Solving problems by searching

Chapter 3

Why Search?

- To achieve goals or to maximize our utility we need to predict what the result of our actions in the future will be.
- There are many sequences of actions, each with their own utility.
- We want to find, or search for, the best one.

Search Overview

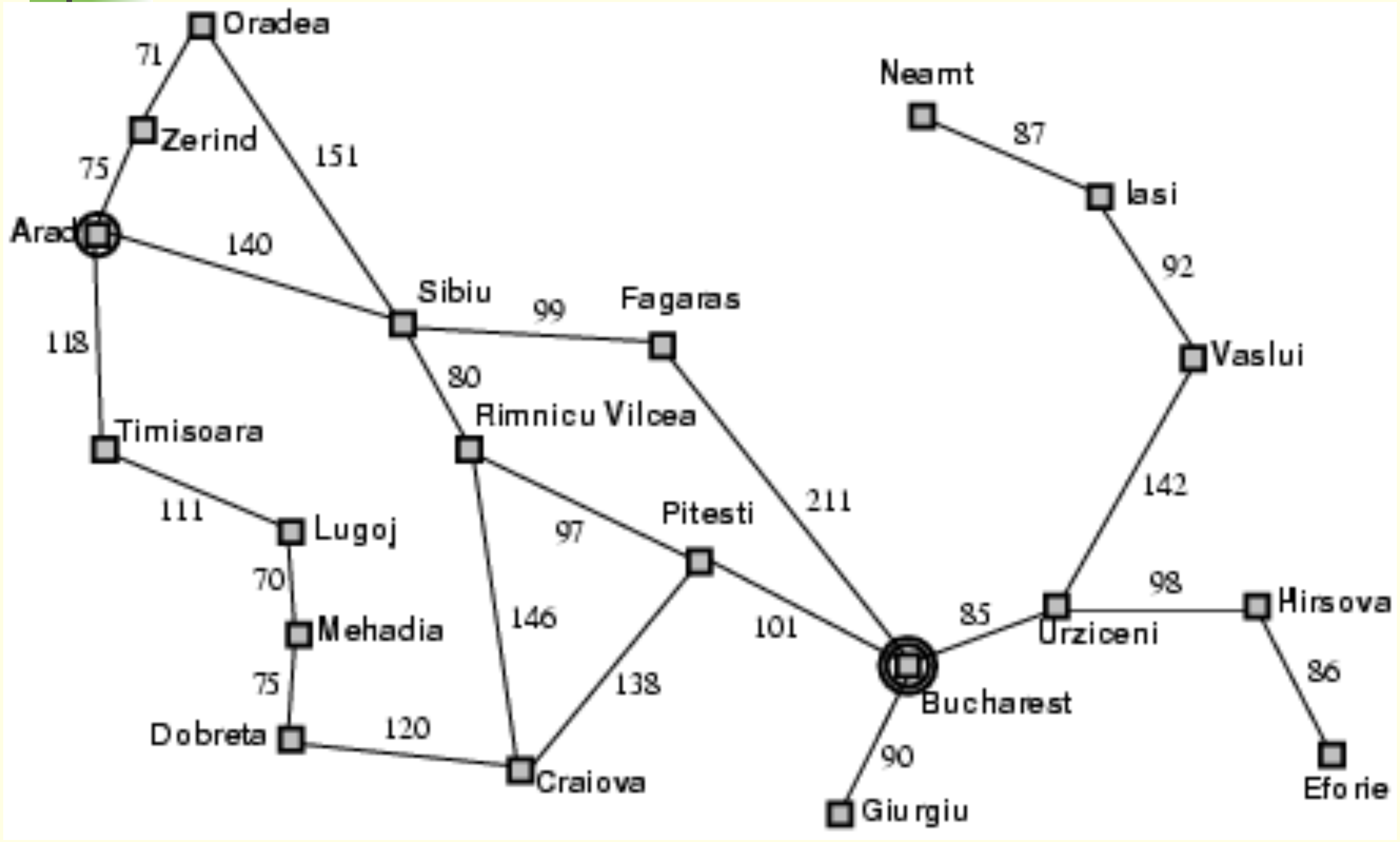
- Watch this video on search algorithms:

http://videolectures.net/aaai2010_thayer_bis/

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 or choose next city
- Find solution:
  - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Example: Romania



Task Environment

- Static / Dynamic

  Previous problem was static: no attention to changes in environment
- Observable / Partially Observable / Unobservable

  Previous problem was observable: it knew its states at all times.
- Deterministic / Stochastic

  Previous problem was deterministic: no new percepts
  were necessary, we can predict the future perfectly given our actions
- Discrete / continuous

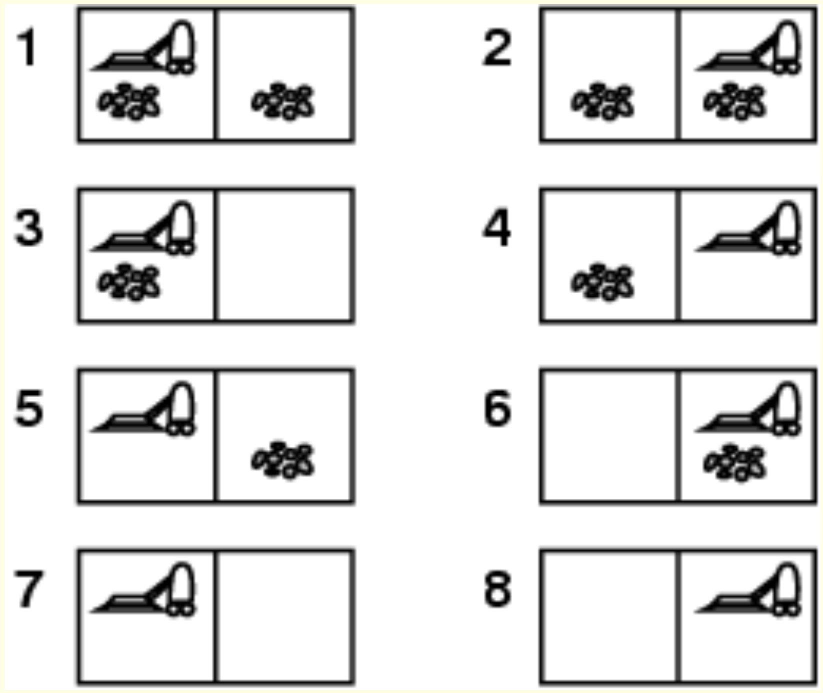
  Previous problem was discrete: we can enumerate all possibilities
- Single Agent

  No other agents interacting with your cost function
- Sequential

  Decisions depend on past decisions

Example: vacuum world

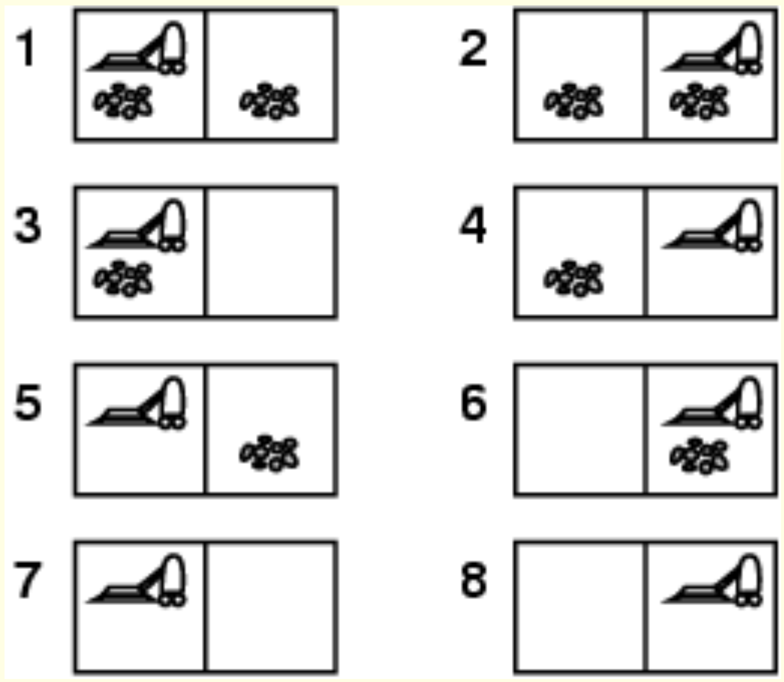
- Observable, start in #5.
  Solution?



Example: vacuum world

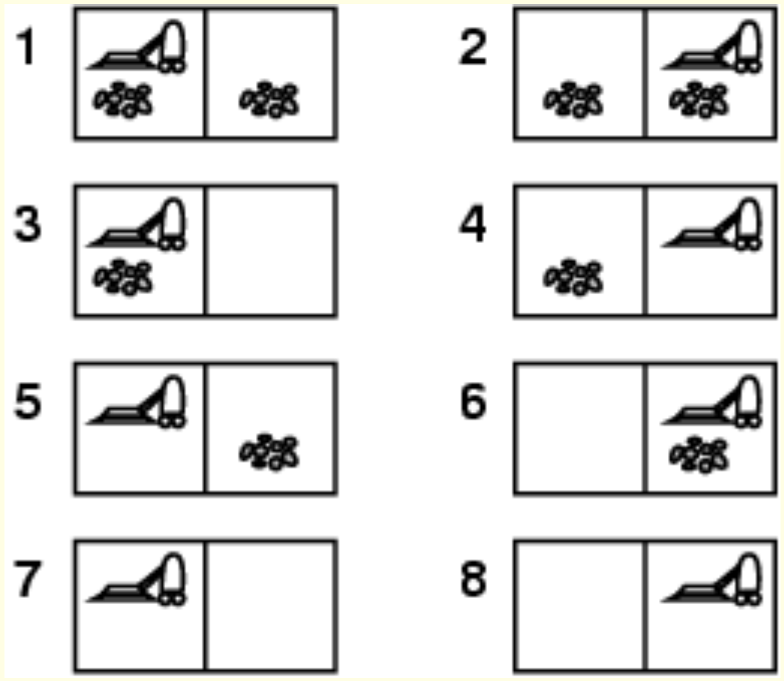
- Observable, start in #5.
  Solution? *[Right, Suck]*

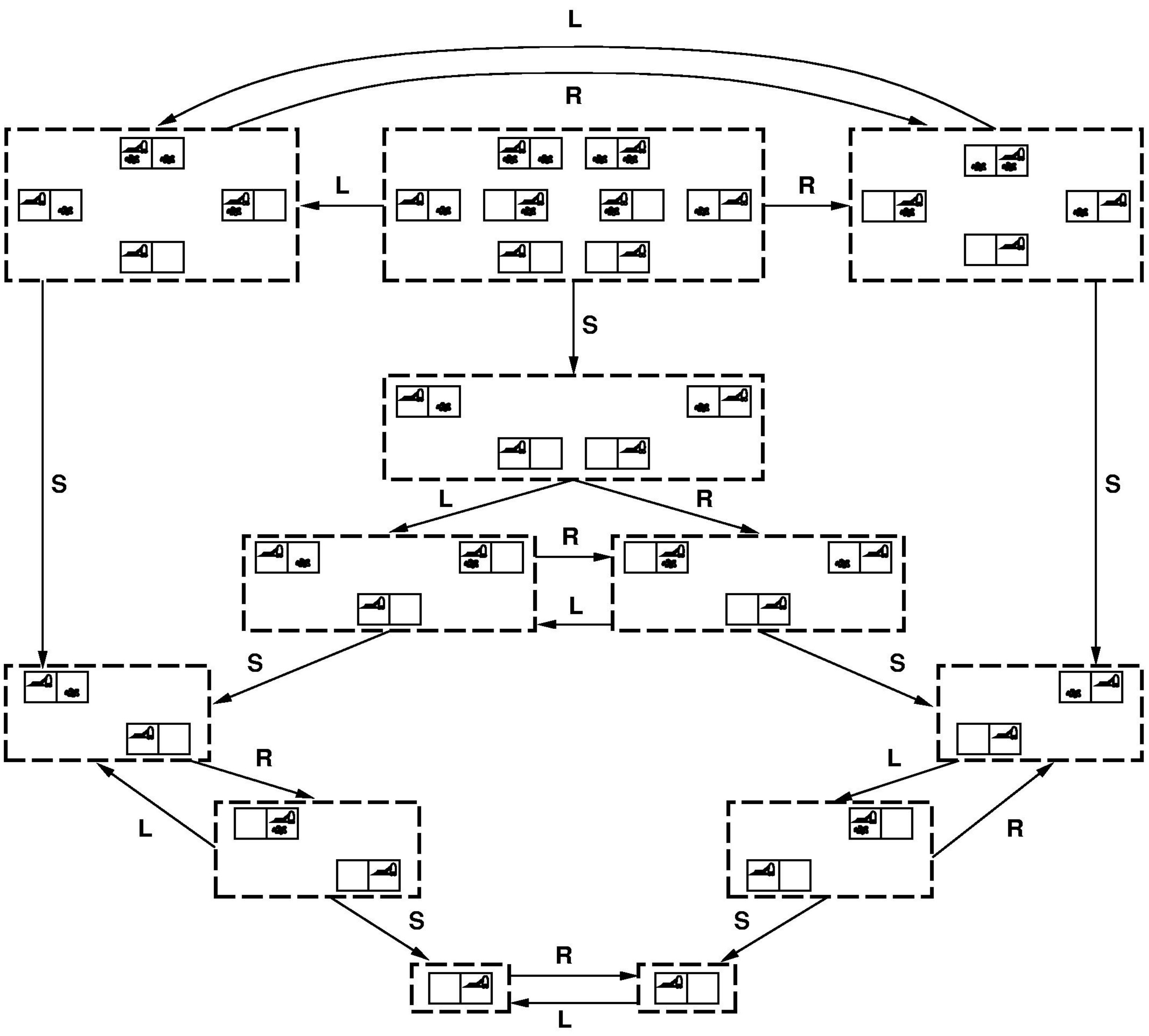
- Unobservable, start in {*1,2,3,4,5,6,7,8*} e.g.,
  Solution?



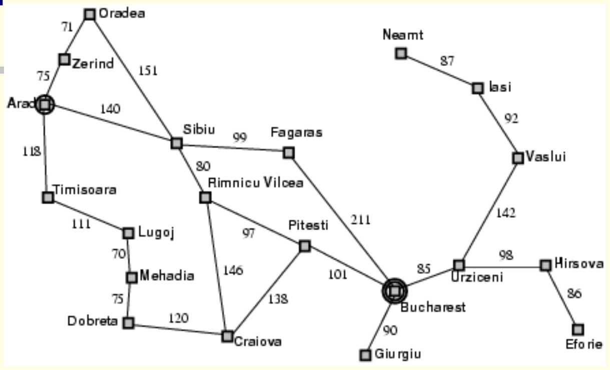
Example: vacuum world

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




L
R
L
R
S
S
S
L
R
R
L
S
S
R
L
L
R
S
S
R
L

Problem Formulation



A problem is defined by four items:

initial state e.g., "at Arad"

actions set of possible actions in current state x.

transition model *Result(x,a)* = state that follows from applying action a in state x. e.g., *Result(X2 (Arad), A4 (Arad → Zerind))* = X3 (Zerind)

goal test, e.g., *x* = "at Bucharest", *Checkmate(x)*

path cost (additive)

- e.g., sum of distances, number of actions executed, etc.
- *c(x,a,y)* is the step cost, assumed to be ≥ 0

A solution is a sequence of actions leading from the initial state to a goal state

Selecting a state space

- Real world is absurdly complex
  - state space must be abstracted for problem solving

- (Abstract) state ← set of real states

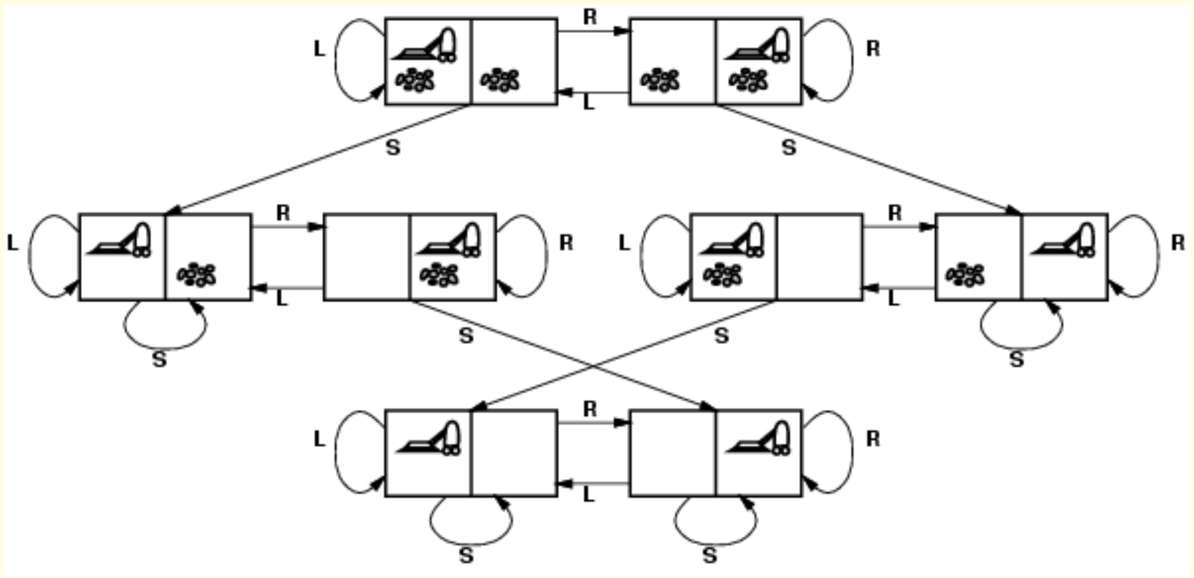
- (Abstract) action ← complex combination of real actions
  - e.g., "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.

- 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

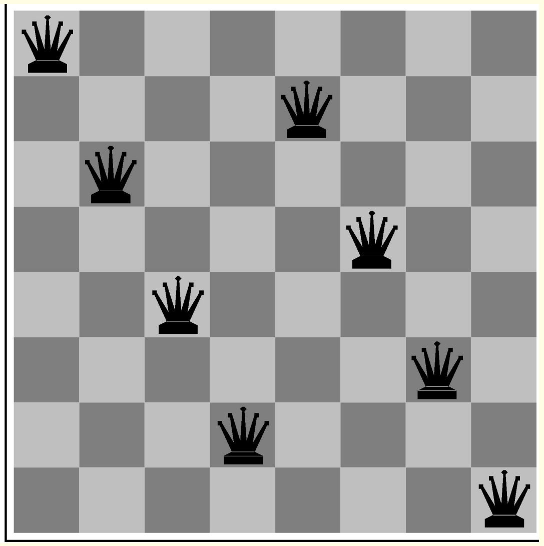
- Each abstract action should be "easier" than the original problem

Vacuum world state space graph



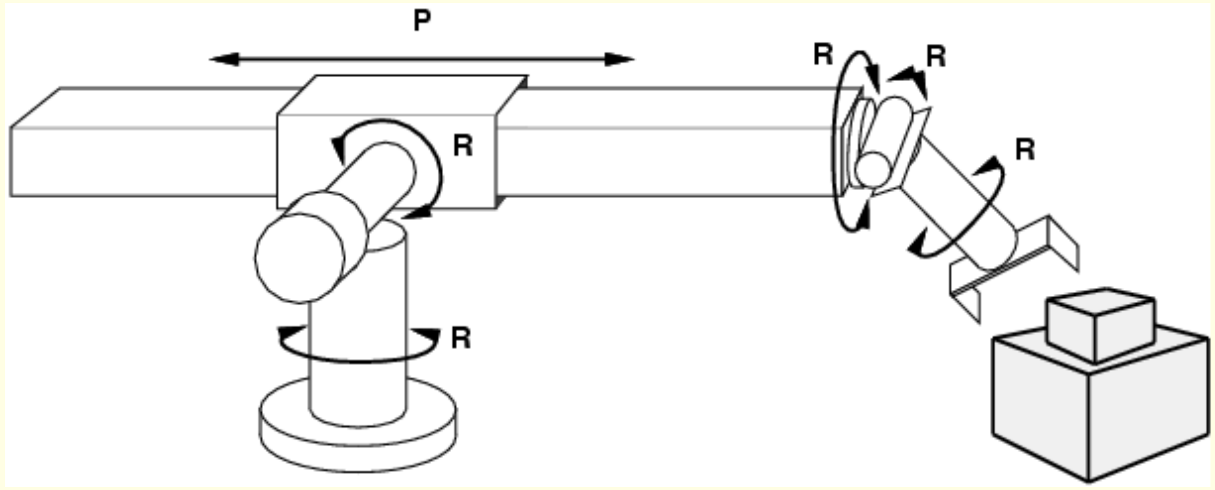
- states? discrete: dirt and robot location
- initial state? any
- actions? *Left*, *Right*, *Suck*
- goal test? no dirt at all locations
- path cost? 1 per action

Example: 8-Queens



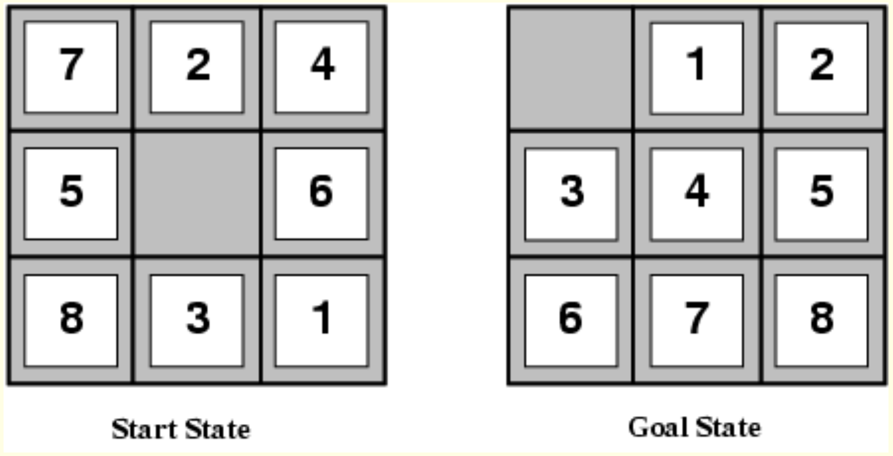
- states? -any arrangement of n<=8 queens
  -*or* arrangements of n<=8 queens in leftmost n columns, 1 per column, such that no queen attacks any other.
- initial state? no queens on the board
- actions? -add queen to any empty square
  -*or* add queen to leftmost empty square such that it is not attacked by other queens.
- goal test? 8 queens on the board, none attacked.
- path cost? 1 per move

Example: robotic assembly



- states?: real-valued coordinates of robot joint angles parts of the object to be assembled
- initial state?: rest configuration
- actions?: continuous motions of robot joints
- goal test?: complete assembly
- path cost?: time to execute+energy used

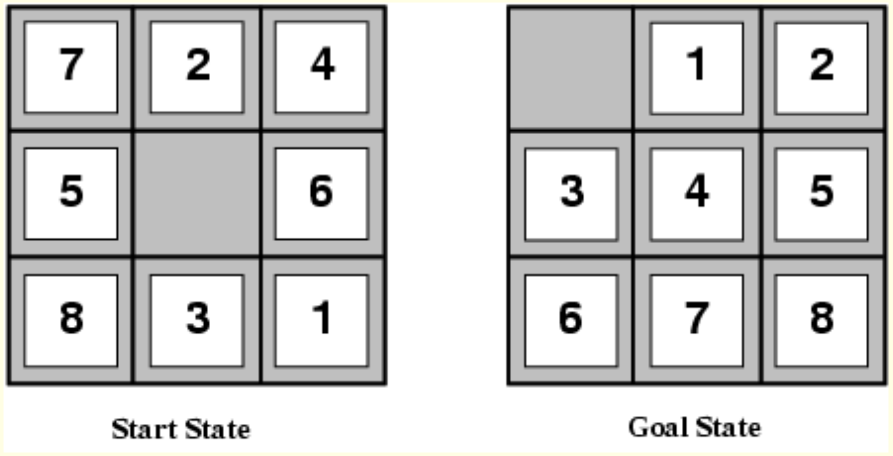
Example: The 8-puzzle



- states?
- initial state?
- actions?
- goal test?
- path cost?

Try yourselves

Example: The 8-puzzle



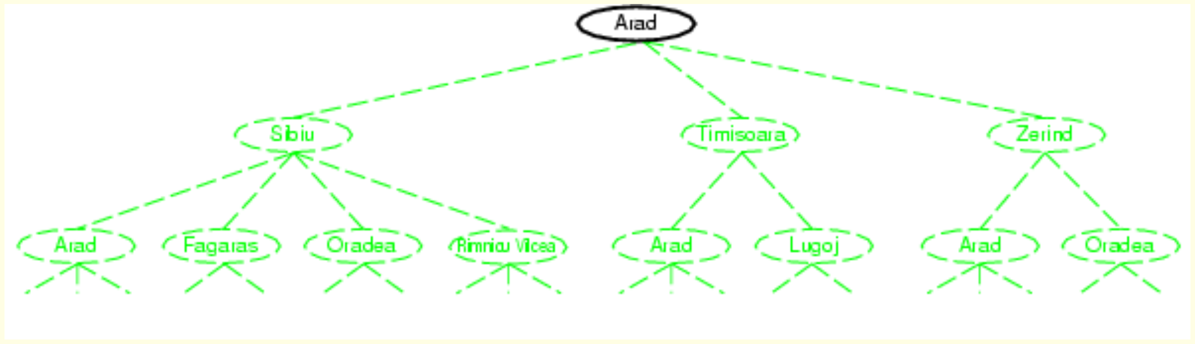
- states? locations of tiles
- initial state? given
- actions? move blank left, right, up, down
- goal test? goal state (given)
- path cost? 1 per move

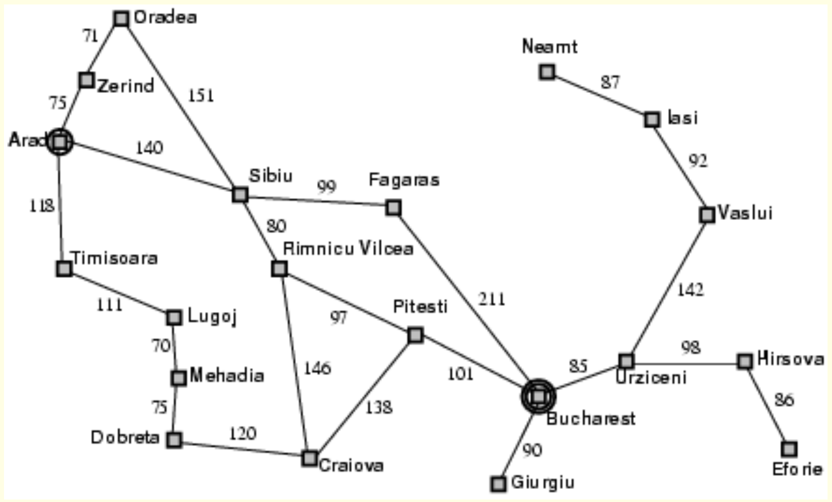
[Note: optimal solution of *n*-Puzzle family is NP-hard]

Tree search algorithms

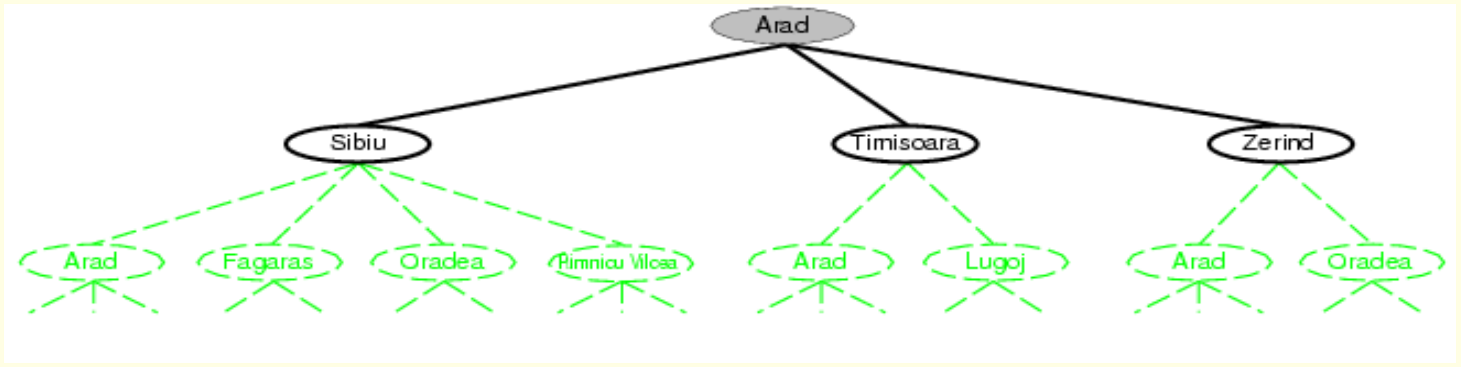
- Basic idea:
  - Exploration of state space by generating successors of already-explored states (a.k.a.~expanding states).
  - Every states is evaluated: *is it a goal state*?

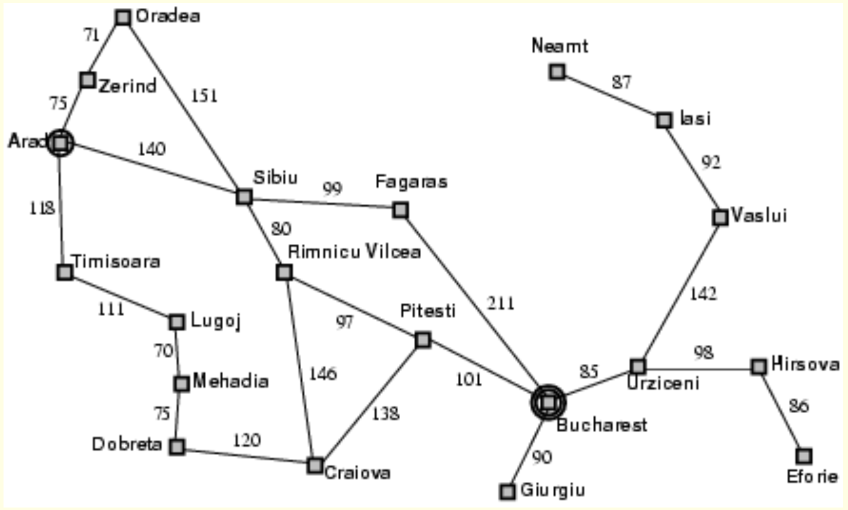
Tree search example



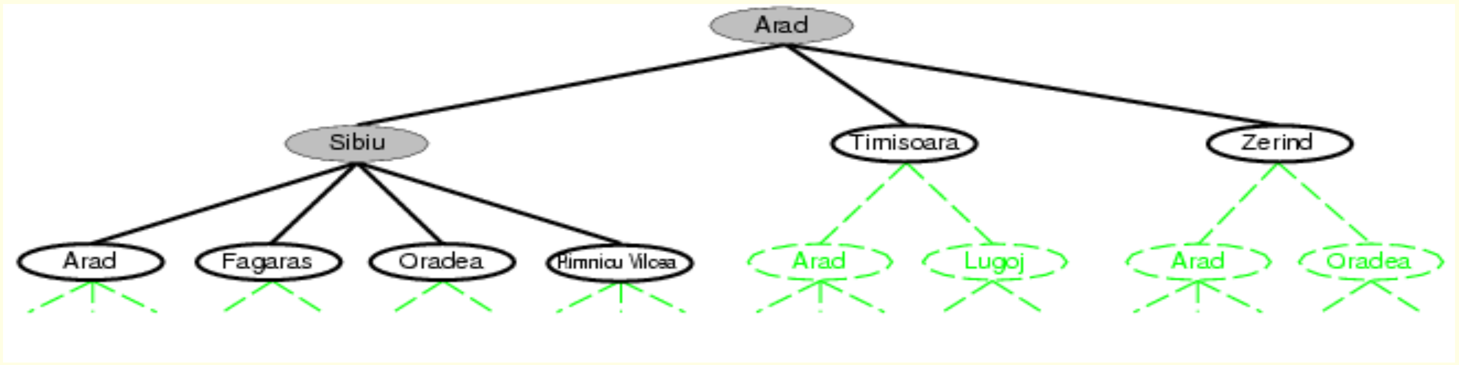


Tree search example





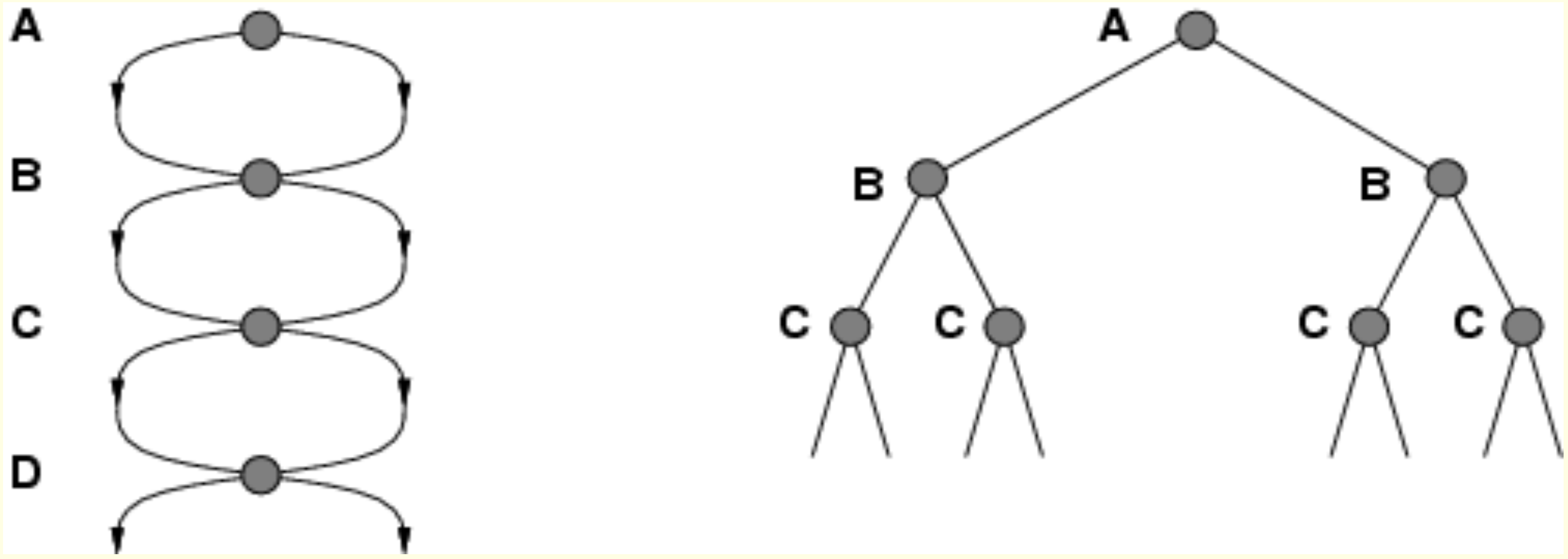
Tree search example



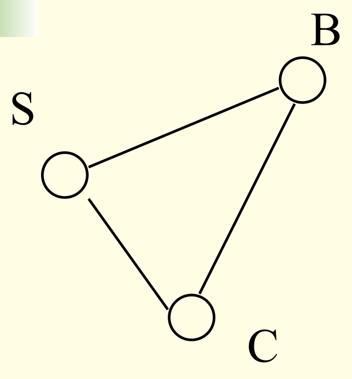
```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the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

Repeated states

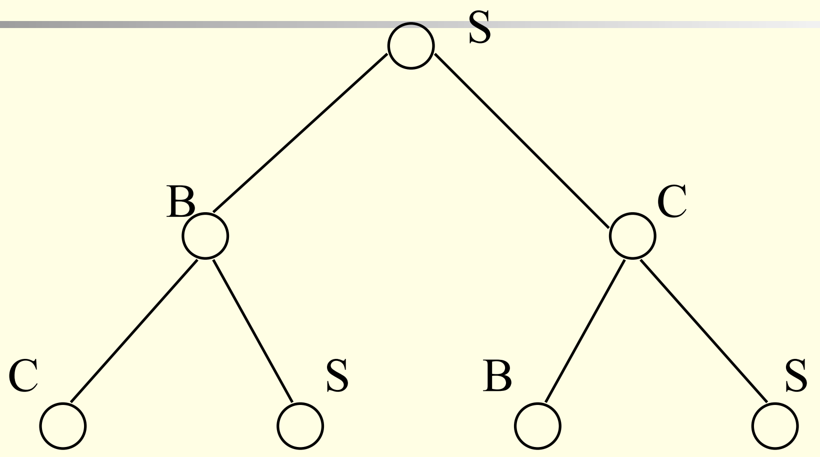
- Failure to detect repeated states can turn a linear problem into an exponential one!



Solutions to Repeated States



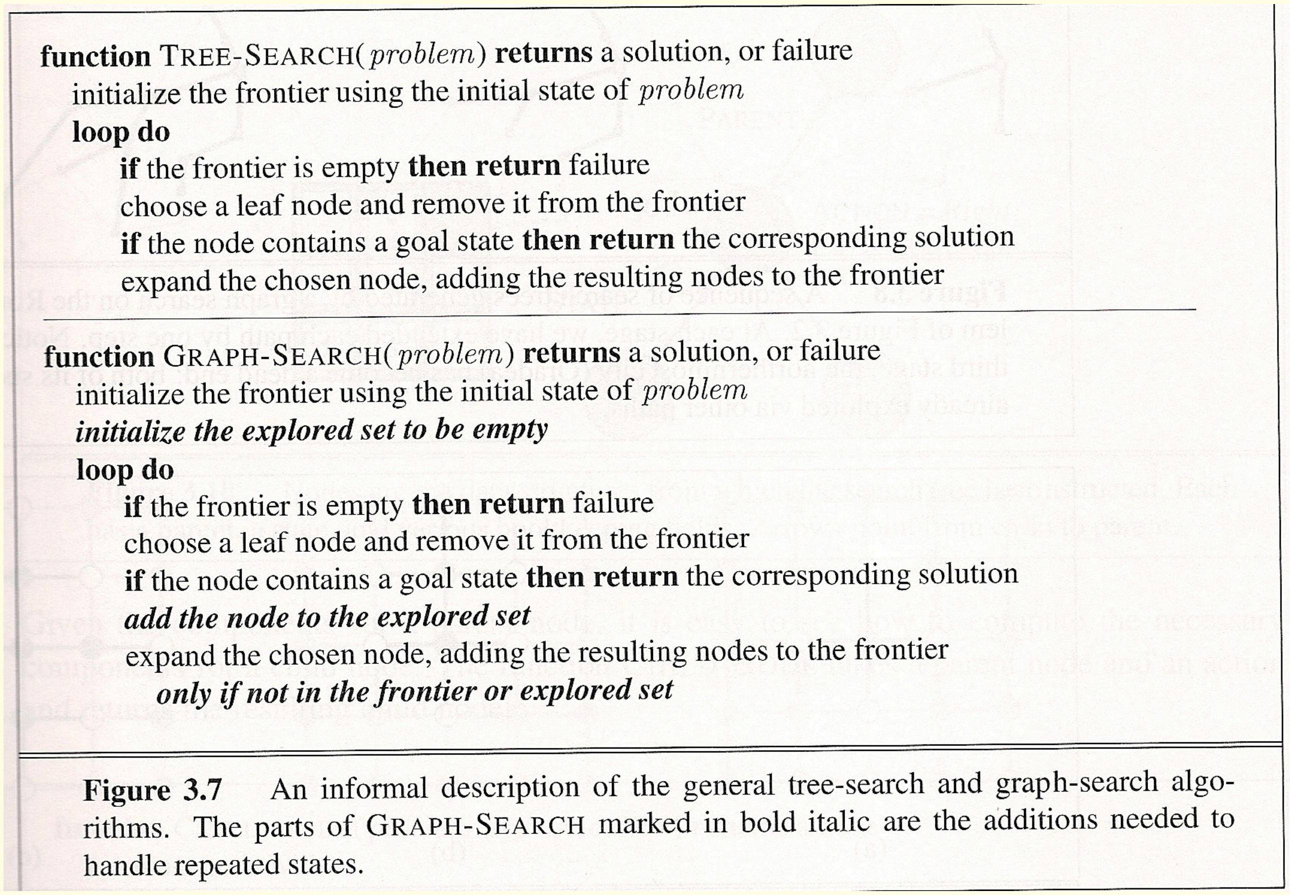
State Space



Example of a Search Tree

- Graph search ← optimal but memory inefficient
  - never generate a state generated before
    - must keep track of all possible states (uses a lot of memory)
    - e.g., 8-puzzle problem, we have 9! = 362,880 states

Graph Search vs Tree 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
```

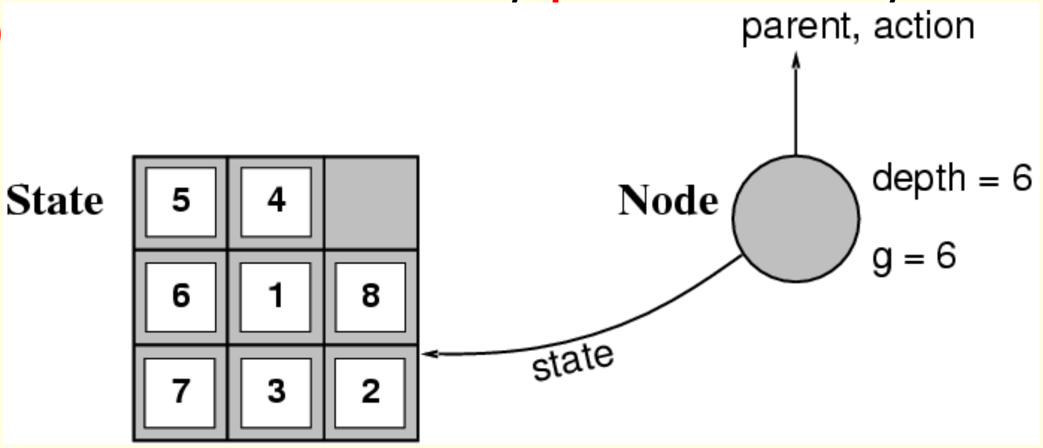
```
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
      if the frontier is empty then return failure
      choose a leaf node and remove it from the frontier
      if the node 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
```

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

(see also blog entry: http://qaintroai.blogspot.com/2009/10/q-how-is-graph-search-different-from.html)

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 contains info such as: state, parent node, action, path cost *g(x)*, dep



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

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?

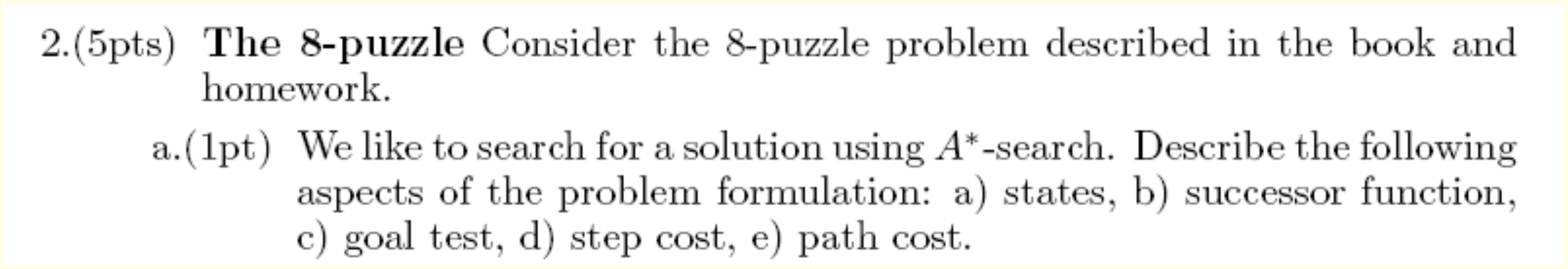
- Time and space complexity are measured in terms of
  - *b:* maximum branching factor of the search tree
  - *d:* depth of the least-cost solution
  - *m*: maximum depth of the state space (may be ∞)

1.(30pts) **Search** Consider a fantasy chess piece called "*jumper*". It can move up, down, left, right, or it can stay wherever it is. Consider k such jumpers on an infinite chessboard at positions $s_1, s_2, ..., s_k$. The goal is to move these jumpers as fast as possible to positions $g_1, g_2, ... g_k$. In each move, you are allowed to move any number of jumpers simultaneously, but 2 or more jumpers can never occupy the same square.

a.(5pts) Formulate the above problem as a search problem, i.e. describe a state, the initial state, an action, the goal test and a path-cost.

b.(5pts) Set $k = 1$, i.e. a single jumper on the board. What is the maximal branching factor for this problem?

d.(5pts) Now consider general k. What is now the maximal branching factor? Remember that up to k jumpers can move simultaneously.



2.(5pts) **The 8-puzzle** Consider the 8-puzzle problem described in the book and homework.

a.(1pt) We like to search for a solution using A^*-search. Describe the following aspects of the problem formulation: a) states, b) successor function, c) goal test, d) step cost, e) path cost.

Uninformed Search

Complexity Recap (app.A)

- We often want to characterize algorithms independent of their implementation.

- *"This algorithm took 1 hour and 43 seconds on my laptop".*
Is not very useful, because tomorrow computers are faster.

- Better is:
"This algorithm takes O(nlog(n)) time to run and O(n) to store".
because this statement abstracts away from irrelevant details.

Time(n) = O(f(n)) means:
Time(n) < constant x f(n) for n>n0 for some n0
Space(n) idem.

n is some variable which characterizes the size of the problem,
e.g. number of data-points, number of dimensions, branching-factor of search tree, etc.

- Worst case analysis versus average case analyis

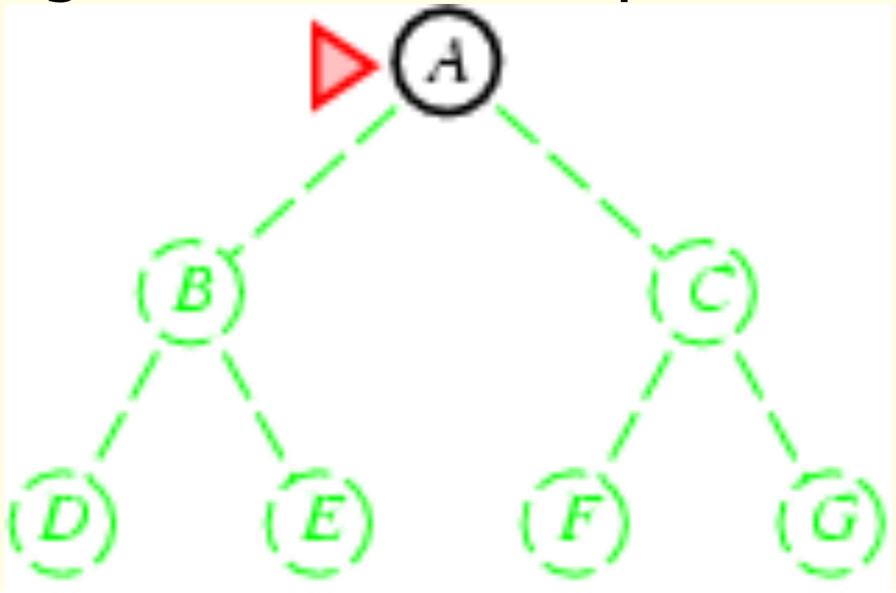
Uninformed search strategies

- Uninformed: While searching you have no clue whether one non-goal state is better than any other. Your search is blind. You don’t know if your current exploration is likely to be fruitful.
- Various blind strategies:
- Breadth-first search
- Uniform-cost search
- Depth-first search
- Iterative deepening search

Breadth-first search

- Expand shallowest unexpanded node
- Fringe: nodes waiting in a queue to be explored
- Implementation:
  - *fringe* is a first-in-first-out (FIFO) queue, i.e., new successors go at end of the queue.

Is A a goal state?

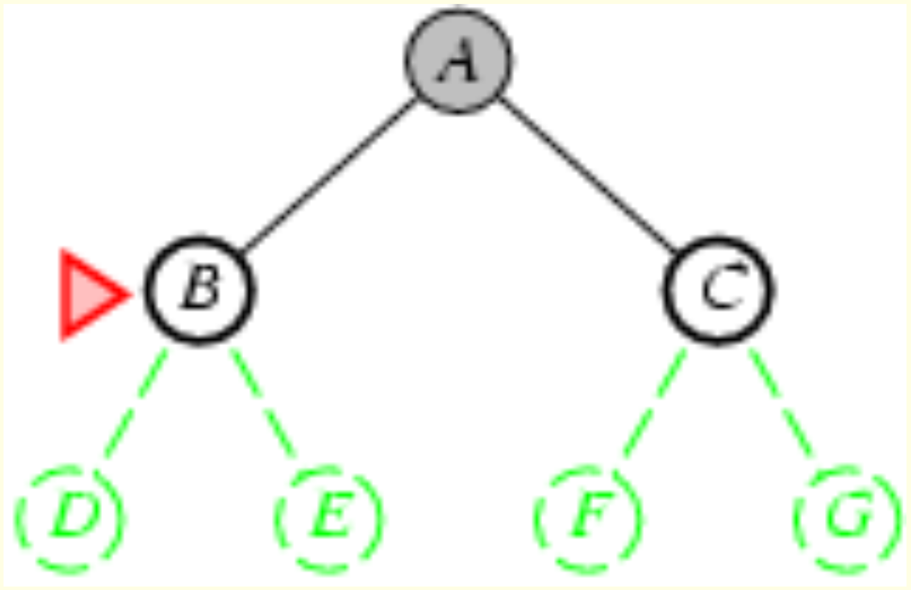


Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end

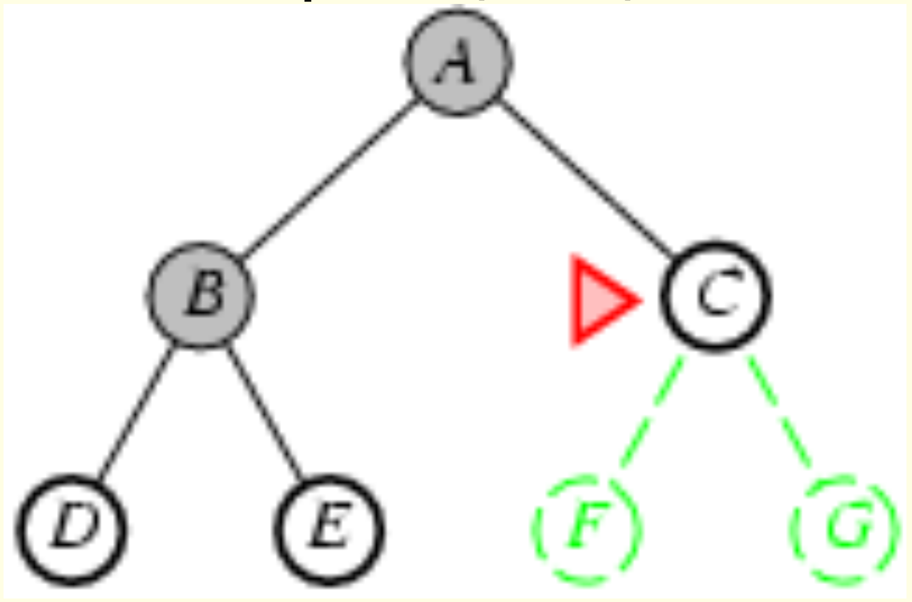
Expand:
fringe = [B,C]

Is B a goal state?



Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



Expand:
fringe=[C,D,E]

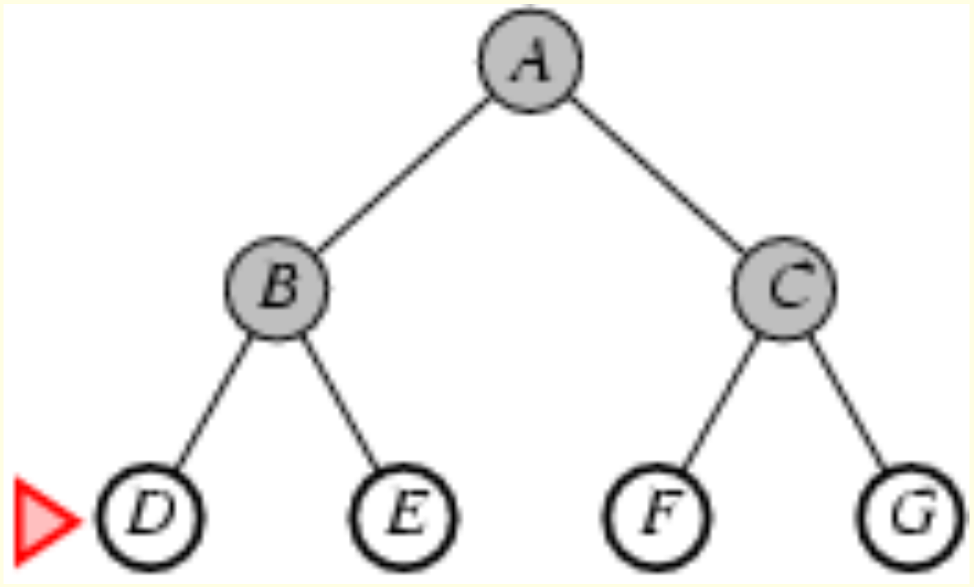
Is C a goal state?

Breadth-first search

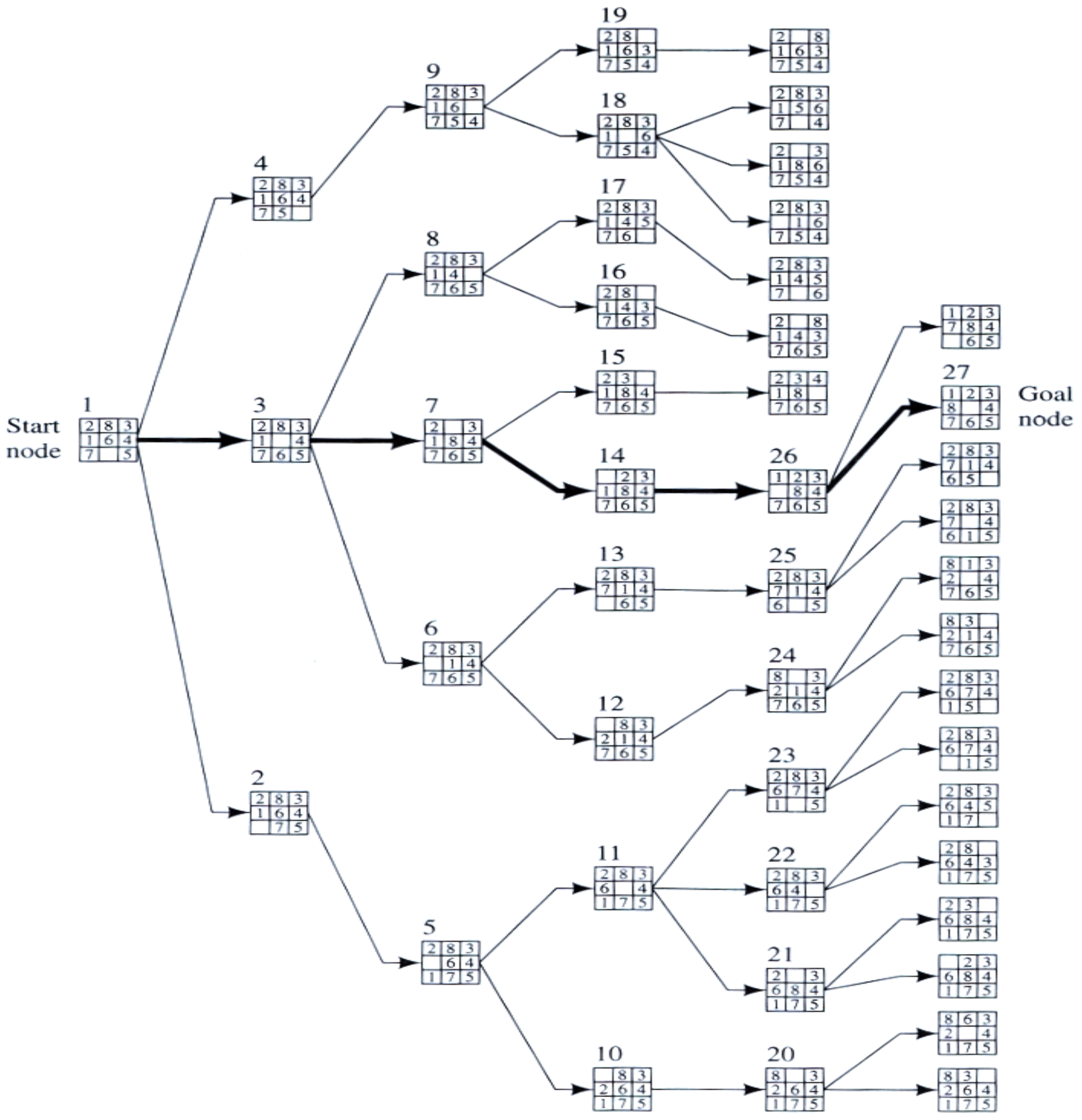
- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end

Expand:
fringe=[D,E,F,G]

Is D a goal state?



Example BFS



Start node
Goal node
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27

Properties of breadth-first search

- Complete? Yes it always reaches goal (if *b* is finite)
- Time? $1+b+b^2+b^3+\ldots+b^d + (b^{d+1}-b)) = O(b^{d+1})$
  (this is the number of nodes we generate)
- Space? $O(b^{d+1})$ (keeps every node in memory, either in fringe or on a path to fringe).
- Optimal? Yes (if we guarantee that deeper solutions are less optimal, e.g. step-cost=1).

- Space is the bigger problem (more than time)

Note: in the new edition Space & Time complexity was $O(b^d)$ because we postpone the expansion.

Uniform-cost search

Breadth-first is only optimal if step costs is increasing with depth (e.g. constant). Can we guarantee optimality for any step cost?

Uniform-cost Search: Expand node with smallest path cost g(n).

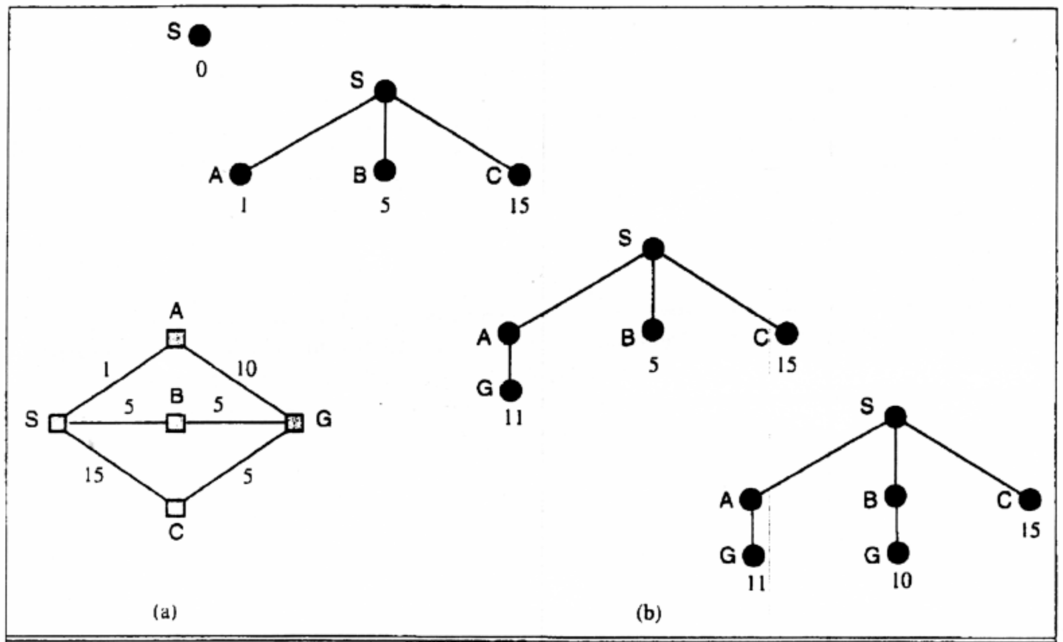


Figure 3.13 A route-finding problem. (a) The state space, showing the cost for each operator. (b) Progression of the search. Each node is labelled with $g(n)$. At the next step, the goal node with $g = 10$ will be selected.

Proof Completeness:

Given that every step will cost more than 0, and assuming a finite branching factor, there is a finite number of expansions required before the total path cost is equal to the path cost of the goal state. Hence, we will reach it.

Proof of optimality given completeness:

Assume UCS is not optimal.
Then there must be an (optimal) goal state with path cost smaller than the found (suboptimal) goal state (invoking completeness).
However, this is impossible because UCS would have expanded that node first by definition.
Contradiction.

Uniform-cost search

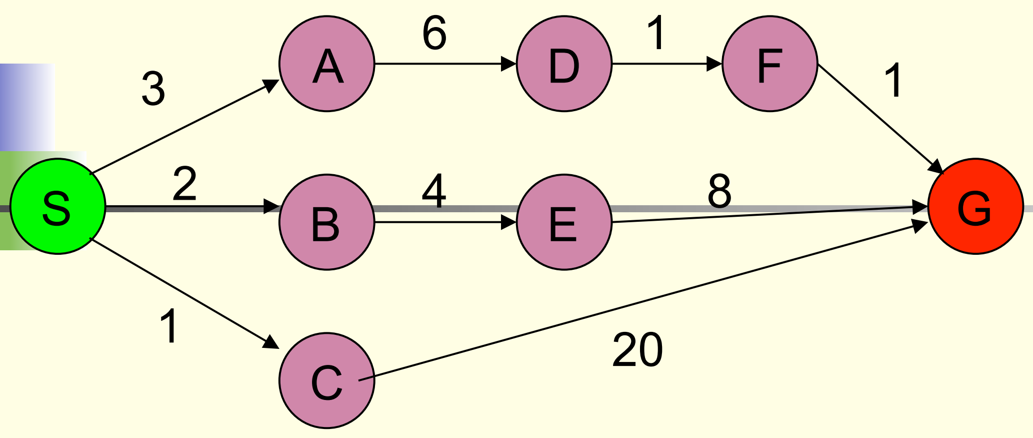
Implementation: *fringe* = queue ordered by path cost
Equivalent to breadth-first if all step costs all equal.

Complete? Yes, if step cost ≥ ε
(otherwise it can get stuck in infinite loops)

Time? # of nodes with *path cost* ≤ cost of optimal solution.

Space? # of nodes with path cost ≤ cost of optimal solution.

Optimal? Yes, for any step cost ≥ ε



The graph above shows the step-costs for different paths going from the start (S) to the goal (G).

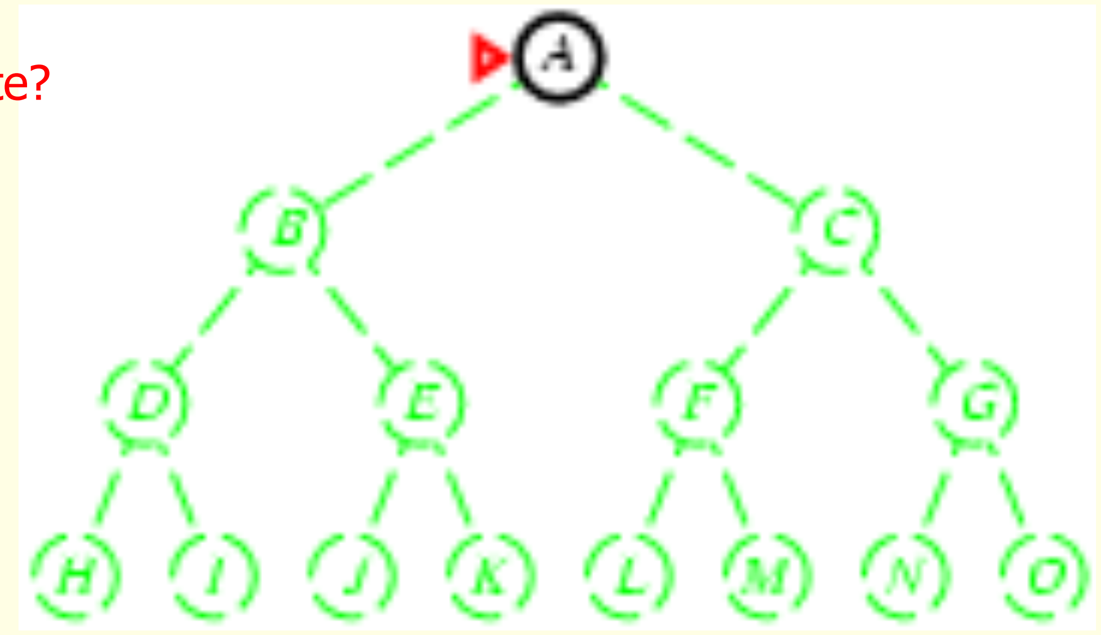
Use uniform cost search to find the optimal path to the goal.

Exercise

Depth-first search

- Expand *deepest* unexpanded node
- Implementation:
  - *fringe* = Last In First Out (LIPO) queue, i.e., put successors at front

Is A a goal state?

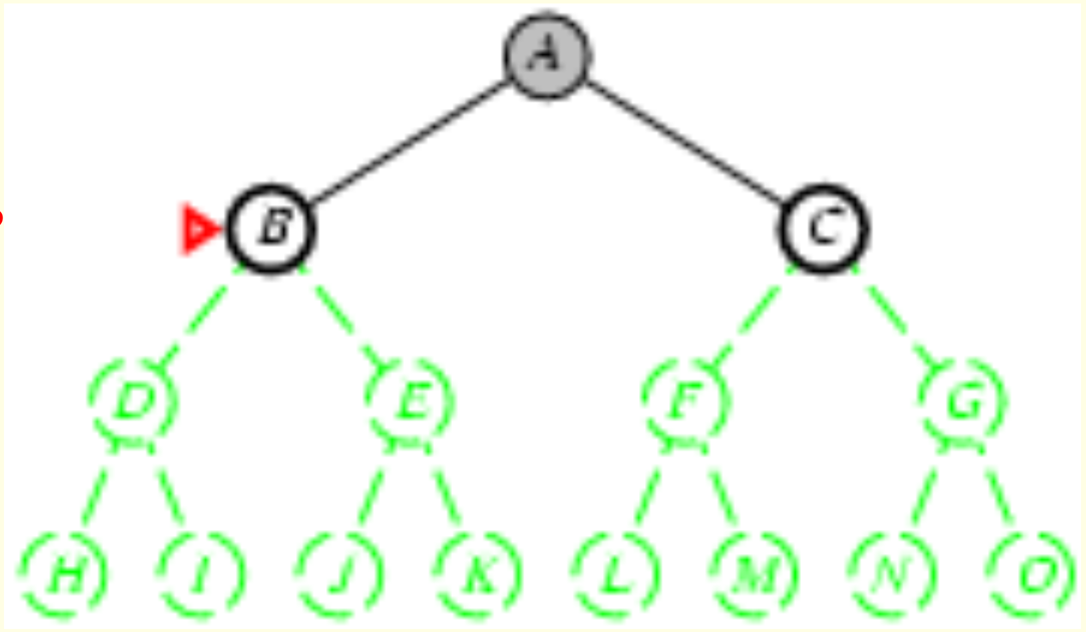


Depth-first search

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

queue=[B,C]

Is B a goal state?

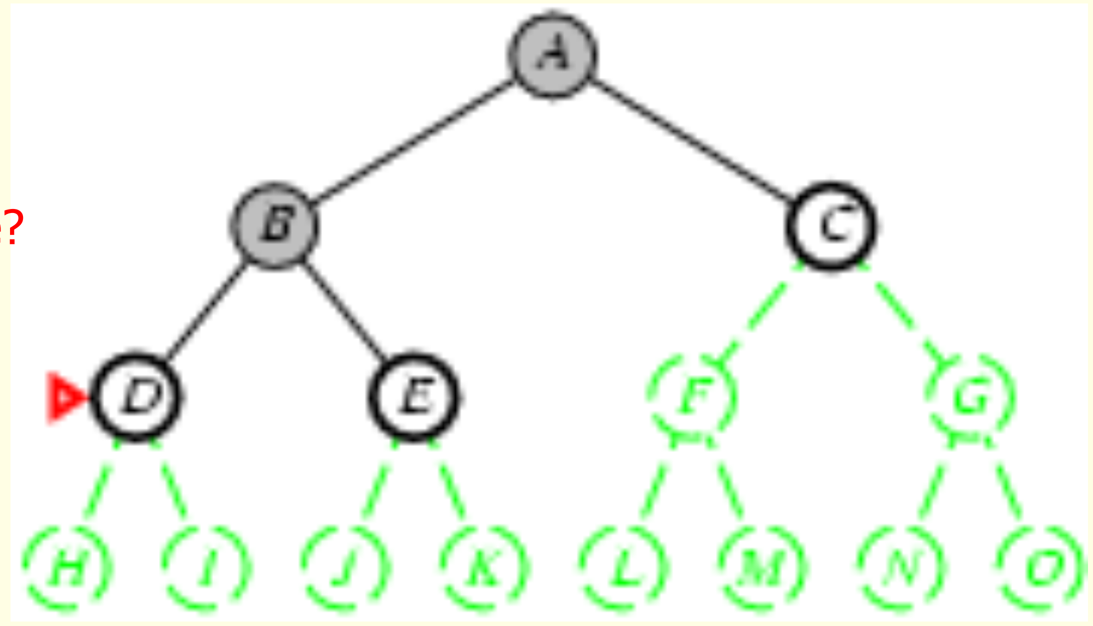


Depth-first search

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

queue=[D,E,C]

Is D = goal state?

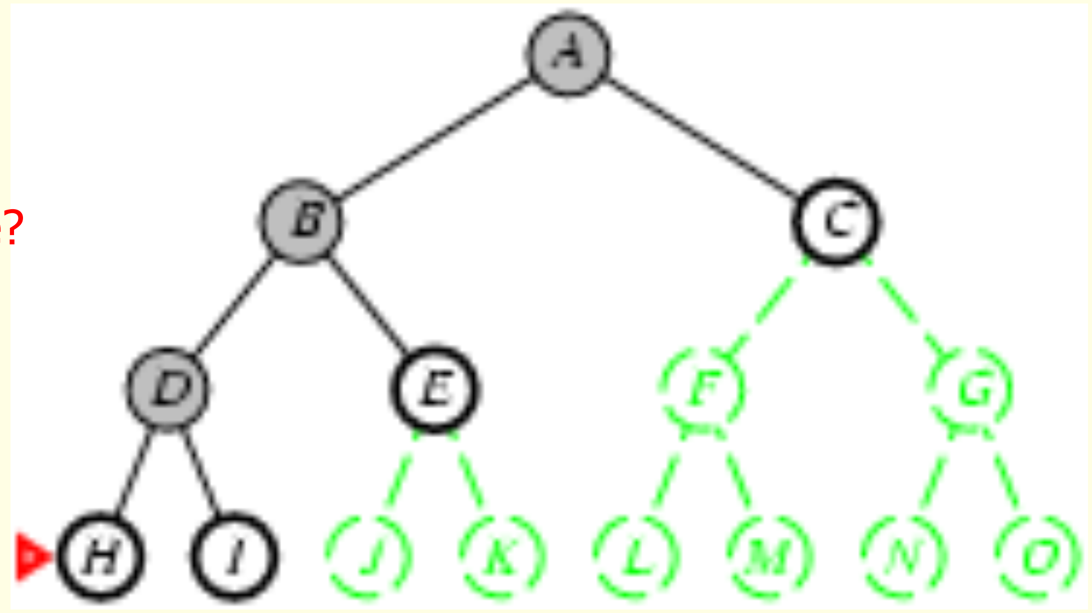


Depth-first search

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

queue=[H,I,E,C]

Is H = goal state?

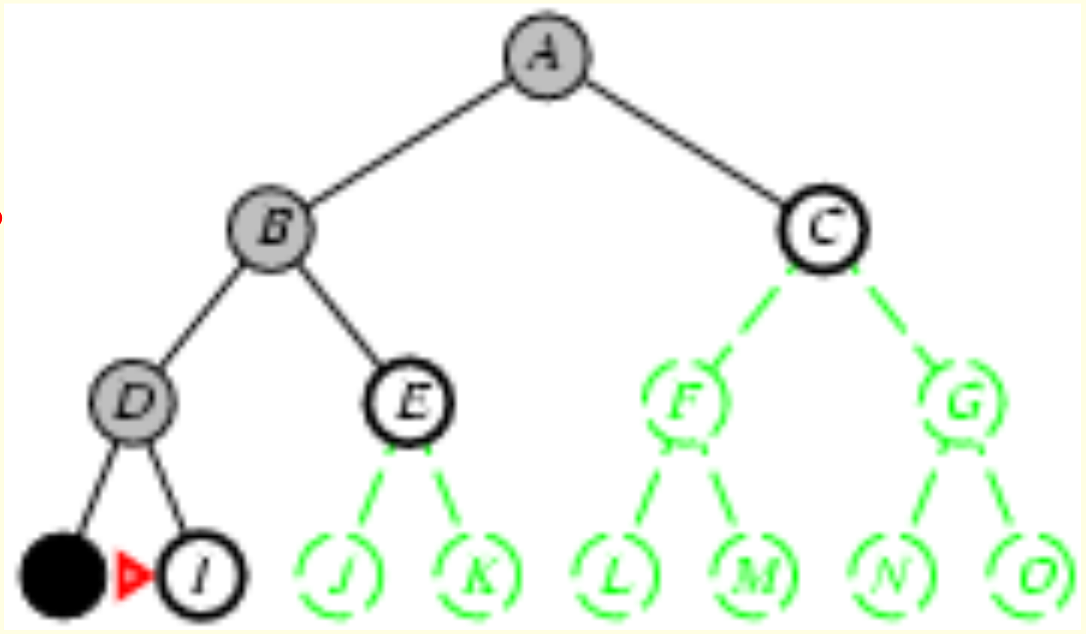


Depth-first search

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

queue=[I,E,C]

Is I = goal state?

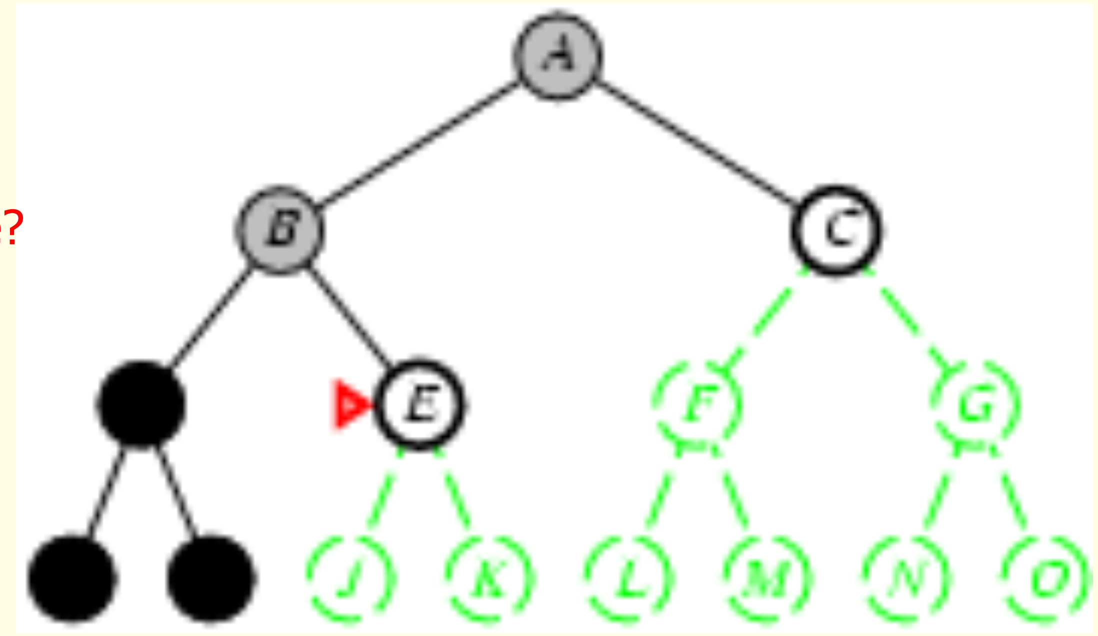


Depth-first search

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

queue=[E,C]

Is E = goal state?

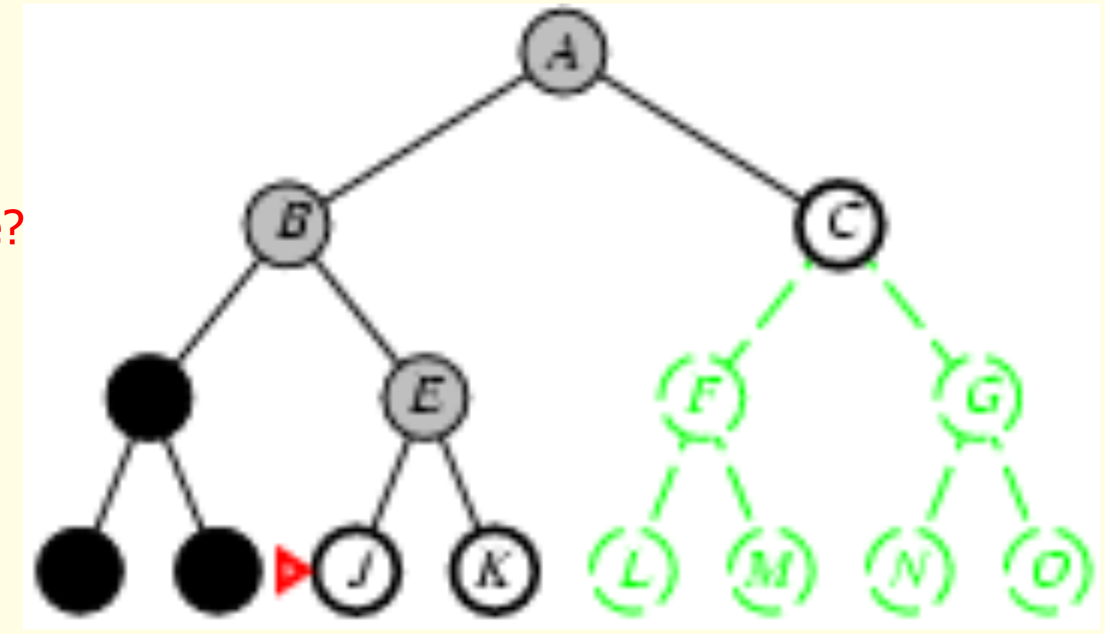


Depth-first search

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

queue=[J,K,C]

Is J = goal state?

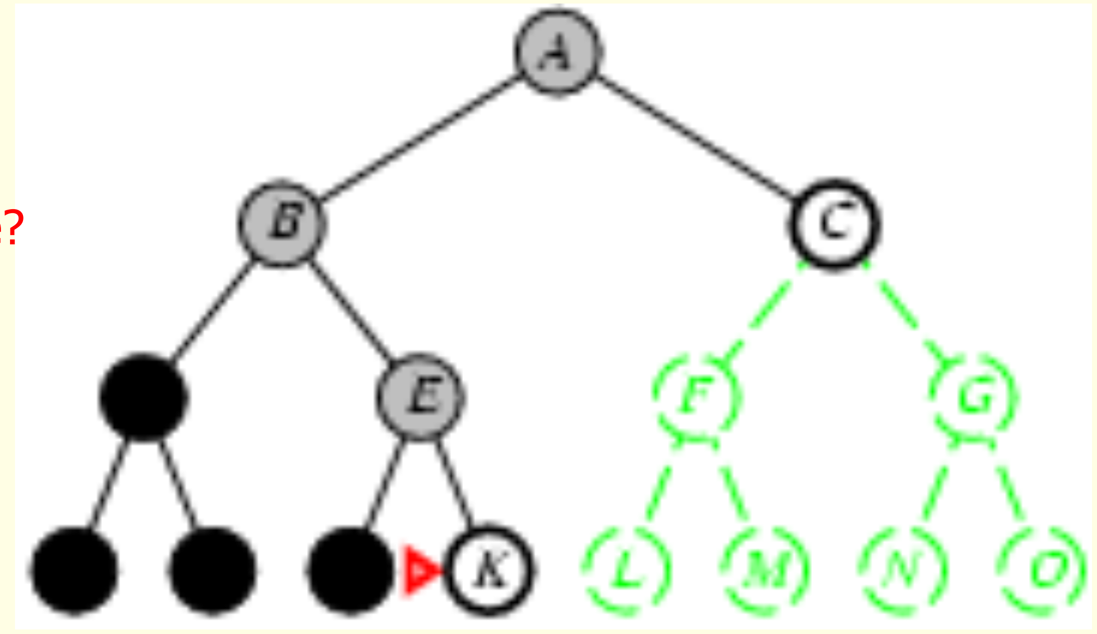


Depth-first search

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

queue=[K,C]

Is K = goal state?

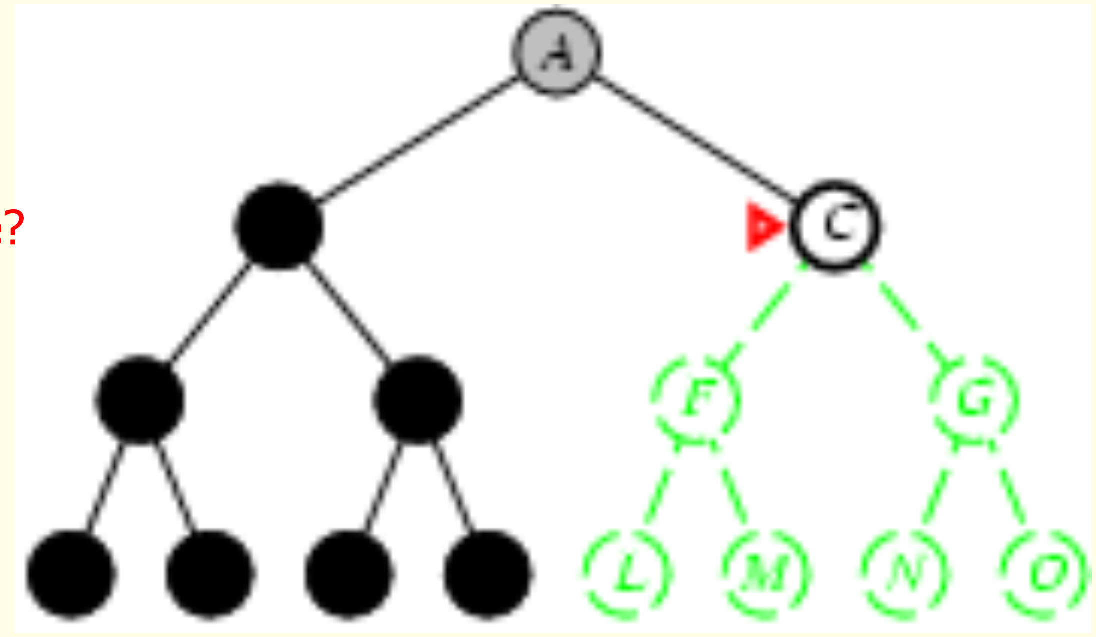


Depth-first search

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

queue=[C]

Is C = goal state?

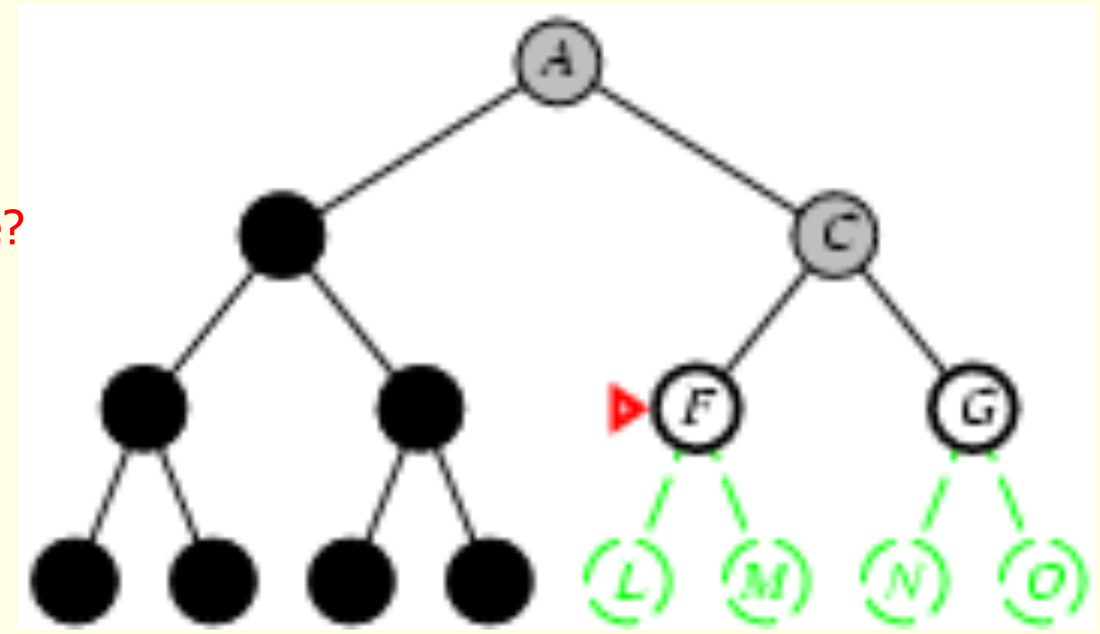


Depth-first search

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

queue=[F,G]

Is F = goal state?

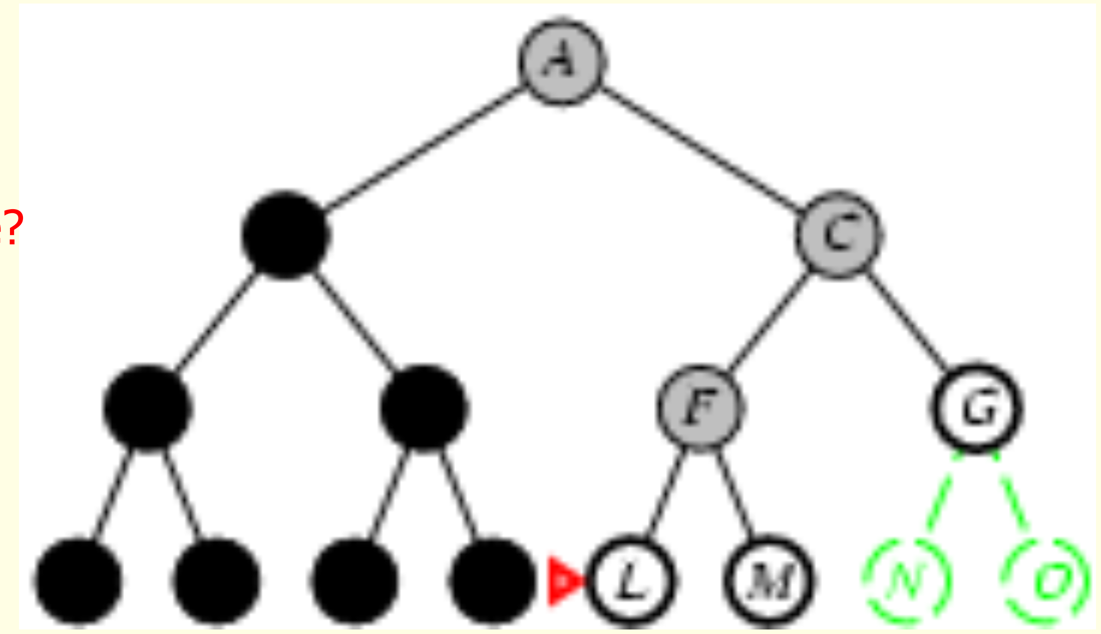


Depth-first search

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

queue=[L,M,G]

Is L = goal state?

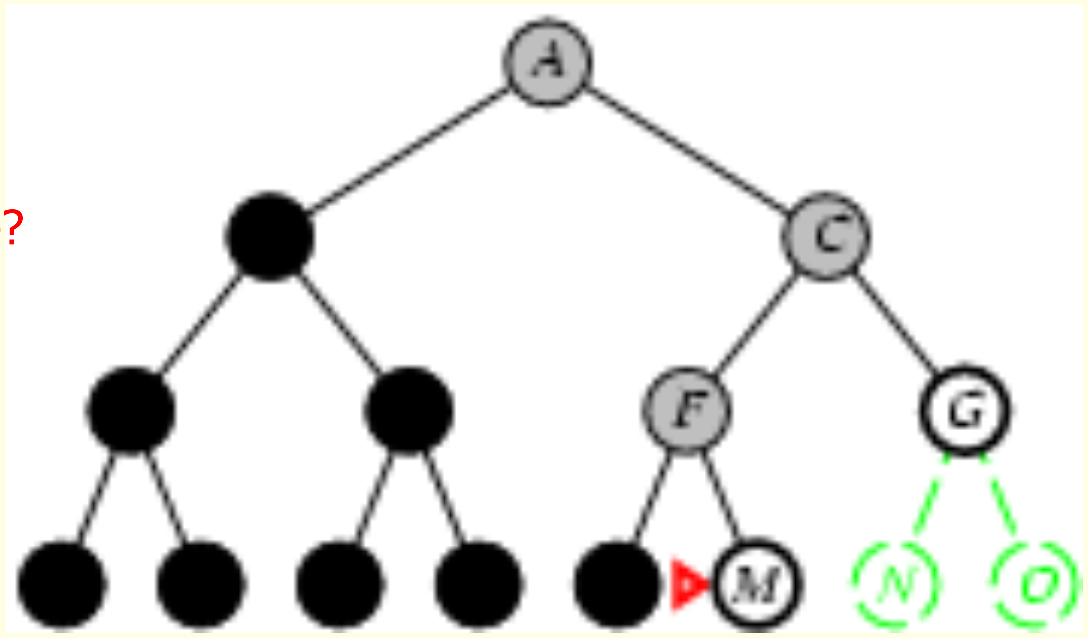


Depth-first search

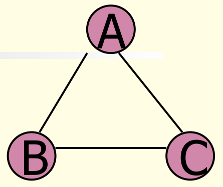
- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

queue=[M,G]

Is M = goal state?



Properties of depth-first search



- Complete? No: fails in infinite-depth spaces

  Can modify to avoid repeated states along path
- Time? $O(b^m)$ with m=maximum depth
- 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! (we only need to remember a single path + expanded unexplored nodes)
- Optimal? No (It may find a non-optimal goal first)

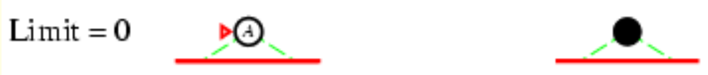
Iterative deepening search

- To avoid the infinite depth problem of DFS, we can decide to only search until depth L, i.e. we don't expand beyond depth L.
  → Depth-Limited Search

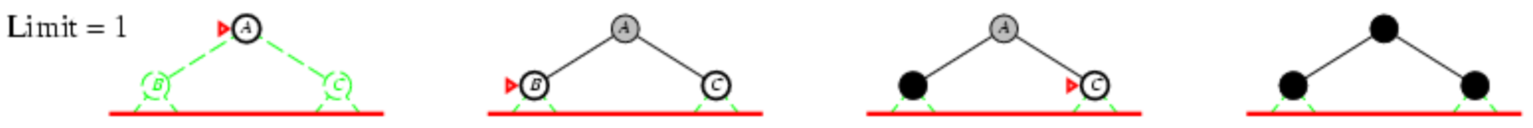
- What if solution is deeper than L? → Increase L iteratively.
  → Iterative Deepening Search

- As we shall see: this inherits the memory advantage of Depth-First search, and is better in terms of time complexity than Breadth first search.

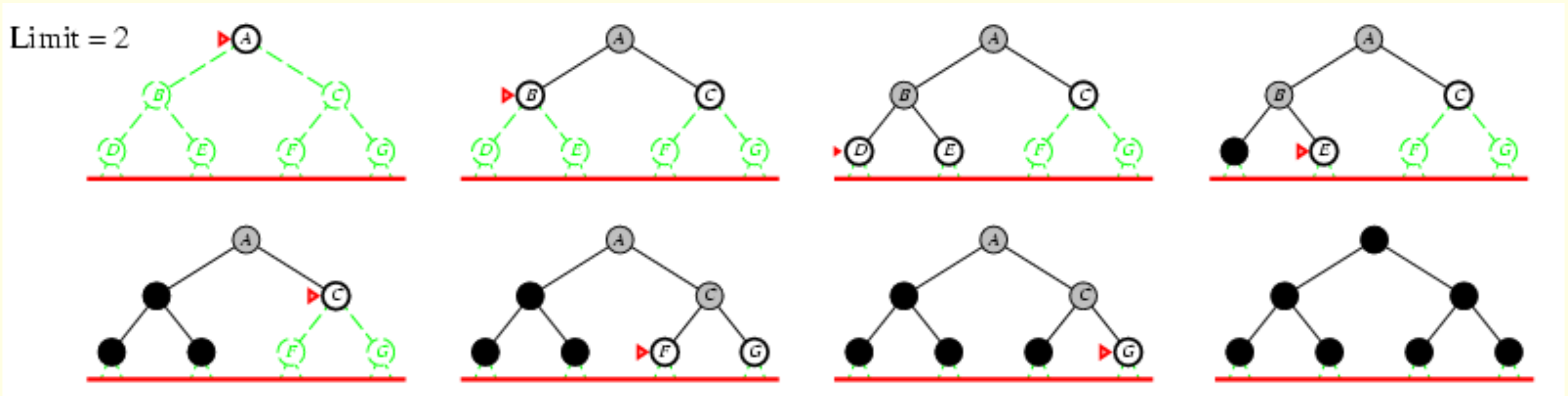
Iterative deepening search *L*=0



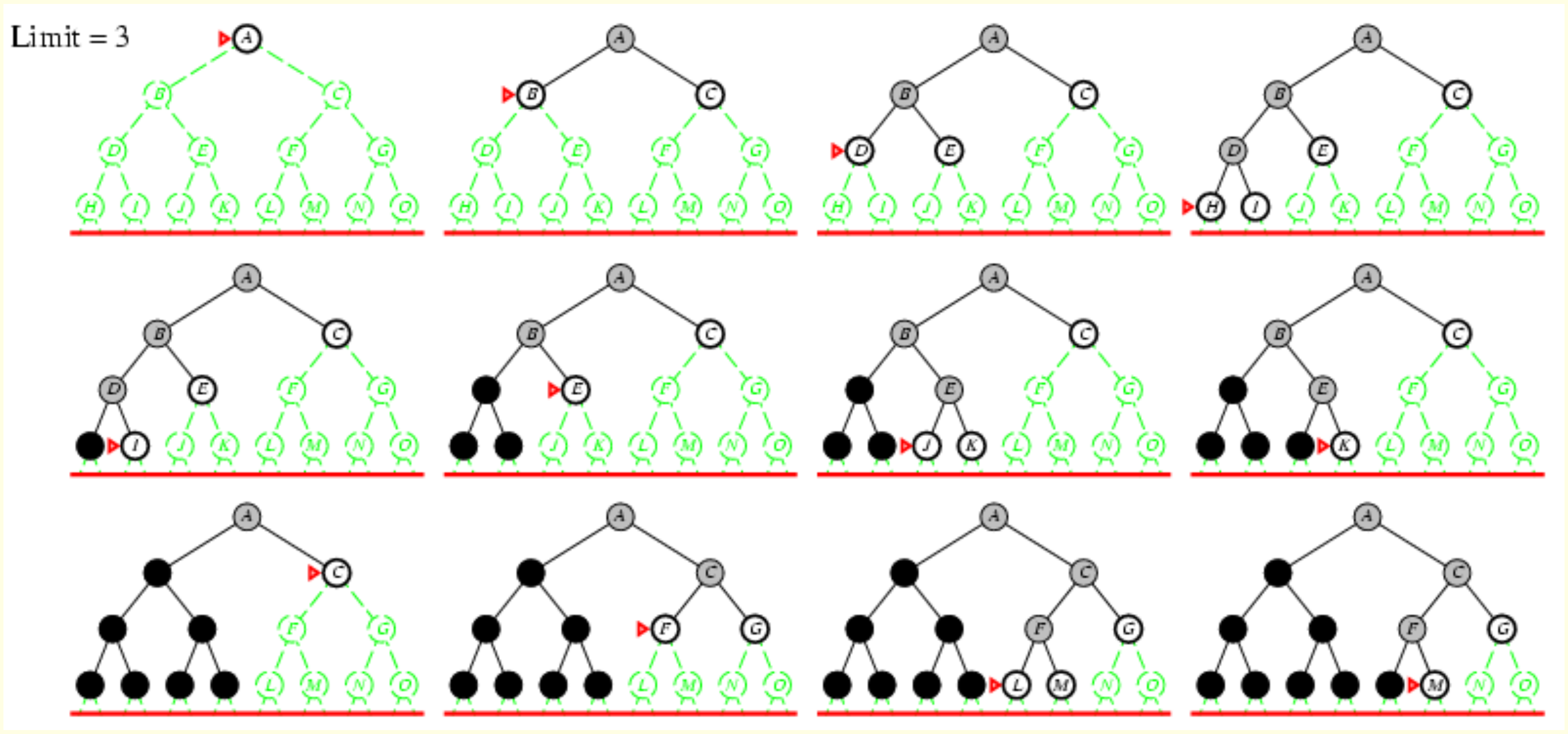
Iterative deepening search *L*=1



Iterative deepening search L=2



Iterative Deepening Search *L*=3



Iterative deepening search

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

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

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

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

$$O(b^d) \neq O(b^{d+1})$$

BFS

- 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 = 123,450
  - NBFS = ... = 1,111,100

Note: BFS can also be adapted to be $O(b^d)$ by waiting to expand until all nodes at depth d are checked

Properties of iterative deepening search

- Complete? Yes
- Time? $O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, if step cost = 1 or increasing function of depth.

Bidirectional Search

- Idea
  - simultaneously search forward from S and backwards from G
  - stop when both “meet in the middle”
  - need to keep track of the intersection of 2 open sets of nodes
- What does searching backwards from G mean
  - need a way to specify the predecessors of G
    - this can be difficult,
    - e.g., predecessors of checkmate in chess?
  - which to take if there are multiple goal states?
  - where to start if there is only a goal test, no explicit list?

Bi-Directional Search

Complexity: time and space complexity are: $O(b^{d/2})$

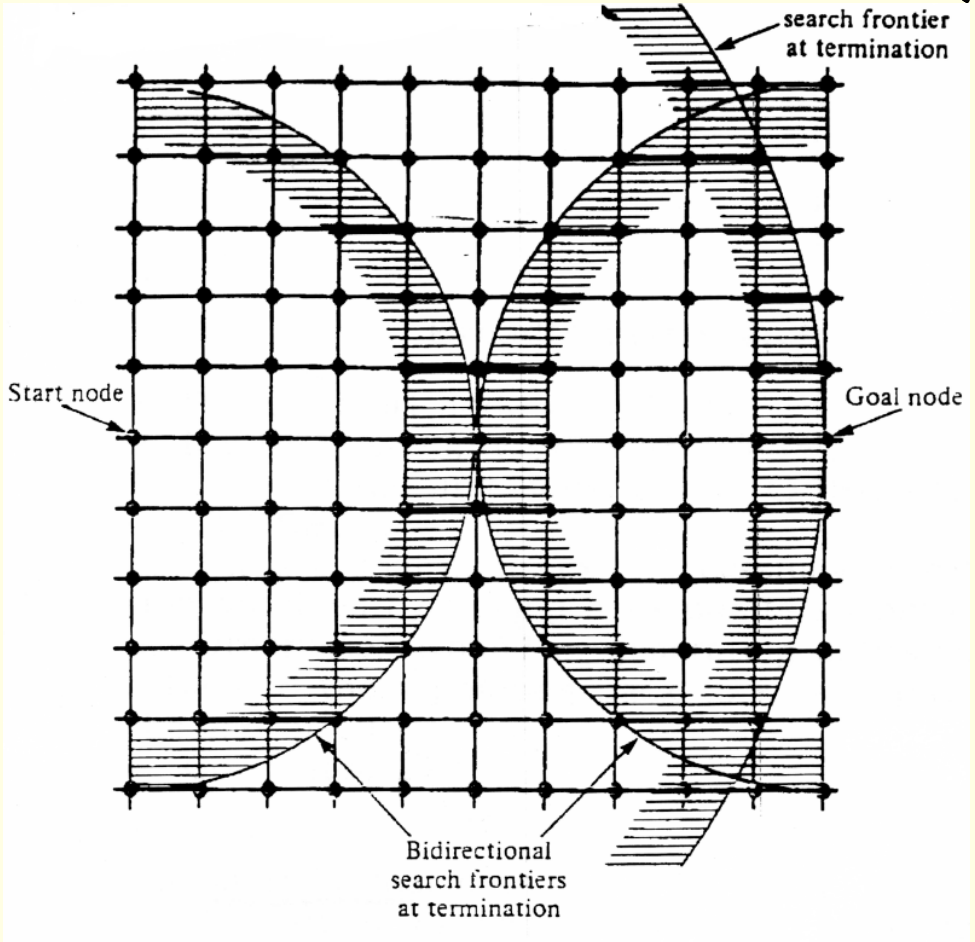
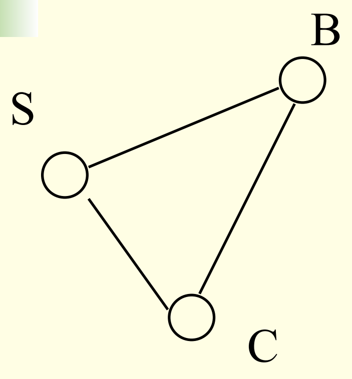
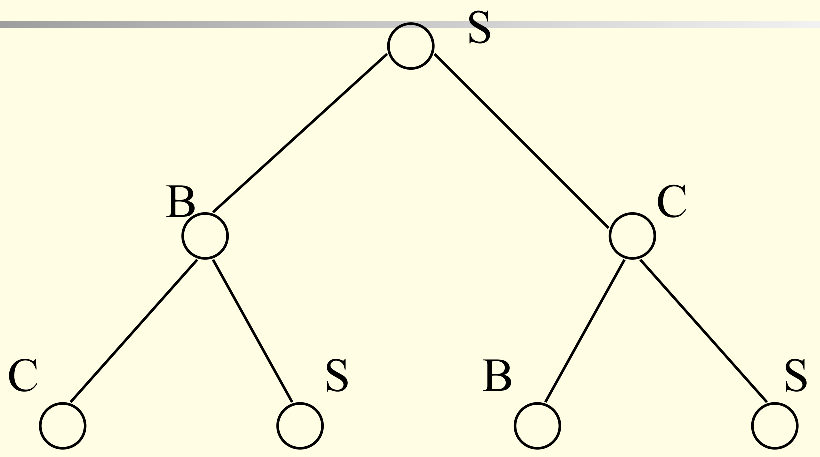

search frontier at termination
Start node
Goal node
Bidirectional search frontiers at termination

Fig. 2.10 Bidirectional and unidirectional breadth-first searches.

Graph Search vs Tree Search



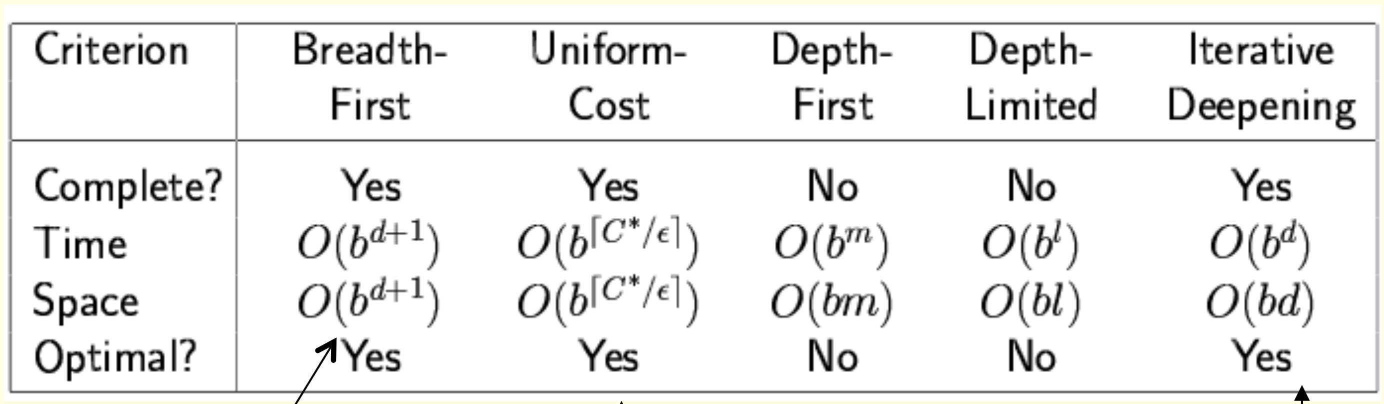
State Space



Example of a Search Tree

- Graph search ← optimal but memory inefficient
  - never generate a state generated before
    - must keep track of all possible states (uses a lot of memory)
    - e.g., 8-puzzle problem, we have 9! = 362,880 states
    - approximation for DFS/DLS: only avoid states in its (limited) memory: avoid looping paths.
    - Graph search optimal for BFS and UCS *(do you understand why?)*

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

Third edition has $O(b^d)$

even complete if step cost is not increasing with depth.

preferred uninformed search strategy

Summary

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

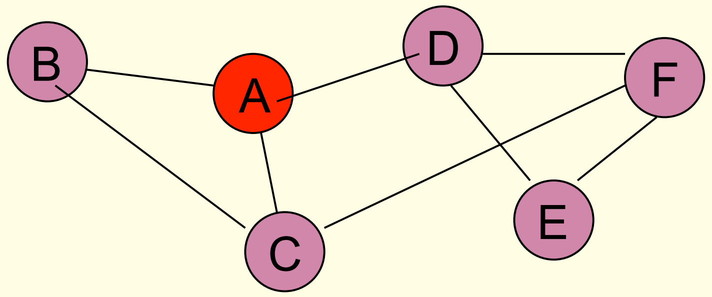
- Variety of uninformed search strategies

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

http://www.cs.rmit.edu.au/AI-Search/Product/
http://aima.cs.berkeley.edu/demos.html (for more demos)

Exercise

2. Consider the graph below:



a) [2pt] Draw the first 3 levels of the full search tree with root node given by A. Use graph search, i.e. avoid repeated states.

b) [2pt] Give an order in which we visit nodes if we search the tree breadth first.

c) [2pt] Express time and space complexity for general breadth-first search in terms of the branching factor, b, and the depth of the goal state, d.

d) [2pt] If the step-cost for a search problem is *not* constant, is breadth first search always optimal? Is BFS graph search optimal?

e) [2pt] Now assume the constant step-cost. Is BSF tree search optimal? Is BFS graph search optimal?

Next time

Questions?