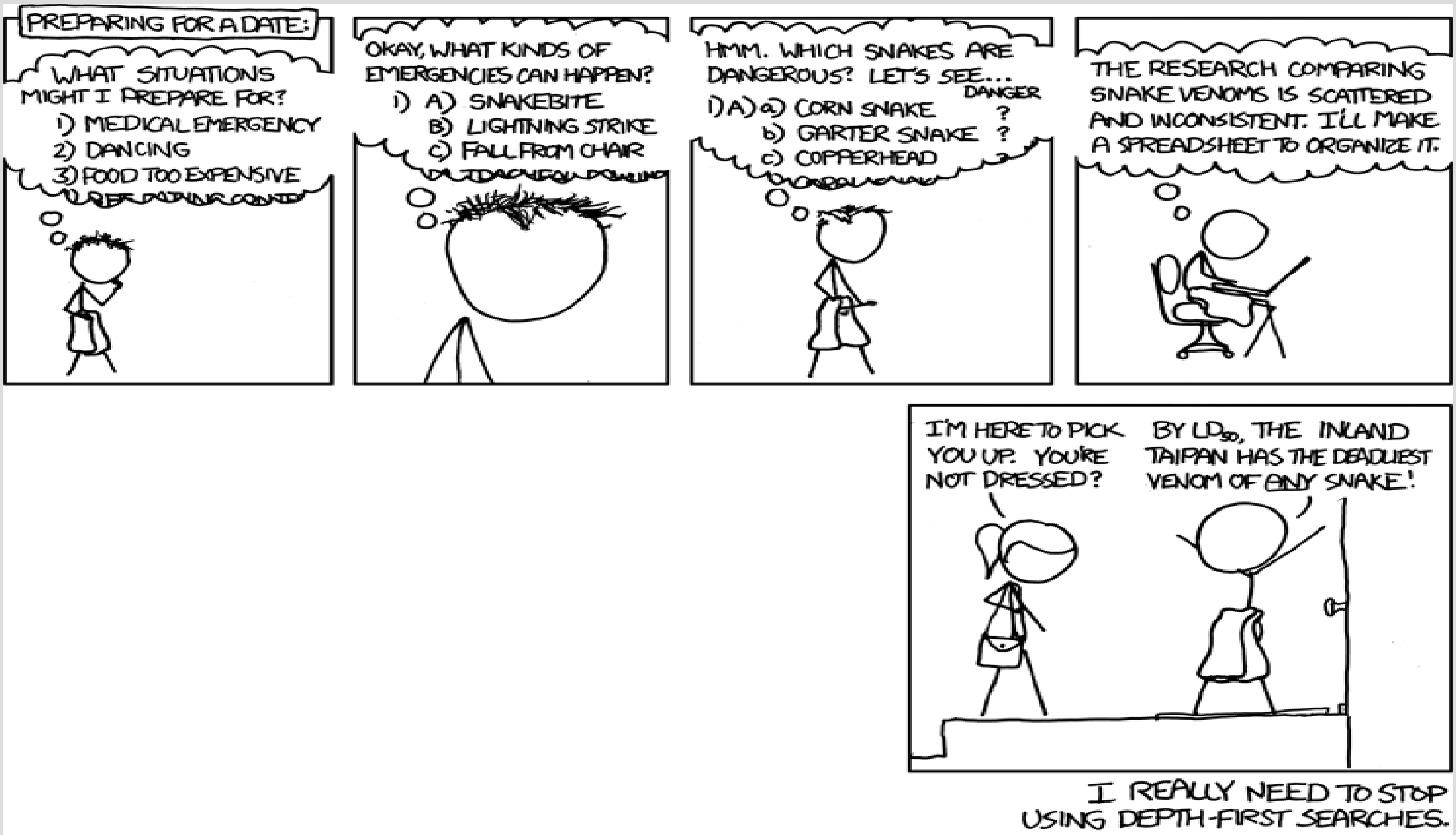
Uninformed Search (Ch. 3-3.4)



Terminology review

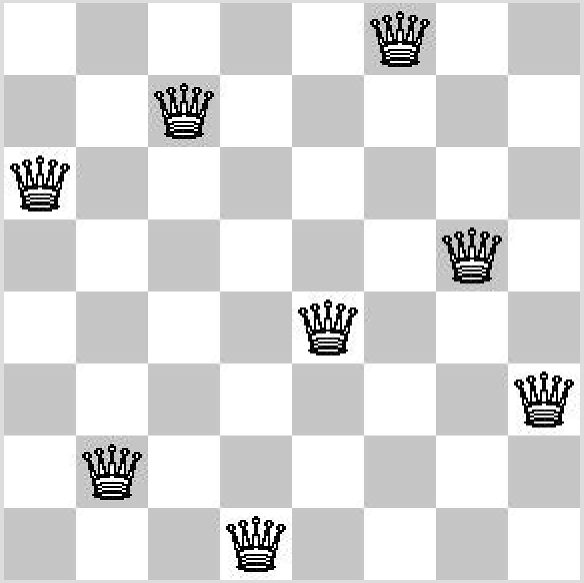
State: a representation of a possible configuration of our problem

Action:
-how our agent interacts with the problem
-can be different depending on which state

Actions and states fundamentally connected (actions tell how to move between states)
E.g. Result(S,a)=S', with S&S' states, a=action

Small examples

8-Queens: how to fit 8 queens on a 8x8 board so no 2 queens can capture each other



Two ways to model this:

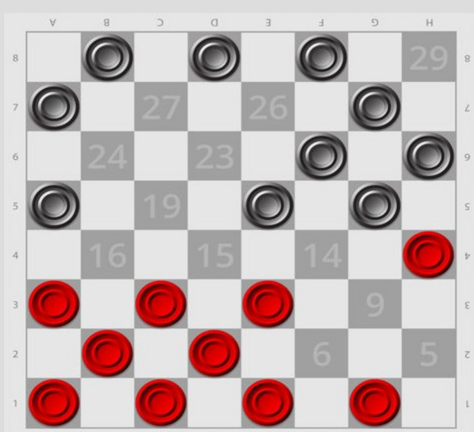
Incremental = each action is to
  add a queen to the board
  (1.8 x 10^{14} states)

Complete state formulation = all 8 queens start
  on board, action = move a queen
  (2057 states)

Small examples

Incremental approaches generally have easier conditions to check (i.e. can stop if invalid)

Complete state formulations run faster as fewer states, but harder to compare (i.e. need to decide "which state is better")



vs.

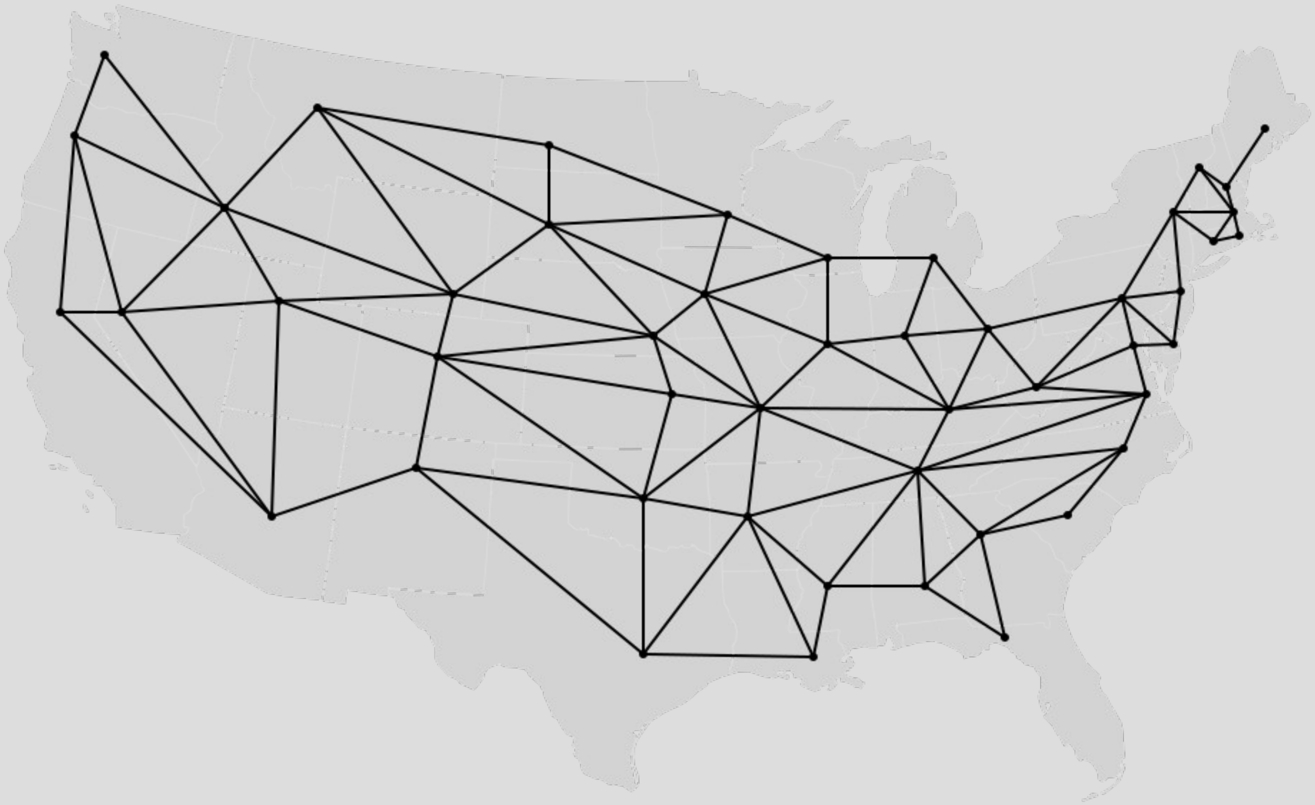
Real world examples

Directions/traveling (land or air)

Model choices: only have interstates?
Add smaller roads, with increased cost?
(pointless if they are never taken)

Real world examples

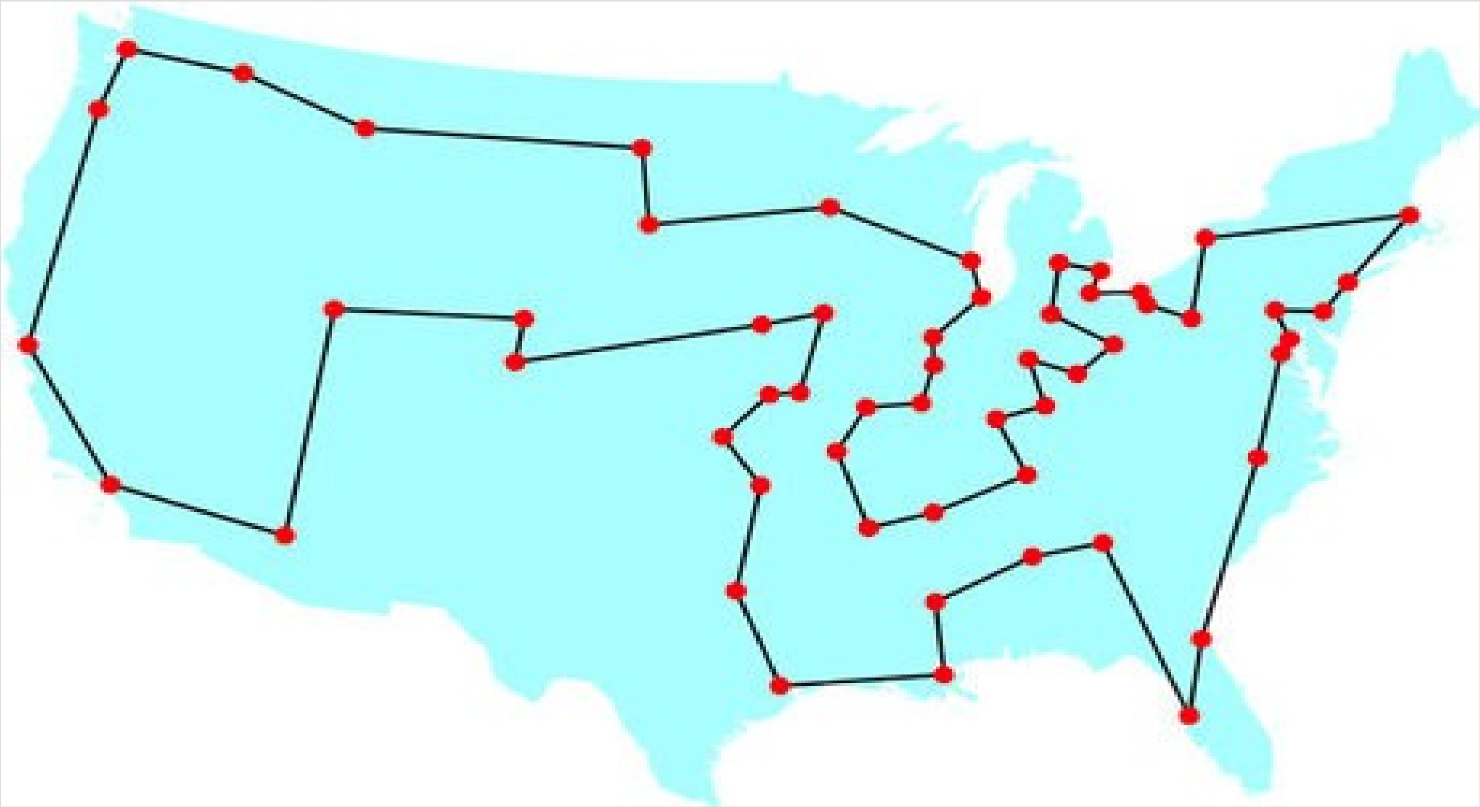
Touring problem: visit each place at least once, end up at starting location



Goal: Minimize distance traveled

Real world examples

Traveling salesperson problem (TSP): Visit each location exactly once and return to start



Goal: Minimize distance traveled

Search algorithm

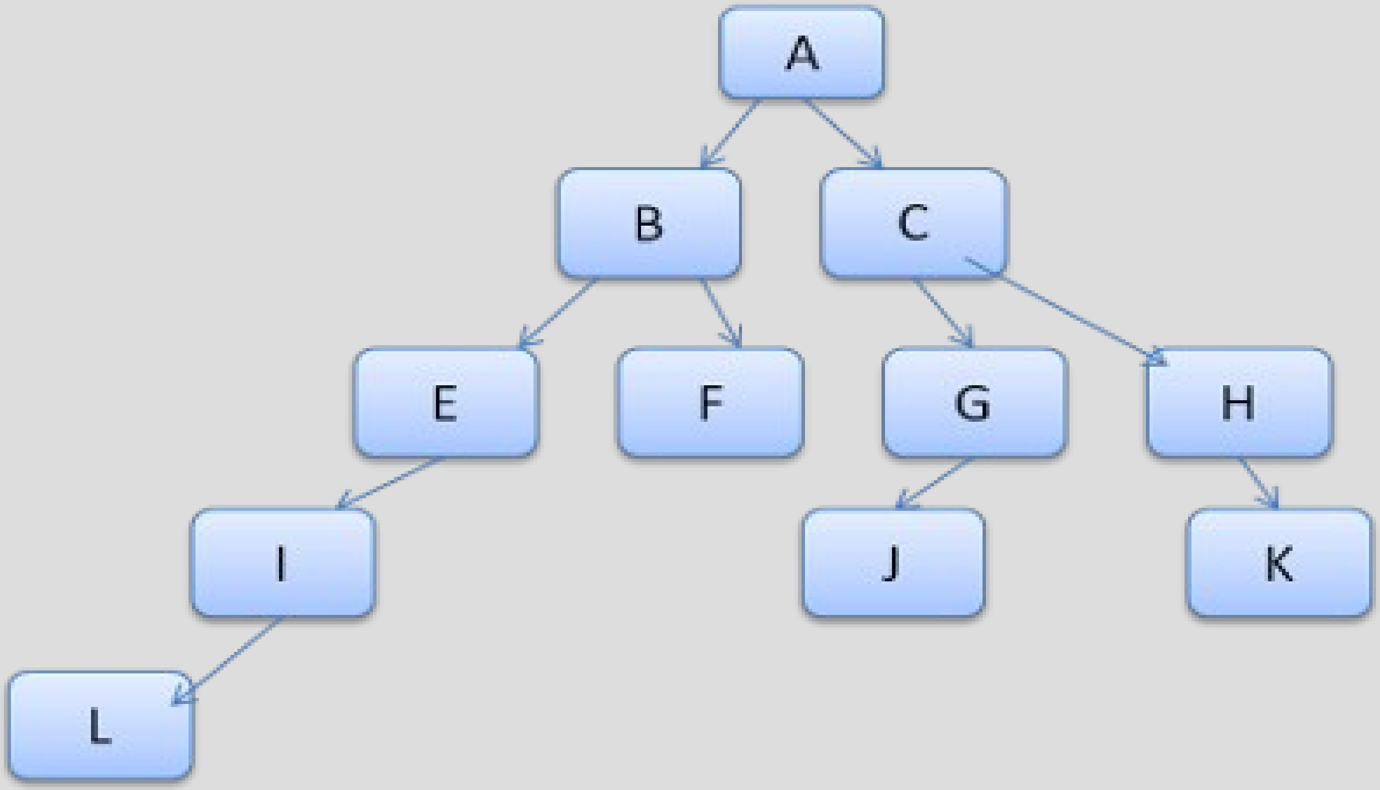
To search, we will build a tree with the root as the initial state

```
function tree-search(root-node)
  fringe ← successors(root-node)
  while ( notempty(fringe) )
      {node ← remove-first(fringe)
       state ← state(node)
       if goal-test(state) return solution(node)
       fringe ← insert-all(successors(node),fringe) }
  return failure
end tree-search
```

Any problems with this?

Search algorithm

You have the choice to search the "action space" or the "state space"

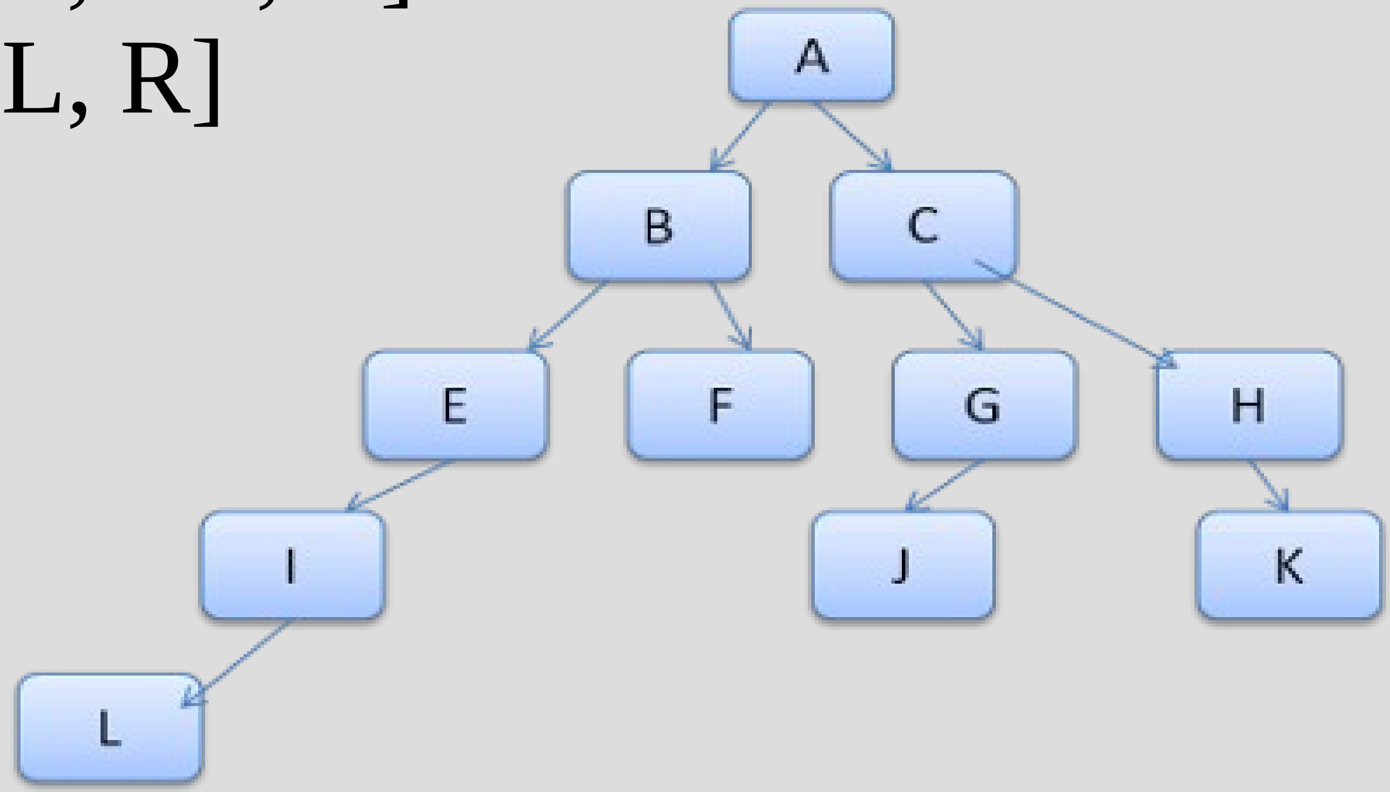


Search algorithm

Action search: (root A), red is explored

1. [L, R]
2. [LL, LR, R]
3. [LL, R]

...

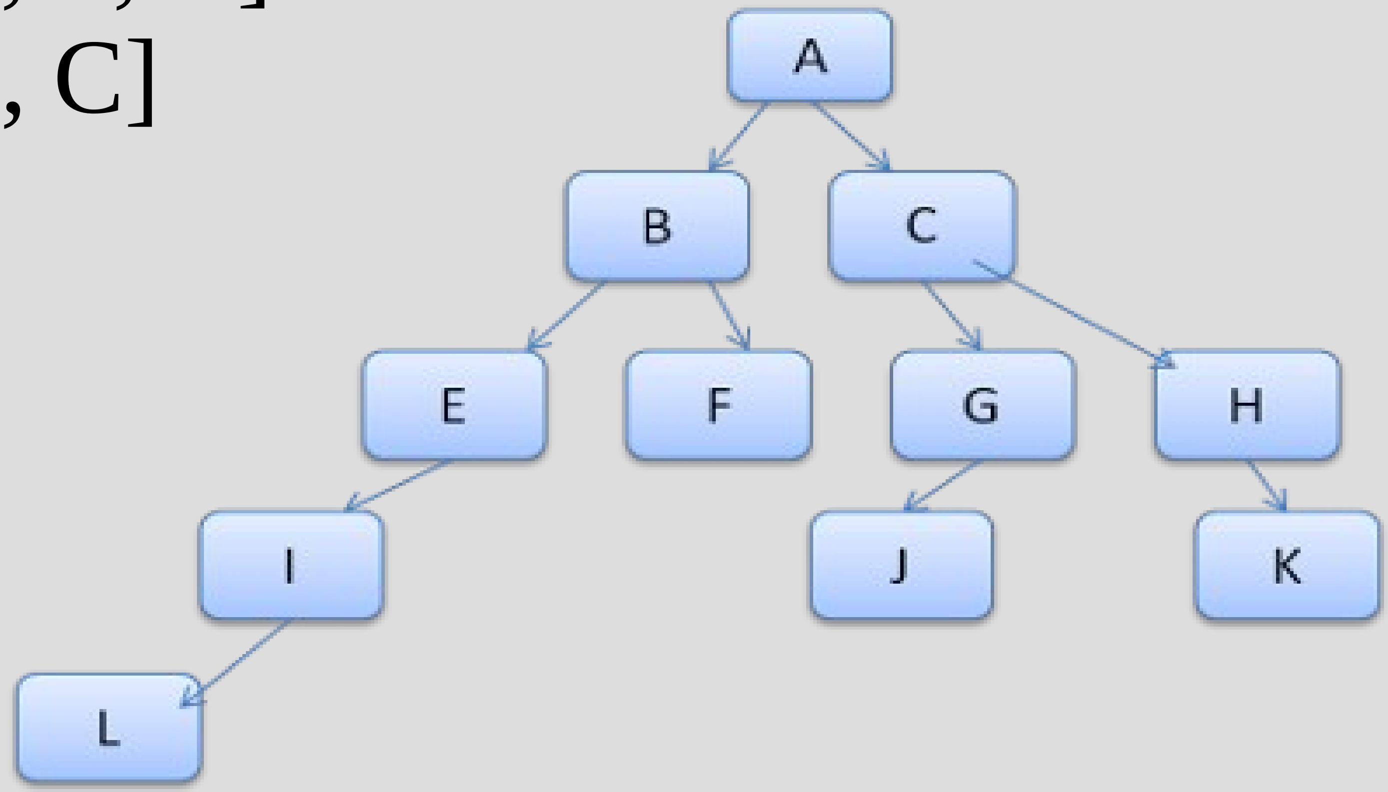


Search algorithm

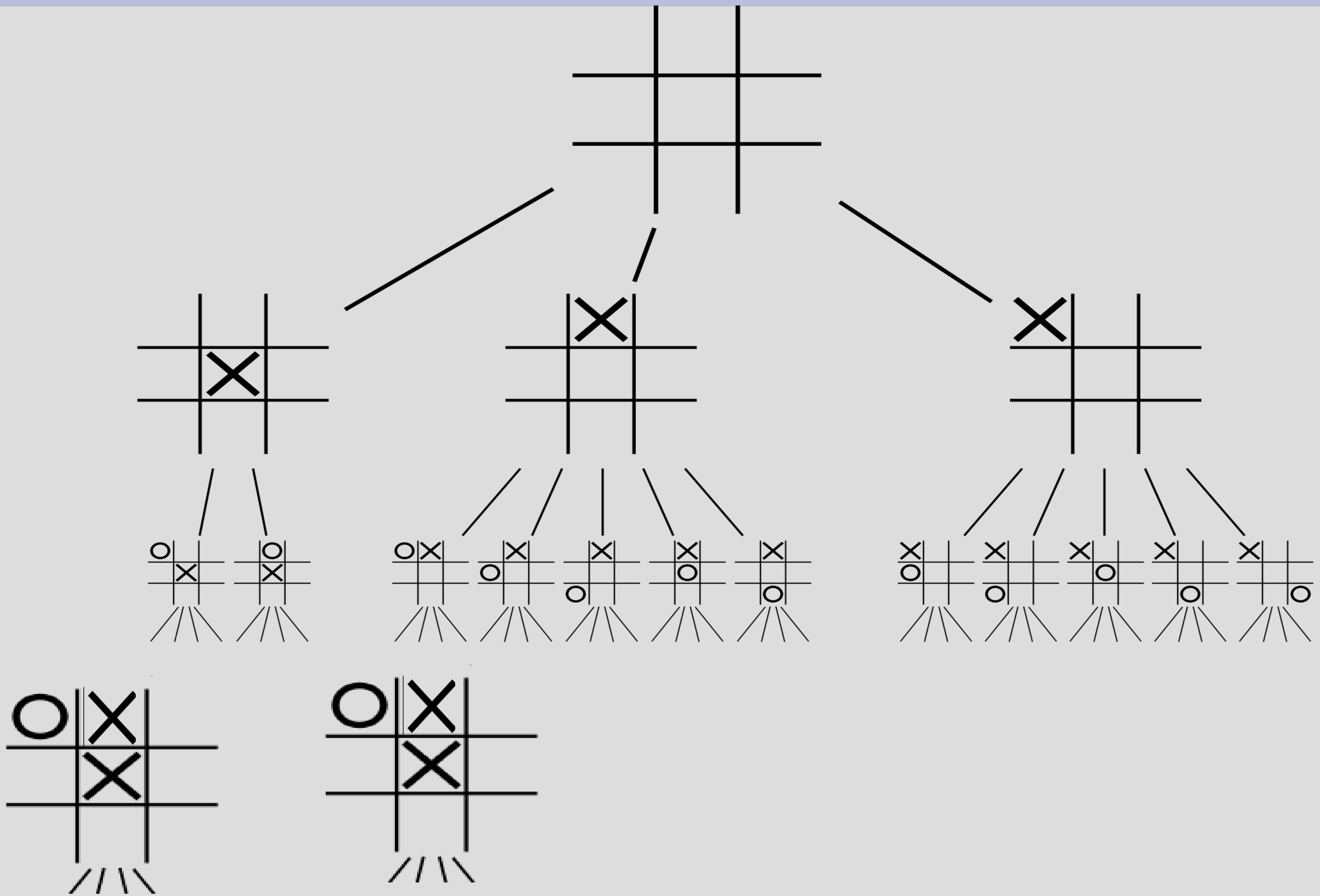
State search: (root A), red is explored

1. [B, C]
2. [E, F, C]
3. [E, C]

...



Search algorithm



Search algorithm

8-queens can actually be generalized to the question:
Can you fit n queens on a z by z board?

Except for a couple of small size boards, you can fit z queens on a z by z board

This can be done fairly easily with recursion

(See: nqueens.cpp)

Search algorithm

We can remove visiting states multiple times by doing this:

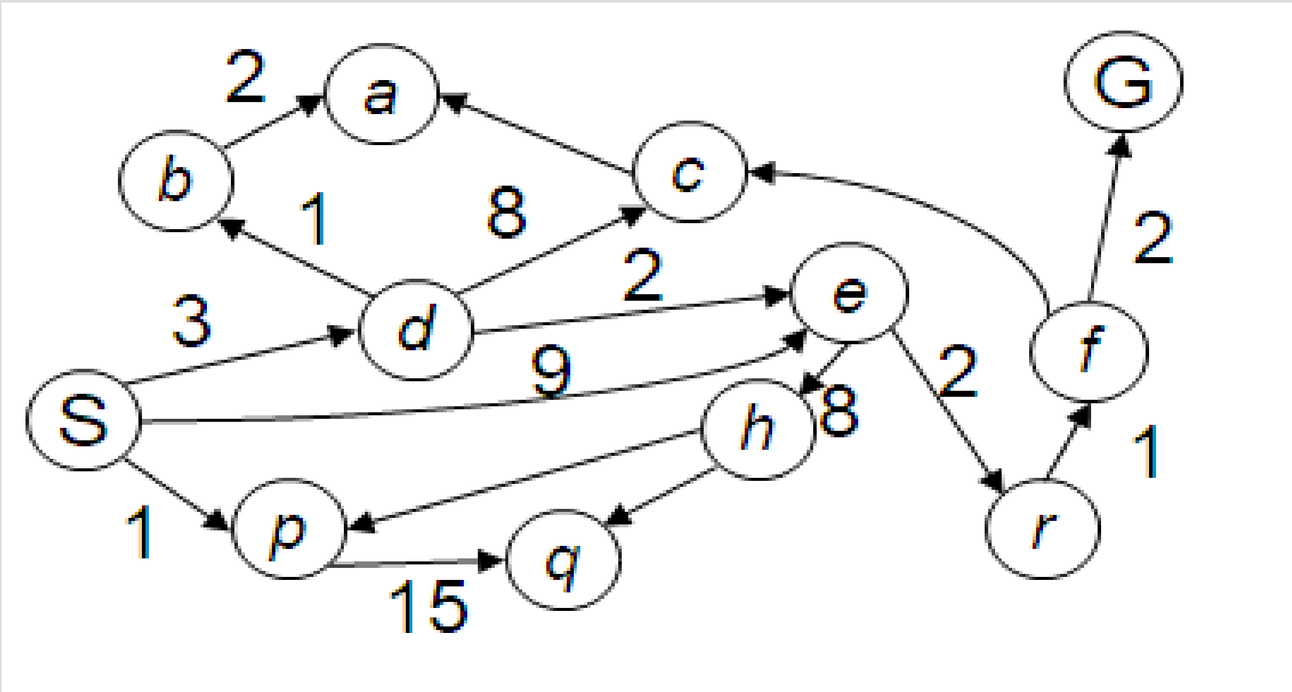
```
function tree-search(root-node)
  fringe ← successors(root-node)
  explored ← empty
  while ( notempty(fringe) )
       {node ← remove-first(fringe)
          state ← state(node)
          if goal-test(state) return solution(node)
          explored ← insert(node,explored)
          fringe ← insert-all(successors(node),fringe,  if node not in explored)
       }
  return failure
end tree-search
```

But this is still not necessarily all that great...

Search algorithm

We will go over 4 general algorithms...

Assume we know: states (root/start and goal), actions, and cost of each action



Search algorithm

Next we will introduce and compare some tree search algorithms

These all assume nodes have 4 properties:

1. The current state
2. Their parent state (and action for transition)
3. Children from this node (result of actions)
4. Cost to reach this node (from root)

Search algorithm

When we find a goal state, we can back track via the parent to get the sequence

To keep track of the unexplored nodes, we will use a queue (of various types)

The explored set is probably best as a hash table for quick lookup (have to ensure similar states reached via alternative paths are the same in the hash, can be done by sorting)

Search algorithm

The search algorithms metrics/criteria:

1. Completeness (does it terminate with a valid solution)
2. Optimality (is the answer the best solution)
3. Time (in big-O notation)
4. Space (big-O)

b = maximum branching factor
d = minimum depth of a goal
m = maximum length of any path

Uninformed search

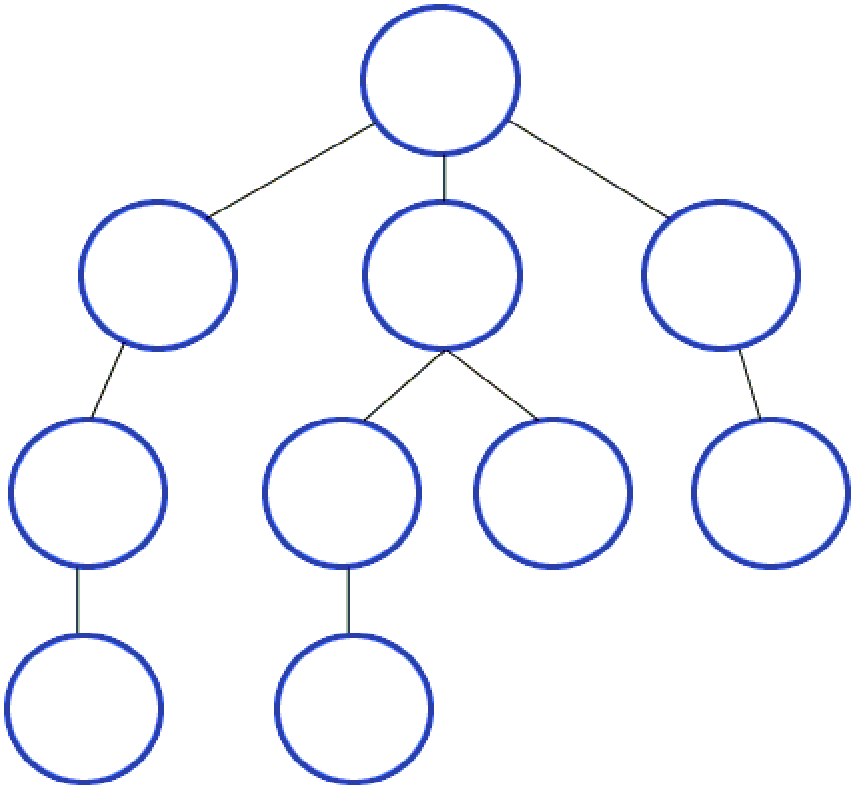
Today, we will focus on uninformed search, which only have the node information (4 parts) (no known relationship between costs)

Next time we will continue with informed searches that assume they have access to additional structures of the problem (i.e. if costs were distances between cities, you could also compute the distance “as the bird flies”)

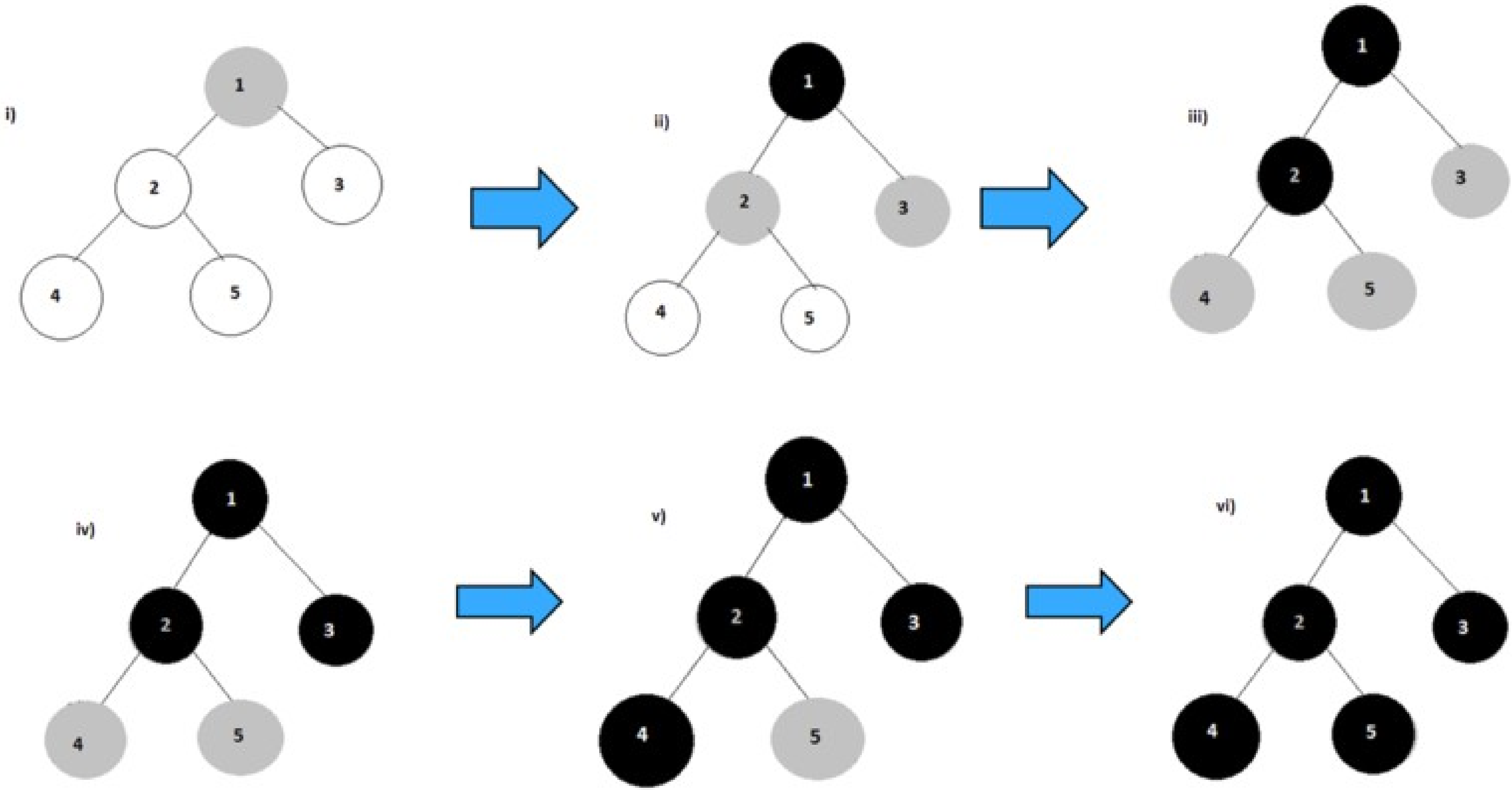
Breadth first search

Breadth first search checks all states which are reached with the fewest actions first

(i.e. will check all states that can be reached by a single action from the start, next all states that can be reached by two actions, then three...)



Breadth first search



(see: https://www.youtube.com/watch?v=5UfMU9TsoEM)
(see: https://www.youtube.com/watch?v=nI0dT288VLs)

Breadth first search

BFS can be implemented by using a simple FIFO (first in, first out) queue to track the fringe/frontier/unexplored nodes

Metrics for BFS:

Complete (i.e. guaranteed to find solution if exists)

Non-optimal (unless uniform path cost)

Time complexity = $O(b^d)$

Space complexity = $O(b^d)$

Breadth first search

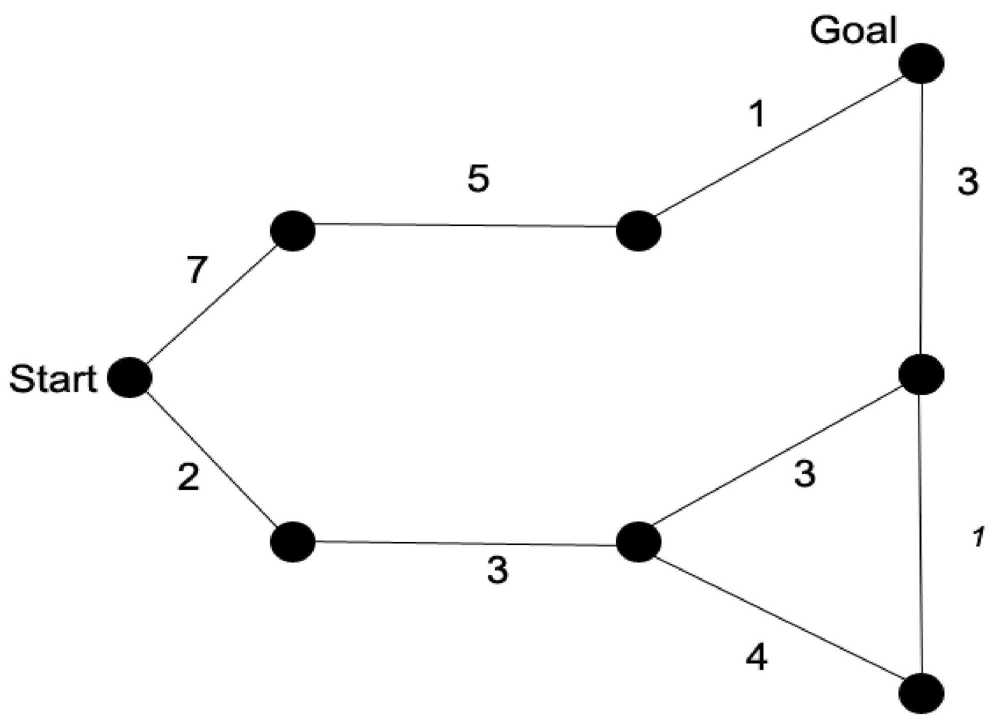
Exponential problems are not very fun, as seen in this picture:

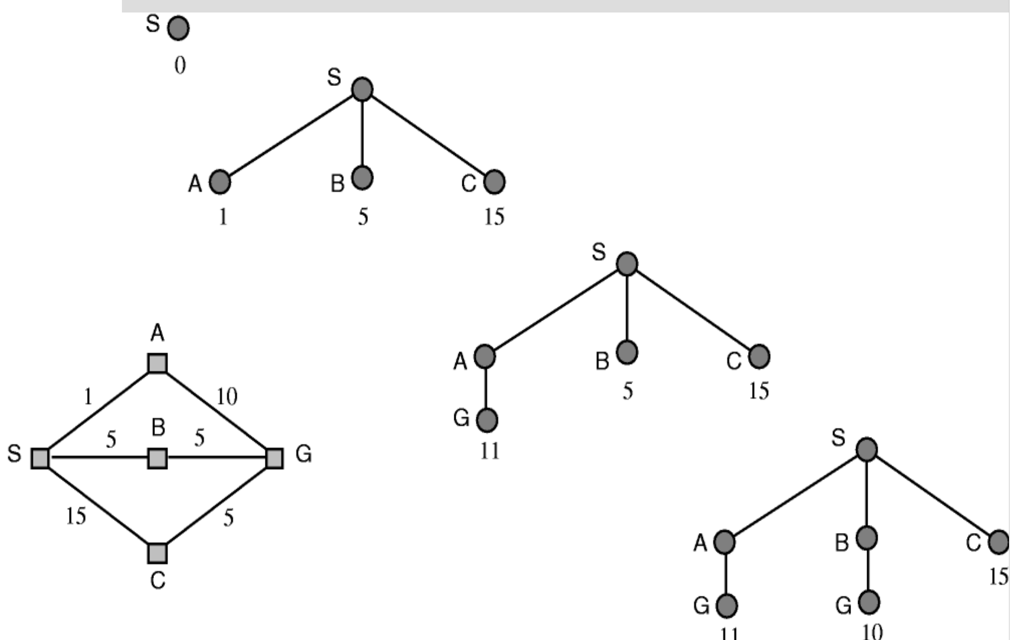
Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

Uniform-cost search

Uniform-cost search also does a queue, but uses a priority queue based on the cost (the lowest cost node is chosen to be explored)





(a) (b)

Uniform-cost search

The only modification is when exploring a node we cannot disregard it if it has already been explored by another node

We might have found a shorter path and thus need to update the cost on that node

We also do not terminate when we find a goal, but instead when the goal has the lowest cost in the queue.

Uniform-cost search

UCS is..

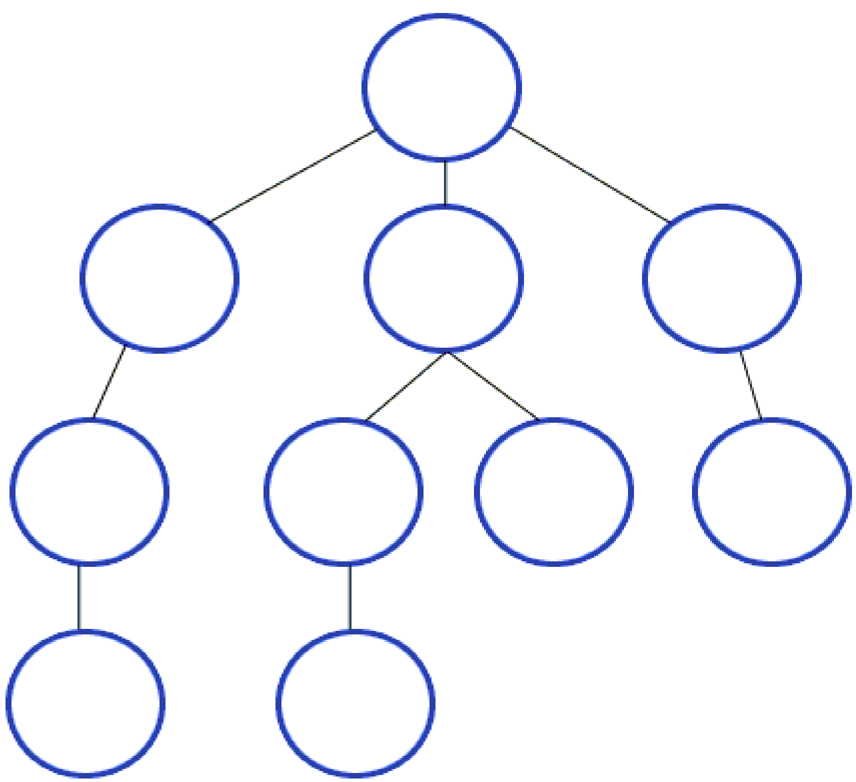
1. Complete (if costs strictly greater than 0)
2. Optimal

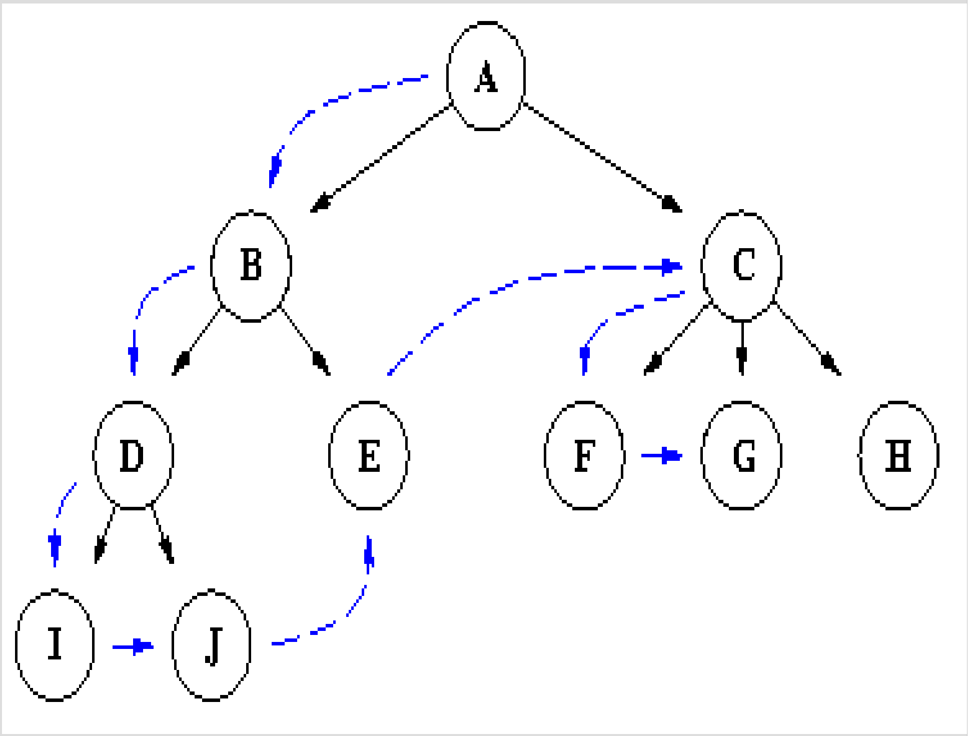
However....

3&4. Time complexity = space complexity = $O(b^{1+C^*/\min(\text{path cost})})$, where C* cost of optimal solution (much worse than BFS)

Depth first search

DFS is same as BFS except with a FILO (or LIFO) instead of a FIFO queue





Depth first search

Metrics:

1. Might not terminate (not correct) (e.g. in vacuum world, if first expand is action L)
2. Non-optimal (just... no)
3. Time complexity = $O(b^d)$
4. Space complexity = $O(b*d)$

Only way this is better than BFS is the space complexity...

Depth limited search

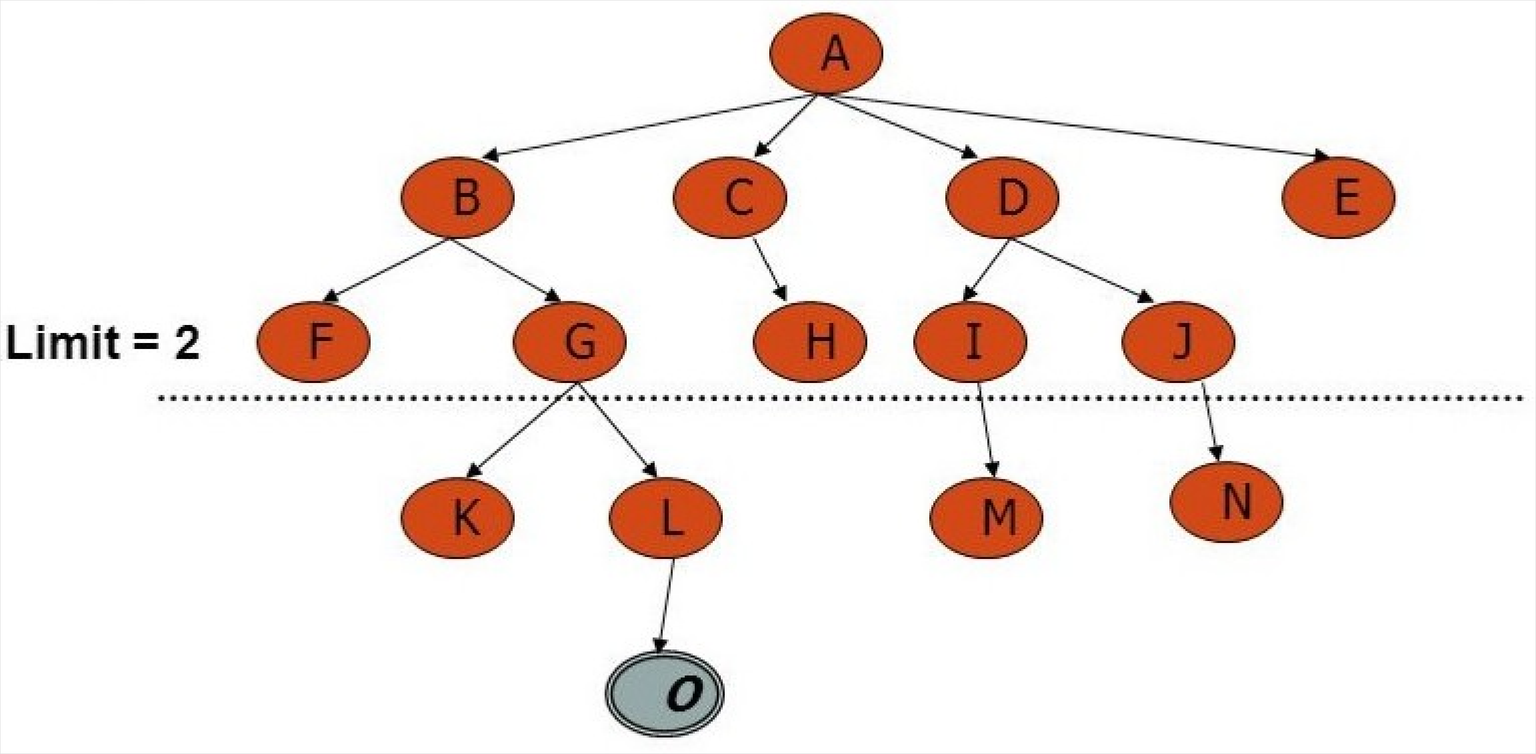
DFS by itself is not great, but it has two (very) useful modifications

Depth limited search runs normal DFS, but if it is at a specified depth limit, you cannot have children (i.e. take another action)

Typically with a little more knowledge, you can create a reasonable limit and makes the algorithm correct

Depth limited search

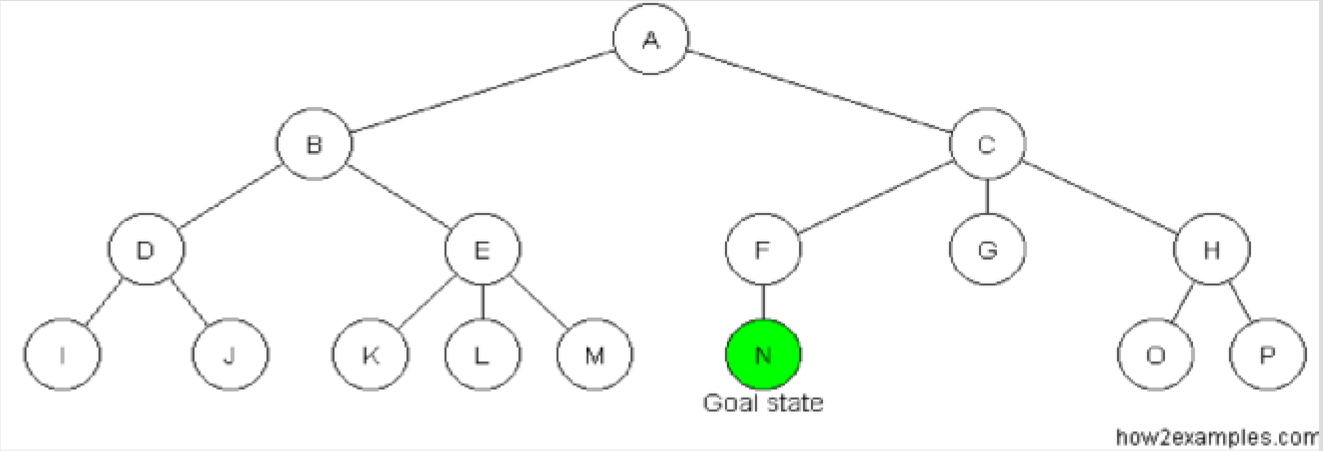
However, if you pick the depth limit before d, you will not find a solution (not correct, but will terminate)



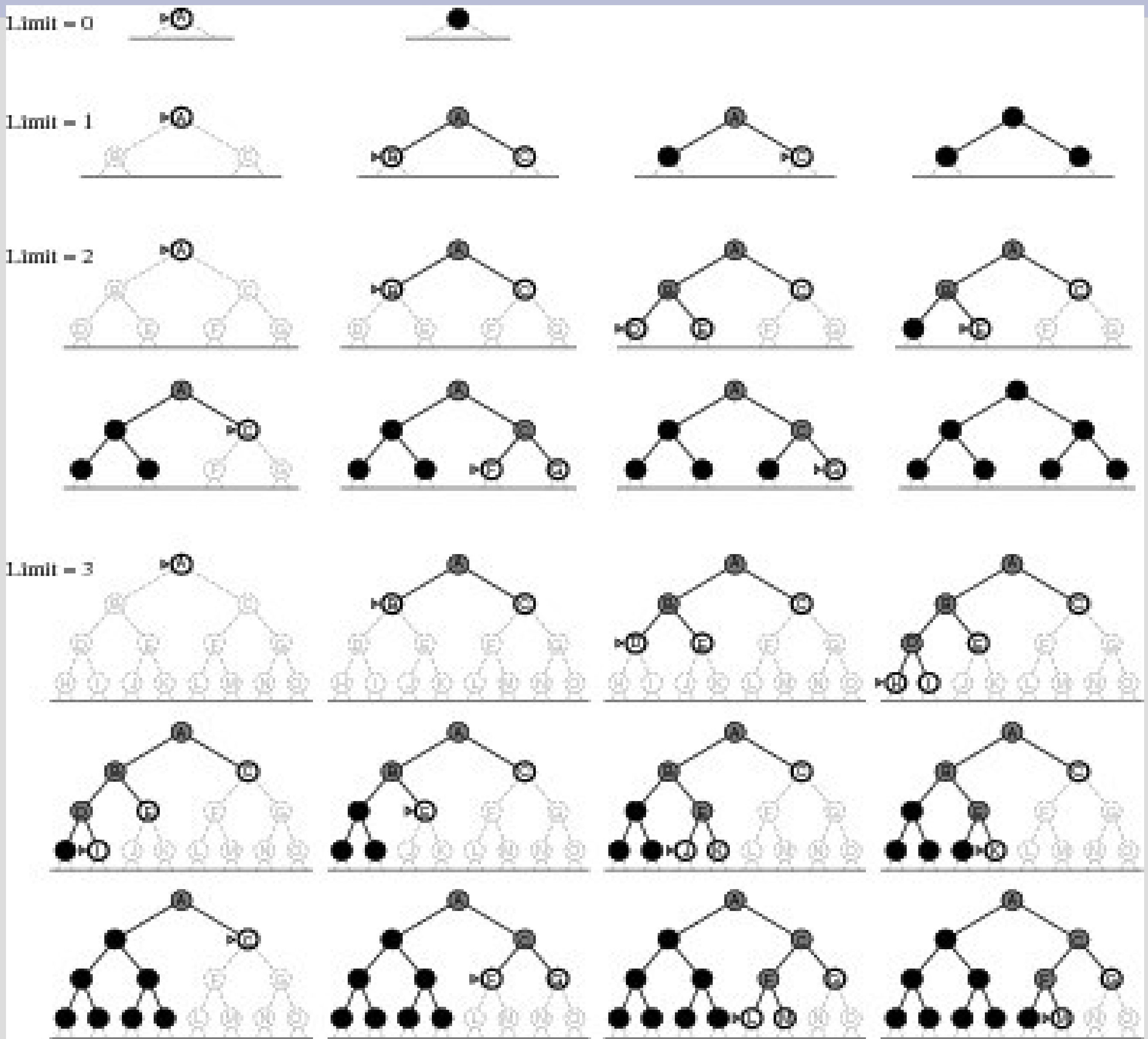
Iterative deepening DFS

Probably the most useful uninformed search is iterative deepening DFS

This search performs depth limited search with maximum depth 1, then maximum depth 2, then 3... until it finds a solution



Iterative deepening DFS



Iterative deepening DFS

The first few states do get re-checked multiple times in IDS, however it is not too many

When you find the solution at depth d, depth 1 is expanded d times (at most b of them)

The second depth are expanded d-1 times (at most b^2 of them)

Thus $d \cdot b + (d-1) \cdot b^2 + ... + 1 \cdot b^d = O(b^d)$

Iterative deepening DFS

Metrics:
1. Complete
2. Non-optimal (unless uniform cost)
3. $O(b^d)$
4. O(bd)

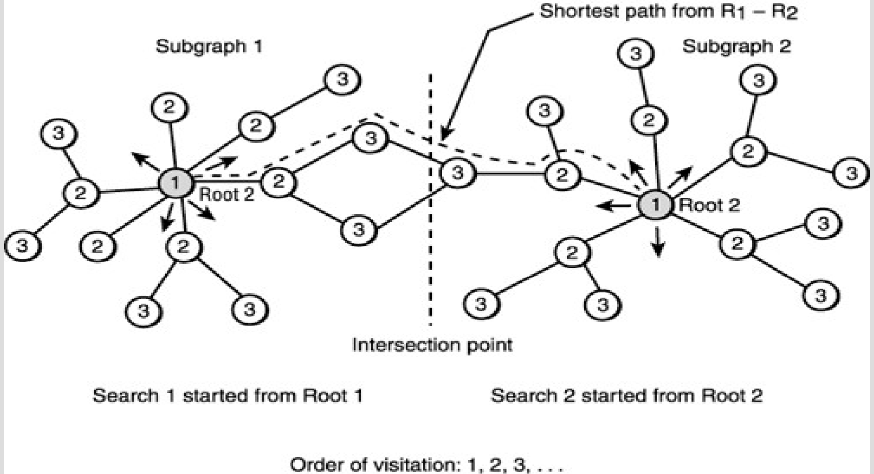
Thus IDS is better in every way than BFS (asymptotically)

Best uninformed we will talk about

Bidirectional search

Bidirectional search starts from both the goal and start (using BFS) until the trees meet

This is better as $2*(b^{d/2}) < b^d$
(the space is much worse than IDS, so only applicable to small problems)



Summary of algorithms
Fig. 3.21, p. 91

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening DLS	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{\lfloor 1+C^*/\varepsilon \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{\lfloor 1+C^*/\varepsilon \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

l = depth limit

Generally the preferred uninformed search strategy

There are a number of footnotes, caveats, and assumptions.
See Fig. 3.21, p. 91.

[a] complete if b is finite
[b] complete if step costs $\geq \varepsilon > 0$
[c] optimal if step costs are all identical
(also if path cost non-decreasing function of depth only)
[d] if both directions use breadth-first search
(also if both directions use uniform-cost search with step costs $\geq \varepsilon > 0$)