Chess every piece must be on one square or the other, not half-way in between.

Fully Observable vs. Partially Observable

Fully Observable: Agent percept contains all relevant information about the world.

Partially Observable: Some relevant information is hidden from the agent.

- Chess is Fully Observable, RoboCup 2-Legged and 4-Legged League are only Partially Observable.

Deterministic vs. Stochastic

Deterministic: The current state uniquely determines the next state.

Stochastic: There is some random element involved.

- Chess is Deterministic; Robocup is Stochastic.
- Note: The non-determinisim partly arises because the physics can only be modelled with limited precision. However, even if it could be modelled perfectly, there would still be randomness due to quantum mechanical effects.

Episodic vs. Sequential

Episodic: Every action by the agent is evaluated independently.

Sequential: The agent is evaluated based on a long sequence of actions.

- Both Chess and Robocup are considered Sequential, because evaluation only happens at the end of a game, and it is necessary to plan several steps ahead in order to play the game well.
- One example of an Episodic task would be scanning a medical image of a patient and predicting whether the patient has cancer. In this case, every image is independent of the previous image.

Known vs. Unknown

Known: The rules of the game, or physics/dynamics of the environment, are known to the agent.

Unknown: The rules of the game, or the "world model", are not fully known to the agent.

- Be careful not to confuse "Known" with "Observable".
- Both Chess and Robocup are Known.
- By contrast, video games like Pong, PacMan or Infinite Mario are sometimes set up in such a way that the dynamics of the environment are Unknown to the agent.

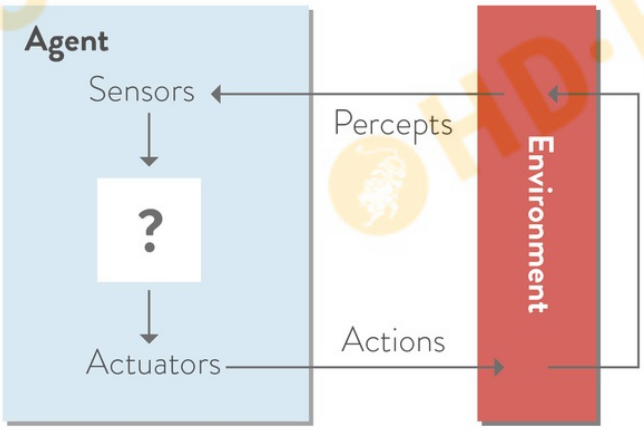
Single-Agent vs. Multi-Agent

Both Chess and Robocup are Multi-Agent.

Agent Types

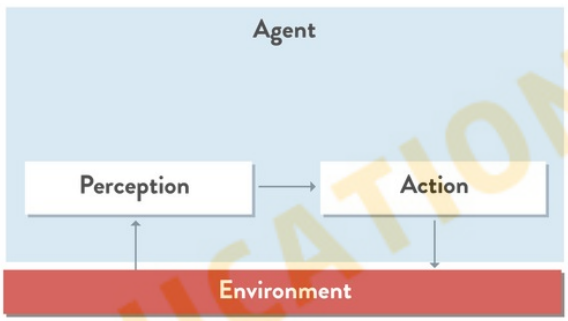
In this course we will consider five different types of agent:

- Reactive Agent
- Model-Based Agent
- Planning Agent
- Game Playing Agent
- Learning Agen



Different agent types are distinguished according to what type of processing occurs between the Sensors and Actuators.

1. Reactive Agent

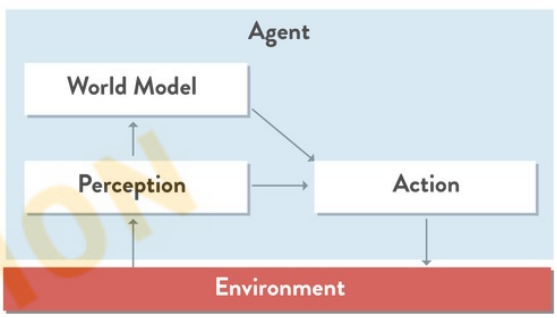


- Reactive agents choose their next action based only on what they currently perceive, using a “policy” or set of rules which are simple to apply.
- Consider the task of navigating your way out of a maze. The best strategy for a Reactive Agent is called the "Right Hand Rule", i.e. put your right hand on the wall and follow it around until you (hopefully) get out of the maze.
- This works in a lot of situations, but if there is an "island" in the maze, you can keep going around in circles forever.

Limitations of Reactive Agents

- Reactive Agents have no memory or “state”
- They are unable to base decision on previous observations
- They may repeat the same sequence of actions over and over

2. Model-Based Agent



- A Model-Based Agent can keep a "map" of the places it has visited, and remember what it perceived there. It can store this information in a World Model.
- In the Maze example, the agent can build a map of the places it has previously visited, and make sure not to explore the same path twice.

Limitations of Model-Based Agents

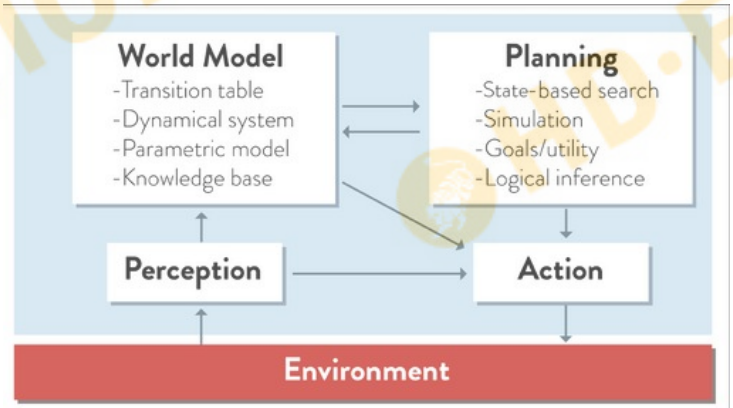
Sometimes we may need to plan several steps into the future. An agent with a world model but no planning can look into the past, but not into the future (or, perhaps one step into the future, but no further than that). As a result, it will perform poorly when the task requires any of the following:

- Searching several moves ahead (Chess, Rubik’s cube)
- Completing complex tasks requiring many individual steps (cooking a meal, assembling a watch)
- Applying logical reasoning to achieve goals (travel to New York)

3. Planning Agent

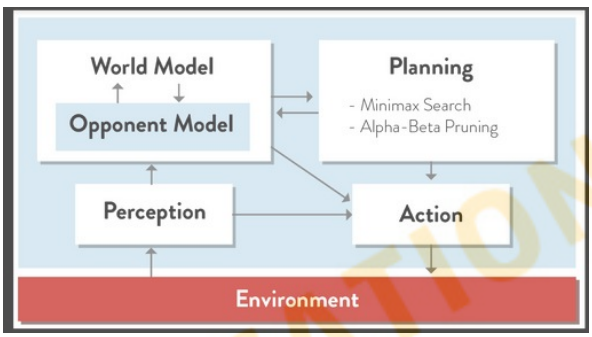
For a Planning Agent, we introduce a Planning module, which interacts with the World Model in order to reason about the effect of future actions.

Different types of planning are appropriate for different types of world model



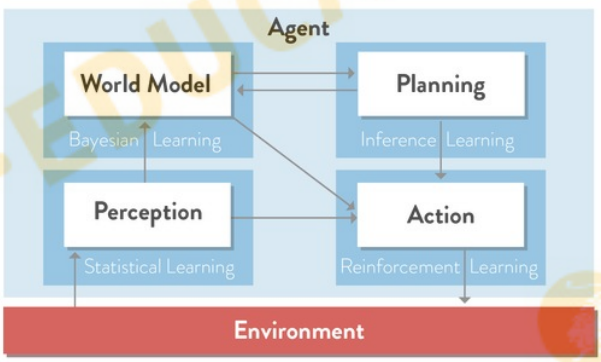
4. Game Playing Agent

For a Game Playing Agent (or Adversarial Agent) the World model must include a model of the opposing agent (known as an "Opponent Model"). An appropriate type of planning is minimax search, often enhanced with alpha-beta pruning (discussed in Week 5).



5. Learning Agent

We think of learning not as a separate module, but rather a set of techniques for improving the existing modules. Different types of learning are appropriate for different modules. In this course we will concentrate on Statistical Learning (Week 9) and Reinforcement Learning (COMP3411/9814 Extension Material, Week 9).



Learning is necessary because:

- it may be difficult or even impossible for a human to
- the agent may need to adapt to new situations without being re-programmed by a human
- We must distinguish complexity of learning from complexity of application.
- The policy for the simulated hockey player took several hours of computation to learn (in this case, by evolutionary computation). But, once in is learned, it can be applied in real time.