

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 $\mathbf{S} \rightarrow \{\mathbf{A}, \mathbf{B}\}$
 - When we come to expand $\mathbf{A} \rightarrow \{\dots\}$ we omit \mathbf{S}
 - Obviously, \mathbf{S} is reachable from \mathbf{A} , but adding it into the expansion for \mathbf{A} causes a loop in our tree**
- We order our nodes lexicographically
 - Prefer to write $\{\mathbf{A}, \mathbf{B}, \mathbf{C}\}$ rather than $\{\mathbf{C}, \mathbf{A}, \mathbf{B}\}$ 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

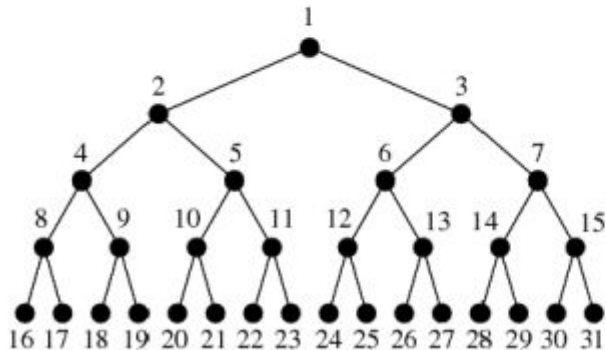
- Generally speaking, we don't need to worry about **how** a search tree / graph is constructed
 - We just assume, for now, that it gets constructed
- Tree/graph construction, and the data structures involved, tend to obfuscate important details
- So, let's assume we have the search tree in memory already.

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.

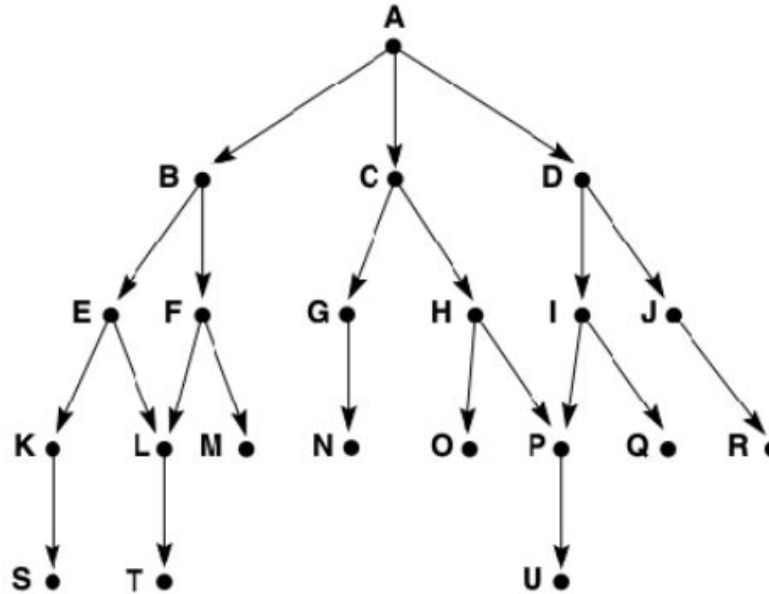
Number of leaf nodes: b^d

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

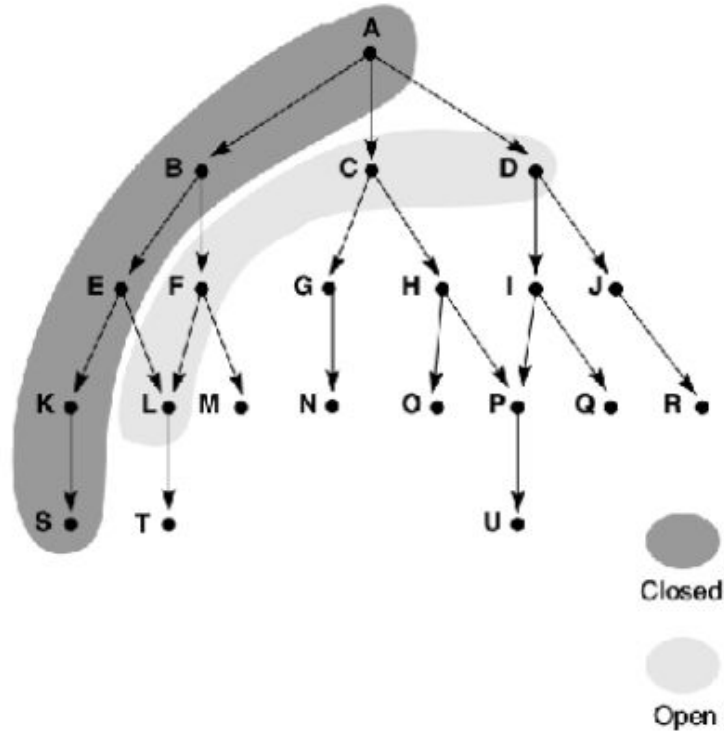


Search Objective

- 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.



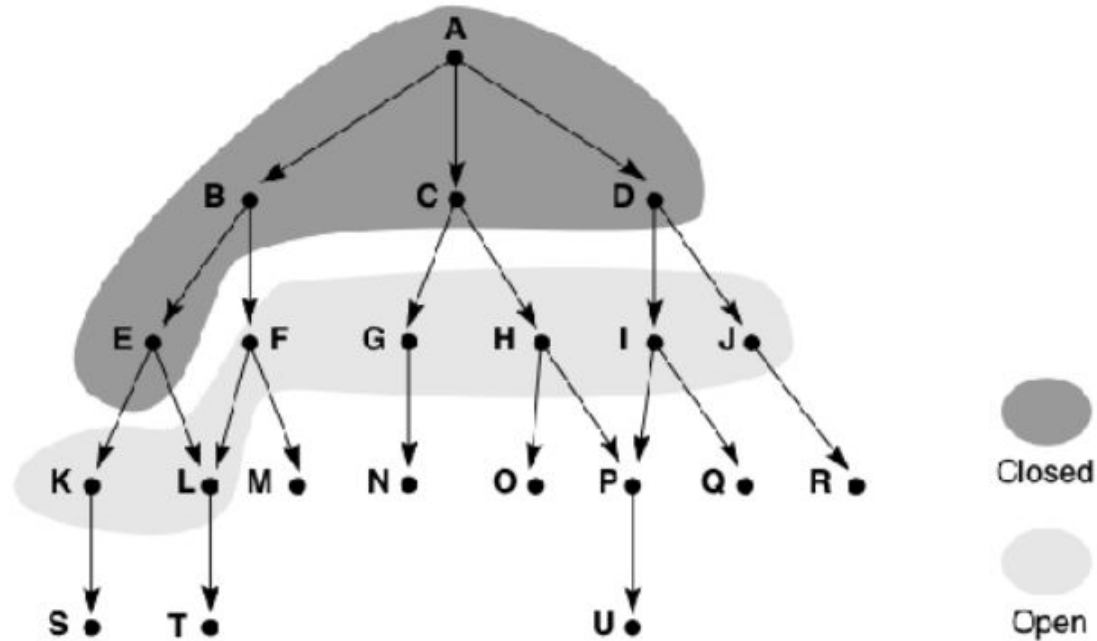
Breadth First Search

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]
H [O, P]
```

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

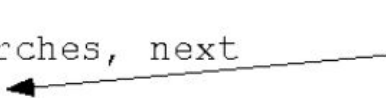
Performance of DFS

- **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 **$b \times n$** 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

Iterative 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,
 - \Rightarrow states have a high average number of children,
 - combinatorial explosion may prevent the algorithm from finding a solution using available memory.

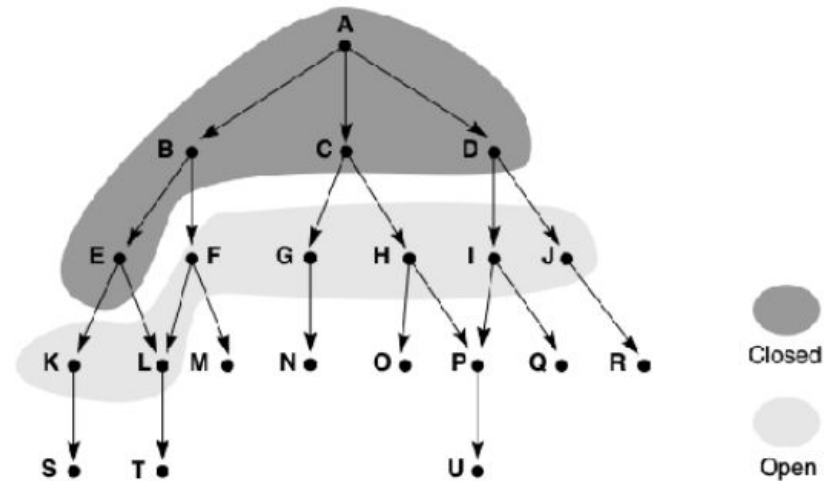
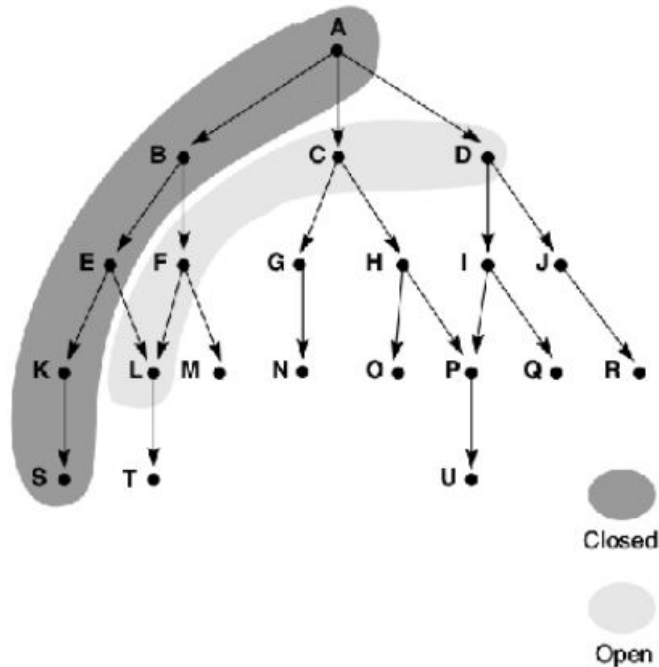
Performance of BFS

- 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

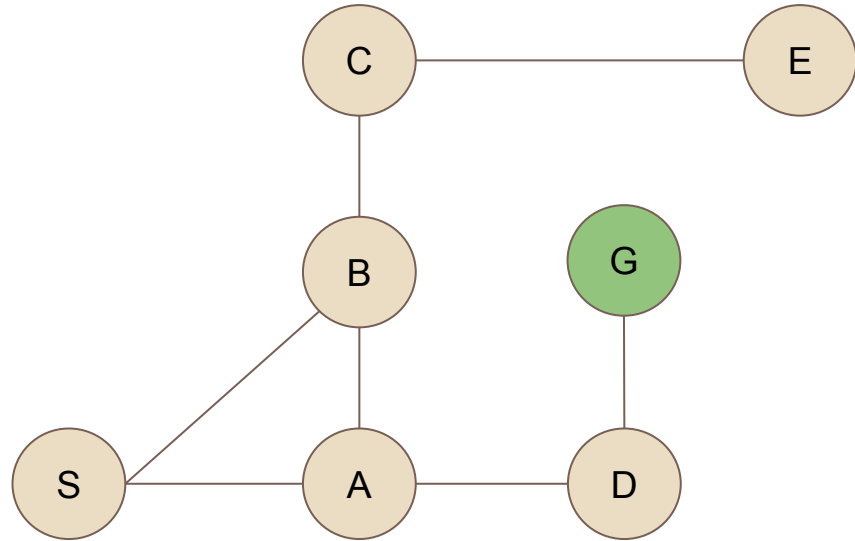
- Breadth first search always finds the shortest path to the goal.
 - Methodical
 - But ... needs to have a large todo list
 - Too large for large trees (too much memory)
- Depth first search doesn't need to store very much, but can easily miss a nearby path.

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.

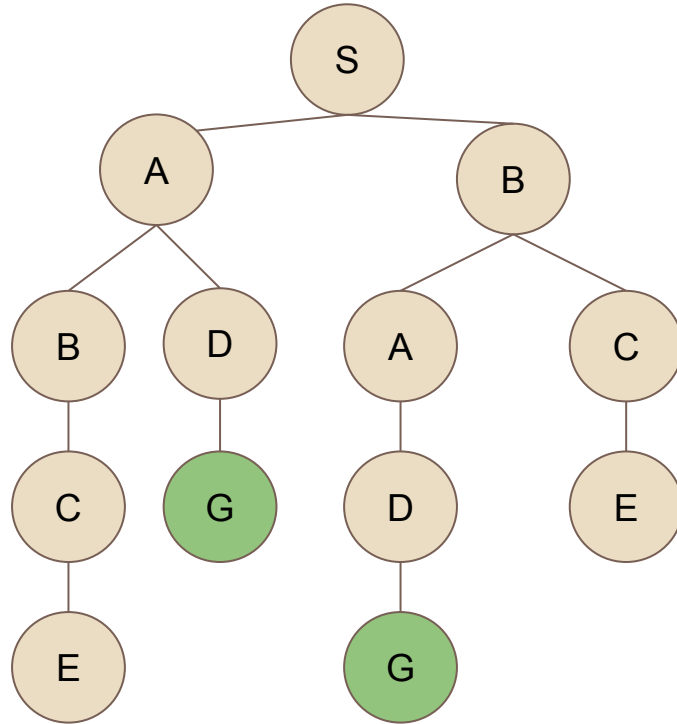
Aside



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?