Artificial Intelligence

AI2002

Rushda Muneer

Spring 2026

Searching for Solutions

- The possible action sequences starting at the initial state form a **search tree** with the initial state at the root;
- The **branches** are actions and the **nodes** correspond to states in the state space of the problem.
- The first step is to **test** whether the root is a goal state.
- Then we **expand** the current state; that is, applying each legal action to the current state, thereby **generating** a new set of states.
- Search algorithms all share this basic structure; they vary primarily according to how they choose which state to expand next—the so-called **search strategy**.

Infrastructure for search algorithms

- For each node n of the tree, we have a structure that contains four components:
- **n.STATE:** the state in the state space to which the node corresponds;
- **n.PARENT:** the node in the search tree that generated this node;
- **n.ACTION:** the action that was applied to the parent to generate the node;
- **n.PATH-COST:** the cost, traditionally denoted by g(n), of the path from the initial state to the node, as indicated by the parent pointers.

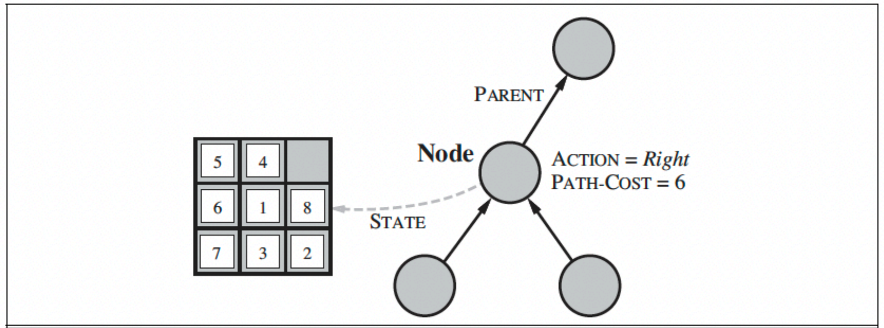


Figure 3.10 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

Infrastructure for search algorithms

- **The frontier needs to be stored** in such a way that the **search algorith**m can easily **choose the next node to expand** according to its preferred strategy. The appropriate data structure for this is a queue. The operations on a queue are as follows:
- **EMPTY?(queue)** returns true only if there are no more elements in the queue.
- **POP(queue)** removes the first element of the queue and returns it.
- **INSERT(element, queue)** inserts an element and returns the resulting queue.
- Three common variants are:
- The first-in, first-out or **FIFO** queue, which pops the oldest element of the queue;
- The last-in, first-out or **LIFO** queue (also known as a stack), which pops the newest element of the queue;
- The **priority** queue, which pops the element of the queue with the highest priority according to some ordering function.

Measuring problem-solving performance

- We can evaluate an algorithm’s performance in four ways:
- **Completeness:** Is the algorithm guaranteed to find a solution when there is one?
- **Optimality:** Does the strategy find the optimal solution?
- **Time complexity:** How long does it take to find a solution?
- **Space complexity:** How much memory is needed to perform the search?

Strategies for State Space Search

- A state space may be searched **in two directions**: from the **given data** of a problem instance **toward** a **goal** or from a **goal** back to the **data**.
- **Data Driven Search** (forward chaining)
  - The problem solver begins with the **given facts of the problem** and a **set of legal moves or rules** for **changing state**.
  - **Search** continues by **applying rules to facts** to **produce new facts**, which in turn are used by the **rules to generate more new facts**.
  - The process continues until (we hope!) it generates a **path** that satisfies the **goal condition**.

Strategies for State Space Search

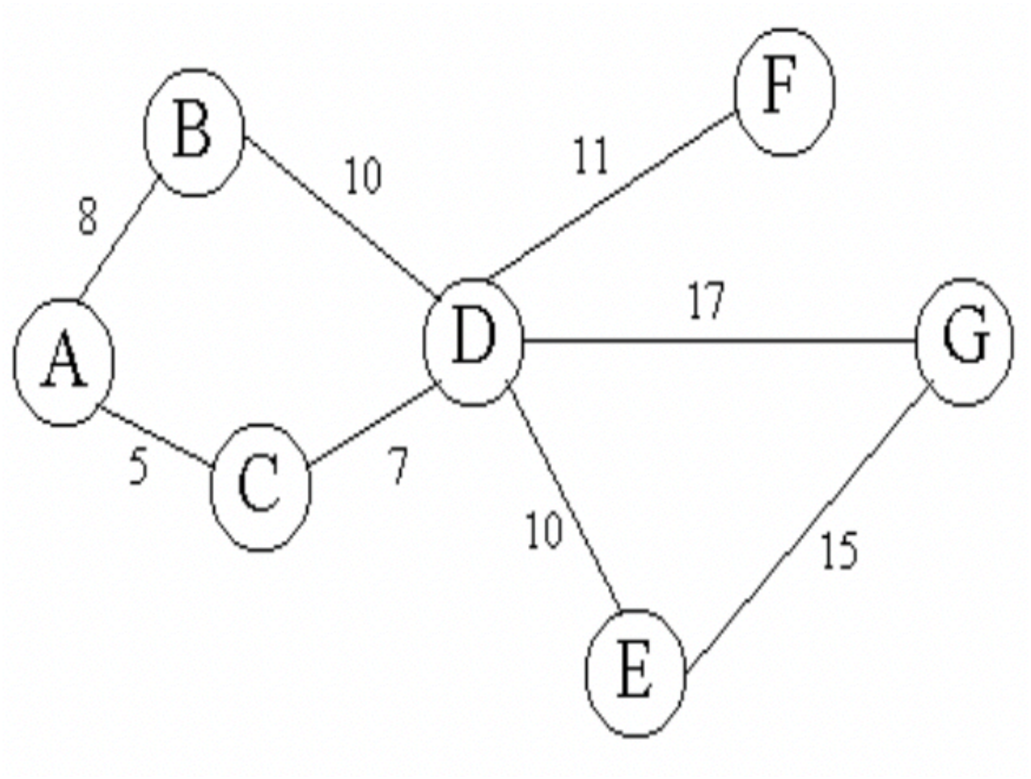
- **Goal Driven Search** (backward chaining)
  - Take the **goal** we want to reach.
  - See what **rules or legal moves** could be used to generate this goal and determine **what conditions must be true** to use them.
  - These conditions become new “**goals**” or “**sub-goals**” for the search.
  - Search continues, working **backwards** through **successive sub-goals** until (again we hope!) it works it way back to the **given data** of the problem.
- Both searches, **search the same state space graph**, but the **order** and **actual number of states** searched can **differ**.
- The **preferred strategy** is determined by the **properties** of the problem e.g. **complexity** of the rules, the **shape** of the state space and, the **nature** and **availability** of problem data.

Blind Search Methods (Uninformed)

- **Blind search methods** are the methods that make **no use of any information** that may be available about **where the goal** is in the search space.
- They **blindly search through the tree** until they stumble across a **solution**.
- If they **stop at this solution** it may **not be the optimum solution** if there are **several possible answers**.
- The search can be shown as a **tree**, as it searches from **one state to another**.

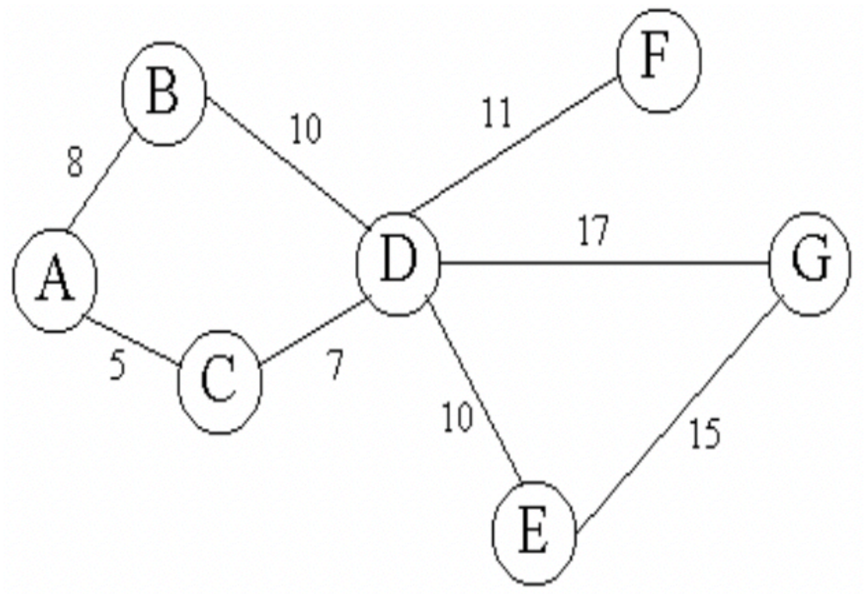
Search Methods

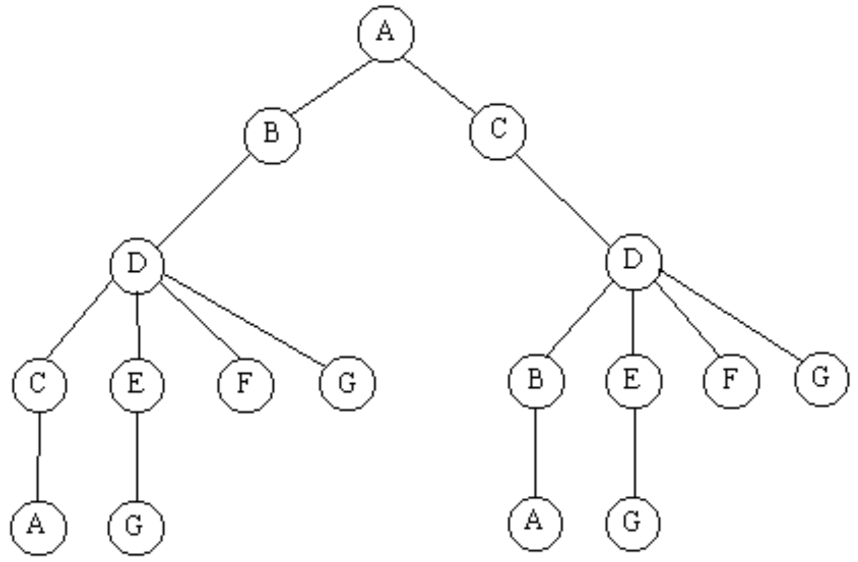
- To demonstrate some different methods of searching let us consider the following **example**.
- Given the map shown, how can we get from **A to G**?
- The **constraint** that each place may be **visited only once** will be added to **restrict the search tree to a finite size**.
- We shall also place the **restriction** that **you may not go immediately back** to the node you came from, e.g.. from **A to B** and back to **A**.



Tree Representation

- The algorithm starts at an **initial state** and works its way **down the tree** until it **finds a node** that satisfies the problem.
- The way a method navigates the tree differentiates it from other methods.



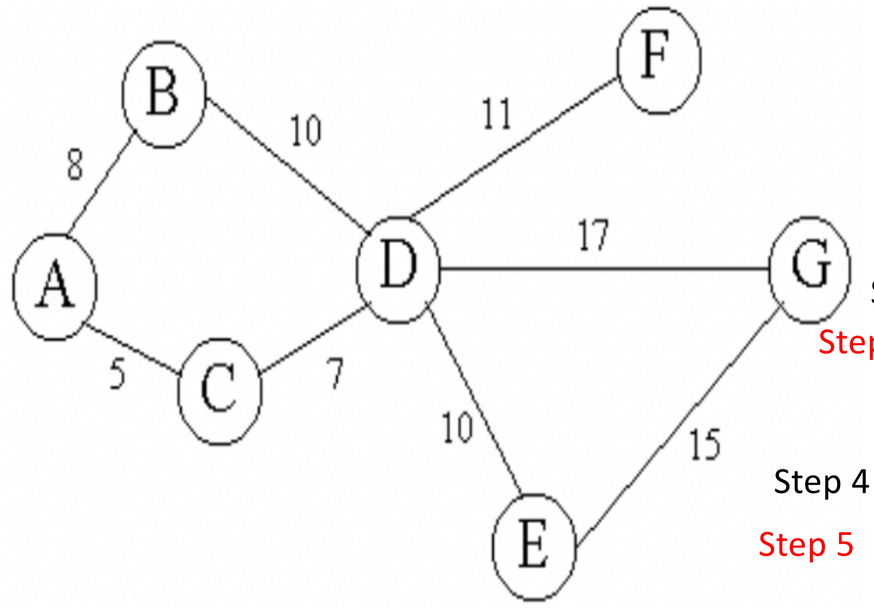


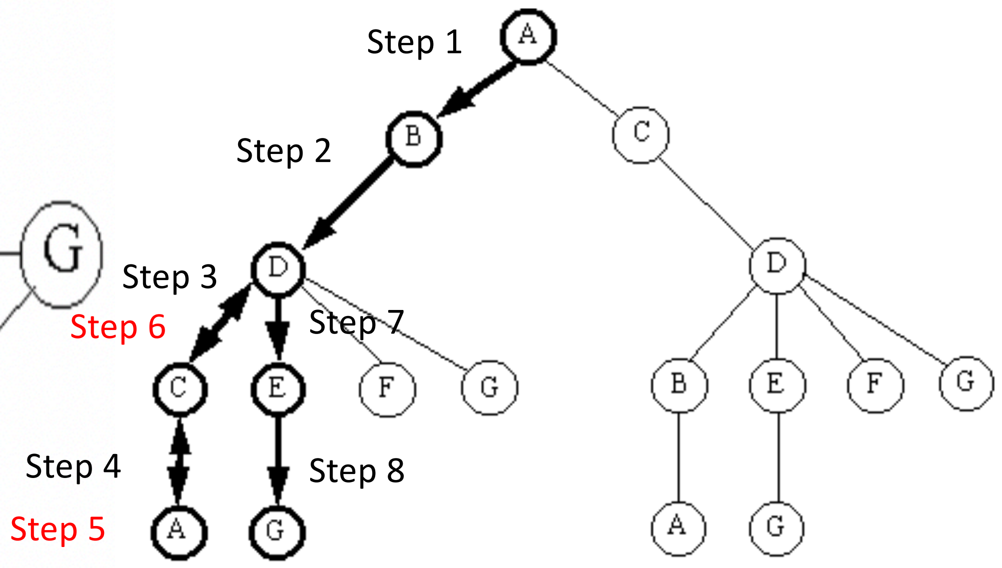
Depth-First Search

- The **most obvious way** to look at the problem is to **start at one node** and **go down the branches** in an **ordered** fashion until a **solution** is found.
- We can define **a simple search algorithm** to go **down** the tree, always taking the **leftmost** branch until the **bottom** is hit.
- If this is the **goal then stop**, if not then **go back up the tree** until a **junction** is found, then take the **next leftmost branch** that **hasn't** already been **traversed**
- If there is **no unused branch**, climb back up another junction.
- Keep on with this strategy until **either a solution is found** or until the **tree is exhausted.**

Depth-First Search

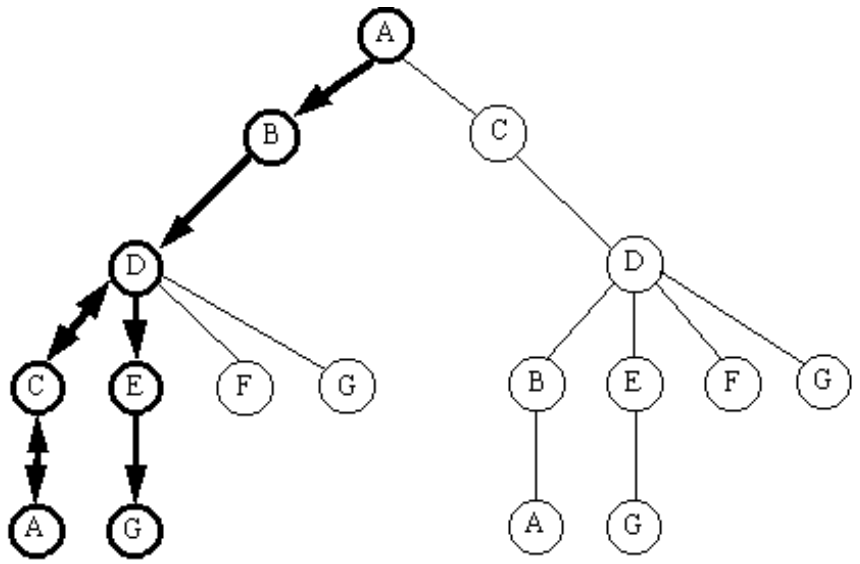
- As you can see, the search goes down the **left-hand side** and gets to **A at the end of one route**, so it backtracks and **finds G** and stops.
- The process took **8 steps** to get there if each step back up is counted.





DFS Algorithm (LIFO)

```
begin
    Open = [Start];
    Closed = [];
    While open <> [] do
    begin
        remove leftmost state from open, call it X;
        if X is a goal then return(success)
        else begin
            generate children of X;
            put X on closed;
            discard children of X if already on open or closed;
            put remaining children on left end of open
        end
    end
    return(failure)
end
```

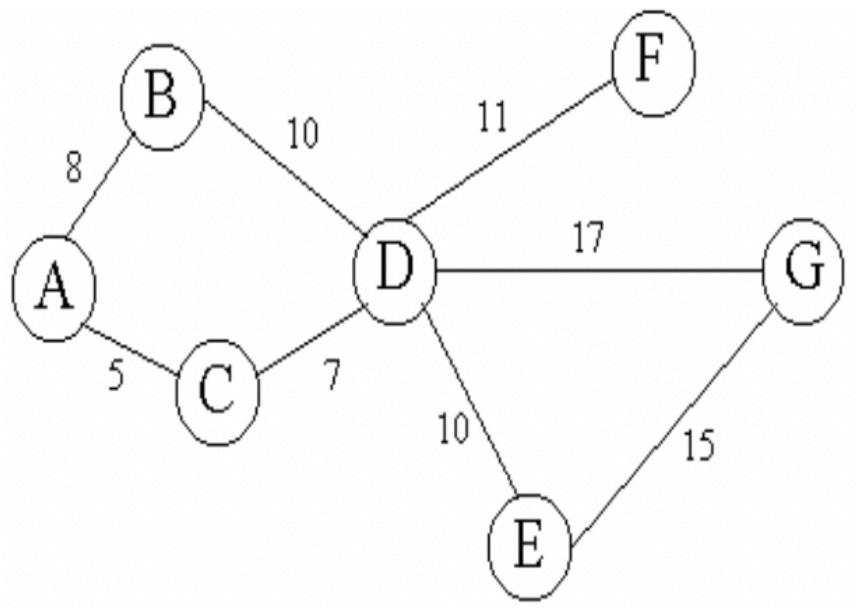


1. Open = [A]; closed = [ ];
2. Open = [B,C]; closed = [A];
3. Open = [D,C]; closed = [B,A];
4. Open = [E,F,G,C]; closed = [D,B,A];
5. Open = [F,G,C]; closed = [E,D,B,A];
6. Open = [G,C]; closed = [F,E,D,B,A];

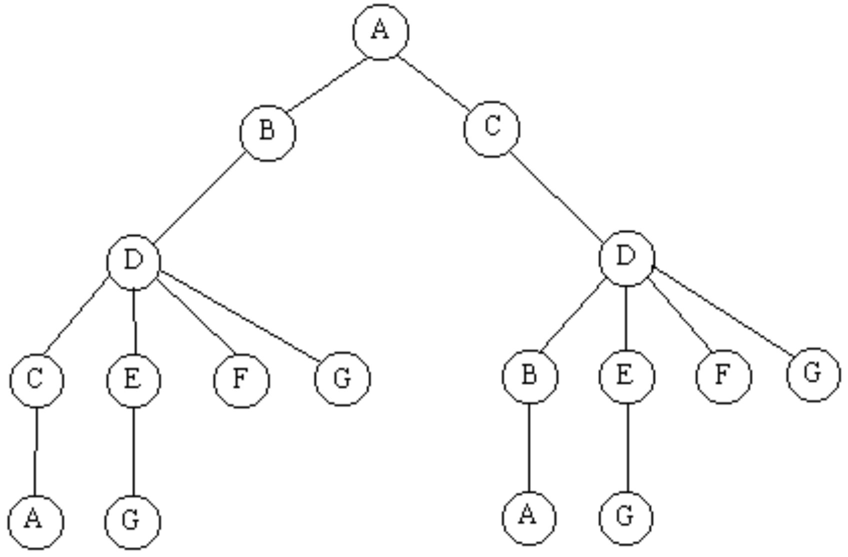
Breadth-First Search

- An **alternative approach** to finding our solution is to look at **one level of the tree** at a time.
- We look at **one level of nodes** and see if there is a **solution** to the problem, if there is we exit.
- If there **isn't a solution** we go down to the **next level of nodes** and try these.

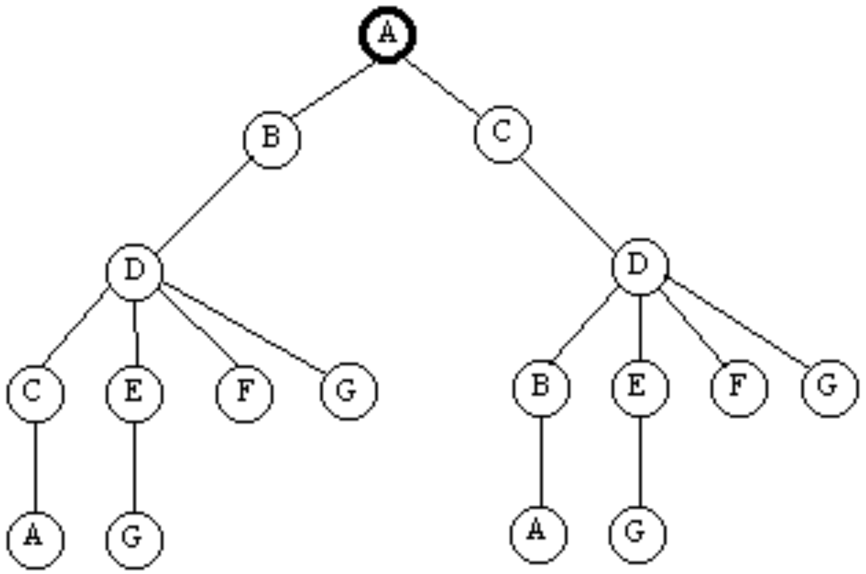
Search Space



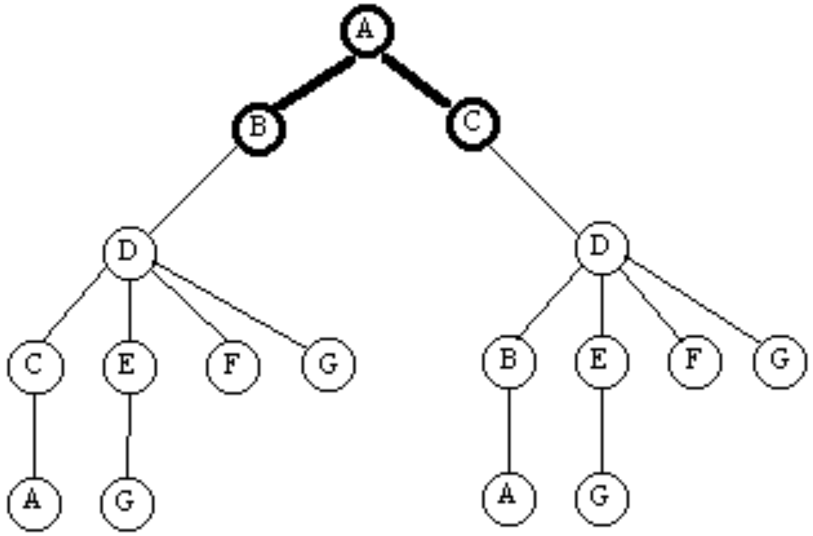
Tree Representation



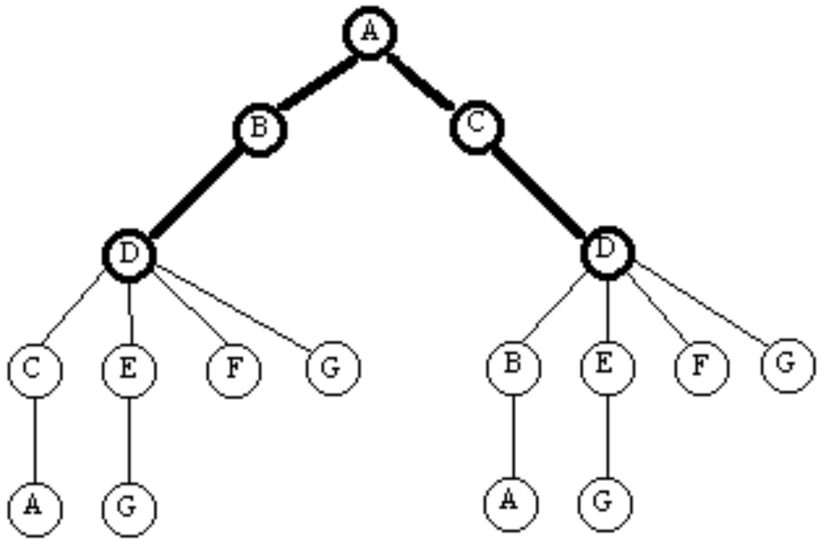
Level 0



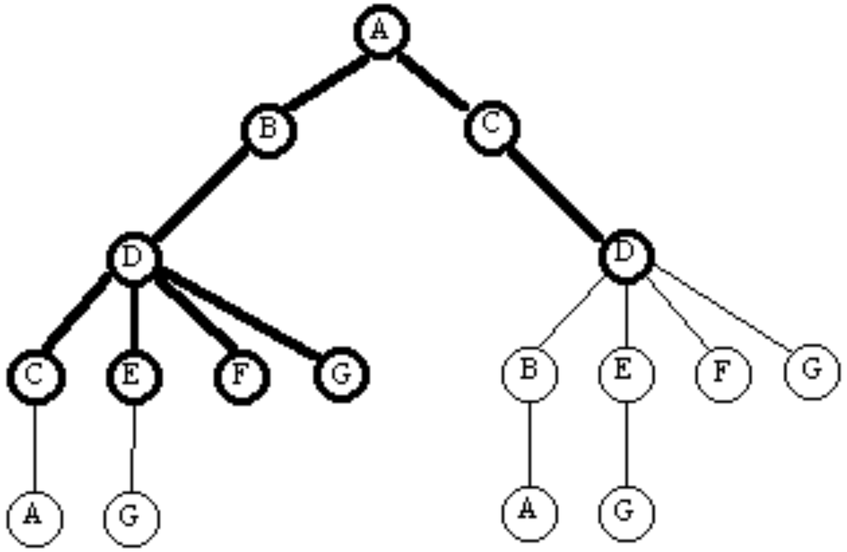
Level 1



Level 2



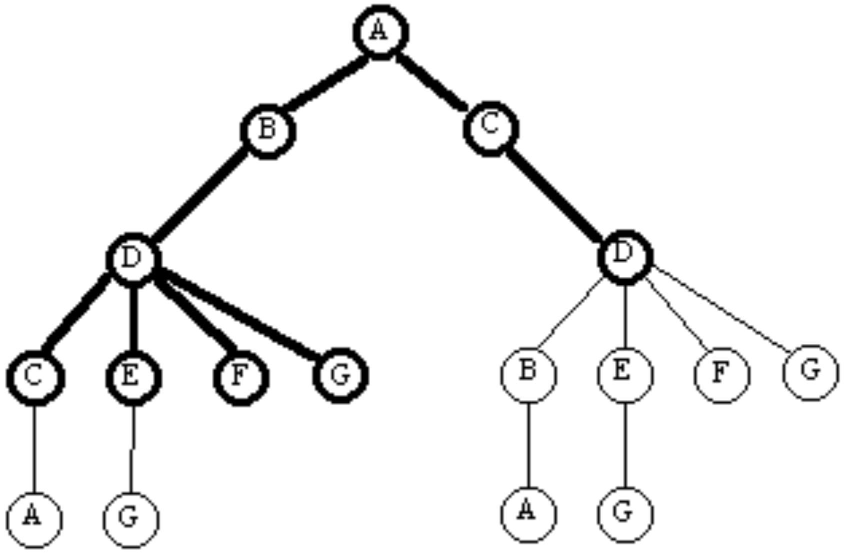
Level 3



solution is found, after examining 9 nodes in total

BFS Algorithm (FIFO)

```
• begin
    • Open = [Start];
    • Closed = [];
    • While open <> [] do
    • begin
        • remove leftmost state from open, call it X;
        • if X is a goal then return(success)
        • else begin
            • generate children of X;
            • put X on closed;
            • discard children of X if already on open or closed;
            • put remaining children on right end of open
        • end
    • end
    • return(failure)
• end
```



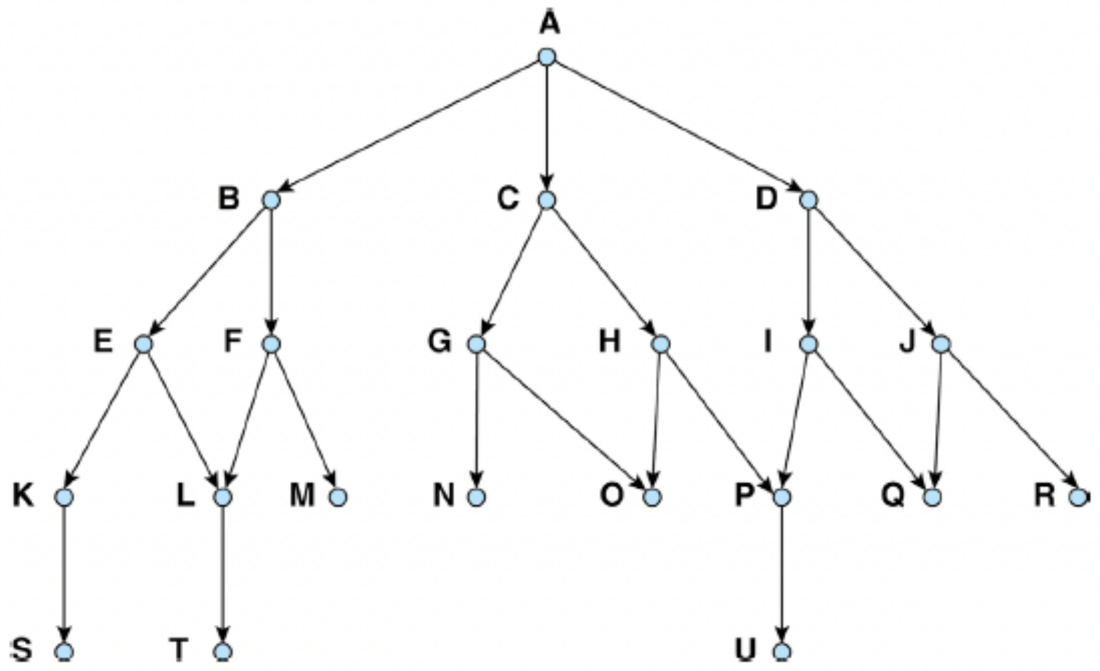
1. Open = [A]; closed = [ ];
2. Open = [B,C]; closed = [A];
3. Open = [C,D]; closed = [B,A];
4. Open = [D]; closed = [C,B,A];
5. Open = [E,F,G]; closed = [D,C,B,A];
6. Open = [F,G]; closed = [E,D,C,B,A];
7. Open = [G]; closed = [F,E,D,C,B,A];

Comparison of DFS and BFS

- Consider the case, though, of **removing the constraints** we placed on the **problem**; we are faced **with some branches that go on to infinity.**
- If a **depth-first search fell-down one of these branches**, which in this problem it certainly would, we would **never get an answer**.
- **Breadth-first** will **always find a solution** if there is **one**, even if **some of the branches are unending**, in fact it will **always find the solution nearest to the start level**.

Activity

- Perform DFS and BFS state space search on the given tree.
- Suppose the state space is to be searched, starting at node “A” to goal “G”.
- List the nodes of the state space in the order that they are expanded for each of the methods and the order that children of each expanded node are added to the queue.
- Assume that the search considers child nodes from left to right.



Algorithms

DFS

```
• begin
    • Open = [Start];
    • Closed = [];
    • While open <> [] do
    • begin
        • remove leftmost state from open, call it X;
        • if X is a goal then return(success)
        • else begin
            • generate children of X;
            • put X on closed;
            • discard children of X if already on open or
              closed;
            • put remaining children on left end of open
        • end
    • end
    • return(failure)
• end
```

BFS

```
• begin
    • Open = [Start];
    • Closed = [];
    • While open <> [] do
    • begin
        • remove leftmost state from open, call it X;
        • if X is a goal then return(success)
        • else begin
            • generate children of X;
            • put X on closed;
            • discard children of X if already on open or
              closed;
            • put remaining children on right end of open
        • end
    • end
    • return(failure)
• end
```

Homework Reading

- AI a Modern Approach Chapter 3 (3.4.3 , 3.4.1)
- AI G.F. Lugar Chapter 3 (3.2)