CA318
Advanced Algorithms and AI Search

Depth First Search; Breadth First Search;

Searching Graphs/Trees

Depth First Search; Breadth First Search; Space considerations of each;

Overview

- In this lecture and the next we cover algorithms for
  - **depth-first** and **breadth-first** search,
- followed by several refinements:
  - keeping track of nodes already considered,
  - hill climbing, and beam search.

Some conventions 1/3

- Search trees never “bite their own tail”
  - That is, suppose **S** -> { **A**, **B** }
  - When we come to expand **A** -> {...} we omit **S**
  - Obviously, **S** is reachable from **A**, but adding it into the expansion for **A** causes a loop in our tree**
- We order our nodes lexicographically
  - Prefer to write **{ A, B, C }** rather than **{ C,A,B }** etc

In graph theory this is called a **directed acyclical graph or DAG

Some conventions 2/3

- Search is **not** about maps
    - Search is about choices you make when you are trying to make decisions
- By convention, when faced with two or more choices (or branches)
    - We go down the left branch first
    - Then the others from left to right

Some conventions 3/3

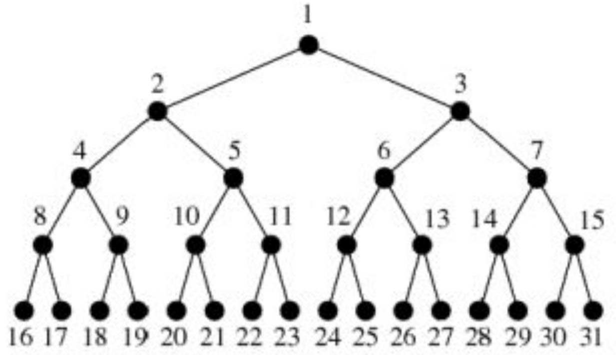
- Generally speaking, we don’t need to worry about **how** a search tree / graph is constructed
  - We just assume, for now, that it gets constructed
- Tree/graph construction, and the data structures involved, tend to obfuscate important details
- So, lets assume we have the search tree in memory already.

Depth and Branching Factor

- The depth of a tree is the number of levels in the tree
- The branching factor is the number of branches at each level
- So a binary tree of depth 10 would have
  - 2^10 nodes in total = 1024
  - The mean branching factor of the 15-puzzle is 2.1304
  - But a 15 puzzle is a huge search space 16!/2 = **$1.046139494 \times 10^{13}$**

Binary Tree Branching Factor

- The branching factor of a tree is the number of children each node has.
  - Usually this is not constant.
  - A full binary tree has a branching factor of two.



Each level has twice as many nodes as the previous level.

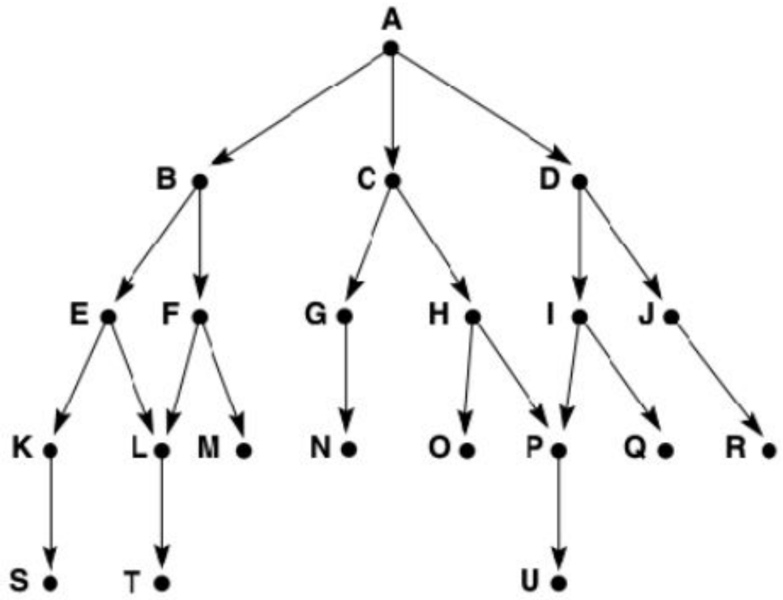
Number of leaf nodes: b^d

Depth First Search

Using a LIFO Queue (ie, a Stack)

Motivating Example

Consider the following tree, that we will use for DFS and BFS examples

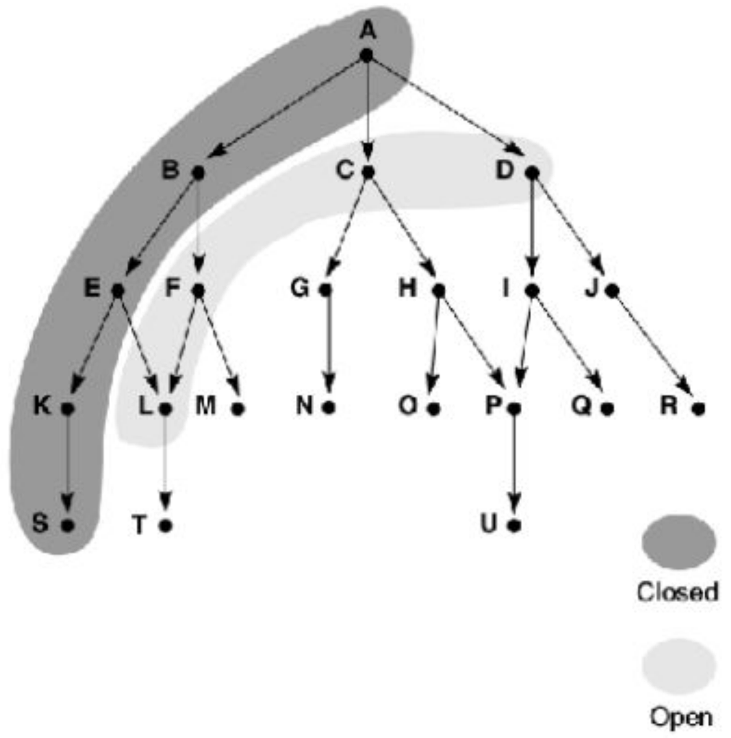


Search Objective

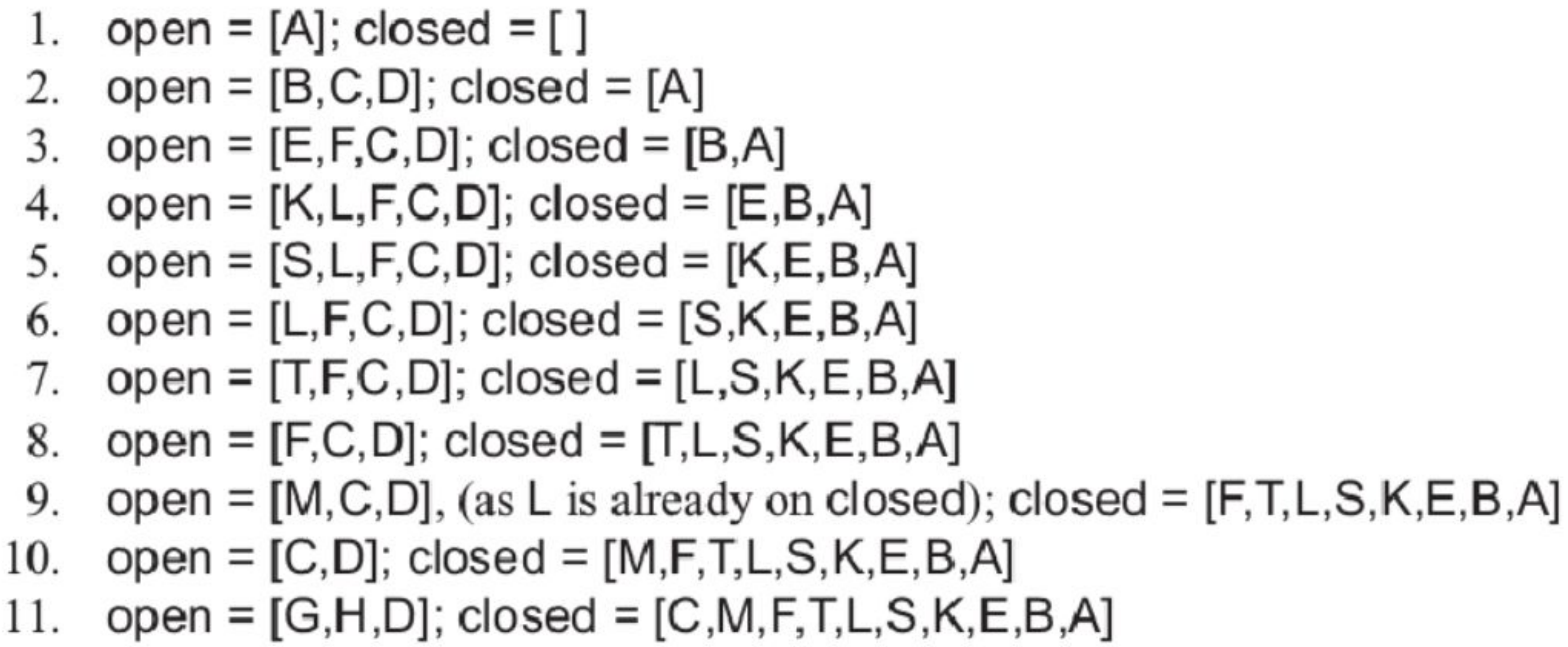
- We will start at **A**
- Our objective is node **U**
- We will stop when we find **U**
    - Or when we reach the end of the search space
- An **open** node is node yet visited
    - A **closed** node is one already visited

DFS open and closed nodes

Note the appearance - a deep down the leftmost branches dive until we can go no further, or we find the objective



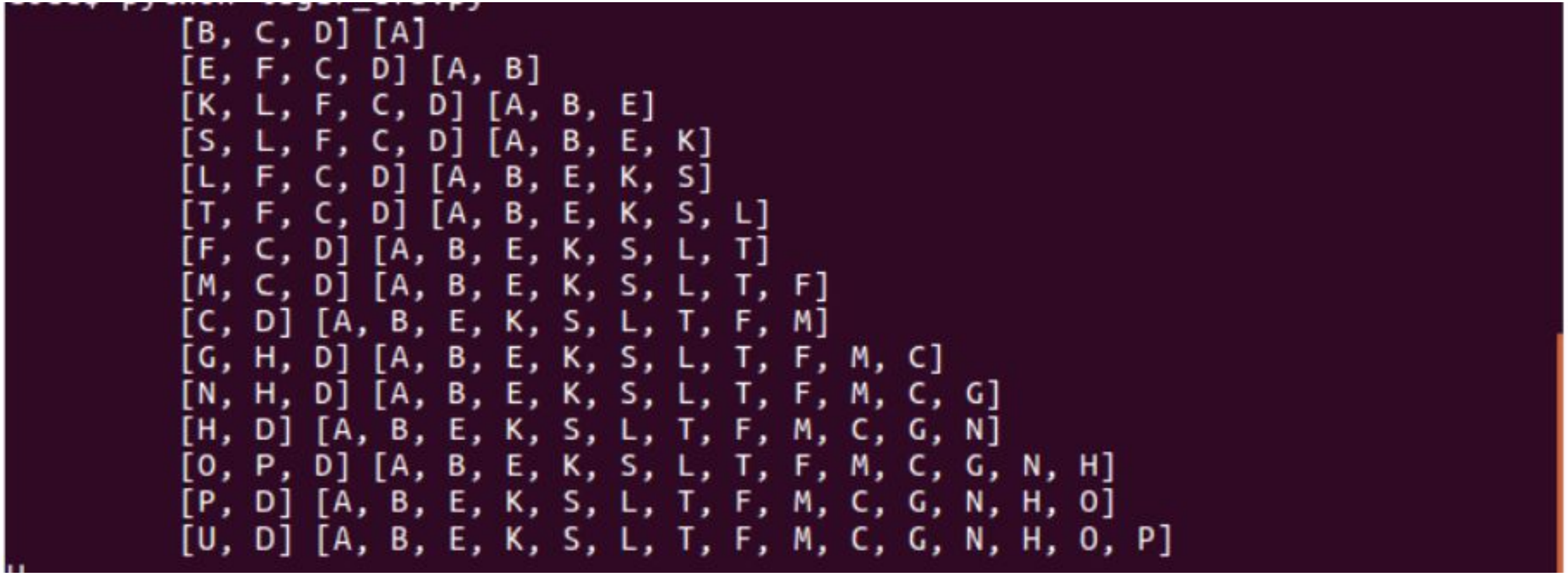
Open and Closed nodes as search proceeds



1. open = [A]; closed = [ ]
2. open = [B,C,D]; closed = [A]
3. open = [E,F,C,D]; closed = [B,A]
4. open = [K,L,F,C,D]; closed = [E,B,A]
5. open = [S,L,F,C,D]; closed = [K,E,B,A]
6. open = [L,F,C,D]; closed = [S,K,E,B,A]
7. open = [T,F,C,D]; closed = [L,S,K,E,B,A]
8. open = [F,C,D]; closed = [T,L,S,K,E,B,A]
9. open = [M,C,D], (as L is already on closed); closed = [F,T,L,S,K,E,B,A]
10. open = [C,D]; closed = [M,F,T,L,S,K,E,B,A]
11. open = [G,H,D]; closed = [C,M,F,T,L,S,K,E,B,A]

Complete Expansion of Open and Closed

Nodes examined starting at **A** and looking for **U**



```
[B, C, D] [A]
[E, F, C, D] [A, B]
[K, L, F, C, D] [A, B, E]
[S, L, F, C, D] [A, B, E, K]
[L, F, C, D] [A, B, E, K, S]
[T, F, C, D] [A, B, E, K, S, L]
[F, C, D] [A, B, E, K, S, L, T]
[M, C, D] [A, B, E, K, S, L, T, F]
[C, D] [A, B, E, K, S, L, T, F, M]
[G, H, D] [A, B, E, K, S, L, T, F, M, C]
[N, H, D] [A, B, E, K, S, L, T, F, M, C, G]
[H, D] [A, B, E, K, S, L, T, F, M, C, G, N]
[O, P, D] [A, B, E, K, S, L, T, F, M, C, G, N, H]
[P, D] [A, B, E, K, S, L, T, F, M, C, G, N, H, O]
[U, D] [A, B, E, K, S, L, T, F, M, C, G, N, H, O, P]
```

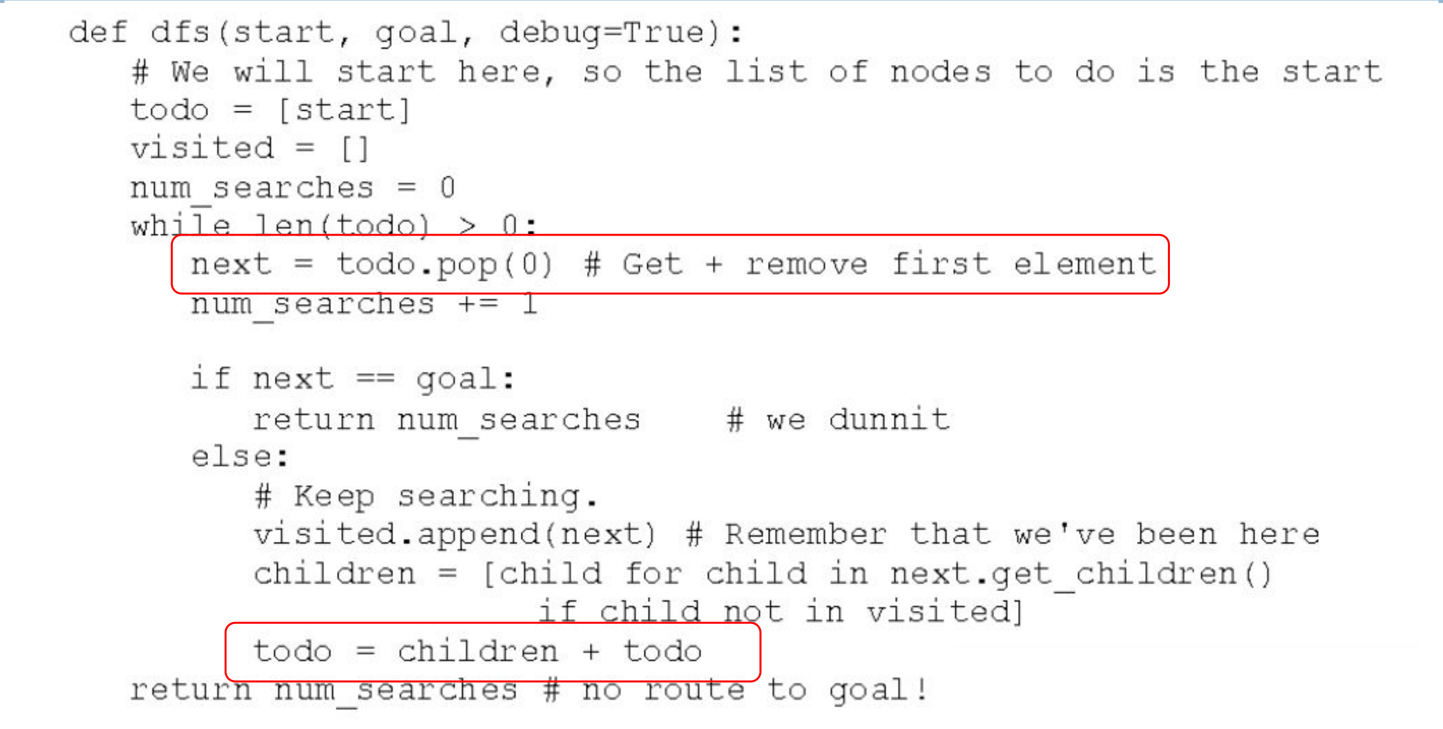
DFS features

- In order to continue down the search tree, we need to use a Last In First Out (**LIFO**) structure
  - Really this is a stack (push and pop)
- The child nodes of the current node are always placed first into the Queue
- Each node **n** is removed by **pop**'ing it (remove from position 0)
- And then we repeat this

DFS Bounds

- Of course, a very deep tree can be handled by limiting the depth of the search

DFS implementation in Python



```
def dfs(start, goal, debug=True):
    # We will start here, so the list of nodes to do is the start
    todo = [start]
    visited = []
    num_searches = 0
    while len(todo) > 0:
        next = todo.pop(0) # Get + remove first element
        num_searches += 1

        if next == goal:
            return num_searches     # we dunnit
        else:
            # Keep searching.
            visited.append(next) # Remember that we've been here
            children = [child for child in next.get_children()
                             if child not in visited]
            todo = children + todo
    return num_searches # no route to goal!
```

DFS Features

- Depth-first search gets quickly into a deep search space.
- If it is known that the solution path will be long,
    - depth-first search will not waste time searching a large
    - number of “shallow” states in the graph.
- On the other hand,
    - depth-first search can get “lost” deep in a graph,
    - missing shorter paths to a goal or even becoming stuck in an infinitely long path that does not lead to a goal.

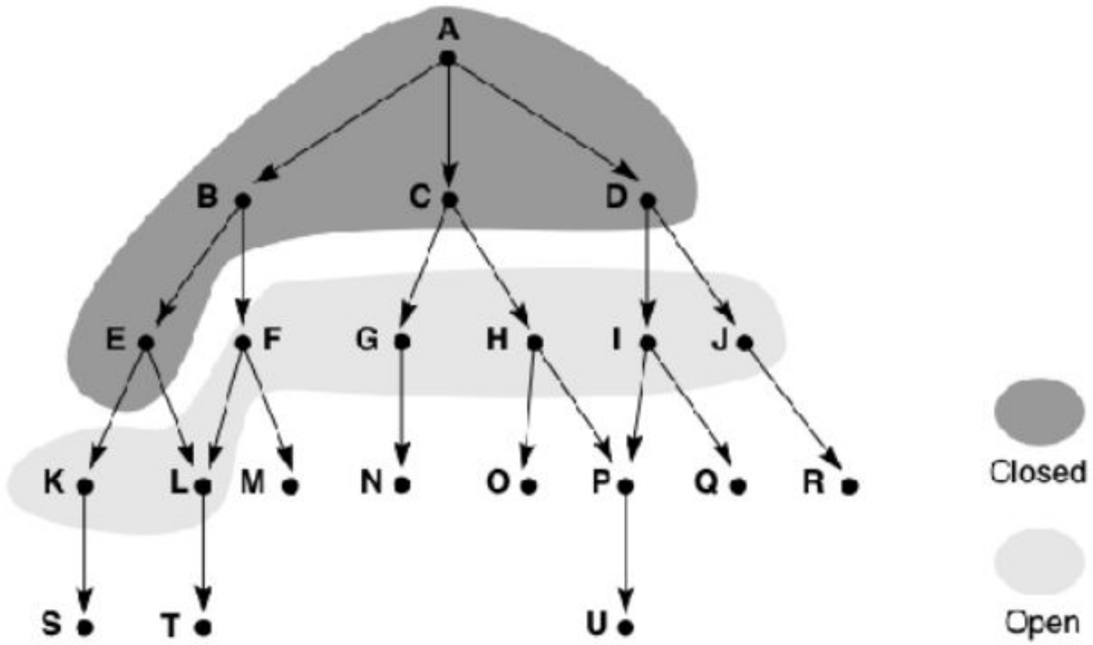
Breadth First Seach

A FIFO Queue (ie, a proper Queue)

BFS

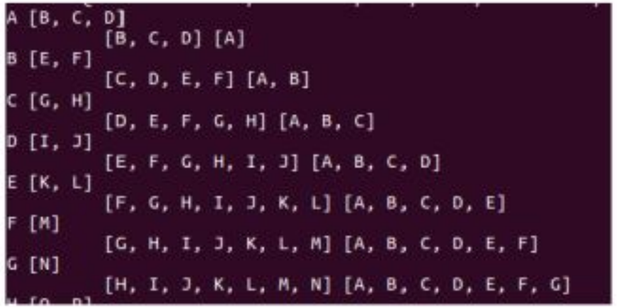
- BFS is actually very similar to DFS from the algorithm point of view
- Wherease, in DFS we place the children of the current node at the **front** of the queue
- In BFS we place the children at the end of the queue
- Suppose **A** has children { **B, C** } and **C** has children { **D, E** }
  - DFS the open queue is { **D, E, B, C** }
  - BFS the open queue is { **B,C,D,E** }

BFS open and closed nodes



BFS open and closed nodes

1. open = [A]; closed = [ ]
2. open = [B,C,D]; closed = [A]
3. open = [C,D,E,F]; closed = [B,A]
4. open = [D,E,F,G,H]; closed = [C,B,A]
5. open = [E,F,G,H,I,J]; closed = [D,C,B,A]
6. open = [F,G,H,I,J,K,L]; closed = [E,D,C,B,A]
7. open = [G,H,I,J,K,L,M] (as L is already on open); closed = [F,E,D,C,B,A]
8. open = [H,I,J,K,L,M,N]; closed = [G,F,E,D,C,B,A]
9. and so on until either U is found or open = [ ].



```
A [B, C, D]
         [B, C, D] [A]
B [E, F]
         [C, D, E, F] [A, B]
C [G, H]
         [D, E, F, G, H] [A, B, C]
D [I, J]
         [E, F, G, H, I, J] [A, B, C, D]
E [K, L]
         [F, G, H, I, J, K, L] [A, B, C, D, E]
F [M]
         [G, H, I, J, K, L, M] [A, B, C, D, E, F]
G [N]
         [H, I, J, K, L, M, N] [A, B, C, D, E, F, G]
```

BFS Features

- BFS considers every node at each level of the graph before
- going deeper into the space
- All states are first reached along the shortest path from the
- start state.
- Breadth-first search is therefore guaranteed to find the shortest path from the start state to the goal
    - Because DFS terminates when it finds a solution
    - But this may not be the optimal solution

BFS Features

- Furthermore, because all states are first found along the shortest path
    - any states encountered a second time are found along a path of equal or greater length.

BFS in Python note the DFS similarity

```
def bfs(tree, start, goal):
    # We will start here, so the list of nodes to do is the start
    todo = [start]
    visited = []
    while len(todo) > 0:
        next = todo.pop(0) # Get next element from queue
        if next == goal:
            return goal     # we dunnit
        else:
            # Keep searching.
            visited.append(next) # Remember that we've been here
            children = [child for child in next.get_children()
                            if child not in visited]

            todo += children
    return None # no route to goal
```

Performance of DFS and BFS

Performance of DFS

- **DFS is much more efficient for search spaces with many branches**
  - it does not have to keep all the nodes at a given level on the open list.
  - The space usage of depth-first search open list is a **linear** function of the length of the path.
  - At each level, open retains only the children of a single state.
  - If a graph has an average of **b** children per state, this requires a total space usage of **b × n** states to go **n** levels deep into the space.

Improving DFS by controlling the depth

- When it is known that a solution lies within a certain depth or when time constraints exist
    - such as those that occur in an extremely large space like chess,
    - limit the number of states that can be considered;
- then a depth-first search with a depth bound may be most appropriate

DFS with Depth Bounding

```
def dfs_depth_bound(start, goal, max_depth):
   todo = [(start, 0)] # start node and depth of zero.
   visited = []
   while len(todo) > 0:
      next, depth = todo.pop(0) # next item on queue

      if next == goal:
         return depth, num_searches, next
      elif depth < max_depth:
         # Keep searching.
         depth += 1
         visited.append(next) # Remember that we've been here
         children = [(child, depth) for child in next.get_children()
                        if child not in visited]
         todo = children + todo

   return depth, num_searches, None # no route to goal
```

Terminate if too deep

Itertative Deepening

- This performs a depth-first search of the space with a depth bound of 1.
- If it fails to find a goal, it performs another depth-first search with a depth bound of 2.
- This continues, increasing the depth bound by one each time.
- At each iteration, the algorithm performs a complete depth-first search to the current depth bound.
- No information about the state space is retained between iterations

Improving DFS Iterative Deepening

```
def main():
    start = Puzzle8Node('2831647 5')
    goal = Puzzle8Node('1238 4765')
    max_depth = 1
    while True:
        print("Deepening", max_depth)

        depth, node = dfs_depth_bound(start, goal, max_depth)
        if node != None:
            # Finished!
            print("Done")
            print(depth)
            break

        max_depth += 1 # Try one deeper
```

Performance of BFS

- Because it always examines all the nodes at level **n** before proceeding to level **n +1**,
    - breadth-first search always finds the shortest path to a goal node.
- In a problem where it is known that a simple solution exists, this solution will be found
- **Unfortunately**, if there is a bad branching factor,
    - => states have a high average number of children,
    - combinatorial explosion may prevent the algorithm from finding a solution using available memory.

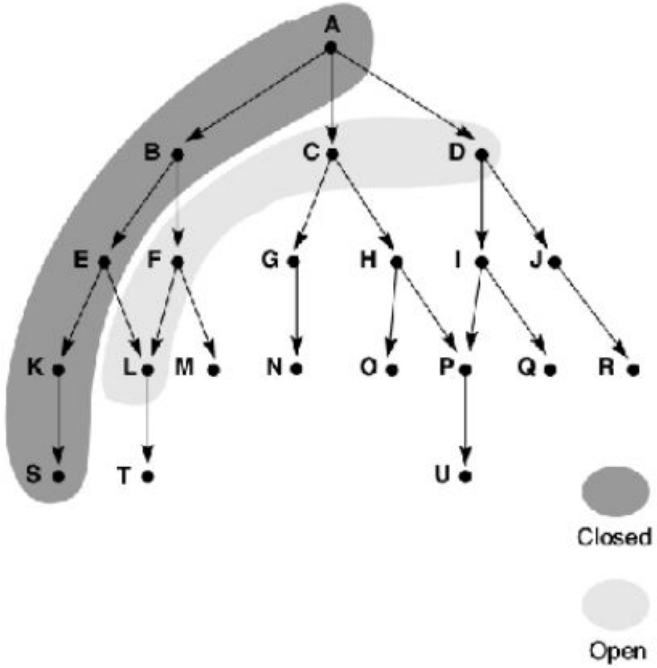
Performance of BFS

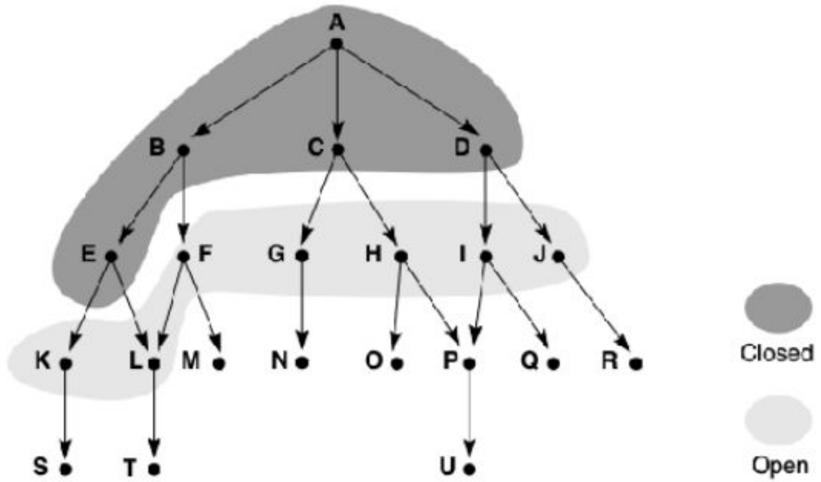
- This is due to the fact that all unexpanded nodes for each level of the search must be kept on the open list.
- For deep searches,
  - or state spaces with a high branching factor,
  - this can become quite cumbersome
- So, there is always a space concern with BFS

Exponential Space Utilization

- The space utilization of breadth-first search, measured in terms of the number of states on open, is an **exponential** function of the length of the path at any time.
- If each state has an average of **b** children, the number of states on a given level is **b** times the number of states on the previous level.
    - Quick example:
        - Avg #Children = 6, depth = 10: 6^10 = 60466176 in the open state
        - So this scales pretty badly

Comparison DFS and BFS





In Summary

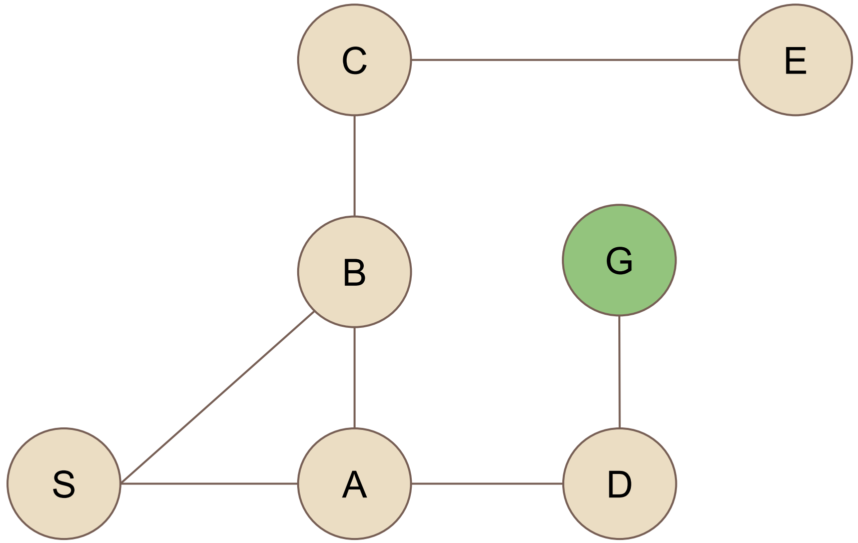
- Breadth first search always finds the shortest path to the goal.
  - Methodical
  - But ... needs to have a large todo list
    - Too large for large trees (too much memory)
- Depth first search doesn’t need to store very much, but can easily miss a nearby path.

Additional Example

Finding the Shortest Path from S - G

Traversing a graph

Consider the following graph. We wish to travel from S to G with the **fewest** moves



Solving with your eyes

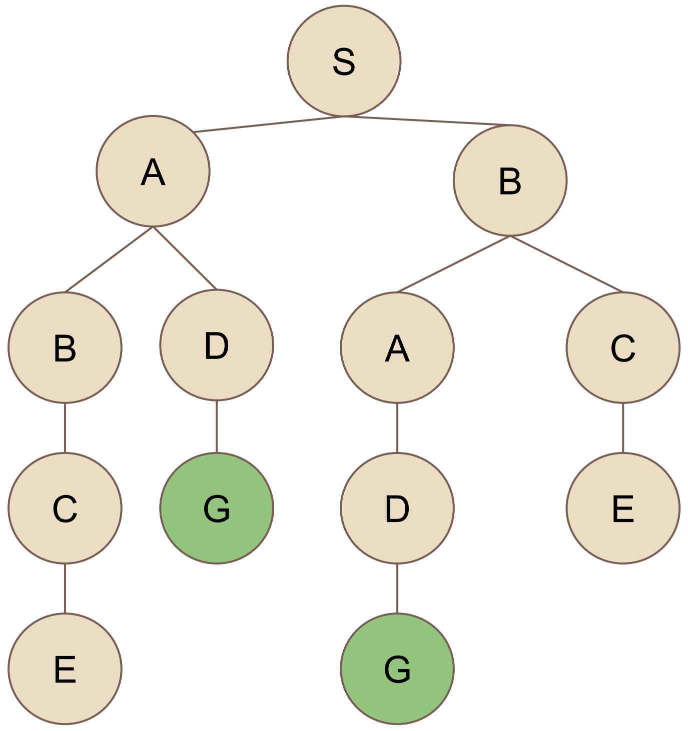
- If I said to you, please find a path from S to G, you would, within a few seconds, find a pretty good path--maybe not the optimal one, but a pretty good one--**using your eyes.**
- But we don't know how that works.
- But we do know that problem solving with the eyes is an important part of our total intelligence.

Aside

We'll never have a complete theory of human intelligence until we can understand the contributions of the human visual system to solving everyday problems like finding a pretty good path in the map.

Remember, we don’t have visually grounded algorithms, so we need another approach

Lets develop the tree for this search



Lets Do DFS and BFS here

- What is the path to the goal using DFS?
  - -> A-B-C-E [**this fails…so back-track**]
  - -> A-D-G [**goal found** - **end**]
- What is the path to the goal using DFS?
  - -> A-B [**no goal so move down**]
  - -> B-D-A-C [**no goal so move down**]
  - -> C-G-D-E [**goal found - end**]

Aside - Hill Climbing

- In DFS, we broke ties lexicographically
  - That is - if we have to expand A or B, we expand A
- In Hill Climbing, we use a Heuristic (a rule of thumb)
  - Expand the node that is closer to the goal
- This often produces a superior result, in cases where we know how far away we are
  - This is how you and I would solve a map search
- But it can go wrong - it could lead us to node “near” to our goal, but without a direct path…in otherwords, a dead end

Finally - improvements to BFS and DFS

Both DFS and BFS are pretty bad - because they never check to see if a terminating node has already been extended.

If a path terminates in a node, and if some other path previously terminated in that node and got extended-- we're not going to do it again.

This improvement is the subject of next weeks class!

Thank You

Questions?

CA318
Advanced Algorithms and AI Search

Branch and Bound; Admissible Heuristics; A*

Searching Graphs/Trees (Part 2)

Branch and Bound; Admissible Heuristics; A*

Branch and Bound

- We will now refine DFS and BFS into a single algorithm general known as **Branch and Bound**
- To make this a little clearer, we will attach a cost to each of the edges in the graph
- At each iteration, we will extend **the shortest path** that can be extended
- So we will consider only **how far we have come**
    - and not how far we have left to go.

Branch and Bound Refinements

We will then refine our branch and bound in the following three ways:

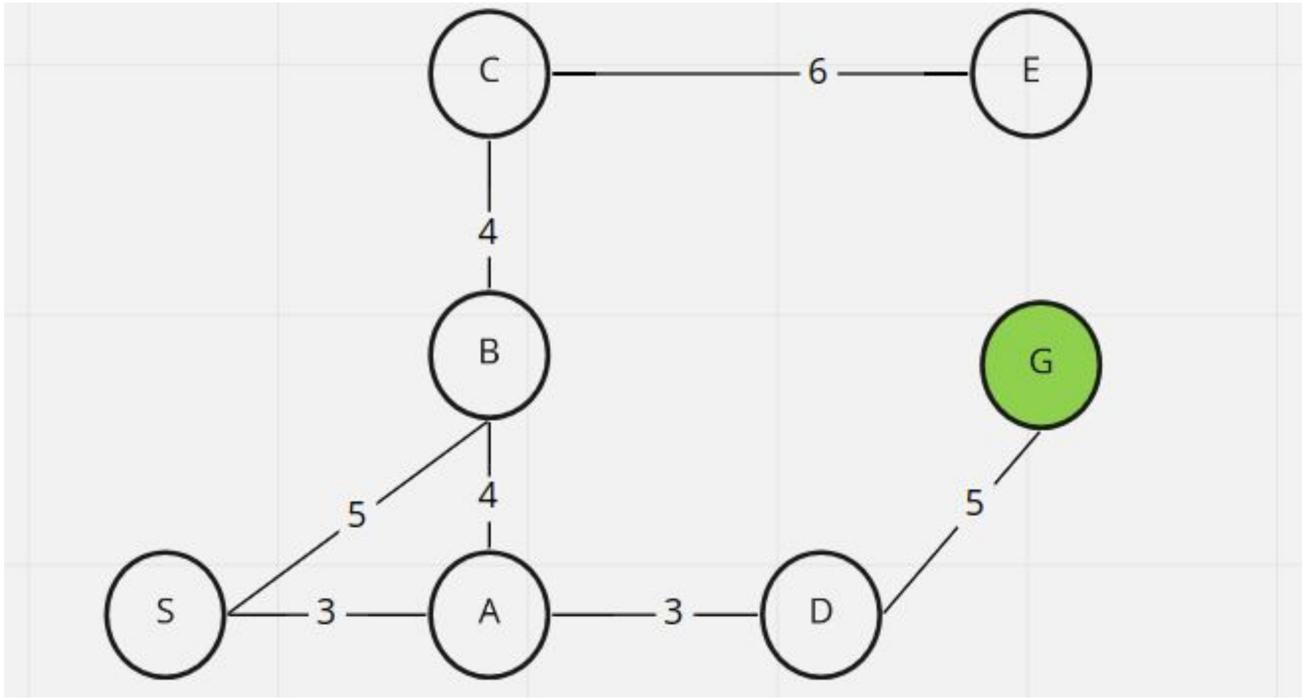
- Test for already extended nodes
- Use an admissable heuristic to estimate remaining distance
- Combine both already extended test and admissible heuristic to yield
    - A*

Naive Branch and Bound

A graph with costs

Map Search

Consider the following graph, with associated "costs". We wish to travel from S to G with the **lowest cost**

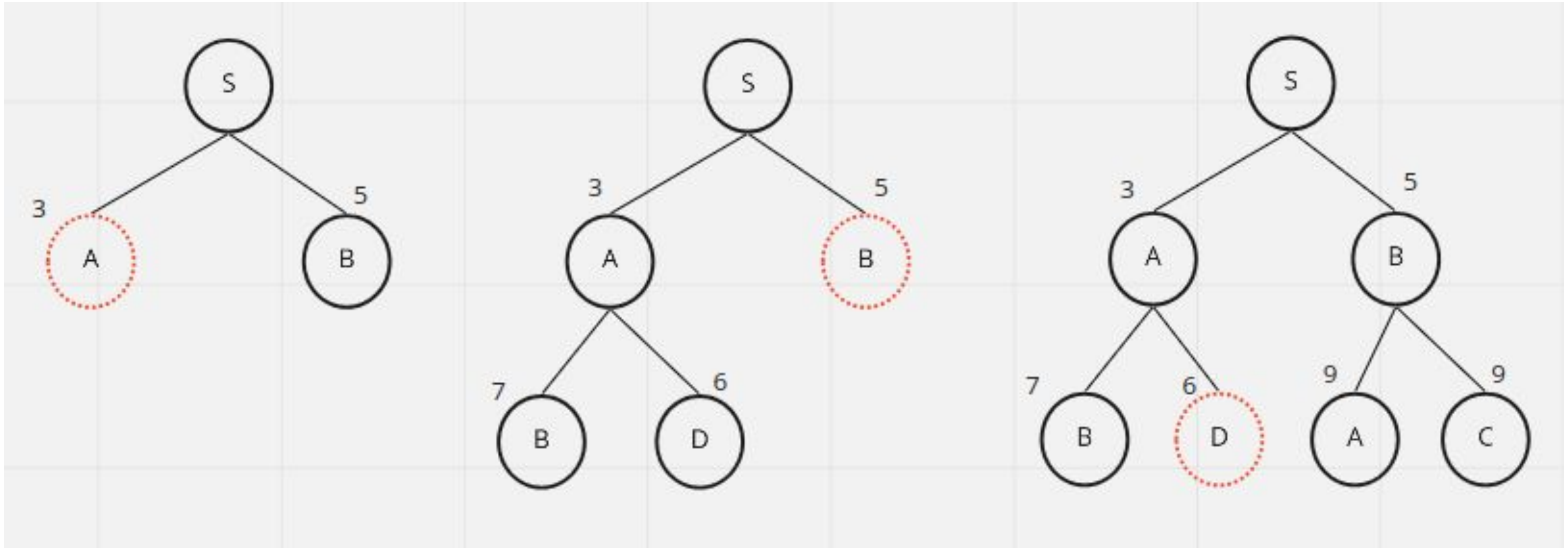


Objectives

- We want to travel from **S** to **G** with along the lowest cost route such that
  - we find the **optimal** route
  - we avoid the space complexity of BFS
  - we avoid the non-optimal solution of DFS

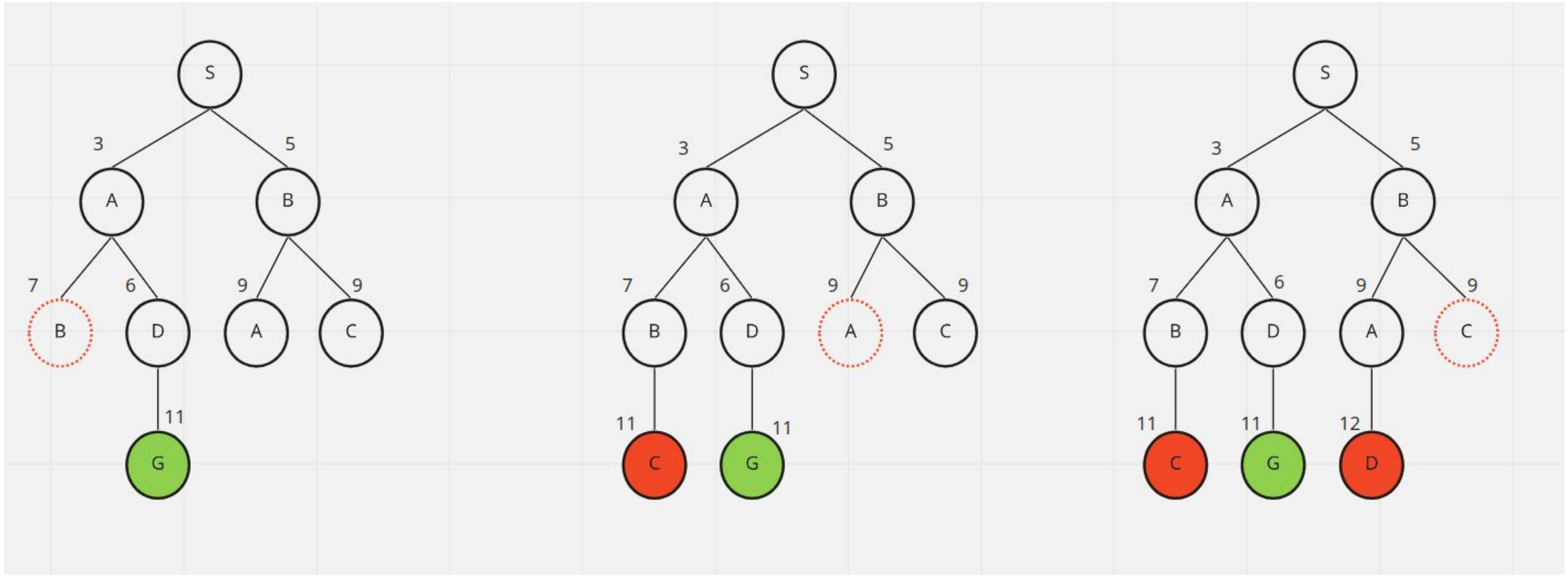
=> We will use a combination of both BFS and DFS, and use the path costs as a guide

Path Evaluation … 1



The first three node extensions

Path Evaluation … 2

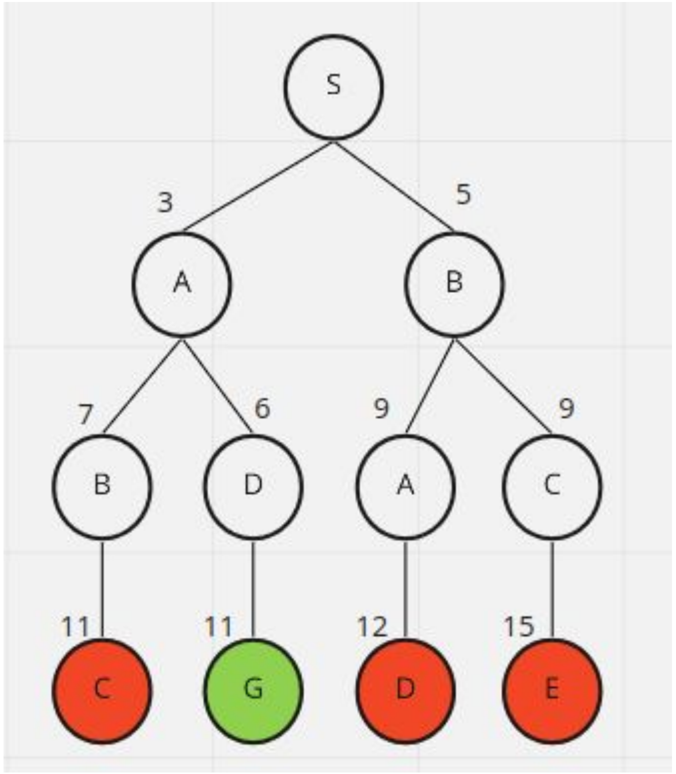


The next three node extensions

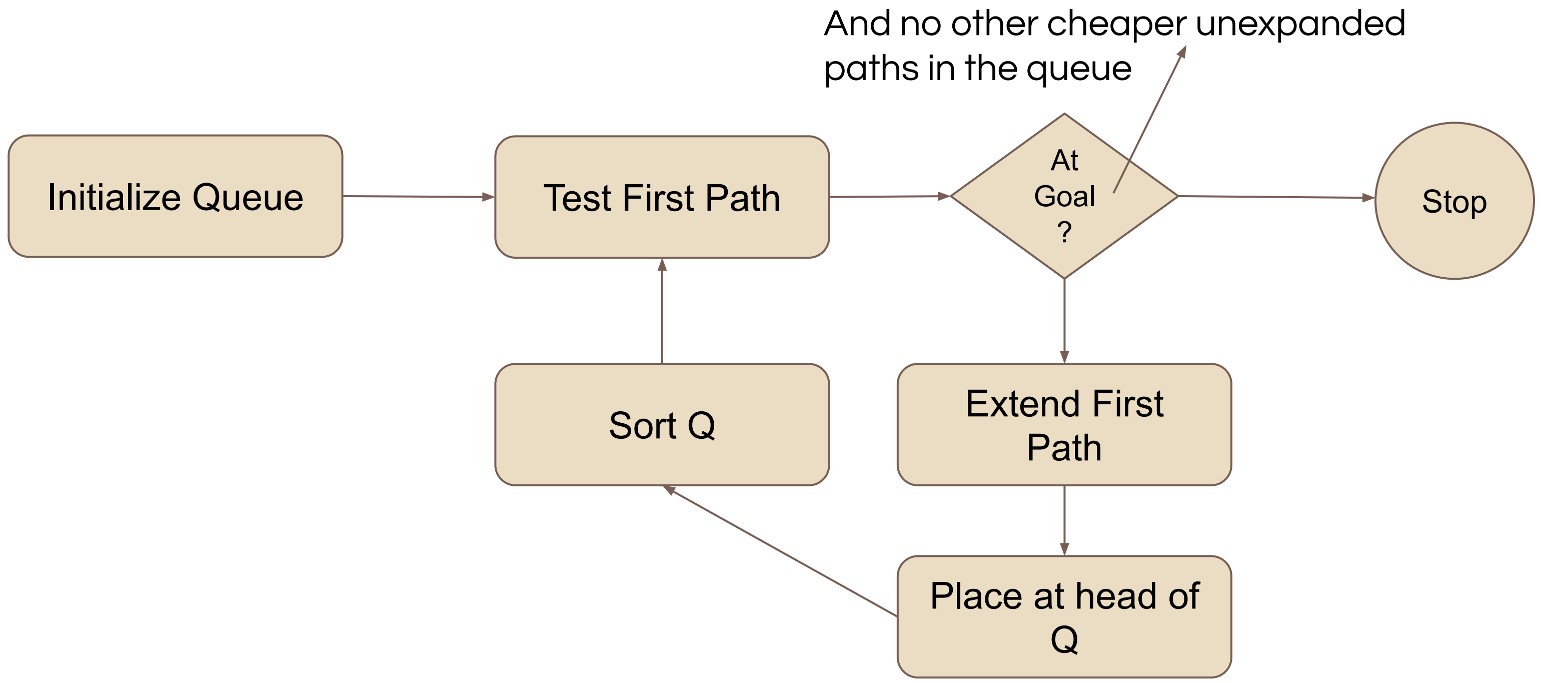
Path Evaluation … 3

Now we stop without further expansion of C, D or E**

**why?

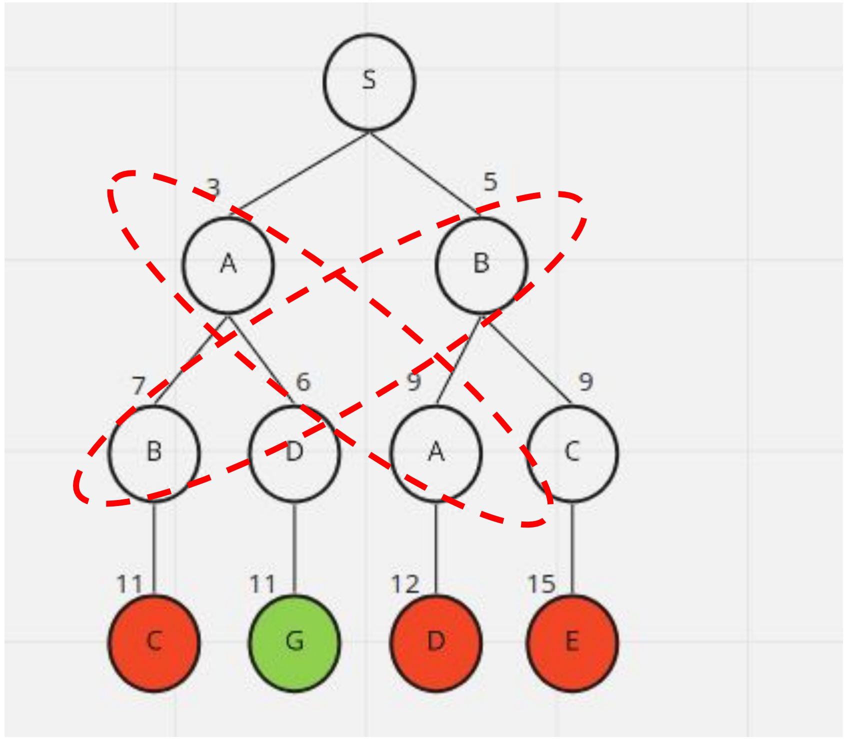


Flowchart of Branch and Bound



Naive Branch and Bound

In the previous example you can see we evaluated paths from **A** and **B** more than once



Duplication of Work

- We extended paths that went through A more than once.
- Would it ever make sense to extend A a second time?
    - No because we've already extended a path that got there with less distance.
- Will it ever make sense to extend B more than once?
    - No because we've already extended another path that gets to be by a shorter distance.

So if we keep an extended list, we can add that to branch to our advantage.

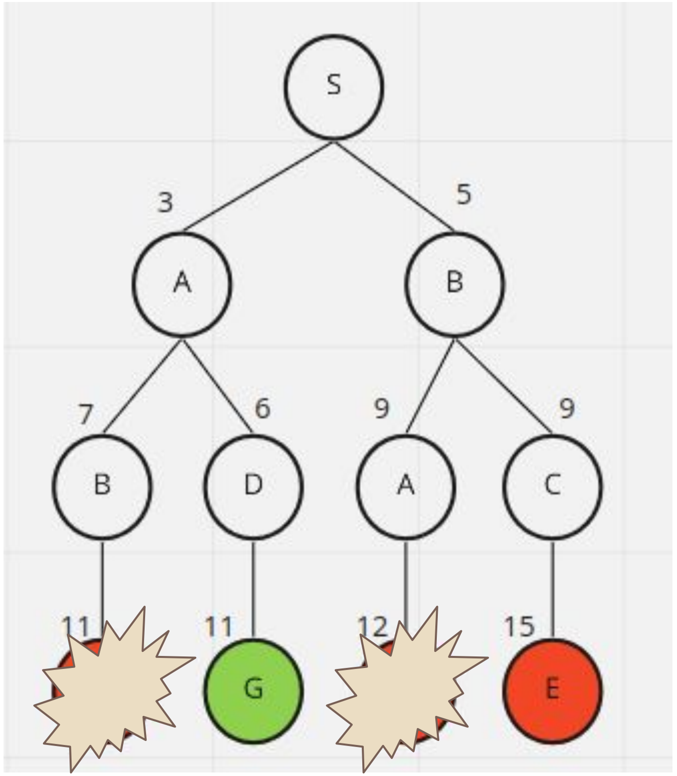
Already Extended [list]

So here we can see the end of the search graph in branch and bound.

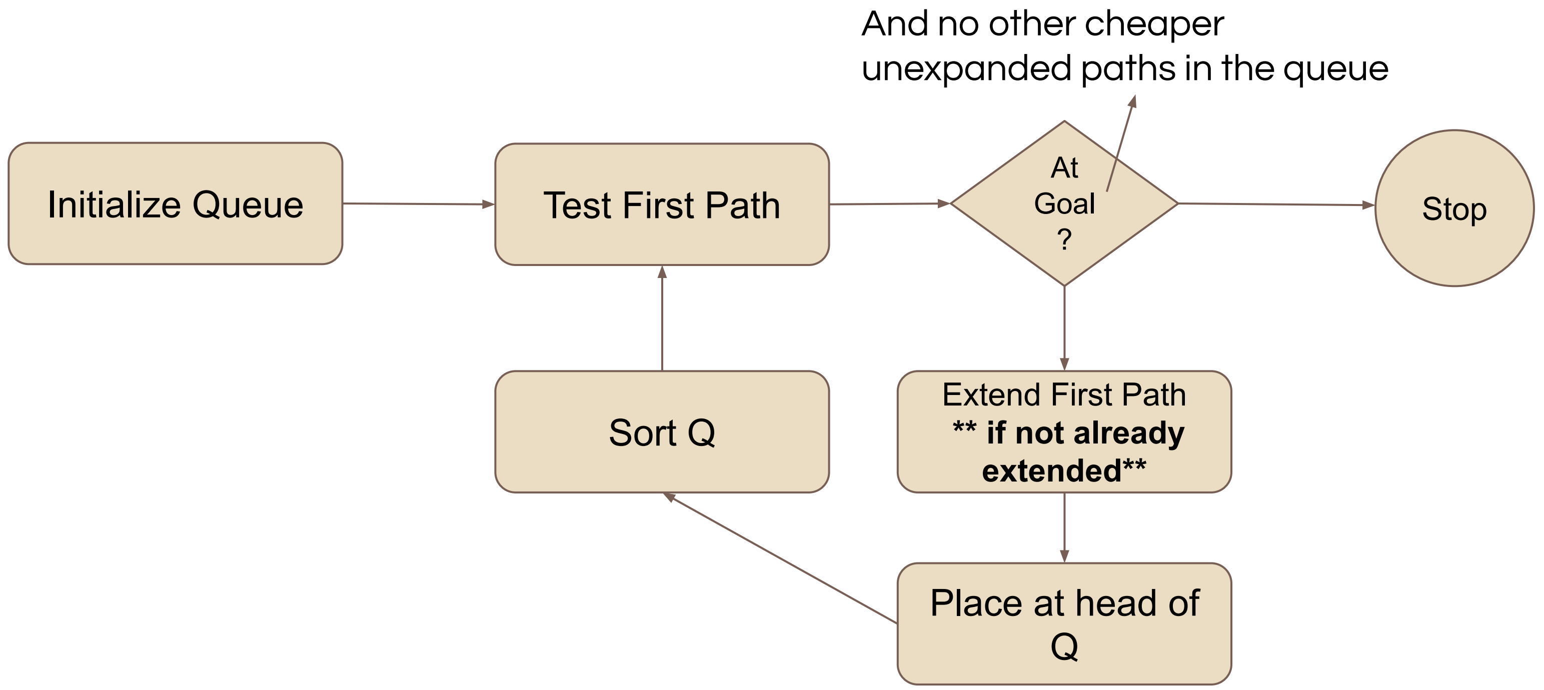
If we keep track of the already extended paths, we see both **A** and **B** have already been expanded because they were both **terminating** nodes on some previously evaluated path.

if we were to expand A a second time it could not possibly result in an shortest path

So we save some work here by using the already extended list



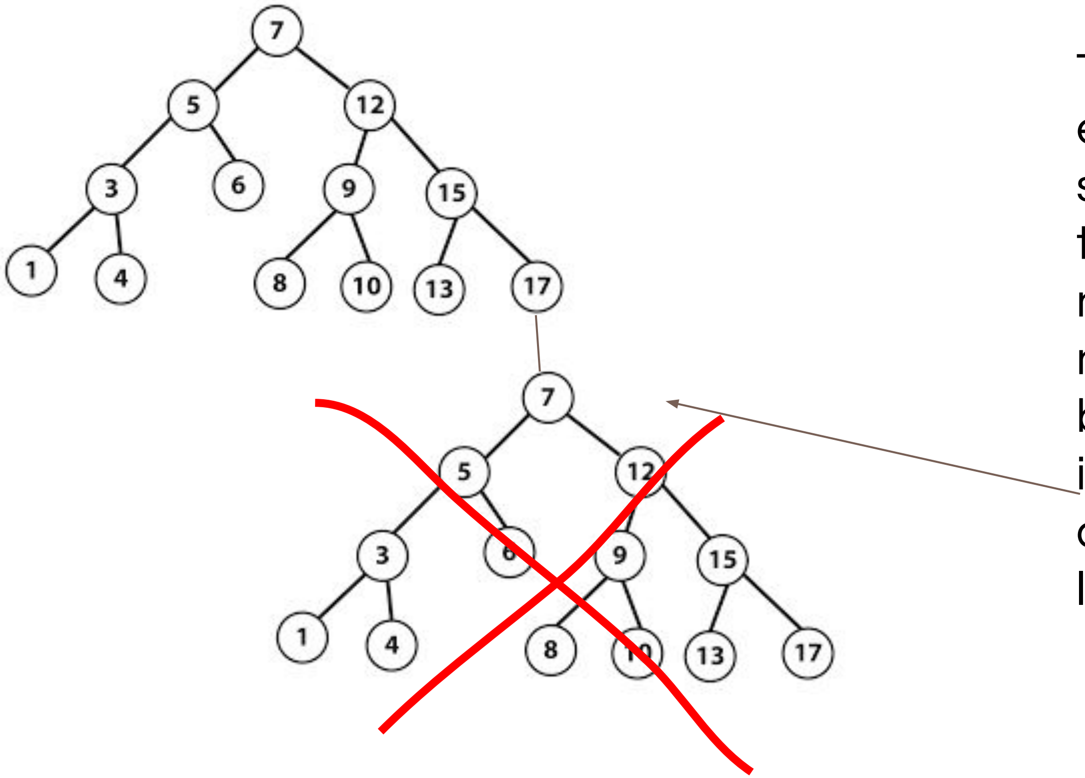
Branch and Bound with Extended List


And no other cheaper unexpanded paths in the queue
Initialize Queue
Test First Path
At Goal ?
Stop
Sort Q
Extend First Path ** if not already extended**
Place at head of Q

Pruning Optimization

- If you compare the two search trees:
  - you can see that there might be vast areas of subtrees tree that are pruned away
  - and these sub-graphs don't have to be examined all.
- Here is a very compelling tree with pruned optimizations

Pruning Optimization

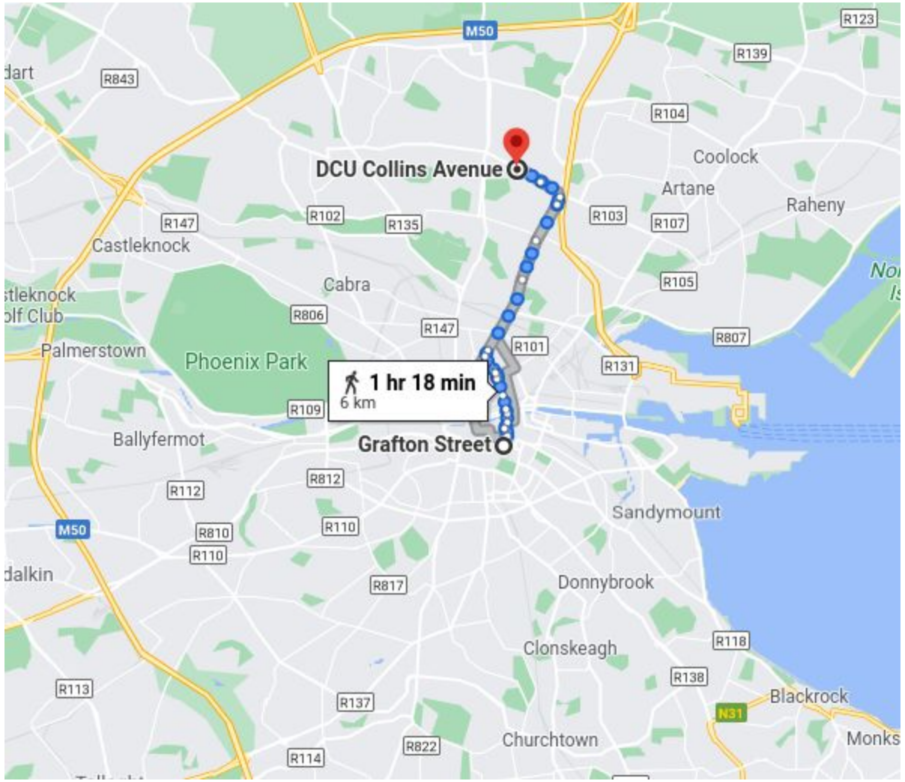


This is a contrived example, but it serves to illustrate the point - we need to improve naive branch and bound by including an already extended list

Problems Remain

Even when we include the **already extended list** concept, some obvious problems remain.

The main problem is obvious when we reorient the problem as a real-world map:



Going Away from the Goal

- The improved-naive branch and bound (as defined) has no concept of **remaining distance to the goal**
- In fact, the algorithm could spend a considerable amount of time evaluating paths **that take it away from the goal**
- This is equivalent to searching for shortest [walking] path from DCU to Grafton Street by evaluating routes through
  - Ballymun
  - IKEA / Poppintree
  - Santry
  - Malahide

Distance

- We would never do that**
- Because we intuitively have this idea of a **distance** - even if this notion is a little ambiguous
- We need a similar concept in our branch and bound
- It is called the Airline Cost, and it is a type of Admissable Heuristic
- **A Heuristic is a rule of thumb**

**would we??

Admissible Heuristics

“Airline Costs”

Admissible Heuristic

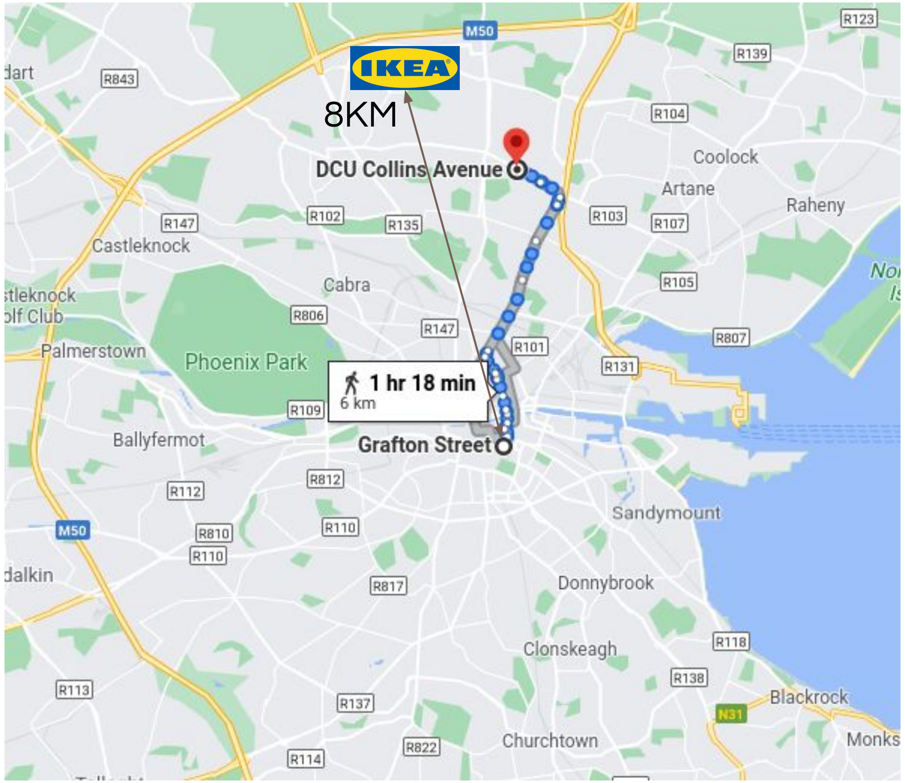
If the heuristic estimate is guaranteed to be less than the actual distance, that's called an **admissible heuristic.**

“Admissible” because you can use it for the purpose of make branch and bound more efficient.

Airline Cost

Looking again at this map we could annotate some admissible heuristics cost

This is sometimes referred to as the Airline Cost (because its a relatively straight path)

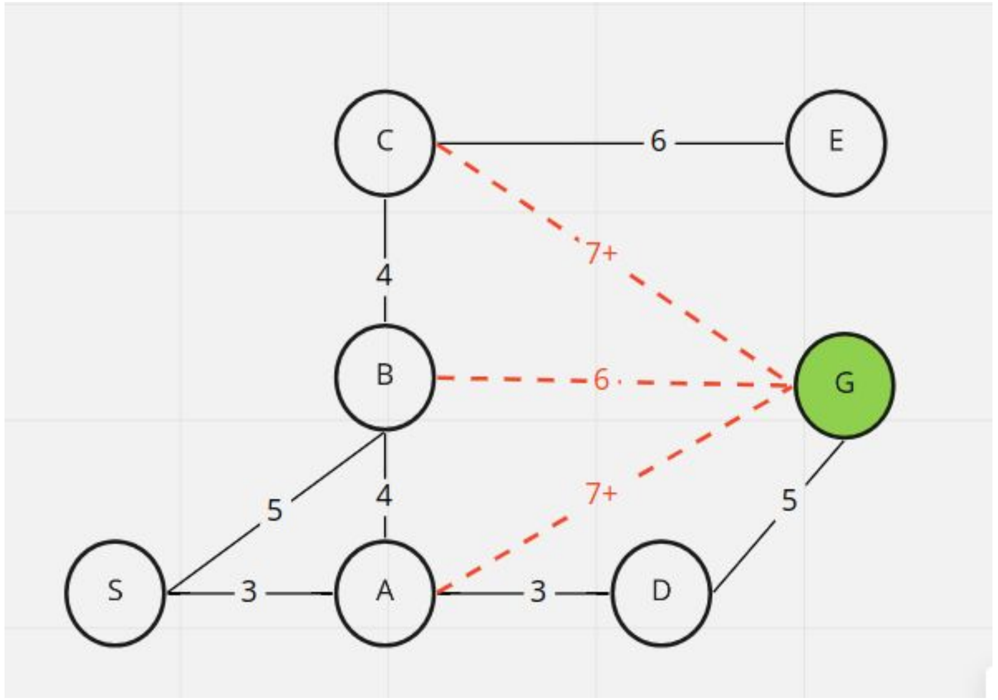


Attach an “Airline” cost

In some cases we can guess an **admissable heuristic**. In this case, a bee-line or airline distance that we attach to the graph to help with the heuristic.

Shown here in red dashed line

Lets now do this for our original problem



Airline Costs

So long as our estimates are all below the actual cost then they are admissible.

So we can create a table of these estimates now:

Node	Airline Cost
A	At least 7 (= 7^+)
B	Exactly 6
C	At least 7 (= 7^+)

Improved Naive Branch and Bound

To simplify the process, we will return to the Naive Branch and Bound **without** the use of the expanded list in order to demonstrate how it improves efficiency

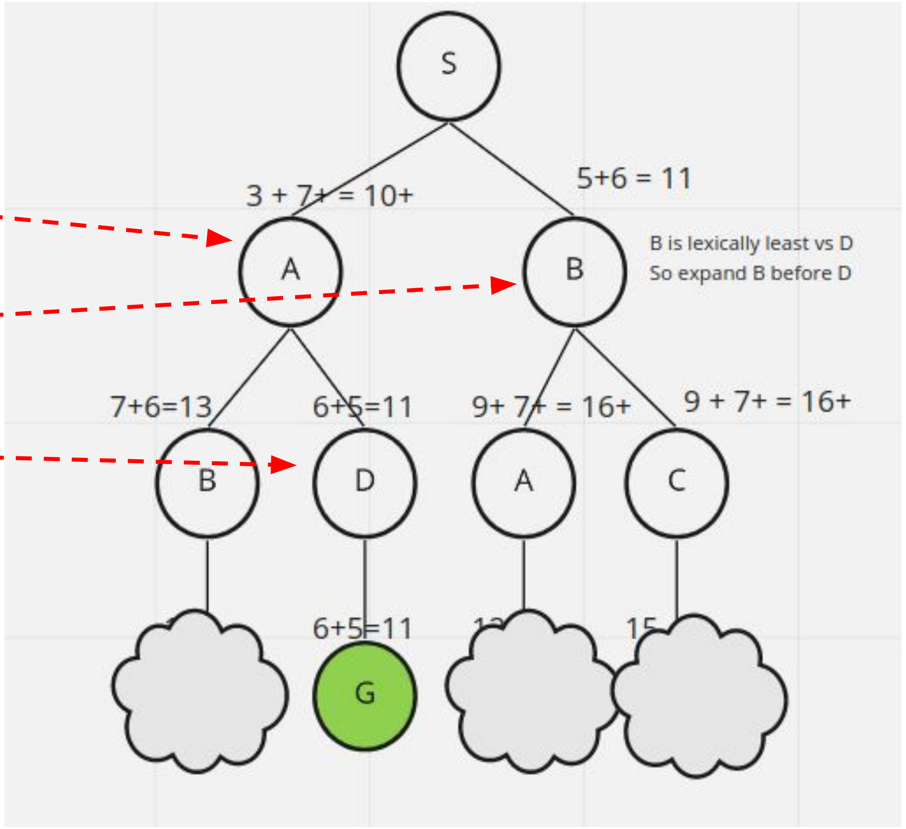
Branch and Bound with Airline Costs

1. Shortest potential path, so expand

2. Shortest potential path, so expand

3. Shortest potential path, so expand

Not evaluated ☁



Branch and Bound with Airline Costs

- We can see that branch and bound with airline costs saves us a lot of time
- This is equivalent to walking from DCU to Grafton street, always using a street that brings us closer to Grafton street.
- So we would always want to use Branch and Bound with Admissible Heuristics
- But we can see that in the previous example, we expanded A and B unnecessarily.
- So we really do need our **Already Extended List** logic back in

Finally...

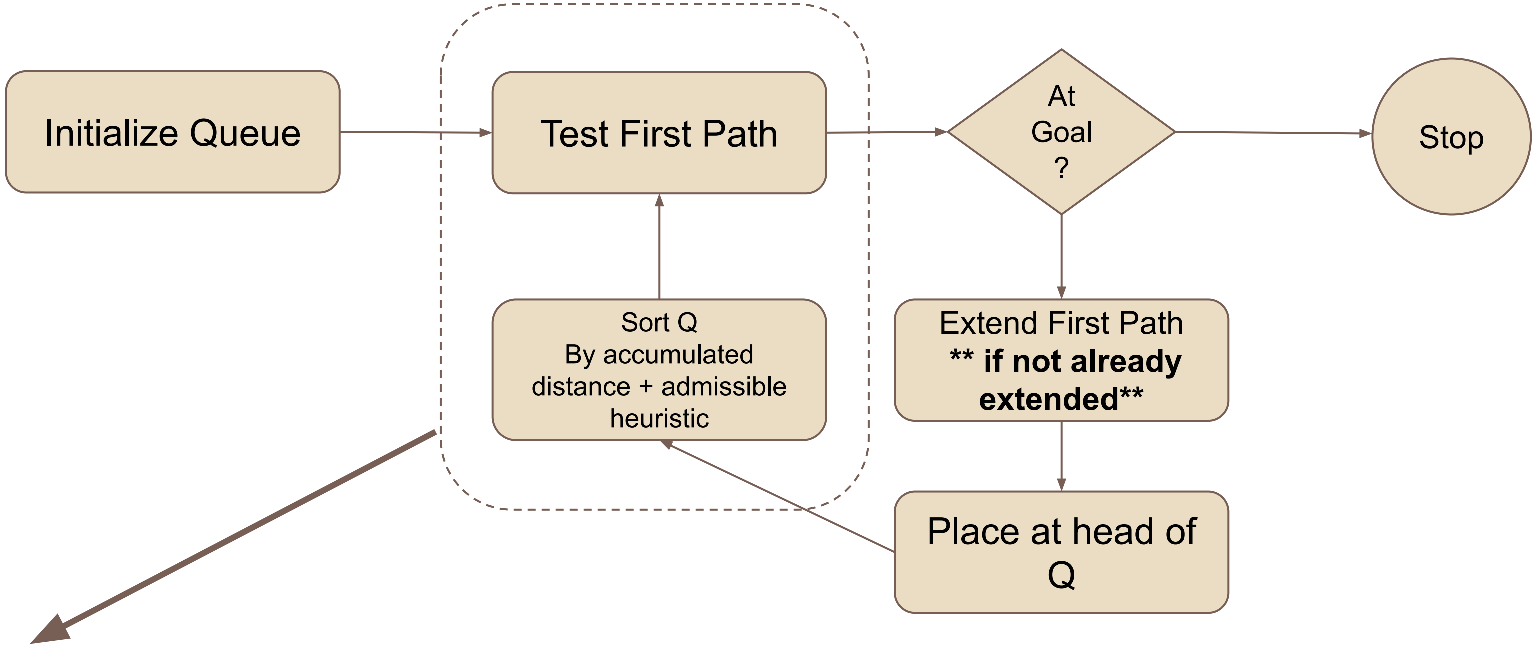
- So we now have Branch and Bound with:
  - Already Extended List
  - Admissible Heuristic

This is A*

A* Search

Branch and Bound with Already Extended and Admissible Heuristics

The New Flowchart = A*


Initialize Queue
Test First Path
At Goal ?
Stop
Sort Q By accumulated distance + admissible heuristic
Extend First Path ** if not already extended**
Place at head of Q

Aside: do we really need to sort the Queue? - why might we not want to sort the Queue? What alternatives are there if we don't sort the Queue?

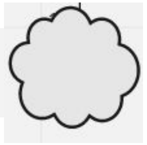
The Pruned A* Search Tree

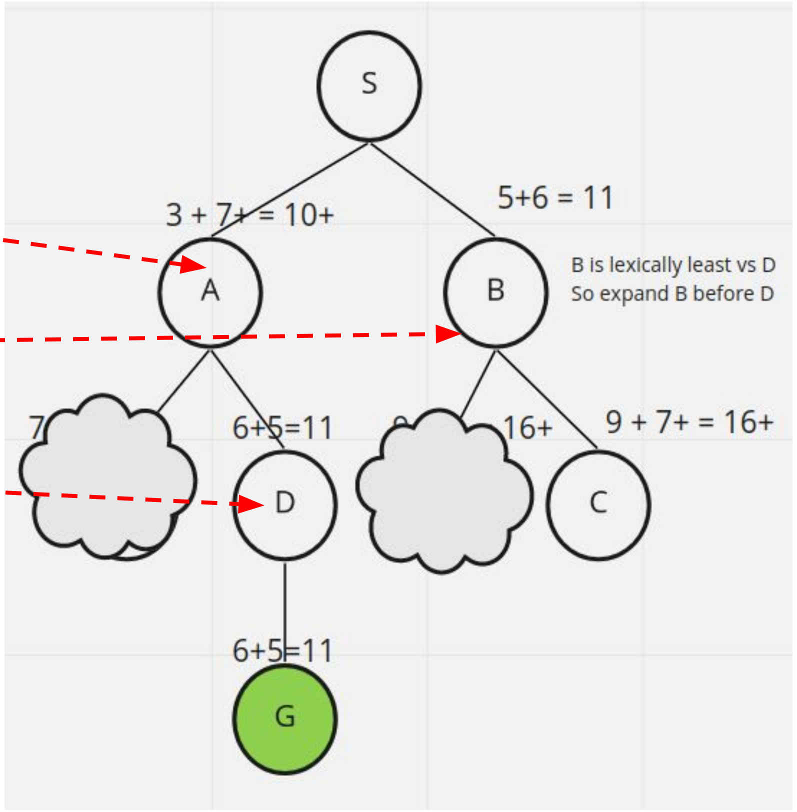
1. Shortest potential path, so expand

2. Shortest potential path, so expand

3. Shortest potential path, so expand

Not evaluated





The Pruned A* Search Tree

We now have our A* Search tree, it finds the optimal shortest path:

- Does not expand already expanded paths
- Applies admissible heuristics

So this looks like the perfect solution.

But there is one observation here, concerning admissible heuristics.

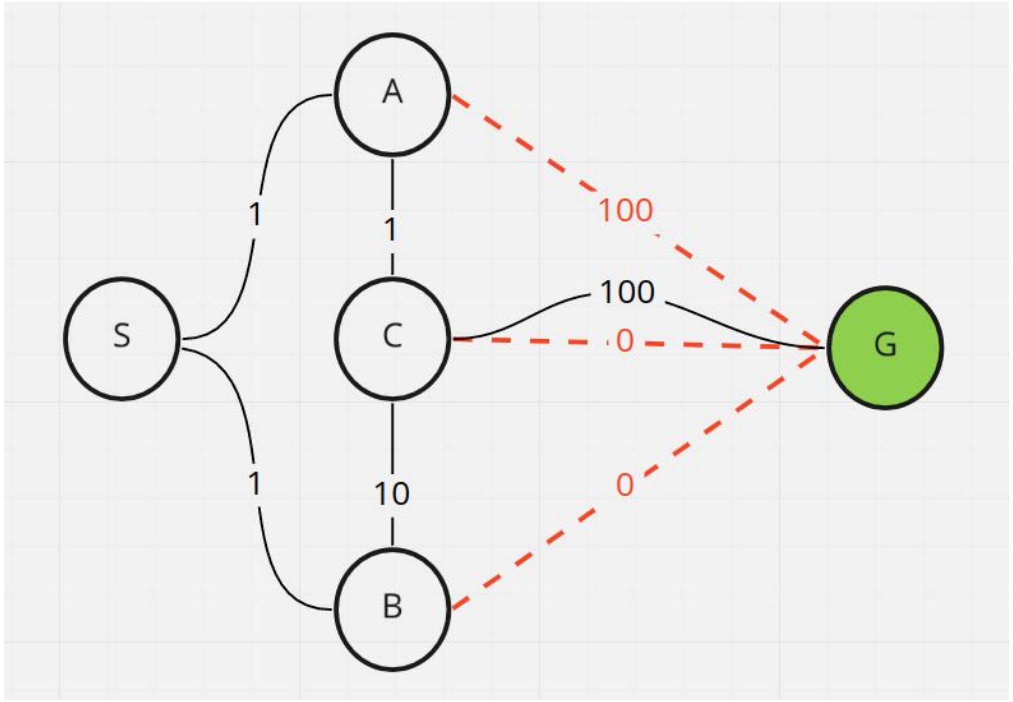
Admissible Heuristics Fail

Sometimes, the Admissible Heuristic approach can fail.

In a map path finding scenario it is reliable.

But not all search problems are map problems.

Some searches are in **non-Euclidean Spaces , eg**



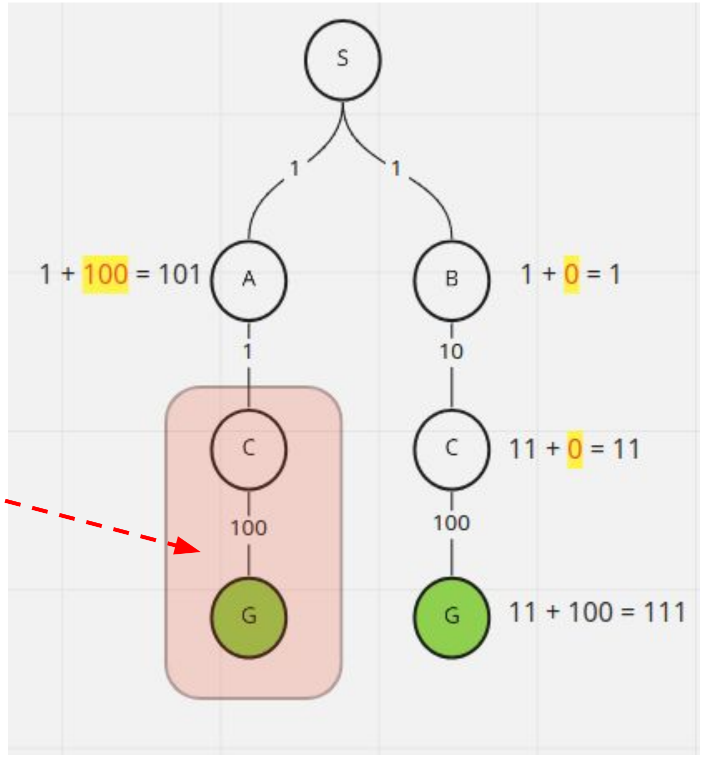
Is it acceptable to use 0? Is 0 **admissible**? Is the 100 acceptable?

A* Search Path

So obviously we have a problem here.
We are working in a non-Euclidean context, we have a path S-B-C-G=111 but a shorter path existed: S-A-C-G=102

This is an obvious situation where A* fails.

The lesson here is - A* does well with maps, but may cause problems in **non-Map** situations



Admissible and Consistent Heuristic

More Formally:

Admissible: $|H(x,G)| \leq D(x,G)$

Consistent: $|H(x,G) - H(y,G)| \leq D(x,y)$

H(A,G) = 100

H(B,G) = 0

D(A,B) = 2

| 100 | > 2 ---> **Fails Consistency**

How do we fix this?

Cost optimal A*

With an inadmissible heuristic, A* may or may not be cost-optimal.

Admissible and Consistent

- Every consistent heuristic is admissible (but not vice versa), so with a consistent heuristic, A* is cost-optimal.
- **In addition, with a consistent heuristic, the first time we reach a state it will be on an optimal path**
    - so we never have to re-add a state to the queue
    - and never have to change an entry in reached.
- But with an inconsistent heuristic, we may end up reaching the goal but in a non-optimal way

Admissible and Consistent

- These complications have led many implementers to avoid inconsistent heuristics
- However, the worst effects rarely happen in practice, and one shouldn’t be afraid of inconsistent heuristics

A* Proof of Optimality

A* is cost-optimal, which we can show with a proof by contradiction.

- Suppose the optimal path has cost **C***, but the algorithm returns a path with cost **C** > **C***.
- Then there must be some node **n** which is on the optimal path and is unexpanded (why?)
- So then, using the notation **d*(n)** to mean the cost of the optimal path from the start to **n**, and **h*(n)** to mean the cost of the optimal path from **n** to the nearest goal then

Proof, by contradiction

$$f(n) > C^*(otherwise\ n\ would\ be\ expanded)$$

$$f(n) = d(n) + h(n)$$

$$f(n) = d(n)^* + h(n)$$

$$f(n) \leq d(n)^* + h(n)^*\ (by\ admissibility:\ \ h(n) \leq h(n)^*)$$

$$f(n) \leq C^*\ (by\ definition\ C^* = h(n)^* + h(n)^*)$$

Detour

Satisficing search

Satisficing search

- A* search has many good qualities, but it expands a lot of nodes.
- We can explore fewer nodes (taking less time and space) if we are willing to accept solutions that are suboptimal, but are “good enough”
    - what we call **satisficing** solutions
- If we allow A* search to use an inadmissible heuristic (one that may overestimate)
    - then we risk missing the optimal solution

Detour Indexes

- For example, road engineers know the concept of a detour index,
  - a multiplier applied to the straight-line distance to account for the typical curvature of roads.
- A detour index of 1.3 means that if two cities are 10 miles apart in straight-line distance, a good estimate of the best path between them is 13 miles.
- For most localities, the detour index ranges between 1.2 and 1.6

Weighted A*

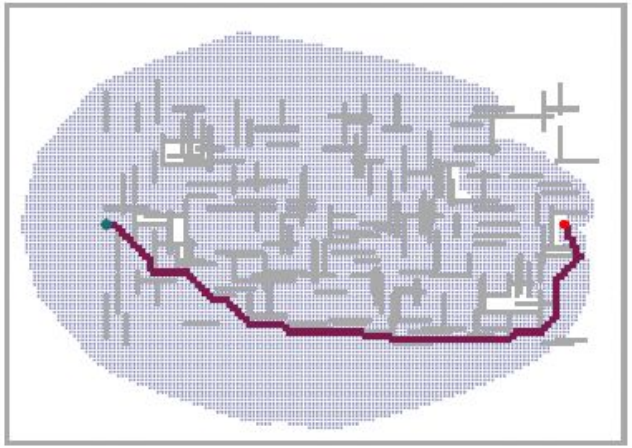
- We can apply this idea to any problem, not just ones involving roads, with an approach called **weighted A*** search
  - we weight the heuristic value more heavily, giving us the evaluation function

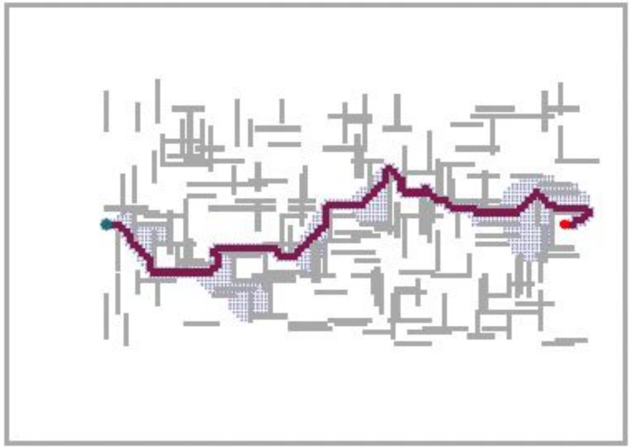
$$f(n) = g(n) + W \times h(n)$$

for some W > 1

Here we have a weight parameter that can be used to govern the balance between solution quality and search effort

Weighted A* Example


(a)


(b)

Two searches on the same grid: (a) an A* search and (b) a weighted A* search with weight W = 2. The gray bars are obstacles, the purple line is the path from the green start to red goal, and the small dots are states that were reached by each search. On this particular problem, weighted A* explores 7 times fewer states and finds a path that is 5% more costly

Observation

- Satisficing is a decision-making strategy that aims for a **satisfactory** or **adequate result**
    - rather than the **optimal** solution.
- Instead of putting maximum exertion toward attaining the ideal outcome
    - satisficing focuses on pragmatic effort when confronted with tasks.

Thank you

Any Questions?

CA318
Advanced Algorithms and AI Search

Games, Minimax, Alpha-Beta Pruning; Anytime Algorithms

Games

Game Playing Strategies

[non-]Adversarial Games

- In the previous lectures we focused on optimal search strategies when there was a single, definable goal (**G**)
- Even though these search strategies had different implementations, they were characterized by all having a **non-adversarial** nature
- In other words, when searching from the optimal path from **S-G**, we did not need to worry about an opponent blocking or impeding our progress

Adversarial Games

Now we move on to a new problem:

- How to search for the optimal solution in a space in which an opponent (or set of opponents) seeks to prevent this.
- These are **competitive environments,**
    - two or more agents have conflicting goals, giving rise to adversarial search problems.
- These classes of problem are called **Adversarial Search and Games**

Game Theory - 3 Views

There are at least three ways to analyze multi-agent environments:

#1 A very large number of agents:

- consider them in the aggregate as an **economy**,
- allowing us to do things like predict that increasing demand will cause prices to rise, without having to predict the action of any individual agent
- These are sometimes called **zero-sum** games

Game Theory - 3 Views

#2 A stochastic hostile environment:

- adversarial agents are a part of the environment
  - a part that makes the environment nondeterministic.
- But if we model the adversaries in the same way that, say, rain sometimes falls and sometimes doesn’t
- we miss the idea that our adversaries are actively trying to defeat us,
  - whereas the rain supposedly has no such intention

Game Theory - 3 Views

#3 A stochastic hostile environment:

We explicitly model the adversarial agents with the techniques of adversarial game-tree search.

This is the approach we will follow here.

2 Player Zero Sum Games

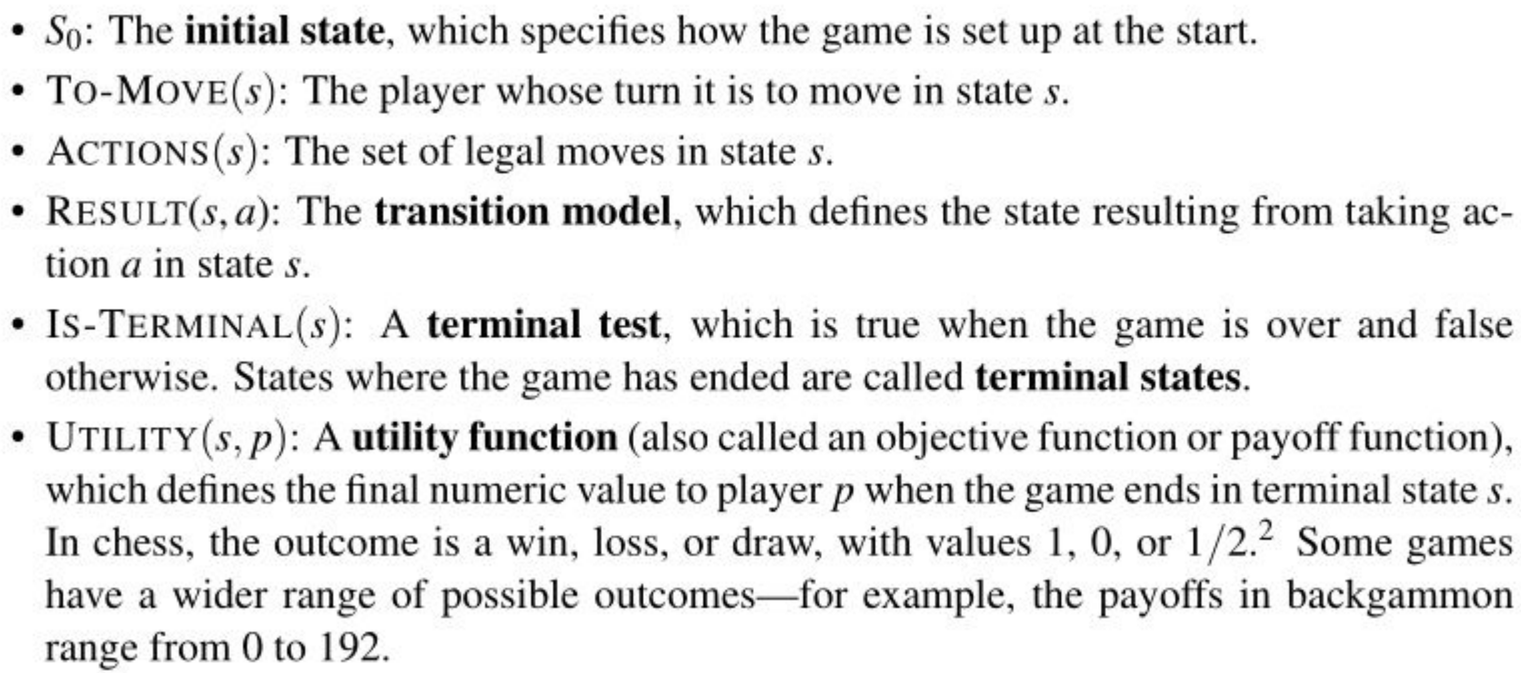
perfect information; no win-win

No win-win

- The games most commonly studied within AI (such as chess and Go) are what game theorists call **deterministic**, two-player, turn-taking
- “Perfect information” is a synonym for “fully observable,”
- “zero-sum” means that what is good for one player is just as bad for the other:
    - there is no “win-win” outcome.

Definition of a Game

A game can be formally defined with the following elements:



- S_0: The **initial state**, which specifies how the game is set up at the start.
- TO-MOVE(s): The player whose turn it is to move in state s.
- ACTIONS(s): The set of legal moves in state s.
- RESULT(s, a): The **transition model**, which defines the state resulting from taking action a in state s.
- IS-TERMINAL(s): A **terminal test**, which is true when the game is over and false otherwise. States where the game has ended are called **terminal states**.
- UTILITY(s, p): A **utility function** (also called an objective function or payoff function), which defines the final numeric value to player p when the game ends in terminal state s. In chess, the outcome is a win, loss, or draw, with values 1, 0, or 1/2.[2] Some games have a wider range of possible outcomes—for example, the payoffs in backgammon range from 0 to 192.

Approaches to Playing Games

Using a computer to play a game

Ways to play a board game

There are four methods for a computer to play a game.

- Analysis + Strategy + Tactics = Move
- IF-THEN rules
- Look Ahead and Evaluate
- Brute Force - Evaluate the entire tree of Possibilities
- Look ahead as far as possible

#1 Analysis + Strategy + Tactic = Move

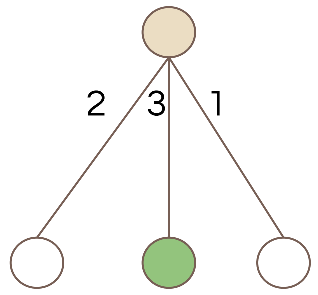
Analysis + Strategy + Tactics = Move

- This method is not computationally successful (or even possible)
- This is how you would play a game
- It has been tried many times and nearly always results in failure
- We don’t have an adequate way of understanding how a human does this

#2 IF-THEN Rules

IF-THEN rules

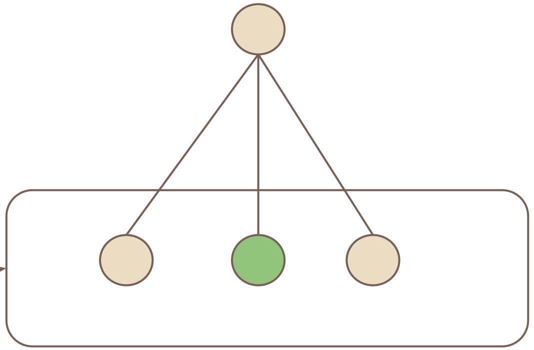
- Look at the board
- Rank the possible **moves**
- Pick the highest rank **move**
    - Doesn't work for chess
    - Does work for Draughts



The choice is obvious here

#3 Lookahead and Evaluate

Here we look ahead at the possible board situations reachable from the root and we ask **which of these situations is best?**



Board Static States

In order to make a decision here, we need a way to evaluate the board state.

Every board configuration is defined by a tuple of features

$$S = g(f_1, f_2, ..., f_N)$$

Each f maps to a chess characteristic such as King Safety, Pawn Structure, Number of Pieces etc.

Linear Scoring Polynomial

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

We are probably going we weight each feature f with some constant a, add these together, and compute a static value **S** for the board in each lookahead, given a board state **s**.

S is more formally called a **weighted linear function**

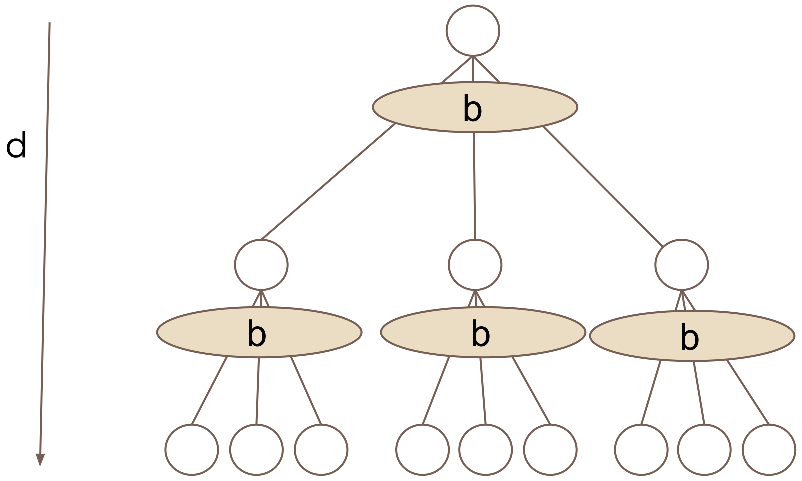
#4 Brute Force

- The idea here is to enumerate all possible paths, and pick the best path for the given situation.
- Before we use this approach however, we need to make sure that enumerating the entire search space is feasible
- Feasibility comes in two parts:
  - Storage feasibility
  - Compute feasibility
- To decide whether it is safe to brute force a problem, lets recap on some important information

Branching Factor and Levels

Branching Factor: The average number of children per node (**b**)

Depth: the total number of levels in the tree (**d**)



Terminal / Leaf Nodes

So the number of Terminal / Leaf nodes is a function of the depth and the branching factor

$$L_d = b^d, d = \{0, 1, 2..., N - 1\}$$

Q: How many leaf nodes do we have where **d**=2 and **b**=3

Back to brute force chess

- Lets assume we have two players playing chess, each player makes 60 moves,
  - giving us a depth of 120
- Now, what is the average branching factor in chess?
  - Most people say this is around 10
- This gives us the number of leaf nodes for Chess Brute Force:

$$L = 10^{120}$$

Is this Feasible?

Lets see how feasible 10^{120} really is:

Size	Task
1×10^{80}	All the atoms in the universe
3×10^{7}	Seconds in a year
1×10^{9}	Nano seconds in a second
1×10^{10}	Number of years since universe began
10^{106}	= Every atom completes one computation every nanosecond since the beginning of time

Brute force - conclusion…

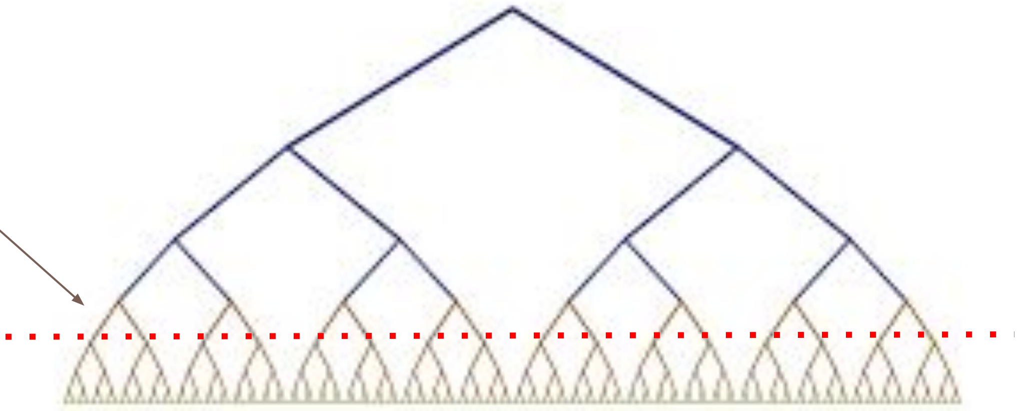
So, if all of the atoms in the universe were doing static evaluations at nanosecond speeds since the beginning of the Big Bang, we'd still be 14 orders of magnitudes short.

#5 Look ahead as far as possible

- As brute force is not feasible, we will look ahead
    - **as far as possible**
- And statically evaluate each of the leaf nodes found
- But how do we account for the adversarial nature of the game?
    - By alternately **minimizing** and **maximizing** the static values **S**

Limit the lookahead

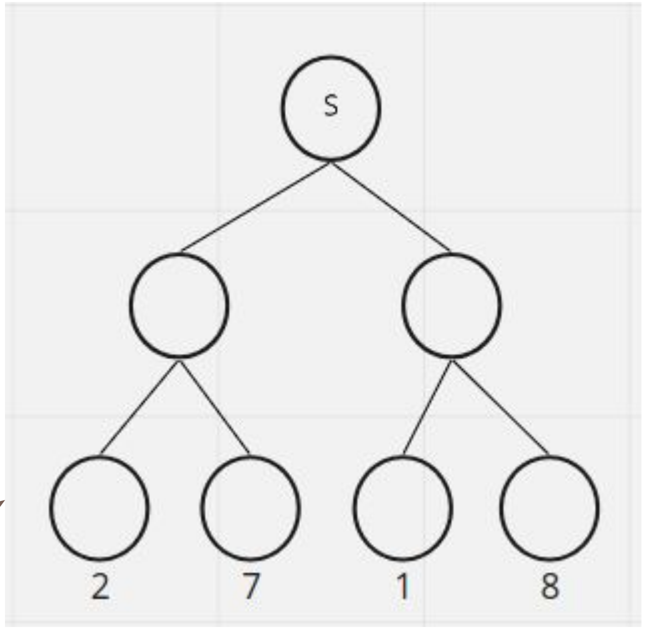
We will lookahead only so far as is computationally feasible (red dashed line), leaving the deeper leaf nodes unevaluated



Adversarial Games

Now we can model adversarial games by using a tree **d**=2 and **b**=2.

The principles shown here apply to all trees of arbitrary **b** and **d**.



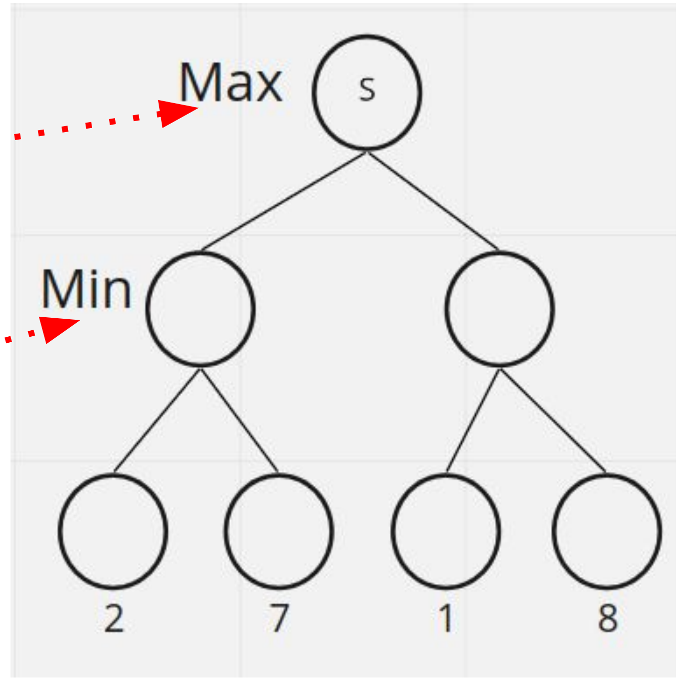
The value of the board from the perspective of the player at the top (ie, the computer)

Min and Max

The player at the top wants to drive the play - as much as possible - towards the big numbers.

We call this player the **maximizing** playrer

But the adversary, or opponent, does not have that objective. They wish to drive towards the small numbers. We call this the **minimizing** player

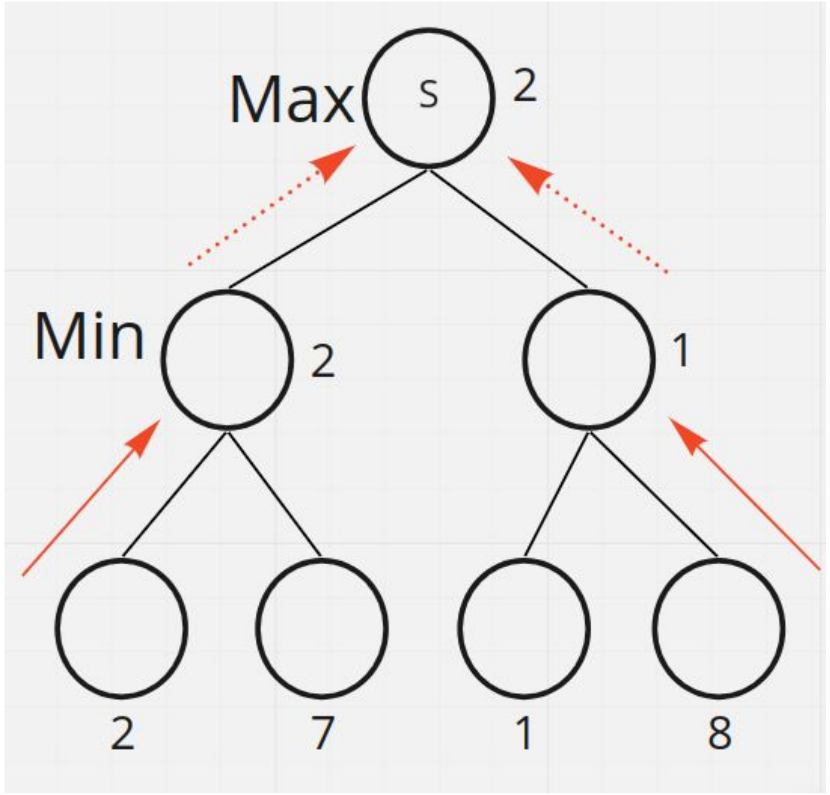


Lets step through this

This the minimax algorithm.

It's very simple.

You go down to the bottom of the tree, you compute static values, you back them up level by level, and then you decide where to go.



Compromise

- And the minimizer goes to the left, too, so the play ends up here:
  - far short of the 8 that the maximizer wanted
  - and less than the 1 that the minimizer wanted
- This is an adversarial game.
- You're competing with another player.
- So, you don't expect to get what you want

In essence, in this case, the game ends in a compromise or averaging effect

Complexity

Once again, lets reflect on the complexity here

Minimax (with ***n*** levels) has a complexity of:

$$\Theta(b^n)$$

Clearly this is **highly undesirable**. So we will try to improve this complexity by using ***αβ*** pruning

Aside

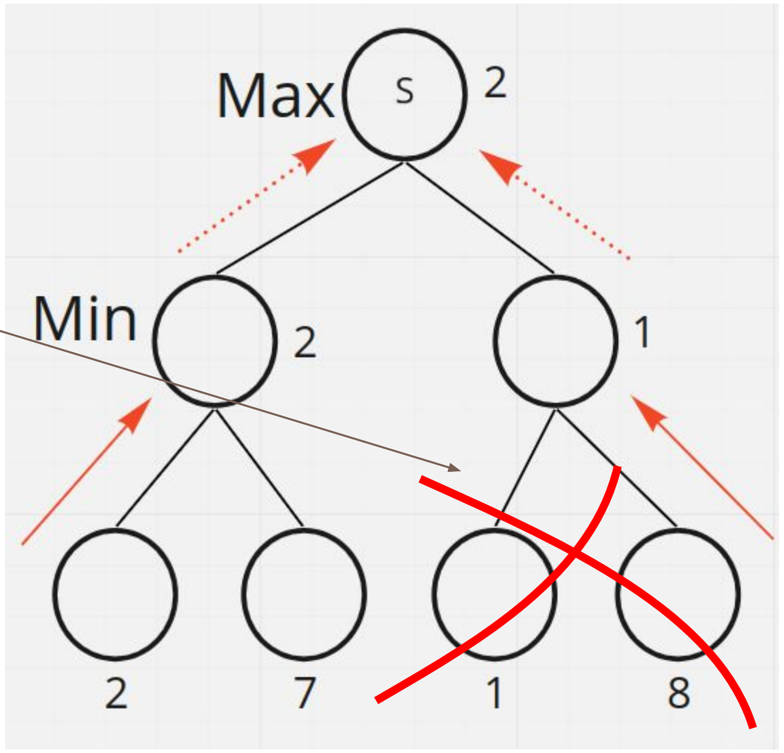
- A chess player that can get down 6 levels deep is probably a reasonable player
- But a player that can get down 15 or 16 levels deep, you could beat a grand master!
- Research has shown that if you get far enough down, the only static metric that really matters is piece count

αβ Pruning Search

Similar to Branch and Bound

αβ pruning

- The minimizer pushes **LHS**=2 and **RHS**=1 up the tree.
- When the Maximizer sees the **RHS**=1 it treats this right branch as if it doesnt exist.
- Thereby eliminating possibly huge computational effort
- The 8 doesn't matter and indeed could have been very large.
- **The RHS is nonetheless pruned**



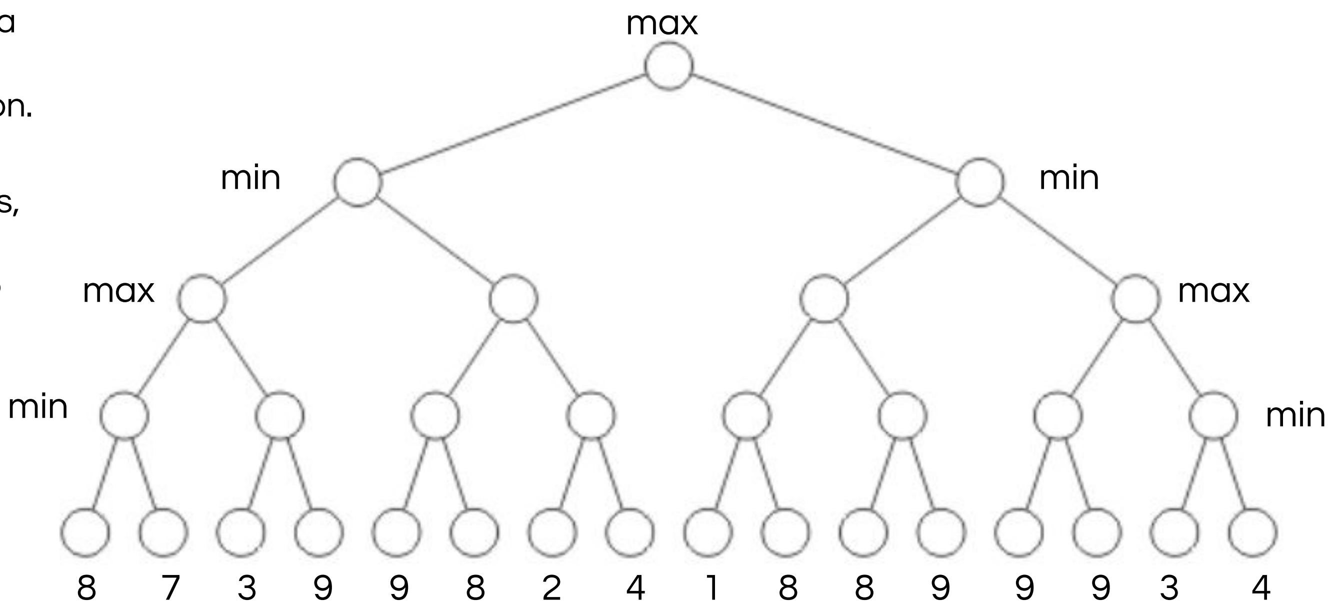
αβ Comments

- The ***αβ*** pruning approach is a layer on top minimax
- It is not a replacement for minimax
- Just like we added features to branch and bound to make it more efficient

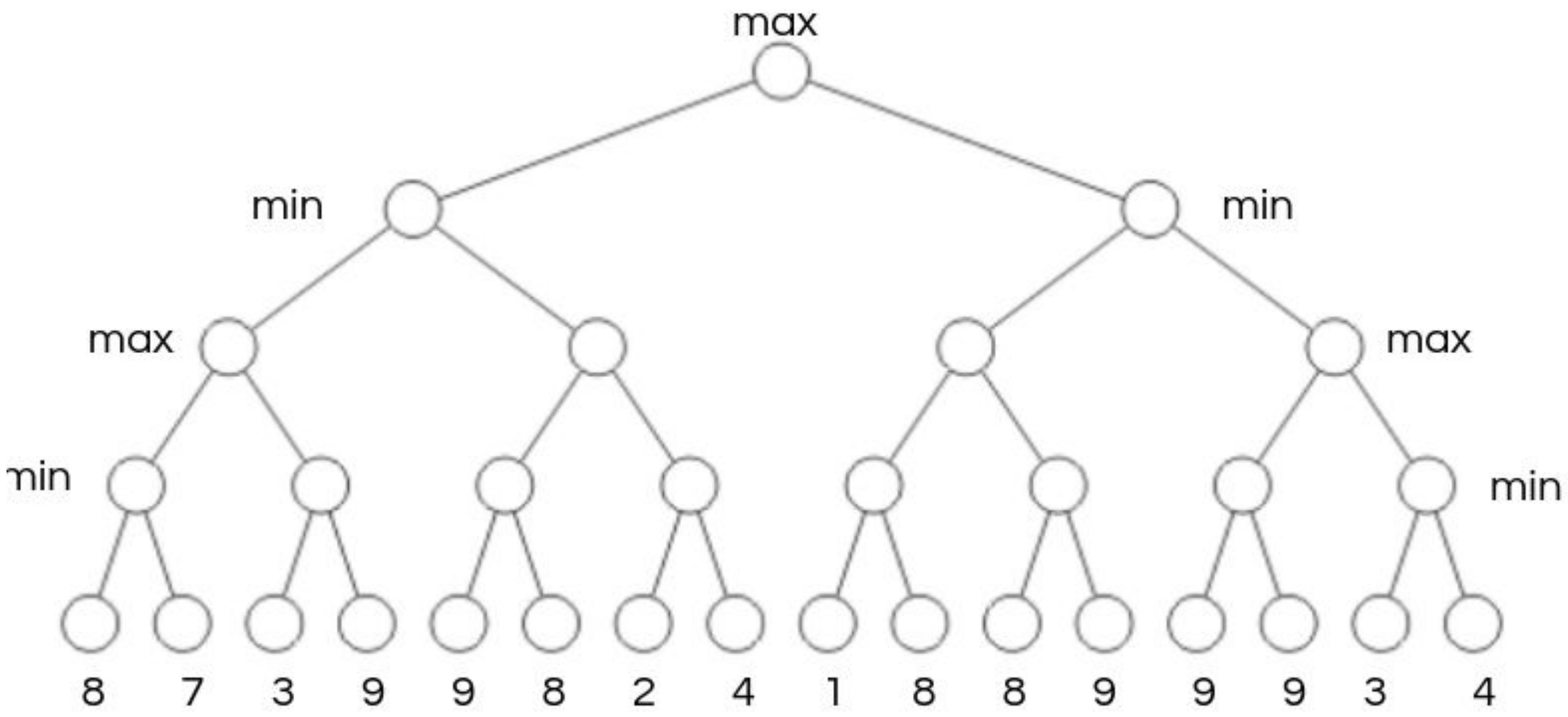
Tree of d=4

Now that we are clear on minimax with ***αβ*** pruning lets consider a case where d=4, a more complex situation.

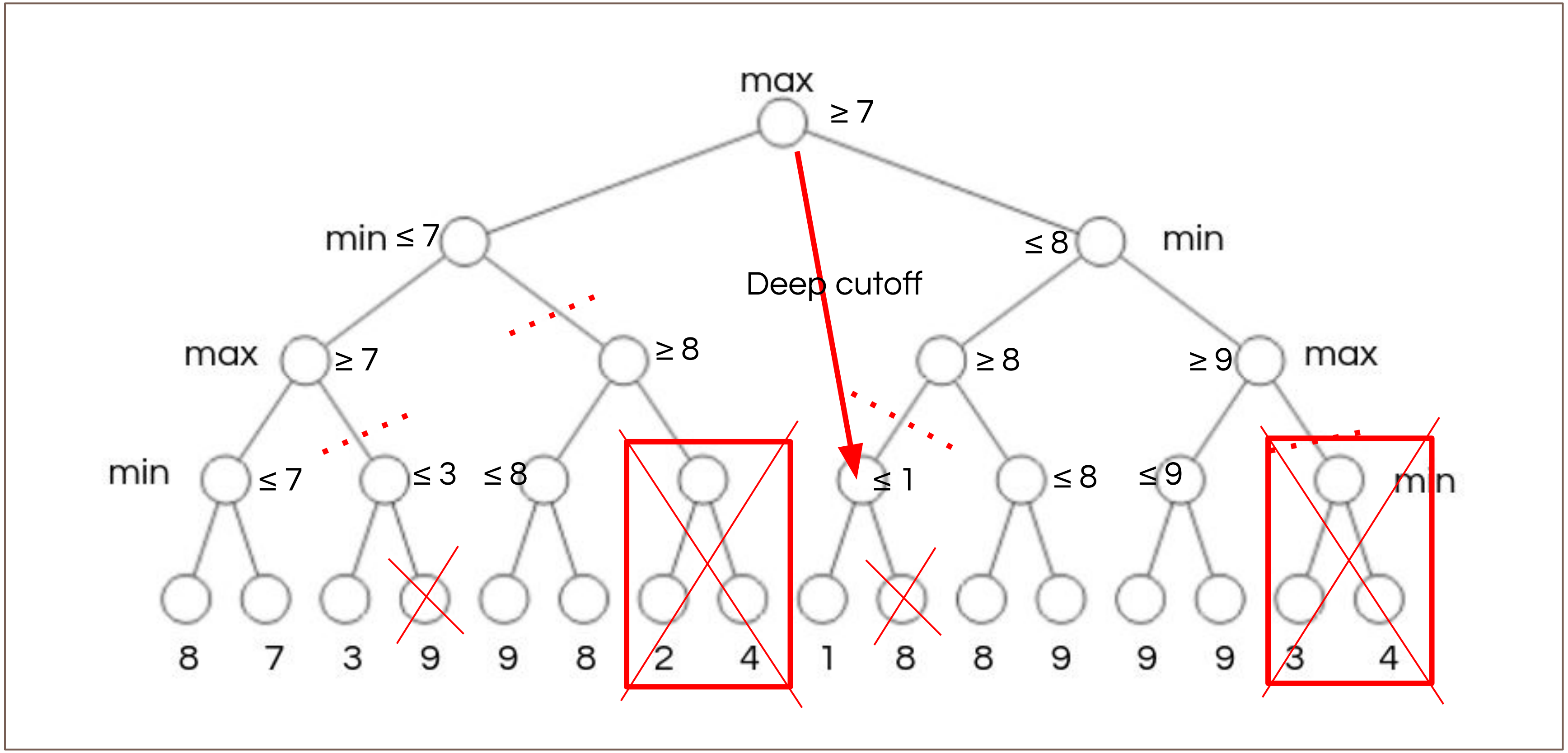
We will also **assume lazy** evaluation. That is, we won't evaluate a leaf node if there is no sense in doing so



Lets work through this



Solution


max
≥ 7
min ≤ 7
≤ 8
min
Deep cutoff
max
≥ 7
≥ 8
≥ 8
≥ 9
max
min
≤ 7
≤ 3
≤ 8
≤ 1
≤ 8
≤ 9
min
8
7
3
9
9
8
2
4
1
8
8
9
9
9
3
4

Final Solution

- The value of 8 the we finally arrive at at the top of the tree is not the minimum and its not the maximum
- It's the compromise number that's arrived
  - by virtue of the fact that this is an adversarial situation.

How good is this solution

In an optimal arrangement, and using ***αβ*** pruning, the actual number of static evaluations is given to us by:

$$S \approx 2b^{d/2}$$

The **d/2** exponent is what makes the difference from having to stop at **d=7** and being able to continue to **d=14**

Question

If we go down the same number of levels, how much less work do we do for **b=4, d=6**

$S \approx 2b^d$ versus $S \approx 2b^{d/2}$

?

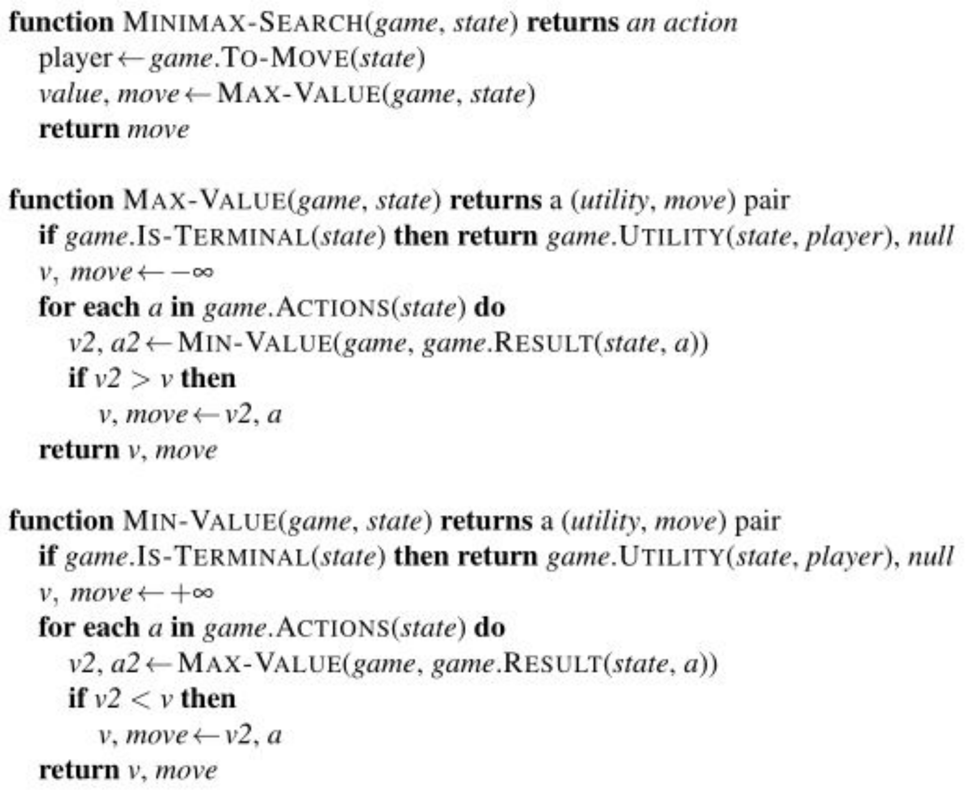
Answer

```
> d <- 6
> b <- 4
> b^d
[1] 4096
> b^(d/2)
[1] 64
> 4096/64
[1] 64
```

Pure minimax algorithm

An algorithm for calculating the optimal move using minimax—the move that leads to a terminal state with maximum utility, under the assumption that the opponent playa to minimize utility.

The functions M AX -VALUE and M IN -VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there



```
function MINIMAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v, move ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v, move ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Minimax with ***αβ*** pruning Algorithm

The alpha–beta search algorithm. Notice that these functions are the same as the MINIMAX -S EARCH functions, except that we maintain bounds in the variables α and β, and use them to cut off search when a value is outside the bounds

```
function ALPHA-BETA-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state, $-\infty$, $+\infty$)
  return move

function MAX-VALUE(game, state, $\alpha$, $\beta$) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  $v \leftarrow -\infty$
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), $\alpha$, $\beta$)
    if $v2 > v$ then
      v, move ← v2, a
      $\alpha \leftarrow$ MAX($\alpha$, v)
    if $v \geq \beta$ then return v, move
  return v, move

function MIN-VALUE(game, state, $\alpha$, $\beta$) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  $v \leftarrow +\infty$
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), $\alpha$, $\beta$)
    if $v2 < v$ then
      v, move ← v2, a
      $\beta \leftarrow$ MIN($\beta$, v)
    if $v \leq \alpha$ then return v, move
  return v, move
```

Progressive Deepening

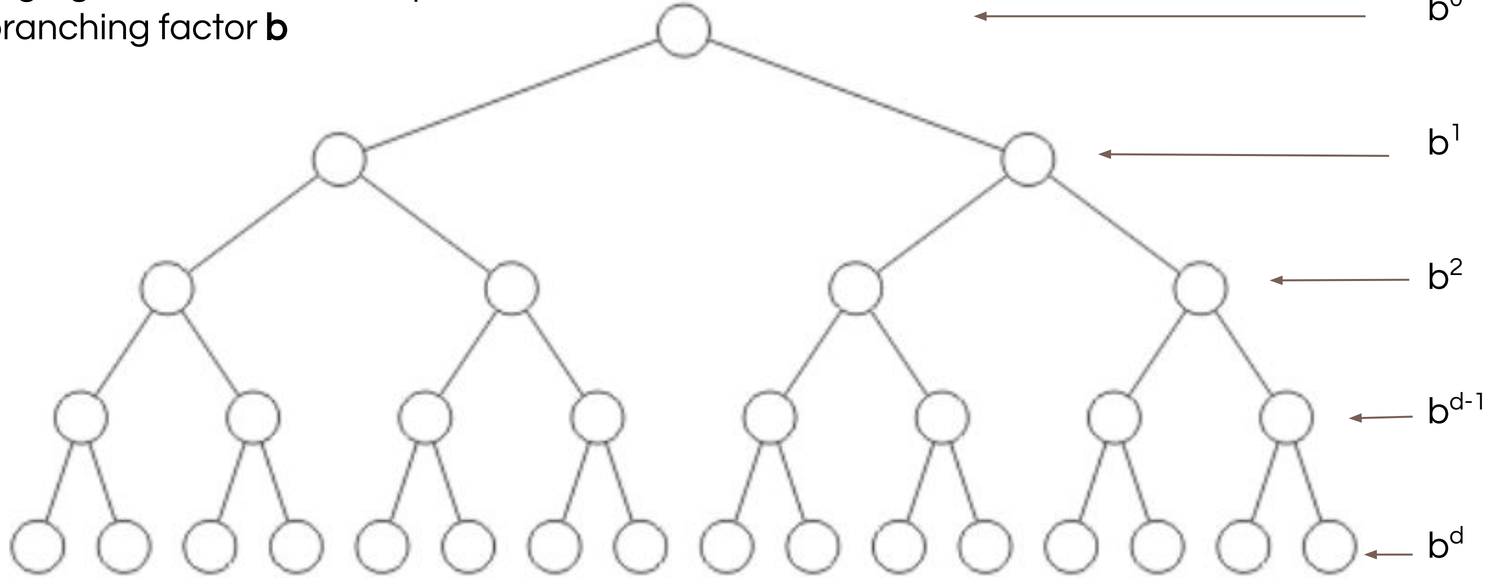
Anytime algorithms

Branching Factor

- Branching factor changes: ie, not constant
- It changes with each move
- It also changes with each game
- We will now consider a final improvement to minimax, known as progressive deepening
- Progressive Deepening is a type of **Anytime Algorithm**
  - No matter when it is called, it always has an answer

Progressive Deepening

Looking again at our tree of depth **d** and branching factor **b**



Bounding the computation time

- Clearly, descending the tree incurs more computations
- But we need to descend as deeply as possible to ensure the best results
- How do we handle the situation where, at level **d**, we spend too much time computing, and the clock expires?
    - We calculate $\mathbf{S = b^d}$ but run out of time
- Clearly we need to compute a solution at
    - $\mathbf{S = b^{d-1}}$ … but what if we run out of time here?
    - Calculate a $\mathbf{S = b^{d-2}}$

How much calculation do we save?

- Suppose **b**=10, we calculate **d**=4:
  - $S = 10^4 = 10000$
- But if we run out of time, or if the branching factor at **d** is too large, lets calculate S for the previous level **d-1**:
  - $S = 10^3 = 1000$
- So by calculating at **d-1** then we compute only 10% of the workload at **d**
- More generally, we compute **1/b** for each previous level
  - $S = 10^2 = 100$

We always have a move...

This is how much we spend getting our **anytime** insurance policy

$$S = 1 + b + b^2 + ... + b^{d-1}$$

This means

$$bS = b + b^2 + ... + b^d$$

$$bS - S = b^d - 1$$

$$S = \frac{b^d - 1}{b - 1} \approx b^{d-1}$$

What this means

So, with an approximation factored in

- the amount of computation needed to do insurance policies at every level
- is not much different from the amount of computation needed to get an insurance policy at just one level,
    - the penultimate level.

This is progressive deepening

Uneven Tree Development

- So far, we've pretended that the tree always grows in an even way to a fixed level.
- But there's no particular reason why that has to be so.
- Some situation down at the bottom of the tree may be particularly dynamic.
- In the very next move, you might be able to capture the opponent's Queen.

Uneven Tree Development

- So, in circumstances like that, you want to expand a little extra search.
- There's no particular reason to have the search go down to a fixed level.
- Instead, you can develop the tree in a way that gives you the most confidence that your backed-up numbers are correct.

Deep Blue Chess Player

Uses:

- Minimax
- $\boldsymbol{\alpha\beta}$ Pruning
- Progressive Deepening
- Parallel Evaluation
- Uneven Tree Development

Finally … 1/2

- Is this a model of anything that goes on in our own heads?
- Is this a model of any kind of human intelligence?
- Is going down 14 levels what human chess players do when they win the world championship?

Finally ... 2/2

- This is substituting raw power for sophistication.
- When a human chess master plays the game,
  - they have a great deal of chess knowledge in their head
  - and they recognize patterns.
- It's very clear that they've Chess masters develop a repertoire of chess knowledge
  - that makes it possible for them to recognize situations and play the game
- The play much more like our very first option #1.

Thank you

Any Questions?

CA318
Advanced Algorithms and AI Search

Introduction to Dynamic Programming

Dynamic Programming

Memoization; Subproblems; Fibonacci Sequence

Dynamic Programming

- Dynamic Programming is a general and sometimes quite powerful method for the development of algorithms
- It can be used to improve the complexity of Brute Force algorithms
  - By removing redundant computations
  - By “remembering” computational states
    - In the form of memos
    - This introduces the term **memoization**

Memoization and Subproblems

As mentioned, Dynamic Programming introduces us to two new concepts in algorithm design:

- Memoization
- Subproblems

Aside

- The term Dynamic Programming is something of a misnomer:
  - The word programming => optimization
  - Dynamic => [sometimes] recursion

Fibonacci numbers

A useful starting point for DP.

Fibonacci Sequence

We are all familiar with the **Fibonacci Sequence**.

It is computed using the well known recurrence:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

Goal: Compute F_n for some arbitrary n

Note: This definition is **recursive**, as we shall see now

Naive Recursive Implementation

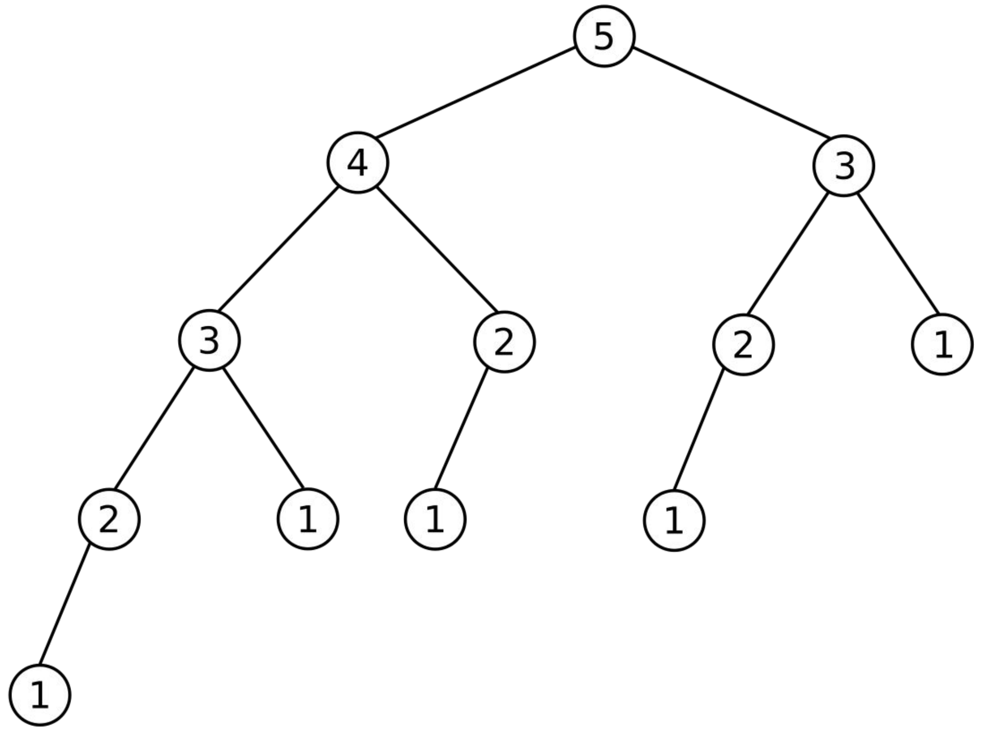
```
fib(n):
    if (n == 0): f = 0
    else if (n == 1): f = 1
    else: f = fib(n-1) + fib(n-2)
    return f
```

Recursive component

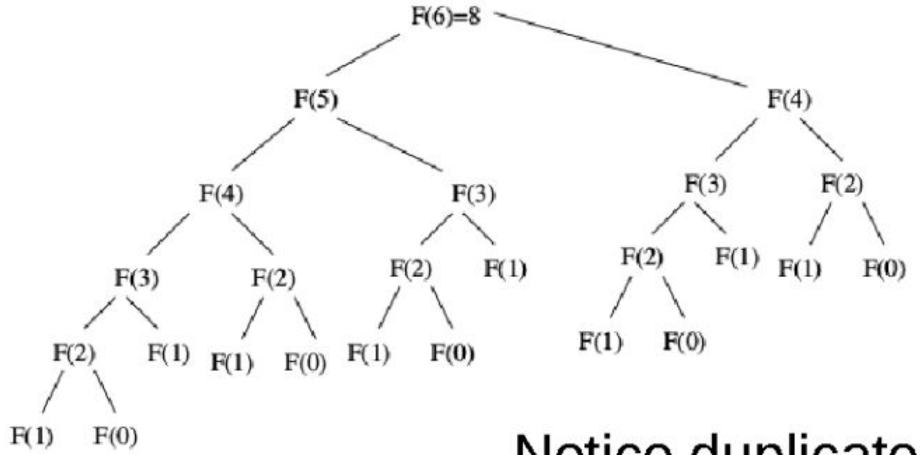
Complexity Calculating F_5

Any suggestions for the complexity of this naive implementation?

Does it remind you of anything?



Problems with Naive Fibonacci



Notice duplicate work:

- f(4) calculated twice
- f(3) calculated 3 times
- f(2) calculate 5 times

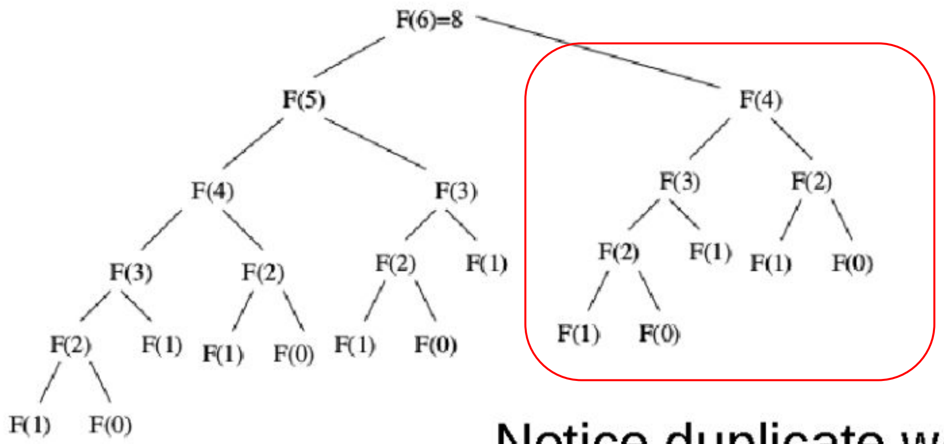
How much duplicate work is done?

We can approximate the answer by using our knowledge of trees where the number of nodes is:

2^{n-1}

But we note that redundant calculations dont just happen in one place, they happen all over. Therefore the redundancy is at least

Which is not good



Notice duplicate work:

- f(4) calculated twice
- f(3) calculated 3 times
- f(2) calculate 5 times

Memoization

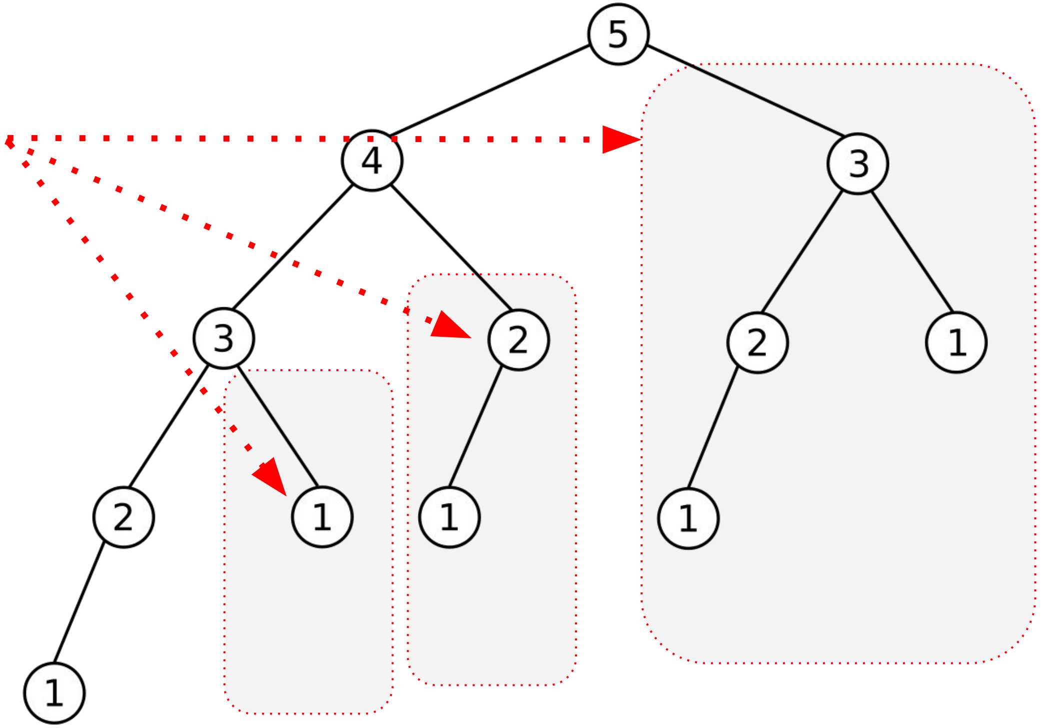
- The idea of memoization is to introduce a data structure (usally called a dictionary):
  - to store the results of already computed values.
- This dictionary has two important properties
  - Constant time lookup speed **O(1)**
  - Constant time insertion speed **O(1)**

Memoized Fibonacci

```
memo = {}
fib(n):
    if n in memo: return memo[n]
    else:
    if (n == 0): f = 0
    else if (n == 1): f = 1
        else: f = fib(n-1) + fib(n-2)
        memo[n] = f
    return f
```

Improvements

Whole sections of the recurrence tree are eliminated with this small improvement



Improvement

`fib(k)` only recurses the first time it is called, for all *k*. All other calculations of *k* are O(1) lookups

The total number of non-memoized calls for any number **n** is `fib(1),fib(2),...,fib(n)` As we can see from the previous slide

Therefore **the complexity of non-memoized work is ... ?**

How much work in `fib(k)` ?

Lets stop for a minute and consider how much work is done in the function `fib(k)`

…any ideas?

```
memo = {}
fib(n):
    if n in memo: return memo[n]
    else:
        if (n == 0): f = 0
        else if (n == 1): f = 1
        else: f = fib(n-1) + fib(n-2)
        memo[n] = f
    return f
```

Complexity for Memoized Fibonacci

$$fib(n) = \Theta(n)$$

In fact, there is a better complexity for computing fibonacci, that uses on the order of

$$fib(n) = \Theta(log\ n)$$

But this is outside the scope of CA318

Subproblems and Memoization

- So we have seen memoization.
- Sub-problems are just small problems along the way to solving the large problem.
- So in the case of `fib(5)` the subproblems are:
    - `fib(4)`
    - `fib(3)`
    - `fib(2)`
    - `fib(1)`
- So we memoize the subproblems
- Subproblems are **not** the goal!

Subproblems

- The subproblems were solved recursively.
- Therefore, we can conclude by saying that dynamic programming is
    - Memoization, and
    - Recursion
- Dynamic Progrmamming - the running time of any solution is:
    - **# of subproblems x time spent per subproblem, eg,**

$$fib(n) = n\,\Theta(1)$$

Subproblems

- Please remember, subproblems are not (mathematically speaking) different function calls in the code
    - **or** different parts of a program taken conditionally based on the code execution path
- Subproblems are **repeatedly executing identical blocks of code**

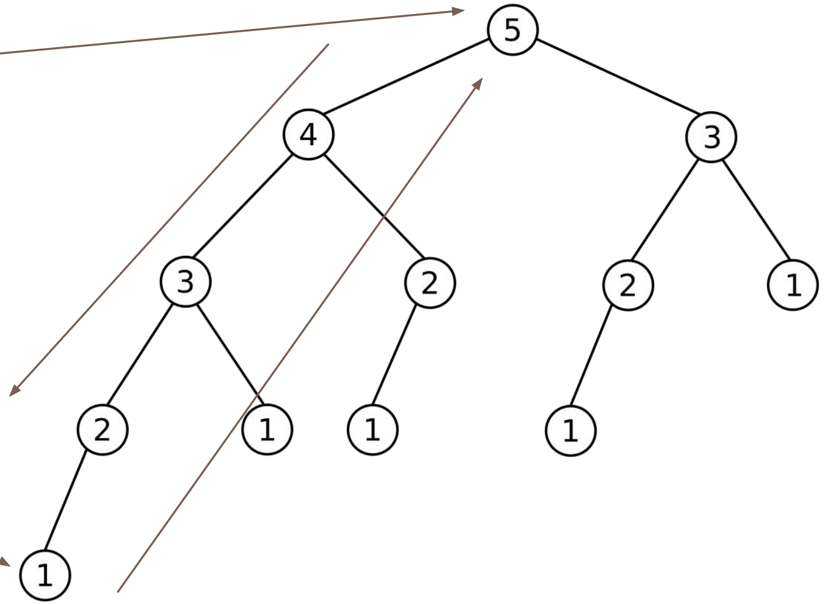
Bottom-up DP Algorithms

An alternative to recursion/memoization

Recursion Starts at the top

The fibonacci tree generation started at the top and recursed until it could go no further. This is **top-down**

But another approach is **bottom-up**. Start here and work your way up. This is called **bottom-up**, but does not use recursion



Bottom-up

If you think about it, bottom-up does not need to use recursion. It can use another approach. So we **unroll** the recursion and write it into a `for`-loop instead

```
fib = {}
fibBU(n):
    for k in range(1:n):
        if k <= 2: f = 1
        else:
            f = fib[k-1] + fib[k-2]
            fib[k] = f
    return fib[k]
```

The same code as the recursive definition

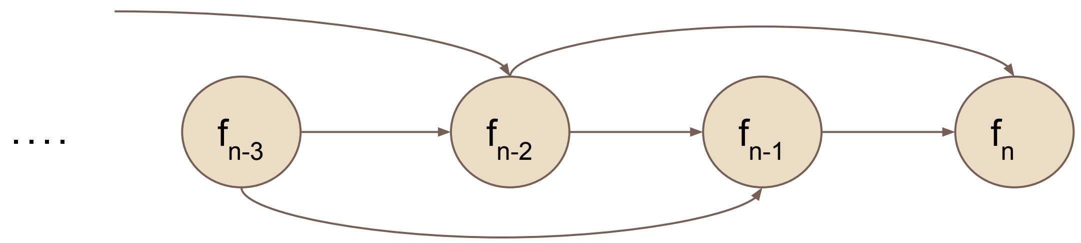
Complexity Check

- Whats the complexity of this approach?
- Is it faster or slower than the recursive definintion?
- Which do you prefer?

Equivalent Computation

In the general cases:

- Bottom-up does exactly the same computation as recursion
- It is merely a topological sort of the subproblem dependency DAG



Space Efficiency

- Bottom-up also has another advantage:
- It is more efficient with space (storage)
- Of course, space is not always a concern nowadays
    - But if **n** is extremely large, then space efficiency is always a good idea

What is the space complexity of the bottom up approach?

Calculating Complexity

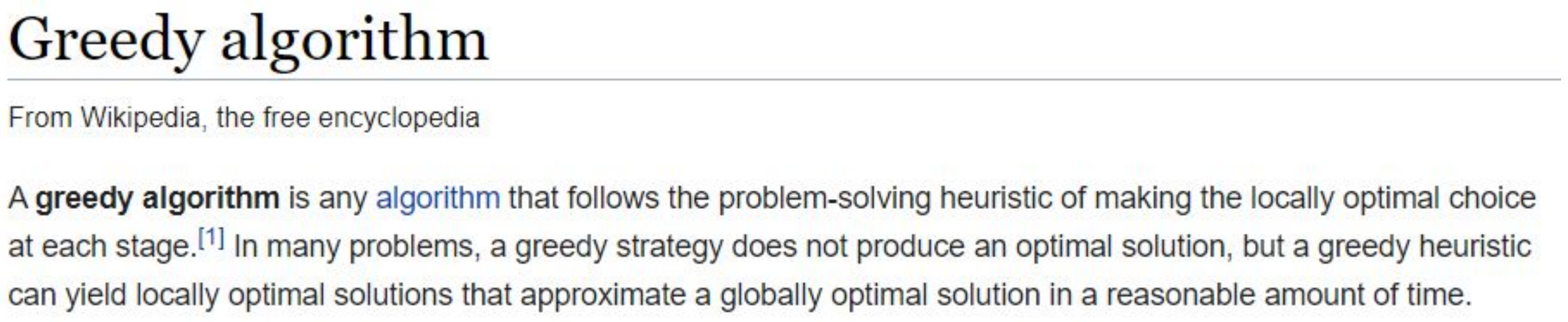
- As mentioned the complexity for this algorithm is the same as the recursive solution
- But something rather interesting to note here,
  - the complexity is obvious
  - there is no recurrence, no memoization,
- so the complexity calculation is trivial:

$$fibBU(n) = n\,\Theta(1)$$

Greedy Algorithms

Making change; Local Optimas; Knapsack Problem

Greedy Algorithms



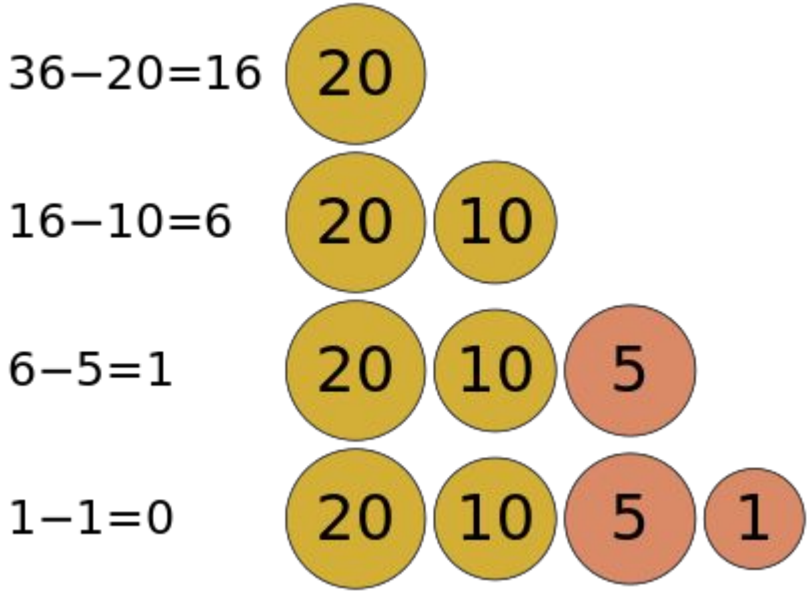
Greedy algorithm

From Wikipedia, the free encyclopedia

A **greedy algorithm** is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage.[1] In many problems, a greedy strategy does not produce an optimal solution, but a greedy heuristic can yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.

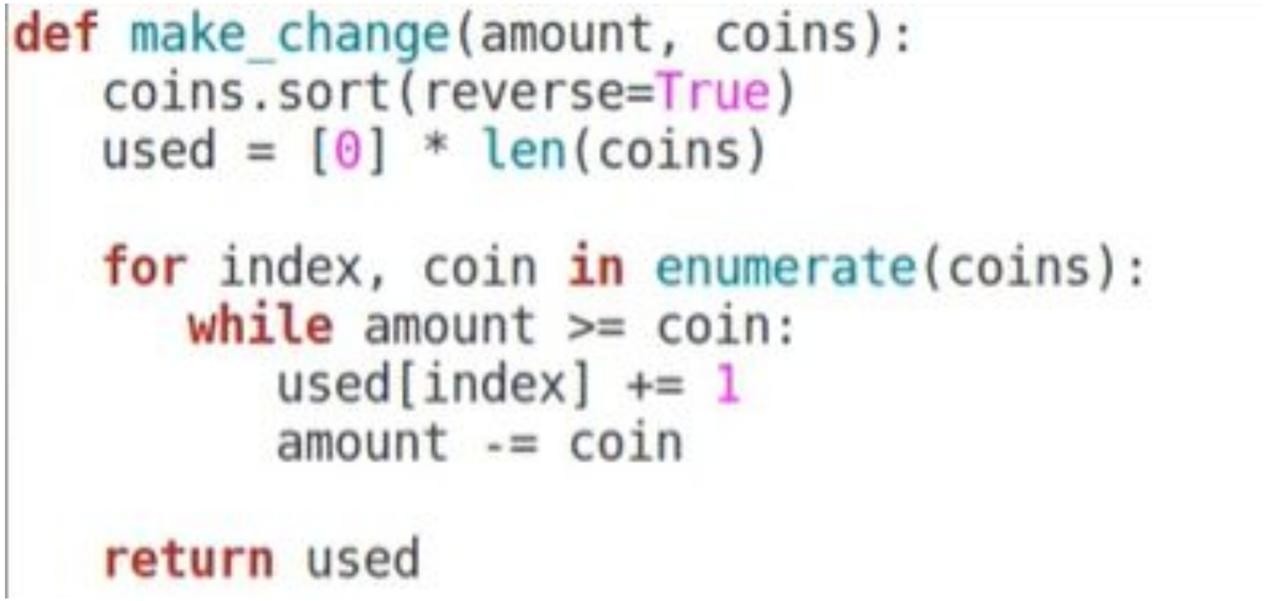
Example

We want to make 36c from the coins 20,10,5,1 as shown here, using as few coins as possible



Naive Implementation

An naive implementation that does not use any dynamic programming efficiencies seen earlier



```
def make_change(amount, coins):
    coins.sort(reverse=True)
    used = [0] * len(coins)

    for index, coin in enumerate(coins):
        while amount >= coin:
            used[index] += 1
            amount -= coin

    return used
```

What change is computed

For the amount A = 767

- Coin set C = [200,100, 50, 20,10, 5, 2,1]
- Output: O = [3, 1, 1, 0, 1, 1, 1, 0]
- Which is correct

For the amount A = 101

- Coin set C = [200,100, 50, 20,10, 5, 2,1]
- Output: A = [0, 1, 0, 0, 0, 0, 0, 1]
- Which is correct

Greedy makes a mistake

For the amount A = 6

- Coin set C = [4,3,1]
- Output: O = [1, 0, 2]
- Which is incorrect - it results in too many coins being used

What has happened here is that greedy has made locally optimal decisions until it no longer can. But it has lost sight of the objective. We need a new method.

Problems

- Obviously, the greedy algorithm as shown here fails in certain cases.
- This is qualitatively different to naive fibonacci - which worked correctly in every case, but with a bad complexity
- But we can still use Dynamic Programming here:
    - Recursion
    - Memoize

Greedy fails

For the amount 36

- Coin set [25, 10, 4, 2]
- Output: [1, 1, `--- ERROR---`]
    - So greedy simply fails
    - Telling us no solution exits
    - But a solution **does** exist
- So now we can see where greedy returns a solution that is not optimal, and greedy simply fails (above)
- Greedy is not guaranteed to work - so we need another approach

Dynamic Programming

If a solution does exist, DP is guaranteed to **always find the optimal solution**

This is in contrast to greedy algorithms that offer no such assurance

Computation Tree

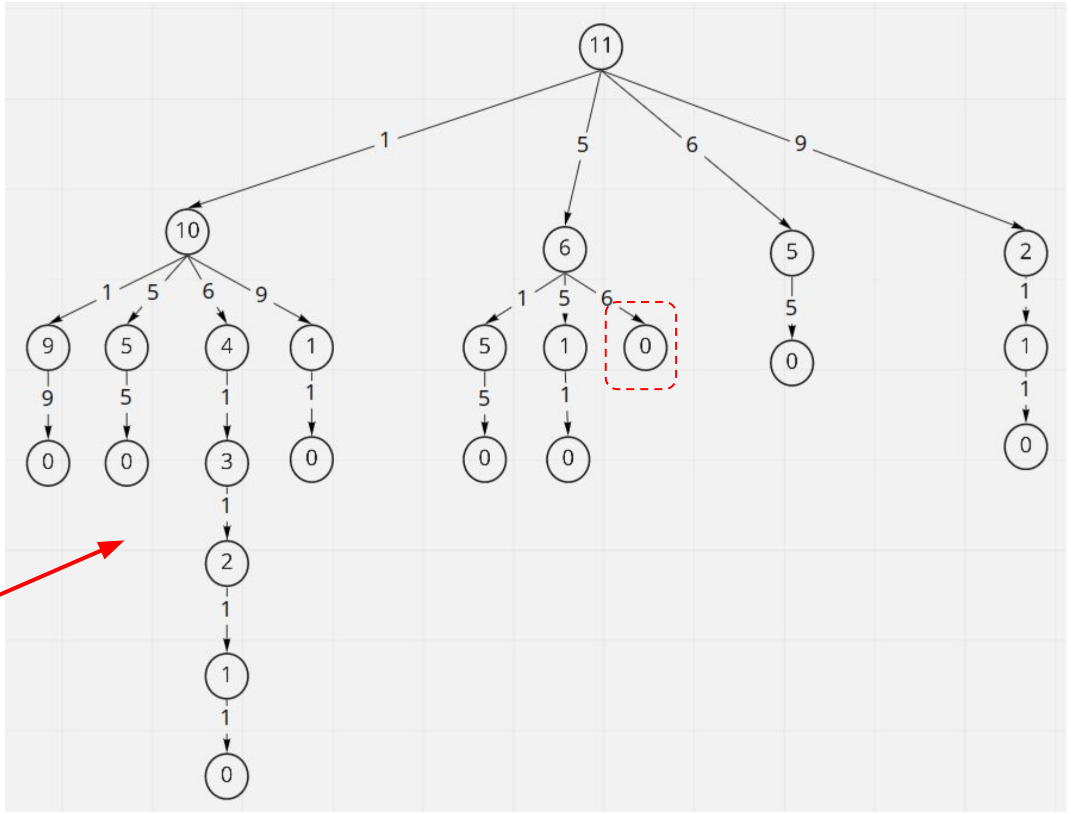
Lets look at this again, using the following amount and coin configuration

A = 11, C = { 1, 5, 6, 9 }

We can see that the greedy algorithm fails to get the optimal solution (9,1,1) wheras we wanted (5 ,6).

Lets recast this in the form of a tree and use our recursion.

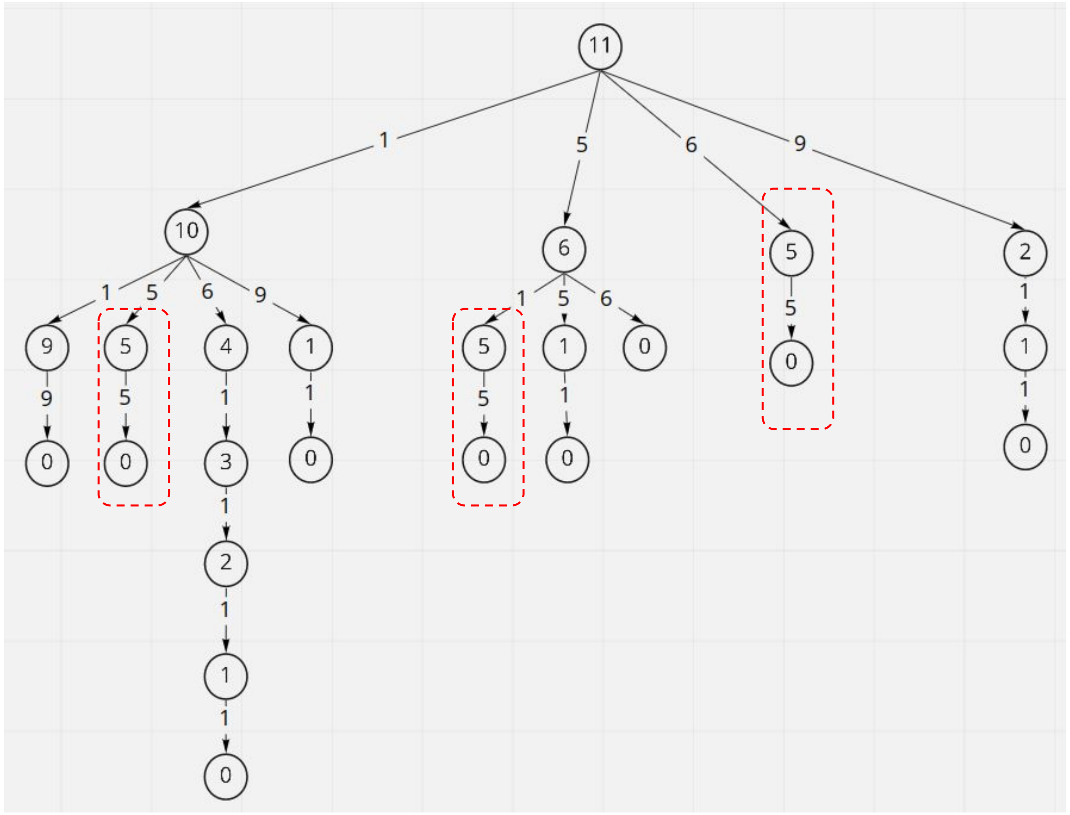
Q: What is the complexity of this tree as shown?



Computation Tree

Once again, repeated computations are obvious.

So recursion on its own does not solve this efficiently



The importance of memo

- This solution has en extremely poor performance without the use of memo
- If you implement it without memo, it could take many seconds to solve the question
- With memo the performance will be instantaneous
- In this example it is slightly clearer if we use loop unrolling rather than recursion

Example

Let us now apply the bottom up method to solve a new problem

A = 10, C = { 1, 5, 6, 9 }, “bottom-up” (no recursion):

j^{th} Amount $\longrightarrow$

i^{th} coin $\downarrow$

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

Step by Step

A = 10, C = { 1, 5, 6, 9 }, "bottom-up" (no recursion):

j^{th} Amount →

i^{th} coin ↓

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

Bottom-upImplementation

The following is a dynamic programming implementation (with Python 3) which uses a matrix to keep track of the optimal solutions to sub-problems, and returns the minimum number of coins, or "Infinity" if there is no way to make change with the coins given.

A second matrix may be used to obtain the set of coins for the optimal solution.

```python
def _get_change_making_matrix(set_of_coins, r: int):
    m = [[0 for _ in range(r + 1)] for _ in range(len(set_of_coins) + 1)]
    for i in range(1, r + 1):
        m[0][i] = float('inf')  # By default there is no way of making change
    return m

def change_making(coins, n: int):
    """This function assumes that all coins are available infinitely.
    n is the number to obtain with the fewest coins.
    coins is a list or tuple with the available denominations.
    """
    m = _get_change_making_matrix(coins, n)
    for c in range(1, len(coins) + 1):
        for r in range(1, n + 1):
            # Just use the coin coins[c - 1].
            if coins[c - 1] == r:
                m[c][r] = 1
            # coins[c - 1] cannot be included.
            # Use the previous solution for making r,
            # excluding coins[c - 1].
            elif coins[c - 1] > r:
                m[c][r] = m[c - 1][r]
            # coins[c - 1] can be used.
            # Decide which one of the following solutions is the best:
            # 1. Using the previous solution for making r (without using coins[c - 1]).
            # 2. Using the previous solution for making r - coins[c - 1] (without
            #      using coins[c - 1]) plus this 1 extra coin.
            else:
                m[c][r] = min(m[c - 1][r], 1 + m[c][r - coins[c - 1]])
    return m[-1][-1]
```

Ref
https://en.wikipedia.org/wiki/Change-making_problem

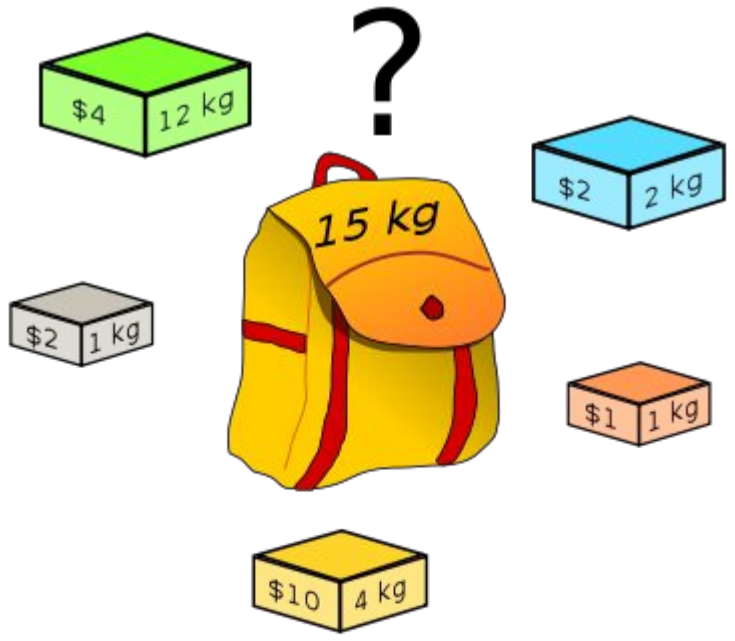
Bottom-up Implementation

Using the previous example where we derived an estimate for the complexity of fibonacci, can you work out a complexity estimate for the recursive memoized making change implementation on the previous slide?

Knapsack Problem

The knapsack problem is a problem in combinatorial optimization:

Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.



Finally

The knapsack problem is similar to the coin changing problem, except that we have to model a weight **and** a cost property rather than a size.

Both of these problems are problems of **Optimization** - where you are looking for the maximum/minimum function. We will revisit this in a future lecture

Thank you

Any Questions

CA318
Advanced Algorithms and AI Search

Introduction to Learning: Nearest Neighbours & Identity Trees

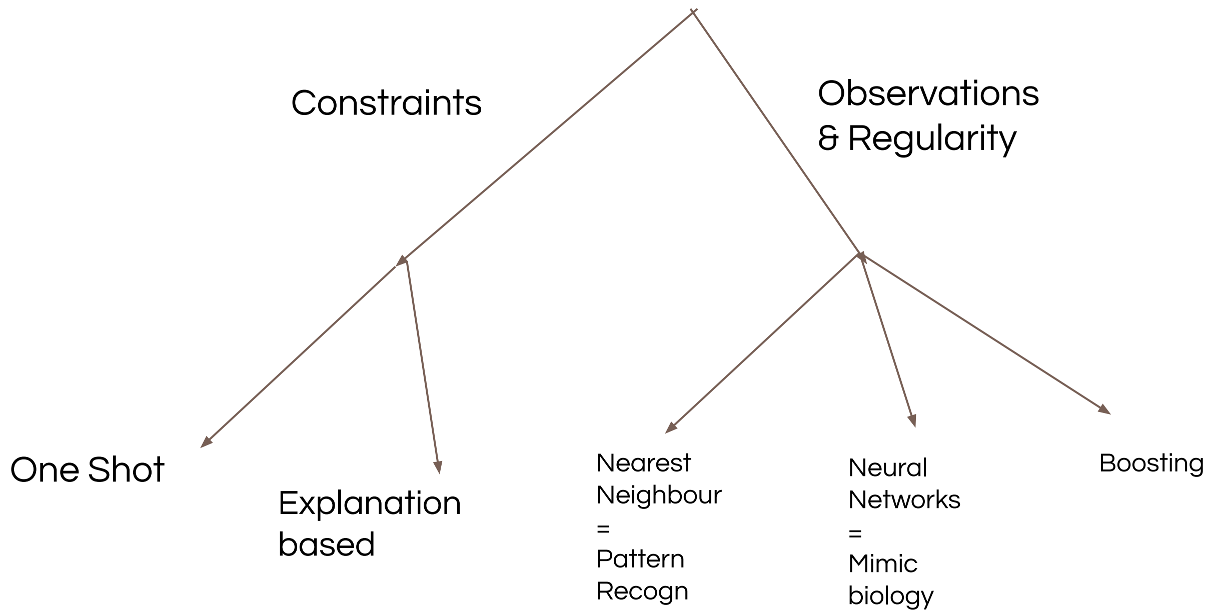
Learning: Part 1

Nearest neighbours; Decision Boundaries

Types of Learning

- Of course, our objective is always to understand how humans learn so well
- We can then mimic this approach
- The objective being to make a machine as capable as learning as a human is
- Our first step then is to characterize learning methods

Types of Learning


Constraints
Observations & Regularity
One Shot
Explanation based
Nearest Neighbour = Pattern Recogn
Neural Networks = Mimic biology
Boosting

Nearest Neighbour Learning

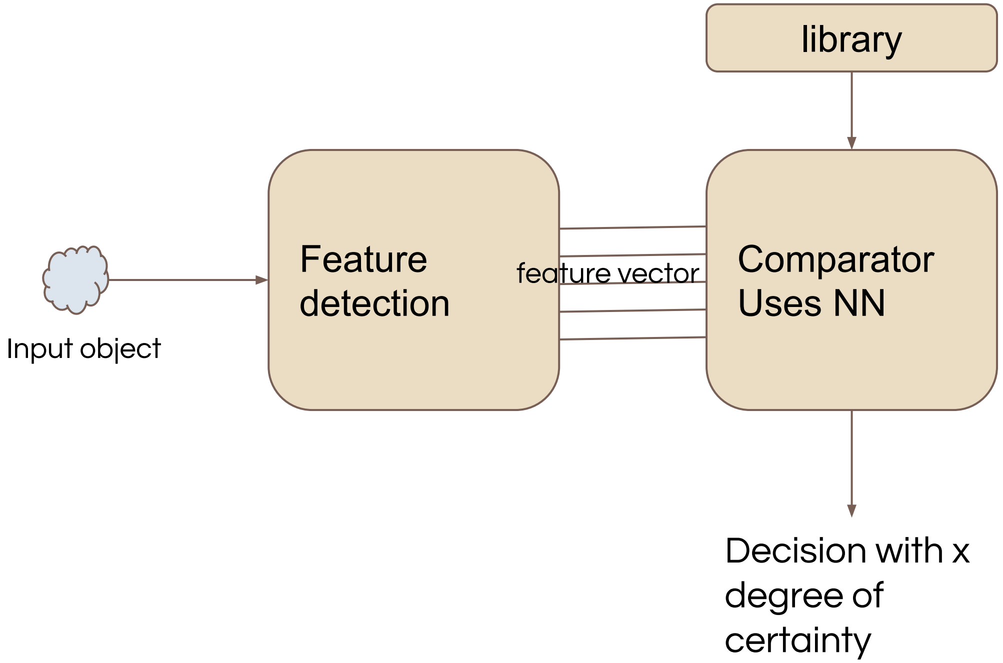
The key components of NNL are:

- Feature Detection
- Feature Comparator
  - Library of vectors
- Recognition with **x** degree of certainty

Decision Flowchart

An input object is scanned and the key features are computed in the form of a feature vector.

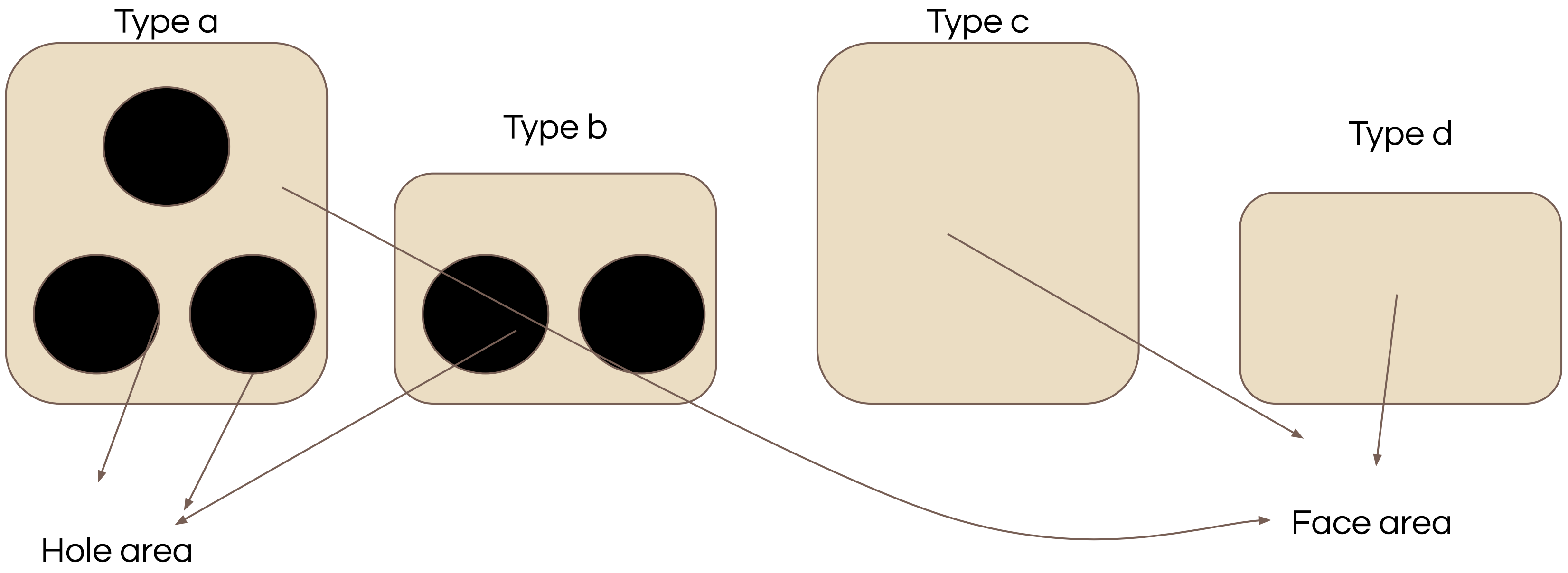
The vector is passed to a comparator, which computes a similarity measure based on NN distance, and a decision with x degree of certainty is computed



Example

- To make this process more concrete, lets consider the situation of a company that makes widgets.
- These widgets have two basic features
  - Area of the widget face A_f
  - Area of the holes in the widget A_h
- Suppose 4 widgets **a,b,c,d** are created on an assembly line
  - We want to **classify** each widget
  - Then we sort the widgets for packing and distribution

New Type of Widget



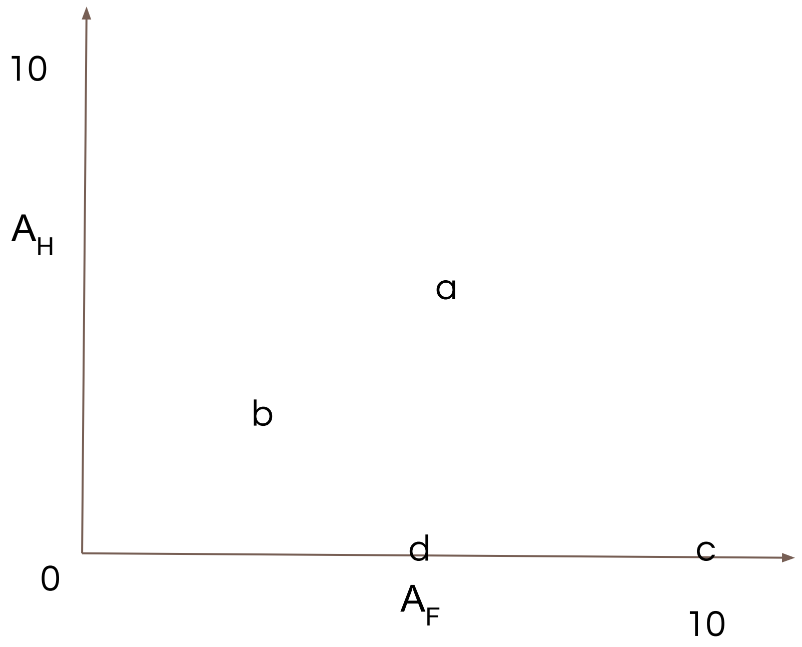
Example Features

Widget feature vector

Widget	A_F	A_H
a	5	5
b	2	2
c	10	0
d	5	0

Plotting the vectors

These vectors are 2 dimensional, so we plot these on the standard cartesian plane:



A New Widget **e**

Suppose a new widget (e) is scanned and and we find its feature vector as follows:

Widget	A_F	A_H
a	5	5
b	2	2
c	10	0
d	5	0
e	4	1

Classifying **e**

Our first solution is to compute how far **e** is from all the other feature vectors and see which one is its nearest neighbour:

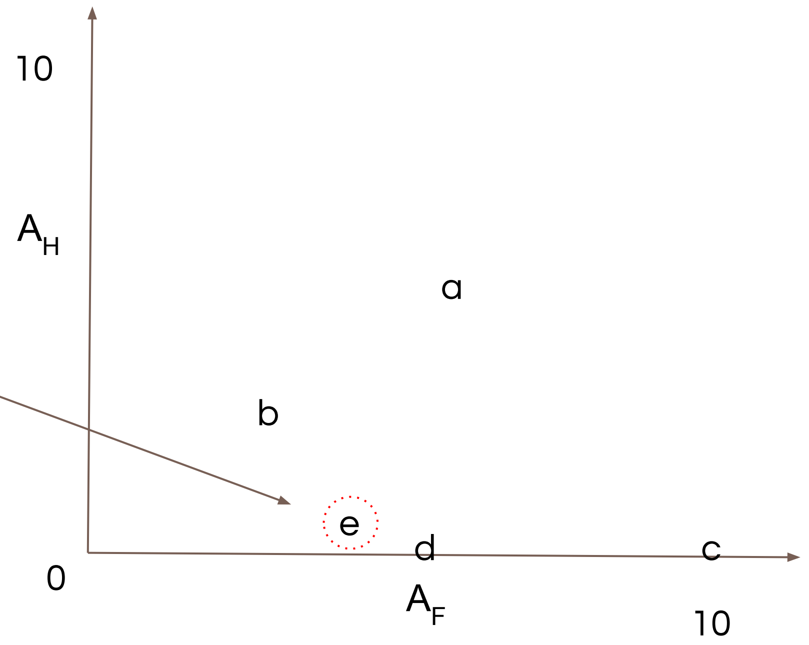
Widget	A_F (x)	A_H (y)	d_e
a	5	5	4.12
b	2	2	2.23
c	10	0	6.08
d	5	0	$\boxed{1.41}$
e	4	1	0

$$d_e = \sqrt{(x_i - x_e)^2 + (y_i - y_e)^2}$$

Choose the minimum

Plotting **e**

As you can see, e is clearly closest to d



Problems with distance **d**

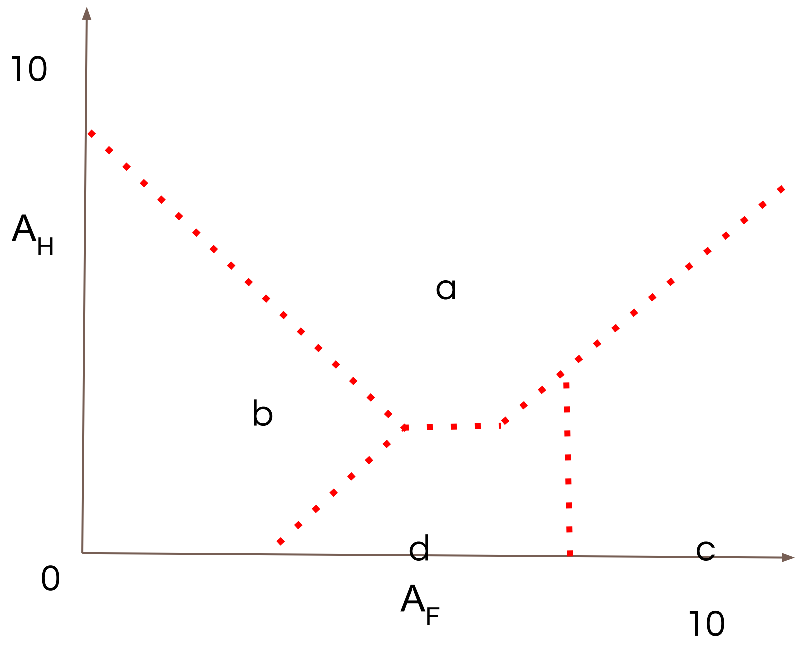
One of the problems with the previous approach, where we used the formula:

$$d_e = \sqrt{(x_i - x_e)^2 + (y_i - y_e)^2}$$

- Is that for the most part, the widgets themselves are not going to be exactly located in the **n**-dimensional space of features.
- They will have some degree of variance
- So instead of exact points we should use a bounding box around the features instead

Bisect the space

So we construct perpendicular bisectors between each of the pairs of features thus:

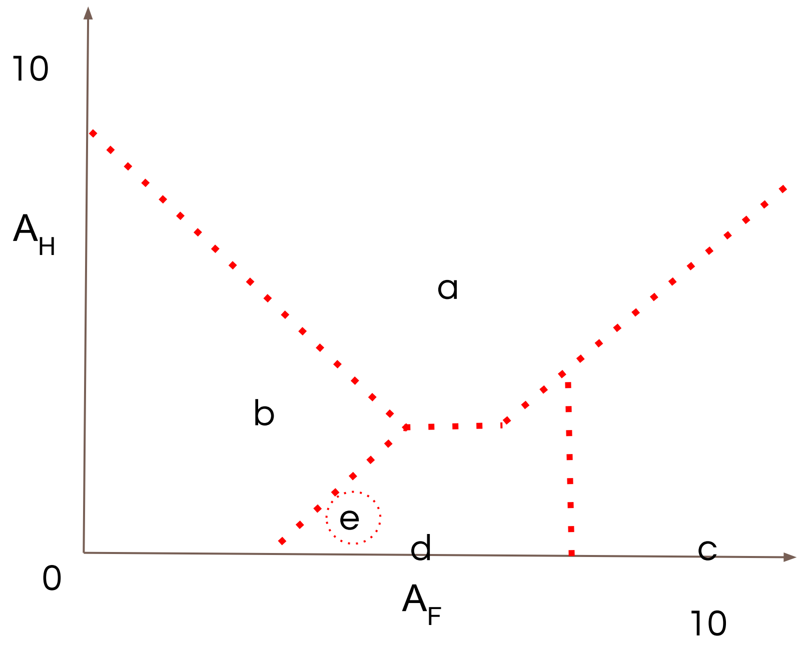


Decision Boundaries

Each of these bisections is called a decision boundary.

Now we have another way to answer the question **which widget does e belong to**…what is it?

Do we get the same answer? - **yes**



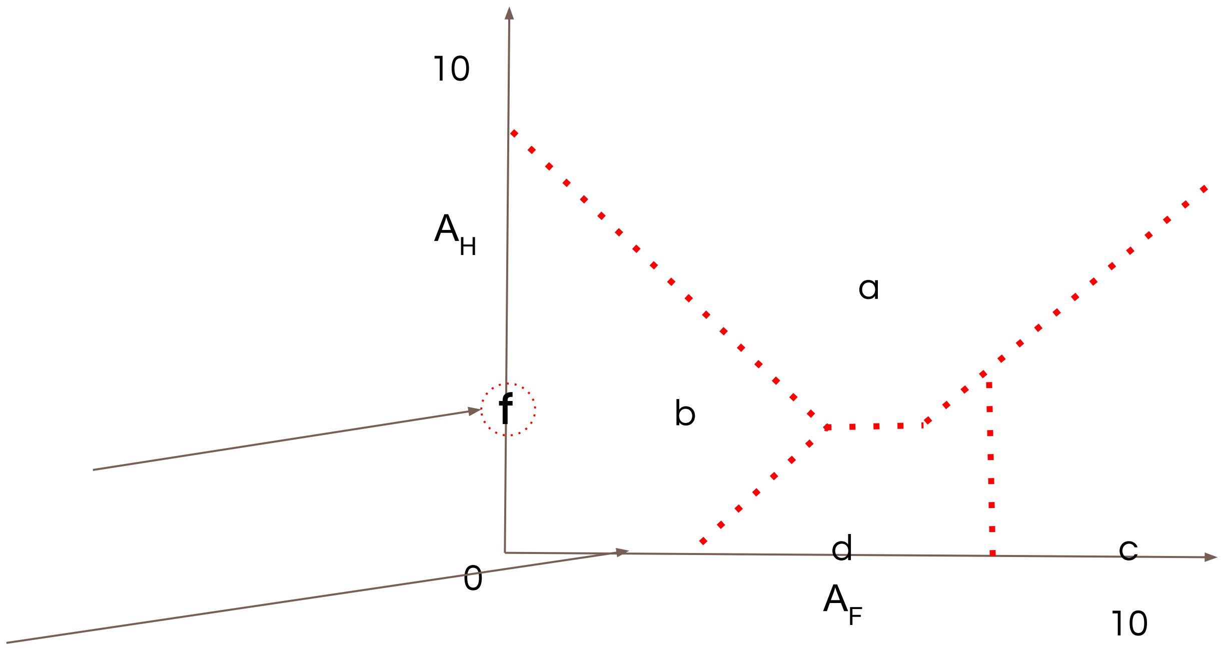
Another widget! (**f**)

- Suppose yet one more widget is scanned but we instead of getting a complete vector of information, we only get the A_H data
- Suppose it turns out that for the widget **f** $A_H = 2$
- What could we say about A_F?
- We could say that
  - “If something is similar in some respects, its likely to be similar in other respects”
  - So therefore, [weakly] we could say for f that $A_F=2$

Computing A_F for **f**

"If something is similar in some respects, its likely to be similar in other respects"

=> the area of the face of **f** is likely to be 2



Education and Learning

- Medical cases
- Legal cases
- Business cases
- -etc-

We will often apply the principle: “If something is similar in some respects its likely to be similar in other respects”

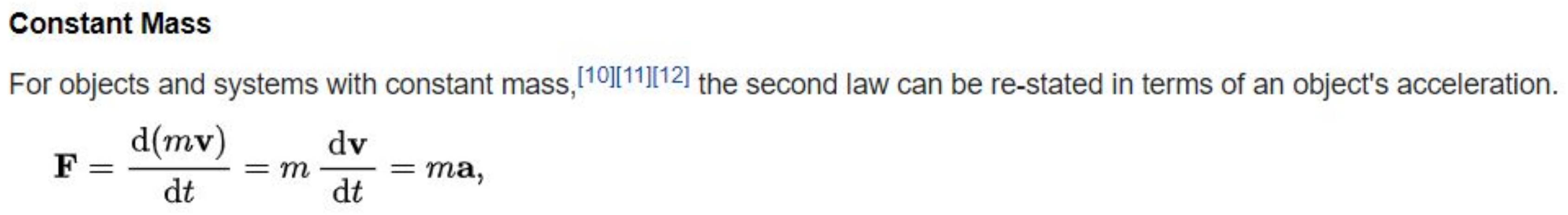
This is one of the key features of education and learning

Throwing a ball

Its easy…right?

Newtons 2nd Law

In order compute the force required to throw a ball we could reference Newton’s 2nd Law:



Constant Mass

For objects and systems with constant mass,[10][11][12] the second law can be re-stated in terms of an object's acceleration.

$$\mathbf{F} = \frac{\mathrm{d}(m\mathbf{v})}{\mathrm{d}t} = m\,\frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t} = m\mathbf{a},$$

But for a computer to really throw an actual ball, the maths is much more complicated

Learning by maths models

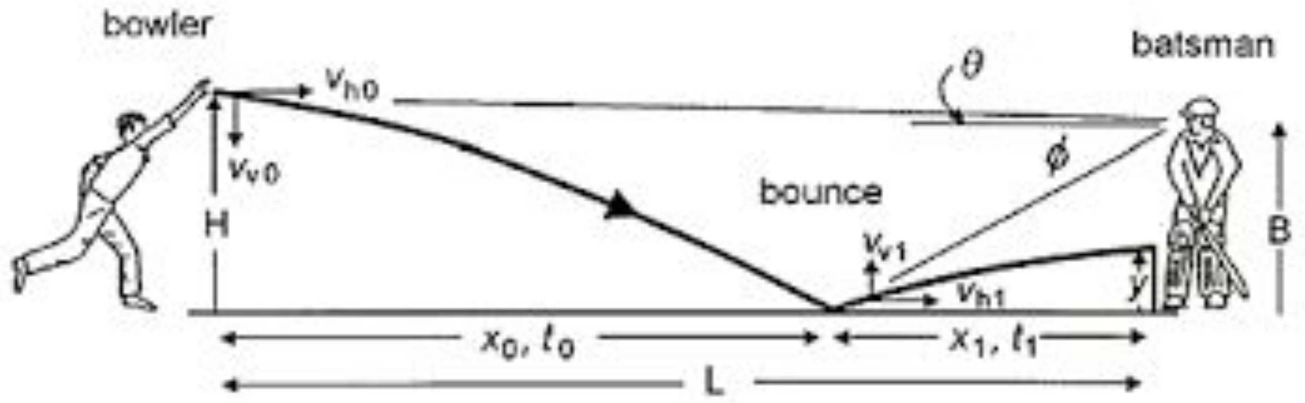
The equation to model how a computer controlled arm should throw a ball a certain distance is given to us by this rather complex mathematical formula

$$\begin{aligned}
\tau_1 =& \ddot{\theta}_1\left(I_1 + I_2 + m_2 l_1 l_2 \cos\theta_2 + \frac{m_1 l_1^2 + m_2 l_2^2}{4} + m_2 l_1^2\right) \\
&+ \ddot{\theta}_2\left(I_2 + \frac{m_2 l_2^2}{4} + \frac{m_2 l_1 l_2}{2}\cos\theta_2\right) \\
&- \dot{\theta}_2^2 \frac{m_2 l_1 l_2}{2}\sin\theta_2 \\
&- \dot{\theta}_1 \dot{\theta}_2 m_2 l_1 l_2 \sin\theta_2, \\
\tau_2 =& \ddot{\theta}_1\left(I_2 + \frac{m_2 l_1 l_2}{2}\cos\theta_2 + \frac{m_2 l_2^2}{4}\right) \\
&+ \ddot{\theta}_2\left(I_2 + \frac{m_2 l_2^2}{4}\right) \\
&+ \dot{\theta}_1^2 \frac{m_2 l_1 l_2}{2}\sin\theta_2.
\end{aligned}$$

Learning by maths models

- But when you were a child and learned to throw a ball, you did not do it by referencing a Newton’s 2nd Law,
    - Or the more complex mathematical model
- So how do we do learn how to it?
- Humans learn by subconsciously populating a table of values from their experience
- Each time we learn a new experience we store this in a table
- When we want to reproduce the result, we lookup the table

Cricket Time



Bowler Parameters

This would be a huge table for a professional athlete

wind	distance	height	speed	...	T_{wrist}	T_{arm}	$T_{shoulder}$
2	5	3	5	...	x	y	z
1	3	4	6	...	x	y	z
					x	y	z

Lookup answer

Look up the table

- Traditionally, it was felt that these parameter tables would be too huge for quick computer look up
- So the idea of having the computer learn parameters was considered too costly in space and time
    - But nowadays we can store infinite quantities of data with very little latency in look ups
- So we can teach a computer to do something by having it store previous successful vales
    - And just look them up

Tables are huge

- Realistically, such tables of parameters could be 10^{12} in size
- But our brains are capable of storing at least 10^{14} so of course humans can comfortably do this.

Q: How much SSD storage is needed to store a 10^{12} table of bytes? - is it feasible on a desktop computer?

Complexity - again

What would be the complexity of a table look up with **n** rows, assuming we had **m** parameters?

- Bad design: **?**
- Good design: **?**

Learning: Part 2

Identification Trees, Disorder and Entropy

Nearest Neighbours again

Nearest neighbour calculations is perfectly feasible if we can create a numeric vector of features:

v is a vector of features in 4-dimensional space

$$v = \begin{pmatrix} 3 \\ 4 \\ 1 \\ 7 \end{pmatrix}$$

Feature Space

- The problem here is that many features sets cannot be adequately described in terms of vectors of integers
- Remember, each integer in **v** is a point in **n**-dimensional space
- So in nearest neighbours we can easily subtract the $\mathbf{v}_i$ and $\mathbf{u}_i$ to get the distance between the **i**-th point in **u** and **v**
- But many feature sets cannot be described numerically like this

Problem

- Suppose we wish to identify an airplane based on characteristics.
- Specifically, we want to know the important characteristics needed to identify a Boeing 787
- Lets suppose there is some uncertainty
  - For example, suppose different iterations of the B787 had somewhat different characteristics
- Suppose we visit the airport and collect some data on airplanes

Airplane Feature Set

Narrow Body	Engine Shape	Winglets	Engine Noise	B787
Yes	cerated	?	quiet	No
Yes	straight	Yes	quiet	No
No	straight	?	quiet	Yes
No	curved	No	loud	Yes
No	curved	?	medium	Yes
No	cerated	Yes	loud	No
No	straight	Yes	loud	No
Yes	curved	?	medium	No

No distances here

Clearly we cannot do any form of distance measurements - but we still want to perform recognition of a **B787 with some degree of certainty.**

How do we recognize here?

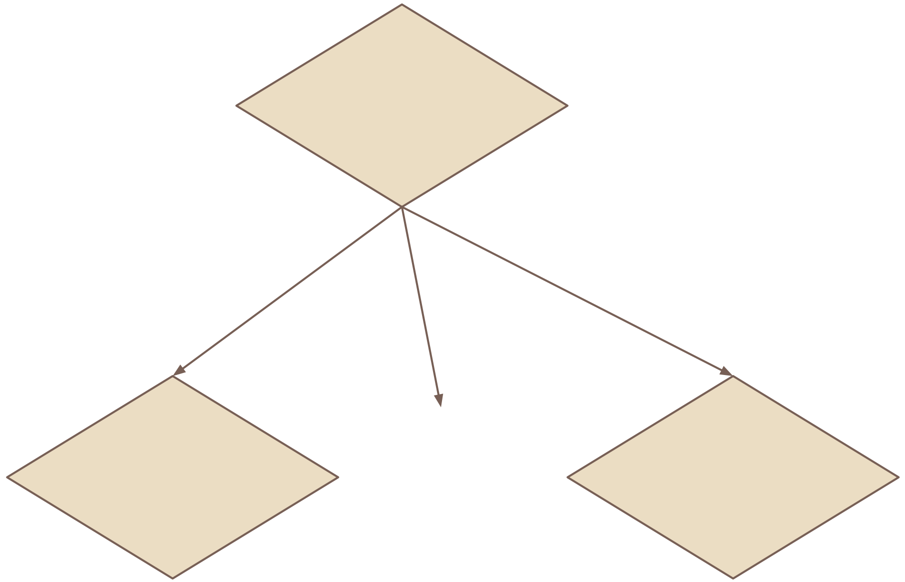
- We want to recognize the B787
- But our data is non numeric
- Therefore distance metrics from NNL does not apply
- We also have data where the answer is unknow (?)
- Some features are more significant than others
- Cost - collecting some of these data is more costly than others
    - eg: Engine noise
- What can we do?

Tests

- Each of the features in the database can be considered a **test** of some kind
- We take our samples, and we arrange our tree of tests
- We process each sample and test the outcome
    - **We rate each test according to the number of individuals it puts into homogeneous sets**
    - We pick the test with the best rating
- We branch according to the outcome of the tests

Identification Tree

We arrange the tests like this - and check each sample against the logic. This is called an **identification tree**



Identification Tree

We need to make a **good** identification tree

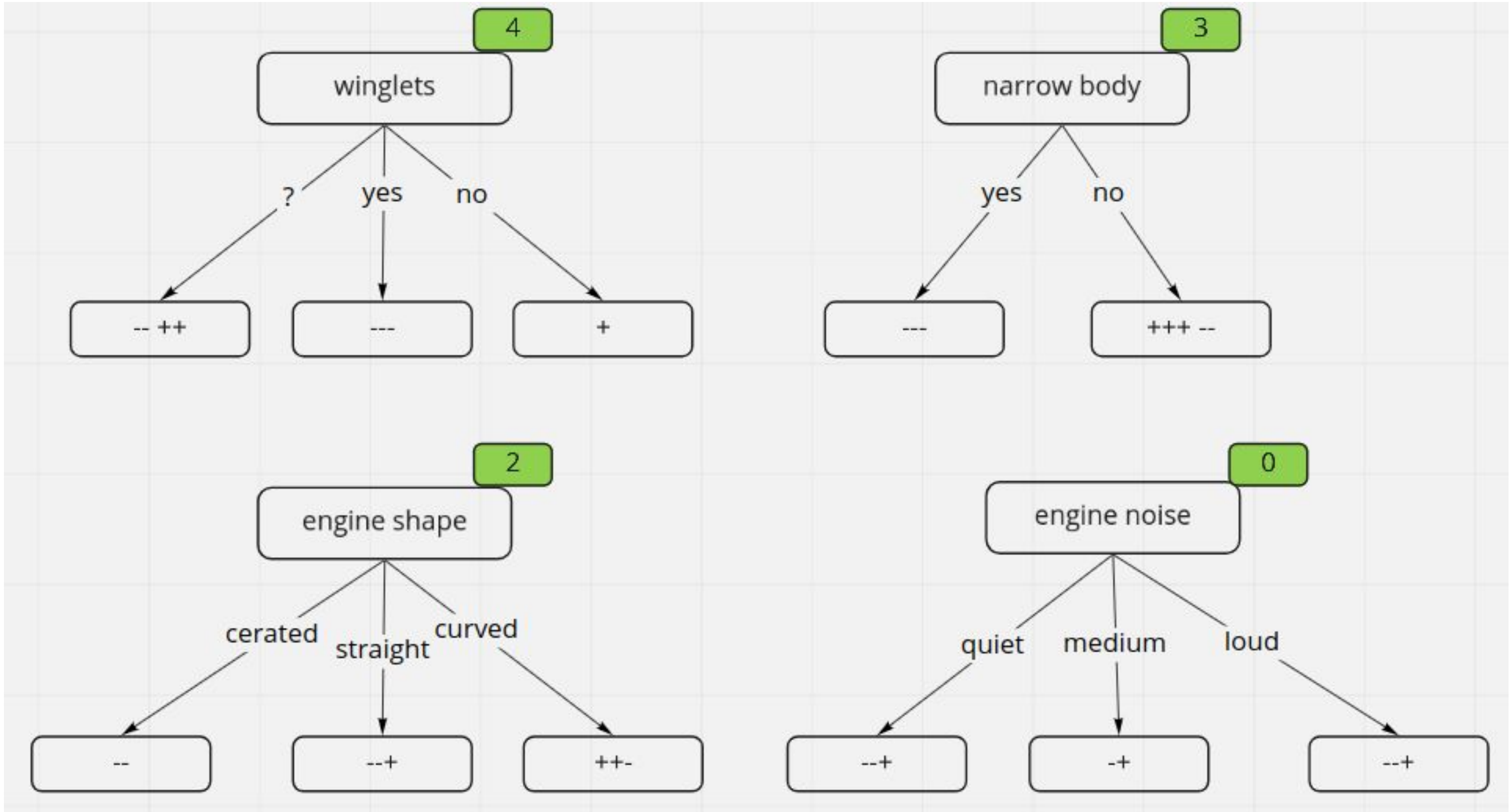
- Small
  - Occams Razor
- Low cost
- Uniform subsets at leaf level
  - At the leaf level, we should have all the B787 aircraft identified together
  - And all the non B787 aircraft identified together

Brute force or Heuristic Methods

- Even though the dataset shown is tiny, in general, these trees cannot be enumerated in a brute force way
- Because trees are generally speaking exponential problems
- So we need a better approach
- Perhaps, a **heuristic**** approach

**of course, heuristics can be dangerous

Each Test with Homogeneous Count



Scores

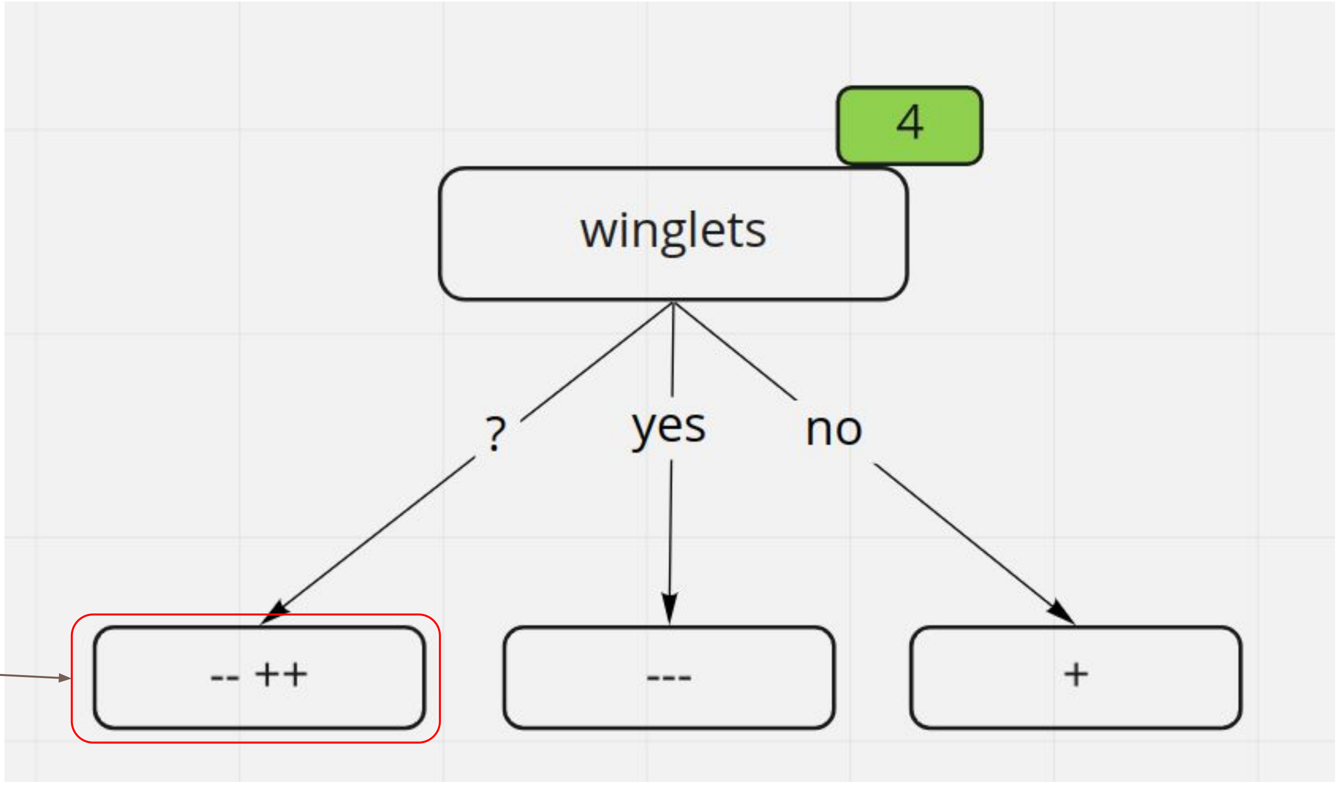
- Winglets: 4
- Narrow Body: 3
- Engine Shape: 2
- Engine Noise: 0

The **winglets** test produced the most homogeneous solution

Identity Tree 1/2

So we now start with the winglets test, eliminate **yes** and **no** rows from the table, and iterate again

non-uniform

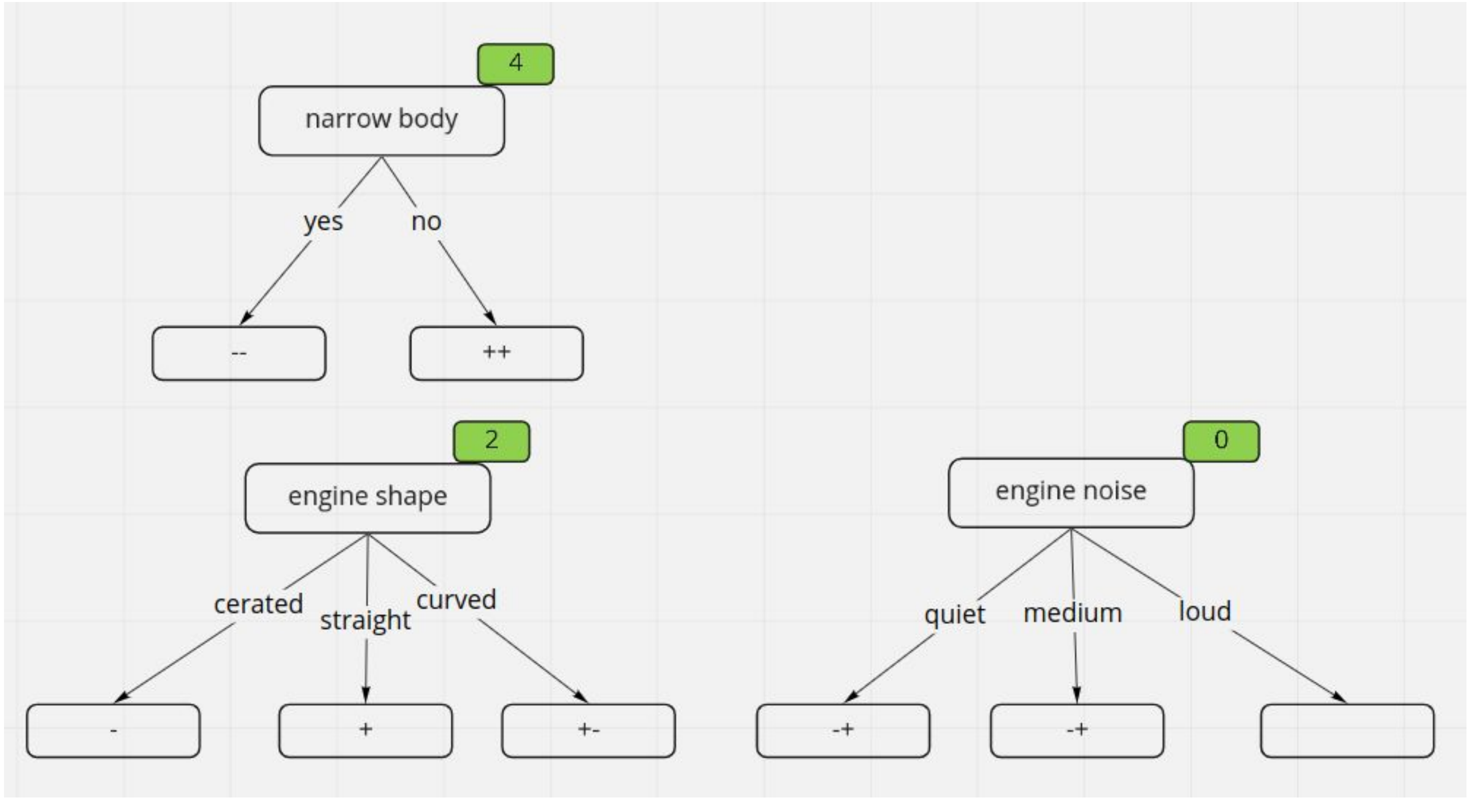


Reduce the test table

Narrow Body	Engine Shape	Winglets	Engine Noise	B787
Yes	cerated	?	quiet	No
~~Yes~~	~~straight~~	~~Yes~~	~~quiet~~	No
No	straight	?	quiet	Yes
~~No~~	~~curved~~	~~No~~	~~loud~~	Yes
No	curved	?	medium	Yes
~~No~~	~~cerated~~	~~Yes~~	~~loud~~	No
~~No~~	~~straight~~	~~Yes~~	~~loud~~	No
Yes	curved	?	medium	No

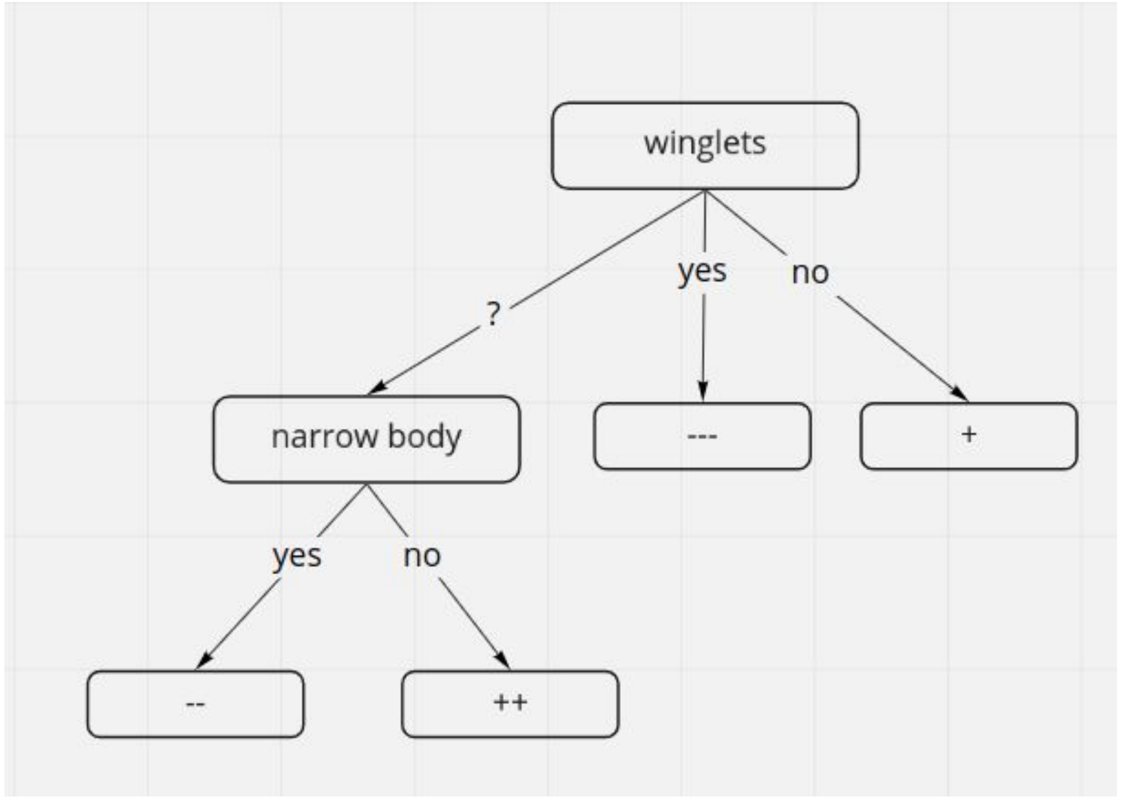
Run the 3 tests again

Select the best test, which in this case is the **narrow body** test



Identity Tree 2/2

Now we have a measure of **certainty**, which will guide us in the classification task



Conclusion: Probabilities

Winglets=**Y**:

P(3/8) it is not B787

Winglets=**N**:

P(1/8) it is B787

Winglets=**?** & Narrow Body=**Y**

P(2/8) it is not B787

Winglets=**?** & Narrow Body=**N**

P(2/8) it is B787

Disorder

Working with large data sets

Large data sets

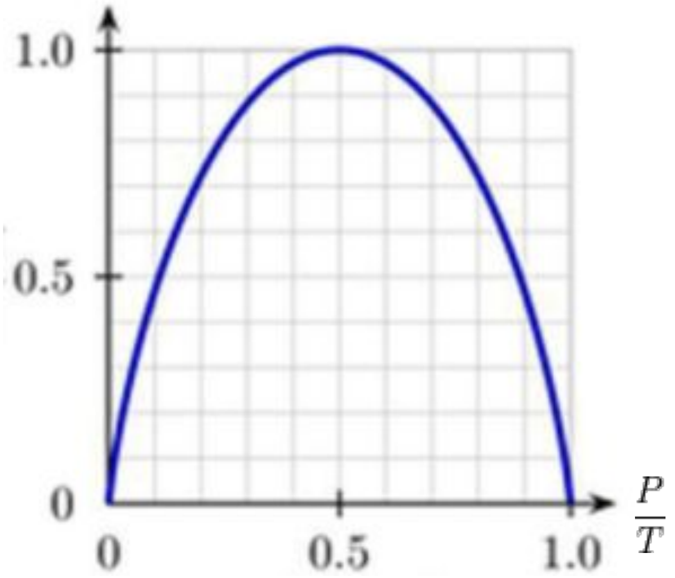
- One of the problems we face when dealing with a large data set is that of how do we start?
- How do we find a homogeneous partition in a large data set
- This is not easy to do
- We can use a measure called **disorder**

Information Theory: Set Entropy

D(S) = disortder of a set S:

$$D(S) = -\frac{P}{T}\, log_2 \frac{P}{T} - \frac{N}{T}\, log_2 \frac{N}{T}$$

P = total positives in S
N = total negatives in S
T = total in S



Finally

When we use set entropy

$$D(S) = -\frac{P}{T} log_2 \frac{P}{T} - \frac{N}{T} log_2 \frac{N}{T}$$

Weighted for the number of edges as a percentage of the total in the decision set, we find the same test partition logic as we did for the previous methods.

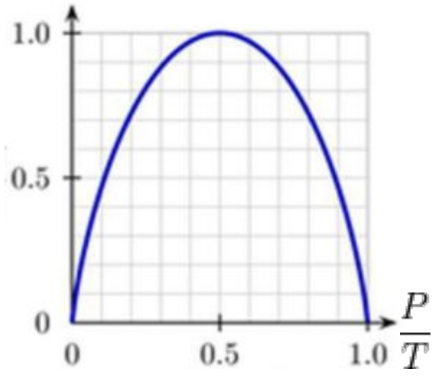
Worked example

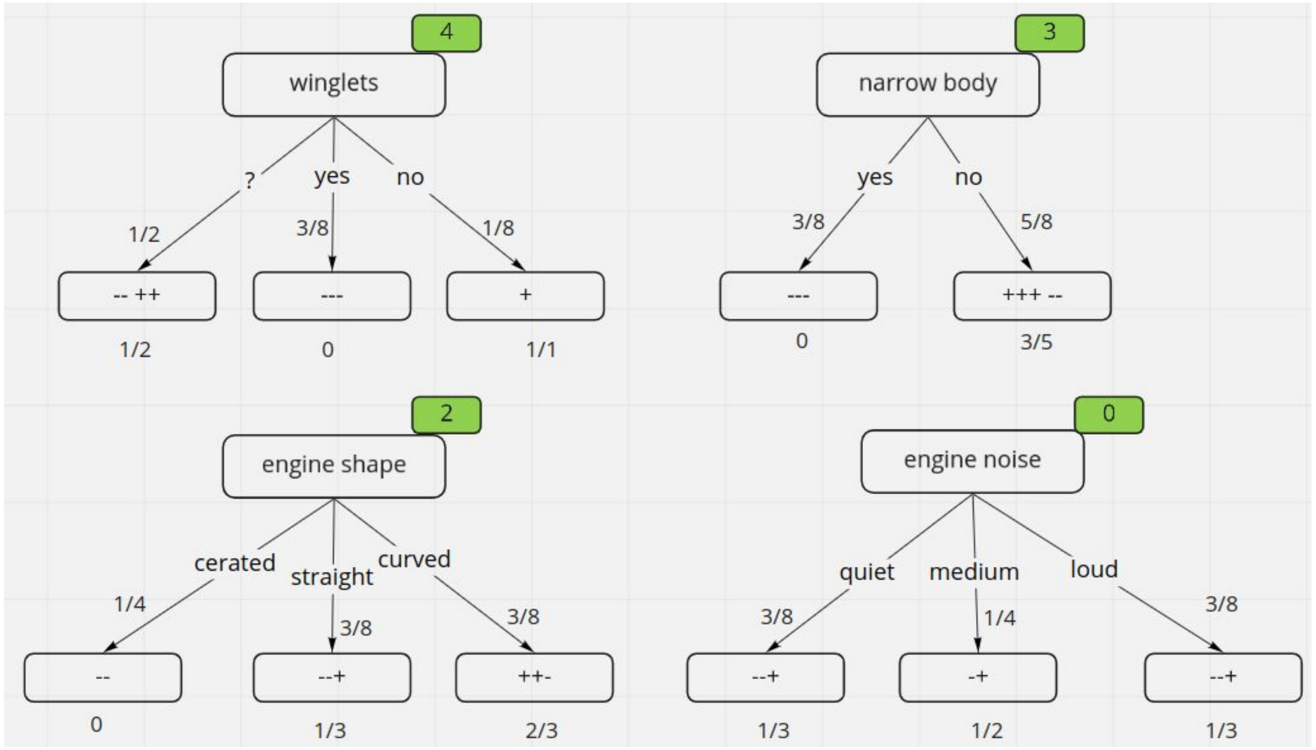
- We can now use the density function in the previous example
- to rank our tests and
- validate our decision to choose the **winglets** test
- We will use graph itself rather than the log function calculation

Worked example

Each test outcome is weighted in proportion to the number of elements in the outcome divided by the total number of elements (N)

In the example N=8





Worked example

We will complete this in class

Finally

- When using Identification Trees we need to create tests with the lowest disorder
- For this we use the entropy density function
- Weighted for the number of elements in the set divided by the set size
- This is the best method for bootstrapping Identification Trees

Thank You

Questions

CA318
Advanced Algorithms and AI Search

Theory of Neural Networks

Theory of Neural Networks

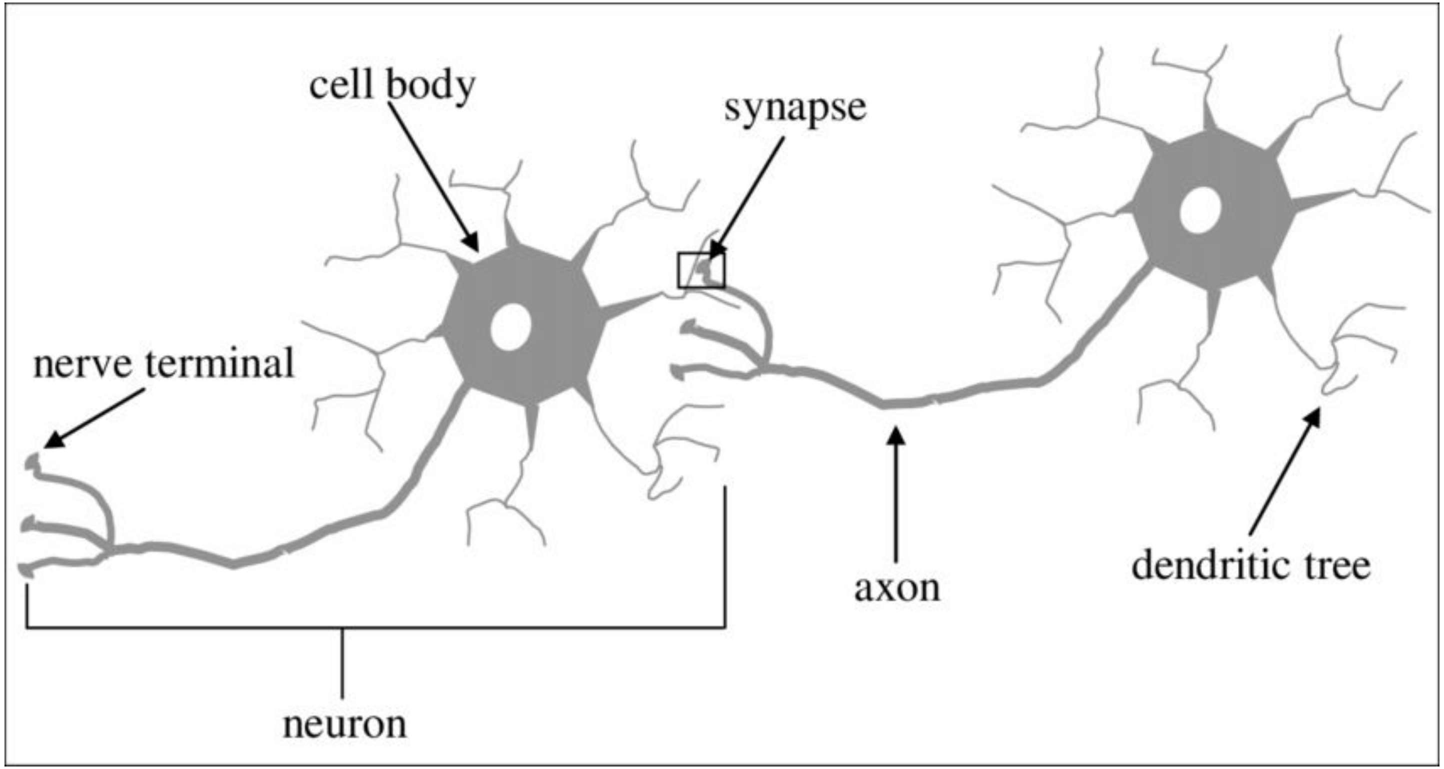
Biologically Inspired Learning

Biologically Inspired Learning

- Although we have studied many elegant methods of
  - searching
  - optimization
  - classification
- We still have not adequately explained how **learning** arises
  - The human brain learns well
  - So why not try to model this in software
- This is the goal of Neural Networks and Deep Learning

Neuron Structure

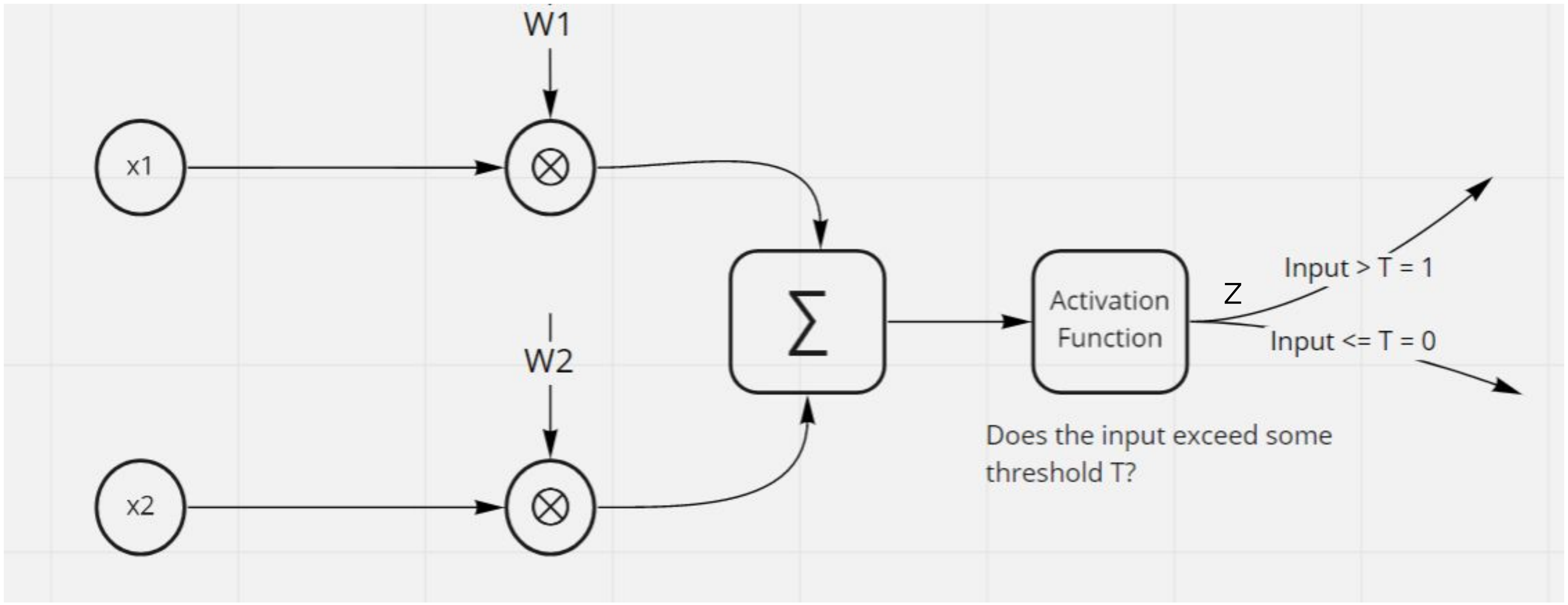
There are 86 billion neurons in the human brain



Modelling a Neuron

- Neurons firing is binary
  - Given some stimulus event
  - They either fire ⊗
  - or do not fire
- When neurons connect to each other, they do so with some level of strength
  - Synaptic Strength
- Lets consider how two neurons might be connected in our model

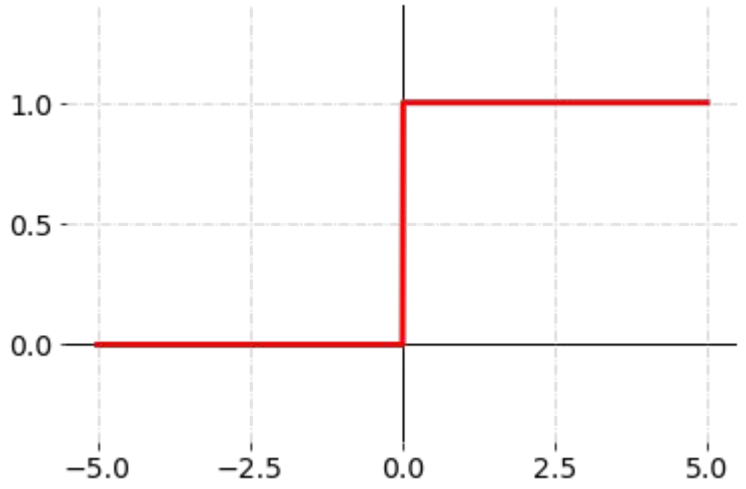
Two Neurons Connected



Activation Function

The activation function defined in this initial example is a binary activation (step) function.

This is the most simplistic approach possible, but we shall see later, it is not very desirable



Model Charateristics

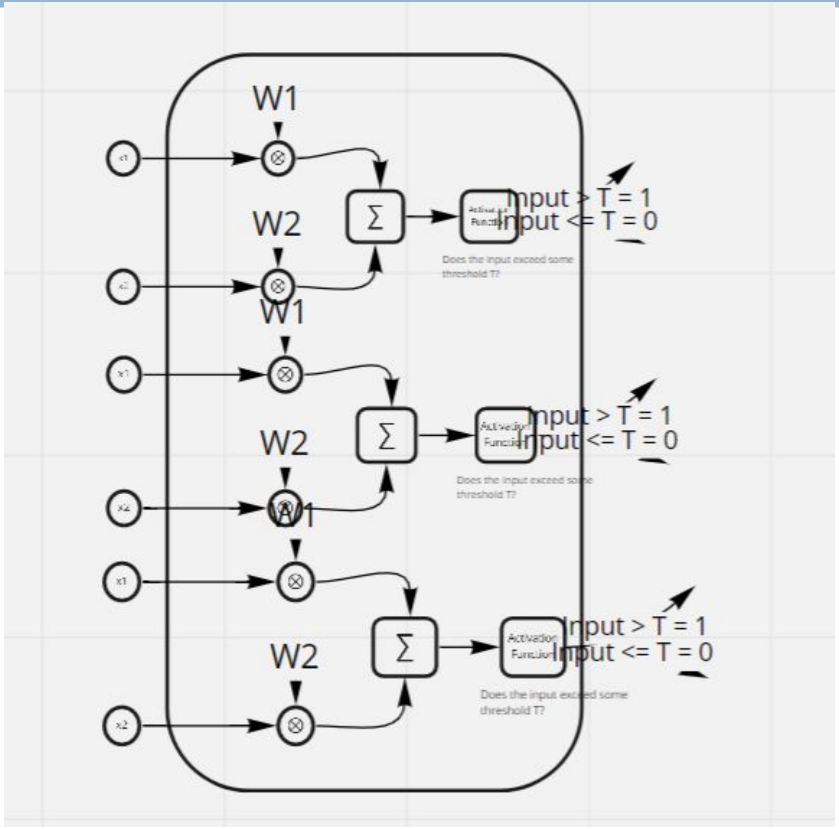
- All or none
- Cumulative Influence
- Synaptic Weight

- This is model that is inspired by the human brain
- But it is not clear that this model is the essential parameters of how a human can do what they do
  - We may be missing many other parameters including timing characteristics

Another View

A "view" of your head, which could be thought of as a very large function calculator:

$$\bar{z} = f(\bar{x}, \bar{w}, \bar{T})$$



Objectives

$$\bar{z} = f(\bar{x}, \bar{w}, \bar{T})$$

What we need to do is to adjust the vectors **w** and **T,** until the vector **z** takes the form of what we want to see.

There may be millions of elements in the vector **w**.

NN as a function approximator

$$\bar{z} = f(\bar{x}, \bar{w}, \bar{T})$$

Clearly, we can think of the neural network as being a **function approximator**.

Desired Output

- If we consider our NN as a **function approximator**
- we can compute the desired output as some kind of ideal function **g**
- **g** yields the desired output vector **d** from the input **x**:

$$\bar{d} = g(\bar{x})$$

How well are we doing?

From the simple observations on the previous slides:

- We can figure out how well our function approximator is doing, by comparing the actual value vector (**z**) with the desired value vector (**d**) in a special function
    - This is called a **performance function**

Performance function ***P***

$$P : \bar{d} \; \bar{z}$$

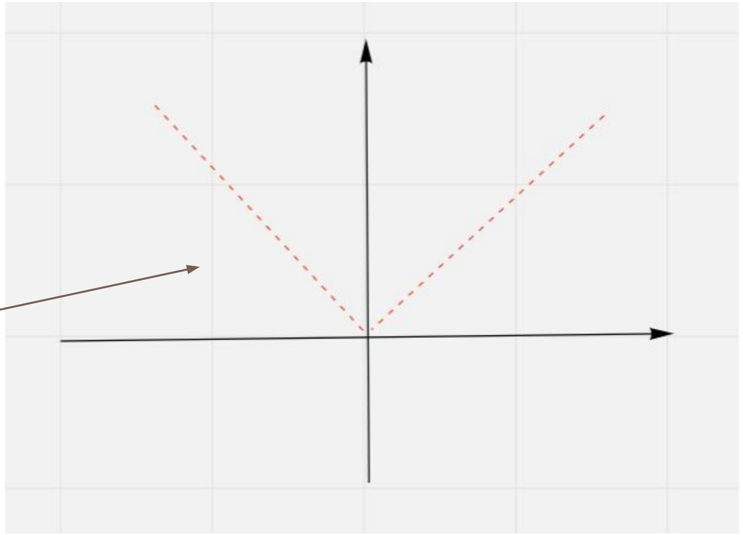
So clearly some performance function (**P**) exists, that does some kind of operation or measurement on the vectors **d** and **z** and reports a quantitative measurement of performance

Performance function ***P***

One simple implementation of P is to simply measure the magnitude of the difference thus:

$$P : ||\bar{d} - \bar{z}||$$

In turn, **P** yields a performance function thus

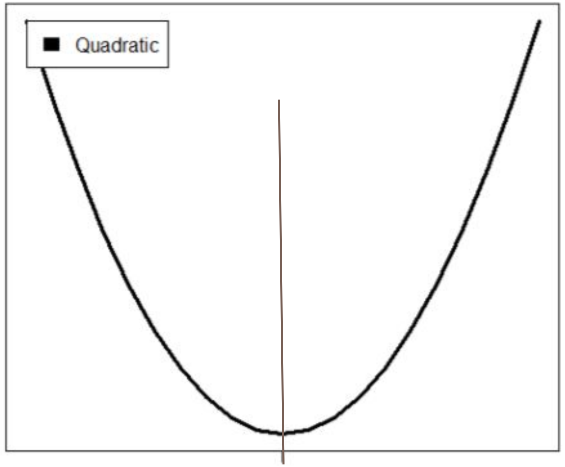


Performance function ***P***

However, this kind of function is awkward to deal with mathematically, so we square the magnitude of the difference:

$$P : ||\bar{d} - \bar{z}||^2$$

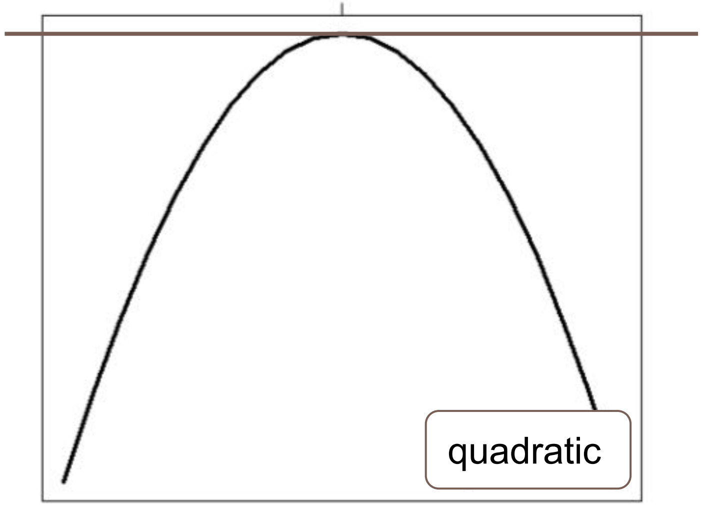
In turn, **P** yields a performance function thus



Refinement of ***P***

We can now slightly refine **P** to express the idea that **P** returns a negative quantity which gets closer to zero the better input and output matches:

$$P : -||\bar{d} - \bar{z}||^2$$



Optimization (again)

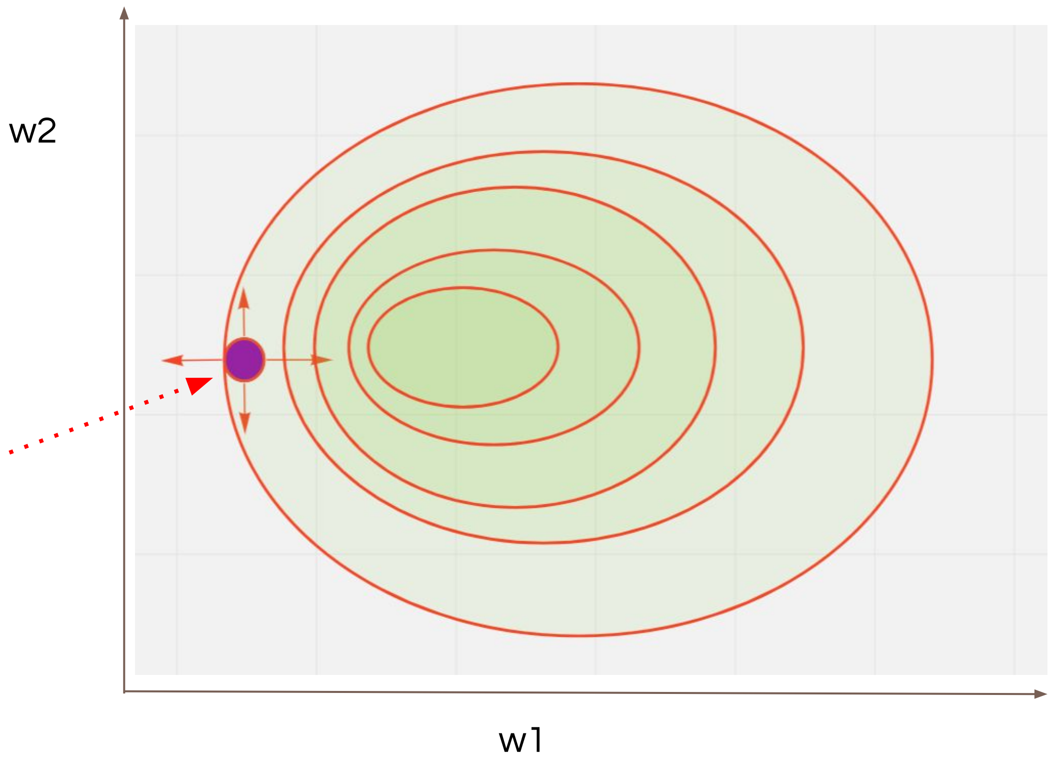
We can now see that the objective is to find the minimum value for the performance function **P**

So we seek to satisfy this **constraint** to find the best vector **w**:

$$P : min(- ||\bar{d} - \bar{z}||^2)$$

Contour Map

So we can see a way to improve the weights…by implementing a Hill Climbing **heuristic**, from the starting point shown here



Simplification

A problem with the

$$\bar{z} = f(\bar{x}, \bar{w}, \bar{T})$$

is that threshold measures are not easy to deal with. So we will replace the threshold value **T** with an equivalent weight **w_T** shown here

$$\bar{z} = f(\bar{x}, \bar{w})$$

Intractability

Unfortunately, another problem arises when we have several million weights, we cannot adjust each of them simultaneously as this quickly becomes intractable (a search in n-dimensional space)

Instead, we use partial derivates to tell us which of the weights as the most influence, and then we modify that weight only.

Partial Derivates - Hill Climbing

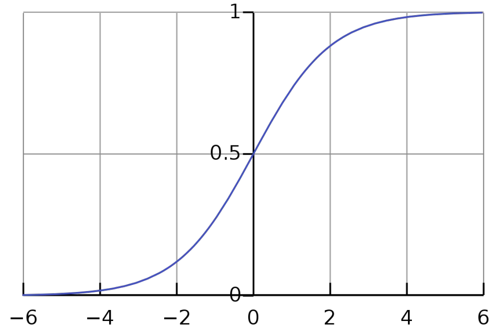
$$\Delta\bar{w} = r\left(\frac{\partial P}{\partial w_1}i + \frac{\partial P}{\partial w_2}j\right)$$

Here, the change in **w** is dependent on some rate constant **r** into the partial derivates shown here

Sigmoid Activation Functions

- Finally, we need to replace the binary activation function with a smoother (sigmoid) activation function.
- This is because it is mathematically inconvenient to perform Hill Climbing when our functions contains discontinuities.
- Sigmoid activation functions are generally of the form:

$$S(\alpha) = \frac{1}{1 + e^{-\alpha}}$$



Deep Learning

An Introduction

Modelling the Brain

- Deep learning has its origins in early work that tried to model networks of neurons in the brain (1943) with computational circuits.
- For this reason, the networks trained by deep learning methods are often called **neural networks**
    - even though the resemblance to real neural cells and structures is superficial

Linear Regression

- Although methods such as linear and logistic regression can handle a large number of input variables,
    - the computation path from each input to the output is very short: multiplication by a single weight, then adding into the aggregate output.
- Moreover, the different input variables contribute independently to the output, without interacting with each other

Linear Regression

They can represent only linear functions and boundaries in
the input space, whereas most real-world concepts are far more complex.

Long Paths for Complex Problems

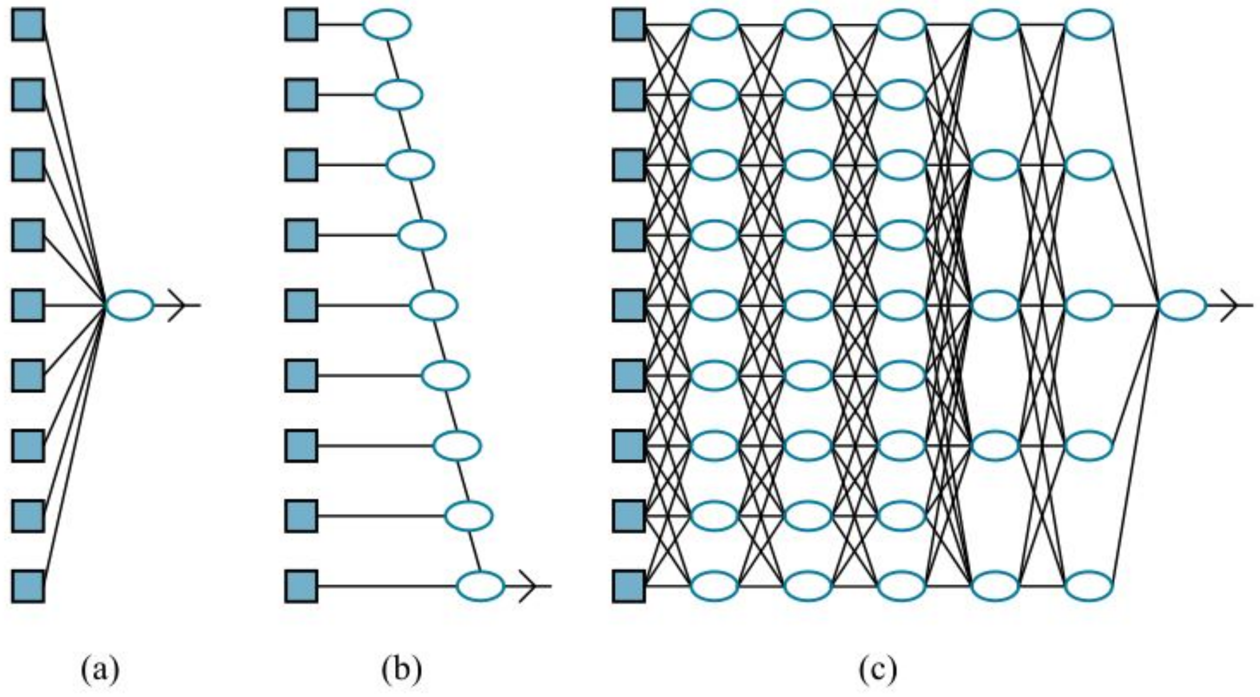
- The basic idea of deep learning is to train circuits such that the computation paths are long, allowing all the input variables to interact in complex ways
    - See Fig (c).
- These circuit models turn out to be sufficiently expressive to capture the complexity of real-world data for many important kinds of learning problems.

Network Depth

(a) A shallow model, such as linear regression, has short computation paths

between inputs and output. (b) A decision list network has some long paths for some possible input values, but most paths are short.

(c) A deep learning network has longer computation paths, allowing each variable to interact with all the other

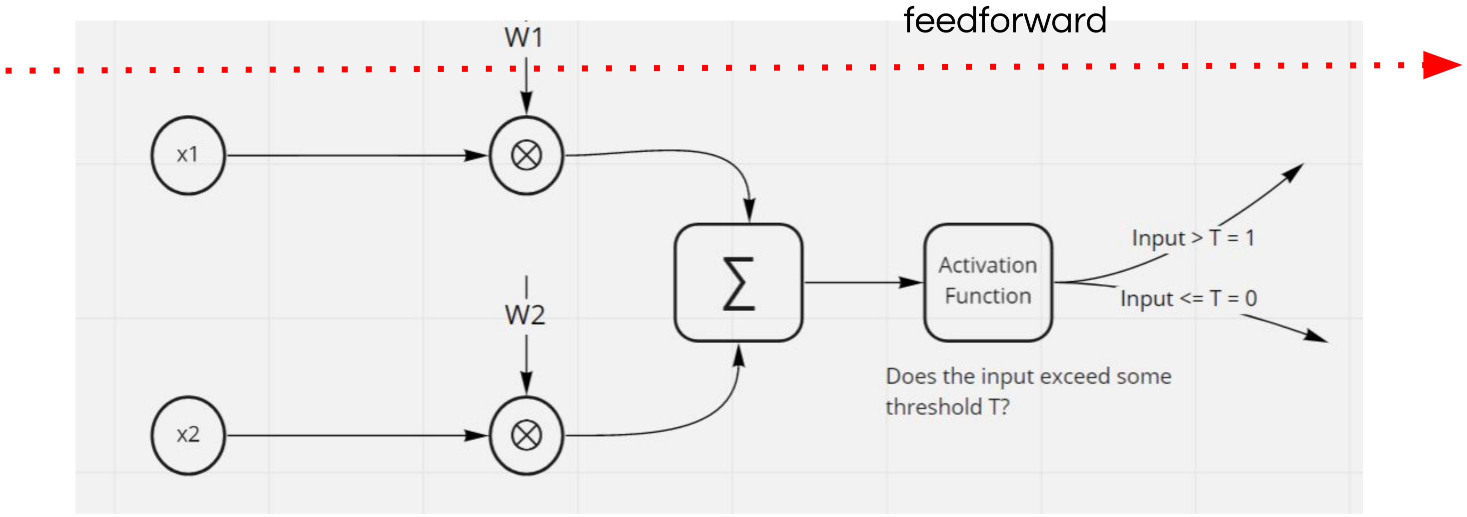


Simple Feedforward Networks

Connections only in one direction

Feedforward Network

In the previous section we connected our trival neural network in such a way that the data flowed from left to right, otherwise known as a Feedforward Network



Feedforward Network

- Each node computes a function of its inputs
    - then passes the result to its successors in the network.
- Information flows through the network from the input nodes to the output nodes
    - there are no loops

Recurrent networks

- A recurrent network, on the other hand, feeds its intermediate or final outputs back into its own inputs.
- This means that the values within the network form a dynamical system
  - that has internal state or memory.
- We will consider these in the next lecture

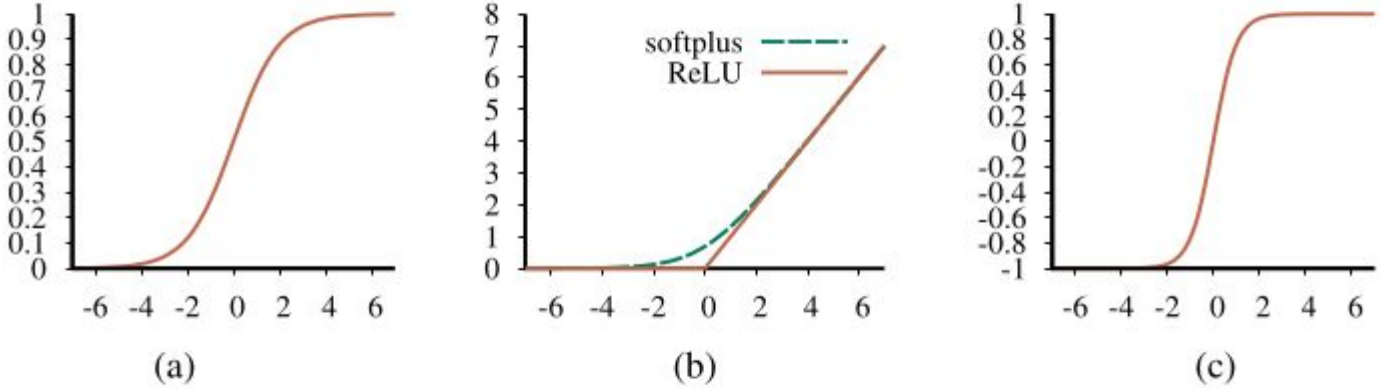
More Activation Functions

In the previous section we discussed sigmoid based activation functions:

$$S(\alpha) = \frac{1}{1 + e^{-\alpha}}$$

However, sigmoid activation functions are not the only kind. Typically we see two others: **tanh** and **softplus/ReLU**

More Activation Functions

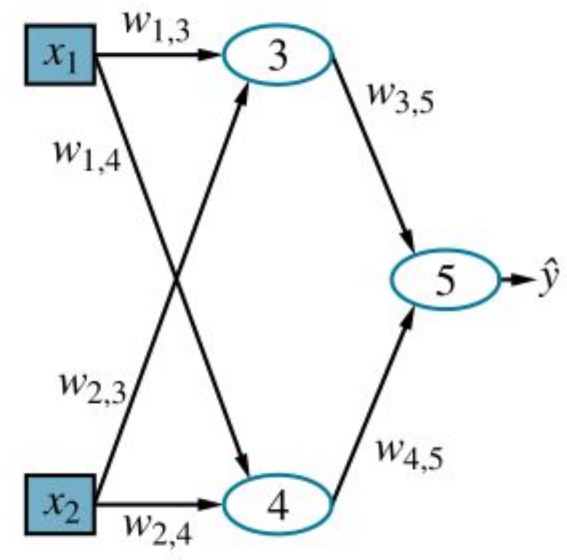


Activation functions commonly used in deep learning systems: (a) the logistic or sigmoid function; (b) the ReLU function and the softplus function; (c) the tanh function.

Activation Functions

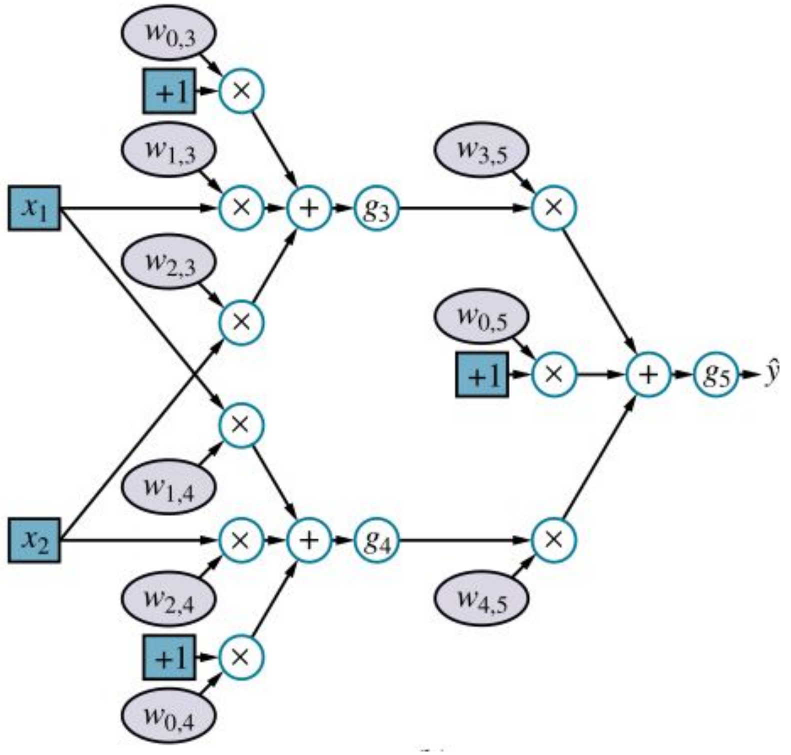
Sigmoid	$S(\alpha) = \dfrac{1}{1+e^{-\alpha}}$
ReLU	$\mathrm{ReLU}(x) = \max(0, x)\,.$
Softplus	$\mathrm{softplus}(x) = \log(1+e^{x})\,.$
tanh	$\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}\,.$

Hidden Layers



A neural network with two inputs, one hidden layer of two units, and one output unit.

Computational Expansion



The network is unpacked into its full computation graph.

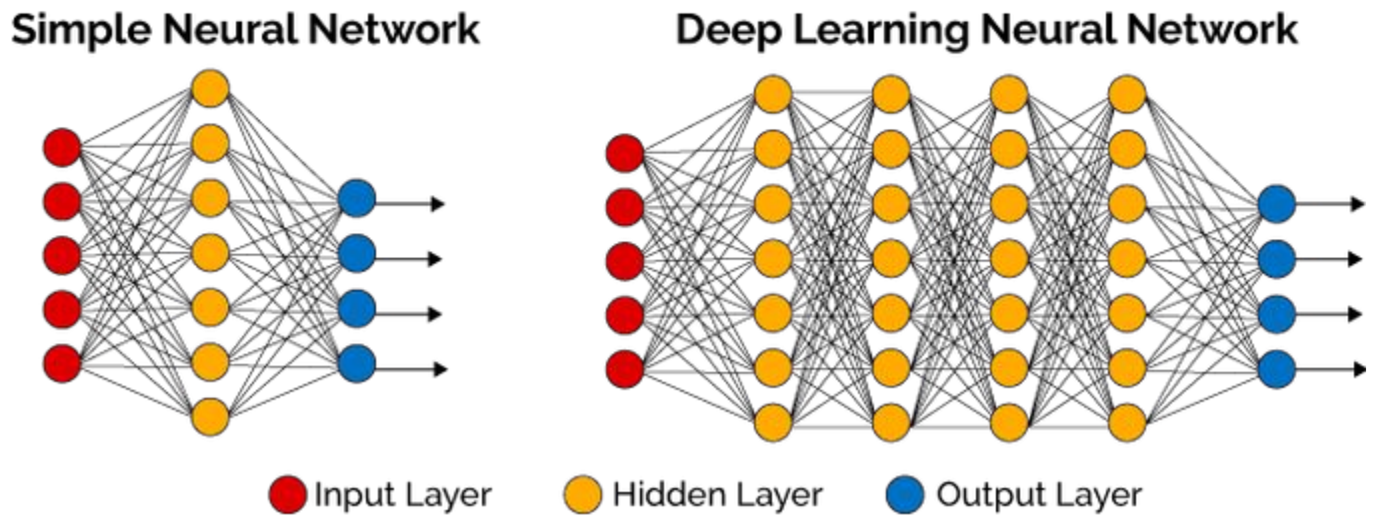
Hidden layers are those not input or output layers.

In this model there is **one** hidden layer.

Hidden Layers

- Hidden layers allow for the function of a neural network to be broken down into specific transformations of the data.
- Each hidden layer function is specialized to produce a defined output.
- For example, a hidden layer functions that are used to identify human eyes and ears may be used in conjunction by subsequent layers to identify faces in images.
- While the functions to identify eyes alone are not enough to independently recognize objects, they can function jointly within a neural network.

Hidden Layers



Ref: https://deepai.org/machine-learning-glossary-and-terms/hidden-layer-machine-learning

Function Approximator

Coupling multiple units together into a network creates a complex function that is a composition of the algebraic expressions represented by the individual units.

For example, the network shown here represents a function h_w (**x**), parameterized by weights **w**, that maps a two-element input vector x to a scalar output value **ŷ**.

$$\begin{aligned}\hat{y} &= g_5(in_5) \\ &= g_5(w_{0,5} + w_{3,5}g_3(in_3) + w_{4,5}g_4(in_4)) \\ &= g_5(w_{0,5} + w_{3,5}g_3(w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2) + w_{4,5}g_4(w_{0,4} + w_{1,4}x_1 + w_{2,4}x_2))\end{aligned}$$

Computational Graph

- The computational graph makes each element of the overall computation explicit.
- It also distinguishes between the inputs (in blue) and the weights (in light mauve):
  - the weights can be adjusted to make the output ŷ agree more closely with the true value y in the training data.
- Each weight is like a volume control knob that determines how much the next node in the graph hears from that particular predecessor in the graph

Graph Complexity

- The computation graph here is relatively small and shallow,
- but the same idea applies to all forms of deep learning:
  - we construct computation graphs and adjust their weights to fit the data.
- The graph above is also fully connected,
  - meaning that every node in each layer is connected to every node in the next layer.
  - Except for nodes leading into the output layer

Graph Connectedness

- The default connection (or wiring) of the graph is **connected**
    - every node in each layer is connected to every node in the next layer.
- This is in some sense the default,
- Choosing the connectivity of the network is also important in achieving effective learning

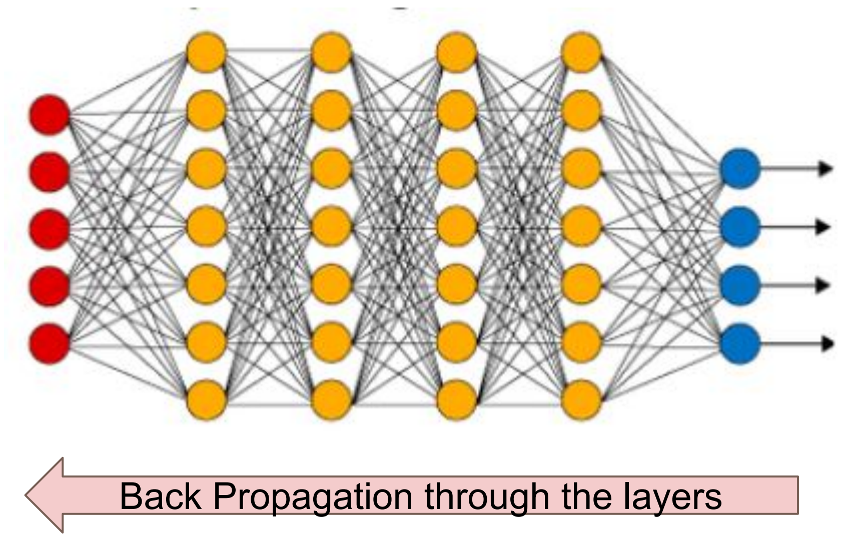
Gradients and learning

- For the weights leading into units in the output layer the ones that produce the output of the network,
  - the gradient calculation is essentially identical to the process the partial differential model seen earlier
- However, units in the hidden layers, which are not directly connected to the outputs, calculating the decent gradient is more complicated.

Backpropagation

Clearly, the error function is computed first at the output layer.

In order to improve the output layer the error function for the hidden layers are computed in reverse order from the output back to the first hidden layer - this is called **back propagation**



Backpropagation

- It is important to remember that Back Propagation is a weight adjustment process.
- It does not mean connecting the output of nodes in later layers with nodes in earlier layers.
- This type of network is called a Recurrent Neural Network and will be the subject of our next lecture
- BP works by iterating backward from the last layer to
    - avoid redundant calculations of intermediate terms in the chain rule
- BP is an example of **dynamic programming**

Backpropagation

At the last layer L, we define a Cost Function as follows, where a$^{(L)}$ is the activation function of the last layer

$$C_0 = (a^{(L)} - y)^2$$

Note though that the input to a$^{(L)}$ can be represented as z$^{(L)}$

Thus a$^{(L)}$ is really this, where **σ** is ReLU for example

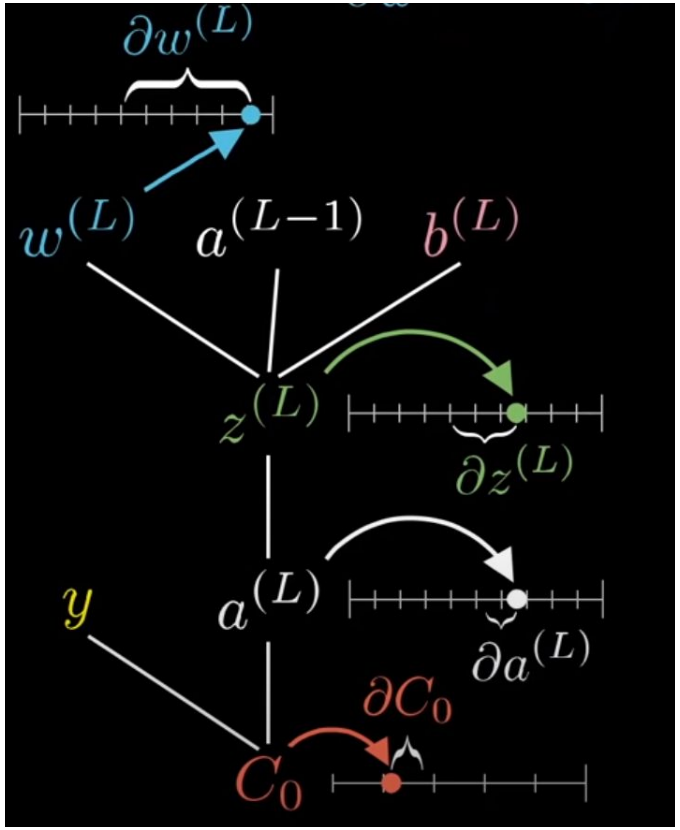
$$a^{(L)} = \sigma(z^{(L)})$$

Backpropagation

If the weights at L are written as w$^{(L)}$, we can think of z$^{(L)}$ as

$$z^{(L)} = w^{(L)} a^{(L-1)}$$

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C_0}{\partial a^{(L)}}$$



Backpropagation

If the weights at L are written as $w^{(L)}$, we can think of $z^{(L)}$ as

$$z^{(L)} = w^{(L)} a^{(L-1)}$$

And the total cost function is written now as a partial derivative using the chain rule

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C_0}{\partial a^{(L)}}$$

Thank You

Any Questions?