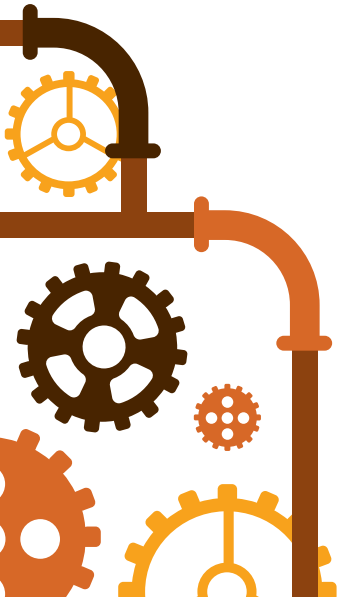
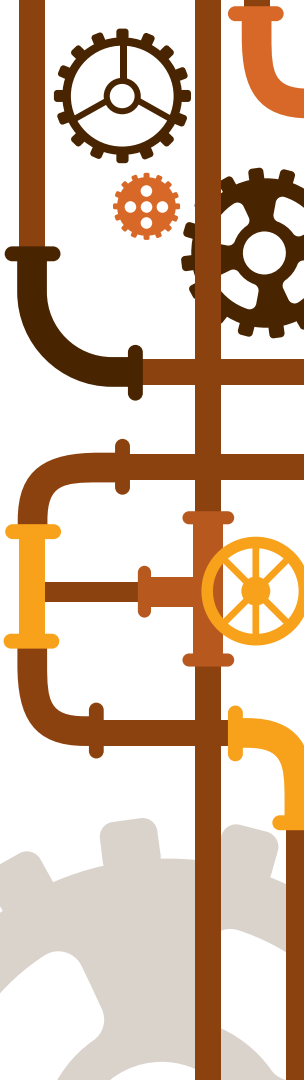
CS 482/628 AI: Problem Solving and Search

Fall 2021

Problem-solving agents

Restricted form of general agent:

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

  state ← UPDATE-STATE(state, p)
  if s is empty then
      g ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, g)
      s ← SEARCH(problem)
  action ← RECOMMENDATION(s, state)
  s ← REMAINDER(s, state)
  return action
```

Note: this is *offline* problem solving.
Online problem solving involves acting without complete knowledge of the problem and solution.

Example: Romania

On holiday in Romania; currently in Arad.
Flight leaves tomorrow from Bucharest

Formulate goal:

be in Bucharest

Formulate problem:

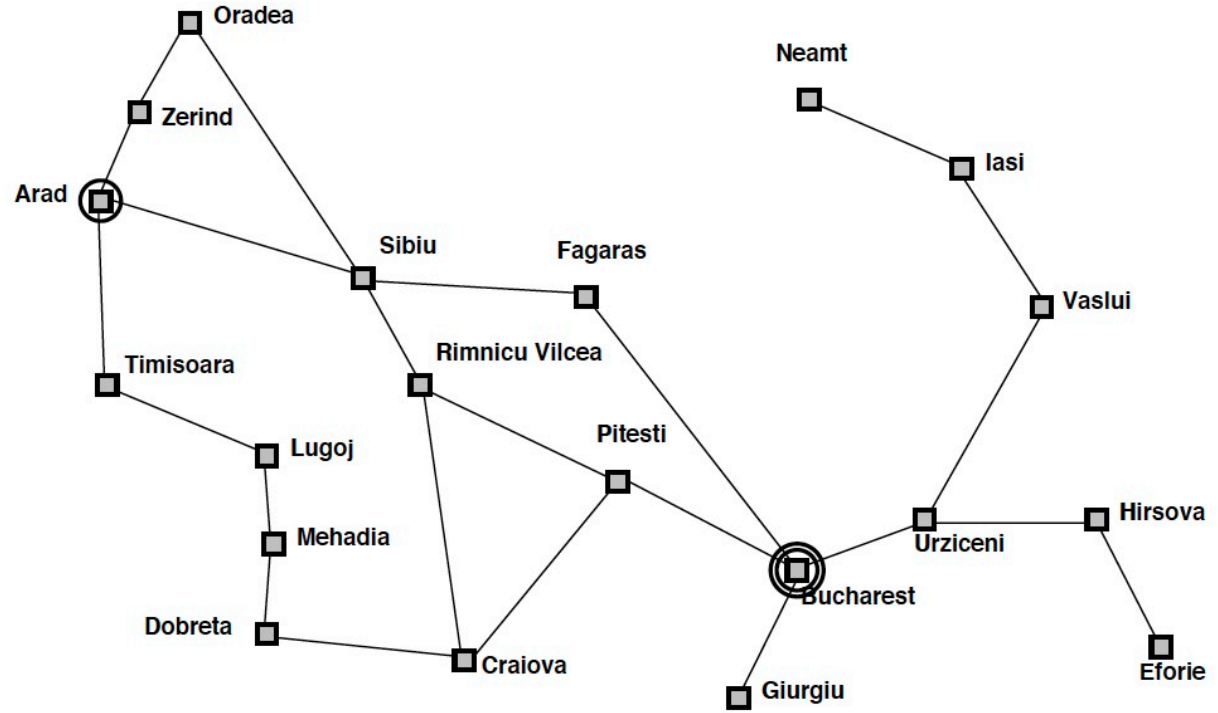
states: various cities

operators: drive between cities

Find solution:

sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Example: Romania



Problem types

Deterministic, accessible $\Longrightarrow$ *single-state problem*
Deterministic, inaccessible $\Longrightarrow$ *multiple-state problem*

Nondeterministic, inaccessible $\Longrightarrow$ *contingency problem*
- must use sensors during execution
- solution is a *tree* or *policy*
- often *interleave* search, execution

Unknown state space $\Longrightarrow$ *exploration problem* ("online")

The agent doesn't know what effect its actions will have

Single-state problem formulation

A *problem* is defined by four items:

initial state e.g., "at Arad"

operators (or *successor function* $S(x)$)
e.g., Arad $\rightarrow$ Zerind Arad $\rightarrow$ Sibiu etc.

goal test, can be
explicit, e.g., $x =$ "at Bucharest"
implicit, e.g., $NoDirt(x)$

path cost (additive)
e.g., sum of distances, number of operators executed, etc.

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

Selecting a state space

Real world is absurdly complex

$\Rightarrow$ state space must be *abstracted* for problem solving

(Abstract) state = set of real states

(Abstract) operator = 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"

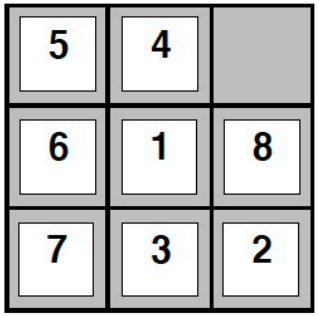
must get to *some* real state "in Zerind"

(Abstract) solution =

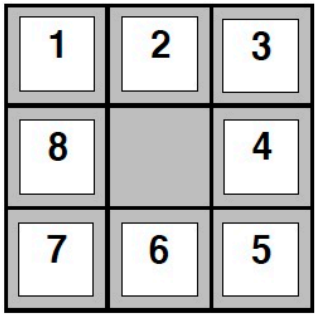
set of real paths that are solutions in the real world

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

Example: The 8-puzzle



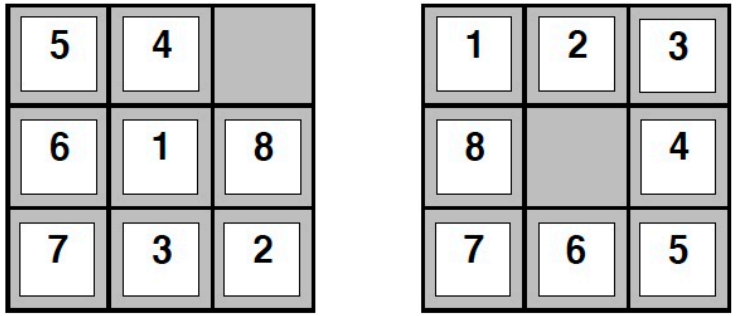
Start State



Goal State

states??
operators??
goal test??
path cost??

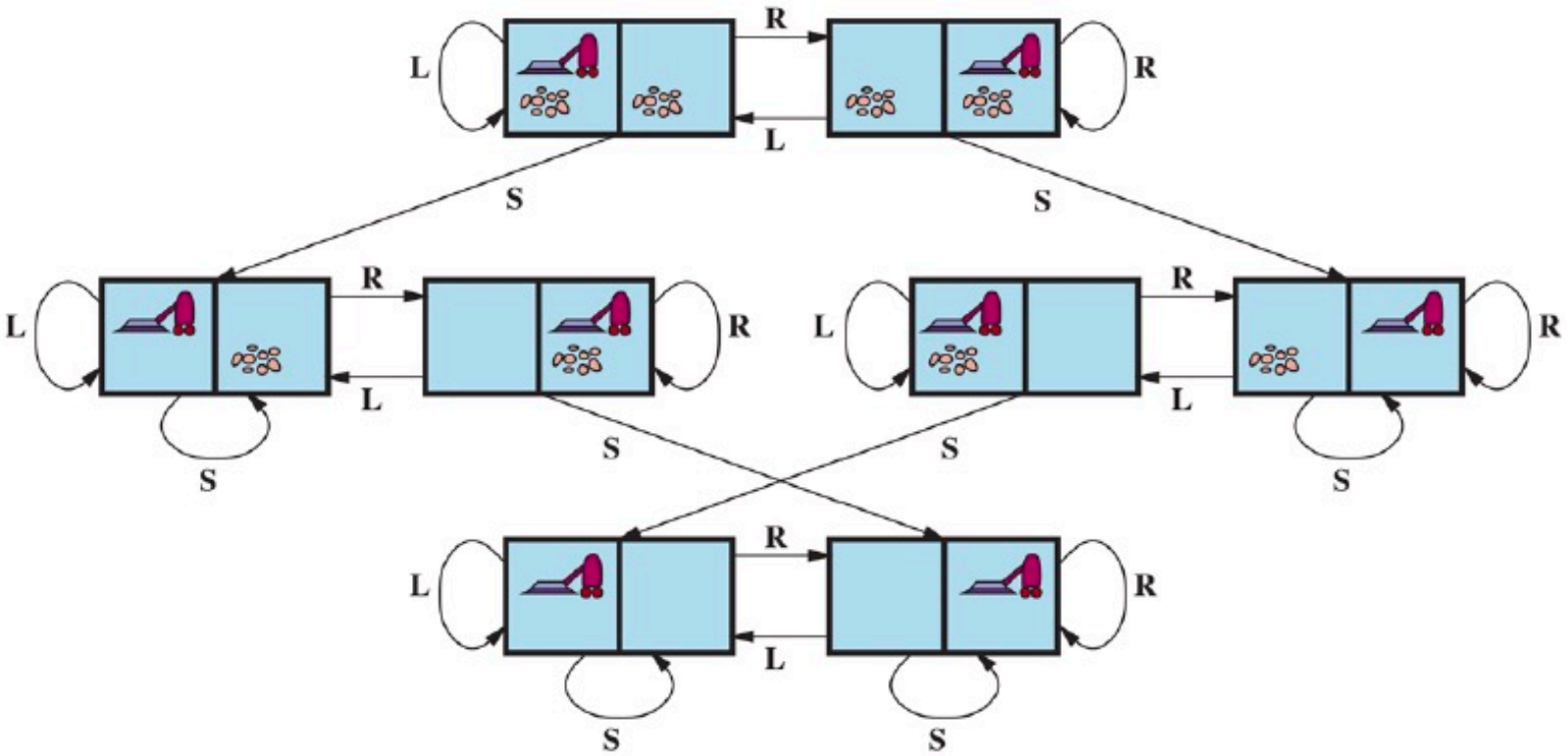
Example: The 8-puzzle



states??: integer locations of tiles (ignore intermediate positions)
operators??: move blank left, right, up, down (ignore unjamming etc.)
goal test??: = goal state (given)
path cost??: 1 per move

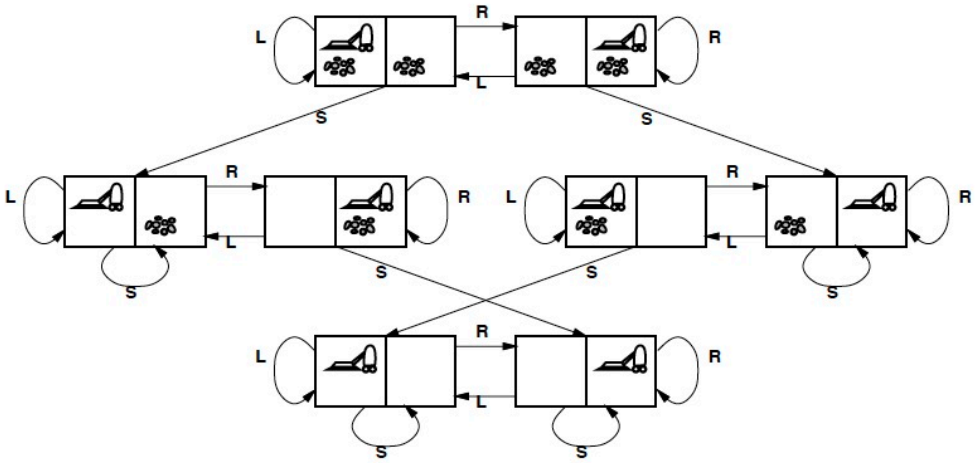
[Note: optimal solution of n-Puzzle family is NP-hard]

Figure 3.2



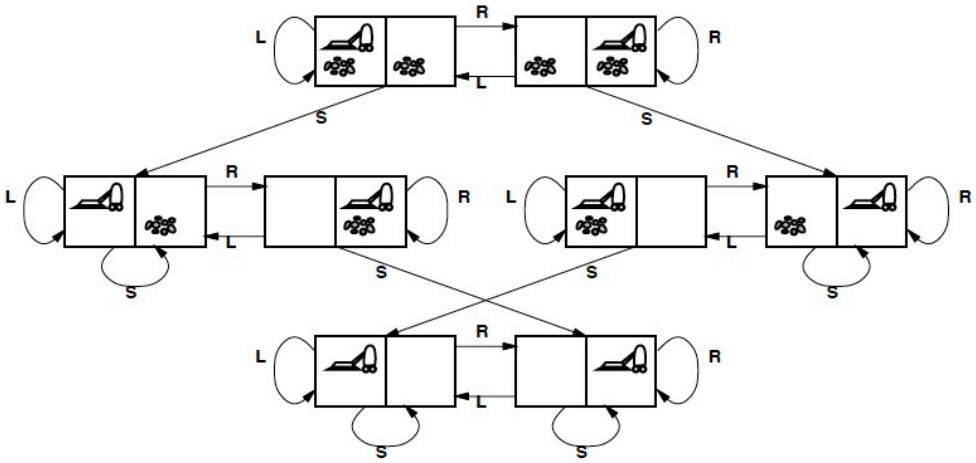
The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: L = *Left*, R = *Right*, S = *Suck*.

Example: vacuum world state space graph



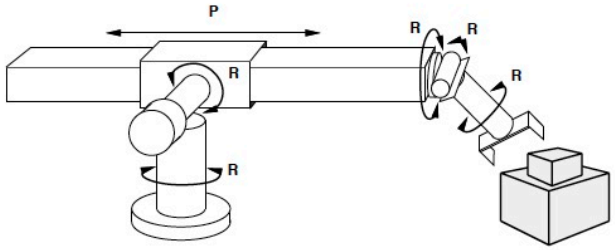
states??
operators??
goal test??
path cost??

Example: vacuum world state space graph



states??: integer dirt and robot locations (ignore dirt *amounts*)
operators??: *Left*, *Right*, *Suck*
goal test??: no dirt
path cost??: 1 per operator

Example: robotic assembly



states??: real-valued coordinates of
robot joint angles
parts of the object to be assembled

operators??: continuous motions of robot joints

goal test??: complete assembly *with no robot included!*

path cost??: time to execute

Search algorithms

Basic idea:

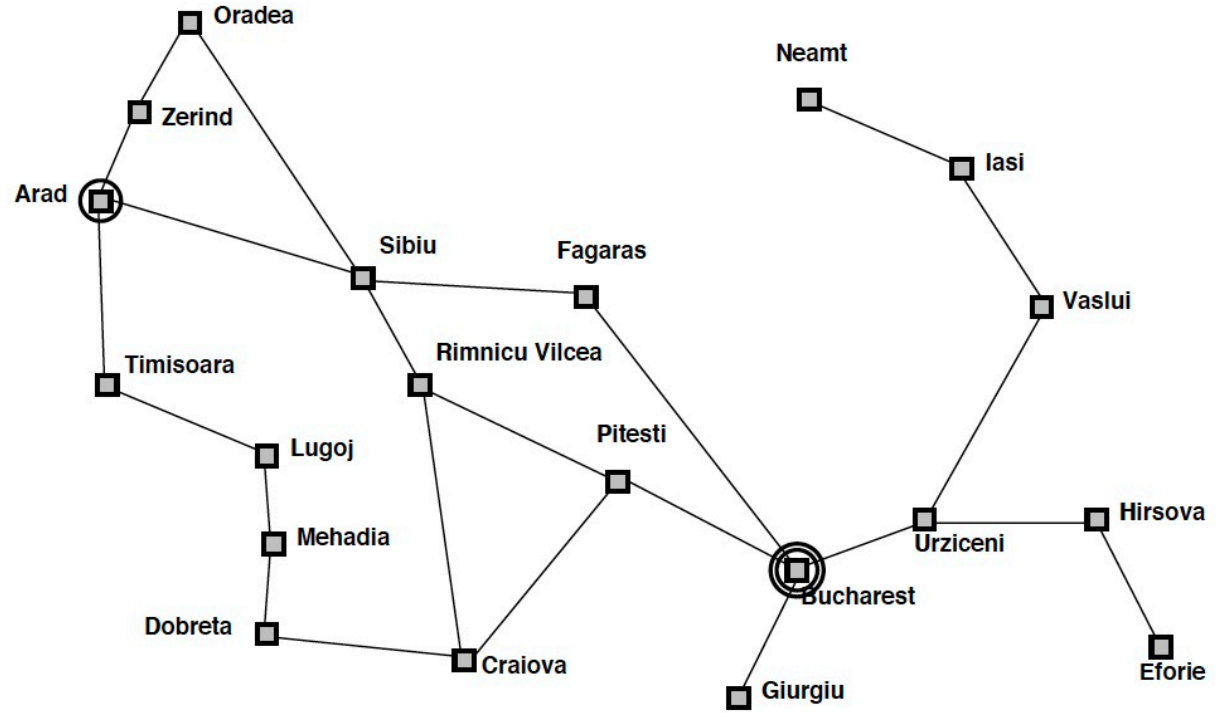
offline, simulated exploration of state space

by generating successors of already-explored states

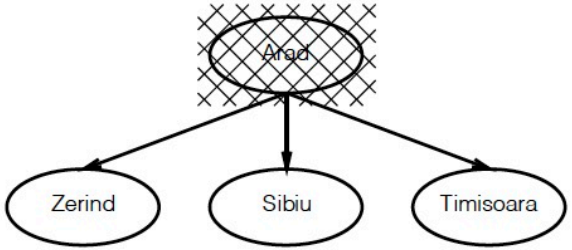
(a.k.a. *expanding* states)

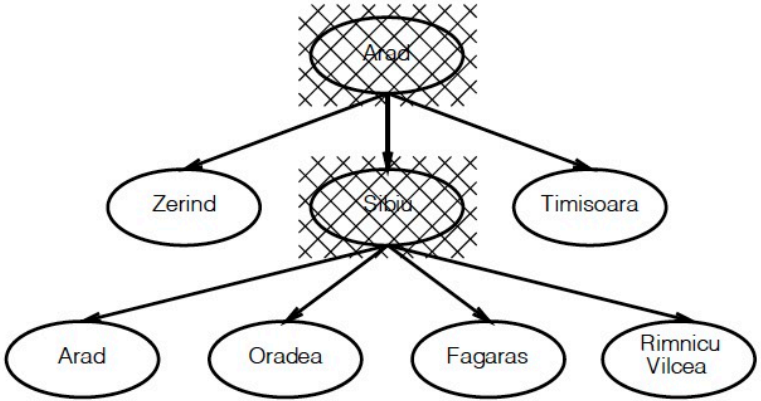
```
function GENERAL-SEARCH( problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains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

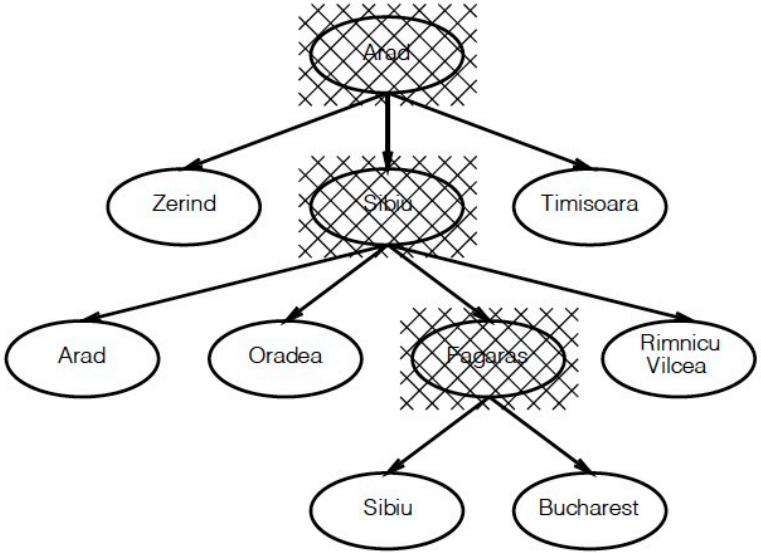
Example: Romania



General search example


Arad
Zerind
Sibiu
Timisoara


Arad
Zerind
Sibiu
Timisoara
Arad
Oradea
Fagaras
Rimnicu
Vilcea

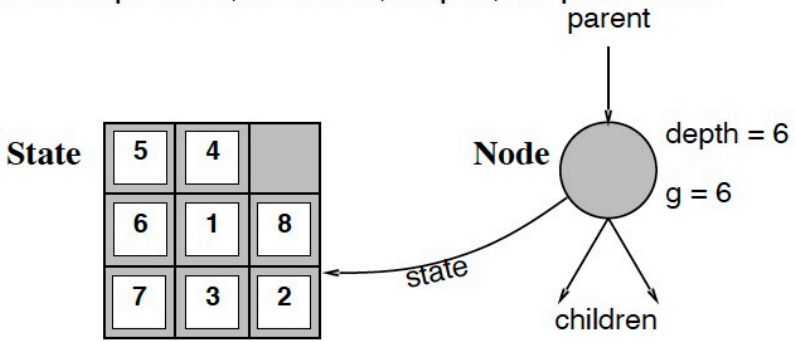

Arad
Zerind
Sibiu
Timisoara
Arad
Oradea
Fagaras
Rimnicu Vilcea
Sibiu
Bucharest

Implementation of search algorithms

```
function GENERAL-SEARCH( problem, QUEUING-FN) returns a solution, or failure
    nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
    loop do
        if nodes is empty then return failure
        node ← REMOVE-FRONT(nodes)
        if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
        nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
    end
```

Implementation contd: 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 *parent*, *children*, *depth*, *path cost* $g(x)$
States do not have parents, children, depth, or path cost!



The EXPAND function creates new nodes, filling in the various fields and using the OPERATORS (or SUCCESSORFN) of the problem to create the corresponding states.

Search strategies

A strategy is defined by picking the *order of node expansion*

Strategies are evaluated along the following dimensions:

- completeness—does it always find a solution if one exists?
- time complexity—number of nodes generated/expanded
- space complexity—maximum number of nodes in memory
- optimality—does it always find a least-cost solution?

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 ∞)

Measuring Performance

Completeness: is the strategy guaranteed to find a solution when one exists?

Time Complexity: how long does it take to find a solution?

Space Complexity: how much memory does it require to perform the search?

Optimality: Does the strategy find the best-quality solution when more than one solution exists?

Un-informed

Uninformed search strategies

Uninformed strategies use only the information available in the problem definition

Breadth-first search

Uniform-cost search

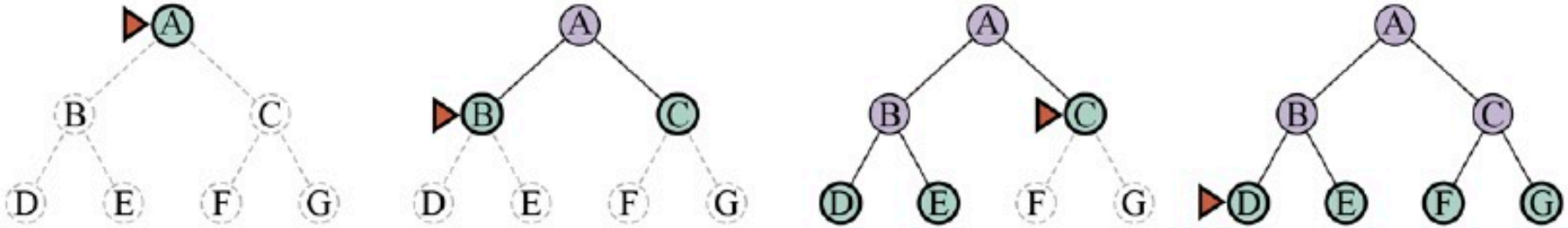
Depth-first search

Depth-limited search

Iterative deepening search

Breadth-first search

Figure 3.8



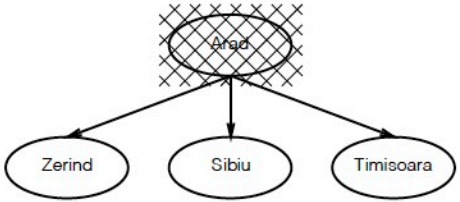
Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by the triangular marker.

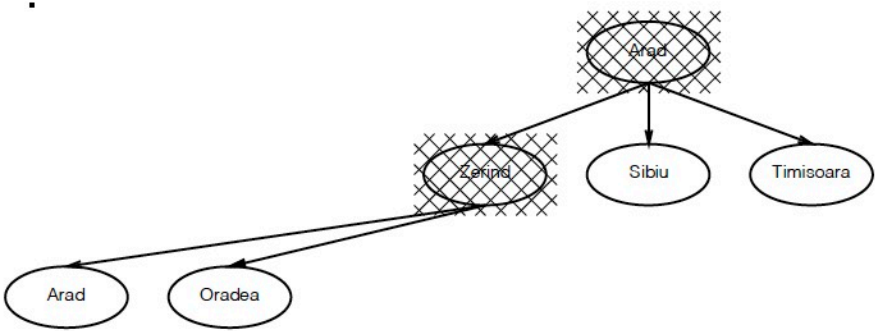
Breadth-first search

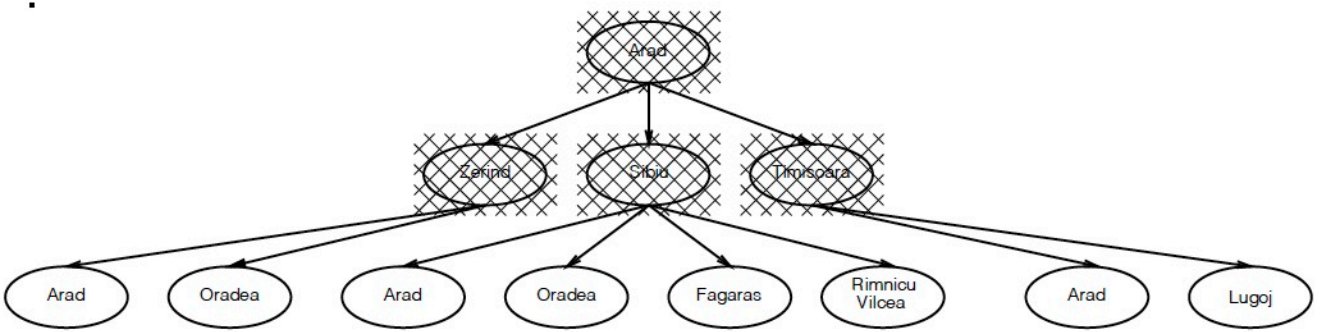
Expand shallowest unexpanded node

Implementation:

QUEUEINGFN = put successors at end of queue


Arad
Zerind
Sibiu
Timisoara


Arad
Zerind
Sibiu
Timisoara
Arad
Oradea


Arad
Zerind
Sibiu
Timisoara
Arad
Oradea
Arad
Oradea
Fagaras
Rimnicu Vilcea
Arad
Lugoj

Properties of breadth-first search

Complete??

Time??

Space??

Optimal??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$, i.e., exponential in d

Space?? $O(b^d)$ (keeps every node in memory)

Optimal?? Yes (if cost $= 1$ per step); not optimal in general

Space is the big problem; can easily generate nodes at 1MB/sec
so 24hrs $=$ 86GB.

Uninformed search strategies

Uninformed strategies use only the information available in the problem definition

Breadth-first search

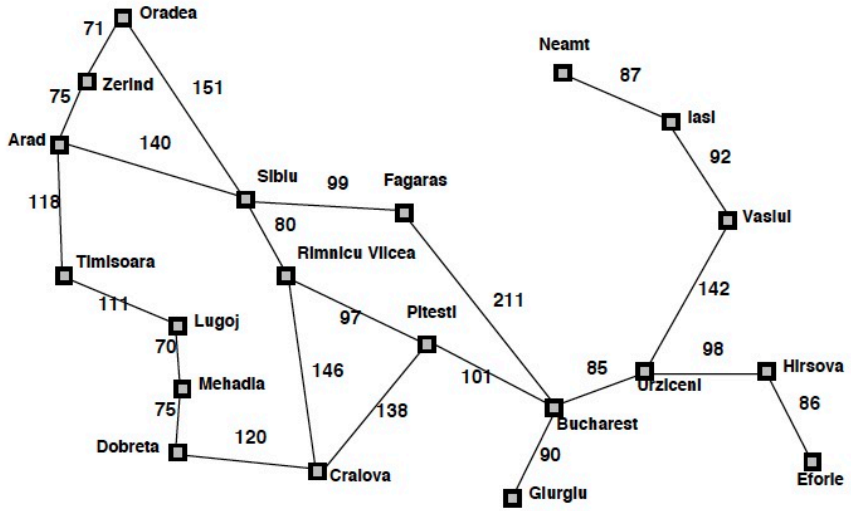
Uniform-cost search

Depth-first search

Depth-limited search

Iterative deepening search

Romania with step costs in km



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Uniform-cost search

Expand least-cost unexpanded node

Also called Dijkstra's algorithm

Implementation:

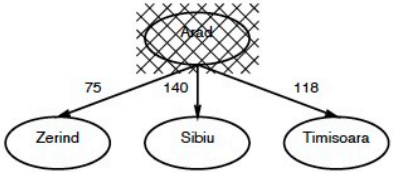
QUEUEINGFN = insert in order of increasing path cost

Uniform-cost search

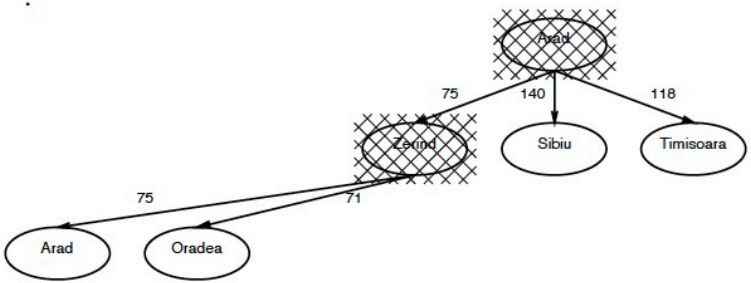
.

.

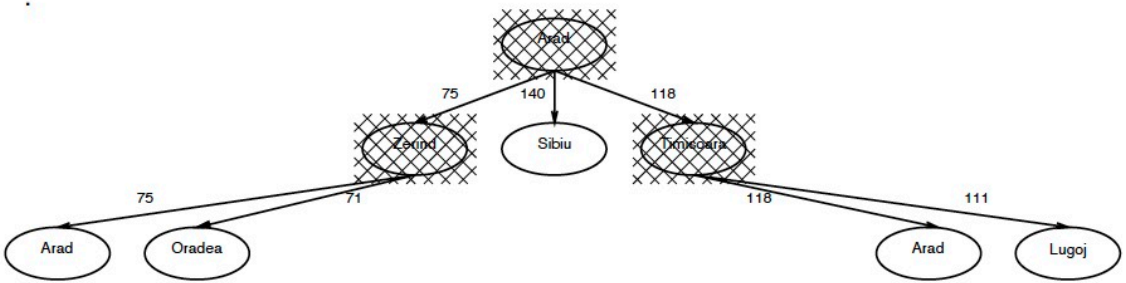
.



Uniform-cost search



Uniform-cost search



Properties of uniform-cost search

Complete?? Yes, if step cost $\geq \epsilon$

Time?? # of nodes with $g \leq$ cost of optimal solution

Space?? # of nodes with $g \leq$ cost of optimal solution

Optimal?? Yes

The complexity of uniform-cost search is characterized in terms of C^*, the cost of the optimal solution,[8] and ϵ, a lower bound on the cost of each action, with $\epsilon > 0$. Then the algorithm's worst-case time and space complexity is $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, which can be much greater than b^d. This is because uniform-cost search can explore large trees of actions with low costs before exploring paths involving a high-cost and perhaps useful action. When all action costs are equal, $b^{1+\lfloor C^*/\epsilon \rfloor}$ is just b^{d+1}, and uniform-cost search is similar to breadth-first search.

Uniform-cost search is complete and is cost-optimal, because the first solution it finds will have a cost that is at least as low as the cost of any other node in the frontier. Uniform-cost search considers all paths systematically in order of increasing cost, never getting caught going down a single infinite path (assuming that all action costs are $> \epsilon > 0$).

Uninformed search strategies

Uninformed strategies use only the information available in the problem definition

Breadth-first search

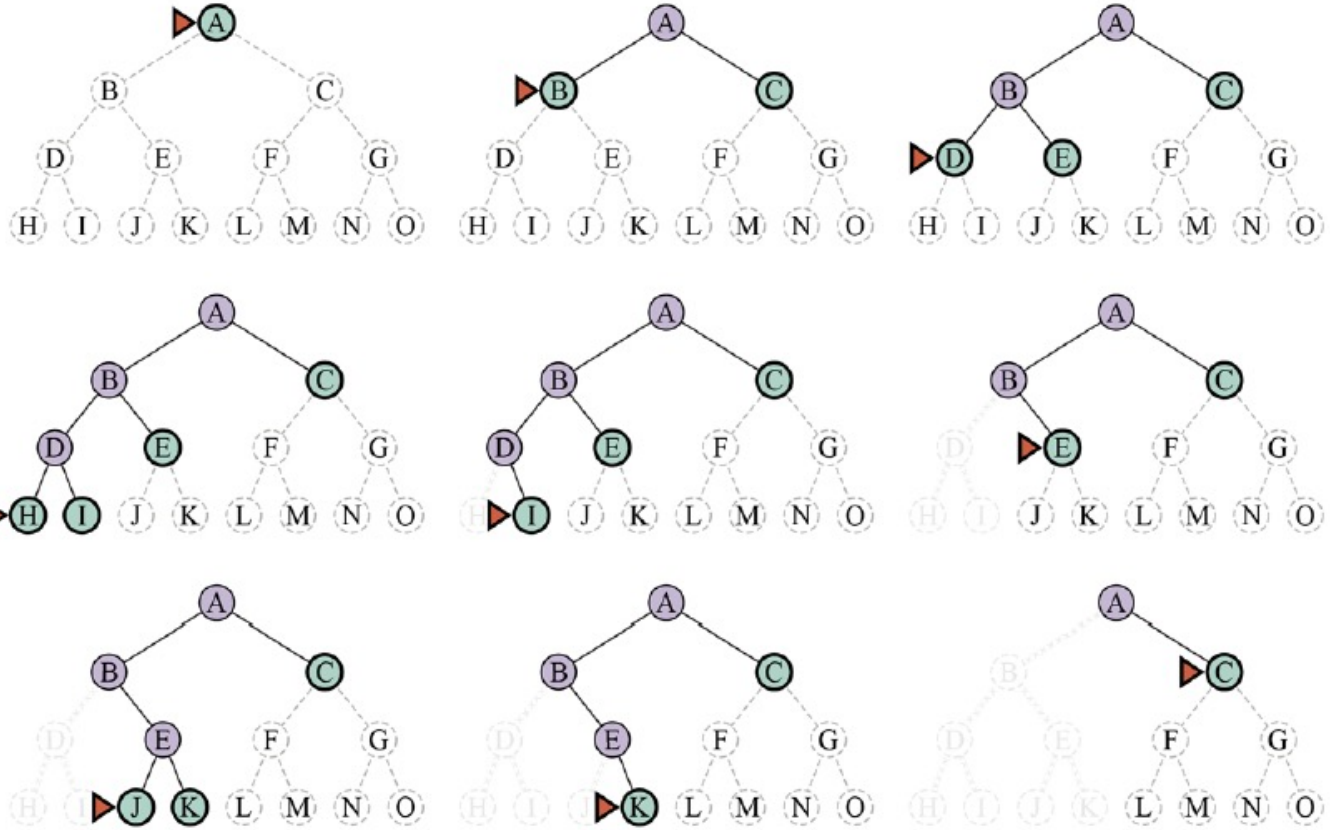
Uniform-cost search

Depth-first search

Depth-limited search

Iterative deepening search

Depth-first search



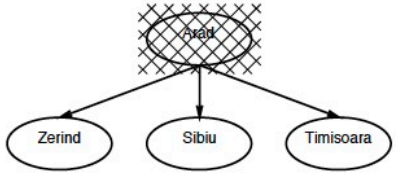
Depth-first search

Expand deepest unexpanded node

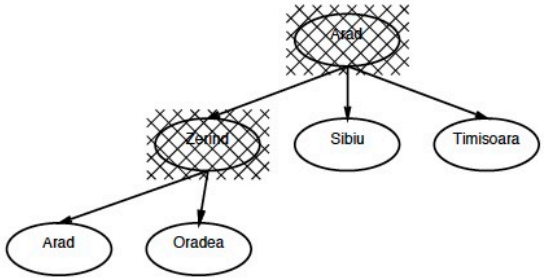
Implementation:

QUEUEINGFN = insert successors at front of queue

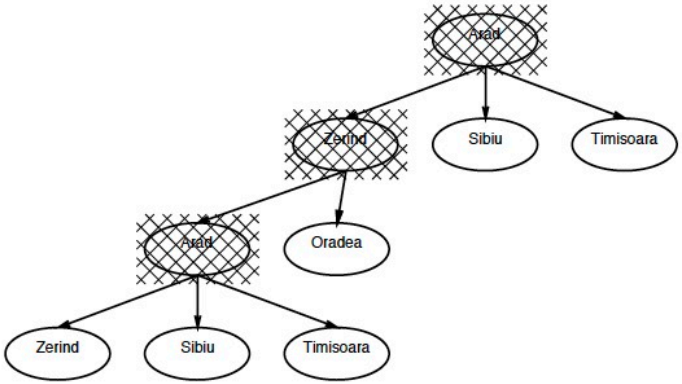
Depth-first search



Depth-first search



Depth-first search



I.e., depth-first search can perform infinite cyclic excursions
Need a finite, non-cyclic search space (or repeated-state checking)

Properties of depth-first search

Complete??

Time??

Space??

Optimal??

Properties of depth-first search

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

b—maximum branching factor of the search tree
d—depth of the least-cost solution
m—maximum depth of the state space (may be ∞)

Uninformed search strategies

Uninformed 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

Depth-limited search

= depth-first search with depth limit l

Implementation:

Nodes at depth l have no successors

Uninformed search strategies

Uninformed 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

Iterative deepening search

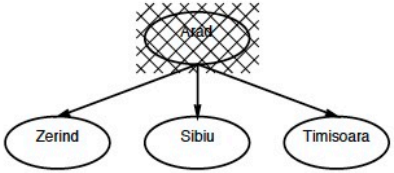
```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution sequence
    inputs: problem, a problem

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

Iterative deepening search $l = 0$

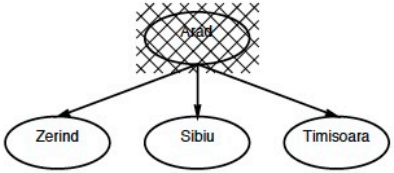
Iterative deepening search $l = 1$

Iterative deepening search $l = 1$

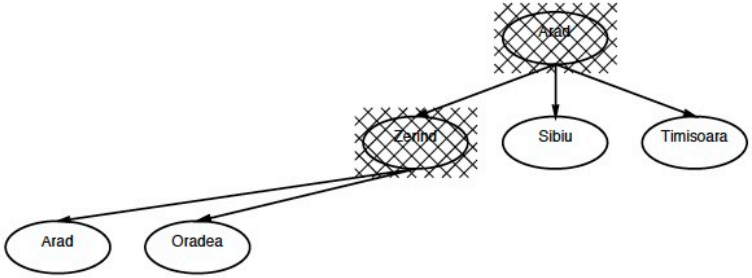


Iterative deepening search $l = 2$

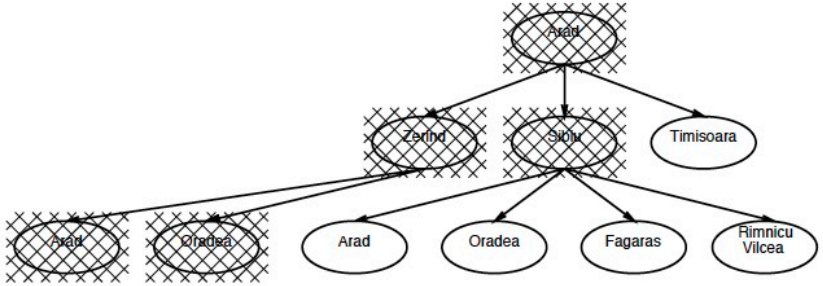
Iterative deepening search $l = 2$



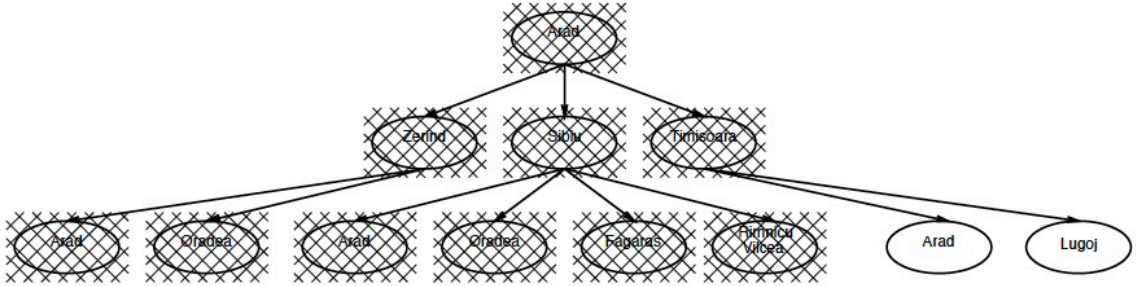
Iterative deepening search $l = 2$



Iterative deepening search $l = 2$



Iterative deepening search $l = 2$



Properties of iterative deepening search

Complete??

Time??

Space??

Optimal??

Properties of iterative deepening search

Complete?? Yes

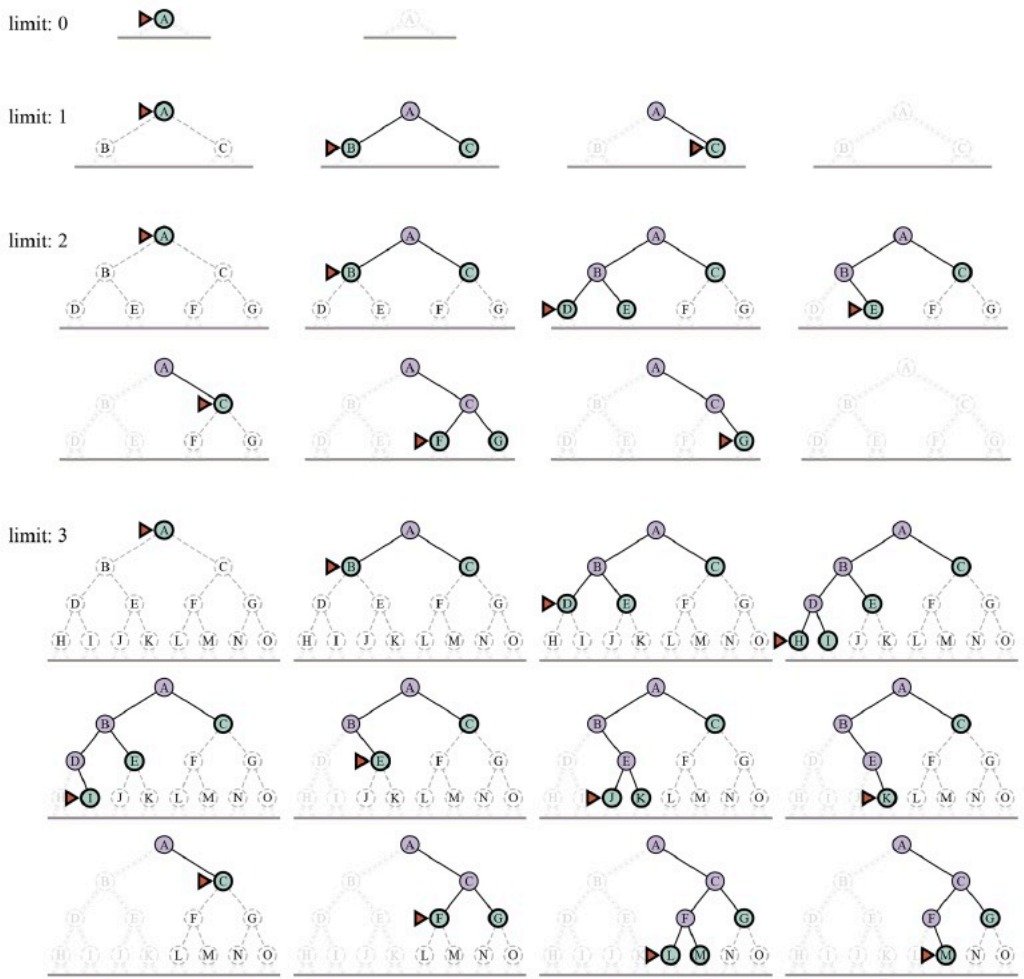
Time?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal?? Yes, if step cost $= 1$

Can be modified to explore uniform-cost tree

Figure 3.13



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

Uninformed search strategies

Uninformed strategies use only the information available in the problem definition

Breadth-first search

Uniform-cost search

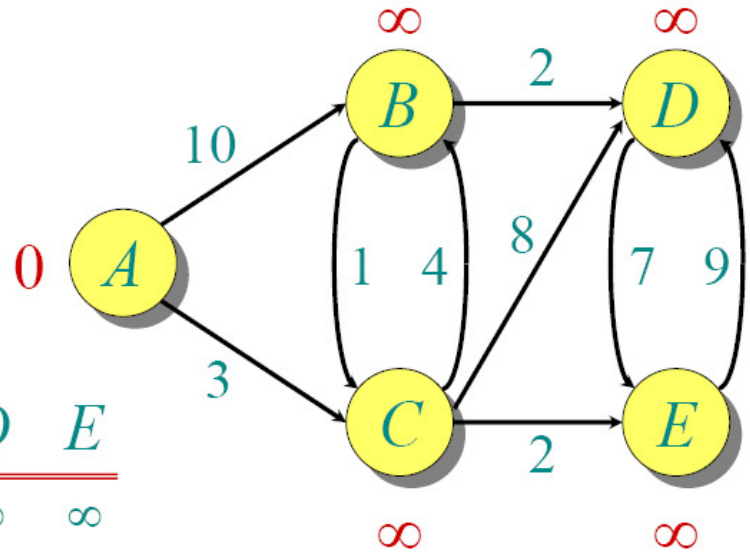
Depth-first search

Depth-limited search

Iterative deepening search

Exercise: uniform-cost search (Dijkstra)

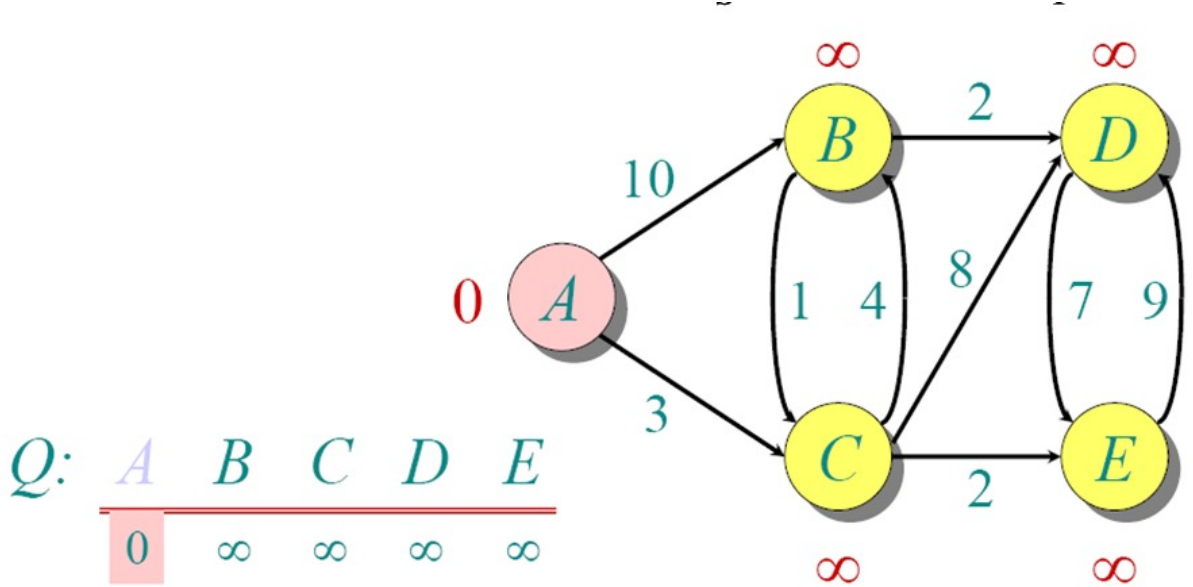
Initialize:



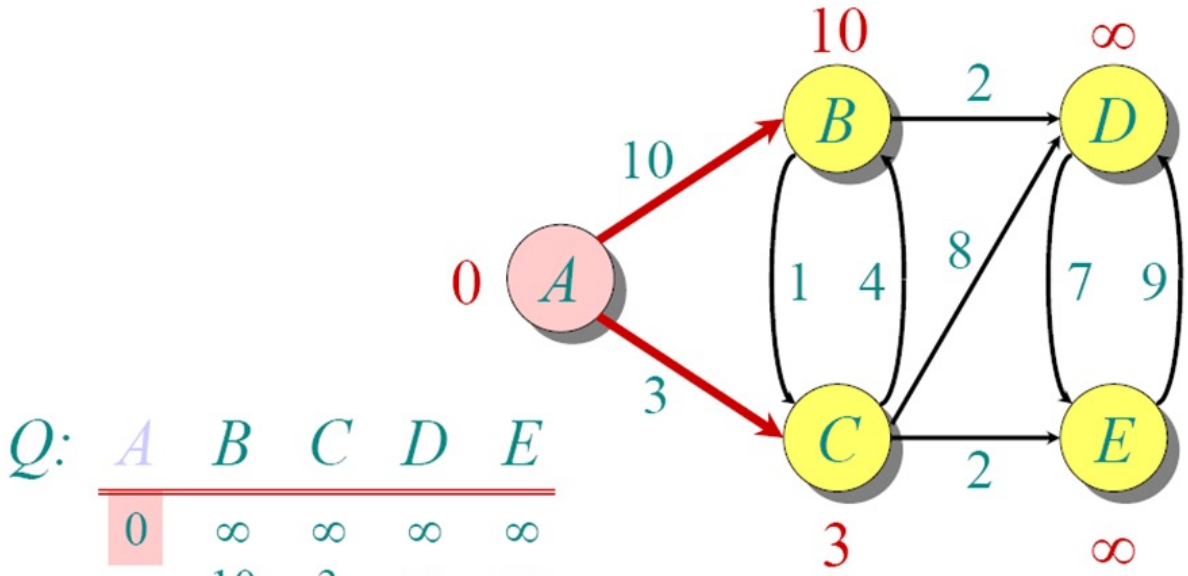
Q:	A	B	C	D	E
	0	∞	∞	∞	∞

S: {}

Exercise: uniform-cost search (Dijkstra)



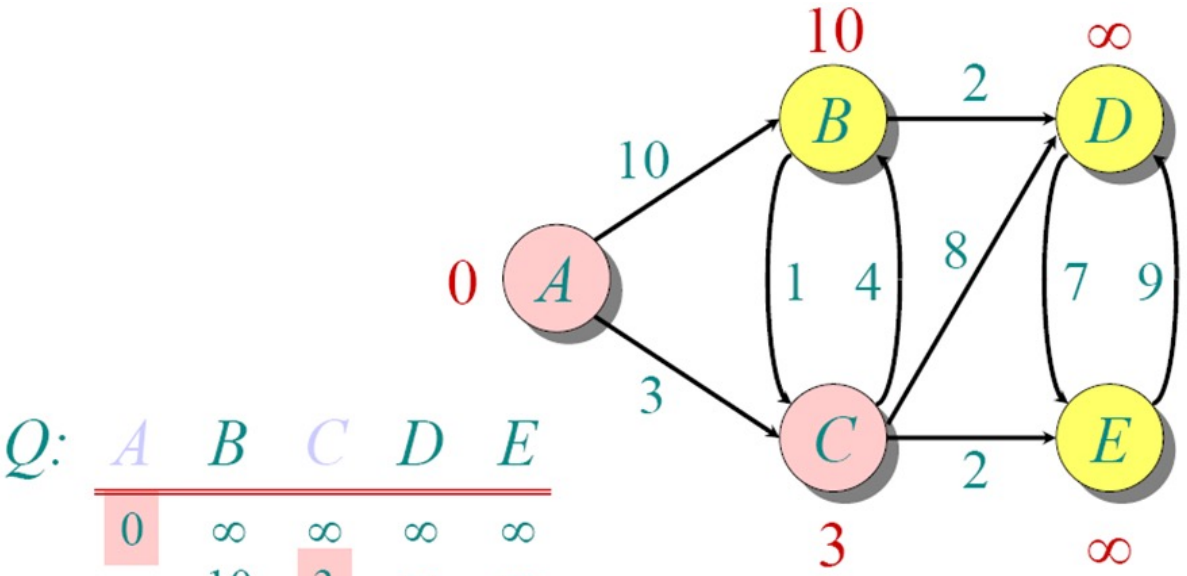
Exercise: uniform-cost search (Dijkstra)



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A }

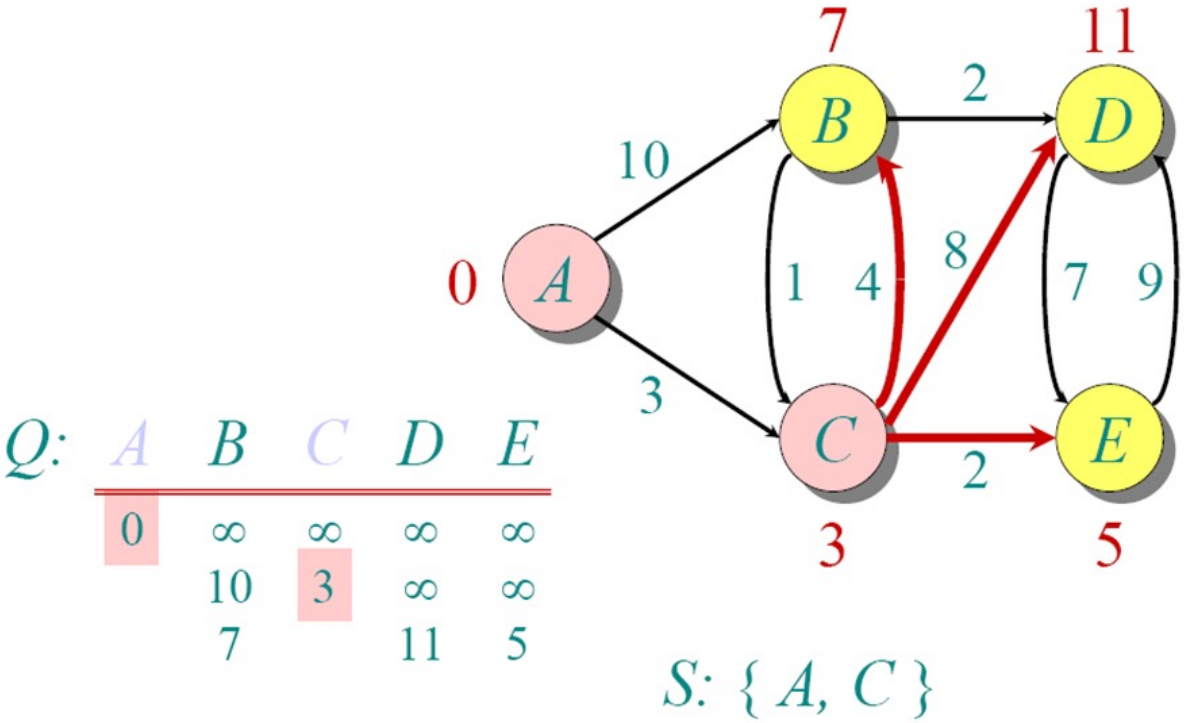
Exercise: uniform-cost search (Dijkstra)



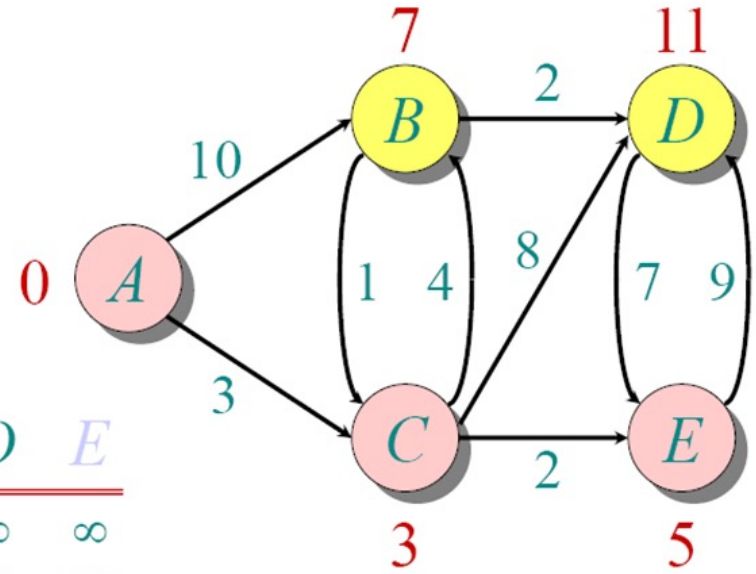
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A, C }

Exercise: uniform-cost search (Dijkstra)



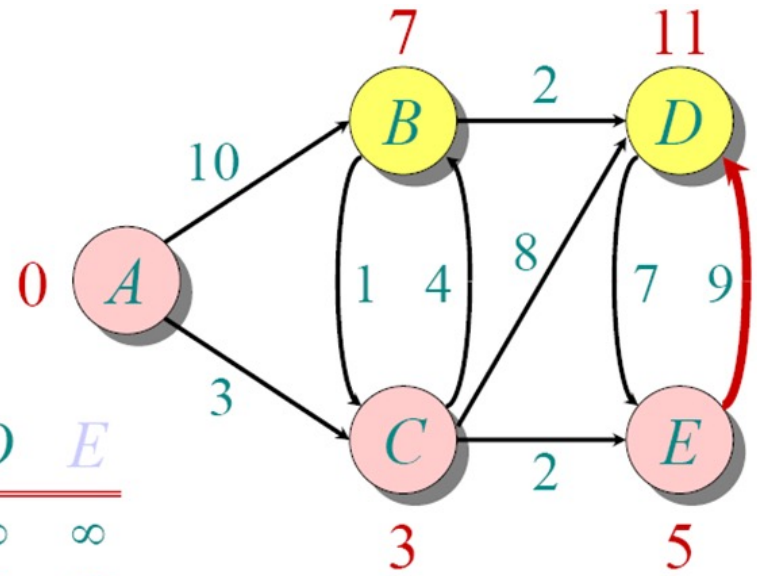
Exercise: uniform-cost search (Dijkstra)



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { A, C, E }

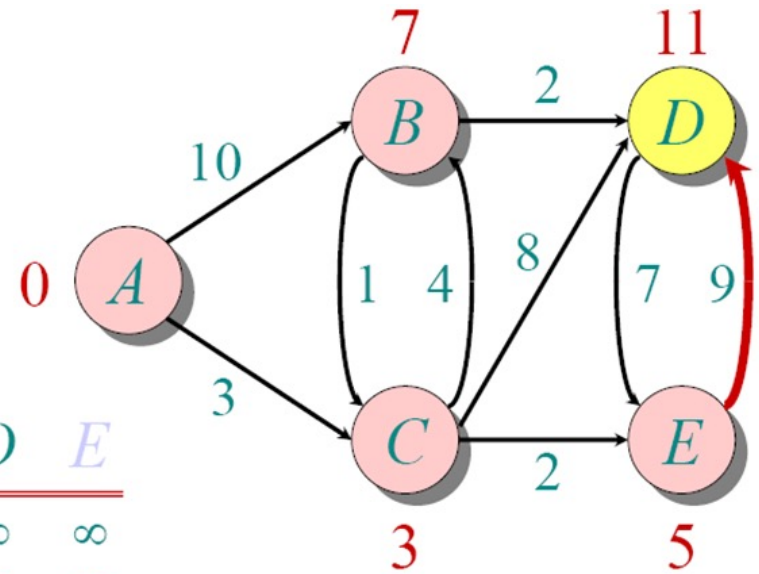
Exercise: uniform-cost search (Dijkstra)



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E }

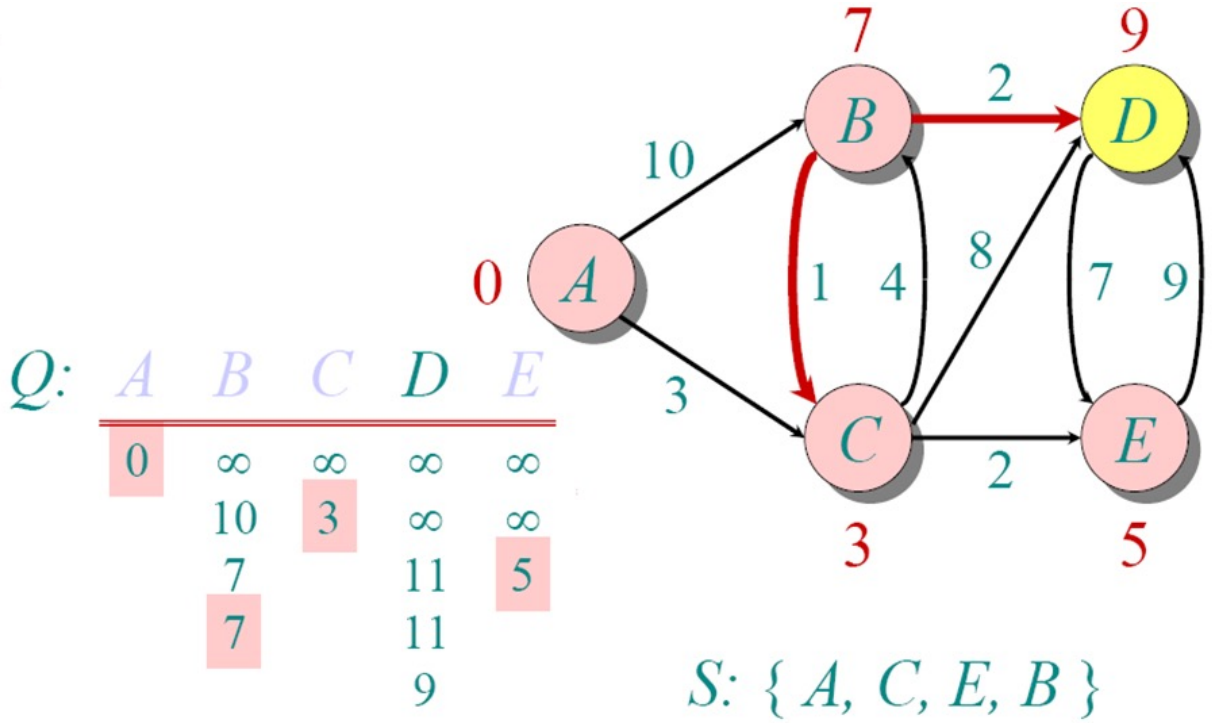
Exercise: uniform-cost search (Dijkstra)



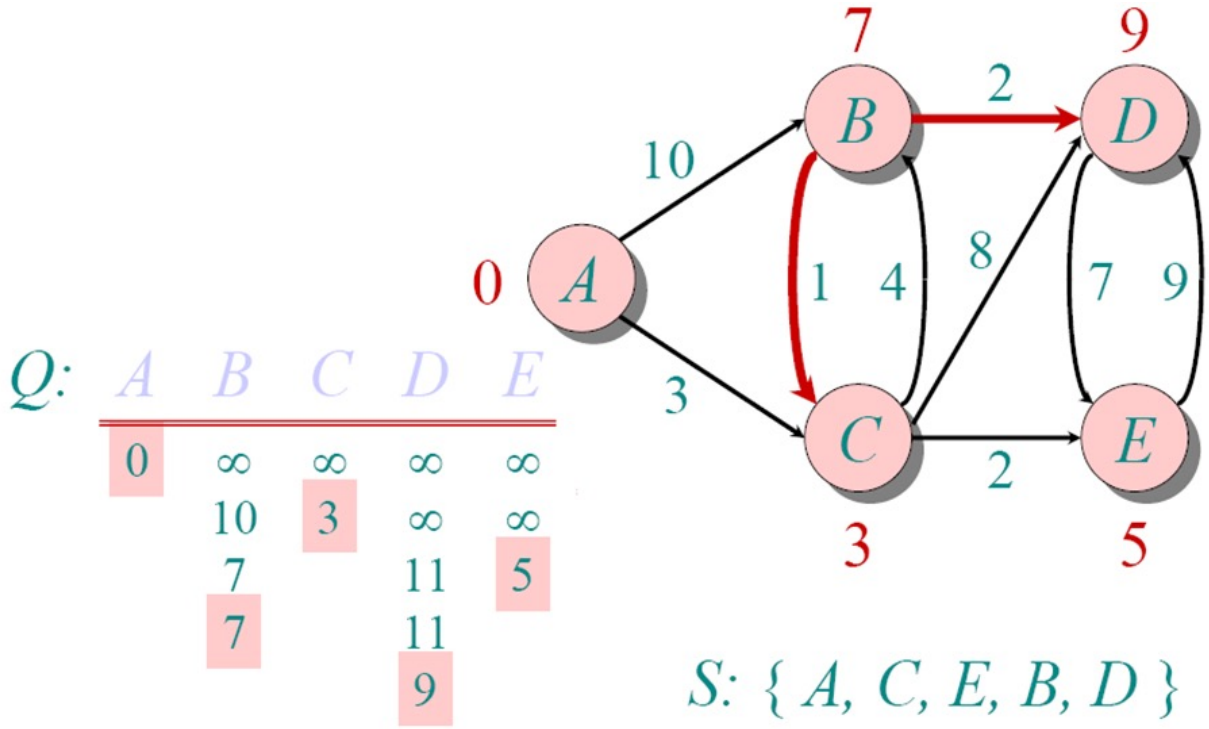
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E, B }

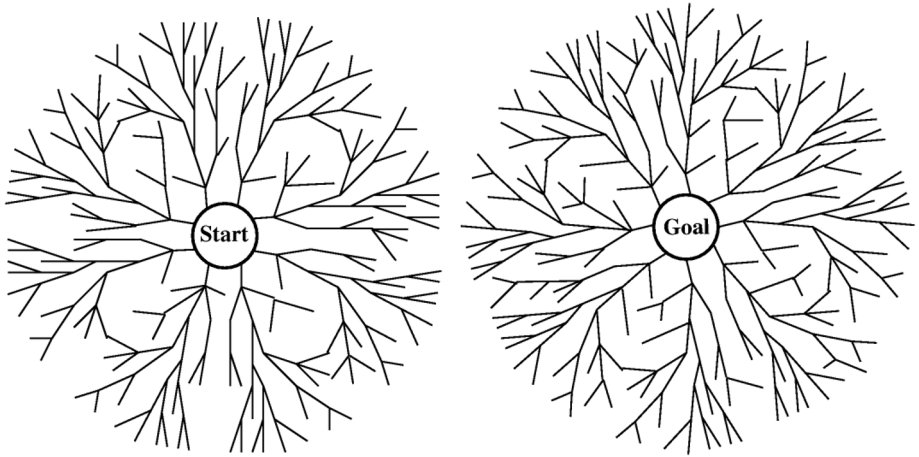
Exercise: uniform-cost search (Dijkstra)



Exercise: uniform-cost search (Dijkstra)



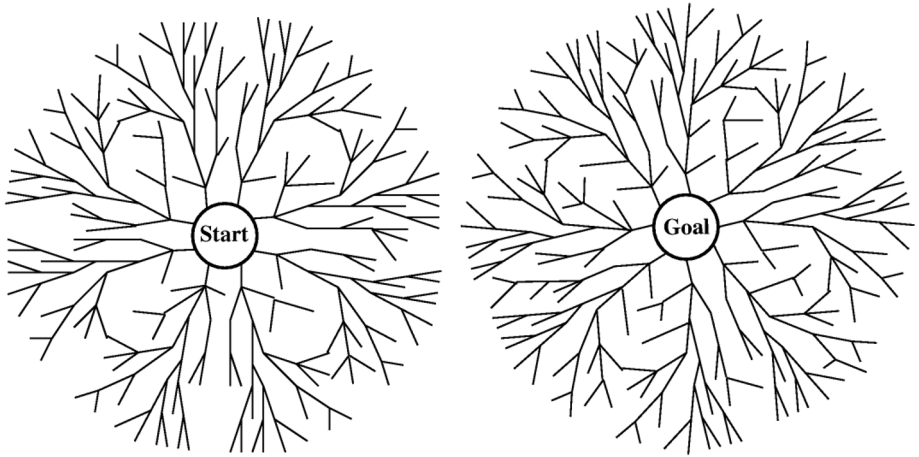
Bidirectional Search



If you can work backward from the solution, then you can limit the search depth
With a solution at depth d, then find a solution in $O(2b^{d/2})=O(b^{d/2})$ steps
Vast improvement over $O(b^{d})$ steps with BFS!

b = 10, d = 6
BFS: 1,111,111 nodes; BDS: 2,222 nodes

Bidirectional Search

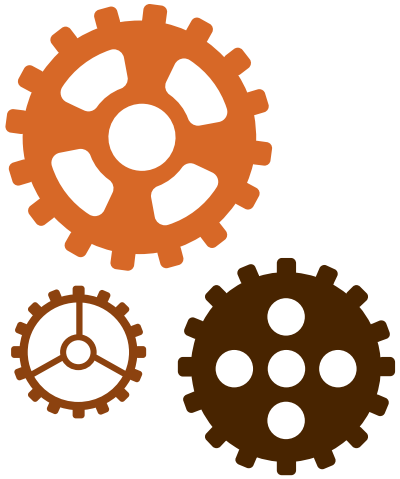


For this to work, we need to keep track of two frontiers and two tables of reached states, and we need to be able to reason backwards: if state ***s***' is a successor of ***s*** in the forward direction, then we need to know that ***s*** is a successor of ***s***' in the backward direction. We have a solution when the two frontiers collide.[9]

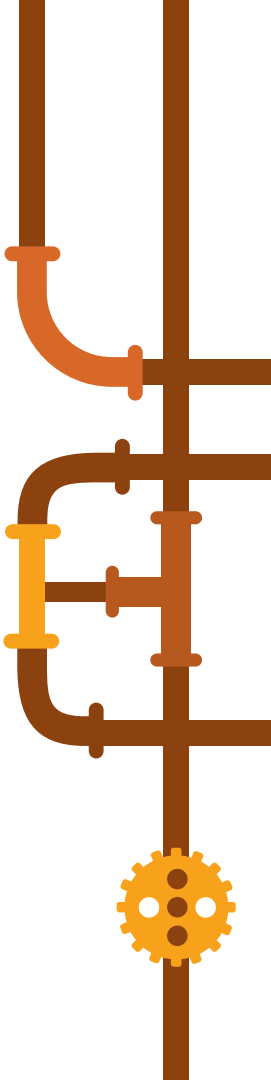
Comparison of Techniques

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Evaluation of search algorithms. b is the branching factor; m is the maximum depth of the search tree; d is the depth of the shallowest solution, or is m when there is no solution; ℓ is the depth limit. Superscript caveats are as follows: [1] complete if b is finite, and the state space either has a solution or is finite. [2] complete if all action costs are $\geq \varepsilon > 0$; [3] cost-optimal if action costs are all identical; [4] if both directions are breadth-first or uniform-cost.



still not as smart as it could be...



Next Chapter

More intelligent search strategies

- Best-first search
- A* search
- Heuristic search

Applications

- Playing games
- Constraint satisfaction problems

Next class: read R & N: chapter 4