ARTIFICIAL INTELLIGENCE

Solving problems by searching-2

Chapter 3

Assoc. Prof. Zeynep ORMAN

Sample Questions for Lecture 3

- Consider the given problems as a state space search problem. Choose a formulation that is precise enough to be implemented.

a) How would you represent a state?
b) Describe the initial state and the goal test.
c) Describe the successor function

Sample Questions for Lecture 3

1. You have to color a planar map using only four colours, in such a way that no two adjacent regions have the same colour.

The map is represented as a graph. Each region corresponds to vertex of the graph.

If two regions are adjacent, there is an edge connecting the corresponding vertices.

Sample Questions for Lecture 3

1. You have to color a planar map using only four colours, in such a way that no two adjacent regions have the same colour.

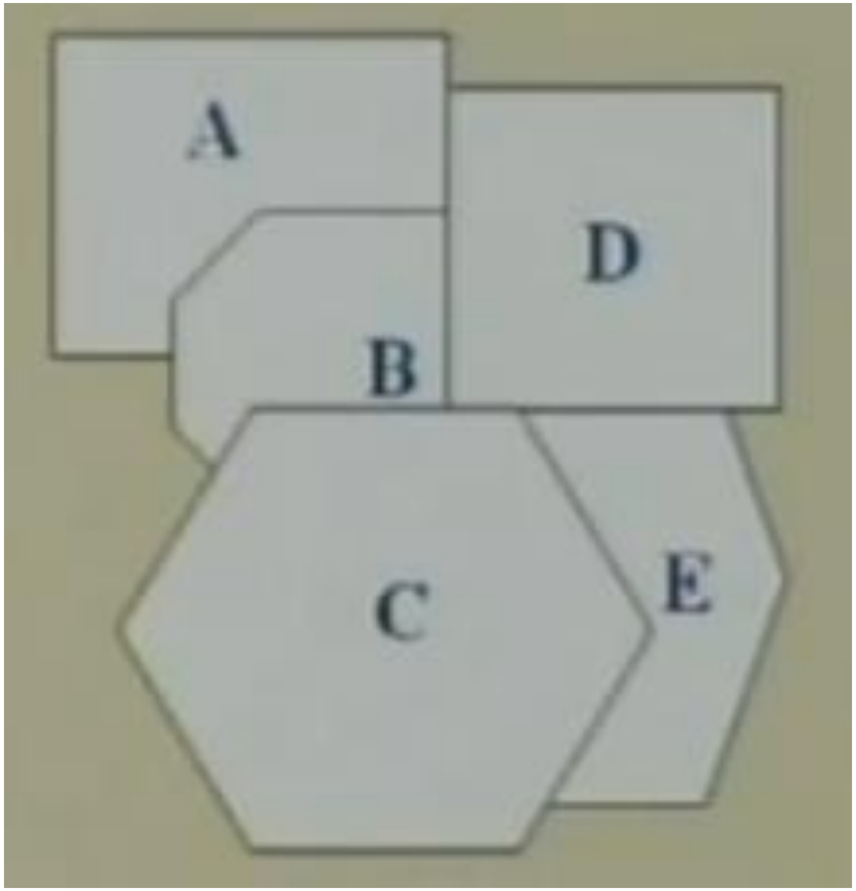
The vertices are named <v1, v2, …., vN>

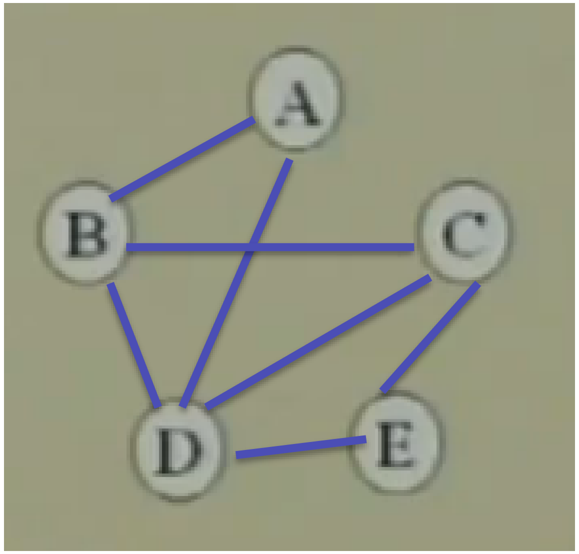
The colours are represented by c1, c2, c3, c4.

A state is represented as a N-tuple

{c1, x, c1, c3, x, x, x, …..}

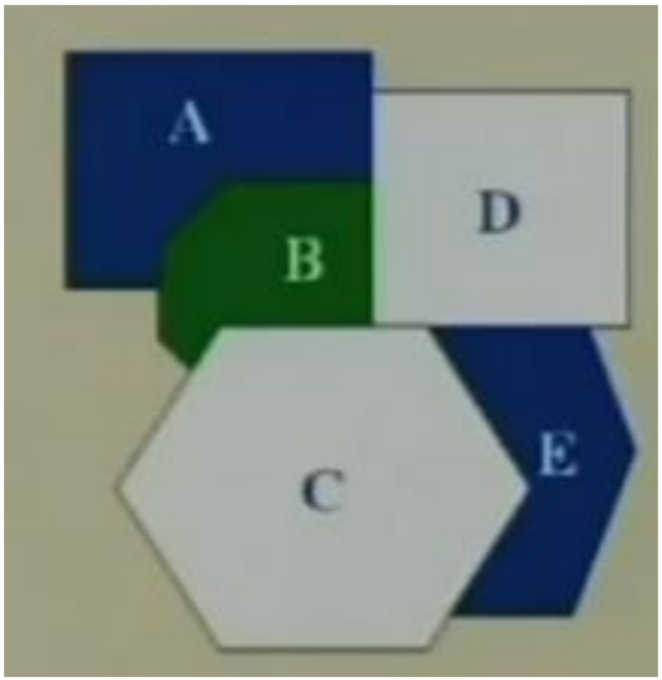
Map Coloring Problem

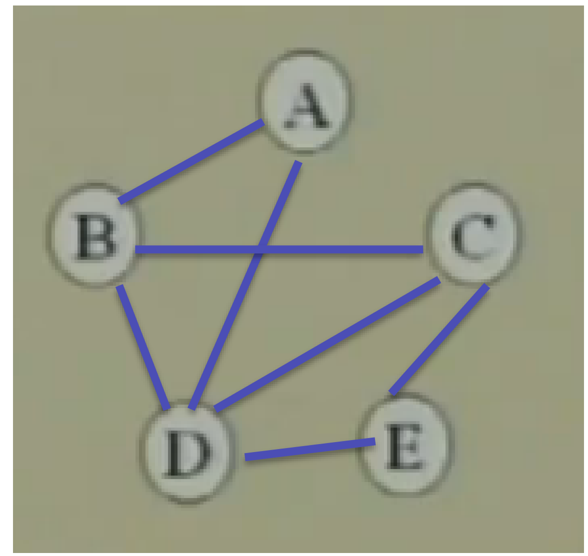




This graph represents the map.

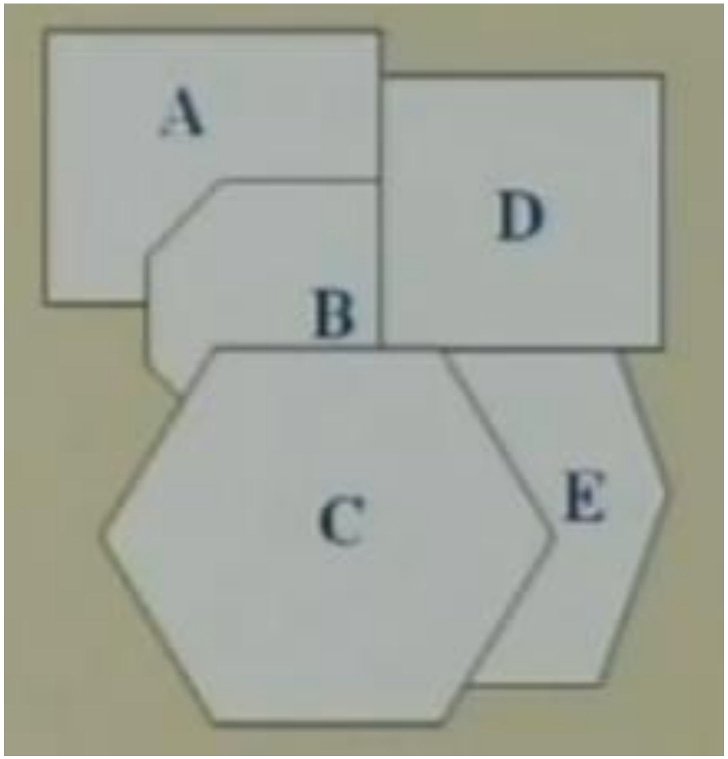
Map Coloring Problem

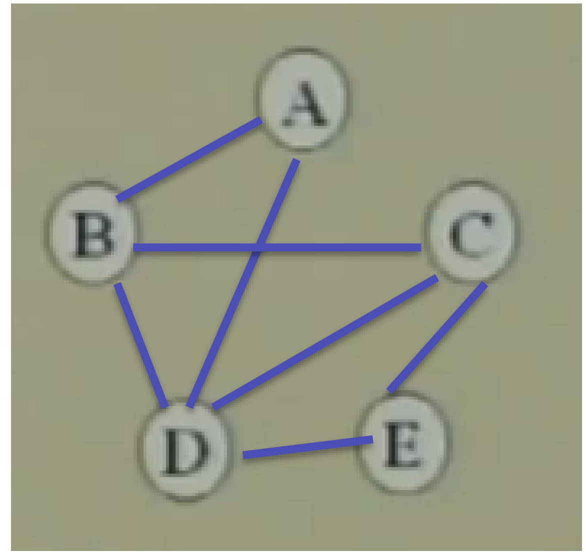




Representation of current state
{blue, green, x, x, blue}

Map Coloring Problem





Initial state

{x, x, x, x, x}

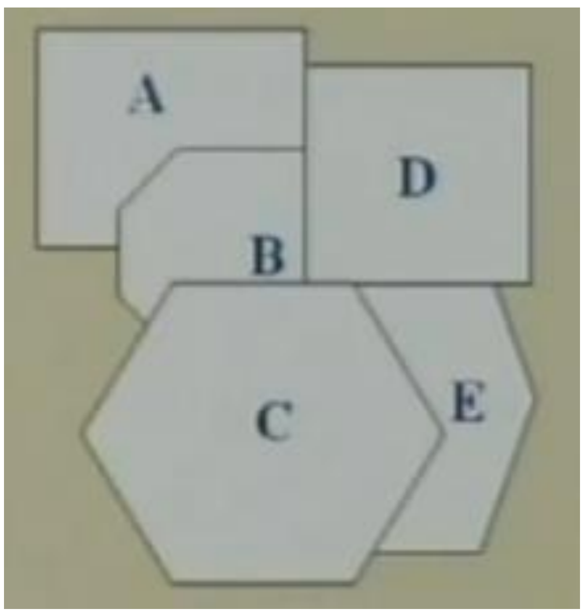
Goal test

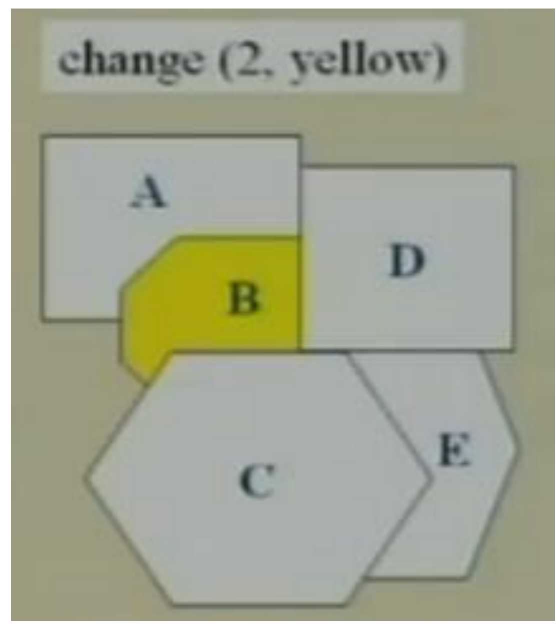
if r_i and r_j are adjacent

colour(i) $\neq$ colur(j)

Successor function

- Change the colour of a state i to c.
  - change (i, c)

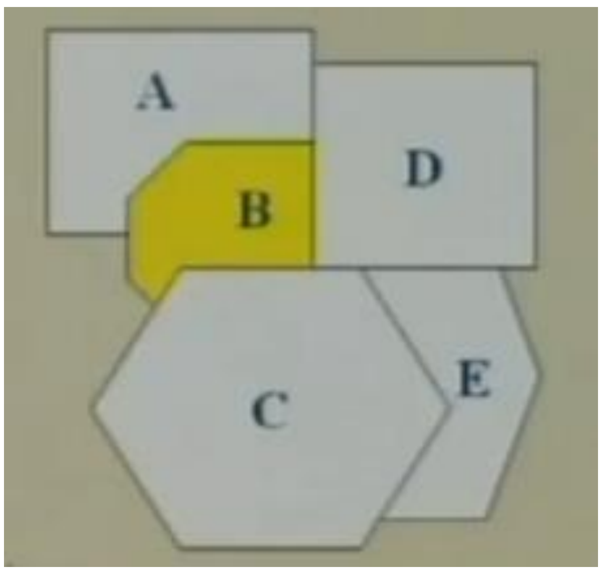


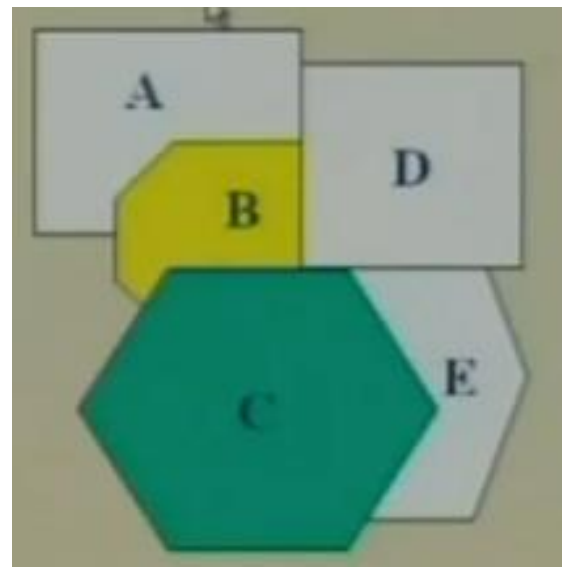


Successor function

- Change the colour of a state i to c.
  - change (i, c)

change (3, green)

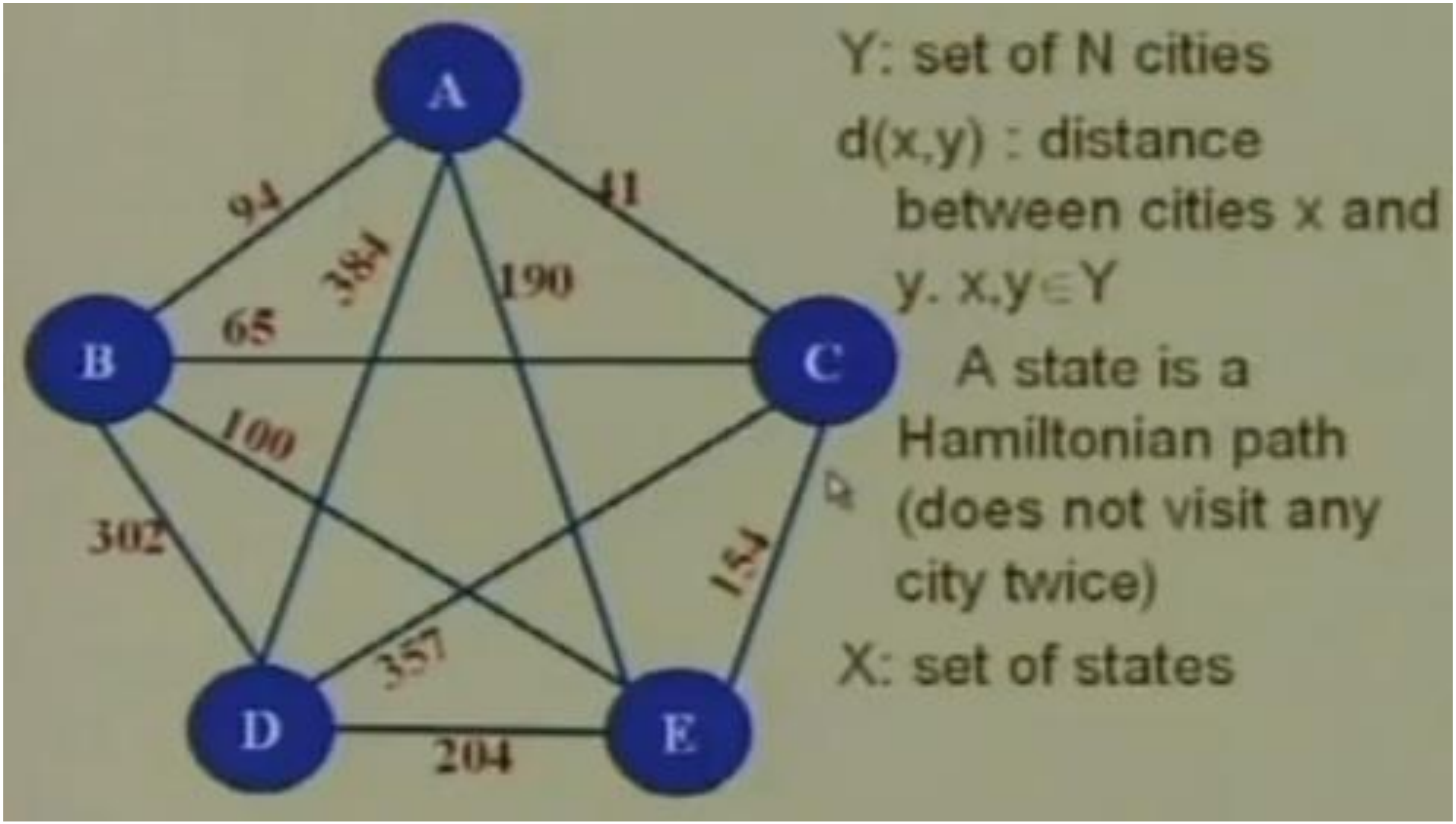




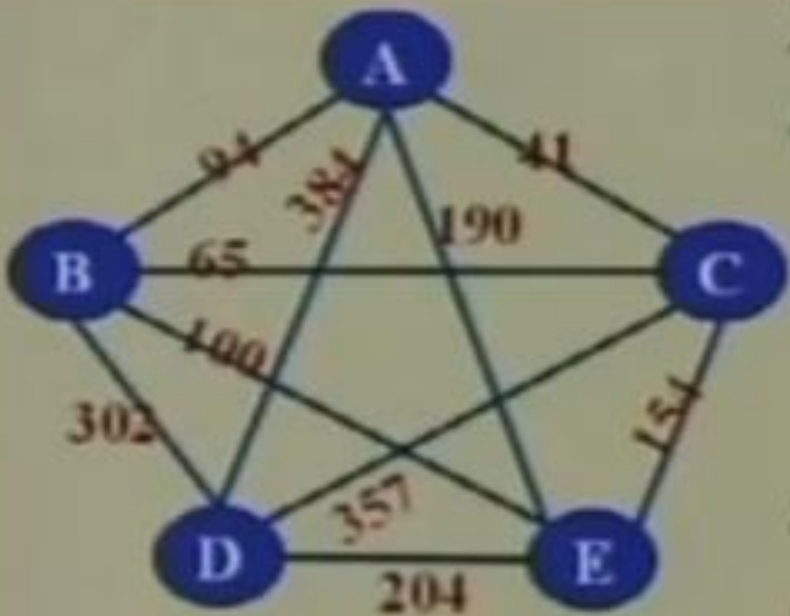
Sample Questions for Lecture 3

2. In the traveling salesperson problem (TSP), there is a map involving N cities some of which are connected by roads. The aim is to find the shortest tour that starts from a city, visits all the cities exactly once and comes back to the starting city.

TSP


A
B
C
D
E
94
41
384
190
65
100
302
154
357
204
Y: set of N cities
d(x,y) : distance between cities x and y. x,y∈Y
A state is a Hamiltonian path (does not visit any city twice)
X: set of states

TSP



X: set of states. X = $\{(x_1, x_2, \ldots, x_n) \mid$

$n=1, \ldots, N+1,$

$xi \in Y$ for all I,

$xi \neq xj$ unless $i=1, j=N+1\}$

Successors of state $(x_1, x_2, \ldots, x_n)$:

$$\delta(x_1, x_2, \ldots, x_n) = \{(x_1, x_2, \ldots, x_n, x_{n+1}) \mid xn+1 \in Y$$

$$x_{n+1} \neq x_i \text{ for all } 1 \leq i \leq n\}$$

The set of goal states include all states of length N+1

Sample Questions for Lecture 3

3. Missionaries and Cannibals problem:

3 missionaries & 3 cannibals are on one side of the river. The boat can carry two. Missionaries must never be outnumbered by cannibals.

Give a plan for all cross the river.

State: <M, C, B>

M: no of missionaries on the left bank

C: no of cannibals on the left bank

B: position of the boat: L or R

Missionaries and Cannibals

Initial State: <3, 3, L>

Goal State: <0, 0, R>

Operators:

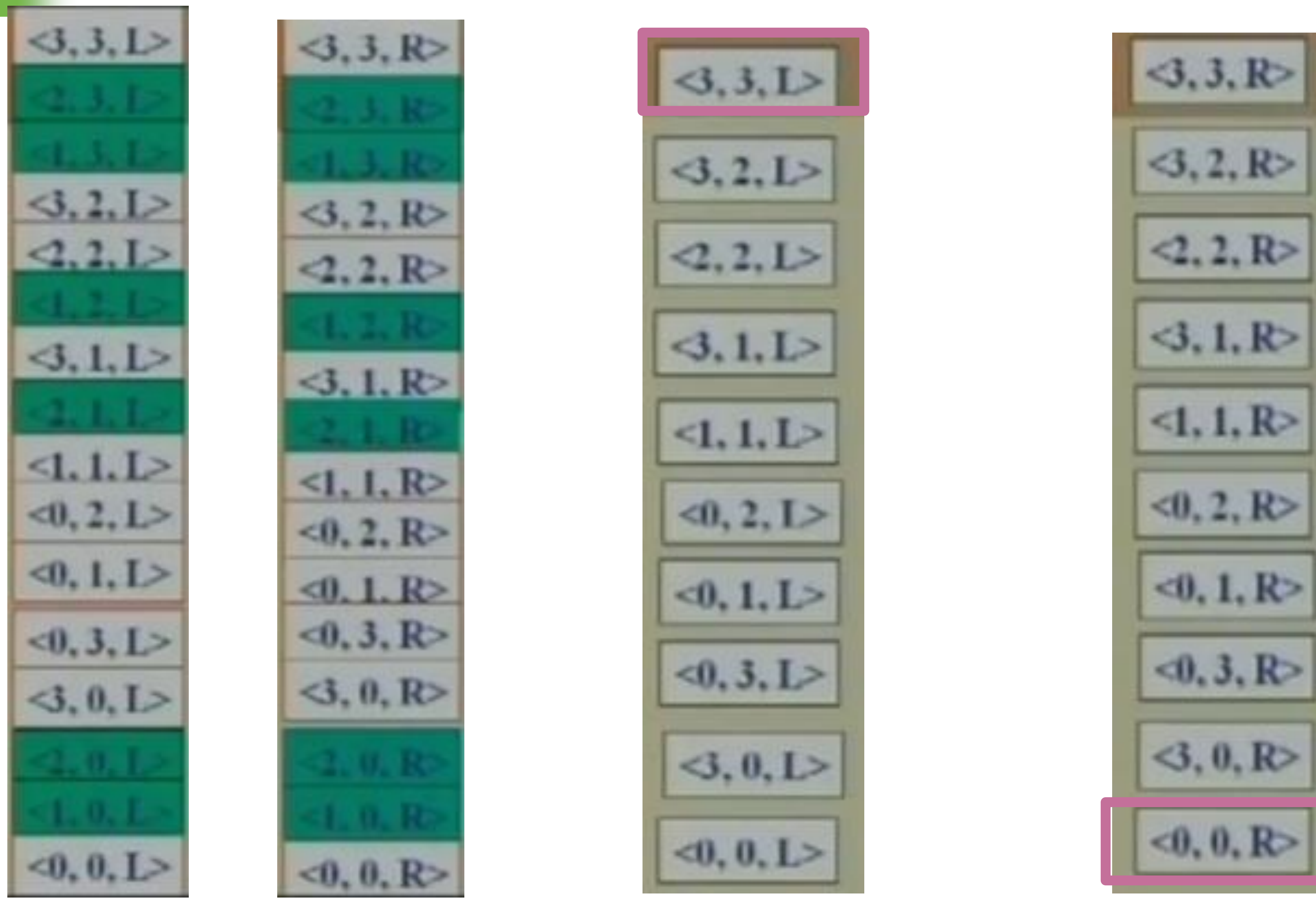
m: no of missionaries on the boat

n: no of cannibals on the boat

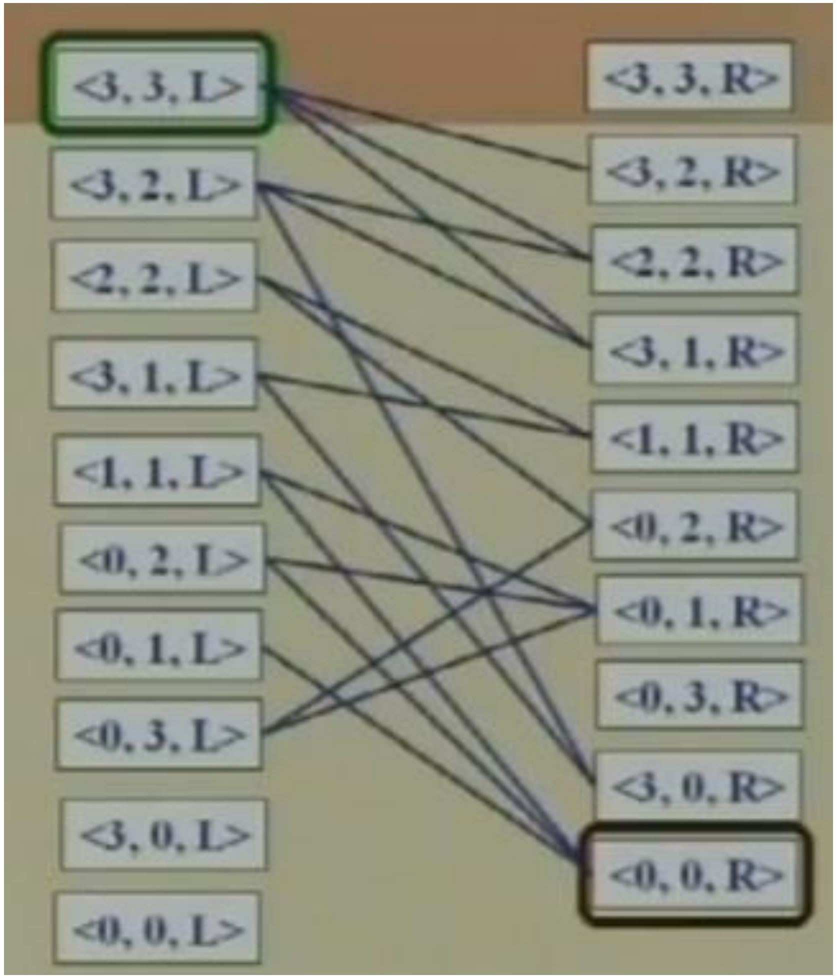
Valid operators in terms of <m,n>

<1,0> <2,0> <1,1> <0,1> < 0,2 >

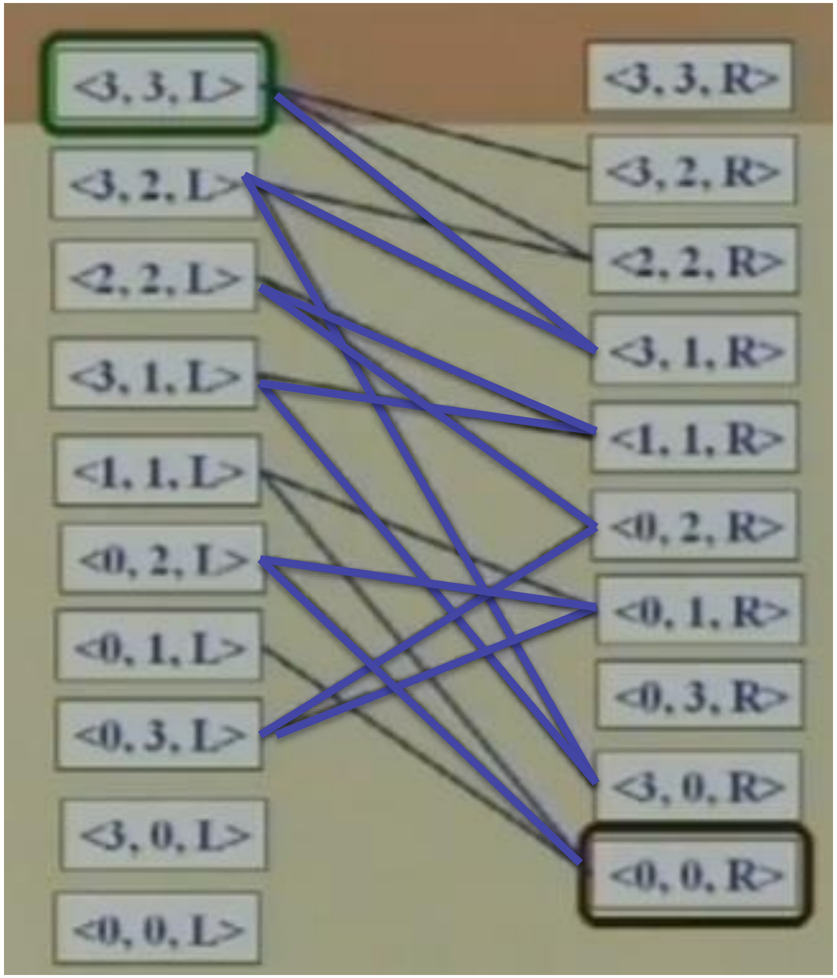
Missionaries and Cannibals



Missionaries and Cannibals



Solution to Missionaries and Cannibals



Outline

- Basic search algorithms
  - Uninformed search – they are given no information about the problem other than its definition
    - depth-first, breadth-first
  - Informed search – have some idea of where to look for solutions
    - best-first, hill-climbing

Outline

- Uninformed search strategies
  - For each of the uninformed search algorithms, you will learn
    - The algorithm
    - The time and space complexities
    - When to select a particular strategy

Outline

- At the end of this lesson, the student should be able to do the following:
  - Analyze a given problem and identify the most suitable search strategy for the problem.
  - Given a problem, apply one of this strategies to find a solution to the problem.

Search problem representation

S: set of states

Initial state $s_0 \in S$

A: $S \rightarrow S$ operators/ actions

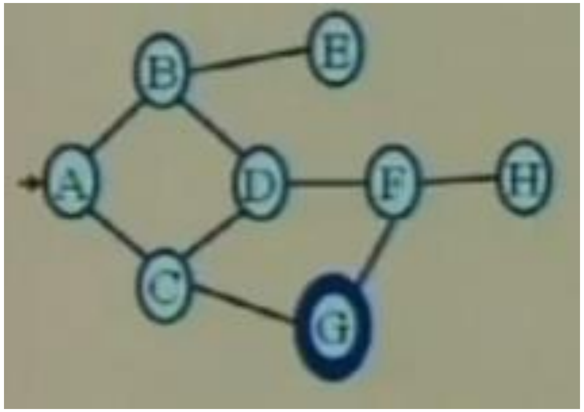
G : goal

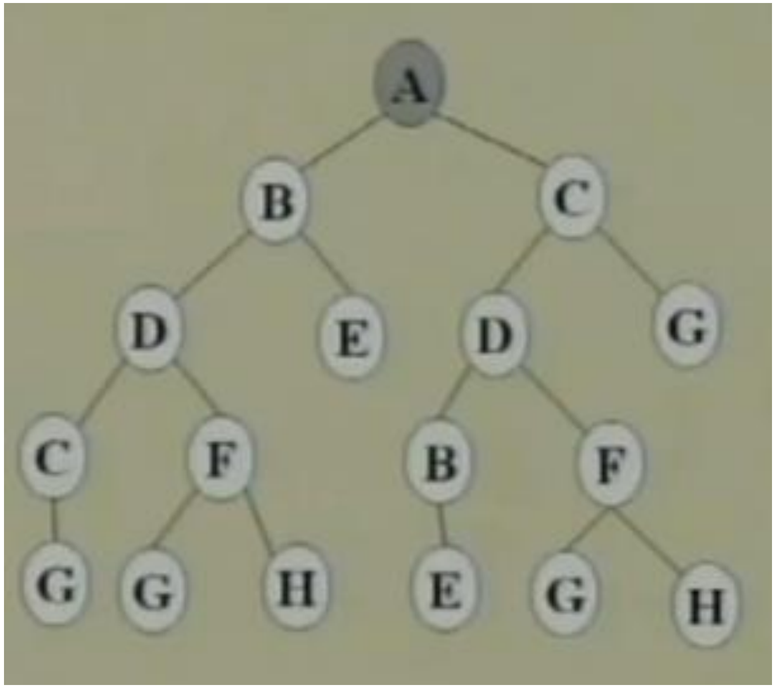
Search problem: $\{S, s_0, A, G\}$

- A *plan* is a sequence of actions.
  $P = \{a_0, a_1, \ldots, a_N\}$
  which leads to traversing a number of states
  $\{s_0, s_1, \ldots, s_{N+1} \in g\}$
- Path cost : path $\rightarrow$ positive number

Search Tree

- List all possible paths
- Eliminate cycles from paths
- Result: A search tree





Basic search algorithm

Let *fringe* be a list containing the initial state

Loop

    if *fringe* is empty return *failure*

    Node ← remove-first (*fringe*)

    if Node is a *goal*

        then return the path from initial state to Node

    else generate all successors of Node, and

        merge the newly generated nodes into *fringe*

End Loop

Search Strategies

- Uninformed search strategies – blind search
  - they have no additional information about states – only the problem definition.
  - all they can do is generate successors and distinguish a goal state from a nongoal state.
- Informed search strategies – heuristic search
  - they know whether a nongoal state is “more promising” than another.
- All search strategies are distinguished by the order in which nodes are expanded.

Uninformed search strategies

- Uninformed search strategies use only the information available in the problem definition.
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search

Search strategies

- A search strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: number of nodes generated/expanded
  - space complexity: maximum number of nodes in memory
  - optimality: does it always find a least-cost solution?

Search strategies

- Time and space complexity are measured in terms of
  - *b:* maximum branching factor of the search tree
  - *d:* depth of the least-cost solution
  - C*: path cost of the least-cost solution
  - *m*: maximum depth of the state space (may be ∞)

Breadth-first search

- **Breadth-first search** is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then *their* successors, and so on.
- In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level.
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end
  - TREE-SEARCH(problem, FIFO-QUEUE())

Breadth-first search

Let *fringe* be a list containing the initial state

Loop

  if *fringe* is empty return *failure*

  Node ← remove-first (*fringe*)

  if Node is a *goal*

    then return the path from initial state to Node

  else generate all successors of Node, and

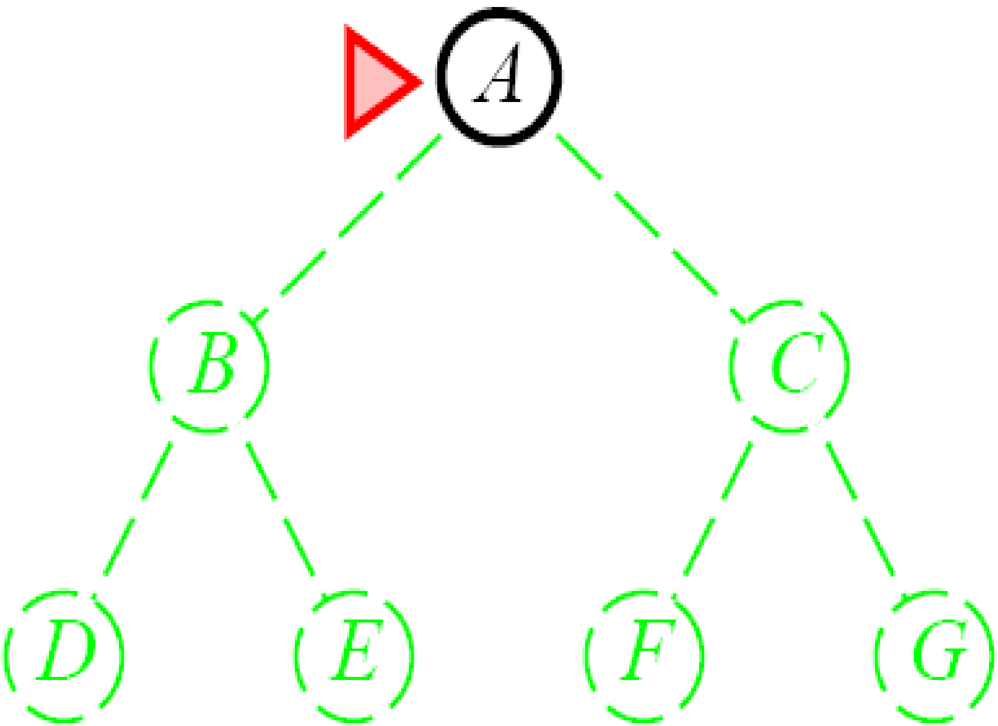
    merge the newly generated nodes into *fringe*

End Loop

Expand shallowest node first

Add generated nodes to the back of fringe

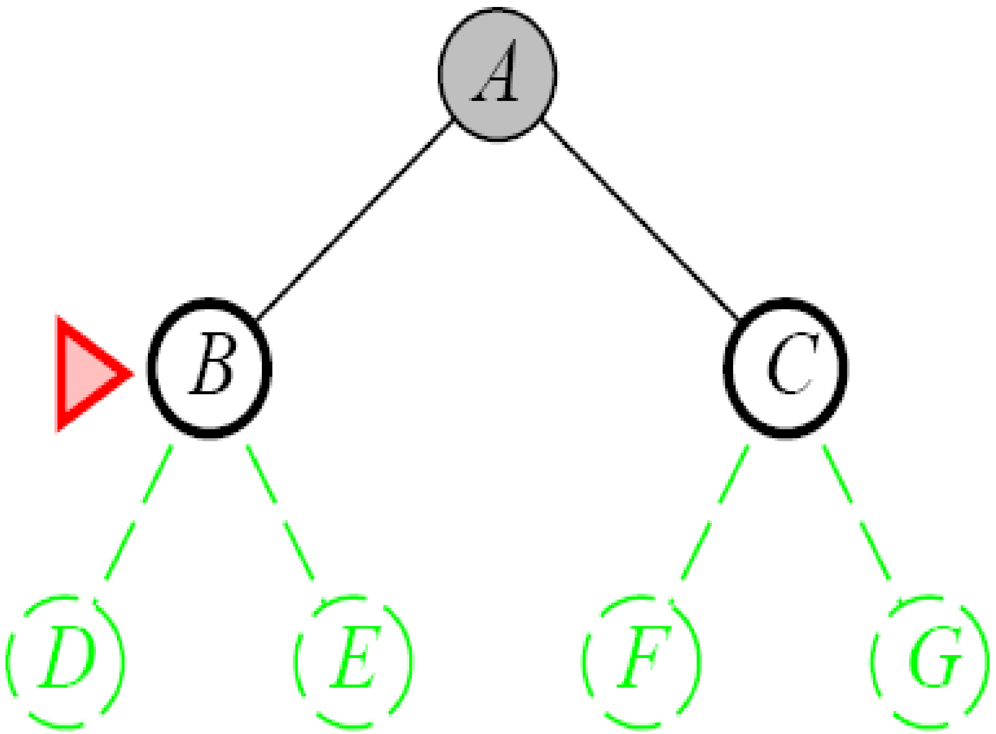
Breadth-first search



- Move downwards, level by level, until goal is reached.

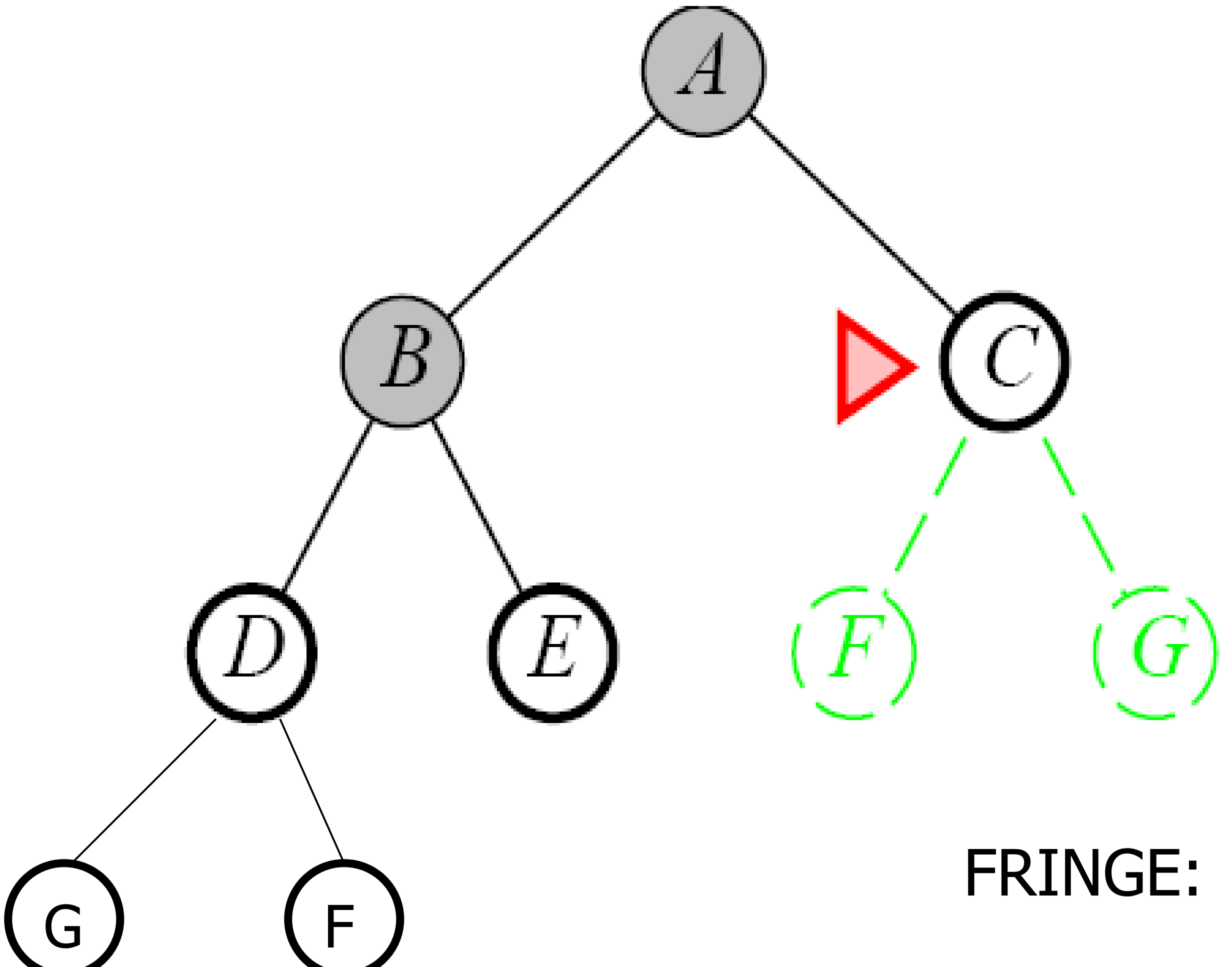
FRINGE: A

Breadth-first search



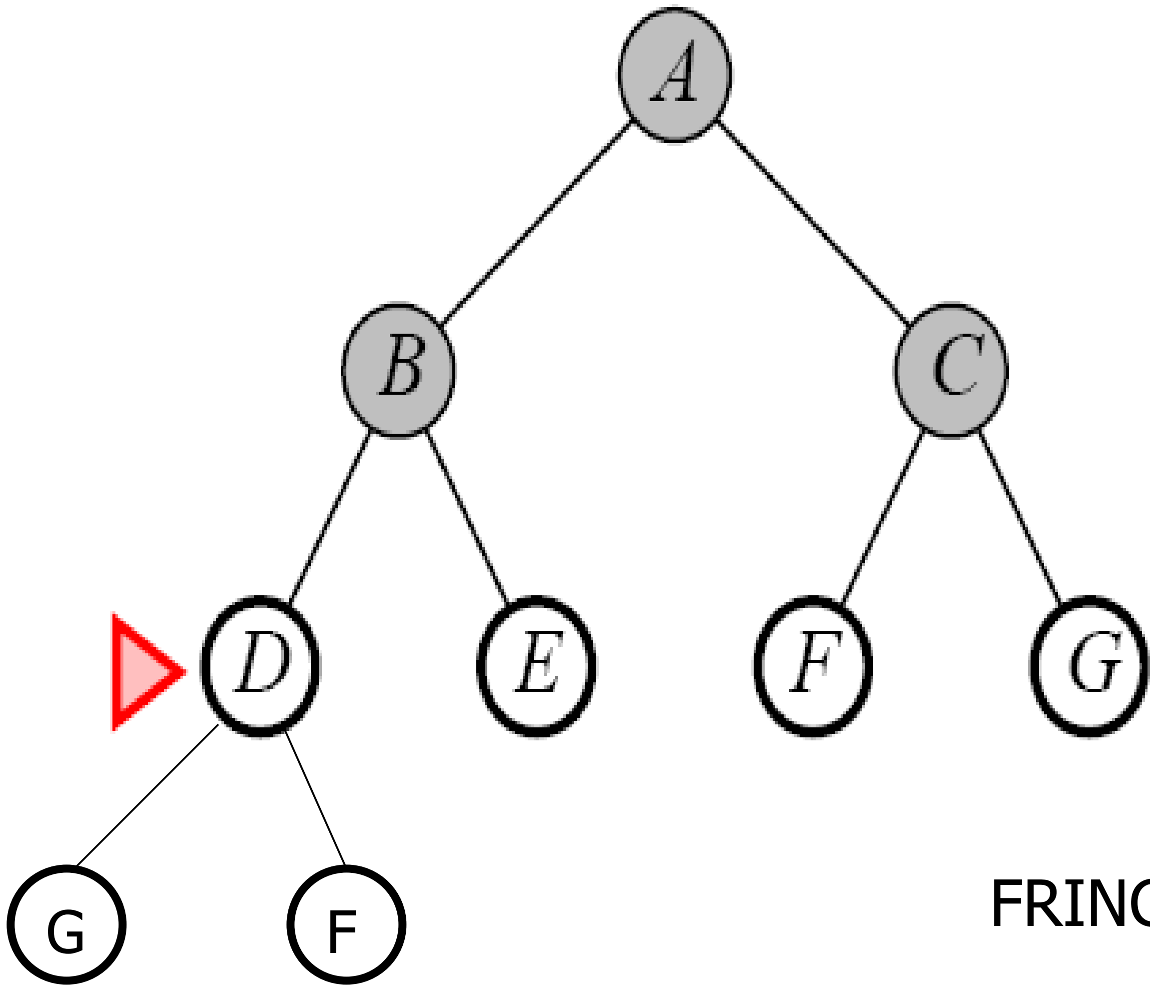
FRINGE: B C

Breadth-first search



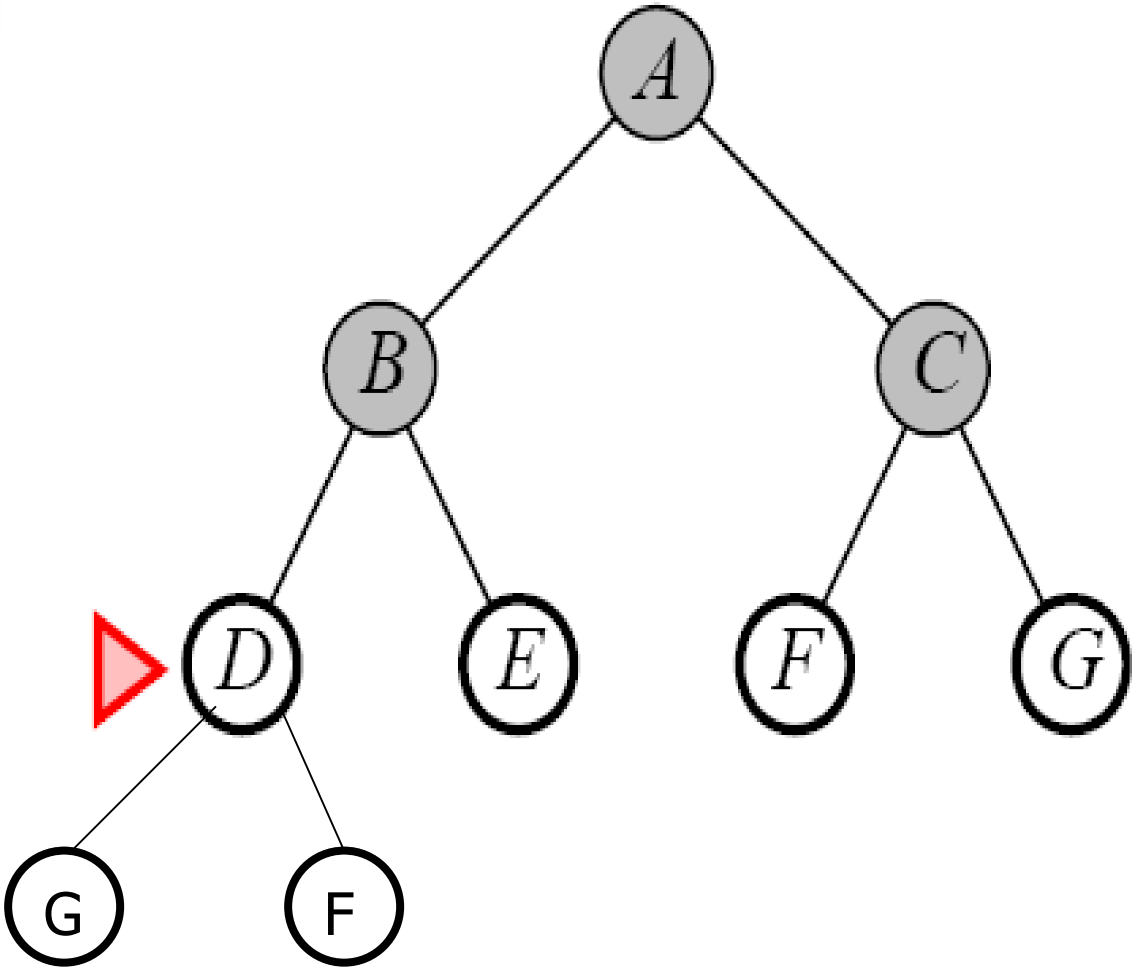
FRINGE: C D E

Breadth-first search



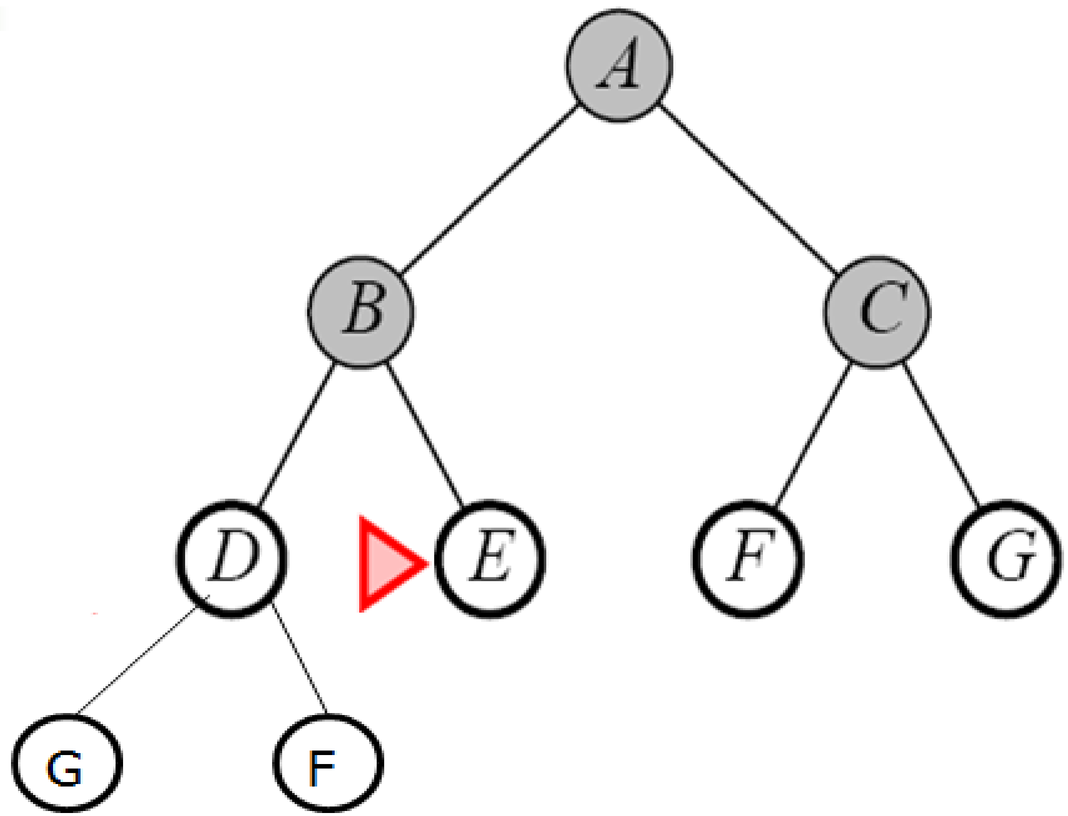
FRINGE: D E F G

Breadth-first search



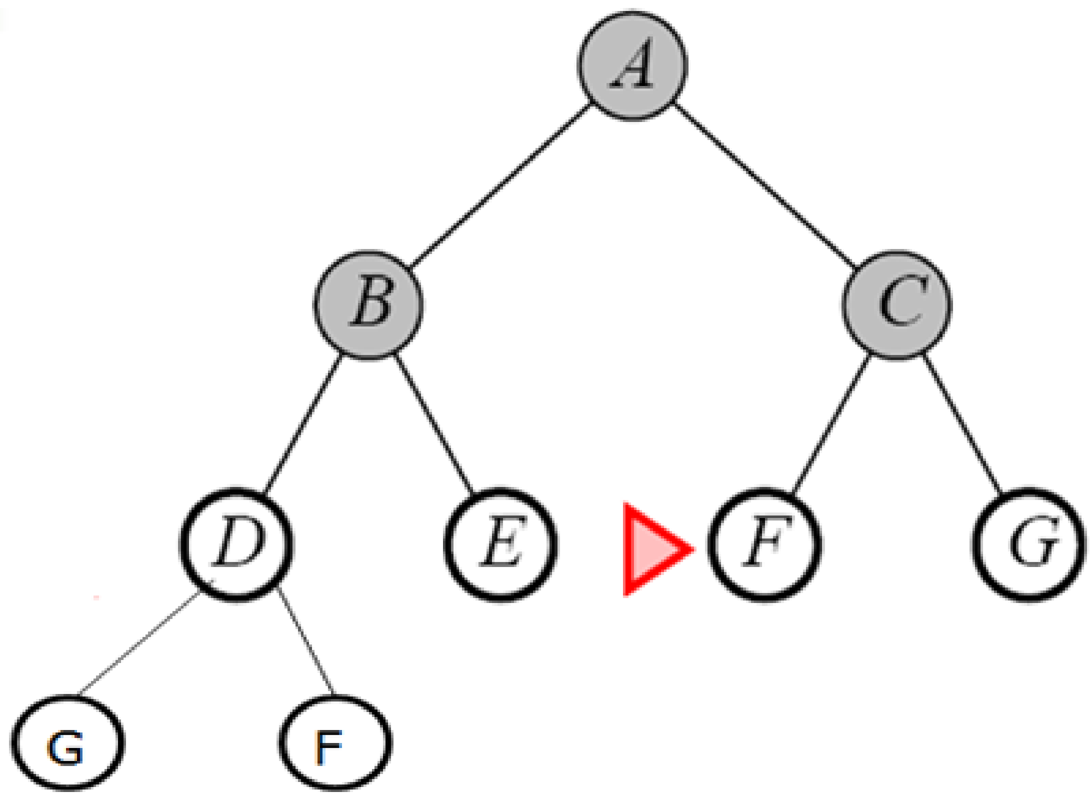
FRINGE: D E F G G F

Breadth-first search



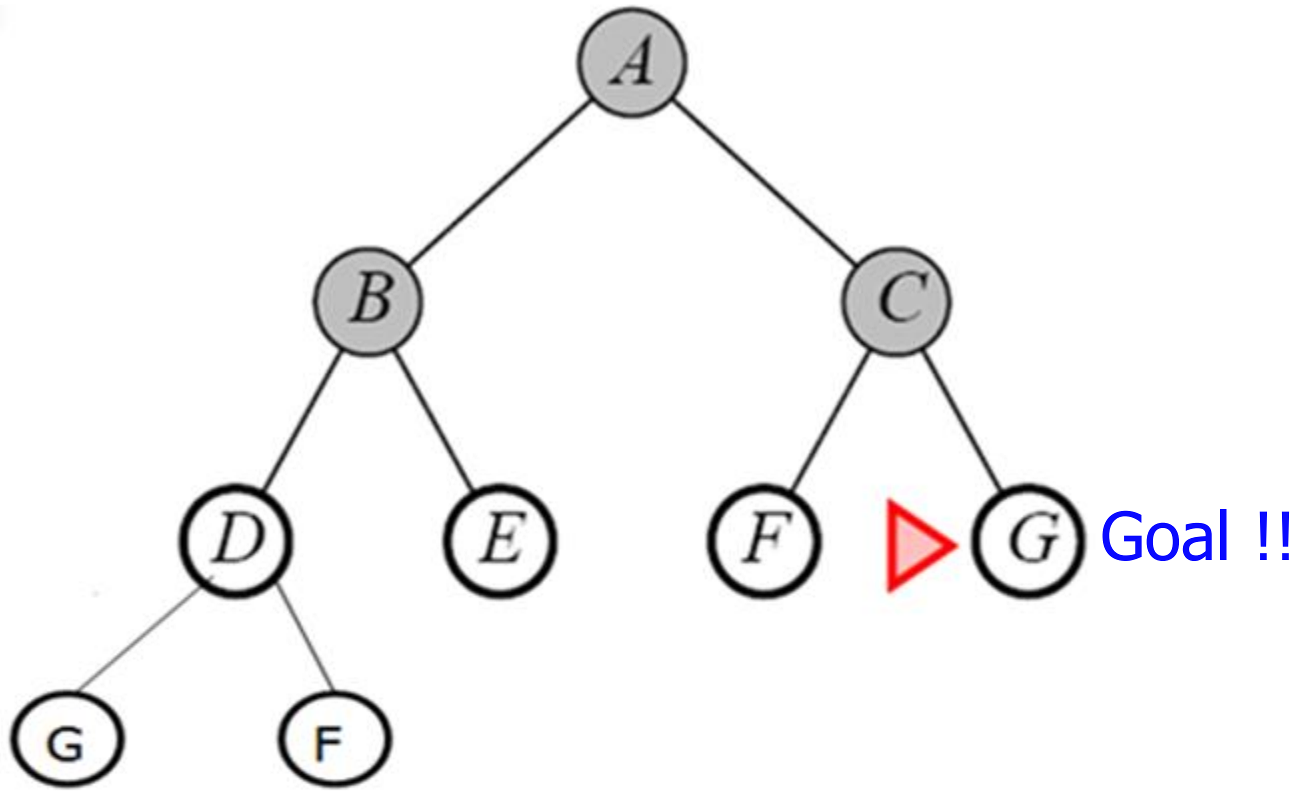
FRINGE: E F G G F

Breadth-first search



FRINGE: F G G F

Breadth-first search



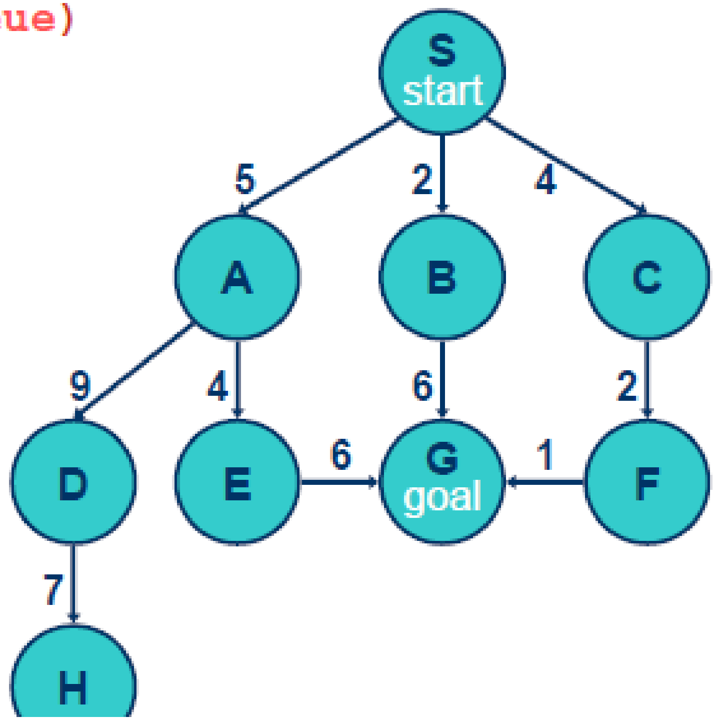
FRINGE: G C F

Breadth-first search

generalSearch(problem, queue)

# of nodes tested: 0, expanded: 0

expnd. node	nodes list
	{S}

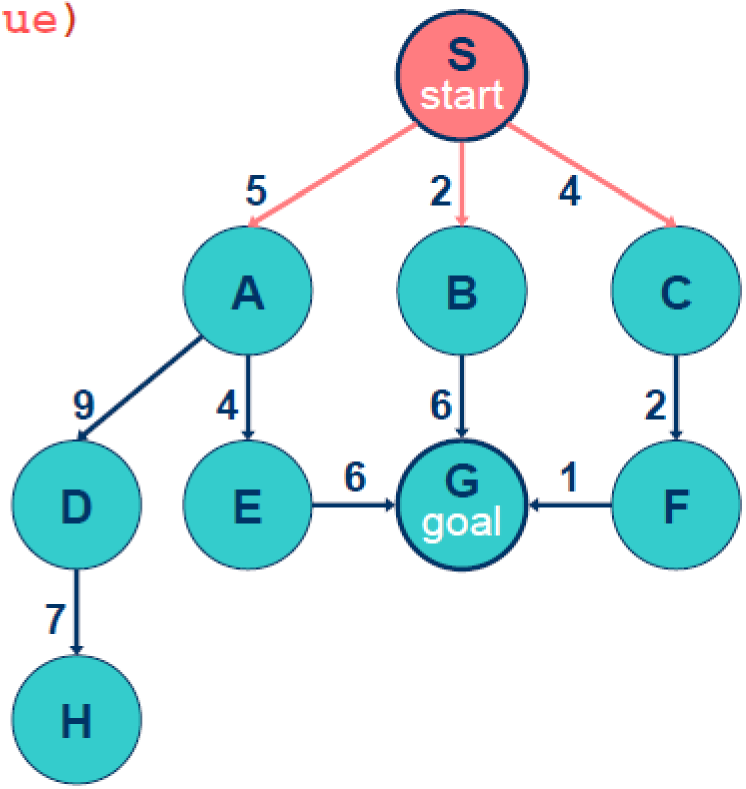


Breadth-first search

```
generalSearch(problem, queue)
```

# of nodes tested: 1, expanded: 1

expnd. node	nodes list
	{S}
S not goal	{A,B,C}

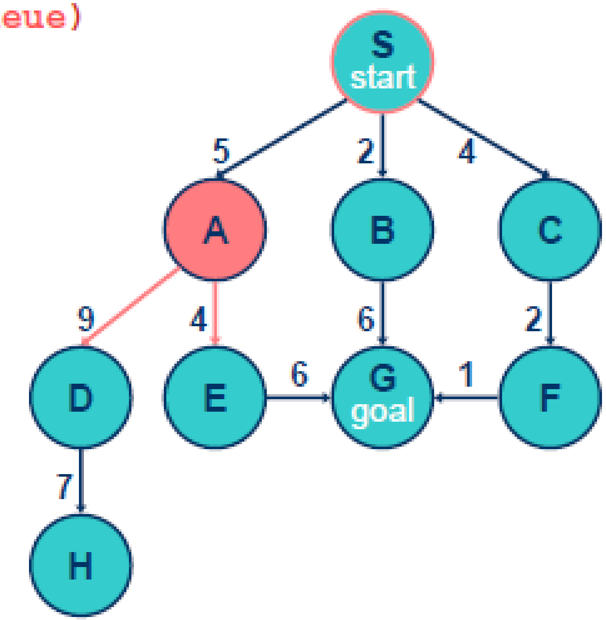


Breadth-first search

`generalSearch(problem, queue)`

# of nodes tested: 2, expanded: 2

expnd. node	nodes list
	{S}
S	{A,B,C}
A not goal	{B,C,D,E}

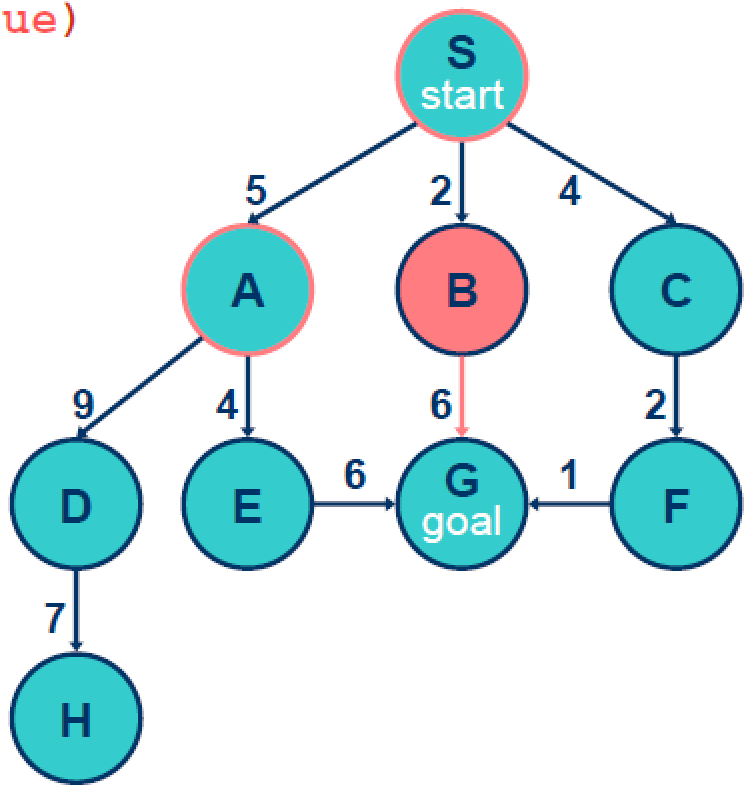


Breadth-first search

`generalSearch(problem, queue)`

# of nodes tested: 3, expanded: 3

expnd. node	nodes list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B not goal	{C,D,E,G}

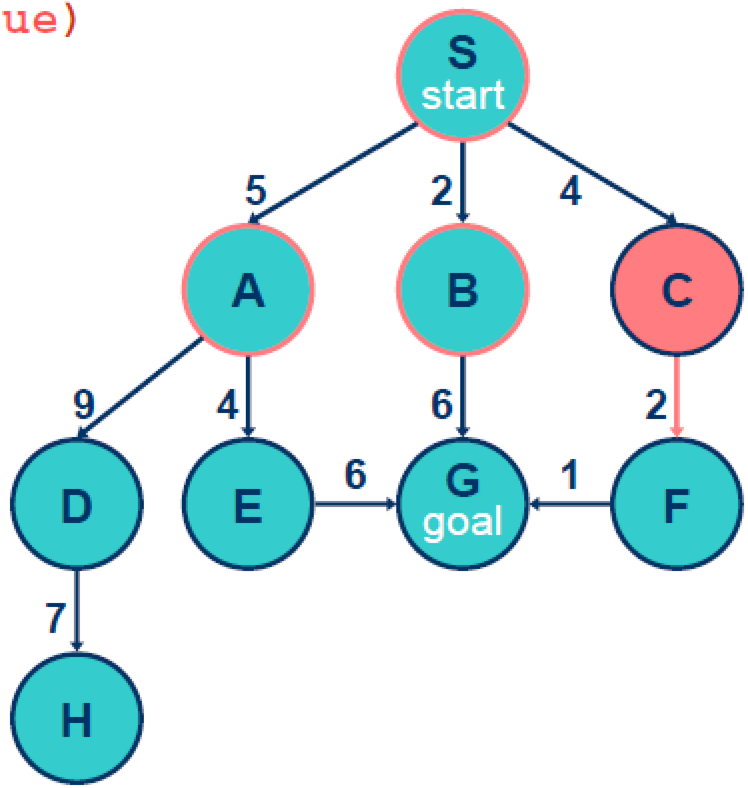


Breadth-first search

`generalSearch(problem, queue)`

# of nodes tested: 4, expanded: 4

expnd. node	nodes list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B	{C,D,E,G}
C not goal	{D,E,G,F}

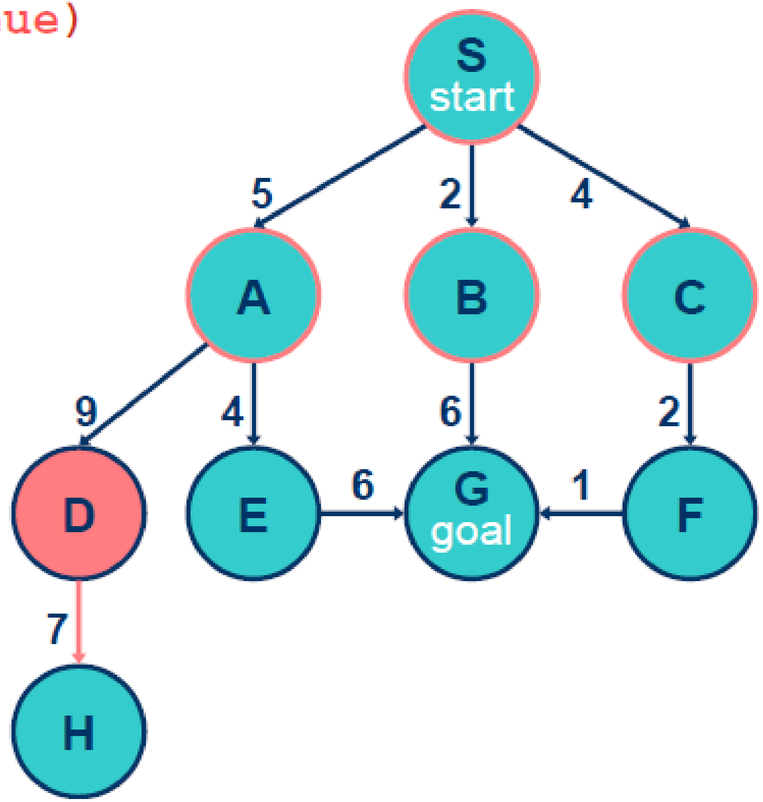


Breadth-first search

`generalSearch(problem, queue)`

# of nodes tested: 5, expanded: 5

expnd. node	nodes list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B	{C,D,E,G}
C	{D,E,G,F}
D not goal	{E,G,F,H}

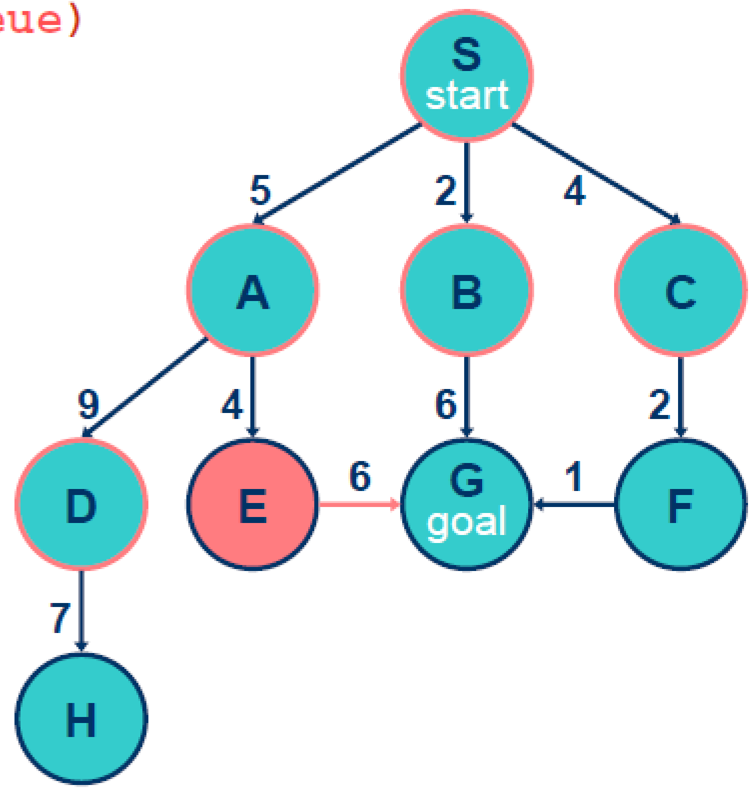


Breadth-first search

`generalSearch(problem, queue)`

# of nodes tested: 6, expanded: 6

expnd. node	nodes list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B	{C,D,E,G}
C	{D,E,G,F}
D	{E,G,F,H}
E not goal	{G,F,H,G}

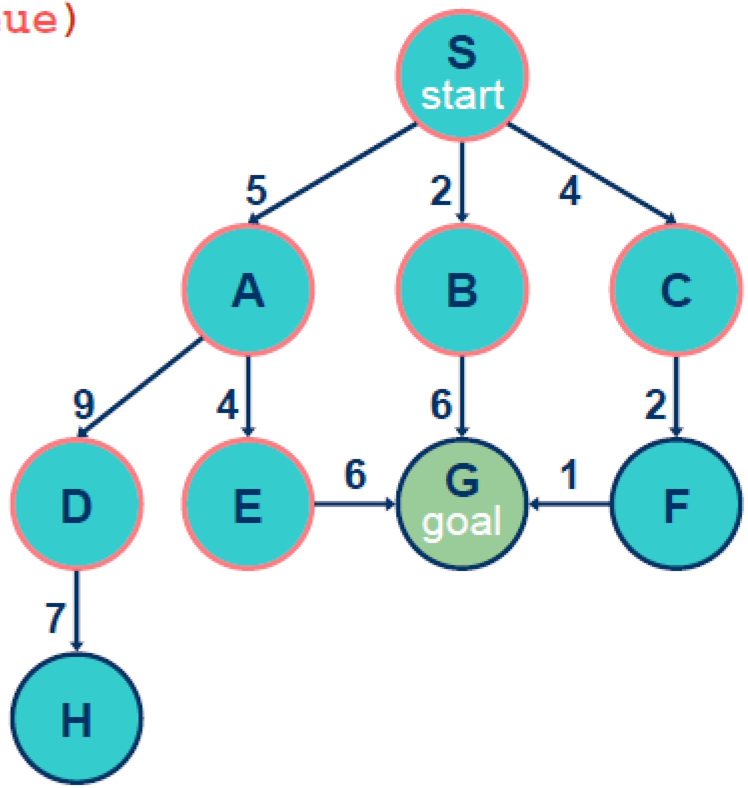


Breadth-first search

`generalSearch(problem, queue)`

# of nodes tested: 7, expanded: 6

expnd. node	nodes list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B	{C,D,E,G}
C	{D,E,G,F}
D	{E,G,F,H}
E	{G,F,H,G}
G goal	{F,H,G} no expand

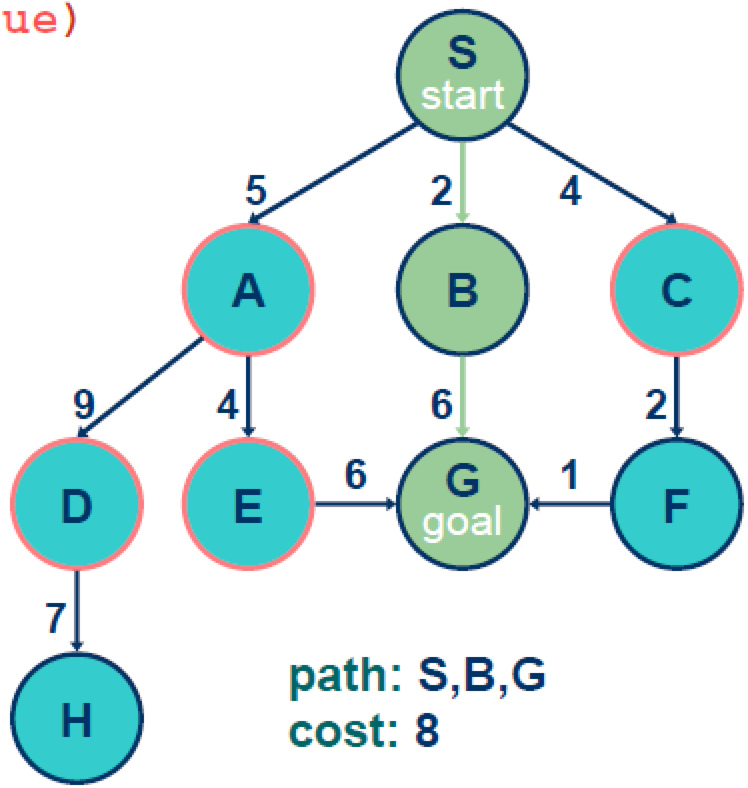


Breadth-first search

`generalSearch(problem, queue)`

# of nodes tested: 7, expanded: 6

expnd. node	nodes list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B	{C,D,E,G}
C	{D,E,G,F}
D	{E,G,F,H}
E	{G,F,H,G}
G	{F,H,G}



path: S,B,G
cost: 8

Properties of breadth-first search

- Complete? Yes (if b is finite)
- Time? $1+b+b^2+b^3+\ldots+b^d+(b^{d+1}-b) = O(b^d)$
- Space? $O(b^d)$ (keeps every node in memory)
- Optimal? Yes (if cost = 1 per step)
  No, unless step costs are constant

- Space is the bigger problem (more than time)

Time and memory requirements for breadth-first search

Depth	Nodes	Time	Memory
2	1100	.11 seconds	1 megabyte
4	111,100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabytes
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3,523 years	1 exabyte

The numbers shown assume branching factor b=10, 10,000 nodes/second, 1000 bytes/node.

- 31 hours would not be too long to wait for the solution to an important problem of depth 8, but only few computers have the terabyte of main memory.

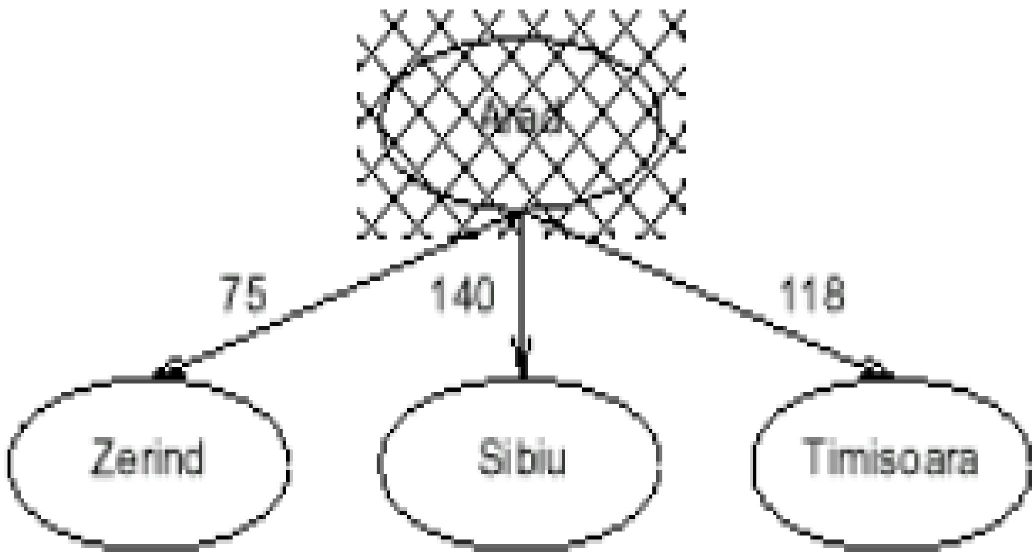
Time and memory requirements for breadth-first search

- If your problem has a solution at depth 12, then it will take 35 years for breadth-first search to find it.
- Indeed with any uninformed search, it will approximately take the same amount of time.
- So in general, exponential-complexity search problems can not be solved by uninformed methods.
- The memory requirements are a bigger problem for breadth-first search than is the execuation time.
- Time requirements are still a major factor.

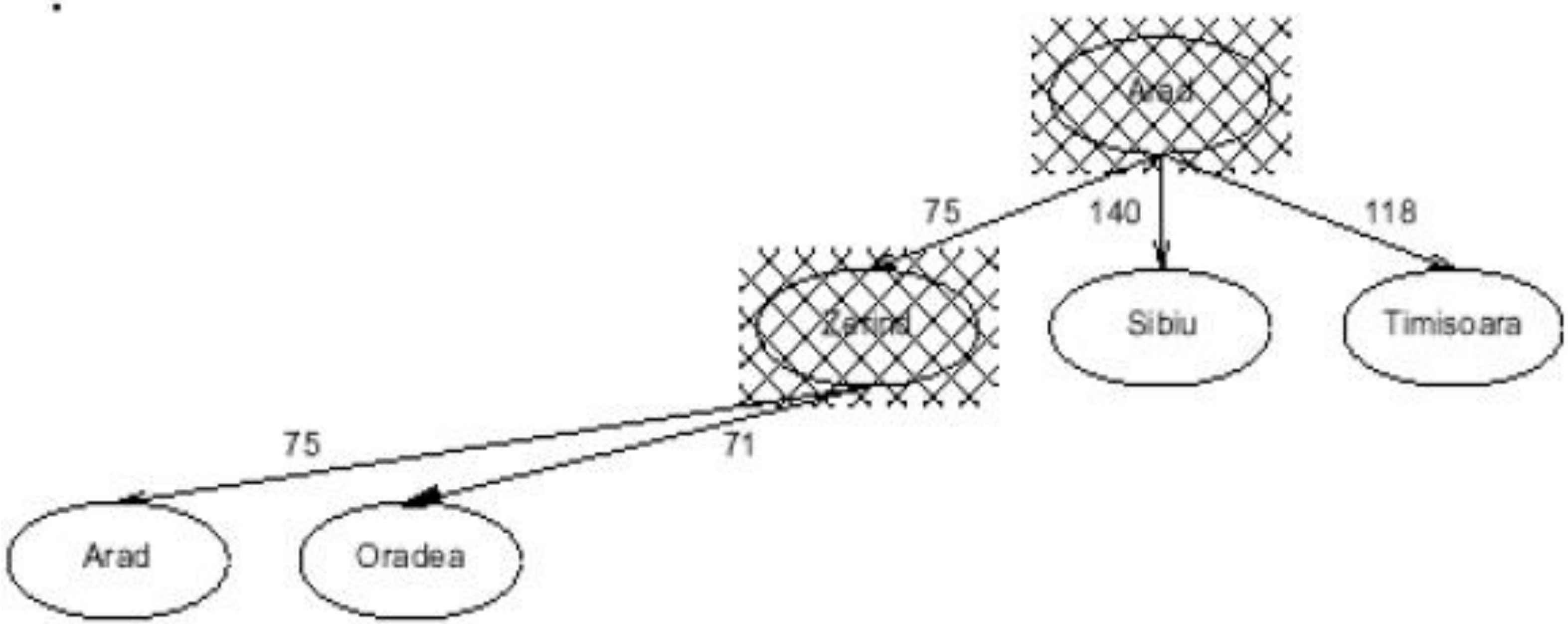
Uniform-cost search

- Uniform-cost search expands the node with the lowest path cost.
- A refinement of the breadth-first strategy:
  - Equivalent to breadth-first if step costs all equal.
- Implementation:
  - *fringe* = queue ordered by path cost

Uniform-cost search



Uniform-cost search



Uniform-cost search

- Uniform-cost search does not care about the number of steps of a path, but only about their total cost.
- Therefore, it will get stuck in an infinite loop if it never expands a node that has a zero-cost action leading back to the same state. (e.g. NoOp action)
- We can guarantee completeness provided the cost of every step is greater than or equal to some small constant ε.
  - This condition is also sufficient to ensure optimality.

Uniform-cost search

- Uniform-cost search is guided by path costs rather than depths → its complexity cannot be characterized in terms of b and d.
- Instead, evaluate complexity in terms of C^* (the cost of the optimal solution) and ε (assuming that every action costs at least ε)

Uniform-cost search

- Complete? Yes, if step cost ≥ ε
- Time? # of nodes with $g \leq C^*$, $O(b^{\lceil C^*/\varepsilon \rceil})$ where C^* is the cost of the optimal solution
- Space? # of nodes with $g \leq C^*$, $O(b^{\lceil C^*/\varepsilon \rceil})$
- Optimal? Yes – nodes expanded in increasing order of $g(n)$

When all step costs are equal, $b^{\lceil C^*/\varepsilon \rceil}$ is just b^d

Depth-first search

- Expand deepest unexpanded node
- Search proceeds immediately to the deepest level of the search tree → nodes have no successors
- As those nodes are expanded, they are dropped from the fringe

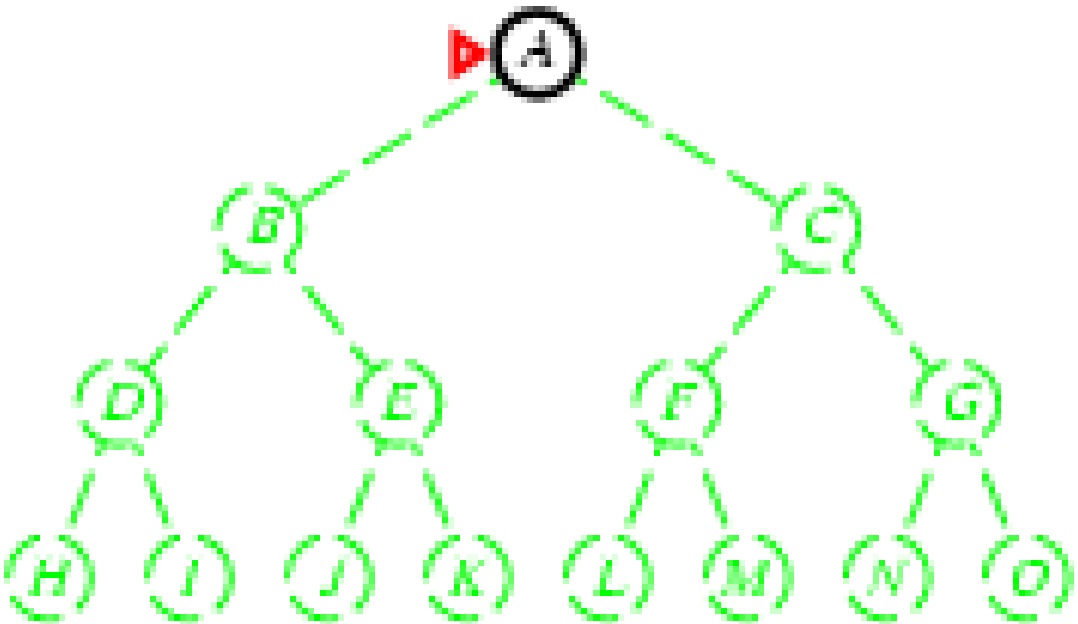
Depth-first search

```
Let fringe be a list containing the initial state
Loop
        if fringe is empty return failure
        Node ← remove-first (fringe)
        if Node is a goal
            then return the path from initial state to Node
        else generate all successors of Node, and
              merge the newly generated nodes into fringe
End Loop
```

Expand deepest node first
Add generated nodes to the front of fringe

Depth-first search

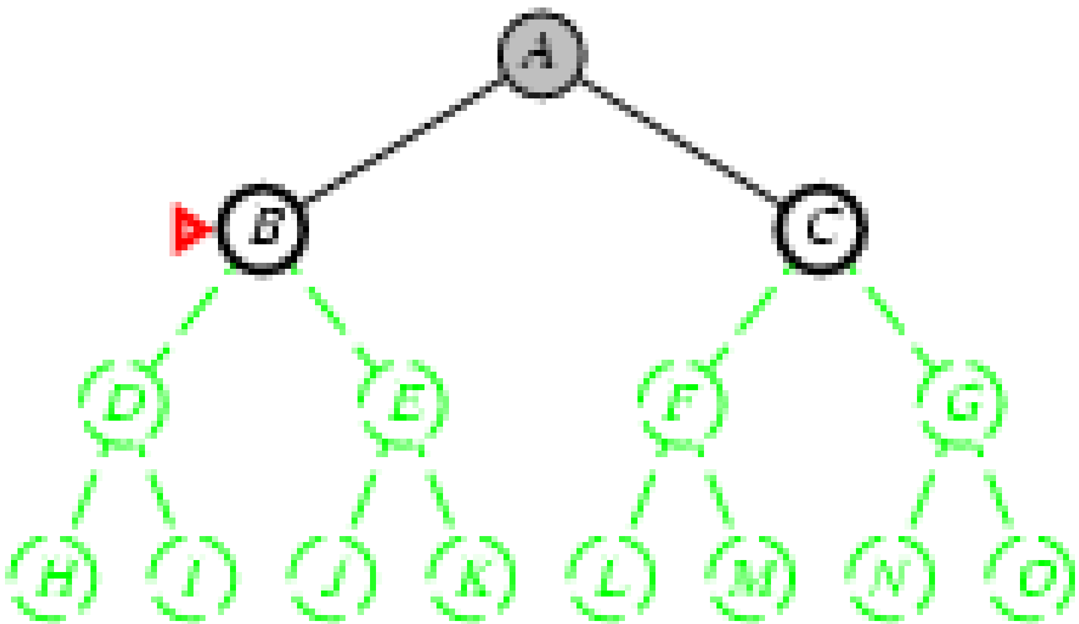
- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



FRINGE: A

Depth-first search

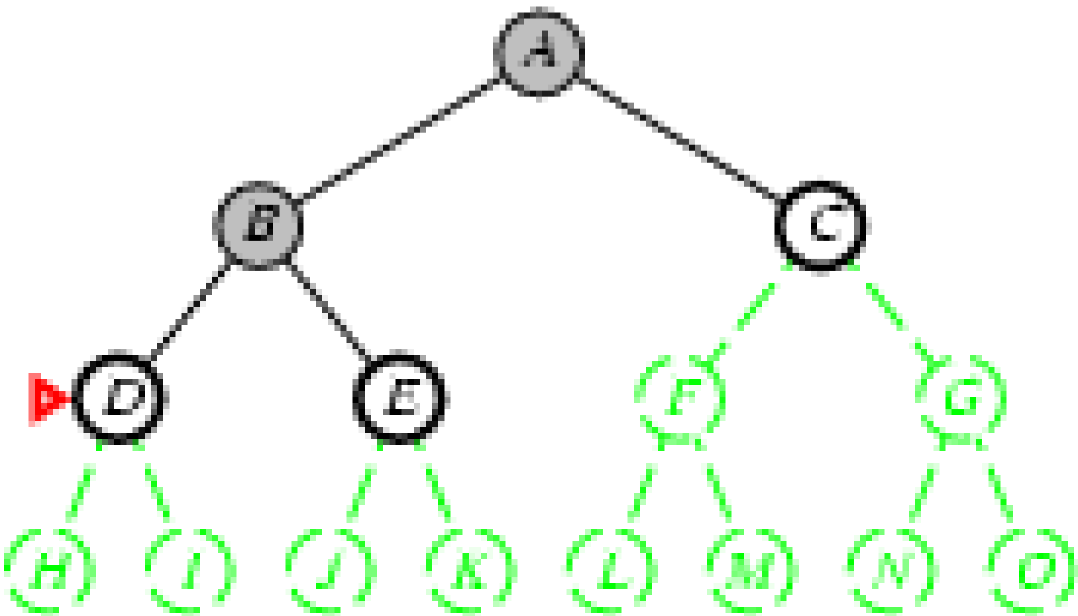
- Expand deepest unexpanded node
  - convention: left-to-right
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



FRINGE: B C

Depth-first search

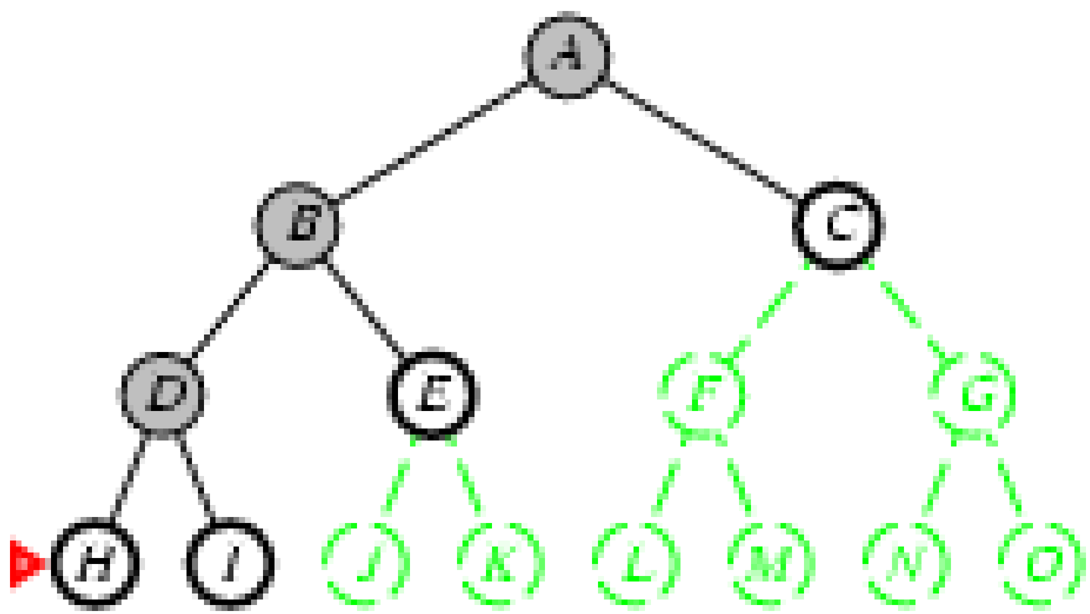
- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



FRINGE: D E C

Depth-first search

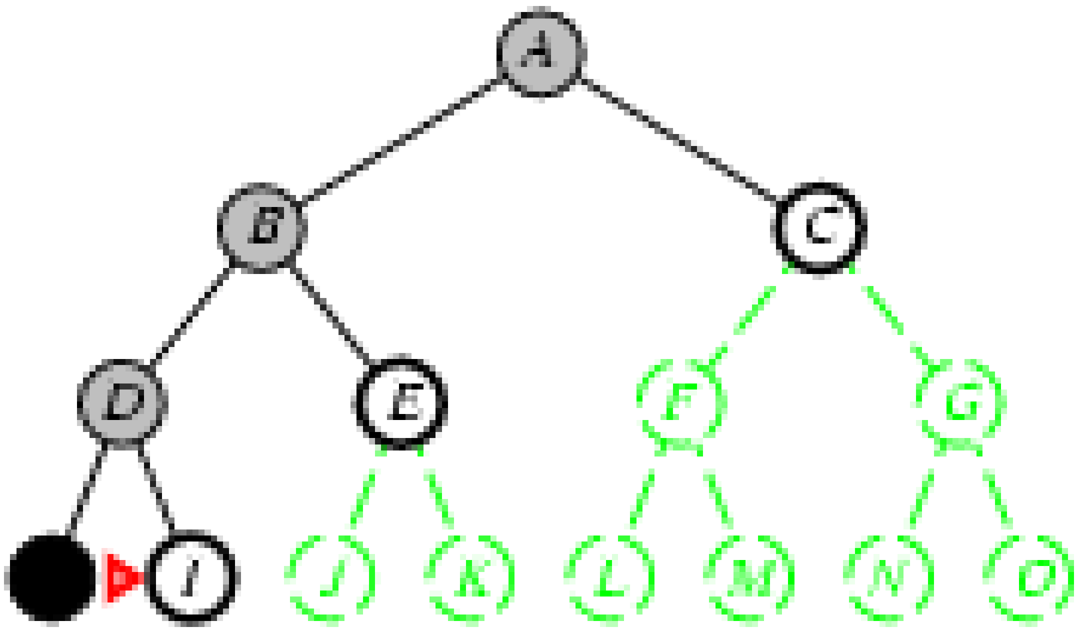
- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



FRINGE: H I E C

Depth-first search

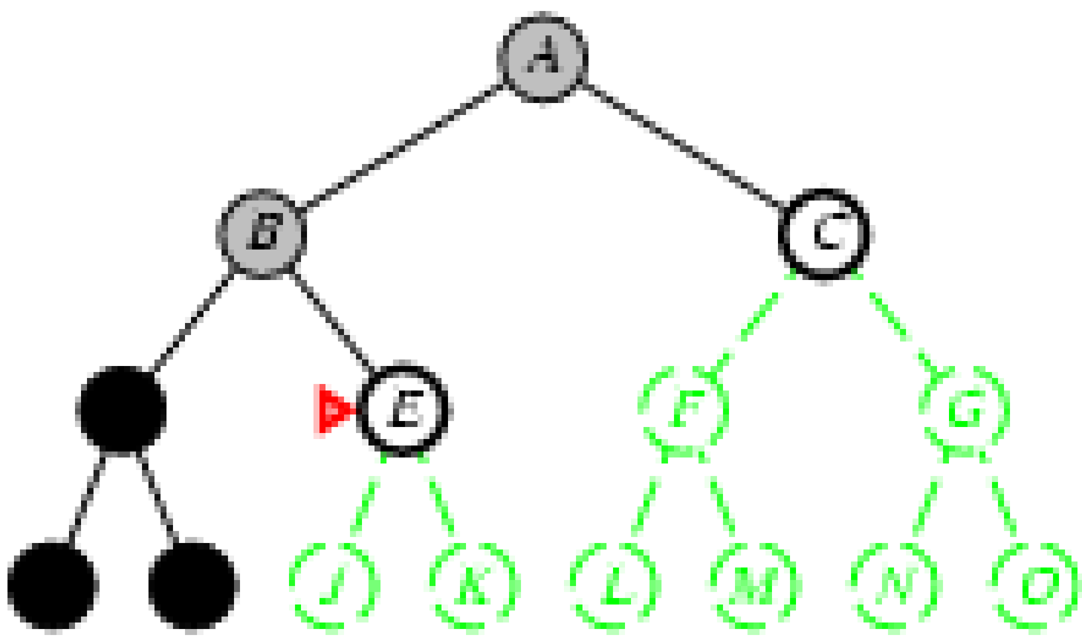
- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



FRINGE: I E C

Depth-first search

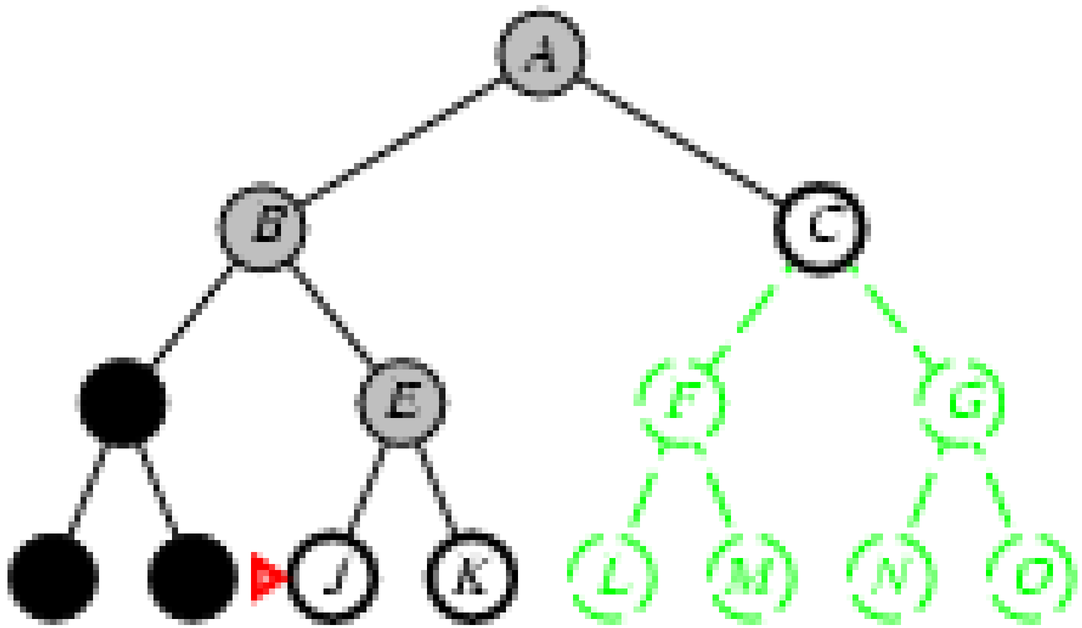
- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



FRINGE: E C

Depth-first search

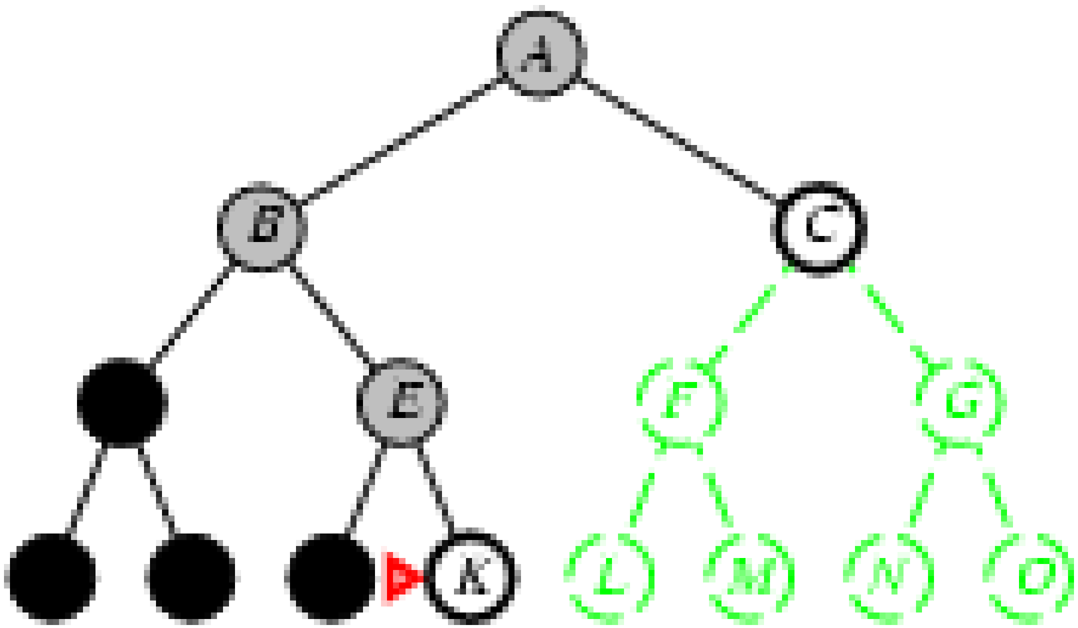
- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



FRINGE: J K C

Depth-first search

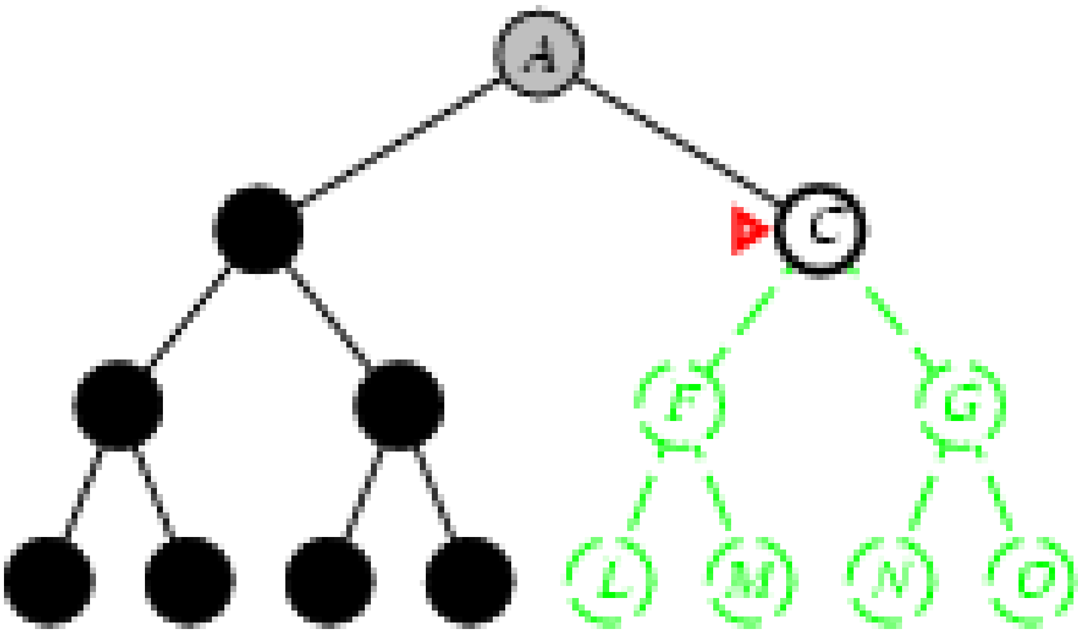
- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



FRINGE: K C

Depth-first search

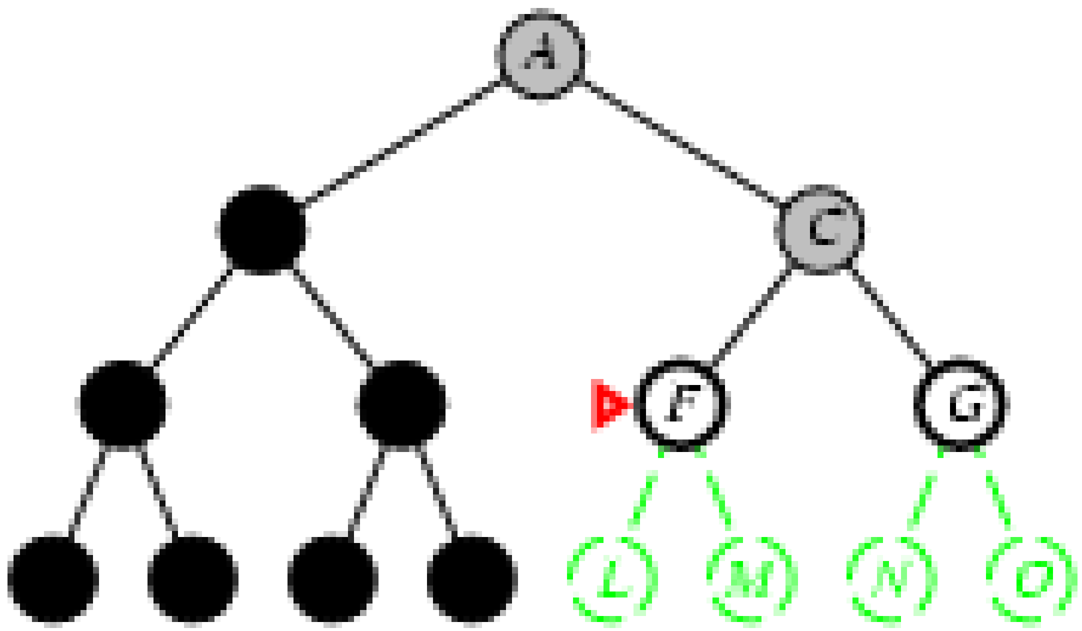
- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



FRINGE: C

Depth-first search

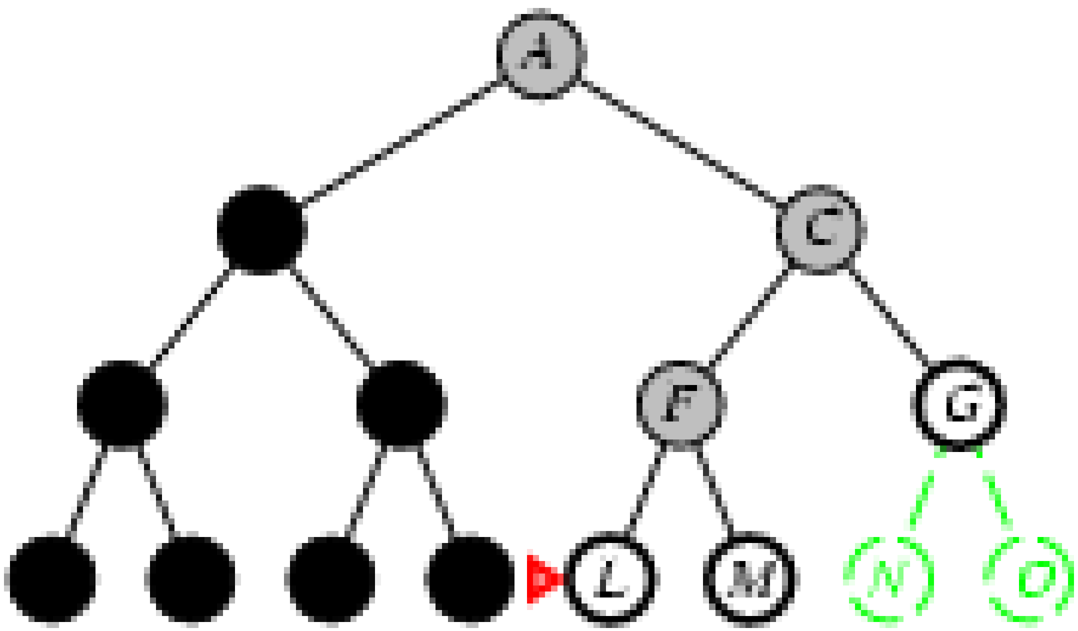
- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



FRINGE: F G

Depth-first search

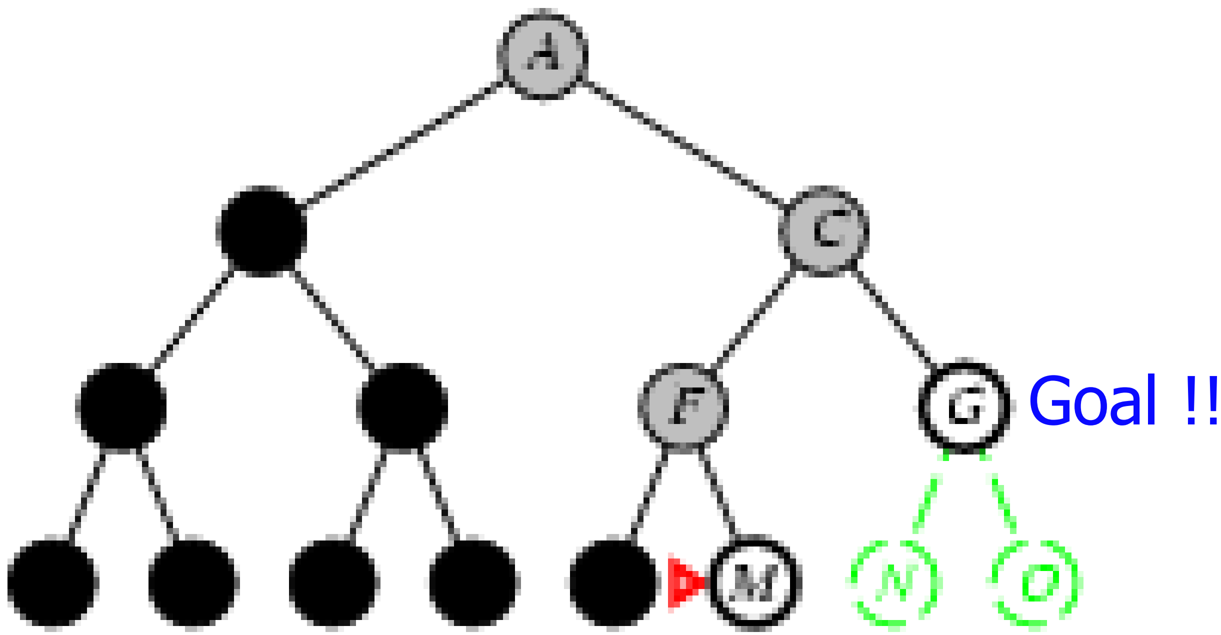
- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



FRINGE: L M G

Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



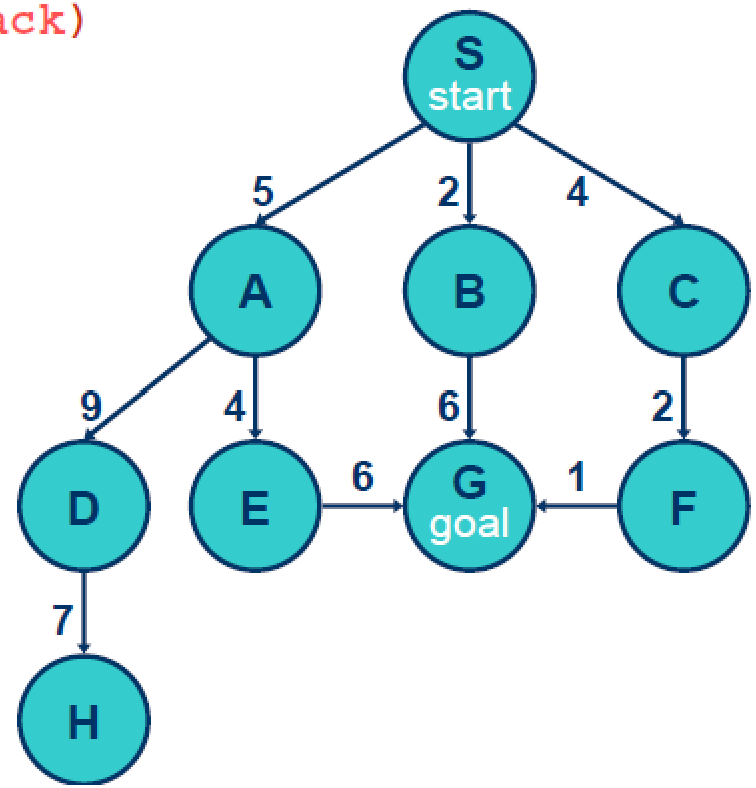
FRINGE: ~~M~~ G

Depth-first

`generalSearch(problem, stack)`

# of nodes tested: 0, expanded: 0

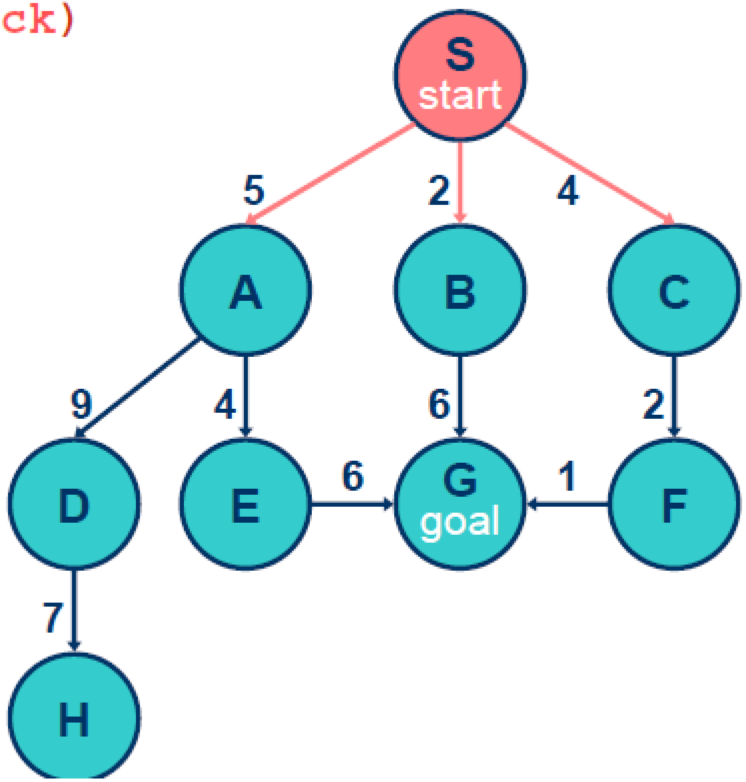
expnd. node	nodes list
	{S}



```
generalSearch(problem, stack)
```

# of nodes tested: 1, expanded: 1

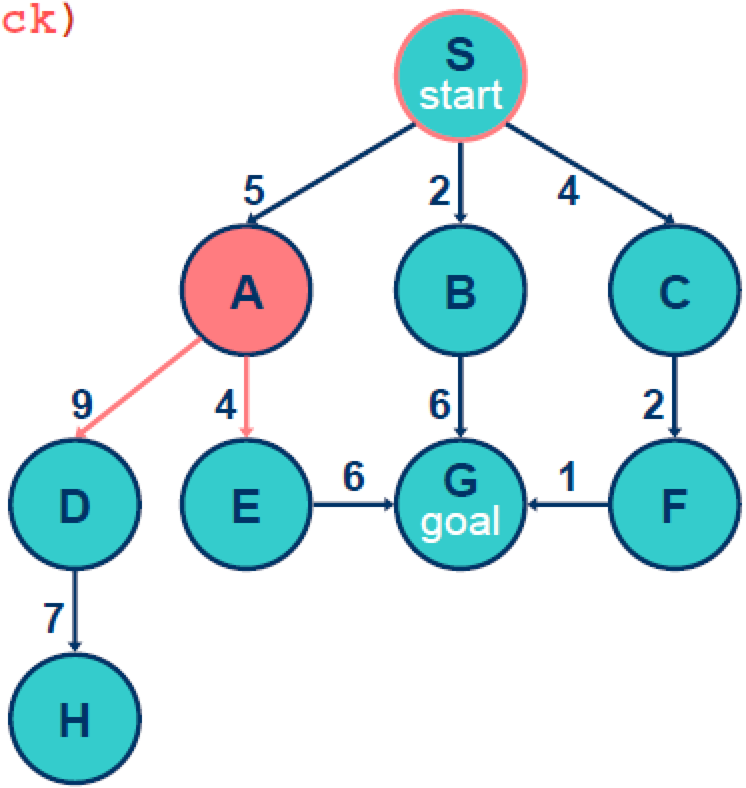
expnd. node	nodes list
	{S}
S not goal	{A,B,C}



```
generalSearch(problem, stack)
```

# of nodes tested: 2, expanded: 2

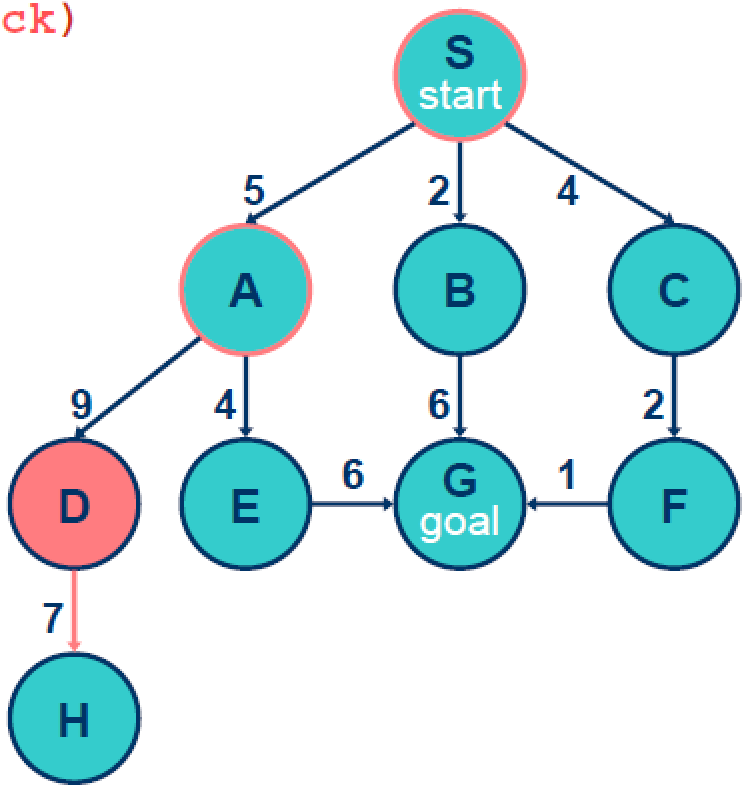
expnd. node	nodes list
	{S}
S	{A,B,C}
A not goal	{D,E,B,C}



```
generalSearch(problem, stack)
```

# of nodes tested: 3, expanded: 3

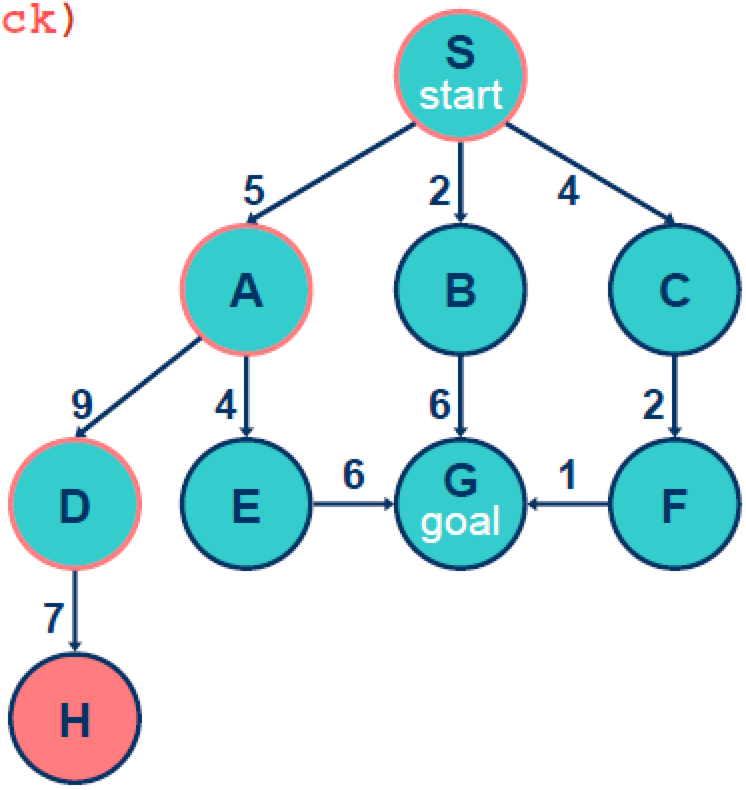
expnd. node	nodes list
	{S}
S	{A,B,C}
A	{D,E,B,C}
D not goal	{H,E,B,C}



`generalSearch(problem, stack)`

\# of nodes tested: 4, expanded: 4

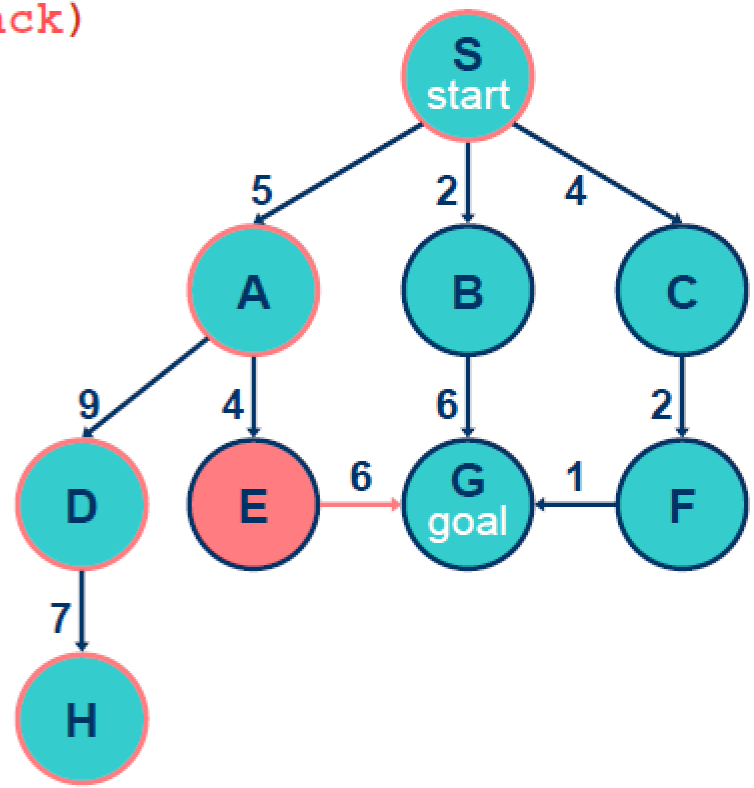
expnd. node	nodes list
	{S}
S	{A,B,C}
A	{D,E,B,C}
D	{H,E,B,C}
H not goal	{E,B,C}



`generalSearch(problem, stack)`

# of nodes tested: 5, expanded: 5

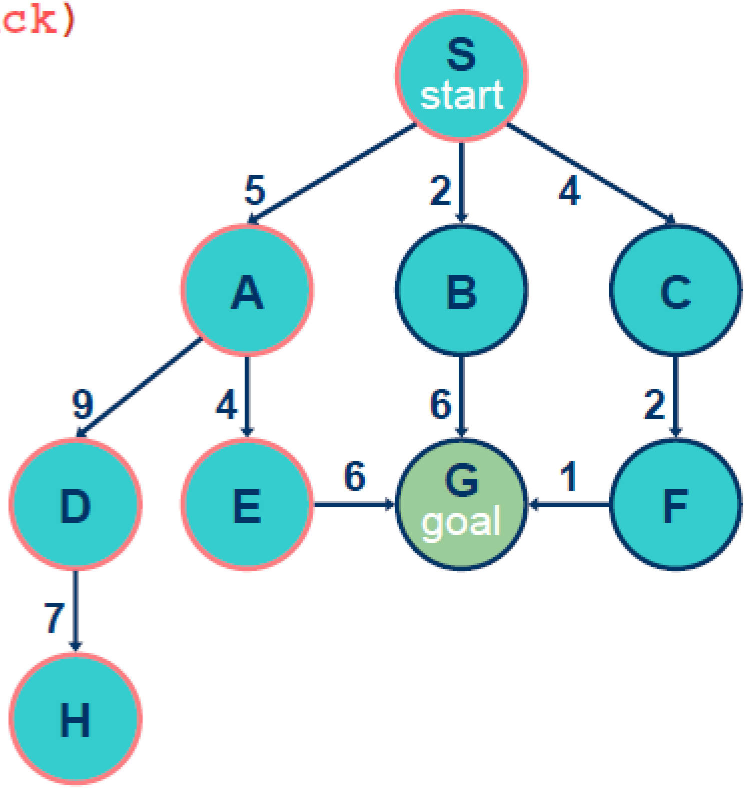
expnd. node	nodes list
	{S}
S	{A,B,C}
A	{D,E,B,C}
D	{H,E,B,C}
H	{E,B,C}
E not goal	{G,B,C}



`generalSearch(problem, stack)`

# of nodes tested: 6, expanded: 5

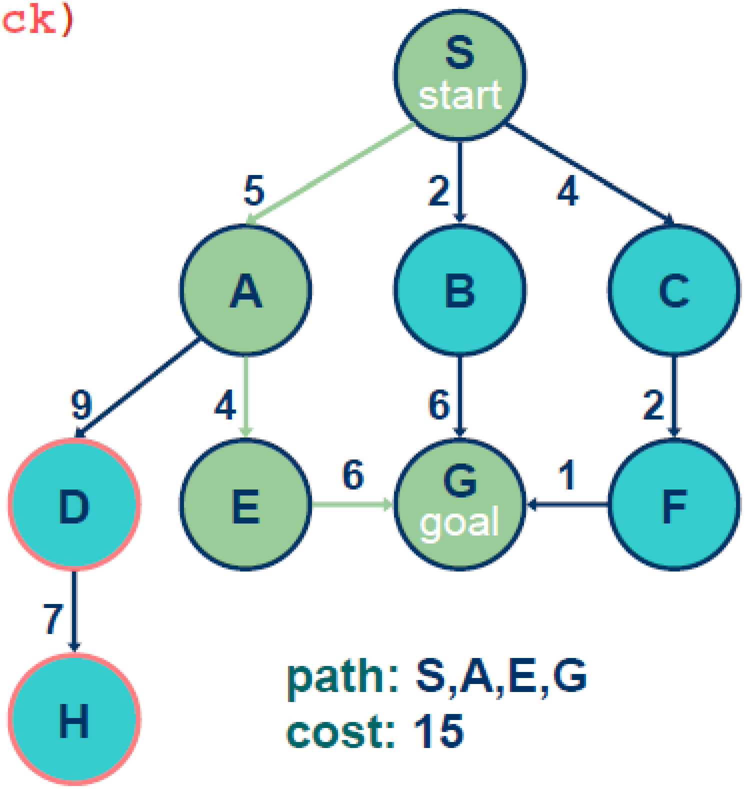
expnd. node	nodes list
	{S}
S	{A,B,C}
A	{D,E,B,C}
D	{H,E,B,C}
H	{E,B,C}
E	{G,B,C}
G goal	{B,C} no expand



`generalSearch(problem, stack)`

# of nodes tested: 6, expanded: 5

expnd. node	nodes list
	{S}
S	{A,B,C}
A	{D,E,B,C}
D	{H,E,B,C}
H	{E,B,C}
E	{G,B,C}
G	{B,C}



path: S,A,E,G
cost: 15

Depth-first search

- Depth-first search has very modest memory requirements.
- It needs to store only a single path from the root to a leaf node
- Once a node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored.

Properties of depth-first search

- Complete? No.
  - if the left subtree were of unbounded depth but contained no solutions, depth-first search would never terminate.
- Time? $O(b^m)$ → *m* is max depth of state space
  - terrible if *m* is much larger than *d*
- Space? $O(bm)$, i.e., linear space!
- Optimal? No
  - e.g. if node C is a goal node, depth-first search will explore the entire left subtree.

Backtracking Search

- A variant of depth-first search → uses still less memory
- Only one successor is generated at a time
- Each partially expanded node remembers which successors to generate next.
  - Only $O(m)$ memory is needed rather than $O(bm)$
  - It is used for problems with large state descriptions e.g. robotic assembly

Depth-limited search

- The problem of unbounded trees can be solved by depth-first search with a predetermined depth limit *l* – depth-limited search.
- **Implementation** : nodes at depth *l* have no successors.
- Unfortunately, if we choose *l* <d, the goal state will be beyond the depth limit.
- Optimal: it does not guarantee to find the least-cost solution, i.e. if we choose *l* >d.

Depth-limited search

- Depth-limits can be based on the knowledge of the problem.
  - e.g. on the map of Romania there are 20 cities. Therefore, we know that if there is a solution, it must be length 19 at the longest, so l =19 is a possible choice.
  - in fact, any city can be reached from any other city in at most 9 steps.
- This number- **diameter-** gives us a better limit.
- However, for most problems we will not know a good depth limit until we have solved the problem.

Depth-limited search

- Complete: if cutoff chosen appropriately then it is guaranteed to find a solution.
- Optimal: it does not guarantee to find the least-cost solution

Iterative deepening search

- Combines the best of breadth-first and depth-first search strategies.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or fail-
ure
    inputs: problem, a problem

    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```

\- Repeatedly applies depth-limited search with increasing limits. It terminates when a solution is found or if the depth-limited search returns *failure*, meaning that no solution exists.

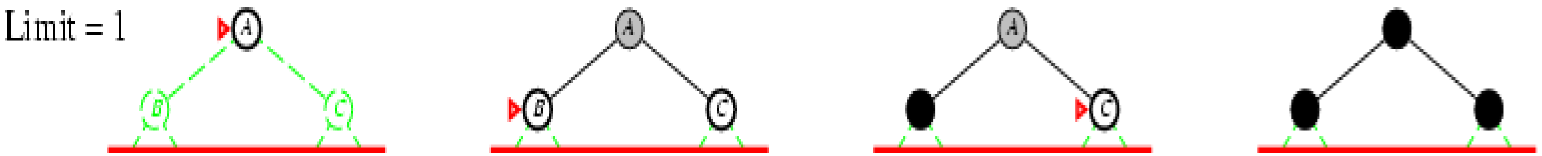
Iterative deepening search

- Iterative deepening search may seem wasteful because so many states are expanded multiple times.
- In practice, however, the overhead of these multiple expansions is small, because most of the nodes are towards leafs (bottom) of the search tree.
- thus, the nodes that are evaluated several times (towards top of tree) are in relatively small number.
- In iterative deepening, nodes at bottom level are expanded once, level above twice, etc. up to root (expanded d+1 times) so total number of expansions is:

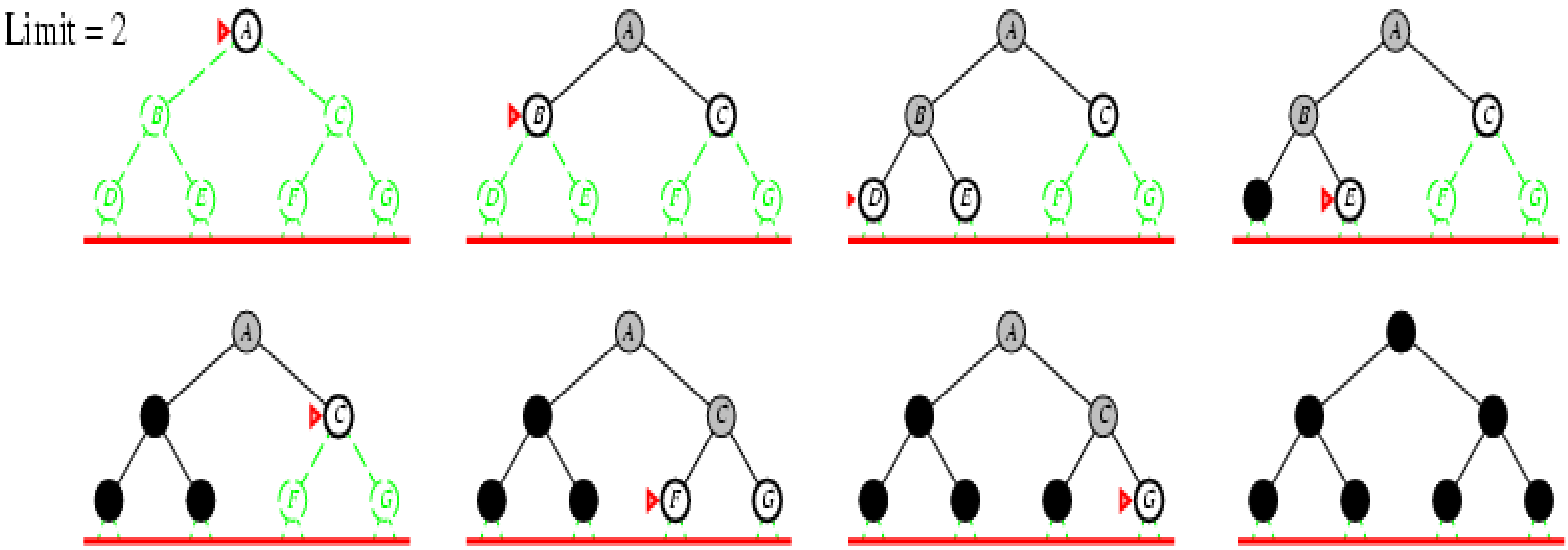
$$N(IDS)=(d+1)b^0+(d)b+(d-1)b^2+\ldots+(1)b^d = O(b^d)$$

Iterative deepening search *l* =0

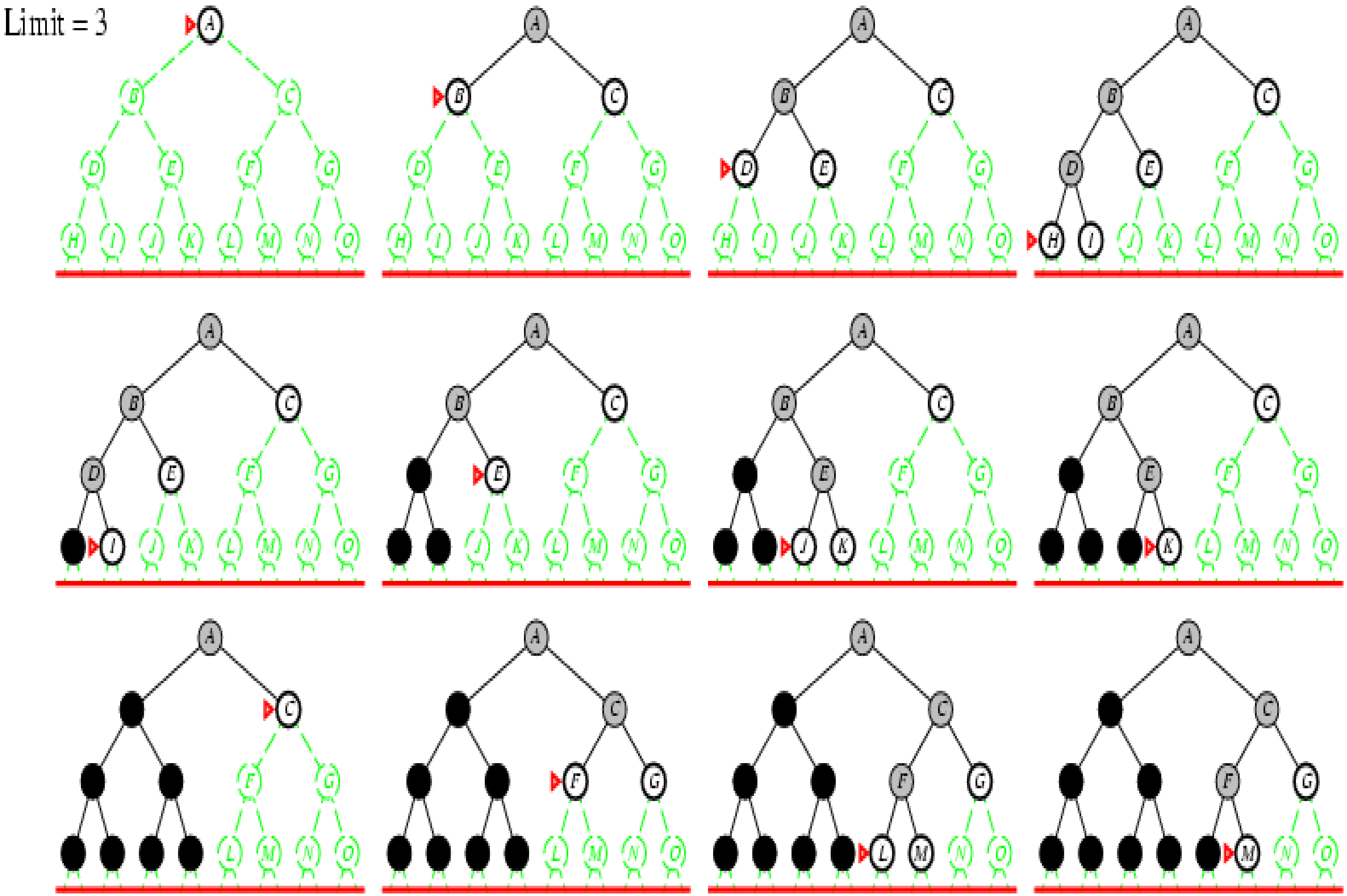
Iterative deepening search $l = 1$



Iterative deepening search *l* =2



Iterative deepening search *l* =3



Iterative deepening search

- Number of nodes generated in a breadth-first search to depth *d* with branching factor *b*:

$$N(BFS) = b^0 + b^1 + b^2 + \ldots + b^d + (b^{d+1} - b)$$

- Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:

$$N(IDS) = (d+1)b^0 + (d)\, b + (d-1)b^2 + \ldots + (1)b^d$$

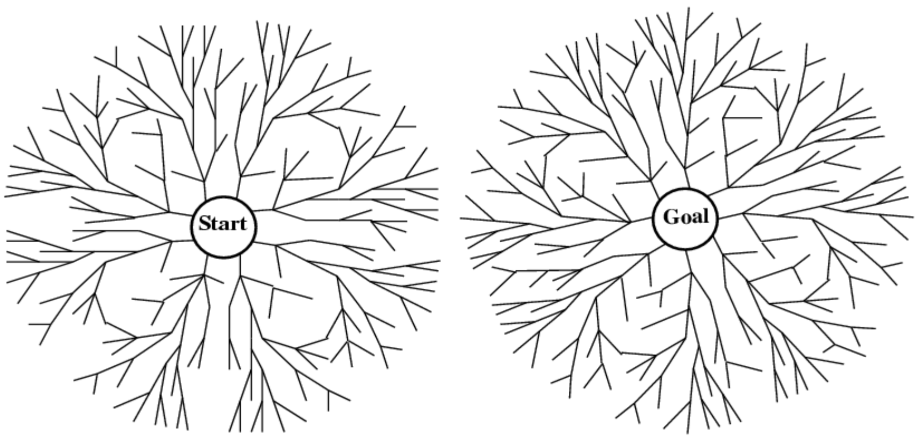
- For $b = 10$, $d = 5$,
  - **N(BFS) =** 1 + 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,101
  - **N(IDS) =** 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456
- Overhead = (123,456 - 111,111)/111,111 = 11%

Properties of iterative deepening search

- Complete? Yes
- Time? $(d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space? $O(bd)$
- Optimal? No, unless step costs are constant

Bidirectional search

- Both search forward from initial state, and backwards from goal.
- Stop when the two searches meet in the middle.
- The motivation is that $b^{d/2} + b^{d/2} < b^d$



  - The area of the two small circles is less than the area of one big circle centered on the start and reaching to the goal.

Bidirectional search

- **Problem:**

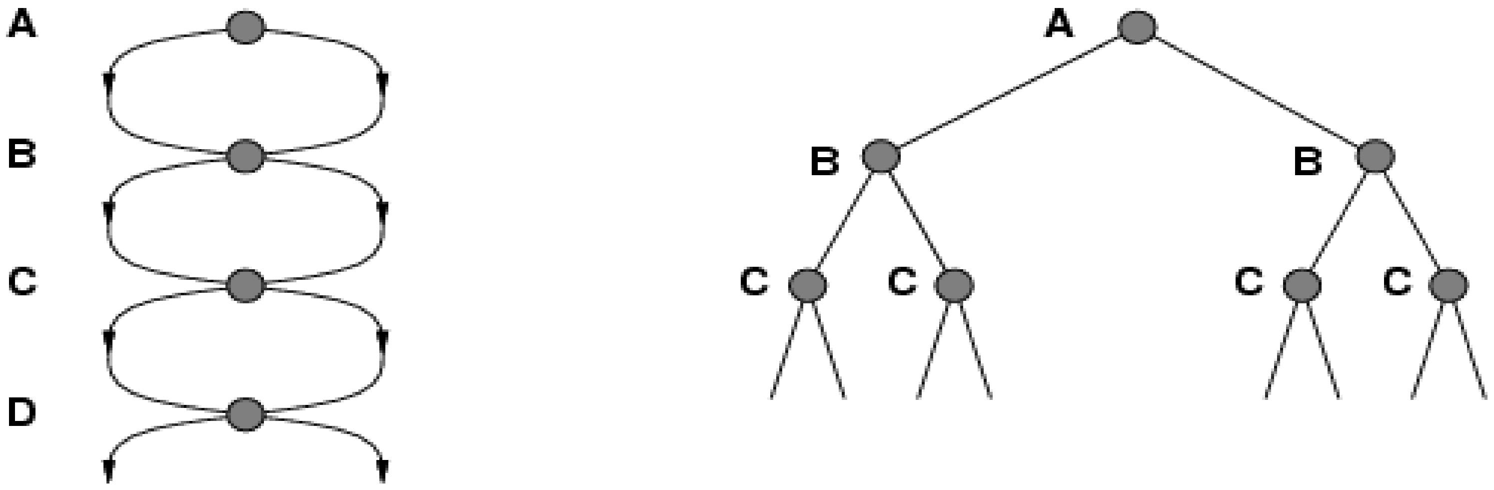
  how do we search backwards from goal??
- predecessor of node n = all nodes that have n as successor, this may not always be easy to compute!
- if several goal states, apply predecessor function to them just as we applied successor (only works well if goals are explicitly known, may be difficult if goals only characterized implicitly).

Summary of algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	Yes, if $l \geq d$	Yes
Time	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes	Yes	No	No	Yes

Avoiding repeated steps

- Up to this point, we have ignored one of the most important complications to the search process:
  - The possibility of wasting time by expanding states that have already been expanded before.
- Failure to detect repeated states can turn a linear problem into an exponential one!



Avoiding repeated states

In increasing order of effectiveness and computational overhead:

- do not return to state we come from.
- do not create paths with cycles.
- do not generate any state that was ever generated before.

Graph search

- If an algorithm remembers every state that it has visited, then it can be implemented as a graph.
- We can modify the TREE-SEARCH algorithm to include a data structure – **closed list**, stores every expanded nodes.
- If the current node matches a node on the closed list, it is discarded instead of being expanded.
- The new algorithm is called **GRAPH-SEARCH**.
- On problems with many repeated steps, GRAPH-SEARCH is much more efficient than TREE-SEARCH.

Graph search

function GRAPH-SEARCH(*problem*, *fringe*) **returns** a solution, or failure

closed ← an empty set
fringe ← INSERT(MAKE-NODE(INITIAL-STATE[*problem*]), *fringe*)
loop do
  if *fringe* is empty **then return** failure
  node ← REMOVE-FRONT(*fringe*)
  if GOAL-TEST[*problem*](STATE[*node*]) **then return** SOLUTION(*node*)
  if STATE[*node*] is not in *closed* **then**
    add STATE[*node*] to *closed*
    fringe ← INSERTALL(EXPAND(*node*, *problem*), *fringe*)

- Use hash table for *closed* — constant-time lookup!
- Makes all algorithms complete in finite spaces!!
- Makes all algorithms worst-case exponential space!!!
- But size of graph often much less than $O(b^d)$!!!!

Summary

- Before an agent can start searching for solutions, it must formulate a **goal** and then use the goal to formulate a **problem**.

- A problem consists of four parts : the **initial state**, a set of **actions**, a **goal test** function and a **path cost** function.

- The environment of the problem is represented by a **state space**. A **path** through the state space from the initial state to a goal state is a **solution**.

Summary

- Problem formulation usually requires **abstracting** away real-world details to define a state space that can feasibly be explored.

- Variety of uninformed search strategies.

- Iterative deepening search uses only linear space and not much more time than other uninformed algorithms

- Graph search can be exponentially more efficient than tree. The GRAPH-SEARCH algorithm eliminates all duplicate states.

NEXT WEEK

- **Informed search algorithms**
  - Best-first search
    - Greedy best-first search
    - A^* search
  - Heuristics
  - Local search algorithms
    - Hill-climbing search
    - Simulated annealing search
    - Local beam search
    - Genetic algorithms