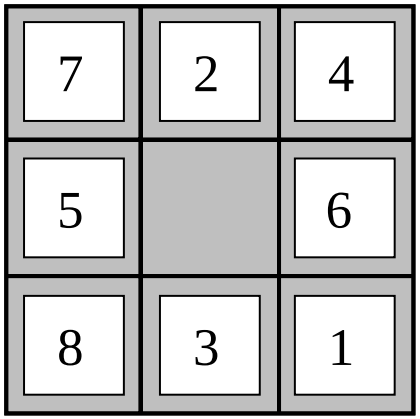
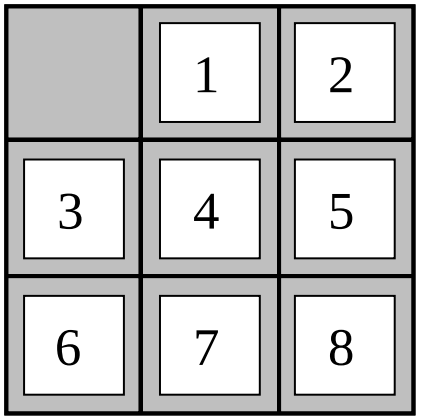
Problem solving and search

Chapter 3

How to Solve a (Simple) Problem



Start State



Goal State

Introduction

Simple goal-based agents can solve problems via searching the state space for a solution, starting from the initial state and terminating when (one of) the goal state(s) are reached.

The search algorithms can be blind or informed (using heuristics).

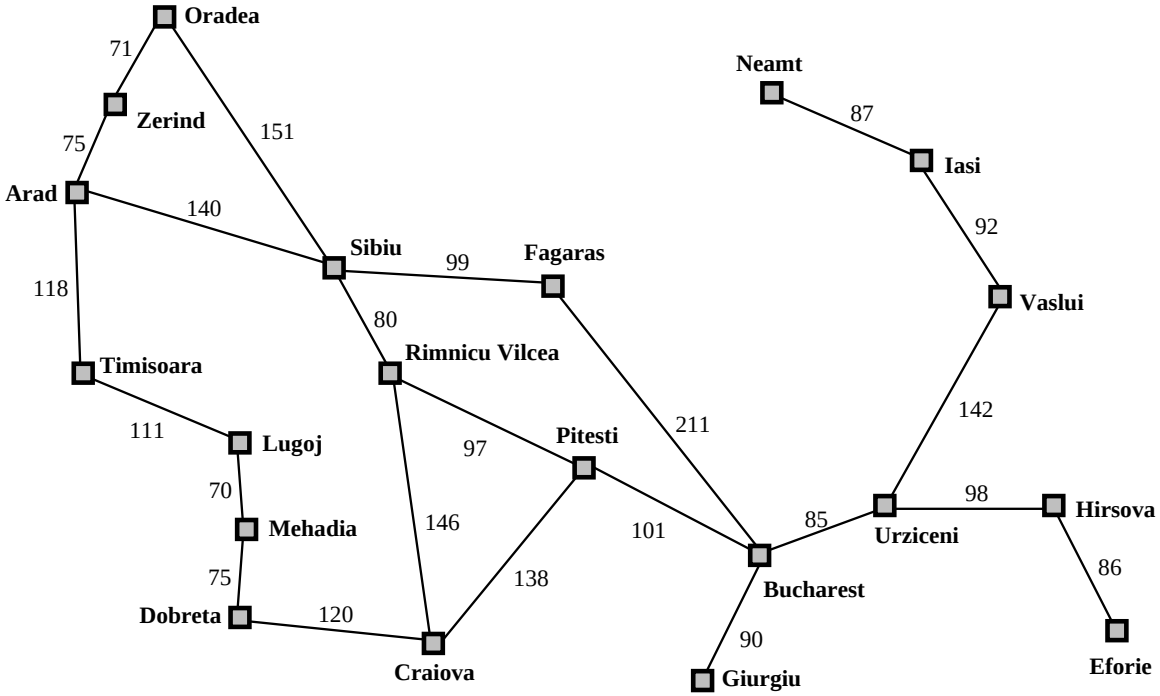
Before we see how we can search the state space of the problem, we need to decide on what the states and operators of a problem are.

$\Rightarrow$ problem formulation

Example: Traveling in Romania

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

You have access to a map.



Example: Romania

Imagine that this is not given as a graph as in here (since you see the solution easily), but as a list of roads from each city to another. This is in fact how a robot will see the map, as a list of edges on a graph (with also associated distances):

Arad to Zerind, Sibiu, Timisoara
Bucharest to Pitesti, Guirgiu, Fagaras, Urziceni
Craiova to Dobreta, Pitesti
Dobreta to Craiova, Mehadia
...
Oradea to Zerind, Sibiu
...
Zerind to Oradea, Arad

Example: Traveling in Romania

Formulate goal:

be in Bucharest

Formulate problem:

Action sequence needs to be in the form of "drive from Arad to ...; drive from ... to ...; ...; drive from ... to Bucharest). Hence the states of the robot, abstracted for this problem are "various cities".

The corresponding operators taking one state to the other are "driving between cities".

Find solution: sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

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.

(Abstract) solution =

set of real paths that are solutions in the real world

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.

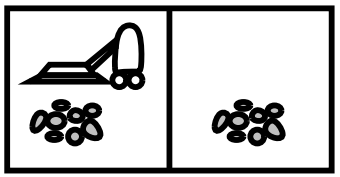
Example: vacuum world

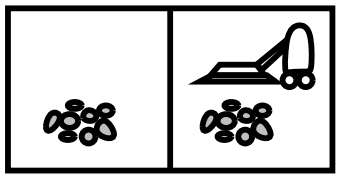
Your robot needs to vacuum a two-room area. Each room may have dirt in it and the robot may be in one of the rooms and move left or right to go to the other room.

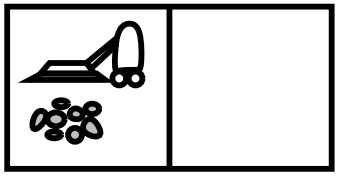
What are the states of this vacuum world?

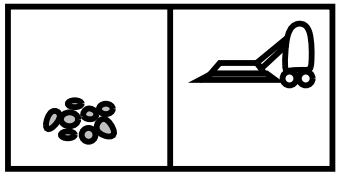
Example: vacuum world

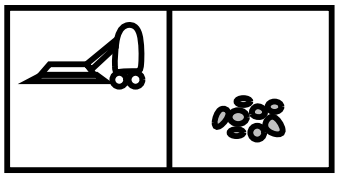
The 8 States:

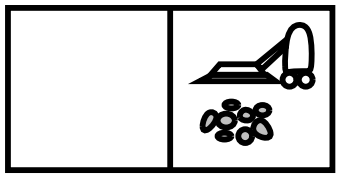
1 

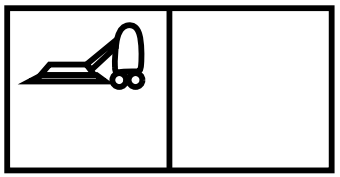
2 

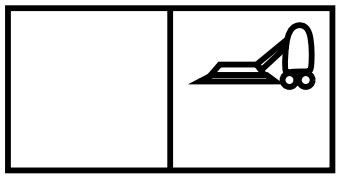
3 

4 

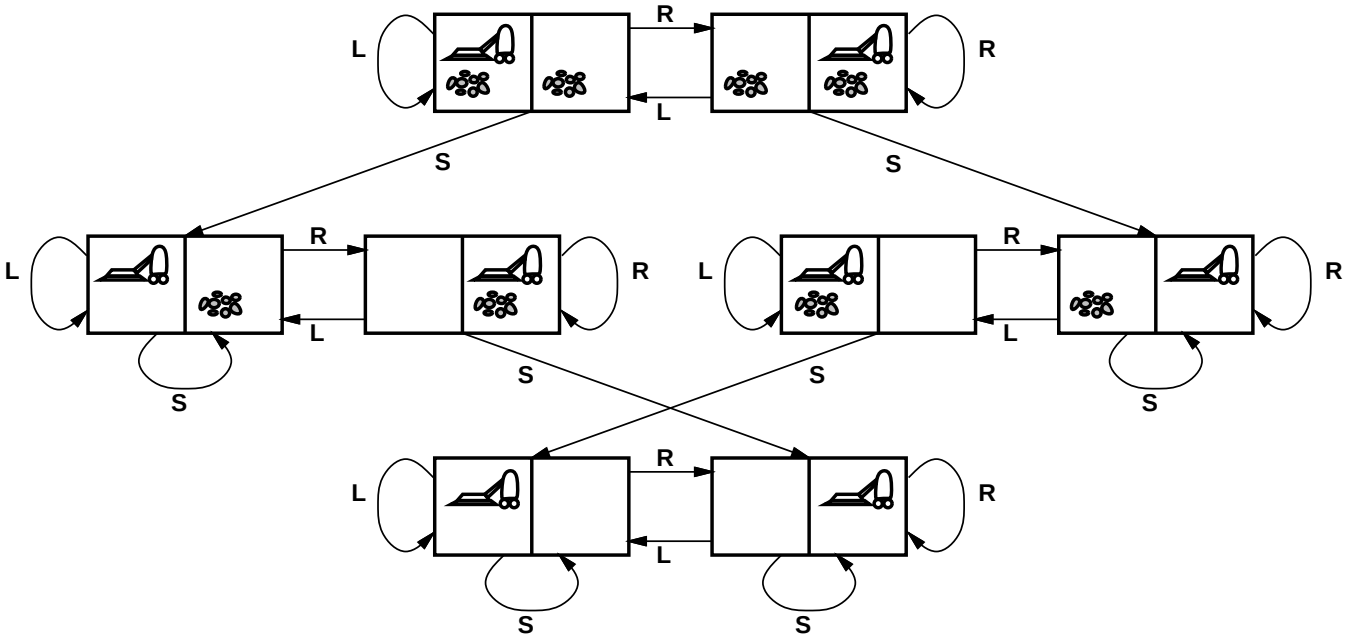
5 

6 

7 

8 

Example: vacuum world



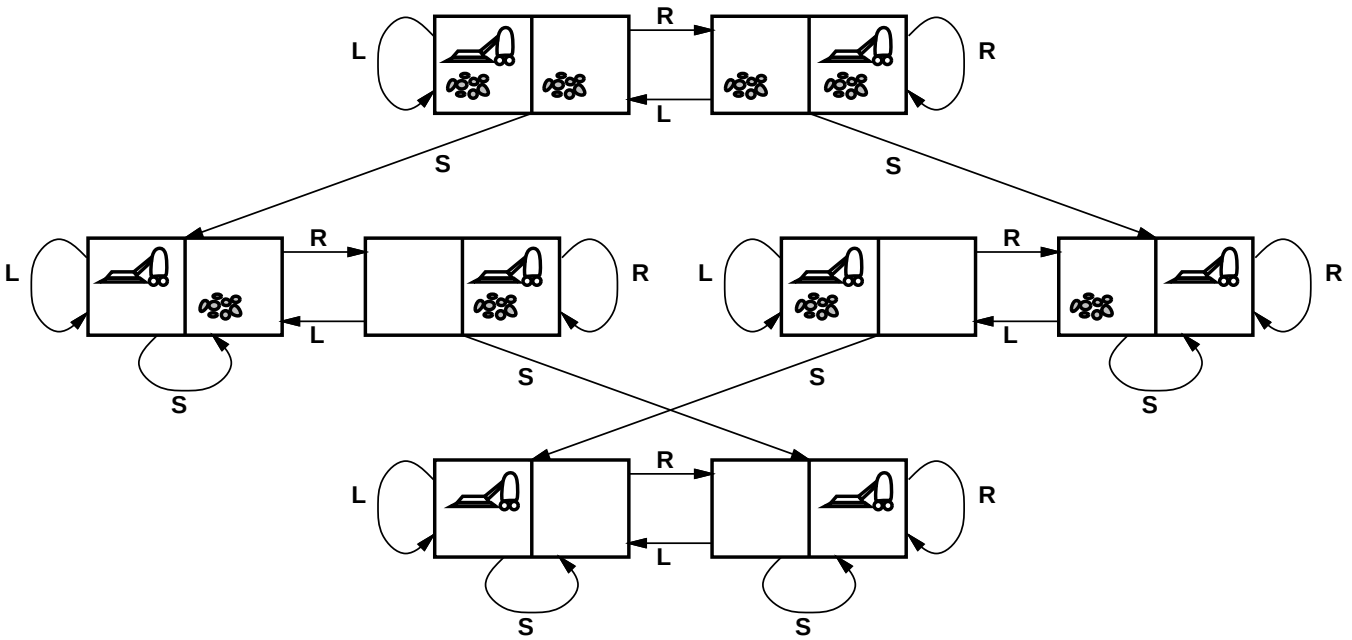
states??

operators??

goal test??

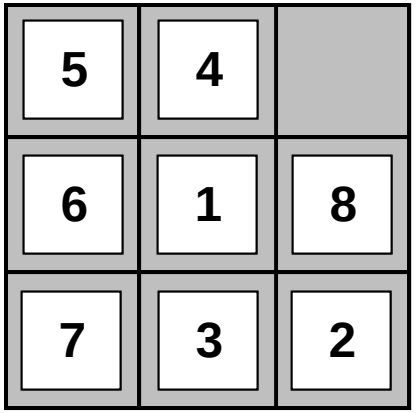
path cost??

Example: vacuum world

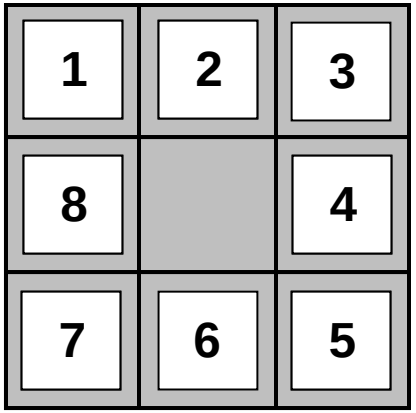


states??: integer dirt and robot locations (ignore dirt *amounts*)
operators??: $Left$, $Right$, $Suck$
goal test??: no dirt
path cost??: 1 per operator

Example: 8-puzzle



Start State



Goal State

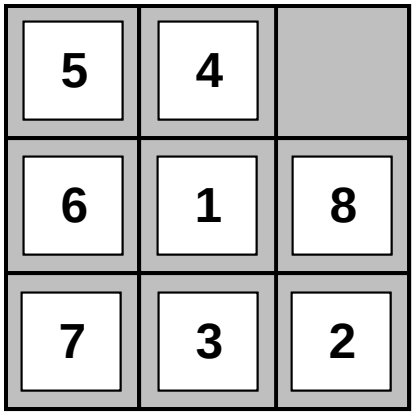
states??

operators??

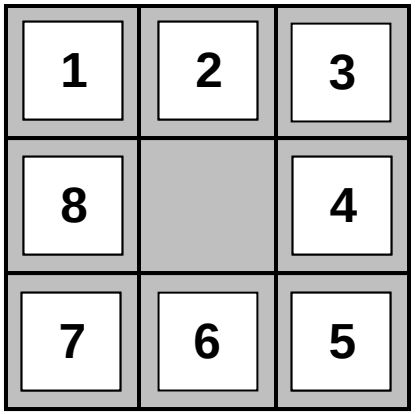
goal test??

path cost??

Example: The 8-puzzle



Start State

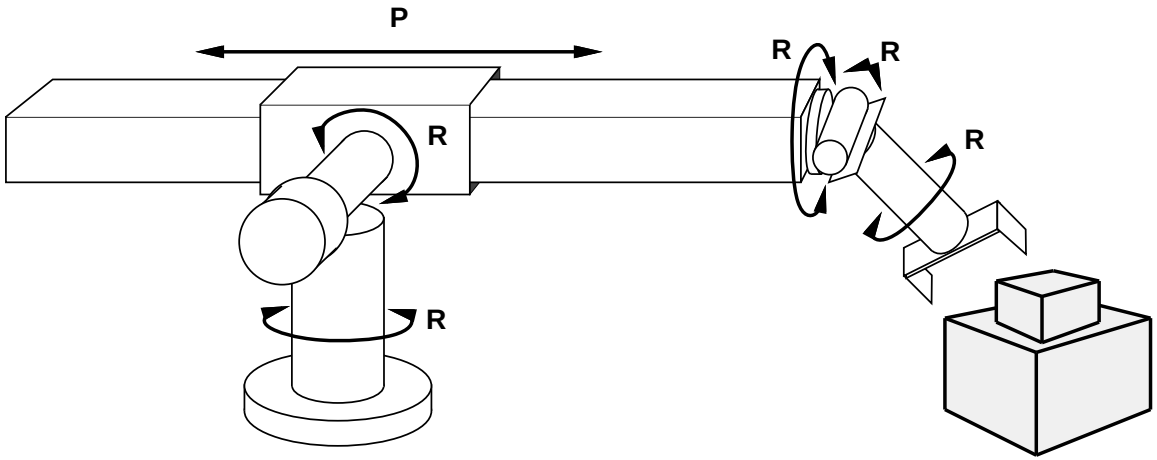


Goal State

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]

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

path cost??: time to execute

Problem-solving agents

Restricted form of general agent, an intelligent agent will solve problems among others):

```
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
```

Implementation of 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*

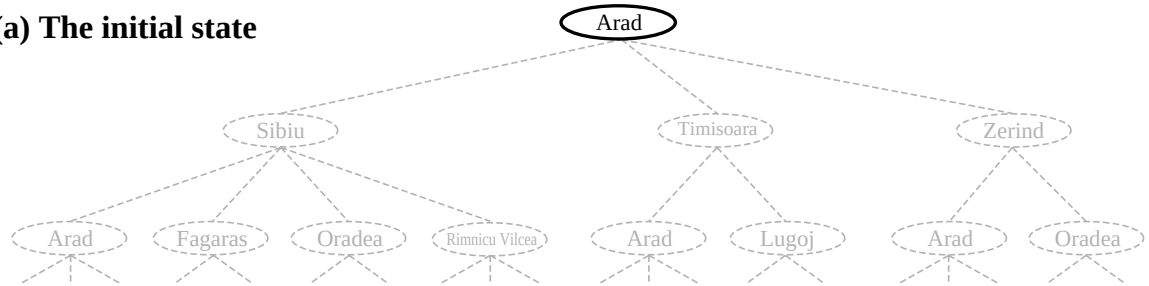
if the node contains a goal state **then return** the corresponding solution

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

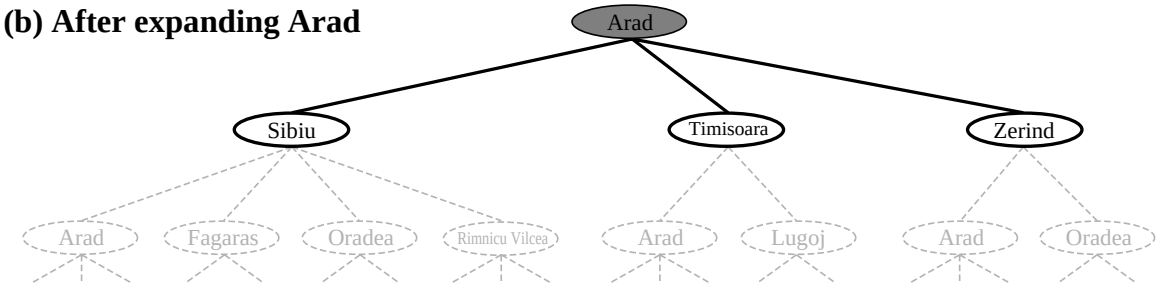
end

Tree search example

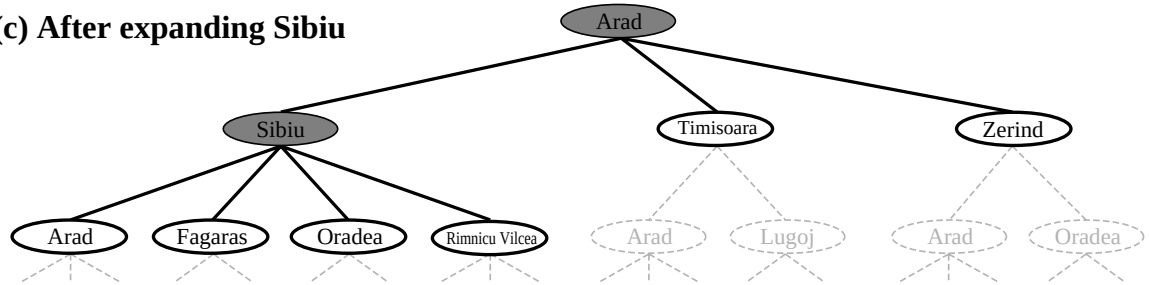
(a) The initial state



(b) After expanding Arad



(c) After expanding Sibiu



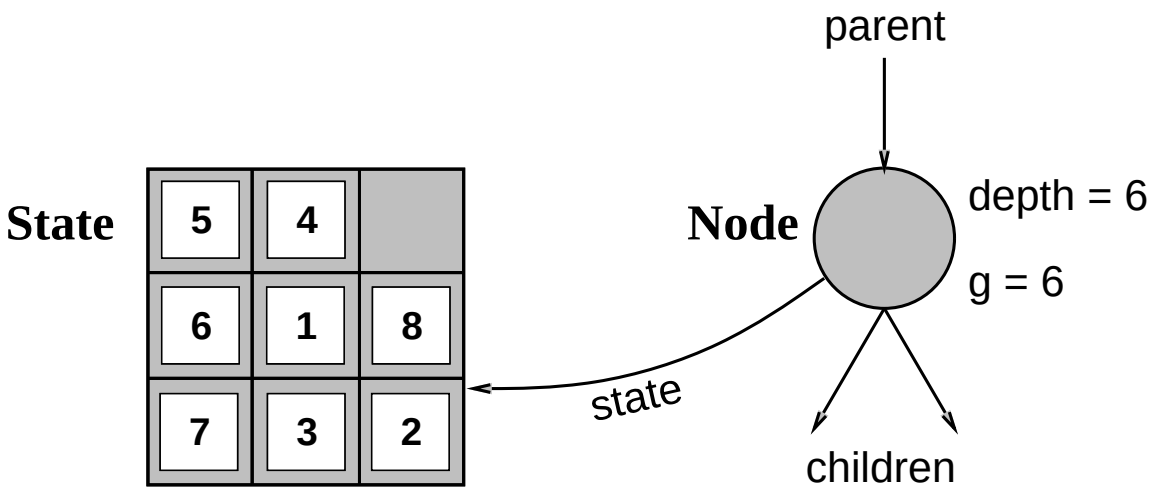
Implementation: states vs. nodes

A *state* is a (representation of) a physical configuration

A *node* is a data structure constituting part of a search tree

includes *state*, *parent*, *children*, *operator*, *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.

Terminology

♢ depth of a node: number of steps from root (starting from depth=0)

♢ path cost: cost of the path from the root to the node

♢ expanding a node: pulling it out from the queue, goal test and expanding (interchangeable with *visiting a node*)

♢ generated nodes: different than nodes expanded!

Implementation of search algorithms

```
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] applied to STATE(node) succeeds return 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
```

Implementation of search algorithms

Notice that we will always take a node from the front of the Queue (called the Fringe), so insertion of the expanded nodes (depending on the Queueing Function) is what distinguishes between different search strategies.

The GeneralSearch (next slide) was the skeleton search algorithm given instead of the TreeSearch in AIMA's (our book) first edition, highlighting the dependence to the Queueing Function:

```
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
```

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—maximum number of nodes generated/expanded

(the slides mostly use visited (goal test and expand if necessary) nodes)

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 (finite)

d—depth of the least-cost solution

m—maximum depth of the state space (may be ∞)

Time Complexity

An algorithm's time complexity is often measured asymptotically. Assume you "process" n items with your algorithm. We say that the time

$T(n)$ of the algorithm is $O(f(n))$ if
(e.g. $T(n)$ is $O(n^2)$)

$$\exists n_0 \text{ such that } T(n) \leq kf(n), \forall n \geq n_0$$

$\diamondsuit$ The time complexity analysis can be done (separately) for the worst case and average case

$\diamondsuit$ In simple terms, it checks what is the dominating factor in the spent time, for large enough problem size (n).

Time Complexity

♢ Some problems can be solved in *polynomial time (P)*. These are considered as "easy" problems (e.g. $O(n), O(logn)$ algorithms.

♢ Some problems do not have a polynomial-time solution, but can be verified in polynomial time if one can guess the solution. They are called *non-deterministic polynomial (NP)* problems.

♢ NP-complete problems: those "harder" NP problems that if you find a polynomial time solution, you can solve all the other NP problems (by reducing one problem into another).

♢ Read Appendix pp.977-979 on time complexity

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

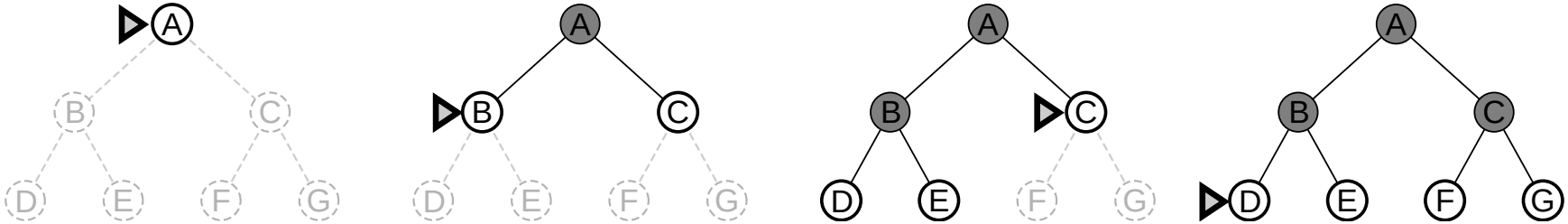
Breadth-first search

Expand shallowest unexpanded node

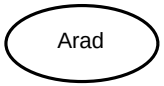
Implementation:

QUEUEINGFN = first in first out (FIFO)

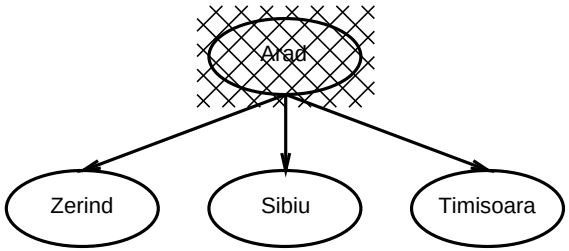
Breadth-first search



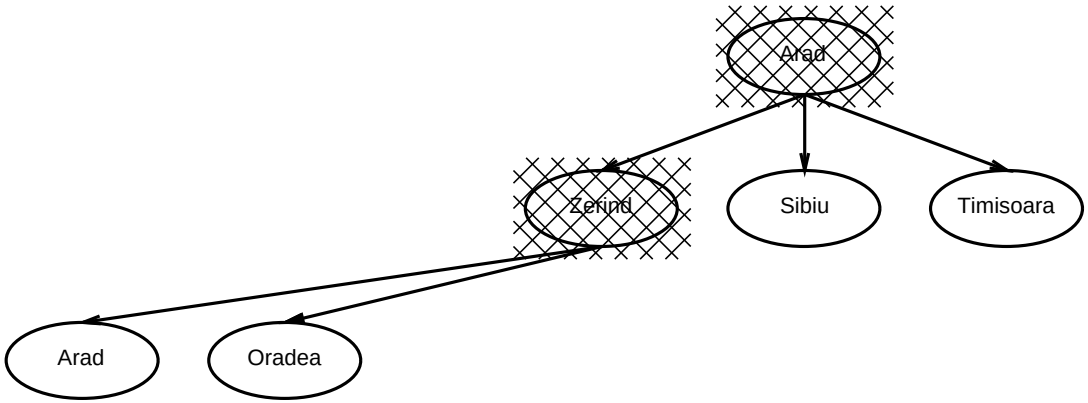
Breadth-first search



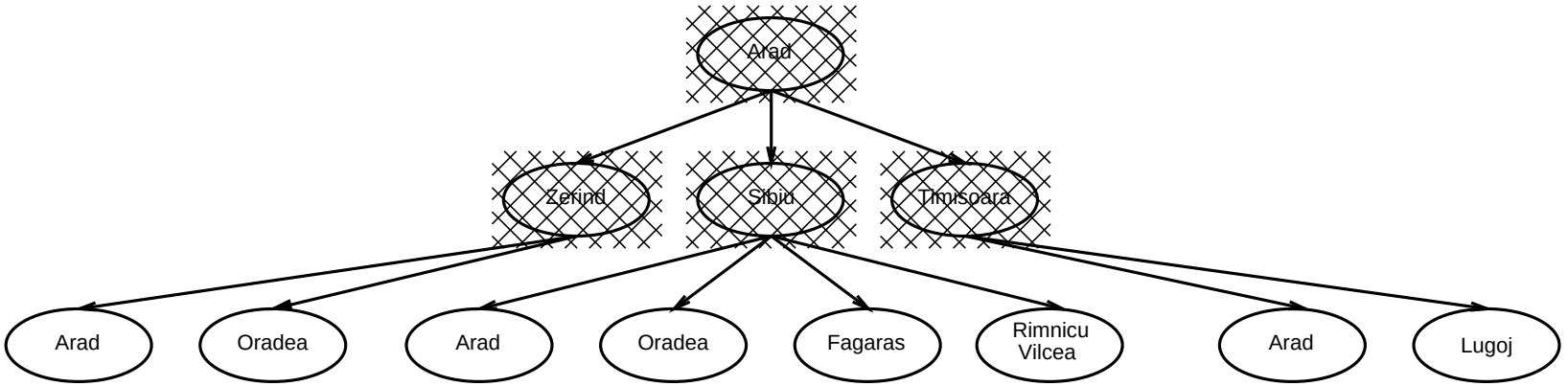
Breadth-first search



Breadth-first search



Breadth-first search



Properties of breadth-first search

Complete??

Time??

Space??

Optimal??

Properties of breadth-first search

Complete?? Yes (if b is finite - otherwise it may be stuck at generating the first level)

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time: ?? $1 + b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b) = O(b^{(d+1)})$, using the basic tree search algorithm.

Time: ?? $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$, using the tree search algorithm that is modified so that last level is not generated in BFS (fig. 3.11, in next slide)

The exact numbers depend on a particular code for the implementation. For instance when the goal test is in the code...

Note: To be precise, we have to specify whether we are talking about visited/expanded or generated nodes.

```
Inaction BREADTH-FIRST-SEARCH (problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL STATE, PATH-COST =0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier) /* chooses the shallowest node in frontier */
      add node.STATE to explored
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          if child.STATE is not in explored or frontier then
              if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
              frontier ← INSERT(child, frontier)
```

Figure 3.11 Breadth-first search on a graph.

Properties of breadth-first search

Space?? $O(b^d)$ for fig.3.11 algorithm

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time: ?? $1 + b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b) = O(b^{(d+1)})$, using the basic tree search algorithm

Time: ?? $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$, using the tree search algorithm that is modified so that last level is not generated in BFS (fig. 3.11)

Space?? $O(b^d)$ for fig.3.11 algorithm

Optimal?? No (Yes if cost $= 1$ per step); not optimal in general

Note: BFS finds the shallowest solution; if the shallowest solution is not the optimal one (step costs are not uniform) than BFS is not optimal.

Time-Space Requirements

Assuming b = 10 and processing speed of 1000 nodes/second (100 bytes/node).

Depth	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	111	.1 seconds	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

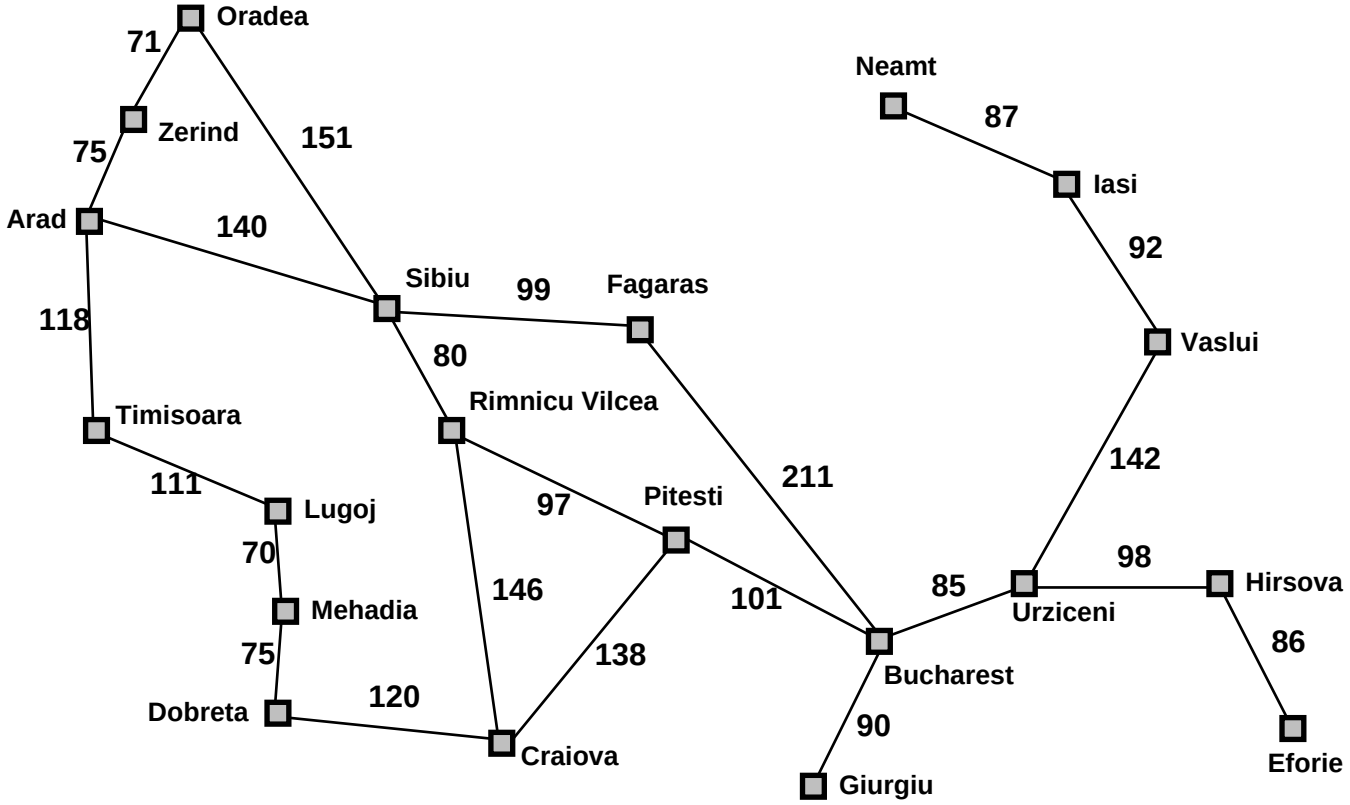
Space is the bigger problem!

Time-Space Requirements

Exponential complexity search problems cannot be solved for all but smallest instances!

Romania with step costs in km

BFS finds the shallowest goal state. What if we have a more general path cost?



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

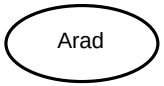
Uniform-cost search

Expand least-cost (path cost) unexpanded node

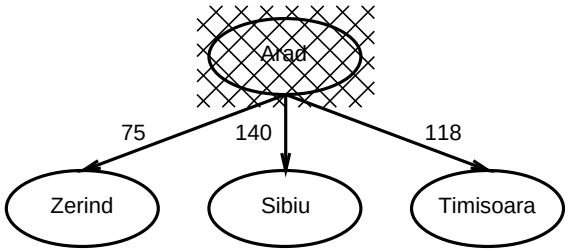
Implementation:

QUEUEINGFN = insert in order of increasing path cost

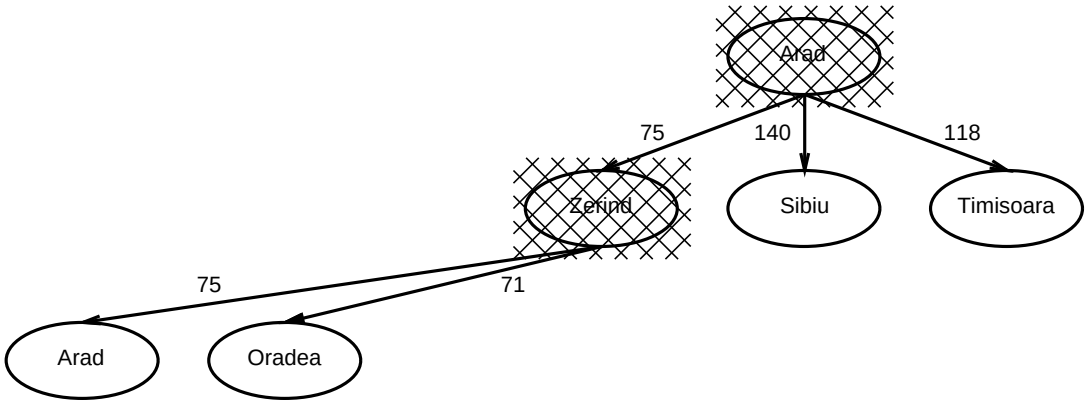
Uniform-cost search



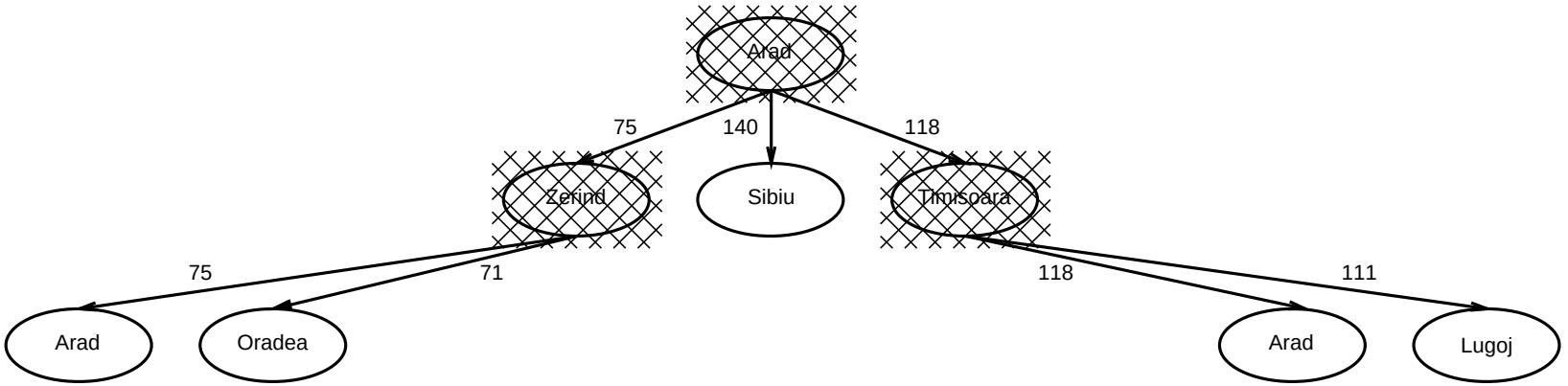
Uniform-cost search



Uniform-cost search



Uniform-cost search



Properties of uniform-cost search

Complete??

Time??

Space??

Optimal??

Note: What would happen if some paths had negative costs?

Properties of uniform-cost search

Complete?? Yes, if step cost $\geq \epsilon$ (nondecreasing)

Time??

Space??

Optimal??

Note: What would happen if some paths had negative costs?

Properties of uniform-cost search

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

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

Space??

Optimal??

If each step costs at least $\epsilon > 0$, then time complexity is $O(b^{\lceil C^*/\epsilon \rceil})$, if the optimum solution has cost C^*

Why?

Properties of uniform-cost search

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

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

Space??

Optimal??

If each step costs at least $\epsilon > 0$, then time complexity is $O(b^{\lceil C^*/\epsilon \rceil})$, if the optimum solution has cost C^*

Why? since the optimum solution would be at a maximum depth of $\lceil C^*/\epsilon \rceil$).

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??

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

Optimality is provided only if we use the algorithm given in Fig. 3.14, which is modified from the basic GRAPH search. Otherwise, it is NOT guaranteed to be optimal.

```
function UNIFORM- COST-SEA RC  (pro blem) returns a solution, or failure
  node 4— a node with STATE = problem.INITIAL STATE, PATH-COST =
  frontier 4— a priority queue ordered by PATH-COST, with node as the only element
  explored 4— an empty set
  loop do
      if EMPTY?( frontier) then return failure
      node ← Poet frontier)  /* chooses the lowest-cost node in frontier e/
      if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
      add node. STATE to explored
      for each action in problem.ACTIONS(node.STATE) do
          child 44— CHILE-NODE(problem, node, action)
          if child . STATE is not in explored or frontier then
              frontier  INSERT( child, frontier)
          else if child.STATE is in frontier with higher PATH-COST then
              replace that frontier node with child
```

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

BFS versus uniform-cost search

Uniform cost search becomes Breadth-first search when the path cost function g(n) is DEPTH(n)

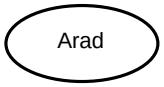
Equivalently, if all the step costs are equal.

Depth-first search

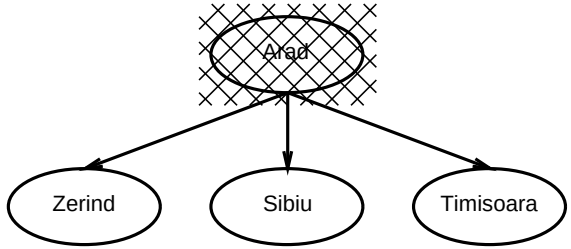
Expand deepest unexpanded node

Implementation:

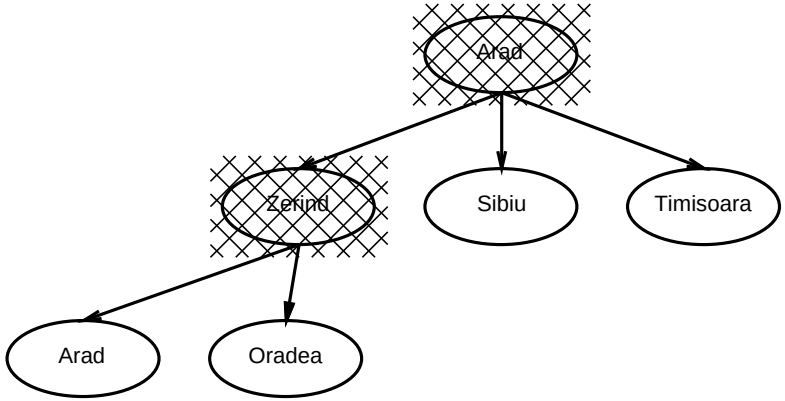
$\textsc{QueueingFn}$ = last in first out (LIFO)



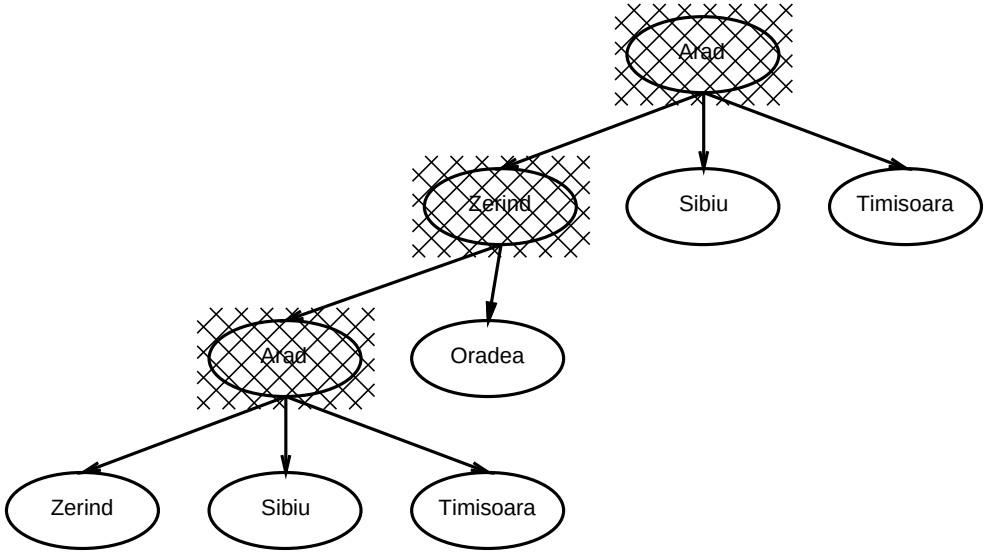
Depth-first search



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??

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??

Optimal??

Notice here that you can find the big-Oh answer by considering the number of nodes in the last level that needs to be considered.

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??

Why? Calculate the size of the Queue assuming that the left-most branch has the maximum depth, m. Now reason that the Queue will never get bigger, wherever the solution may be.

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

BFS vs DFS

Notice that if the problem does not have the issue of very long (possibly infinite) paths, DFS is very advantageous! It has very small memory requirements and it is very easy to program.

Depth-limited search

$=$ depth-first search with depth limit l:
nodes at depth l are treated as if they have no successors

E.g. when we know that there are 20 cities on the map of Romania, there is no need to look beyond depth 19. Compare with the diameter of a problem.

Implementation:

Nodes at depth l have no successors

Depth-limited search - properties

Similar to DFS.

Complete?? yes, if $l \geq d$

Time?? $O(b^l)$

Space?? $O(bl)$

Optimal?? No

Code used:

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

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

Figure 3.17 A recursive implementation of depth-limited tree search.

Iterative deepening search

Can we do away with trying to estimate the limit?

Iterative deepening search

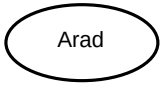
function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution sequence

inputs: *problem*, a problem

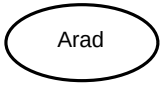
for *depth* $\leftarrow$ 0 **to** ∞ **do**
  result $\leftarrow$ DEPTH-LIMITED-SEARCH(*problem*, *depth*)
  if *result* $\neq$ cutoff **then return** *result*
end

cutoff: no solution within the depth-limit
Failure: no solution at all

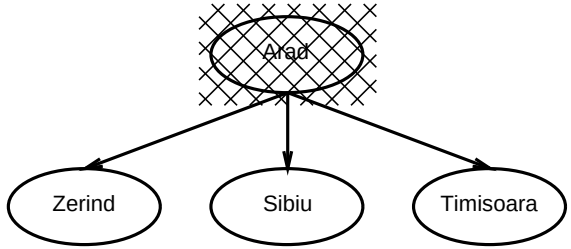
Iterative deepening search $l = 0$



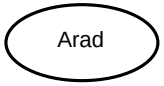
Iterative deepening search $l = 1$



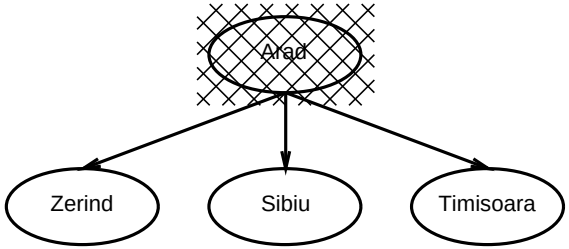
Iterative deepening search $l = 1$



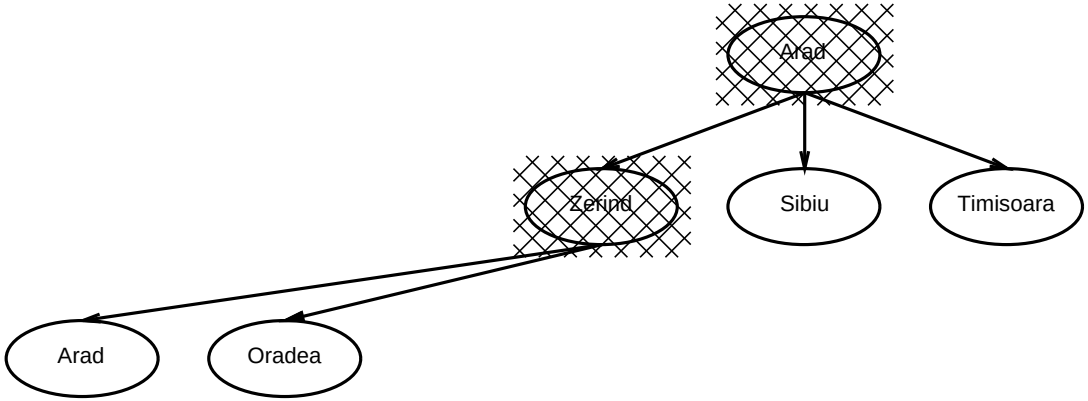
Iterative deepening search $l = 2$



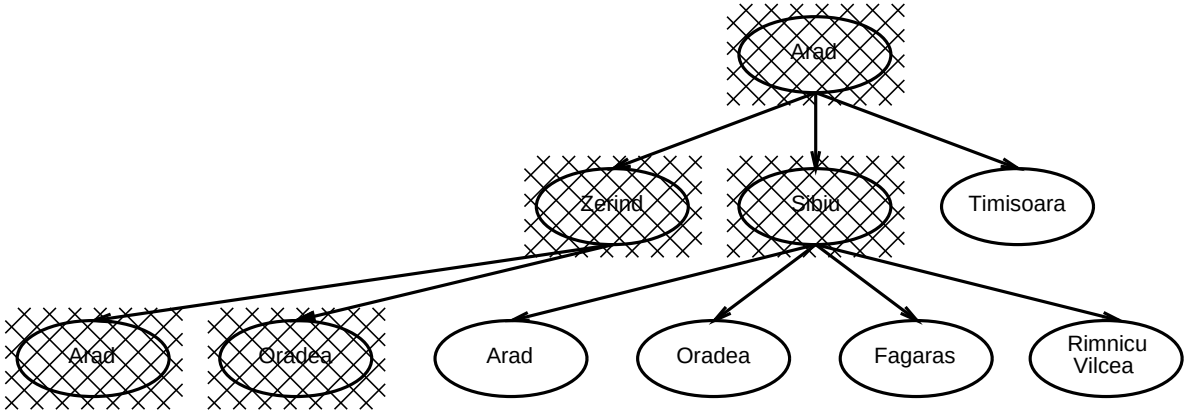
Iterative deepening search $l = 2$



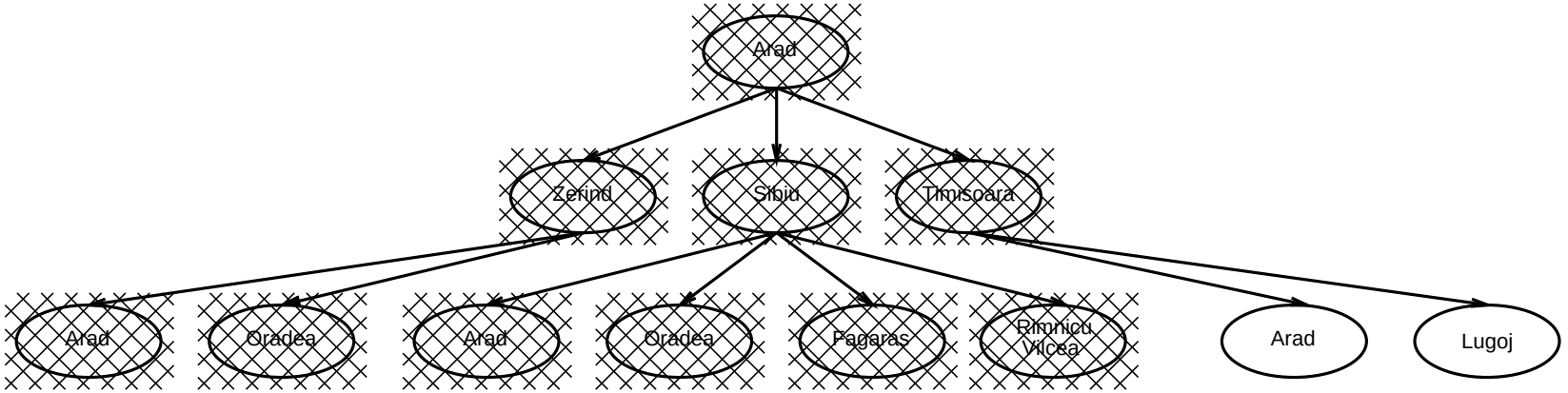
Iterative deepening search $l = 2$



Iterative deepening search $l = 2$



Iterative deepening search $l = 2$

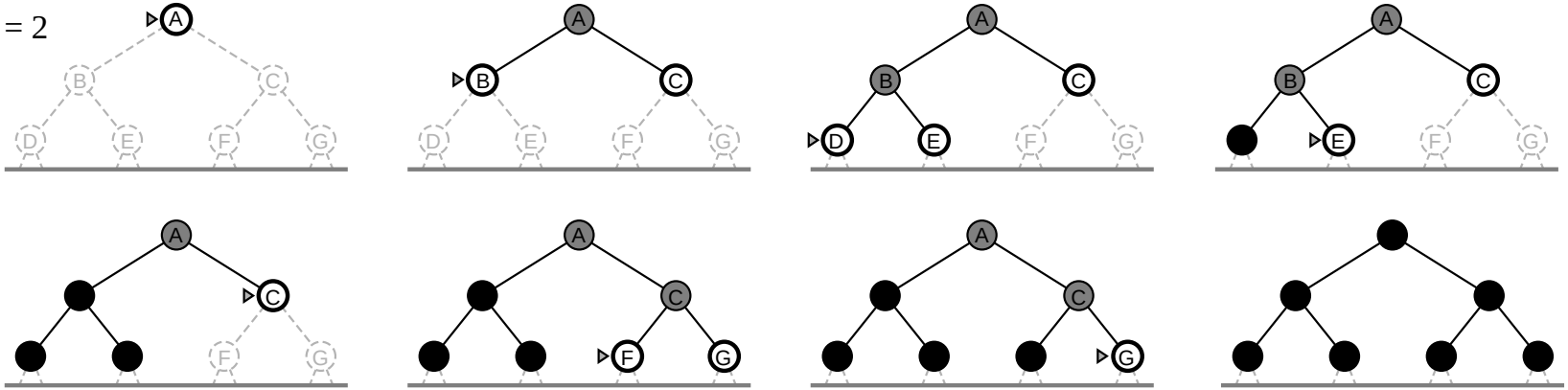


Iterative deepening search

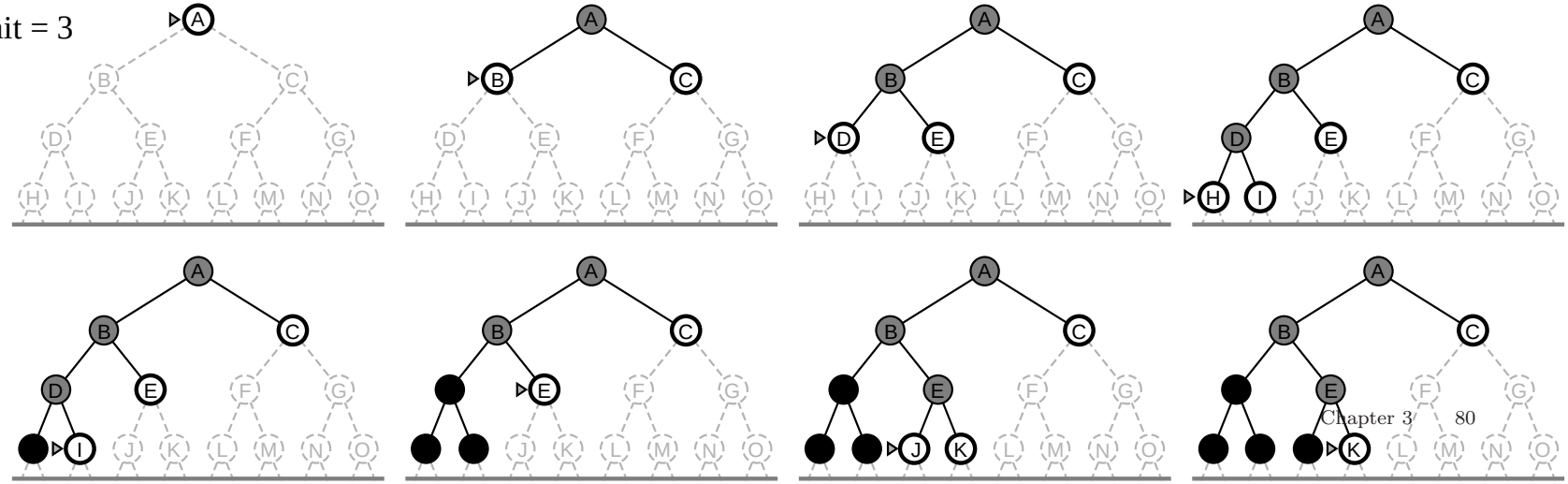
Limit = 0

Limit = 1

Limit = 2



Limit = 3



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

Properties of iterative deepening search

The higher the branching factor, the lower the overhead of repeatedly expanded states (number of leaves dominate).

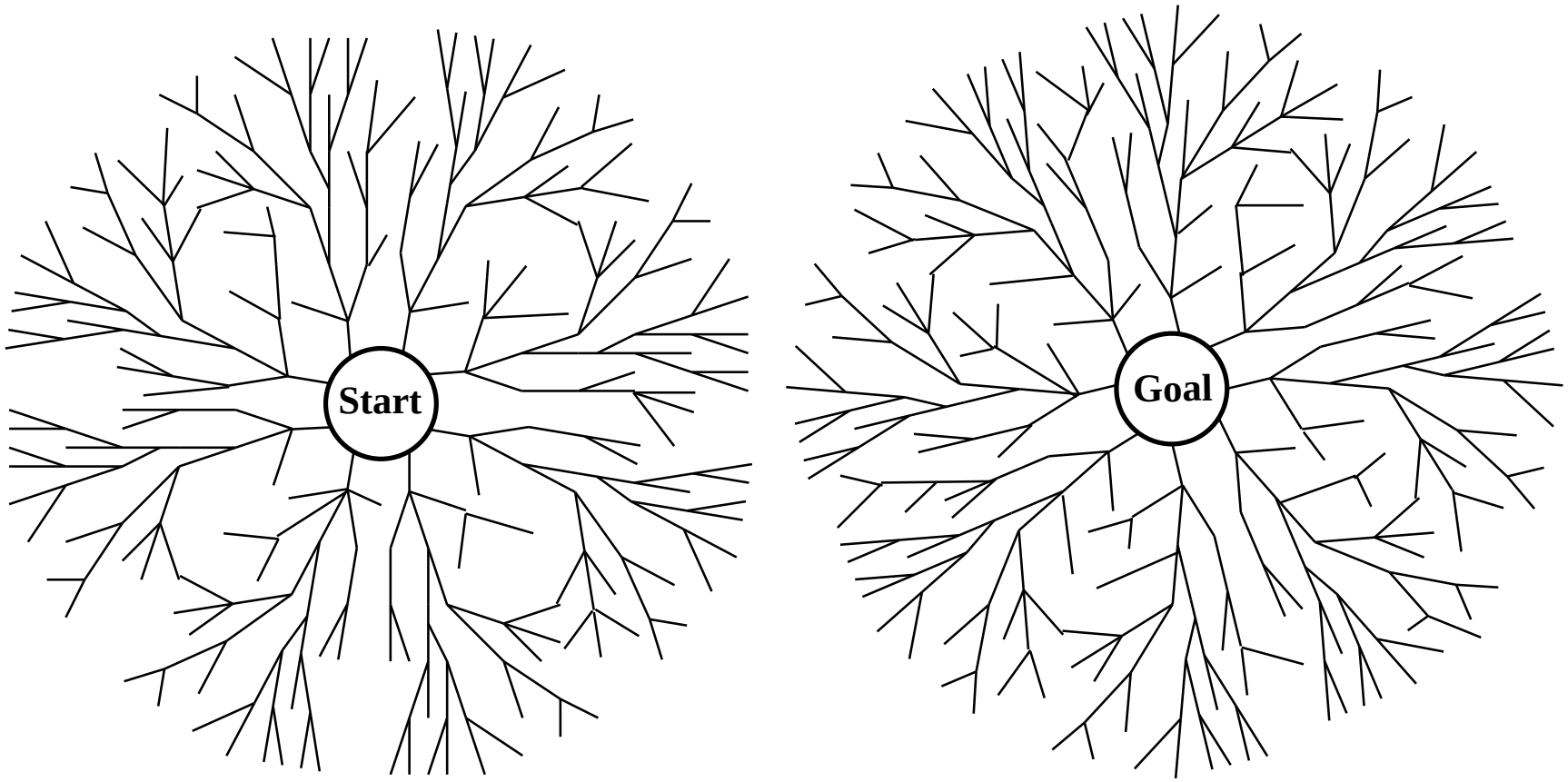
Number of *generated nodes for* $b = 10$ *and* $d = 5$ *:*

$$N(\mathit{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$
$$N(\mathit{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100$$

Preferred method when there is a large search space and the depth of the solution is not known.

Bidirectional search

Simultaneously search both forward from the initial state and backward from the goal state.



Bidirectional search

Need to define predecessors

Operators may not be reversible

What if there are many goal states?

Bidirectional search

Time?

Space?

Bidirectional search

Time? $O(b^{d/2})$

Space? $O(b^{d/2})$

For b=10, d = 6, BFS vs. BDS: million vs 2222 nodes .

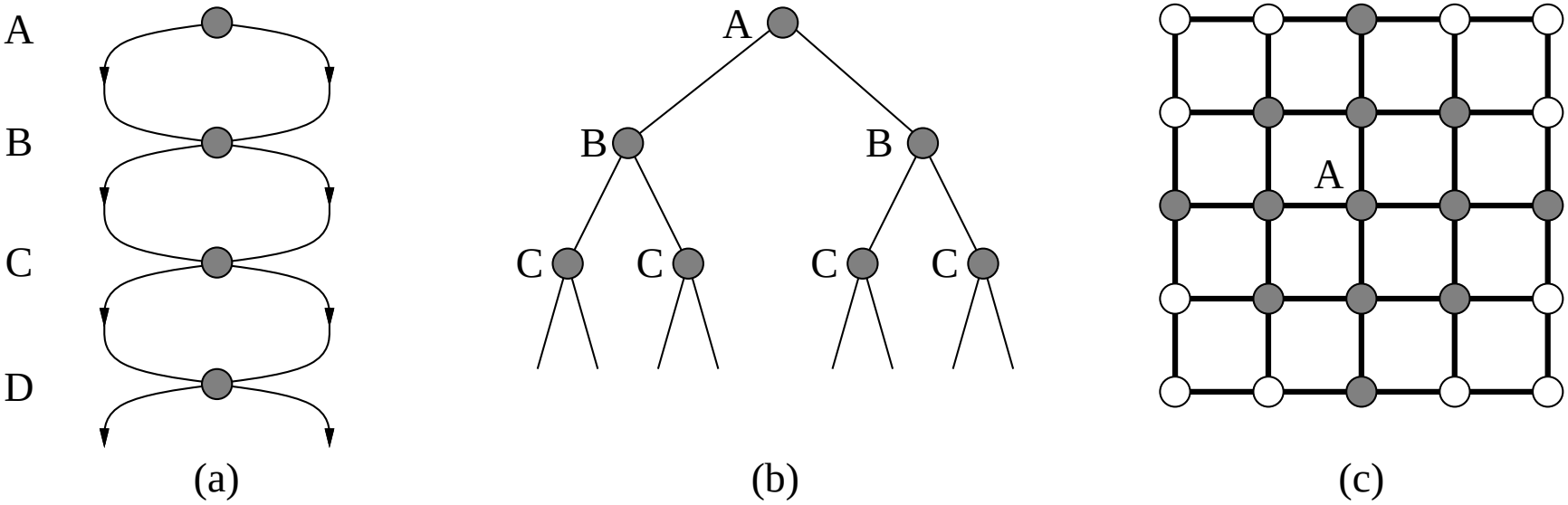
Summary of algorithms

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

Note that conditional Yes^* *and No are not that different: they both do not guarantee completeness, only differ in the strength of the assumptions (b is finite or the max. depth is finite etc.)*

Repeated states

Failure to detect repeated states can turn a linear problem into an exponential one, even for non-looping problems!



Solution: Remember every visited state using a graph search.

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 node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
    end
```

Compare to tree search!

Graph search

Problems with Graph Search:

♢ *Memory Requirements: Increased space requirements for Depth-First search (keep track of states to check for repetition!)*

♢ *Optimality: Basic Graph search deletes the later found path to a repeated state, which could be the path with a shorter cost according to the chosen search strategy (e.g. in iterative deepening, unless modifications are made).*

This was the reason why graph search was modified to guarantee optimality of Uniform Cost in graphs (where a previously found node is replaced with a smaller cost one), in Fig. 3.14

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