Problem solving and search

Chapter 3, Part B: Blind Search Algorithms and Their Analysis

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

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

Core idea: The highest-order term dominates and is given in O(...).
E.g. $1 + b + b^2 + b^3$ is O(b^3)

E.g. $T(n)$ is $O(n^2)$ if $T(n) = 5n^2 + n$

$\diamondsuit$ In ignores what happens for small n (ignoring what happens when $n < n_0$)

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

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

Core idea: We are interested in algorithms that work in polynomial time (of the input parameters such as b, d, m)

Uninformed search strategies

Uninformed strategies use only the information available in the problem definition

- ♢ Breadth-first search
- ♢ Depth-first search
- ♢ Depth-limited search
- ♢ Iterative deepening search
- ♢ Uniform-cost search
- ♢ Bidirectional search

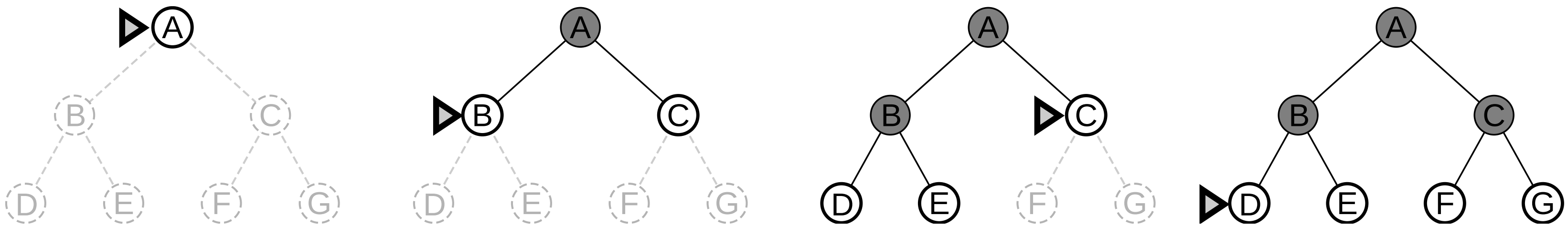
Breadth-first search

Search strategy : Expand the **shallowest** unexpanded node.

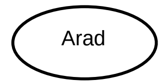
Implementation:

QueueingFn = first in first out (FIFO Queue)

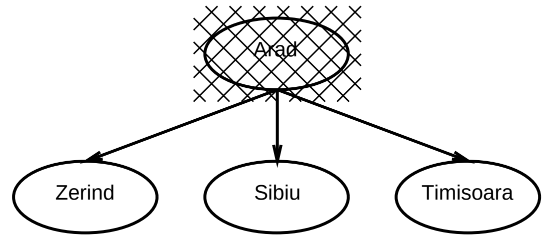
Breadth-first search



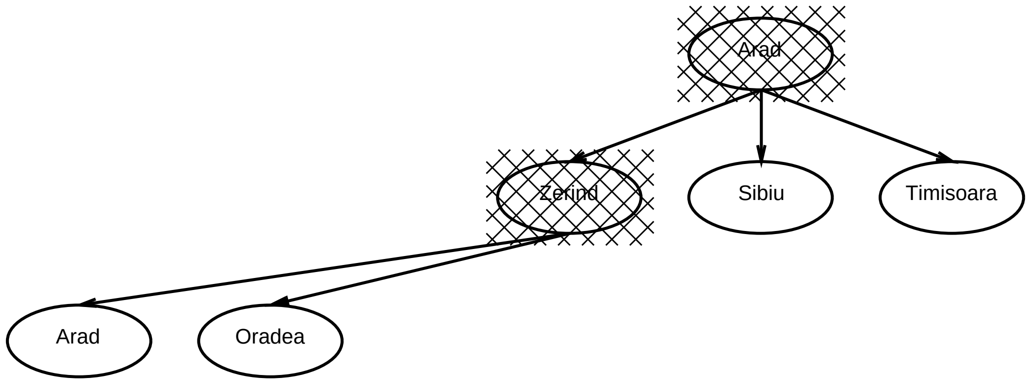
Breadth-first search



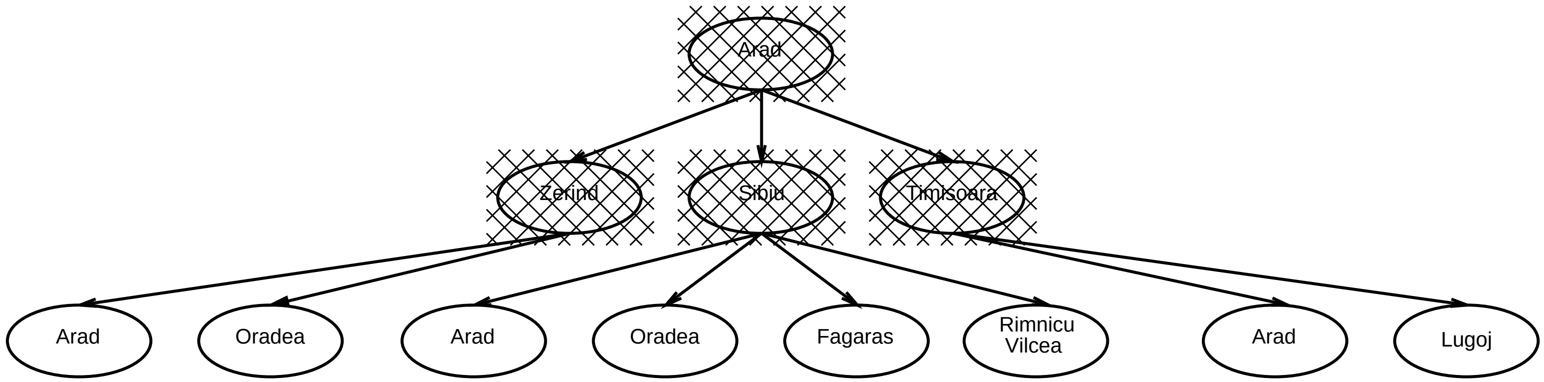
Breadth-first search



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)

Time??

Space??

Optimal??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time ?? $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$ visited nodes (exponential in d)

Time ?? $b + b^2 + b^3 + \ldots + b^d + (b^{(d+1)} - b) = O(b^{(d+1)})$ generated nodes (exponential in $d + 1$) - does not count start state as generated - but we won't care about 1 less one more.

◇ The book sometimes gives time complexity in terms of visited and sometimes in terms of generated nodes. But I will make a distinction between visited and generated nodes so as to be precise and so that you understand the examples in the book.

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., visited nodes is exponential in d

Time ?? $b + b^2 + b^3 + \ldots + b^d + (b^{(d+1)} - b) = O(b^{(d+1)})$, i.e., generated nodes is exponential in $d + 1$

Space?? $O(b^{(d+1)})$

All nodes in the last level (d) are already explored, so level (d+1) is generated.

We also want a precise sum sometimes to see if you understand the tree search algorithm.

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., visited nodes is exponential in d

Time ?? $b + b^2 + b^3 + \ldots + b^d + (b^{(d+1)} - b) = O(b^{(d+1)})$, i.e., generated nodes is exponential in $d + 1$

Space?? $O(b^{(d+1)})$

Optimal?? 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

With even small depths (d=12), both time and space are problematic.

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

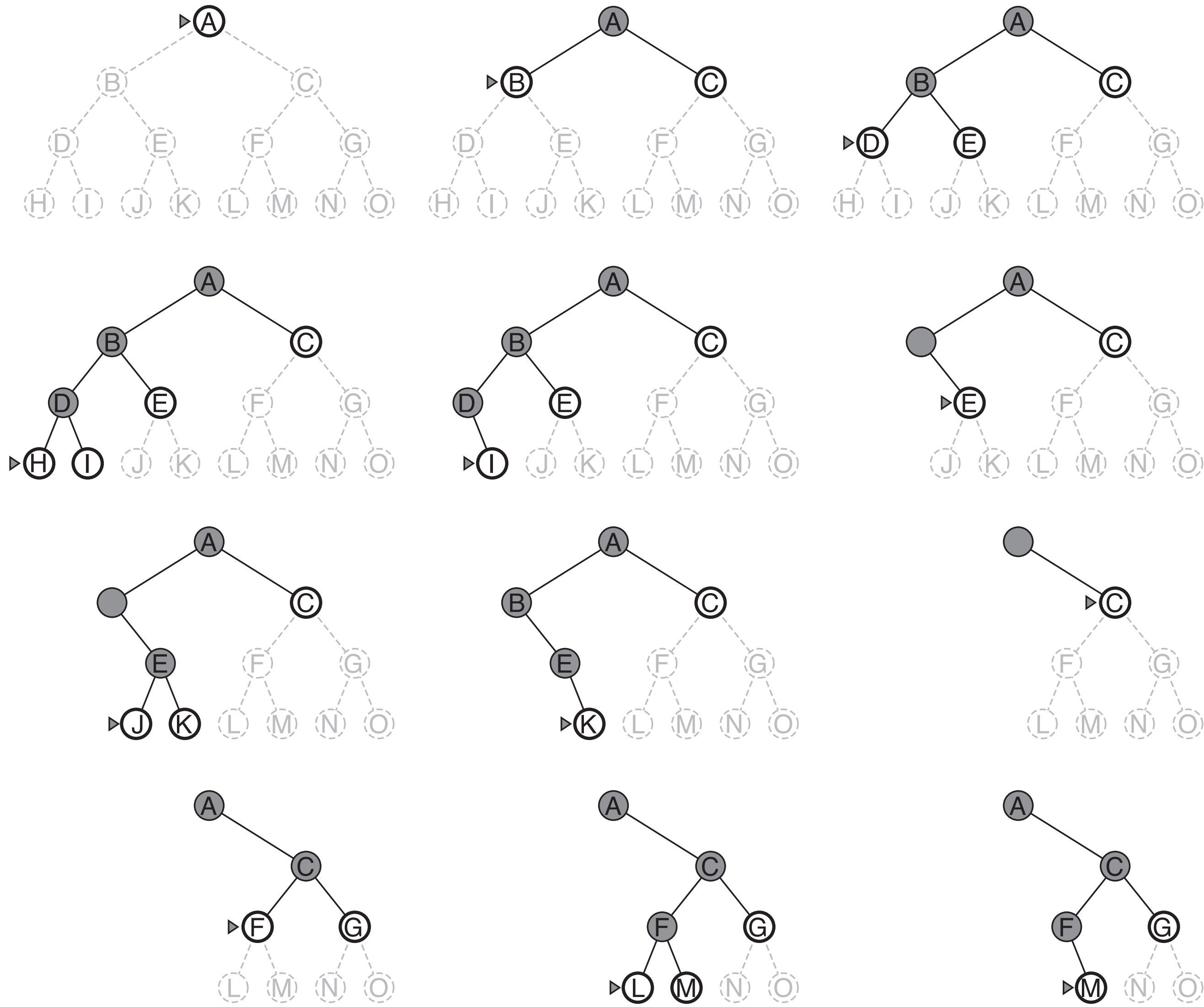
Depth-first search

Search strategy : Expand the **deepest** unexpanded node.

Implementation:

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

Depth-first search



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??

Repeated States

♢ To avoid repeated state, we need to keep a separate data structure to keep track of already explored state. We will call this **explored set**. Sometimes also called **closed set**.

♢ The size of this structure can be as big as the number of different possible states. So we will put a check when a particular state is visited, so we can tell later on if it is a repeated state. Of course the access to this data structure (e.g. an array) shd be instantaneous, otherwise it also adds to time complexity.

♢ If the number of different possible states is too large, then what we do is to keep a smaller array and use hash functions to index into the array and if there is a clash, expand the array via linked lists.

Non-CS people: Repeated state checking requires a new data structure, up to the size of possible different states, but often much less. This size should also be considered when selecting a search algorithm that requires repeated state checking.

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

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

Iterative deepening search

Can we do away with trying to estimate the limit?

Iterative deepening search

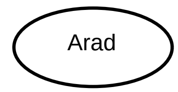
function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution sequence
  inputs: *problem*, a problem

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

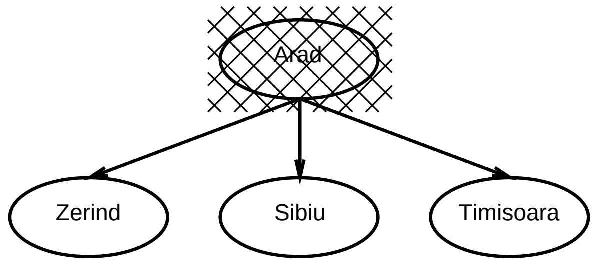
cutoff: no solution within the depth-limit

Failure: no solution at all

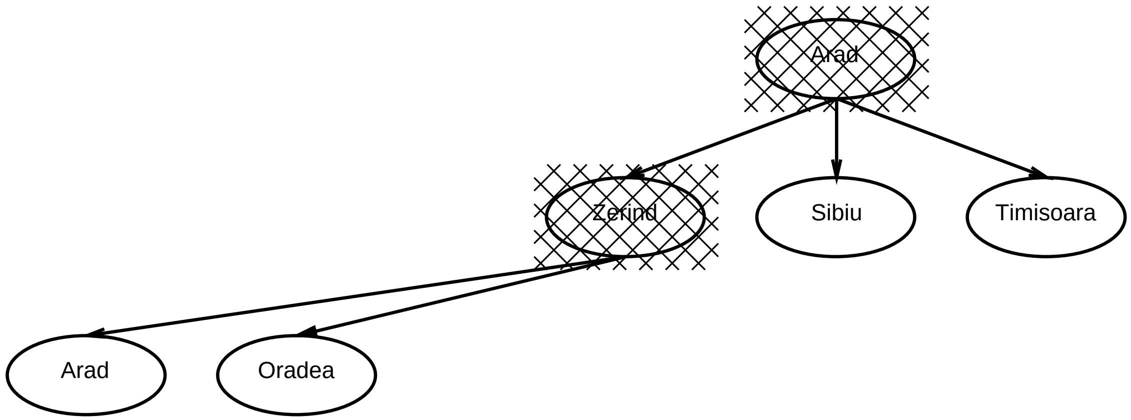
Iterative deepening search $l = 0$



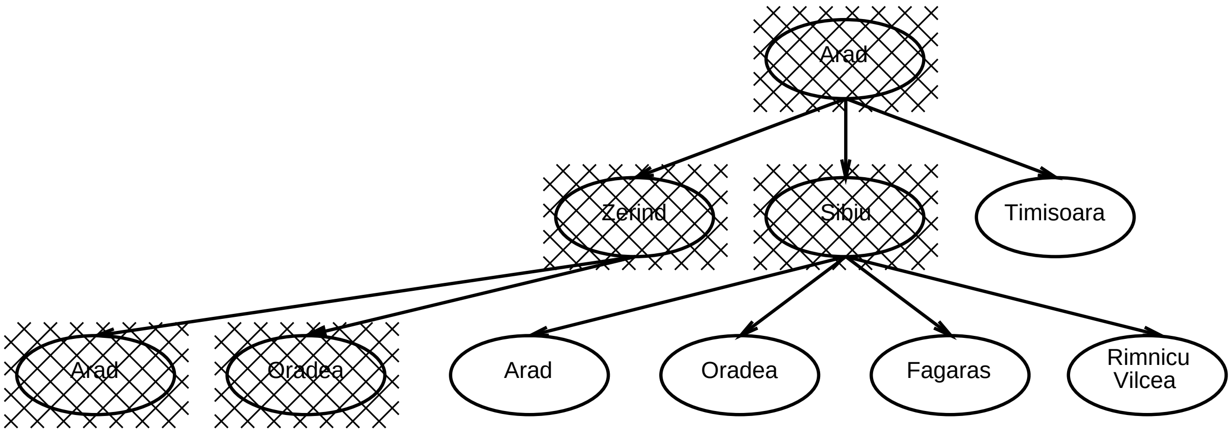
Iterative deepening search $l = 1$



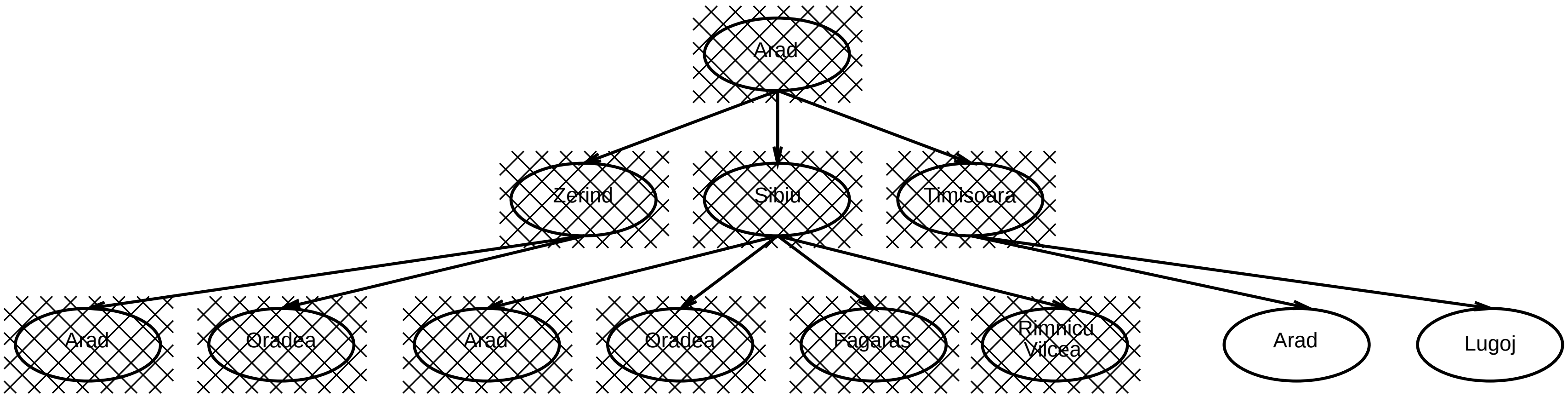
Iterative deepening search $l = 1$



Iterative deepening search $l = 2$



Iterative deepening search $l = 2$



Properties of iterative deepening search

Complete??

Time??

Space??

Optimal??

Properties of iterative deepening search

Complete?? Yes

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??

Optimal??

Note: Number of visited nodes

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??

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 $= 1$, but not in general

Properties of iterative deepening search

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

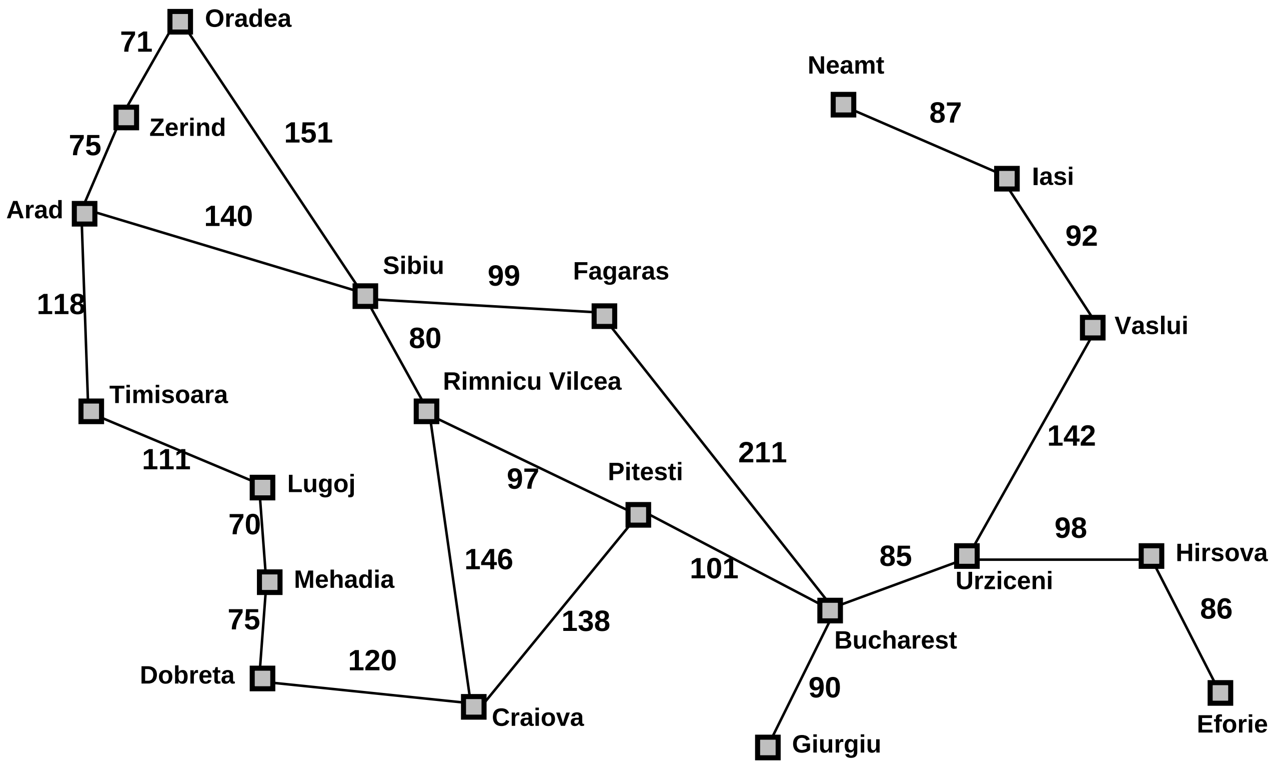
Number of *generated nodes for* $b = 10$ *and* $d = 5$ *:*

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

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

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

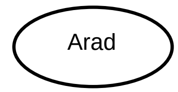
Uniform-cost search

Expand least-cost (path cost) unexpanded node

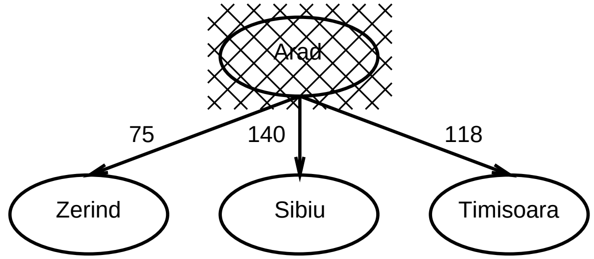
Implementation:

QueueingFn = *insert in order of increasing path cost*

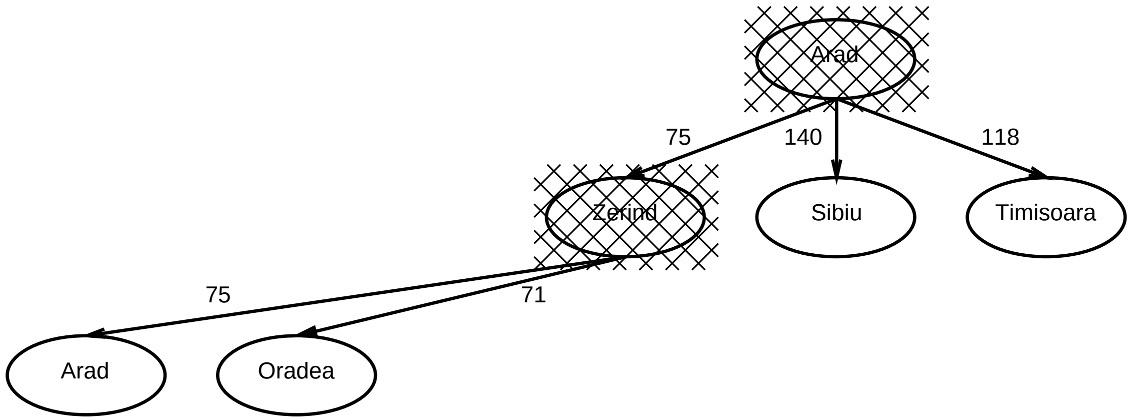
Uniform-cost search



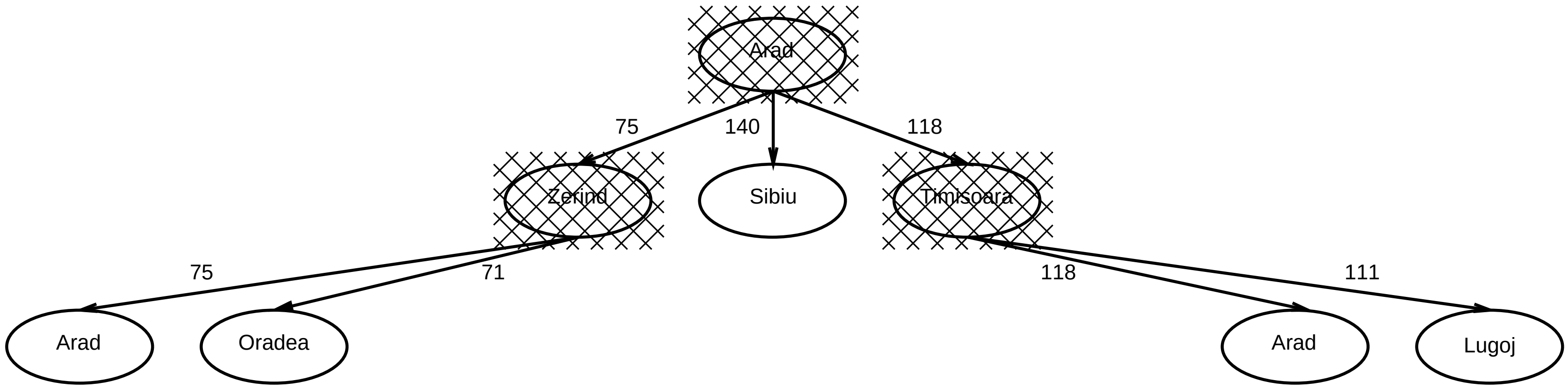
Uniform-cost search



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 $\geq \epsilon$

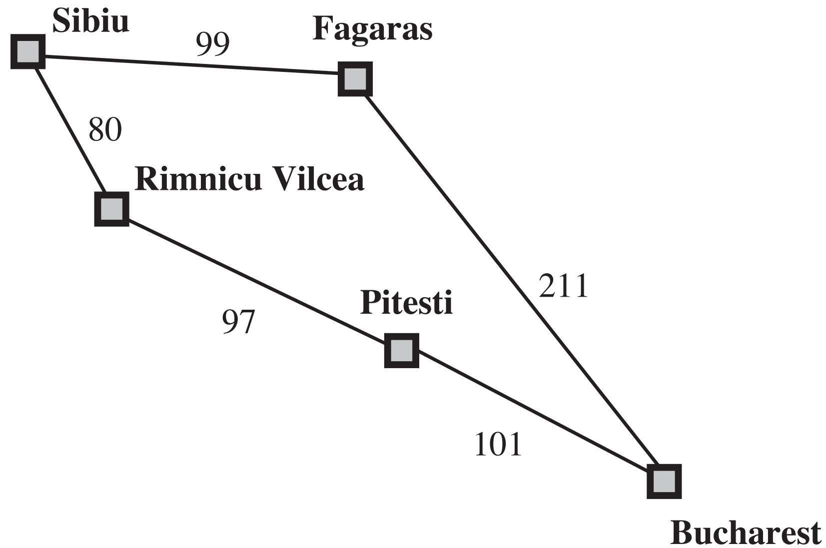
Time?? $\#$ of nodes with $g \leq$ cost of optimal solution

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

Optimal?? Yes with the right implementation (Fig. 3.14) which checks if a shorter path is found to a node in the frontier.

Optimality of uniform-cost search

When a path to Bucharest is found via Fagaras (310km), and later via Pitesti (278km), Bucharest node is replaced to reflect the new found path. So that when it is taken from the fringe and goal-tested, we find the right path.



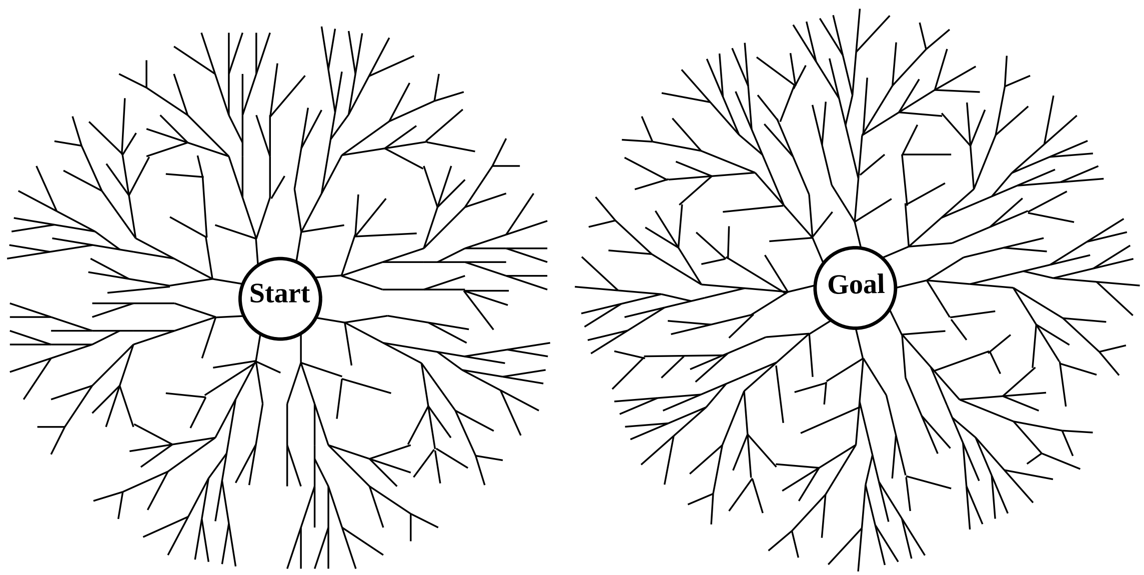
BFS versus uniform-cost search

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.

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 .

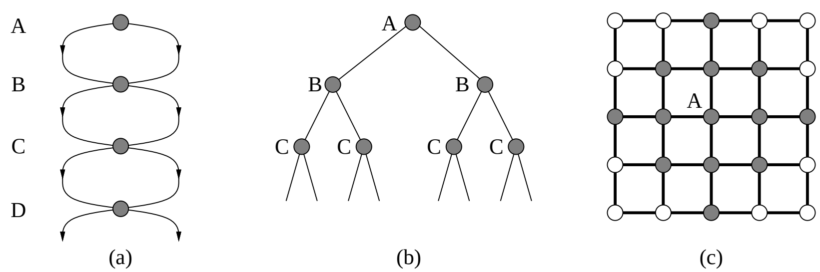
Summary of algorithms

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 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 using a separate ”closed” set.

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
```

```
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
```

Graph search

Side-by-side: These are removed in the new edition, but it may be good to see them side by side. Sometimes we check against just the explored (or closed), sometimes also the fringe.... with small consequences

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.

Graph search

When you should use a graph search: if you cannot avoid cycles (arriving to the same state in more than one path as in shortest travel distance problem).

If not, you can have exponential growth even in linear spaces or you may miss the optimal path.

Graph search

Problems with Graph Search:

◇ *Memory Requirements: increased space requirements for Depth-First search (the state, rather than the node, is checked for repetition!)*

◇ *Optimality: Graph Search deletes the later found path to a repeated state. This could be the path with a shorter cost according to the chosen search strategy (e.g. iterative deepening). however, we can show that with slight changes, Uniform-Cost search remains optimal with a graph-based implementation.*

More Complex Search Problems

Deterministic, fully observable $\Longrightarrow$ single-state problem

Deterministic, partially observable $\Longrightarrow$ multiple-state problem

E.g. The robot cannot tell which room it is in.

Nondeterministic $\Longrightarrow$ contingency problem

E.g. Suck action may not always work.
must use sensors during execution
solution is a tree *or* policy
often interleave *search, execution*

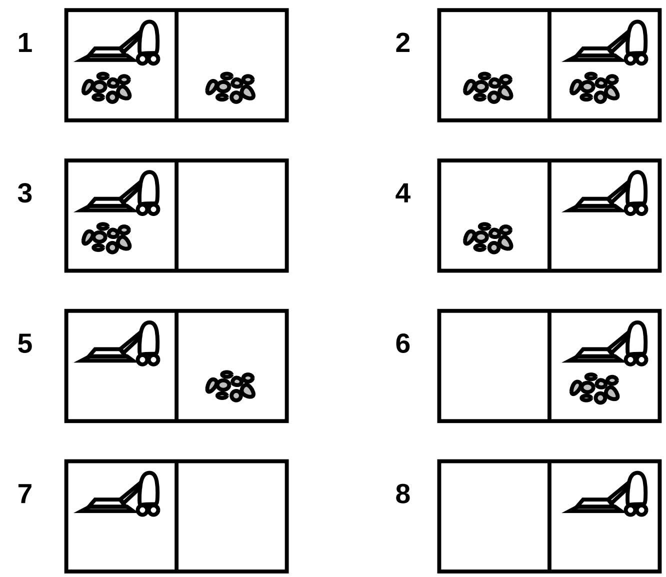
Unknown state space $\Longrightarrow$ exploration problem *("online")*

Example: vacuum world

Single-state problem, start in $\#5$.
Solution: *Right, Suck*

Agent has no sensors: Multiple-state problem,
start in $\{1, 2, 3, 4, 5, 6, 7, 8\}$
e.g., $Right$ *goes to* $\{2, 4, 6, 8\}$.
Solution: *Right, Suck, Left, Suck*

World is stochastic: Contingency problem,
start in $\#5$
Murphy's Law: $Suck$ *can dirty a clean carpet*
Local sensing: dirt, location.
Solution: *Right, if ([B,Dirty], Suck)*



Summary

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

Variety of uninformed search strategies

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