INFO 6205
Program Structure and Algorithms

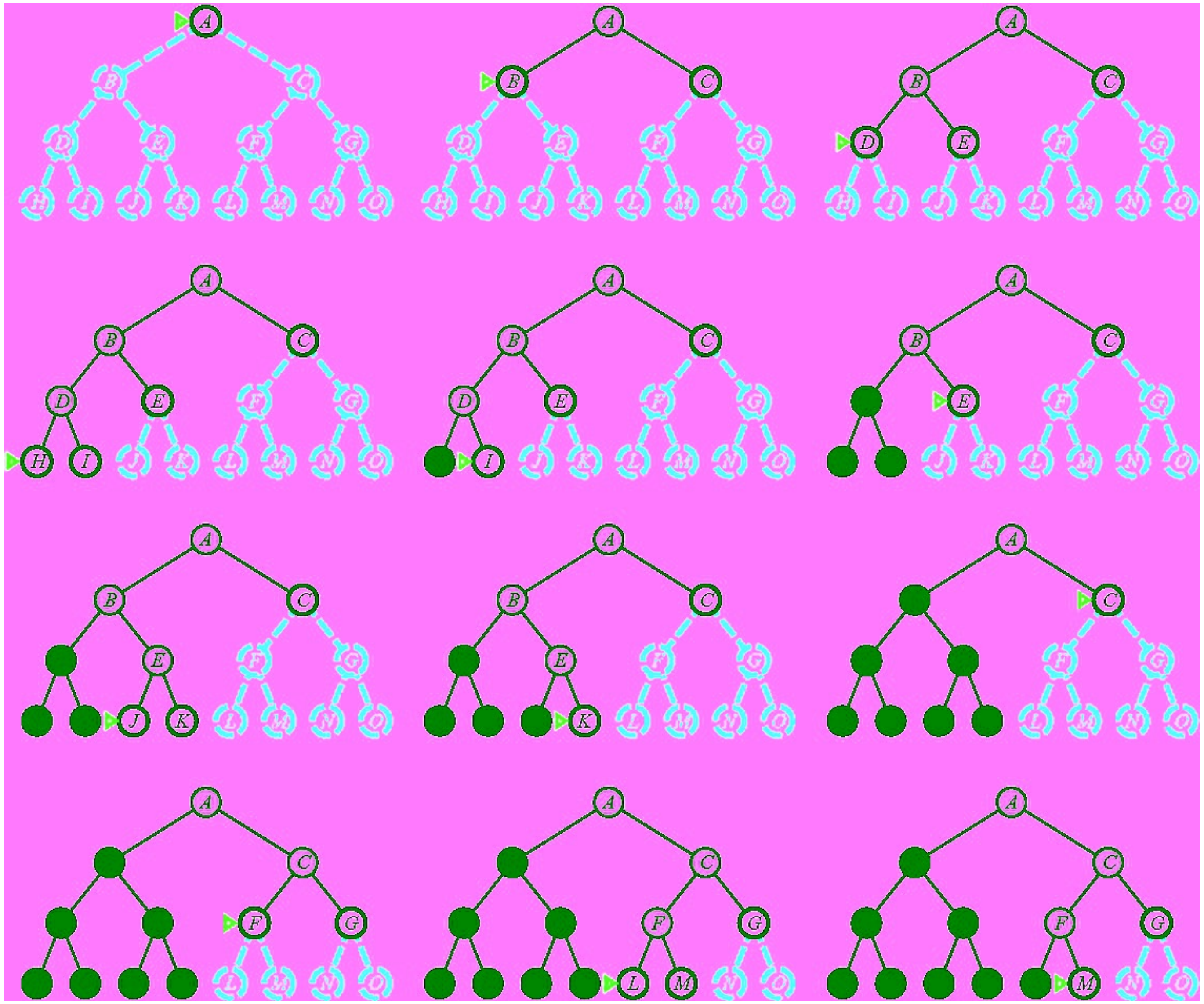
Depth-First Search

Nik Bear Brown

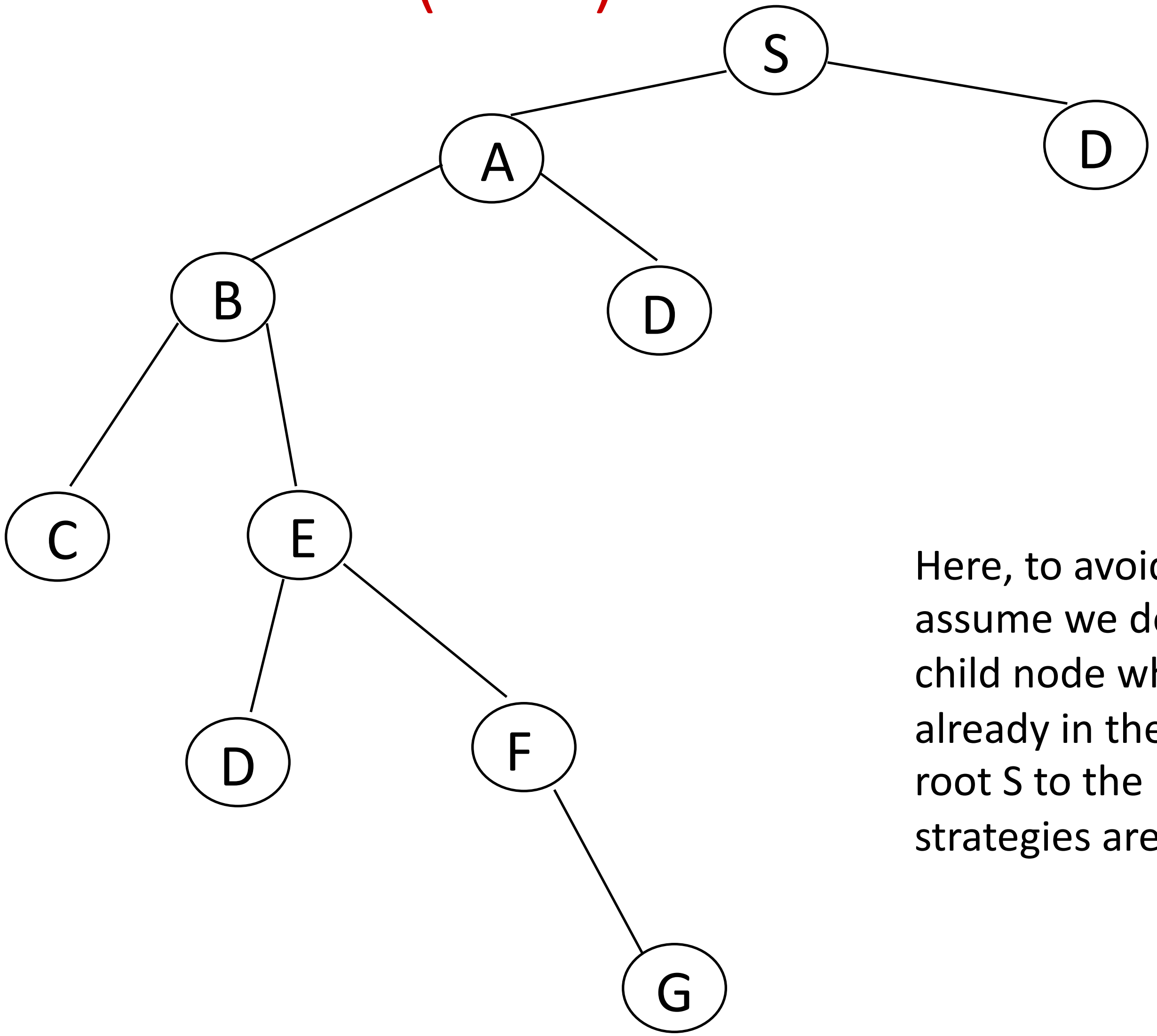
Topics

- Depth-First Search

Depth-First Search



Depth First Search (DFS)



Here, to avoid repeated states assume we don't expand any child node which appears already in the path from the root S to the parent. (Other strategies are also possible)

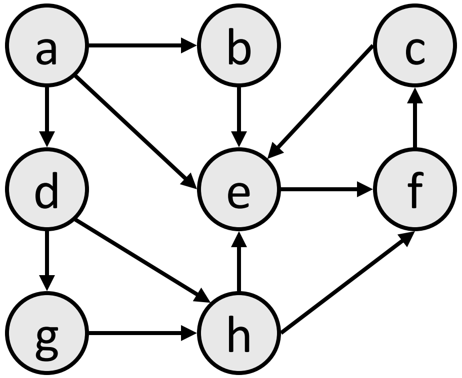
Pseudocode for Depth-First Search

```
Initialize: Let Q = {S}
While Q is not empty
        pull Q1, the first element in Q
        if Q1 is a goal
                report(success) and quit
        else
                child_nodes = expand(Q1)
                eliminate child_nodes which represent loops
                put remaining child_nodes at the front of Q
        end
Continue
```

Depth-first search

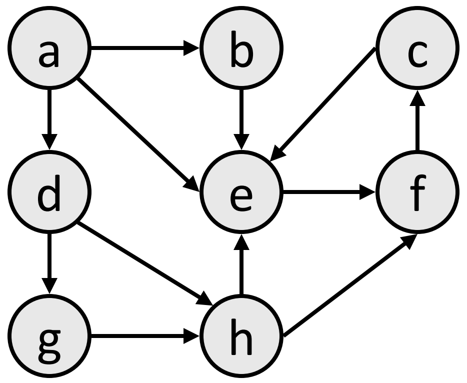
- **depth-first search** (DFS): Finds a path between two vertices by exploring each possible path as far as possible before backtracking.
  - Often implemented recursively.
  - Many graph algorithms involve *visiting* or *marking* vertices.

- Depth-first paths from *a* to all vertices (assuming ABC edge order):
  - to b: {a, b}
  - to c: {a, b, e, f, c}
  - to d: {a, d}
  - to e: {a, b, e}
  - to f: {a, b, e, f}
  - to g: {a, d, g}
  - to h: {a, d, g, h}



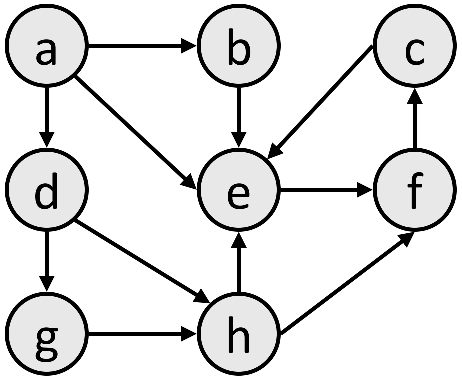
DFS pseudocode

```
function dfs(v1, v2):
    dfs(v1, v2, { }).

function dfs(v1, v2, path):
    path += v1.
    mark v1 as visited.
    if v1 is v2:
        a path is found!

    for each unvisited neighbor n of v1:
      if dfs(n, v2, path) finds a path: a path is found!

    path -= v1.   // path is not found.
```



- The *path* param above is used if you want to have the path available as a list once you are done.
  - Trace dfs(*a*, *f*) in the above graph.

DFS observations



- *discovery*: DFS is guaranteed to find <u>*a*</u> path if one exists.
- *retrieval*: It is easy to retrieve exactly what the path is (the sequence of edges taken) if we find it
- *optimality*: not optimal. DFS is guaranteed to find <u>a</u> path, not necessarily the best/shortest path
  - Example: dfs(a, f) returns {a, d, c, f} rather than {a, d, f}.

Comparing DFS and BFS

- Same Time Complexity, unless…
  - say we have a search problem with
    - goals at some depth d
    - but paths without goals and which have infinite depth (i.e., loops in the search space)
  - in this case DFS never may never find a goal!
      - (it stays on an infinite (non-goal) path forever)
  - BFS does not have this problem
      - it will find the finite depth goals in time $O(b^d)$

- Practical considerations
  - if there are no infinite paths, and many possible goals in the search tree, DFS will work best
  - For large branching factors b, BFS may run out of memory
  - BFS is "safer" if we know there can be loops

Depth-Limited Search

- This is Depth-first Search with a cutoff on the maximum depth of any path
  - i.e., implement the usual DFS algorithm
  - when any path gets to be of length m, then do not expand this path any further and backup
  - this will systematically explore a search tree of depth m

- Properties of DLS
  - Time complexity = O(b^m), Space complexity = O(bm)
  - If goal state is within m steps from S:
    - DLS is complete
    - e.g., with N cities, we know that if there is a path to goal state G it can be of length N-1 at most
  - But usually we don't know where the goal is!
    - if goal state is more than m steps from S, DLS is incomplete!
    - => the big problem is how to choose the value of m

Iterative Deepening Search

- Basic Idea:
  - we can run DFS with a maximum depth constraint, m
    - i.e., DFS algorithm but it **backs-up at depth m**
    - this avoids the problem of infinite paths
  - But how do we choose m in practice? say m < d (!!)
  - We can run DFS multiple times, gradually increasing m
    - this is known as Iterative Deepening Search

Procedure

```
for m = 1 to infinity
        if (depth-first search with max-depth = m ) returns success
                then report (success) and quit
        else
                continue
end
```

Iterative Deepening Search

- Complexity
  - Space complexity = O(bd)
    - (since its like depth first search run different times)
  - Time Complexity
    - $1 + (1+b) + (1 + b+b^2) + ...(1 + b+....b^d)$
      $= O(b^d)$
      (i.e., the same as BFS or DFS in the the worst case)

    - The overhead in repeated searching of the same subtrees is small relative to the overall time
      - e.g., for b=10, only takes about 11% more time than DFS

- A useful practical method
  - combines
    - guarantee of finding a solution if one exists (as in BFS)
    - space efficiency, O(bd) of DFS

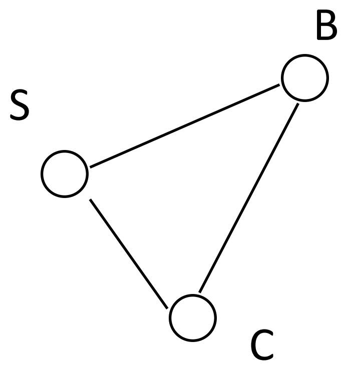
Bidirectional Search

- Idea
  - simultaneously search forward from S and backwards from G
  - stop when both “meet in the middle”
  - need to keep track of the intersection of 2 open sets of nodes

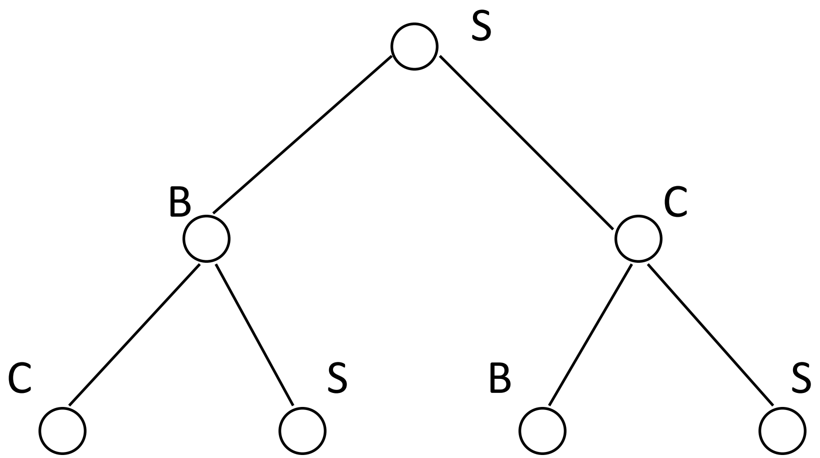
- What does searching backwards from G mean
  - need a way to specify the predecessors of G
    - this can be difficult,
    - e.g., predecessors of checkmate in chess?
  - what if there are multiple goal states?
  - what if there is only a goal test, no explicit list?

- Complexity
  - time complexity is $O(2\ b^{(d/2)}) = O(b^{(d/2)})$ steps
  - memory complexity is the same

Repeated States



State Space



Example of a Search Tree

- For many problems we can have repeated states in the search tree
  - i.e., the same state can be gotten to by different paths
  - => same state appears in multiple places in the tree
    - this is inefficient, we want to avoid it
- How inefficient can this be?
  - a problem with a finite number of states can have an infinite search tree!

Techniques for Avoiding Repeated States

- Method 1
  - when expanding, do not allow return to parent state
  - (but this will not avoid “triangle loops” for example)
- Method 2
  - do not create paths containing cycles (loops)
  - i.e., do not keep any child-node which is also an ancestor in the tree
- Method 3
  - never generate a state generated before
    - only method which is guaranteed to always avoid repeated states
    - must keep track of all possible states (uses a lot of memory)
    - e.g., 8-puzzle problem, we have 9! = 362,880 states
- Methods 1 and 2 are most practical, work well on most problems