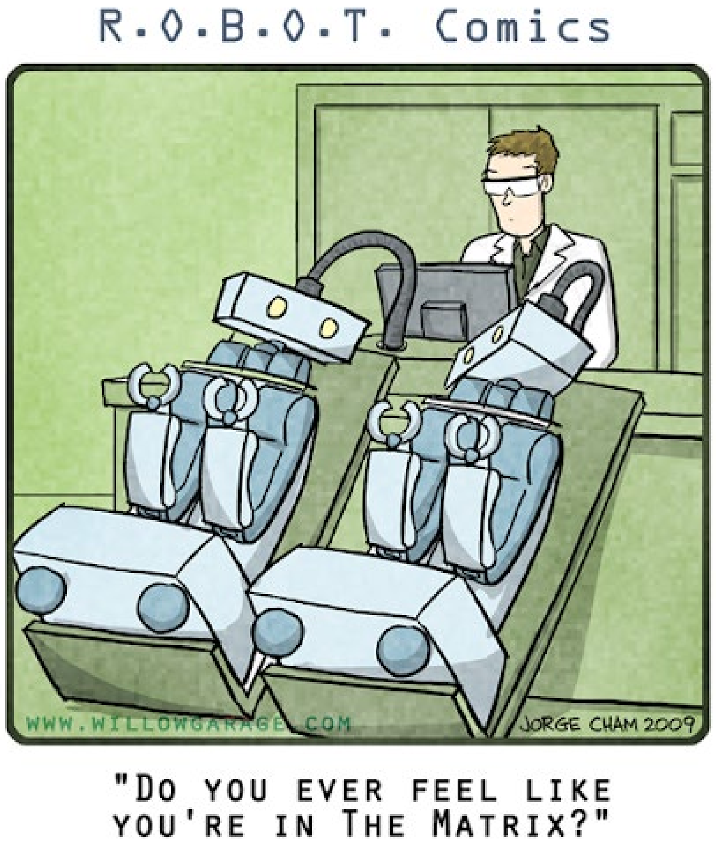


CS 4649/7649 Robot Intelligence: Planning

Foundations II: Complexity of State Space Search

Asst. Prof. Matthew Gombolay

Slides adapted from:
6.034 Tomas Lozano Perez
16.410 Brian Williams
Russell and Norvig AIMA

24 AUG 2020

Assignments

- Due Today, 8/24
  - Read Ch. 4 in Russel & Norvig
- Due Wednesday, 8/26
  - Read Ch. 6 in Russel & Norvig
  - Pset1 due at 1:59 PM Eastern

Outline:

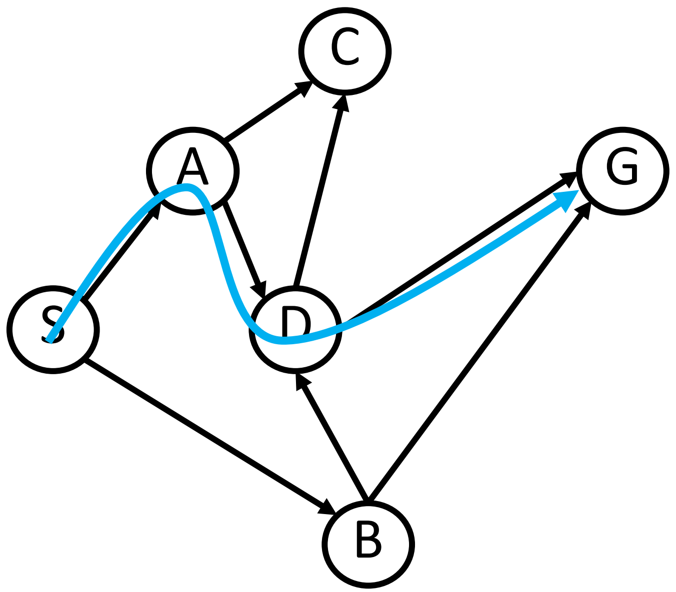
<u>Complexity of Statespace Search</u>

- Review
- Analysis
  - Depth-first search
  - Breadth-first search
- Iterative deepening

Formalizing Graph Search

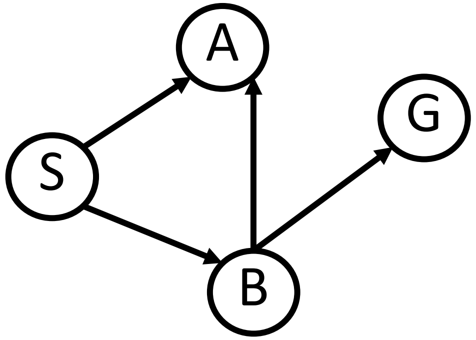
Input: A search problem P $= \langle G, v_s, v_g \rangle$ where

- Graph, $G = \langle V, E \rangle$
  - V is set of Vertices
  - E is set of Edges
- Start vertex $\mathrm{v_s} \in V$
- Goal vertex $\mathrm{v_g} \in V$

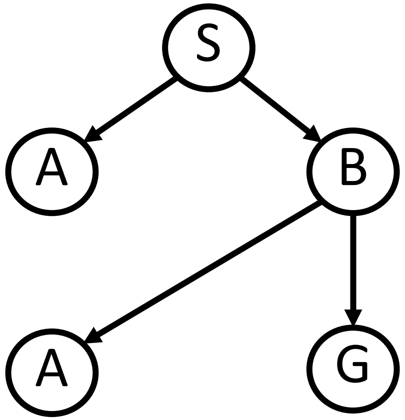


Output: A simple path, e.g. $P = \langle v_s, v_A, \ldots, v_g \rangle$, in G from v_s to v_g
(i.e., $\langle v_i, v_{i+1} \rangle \in E$ and $v_i \neq v_j$ if $i \neq j$)

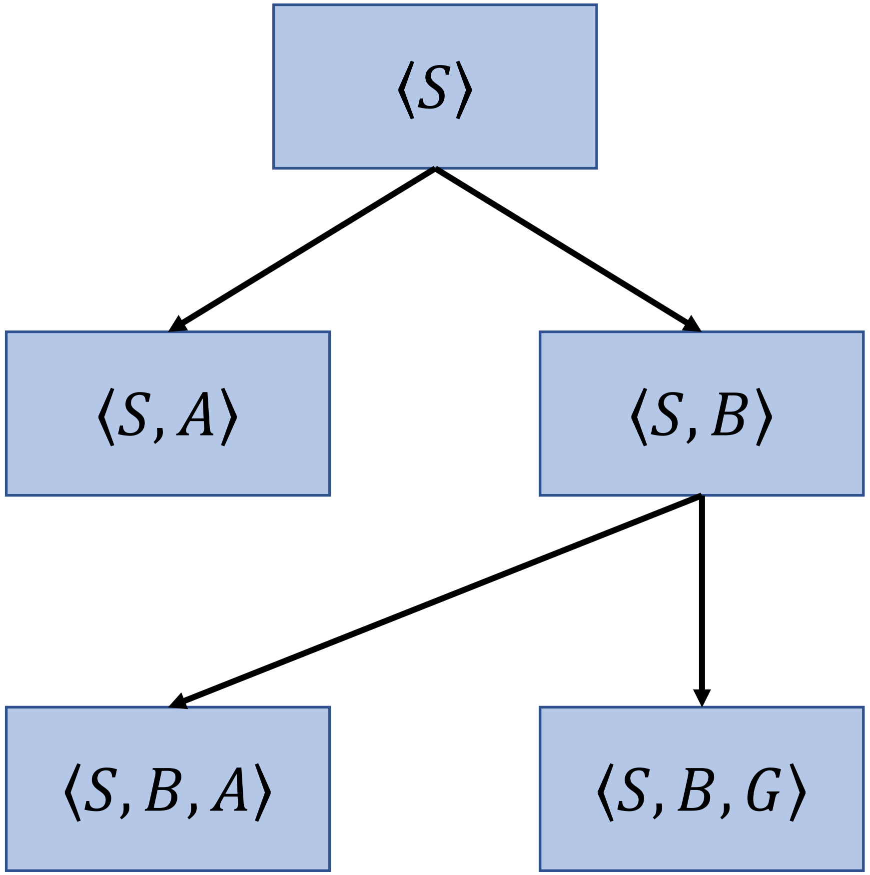
Formalizing Graph Search



Graph search is a kind of tree search



Graph search is a kind of state space search



Simple Search Algorithm

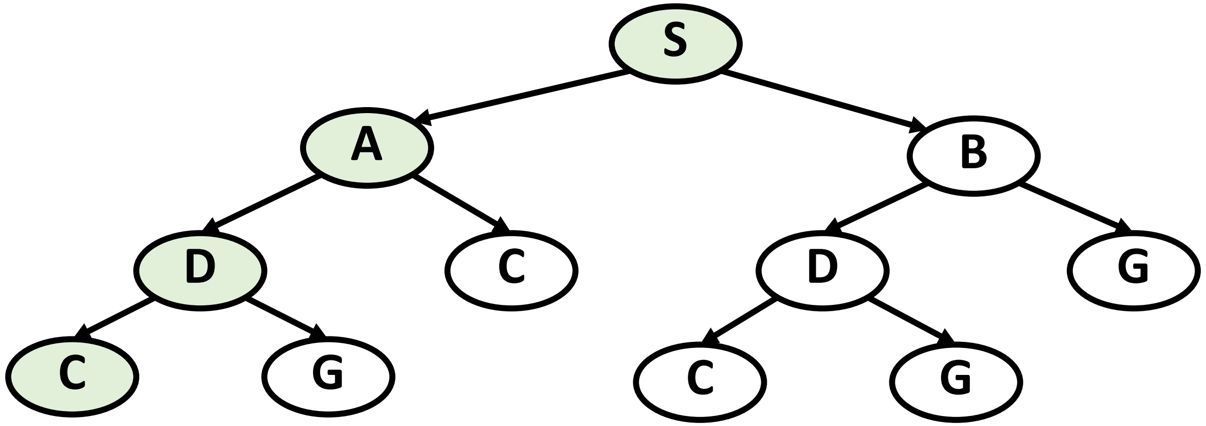
Let Q be a list of partial paths,

S be the start node and

G be the goal node.

1. Initialize Q with a partial path <S>; set Visited = {}
2. If Q is empty, fail. Else, pick a partial path N from Q
3. If head(N) = G, return N //Goal reach!
4. Else:
   a) Remove N from Q
   b) Find all children of head(N) not in Visited and create a one-step extension of N to each child
   c) Add all extended paths to Q
   d) Add Children of head(N) to Visited
   e) GOTO step 2

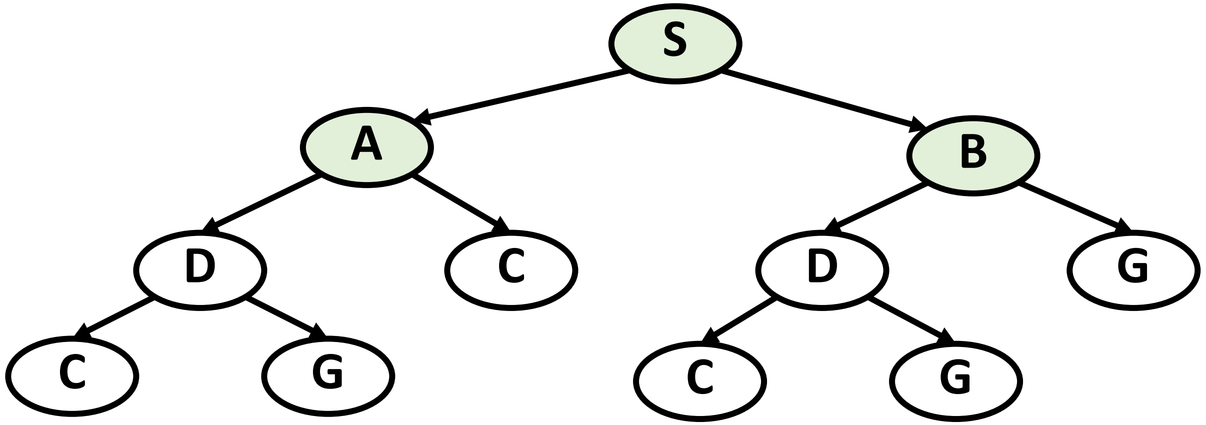
Solution: Depth First Search (DFS)



Depth-first:

- Add path extension to front of Q
- Pick first element of Q

Solution: Breadth First Search (BFS)



Breadth-first:

- Add path extension to back of Q
- Pick first element of Q

Outline:

Complexity of Statespace Search

- Review
- Analysis
  - Depth-first search
  - Breadth-first search
- Iterative deepening

Elements of Algorithmic Design

Description: (Last Wednesday)

- Problem statement
- Stylized pseudo code, sufficient to analyze and implement the algorithm

Analysis:

- Performance:
  - Time complexity
    - How long does it take to find a solution?
  - Space complexity:
    - How much memory does it need to perform search?
- Correctness:
  - Soundness:
    - When a solution is returned, is it guaranteed to be correct?
  - Completeness:
    - Is the algorithm guaranteed to find a solution when there is one?

Performance Analysis

Analysis of run-time and resource usage:

- Helps to understand scalability
- Draws line between feasible and impossible

    - A function of program input
    - Parameterized by input size
    - Seeks upper (lower, average) bound

Types of Analyses

Worst-case:

- T(n) = maximum time of algorithm of any input of size n

Average-case:

- T(n) = expected time of algorithm over all inputs of size n
  - Typically requires statistical distribution of inputs

Best-case:

- T(n) = minimum time of algorithm on any input

Analysis uses Machine-independent Time & Space

Performance depends on computer speed:

- Relative speed (run on same machines)
- Absolute speed (on different machines)

Big Idea:

- Ignore machine-dependent constraints
- Look at growth of T(n) as $n \to \infty$

"Asymptotic Analysis"

Asymptotic Notation

O-notation (upper bounds):

- $2n^2 = O(n^3)$ means $2n^2 \leq cn^3$ for sufficiently large *c* and *n*
- $f(n) = O\big(g(n)\big)$ if there exists constants $c > 0, n_o > 0$ *such that* $0 \leq f(n) \leq c * g(n)$ for all $n \geq n_o$

Set Definition of O-notation

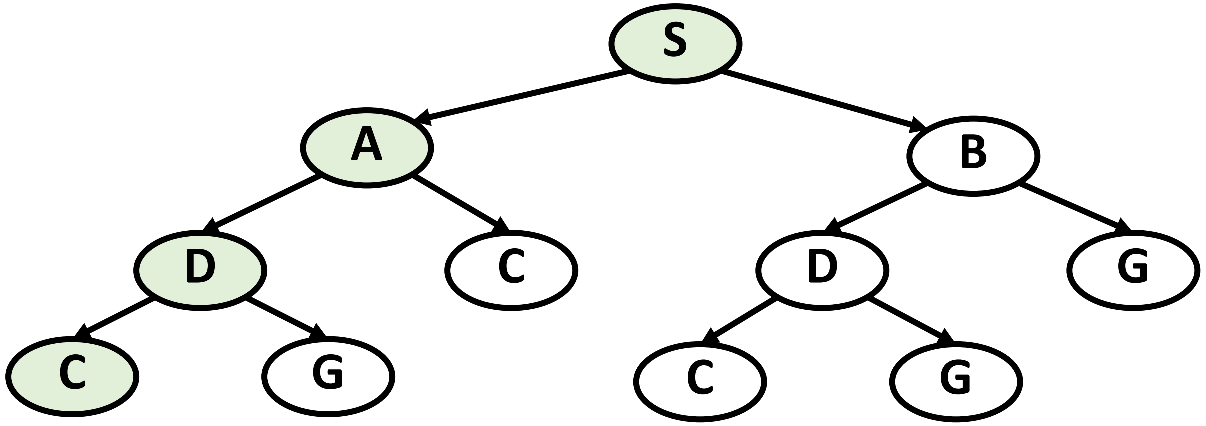
$$O(n^3) = \{\text{all functions bounded by } cn^3\}$$

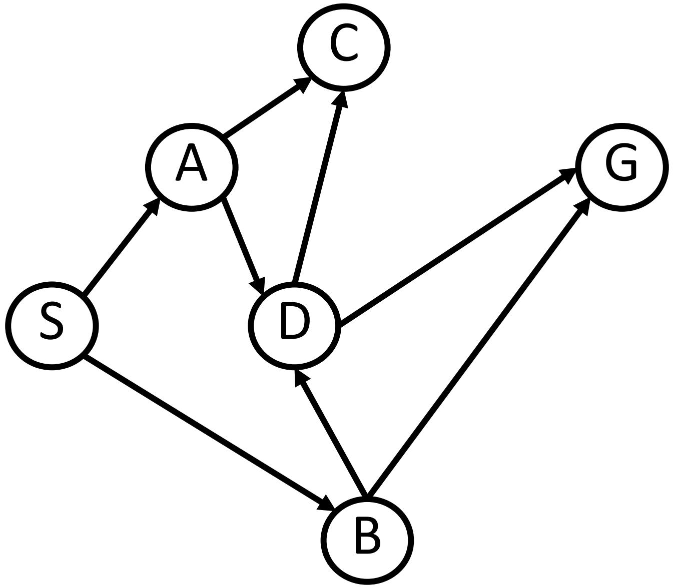
$$2n^2 \in O(n^3)$$

$$O\big(g(n)\big) = \{f(n) | \text{there exists constants } c > 0, n_o > 0 \text{ such that } 0 \leq f(n) \leq c * g(n) \text{ for all n} \geq n_o\}$$

Performance and Resource Usage

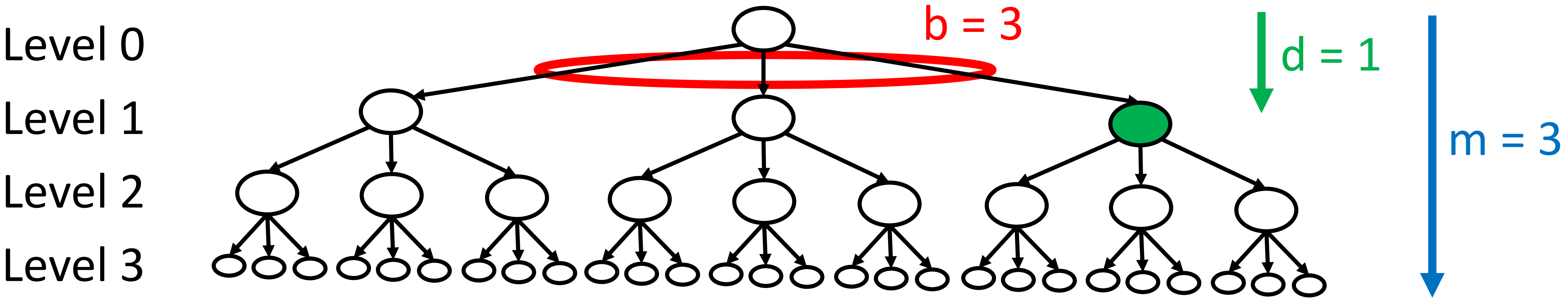
Which is better, DFS or BFS?





Search Method	Worst Time	Worst Space	Shortest Path?	Guaranteed to find path?
DFS				
BFS				

Analyzing Time and Space Complexity of Search in Terms of Trees

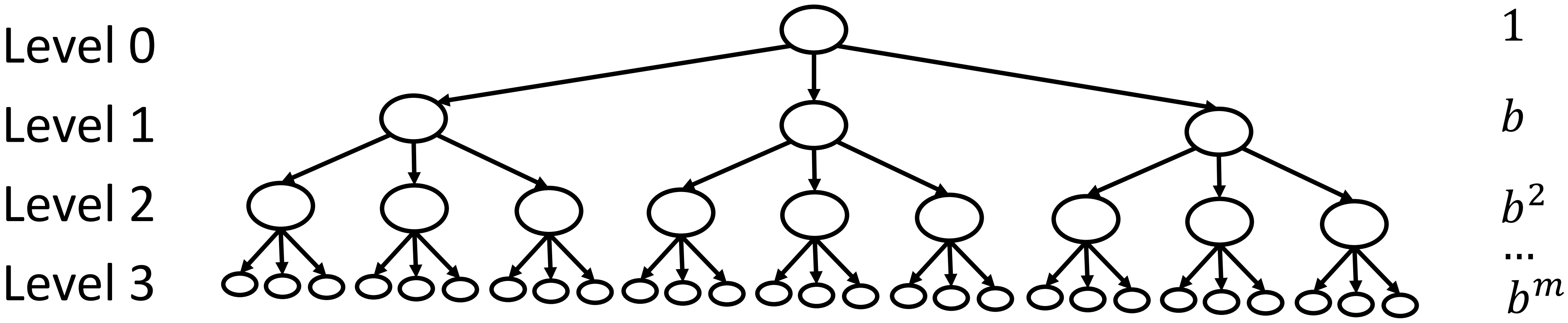


b = maximum branching factor (number of children)

d = depth of shallowest goal node

m = maximum length of any path in the state space

Analyzing Time and Space Complexity of Search in Terms of Trees



$$T_{dfs} = [\cancel{b^m} + \ \ldots + \cancel{b} + 1\,] * c_{dfs}$$

where c_{dfs} is time per node

(solve recurrence)

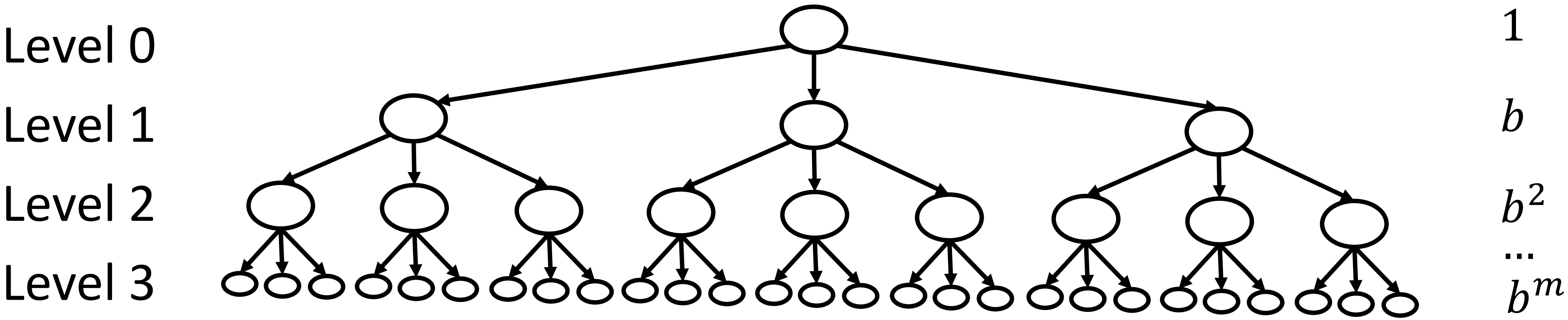
$$b * T_{dfs} = [b^{m+1} \ + \cancel{b^m} \ldots + \cancel{b} + 0\,] * c_{dfs}$$

$$(b-1) * T_{dfs} = [b^{m+1} \ - 1\,] * c_{dfs}$$

$$T_{dfs} = \frac{[b^{m+1} \ - 1\,]}{[b-1]} * c_{dfs}$$

Cost Using Order Notation

Worst case time T is proportional to number of nodes visited



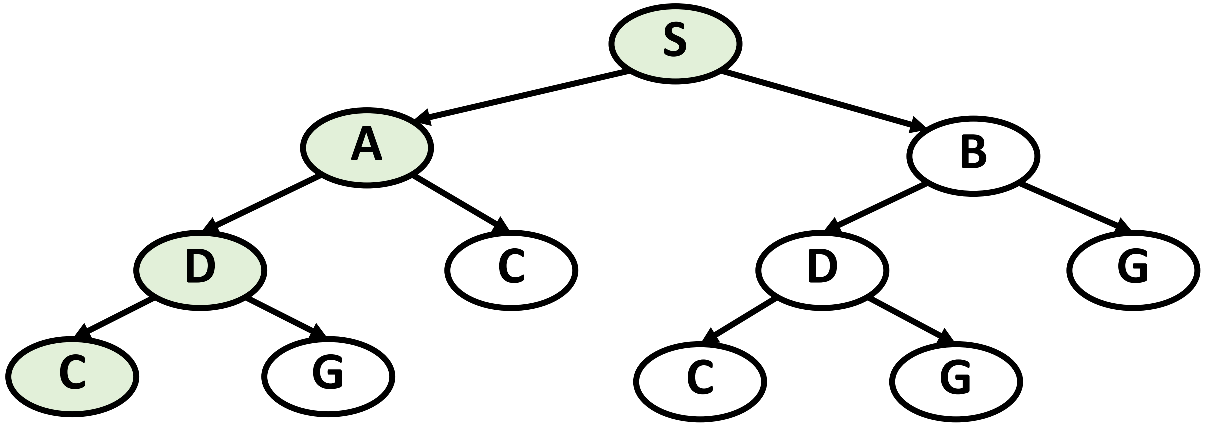
Order Notation

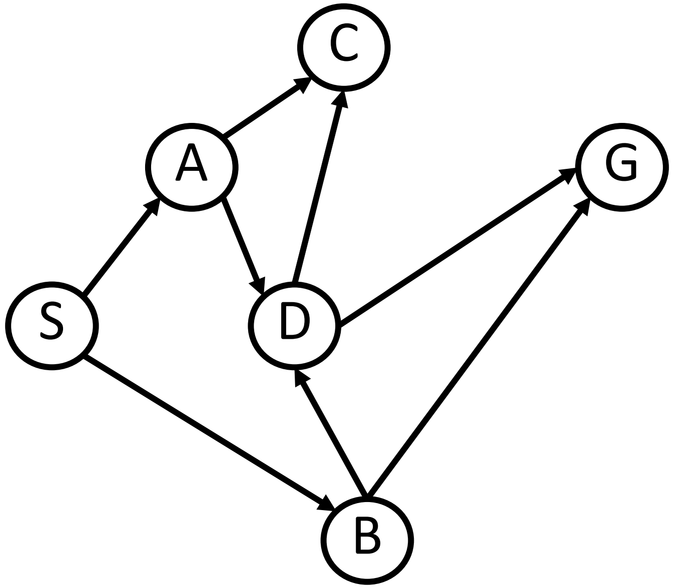
- $T(n) = O\big(e(n)\big)$ if $\mathrm{T} \leq c * e$ for some sufficiently large c & e

$$T_{dfs} = \frac{b^{m+1}}{\mathrm{b}-1} * \mathrm{c}_{\mathrm{dfs}}$$
$$= O(b^{m+1})$$
$$\sim O(b^m) \text{ as } b \to \infty \text{ (used in some texts)}$$

Performance and Resource Usage

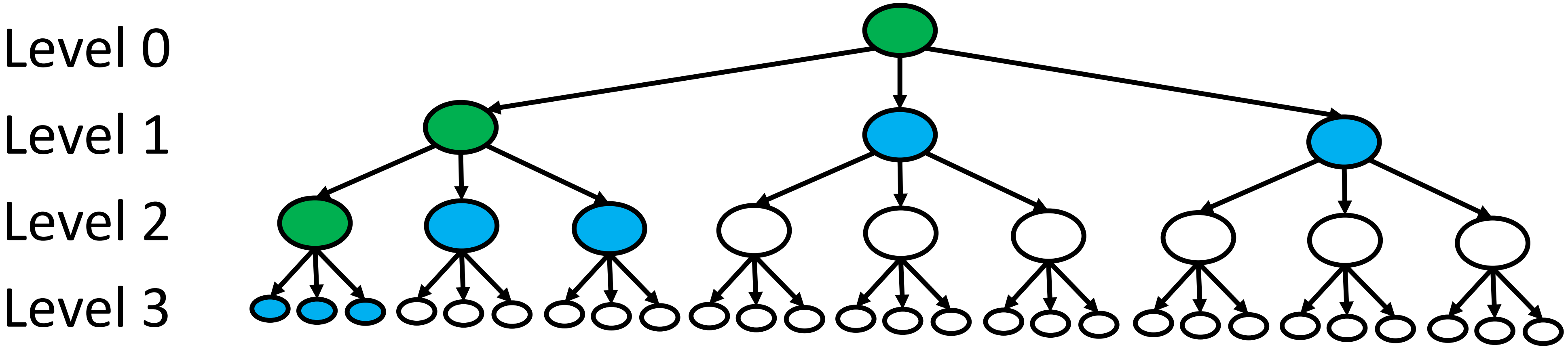
Which is better, DFS or BFS?





Search Method	Worst Time	Worst Space	Shortest Path?	Guaranteed to find path?
DFS	$\sim b^m$			
BFS				

Worst Case Space for Depth-first Search



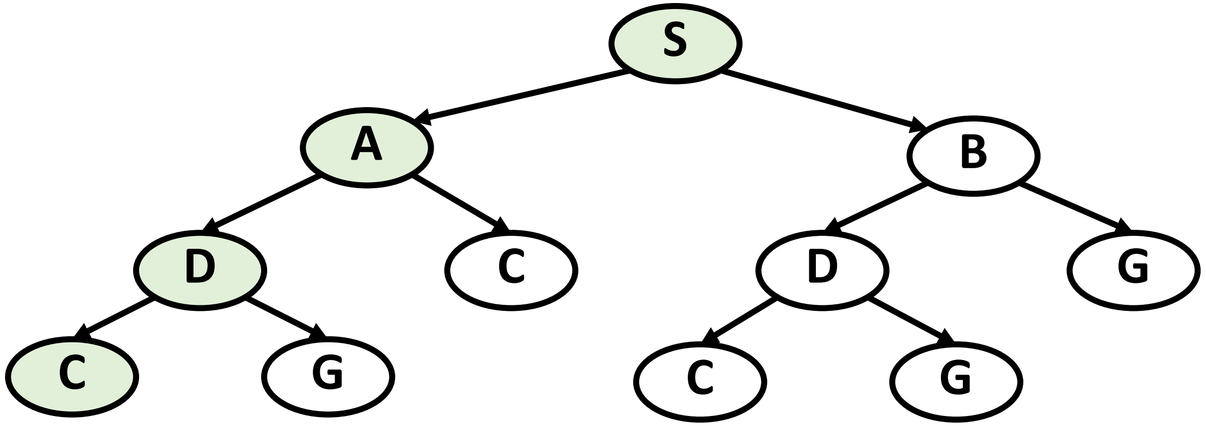
- If a node is queued, its parent and siblings have been queued, and its parent is de-queued.

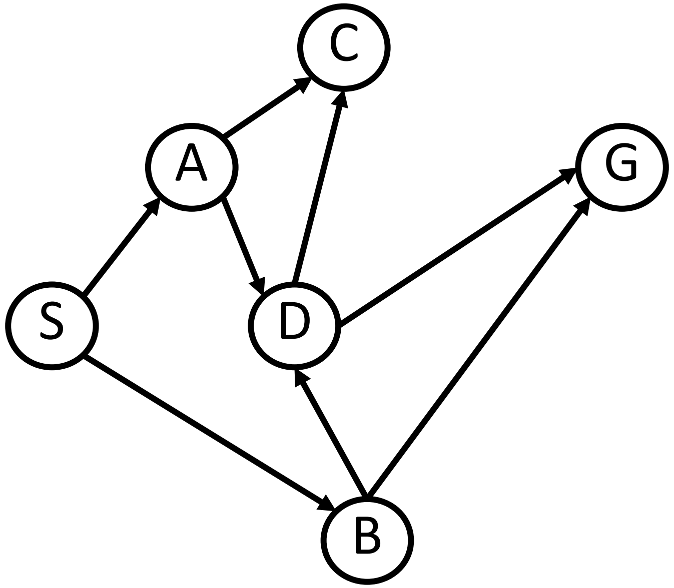
  $S_{dfs} \geq [(b-1) * m + 1] * c_{dfs}$ where c_{dfs} is space per node.
- At most one sibling of a node has its children queued.

  $\rightarrow [(b-1) * m + 1] * c_{dfs}$
- $S_{dfs} = O(b * m)$ //Add visited list!

Performance and Resource Usage

Which is better, DFS or BFS?

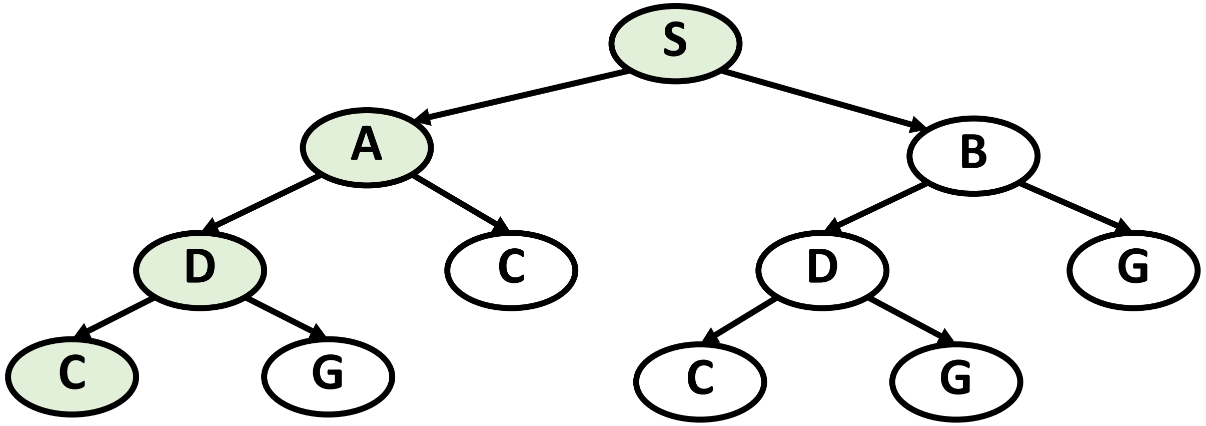


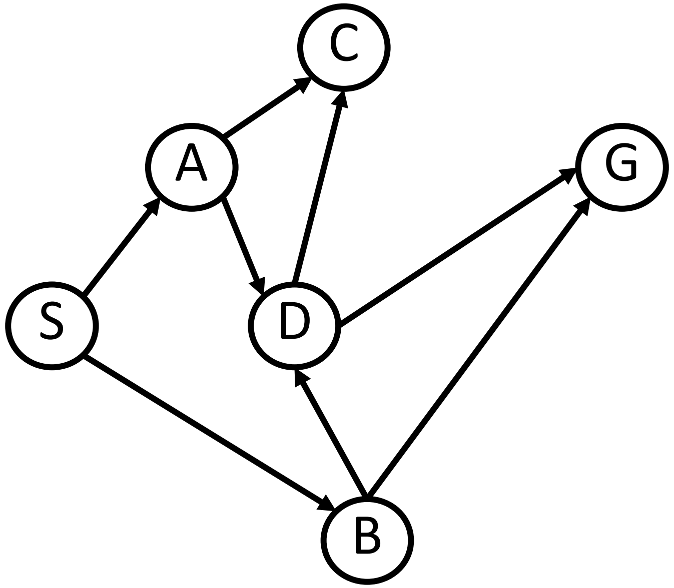


Search Method	Worst Time	Worst Space	Shortest Path?	Guaranteed to find path?
DFS	$\sim b^m$	$b * m$		
BFS				

Performance and Resource Usage

Which is better, DFS or BFS?

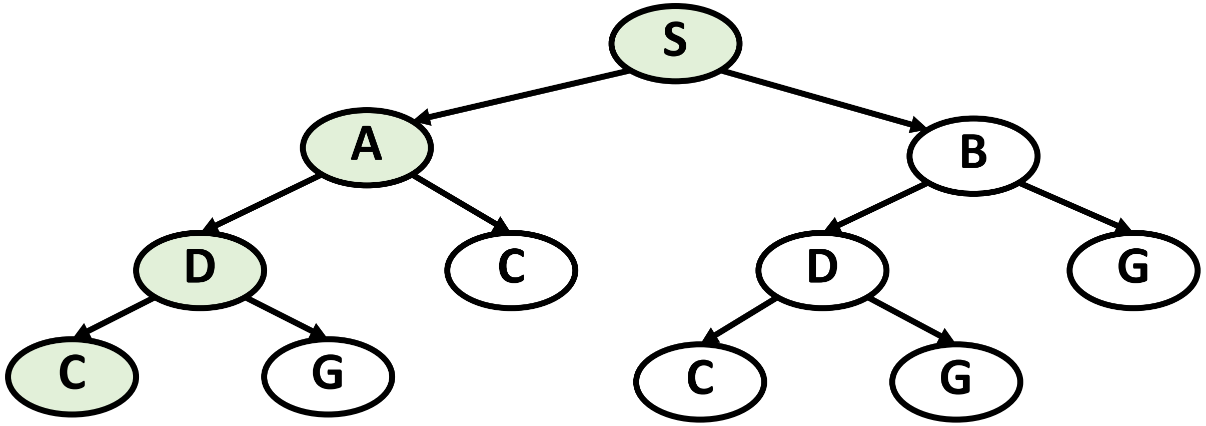


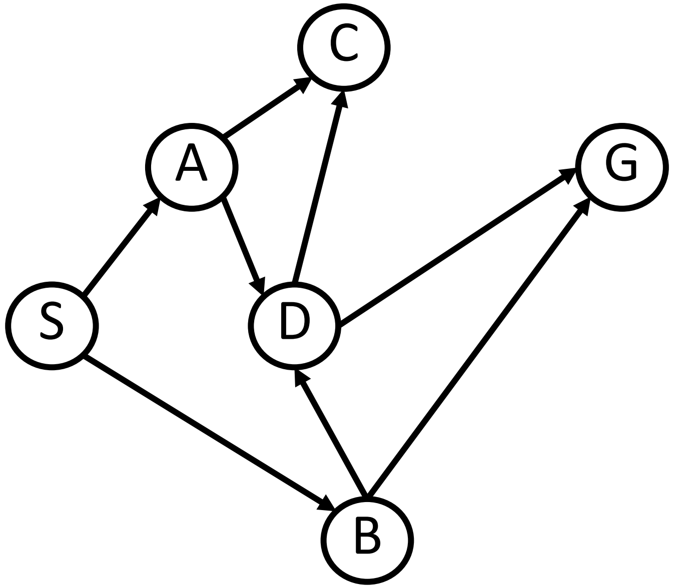


Search Method	Worst Time	Worst Space	Shortest Path?	Guaranteed to find path?
DFS	$\sim b^m$	$b * m$	No	
BFS				

Performance and Resource Usage

Which is better, DFS or BFS?

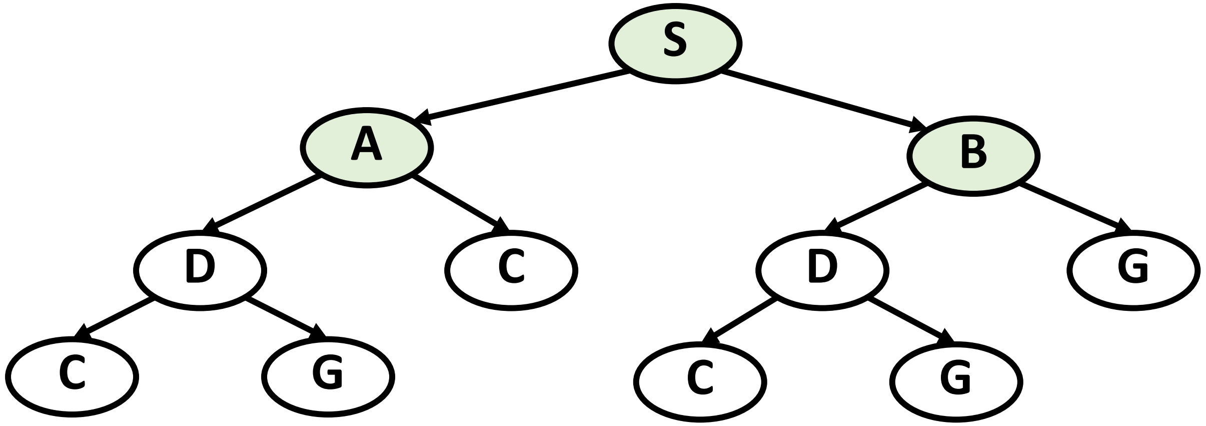


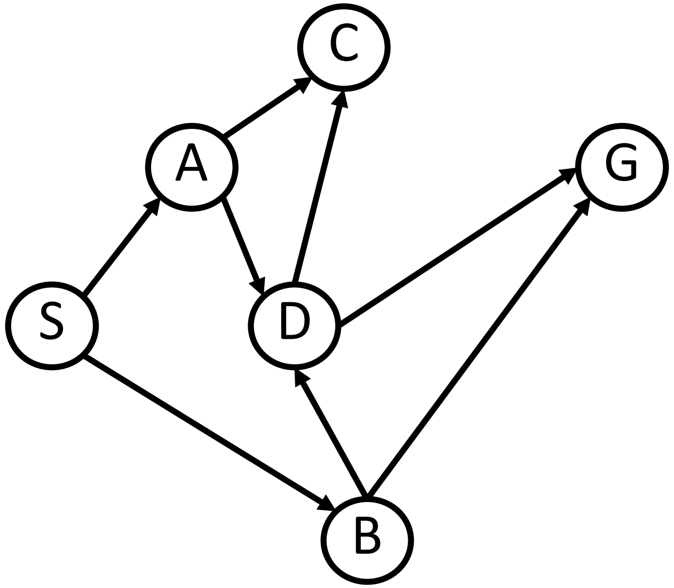


Search Method	Worst Time	Worst Space	Shortest Path?	Guaranteed to find path?
DFS	$\sim b^m$	$b * m$	No	Yes (for finite graph)
BFS				

Performance and Resource Usage

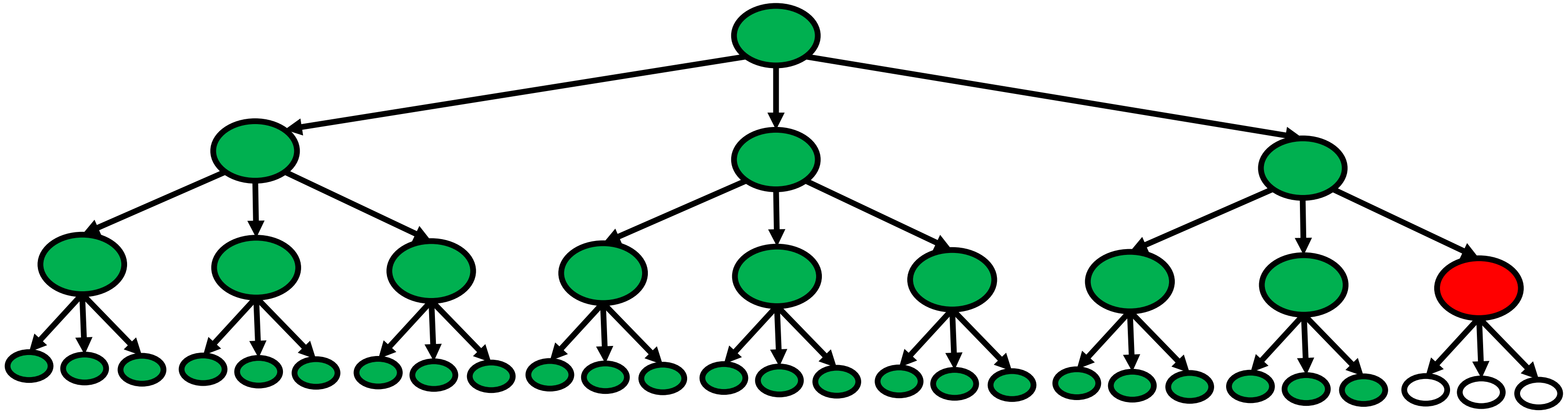
Which is better, DFS or BFS?





Search Method	Worst Time	Worst Space	Shortest Path?	Guaranteed to find path?
DFS	$\sim b^m$	$b * m$	No	Yes (for finite graph)
BFS				

Worst case time for BFS



Level 0 — 1

Level 1 — b

$\cdots$

Level 2 — b^d

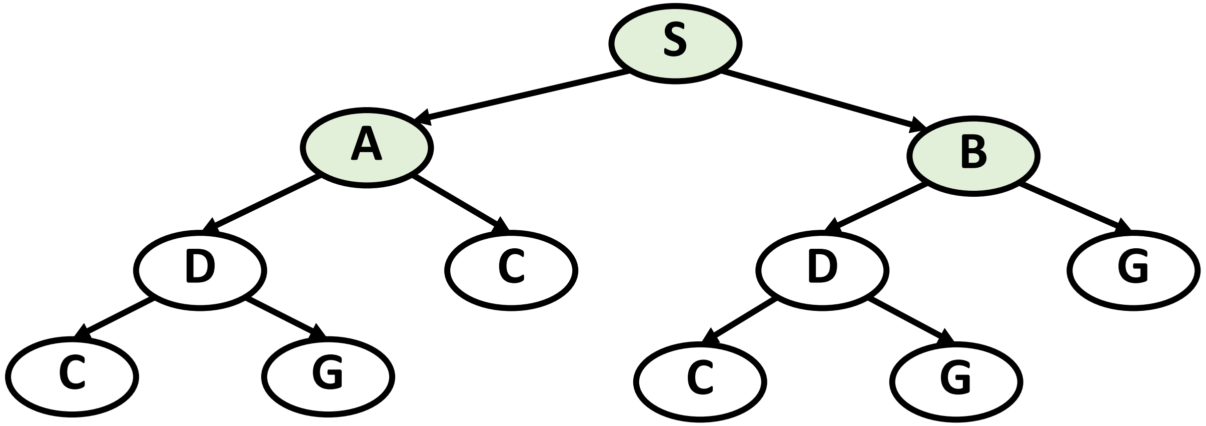
Level 3 — $b^{d+1} - b$

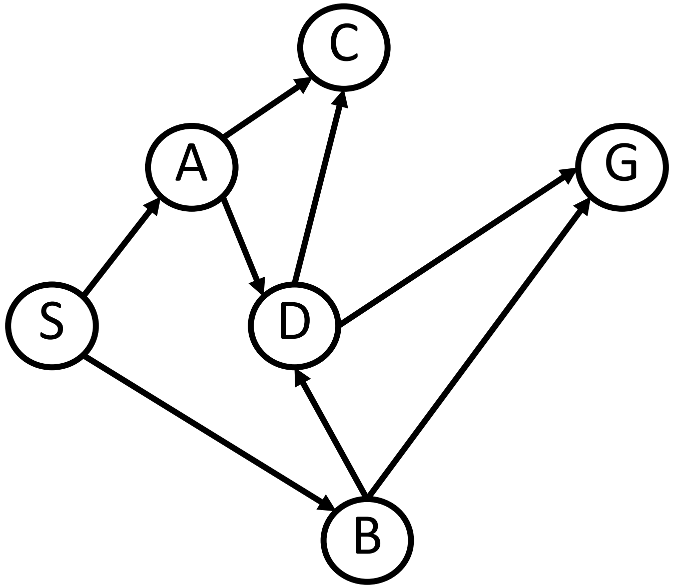
Consider case where solution is at level d (absolute worst is m)

$$T_{bfs} = [b^{d+1} + b^d + \cdots + b + 1 - b] * c_{bfs}$$
$$= [b^{d+2} - b^2 + b - 1]/[b - 1] * c_{bfs}$$
$$= O[b^{d+2}]$$
$$\sim O\left(b^{d+1}\right) \text{ as } b \to \infty$$

Performance and Resource Usage

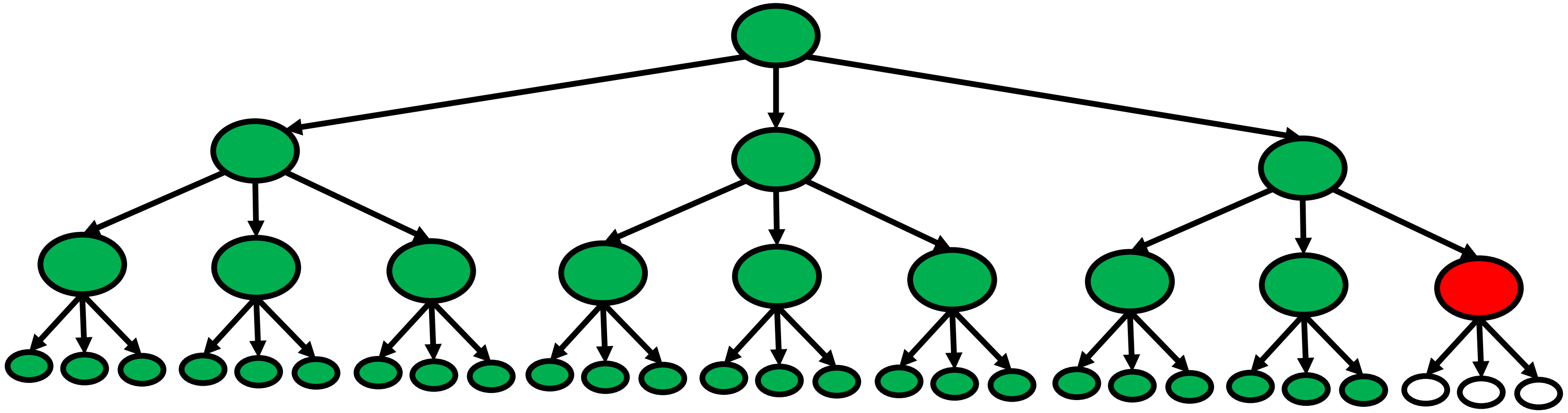
Which is better, DFS or BFS?





Search Method	Worst Time	Worst Space	Shortest Path?	Guaranteed to find path?
DFS	$\sim b^m$	$b * m$	No	Yes (for finite graph)
BFS	$\sim b^{d+1}$			

Worst case space for BFS



Level 0 — 1

Level 1 — b

$\cdots$

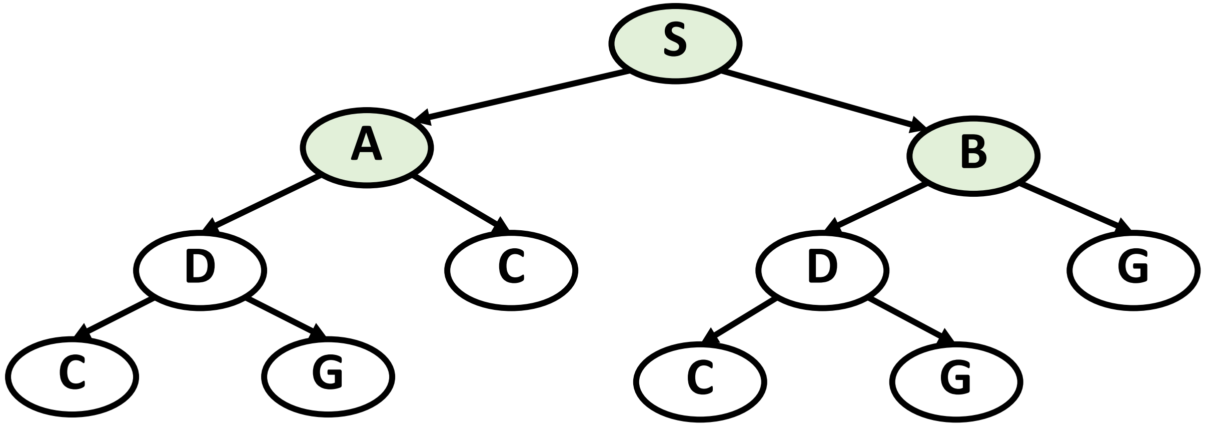
Level 2 — b^d

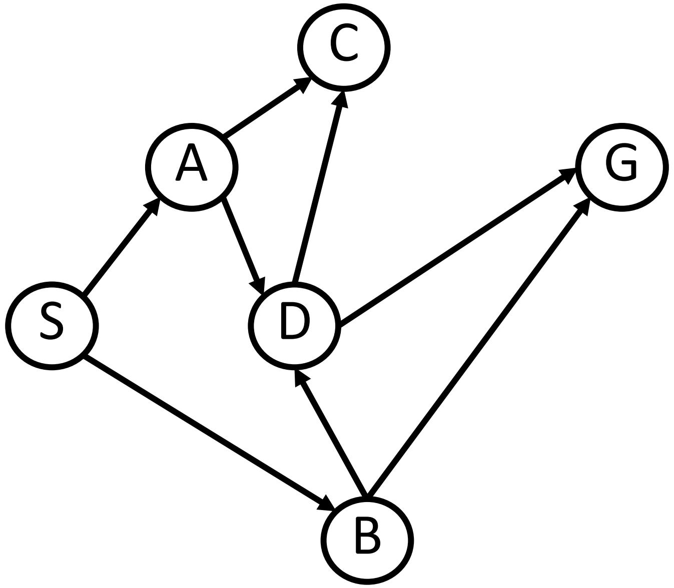
Level 3 — $b^{d+1} - b$

$$S_{bfs} = [b^{d+1} - b + 1] * c_{bfs}$$
$$= O\left(b^{d+1}\right)$$

Performance and Resource Usage

Which is better, DFS or BFS?

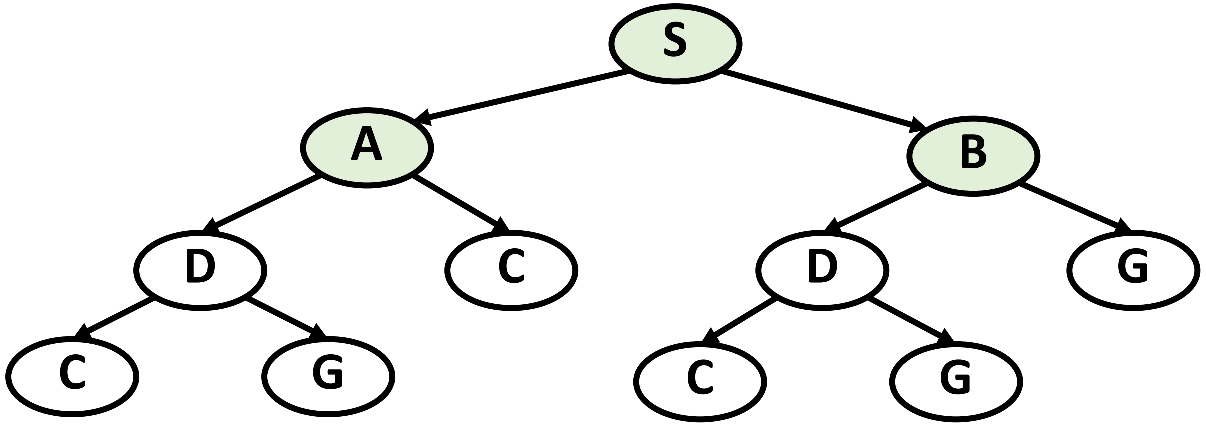


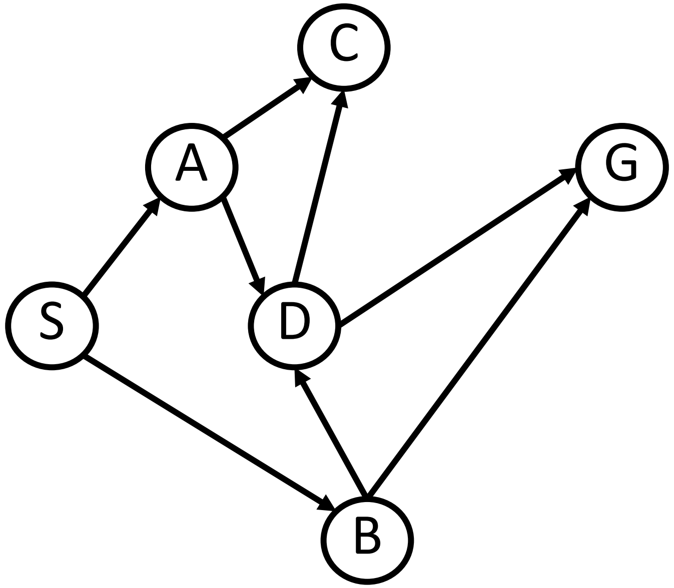


Search Method	Worst Time	Worst Space	Shortest Path?	Guaranteed to find path?
DFS	$\sim b^m$	$b * m$	No	Yes (for finite graph)
BFS	$\sim b^{d+1}$	b^{d+1}		

Performance and Resource Usage

Which is better, DFS or BFS?

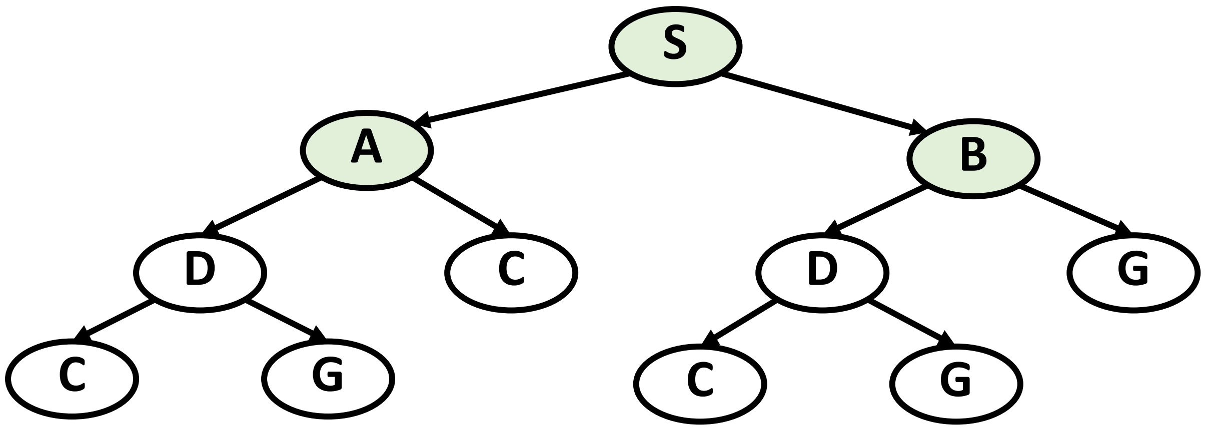


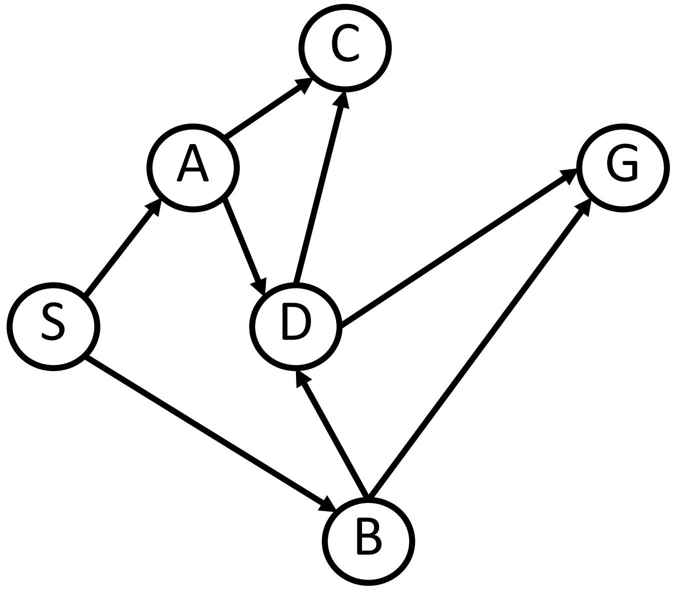


Search Method	Worst Time	Worst Space	Shortest Path?	Guaranteed to find path?
DFS	$\sim b^m$	$b * m$	No	Yes (for finite graph)
BFS	$\sim b^{d+1}$	b^{d+1}	Yes (at unit length)	

Performance and Resource Usage

Which is better, DFS or BFS?

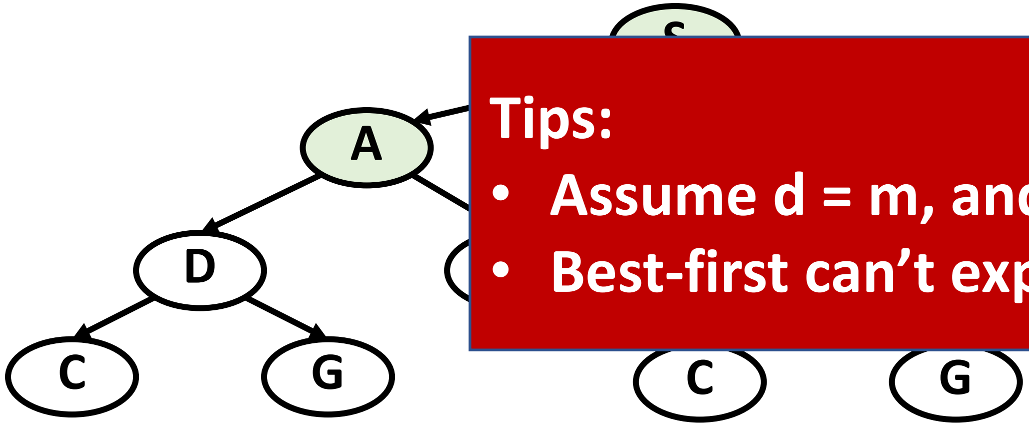




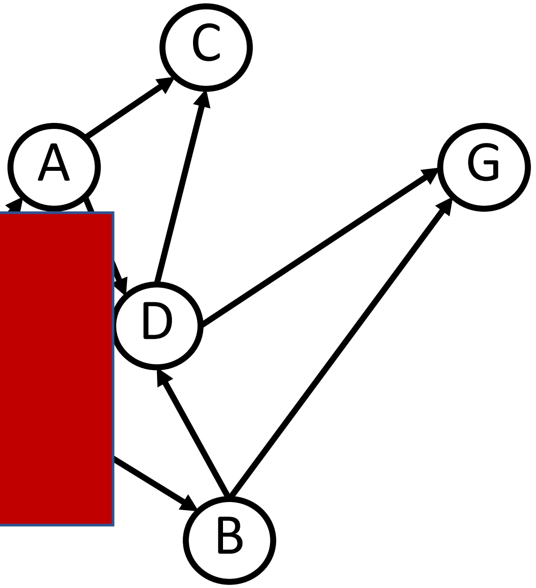
Search Method	Worst Time	Worst Space	Shortest Path?	Guaranteed to find path?
DFS	$\sim b^m$	$b * m$	No	Yes (for finite graph)
BFS	$\sim b^{d+1}$	b^{d+1}	Yes (at unit length)	Yes

Performance and Resource Usage

Which is better, DFS or BFS?



Tips:
- **Assume d = m, and call both m**
- **Best-first can't expand to level m+1, just m**



Search Method	Worst Time	Worst Space	Shortest Path?	Guaranteed to find path?
DFS	$\sim b^m$	$b * m$	No	Yes (for finite graph)
BFS	$\sim b^{d+1}$	b^{d+1}	Yes (at unit length)	Yes

Growth for BFS

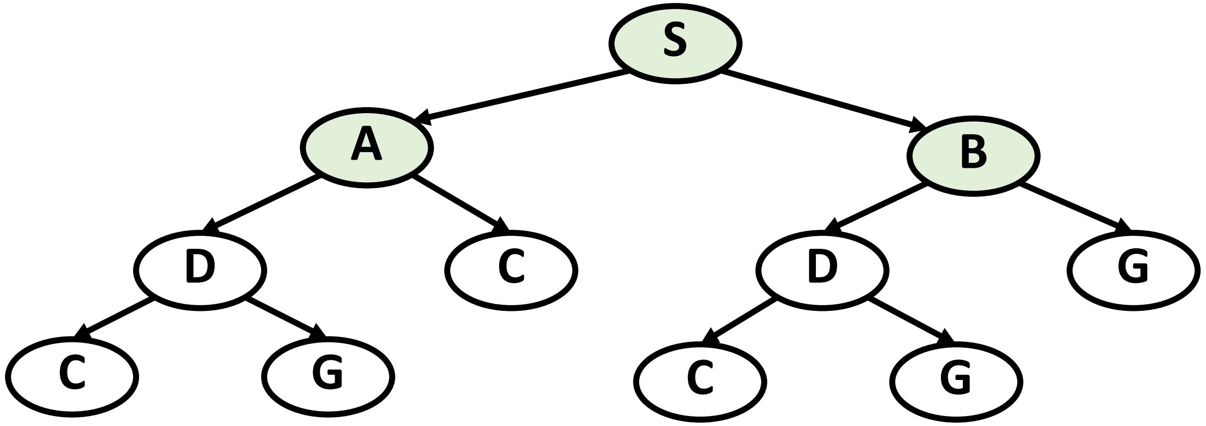
b = 10; 10,000 nodes/sec; 1000 bytes/node

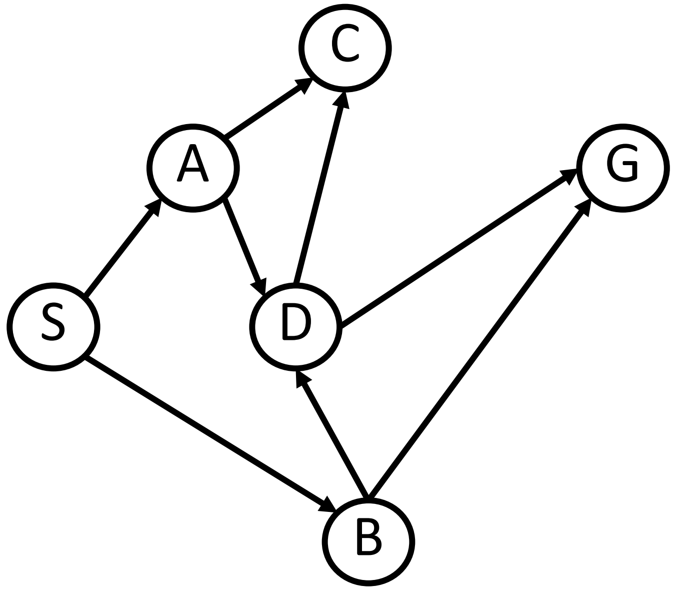
Depth	Nodes	Time	Memory
2	1,100	.11 seconds	1 megabyte
4	111,100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabyte
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3,523 years	1 exabyte

Credit: Brian Williams

How do we get the best of both worlds?

Which is better, DFS or BFS?





Search Method	Worst Time	Worst Space	Shortest Path?	Guaranteed to find path?
DFS	$\cancel{\sim b^m}$	$b * m$	~~No~~	~~Yes (for finite graph)~~
BFS	$\sim b^{d+1}$	$\cancel{b^{d+1}}$	Yes (at unit length)	Yes

Outline:

Complexity of Statespace Search

- Review
- Analysis
  - Depth-first search
  - Breadth-first search
- Iterative deepening

Iterative Deepening Search (IDS)

a.k.a. “Depth-limited Search”

Idea: Explore tree in breath-first order, using DFS

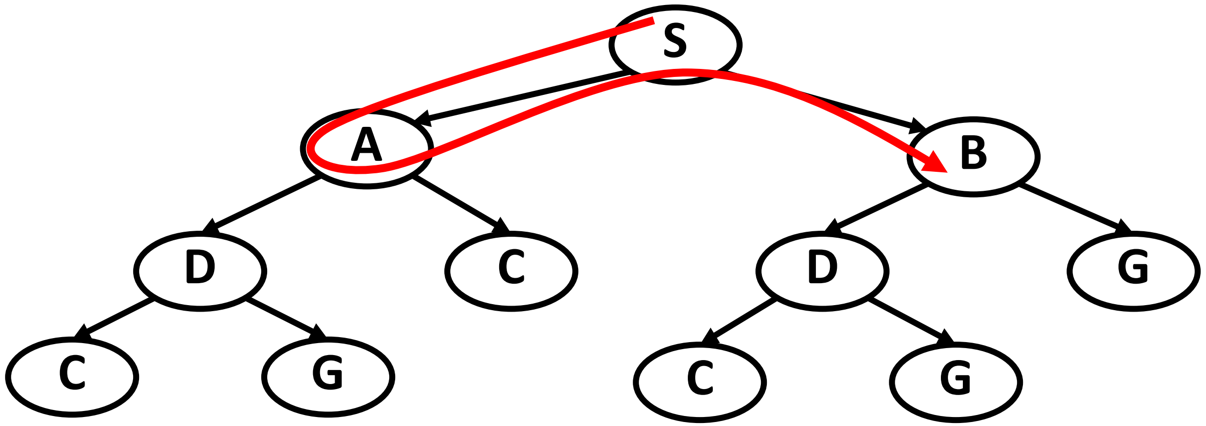
→ Search to tree depth 1, …

Level 0

Level 1

Level 2

Level 3



Iterative Deepening Search (IDS)

a.k.a. "Depth-limited Search"

Idea: Explore tree in breath-first order, using DFS

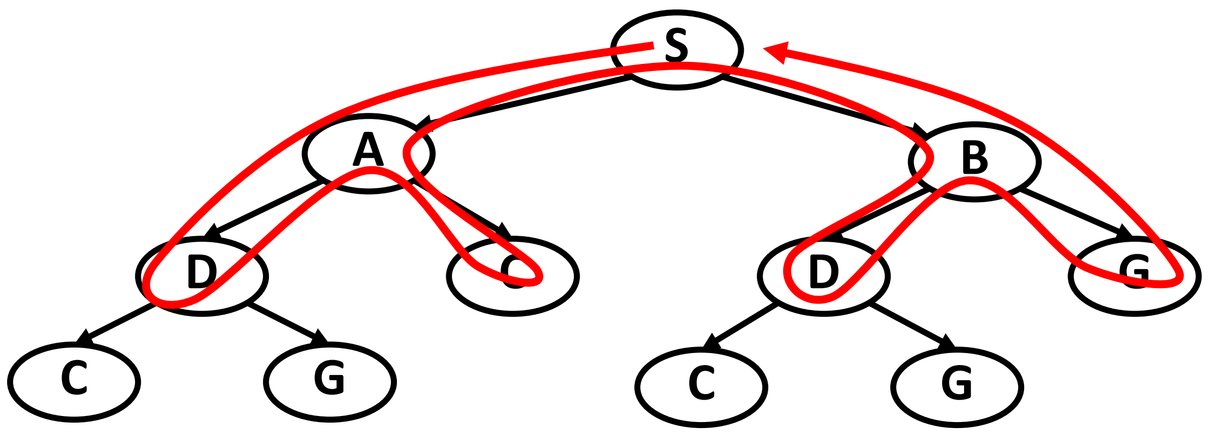
→ Search to tree depth 1, then 2, …

Level 0

Level 1

Level 2

Level 3



Iterative Deepening Search (IDS)

a.k.a. “Depth-limited Search”

Idea: Explore tree in breath-first order, using DFS

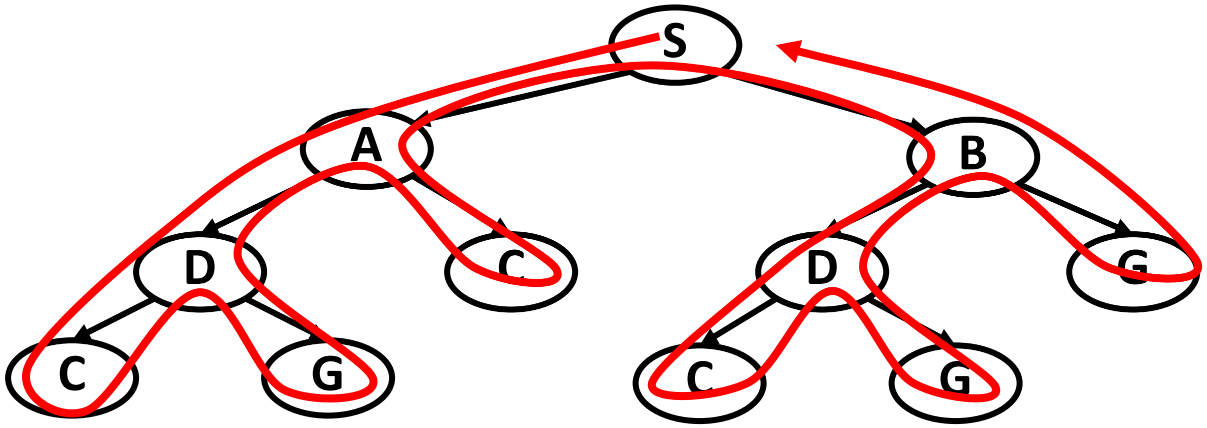
→ Search to tree depth 1, then 2, then 3, …

Level 0

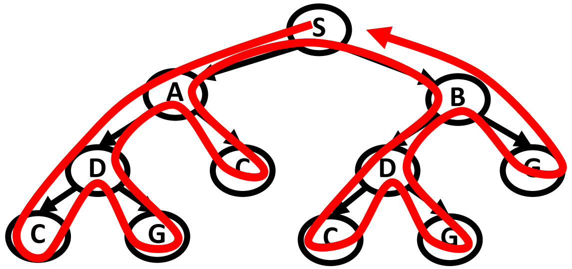
Level 1

Level 2

Level 3



Compare speed of BFS vs. IDS



BFS: $T_{bfs} = 1 + b + b^2 + \cdots + b^d + \left(b^{d+1} - b\right)$

$\sim O\left(b^{d+1}\right)$

IDS: $T_{ids} = (d+1)(1) + (\cancel{d})b + (\cancel{d} - 1)b^2 + \cdots \quad + 2b^{d-1} + \cancel{b^d}$

$bT_{ids} = \qquad (\cancel{d} + 1)(b) + \quad (\cancel{d})b^2 + (d-1)b^3 + \cdots + \cancel{2}b^d + b^{d+1}$

$(b-1)T_{ids} = d + 1 + b + b^2 + \cdots + b^d + b^{d+1}$

$= d + \frac{1-b^{d+2}}{1-b}$

$T_{ids} = \frac{d}{1-b} + \frac{1-b^{d+2}}{(1-b)^2} \sim O\left(\frac{b^{d+2}}{b^2}\right)$

$\sim O\left(b^d\right)$ for large b

Remember... (for r ≠ 1)

$$\sum_{k=0}^{n-1} r^k = \frac{1-r^n}{1-r}$$

→ Iterative deepening performs better than breadth-first!

Summary

- Most problem solving tasks may be encoded as state space search
- Basic data structures for search are graphs and search trees
- DFS and BFS may be framed as instances of generic search strategies
- Cycle detection is required to achieve efficiency and completeness

New:

- Complexity analysis shows that BFS is preferred in terms of optimality and time, while DFS is preferred in terms of space
- IDS draws the best from DFS and BFS

Mid-Lecture Break

[DARPA AlphaDog Competition - Heron vs Banger - Round 5]

The mystique of the “knight”…no more?

Elements of Algorithmic Design

Description: (Last Wednesday)

- Problem statement
- Stylized pseudo code, sufficient to analyze and implement the algorithm

Analysis:

- Performance:
    - Time complexity
        - How long does it take to find a solution?
    - Space complexity:
        - How much memory does it need to perform search?
- Correctness:
    - Soundness:
        - When a solution is returned, is it guaranteed to be correct?
    - Completeness:
        - Is the algorithm guaranteed to find a solution when there is one?

Proof by Invariance

A common technique in algorithm analysis

- Show that a certain property holds throughout an algorithm
- Assume that the property holds initially
- Show that in any step that the algorithm takes, the property still holds
- The, property holds forever.

Proving statements about algorithms

- Correctness of simplest algorithms may be very hard to prove…

- Collatz conjecture:
  - Algorithms (Half or Triple Plus One – HOTPO)
    - Give an integer n
    1. If n is even, then $n = \frac{n}{2}$
    2. If n is odd, then $n = 3n + 1$
    3. If $n = 1$, then terminate, else go to Step 1

  - Conjecture: For any n, the algorithm always terminates (with n = 1)

Proving statements about algorithms

Collatz conjecture:

- First proposed in 1937

- It is not known whether the conjecture is true or false

Paul Erdős (1913-1996), a famous number theorist said, “Mathematics is not yet ready for such problems” in 1956.

Jeffrey Lagarias said, “This is…completely out of reach of present day mathematics” in 2010.

Probabilistic/Asymptotic Types

Probabilistic Completeness

- The algorithm returns a solution, if one exists, with probability approach one as the number of iterations increases.
- If there is no solution, it may run forever.

Probabilistic Soundness

- The probability that the "solution" reported solve the problem approaches one as the number of iterations increases.

Asymptotic Optimality

- The algorithm does not necessarily return an optimal solution, but the cost of the solution reported approaches the optimal as the number of iterations increases.

Soundness & Completeness Theorems

We would like to prove the following two theorems:

Theorem 1 (Soundness):
- Simple search algorithm is sound.

Theorem 2 (Completeness):
- Simple search algorithm is complete.

We will use a blend of proof techniques for proving them.

Soundness & Completeness Theorems

Theorem 1 (Soundness):

- Simple search algorithm is sound.

Let us prove 3 lemmas before proving this theorem

A lemma towards the proof

- **Lemma 1**: If $\langle v_1, v_2, \ldots, v_k \rangle$ is the path in the queue at any given time, then $v_k = S$
- **Proof:** (by invariance)
  - *Base case:* Initially, there is only $\langle S \rangle$ in the queue. Hence, the invariant holds.
  - *Induction step:* Let us check that the invariant continues to hold in every step of the algorithm

Simple Search Algorithm

Invariant: **If $\langle v_1, v_2, \dots, v_k \rangle$ is the path in the queue at any given time, then $v_k = S$.**

1. Initialize Q with a partial path <S>; set Visited = {}
2. If Q is empty, fail. Else, pick a partial path N from Q
3. If head(N) = G, return N   //Goal reach!
4. Else:
   a) Remove N from Q
   b) Find all children of head(N) not in Visited and create a on-step extension of N to each child
   c) Add all extended paths to Q
   d) Add Children of head(N) to Visited
   e) GOTO step 2

Simple Search Algorithm

Invariant: **If $\langle v_1, v_2, \dots, v_k \rangle$ is the path in the queue at any given time, then $v_k = S$.**

1. Initialize Q with a partial path <S>; set Visited = {}
2. **If Q is empty, fail. Else, pick a partial path N from Q**
3. If head(N) = G, return N //Goal reach!
4. Else:
   a) Remove N from Q
   b) Find all children of head(N) not in Visited and create a on-step extension of N to each child
   c) Add all extended paths to Q
   d) Add Children of head(N) to Visited
   e) GOTO step 2

Before this line: assume that invariant holds.

After this line: show that this invariant is still true.

→ In this case, no new path is added to the queue.

Simple Search Algorithm

Invariant: **If $\langle v_1, v_2, \ldots, v_k \rangle$ is the path in the queue at any given time, then $v_k = S$.**

1. Initialize Q with a partial path <S>; set Visited = {}
2. If Q is empty, fail. Else, pick a partial path N from Q
3. **If head(N) = G, return N //Goal reach!**
4. Else:
   a) Remove N from Q
   b) Find all children of head(N) not in Visited and create a on-step extension of N to each child
   c) Add all extended paths to Q
   d) Add Children of head(N) to Visited
   e) GOTO step 2

Before this line: assume that invariant holds.

After this line: show that this invariant is still true.

→ In this case, no new path is added to the queue.

Simple Search Algorithm

Invariant: **If $\langle v_1, v_2, \dots, v_k \rangle$ is the path in the queue at any given time, then $v_k = S$.**

1. Initialize Q with a partial path <S>; set Visited = {}
2. If Q is empty, fail. Else, pick a partial path N from Q
3. If head(N) = G, return N //Goal reach!
4. **Else:**
   a) **Remove N from Q**
   b) Find all children of head(N) not in Visited and create a on-step extension of N to each child
   c) Add all extended paths to Q
   d) Add Children of head(N) to Visited
   e) GOTO step 2

Before this line: assume that invariant holds.

After this line: show that this invariant is still true.

→ In this case, no new path is added to the queue (one is removed).

Simple Search Algorithm

Invariant: **If $\langle v_1, v_2, \dots, v_k \rangle$ is the path in the queue at any given time, then $v_k = S$.**

1. Initialize Q with a partial path <S>; set Visited = {}
2. If Q is empty, fail. Else, pick a partial path N from Q
3. If head(N) = G, return N //Goal reach!
4. **Else:**
   a) Remove N from Q
   b) **Find all children of head(N) not in Visited and create a on-step extension of N to each child**
   c) **Add all extended paths to Q**
   d) Add Children of head(N) to Visited
   e) GOTO step 2

Before this line: assume that invariant holds.

After this line: show that this invariant is still true.

→ Several paths added, each satisfy the invariant since N Satisfies it.

Simple Search Algorithm

Invariant: **If $\langle v_1, v_2, \dots, v_k \rangle$ is the path in the queue at any given time, then $v_k = S$.**

1. Initialize Q with a partial path <S>; set Visited = {}
2. If Q is empty, fail. Else, pick a partial path N from Q
3. If head(N) = G, return N //Goal reach!
4. **Else:**
   a) Remove N from Q
   b) Find all children of head(N) not in Visited and create a on-step extension of N to each child
   c) Add all extended paths to Q
   d) **Add Children of head(N) to Visited**
   e) GOTO step 2

Before this line: assume that invariant holds.

After this line: show that this invariant is still true.

→ In this case, no new path is added to the queue.

Simple Search Algorithm

Invariant: **If $\langle v_1, v_2, \dots, v_k \rangle$ is the path in the queue at any given time, then $v_k = S$.**

1. Initialize Q with a partial path <S>; set Visited = {}
2. If Q is empty, fail. Else, pick a partial path N from Q
3. If head(N) = G, return N //Goal reach!
4. **Else:**
   a) Remove N from Q
   b) Find all children of head(N) not in Visited and create a on-step extension of N to each child
   c) Add all extended paths to Q
   d) Add Children of head(N) to Visited
   e) **GOTO step 2**

Before this line: assume that invariant holds.

After this line: show that this invariant is still true.

→ In this case, no new path is added to the queue.

Another lemma towards the proof

- **Definition:** A path $\langle v_1, v_2, \ldots, v_k \rangle$ is valid if $\langle v_{i-1}, v_i \rangle \in E$ for all $i \in \{1,2, \ldots, k\}$

- **Lemma 2**: If $\langle v_1, v_2, \ldots, v_k \rangle$ is a path in the queue at any given time, then it is valid.

- **Proof:** (by invariance)
  - *Base case:* Initially there is only one path $\langle S \rangle$, which is valid. Hence, the invariant holds.
  - *Induction step:* Let us check that the invariant continues to hold in every step of the algorithm.

Simple Search Algorithm

Invariant: If $\langle v_1, v_2, \dots, v_k \rangle$ is a path in the queue at any given time, then it is valid.

1. Initialize Q with a partial path <S>; set Visited = {}
2. If Q is empty, fail. Else, pick a partial path N from Q
3. If head(N) = G, return N //Goal reach!
4. Else:
   a) Remove N from Q
   b) Find all children of head(N) not in Visited and create a on-step extension of N to each child
   c) Add all extended paths to Q
   d) Add Children of head(N) to Visited
   e) GOTO step 2

Simple Search Algorithm

Invariant: If $\langle v_1, v_2, \ldots, v_k \rangle$ is a path in the queue at any given time, then it is valid.

1. **Initialize Q with a partial path <S>; set Visited = {}**
2. If Q is empty, fail. Else, pick a partial path N from Q
3. If head(N) = G, return N //Goal reach!
4. Else:
   a) Remove N from Q
   b) Find all children of head(N) not in Visited and create a on-step extension of N to each child
   c) Add all extended paths to Q
   d) Add Children of head(N) to Visited
   e) GOTO step 2

Before this line: assume that invariant holds.

After this line: show that this invariant is still true.

→ In this case, no new path is added to the queue.

Simple Search Algorithm

Invariant: If $\langle v_1, v_2, \dots, v_k \rangle$ is a path in the queue at any given time, then it is valid.

1. Initialize Q with a partial path <S>; set Visited = {}
2. **If Q is empty, fail. Else, pick a partial path N from Q**
3. If head(N) = G, return N //Goal reach!
4. Else:
   a) Remove N from Q
   b) Find all children of head(N) not in Visited and create a on-step extension of N to each child
   c) Add all extended paths to Q
   d) Add Children of head(N) to Visited
   e) GOTO step 2

Before this line: assume that invariant holds.

After this line: show that this invariant is still true.

→ In this case, no new path is added to the queue.

Simple Search Algorithm

Invariant: If $\langle v_1, v_2, \ldots, v_k \rangle$ is a path in the queue at any given time, then it is valid.

1. Initialize Q with a partial path <S>; set Visited = {}
2. If Q is empty, fail. Else, pick a partial path N from Q
3. **If head(N) = G, return N   //Goal reach!**
4. Else:
   a) Remove N from Q
   b) Find all children of head(N) not in Visited and create a on-step extension of N to each child
   c) Add all extended paths to Q
   d) Add Children of head(N) to Visited
   e) GOTO step 2

Before this line: assume that invariant holds.

After this line: show that this invariant is still true.

→ In this case, no new path is added to the queue.

Simple Search Algorithm

Invariant: If $\langle v_1, v_2, \ldots, v_k \rangle$ is a path in the queue at any given time, then it is valid.

1. Initialize Q with a partial path <S>; set Visited = {}
2. If Q is empty, fail. Else, pick a partial path N from Q
3. If head(N) = G, return N //Goal reach!
4. **Else:**
   a) **Remove N from Q**
   b) Find all children of head(N) not in Visited and create a on-step extension of N to each child
   c) Add all extended paths to Q
   d) Add Children of head(N) to Visited
   e) GOTO step 2

Before this line: assume that invariant holds.

After this line: show that this invariant is still true.

→ In this case, no new path is added to the queue.

Simple Search Algorithm

Invariant: If $\langle v_1, v_2, \dots, v_k \rangle$ is a path in the queue at any given time, then it is valid.

1. Initialize Q with a partial path <S>; set Visited = {}
2. If Q is empty, fail. Else, pick a partial path N from Q
3. If head(N) = G, return N //Goal reach!
4. **Else:**
   a) Remove N from Q
   b) **Find all children of head(N) not in Visited and create a on-step extension of N to each child**
   c) **Add all extended paths to Q**
   d) Add Children of head(N) to Visited
   e) GOTO step 2

Before this line: assume that invariant holds.

After this line: show that this invariant is still true.

→ Note that validity holds for all newly added path (from Line 4.b)

Simple Search Algorithm

Invariant: If $\langle v_1, v_2, \dots, v_k \rangle$ is a path in the queue at any given time, then it is valid.

1. Initialize Q with a partial path <S>; set Visited = {}
2. If Q is empty, fail. Else, pick a partial path N from Q
3. If head(N) = G, return N //Goal reach!
4. **Else:**
   a) Remove N from Q
   b) Find all children of head(N) not in Visited and create a on-step extension of N to each child
   c) Add all extended paths to Q
   d) **Add Children of head(N) to Visited**
   e) GOTO step 2

Before this line: assume that invariant holds.

After this line: show that this invariant is still true.

→ In this case, no new path is added to the queue.

Simple Search Algorithm

Invariant: If $\langle v_1, v_2, \dots, v_k \rangle$ is a path in the queue at any given time, then it is valid.

1. Initialize Q with a partial path <S>; set Visited = {}
2. If Q is empty, fail. Else, pick a partial path N from Q
3. If head(N) = G, return N //Goal reach!
4. **Else:**
   a) Remove N from Q
   b) Find all children of head(N) not in Visited and create a on-step extension of N to each child
   c) Add all extended paths to Q
   d) Add Children of head(N) to Visited
   e) **GOTO step 2**

Before this line: assume that invariant holds.

After this line: show that this invariant is still true.

→ In this case, no new path is added to the queue.

Simple Search Algorithm

Invariant: If $\langle v_1, v_2, \ldots, v_k \rangle$ is a path in the queue at any given time, then it is valid.

1. Initialize Q with a partial path <S>; set Visited = {}
2. If Q is empty, fail. Else, pick a partial path N from Q
3. If head(N) = G, return N //Goal reach!
4. Else:
   a) Remove N from Q
   b) Find all children of head(N) not in Visited and create a on-step extension of N to each child
   c) Add all extended paths to Q
   d) Add Children of head(N) to Visited
   e) GOTO step 2

Before this line: assume that invariant holds.

After this line: show that this invariant is still true.

→ In this case, no new path is added to the queue.

Yet another lemma towards the proof

- **Lemma 3**: If $\langle v_1, v_2, \ldots, v_k \rangle$ is a path in the queue at any given time, then it is a simple path (contains no cycles)

- **Proof:** (by invariance)
  - *Base case:* Initially there is only one path $\langle S \rangle$, which is valid. Hence, the invariant holds.
  - *Induction step:* Let us check that the invariant continues to hold in every step of the algorithm.

Simple Search Algorithm

Invariant: If $\langle v_1, v_2, \dots, v_k \rangle$ is a path in the queue at any given time, then it is a simple path.

1. Initialize Q with a partial path <S>; set Visited = {}
2. If Q is empty, fail. Else, pick a partial path N from Q
3. If head(N) = G, return N //Goal reach!
4. Else:
   a) Remove N from Q
   b) Find all children of head(N) not in Visited and create a on-step extension of N to each child
   c) Add all extended paths to Q
   d) Add Children of head(N) to Visited
   e) GOTO step 2

Simple Search Algorithm

Invariant: If $\langle v_1, v_2, \dots, v_k \rangle$ is a path in the queue at any given time, then it is simple path.

1. **Initialize Q with a partial path <S>; set Visited = {}**
2. If Q is empty, fail. Else, pick a partial path N from Q
3. If head(N) = G, return N //Goal reach!
4. Else:
   a) Remove N from Q
   b) Find all children of head(N) not in Visited and create a on-step extension of N to each child
   c) Add all extended paths to Q
   d) Add Children of head(N) to Visited
   e) GOTO step 2

Before this line: assume that invariant holds.

After this line: show that this invariant is still true.

→ In this case, no new path is added to the queue.

Simple Search Algorithm

Invariant: If $\langle v_1, v_2, \dots, v_k \rangle$ is a path in the queue at any given time, then it is simple path.

1. Initialize Q with a partial path <S>; set Visited = {}
2. **If Q is empty, fail. Else, pick a partial path N from Q**
3. If head(N) = G, return N //Goal reach!
4. Else:
   a) Remove N from Q
   b) Find all children of head(N) not in Visited and create a on-step extension of N to each child
   c) Add all extended paths to Q
   d) Add Children of head(N) to Visited
   e) GOTO step 2

Before this line: assume that invariant holds.

After this line: show that this invariant is still true.

→ In this case, no new path is added to the queue.

Simple Search Algorithm

Invariant: If $\langle v_1, v_2, \ldots, v_k \rangle$ is a path in the queue at any given time, then it is simple path.

1. Initialize Q with a partial path <S>; set Visited = {}
2. If Q is empty, fail. Else, pick a partial path N from Q
3. **If head(N) = G, return N   //Goal reach!**
4. Else:
   a) Remove N from Q
   b) Find all children of head(N) not in Visited and create a on-step extension of N to each child
   c) Add all extended paths to Q
   d) Add Children of head(N) to Visited
   e) GOTO step 2

Before this line: assume that invariant holds.

After this line: show that this invariant is still true.

→ In this case, no new path is added to the queue.

Simple Search Algorithm

Invariant: If $\langle v_1, v_2, \dots, v_k \rangle$ is a path in the queue at any given time, then it is simple path.

1. Initialize Q with a partial path <S>; set Visited = {}
2. If Q is empty, fail. Else, pick a partial path N from Q
3. If head(N) = G, return N //Goal reach!
4. **Else:**
   a) **Remove N from Q**
   b) Find all children of head(N) not in Visited and create a on-step extension of N to each child
   c) Add all extended paths to Q
   d) Add Children of head(N) to Visited
   e) GOTO step 2

Before this line: assume that invariant holds.

After this line: show that this invariant is still true.

→ In this case, no new path is added to the queue.

Simple Search Algorithm

Invariant: If $\langle v_1, v_2, \dots, v_k \rangle$ is a path in the queue at any given time, then it is simple path.

1. Initialize Q with a partial path <S>; set Visited = {}
2. If Q is empty, fail. Else, pick a partial path N from Q
3. If head(N) = G, return N //Goal reach!
4. **Else:**
   a) Remove N from Q
   b) **Find all children of head(N) not in Visited and create a on-step extension of N to each child**
   c) **Add all extended paths to Q**
   d) Add Children of head(N) to Visited
   e) GOTO step 2

We would like to show that each newly added path is simple assuming N is simple.

Proof: (by contradiction) Assume one path is not simple. Then, a child of head(N) appears in N. But this contradicts step for 4.b

Simple Search Algorithm

Invariant: If $\langle v_1, v_2, \dots, v_k \rangle$ is a path in the queue at any given time, then it is simple path.

1. Initialize Q with a partial path <S>; set Visited = {}
2. If Q is empty, fail. Else, pick a partial path N from Q
3. If head(N) = G, return N //Goal reach!
4. **Else:**
   a) Remove N from Q
   b) Find all children of head(N) not in Visited and create a on-step extension of N to each child
   c) Add all extended paths to Q
   d) **Add Children of head(N) to Visited**
   e) GOTO step 2

Before this line: assume that invariant holds.

After this line: show that this invariant is still true.

→ In this case, no new path is added to the queue.

Simple Search Algorithm

Invariant: If $\langle v_1, v_2, \dots, v_k \rangle$ is a path in the queue at any given time, then it is simple path.

1. Initialize Q with a partial path <S>; set Visited = {}
2. If Q is empty, fail. Else, pick a partial path N from Q
3. If head(N) = G, return N //Goal reach!
4. **Else:**
   a) Remove N from Q
   b) Find all children of head(N) not in Visited and create a on-step extension of N to each child
   c) Add all extended paths to Q
   d) Add Children of head(N) to Visited
   e) **GOTO step 2**

Before this line: assume that invariant holds.

After this line: show that this invariant is still true.

→ In this case, no new path is added to the queue.

Simple Search Algorithm

Invariant: If $\langle v_1, v_2, \ldots, v_k \rangle$ is a path in the queue at any given time, then it is simple path.

1. Initialize Q with a partial path <S>; set Visited = {}
2. If Q is empty, fail. Else, pick a partial path N from Q
3. If head(N) = G, return N   //Goal reach!
4. Else:
   a) Remove N from Q
   b) Find all children of head(N) not in Visited and create a on-step extension of N to each child
   c) Add all extended paths to Q
   d) Add Children of head(N) to Visited
   e) GOTO step 2

Before this line: assume that invariant holds.

After this line: show that this invariant is still true.

→ In this case, no new path is added to the queue.

Soundness & Completeness Theorems

Theorem 1 (Soundness):

- Simple search algorithm is sound.

Proof: by contradiction…

Proof of Soundness

Assume that the search algorithm is ***not*** **sound:**

Let the returned path be $\langle v_1, v_2, \ldots, v_k \rangle$

Then, one of the following must be TRUE:

1) Returned path does not start with S:

$$v_k \neq S$$

2) Returned path does not contain G at head:

$$v_o \neq G$$

3) Some transition in the returned path is not valid

$$\langle v_{i-1}, v_i \rangle \notin E \text{ for some } i \in \{1,2, \ldots, v_k\}$$

4) Returned path is not simple:

$$v_i = v_j \text{ for some } i, j \in \{0,1, \ldots, k\} \text{ with } i \neq j$$

Proof of Soundness

1) Returned path does not start with S: $v_k \neq S$

But this contradicts Lemma 1!

Lemma 1: If $\langle v_1, v_2, \dots, v_k \rangle$ is a path in the queue at any given time, then $v_k = S$.

Proof of Soundness

2) Returned path does not contain G at head: $v_o \neq G$

But, the returned path has the property that Head(N) = G

Recall Lines 2-3 of the psuedocode

Simple Search Algorithm

Invariant: If $\langle v_1, v_2, \dots, v_k \rangle$ is a path in the queue at any given time, then it is simple path.

1. Initialize Q with a partial path <S>; set Visited = {}
2. **If Q is empty, fail. Else, pick a partial path N from Q**
3. **If head(N) = G, return N   //Goal reach!**
4. Else:
   a) Remove N from Q
   b) Find all children of head(N) not in Visited and create a on-step extension of N to each child
   c) Add all extended paths to Q
   d) Add Children of head(N) to Visited
   e) GOTO step 2

Proof of Soundness

3) Some transition in the returned path is not valid:

$\langle v_{i-1}, v_i \rangle \notin E$ for some $i \in \{1,2,\ldots, v_k\}$

Contradicts Lemma 2!

Lemma 2: If $\langle v_1, v_2, \ldots, v_k \rangle$ is a path in the queue at any given time, then it is valid.

Proof of Soundness

4) Returned path is not simple:

$v_i = v_j$ for some $i, j \in \{0,1, \dots, k\}$ with $i \neq j$

Contradicts Lemma 3!

Lemma 2: If $\langle v_1, v_2, \dots, v_k \rangle$ is a path in the queue at any given time, then it is a simple path (contains no cycles)

Proof of Soundness

Assume that the search algorithm is ***not* sound:**

Let the returned path be $\langle v_1, v_2, \dots, v_k \rangle$

Then, one of the following must be TRUE:

1) Returned path does not start with S:

$$v_k \neq S$$

2) Returned path does not contain G at head:

$$v_o \neq G$$

3) Some transition in the returned path is not valid

$\langle v_{i-1}, v_i \rangle \notin E$ for some $i \in \{1,2, \dots, v_k\}$

4) Returned path is not simple:

$v_i = v_j$ for some $i, j \in \{0,1, \dots, k\}$ with $i \neq j$

Proof of Soundness

Assume that the search algorithm is ***not*** **sound:**

We reach a contradiction in all cases.

Hence, the simple search algorithm is ***sound.***

Soundness & Completeness Theorems

Theorem 2 (Completeness):

- Simple search algorithm is complete.

Need to prove:

- If there is a path to reach from S to G, then the algorithm returns one path that does so.

Soundness & Completeness Theorems

Theorem 1 (Completeness):

- Simple search algorithm is complete.

Need to prove:

- If there is a path to reach from S to G, then the algorithm returns one path that does so.

Simple Search Algorithm

1. **Initialize Q with a partial path <S>; set Visited = {}**
2. **If Q is empty, fail. Else, pick a partial path N from Q**
3. **If head(N) = G, return N   //Goal reach!**
4. **Else:**
   a) **Remove N from Q**
   b) **Find all children of head(N) not in Visited and create a on-step extension of N to each child**
   c) **Add all extended paths to Q**
   d) **Add Children of head(N) to Visited**
   e) **GOTO step 2sume that invariant holds.**

A common technique in analysis of algorithms

- Let us slightly modify the algorithm

- We will analyze the modified algorithm

- Then “project’ our results to the original algorithm

Simple Search Algorithm

1. **Initialize Q with a partial path <S>; set Visited = {}**
2. **If Q is empty, fail. Else, pick a partial path N from Q**
3. **// If head(N) = G, return N //Goal reach!**
4. **Else:**
   a) **Remove N from Q**
   b) **Find all children of head(N) not in Visited and create a on-step extension of N to each child**
   c) **Add all extended paths to Q**
   d) **Add Children of head(N) to Visited**
   e) **GOTO step 2sume that invariant holds.**

Proof of Completeness

- The modified algorithm terminates when the queue is empty.

- Let us prove a few lemmas regarding the behavior of the modified algorithm

Proof of Completeness

- **Lemma 1**: A path that is taken out of the queue is not placed into the queue again at a later step.
- **Proof:** (using logical deduction)
  - *Another way to state this:* If $\mathrm{p} = \langle v_0, v_1, \ldots, v_k \rangle$ is a path that is taken out of the queue, then $\mathrm{p} = \langle v_0, v_1, \ldots, v_k \rangle$ is not placed in to the queue at later step.
  - Assume that $\mathrm{p} = \langle v_0, v_1, \ldots, v_k \rangle$ is taken out of the queue.
  - Then, p must be placed in to the queue at an earlier step.
  - Then, v_o must be in the visited list at this step.
  - Then, $\mathrm{p} = \langle v_0, v_1, \ldots, v_k \rangle$ can not be placed in to the queue at a later step, since v_o is in the visted list.

Proof of Completeness

Definition: A vertex v is *reachable* from S if there exists a path $\langle v_0, v_1, \ldots, v_k \rangle$ that starts from S and ends at v, i.e., $v_k = S$ and $v_o = v$.

Lemma 2: If a vertex v is reachable from S, then v is placed in to the visited list after a finite number of steps.

Proof of Completeness:

Lemma 2: If a vertex v is reachable from S, then v is placed in to the visited list after a finite number of steps.

Proof: *(by contradiction)*

- Assume v is reachable from S but is never placed on the visited list
- Since v is reachable from S, there exists a path that is of the form $\langle v_0, v_1, \ldots, v_k \rangle$, where $v_o = v$ and $v_k = S$
- Let v_i be the first node (starting from v_k) in the chain that is never added to the visited list.
- Note 1) v_i was not in the visited list before this list
- Note 2) $\langle v_{i+1}, v_i \rangle \in E$

Proof of Completeness:

Proof: *(by contradiction)*

- …
- Since v_{i+1} was in the visited list, the queue included a path $\langle v_{i+1}, v_{i+2}, \dots, v_k \rangle$ (not necessarily the same as before), where $v_k = S$
- This path must have been popped from the queue, since there are only finitely many different partial paths and no path is added twice (by Lemma 1) and v_i was not in the visited list (see Note 1 above)
- Since it is popped from the queue, then $\langle v_0, v_1, \dots, v_k \rangle$ must be placed in to the queue (se Note 2) and v_i placed into the visited list

Proof of Completeness:

Lemma 2: If a vertex v is reachable from S, then v is placed in to the visited list after a finite number of steps.

Proof: *(by contradiction)*

- Assume v is reachable from S but is never placed on the visited list
- Since v is reachable from S, there exists a path that is of the form $\langle v_0, v_1, \ldots, v_k \rangle$, where $v_o = v$ and $v_k = S$
- Let v_i be the first node (starting from v_k) in the chain that is never added to the visited list.
- Note that v_i was not in the visited list before this list
- Note that $\langle v_{i+1}, v_i \rangle \in E$
- Note 1) v_i was not in the visited list before this list
- Note 2) $\langle v_{i+1}, v_i \rangle \in E$
- Since v_{i+1} was in the visited list, the queue included a path $\langle v_{i+1}, v_{i+2}, \ldots, v_k \rangle$ (not necessarily the same as before), where $v_k = S$
- This path must have been popped from the queue, since there are only finitely many different partial paths and no path is added twice (by Lemma 1) and v_i was not in the visited list (see Note 1 above)
- Since it is popped from the queue, then $\langle v_0, v_1, \ldots, v_k \rangle$ must be placed in to the queue (se Note 2) and v_i placed into the visited list

Red statements contradict!

Proof of Completeness

- **Lemma 2**: If vertex v is reachable from S, then v is placed in to the visited list after a finite number of steps.

- **Corollary:** In the modified algorithm, G is placed into visited queue

- *"Project" back to the original algorithm:*
  - This is exactly when the original algorithm terminates

Proof of Completeness

Theorem 2 (Completeness):

Simple search algorithm is complete.

- **Proof:** Follows from Lemma 2 evaluated in the original algorithm.

Simple Search Algorithm

1. **Initialize Q with a partial path <S>; set Visited = {}**
2. **If Q is empty, fail. Else, pick a partial path N from Q**
3. **If head(N) = G, return N //Goal reach!**
4. **Else:**
   a) **Remove N from Q**
   b) **Find all children of head(N) not in Visited and create a on-step extension of N to each child**
   c) **Add all extended paths to Q**
   d) **Add Children of head(N) to Visited**
   e) **GOTO step 2sume that invariant holds.**

Summarize Completeness and Soundness

Hence, we have proven two theorems:

Theorem 1 (Soundness):

Simple search algorithm is sound.

Theorem 2 (Completeness):

Simple search algorithm is complete.

Soundness and completeness is a requirement for most algorithms, although we will use their relaxations often

Back to the Axiomatic Method

Does it really work?

- Essentially all of what we know in mathematics today can be derived from a handful of axioms called the Zarmelo-Frankel set theory with the axiom of Choice (ZFC)

- These axioms were made up by Zarmelo

  (they did not exist *a priori*, unlike physical phenomena)

- We do not know whether these axioms are logically conistent.
  - Sounds crazy! But, happened before…

    Around 1900, B. Russel discovered that he axioms of that time were logically inconsistent, i.e., one could prove a contradiction.

Back to the Axiomatic Method

- ZFC axioms gives one what she/he wants:
  - **Theorem**: 5 + 5 = 10

- However, absurd statements can also be derived:
  - **Theorem** (Banach-Tarski): A ball can be cut into a finite number of pieces and then the pieces can be rearranged to build two balls of the same size of the original.

Clearly, this contradicts our geometric intuition.

Back to the Axiomatic Method

On the fundamental limits of mathematics:

- Godel showed in 1930 that there are some propositions that are true, but do not logically follow from the axioms.
- The axioms are not enough!
- But, Godel also showed that simply adding more axioms does not eliminate the problem. Any set of axioms that is not contradictory will have the same problem!
- Godel’s results are directly related to computation. These results were later used by **Alan Turing** in the 1950s to invent a revolutionary idea: a ***computer***…

Elements of Algorithmic Design

Description: (Last Wednesday)

- Problem statement
- Stylized pseudo code, sufficient to analyze and implement the algorithm

Analysis:

- Performance:
  - Time complexity
    - How long does it take to find a solution?
  - Space complexity:
    - How much memory does it need to perform search?
- Correctness:
  - Soundness:
    - When a solution is returned, is it guaranteed to be correct?
  - Completeness:
    - Is the algorithm guaranteed to find a solution when there is one?