Complexity

- Why worry about complexity of algorithms?

➢ because a problem may be solvable in principle but may take too long to solve in practice

Complexity: Tower of Hanoi

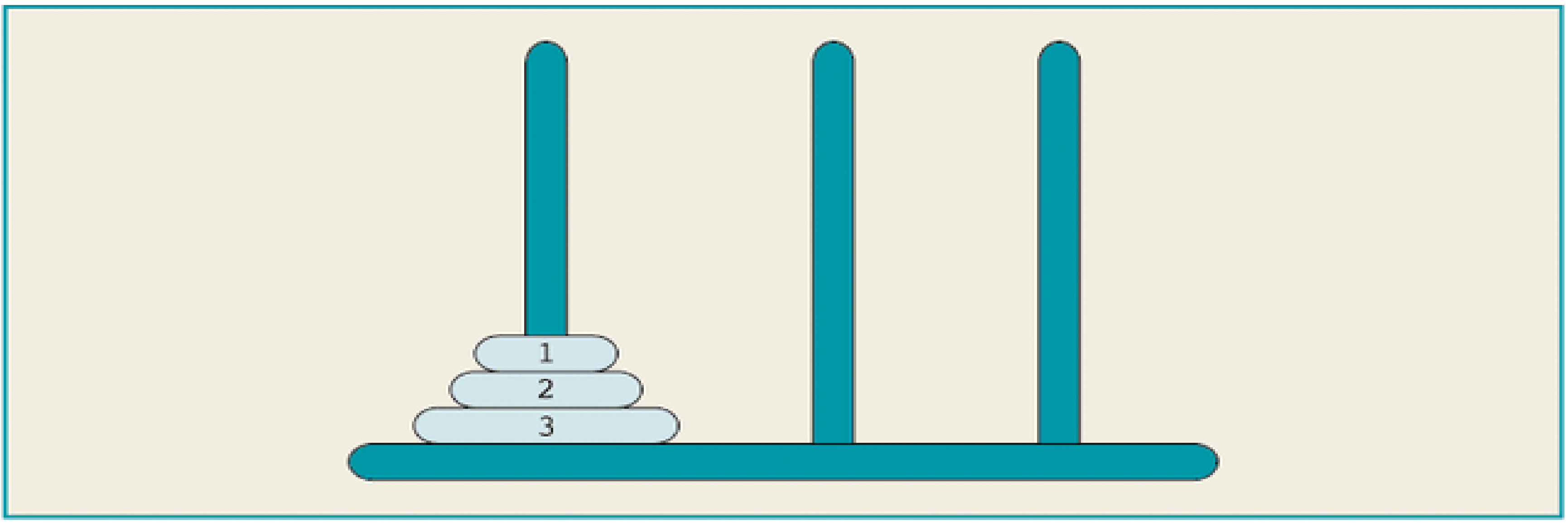


Figure 11-6 Tower of Hanoi problem with three disks

Complexity: Tower of Hanoi

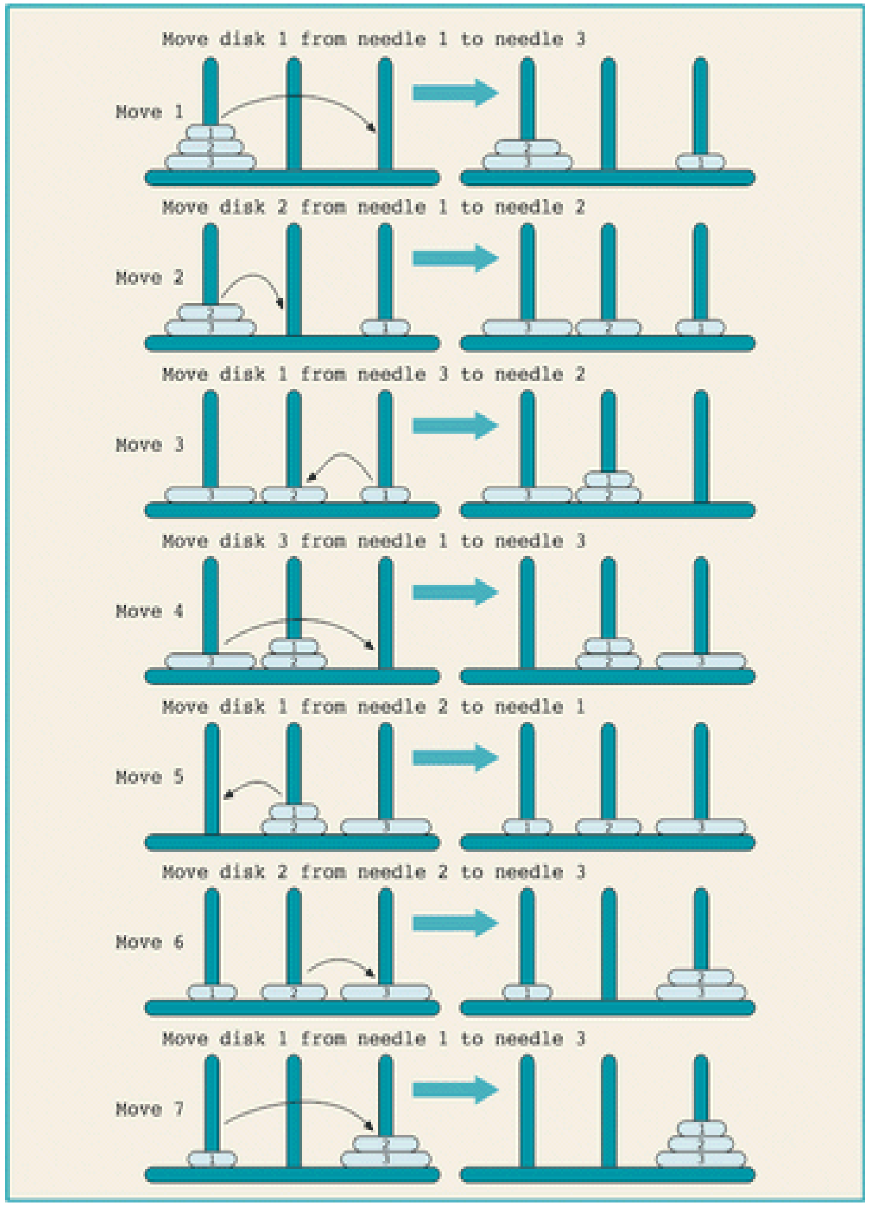


Figure 11-7 Solution of Tower of Hanoi problem with three disks

Complexity: Tower of Hanoi

- 3-disk problem: $2^3 - 1 = 7$ moves
- 64-disk problem: $2^{64} - 1$.
  - $2^{10} = 1024 \approx 1000 = 10^3$,
  - $2^{64} = 2^4 * 2^{60} \approx 2^4 * 10^{18} = 1.6 * 10^{19}$
- One year $\approx 3.2 * 10^7$ seconds

Complexity: Tower of Hanoi

- The wizard's speed = one disk / second

$$1.6 * 10^{19} = 5 * 3.2 * 10^{18} =$$

$$5 * (3.2 * 10^{7}) * 10^{11} =$$

$$(3.2 * 10^{7}) * (5 * 10^{11})$$

Complexity: Tower of Hanoi

- The time required to move all 64 disks from needle 1 to needle 3 is roughly $5 * 10^{11}$ years.

- It is estimated that our universe is about 15 billion $= 1.5 * 10^{10}$ years old.

$$5 * 10^{11} = 50 * 10^{10} \approx 33 * (1.5 * 10^{10}).$$

Complexity: Tower of Hanoi

- Assume: a computer with 1 billion = 10^9 moves/second.
  - Moves/year=$(3.2 * 10^7) * 10^9 = 3.2 * 10^{16}$

- To solve the problem for 64 disks:
  - $2^{64} \approx 1.6 * 10^{19} = 1.6 * 10^{16} * 10^3 =$

    $(3.2 * 10^{16}) * 500$

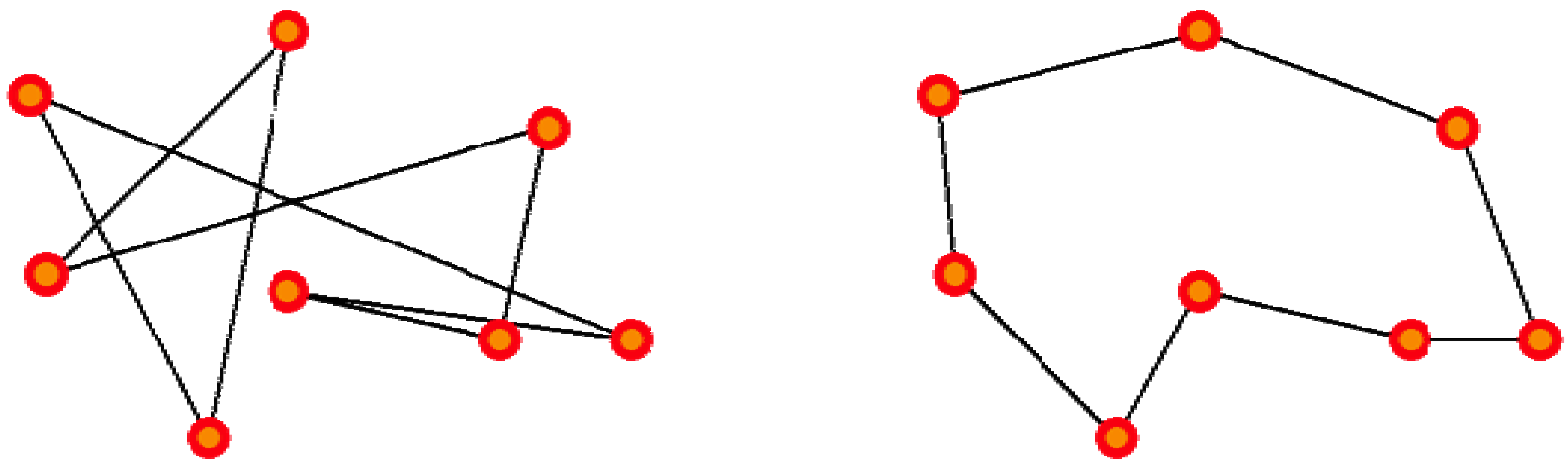
  - 500 years for the computer to generate 2^{64} moves at the rate of 1 billion moves per second.

Complexity

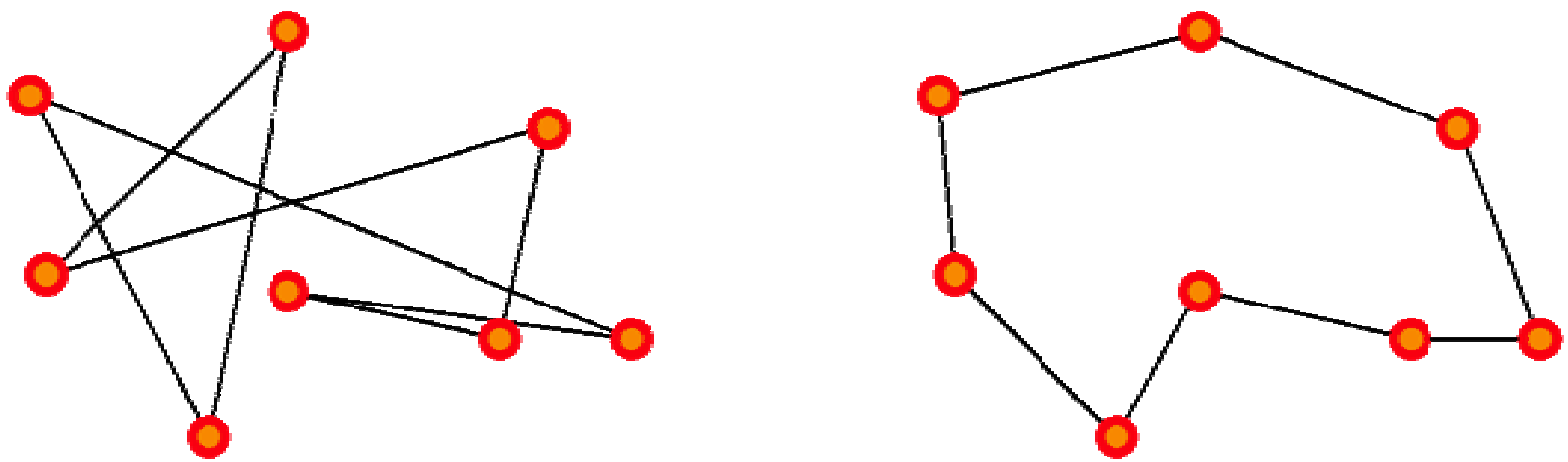
- Why worry about complexity of algorithms?
  - because a problem may be solvable in principle but may take too long to solve in practice
- How can we evaluate the complexity of algorithms?
  - through asymptotic analysis, i.e., estimate time (or number of operations) necessary to solve an instance of size *n* of a problem when *n* tends towards infinity
  - See AIMA, Appendix A.

Complexity example: Traveling Salesman Problem

- There are n cities, with a road of length L_{ij} joining city i to city j.
- The salesman wishes to find a way to visit all cities that is optimal in two ways:
  each city is visited only once, and
  the total route is as short as possible.



Complexity example: Traveling Salesman Problem



This is a *hard* problem: the only known algorithms (so far) to solve it have exponential complexity, that is, the number of operations required to solve it grows as *exp(n)* for *n* cities.

Why is exponential complexity "hard"?

It means that the number of operations necessary to compute the exact solution of the problem grows exponentially with the size of the problem (here, the number of cities).

- exp(1) = 2.72
- exp(10) = $2.20\ 10^{4}$ (daily salesman trip)
- exp(100) = $2.69\ 10^{43}$ (monthly salesman planning)
- exp(500) = $1.40\ 10^{217}$ (music band worldwide tour)
- exp(250,000) = $10^{108,573}$ (fedex, postal services)
- Fastest computer = 10^{12} operations/second

So…

In general, exponential-complexity problems *cannot be solved for any but the smallest instances!*

Complexity

- Polynomial-time (P) problems: we can find algorithms that will solve them in a time (=number of operations) that grows polynomially with the size of the input.

➢ for example: sort n numbers into increasing order: poor algorithms have *n^2* complexity, better ones have *n log(n)* complexity.

Complexity

- Since we did not state what the order of the polynomial is, it could be very large! Are there algorithms that require more than polynomial time?

- Yes (until proof of the contrary); for some algorithms, we do not know of any polynomial-time algorithm to solve them. These belong to the class of nondeterministic-polynomial-time (NP) algorithms (which includes P problems as well as harder ones).

➢ for example: traveling salesman problem.

- In particular, exponential-time algorithms are believed to be NP.

Note on NP-hard problems

- The formal definition of NP problems is:

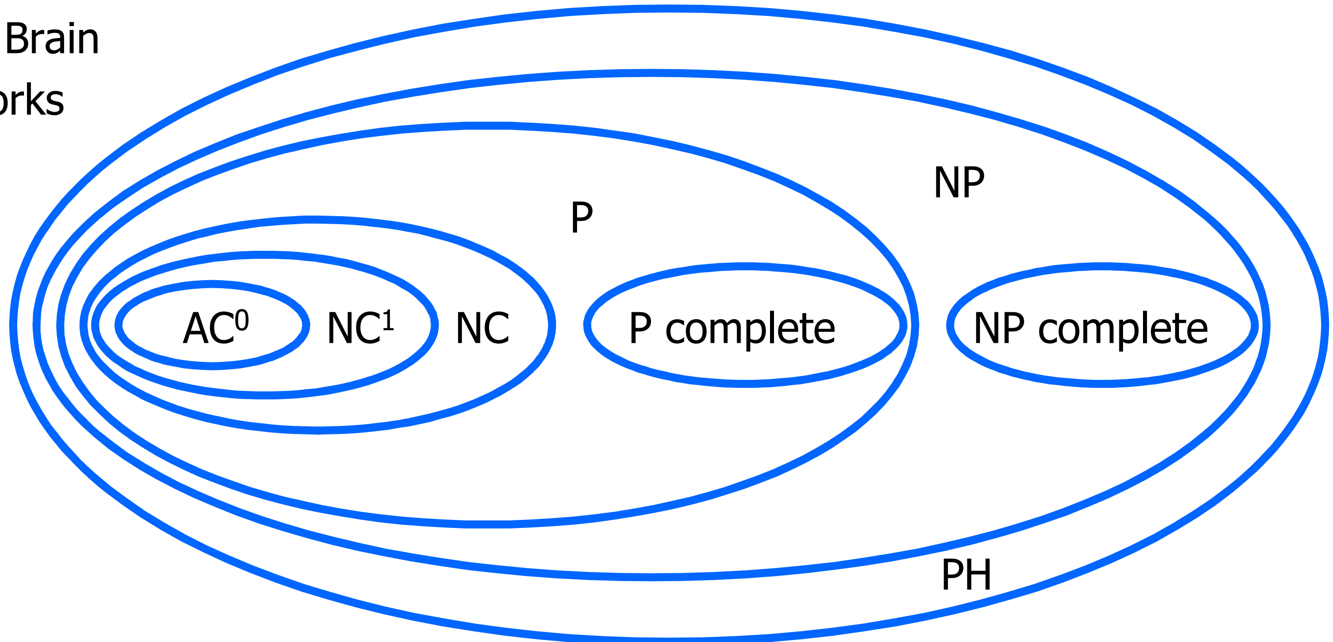
A problem is nondeterministic polynomial if there exists some algorithm that can guess a solution and then verify whether or not the guess is correct in polynomial time.

(one can also state this as these problems being solvable in polynomial time on a nondeterministic Turing machine.)

In practice, until proof of the contrary, this means that known algorithms that run on known computer architectures will take more than polynomial time to solve NP problems.

Polynomial-time hierarchy

- From Handbook of Brain Theory & Neural Networks (Arbib, ed.; MIT Press 1995).



AC^0: can be solved using gates of constant depth
NC^1: can be solved in logarithmic depth using 2-input gates
NC: can be solved by small, fast parallel computer
P: can be solved in polynomial time
P-complete: hardest problems in P; if one of them can be proven to be NC, then P = NC
NP: nondeterministic-polynomial algorithms
NP-complete: hardest NP problems; if one of them can be proven to be P, then NP = P
PH: polynomial-time hierarchy

Remember: 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
```

Queuing-Fn(*queue, elements*) is a queuing function that inserts a set of elements into the queue and determines the order of node expansion. Varieties of the queuing function produce varieties of the search algorithm.

Evaluation of search strategies

- A search strategy is defined by picking the order of node expansion.

- Search algorithms are commonly evaluated according to the following four criteria:
  - **Completeness:** does it always find a solution if one exists?
  - **Time complexity:** how long does it take as function of num. of nodes?
  - **Space complexity:** how much memory does it require?
  - **Optimality:** does it guarantee the least-cost solution?

- Time and space complexity are measured in terms of:
  - b – max branching factor of the search tree
  - d – depth of the least-cost solution
  - m – max depth of the search tree (may be infinity)

Note: Approximations

- In our complexity analysis, we do not take the built-in loop-detection into account.

- The results only 'formally' apply to the variants of our algorithms WITHOUT loop-checks.
- Studying the effect of the loop-checking on the complexity is hard:
  - overhead of the checking MAY or MAY NOT be compensated by the reduction of the size of the tree.

- Also: our analysis DOES NOT take the length (space) of representing paths into account !!

http://www.cs.kuleuven.ac.be/~dannyd/FAI/

Uninformed search strategies

Use only information available in the problem formulation

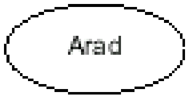
- Breadth-first
- Uniform-cost
- Depth-first
- Depth-limited
- Iterative deepening

Breadth-first search

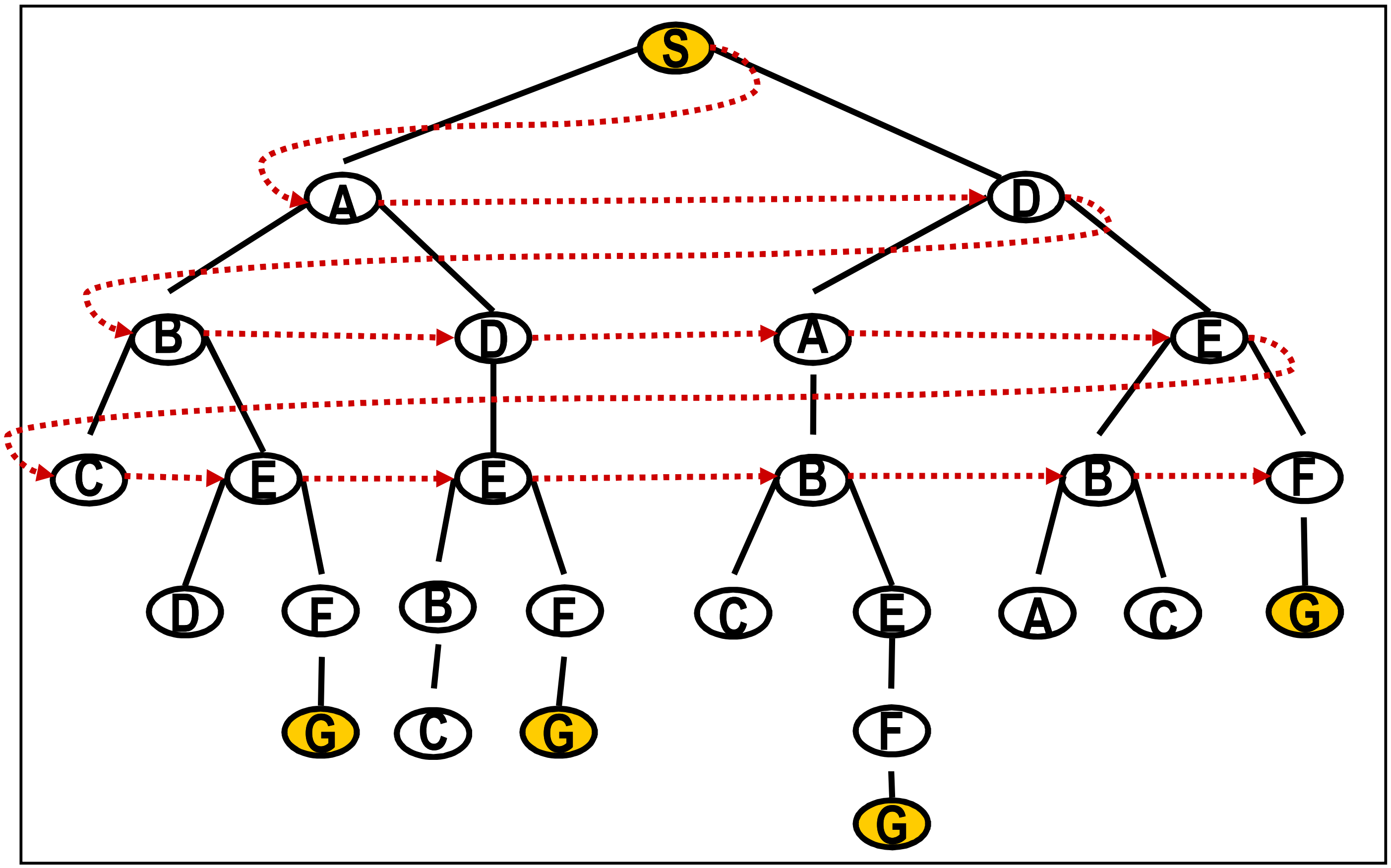
Expand shallowest unexpanded node

Implementation:

$\textsc{QueueingFn}$ = put successors at end of queue

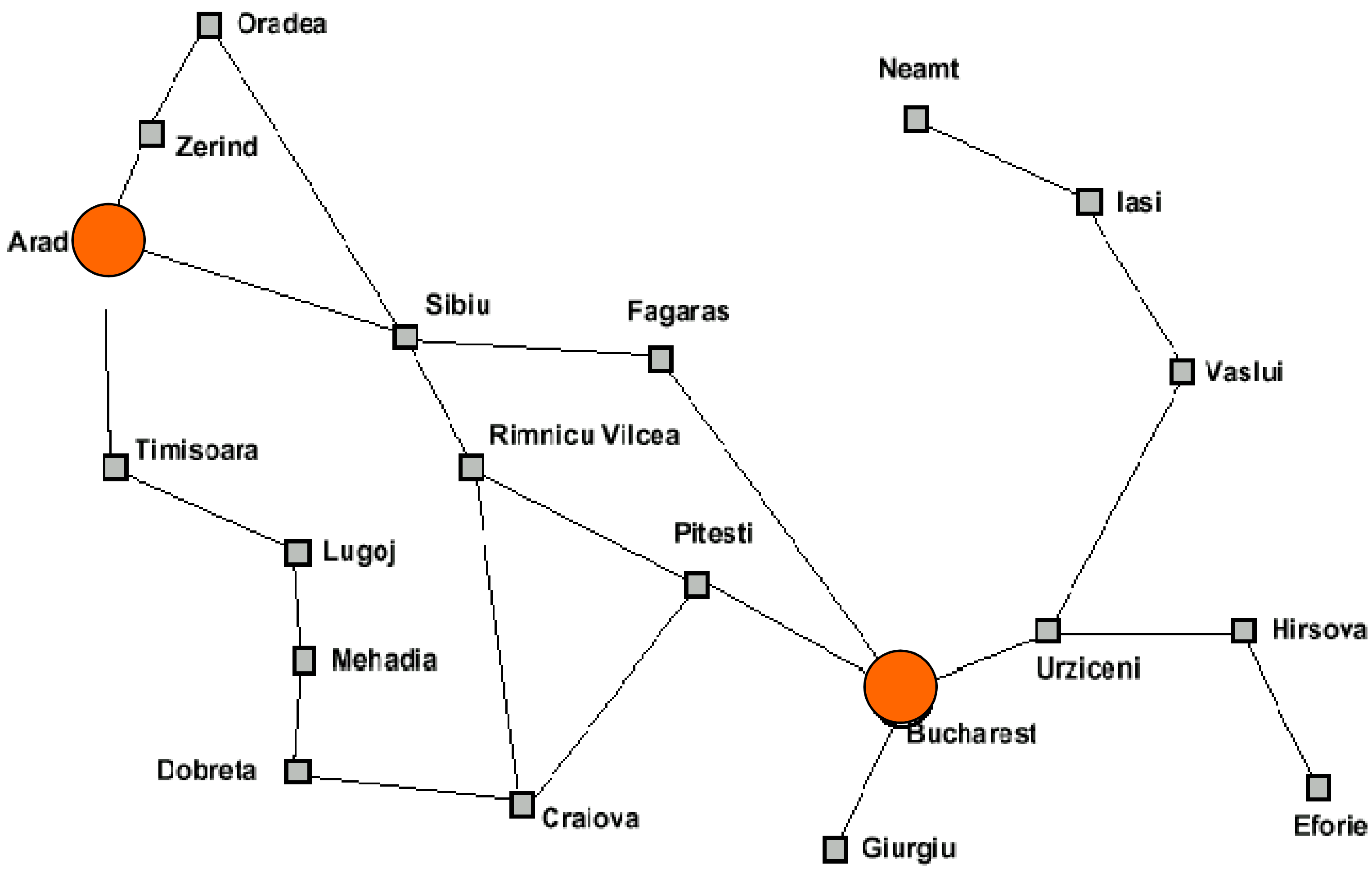


Breadth-first search

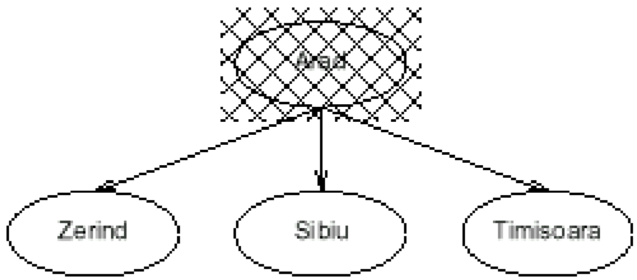


Move downwards, level by level, until goal is reached.

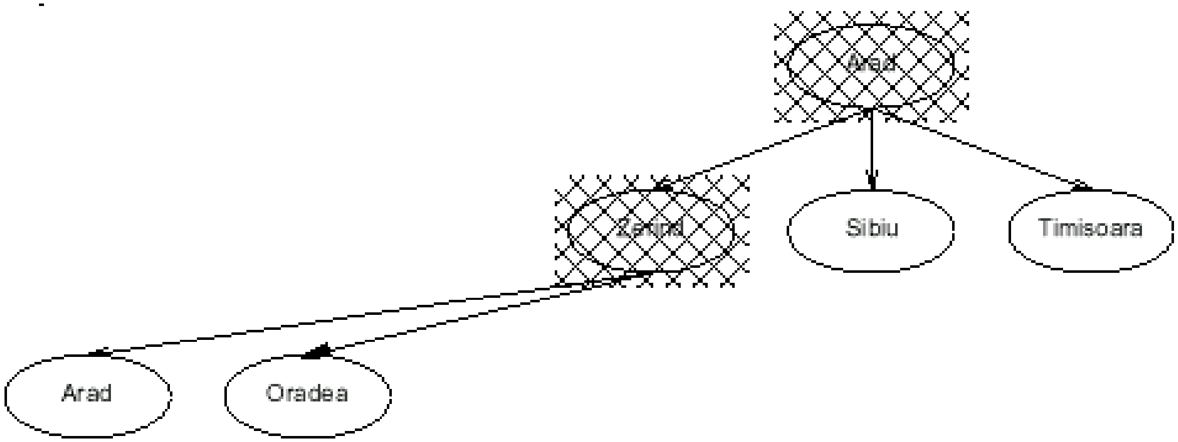
Example: Traveling from Arad To Bucharest



Breadth-first search

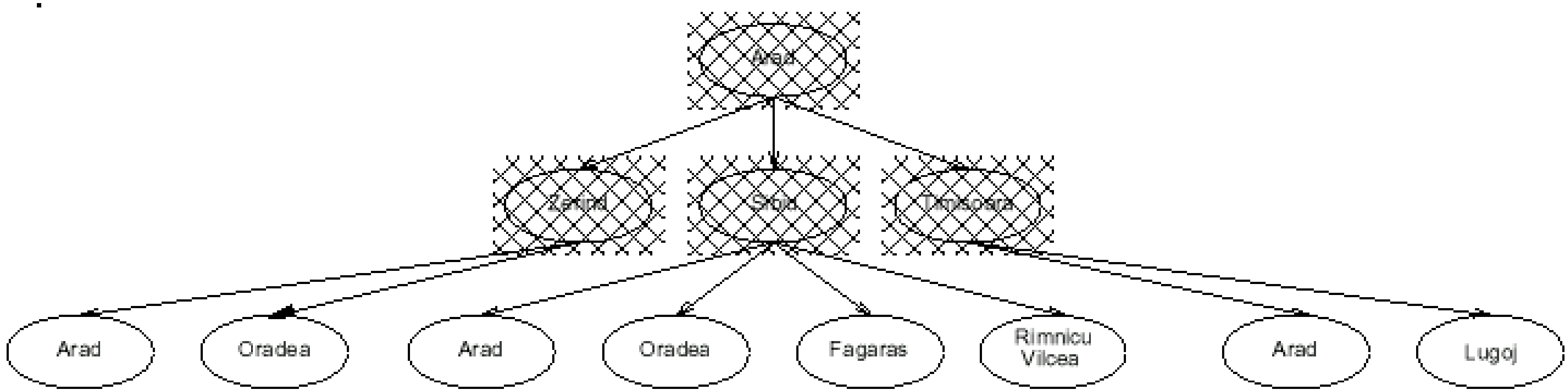


Breadth-first search



Breadth-first search

.



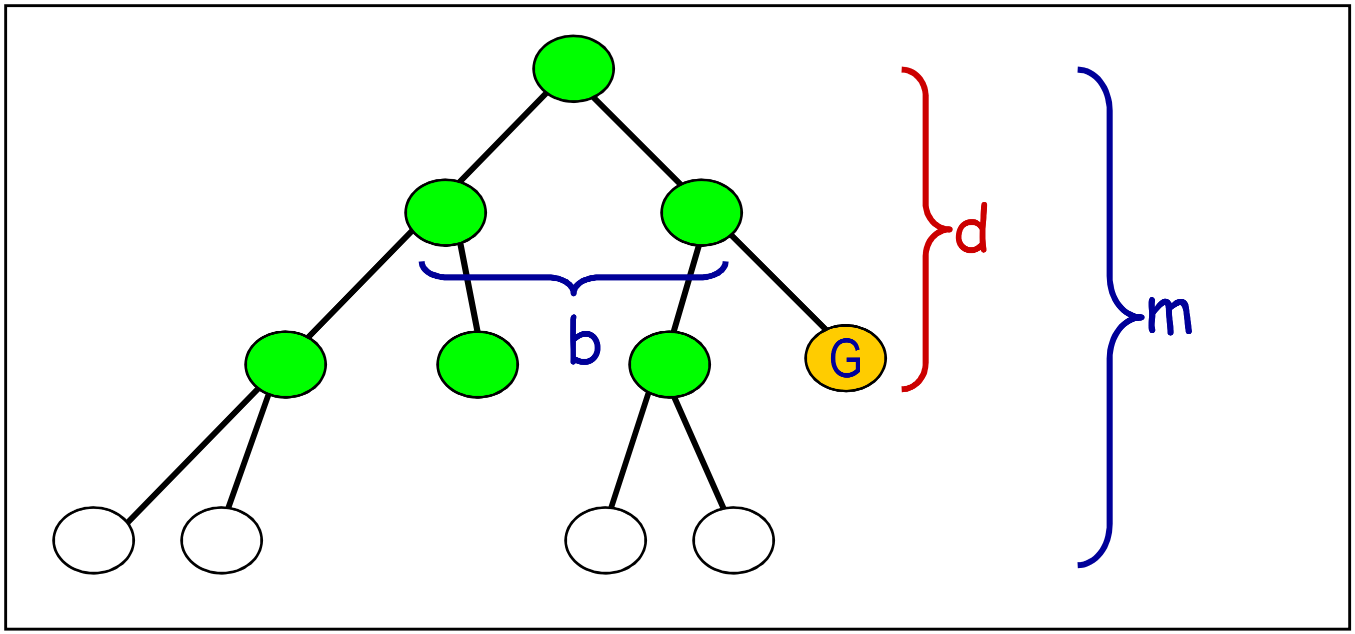
Properties of breadth-first search

- Completeness: Yes, if b is finite
- Time complexity: $1+b+b^2+\ldots+b^d = O(b^d)$, i.e., exponential in d
- Space complexity: $O(b^d)$ (see following slides)
- Optimality: Yes, **assuming cost = 1 per step**

- Search algorithms are commonly evaluated according to the following four criteria:
  - **Completeness:** does it always find a solution if one exists?
  - **Time complexity:** how long does it take as function of num. of nodes?
  - **Space complexity:** how much memory does it require?
  - **Optimality:** does it guarantee the least-cost solution?
- Time and space complexity are measured in terms of:
  - b – max branching factor of the search tree
  - d – depth of the least-cost solution
  - m – max depth of the search tree (may be infinity)

Time complexity of breadth-first search

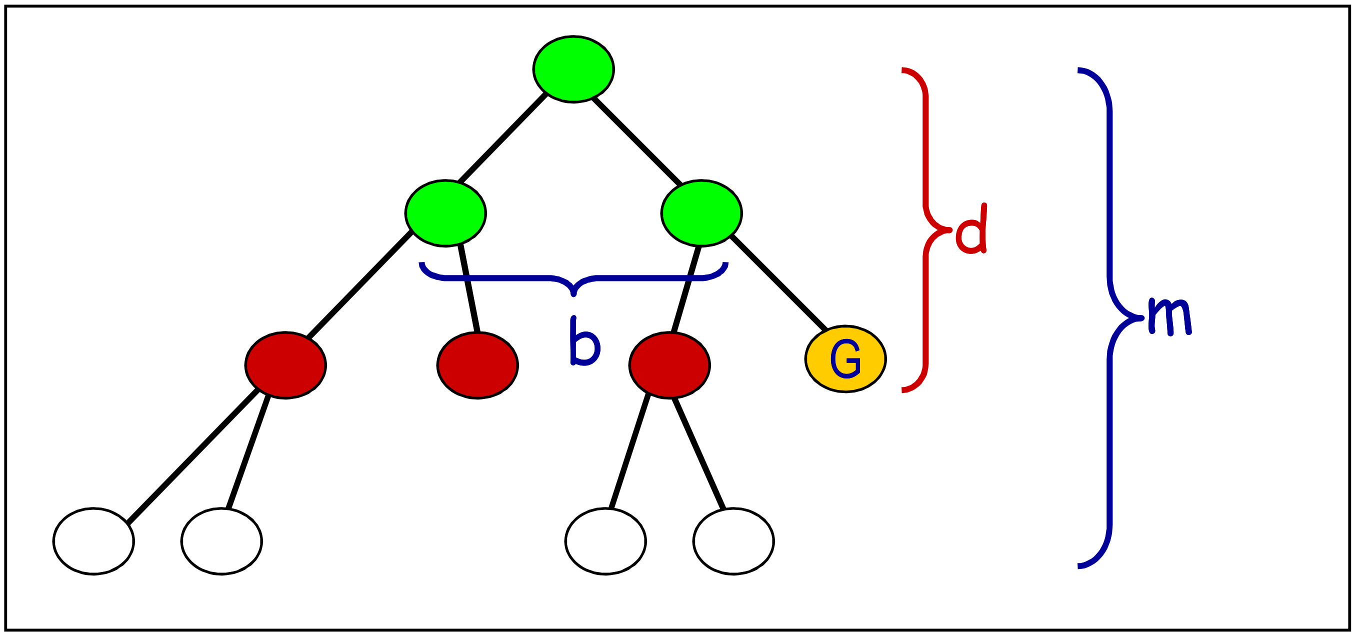
- If a goal node is found on depth d of the tree, all nodes up till that depth are created.



- Thus: $O(b^d)$

Space complexity of breadth-first

- Largest number of nodes in QUEUE is reached on the level d of the goal node.



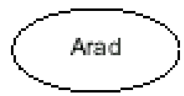
- QUEUE contains all and G nodes. (Thus: 4) .
- In General: b^{d}

Uniform-cost search

Expand least-cost unexpanded node

Implementation:

QueueingFn = insert in order of increasing path cost

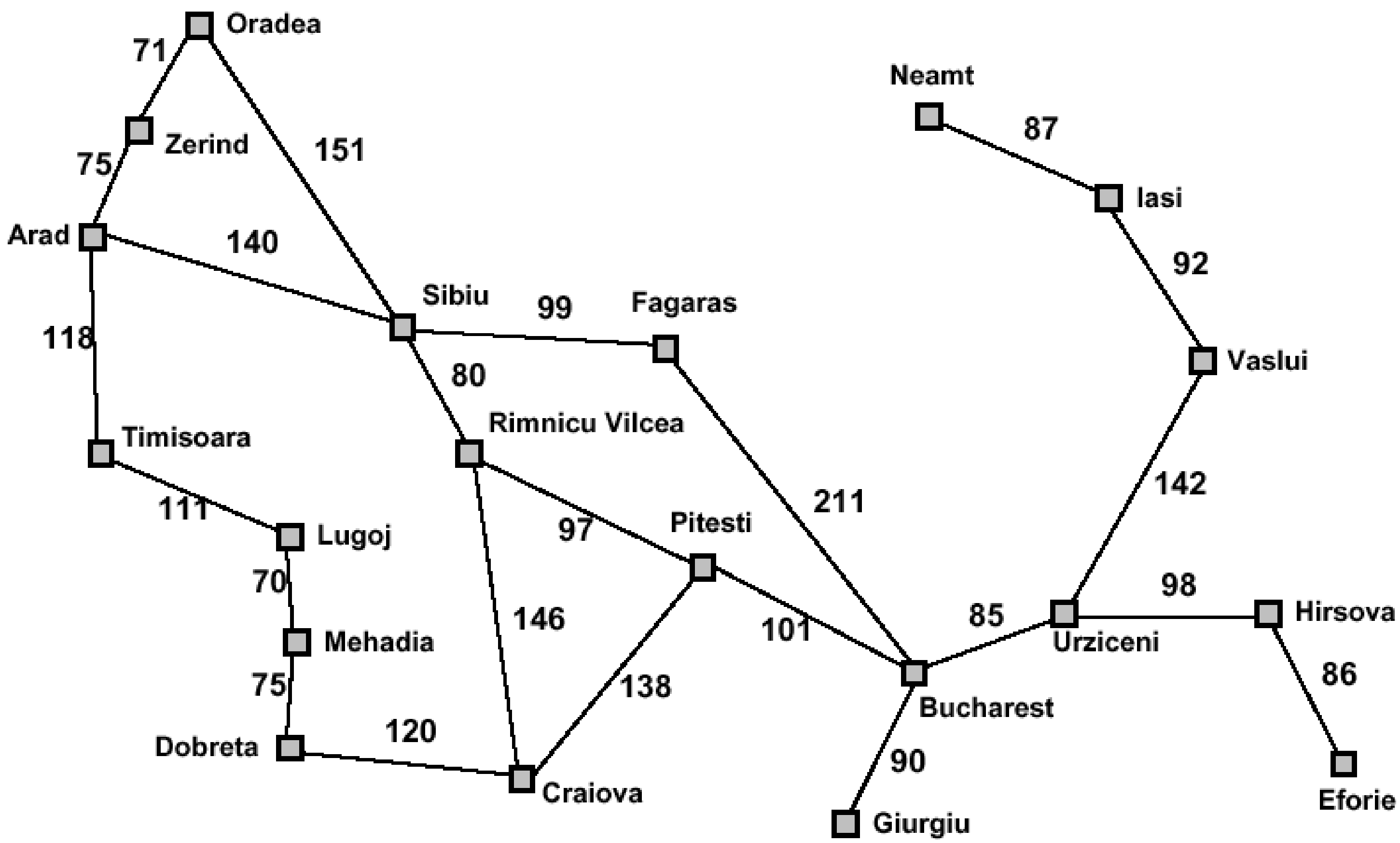


So, the queueing function keeps the node list sorted by increasing path cost, and we expand the first unexpanded node (hence with smallest path cost)

A refinement of the breadth-first strategy:

Breadth-first = uniform-cost with path cost = node depth

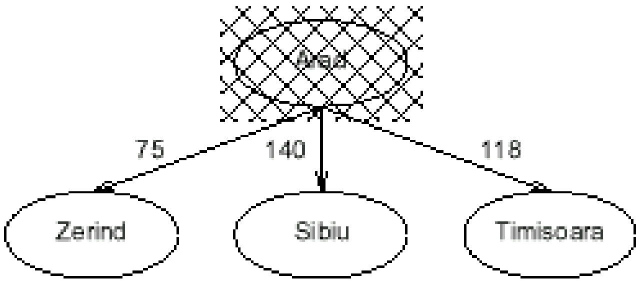
Romania with step costs in km



Straight–line distance to Bucharest

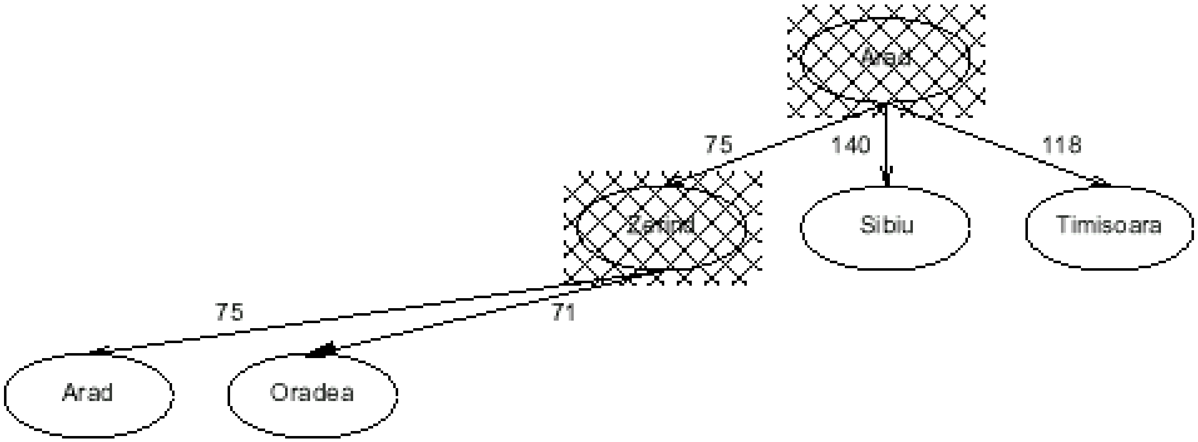
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Uniform-cost search



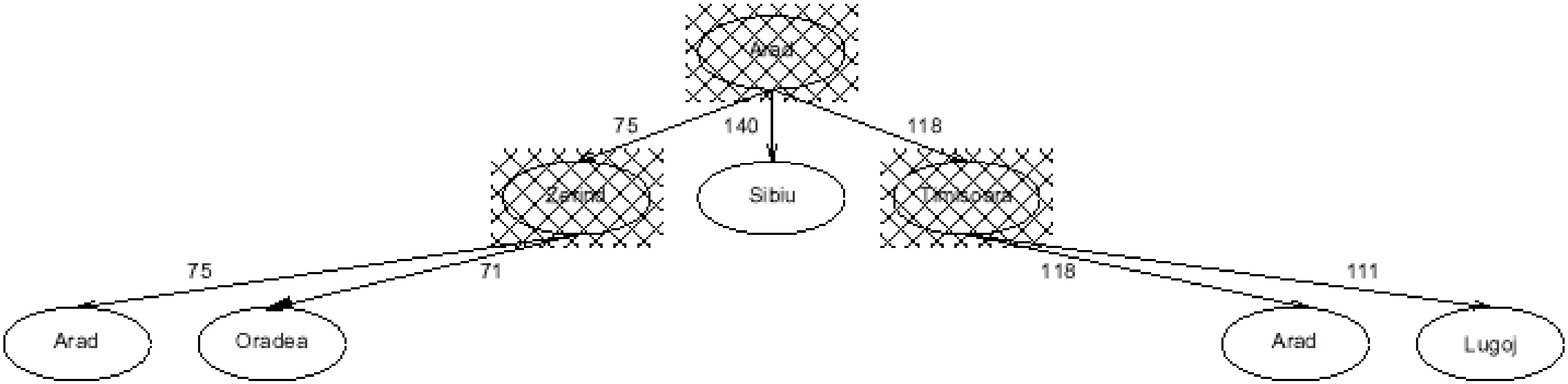
Uniform-cost search

.

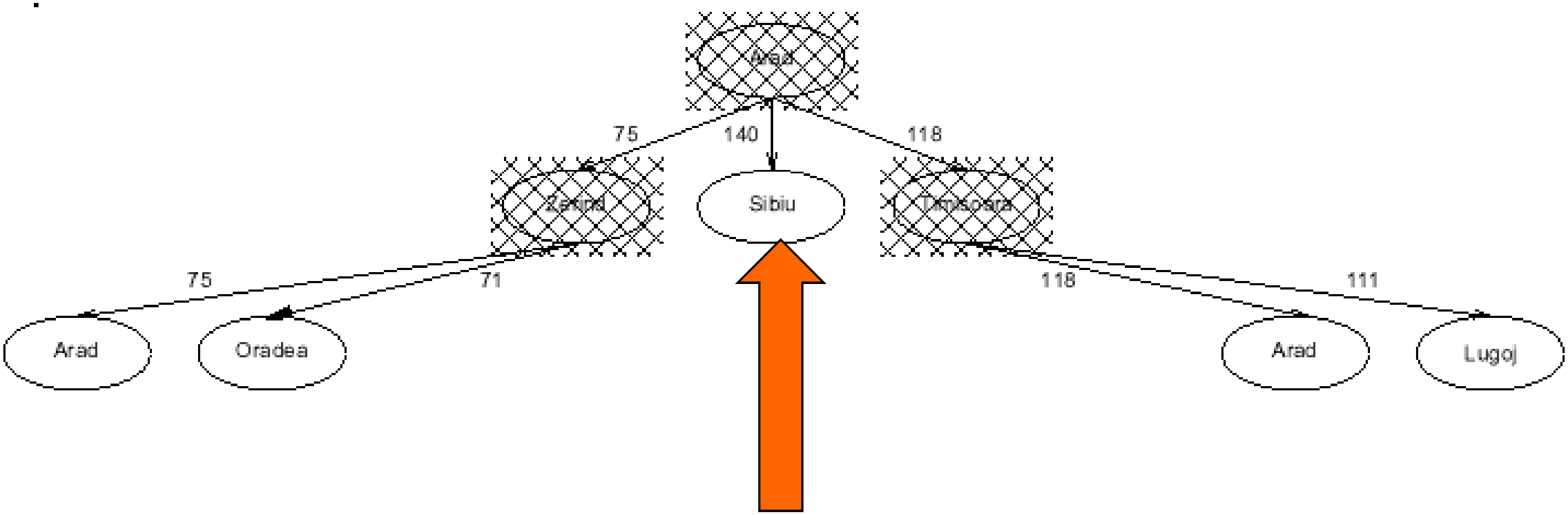


Uniform-cost search

.



Uniform-cost search



Next node to be expanded because of least path cost

Properties of uniform-cost search

- Completeness: Yes, if step cost $\geq \varepsilon > 0$
- Time complexity: # nodes with g $\leq$ cost of optimal solution, $\leq O(b^d)$
- Space complexity: # nodes with g $\leq$ cost of optimal solution, $\leq O(b^d)$
- Optimality: **Can be optimal if you do not terminate the search after reaching a goal state AND if step cost $\geq \varepsilon > 0$**

g(n) is the path cost to node n

Remember:

b = branching factor

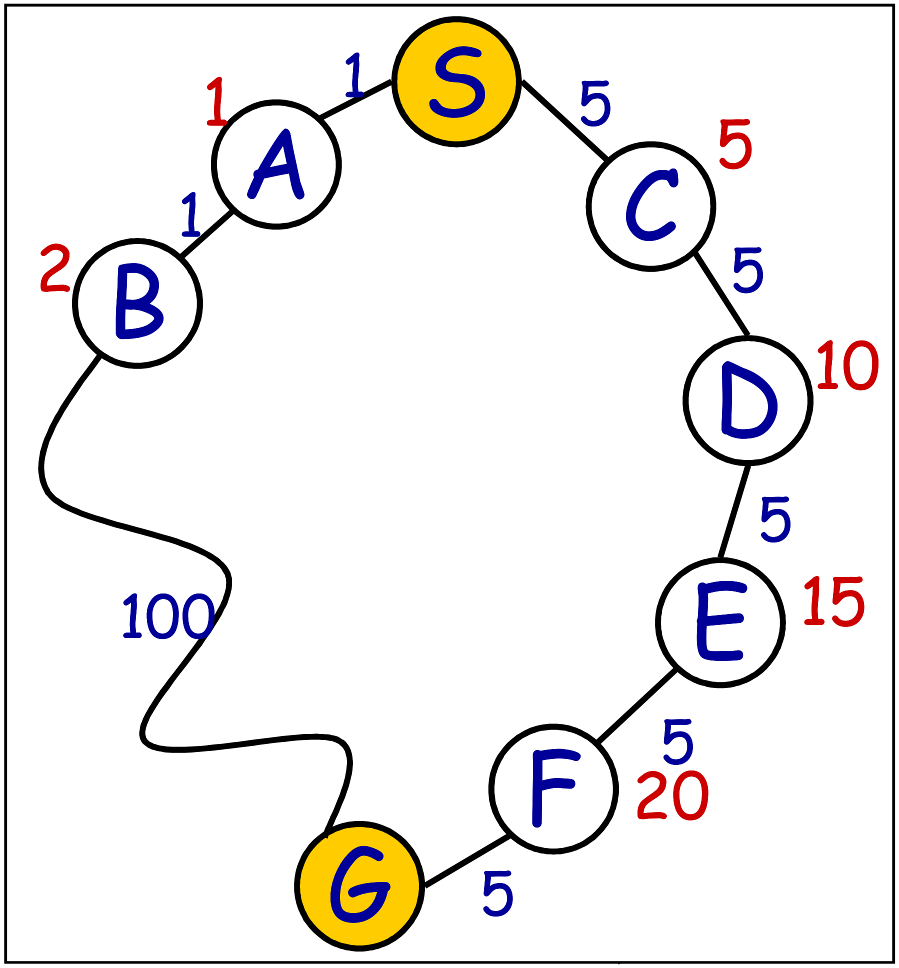
d = depth of least-cost solution

Implementation of uniform-cost search

- Initialize Queue with root node (built from start state)
- Repeat until (Queue empty) or (first node has Goal state):
  - Remove first node from front of Queue
  - Expand node (find its children)
  - Reject those children that have already been considered, to avoid loops
  - Add remaining children to Queue, ***in a way that keeps entire queue sorted by increasing path cost***
- If Goal was reached, return success, otherwise failure

Caution!

- Uniform-cost search **not** optimal if it is terminated when ***any*** node in the queue has goal state.



- Uniform cost returns the path with cost 102 (if first encounter of any goal node is considered a solution), while there is a path with cost 25! So, do not terminate till G itself needs to be expanded

Note: Loop Detection

- In class, we saw that the search may fail or be sub-optimal if:

  - no loop detection: then algorithm runs into infinite cycles
    (A -> B -> A -> B -> …)

  - not queuing-up a node that has a state which we have
    already visited: may yield suboptimal solution

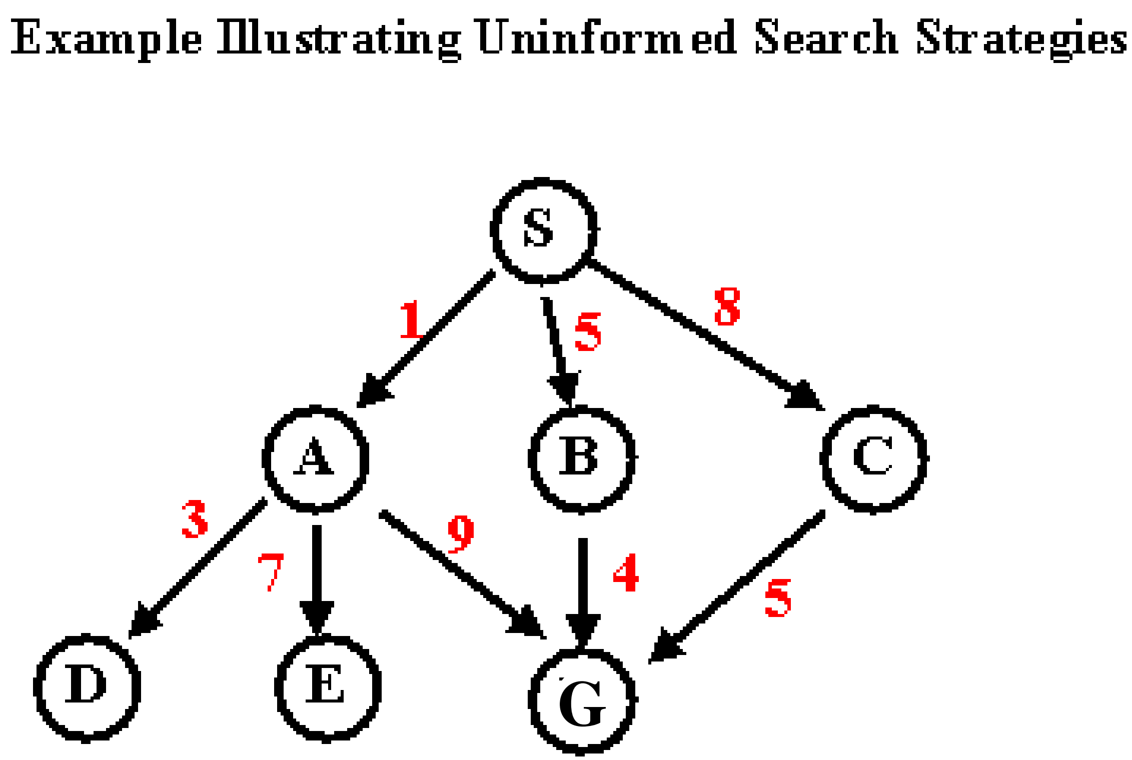
  - simply avoiding to go back to our parent: looks promising,
    but we have not proven that it works

Solution? do not enqueue a node if its state matches the state of any of its parents (assuming path costs>0).

Indeed, if path costs > 0, it will always cost us more to consider a node with that state again than it had already cost us the first time.

Example

From: http://www.csee.umbc.edu/471/current/notes/uninformed-search/



Breadth-First Search Solution

From: http://www.csee.umbc.edu/471/current/notes/uninformed-search/

Breadth-First Search

return GENERAL-SEARCH(problem, ENQUEUE-AT-END)

exp. node	**nodes list**
	{ S }
S	{ A B C }
A	{ B C D E G }
B	{ C D E G G' }
C	{ D E G G' G" }
D	{ E G G' G" }
E	{ G G' G" }
G	{ G' G" }

Solution path found is S A G <-- this G also has cost 10

Number of nodes expanded (including goal node) = 7

Uniform-Cost Search Solution

From: http://www.csee.umbc.edu/471/current/notes/uninformed-search/

Uniform-Cost Search

GENERAL-SEARCH(problem, ENQUEUE-BY-PATH-COST)

exp. node	**nodes list**
	{ S }
S	{ A(1) B(5) C(8) }
A	{ D(4) B(5) C(8) E(8) G(10) } (NB, we don't return G)
D	{ B(5) C(8) E(8) G(10) }
B	{ C(8) E(8) G(9) G(10) }
C	{ E(8) G(9) G(10) G(13) }
E	{ G(9) G(10) G(13) }
G	{ }

Solution path found is S B G <-- this G has cost 9, not 10

Number of nodes expanded (including goal node) = 7

Note: Queuing in Uniform-Cost Search

In the previous example, it is wasteful (but not incorrect) to queue-up three nodes with G state, if our goal if to find the least-cost solution:

Although they represent different paths, we know for sure that the one with smallest path cost (9 in the example) will yield a solution with smaller total path cost than the others.

So we can refine the queueing function by:

- queue-up node if

    1) its state does not match the state of any parent

  and 2) path cost smaller than path cost of any unexpanded node with same state in the queue (and in this case, replace old node with same state by our new node)