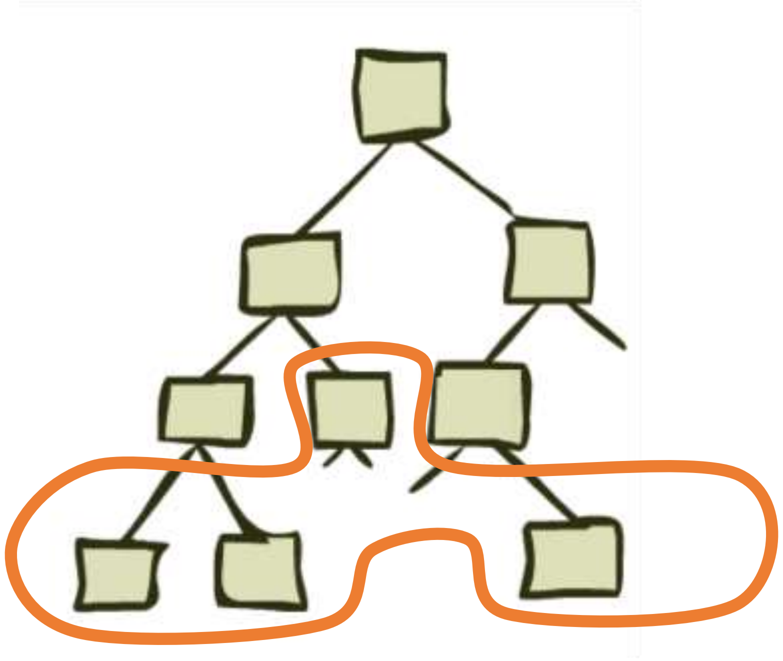
Uninformed Search - I

Breadth First Search (BFS) and Uniform Cost Search (UCS)



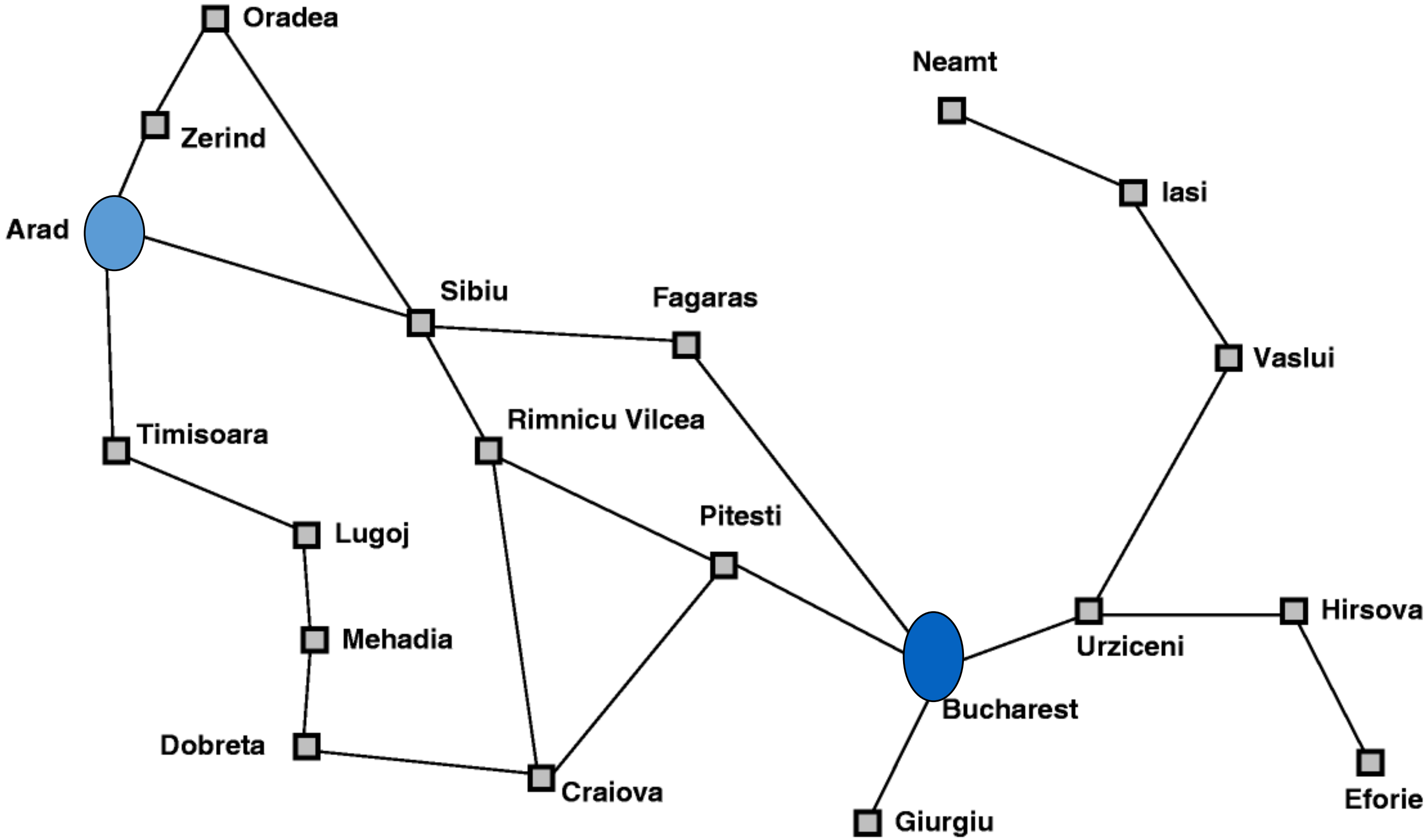
Previous Lecture

- Problem Formulation
- State Space / Search Space Representation
- Searching for Solution

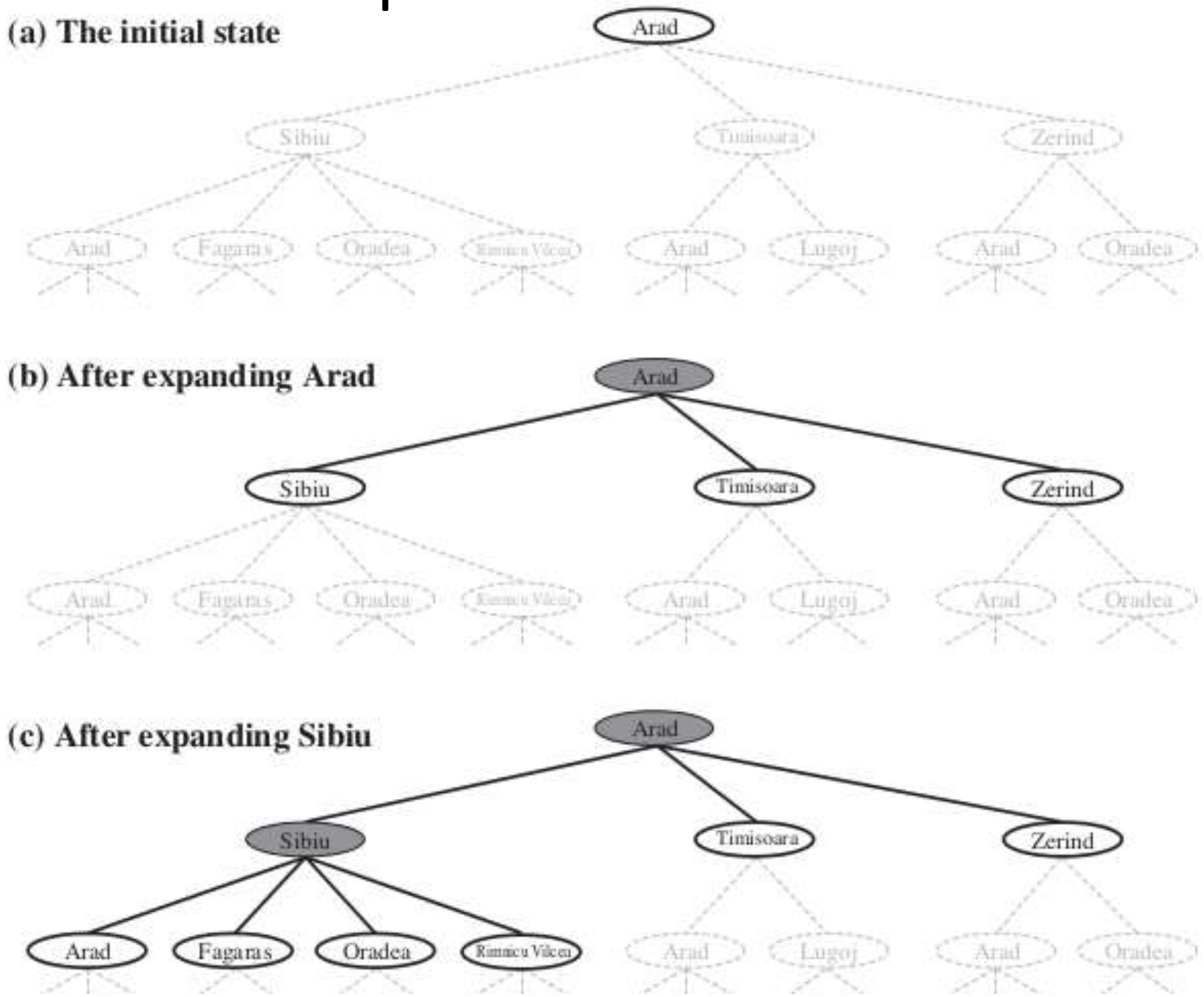
Today's Lecture

- Types of search Algorithms
- Uninformed Search vs. Informed Search
- Uniform Cost Search Algorithm
  - Breadth-first searching
  - Depth-first Search
  - Uniform Cost Search
- Algorithm Complexity

Map searching (navigation)

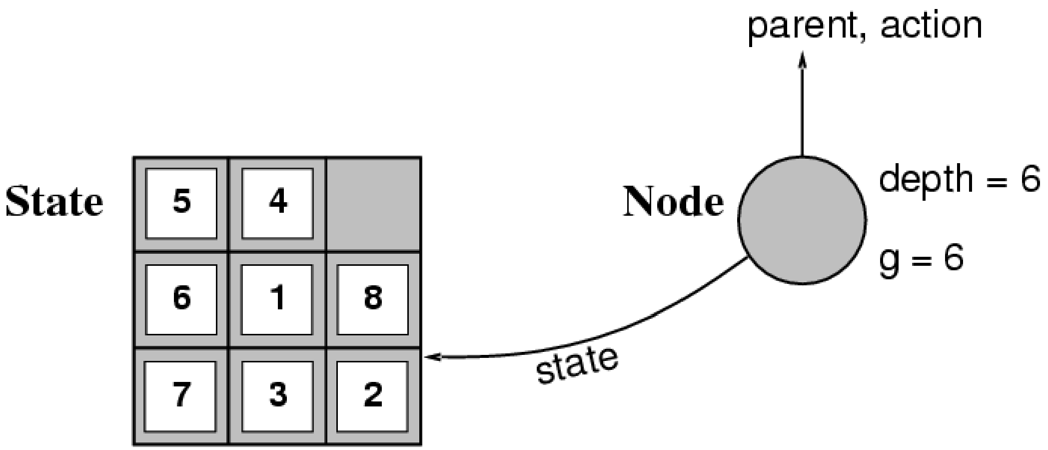


Tree search example



Implementation: STATES vs. NODES

- A state is a (representation of) a physical configuration
- A node is a data structure constituting part of a search tree includes state, parent node, action, path cost *g(x)*, depth



Searching for Solution

function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

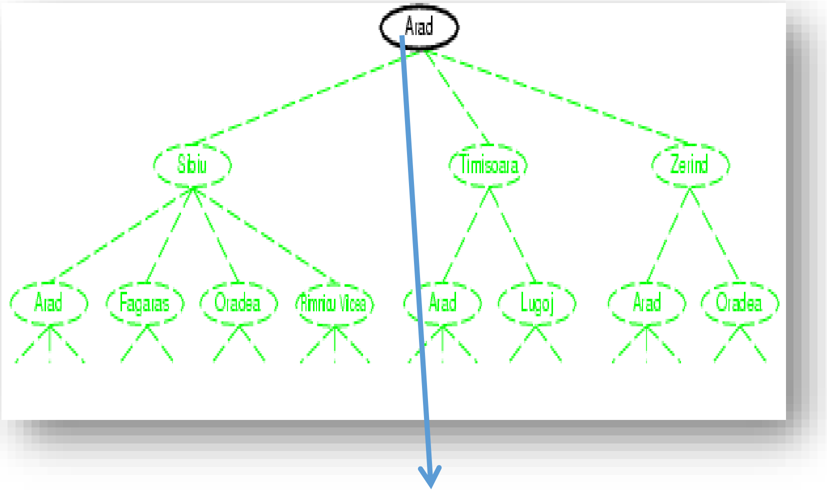
---

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set

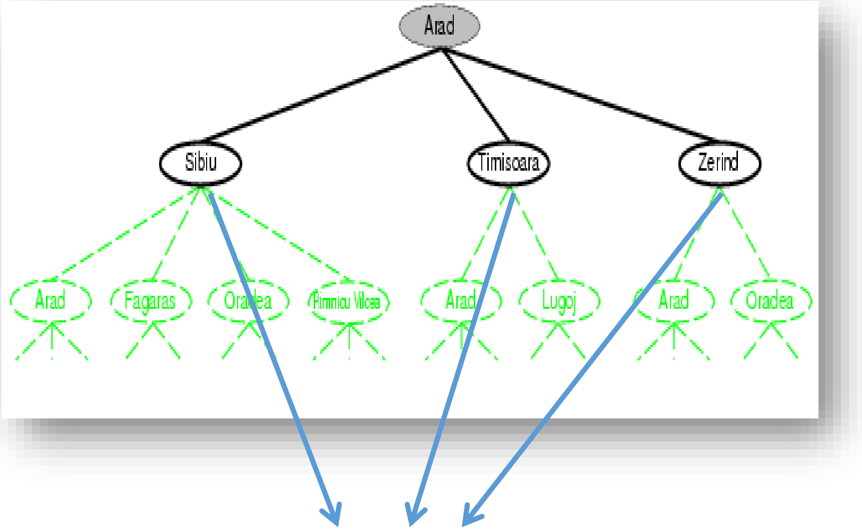
Figure 3.7 An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.

Fringe

- Fringe: The collection of nodes that have been **<u>generated but not yet expanded</u>**
- Each element of the fringe is a leaf node, with (currently) no successors in the tree
- The **search strategy** defines which element to choose from the fringe

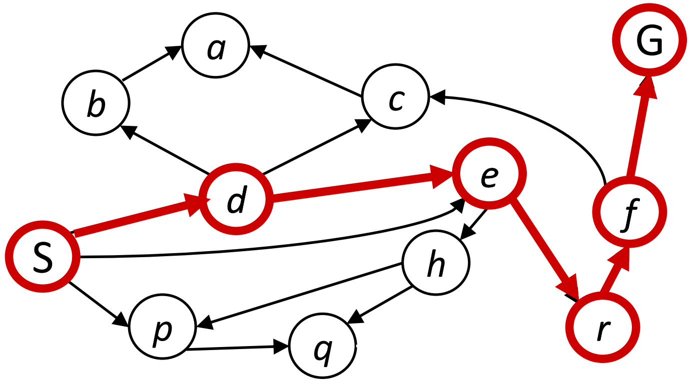


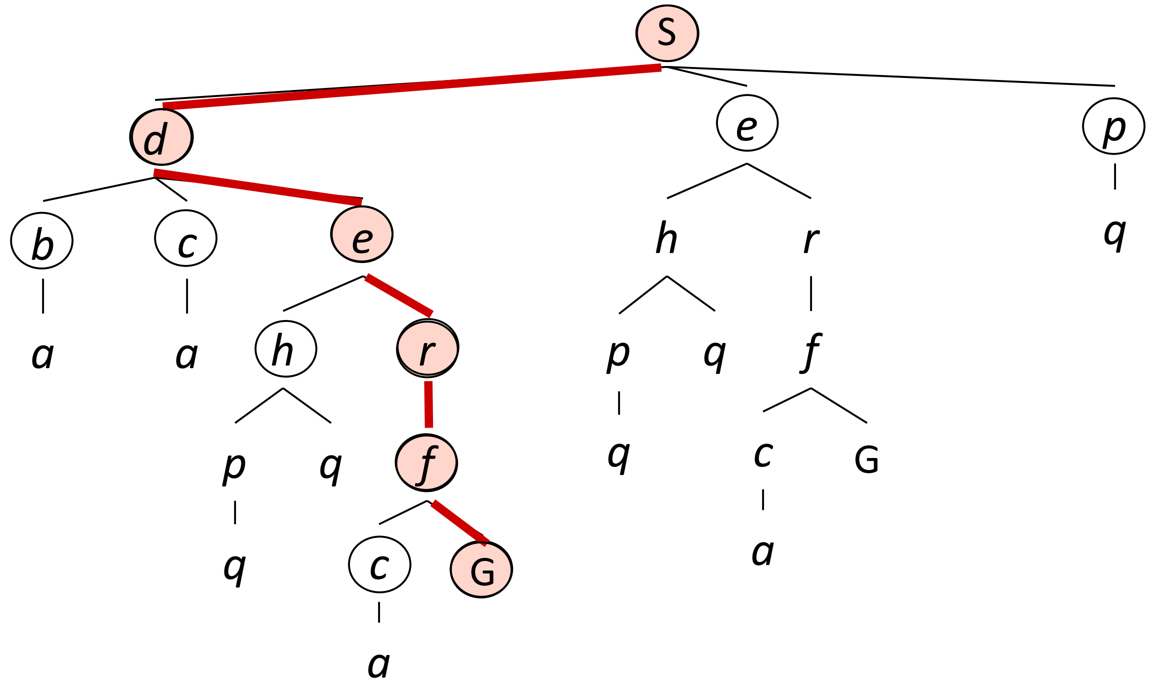
fringe



fringe

Example: Tree Search





~~s~~
~~s → d~~
s → e
s → p
s → d → b
s → d → c
~~s → d → e~~
s → d → e → h
~~s → d → e → r~~
~~s → d → e → r → f~~
s → d → e → r → f → c
~~s → d → e → r → f → G~~

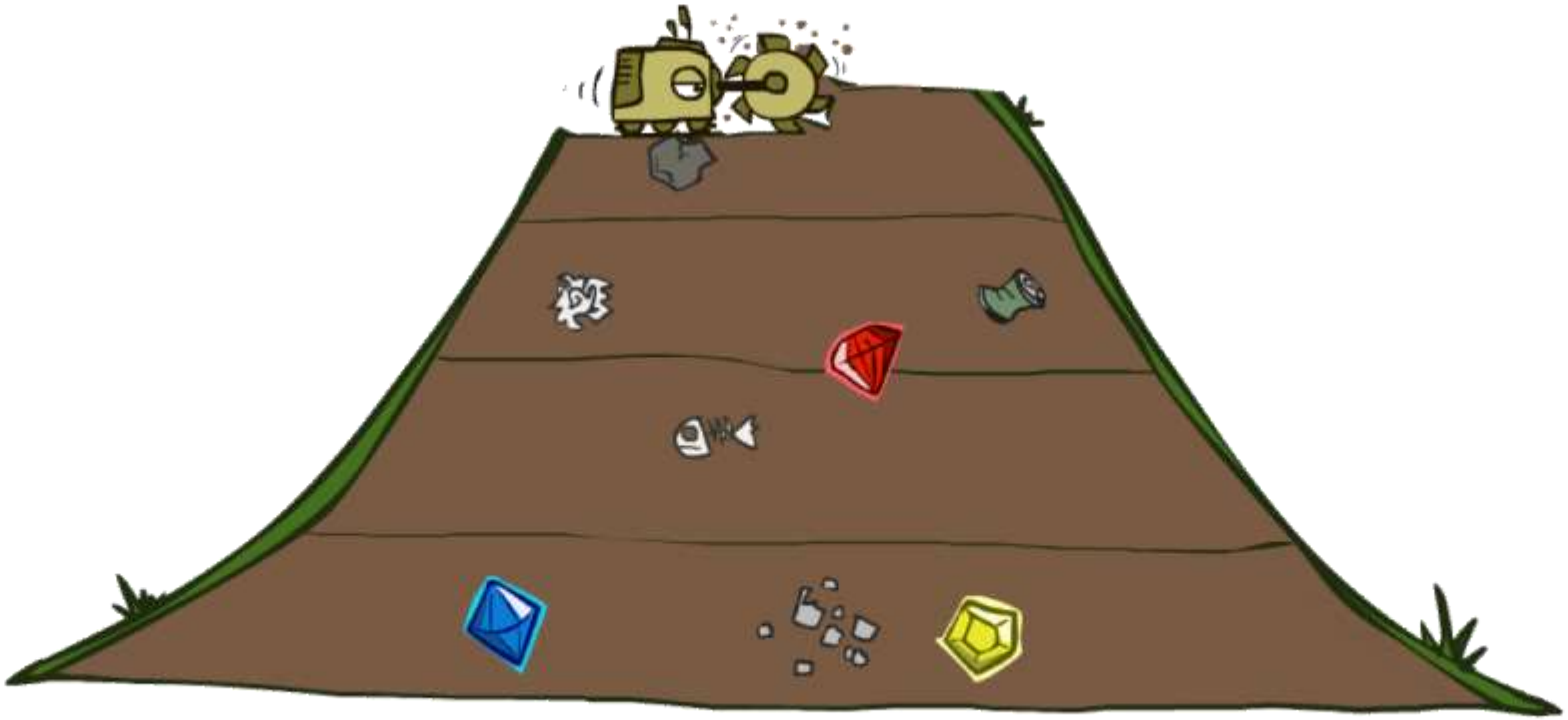
Search Strategy

All search strategies are distinguished by the Order in which nodes are expanded

- **Uninformed search** methods (Blind search) have access only to the problem definition.
  - No notion of the concept of the “right direction”.
  - Can only recognize goal once it’s achieved.

- **Informed search** methods may have access to a **heuristic** function that estimate the cost of a solution from n.
  - uses the given heuristic information to decide whether or not to explore the current state further.

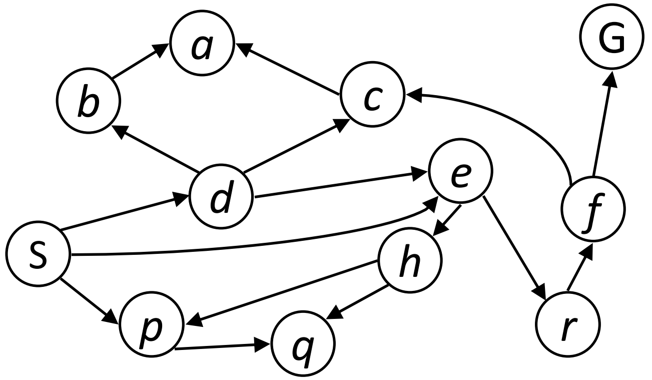
Breadth-First Search



Breadth-First Search

Strategy: expand a shallowest node first

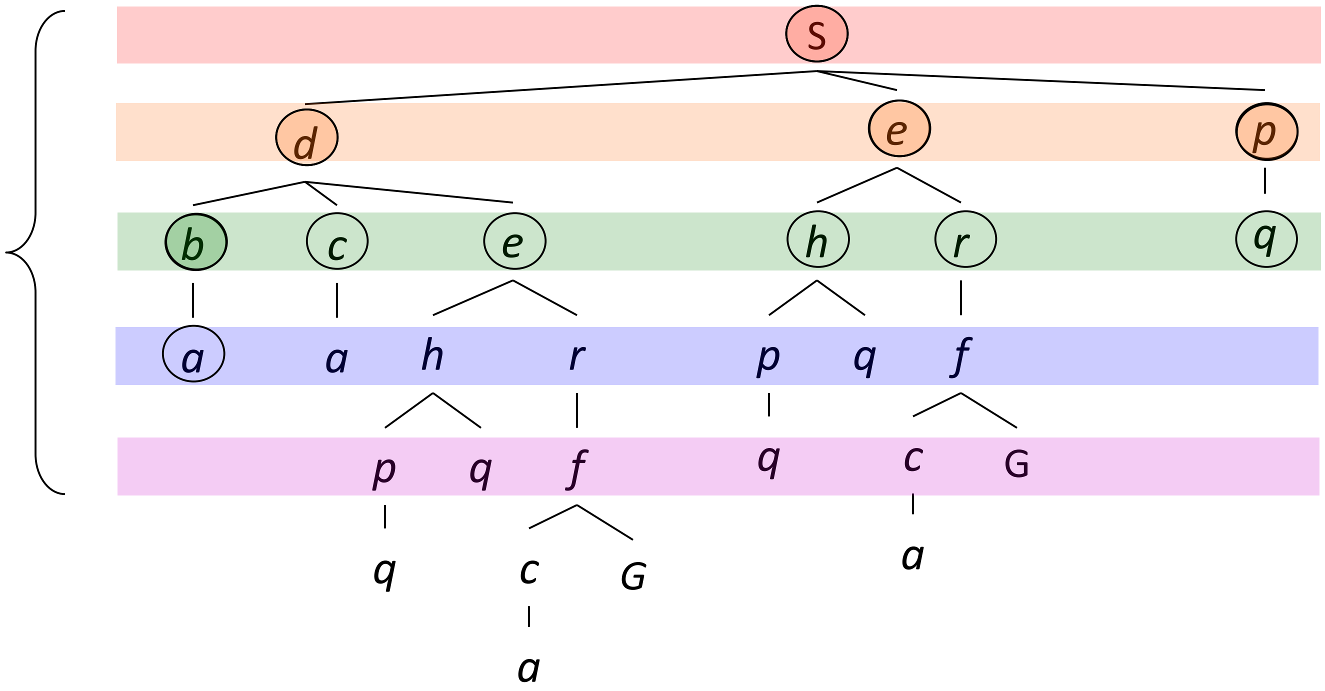
Implementation: Fringe is a FIFO queue



The root node is expanded first, then all the successors of the node are expanded next, then their successors, and so on.

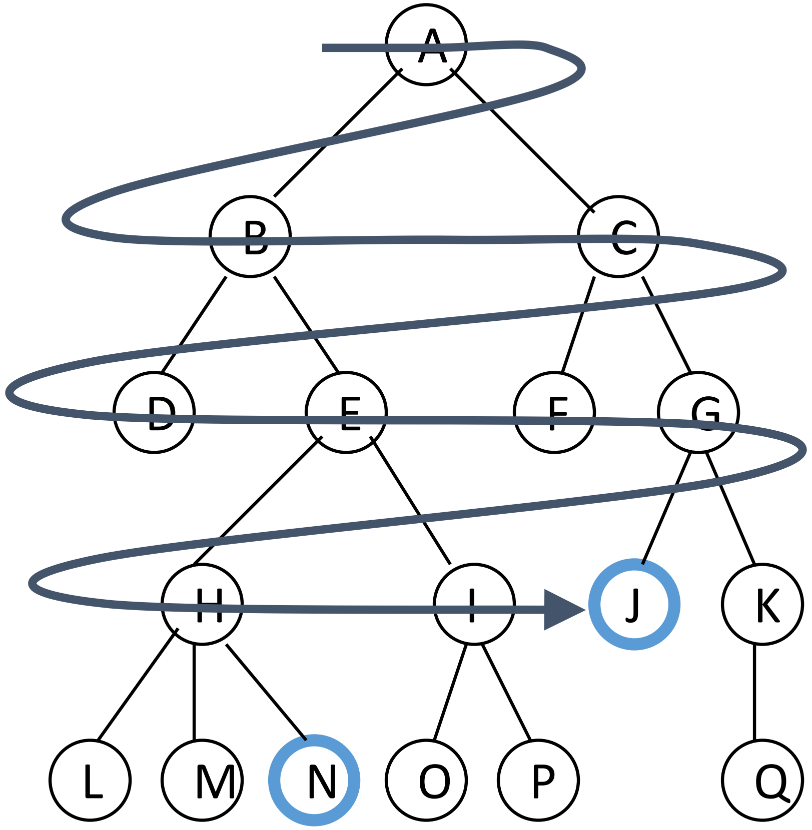
Search

Tiers



Breadth-first searching

- A breadth-first search (BFS) explores nodes nearest the root before exploring nodes further away
- For example, after searching A, then B, then C, the search proceeds with D, E, F, G
- Node are explored in the order A B C D E F G H I J K L M N O P Q
- J will be found before N



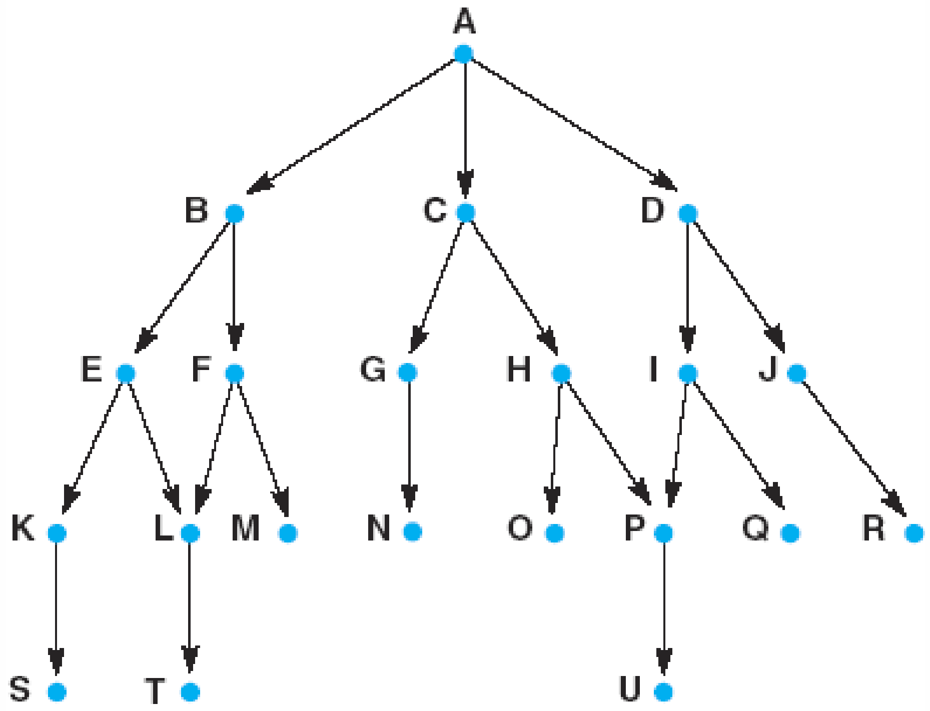
Function breadth_first search algorithm

OPEN lists states that have been generated but whose children have not yet been examined

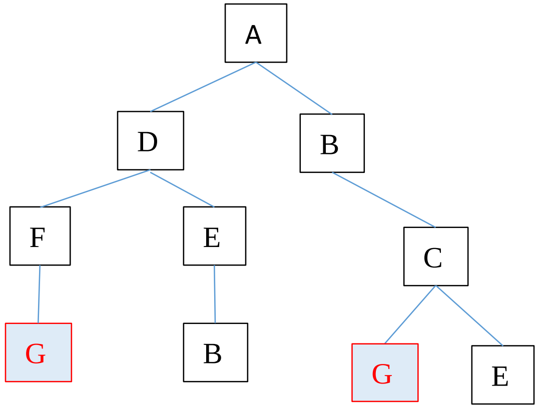
```
function breadth_first_search;

begin
   open := [Start];                                                % initialize
   closed := [ ];
   while open ≠ [ ] do                                             % states remain
      begin
         remove leftmost state from open, call it X;
            if X is a goal then return SUCCESS                     % goal found
               else begin
                  generate children of X;
                  put X on closed;
                  discard children of X if already on open or closed;   % loop check
                  put remaining children on right end of open          % queue
               end
      end
   return FAIL                                                     % no states left
end.
```

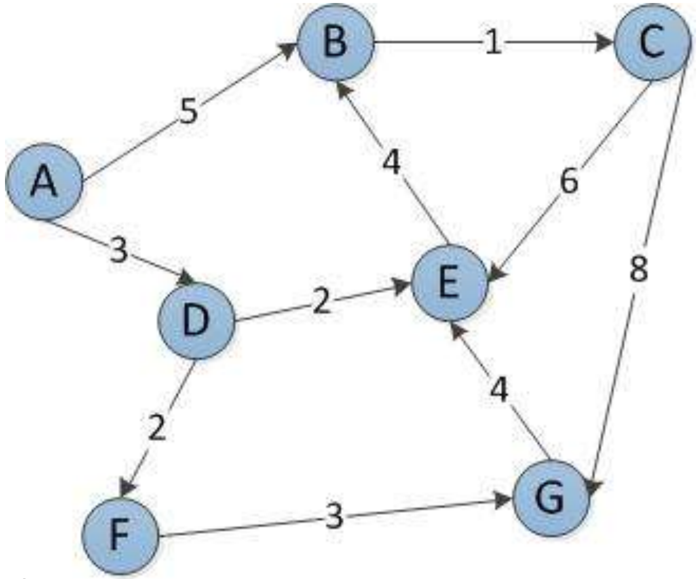
CLOSE records states that have already been examined

A trace of BFS



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** = [ ]



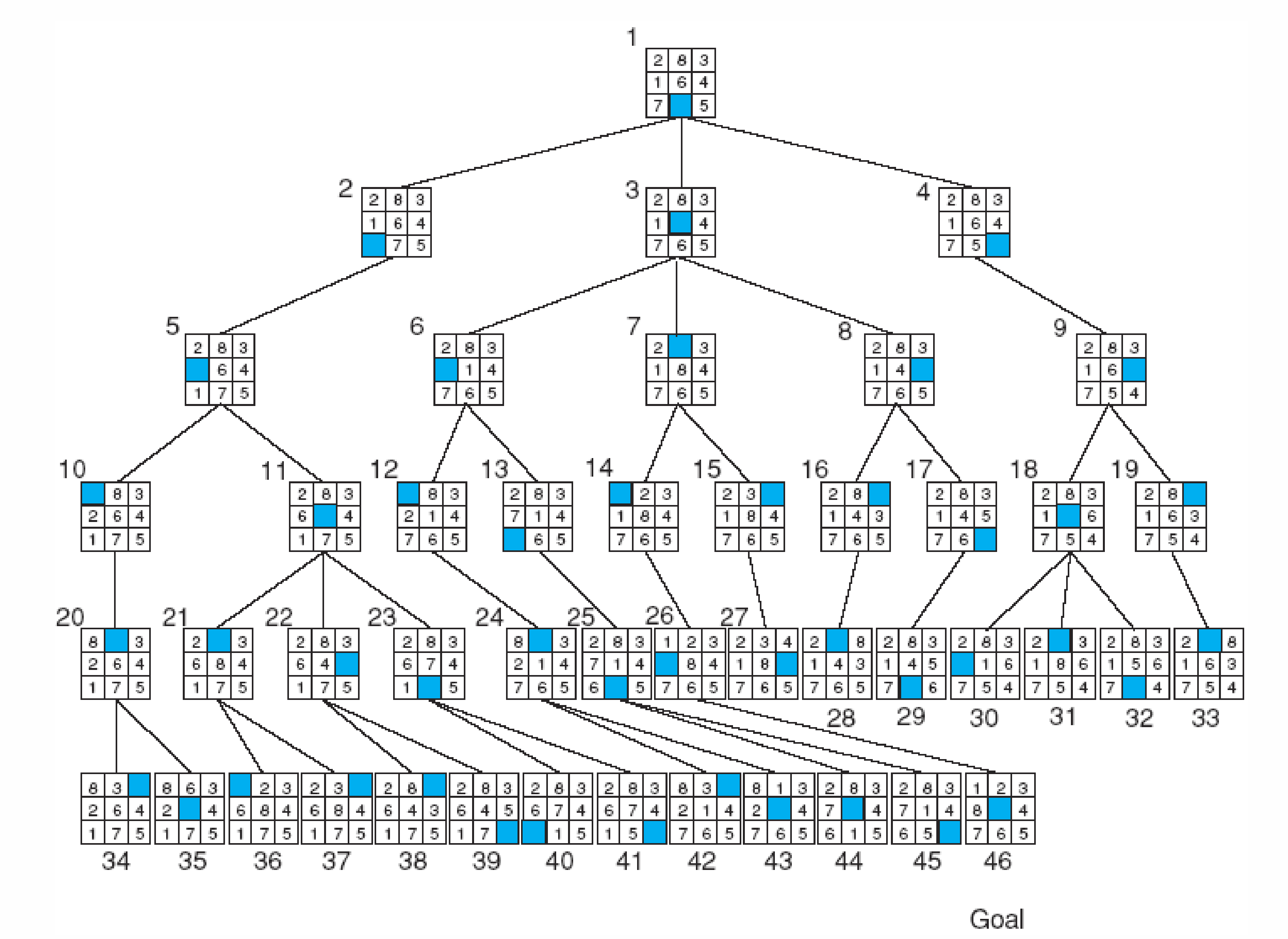


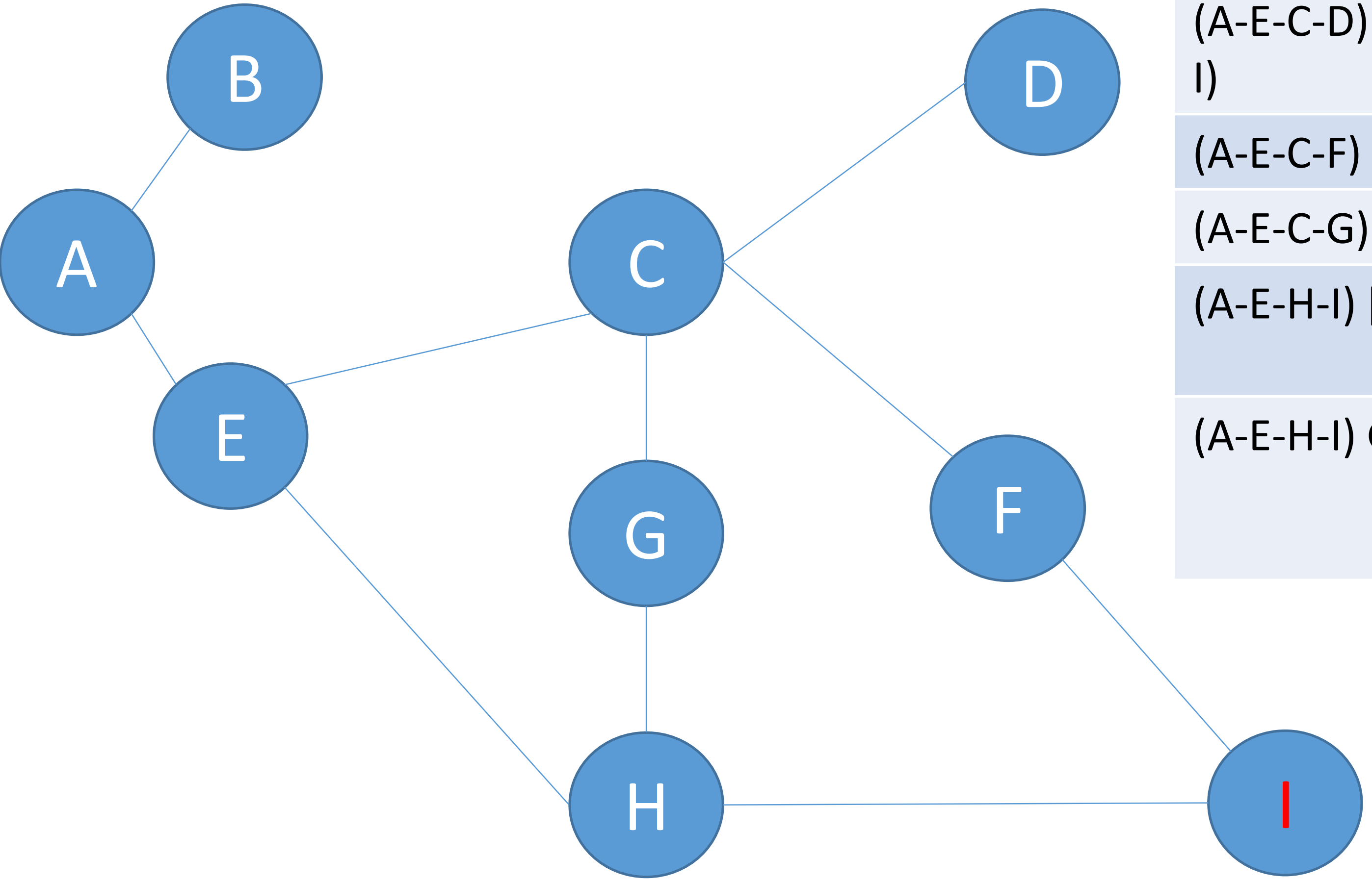
- Start Node: A
- Goal Node: G

Step	Frontier	Expand[*]	Explored: a set of nodes
1	{ A }	A	Ø
2	{(A-D), (A,B)}	D	{A}
3	{(A-B),(A-D-F),(A-D-E)}	B	{A,D}
4	{(A-D-F),(A-D-E),(A-B-C)}	F	{A,D,B}
5	{ (A-D-E),(A-B-C), (A-D-F-G}	E	{A,D,B,F}
6	{(A-B-C),(A-D-F-G)}[*]	C	{A,D,B,F,E}
7	{(A-D-F-G), (A-B-C-G)}[+]	G	{A,D,B,E,F,C}
8	{(A-B-C-G)}	G	{A,D,B,E,F,C,G}
9	Ø		

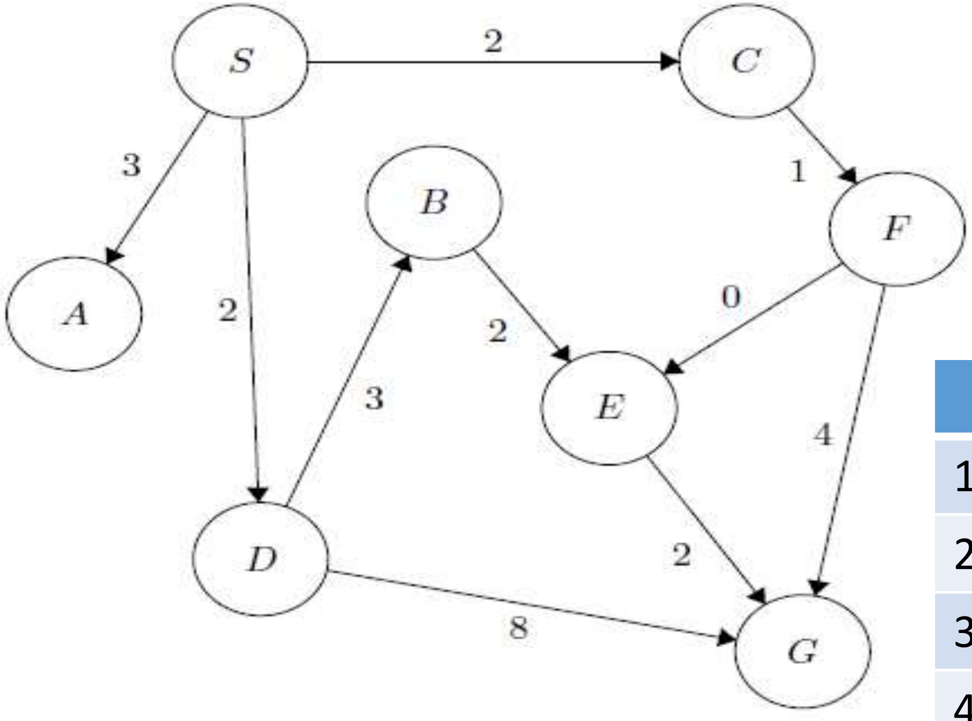
- Visited path: A -> D -> B -> E -> F -> C -> G.
- (*B, +E) is not added to the frontier because it is found in the explored set.

Breadth-first search of the 8-puzzle, showing order in which states were removed from open.





Frontier (BFS)	Expand	Explored
A	A	Empty
(A-B)(A-E)	B	A
(A-E)	E	A,B
(A-E-C)(A-E-H)	C	A,B,E
(A-E-H)(A-E-C-D) (A-E-C-F) (A-E-C-G)	H	A,B,E,C
(A-E-C-D) (A-E-C-F) (A-E-C-G) [*] (A-E-H-I)	D	A,B,E,C,H
(A-E-C-F) (A-E-C-G) (A-E-H-I)	F	A,B,E,C,H,D
(A-E-C-G) (A-E-H-I) [+]	G	A,B,E,C,H,D,F
(A-E-H-I) [-]	I	A,B,E,C,H,D,F,G
(A-E-H-I) Goal Found		A,B,E,C,H,D,F,G, I



	Frontier (BFS)	Expand	Explored
1	S	S	Empty
2	(S-A) (S-C) (S-D)	A	S
3	(S-C)(S-D)	C	S,A
4	(S-D)(S-C-F)	D	S,A,C
5	(S-C-F)(S-D-B)(S-D-G)	F	S,A,C,D
6	(S-D-B)(S-D-G) (S-C-F-E)[*]	B	S,A,C,D,F
7	(S-D-G) (S-C-F-E) [+]	G	S,A,C,D,F,B
8	(S-D-G) Goal Found		S,A,C,D,F,B,G

Search Strategies Evaluation

- Strategies are evaluated along the following dimensions:
  - **COMPLETENESS**: Does it always find a solution if one exists?
  - **OPTIMALITY**: Does it always find a least-cost solution?
  - **TIME COMPLEXITY**: How long does it take to find a solution. Number of nodes generated.
  - **SPACE COMPLEXITY**: Maximum number of nodes in memory

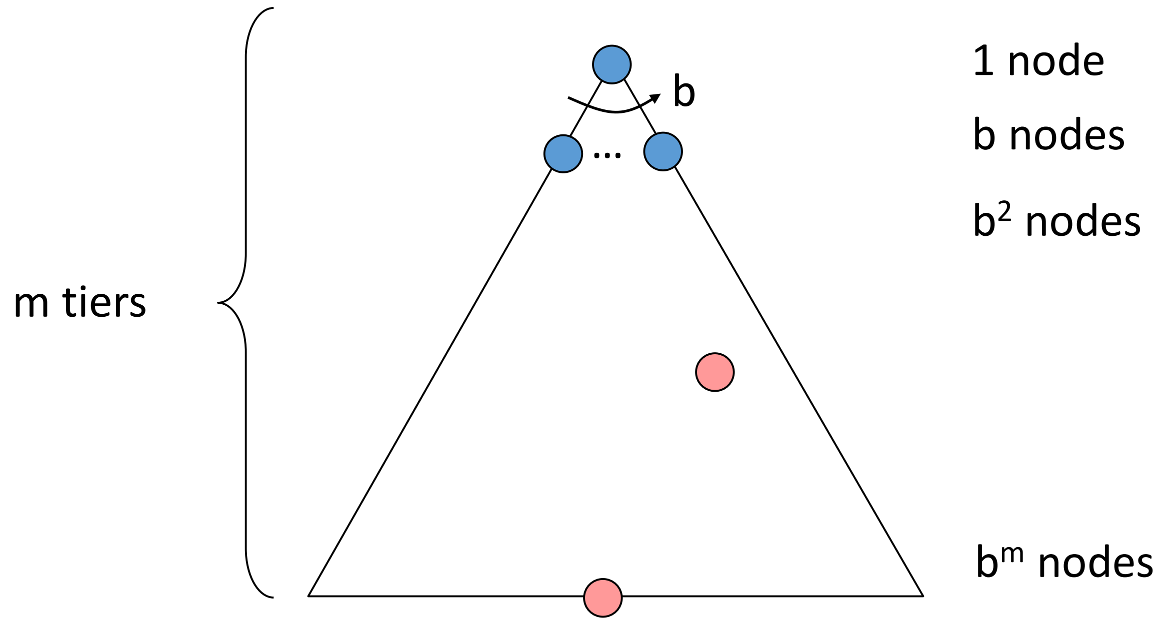
- Time and Space Complexity are measured in terms of:
  - The effective branching factor **b**:
    - Maximum no. of successors of any node
    - The average number of new nodes we create when expanding a new node
  - Depth **d**: Depth of the shallowest goal node
    - The length of a path to goal
  - **m**: the maximum length of any path in the state space.

Search Algorithm Properties

- Complete: Guaranteed to find a solution if one exists?
- Optimal: Guaranteed to find the least cost path?
- Time complexity?
- Space complexity?

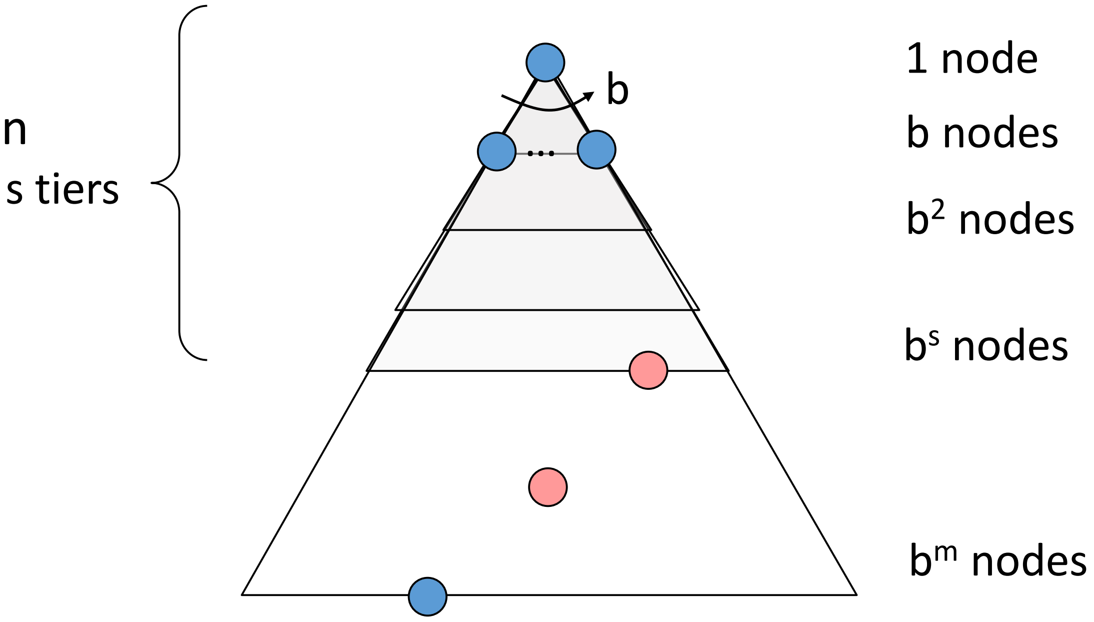
- Example of search tree:
  - b is the branching factor
  - m is the maximum depth
  - solutions at various depths

- Number of nodes in entire tree?
  - $1 + b + b^2 + \ldots. b^m = O(b^m)$



Breadth-First Search (BFS) Properties

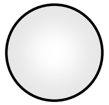
- What nodes does BFS expand?
  - Processes all nodes above shallowest solution
  - Let depth of shallowest solution be s
  - Search takes time $O(b^s)$
- How much space does the fringe take?
  - Has roughly the last tier, so $O(b^s)$
- Is it complete?
  - s must be finite if a solution exists, so yes!
- Is it optimal?
  - Only if costs are all 1 (more on costs later)



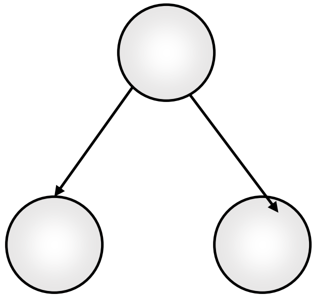
BFS Branching Factor

Branching factor: *number of nodes generated by a node parent (we called here "b")*

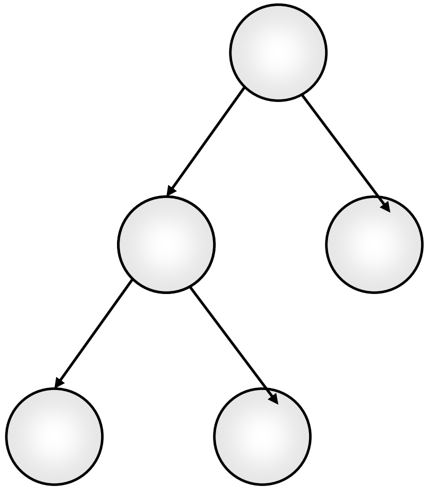
➔ *Here after b=2*



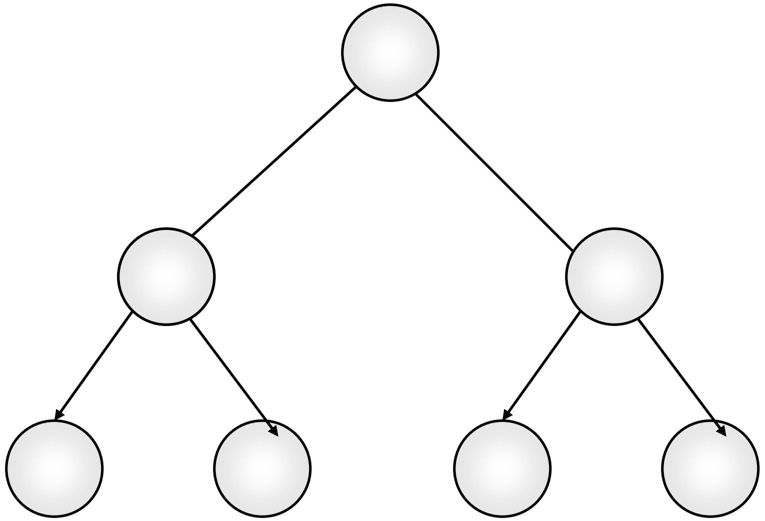
0 expansion



1 expansion



2 expansions



3 expansions

- *The root ➔ generates (b) new nodes*
- *Each of which ➔ generates (b) more nodes*
- *So, the maximum number of nodes expended before finding a solution at level "d", it is :*

  $$1+b+b^2+b^3+....+b^d$$

- *Complexity is exponential = $O(b^d)$*

Time and memory requirement in BFS

- The table assumes that 1 million nodes can be generated per second and that a node requires 1000 bytes of storage, b=10.
- Many search problems fit roughly within these assumptions when run on a modern personal computer.

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

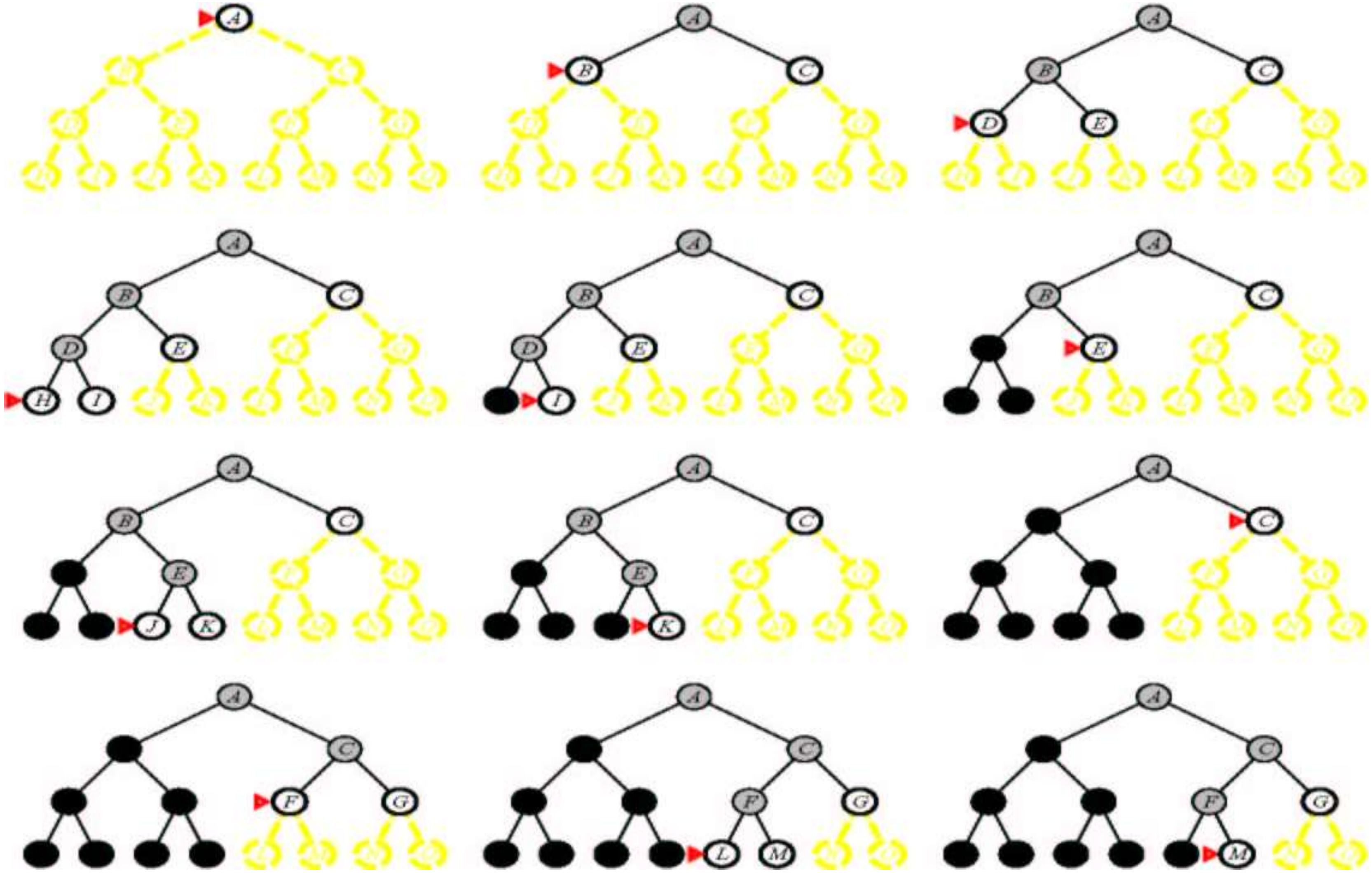
Breadth-first search: two lessons

- The memory requirements are a bigger problem than is the execution time
  - One might wait 13 days for the solution to an important problem with search depth 12, but no personal computer has the petabyte of memory it would take.
- **Time is still a major factor**
  - a solution at depth 16, (given our assumptions) it will take about 350 years for breadth-first search (or indeed any uninformed search) to find it.

Uninformed methods can solve the exponential complexity search problems only for smallest instances

Depth-First Search

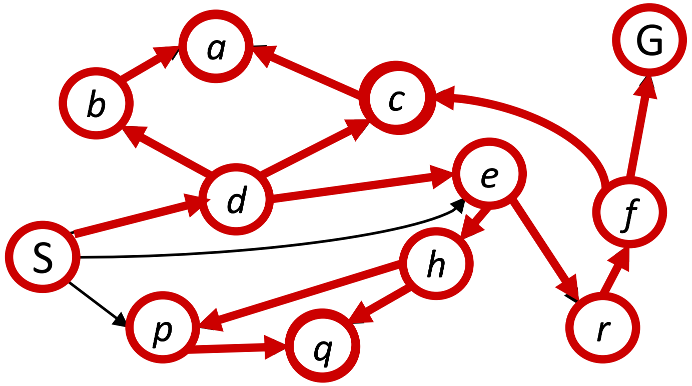
Depth First Search

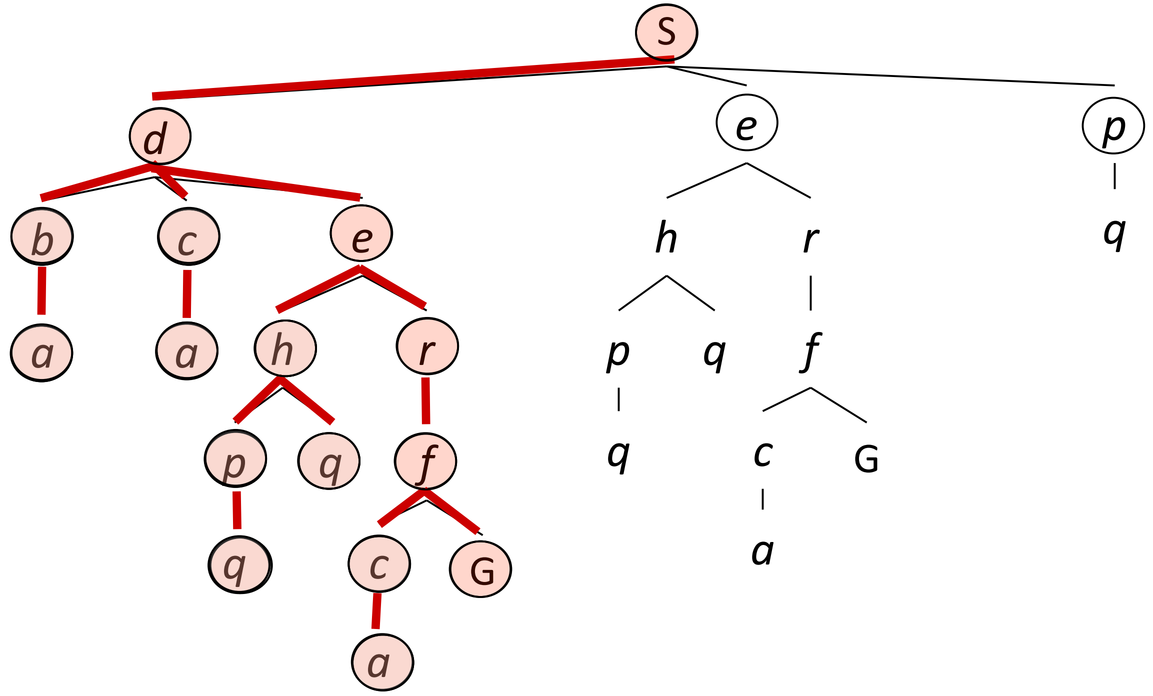


Depth-First Search

Strategy: expand a deepest node first

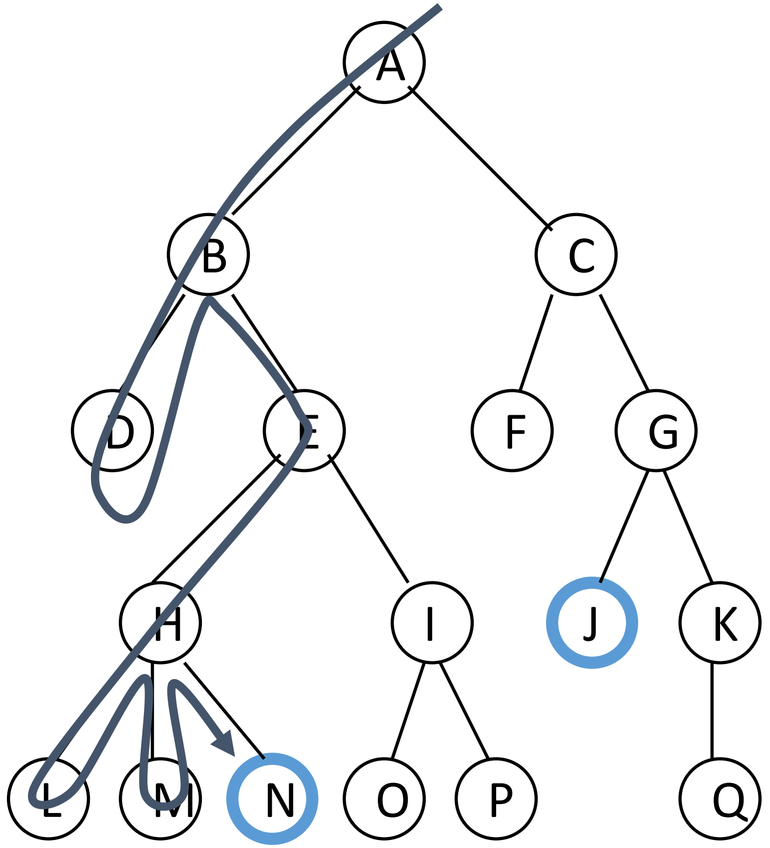
Implementation: Fringe is a LIFO stack





Depth-first searching

- A depth-first search (DFS) explores a path all the way to a leaf before backtracking and exploring another path
- For example, after searching A, then B, then D, the search backtracks and tries another path from B
- Node are explored in the order A B D E H L M N I O P C F G J K Q
- N will be found before J

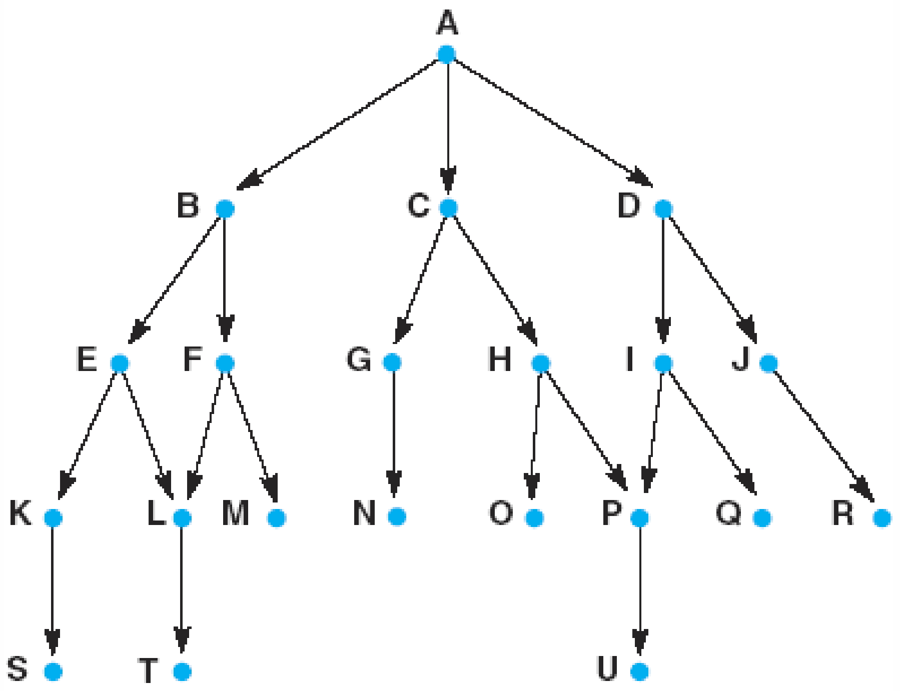


The depth-first search algorithm

```
begin
   open := [Start];                                              % initialize
   closed := [ ];
   while open ≠ [ ] do                                           % states remain
      begin
         remove leftmost state from open, call it X;
         if X is a goal then return SUCCESS                      % goal found
            else begin
               generate children of X;
               put X on closed;
               discard children of X if already on open or closed;   % loop check
               put remaining children on left end of open        % stack
            end
      end;
   return FAIL                                                   % no states left
end.
```

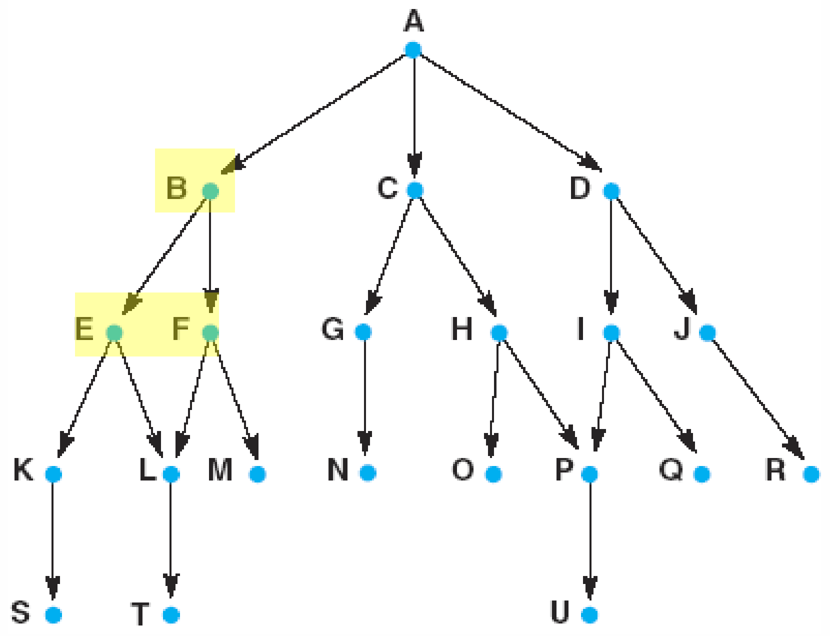
This is the only difference between depth-first and breadth-first.

DFS Algorithm



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]**

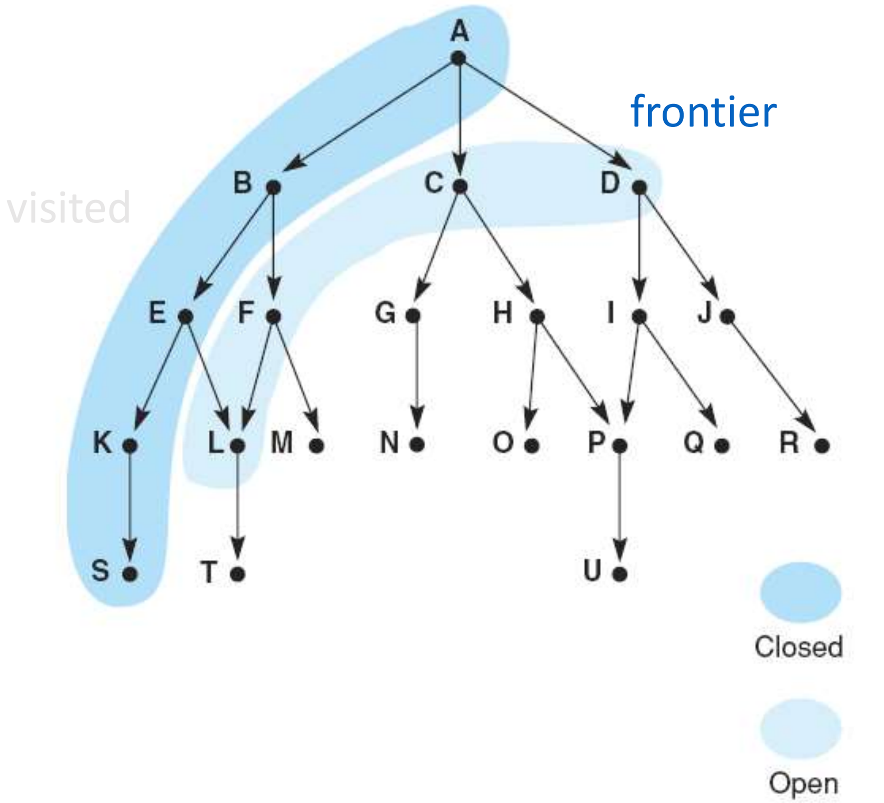
DFS Algorithm



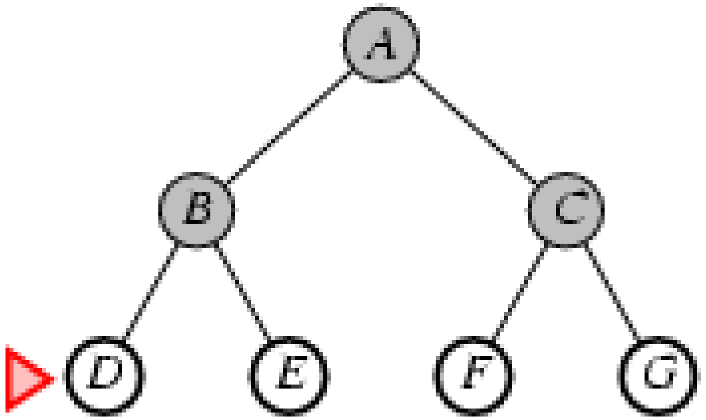
top of stack

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]**

Snap shot at iteration 6



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]



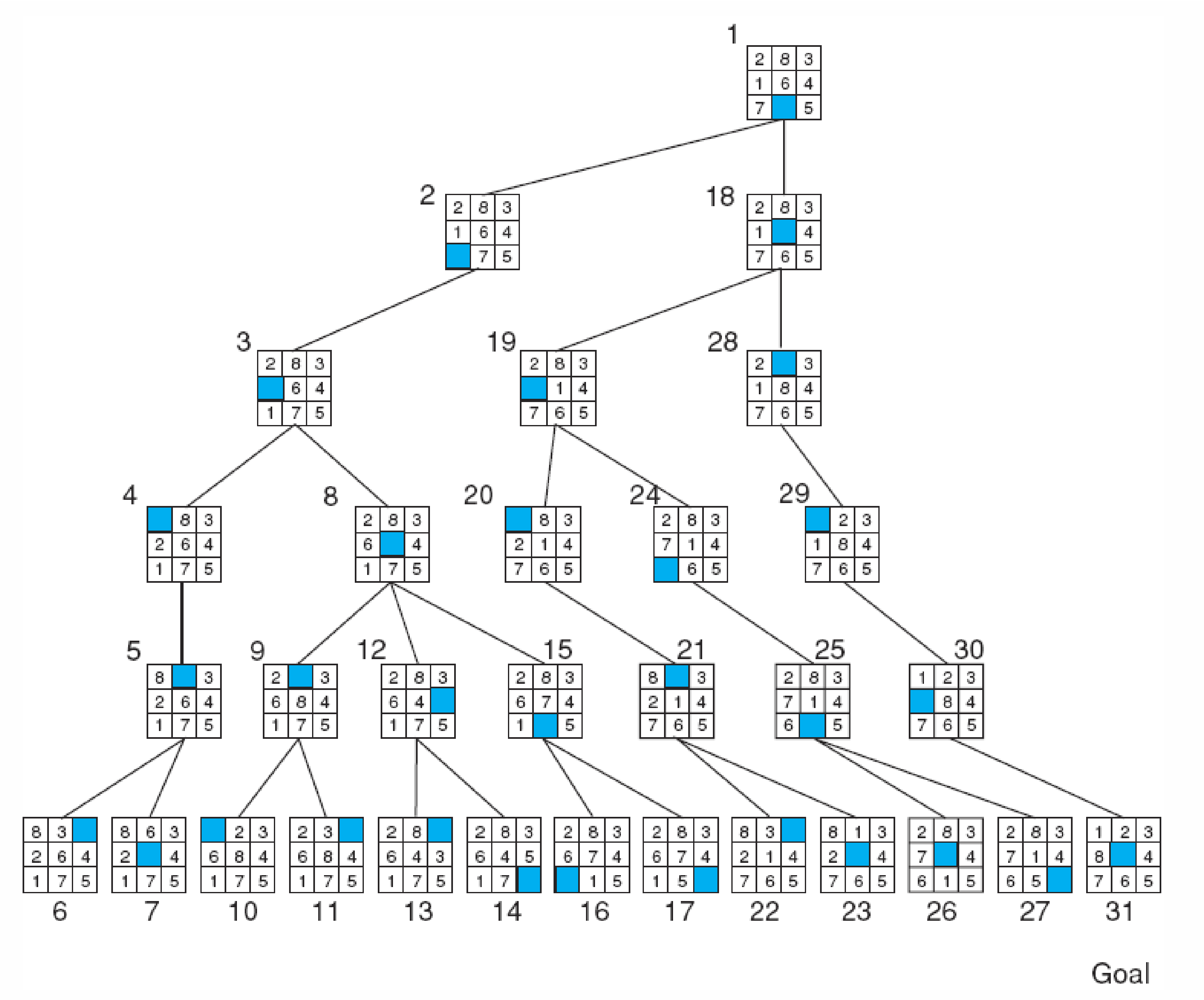
Start Node: A

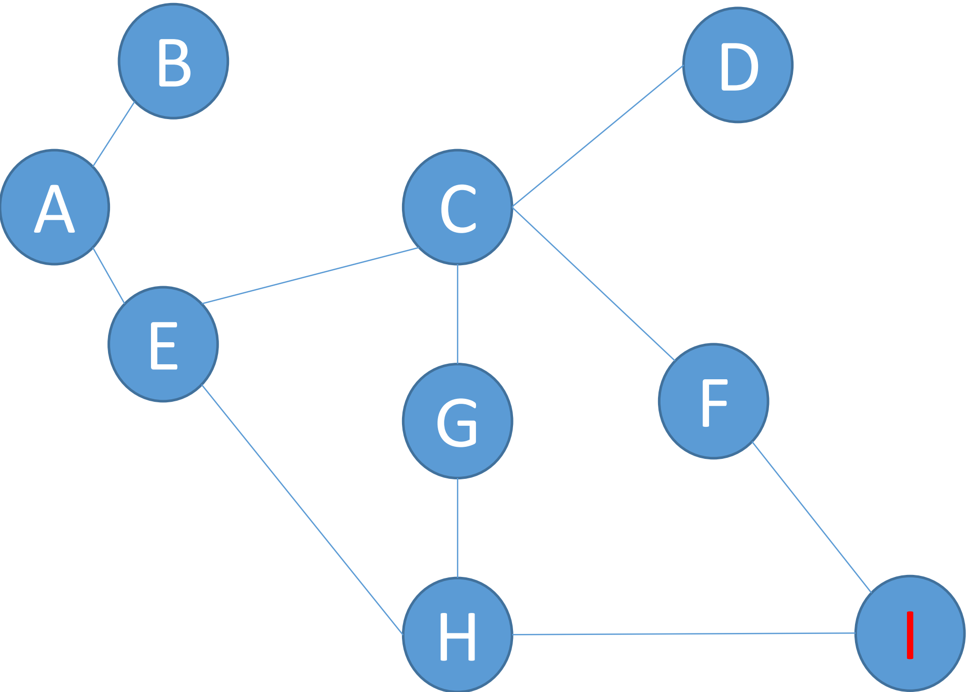
Goal Node: G

Step	Frontier	Expand[*]	Explored: a set of nodes
1	{A}	A	Ø
2	{(A-B),(A-C)}	B	{A}
3	{(A-B-D),(A-B-E),(A-C)}	D	{A,B}
4	{(A-B-E),(A-C)}	E	{A,B,D}
5	{(A-C)}	C	{A,B,D,E}
6	{(A-C-F),(A-C-G)}	F	{A,B,D,E,C}
7	(A-C-G)}	G	{A,D,B,E,C,F,G}
8	Ø		

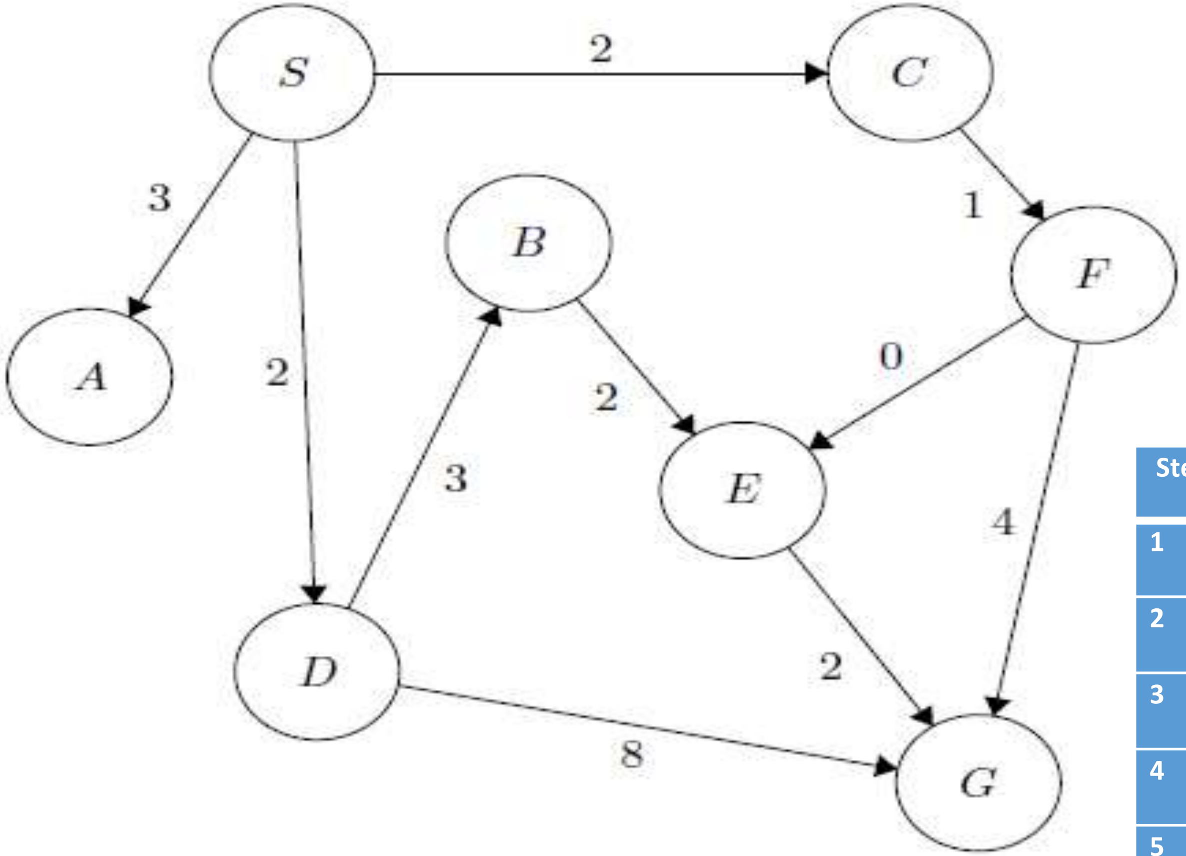
Found the path: A -> C -> G.

Depth-first search of the 8-puzzle with a depth bound of 5.





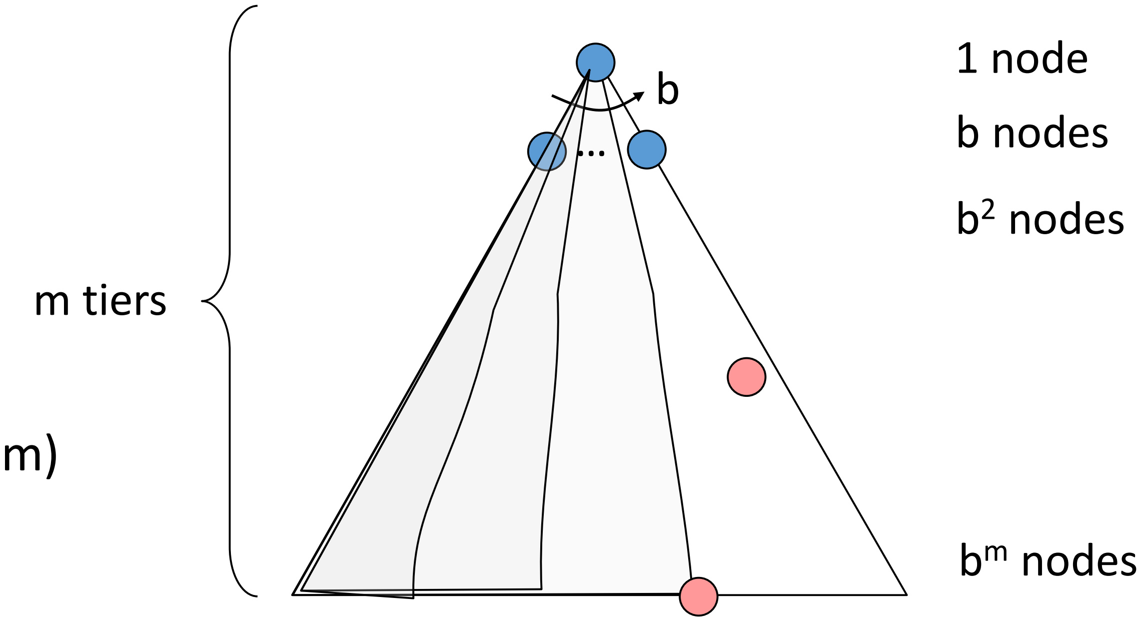
Frontier (DFS)	Expand	Explored
A	A	Empty
(A-B)(A-E)	B	A
(A-E)	E	A,B
(A-E-C)(A-E-H)	C	A,B,E
(A-E-C-D) (A-E-C-F) (A-E-C-G)(A-E-H)	D	A,B,E,C
(A-E-C-F) (A-E-C-G)(A-E-H)	F	A,B,E,C,D
(A-E-C-F-I) (A-E-C-G)(A-E-H)	I	A,B,E,C,D,F
(A-E-C-F-I) Goal Found		



Step	Frontier (DFS)	Expand	Explored: a set of Nodes
1	S	S	Ø
2	(S-A)(S-C)(S-D)	A	S
3	(S-C)(S-D)	C	S-A
4	(S-C-F) (S-D)	F	S-A-C
5	(S-C-F-E)(S-C-F-G)(S-D)	E	S-A-C-F
6	[*](S-C-F-G)(S-D)	G	S-A-C-F-E
7	(S-C-F-G) GOAL FOUND		S-A-C-F-E-G

Depth-First Search (DFS) Properties

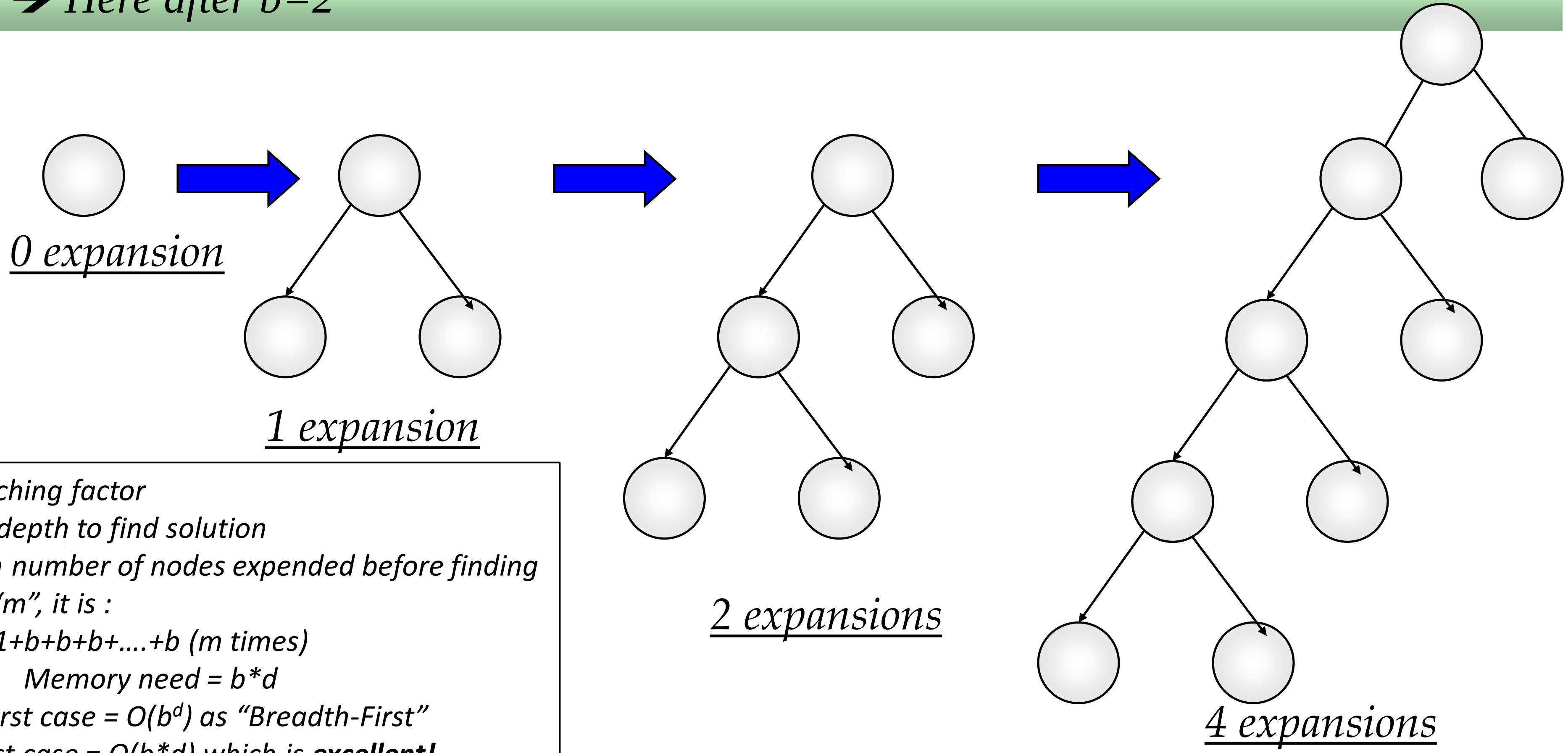
- What nodes DFS expand?
  - Some left prefix of the tree.
  - Could process the whole tree!
  - If m is finite, takes time $O(b^m)$
- How much space does the fringe take?
  - Only has ancestors on path to root, so $O(bm)$
- Is it complete?
  - m could be infinite, so only if we prevent cycles
- Is it optimal?
  - No, it finds the "leftmost" solution, regardless of depth or cost



DFS Branching Factor

Branching factor: *number of nodes generated by a node parent (we called here "b")*

➔ *Here after b=2*



- *Let **b**: is the branching factor*
- *Let **d**: maximum depth to find solution*
- *So, the maximum number of nodes expended before finding a solution at level "m", it is :*

1+b+b+b+….+b (m times)

*Memory need = b*d*

- *Complexity in worst case = $O(b^d)$ as "Breadth-First"*
- *Complexity in best case = O(b*d) which is **excellent!***

Time and memory requirement in DFS

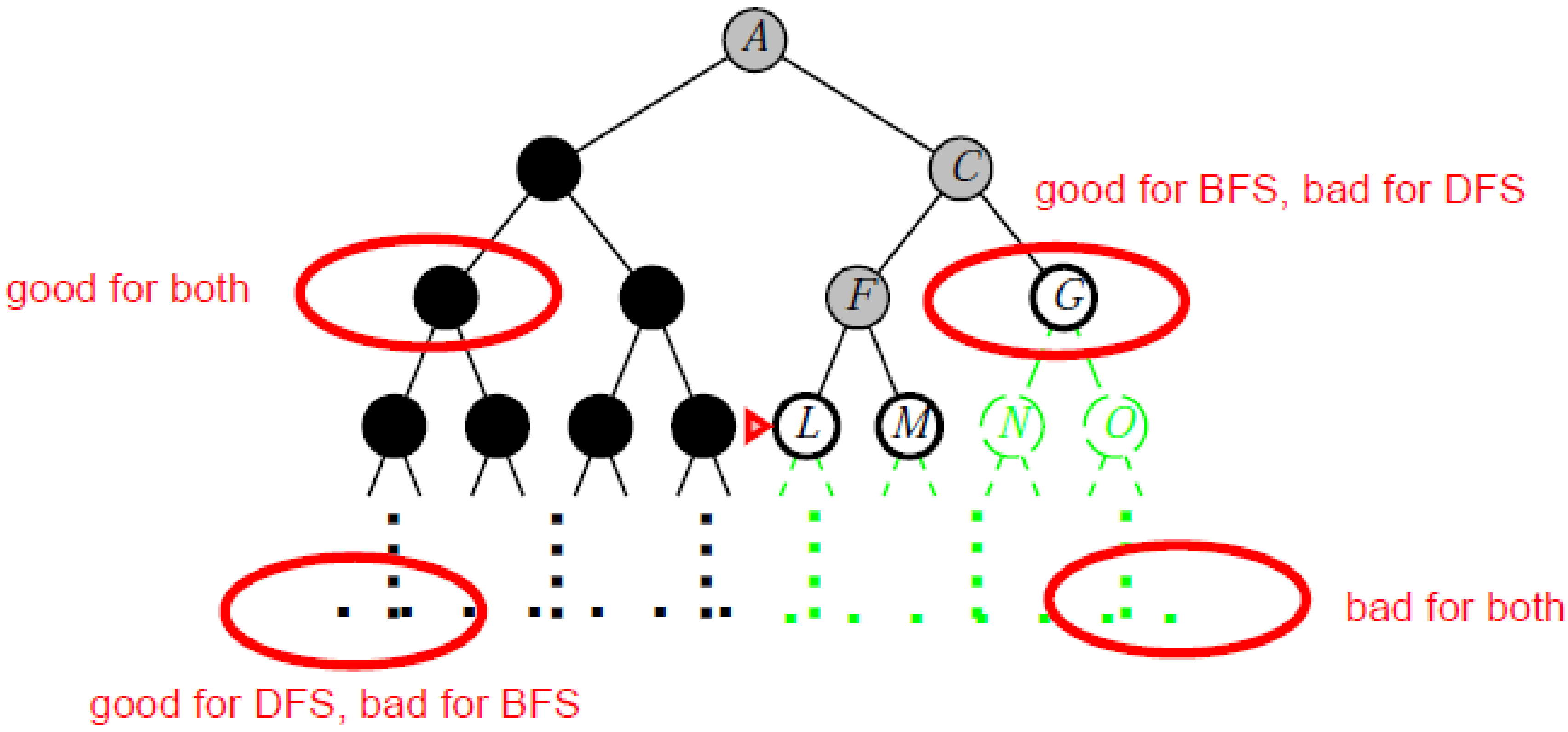
Depth	*Nodes*	*Time (best case)*	*Memory*
0	1	1 millisec	100 bytes
2	20	0.02 sec	2 Kb
4	40	0.04 sec	4 Kb
6	10 * 6	0.06 sec	6 Kb
8	10 * 8	0.08 sec	8 Kb
10	10 *10	0.1 sec	10 Kb
12	10 * 12	0.12 sec	12 Kb
14	10 * 14	0.14 sec	14 Kb

Assume branching factor b=10; 1000 nodes explored/sec and 100 bytes/node

Comparing the ordering of search sequences

- Determine the order of nodes (states) to be examined
- Breadth-first search
  - When a state is examined, all of its children are examined, one after another
  - Explore the search space in a level-by-level fashion
- Depth-first search
  - When a state is examined, all of its children and their descendants are examined before any of its siblings
  - Go deeper into the search space where possible

BFS or DFS


A
C
good for BFS, bad for DFS
good for both
F
G
L
M
N
O
bad for both
good for DFS, bad for BFS

When to use BFS vs. DFS?

- We need the shortest path to a solution

BFS DFS

- There are only solutions at great depth

BFS DFS

- There are some solutions at shallow depth

BFS DFS