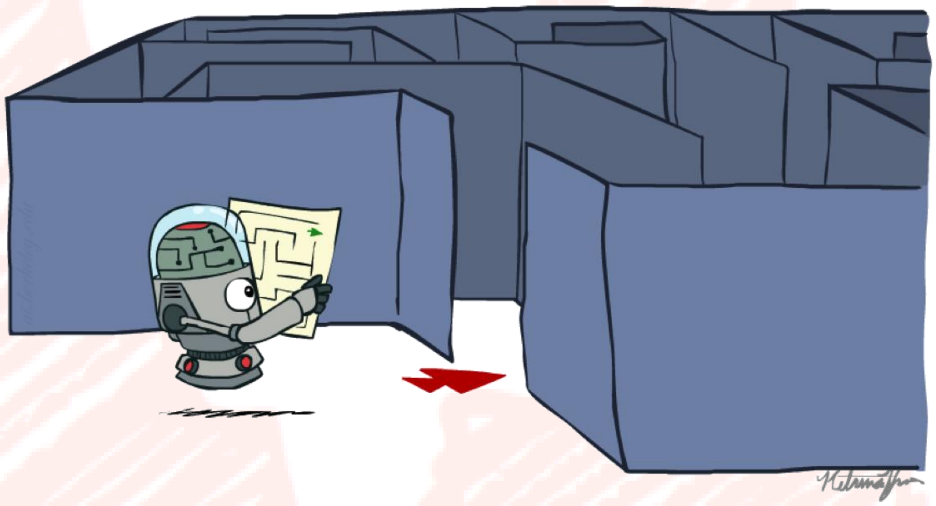
FACULTY OF INFORMATION TECHNOLOGY

Artificial Intelligence Fundamentals (NM TTNT)

Semester 1, 2023/2024

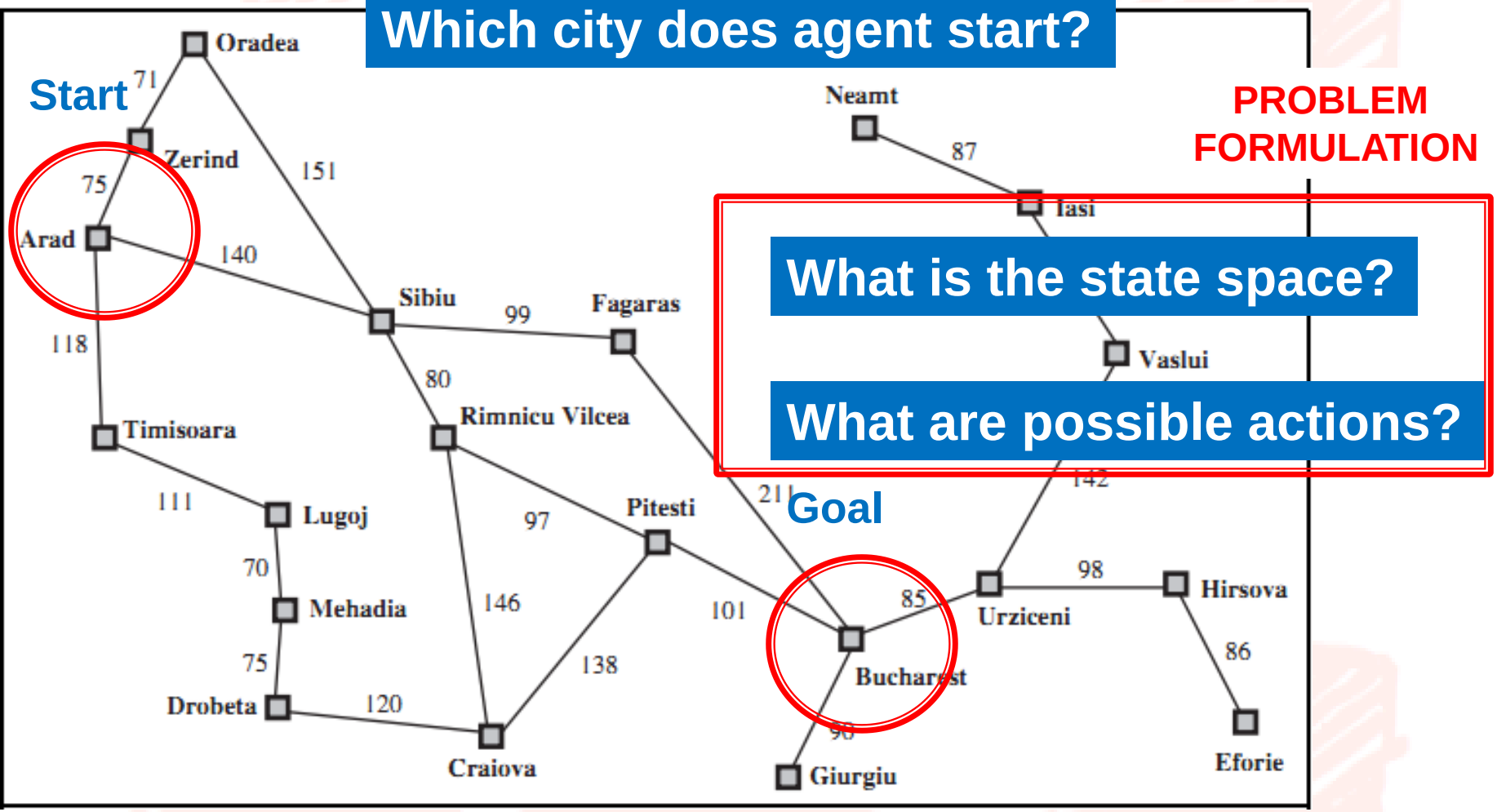
Chapter 3. Uninformed Search



Content

- Problem-solving (**goal-based**) agents
- Problem types
  - Single state (fully observable)
  - Search with partial information
- Problem formulation
  - Example problems
- Basic search algorithms (Uninformed search)
  - Breadth-first search
  - Depth-first search
  - Iterative deepening search
  - Uniform-cost search
  - Depth-limited search
  - Bidirectional search

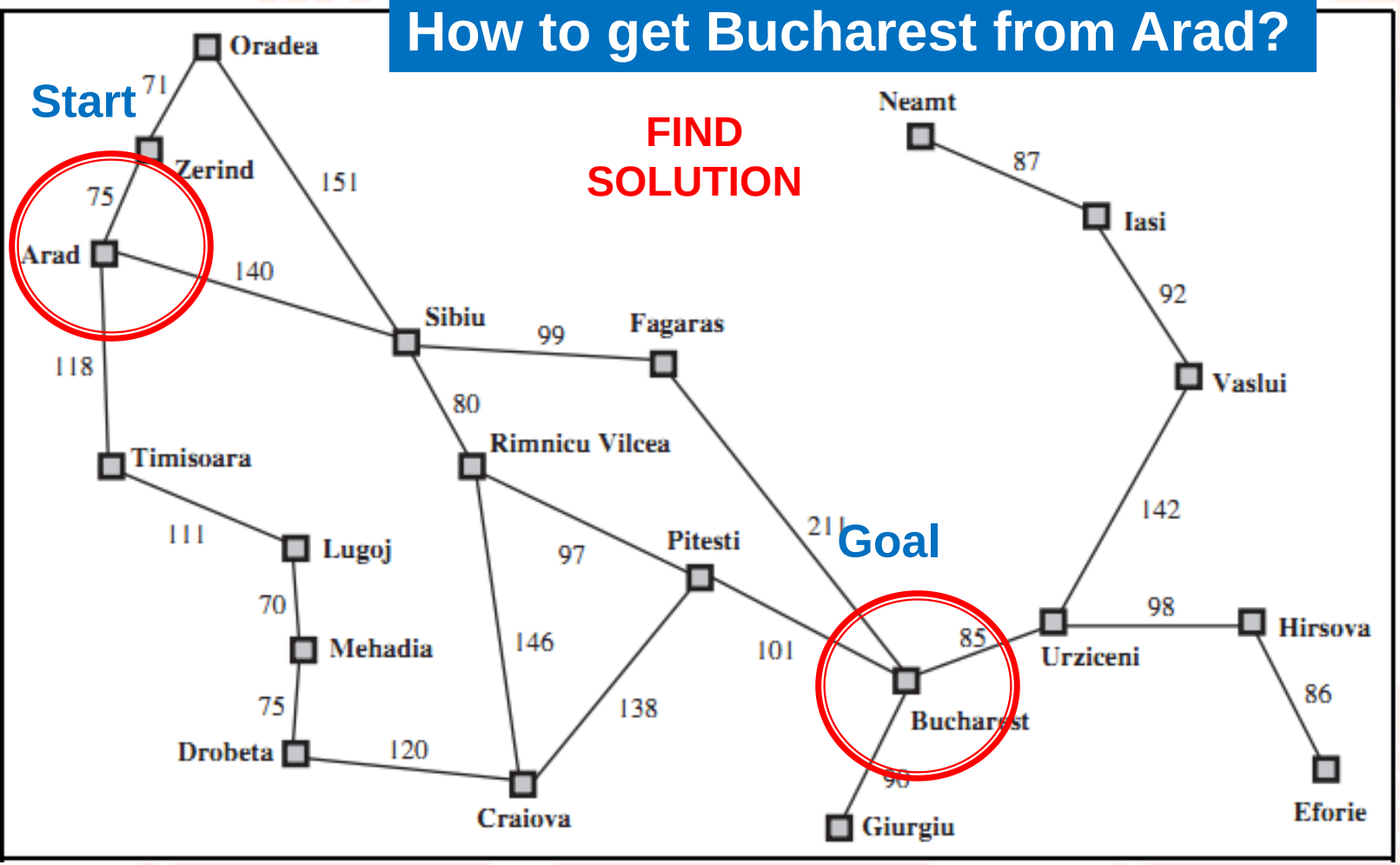
Romania Example



GOAL FORMULATION

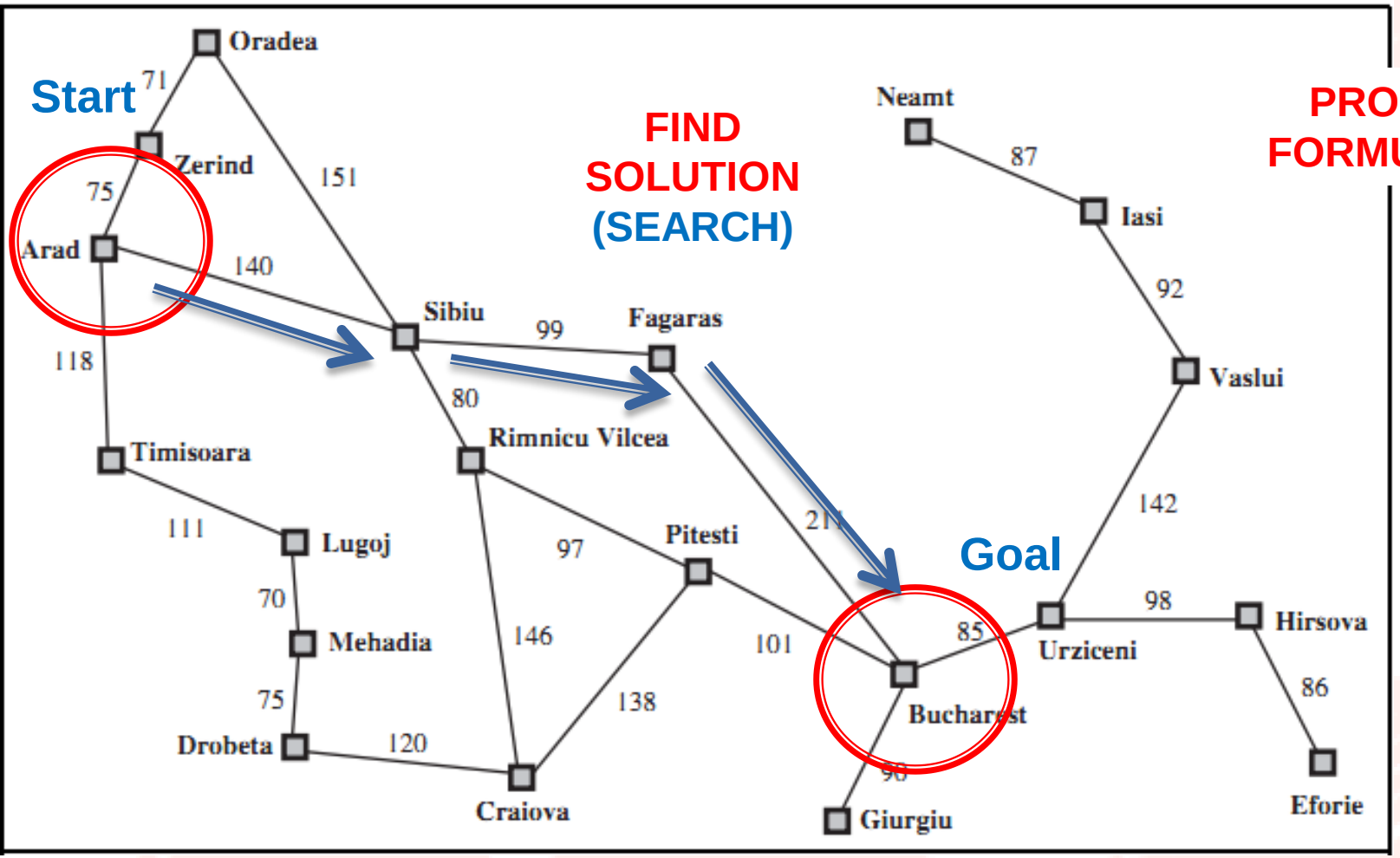
Which city does agent want to be in?

Romania Example (cont.)



GOAL FORMULATION

Romania Example (cont.)



Problem-solving agent

Goal-based agent

- States: the set of all states reachable from the initial state by any sequence of actions
- Initial state: where does the agent start in?
- Actions: possible actions available to the agent
- Transition model
  - specified by a function RESULT(s, a) that returns the state that results from TRANSITION MODEL doing action **a** in state s

RESULT(In(Arad),Go(Zerind)) = In(Zerind)

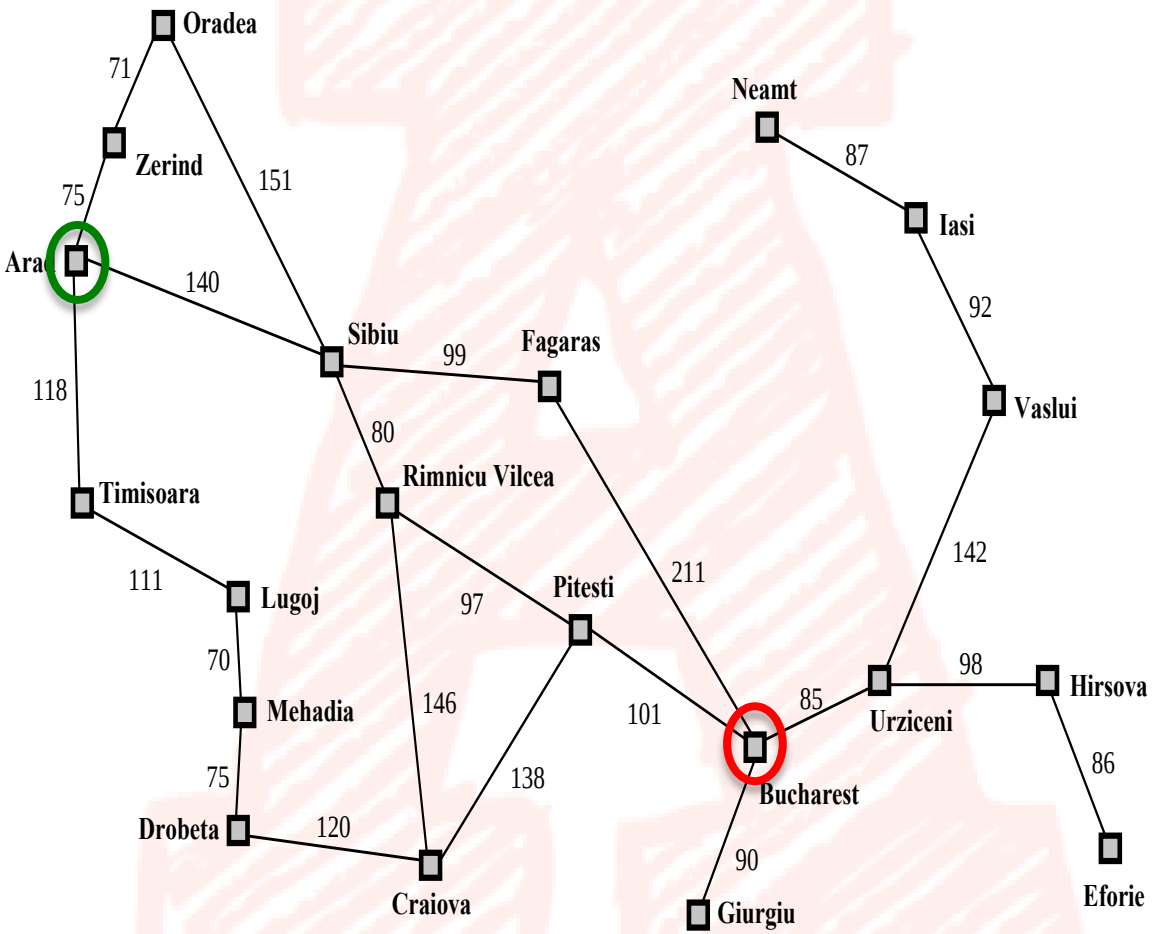
- Goal test
  - determines whether a given state is a goal state.
- Path cost
  - The cost of each path.

Problem-solving agent

- A simple problem-solving agent

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
      if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

Romania Example (cont.)



State space:

Cities

Actions:

Go to adjacent city

Transition model:

Result(In(A), Go(B)) = In(B)

Step cost:

Distance along road link

Start state:

Arad

Goal test:

Is state == Bucharest?

Solution?

Problem types

- Deterministic, fully observable ⇒ *single state problem*

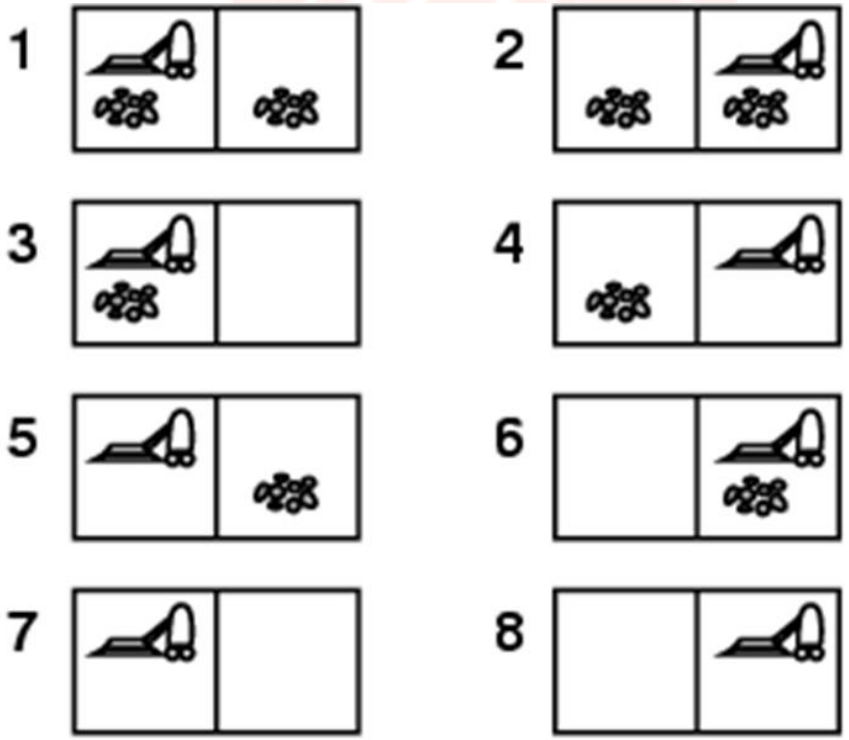
- Partial knowledge of states and actions:
  - Non-observable ⇒ ***sensorless or conformant problem***
    - Agent may have no idea where it is; solution (if any) is a sequence.
  - **Nondeterministic** and/or partially observable ⇒ ***contingency** (ngẫu nhiên)**problem***
    - Percepts provide ***new*** information about current state; solution is a tree or policy; often **interleave** search and execution
  - **Unknown state space** ⇒ ***exploration problem*** ("online")
    - When states and actions of the environment are unknown.

Example: vacuum world single-state

- Single-state, start in #5.

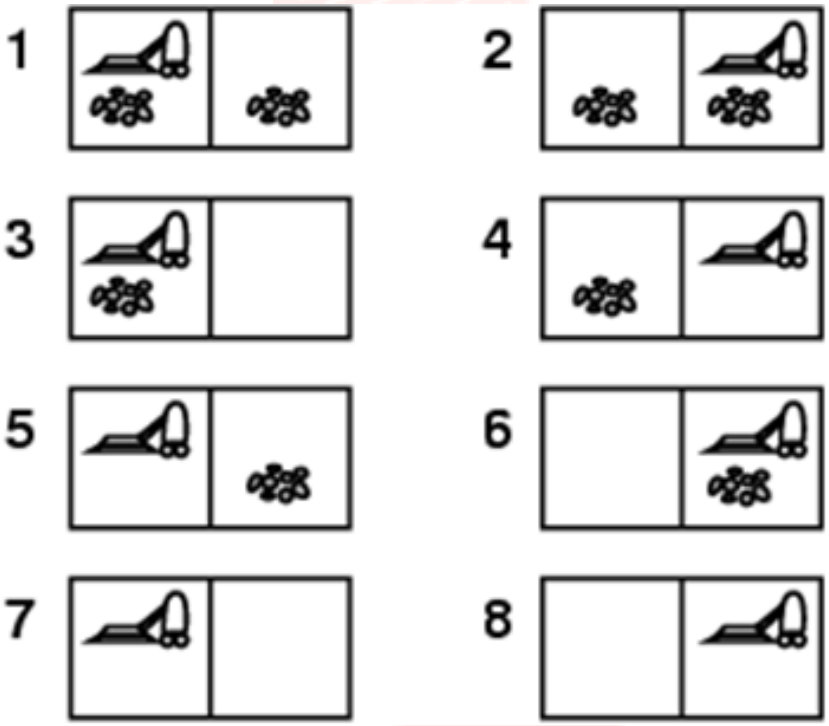
Solution?

[Right, Suck]



Conformant (Sensorless) problems

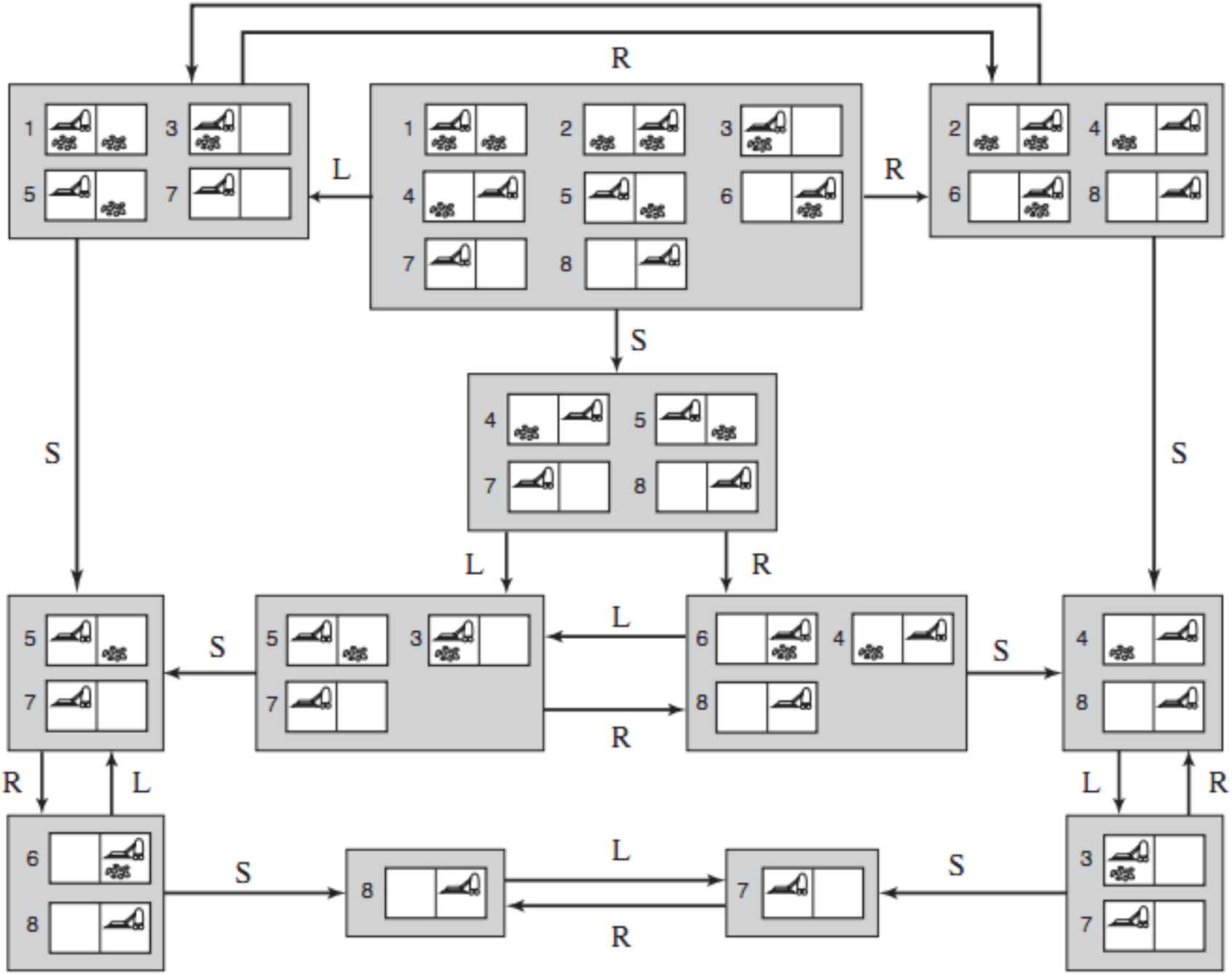
- The agent knows the geography of its world, but doesn't know its location or the distribution of dirt
- Start in {1, 2, 3, 4, 5, 6, 7, 8} e.g Right goes to {2, 4, 6, 8}.
- Solution??
  - [Right, Suck, Left, Suck]



- When the world is not fully observable: reason about a set of states that might be reached

  =belief state

Belief state of vacuum-world (set of states that the agent might be in)

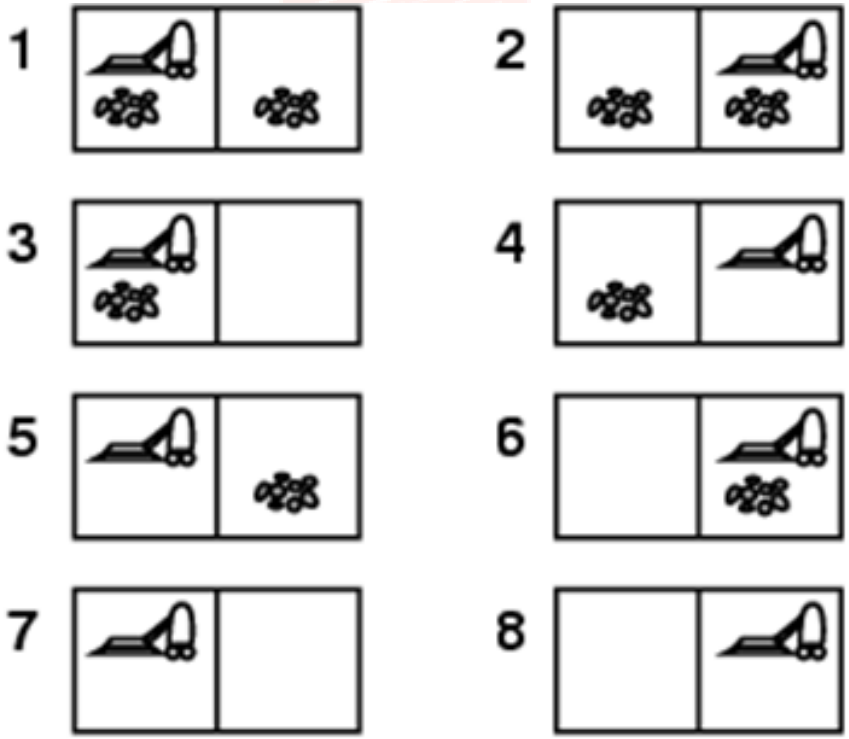


Contingency problems

- **Nondeterministic**:
  - assume Murphy's law, suck can dirty a clean carpet.

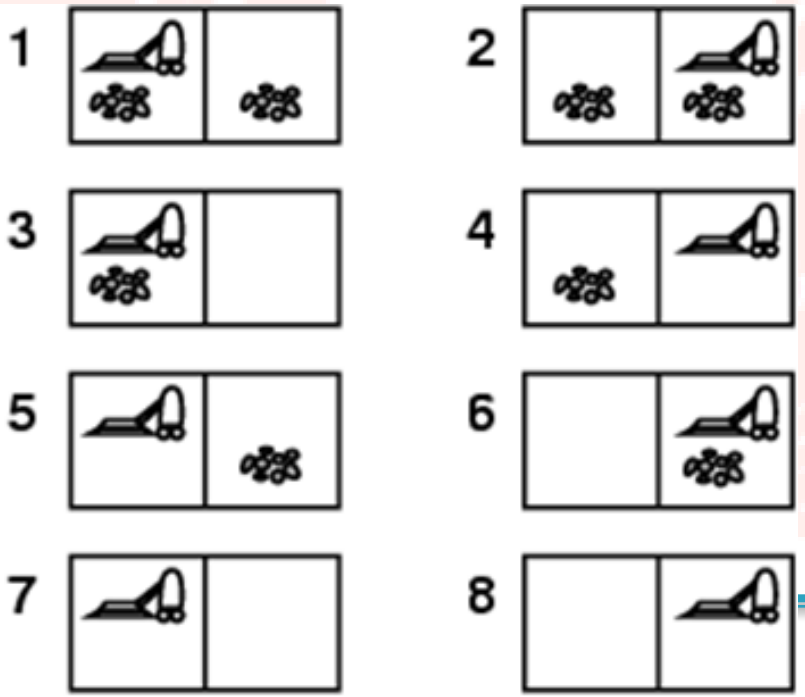
- **Partially observable** (local sensing): dirt, location only.
  - Percept = [L, Dirty] = {1, 3}
  - [Suck] = {5, 7}
  - [Right] ={6, 8}
  - [Suck] in {6}={8} (Success)
  - BUT [Suck] in {8} = failure

- **Solution**??
  - Belief-state: no fixed action sequence guarantees solution



Contingency problems (cont.)

- **Relax requirement**: a problem with fewer restrictions on the actions
  - A solution starting in a state where you're in the left room and it's dirty is:
    - [Suck, Right, if [R, dirty] then Suck]
    - Select actions based on contingencies arising during execution.



Single-state problem formulation

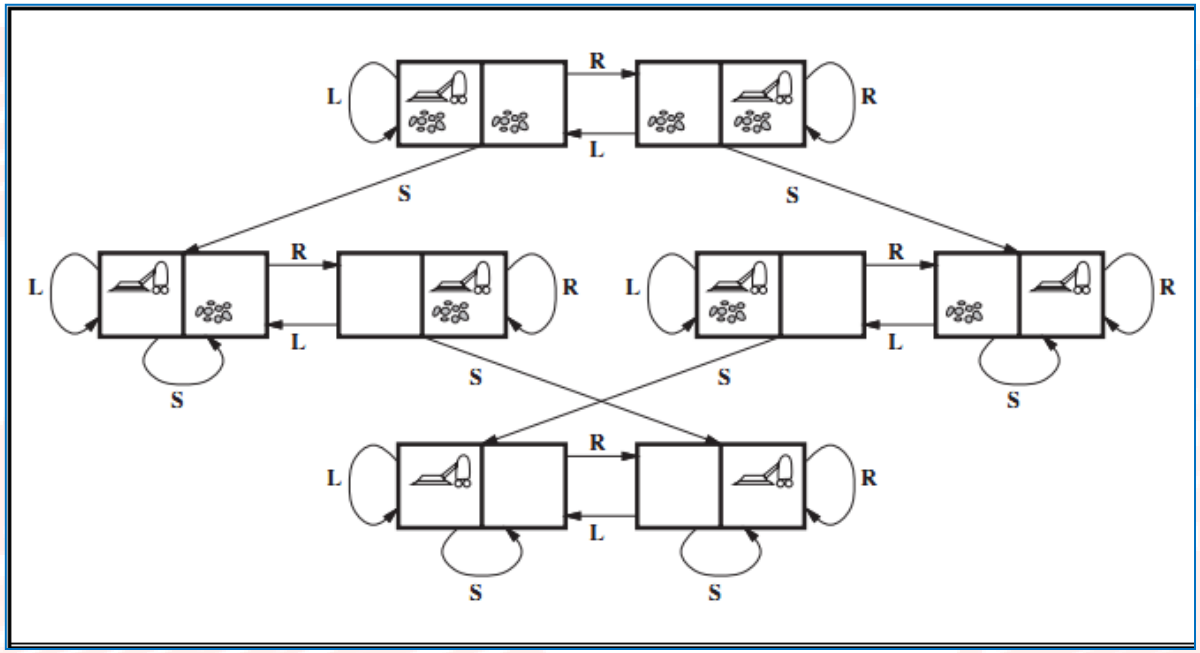
A problem is defined by:

- An initial state, e.g. Arad
- Actions or Successor function S(X) = set of action-state pairs
  - e.g. S(Arad) = {<Arad → Zerind, Zerind>,…}
  - intial state + successor function = state space
- Goal test, can be
  - Explicit, e.g. x = 'at Bucharest'
  - Implicit, e.g. checkmate(x)
- Path cost (additive)
  - e.g. sum of distances, number of actions executed, …
  - c(x,a,y) is the step cost, assumed to be >= 0
- A solution: a sequence of actions from initial to goal state.
  - Optimal solution has the lowest path cost.

Selecting a state space

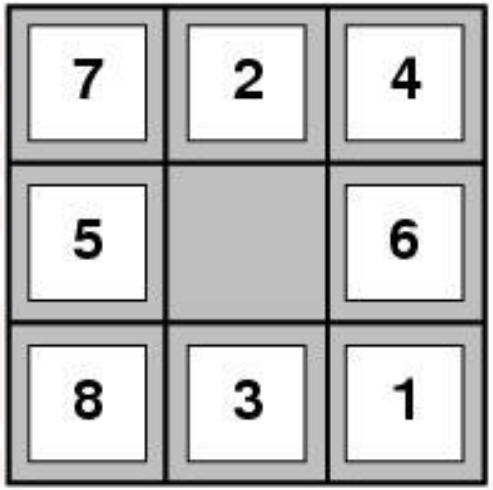
- Real world is absurdly complex.
  - State space must be abstracted for problem solving.
- (Abstract) state = set of real states.
- (Abstract) action = complex combination of real actions.
  - e.g. Arad → Zerind represents a complex set of possible routes, detours, rest stops, etc.
  - The abstraction is valid if the path between two states is reflected in the real world.
- (Abstract) solution = set of real paths that are solutions in the real world.
- Each abstract action should be "easier" than the real problem.

Example: Vacuum world

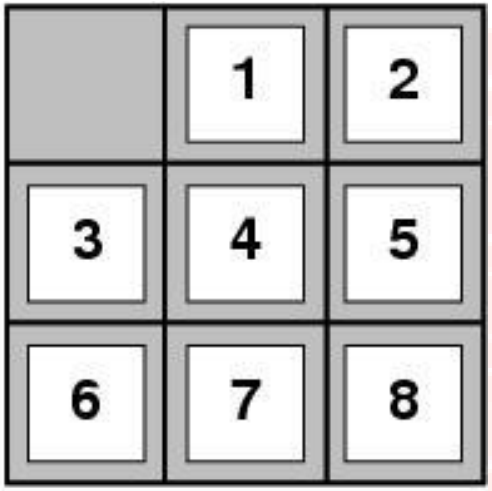


- States: two locations with or without dirt: $2 \times 2^2 = 8$ states.
- Initial state: Any state can be initial
- Actions: {Left, Right, Suck}
- Goal test: Check whether squares are clean.
- Path cost: Number of actions to reach goal.

Example: 8-puzzle



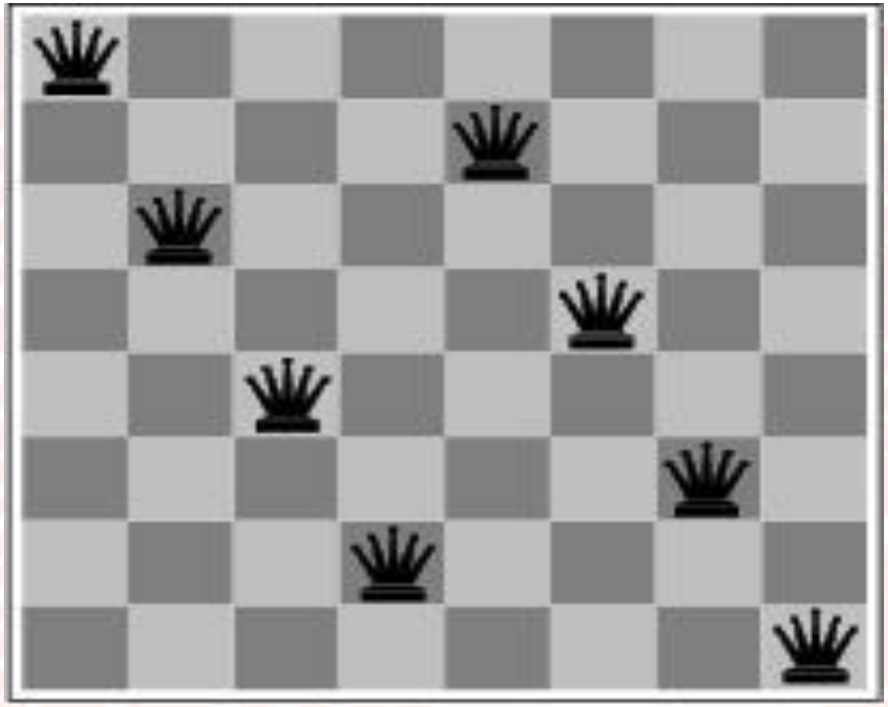
Start State



Goal State

- States: Integer location of each tile (9! States)
- Initial state: Any state can be initial
- Actions: {Left, Right, Up, Down}
- Goal test: Check whether goal configuration is reached
- Path cost: Number of actions to reach goal

Example: 8-queens problem



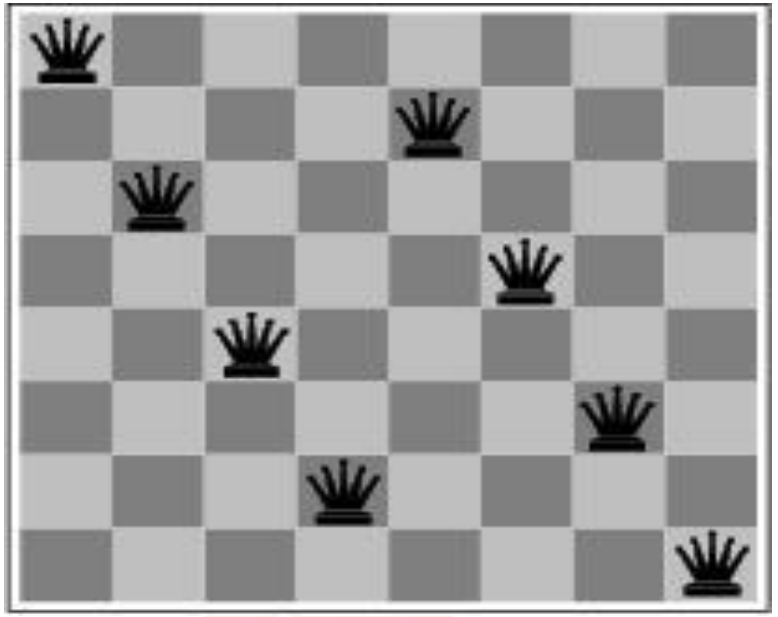
- States:
- Initial state:
- Actions:
- Goal test:
- Path cost:

Incremental formulation vs. complete-state formulation

Example: 8-queens problem

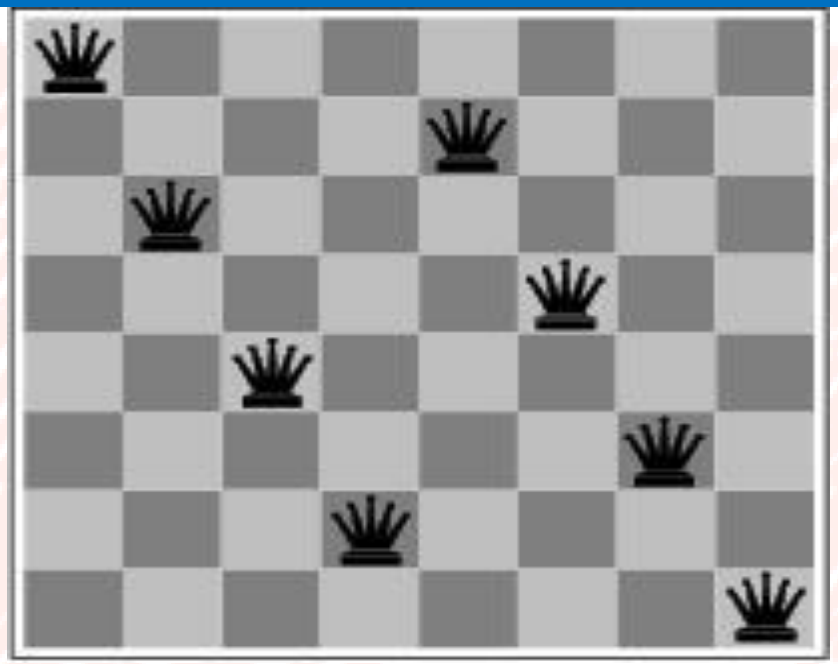
- States: Any arrangement of 0 to 8 queens on the board
- Initial state: No queens
- Actions: Add queen in empty square
- Transition model: Returns the board with a queen added to the specified square
- Goal test: 8 queens on board and none attacked
- Path cost: None

$\approx 3 \times 10^{14}$ possible sequences to investigate



Incremental formulation

Example: 8-queens problem



Complete-state formulation

- States: n ($0 \leq n \leq 8$) queens on the board, one per column in the n leftmost columns with no queen attacking another.
- Actions: Add queen in leftmost empty column such that is not attacking other queens

2057 possible sequences to investigate;

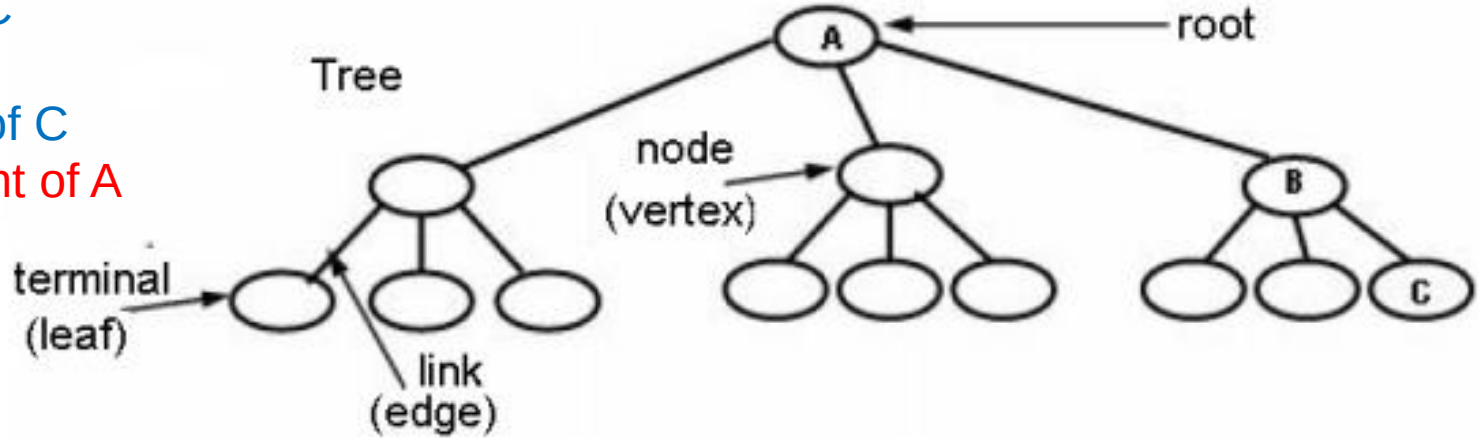
Basic search algorithms

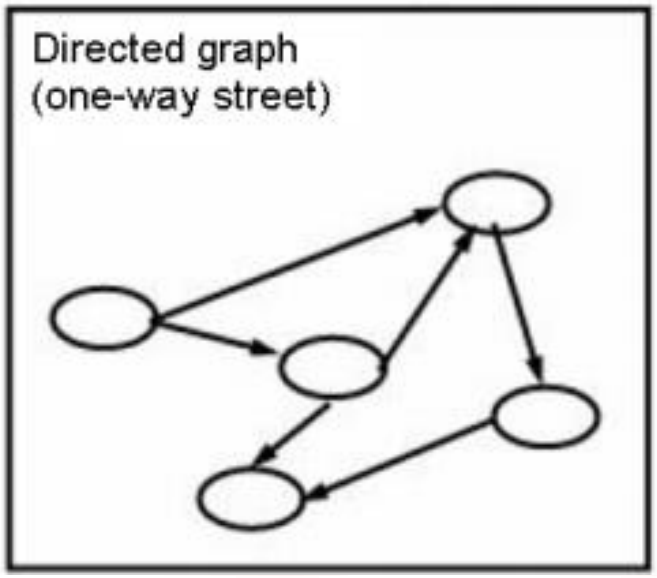
- How do we find the solutions of previous problems?

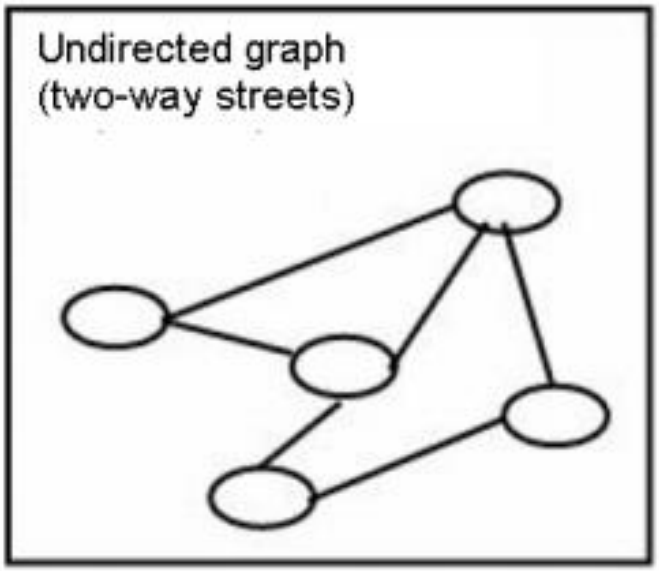
- Search the state space (remember complexity of space depends on state representation)
- Here: search through *explicit tree generation*
  - **ROOT** = initial state.
  - **Nodes** and **leafs** generated through successor function.
- In general search generates a graph (same state through multiple paths)

Tree and Graph

B is parent of C
C is child of B
A is ancestor of C
C is descendant of A

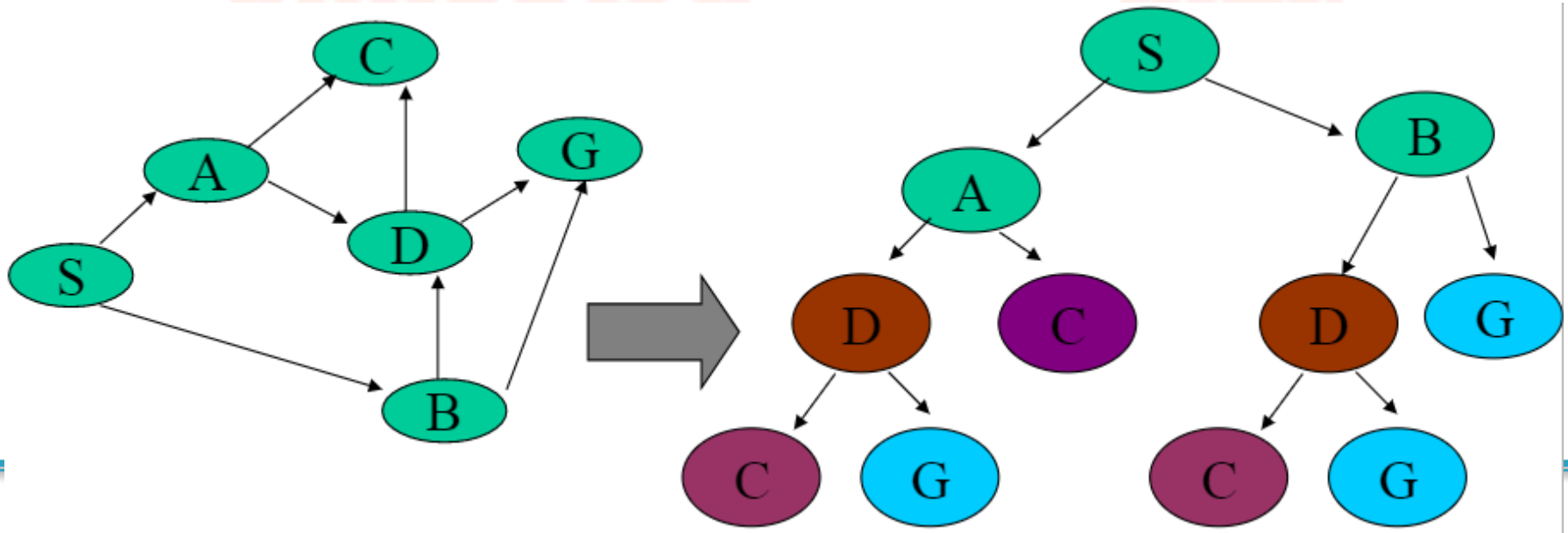




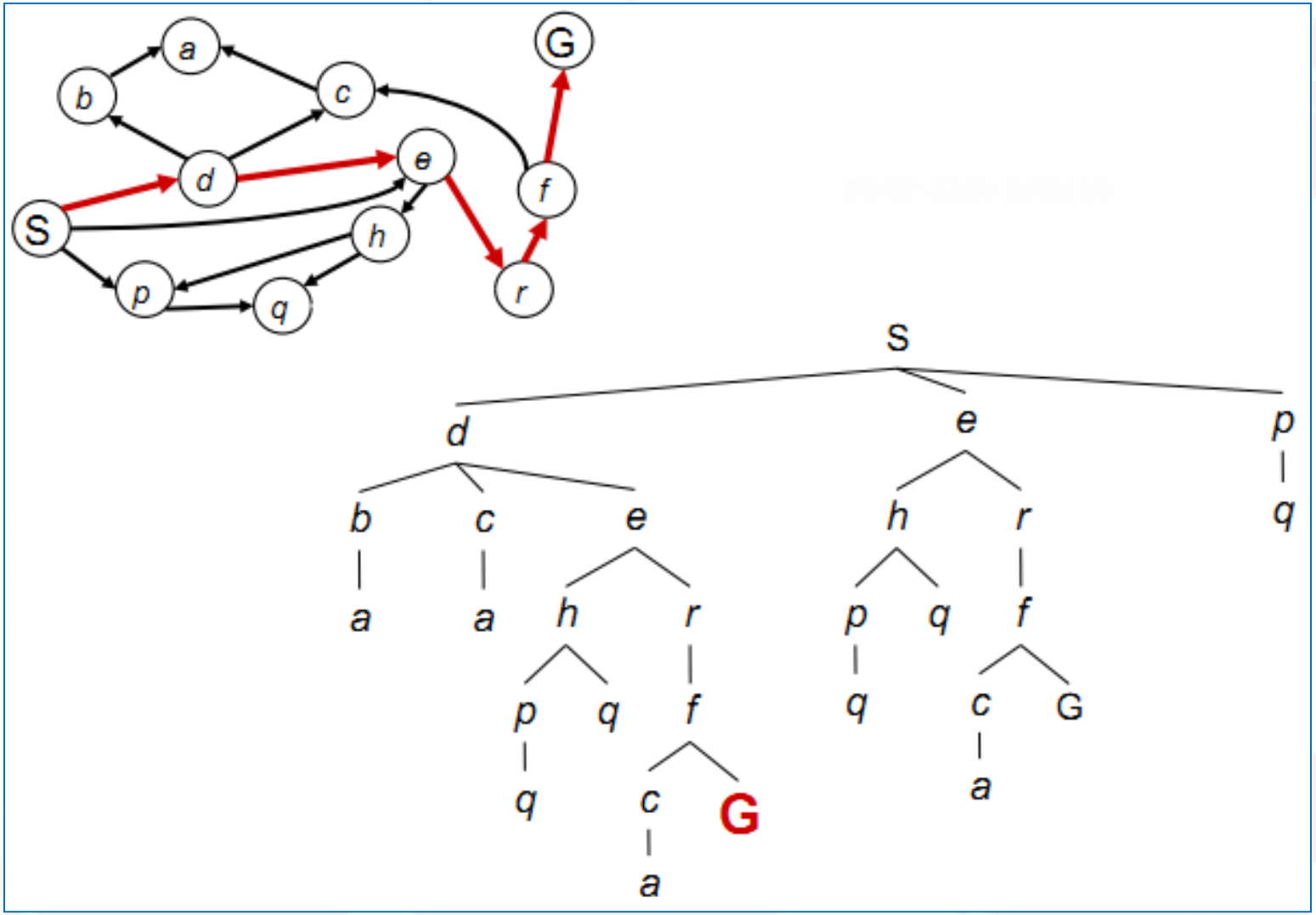


Graph search to Tree search

- Graph Search as Tree Search, Trees are directed graphs without cycles and nodes having <=1 parent
- We can turn graph search problems (from S to G) into tree search problems by:
  - Replacing undirected links by 2 directed links
  - Avoiding loops in path (or keeping track of visited nodes globally)



Graph search to Tree search (cont.)



Tree search vs Graph search

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

Frontier: contains generated nodes which are not yet expanded.

Simple tree search example

function TREE-SEARCH(*problem, strategy*) return a solution or failure

Initialize search tree to the *initial state* of the *problem*

loop do

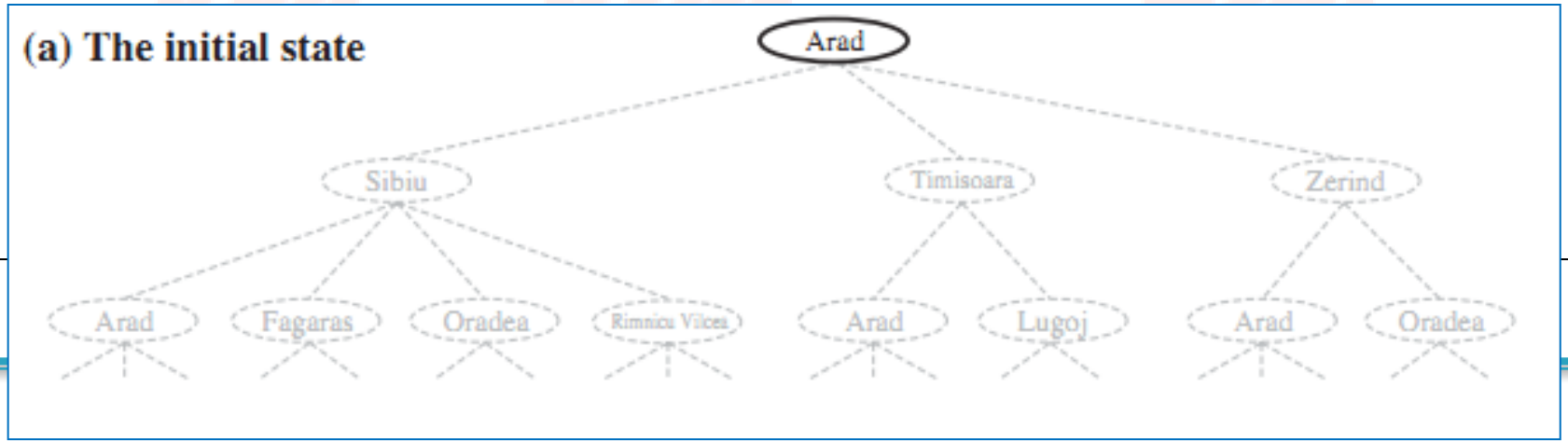
if no candidates for expansion then return ***failure***

choose leaf node for expansion according to ***strategy***

if node contains goal state then return the corresponding ***solution***

else expand chosen node and add resulting nodes to the search tree

end do



Simple tree search example

function TREE-SEARCH(*problem, strategy*) return a solution or failure

Initialize search tree to the *initial state* of the *problem*

loop do

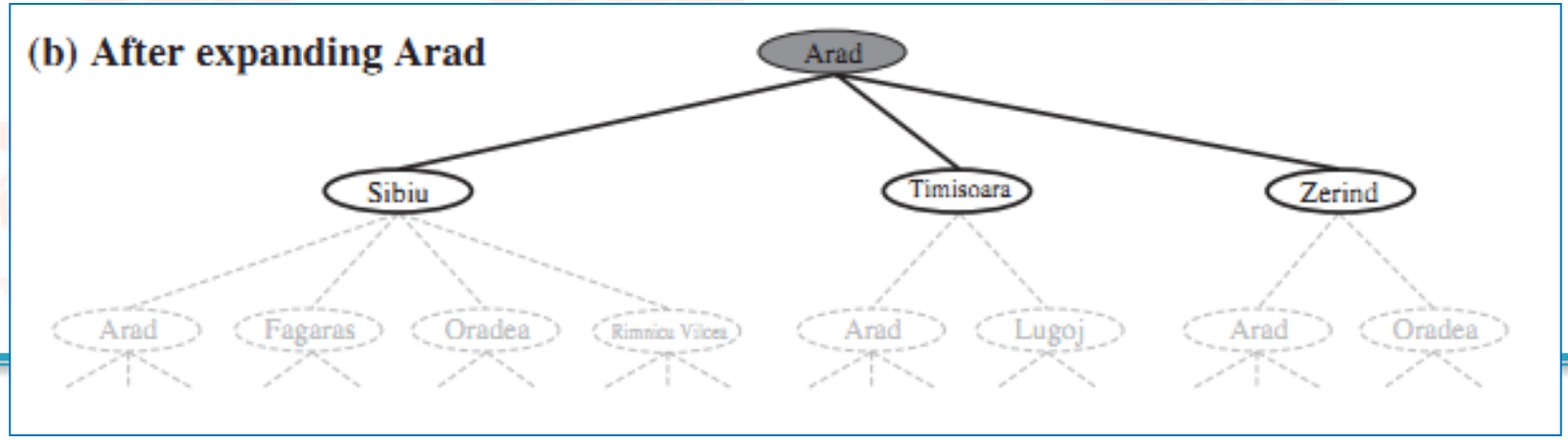
if no candidates for expansion then return ***failure***

choose leaf node for expansion according to *strategy*

if node contains goal state then return the corresponding ***solution***

else expand chosen node and add resulting nodes to the search tree

end do



Simple tree search example

function TREE-SEARCH(*problem, strategy*) return a solution or failure

Initialize search tree to the *initial state* of the *problem*

loop do

if no candidates for expansion then return ***failure***

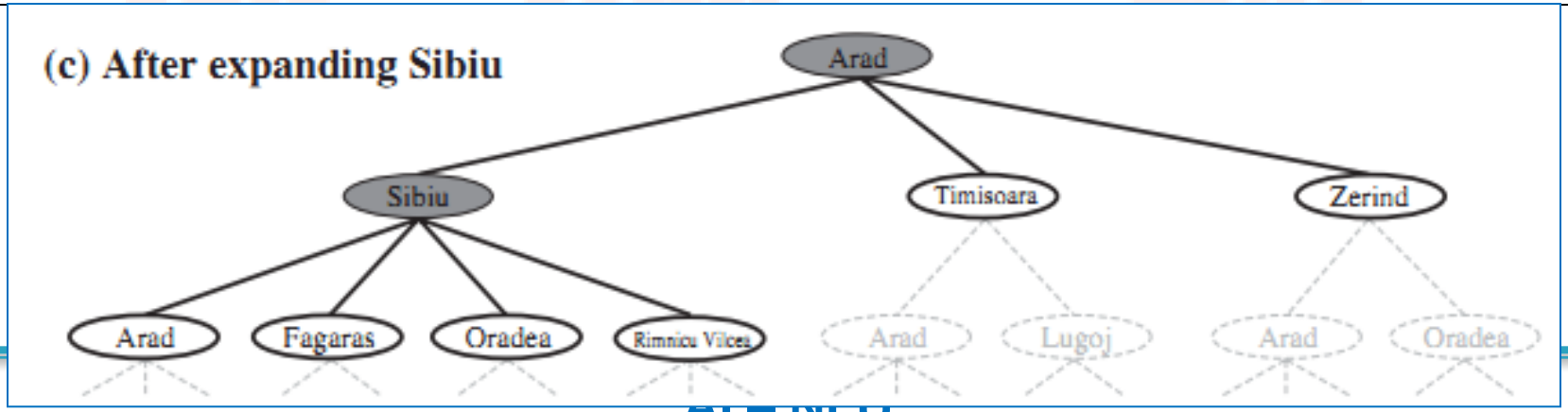
choose leaf node for expansion according to ***strategy***

if node contains goal state then return the corresponding ***solution***

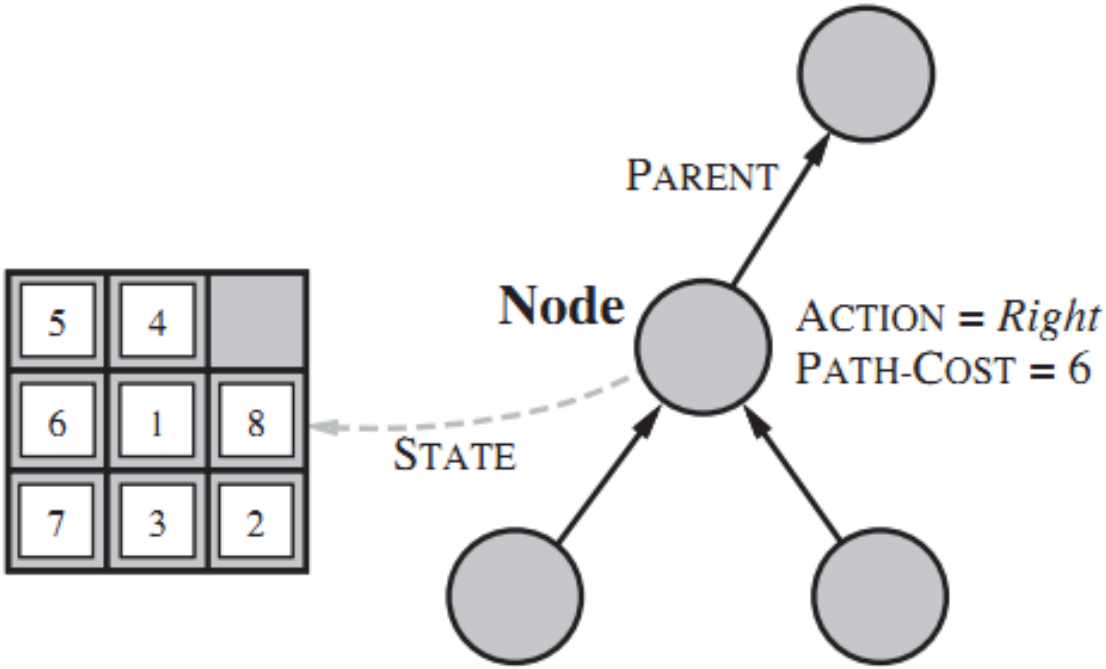
else **expand chosen node and add resulting nodes to the search tree**

end do

Determines search process!!



Infrastructure for search algorithms



- A *state* is a (representation of) a physical configuration
- A *node* is a data structure belong to a search tree include state, parent-node, action, path-cost, …

Tree search algorithm

```
function TREE-SEARCH(problem) return a solution or failure
    frontier ← INSERT(MAKE-NODE(problem.INITIAL-STATE),
    frontier)
    loop do
        if EMPTY?(frontier) then return failure
        node ← REMOVE-FIRST(frontier)
        if problem.GOAL-TEST(node.STATE)
            then return SOLUTION(node)
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            frontier ← INSERT(child, frontier)
```

- ***frontier*** = contains generated nodes which are not yet expanded.
  - ***White nodes*** with black outline

Tree search algorithm (cont.)

- The function CHILD-NODE takes a parent node and an action and returns the resulting child node

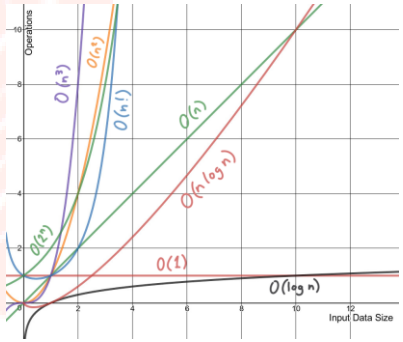
```
function CHILD-NODE(problem, parent, action) returns a node
  return a node with
      STATE = problem.RESULT(parent.STATE, action),
      PARENT = parent, ACTION = action,
      PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

Search strategies

- A strategy is defined by picking the order of node expansion.
- Problem-solving performance is measured 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?

Search strategies (cont.)

- Time and space complexity are measured in terms of problem difficulty defined by:
  - b – maximum branching factor of the search tree (maximum number of successors of any node)
  - d – depth of the least-cost solution (the number of steps along the path from the root)
  - m – the maximum length of any path in the state space (may be ∞)



Uninformed search strategies

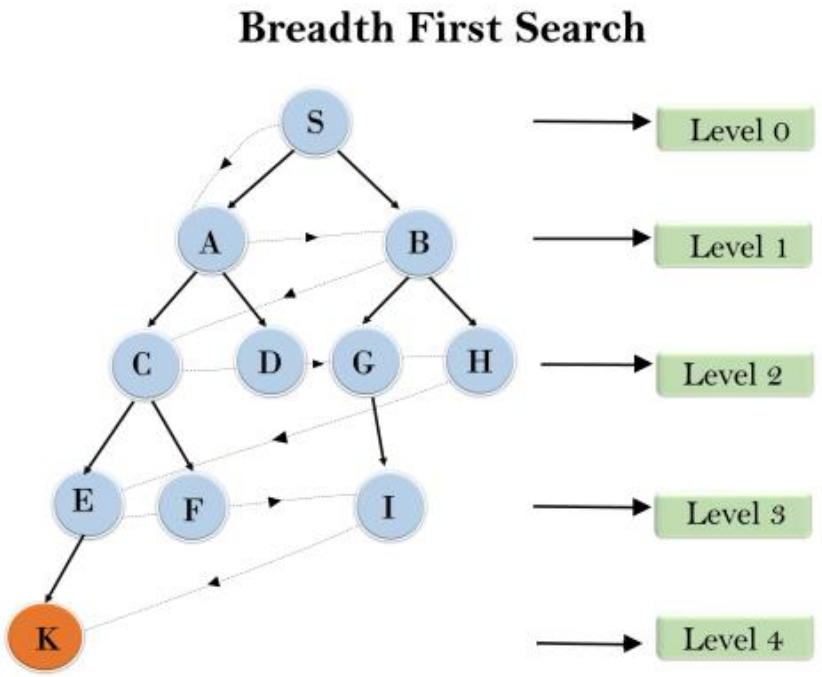
- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search.
- Bidirectional search

Uninformed search strategies

- Other name: blind search
- Only use information available in problem definition
  - When strategies can determine whether one non-goal state is better than another → informed search
- Categories defined by expansion algorithm:

- **Breadth-first search**
- **Uniform-cost search**
- **Depth-first search**
- **Depth-limited search**
- **Iterative deepening search.**
- **Bidirectional search**

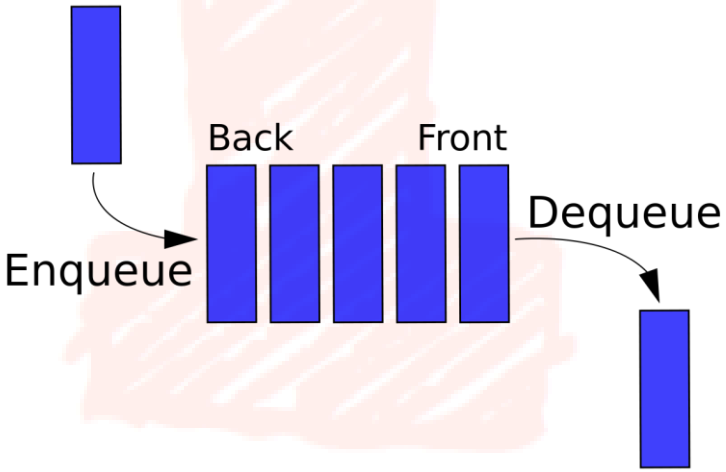
Breadth-First search



Breadth-first search

- Expand shallowest unexpanded node

- Implementation:
  - **frontier** is a FIFO **queue**,
    i.e., new successors go at end

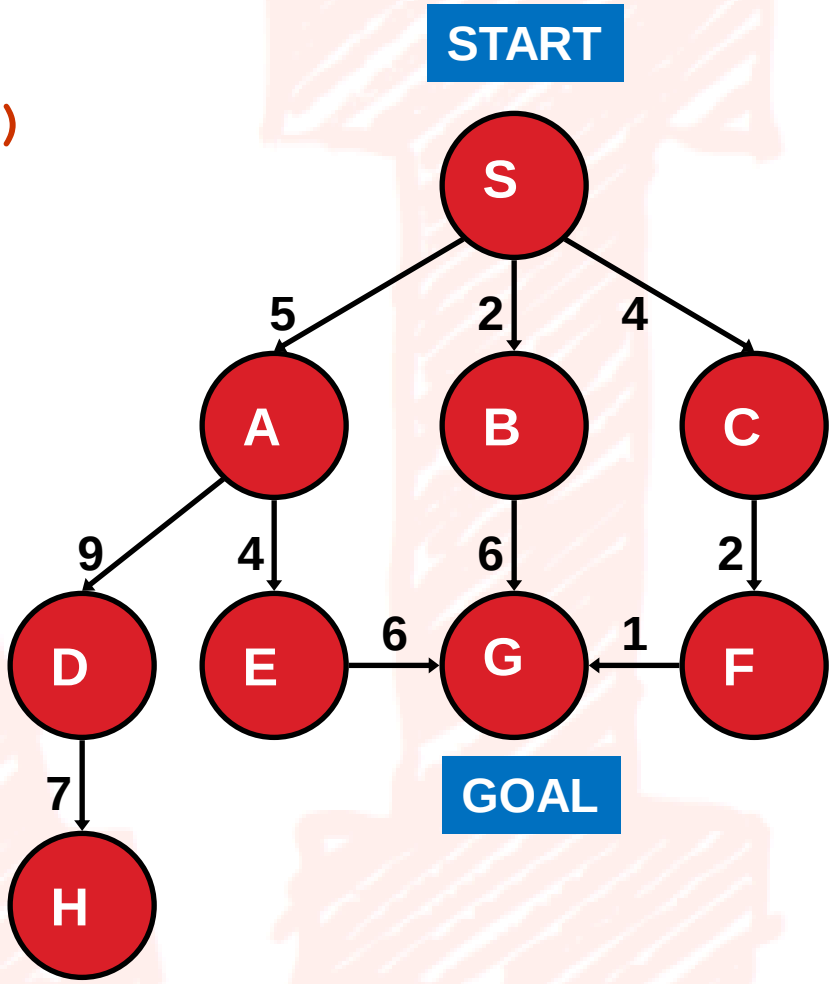


Breadth-First Search (BFS)

TREE-SEARCH(problem, queue)

# of nodes tested: 0, expanded: 0

current	queue
	{S}

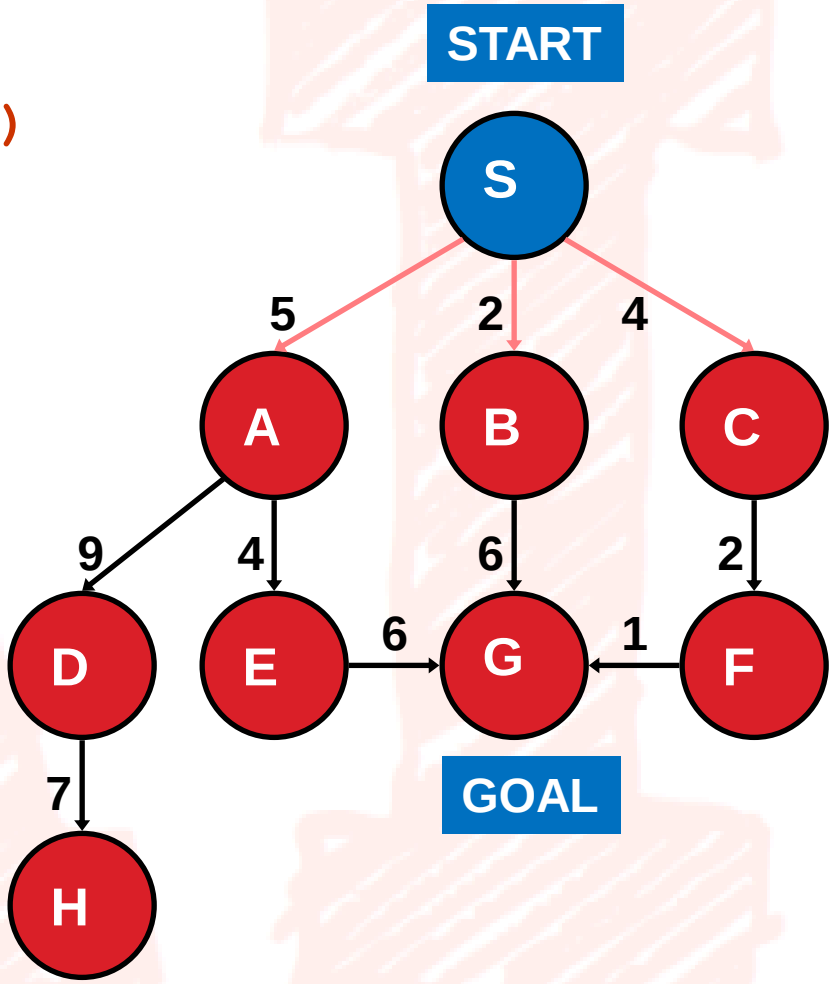


Breadth-First Search (BFS)

TREE-SEARCH(problem, queue)

# of nodes tested: 1, expanded: 1

current	queue
	{S}
S not goal	{A,B,C}

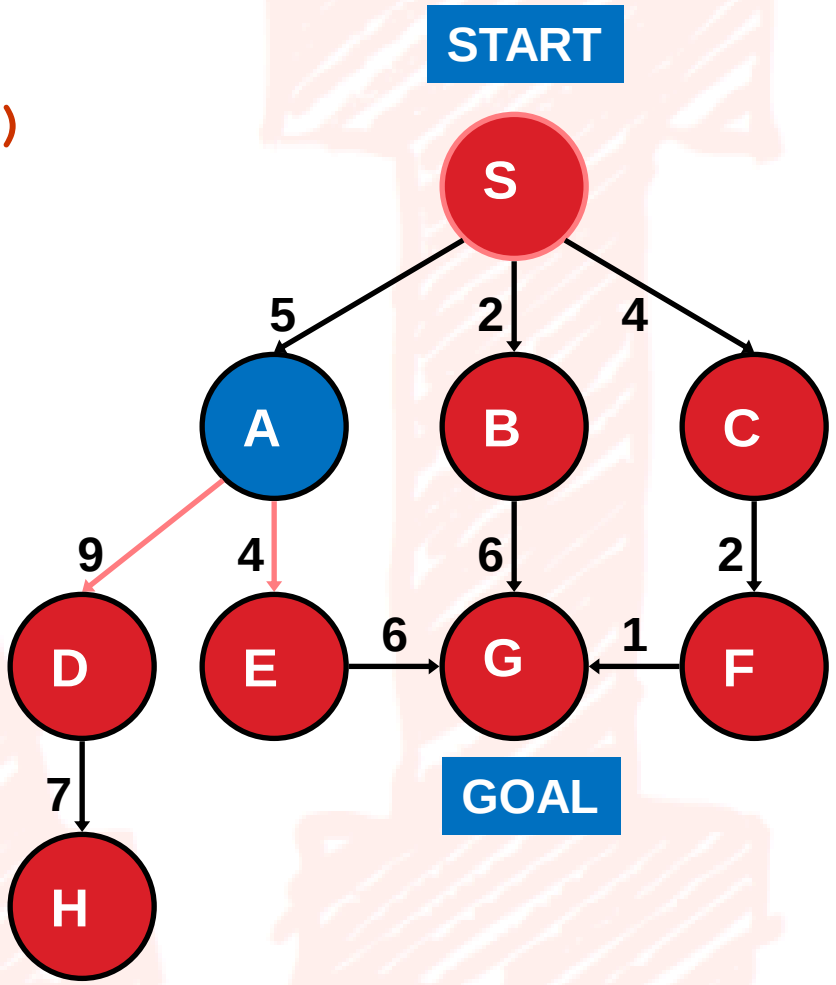


Breadth-First Search (BFS)

TREE-SEARCH(problem, queue)

# of nodes tested: 2, expanded: 2

current	queue
	{S}
S	{A,B,C}
A not goal	{B,C,D,E}

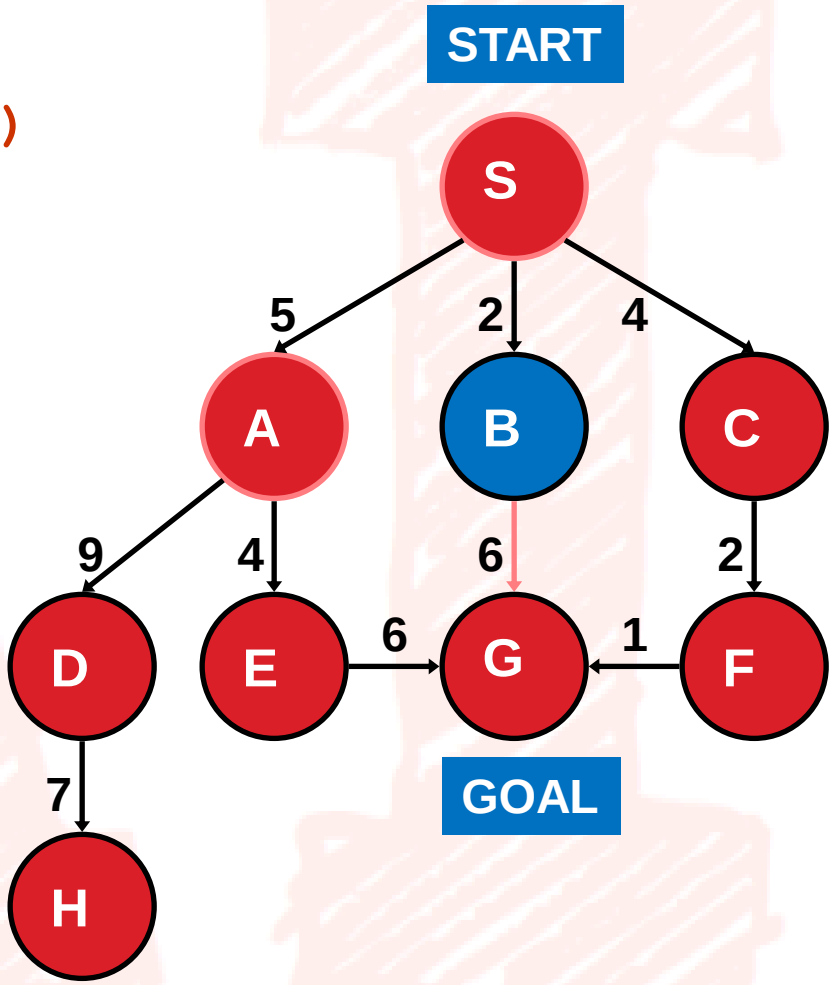


Breadth-First Search (BFS)

TREE-SEARCH(problem, queue)

# of nodes tested: 3, expanded: 3

current	queue
	{S}
S	{A,B,C}
A	{B,C,D,E}
B not goal	{C,D,E,G}

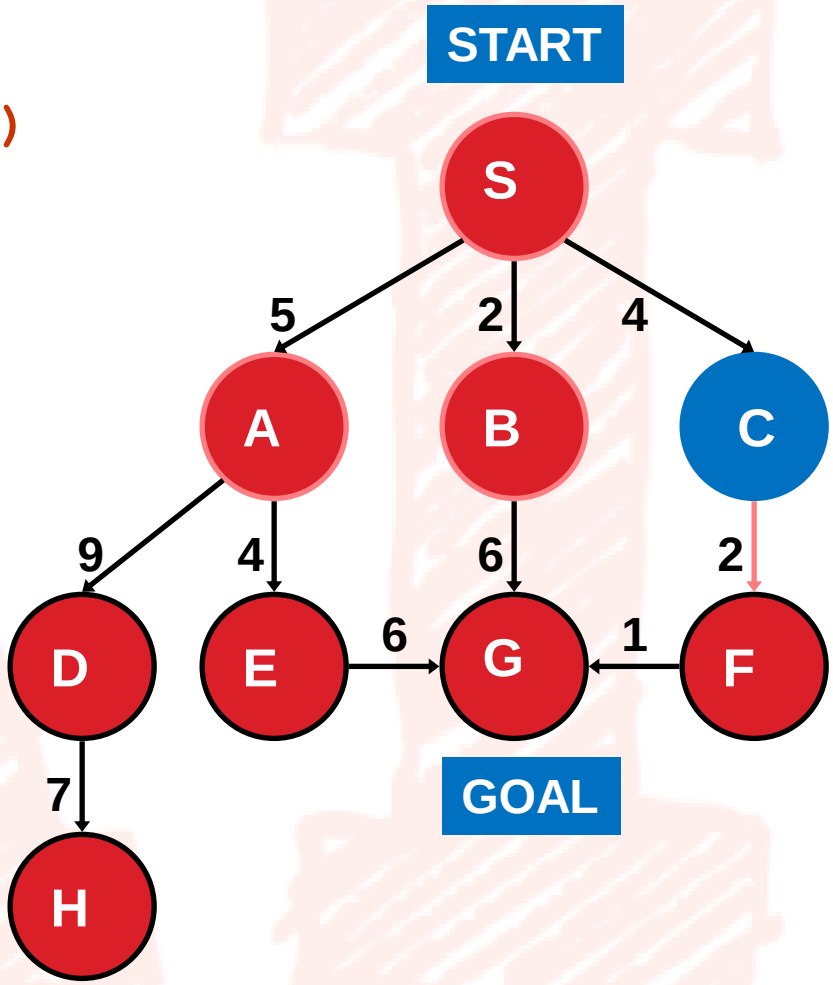


Breadth-First Search (BFS)

TREE-SEARCH(problem, queue)

# of nodes tested: 4, expanded: 4

current	queue
	{S}
S	{A,B,C}
A	{B,C,D,E}
B	{C,D,E,G}
C not goal	{D,E,G,F}

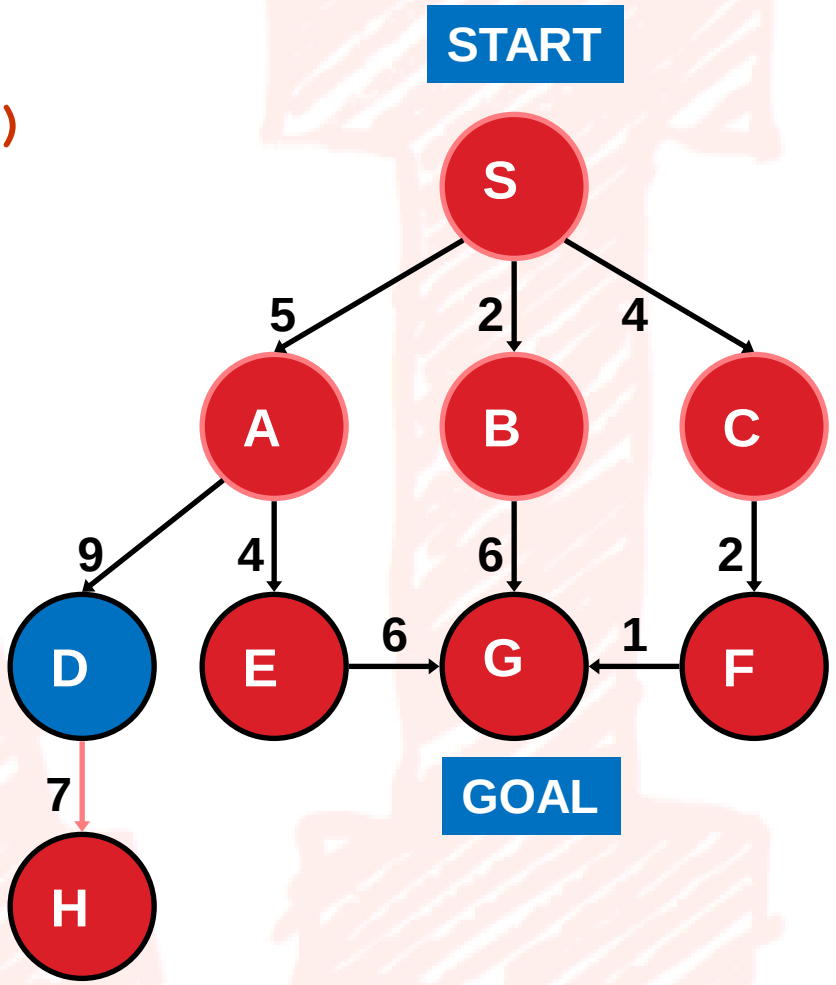


Breadth-First Search (BFS)

TREE-SEARCH(problem, queue)

# of nodes tested: 5, expanded: 5

current	queue
	{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}

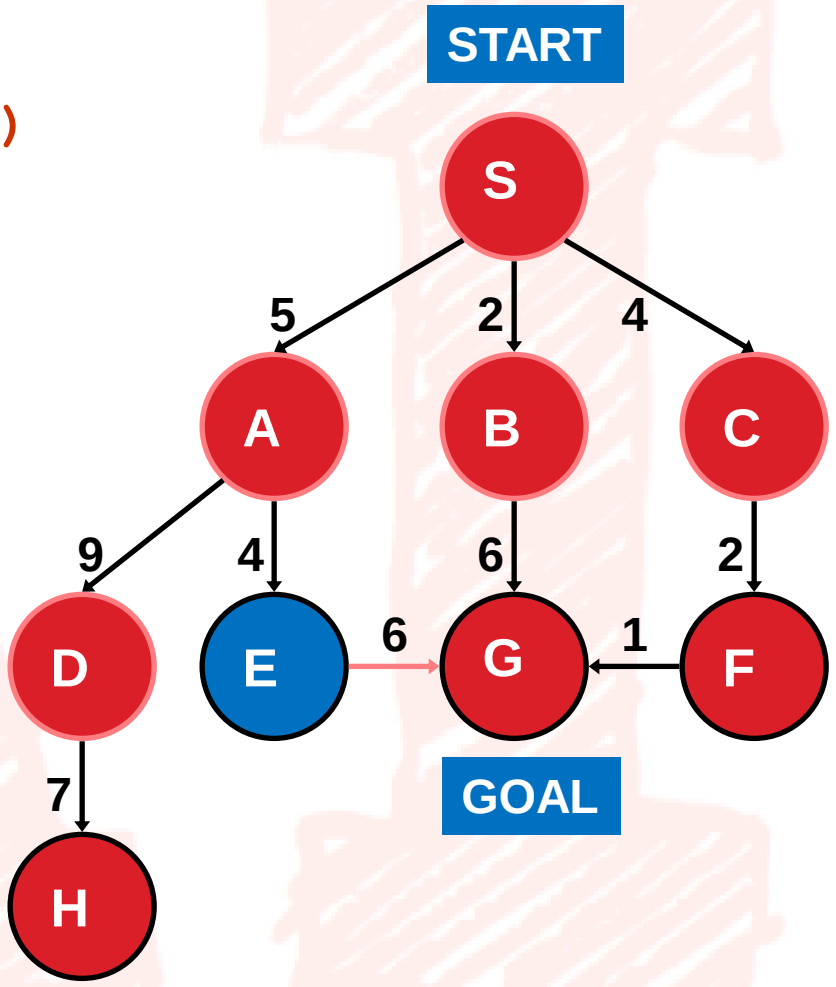


Breadth-First Search (BFS)

TREE-SEARCH(problem, queue)

# of nodes tested: 6, expanded: 6

current	queue
	{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}

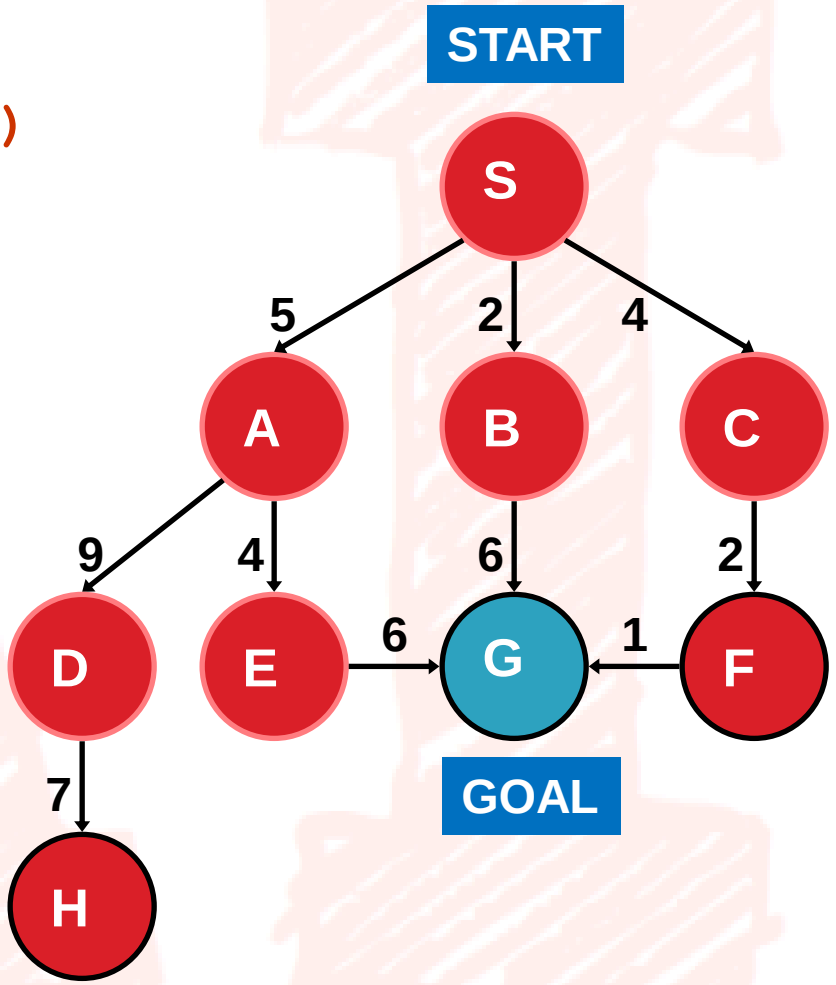


Breadth-First Search (BFS)

TREE-SEARCH(problem, queue)

# of nodes tested: 7, expanded: 6

current	queue
	{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

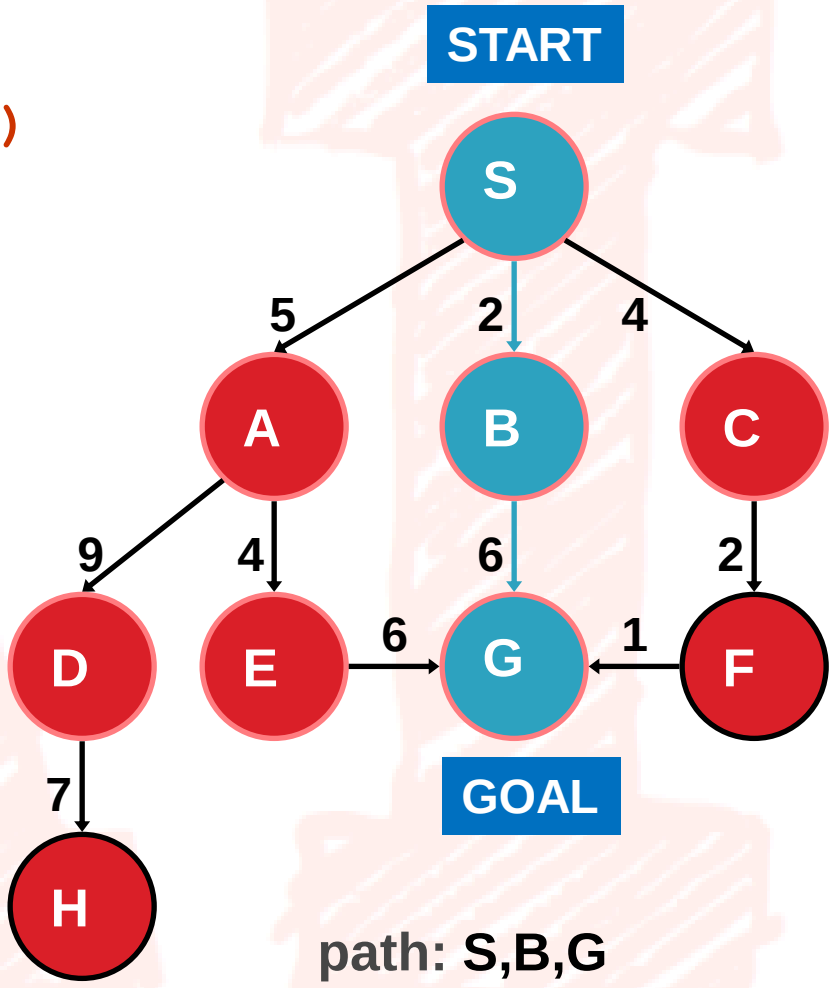


Breadth-First Search (BFS)

TREE-SEARCH(problem, queue)

# of nodes tested: 7, expanded: 6

current	queue
	{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}

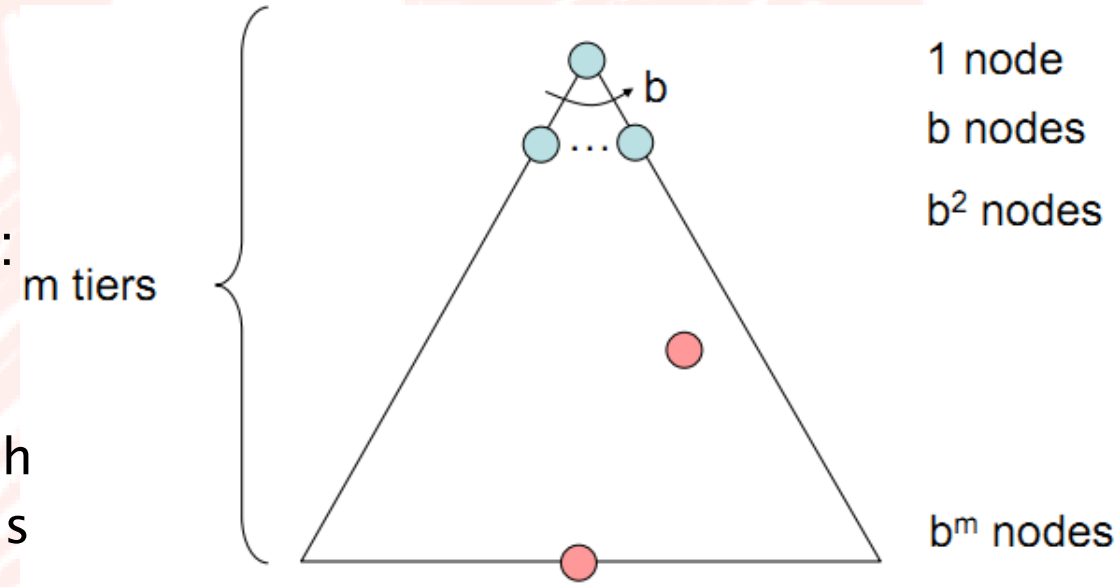


path: **S,B,G**
cost: **8**

Search Algorithm Properties

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

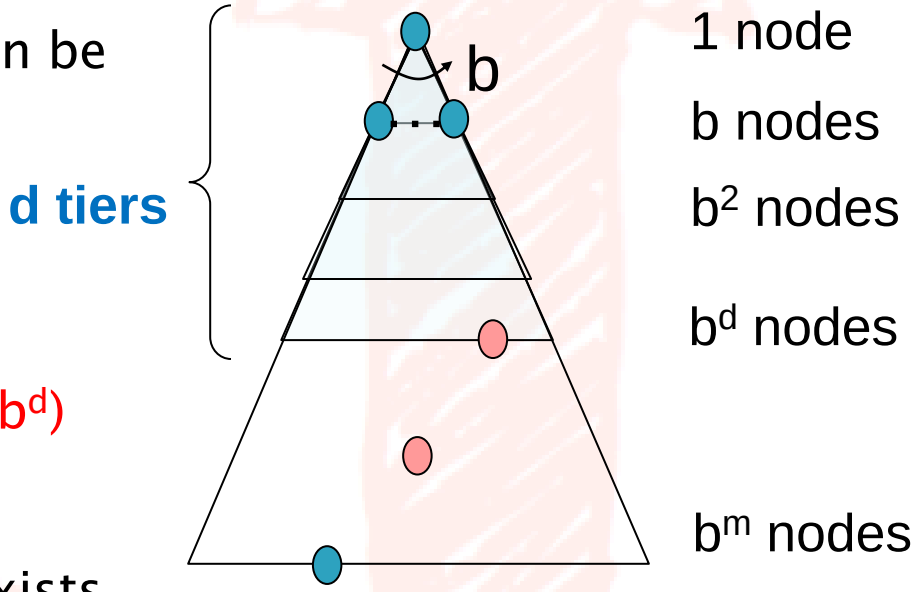
- Cartoon of search tree:
  - b maximum number of successors of any node
  - m is the maximum depth solutions at various depths



- 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 d
  - Search takes time $O(b^d)$
- How much space does the frontier take?
  - Has roughly the last tier, so $O(b^d)$
- Is it complete?
  - d must be finite if a solution exists, so yes!
- Is it optimal?
  - Only if costs are all the same (more on costs later)



Properties of Breadth-first search

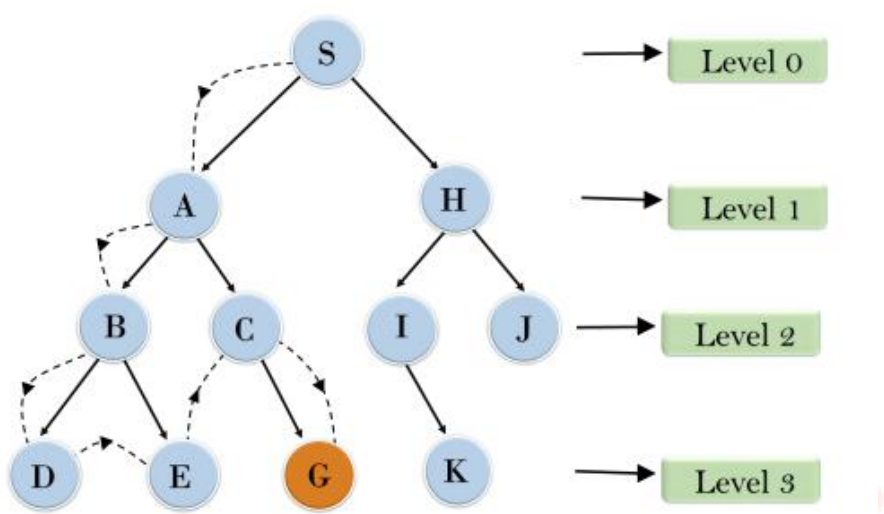
- Complete: Yes (if b is finite)
- Time: $1+b+b^2+b^3+\ldots+b^d=O\left(b^d\right)$
- Space: $O(b^d)$ (keeps every node in memory)
- Optimal: Yes (if cost $=$ 1 per step)
- Space is the bigger problem (more than time)

BFS: evaluation

- Memory requirements are a bigger problem than its execution time.
- Exponential complexity search problems cannot be solved by uninformed search methods for any but the smallest instances.

DEPTH	NODES	TIME	MEMORY
2	1100	0.11 seconds	1 megabyte
4	111100	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}	3523 years	1 exabyte

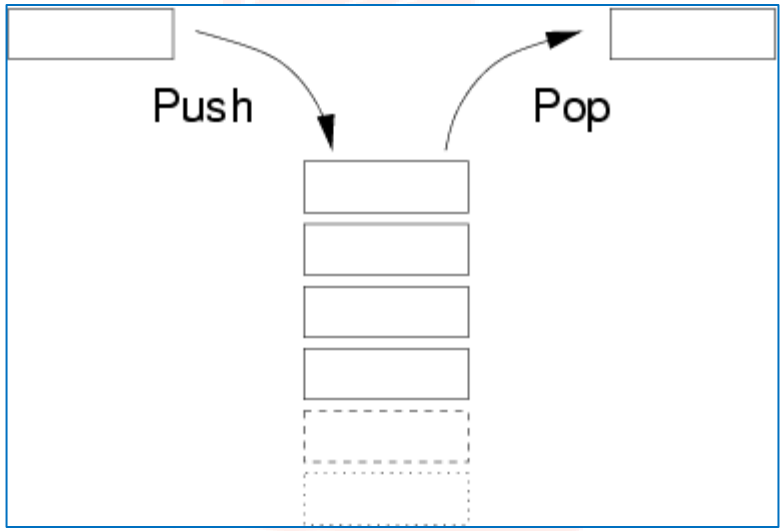
Depth-First search



Depth-first search

- Expand deepest unexpanded node

- Implementation:
  - *frontier* = LIFO (stack),
    i.e., put successors at front

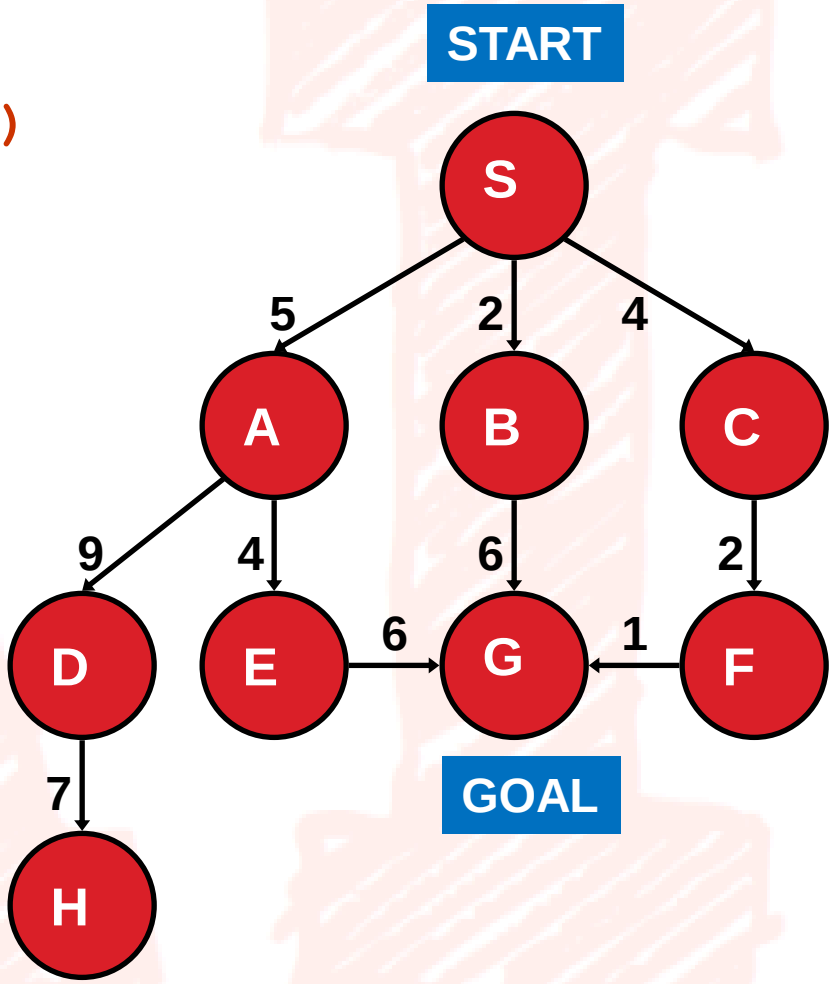


Depth-First Search (DFS)

TREE-SEARCH(problem, stack)

# of nodes tested: 0, expanded: 0

current	stack
	{S}

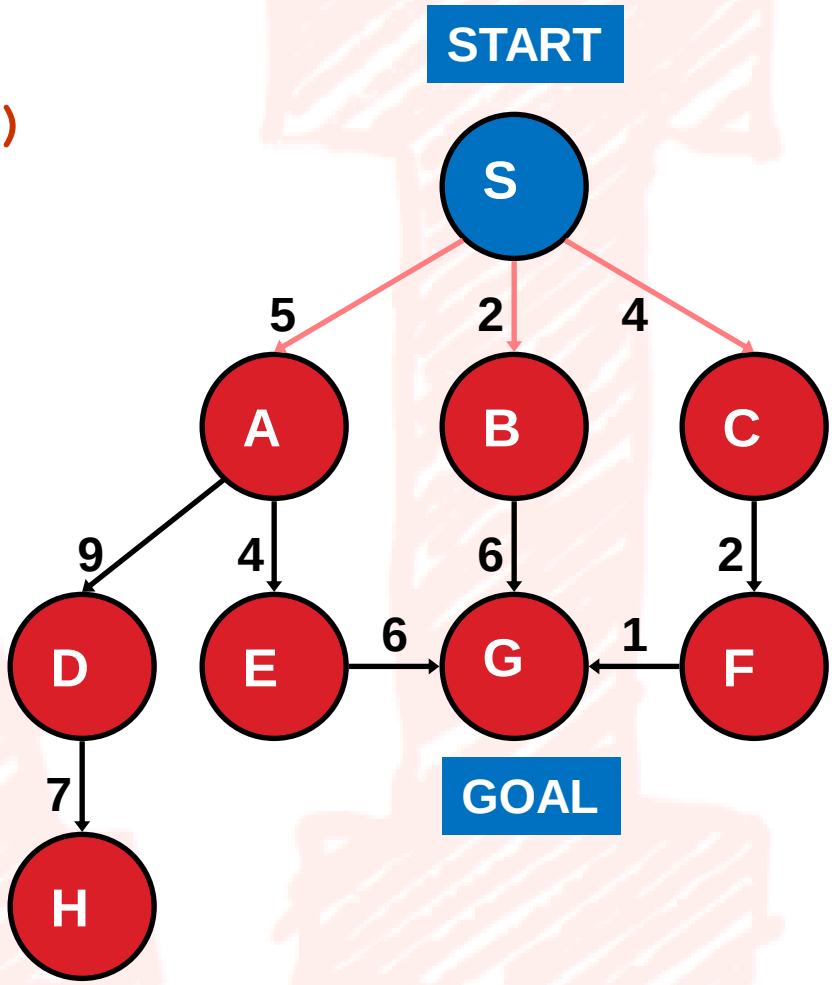


Depth-First Search (DFS)

TREE-SEARCH(problem, stack)

# of nodes tested: 1, expanded: 1

current	stack
	{S}
S not goal	{A,B,C}

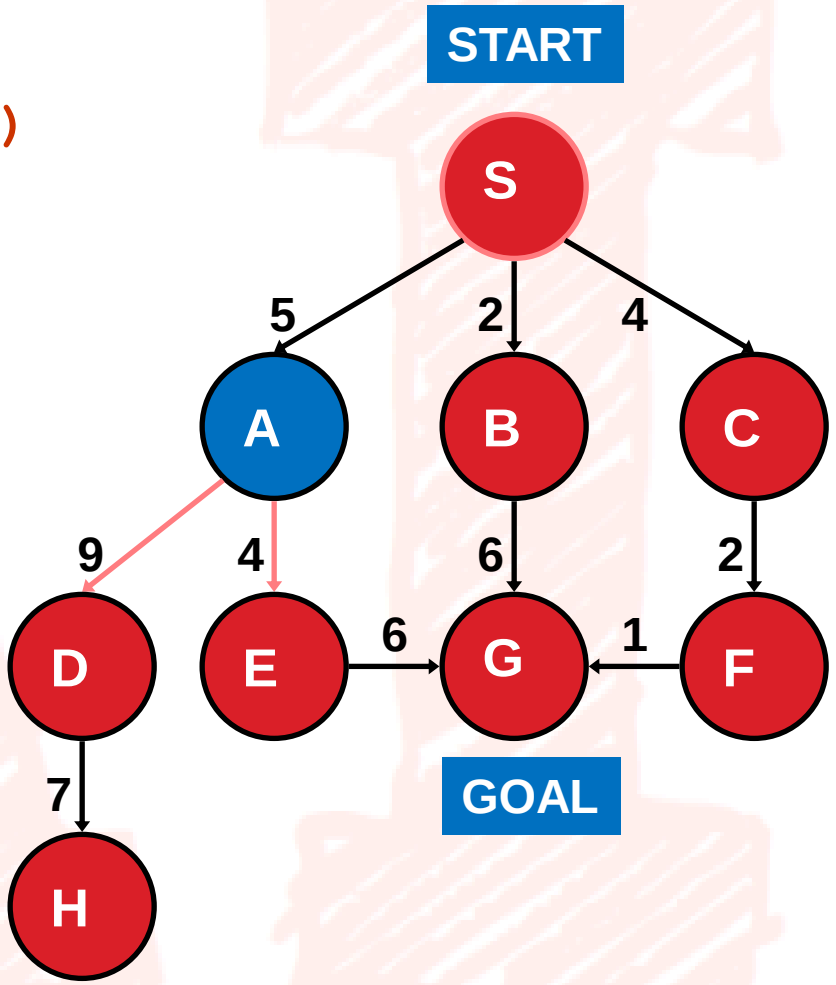


Depth-First Search (DFS)

TREE-SEARCH(problem, stack)

# of nodes tested: 2, expanded: 2

current	stack
	{S}
S	{A,B,C}
A not goal	{D,E,B,C}

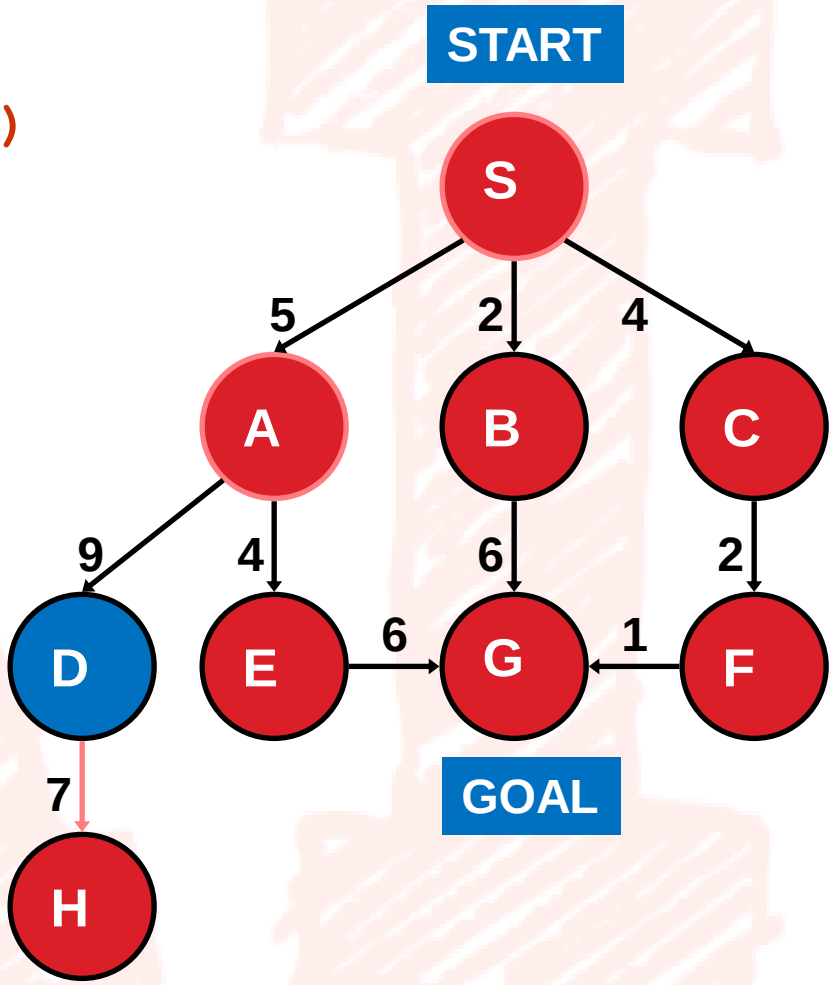


Depth-First Search (DFS)

TREE-SEARCH(problem, stack)

# of nodes tested: 3, expanded: 3

current	stack
	{S}
S	{A,B,C}
A	{D,E,B,C}
D not goal	{H,E,B,C}

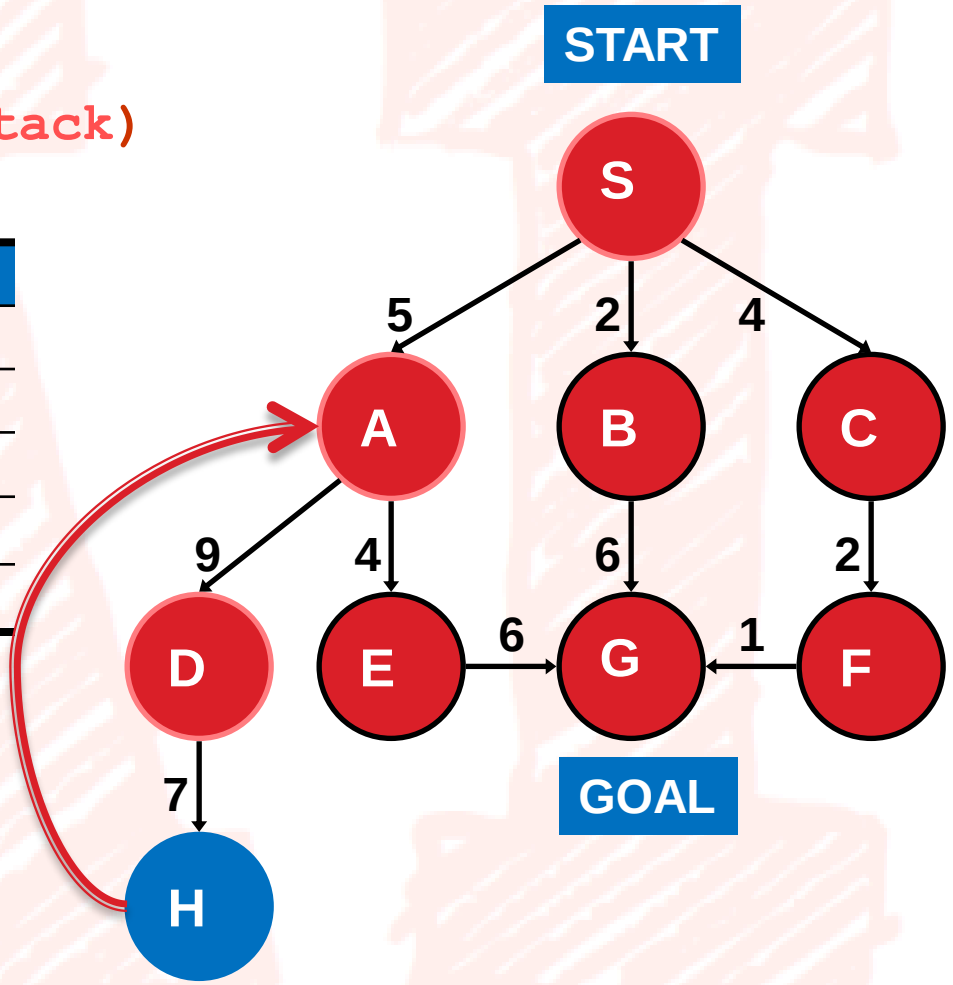


Depth-First Search (DFS)

TREE-SEARCH(problem, stack)

# of nodes tested: 4, expanded: 4

current	stack
	{S}
S	{A,B,C}
A	{D,E,B,C}
D	{H,E,B,C}
H not goal	{E,B,C}

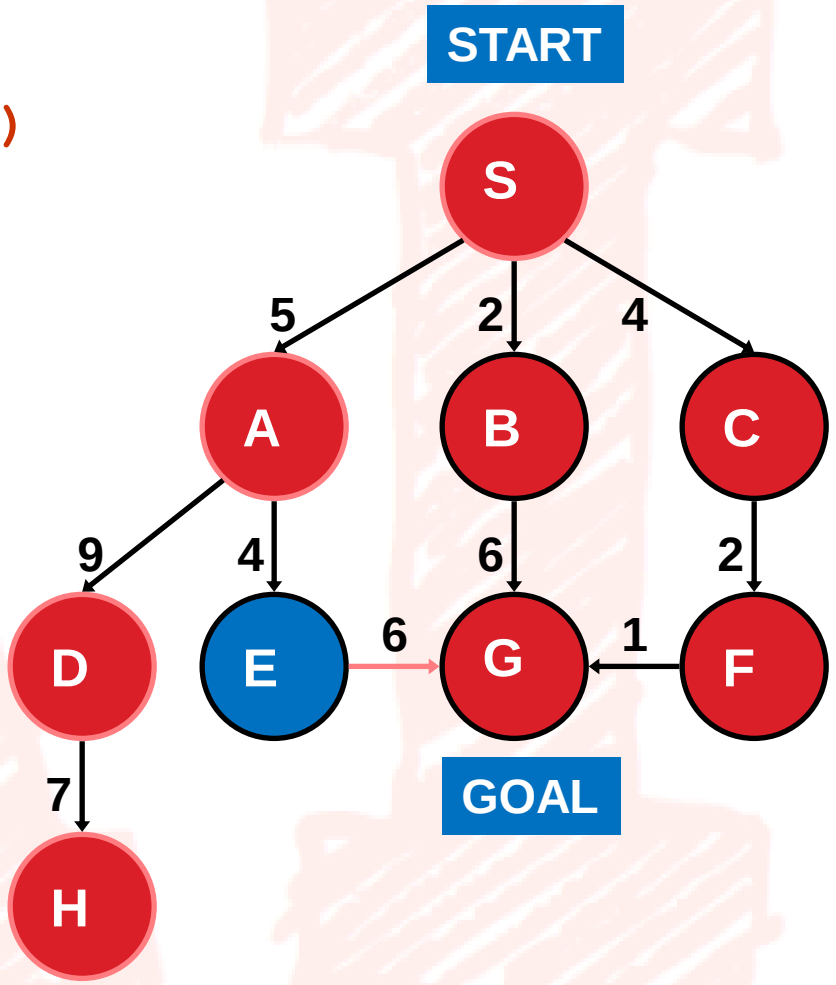


Depth-First Search (DFS)

TREE-SEARCH(problem, stack)

# of nodes tested: 5, expanded: 5

current	stack
	{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} backtracked

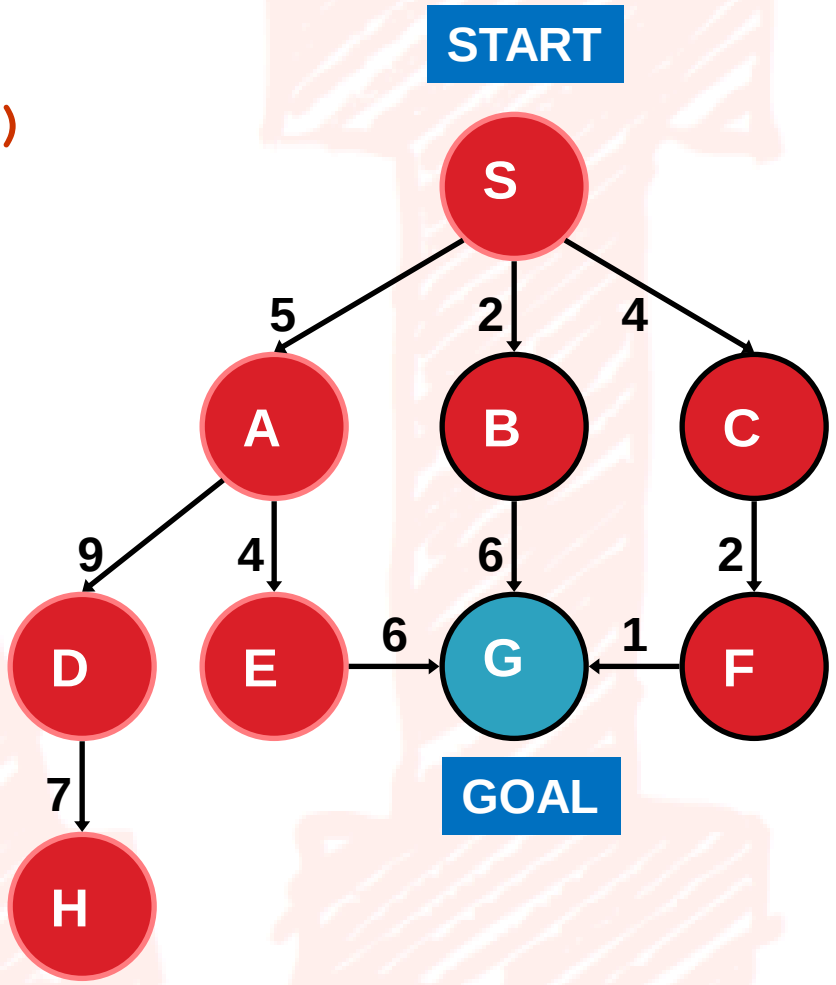


Depth-First Search (DFS)

TREE-SEARCH(problem, stack)

# of nodes tested: 6, expanded: 5

current	stack
	{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

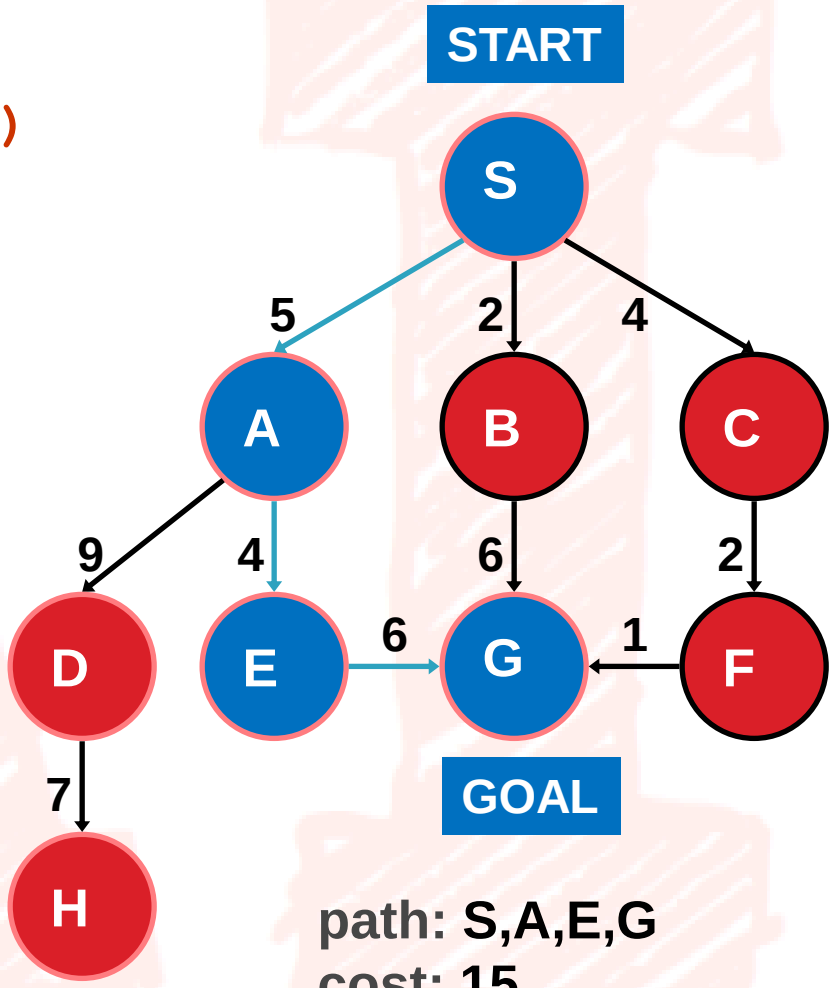


Depth-First Search (DFS)

TREE-SEARCH(problem, stack)

# of nodes tested: 6, expanded: 5

current	stack
	{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}

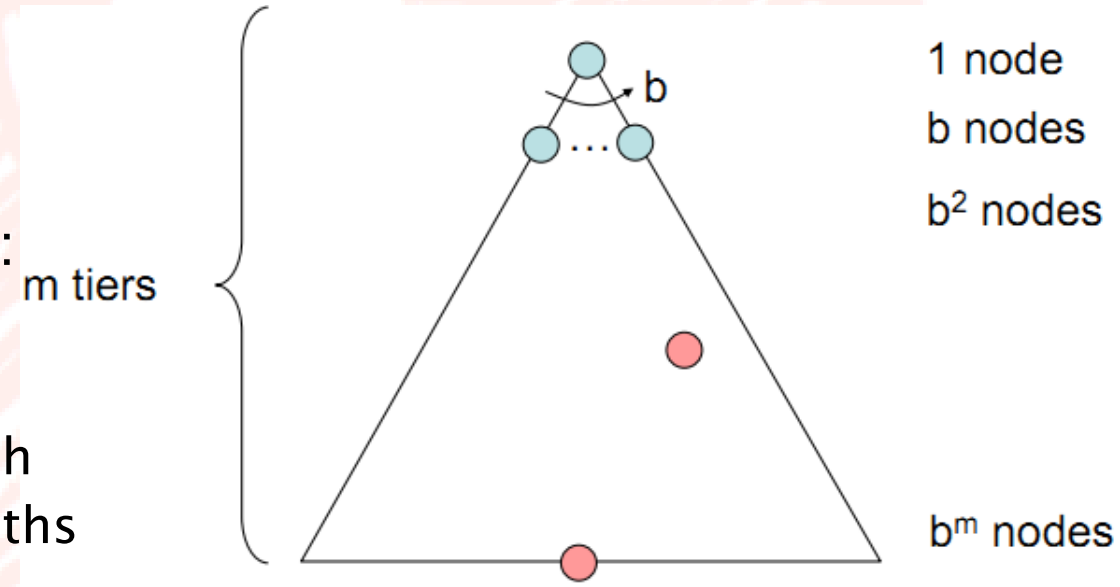


path: **S,A,E,G**
cost: **15**

Search Algorithm Properties

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

- Cartoon of search tree:
  - b maximum number of successors of any node
  - m is the maximum depth
  - solutions at various depths

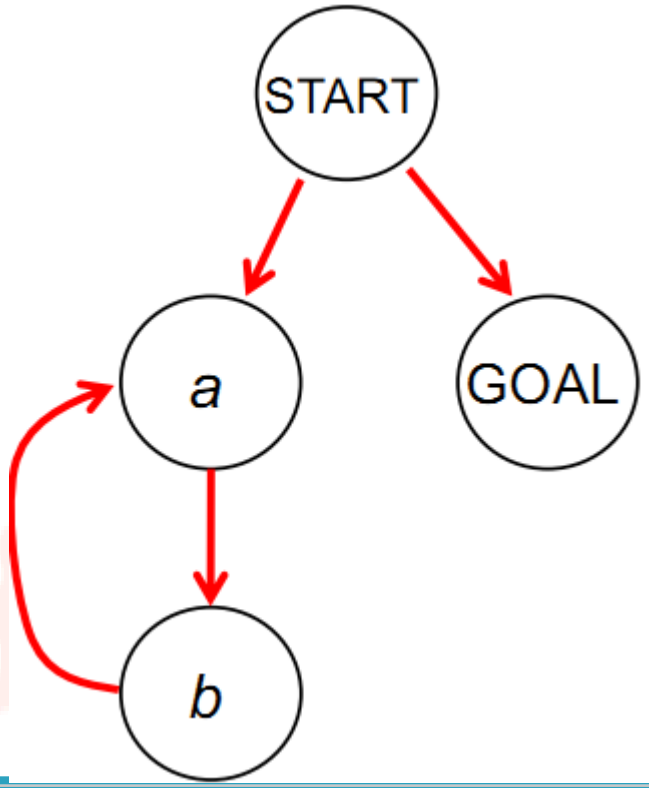


- Number of nodes in entire tree?
  - $1 + b + b^2 + \ldots + b^m = O(b^m)$

Properties of DFS

- Infinite paths make DFS incomplete…
  - How can we fix this?
  - Check new nodes against path from START

Assuming finite tree!!!



Analysis of DFS

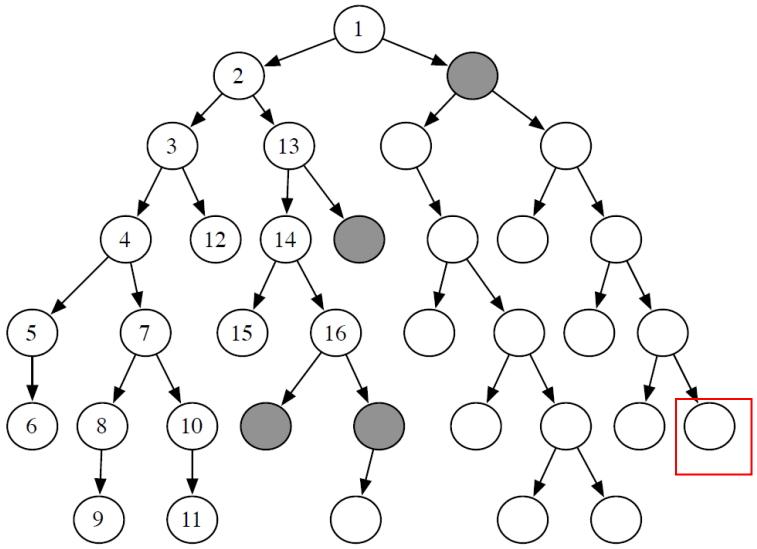
The time complexity of a search algorithm is the worst-case amount of time it will take to run, expressed in terms of

- maximum path length m
- maximum number of successors of any node b.

- What is DFS's time complexity, in terms of m and b ?

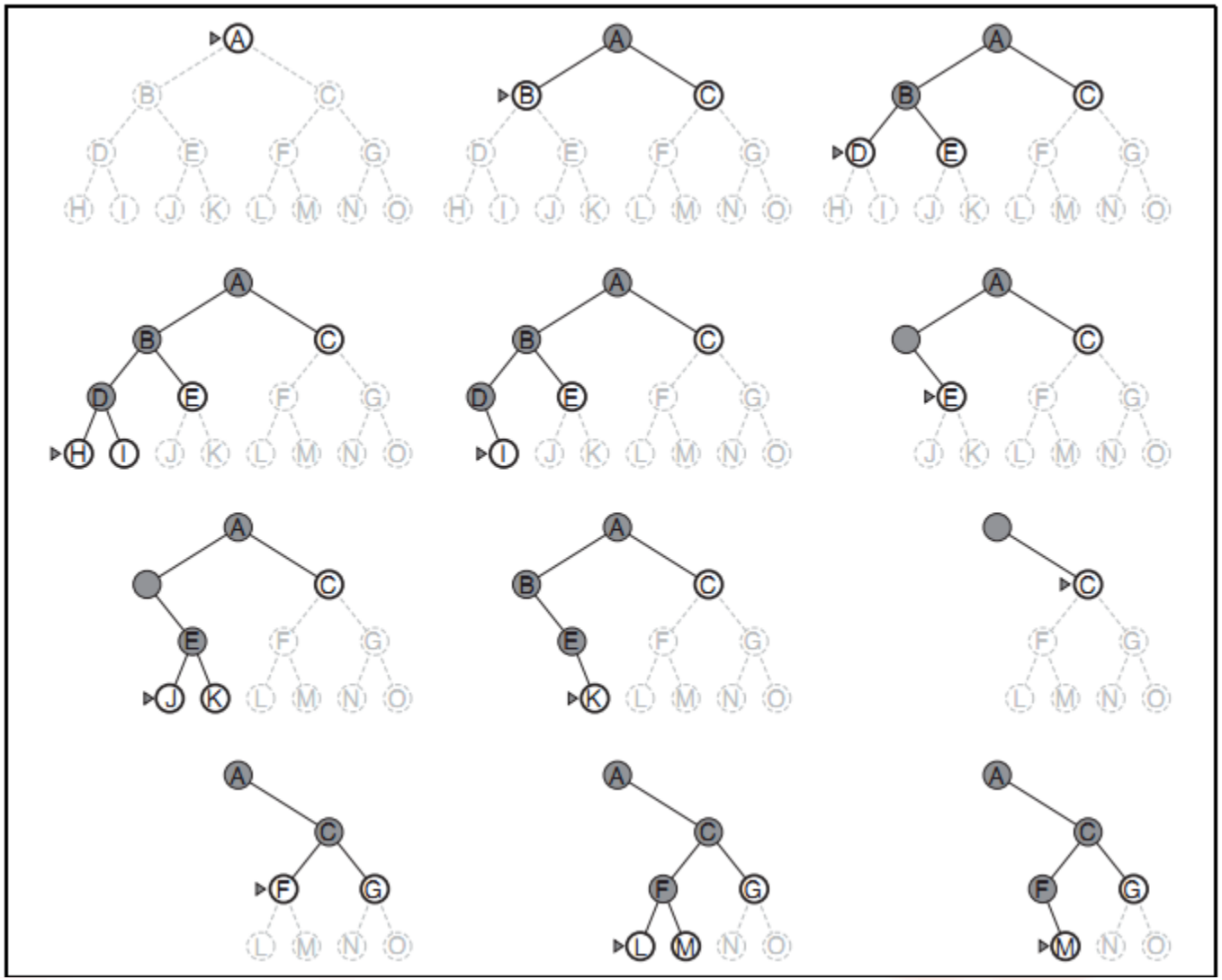
$O(b^m)$

- In the worst case, must examine every node in the tree
  - E.g., single goal node -> red box



Analysis of DFS (cont.)

How about space complexity?



Analysis of DFS (cont.)

The space complexity of a search algorithm is the worst-case amount of memory that the algorithm will use. Expressed in terms of

- maximum path length *m*
- maximum number of successors of any node *b*.

- What is DFS's space complexity, in terms of **m** and **b** ?

O(bm)

- for every node in the path currently explored, DFS maintains a path to its unexplored siblings in the search tree
- Alternative paths that DFS needs to explore

Properties of DFS

- Complete: No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path

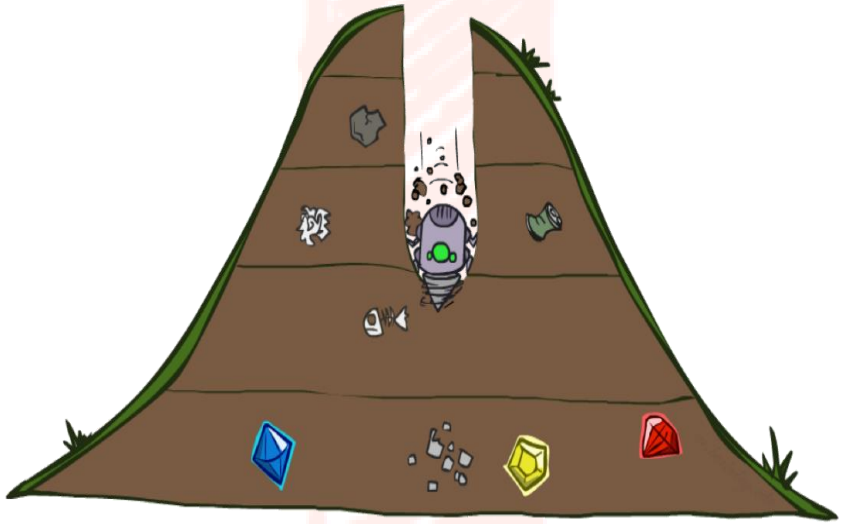
    → complete in finite spaces

- Time: $O(b^m)$: terrible if m is much larger than d
  - but if solutions are dense, may be much faster than breadth-first

- Space: $O(bm)$, i.e., linear space!

- Optimal: No

BFS vs DFS



Tree search algorithm

```
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
```

- ***frontier*** contains generated nodes which are not yet expanded.

Graph search algorithm

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

- ***frontier*** contains generated nodes which are not yet expanded.
- ***explored*** list stores all expanded nodes.

Graph search: evaluation

- Optimality:
  - GRAPH-SEARCH discard newly discovered paths.
    - This may result in a sub-optimal solution
    - YET: when uniform-cost search or breadth-first search with constant step cost
- Time and space complexity:
  - proportional to the size of the state space (may be much smaller than O(bd)).
  - DF- and ID-search with explored list no longer has linear space requirements since all nodes are stored in explored list!!

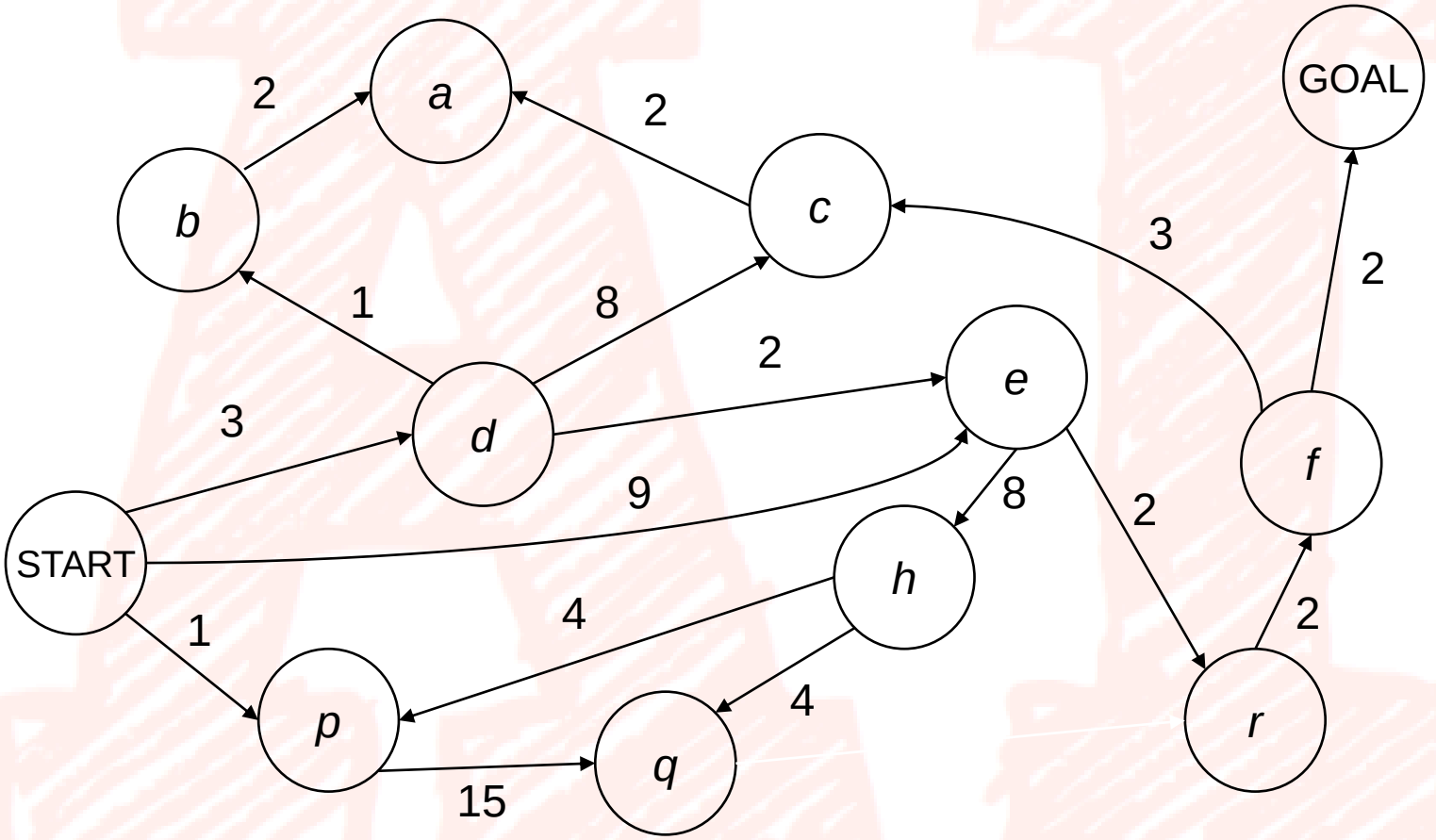
Hints

- On small problems
  - Graph search almost always better than tree search
  - Implement your explored list as a dict or set!

- On many real problems
  - Storage space is a huge concern
  - Graph search impractical

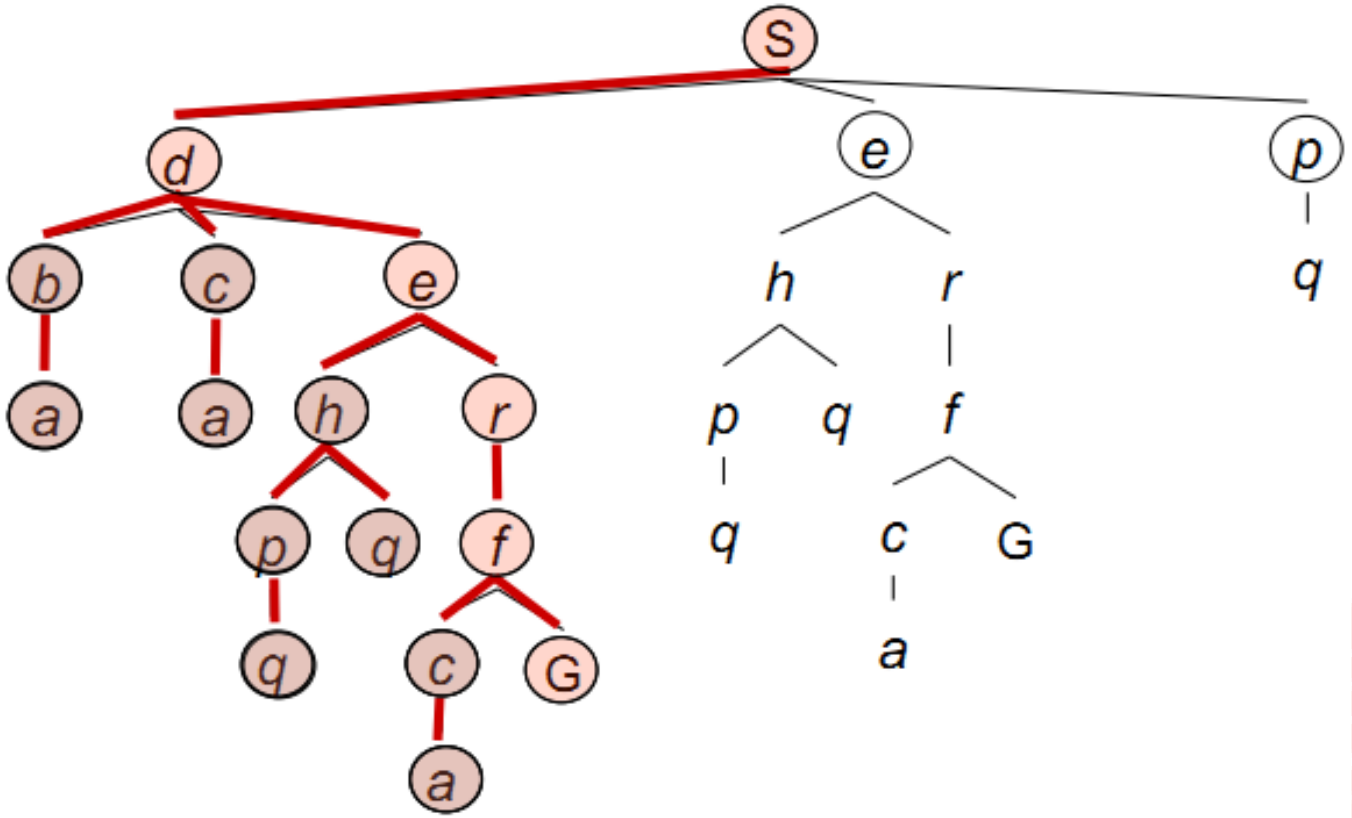
Exercise 1

- BFS, DFS from START to GOAL



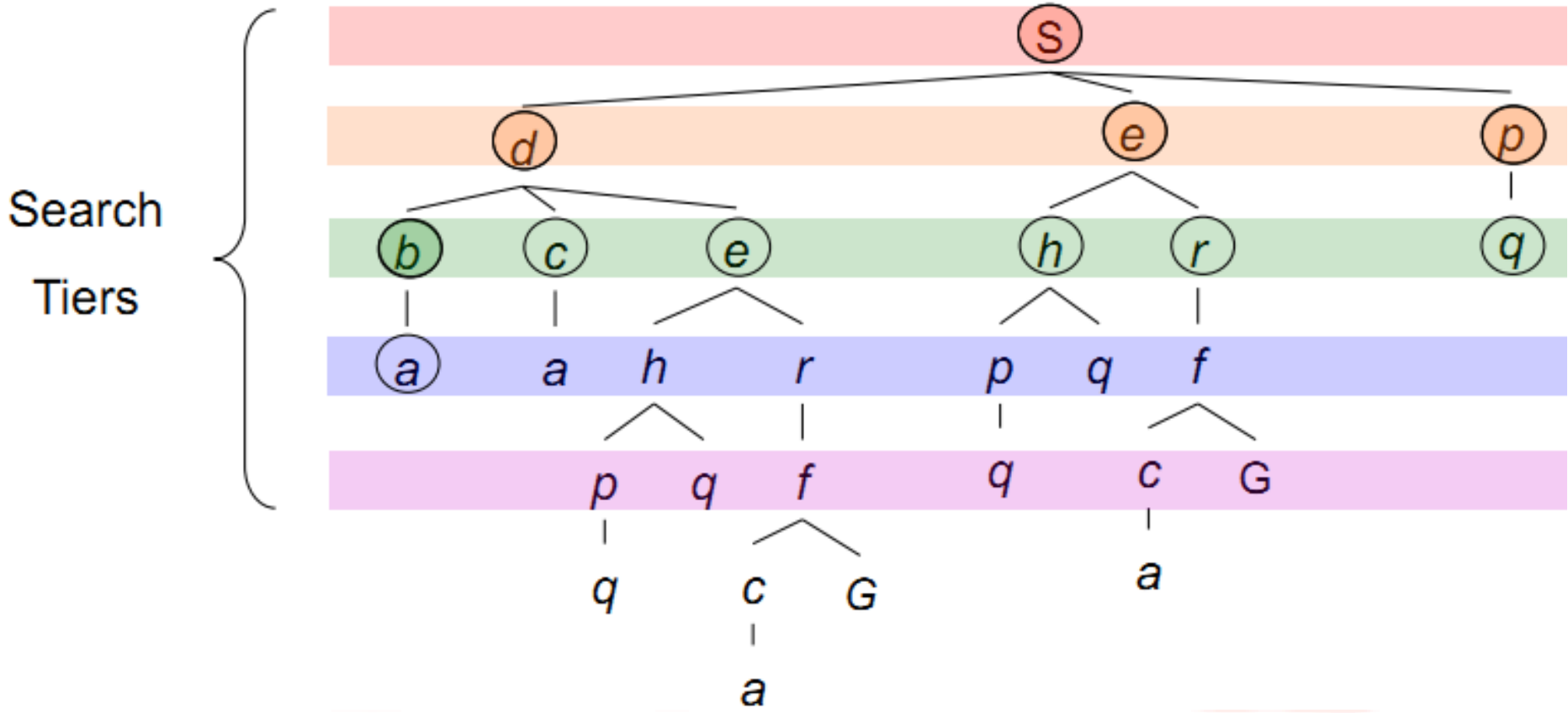
Depth first search

- Expansion order: (S, d, b, a, c, a, e, h, p, q, q, r, f, c, a, G)



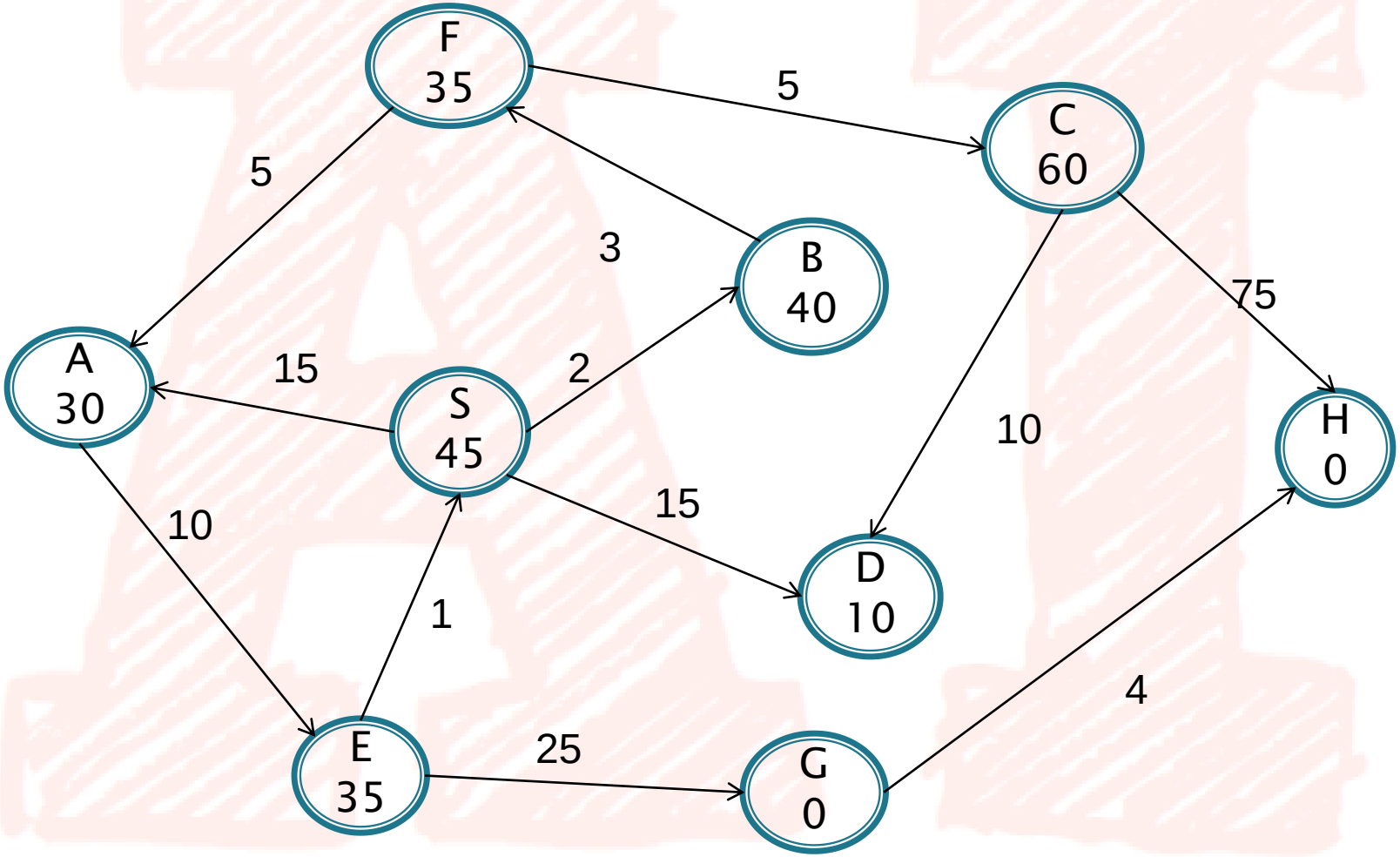
Breadth first search

- Expansion order : (S, d, e, p, b, c, e, h, r, q, a, a, h, r, p, q, f, p, q, f, q, c, G)

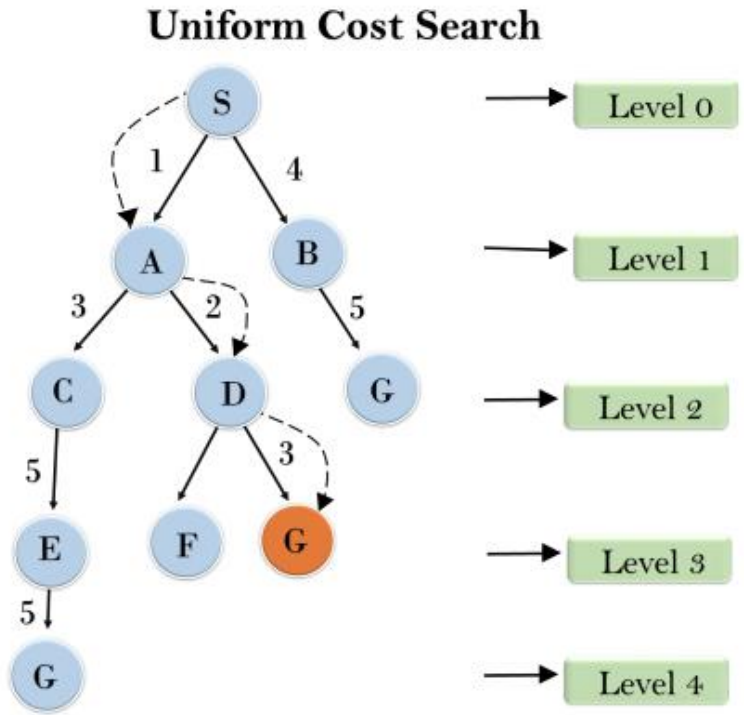


Exercise 2

- BFS, DFS from **S** to **G** (or **H**)



Uniform-Cost Search



Uniform-Cost Search

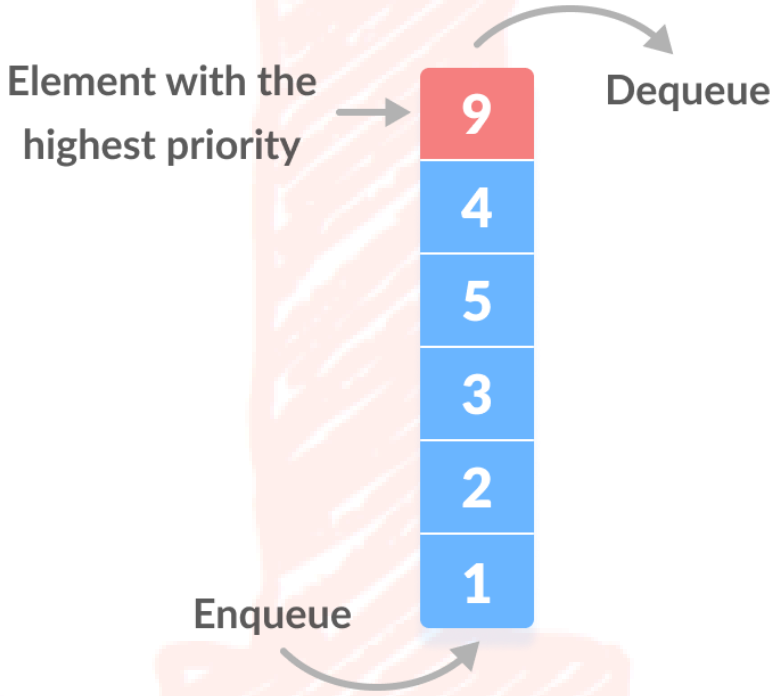
- UCS: uniform-cost search

  Priority queue used to order nodes, **sorted by path cost**

  - Let g(n) = cost of path from start node s to current node n
  - Sort nodes by increasing value of g
  - Only uninformed search that worries about costs
- Called *Dijkstra's Algorithm* in algorithms.
- Similar to *Branch and Bound* Algorithm in operations research literature

Uniform-Cost Search

- Expand least-cost unexpanded node
- Implementation:
  - frontier = queue ordered by path cost
- Equivalent to *BFS if step costs all equal*

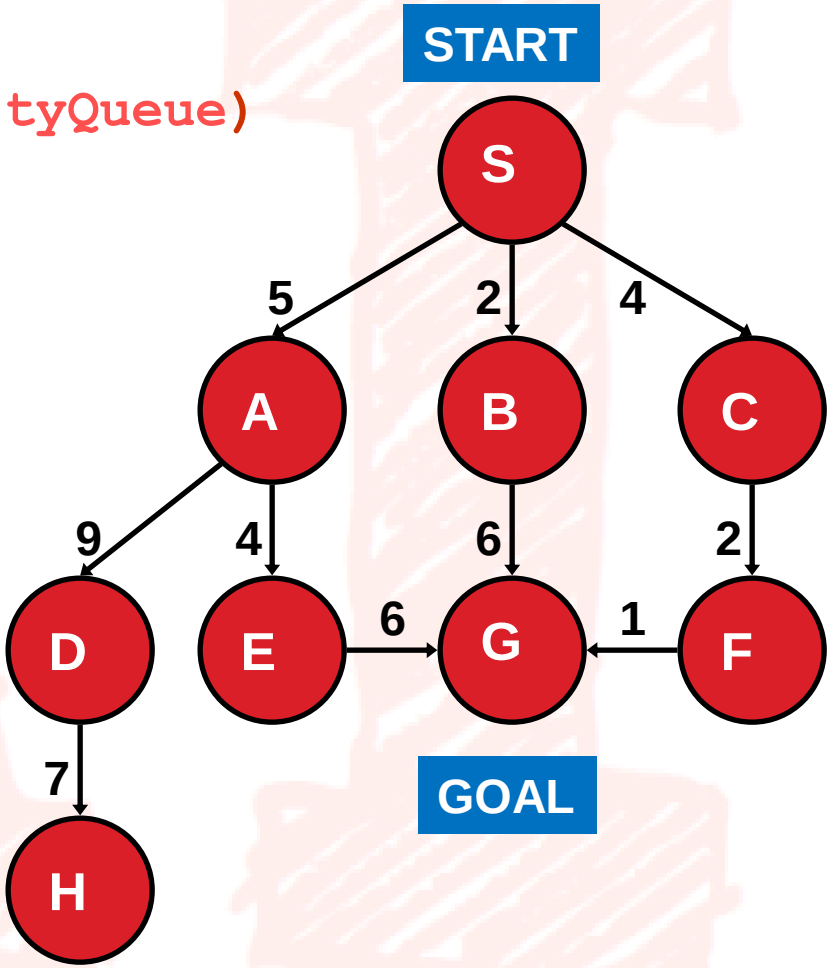


Uniform-Cost Search (UCS)

TREE-SEARCH(problem, priorityQueue)

# of nodes tested: 0, expanded: 0

current	priorityQueue
	{S}

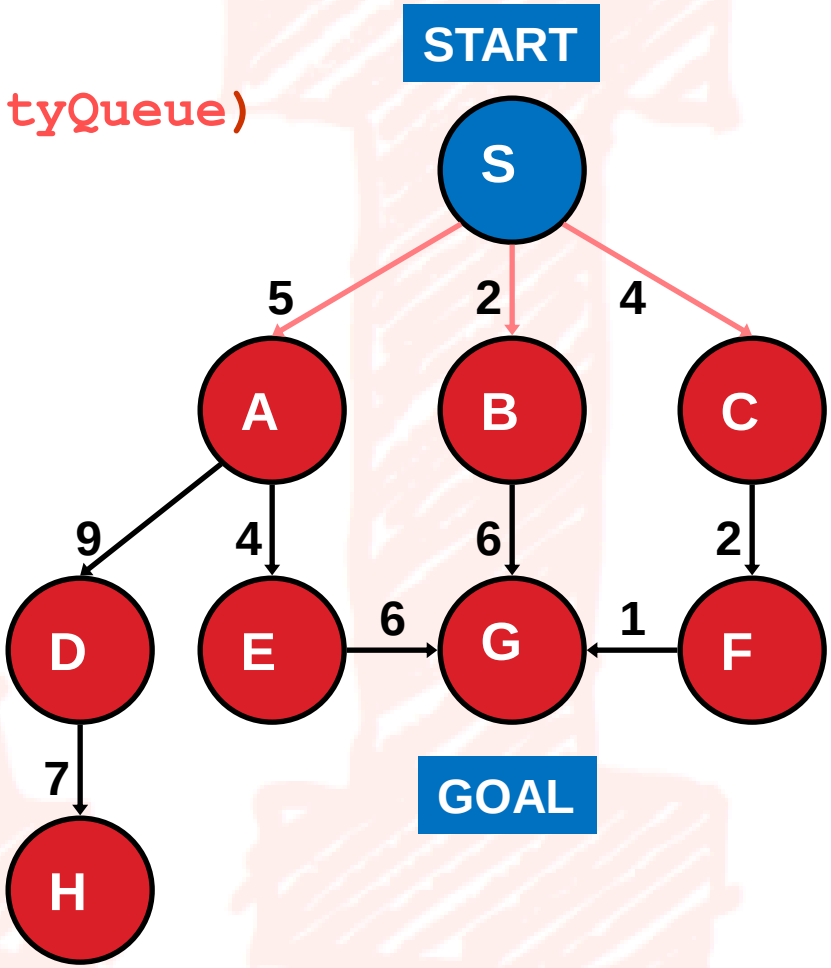


Uniform-Cost Search (UCS)

TREE-SEARCH(problem, priorityQueue)

# of nodes tested: 1, expanded: 1

current	priorityQueue
	{S:0}
S not goal	{B:2,C:4,A:5}

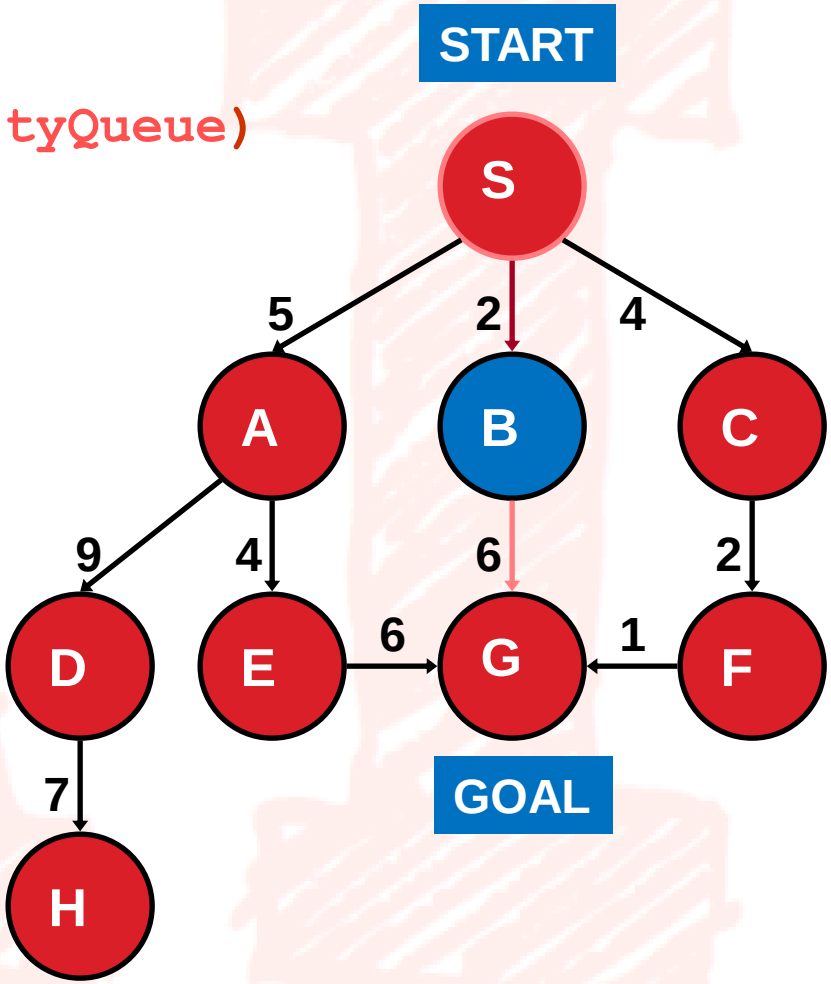


Uniform-Cost Search (UCS)

TREE-SEARCH(problem, priorityQueue)

# of nodes tested: 2, expanded: 2

current	priorityQueue
	{S}
S	{B:2,C:4,A:5}
B not goal	{C:4,A:5,G:2+6}

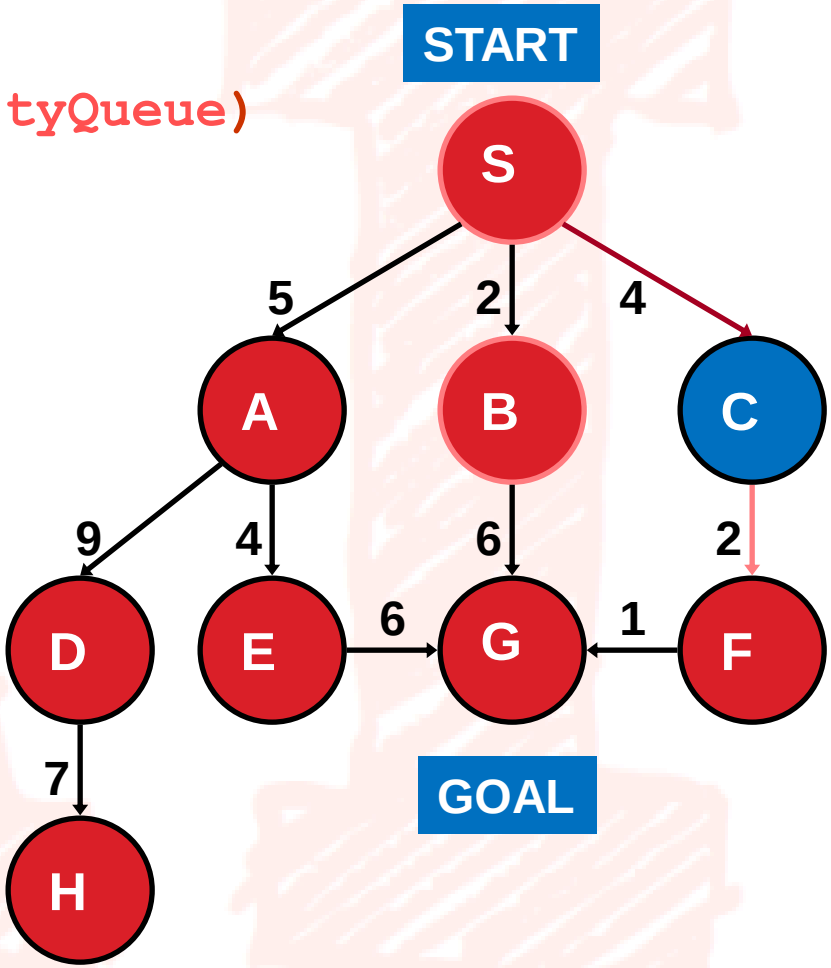


Uniform-Cost Search (UCS)

TREE-SEARCH(problem, priorityQueue)

# of nodes tested: 3, expanded: 3

current	priorityQueue
	{S}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C not goal	{A:5,F:4+2,G:8}

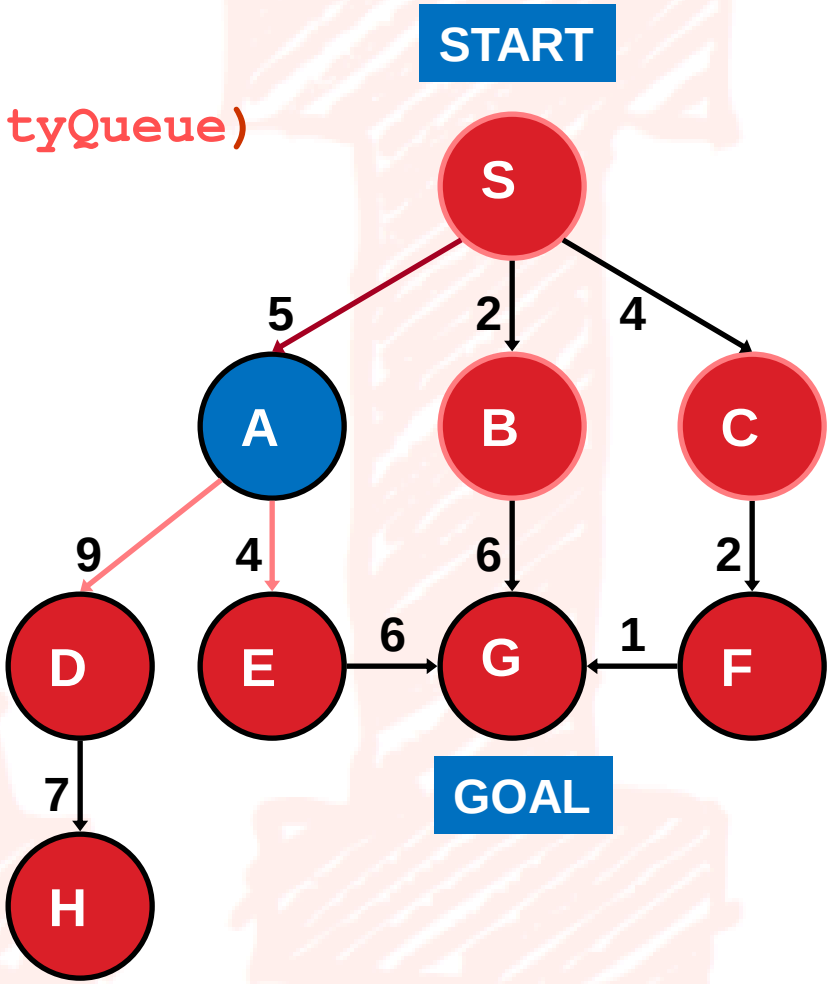


Uniform-Cost Search (UCS)

TREE-SEARCH(problem, priorityQueue)

# of nodes tested: 4, expanded: 4

current	priorityQueue
	{S}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C	{A:5,F:6,G:8}
A not goal	{F:6,G:8,E:5+4, D:5+9}

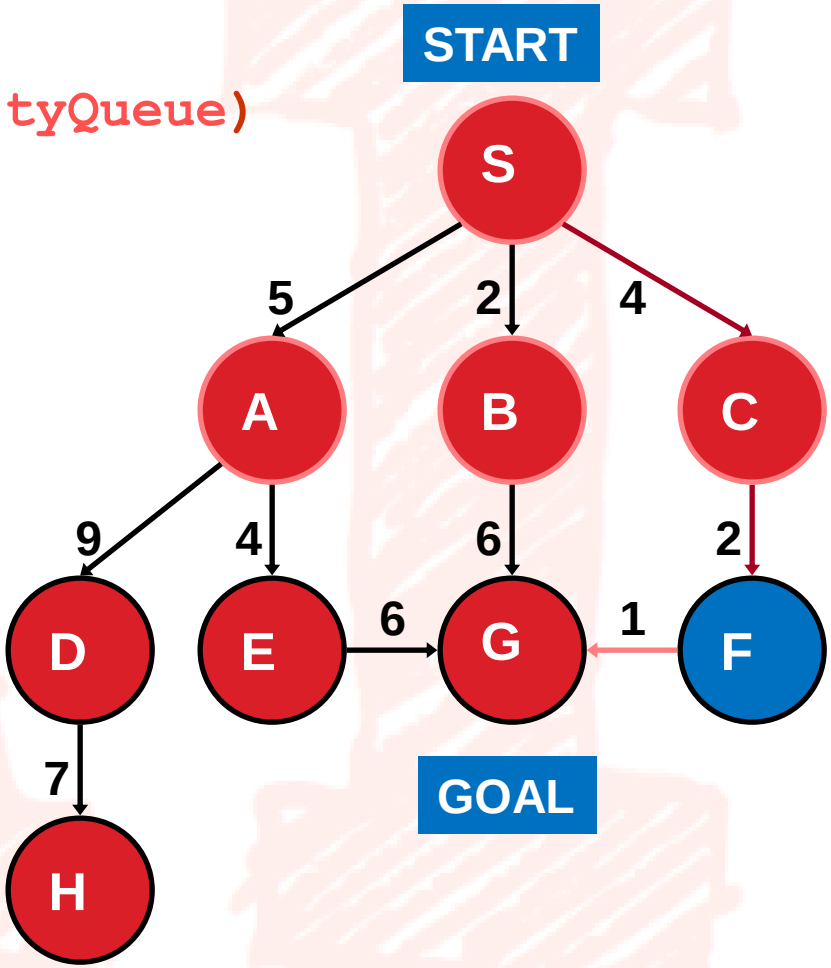


Uniform-Cost Search (UCS)

TREE-SEARCH(problem, priorityQueue)

# of nodes tested: 5, expanded: 5

current	priorityQueue
	{S}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C	{A:5,F:6,G:8}
A	{F:6,G:8,E:9,D:14}
F not goal	{G:6+1,G:8,E:9,D:14}

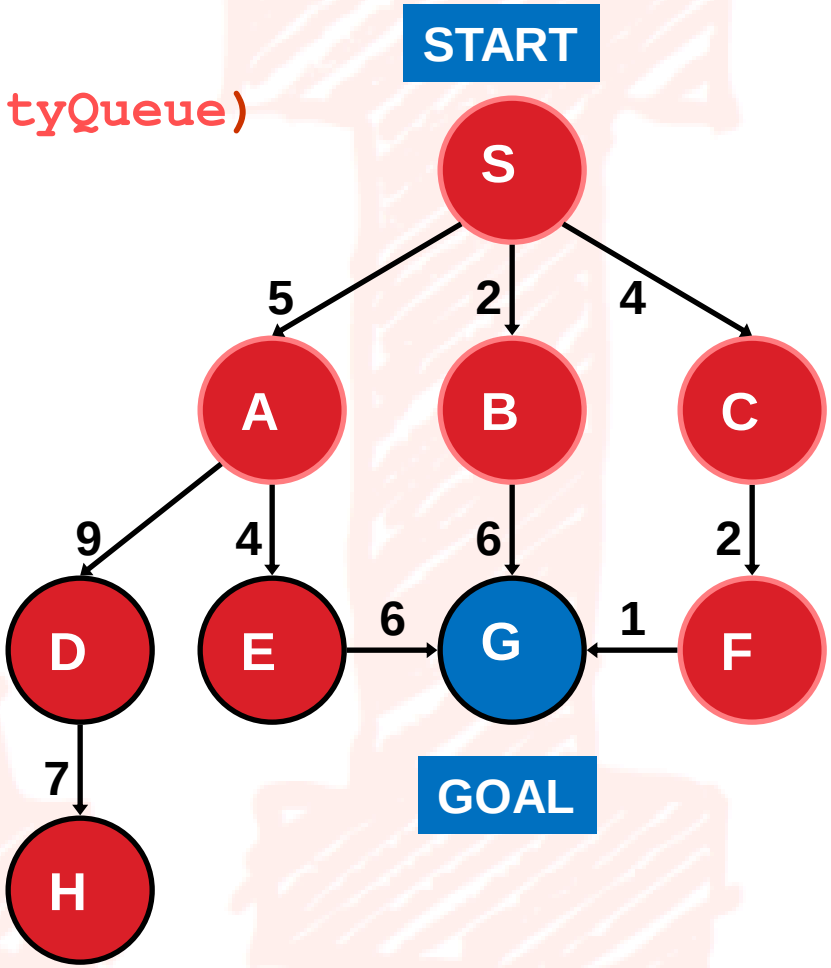


Uniform-Cost Search (UCS)

TREE-SEARCH(problem, priorityQueue)

# of nodes tested: 6, expanded: 5

current	priorityQueue
	{S}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C	{A:5,F:6,G:8}
A	{F:6,G:8,E:9,D:14}
F	{G:7,G:8,E:9,D:14}
G goal	{G:8,E:9,D:14} no expand

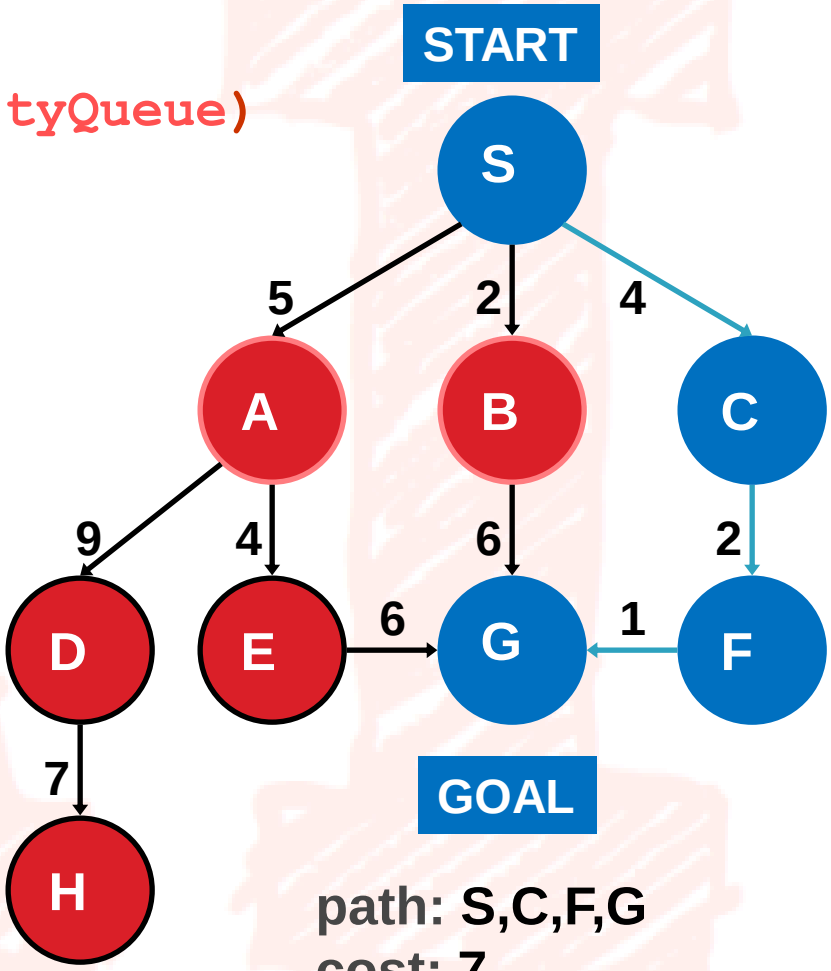


Uniform-Cost Search (UCS)

`TREE-SEARCH(problem, priorityQueue)`

# of nodes tested: 6, expanded: 5

current	priorityQueue
	{S}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C	{A:5,F:6,G:8}
A	{F:6,G:8,E:9,D:14}
F	{G:7,G:8,E:9,D:14}
G	{G:8,E:9,D:14}



path: **S,C,F,G**
cost: **7**

Pseudo code

Graph search

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure

  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

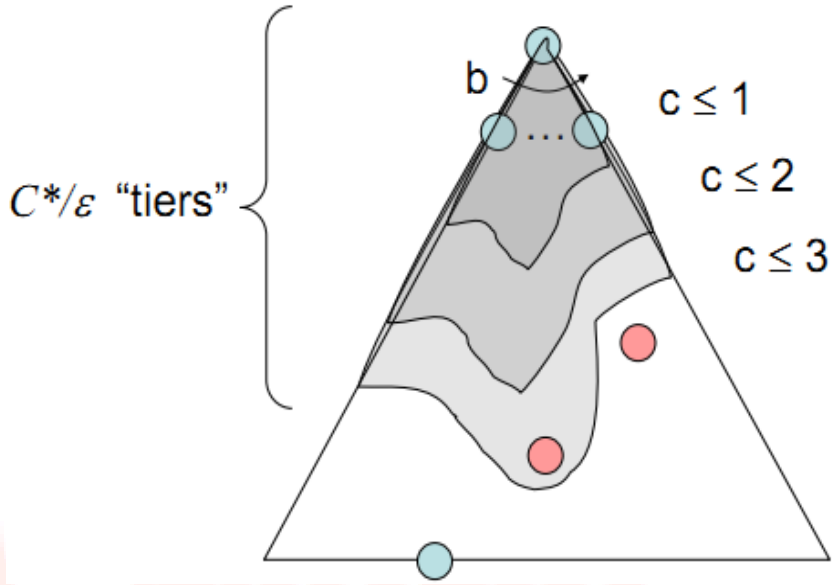
Evaluating UCS

- UCS: guided by path costs rather than depths,
- Let C^* be the cost of the optimal solution, and assume that every action costs at least ϵ.
- The worst-case:
  - time and space complexity:

$$O\left(b^{\left[C^*/\varepsilon\right]}\right)$$

- If all step costs are equal:

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

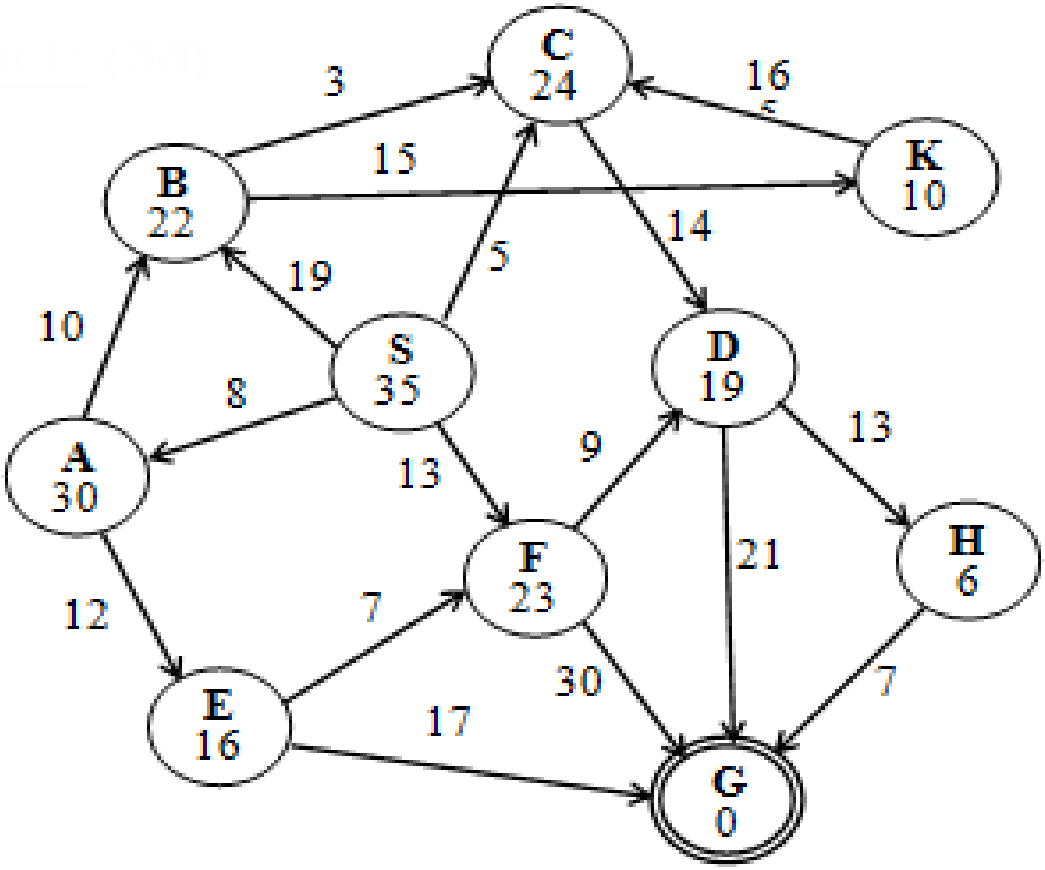


Evaluating UCS (cont.)

- Complete (if b is finite, cost $>= \varepsilon$)
- Optimal: Yes but
  - requires the goal test to be applied when a node is removed from the frontier rather than when the node is first generated when its parent node is expanded
- Time and space complexity: $O\left(b^{\lceil C^*/\varepsilon \rceil}\right)$
  - $O(b^d)$ if assuming all edges have the same cost
- Is it optimal?
  - Yes! (Proof next lecture via A*)

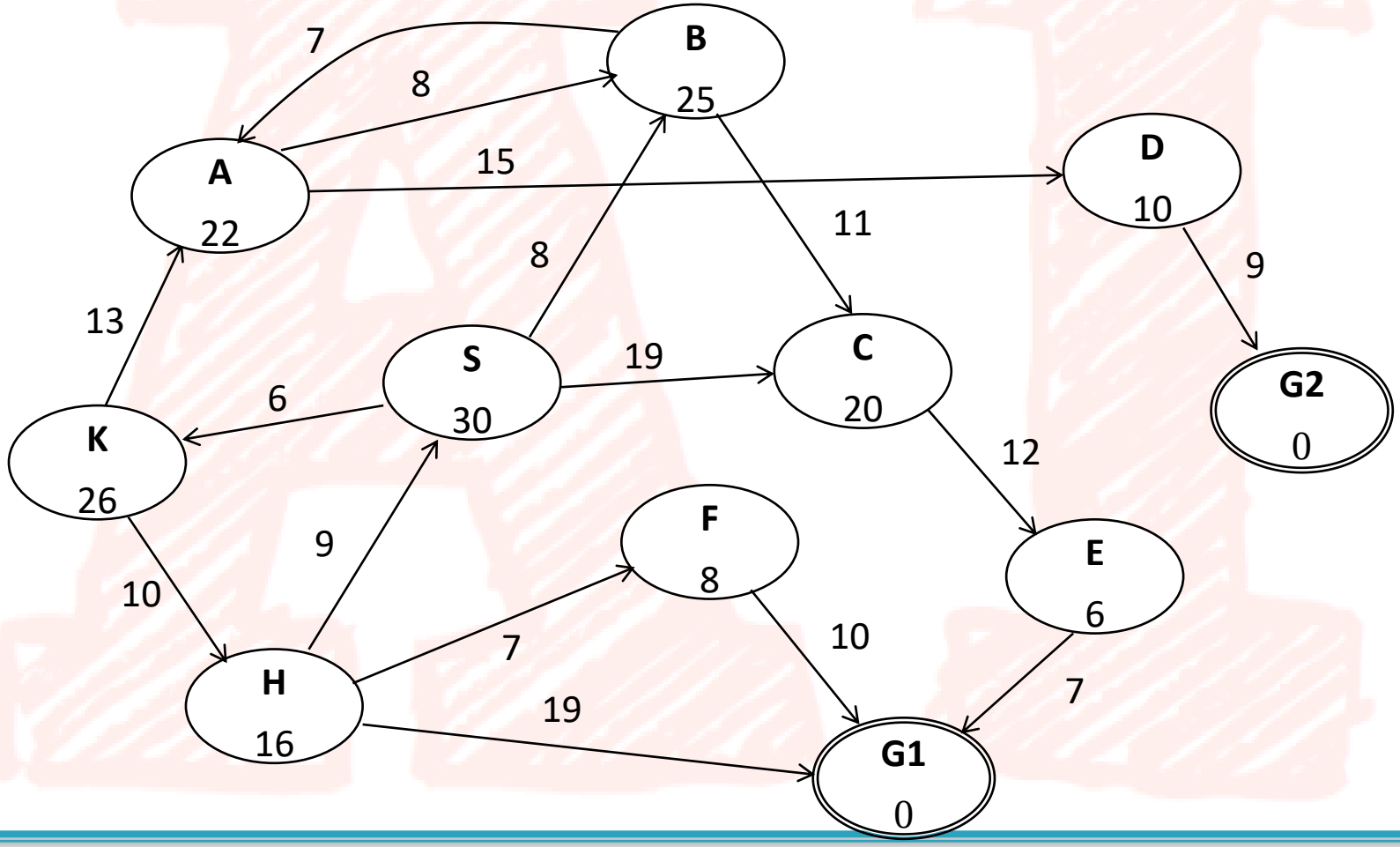
Exercise

- How to get G from S using BFS, DFS, UCS:

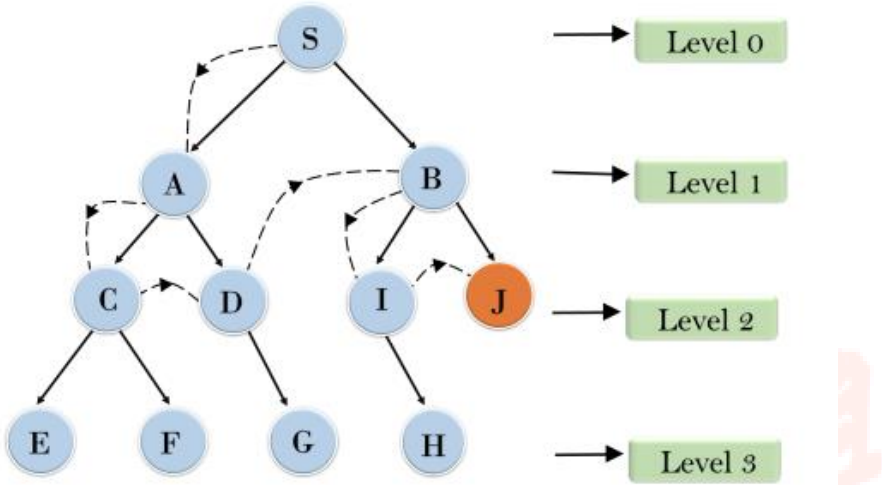


Exercise

- How to get G1(G2) from S using BFS, DFS, UCS:



Depth-limited search



Depth-limited search

- Is DFS with depth limit l
  - i.e. nodes at depth l have no successors.
  - Problem knowledge can be used
- Solves the infinite-path problem.
  - If $l < d$ then incompleteness results.
  - If $l > d$ then not optimal.
- Time complexity: $O(b^l)$
- Space complexity: $O(bl)$

Depth-limited algorithm

- **Recursive implementation:**

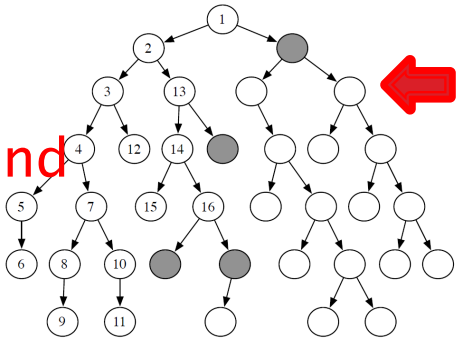
```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

- Failure: indicates no solution;
- Cutoff: indicates no solution within the depth limit.

Iterative Deepening Search

Iterative deepening search

- What?
  - A general strategy to find best depth limit *L*.
    - Goals is found at depth *d*, the depth of the shallowest goal-node.
  - Often used in combination with depth first search
- Combines benefits of DFS and BFS
  - do DFS to depth 0 treat start node as leaf
  - do DFS to depth 1 treat all children of the start node as leaves
  - if no solution found, do DFS to depth 2
  - repeat by increasing depth until a solution found
- Start node is at depth 0



Iterative deepening search (IDS)

- The iterative deepening search algorithm, which 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.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    for depth = 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```

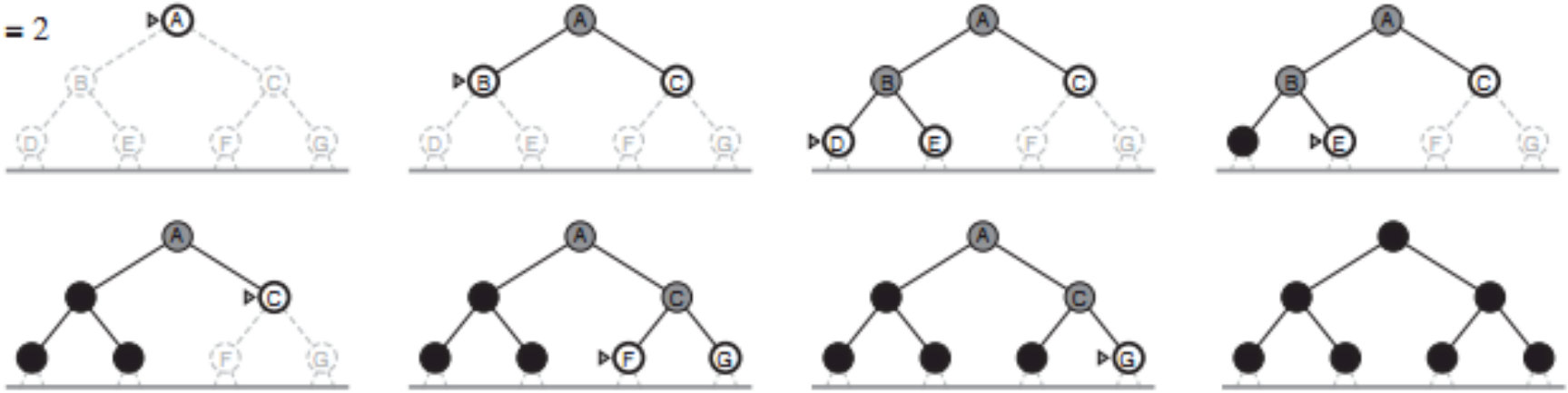
Iterative deepening search L = 0, 1

Limit = 0

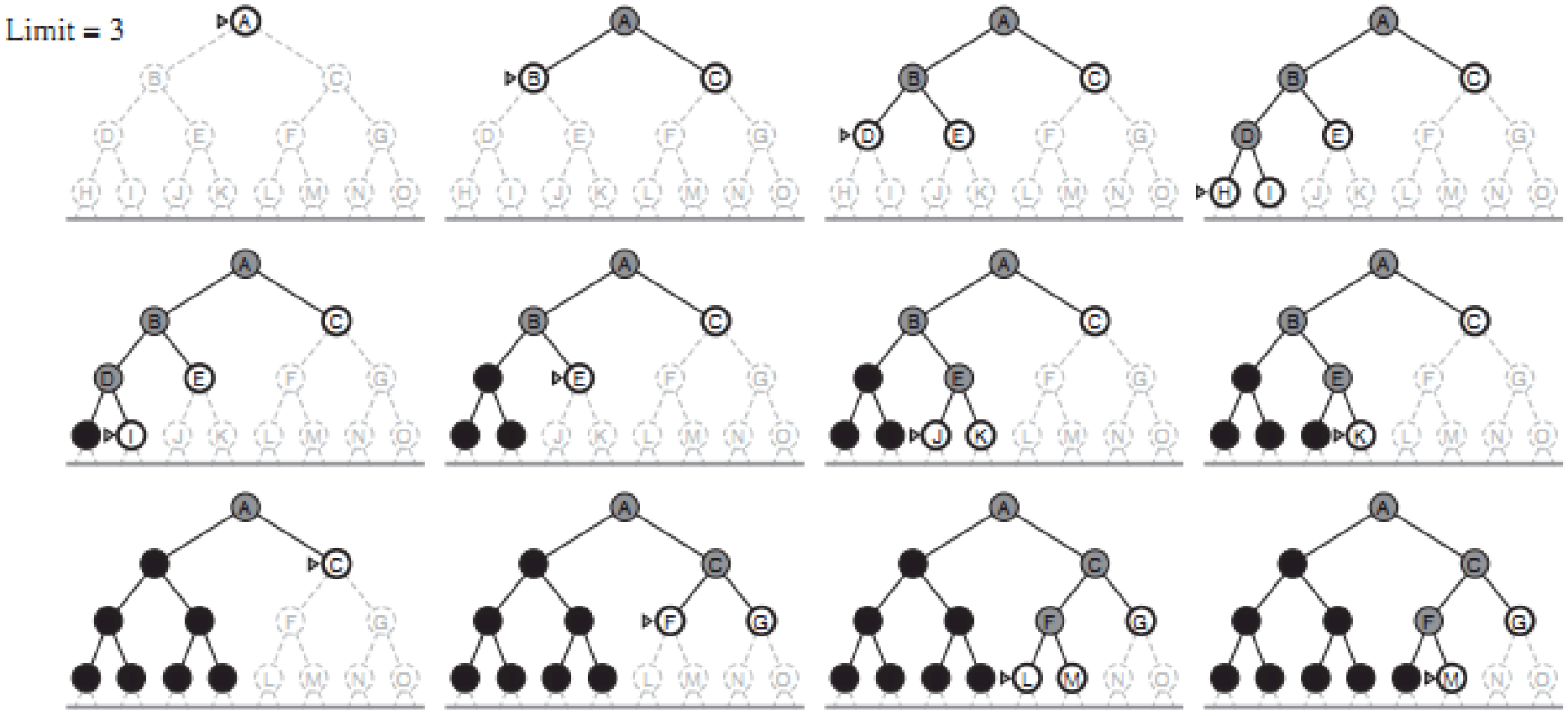
Limit = 1

Iterative deepening search L = 2

Limit = 2



Iterative deepening search L = 3

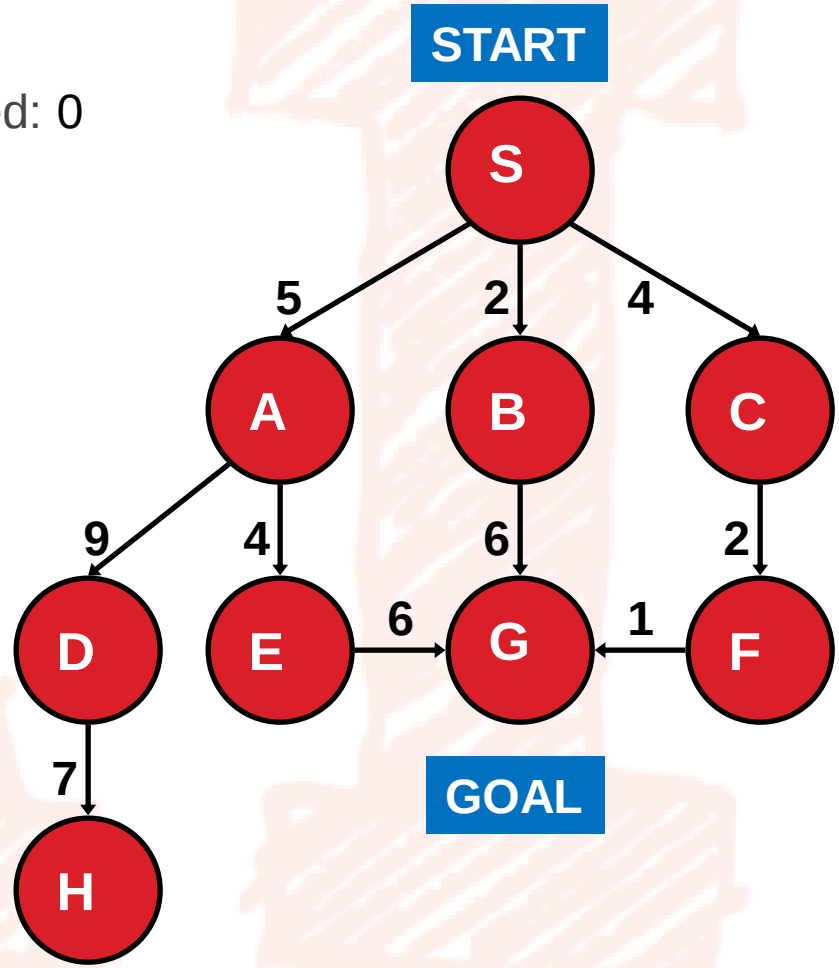


Iterative-Deepening Search (IDS)

`deepeningSearch(problem)`

depth: 0, # of nodes tested: 0, expanded: 0

current	nodes list
	{S}

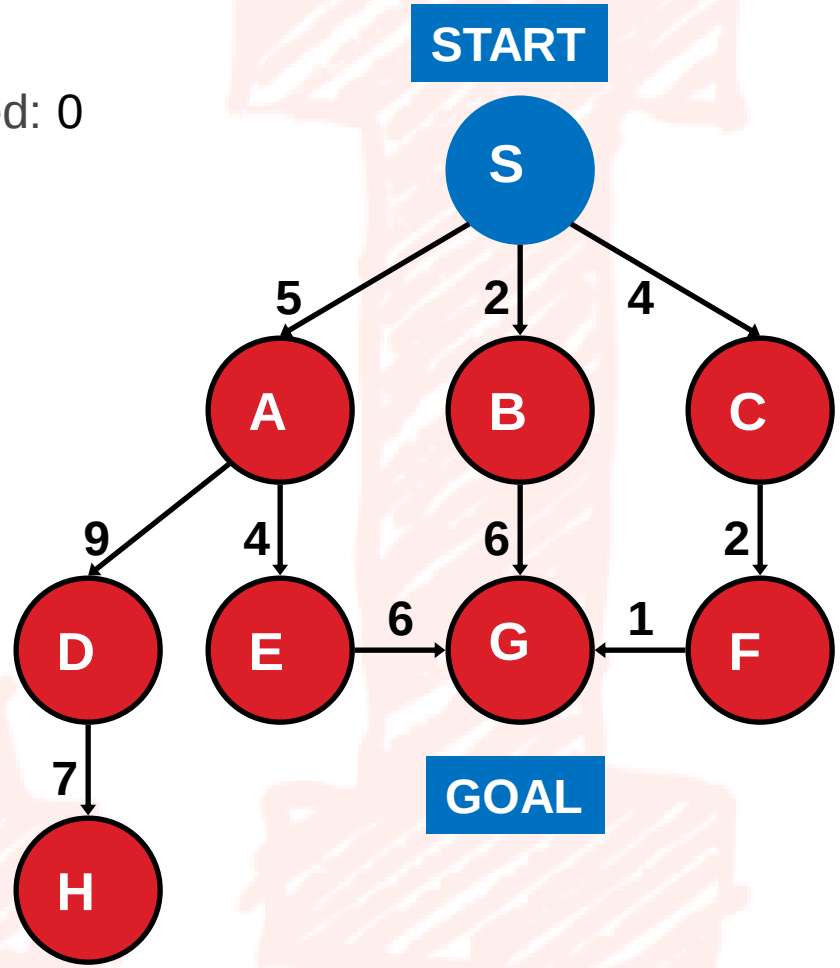


Iterative-Deepening Search (IDS)

`deepeningSearch(problem)`

depth: 0, # of nodes tested: 1, expanded: 0

current	nodes list
	{S}
S not goal	{ } @cutoff-FAIL

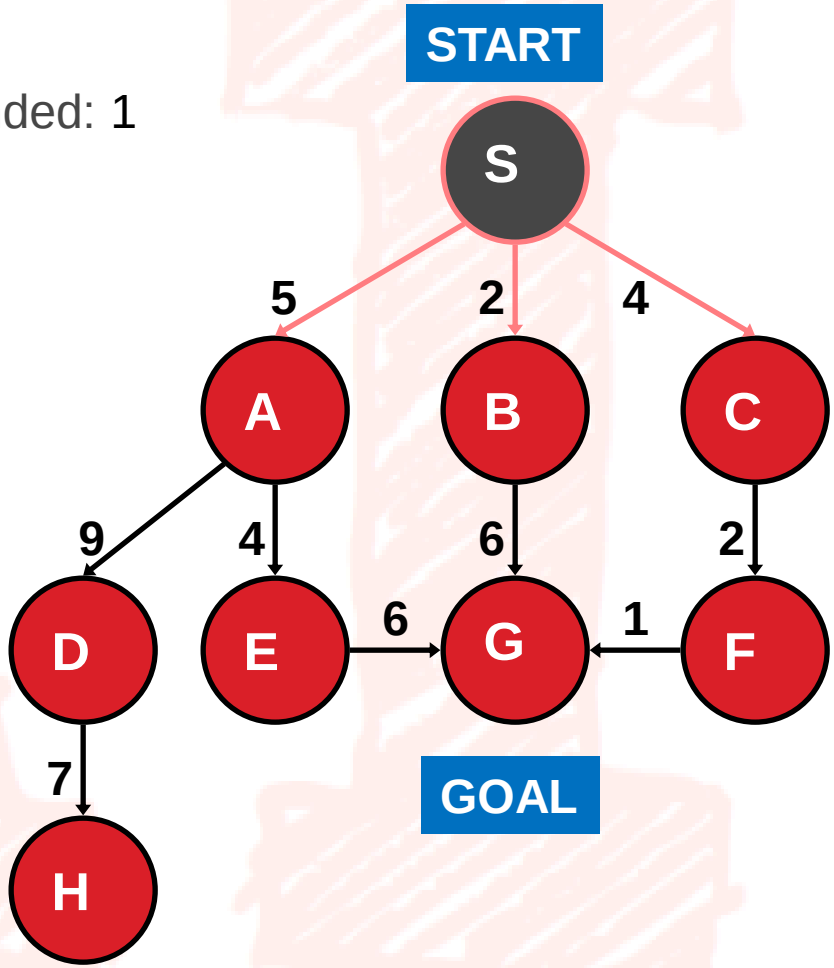


Iterative-Deepening Search (IDS)

`deepeningSearch(problem)`

depth: 1, # of nodes tested: 1(1), expanded: 1

current	nodes list
	{S}
S	{ }
S no test	{A,B,C}

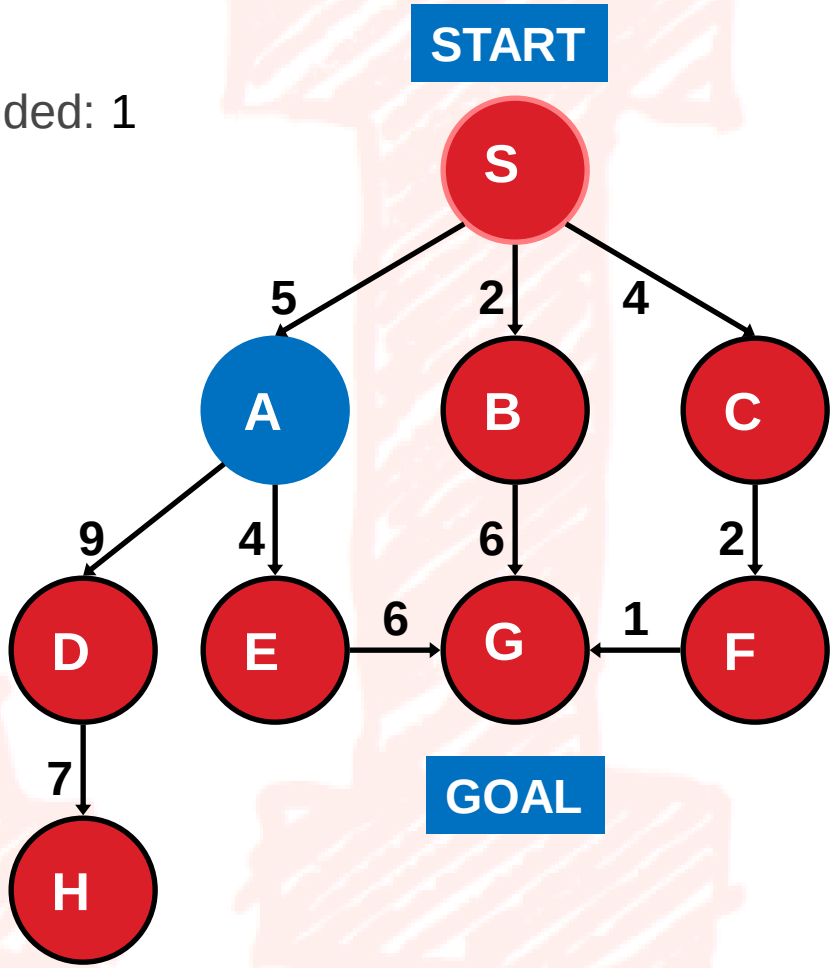


Iterative-Deepening Search (IDS)

`deepeningSearch(problem)`

depth: 1, # of nodes tested: 2(1), expanded: 1

current	nodes list
	{S}
S	{ }
S	{A,B,C}
A not goal	{B,C} @cutoff

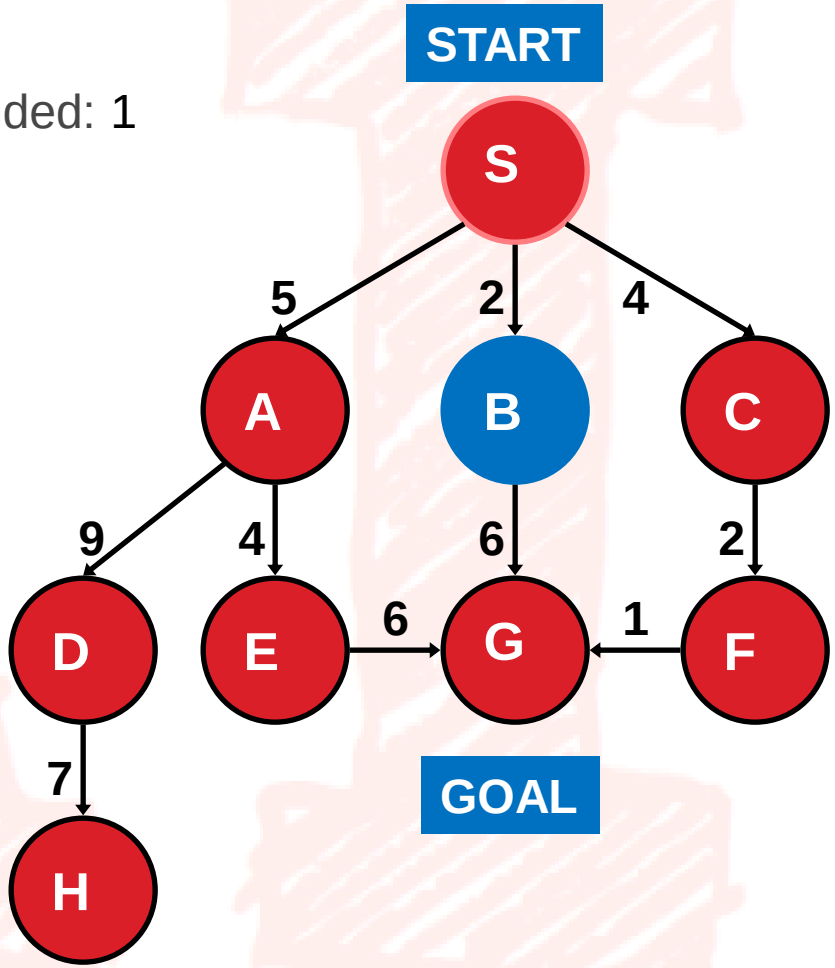


Iterative-Deepening Search (IDS)

`deepeningSearch(problem)`

depth: 1, # of nodes tested: 3(1), expanded: 1

current	nodes list
	{S}
S	{ }
S	{A,B,C}
A	{B,C}
B not goal	{C} @cutoff

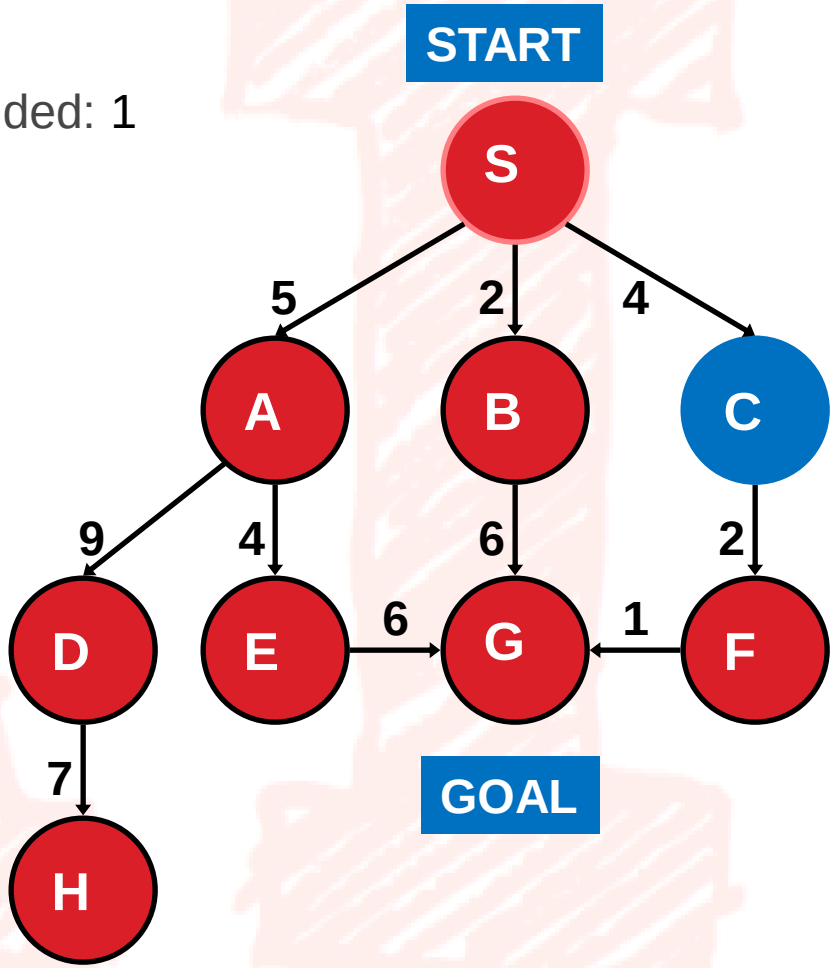


Iterative-Deepening Search (IDS)

`deepeningSearch(problem)`

depth: 1, # of nodes tested: 4(1), expanded: 1

current	nodes list
	{S}
S	{ }
S	{A,B,C}
A	{B,C}
B	{C}
C not goal	{ } @cutoff-FAIL

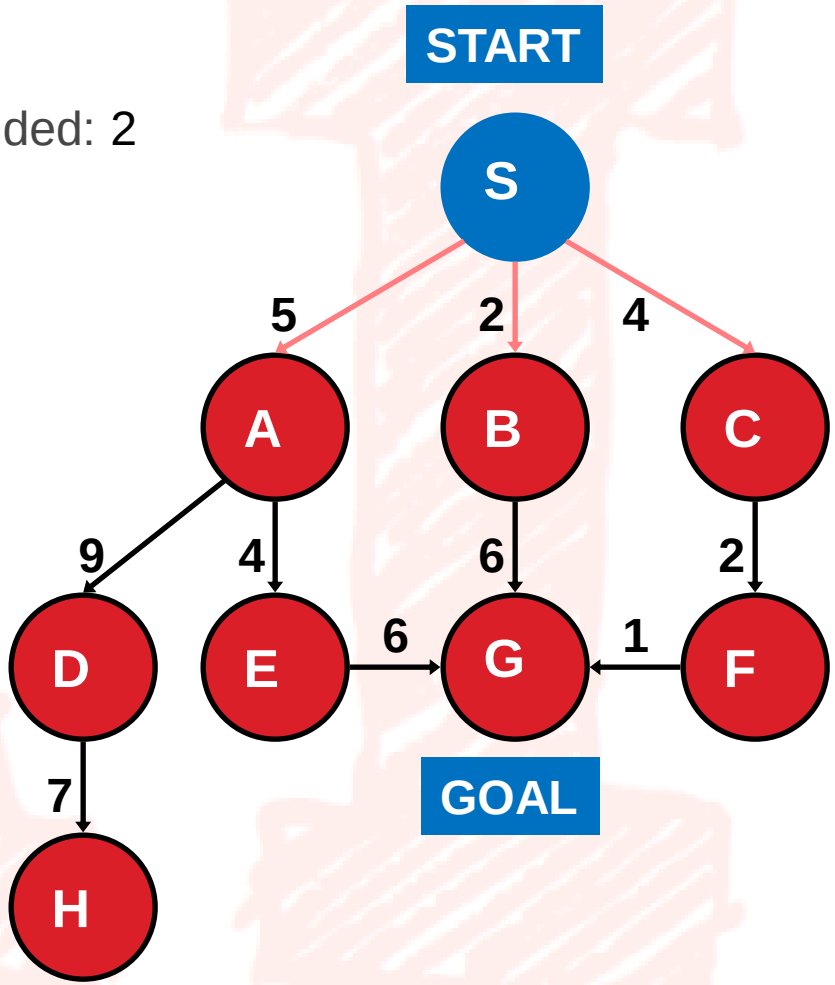


Iterative-Deepening Search (IDS)

`deepeningSearch(problem)`

depth: 2, # of nodes tested: 4(2), expanded: 2

current	nodes list
	{S}
S	{ }
S	{A,B,C}
A	{B,C}
B	{C}
C	{ }
S no test	{A,B,C}

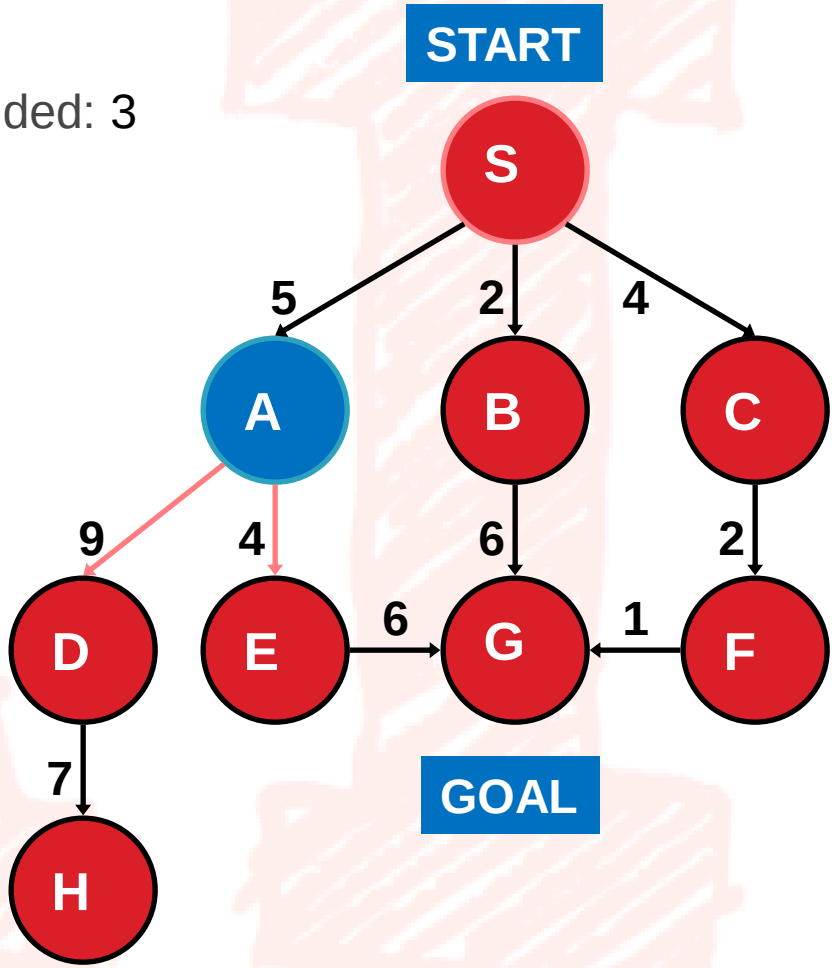


Iterative-Deepening Search (IDS)

`deepeningSearch(problem)`

depth: 2, # of nodes tested: 4(3), expanded: 3

current	nodes list
	{S}
S	{ }
S	{A,B,C}
A	{B,C}
B	{C}
C	{ }
S	{A,B,C}
A no test	{D,E,B,C}

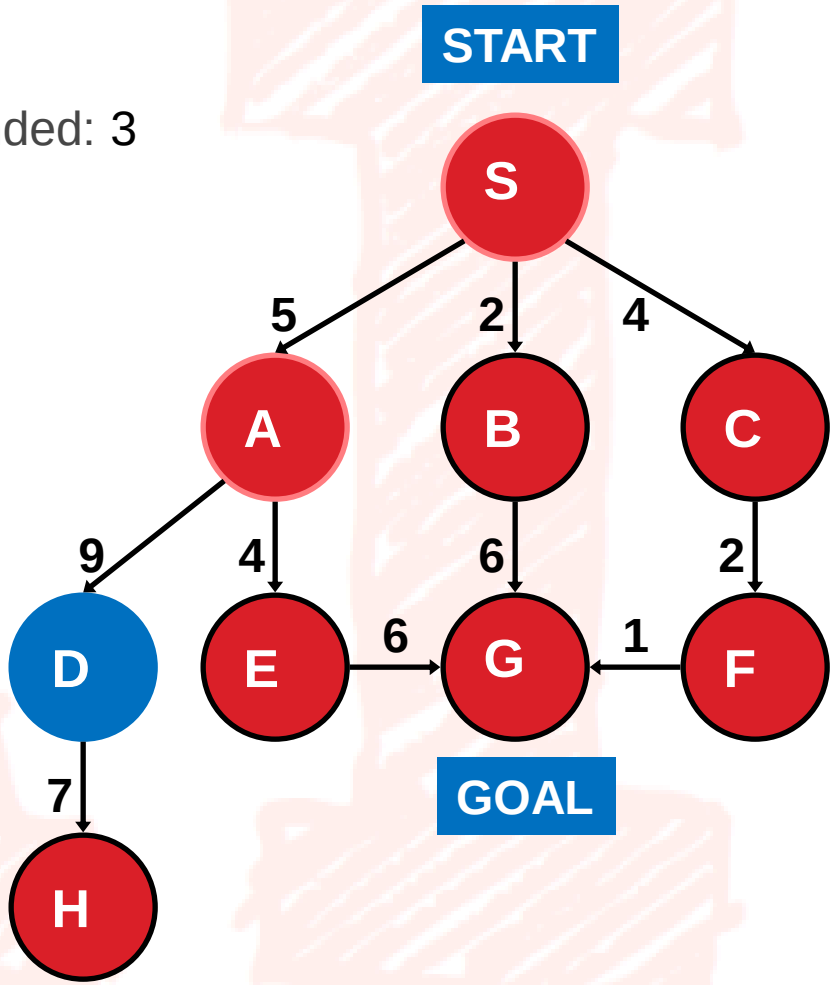


Iterative-Deepening Search (IDS)

`deepeningSearch(problem)`

depth: 2, # of nodes tested: 5(3), expanded: 3

current	nodes list
	{S}
S	{ }
S	{A,B,C}
A	{B,C}
B	{C}
C	{ }
S	{A,B,C}
A	{D,E,B,C}
D not goal	{E,B,C} @cutoff

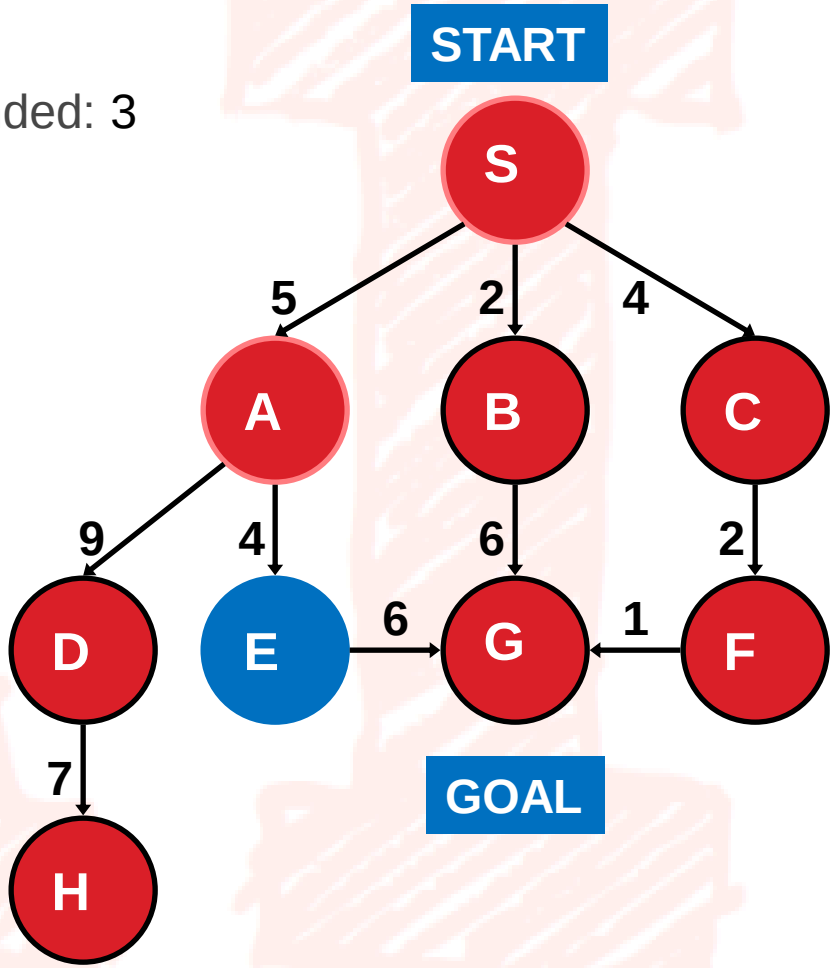


Iterative-Deepening Search (IDS)

`deepeningSearch(problem)`

depth: 2, # of nodes tested: 6(3), expanded: 3

current	nodes list
	{S}
S	{ }
S	{A,B,C}
A	{B,C}
B	{C}
C	{ }
S	{A,B,C}
A	{D,E,B,C}
D	{E,B,C}
E not goal	{B,C} @cutoff

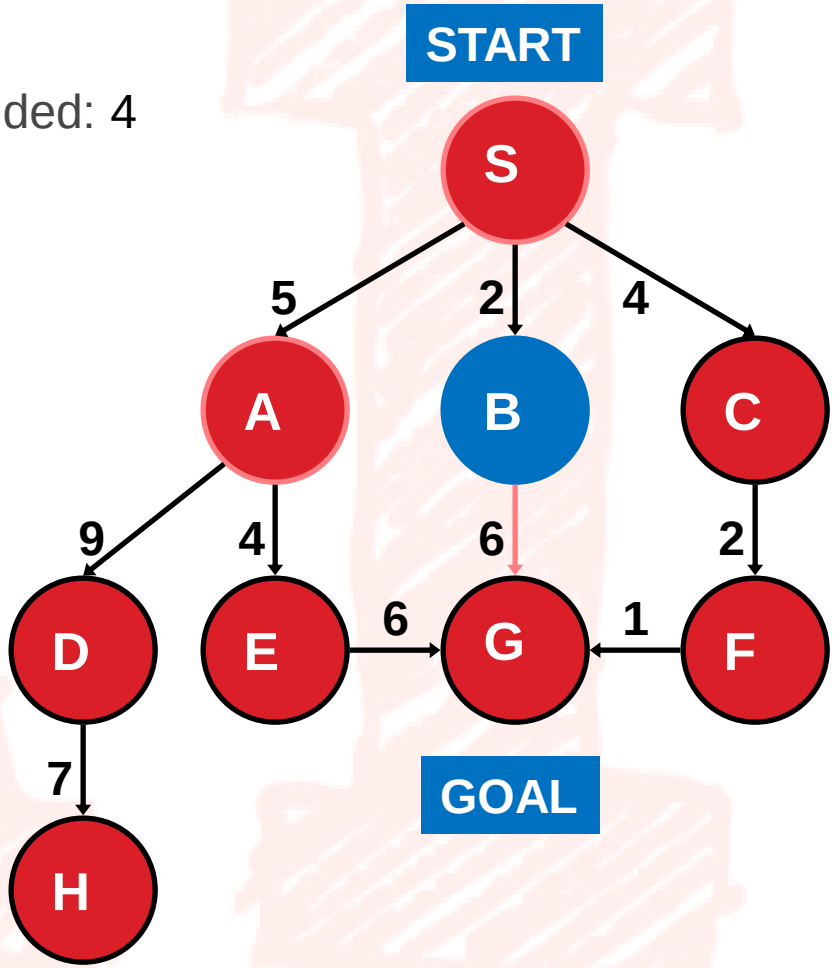


Iterative-Deepening Search (IDS)

`deepeningSearch(problem)`

depth: 2, # of nodes tested: 6(4), expanded: 4

current	nodes list
	{S}
S	{ }
S	{A,B,C}
A	{B,C}
B	{C}
C	{ }
S	{A,B,C}
A	{D,E,B,C}
D	{E,B,C}
E	{B,C}
B no test	{G,C}

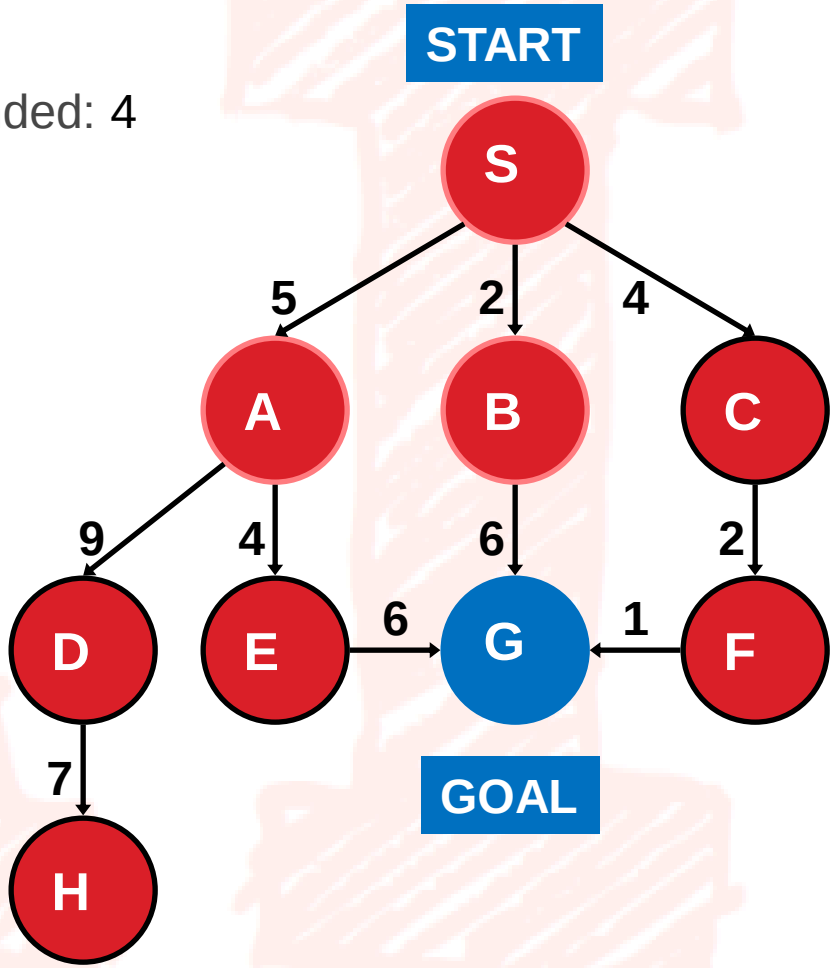


Iterative-Deepening Search (IDS)

`deepeningSearch(problem)`

depth: 2, # of nodes tested: 7(4), expanded: 4

current	nodes list
	{S}
S	{ }
S	{A,B,C}
A	{B,C}
B	{C}
C	{ }
S	{A,B,C}
A	{D,E,B,C}
D	{E,B,C}
E	{B,C}
B	{G,C}
G goal	{C} no expand

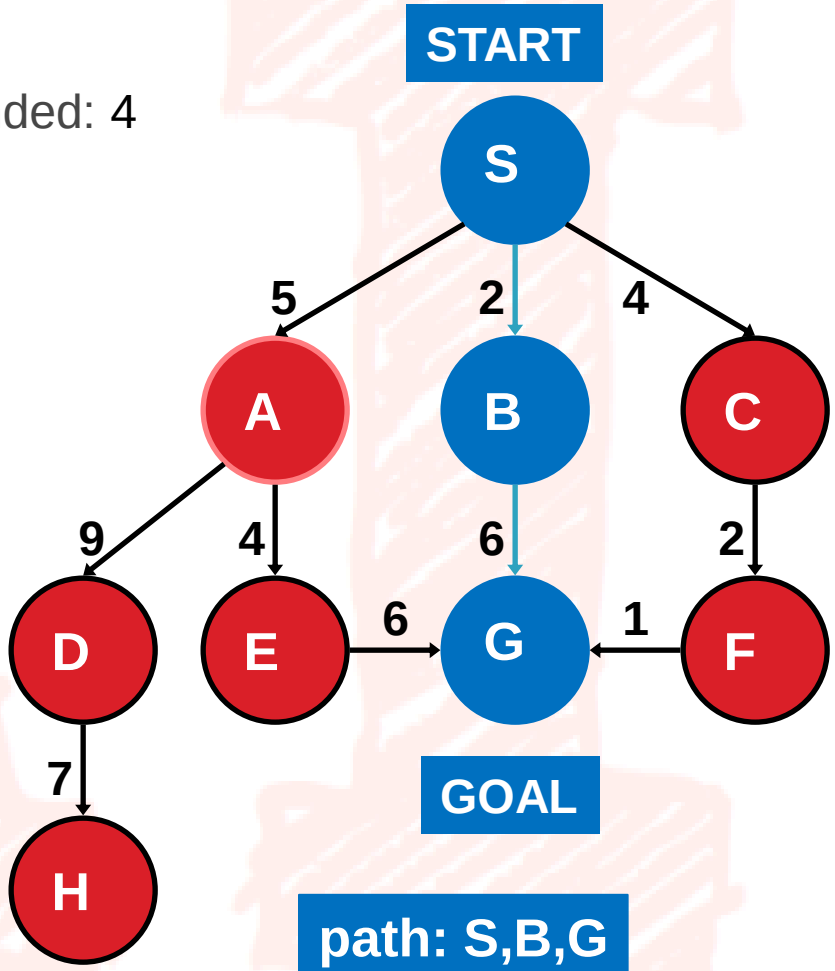


Iterative-Deepening Search (IDS)

`deepeningSearch(problem)`

depth: 2, # of nodes tested: 7(4), expanded: 4

current	nodes list
	{S}
S	{ }
S	{A,B,C}
A	{B,C}
B	{C}
C	{ }
S	{A,B,C}
A	{D,E,B,C}
D	{E,B,C}
E	{B,C}
B	{G,C}
G	{C}



Iterative-Deepening Search (IDS)

- Has advantages of BFS
  - completeness
  - optimality as stated for BFS
- Has advantages of DFS
  - limited space
  - in practice, even with redundant effort it still finds longer paths faster than BFS!
- In general, IDS is the preferred uniformed search strategy for large search spaces when the solution's depth is unknown.

Iterative-Deepening Search (IDS)

- For depth limit $\ell = 0; 1; \ldots$
- Perform a depth-first search with maximum depth ℓ.

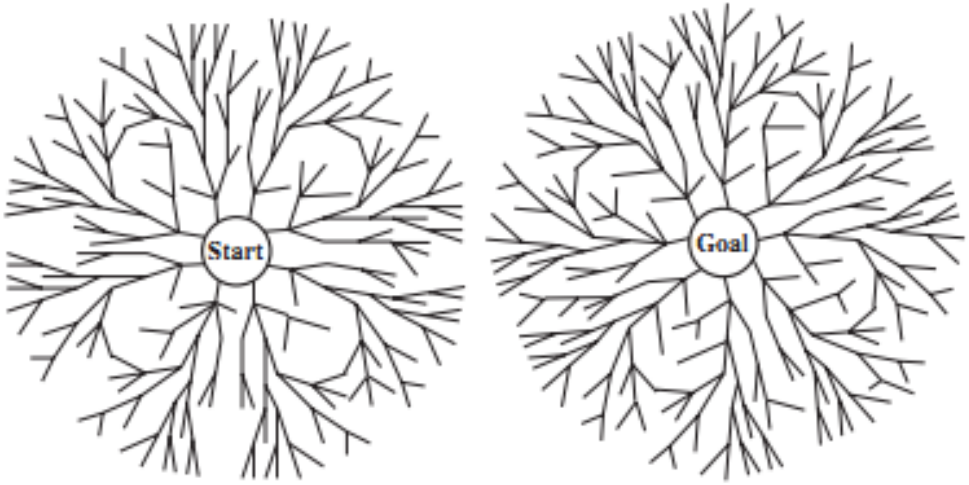
- Complete? Yes if the branching factor (b) is finite.

- Optimal? Yes if all arc costs are the same.

- Time complexity: $O(b^d)$

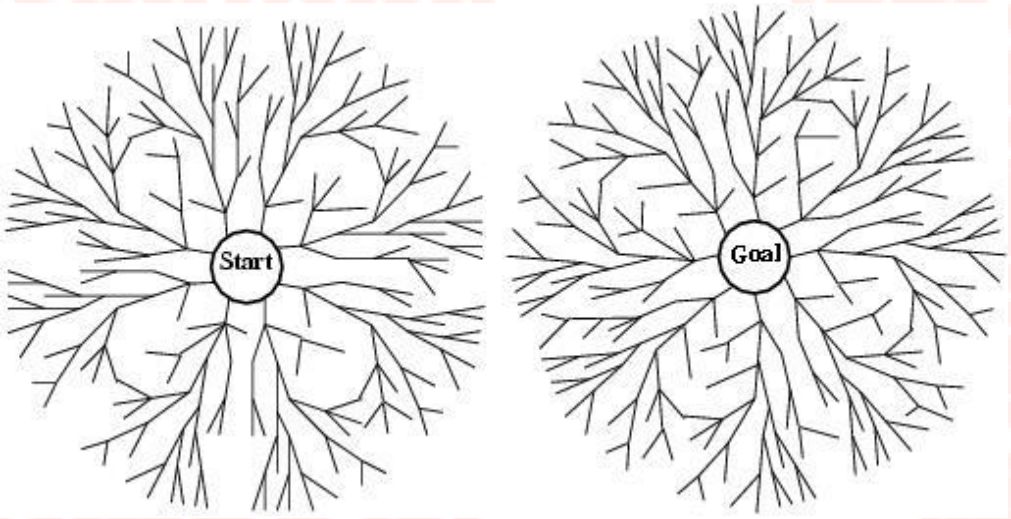
- Space complexity: O(bd)

Bidirectional search



- Two simultaneous searches from start an goal.
  - Motivation: $b^{d/2} + b^{d/2} \neq b^d$
- Check whether **the node belongs to the other frontier** before expansion.
- Space complexity is the most significant weakness.
- Complete and optimal if both searches are Breadth First.

How to search backwards?



- The predecessor of each node should be efficiently computable.
  - When actions are easily reversible.

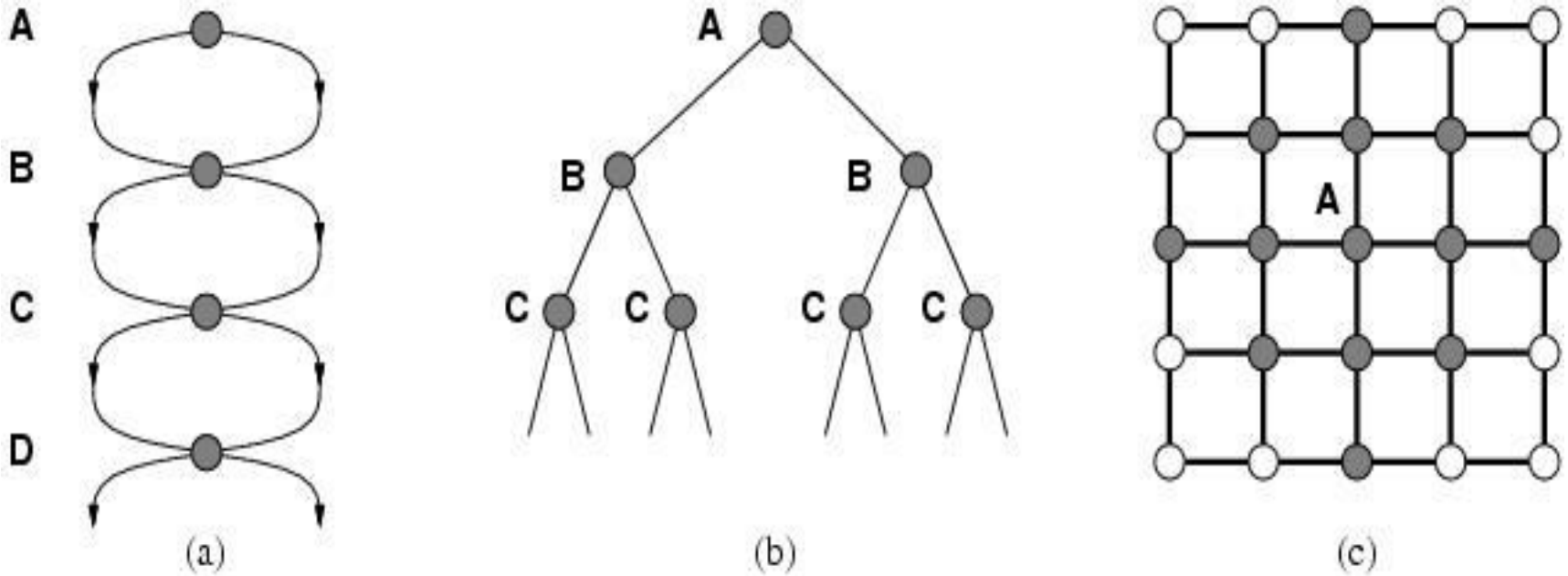
Bidirectional Search (cont.)

- **Advantages**:
  - Bidirectional search is fast.
  - Bidirectional search requires less memory

- **Disadvantages**:
  - Implementation of the bidirectional search tree is difficult.
  - In bidirectional search, one should know the goal state in advance.

Repeated states

- Failure to detect repeated states can turn a solvable problems into unsolvable ones.



Summary of algorithms

- Properties of algorithms

Criterion	Breadth-First	Uniform-cost	Depth-First	Depth-limited	Iterative deepening	Bidirectional search
Complete?	YES*	YES*	NO	YES, if $l \geq d$	YES	YES*
Time	b^d	$b^{C^*/e}$	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	$b^{C^*/e}$	bm	bl	bd	$b^{d/2}$
Optimal?	YES*	YES*	NO	NO	YES	YES

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

FACULTY OF INFORMATION TECHNOLOGY