
Chapter 3

Solving problems by searching

CS 461 – Artificial Intelligence

Pinar Duygulu

Bilkent University, Spring 2008

Slides are mostly adapted from AIMA and MIT Open Courseware

Introduction

- Simple-reflex agents directly maps states to actions.
- Therefore, they cannot operate well in environments where the mapping is too large to store or takes too much to learn
- Goal-based agents can succeed by considering future actions and desirability of their outcomes
- Problem solving agent is a goal-based agent that decides what to do by finding sequences of actions that lead to desirable states

Outline

- Problem-solving agents
- Problem types
- Problem formulation
- Example problems
- Basic search algorithms

Problem solving agents

- Intelligent agents are supposed to maximize their performance measure
- This can be simplified if the agent can adopt a **goal** and aim at satisfying it
- Goals help organize behaviour by limiting the objectives that the agent is trying to achieve
- **Goal formulation**, based on the current situation and the agent's performance measure, is the first step in problem solving
- Goal is a set of states. The agent's task is to find out which sequence of actions will get it to a goal state
- **Problem formulation** is the process of deciding what sorts of actions and states to consider, given a goal

Problem solving agents

- An agent with several immediate options of unknown value can decide what to do by first examining different possible sequences of actions that lead to states of known value, and then choosing the best sequence
- Looking for such a sequence is called **search**
- A search algorithm takes a problem as input and returns a **solution** in the form of action sequence
- Once a solution is found the actions it recommends can be carried out – **execution** phase

Problem solving agents

- “**formulate, search, execute**” design for the agent
- After formulating a goal and a problem to solve the agent calls a search procedure to solve it
- It then uses the solution to guide its actions, doing whatever the solution recommends as the next thing to do (typically the first action in the sequence)
- Then removing that step from the sequence
- Once the solution has been executed, the agent will formulate a new goal

Problem-solving agents

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  static: 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 do
    goal ← FORMULATE-GOAL(state)
    problem ← FORMULATE-PROBLEM(state, goal)
    seq ← SEARCH(problem)
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

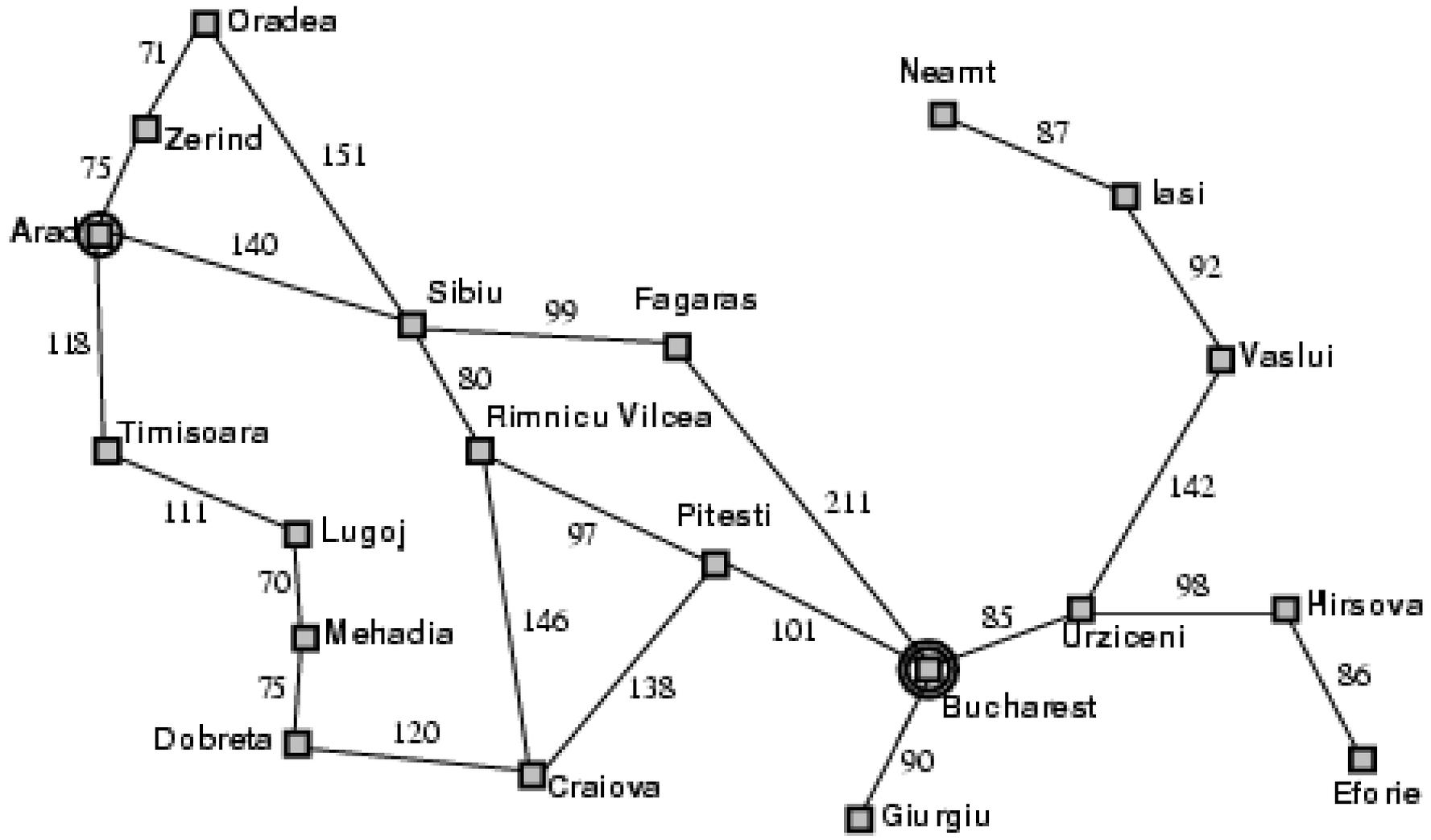
Environment Assumptions

- Static, formulating and solving the problem is done without paying attention to any changes that might be occurring in the environment
- Initial state is known and the environment is observable
- Discrete, enumerate alternative courses of actions
- Deterministic, solutions to problems are single sequences of actions, so they cannot handle any unexpected events, and solutions are executed without paying attention to the percepts

Example: Romania

- On holiday in Romania; currently in Arad.
- Flight leaves tomorrow from Bucharest
- **Formulate goal:**
 - be in Bucharest
- **Formulate problem:**
 - **states:** various cities
 - **actions:** drive between cities
- **Find solution:**
 - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Example: Romania



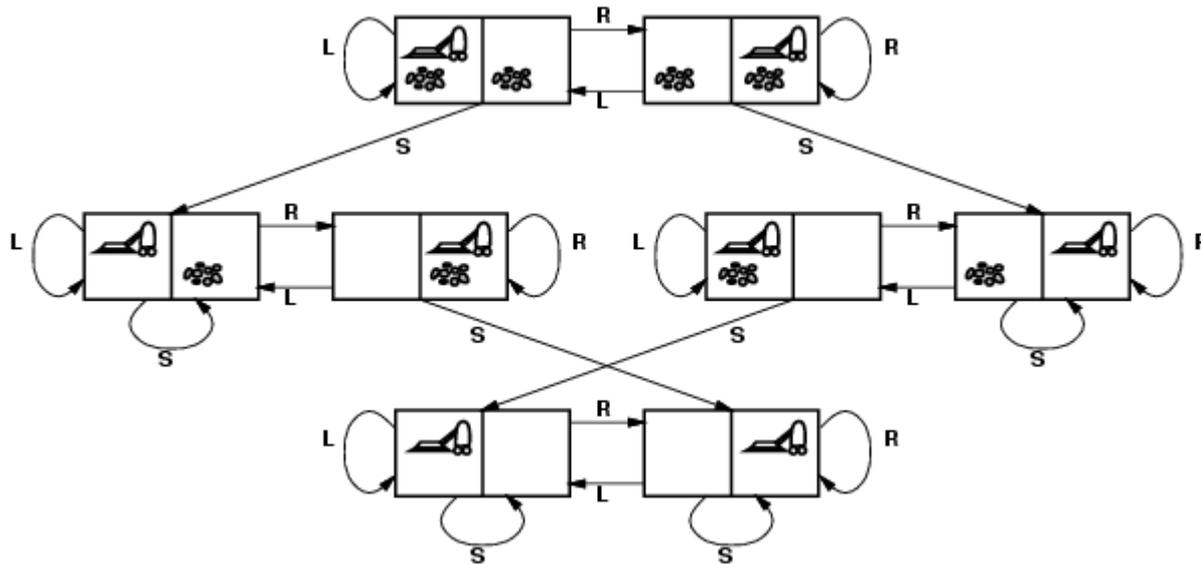
Well-defined problems and solutions

- A problem can be defined formally by four components
 - Initial state that the agent starts in
 - e.g. In(Arad)
 - A description of the possible actions available to the agent
 - Successor function – returns a set of <action,successor> pairs
 - e.g. {<Go(Sibiu),In(Sibiu)>, <Go(Timisoara),In(Timisoara)>, <Go(Zerind), In(Zerind)>}
 - Initial state and the successor function define the state space (a graph in which the nodes are states and the arcs between nodes are actions). A path in state space is a sequence of states connected by a sequence of actions
 - Goal test determines whether a given state is a goal state
 - e.g. {In(Bucharest)}
 - Path cost function that assigns a numeric cost to each path. The cost of a path can be described as the some of the costs of the individual actions along the path – step cost
 - e.g. Time to go Bucharest
-

Problem Formulation

- A solution to a problem is a path from the initial state to the goal state
 - Solution quality is measured by the path cost function and an optimal solution has the lowest path cost among all solutions
 - 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 \rightarrow Zerind" represents a complex set of possible routes, detours, rest stops, etc.
 - For guaranteed realizability, **any** real state "in Arad" must get to **some** real state "in Zerind"
 - (Abstract) solution =
 - set of real paths that are solutions in the real world
 - Each abstract action should be "easier" than the original **problem**
-

Vacuum world state space graph



- states integer dirt and robot location.
 - The agent is in one of two locations, each of which might or might not contain dirt – 8 possible states
- Initial state: any state
- actions *Left, Right, Suck*
- goal test no dirt at all locations
- path cost 1 per action

Example: The 8-puzzle

7	2	4
5		6
8	3	1

Start State

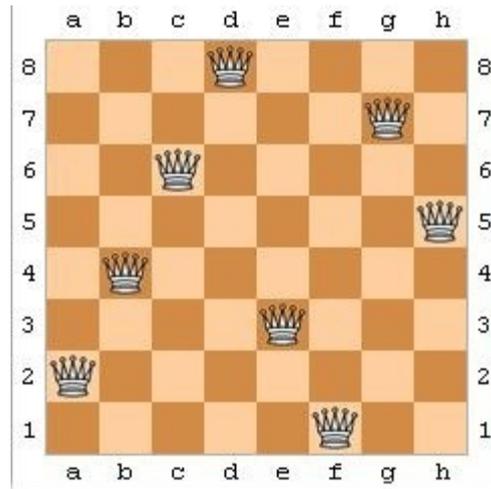
	1	2
3	4	5
6	7	8

Goal State

- states: locations of tiles
- Initial state: any state
- actions: move blank left, right, up, down
- goal test: goal state (given)
- path cost: 1 per move

[Note: optimal solution of n -Puzzle family is NP-hard]

Example: 8-queens problem



- states: any arrangement of 0-8 queens on the board is a state
- Initial state: no queens on the board
- actions: add a queen to any empty square
- goal test: 8 queens are on the board, none attacked

$64 \cdot 63 \cdot \dots \cdot 57 = 1.8 \times 10^{14}$ possible sequences

Example: Route finding problem

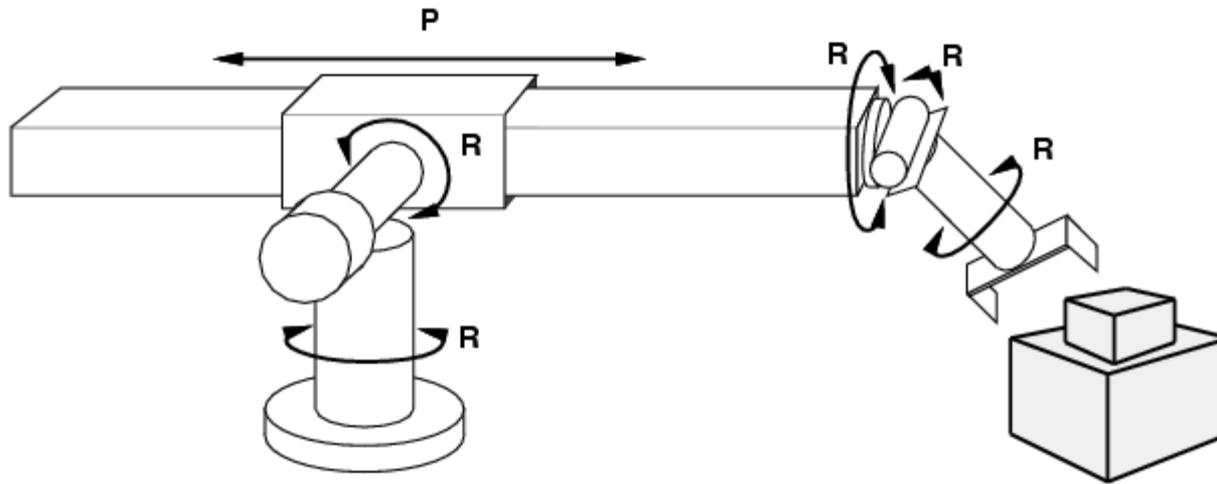
- states: each is represented by a location (e.g. An airport) and the current time
- Initial state: specified by the problem
- Successor function: returns the states resulting from taking any scheduled flight, leaving later than the current time plus the within airport transit time, from the current airport to another
- goal test: are we at the destination by some pre-specified time
- Path cost: monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, etc

- Route finding algorithms are used in a variety of applications, such as routing in computer networks, military operations planning, airline travel planning systems

Other example problems

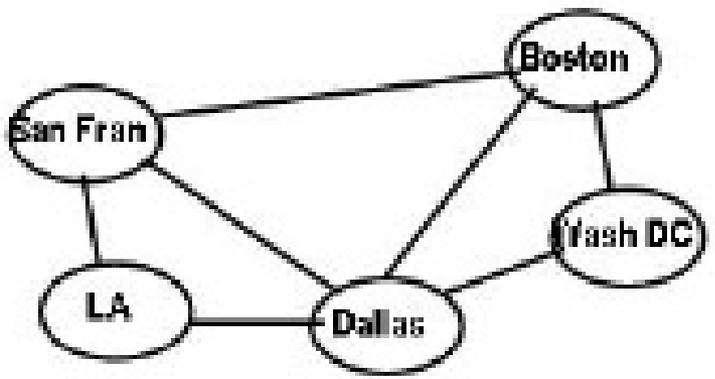
- Touring problems: visit every city at least once, starting and ending at Bucharest
- Travelling salesperson problem (TSP) : each city must be visited exactly once – find the shortest tour
- VLSI layout design: positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield
- Robot navigation
- Internet searching
- Automatic assembly sequencing
- Protein design

Example: robotic assembly



- states: real-valued coordinates of robot joint angles parts of the object to be assembled
- actions: continuous motions of robot joints
- goal test: complete assembly
- path cost: time to execute

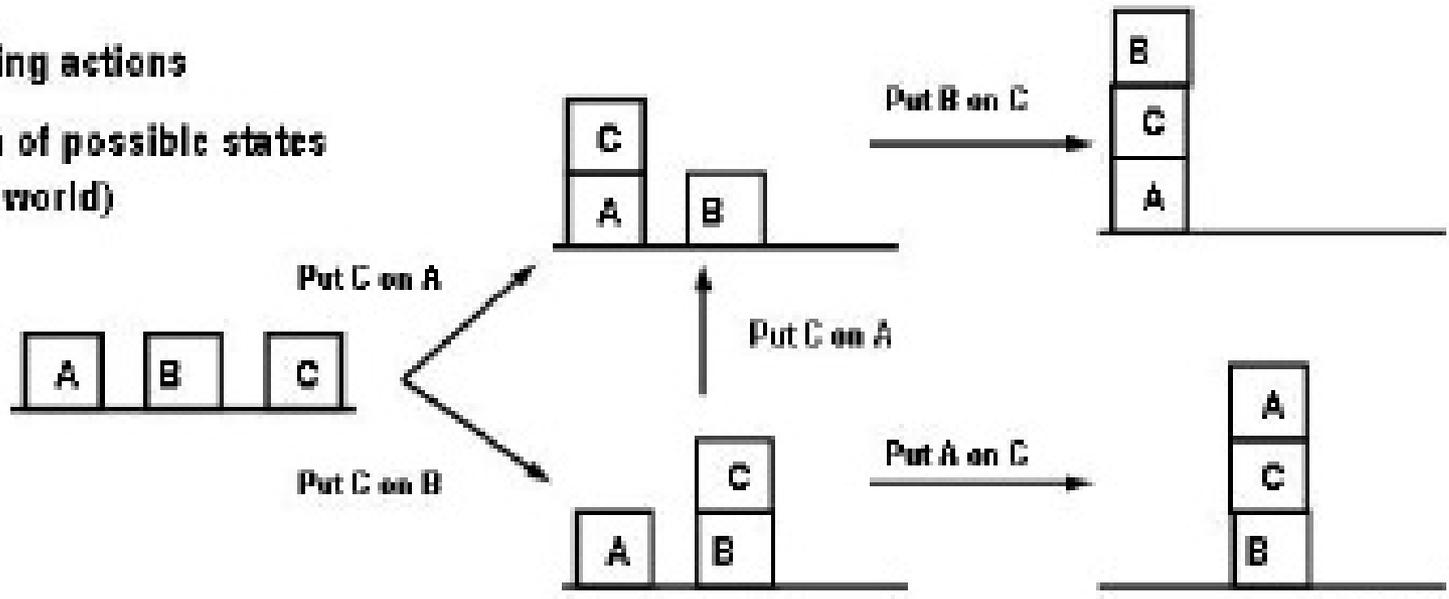
Graphs



Airline Routes

Planning actions

(graph of possible states of the world)



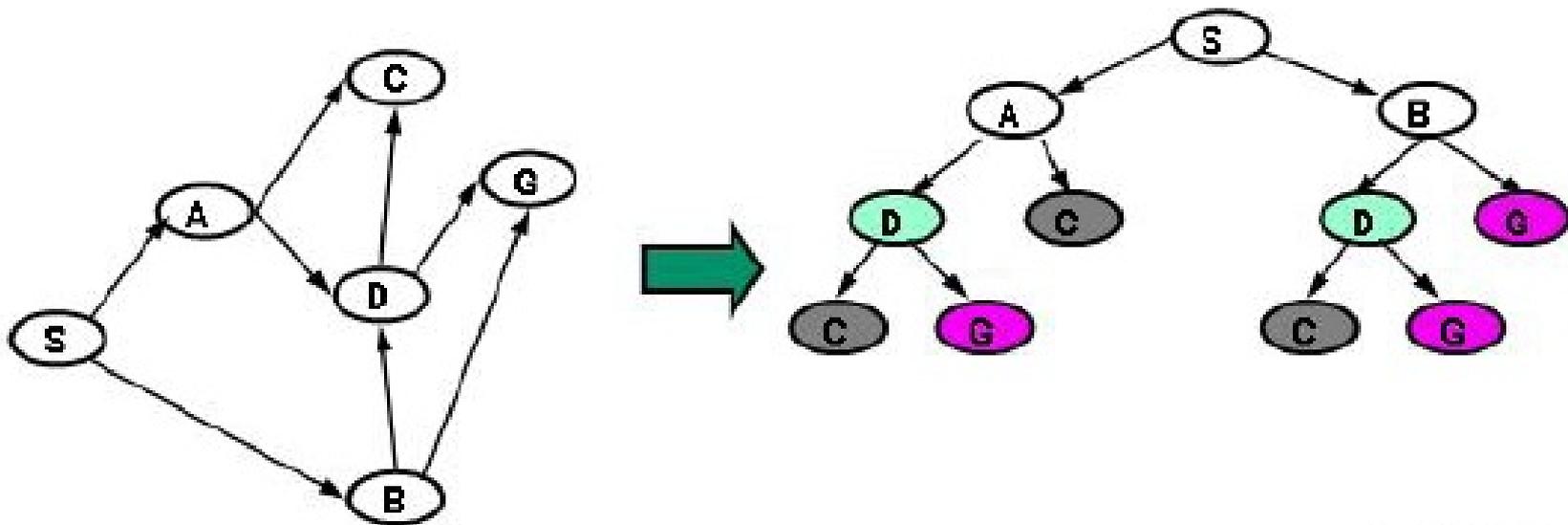
ty - Spring 08 - 4

Problem Solving

- What are the **states**? (All relevant aspects of the problem)
 - Arrangement of parts (to plan an assembly)
 - Positions of trucks (to plan package distribution)
 - City (to plan a trip)
 - Set of facts (e.g. to prove geometry theorem)
- What are the **actions** (operators)? (Deterministic and discrete)
 - Assemble two parts
 - Move a truck to a new position
 - Fly to a new city
 - Apply a theorem to derive new fact
- What is the **goal test**? (Conditions for success)
 - All parts in place
 - All packages delivered
 - Reached destination city
 - Derived goal fact

Graph Search as Tree Search

- Trees are directed graphs without cycles and with 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)



Terminology

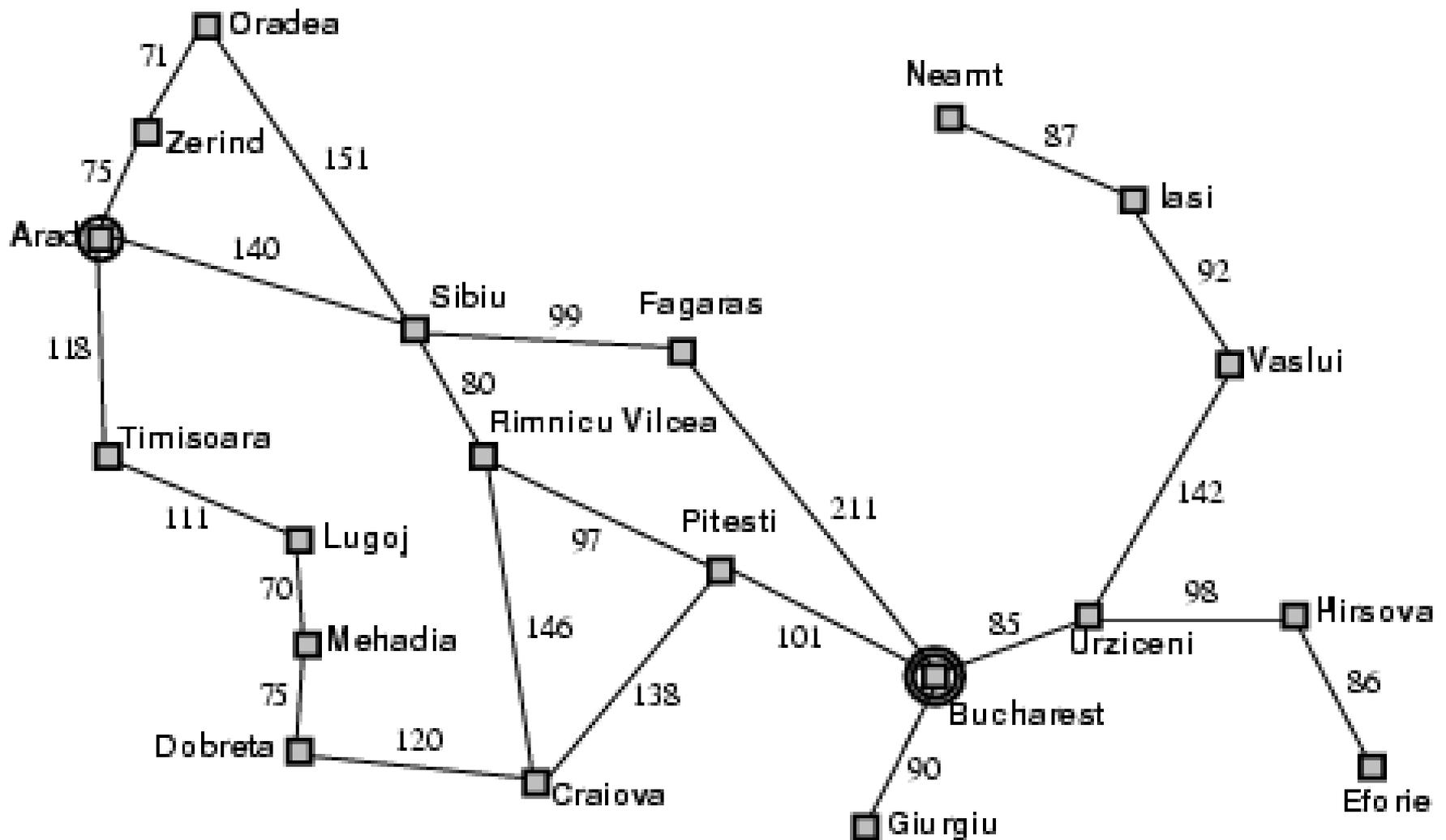
- **State** – Used to refer to the vertices of the underlying **graph** that is being searched, that is, states in the problem domain, for example, a city, an arrangement of blocks or the arrangement of parts in a puzzle.
- **Search Node** – Refers to the vertices of the search **tree** which is being generated by the search algorithm. Each node refers to a state of the world; **many nodes may refer to the same state**. Importantly, a node implicitly represents a path (from the start state of the search to the state associated with the node). Because search nodes are part of a search tree, they have a unique ancestor node (except for the root node).

Tree search algorithms

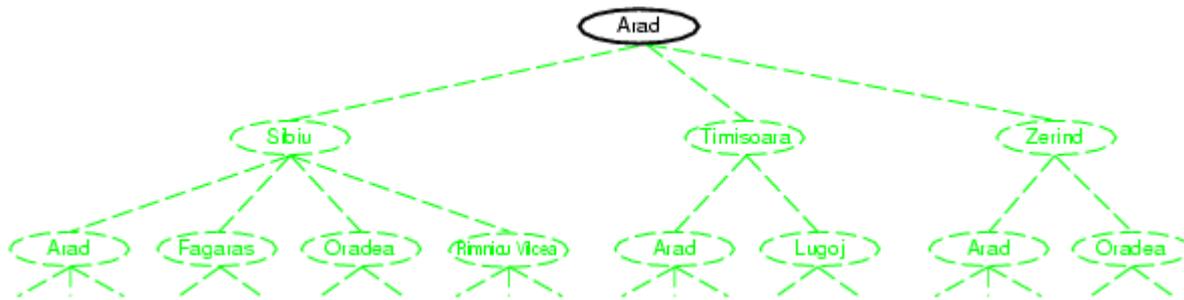
- Basic idea:
 - offline, simulated exploration of state space by generating successors of already-explored states (a.k.a. **expanding** states)

```
function TREE-SEARCH(problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree
```

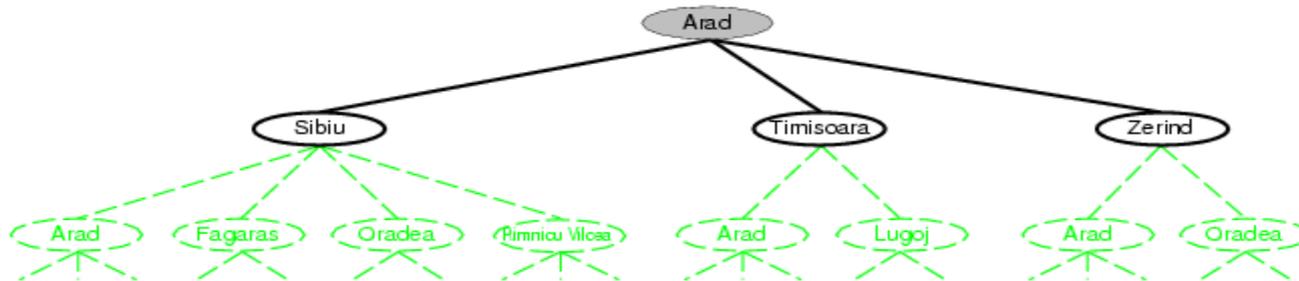
Example: Romania



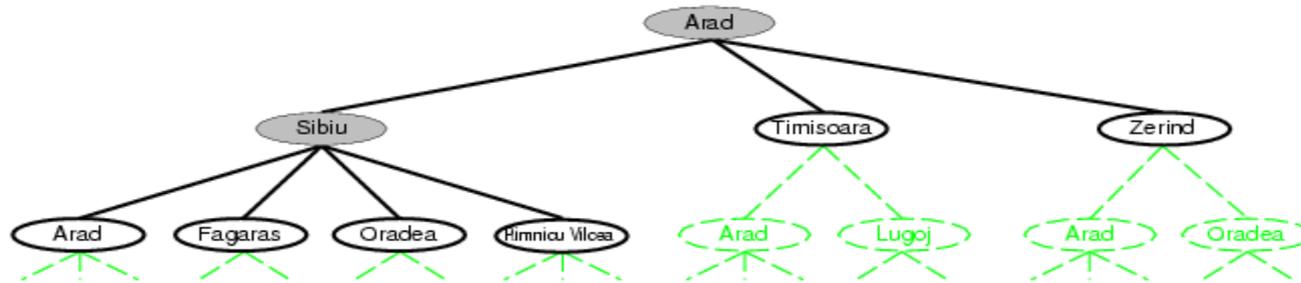
Tree search example



Tree search example



Tree search example

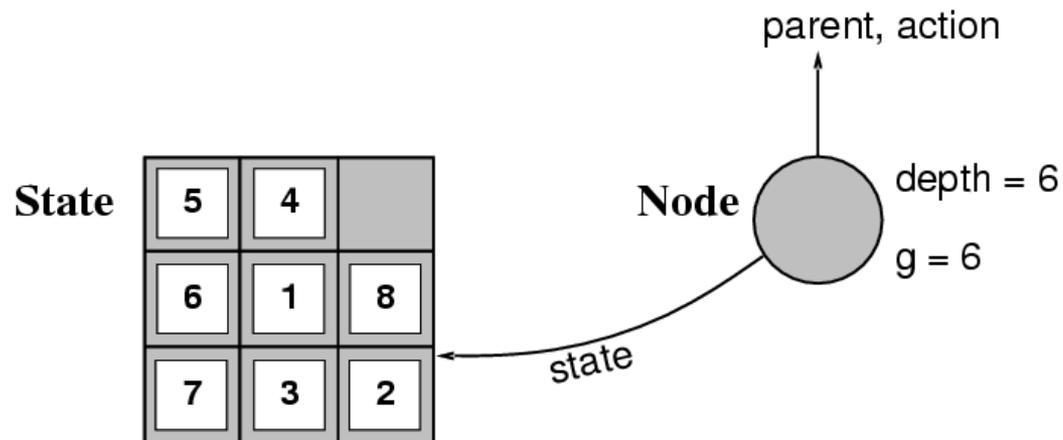


Implementation: Components of a node

- **State:** the state in the state space to which the node corresponds
- **Parent-node:** the node in the search tree that generated this node
- **Action:** the action that was applied to the parent to generate the node
- **Path-cost:** the cost, traditionally denoted by $g(n)$, of the path from the initial state to the node, as indicated by the parent pointers
- **Depth:** the number of steps along the path from the initial state

Implementation: states vs. nodes

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



Implementation: general tree search

- **Fringe:** the collection of nodes that have been generated but not yet been expanded
- Each element of a fringe is a leaf node, a node with no successors
- **Search strategy:** a function that selects the next node to be expanded from fringe
- We assume that the collection of nodes is implemented as a queue
- The operations on the queue are:
 - Make-queue(queue)
 - Empty?(queue)
 - first(queue)
 - remove-first(queue)
 - insert(element, queue)
 - insert-all(elements, queue)

Implementation: general tree search

```

function TREE-SEARCH(problem, fringe) returns a solution, or failure
  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)
    fringe ← INSERTALL(EXPAND(node, problem), fringe)

```

```

function EXPAND(node, problem) returns a set of nodes
  successors ← the empty set
  for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
    s ← a new NODE
    PARENT-NODE[s] ← node; ACTION[s] ← action; STATE[s] ← result
    PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
    DEPTH[s] ← DEPTH[node] + 1
    add s to successors
  return successors

```

- The Expand function creates new nodes, filling in the various fields and using the SuccessorFn of the problem to create the corresponding states.

Simple search algorithms - revisited

- A search node is a path from state X to the start state (e.g. X B A S)
- The state of a search node is the most recent state of the path (e.g. X)
- Let Q be a list of search nodes (e.g. (X B A S) (C B A S)) and S be the start state

- **Algorithm**

1. Initialize Q with search node (S) as only entry, set Visited = (S)
2. If Q is empty, fail. Else pick some search node N from Q
3. If state(N) is a goal, return N (we have reached the goal)
4. Otherwise remove N from Q
5. Find all the children of state(N) not in visited and create all the one-step extensions of N to each descendant
6. Add the extended paths to Q, add children of state(N) to Visited
7. Go to step 2

- **Critical decisions**

- Step 2: picking N from Q
- Step 6: adding extensions of N to Q

Examples: Simple search strategies

- Depth first search
 - Pick first element of Q
 - Add path extensions to front of Q
- Breadth first search
 - Pick first element of Q
 - Add path extensions to the end of Q

Visited versus expanded

- **Visited:** a state M is first visited when a path to M first gets added to Q . In general, a state is said to have been visited if it has ever shown up in a search node in Q . The intuition is that we have briefly visited them to place them on Q , but we have not yet examined them carefully
- **Expanded:** a state M is expanded when it is the state of a search node that is pulled off of Q . At that point, the descendants of M are visited and the path that led to M is extended to the eligible descendants. In principle, a state may be expanded multiple times. We sometimes refer to the search node that led to M as being expanded. However, once a node is expanded, we are done with it, we will not need to expand it again. In fact, we discard it from Q

Testing for the goal

- This algorithm stops (in step 3) when $\text{state}(N) = G$ or, in general when $\text{state}(N)$ satisfies the goal test
- We could have performed the test in step 6 as each extended path is added to Q . This would catch termination earlier
- However, performing the test in step 6 will be incorrect for the optimal searches

Keeping track of visited states

- Keeping track of visited states generally improves time efficiency when searching for graphs, without affecting correctness. Note, however, that substantial additional space may be required to keep track of visited states.
- If all we want to do is find a path from the start to goal, there is no advantage of adding a search node whose state is already the state of another search node
- Any state reachable from the node the second time would have been reachable from that node the first time
- Note that when using Visited, each state will only ever have at most one path to it (search node) in Q
- We'll have to revisit this issue when we look at optimal searching

Implementation issues : The visit List

- Although we speak of a visited list, this is never the preferred implementation
- If the graph states are known ahead of time as an explicit set, then space is allocated in the state itself to keep a mark, which makes both adding Visited and checking if a state is Visited a constant time operation
- Alternatively, as in more common AI, if the states are generated on the fly, then a hash table may be used for efficient detection of previously visited states.
- Note that, in any case, the incremental space cost of a Visited list will be proportional to the number of states, which can be very high in some problems

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
 - **space complexity**: maximum number of nodes in memory
 - **optimality**: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
 - b : maximum branching factor of the search tree
 - d : depth of the least-cost solution
 - m : maximum depth of the state space (may be ∞)

Classes of Search

Class	Name	Operation
Any Path Uninformed	Depth-First Breadth-First	Systematic exploration of whole tree until a goal node is found.
Any Path Informed	Best-First	Uses heuristic measure of goodness of a state, e.g. estimated distance to goal.
Optimal Uninformed	Uniform-Cost	Uses path "length" measure. Finds "shortest" path.
Optimal Informed	A*	Uses path "length" measure and heuristic Finds "shortest" path

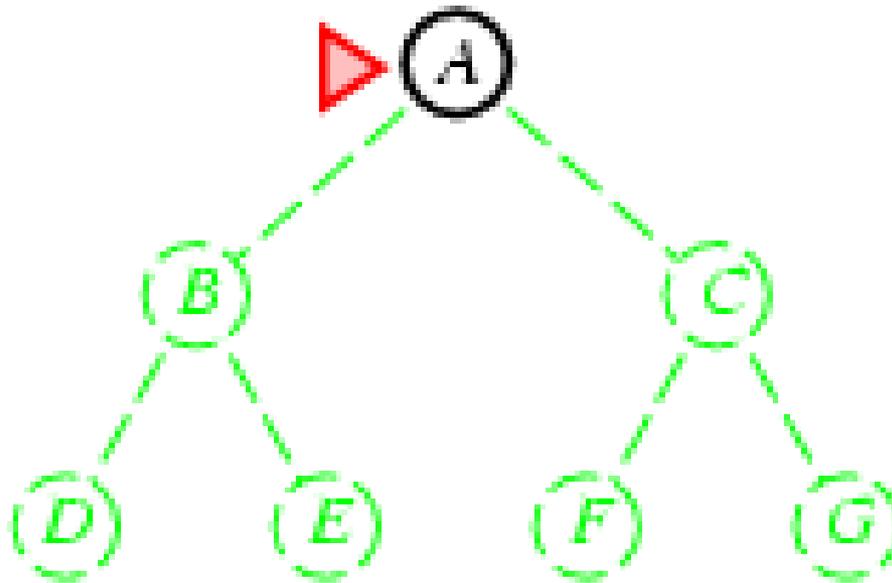
Uninformed search strategies

- **Uninformed** search (blind 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

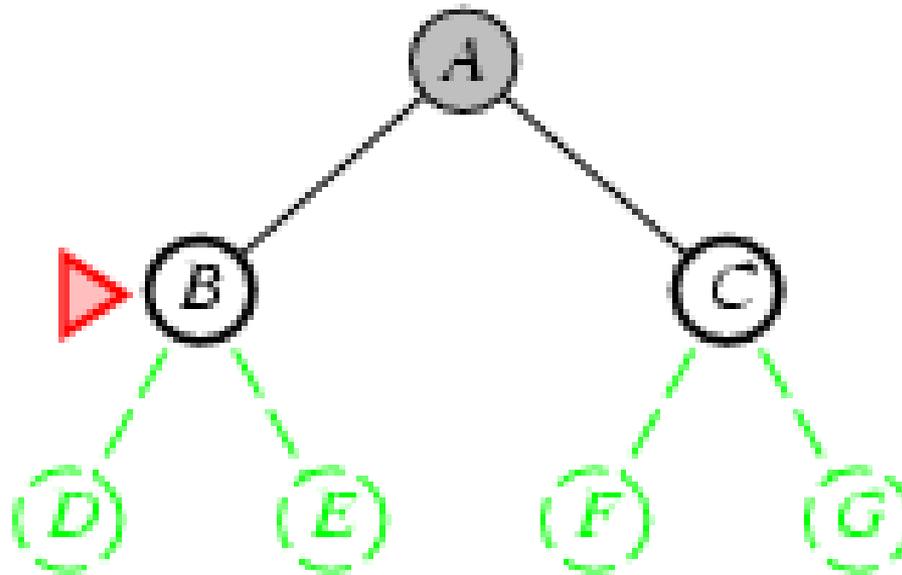
Breadth-first search

- The root node is expanded first, then all the successors of the root node, and 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 are expanded
- Expand shallowest unexpanded node
- **Implementation:**
 - *fringe* is a FIFO queue,
 - the nodes that are visited first will be expanded first
 - All newly generated successors will be put at the end of the queue
 - Shallow nodes are expanded before deeper nodes

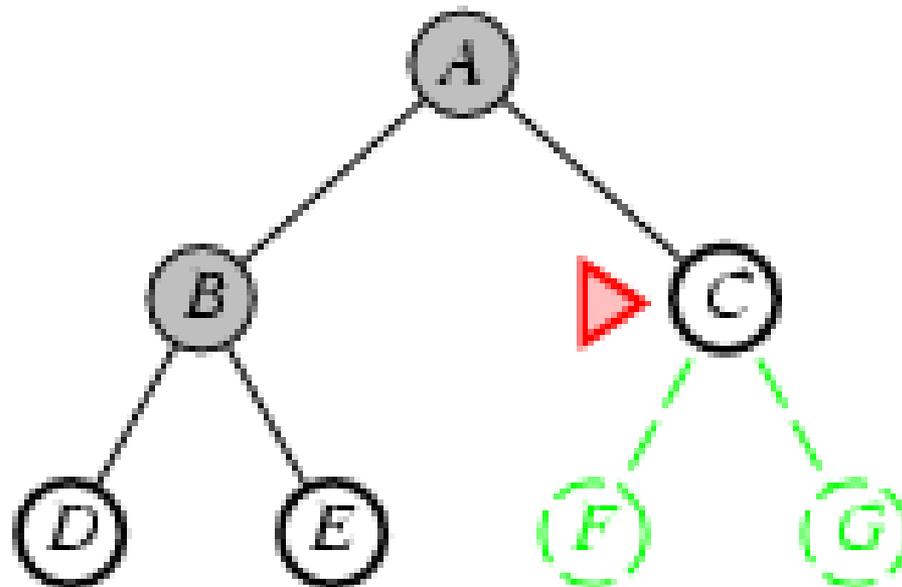
Breadth-first search



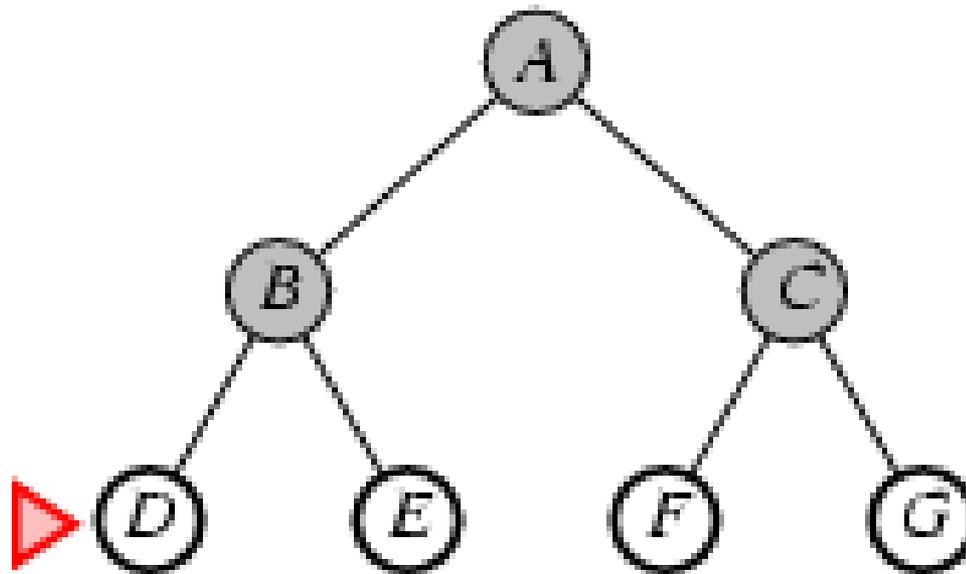
Breadth-first search



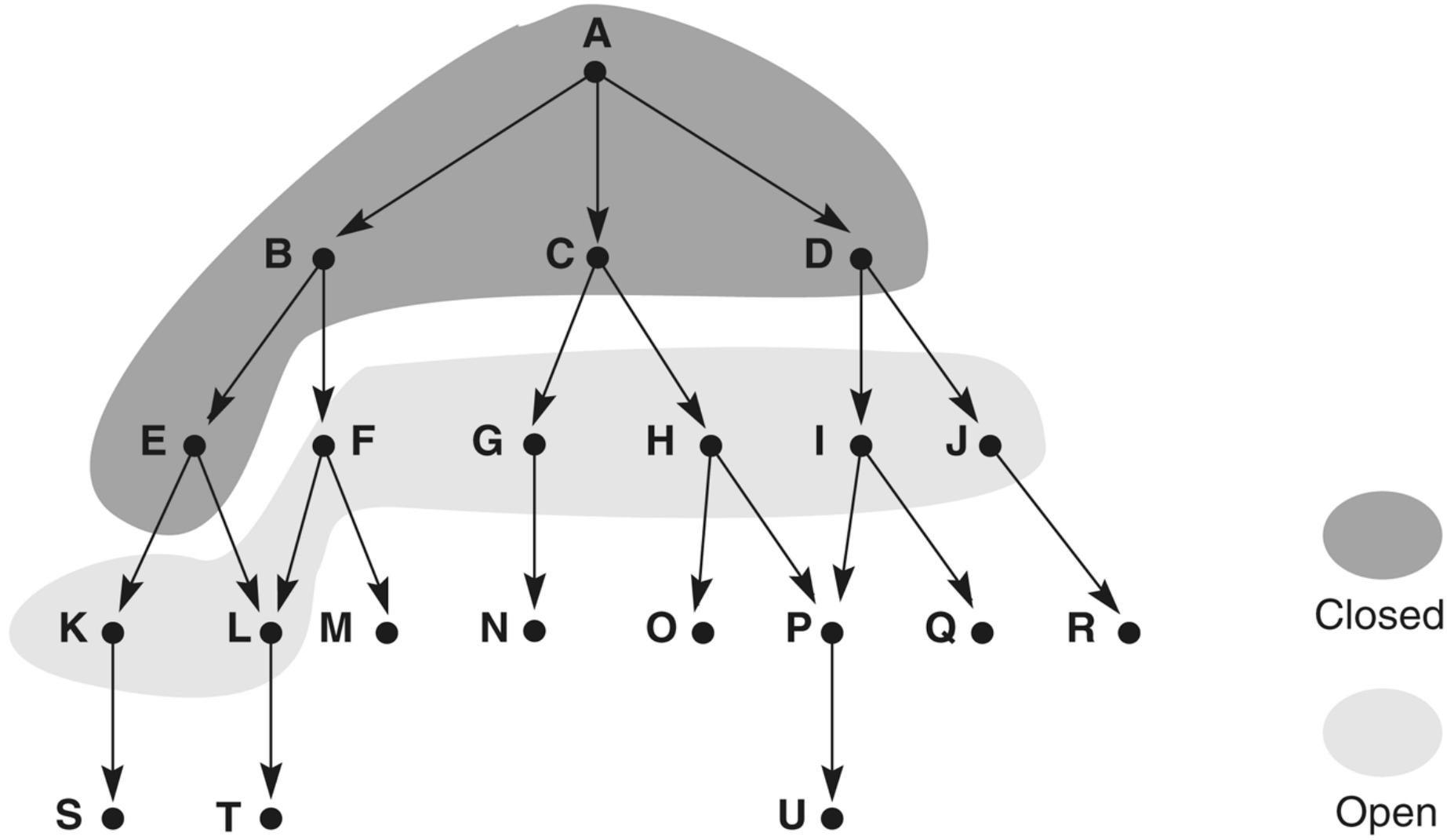
Breadth-first search



Breadth-first search



Breadth-first search



8-puzzle problem

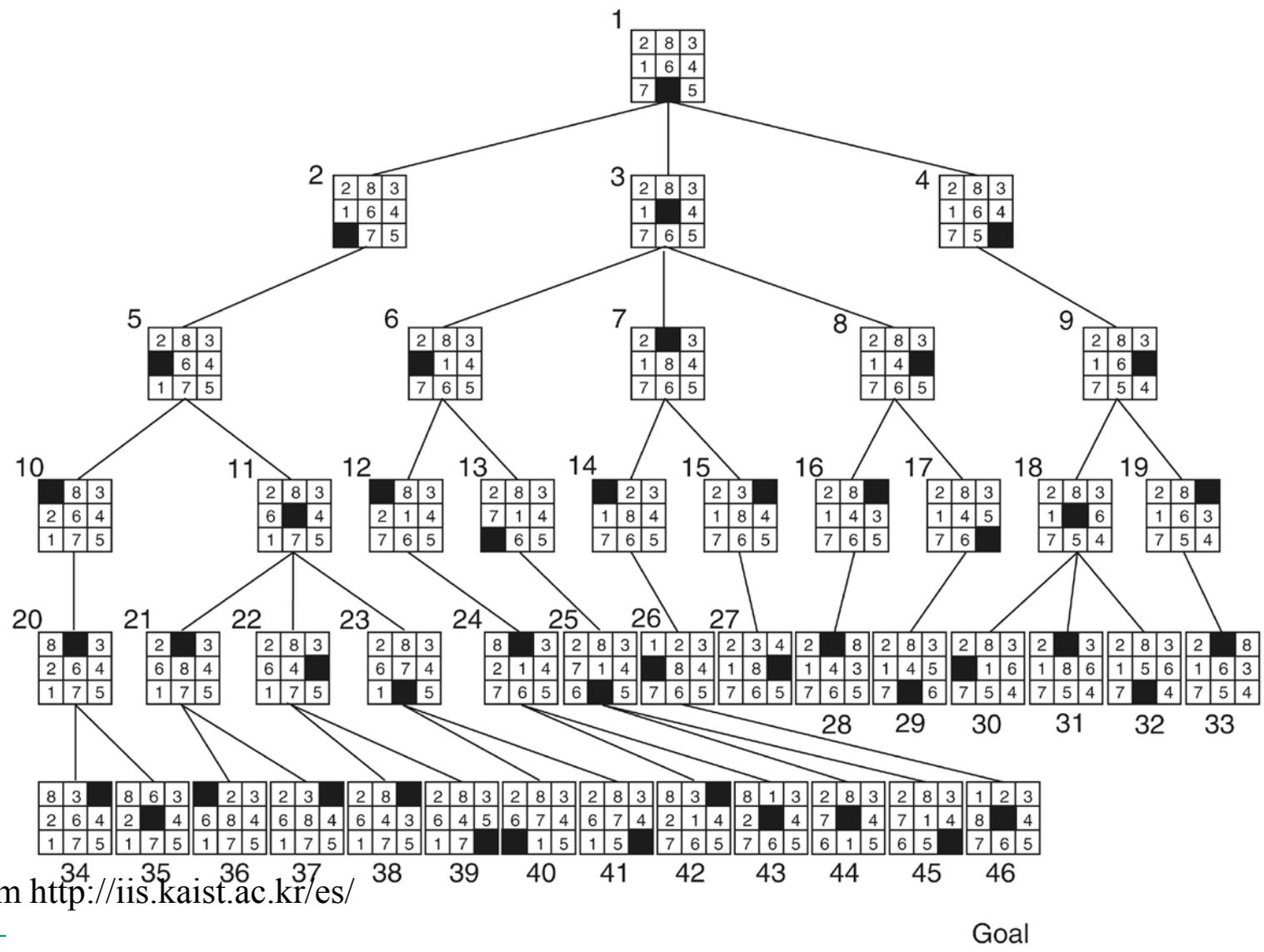
2	8	3
1	6	4
7		5



1	2	3
8		4
7	6	5

Breadth-first search

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

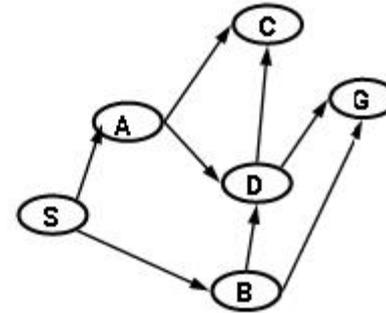


Taken from <http://iis.kaist.ac.kr/es/>

Goal

Breadth first search

	Q	Visited
1	(S)	S
2		
3		
4		
5		
6		



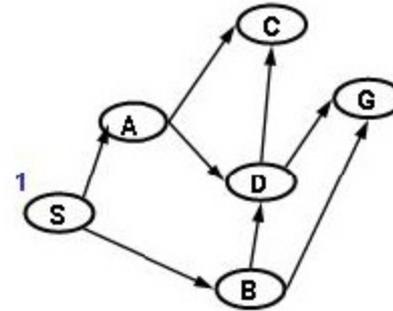
Pick first element of Q; Add path extensions to end of Q

Added paths in blue

We show the paths in **reversed order; the node's state is the first entry.**

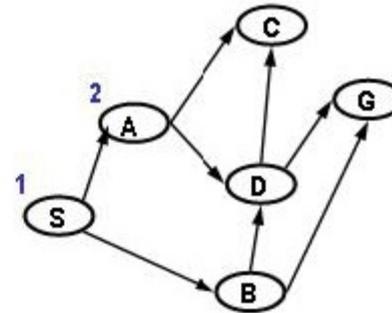
Breadth first search

	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3		
4		
5		
6		



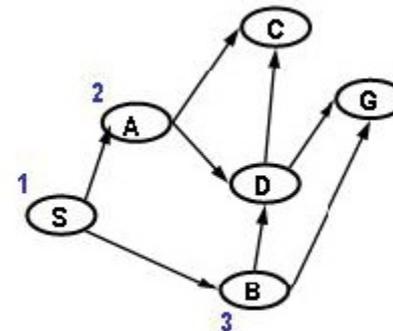
Breadth first search

	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3	(B S) (C A S) (D A S)	C,D,B,A,S
4		
5		
6		



Breadth first search

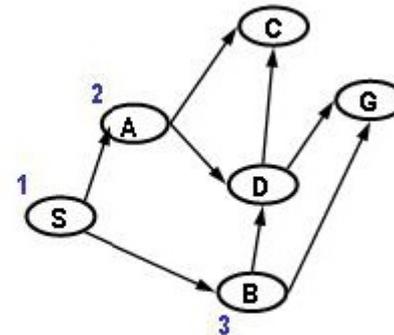
	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3	(B S) (C A S) (D A S)	C,D,B,A,S
4	(C A S) (D A S) (G B S)*	G,C,D,B,A,S
5		
6		



* We could have stopped here, when the first path to the goal was generated.

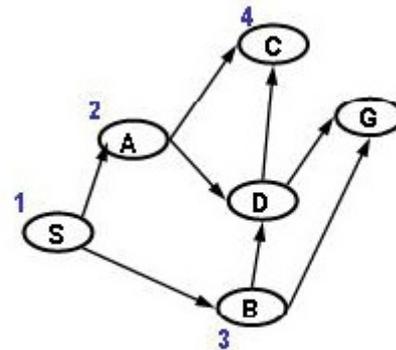
Breadth first search

	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3	(B S) (C A S) (D A S)	C,D,B,A,S
4	(C A S) (D A S) (G B S)*	G,C,D,B,A,S
5		
6		



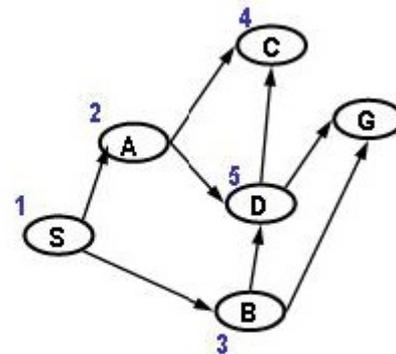
Breadth first search

	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3	(B S) (C A S) (D A S)	C,D,B,A,S
4	(C A S) (D A S) (G B S)*	G,C,D,B,A,S
5	(D A S) (G B S)	G,C,D,B,A,S
6		



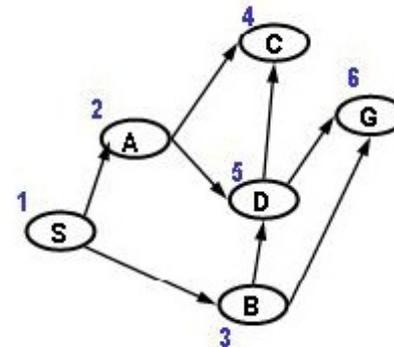
Breadth first search

	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3	(B S) (C A S) (D A S)	C,D,B,A,S
4	(C A S) (D A S) (G B S)*	G,C,D,B,A,S
5	(D A S) (G B S)	G,C,D,B,A,S
6		



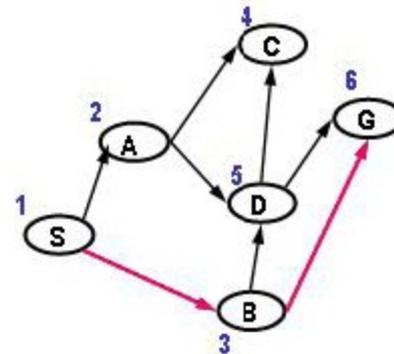
Breadth first search

	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3	(B S) (C A S) (D A S)	C,D,B,A,S
4	(C A S) (D A S) (G B S)*	G,C,D,B,A,S
5	(D A S) (G B S)	G,C,D,B,A,S
6	(G B S)	G,C,D,B,A,S

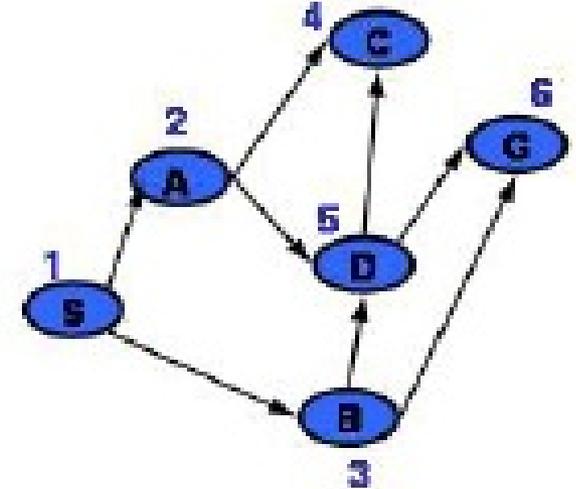
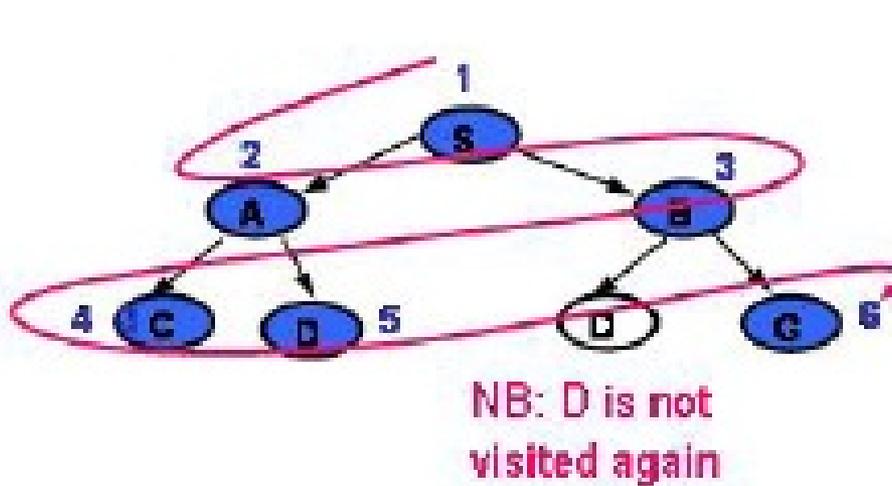


Breadth first search

	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3	(B S) (C A S) (D A S)	C,D,B,A,S
4	(C A S) (D A S) (G B S)*	G,C,D,B,A,S
5	(D A S) (G B S)	G,C,D,B,A,S
6	(G B S)	G,C,D,B,A,S



Breadth first search



Numbers indicate order pulled off of Q (expanded)

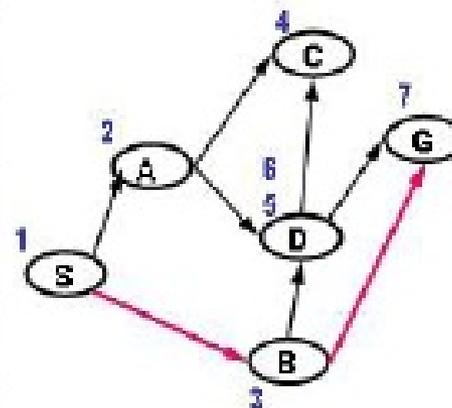
Dark blue fill = Visited & Expanded

Light gray fill = Visited

Breadth first (Without visited list)

Pick first element of Q; Add path extensions to end of Q

	Q
1	(S)
2	(A S) (B S)
3	(B S) (C A S) (D A S)
4	(C A S) (D A S) (D B S) (G B S)*
5	(D A S) (D B S) (G B S)
6	(D B S) (G B S) (C D A S) (G D A S)
7	(G B S) (C D A S) (G D A S) (C D B S) (G D B S)



Added paths in blue

We show the paths in **reversed** order; the node's state is the first entry.

* We could have stopped here, when the first path to the goal was generated.

Properties of breadth-first search

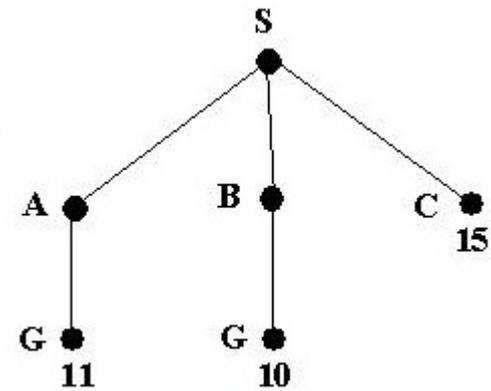
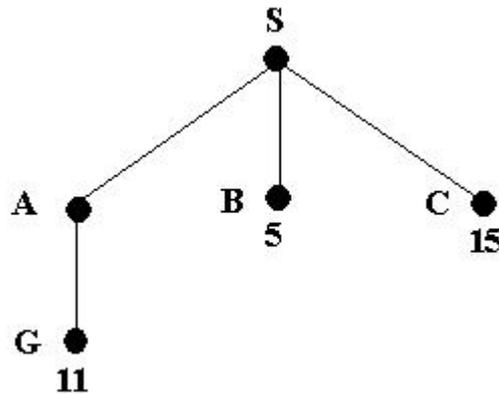
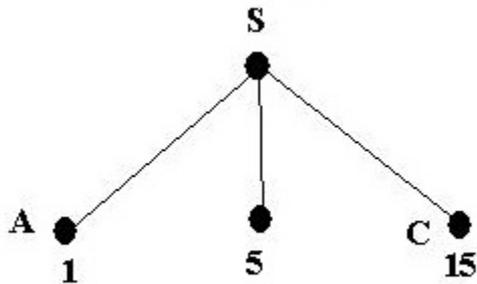
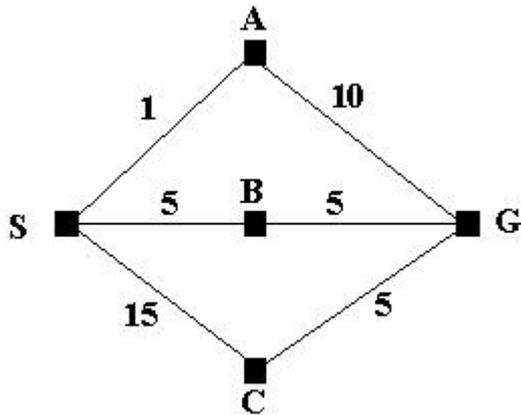
- Complete? Yes (if b is finite)
- Time? $1+b+b^2+b^3+\dots +b^d + b(b^d-1) = O(b^{d+1})$
- Space? $O(b^{d+1})$ (keeps every node in memory)
- Optimal? Yes (if cost = 1 per step)
- **Space** is the bigger problem (more than time)

Depth	Nodes	Time	Memory
0	1	1 millisecond	100 kbytes
2	111	0.1 second	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	10^6	18 minutes	111 megabytes
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11,111 terabytes

Uniform-cost search

- Expand least-cost unexpanded node
- **Implementation:**
 - *fringe* = queue ordered by path cost
- Equivalent to breadth-first if step costs all equal
- **Complete?** Yes, if step cost $\geq \epsilon$
- **Time?** # of nodes with $g \leq$ cost of optimal solution, $O(b^{\text{ceiling}(C^*/\epsilon)})$ where C^* is the cost of the optimal solution
- **Space?** # of nodes with $g \leq$ cost of optimal solution, $O(b^{\text{ceiling}(C^*/\epsilon)})$
- **Optimal?** Yes – nodes expanded in increasing order of $g(n)$

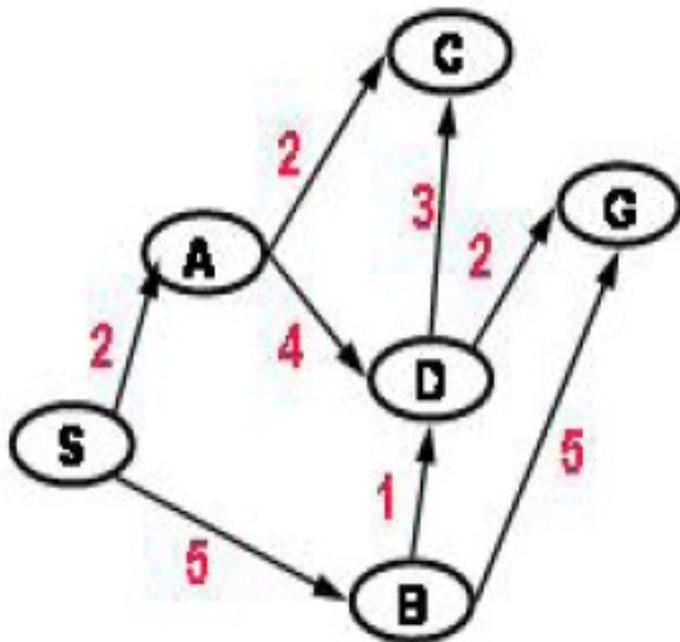
Uniform-cost search



Taken from http://www.cs.nott.ac.uk/~gzk/courses/g5aiai/003blindsearches/blind_searches.htm

Uniform cost search

- Each link has a length or cost (which is always greater than 0)
- We want shortest or least cost path

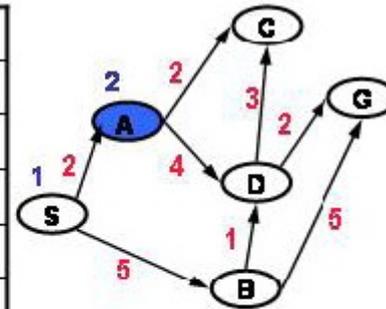


Total path cost:

(S A C)	4
(S B D G)	8
(S A D C)	9

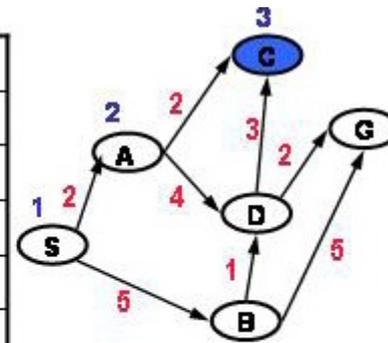
Uniform cost search

	Q
1	(0 S)
2	(2 A S) (5 B S)



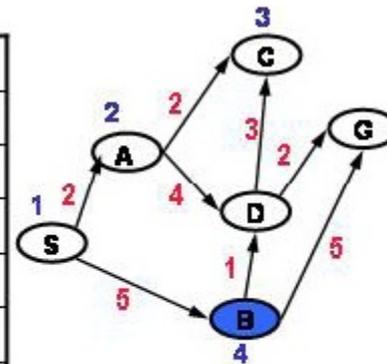
Uniform cost search

	Q
1	(0 S)
2	(2 A S) (5 B S)
3	(4 C A S) (6 D A S) (5 B S)



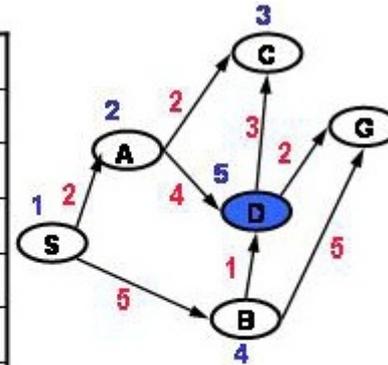
Uniform cost search

	Q
1	(0 S)
2	(2 A S) (5 B S)
3	(4 C A S) (6 D A S) (5 B S)
4	(6 D A S) (5 B S)



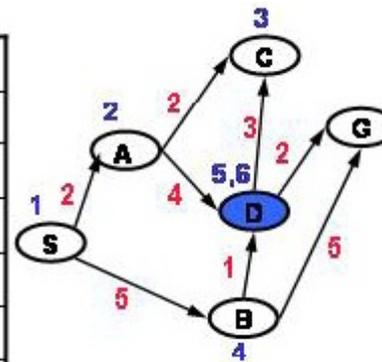
Uniform cost search

	Q
1	(0 S)
2	(2 A S) (5 B S)
3	(4 C A S) (6 D A S) (5 B S)
4	(6 D A S) (5 B S)
5	(6 D B S) (10 G B S) (6 D A S)



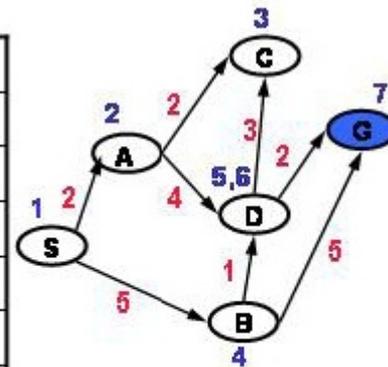
Uniform cost search

	Q
1	<u>(0 S)</u>
2	<u>(2 A S)</u> (5 B S)
3	<u>(4 C A S)</u> (6 D A S) (5 B S)
4	(6 D A S) <u>(5 B S)</u>
5	<u>(6 D B S)</u> (10 G B S) (6 D A S)
6	<u>(8 G D B S)</u> (9 C D B S) (10 G B S) (6 D A S)



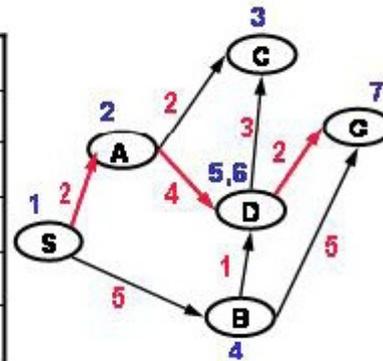
Uniform cost search

	Q
1	<u>(0 S)</u>
2	<u>(2 A S)</u> (5 B S)
3	<u>(4 C A S)</u> (6 D A S) (5 B S)
4	(6 D A S) <u>(5 B S)</u>
5	<u>(6 D B S)</u> (10 G B S) (6 D A S)
6	(8 G D B S) (9 C D B S) (10 G B S) <u>(6 D A S)</u>
7	(8 G D A S) (9 C D A S) (8 G D B S) (9 C D B S) (10 G B S)



Uniform cost search

	Q
1	<u>(0 S)</u>
2	<u>(2 A S)</u> (5 B S)
3	<u>(4 C A S)</u> (6 D A S) (5 B S)
4	(6 D A S) <u>(5 B S)</u>
5	<u>(6 D B S)</u> (10 G B S) (6 D A S)
6	<u>(8 G D B S)</u> (9 C D B S) (10 G B S) (6 D A S)
7	<u>(8 G D A S)</u> (9 C D A S) (8 G D B S) (9 C D B S) (10 G B S)

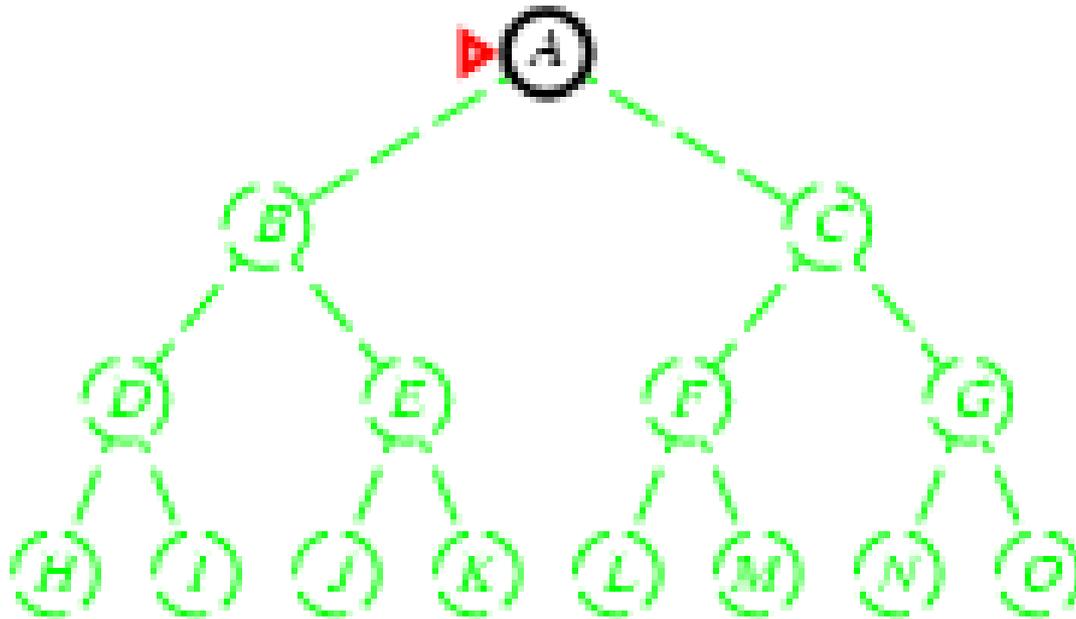


Why not stop on the first goal

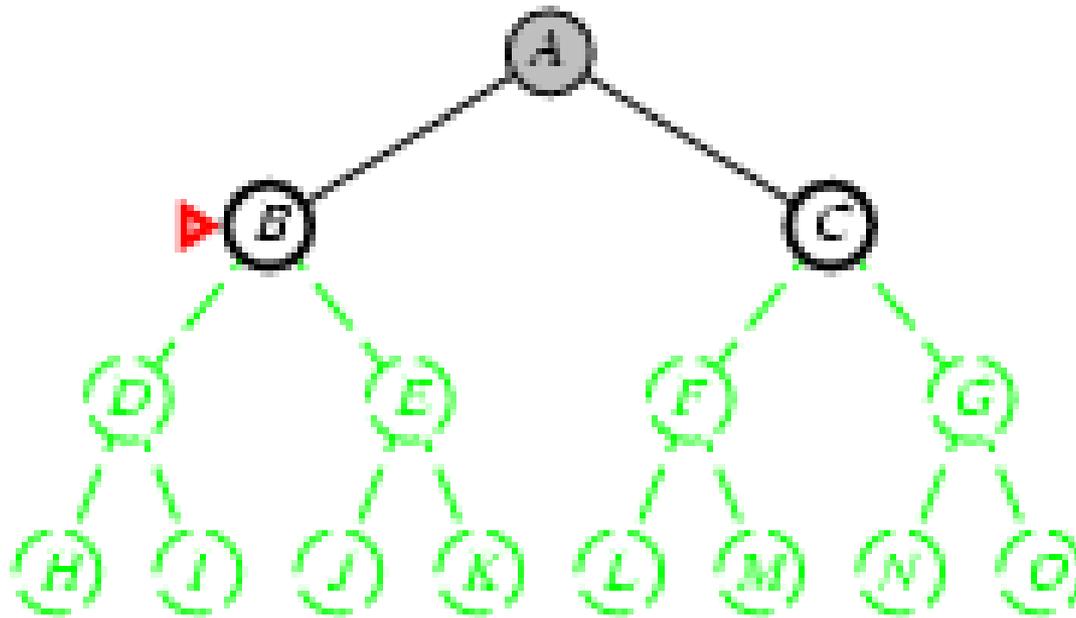
- **When doing Uniform Cost, it is not correct to stop the search when the first path to a goal is generated, that is, when a node whose state is a goal is added to Q.**
- **We must wait until such a path is pulled off the Q and tested in step 3. It is only at this point that we are sure it is the shortest path to a goal since there are no other shorter paths that remain unexpanded.**
- **This contrasts with the Any Path searches where the choice of where to test for a goal was a matter of convenience and efficiency, not correctness.**
- **In the previous example, a path to G was generated at step 5, but it was a different, shorter, path at step 7 that we accepted.**

Depth-first search

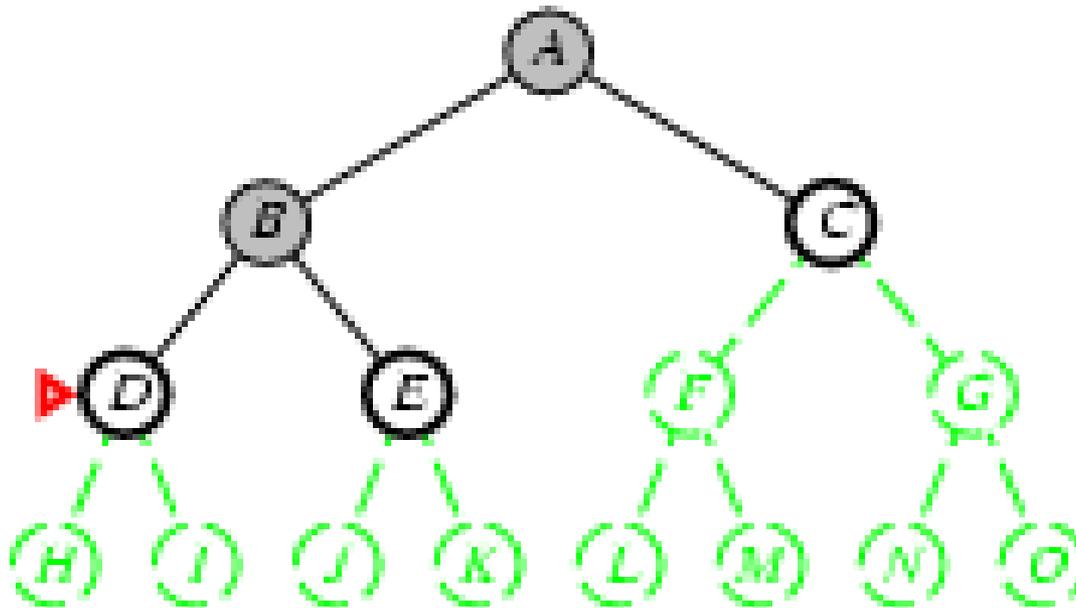
- Expand deepest unexpanded node
- **Implementation:**
 - *fringe* = LIFO queue (stack) , i.e., put successors at front



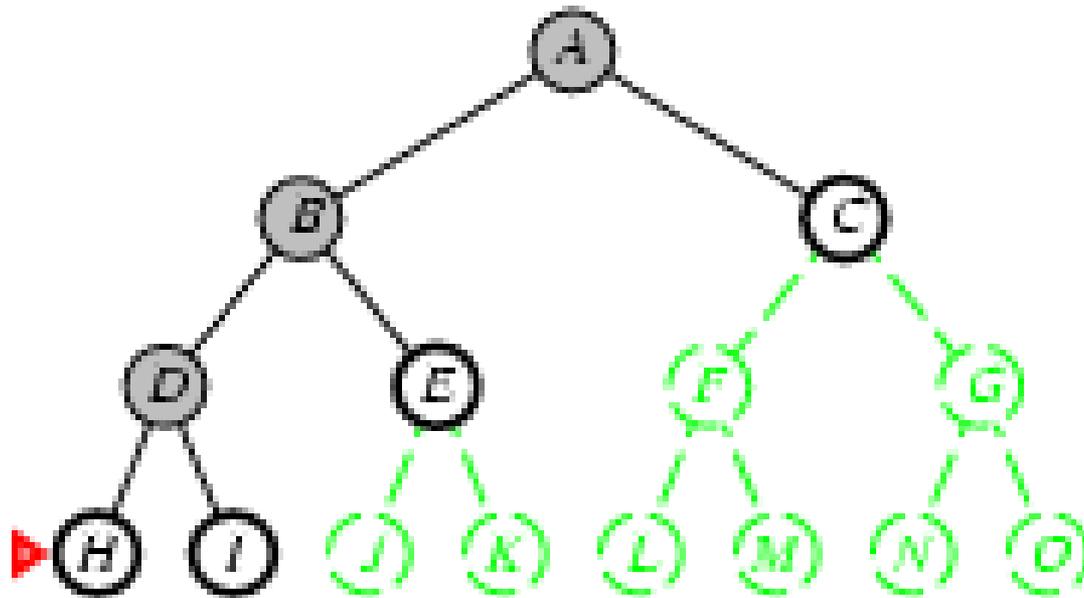
Depth-first search



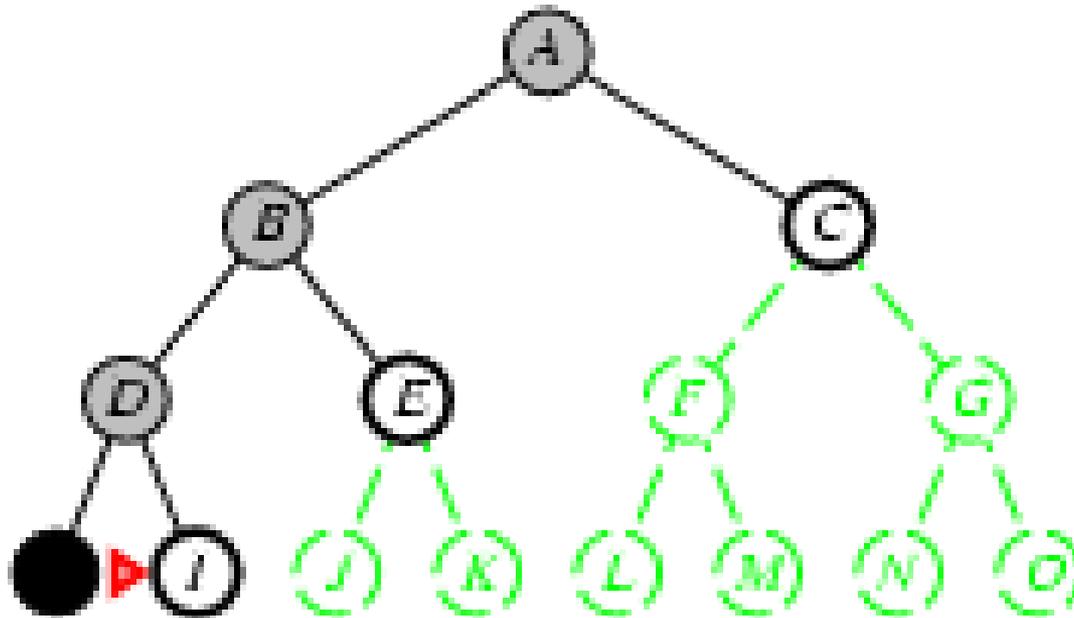
Depth-first search



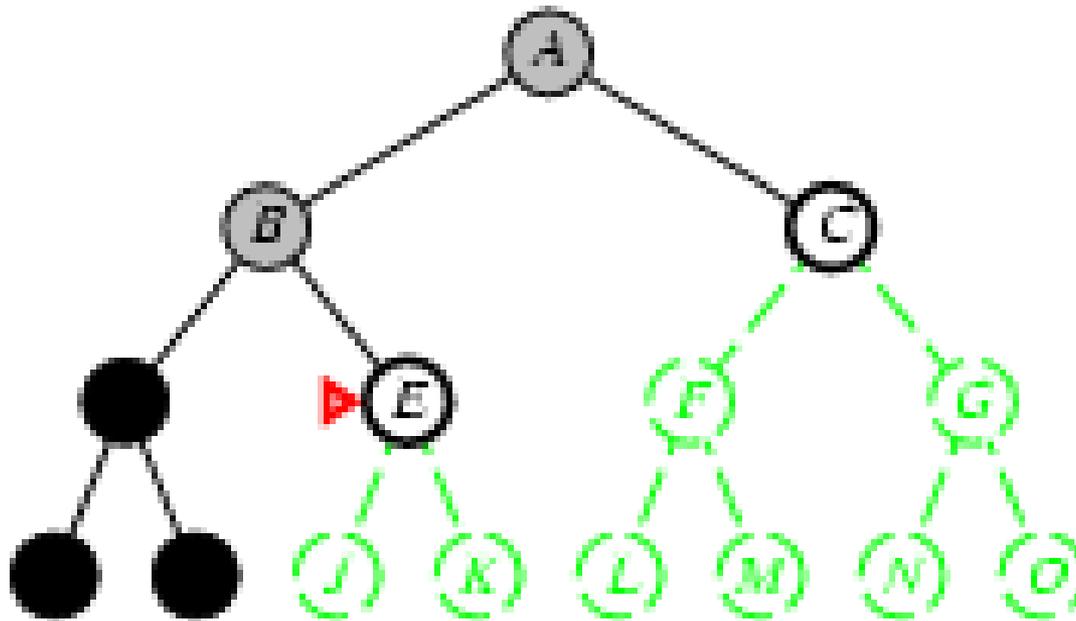
Depth-first search



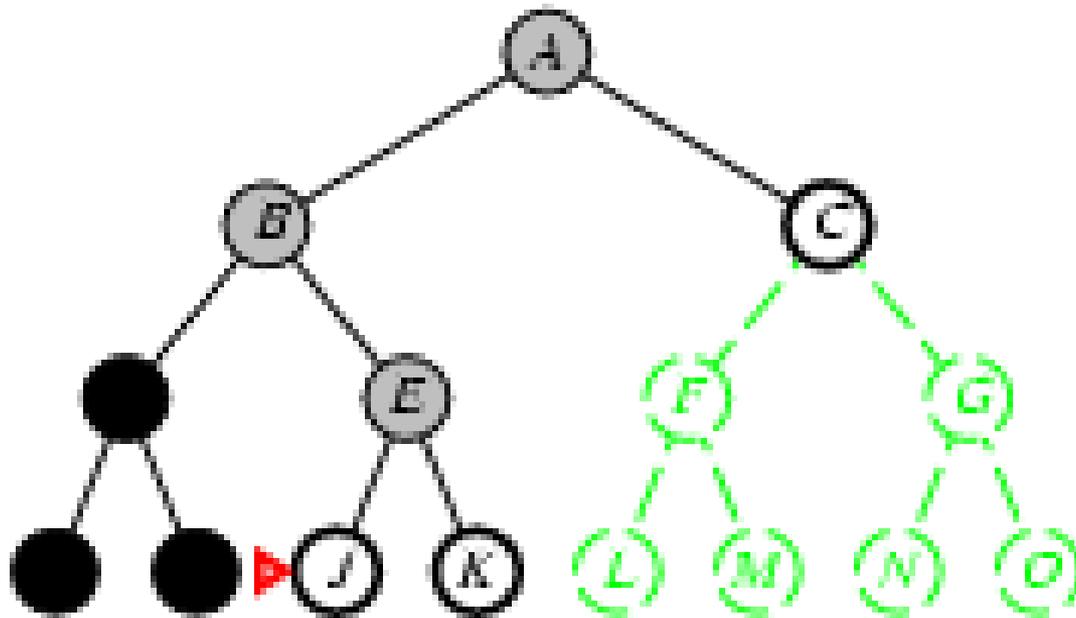
Depth-first search



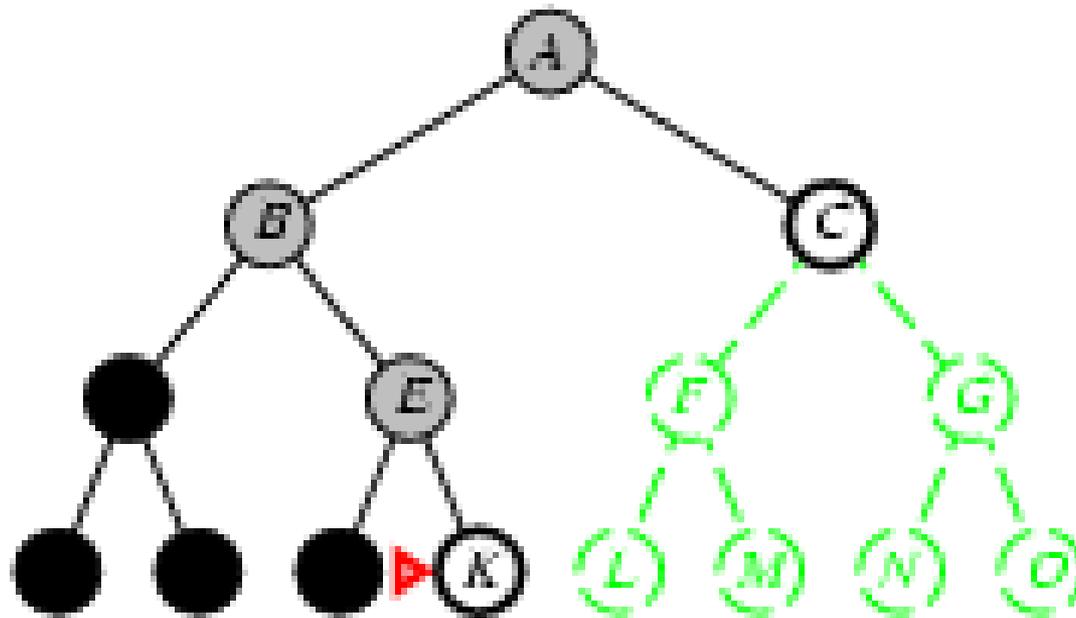
Depth-first search



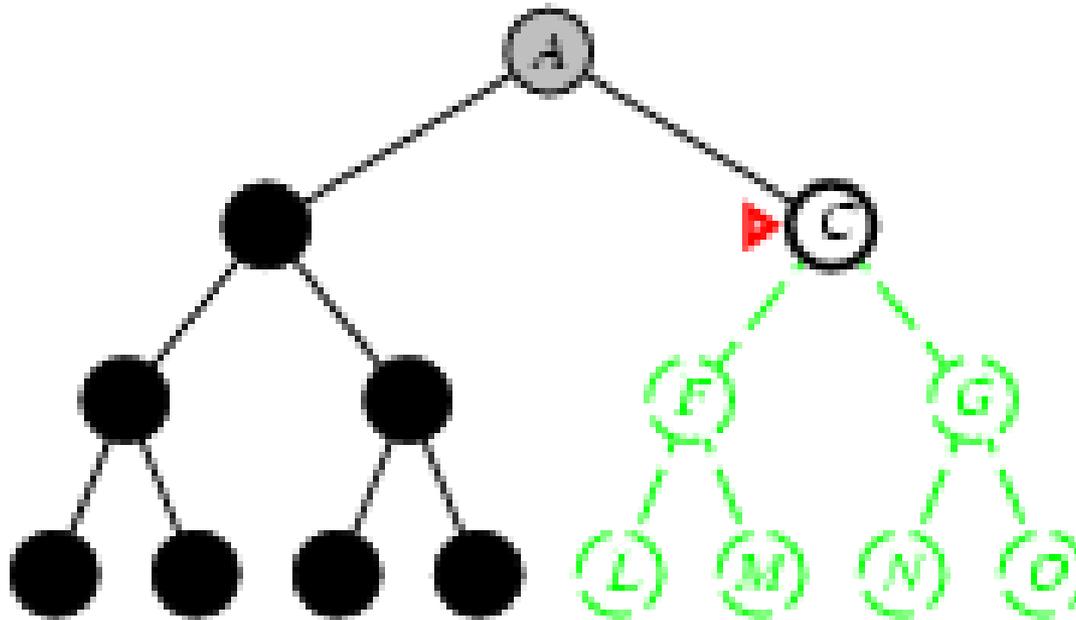
Depth-first search



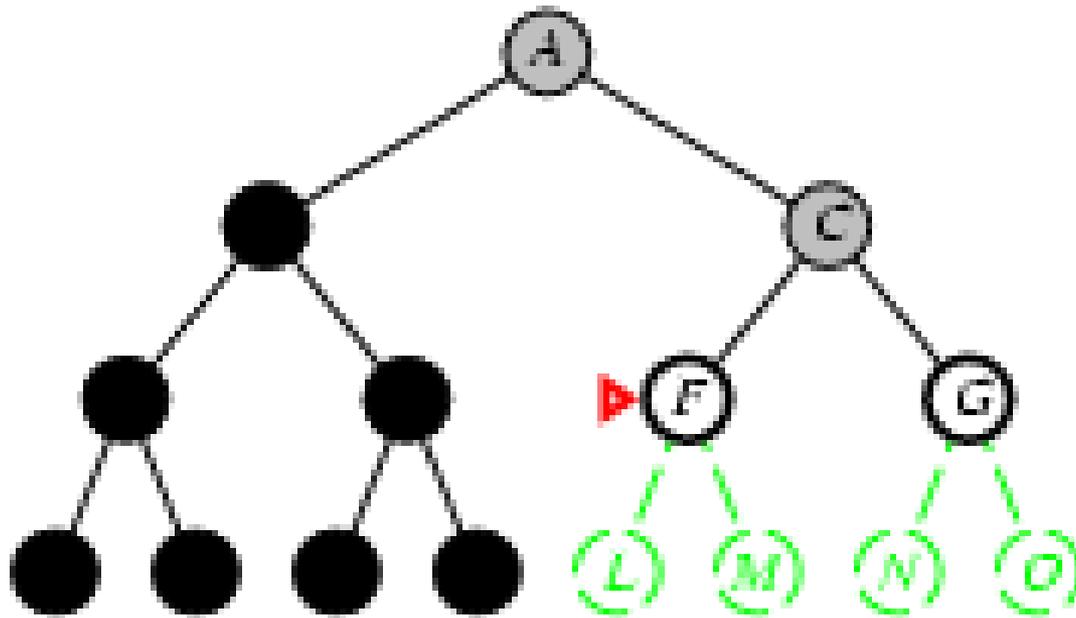
Depth-first search



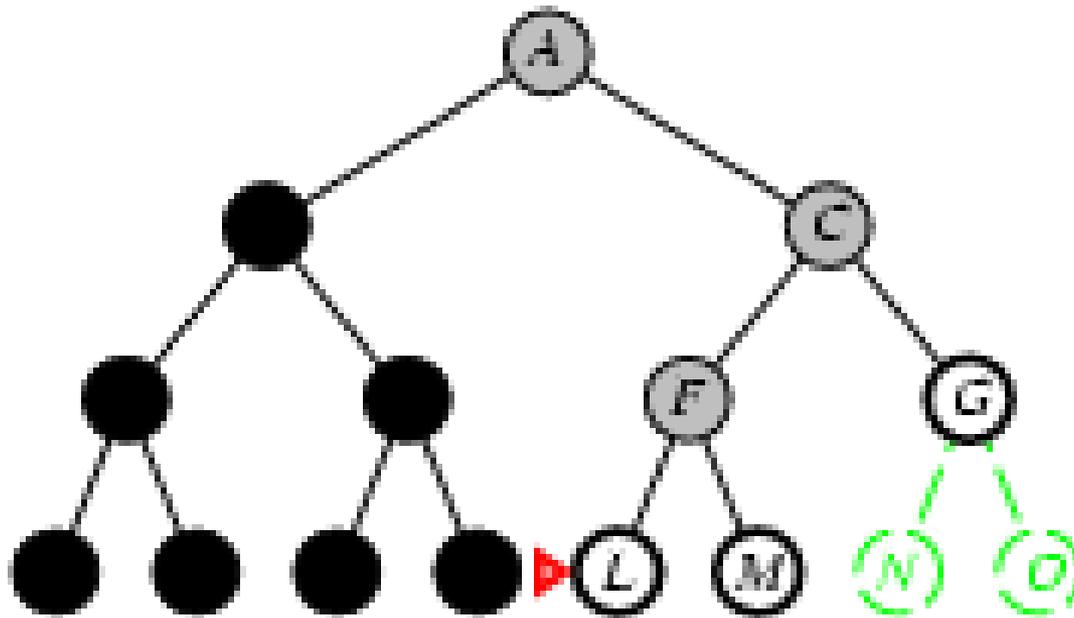
Depth-first search



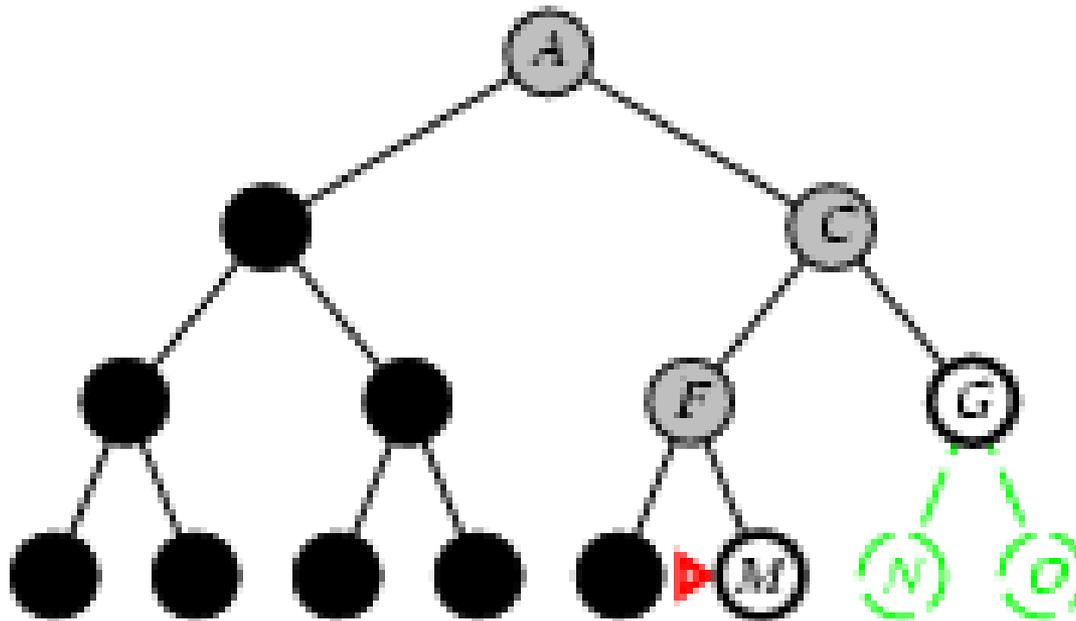
Depth-first search



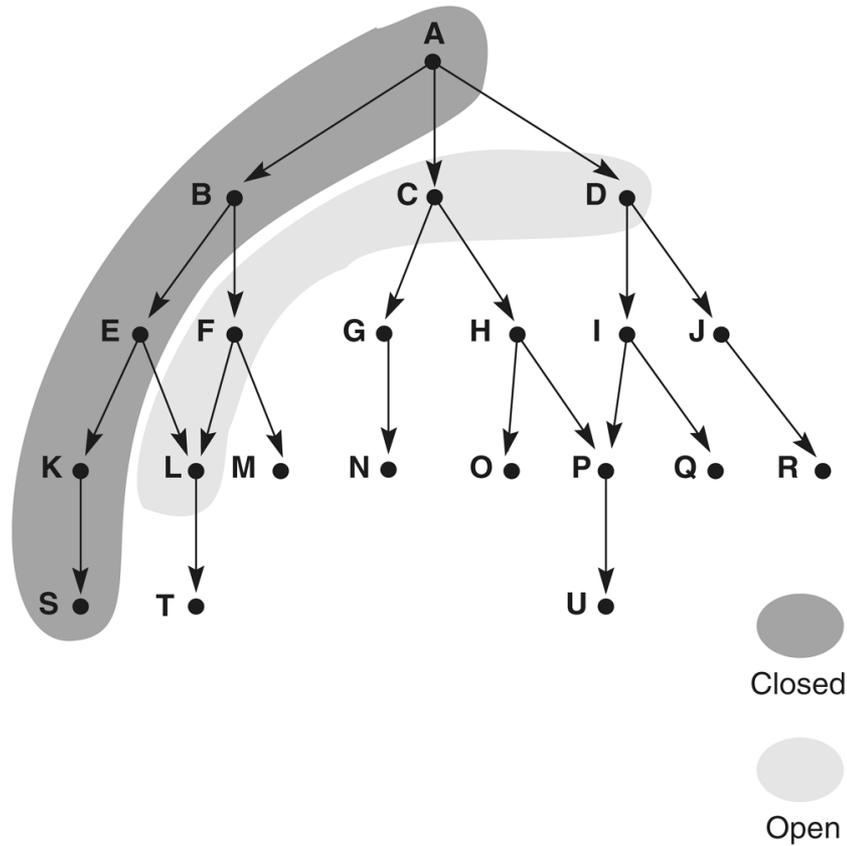
Depth-first search



Depth-first search

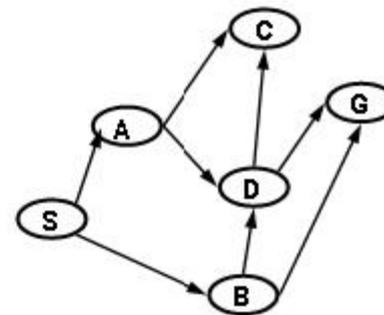


Depth-first search



Depth first search

	Q	Visited
1		
2		
3		
4		
5		



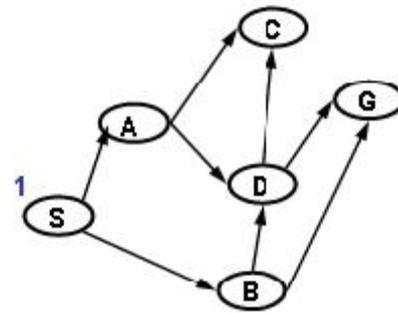
Pick first element of Q; Add path extensions to front of Q

Added paths in **blue**

We show the paths in **reversed** order; the node's state is the first entry.

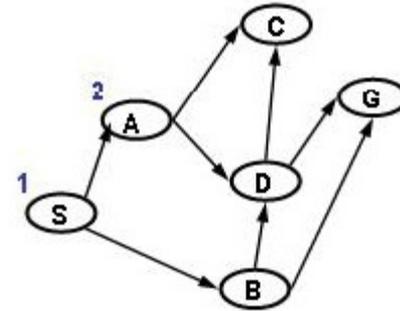
Depth first search

	Q	Visited
1	(S)	S
2		
3		
4		
5		



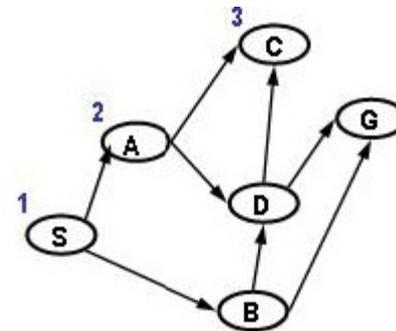
Depth first search

	Q	Visited
1	(S)	S
2	(A S) (B S)	A, B, S
3		
4		
5		



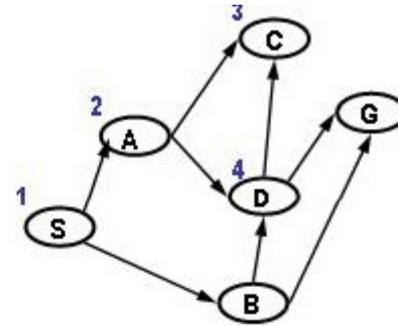
Depth first search

	Q	Visited
1	(S)	S
2	(A S) (B S)	A, B, S
3	(C A S) (D A S) (B S)	C, D, B, A, S
4		
5		



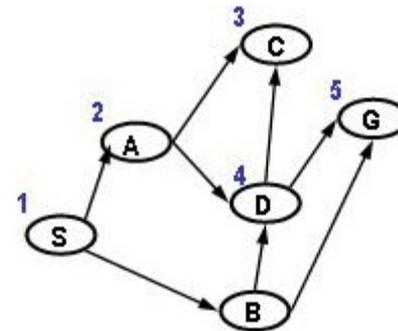
Depth first search

	Q	Visited
1	(S)	S
2	(A S) (B S)	A, B, S
3	(C A S) (D A S) (B S)	C, D, B, A, S
4	(D A S) (B S)	C, D, B, A, S
5		



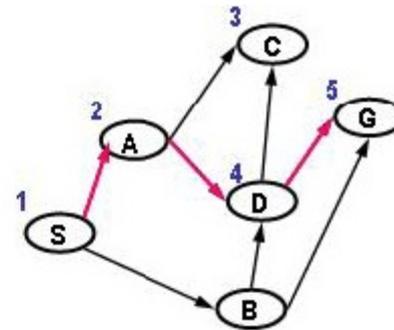
Depth first search

	Q	Visited
1	(S)	S
2	(A S) (B S)	A, B, S
3	(C A S) (D A S) (B S)	C, D, B, A, S
4	(D A S) (B S)	C, D, B, A, S
5	(G D A S) (B S)	G, C, D, B, A, S



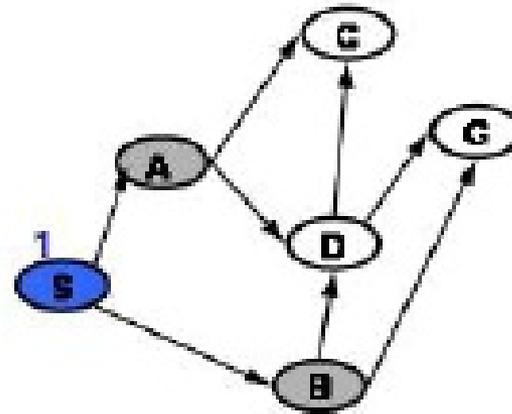
Depth first search

	Q	Visited
1	(S)	S
2	(A S) (B S)	A, B, S
3	(C A S) (D A S) (B S)	C, D, B, A, S
4	(D A S) (B S)	C, D, B, A, S
5	(G D A S) (B S)	G, C, D, B, A, S



Depth-first search

Another (easier?) way to see it



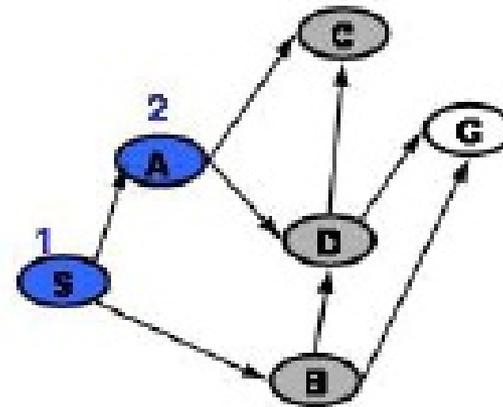
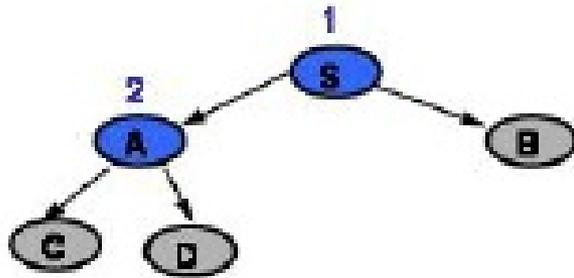
Numbers indicate order pulled off of Q (expanded)

Dark blue fill = Visited & Expanded

Light gray fill = Visited

Depth-first search

Another (easier?) way to see it



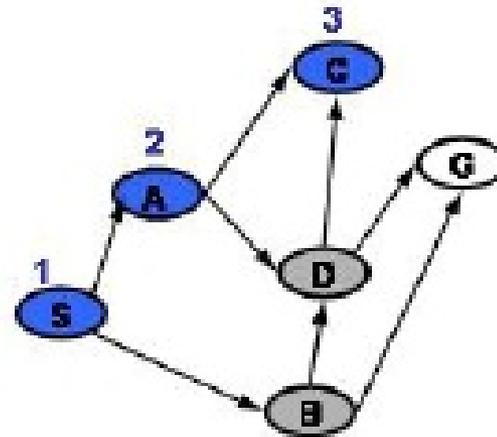
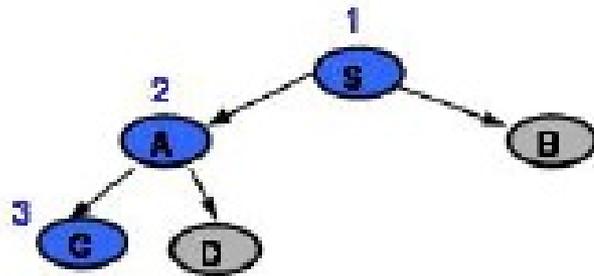
Numbers indicate order pulled off of Q (expanded)

Dark blue fill = Visited & Expanded

Light gray fill = Visited

Depth-first search

Another (easier?) way to see it



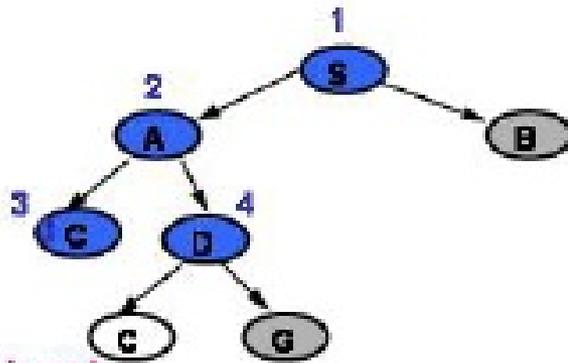
Numbers indicate order pulled off of Q (expanded)

Dark blue fill = Visited & Expanded

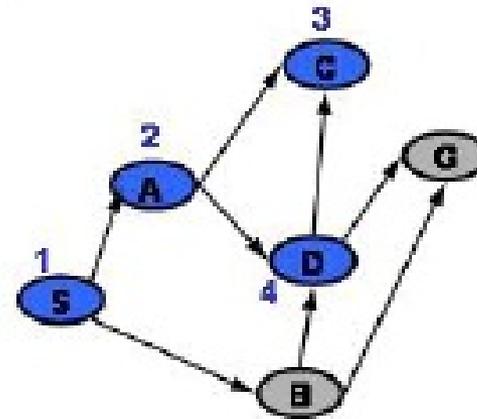
Light gray fill = Visited

Depth-first search

Another (easier?) way to see it



NB: C is not visited again



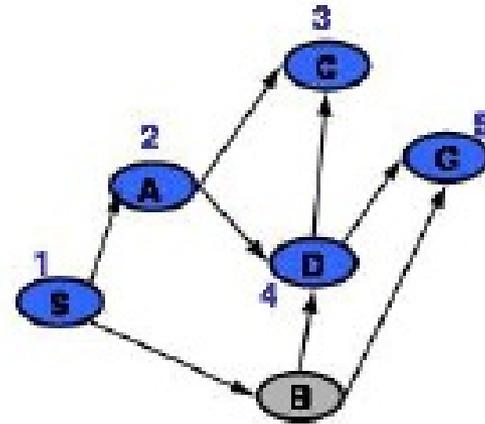
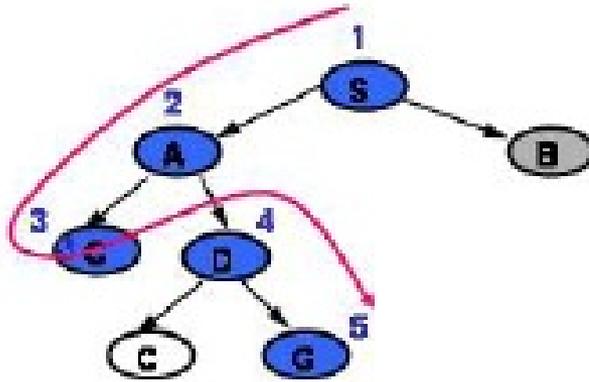
Numbers indicate order pulled off of Q (expanded)

Dark blue fill = Visited & Expanded

Light gray fill = Visited

Depth-first search

Another (easier?) way to see it



Numbers indicate order pulled off of Q (expanded)

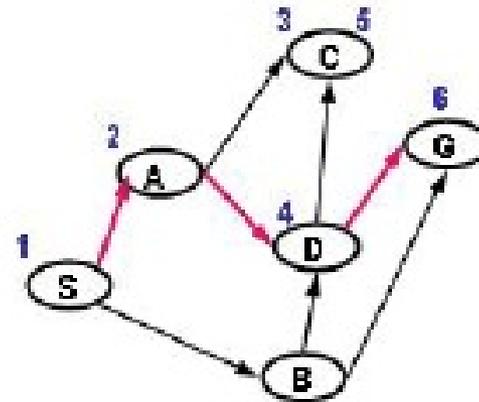
Dark blue fill = Visited & Expanded

Light gray fill = Visited

Depth-first search

Pick first element of Q; Add path extensions to front of Q

	Q
1	(S)
2	(A S) (B S)
3	(C A S) (D A S) (B S)
4	(D A S) (B S)
5	(C D A S) (G D A S) (B S)
6	(G D A S) (B S)



Added paths in blue

We show the paths in reversed order; the node's state is the first entry.

Do not extend a path to a state if the resulting path would have a loop.

Properties of depth-first search

- 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

Depth-limited search

= depth-first search with depth limit l ,
i.e., nodes at depth l have no successors

- Recursive implementation:

```

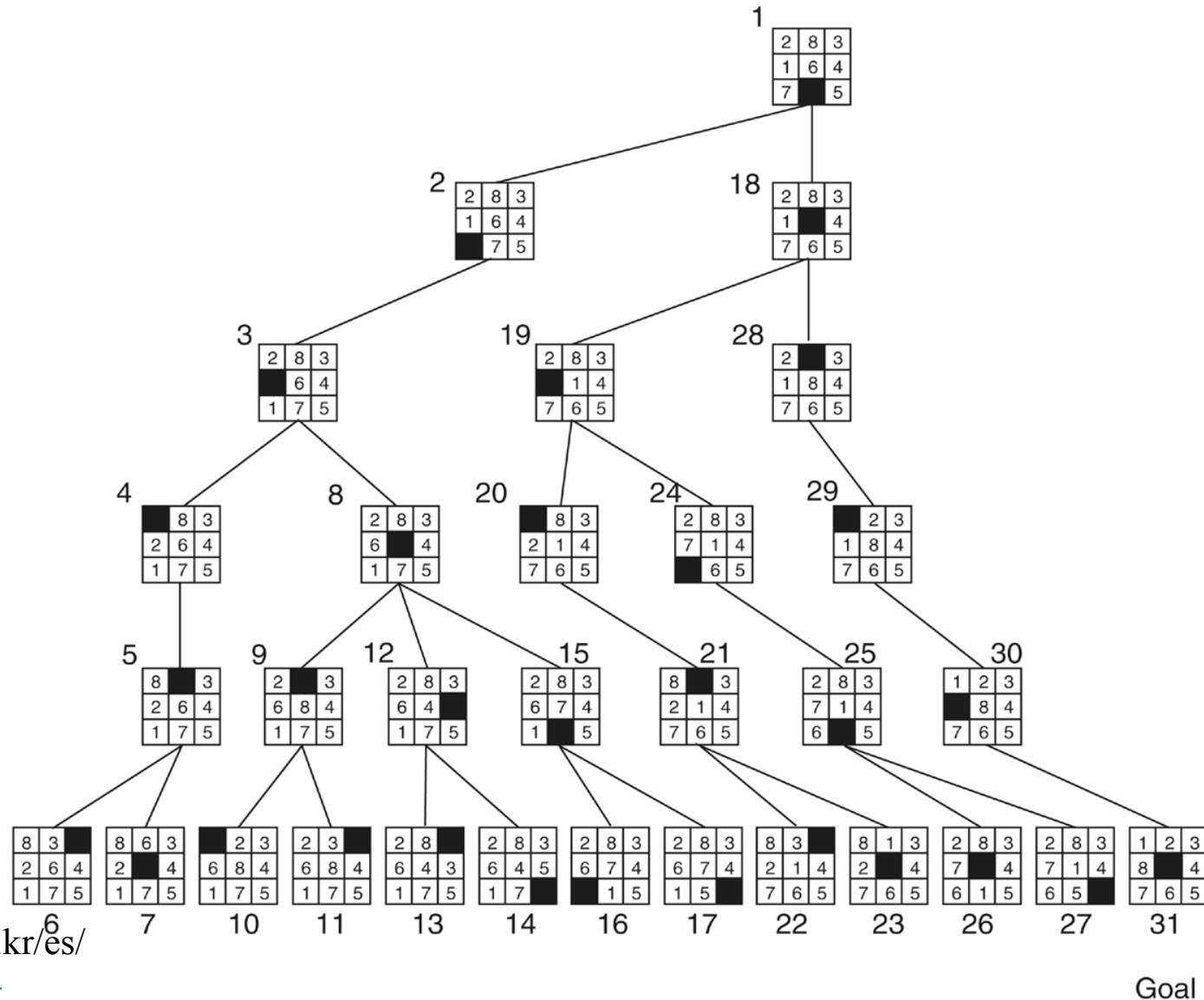
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
  RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
  cutoff-occurred? ← false
  if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
  else if DEPTH[node] = limit then return cutoff
  else for each successor in EXPAND(node, problem) do
    result ← RECURSIVE-DLS(successor, problem, limit)
    if result = cutoff then cutoff-occurred? ← true
    else if result ≠ failure then return result
  if cutoff-occurred? then return cutoff else return failure

```

Depth first search

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



Taken from <http://iis.kaist.ac.kr/es/>

Iterative deepening search

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or fail-  
ure  
  inputs: problem, a problem  
  for depth  $\leftarrow$  0 to  $\infty$  do  
    result  $\leftarrow$  DEPTH-LIMITED-SEARCH(problem, depth)  
    if result  $\neq$  cutoff then return result
```

Iterative deepening search $l = 0$

Limit = 0

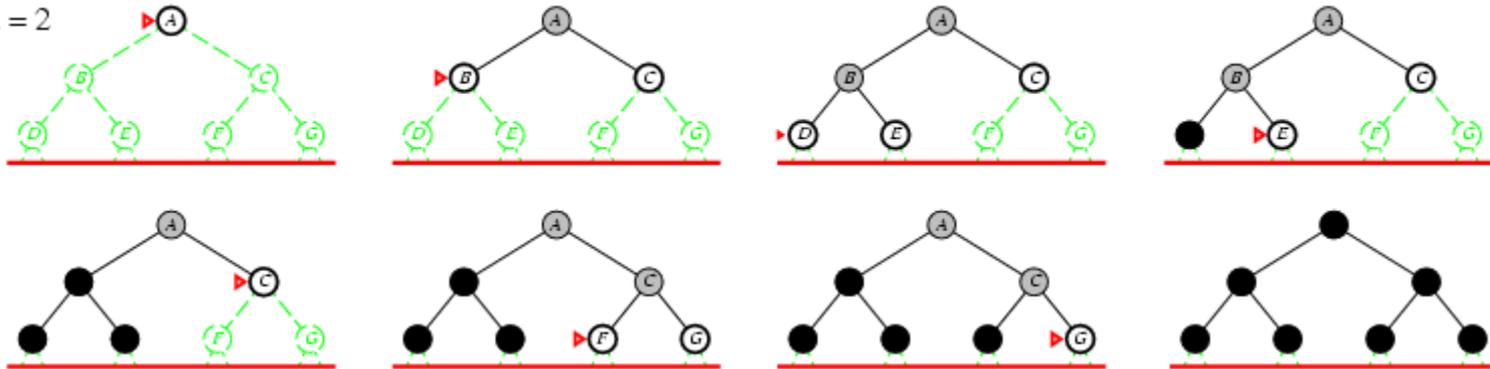


Iterative deepening search $l = 1$



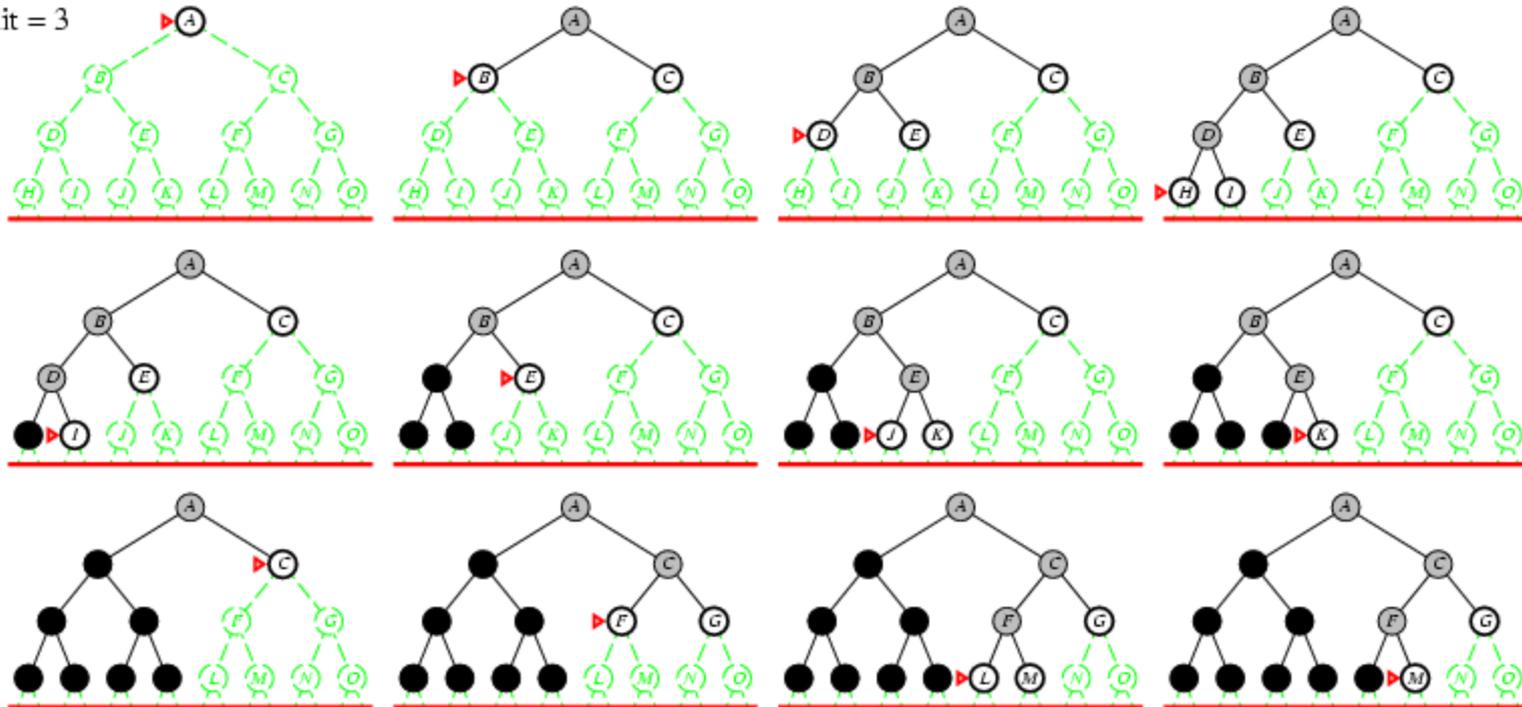
Iterative deepening search $l=2$

Limit = 2



Iterative deepening search $l=3$

Limit = 3



Iterative deepening search

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

$$N_{DLS} = b^0 + b^1 + b^2 + \dots + b^{d-2} + b^{d-1} + b^d$$

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

$$N_{IDS} = (d+1)b^0 + d b^1 + (d-1)b^2 + \dots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

- For $b = 10$, $d = 5$,
 - $N_{DLS} = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111$
 - $N_{IDS} = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456$
- Overhead = $(123,456 - 111,111)/111,111 = 11\%$

Properties of iterative deepening search

- Complete? Yes
- Time? $(d+1)b^0 + d b^1 + (d-1)b^2 + \dots + b^d = O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, if step cost = 1

Bidirectional search

- Run two simultaneous searches – one forward from the initial state, and the other backward from the goal, stopping when two searches meet in the middle
- $b^{d/2} + b^{d/2} < b^d$
- when there is a single goal, it is like forward search
- For more than a single goal – dummy goal state

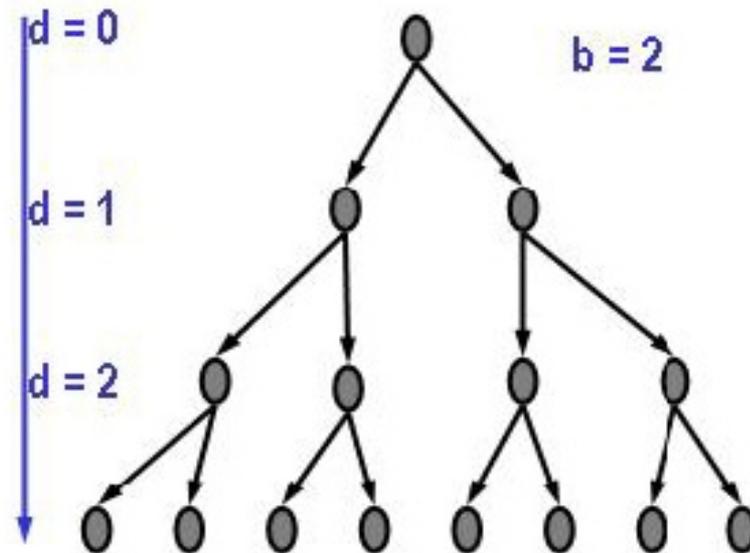
Summary of algorithms

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

Worst Case Running Time

Max Time \propto Max #Visited

- The number of states in the search space may be exponential in some “depth” parameter, e.g. number of actions in a plan, number of moves in a game.
- All the searches, with or without visited list, may have to visit each state at least once, in the worst case.
- So, all searches will have worst case running times that are at least proportional to the total number of states and therefore exponential in the “depth” parameter.



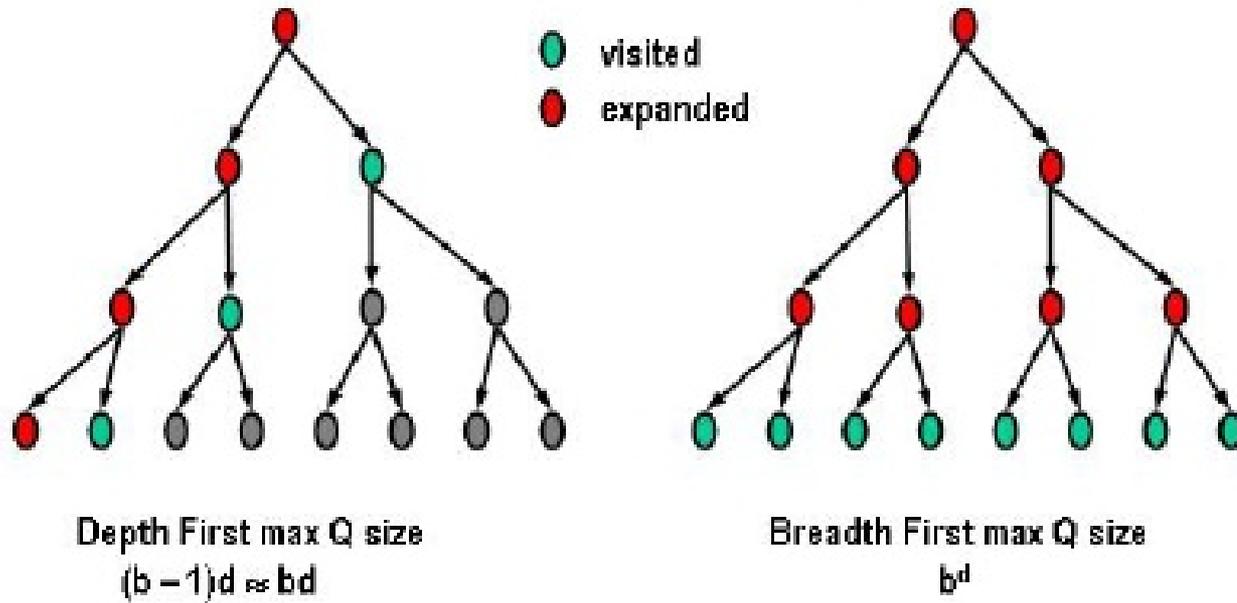
d is depth
 b is branching factor

$$b^d < \frac{(b^{d+1} - 1)}{(b - 1)} < b^{d+1}$$

states in tree

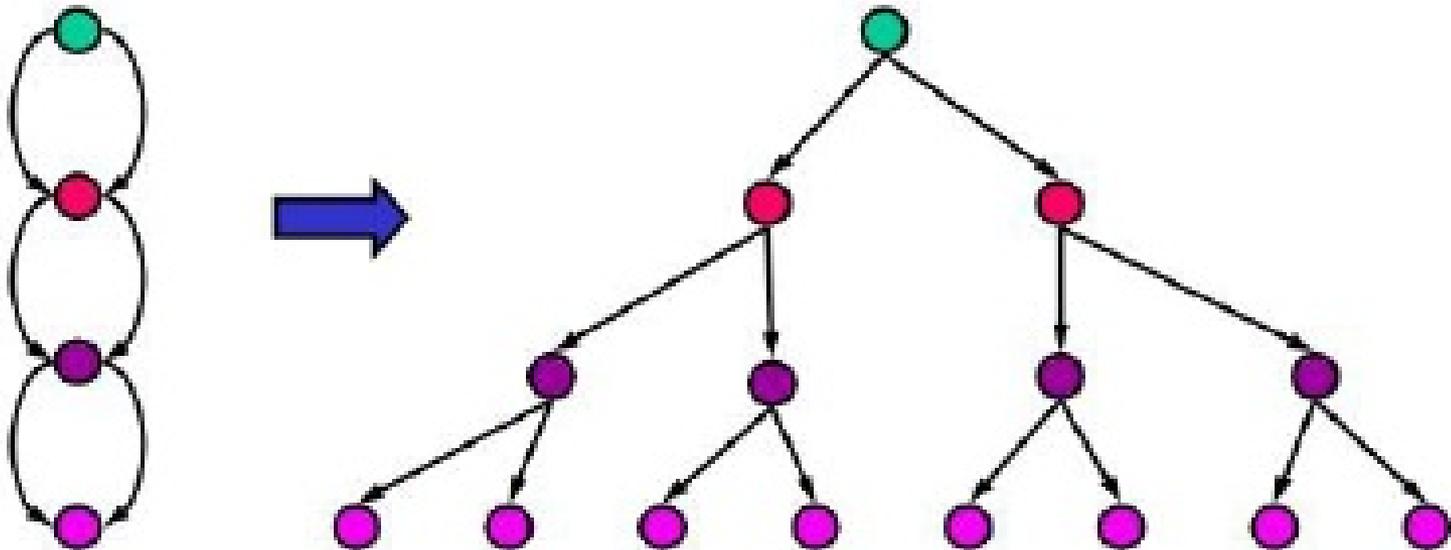
Worst Case Space

Max Q size = Max (#Visited - #Expanded)



Repeated states

- Failure to detect repeated states can turn a linear problem into an exponential one!



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)
```

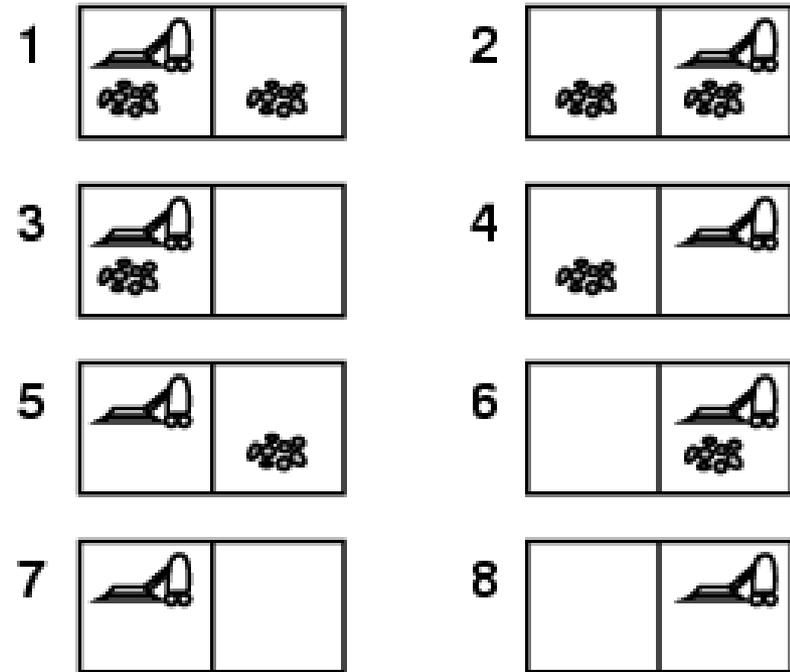
Problem types

- **Deterministic, fully observable** → **single-state problem**
 - Agent knows exactly which state it will be in; solution is a sequence
- **Non-observable** → **sensorless problem (conformant problem)**
 - Agent may have no idea where it is; solution is a sequence
- **Nondeterministic and/or partially observable** → **contingency problem**
 - percepts provide **new** information about current state
 - often **interleave**} search, execution
- **Unknown state space** → **exploration problem**

Example: vacuum world

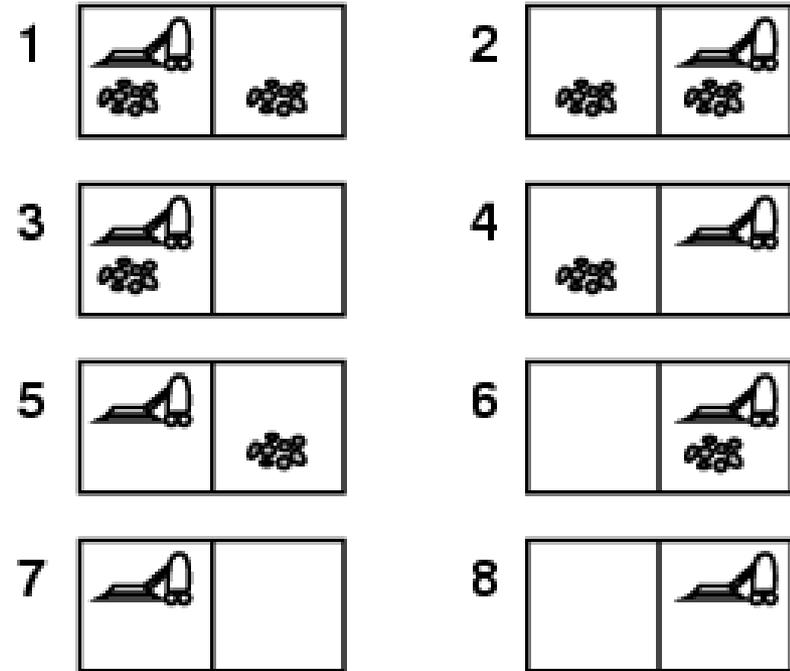
Single-state, start in #5.

Solution? [*Right, Suck*]

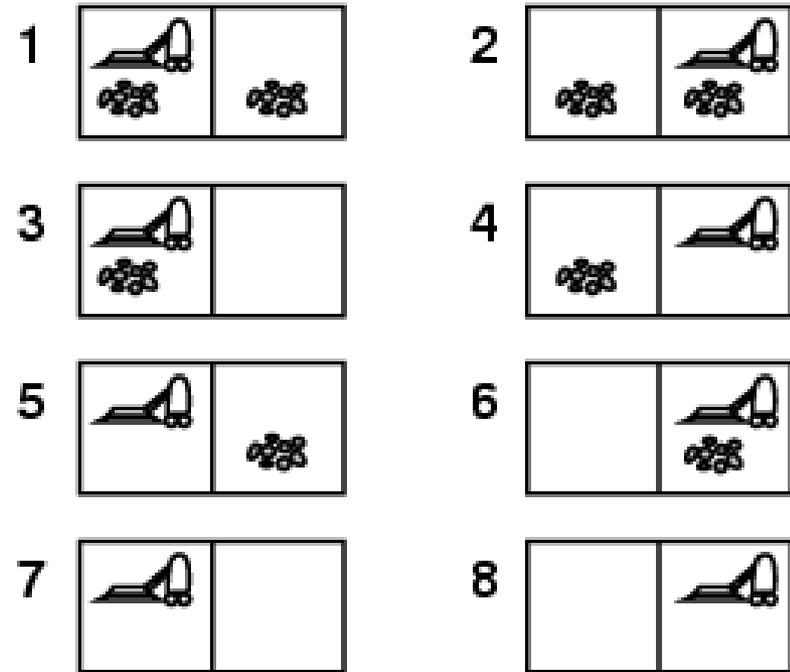


Example: vacuum world

- **Sensorless**, start in $\{1,2,3,4,5,6,7,8\}$ e.g.,
Right goes to $\{2,4,6,8\}$
Solution?
[Right,Suck,Left,Suck]



Example: vacuum world



- Contingency

- Nondeterministic: *Suck* may dirty a clean carpet
 - Partially observable: location, dirt at current location.
 - Percept: $[L, \text{Clean}]$, i.e., start in #5 or #7
- Solution? [*Right, if dirt then Suck*]

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