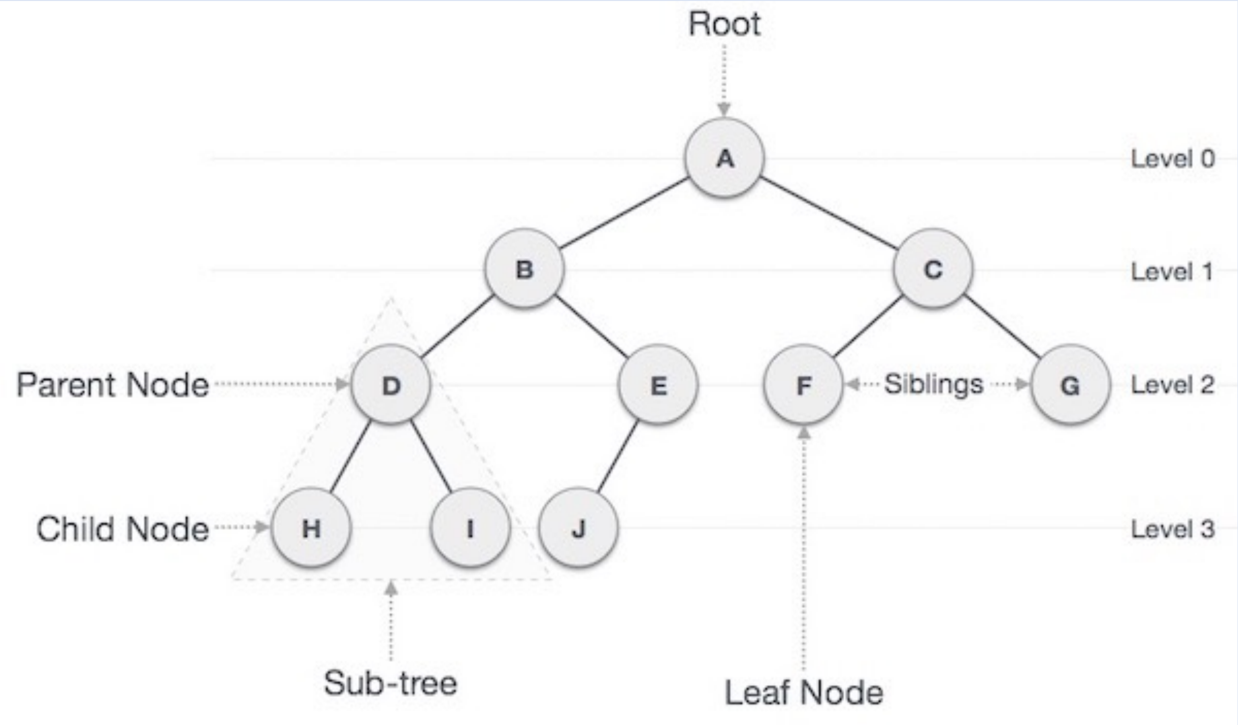
Artificial Intelligence

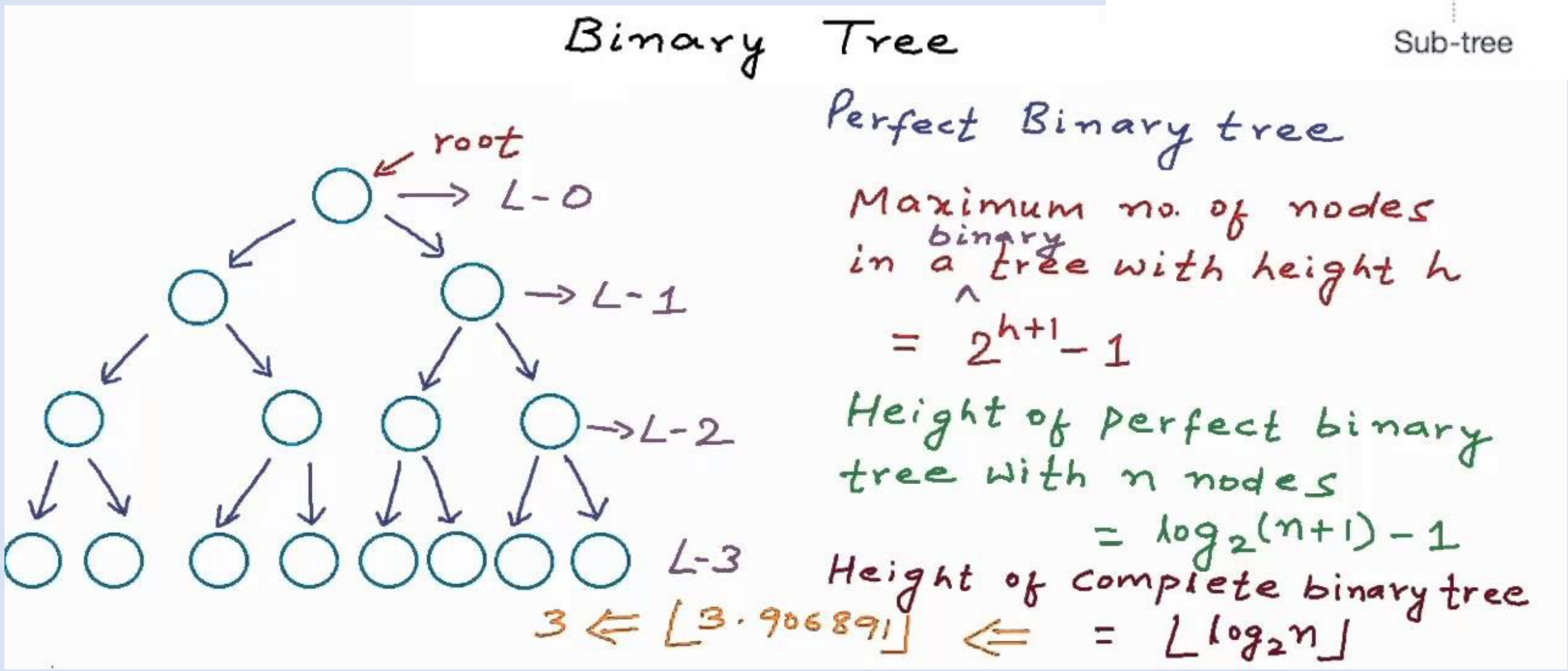
Solving Problems By Searching

Dr. Bilgin Avenoğlu

Search Tree



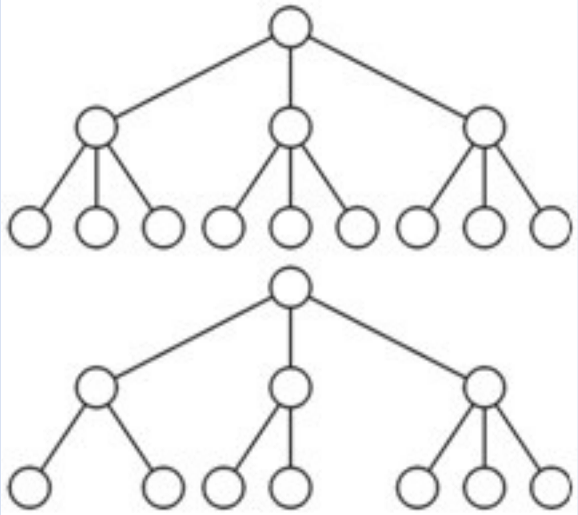
Root
A
B
C
D
E
F
G
H
I
J
Level 0
Level 1
Level 2
Level 3
Parent Node
Child Node
Siblings
Sub-tree
Leaf Node



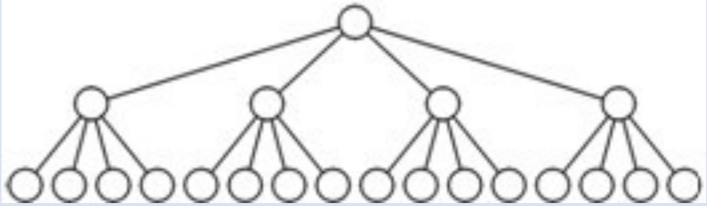
Binary Tree
root
L-0
L-1
L-2
L-3
Perfect Binary tree
Maximum no. of nodes in a binary tree with height h
$= 2^{h+1} - 1$
Height of perfect binary tree with n nodes
$= \log_2(n+1) - 1$
Height of complete binary tree
$3 \Leftarrow \lfloor 3.906891 \rfloor \Leftarrow = \lfloor \log_2 n \rfloor$

N-ary Trees

Ternary (3-ary) Tree



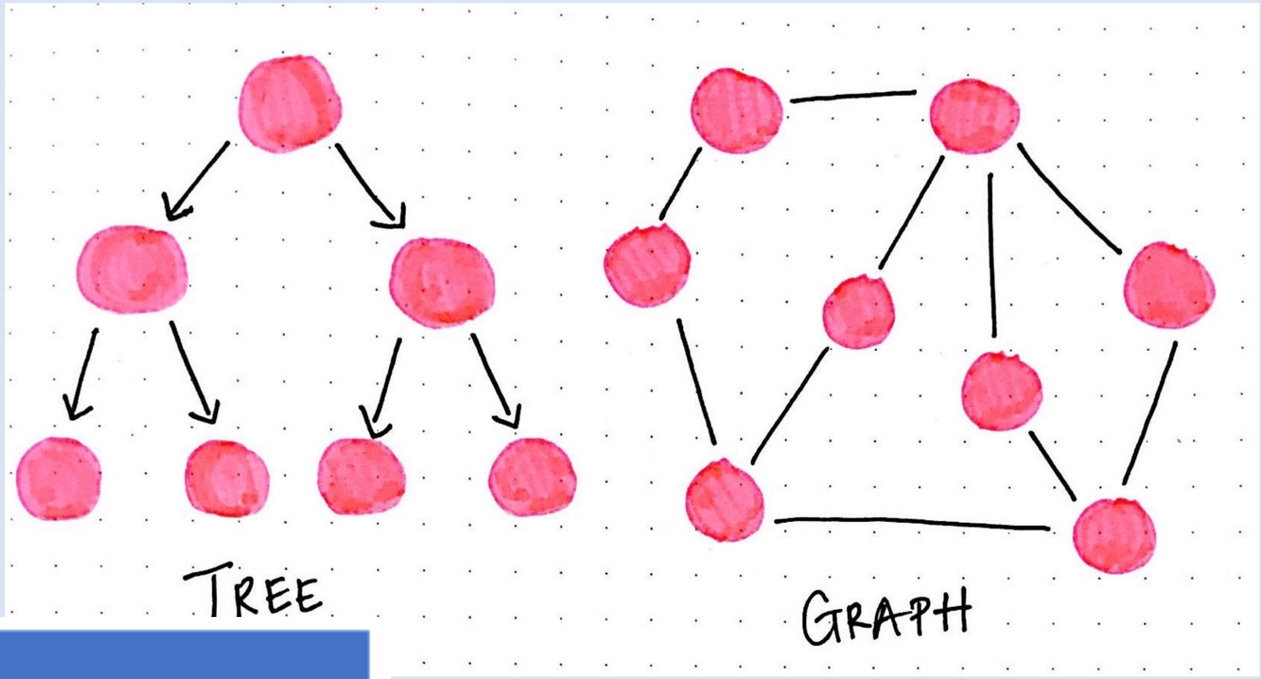
Perfect Quaternary Tree



Number of Nodes in a tree : $1 + b^1 + b^2 + \ldots.. + b^d$

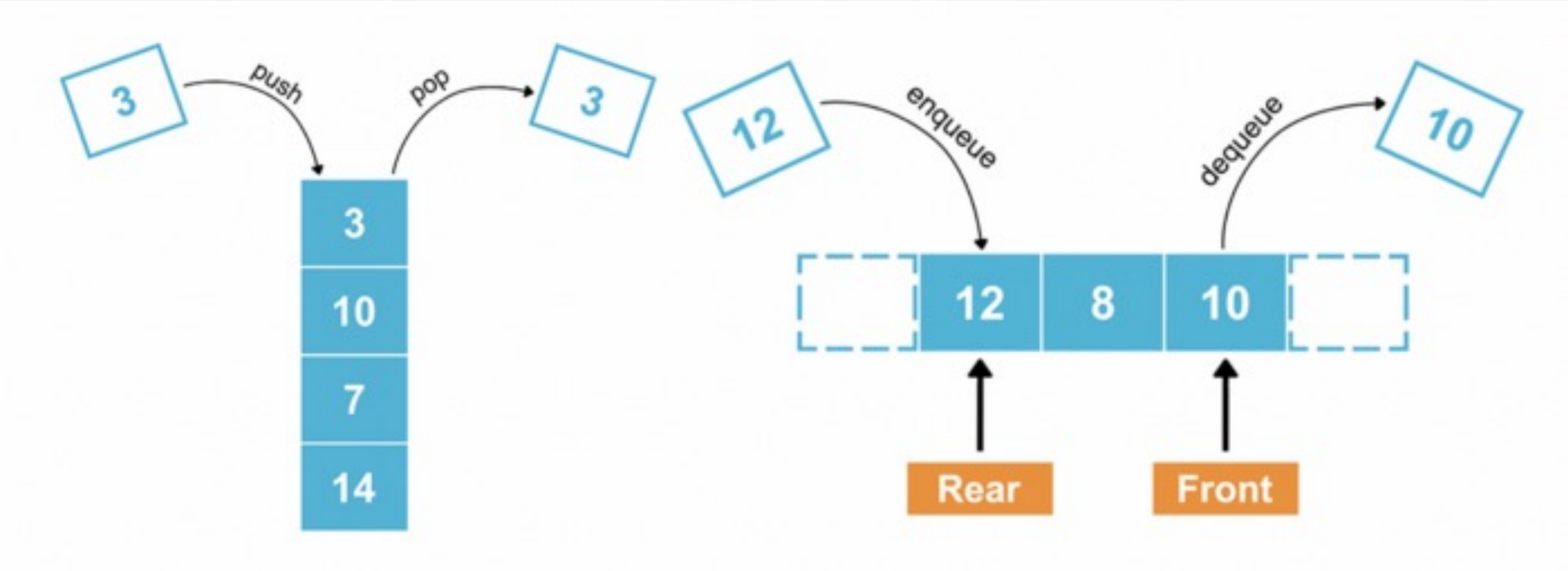
Heigth $h = ceil(log_b((b-1) * n))$

Tree vs. Graph



Comparison	Tree	Graph
Relationship of node	Only one root node. Parent-Child relationship exists.	No root node. No Parent-Child relationship exists.
Path	Only one path between two nodes	One or more paths exist between two nodes
Edge	N – 1 (N = Number of nodes)	Can not defined
Loop	Loop is not allowed	Loop is allowed
Traversal	Preorder, Inorder, Postorder	BFS, DFS
Model type	Hierarchical	Network

Stack & Queue



Asymptotic analysis

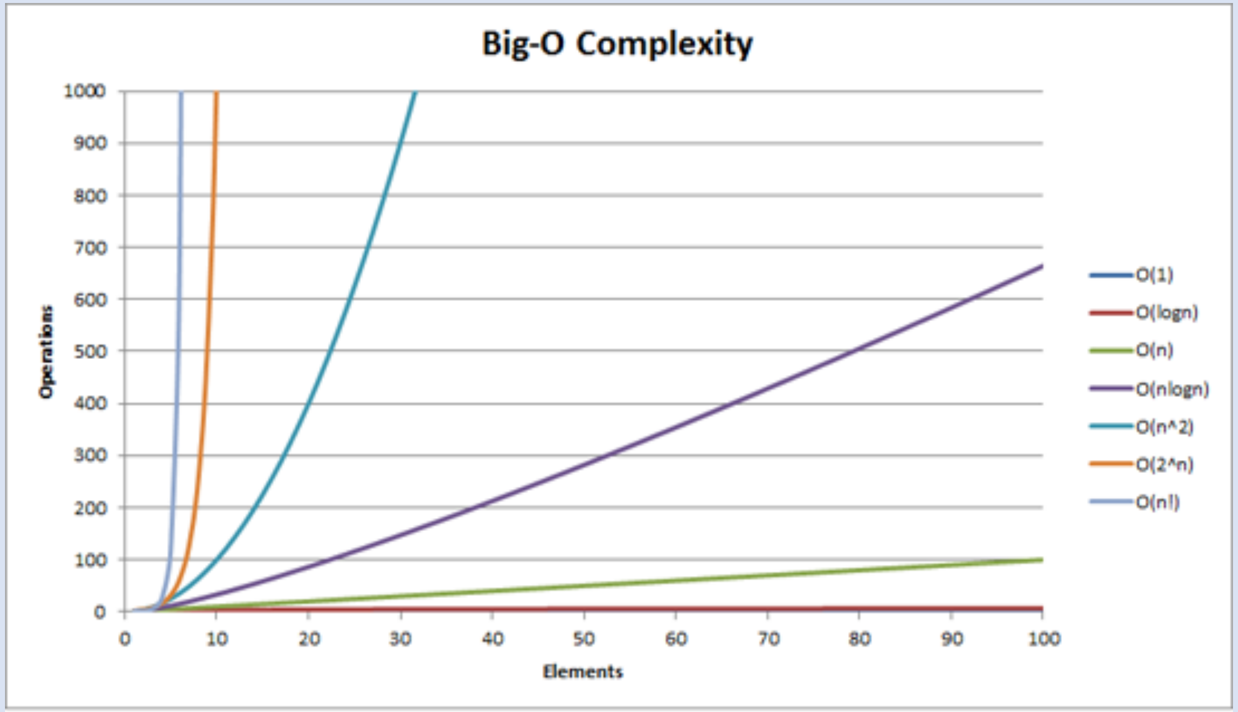
- *O(n)*, meaning that its measure is at most a constant times *n*, with the possible exception of a few small values of n

```
function SUMMATION(sequence) returns a number
    sum ← 0
    for i = 1 to LENGTH(sequence) do
        sum ← sum + sequence[i]
    return sum
```

- As *n* approaches infinity, an *O(n)* algorithm is better than an *O(n^2)* algorithm.

Examples

$O(1)$ – constant-time
$O(\log_2(n))$ – logarithmic-time
$O(n)$ – linear-time
$O(n^2)$ – quadratic-time
$O(n^k)$ – polynomial-time
$O(k^n)$ – exponential-time
$O(n!)$ – factorial-time



Big O Notation	Computations for 10 elements	Computations for 100 elements	Computations for 1000 elements
O(1)	1	1	1
O(log N)	3	6	9
O(N)	10	100	1000
O(N log N)	30	600	9000
O(N^2)	100	10000	1000000
O(2^N)	1024	1.26e+29	1.07e+301
O(N!)	3628800	9.3e+157	4.02e+2567

1. O(1)

```
void printFirstElementOfArray(int arr[])
{
    printf("First element of array = %d",arr[0]);
}
```

This function runs in `O(1)` time (or "constant time") relative to its input. The input array could be 1 item or 1,000 items, but this function would still just require one step.

2. O(n)

```
void printAllElementOfArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        printf("%d\n", arr[i]);
    }
}
```

This function runs in `O(n)` time (or "linear time"), where `n` is the number of items in the array. If the array has 10 items, we have to print 10 times. If it has 1000 items, we have to print 1000 times.

3. O(n^2)

```
void printAllPossibleOrderedPairs(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            printf("%d = %d\n", arr[i], arr[j]);
        }
     }
}
```

Here we're nesting two loops. If our array has `n` items, our outer loop runs `n` times and our inner loop runs `n` times for each iteration of the outer loop, giving us n^2 total prints. Thus this function runs in O(n^2) time (or "quadratic time"). If the array has 10 items, we have to print 100 times. If it has 1000 items, we have to print 1000000 times.

4. $O(2^n)$

```
int fibonacci(int num)
{
    if (num <= 1) return num;
    return fibonacci(num - 2) + fibonacci(num - 1);
}
```

An example of an $O(2^n)$ function is the recursive calculation of Fibonacci numbers. $O(2^n)$ denotes an algorithm whose growth doubles with each addition to the input data set. The growth curve of an $O(2^n)$ function is exponential - starting off very shallow, then rising meteorically.

5. Drop the constants

When you're calculating the big O complexity of something, you just throw out the constants. Like:

```
void printAllItemsTwice(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        printf("%d\n", arr[i]);
    }

    for (int i = 0; i < size; i++)
    {
        printf("%d\n", arr[i]);
    }
}
```

This is `O(2n)`, which we just call `O(n)`.

This is `O(2n)`, which we just call `O(n)`.

```
void printFirstItemThenFirstHalfThenSayHi100Times(int arr[], int size)
{
    printf("First element of array = %d\n",arr[0]);

    for (int i = 0; i < size/2; i++)
    {
        printf("%d\n", arr[i]);
    }

    for (int i = 0; i < 100; i++)
    {
        printf("Hi\n");
    }
}
```

This is `O(1 + n/2 + 100)`, which we just call `O(n)`.

Why can we get away with this? Remember, for big O notation we're looking at what happens as `n` gets arbitrarily large. As `n` gets really big, adding 100 or dividing by 2 has a decreasingly significant effect.

6. Drop the less significant terms

```
void printAllNumbersThenAllPairSums(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        printf("%d\n", arr[i]);
    }

    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            printf("%d\n", arr[i] + arr[j]);
        }
    }
}
```

Here our runtime is $O(n + n^2)$, which we just call $O(n^2)$.

Similarly:

- $O(n^3 + 50n^2 + 10000)$ is $O(n^3)$
- $O((n + 30) * (n + 5))$ is $O(n^2)$

Again, we can get away with this because the less significant terms quickly become, well, less significant as `n` gets big.

7. With Big-O, we're usually talking about the "worst case"

```
bool arrayContainsElement(int arr[], int size, int element)
{
    for (int i = 0; i < size; i++)
    {
        if (arr[i] == element) return true;
    }
    return false;
}
```

Here we might have 100 items in our array, but the first item might be the that element, in this case we would return in just 1 iteration of our loop.

In general we'd say this is `O(n)` runtime and the "worst case" part would be implied. But to be more specific we could say this is worst case `O(n)` and best case `O(1)` runtime. For some algorithms we can also make rigorous statements about the "average case" runtime.

```
for (int i = 1; i <= n; i = 2*i)
  for (int j = 1; j <= n/2; j++)
     k++;
```

O(n log n)

NP and inherently hard problems

- The class of polynomial problems - *P*
  - Problems which can be solved in time $O(n^k)$ for some k - is called *P*.
  - These are sometimes called "easy" problems, because the class contains those problems with running times like $O(\log n)$ and $O(n)$.
  - But it also contains those with time $O(n^{1000})$, so the name "easy" should not be taken too literally.

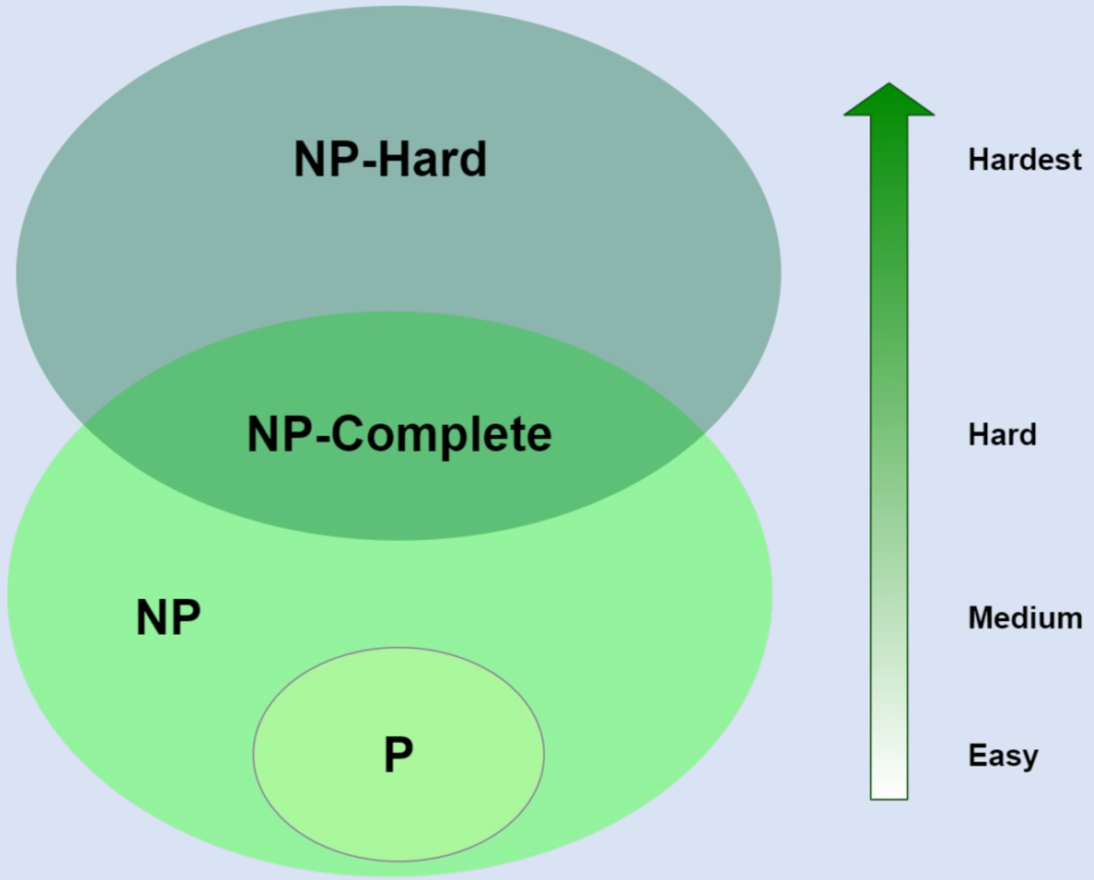
- Ex: Calculating the greatest common divisor.

NP and inherently hard problems

- Another important class of problems is NP
  - If there is some algorithm that can guess a solution and then verify whether a guess is correct in polynomial time.
  - With an arbitrarily large number of processors you can try all the guesses at once, or if you are very lucky and always guess right the first time, then the NP problems become P problems.
- One of the biggest open questions in computer science is whether the class NP is equivalent to the class P when one does not have the luxury of an infinite number of processors or omniscient guessing.
- Most computer scientists are convinced that $P \neq NP$; that NP problems are inherently hard and have no polynomial-time algorithms.
  - But this has never been proven.

NP and inherently hard problems

- *P* problems are quick to solve
- *NP* problems are quick to verify but slow to solve
- *NP-Complete* problems are also quick to verify, slow to solve and can be reduced to any other NP-Complete problem
- *NP-Hard* problems are slow to verify, slow to solve and can be reduced to any other NP problem



Problem-solving Agent

- When the correct action to take is not immediately obvious, an agent may need to plan ahead:
  - to consider a sequence of actions that form a path to a goal state.
- Such an agent is called a problem-solving agent, and the computational process it undertakes is called search.

Types of Search Algorithms

- informed algorithms,
  - the agent can estimate how far it is from the goal.
- uninformed algorithms,
  - where no estimate is available.

Problem-solving Process

- On holiday in Romania; currently in Arad.

- Formulate goal:
  - be in Bucharest

- Formulate problem:
  - states: various cities
  - actions: drive between cities

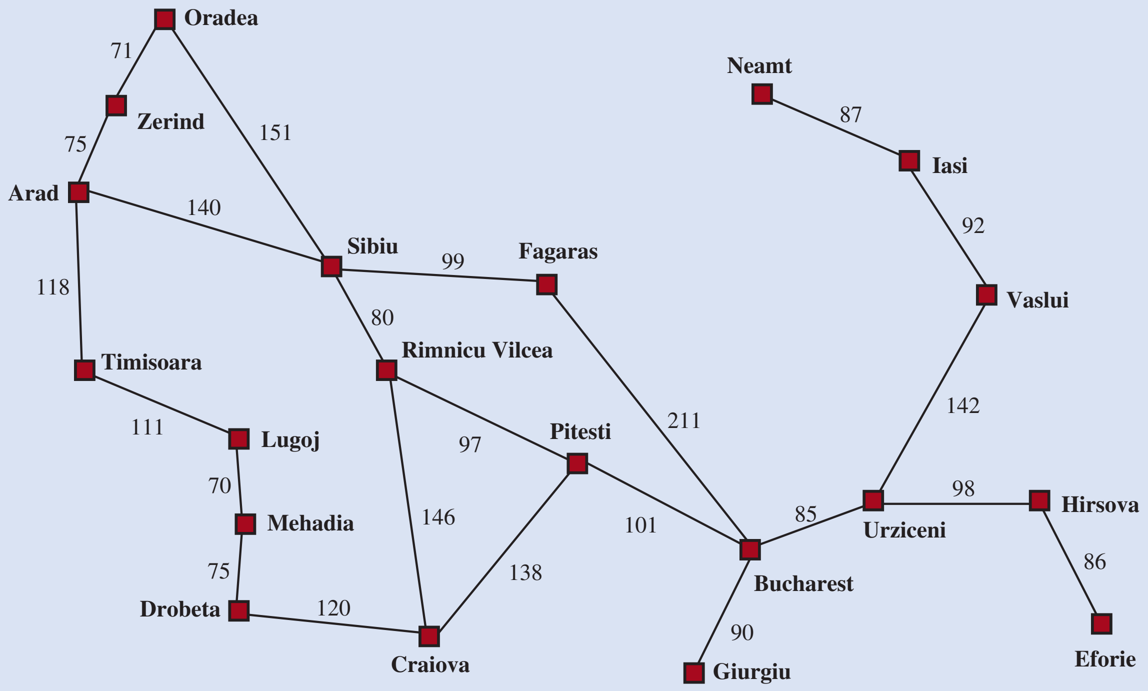


Figure 3.1 A simplified road map of part of Romania, with road distances in miles.

- Find solution:
  - the agent simulates sequences of actions in its model, searching until it finds a sequence of actions that reaches the goal.
  - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Search Problem

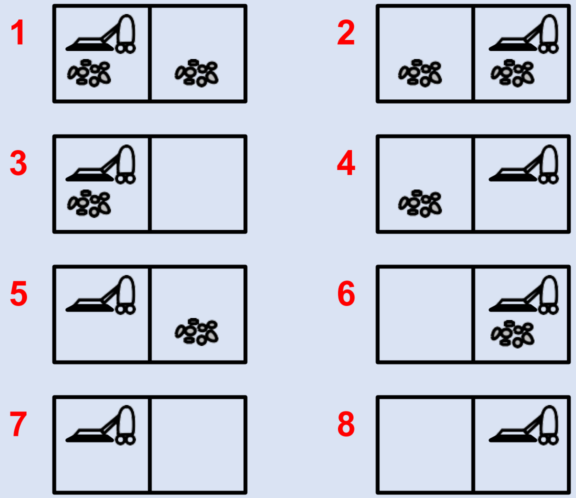
- A search problem can be defined formally as follows:
  - state space - a set of possible states that the environment can be in.
  - initial state - the agent starts in
  - a set of one or more goal states.
  - actions available to the agent, *ACTIONS(s)*
    - *ACTIONS*(Arad) = {ToSibiu, ToTimisoara, ToZerind}.
  - a transition model, which describes what each action does, *RESULT(s,a)*
    - *RESULT*(Arad,ToZerind)=Zerind.
  - an action cost function, denoted by *ACTION-COST(s, a, s')*

Search Problem

- A sequence of actions forms a path
- A solution is a path from the initial state to a goal state.
- We assume that action costs are additive; that is, the total cost of a path is the sum of the individual action costs.
- An optimal solution has the lowest path cost among all solutions.

Example: vacuum world

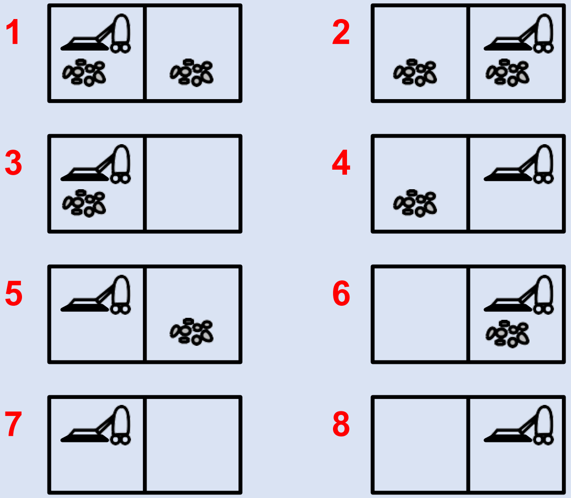
Single-state, start in #5. Solution??



Example: vacuum world

Single-state, start in #5. Solution??
$[Right, Suck]$

Conformant, start in $\{1, 2, 3, 4, 5, 6, 7, 8\}$
e.g., $Right$ goes to $\{2, 4, 6, 8\}$. Solution??

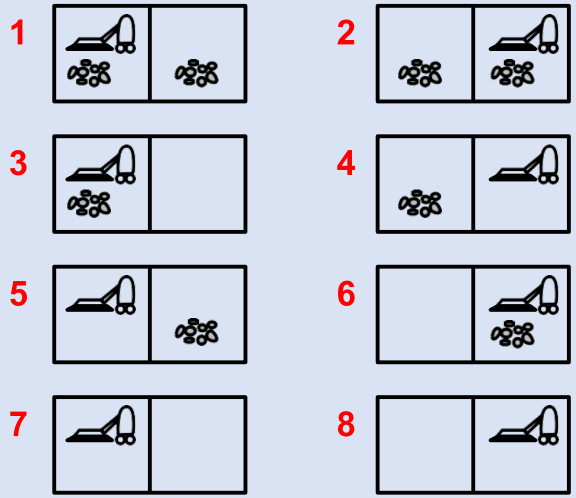


Example: vacuum world

Single-state, start in #5. Solution??
$[Right, Suck]$

Conformant, start in $\{1, 2, 3, 4, 5, 6, 7, 8\}$
e.g., $Right$ goes to $\{2, 4, 6, 8\}$. Solution??
$[Right, Suck, Left, Suck]$

Contingency, start in #5
Murphy's Law: $Suck$ can dirty a clean carpet
Local sensing: dirt, location only.
Solution??

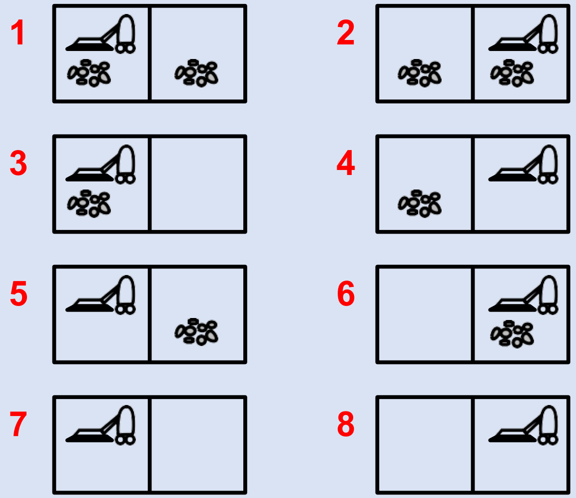


Example: vacuum world

Single-state, start in #5. Solution??
$[Right,\ Suck]$

Conformant, start in $\{1, 2, 3, 4, 5, 6, 7, 8\}$
e.g., $Right$ goes to $\{2, 4, 6, 8\}$. Solution??
$[Right,\ Suck,\ Left,\ Suck]$

Contingency, start in #5
Murphy's Law: $Suck$ can dirty a clean carpet
Local sensing: dirt, location only.
Solution??
$[Right,\ \text{if}\ \ dirt\ \text{then}\ \ Suck]$



State-space

- states??: integer dirt and robot locations (ignore dirt amounts etc.)
- actions??: Left, Right, Suck, NoOp
- goal test??: no dirt
- path cost??: 1 per action (0 for NoOp)

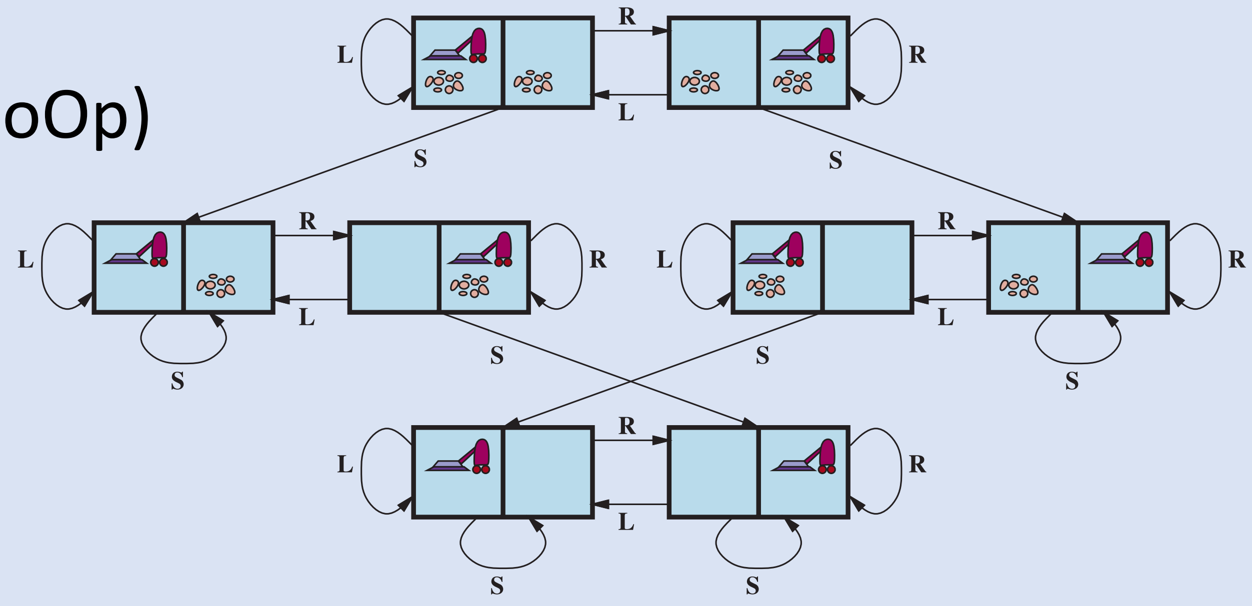


Figure 3.2 The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: L = *Left*, R = *Right*, S = *Suck*.

8-puzzle Game

- states??: integer locations of tiles (ignore intermediate positions)
- actions??: 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]

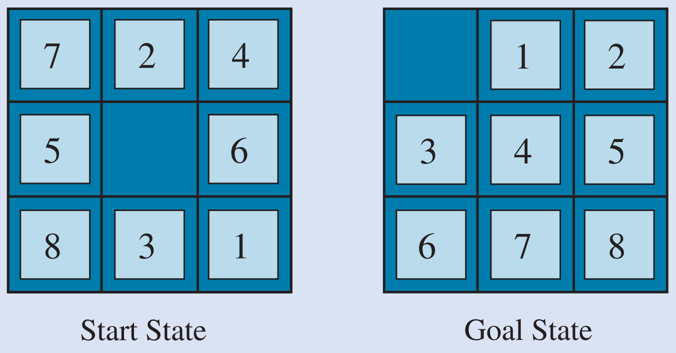


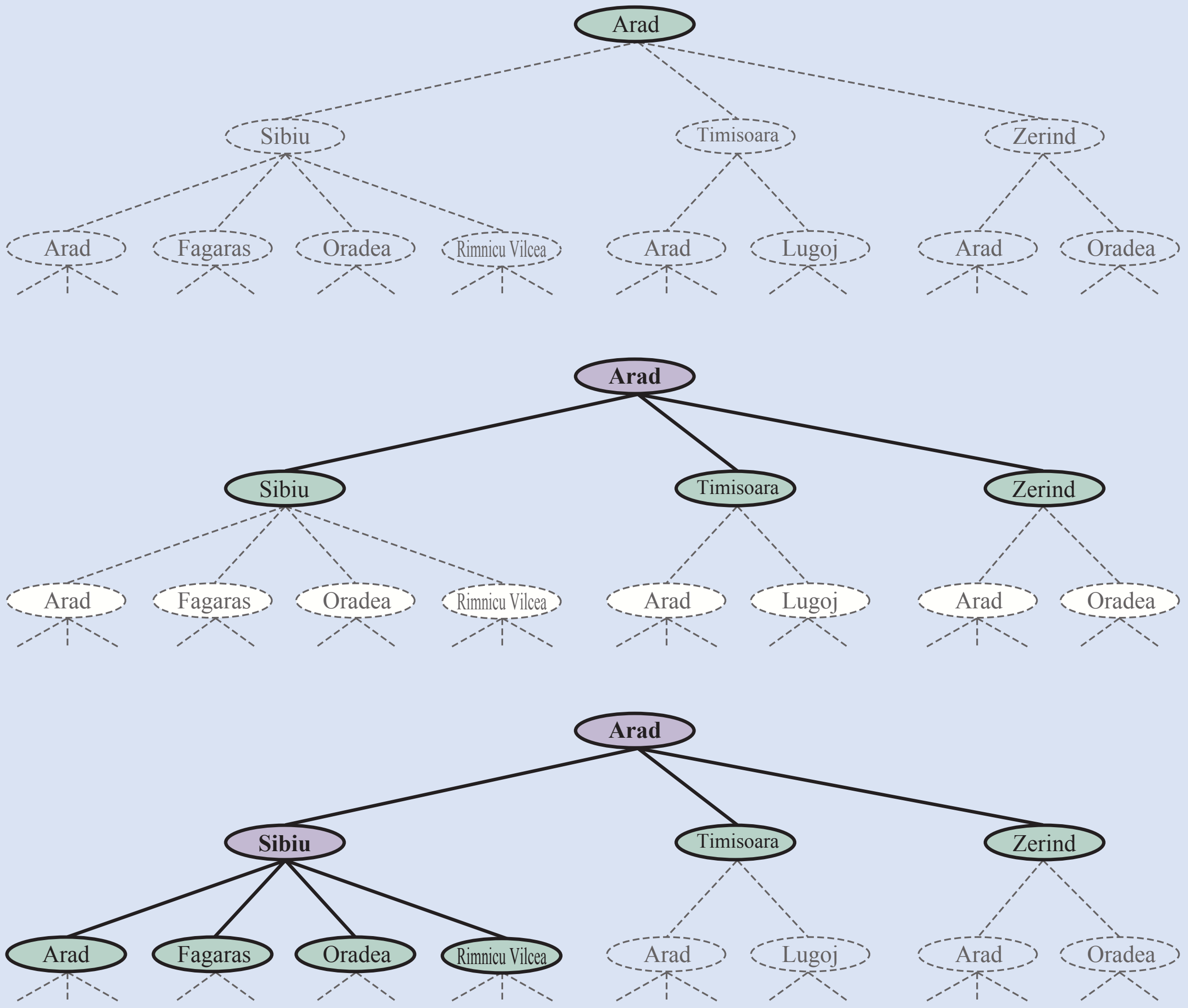
Figure 3.3 A typical instance of the 8-puzzle.

Search Algorithms

- A search algorithm
  - takes a search problem as input
  - returns a solution, or an indication of failure.
- Superimpose a search tree over the state-space graph, forming various paths from the initial state, trying to find a path that reaches a goal state.
  - Each node corresponds to a state in the state space
  - Edges correspond to actions.
  - The root is the initial state of the problem.

Search Tree

- Nodes that have been expanded are lavender with bold letters;
- Nodes on the frontier that have been generated but not yet expanded are in green;
- States of these two types of nodes are said to have been reached.
- Nodes that could be generated next are shown in faint dashed lines.
- Notice in the bottom tree there is a cycle from Arad to Sibiu to Arad;
  - that can't be an optimal path, so search should not continue from there.



Search Tree

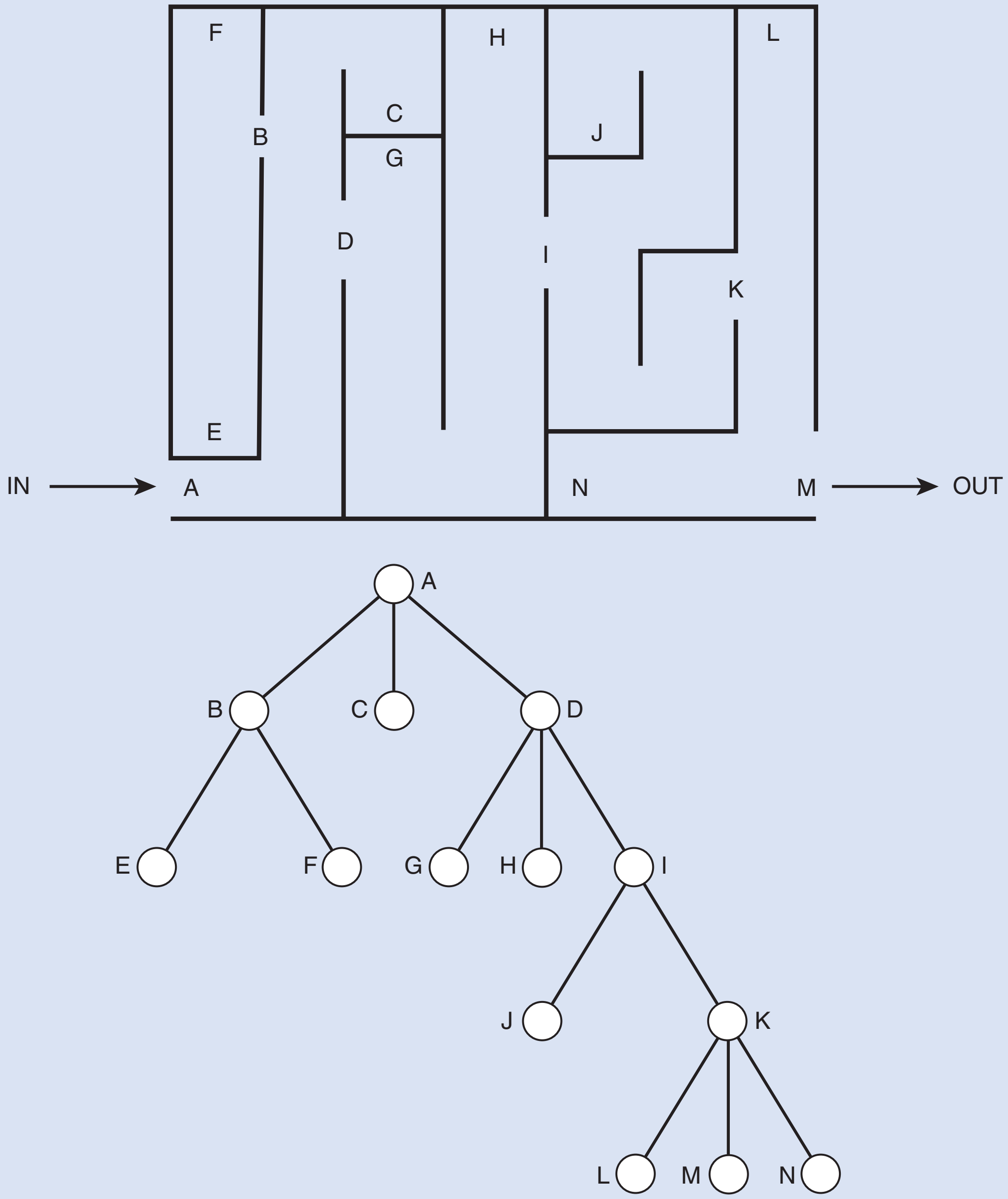


Figure 4.3

A maze and a search tree representation of the maze.

Search Tree

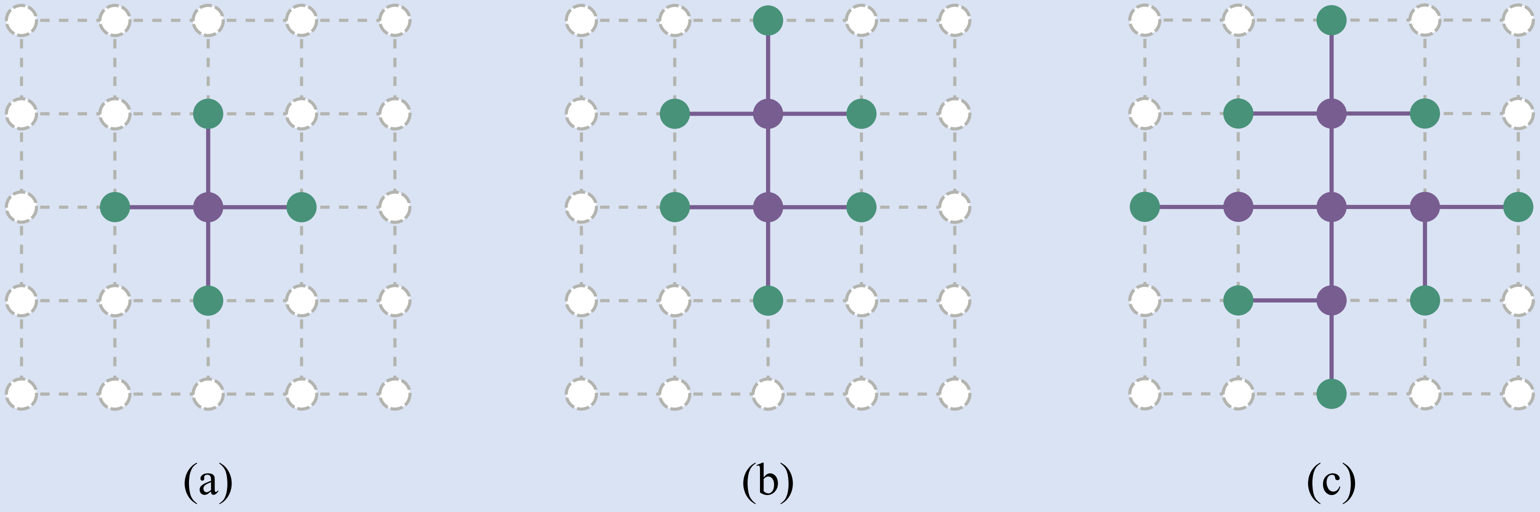


Figure 3.6 The separation property of graph search, illustrated on a rectangular-grid problem. The frontier (green) separates the interior (lavender) from the exterior (faint dashed). The frontier is the set of nodes (and corresponding states) that have been reached but not yet expanded; the interior is the set of nodes (and corresponding states) that have been expanded; and the exterior is the set of states that have not been reached. In (a), just the root has been expanded. In (b), the top frontier node is expanded. In (c), the remaining successors of the root are expanded in clockwise order.

Search data structures

- node.STATE: the state to which the node corresponds;
- node.PARENT: the node in the tree that generated this node;
- node.ACTION: the action that was applied to the parent's state to generate this node;
- node.PATH-COST: the total cost of the path from the initial state to this node.

Search data structures

- We need a data structure to store the frontier. The appropriate choice is a queue of some kind, because the operations on a frontier are:

  - IS-EMPTY(frontier) returns true only if there are no nodes in the frontier.
  - POP(frontier) removes the top node from the frontier and returns it.
  - TOP(frontier) returns (but does not remove) the top node of the frontier.
  - ADD(node, frontier) inserts node into its proper place in the queue.

Search data structures

- Three kinds of queues are used in search algorithms:
  - A priority queue first pops the node with the minimum cost according to some evaluation function, f . It is used in best-first search.
  - A FIFO queue or first-in-first-out queue first pops the node that was added to the queue first; we shall see it is used in breadth-first search.
  - A LIFO queue or last-in-first-out queue (also known as a stack) pops first the most recently added node; we shall see it is used in depth-first search.

Implementation: general tree search

```
function Tree-Search( problem, frontier ) returns a solution, or failure
   frontier ← Insert (Make-Node(Initial-State [problem]), frontier)
   loop do
       if frontier is empty then return failure
       node ← Remove-Front(frontier)
       if Goal-Test(problem, State(node)) then return node
       frontier ← InsertAll(Expand(node, problem), frontier)
```

Measuring problem-solving performance

- Completeness: Is the algorithm guaranteed to find a solution when there is one, and to correctly report failure when there is not?
- Cost optimality: Does it find a solution with the lowest path cost of all solutions?
- Time complexity: How long does it take to find a solution?
  - This can be measured in seconds, or more abstractly by the number of states and actions considered.
- Space complexity: How much memory is needed to perform the search?

- 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 ∞)

Uninformed Search Strategies

- An uninformed search algorithm is given no clue about how close a state is to the goal(s).
- Consider our agent in Arad with the goal of reaching Bucharest.
- An uninformed agent with no knowledge of Romanian geography has no clue whether going to Zerind or Sibiu is a better first step.

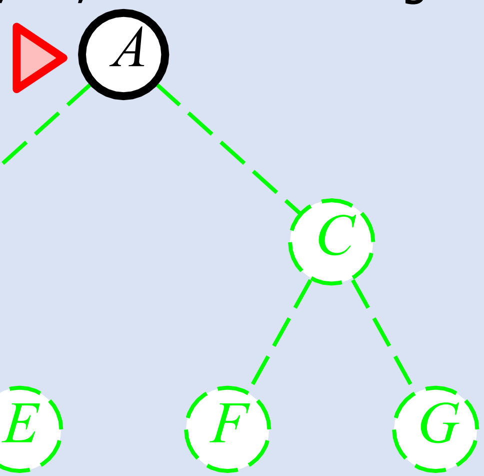
    - Breadth-first search
    - Uniform-cost search
    - Depth-first search
    - Depth-limited search
    - Iterative deepening search

Breadth-first search

Expand shallowest unexpanded node

Implementation:

frontier is a FIFO queue, i.e., new successors go at end

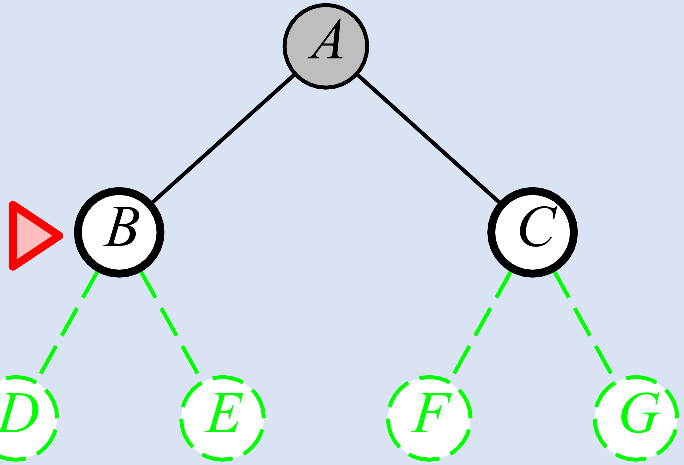


Breadth-first search

Expand shallowest unexpanded node

Implementation:

frontier is a FIFO queue, i.e., new successors go at end

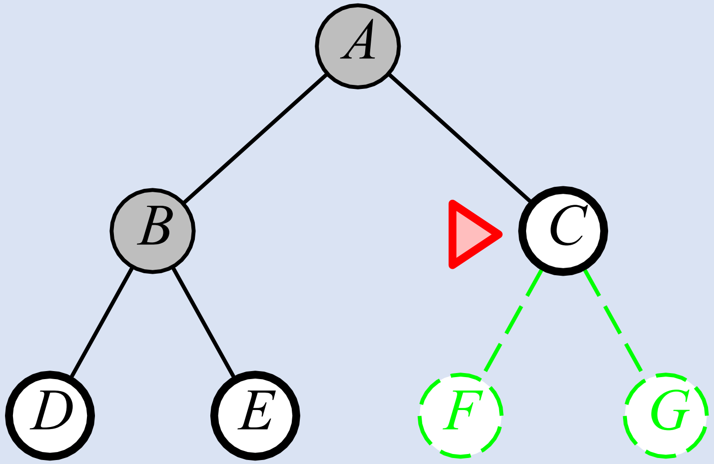


Breadth-first search

Expand shallowest unexpanded node

Implementation:

frontier is a FIFO queue, i.e., new successors go at end

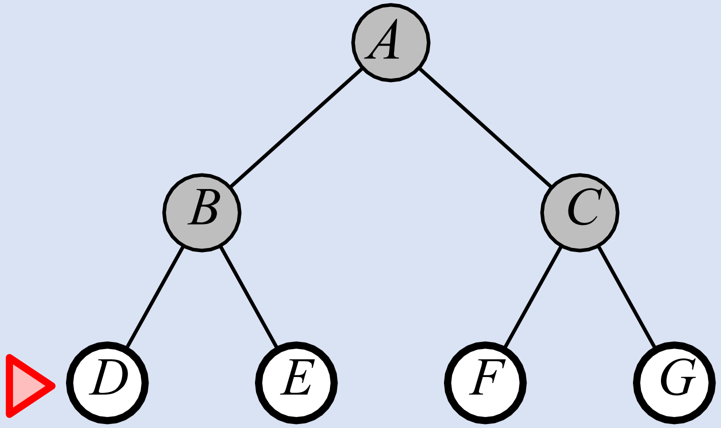


Breadth-first search

Expand shallowest unexpanded node

Implementation:

frontier is a FIFO queue, i.e., new successors go at end



Breadth-first implementation

```
Function breadth ()
{
     queue = [];      // initialize an empty queue
     state = root_node;   // initialize the start state
     while (true)
{
          if is_goal (state)
               then return SUCCESS
          else add_to_back_of_queue (successors (state));
          if queue == []
               then report FAILURE;
          state = queue [0]; // state = first item in queue
          remove_first_item_from (queue);
     }
}
```

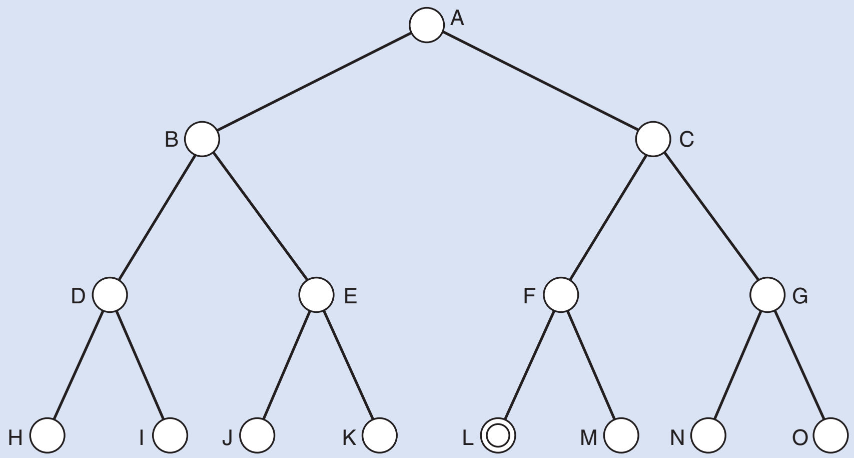


Table 4.3 Analysis of breadth-first search of tree shown in Figure 4.4

Step	State	Queue	Notes
1	A	(empty)	The queue starts out empty, and the initial state is the root node, which is A.
2	A	B,C	The two descendents of A are added to the queue.
3	B	C	
4	B	C,D,E	The two descendents of the current state, B, are added to the back of the queue.
5	C	D,E	
6	C	D,E,F,G	
7	D	E,F,G	
8	D	E,F,G,H,I	
9	E	F,G,H,I	
10	E	F,G,H,I,J,K	
11	F	G,H,I,J,K	
12	F	G,H,I,J,K,L,M	
13	G	H,I,J,K,L,M	
14	G	H,I,J,K,L,M,N,O	
15	H	I,J,K,L,M,N,O	H has no successors, so we have nothing to add to the queue in this state, or in fact for any subsequent states.
16	I	J,K,L,M,N,O	
17	J	K,L,M,N,O	
18	K	L,M,N,O	
19	L	M,N,O	SUCCESS: A goal state has been reached.

Properties of breadth-first search

Complete??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time??

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., exp. in d

Space??

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., exp. in d

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

Optimal??

Properties of breadth-first search

Complete?? Yes (if b is finite)

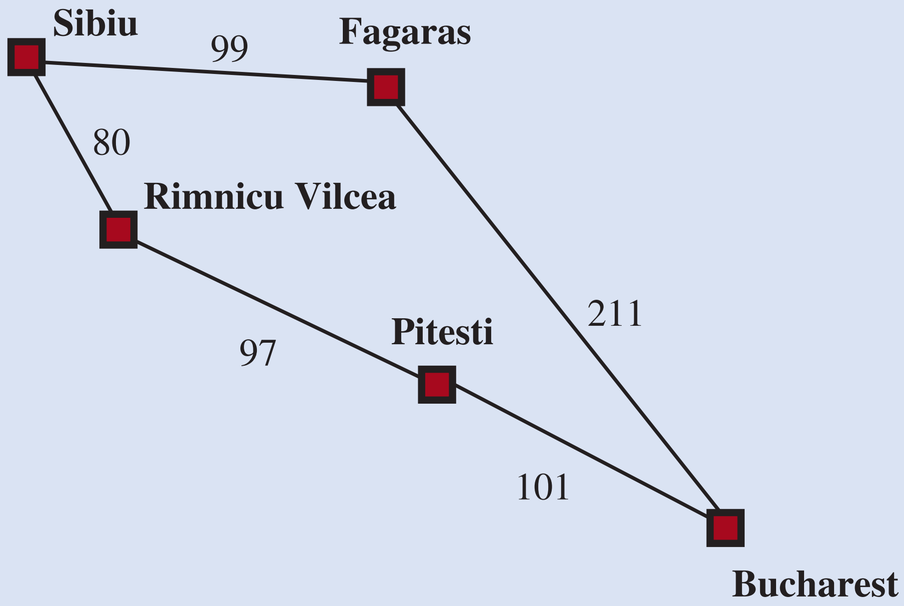
Time?? $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$, i.e., exp. in d

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 100MB/sec
so 24hrs = 8640GB.

Uniform-cost search



Expand least-cost unexpanded node

Implementation:

frontier = queue ordered by path cost, lowest first

Equivalent to breadth-first if step costs all equal

Complete?? Yes, complete, if there is a solution, it will find it.

Time?? **$O(b^{1 + \lceil C^*/\varepsilon \rceil})$**

ε a lower bound on the cost of each action, with $\varepsilon > 0$

where C^* is the cost of the optimal solution

Space?? Same with time complexity

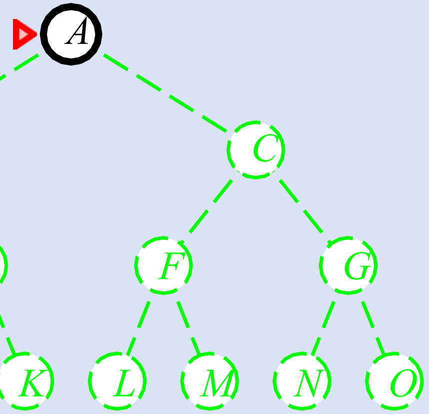
Optimal?? Yes

Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front

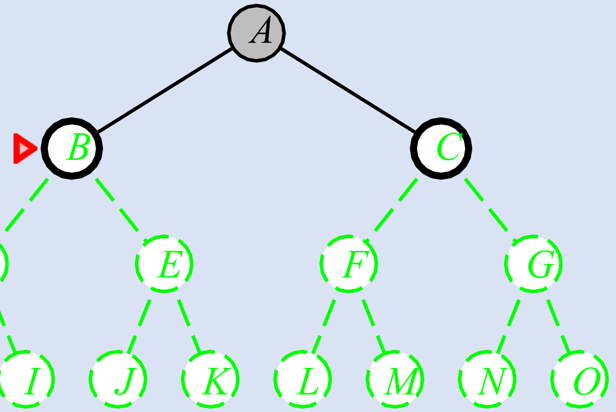


Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front

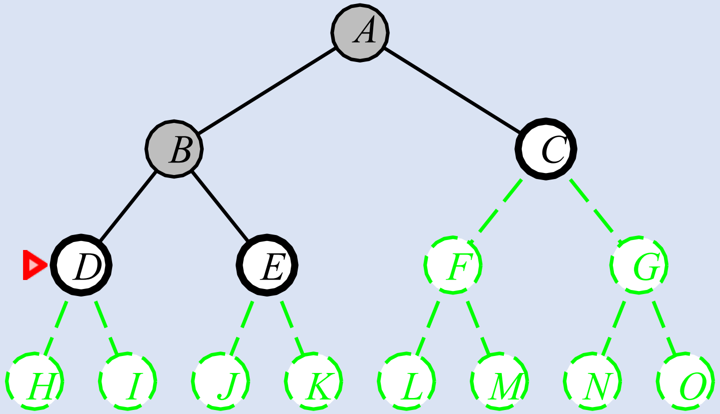


Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front

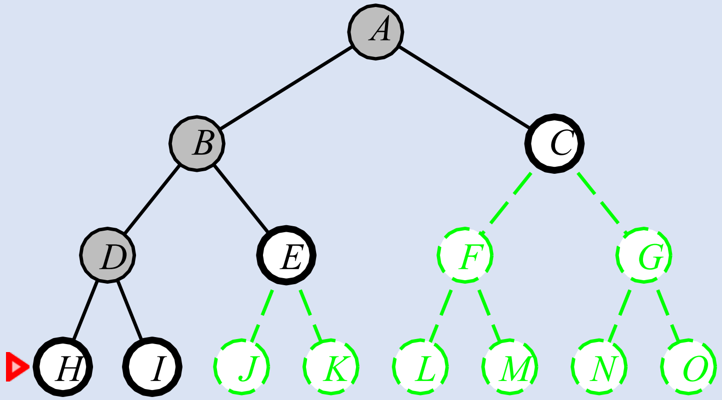


Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front

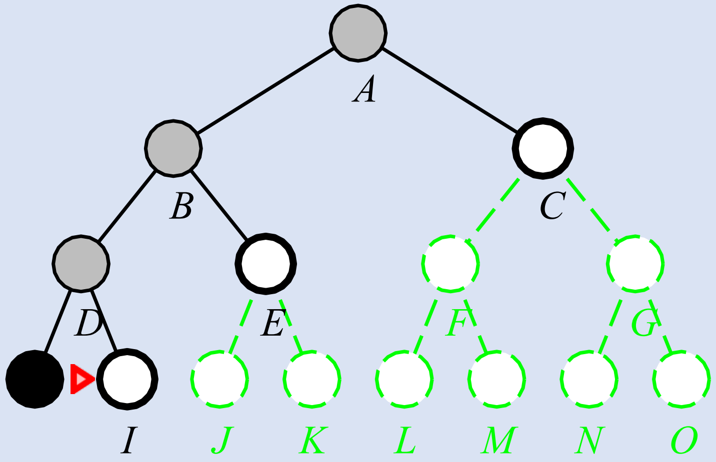


Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front

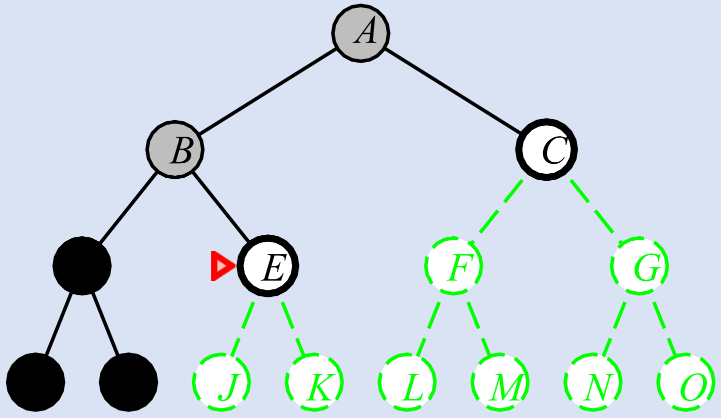


Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front

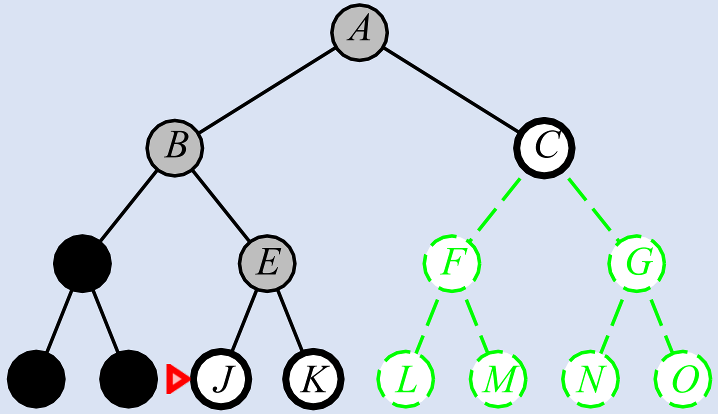


Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front

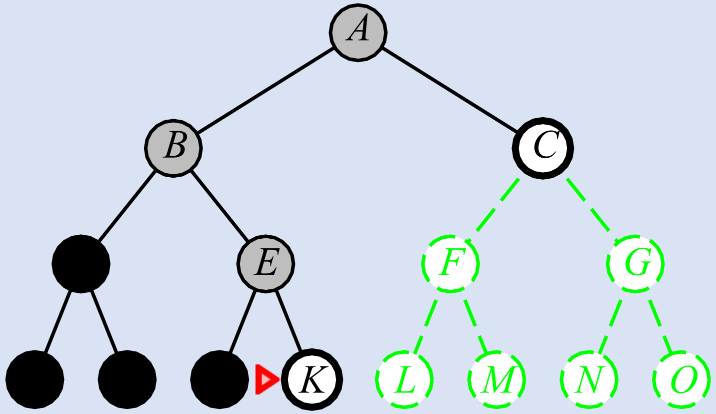


Depth-first search

Expand deepest unexpanded node

Implementation:

frontier= LIFO queue, i.e., put successors at front

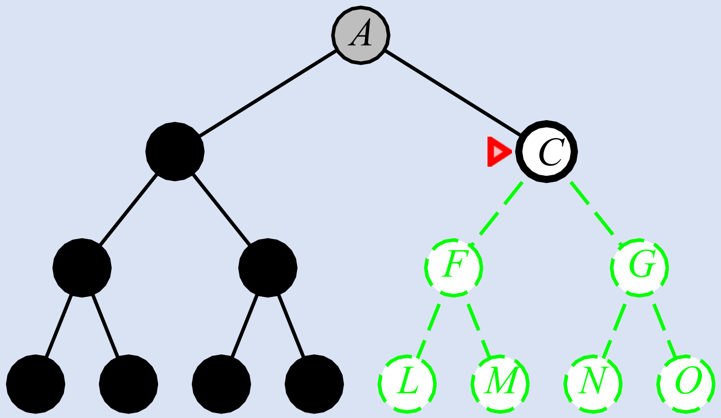


Depth-first search

Expand deepest unexpanded node

Implementation:

frontier= LIFO queue, i.e., put successors at front

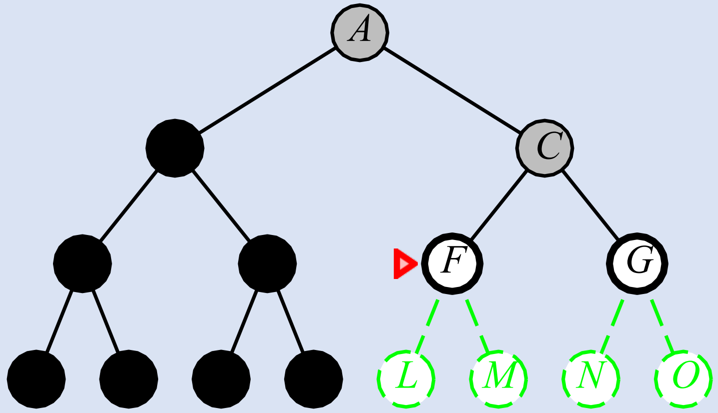


Depth-first search

Expand deepest unexpanded node

Implementation:

frontier= LIFO queue, i.e., put successors at front

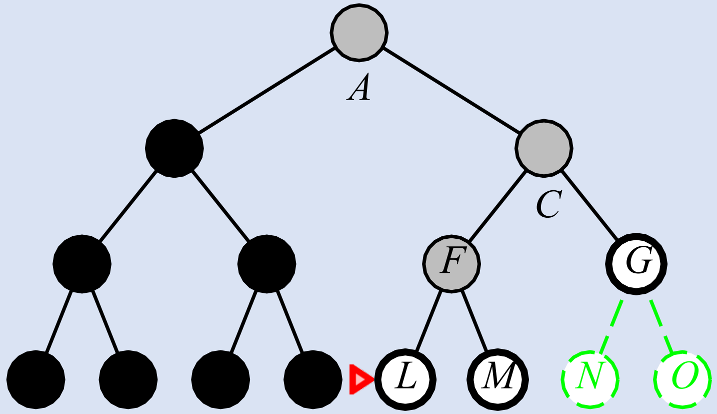


Depth-first search

Expand deepest unexpanded node

Implementation:

frontier= LIFO queue, i.e., put successors at front

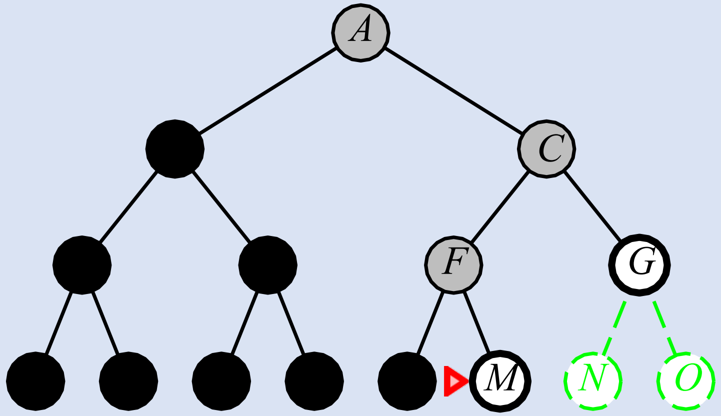


Depth-first search

Expand deepest unexpanded node

Implementation:

frontier= LIFO queue, i.e., put successors at front



Depth-first implementation

```
Function depth ()
{
     queue = [];     // initialize an empty queue
     state = root_node;   // initialize the start state
     while (true)
     {
          if is_goal (state)
               then return SUCCESS
          else add_to_front_of_queue (successors (state));
          if queue == []
               then report FAILURE;
          state = queue [0]; // state = first item in queue
          remove_first_item_from (queue);
     }
}
```

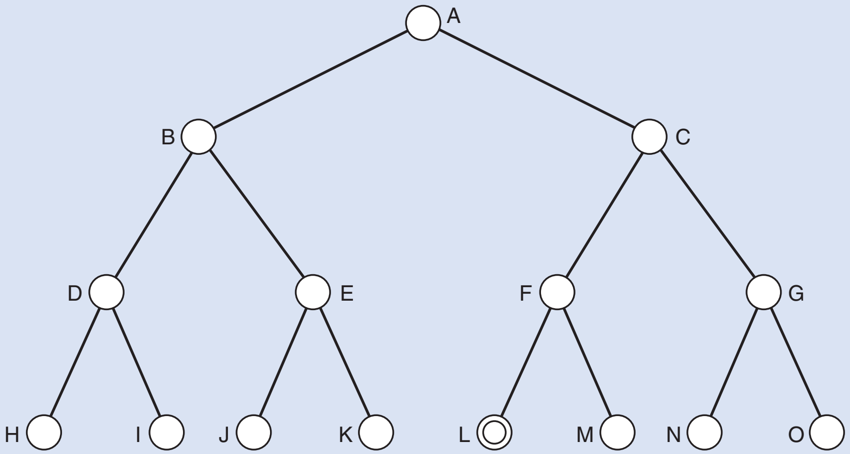


Table 4.2 Analysis of depth-first search of tree shown in Figure 4.5

Step	State	Queue	Notes
1	A	(empty)	The queue starts out empty, and the initial state is the root node, which is A.
2	A	B,C	The successors of A are added to the queue.
3	B	C	
4	B	D,E,C	The successors of the current state, B, are added to the front of the queue.
5	D	E,C	
6	D	H,I,E,C	
7	H	I,E,C	H has no successors, so no new nodes are added to the queue.
8	I	E,C	Similarly, I has no successors.
9	E	C	
10	E	J,K,C	
11	J	K,C	Again, J has no successors.
12	K	C	K has no successors. Now we have explored the entire branch below B, which means we back-track up to C.
13	C	(empty)	The queue is empty, but we are not at the point in the algorithm where this would mean failing because we are about to add successors of C to the queue.
14	C	F,G	
15	F	G	
16	F	L,M,G	
17	L	M,G	SUCCESS: the algorithm ends because a goal node has been located. In this case, it is the only goal node, but the algorithm does not know that and does not know how many nodes were left to explore.

Properties of depth-first search

Complete??

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??

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??

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??

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

Comparison of depth-first and bread-first search

Table 4.1 Comparison of depth-first and breadth-first search

Scenario	Depth first	Breadth first
Some paths are extremely long, or even infinite	Performs badly	Performs well
All paths are of similar length	Performs well	Performs well
All paths are of similar length, and all paths lead to a goal state	Performs well	Wasteful of time and memory
High branching factor	Performance depends on other factors	Performs poorly

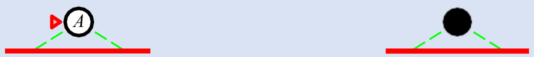
Depth-limited & Iterative deepening search

- Iterative deepening repeatedly applies depth-limited search with increasing limits.
- It returns one of three different types of values:
  - a solution node;
  - or failure, when it has exhausted all nodes and there is no solution at any depth;
  - or cutoff, to mean there might be a solution at a deeper depth than ℓ.
- This is a tree-like search algorithm that does not keep track of reached states, and thus uses much less memory than best-first search, but runs the risk of visiting the same state multiple times on different paths.

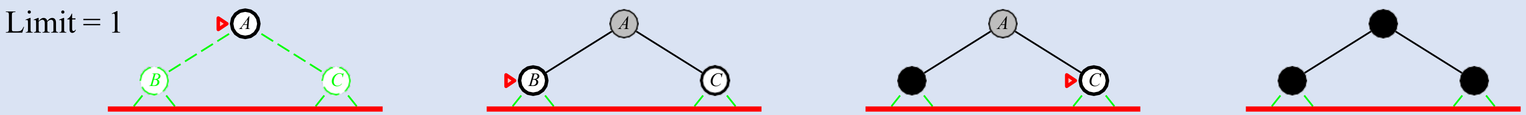
```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution node or failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result

function DEPTH-LIMITED-SEARCH(problem, ℓ) returns a node or failure or cutoff
  frontier ← a LIFO queue (stack) with NODE(problem.INITIAL) as an element
  result ← failure
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node) > ℓ then
      result ← cutoff
    else if not IS-CYCLE(node) do
      for each child in EXPAND(problem, node) do
        add child to frontier
  return result
```

Iterative deepening search l = 0

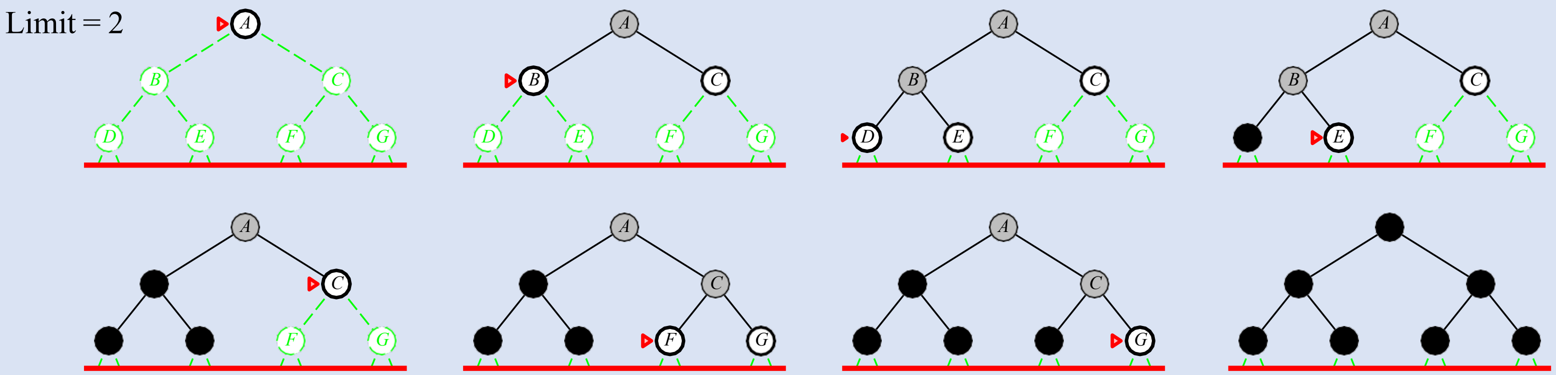
Limit = 0



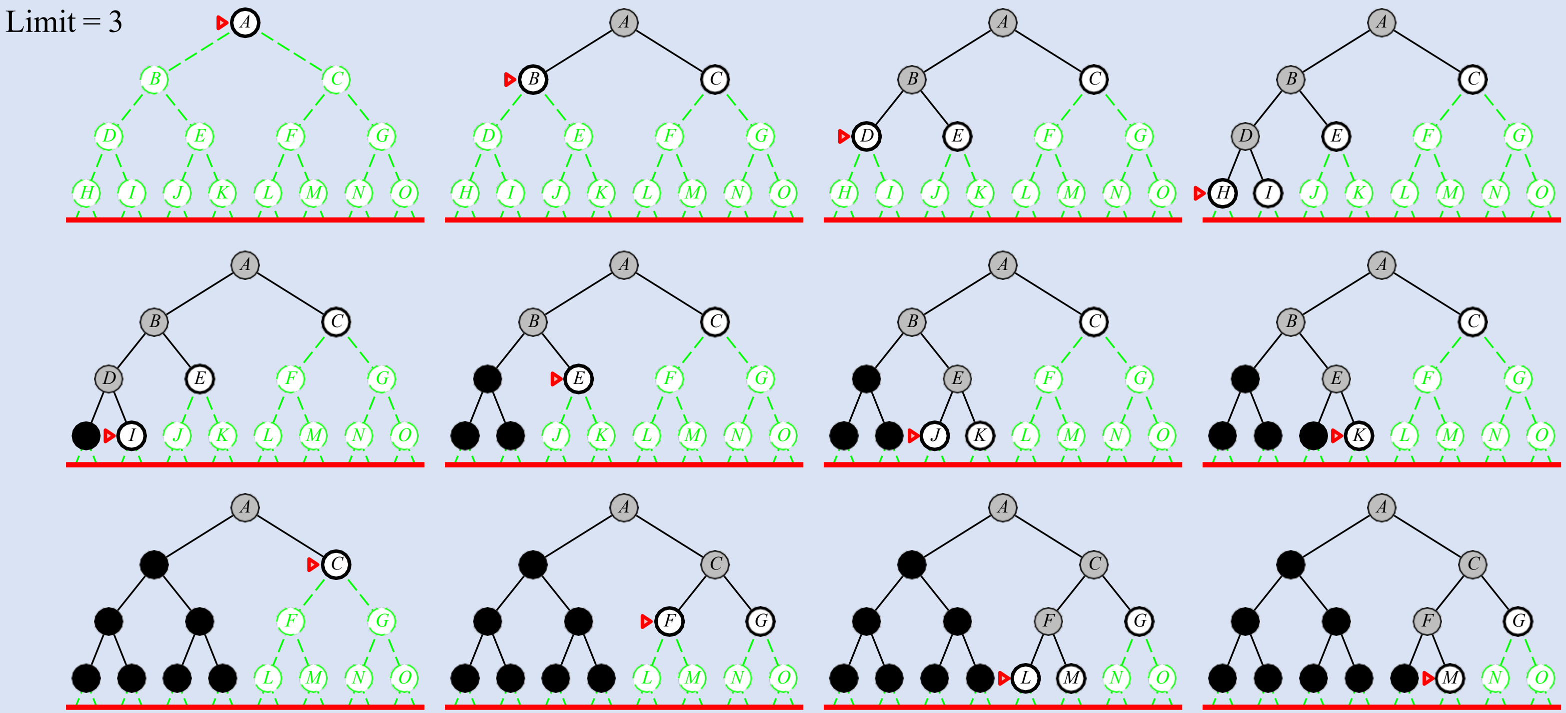
Iterative deepening search l = 1



Iterative deepening search l = 2



Iterative deepening search l = 3



Properties of iterative deepening search

Complete??

Properties of iterative deepening search

Complete?? Yes

Time??

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??

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
Can be modified to explore uniform-cost tree

$$N(\text{IDS}) = (d)b^1 + (d-1)b^2 + (d-2)b^3 \cdots + b^d$$

Numerical comparison for $b = 10$ and $d = 5$, solution at far right leaf:

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

IDS does better because other nodes at depth d are not expanded

BFS can be modified to apply goal test when a node is generated

Summary of Uninformed Search algorithms

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	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Figure 3.15 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 $\geq \epsilon > 0$; [3] cost-optimal if action costs are all identical; [4] if both directions are breadth-first or uniform-cost.

Informed Search Algorithms

- Best-first search
- Greedy best-first search
- A*

Best-first search

- Choose a node, n, with minimum value of some evaluation function, f (n).
- On each iteration we choose a node on the frontier with minimum f(n) value, return it if its state is a goal state, and otherwise apply EXPAND to generate child nodes.
- Each child node is added to the frontier if it has not been reached before, or
  - is re-added if it is now being reached with a path that has a lower path cost than any previous path.

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s′ ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s′)
    yield NODE(STATE=s′, PARENT=node, ACTION=action, PATH-COST=cost)
```

Figure 3.7 The best-first search algorithm, and the function for expanding a node. The data structures used here are described in Section 3.3.2. See Appendix B for **yield**.

Best-first search

```
Function best ()
{
     queue = [];     // initialize an empty queue
     state = root_node;   // initialize the start state
     while (true)
     {
          if is_goal (state)
               then return SUCCESS
          else
          {
               add_to_front_of_queue (successors (state));
               sort (queue);
          }
          if queue == []
               then report FAILURE;
          state = queue [0]; // state = first item in queue
          remove_first_item_from (queue);
     }
}
```

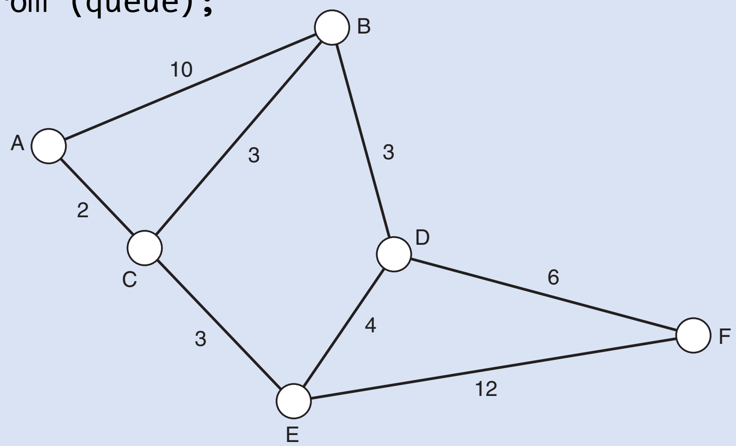
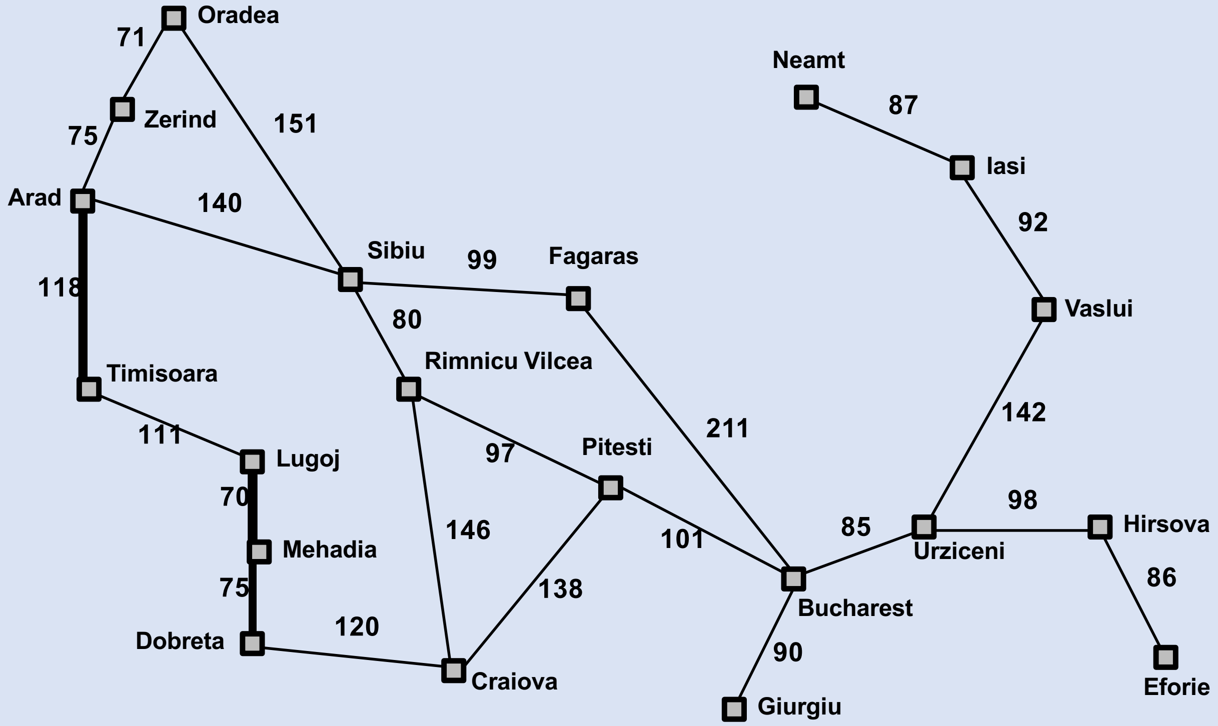


Table 4.5 Analysis of best-first search of tree shown in Figure 4.4

Step	State	Queue	Notes
1	A	(empty)	The queue starts out empty, and the initial state is the root node, which is A.
2	A	B,C	The successors of the current state, B and C, are placed in the queue.
	A	B,C	The queue is sorted, leaving B in front of C because it is closer to the goal state, F.
3	B	C	
4	B	D,C,C	The children of node B are added to the front of the queue.
5	B	D,C,C	The queue is sorted, leaving D at the front because it is closer to the goal node than C.
6	D	C,C	Note that although the queue appears to contain the same node twice, this is just an artifact of the way the search tree was constructed. In fact, those two nodes are distinct and represent different paths on our search tree.
7	D	E,F,C,C	The children of D are added to the front of the queue.
8	D	F,E,C,C	The queue is sorted, moving F to the front.
9	F	E,C,C	SUCCESS: Path is reported as A,B,D,F.

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

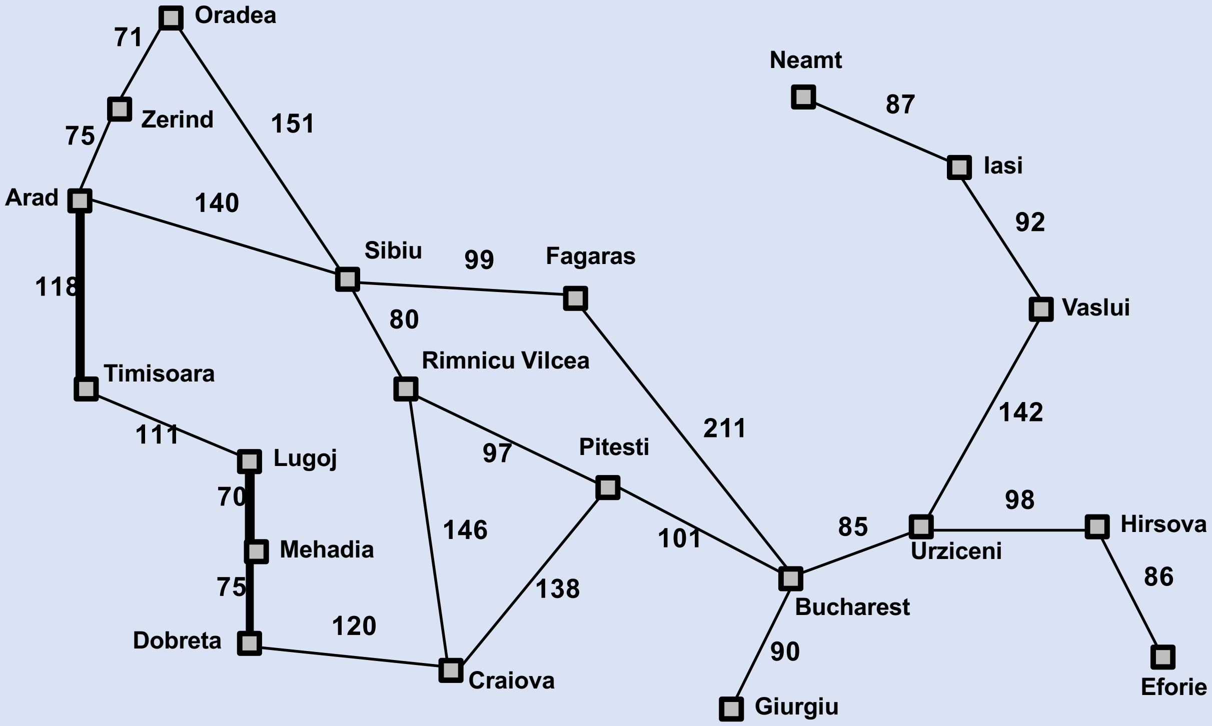
Greedy search

Evaluation function $h(n)$ (heuristic)
= estimate of cost from n to the closest goal

E.g., $h_{\mathrm{SLD}}(n)$ = straight-line distance from n to Bucharest

Greedy search expands the node that appears to be closest to goal

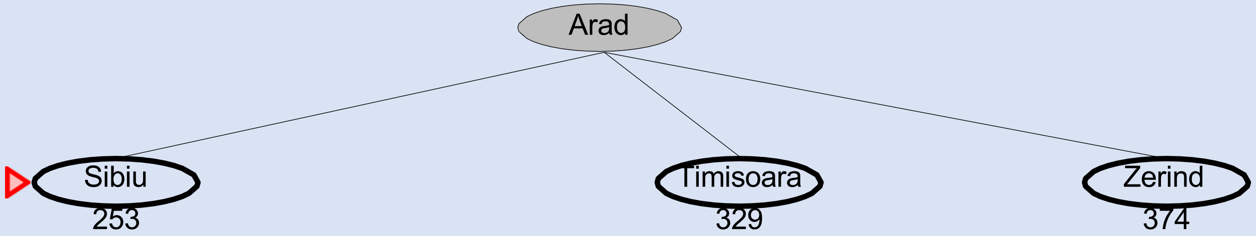
Greedy search example

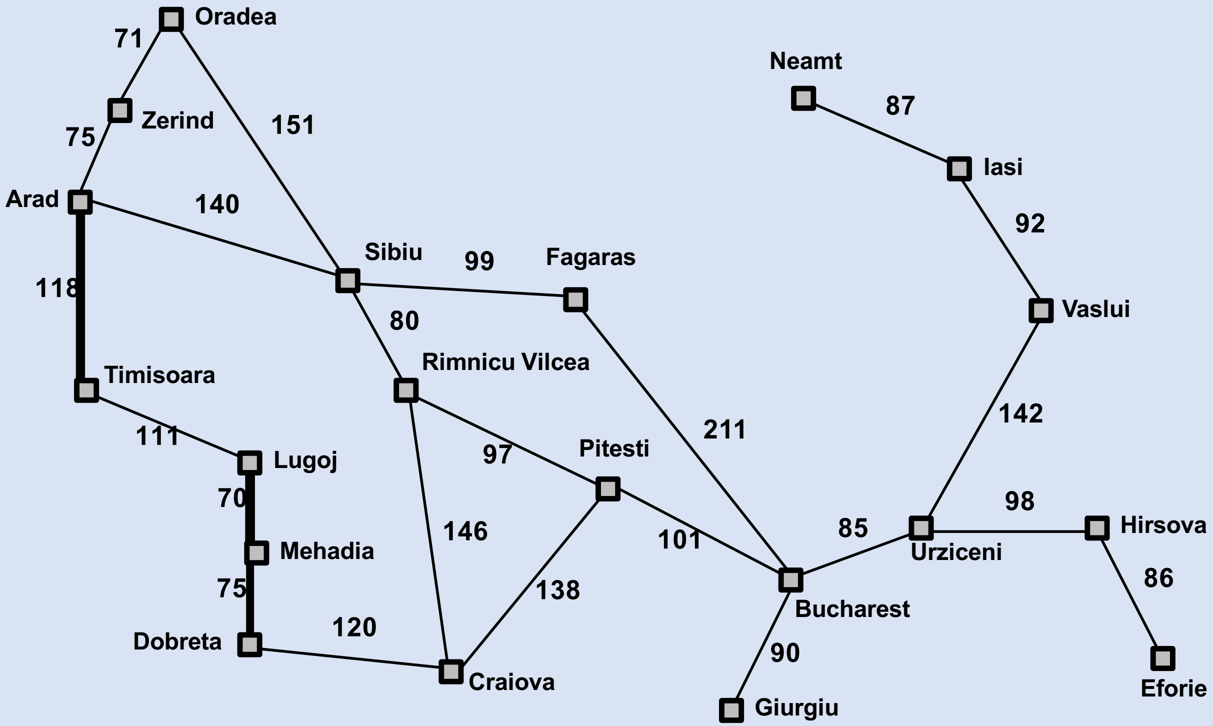


Straight−line distance to Bucharest

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

Greedy search example

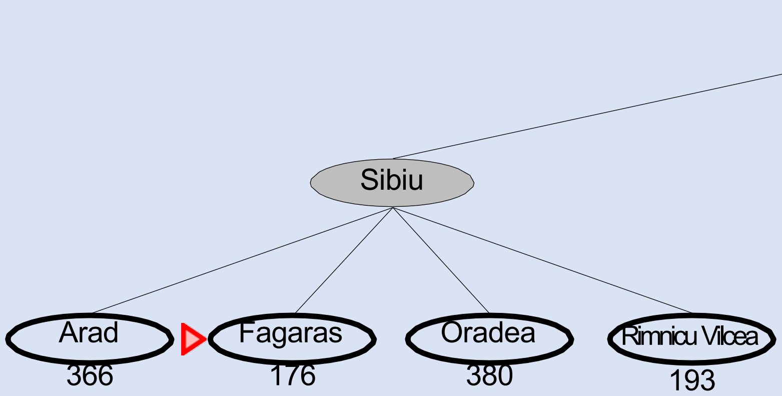


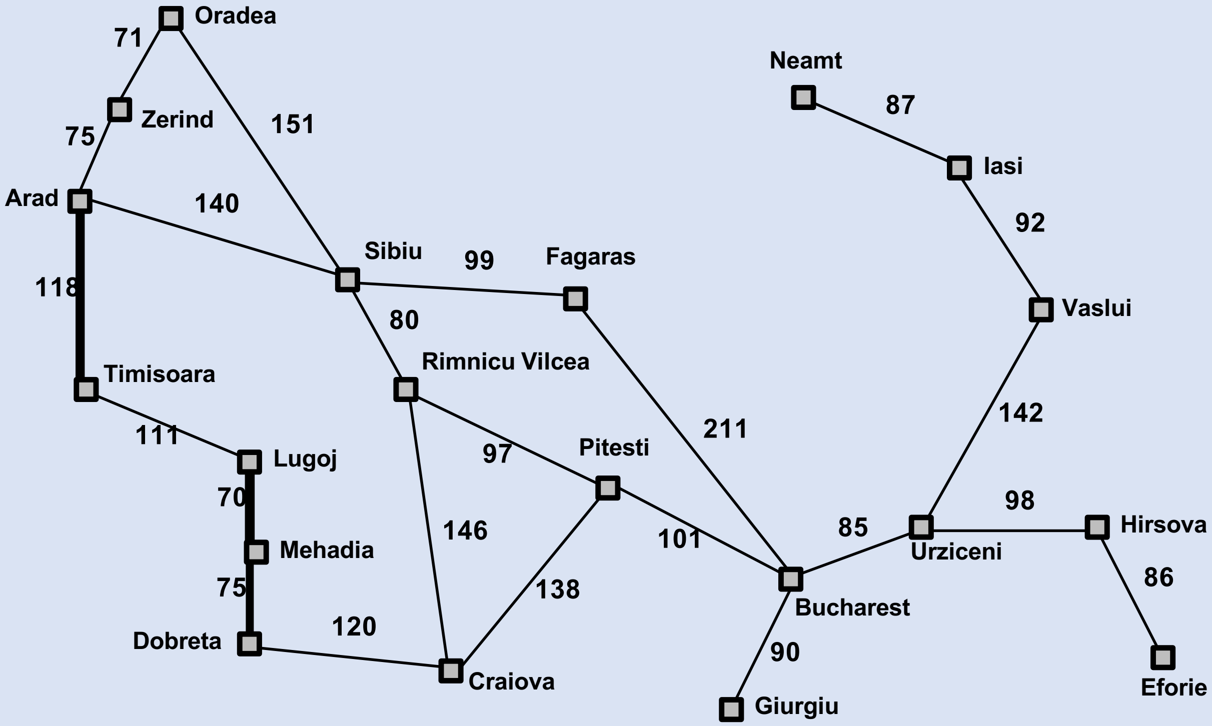


Straight−line distance
to Bucharest

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

Greedy search example

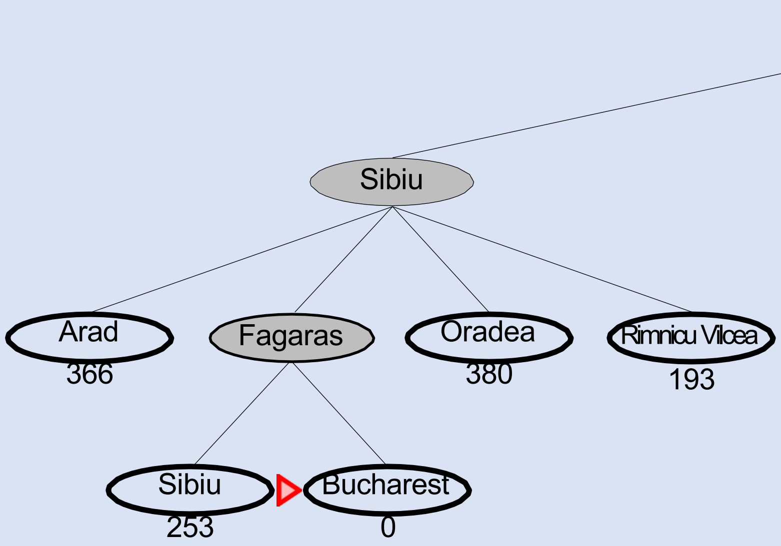


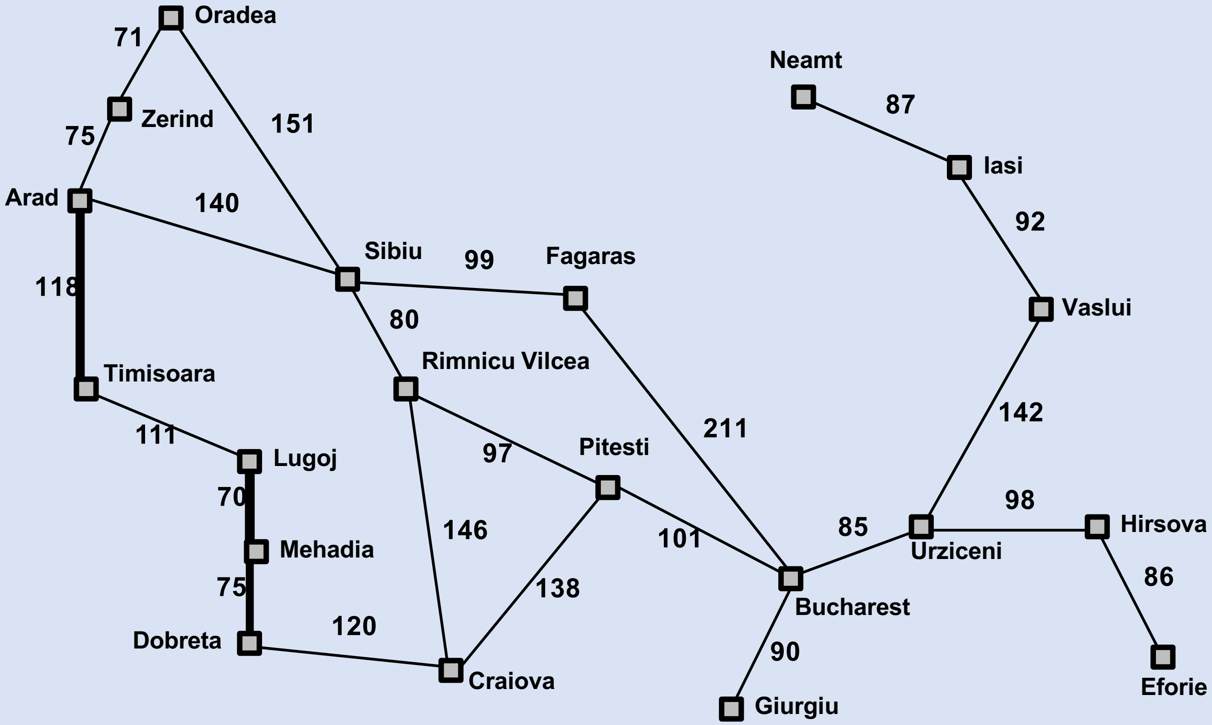


Straight−line distance to Bucharest

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

Greedy search example





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

Properties of greedy search

Complete??

Properties of greedy search

Complete?? No–can get stuck in loops, e.g., with Oradea as goal,

Iasi → Neamt → Iasi → Neamt →

Complete in finite space with repeated-state checking

Time??

Properties of greedy search

Complete?? No–can get stuck in loops, e.g.,

Iasi → Neamt → Iasi → Neamt →

Complete in finite space with repeated-state checking

Time?? $O(b^m)$, but a good heuristic can give dramatic improvement

Space??

Properties of greedy search

Complete?? No–can get stuck in loops, e.g.,

Iasi → Neamt → Iasi → Neamt →

Complete in finite space with repeated-state checking

Time?? $O(b^m)$, but a good heuristic can give dramatic improvement

Space?? $O(b^m)$ - keeps all nodes in memory

Optimal??

Properties of greedy search

Complete?? No–can get stuck in loops, e.g.,

Iasi → Neamt → Iasi → Neamt →

Complete in finite space with repeated-state checking

Time?? $O(b^m)$, but a good heuristic can give dramatic improvement

Space?? $O(b^m)$—keeps all nodes in memory

Optimal?? No

A* search

Idea: avoid expanding paths that are already expensive

Evaluation function $f(n) = g(n) + h(n)$

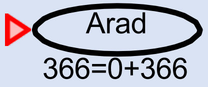
$g(n)$ = cost so far to reach n

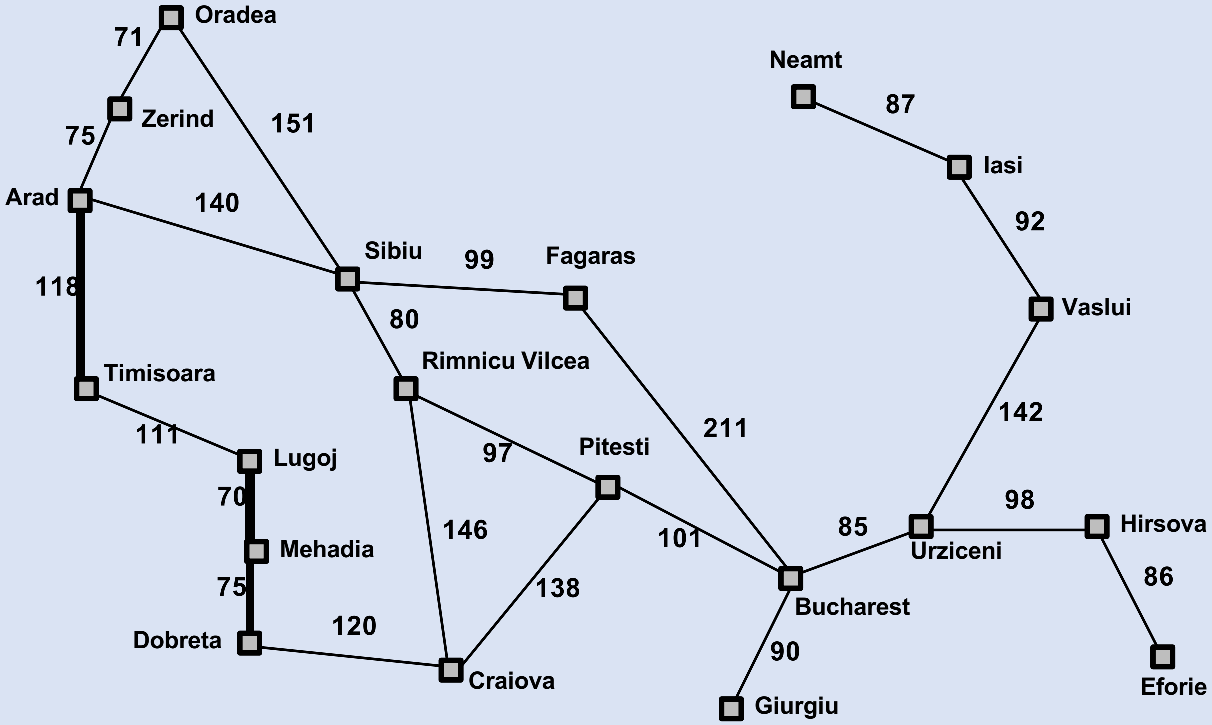
$h(n)$ = estimated cost to goal from n
$f(n)$ = estimated total cost of path through n to goal

A$*$ search uses an admissible heuristic
i.e., $h(n) \leq h^*(n)$ where $h^*(n)$ is the true cost from n.

E.g., $h_{\mathrm{SLD}}(n)$ never overestimates the actual road distance Theorem: A$*$ search is optimal

A* search example

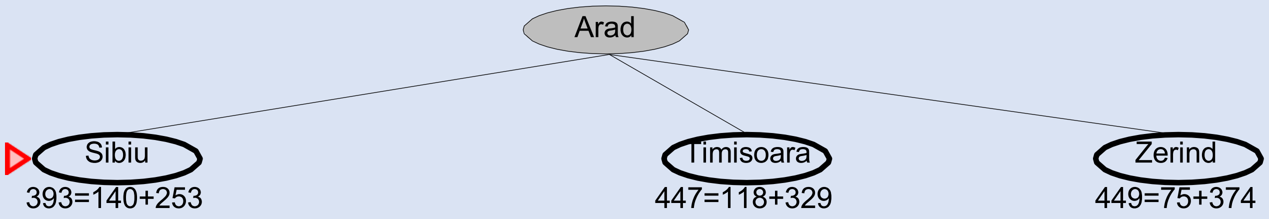


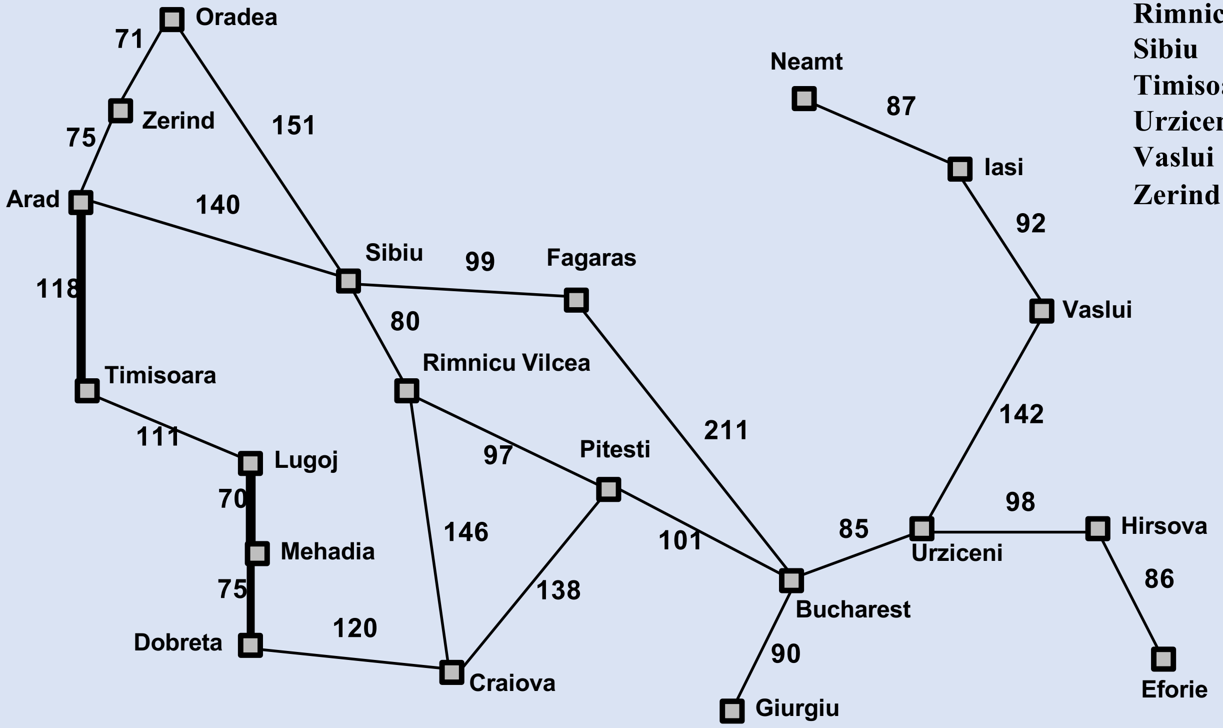


Straight−line distance to Bucharest

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

A* search example

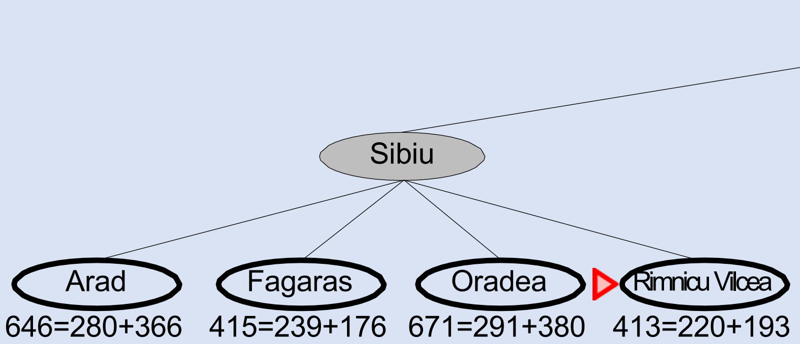


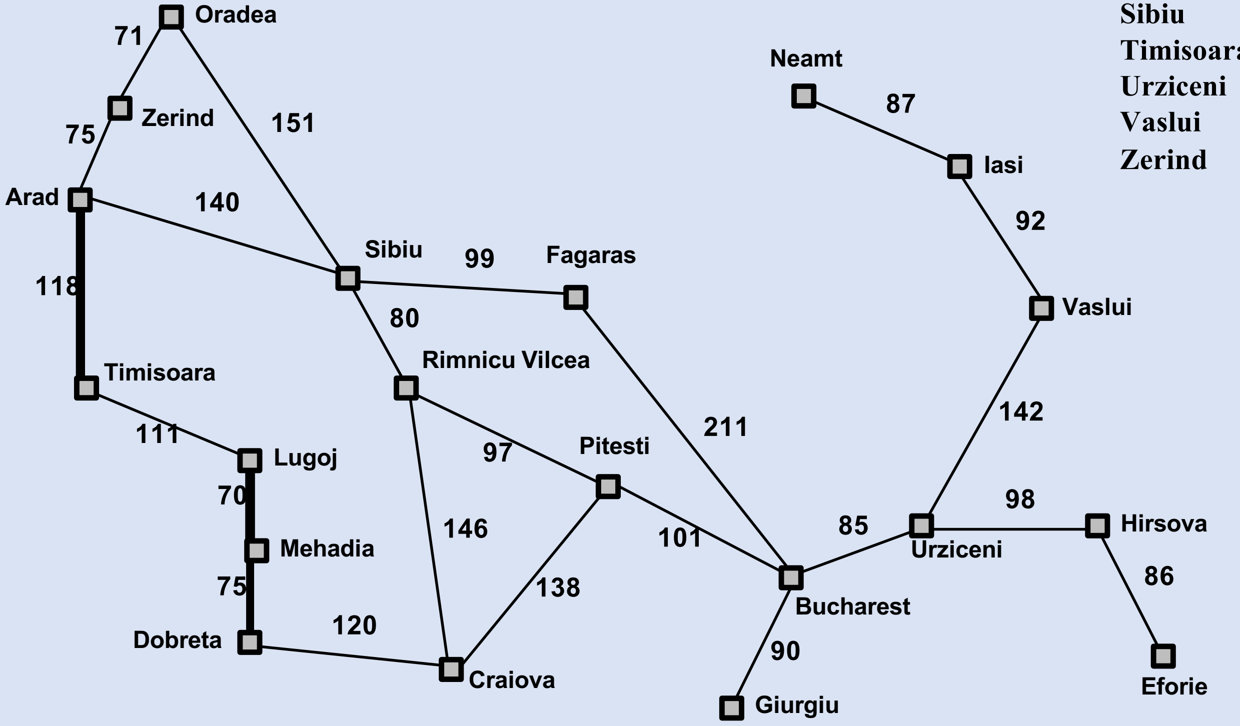


Straight−line distance to Bucharest

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

A* search example

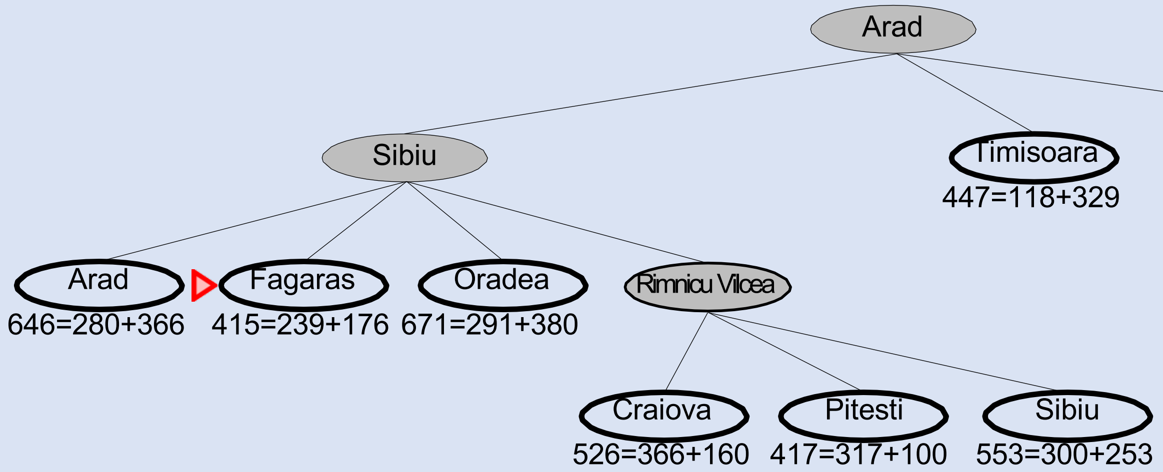




Straight−line distance
to Bucharest

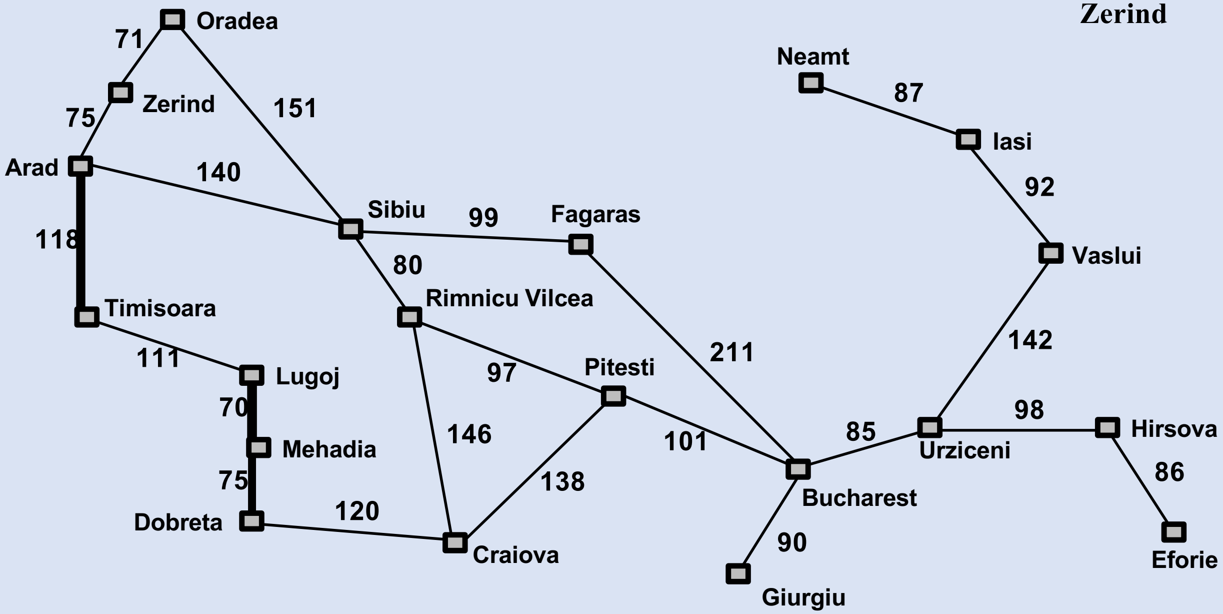
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* search example

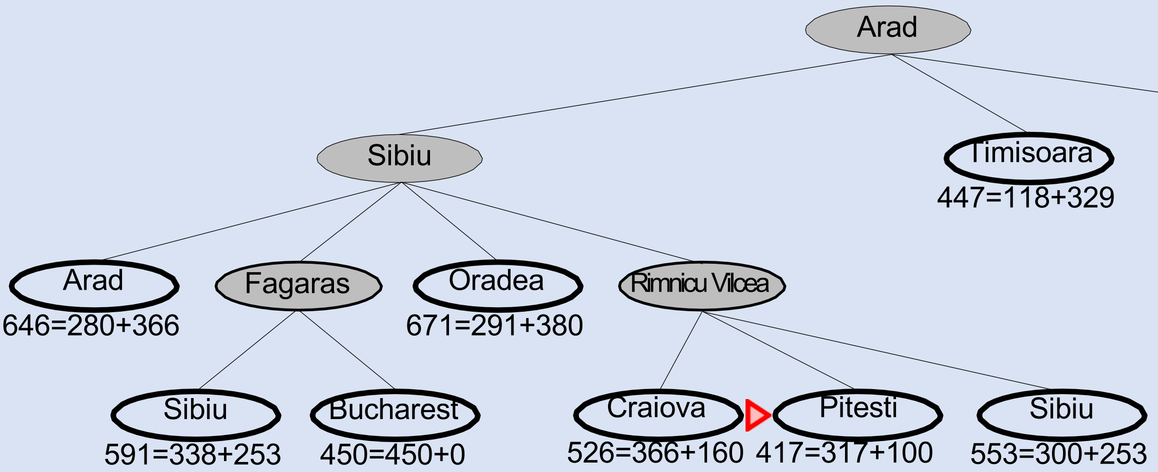


Straight−line distance
to Bucharest

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

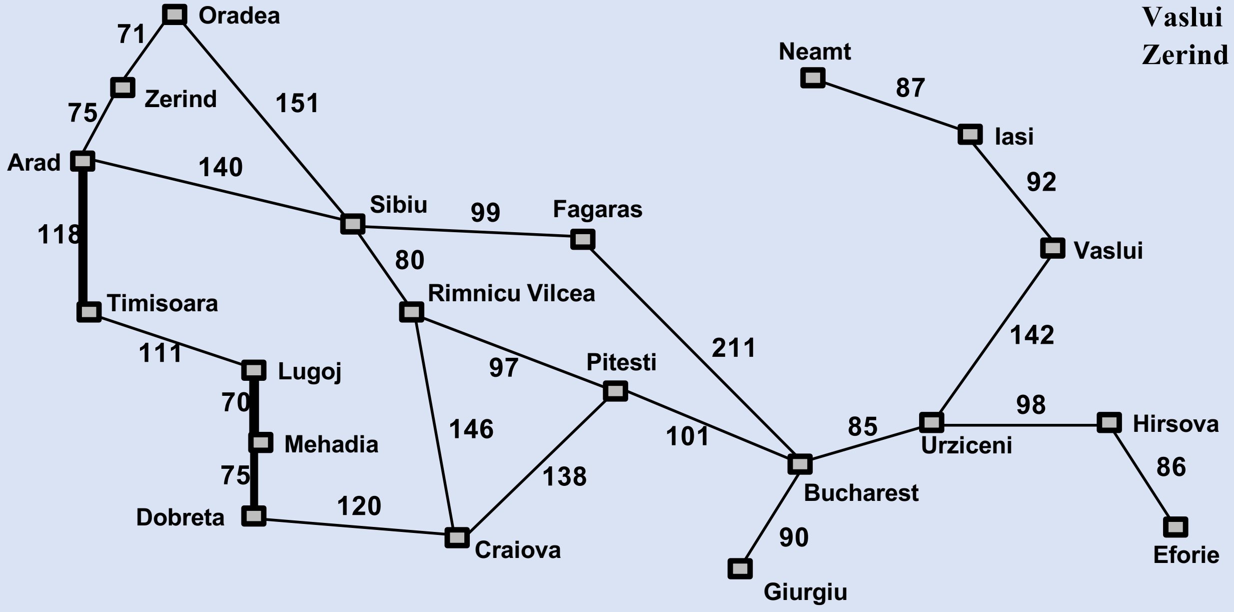


A* search example

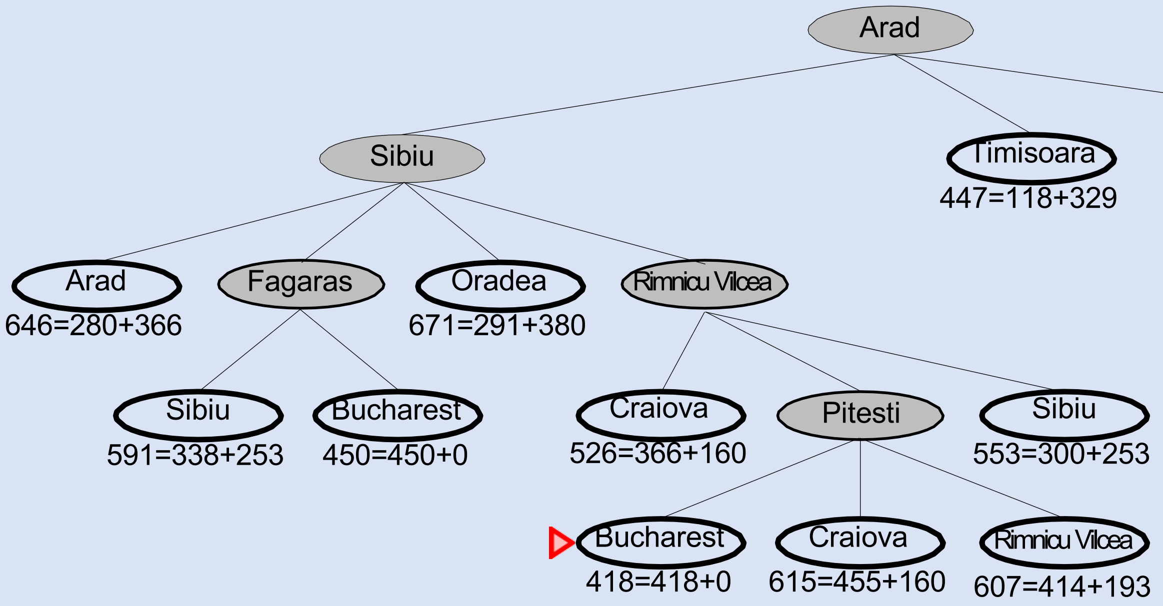


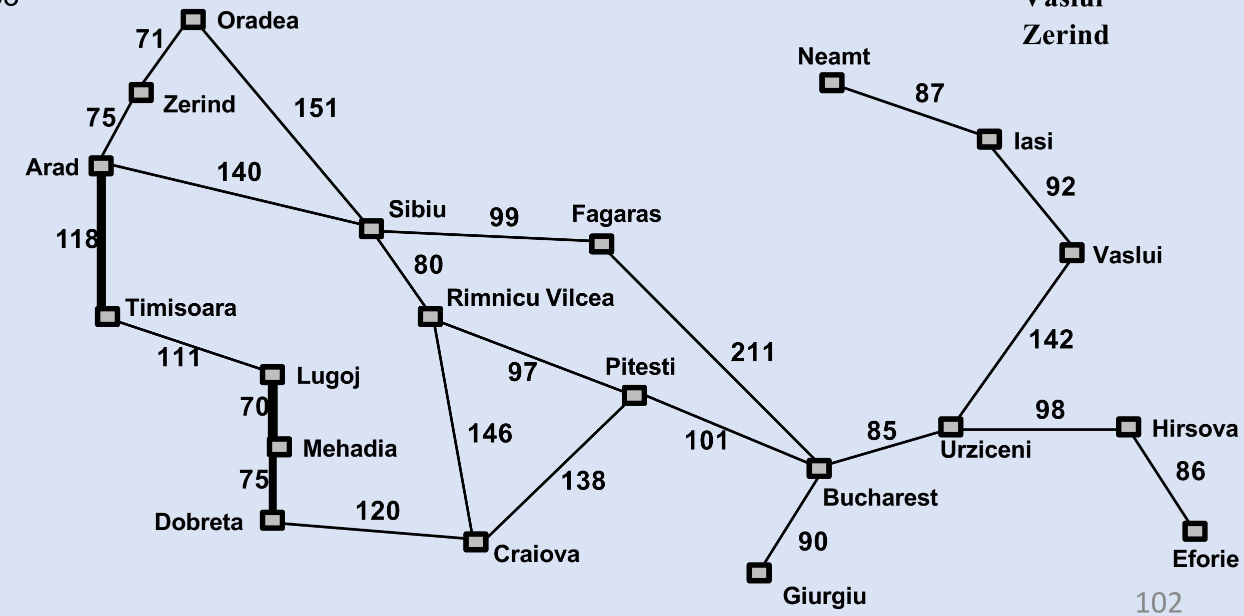
Straight−line distance to Bucharest

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



A* search example





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

Optimality of A∗ (more useful)

- Suppose the optimal path has cost C^*, but the algorithm returns a path with cost $C > C^*$.
- Then there must be some node n which is on the optimal path and is unexpanded
  - If all the nodes on the optimal path had been expanded, then we would have returned that optimal solution.
- Using the notation $g^*(n)$ to mean the cost of the optimal path from the start to n, and $h^*(n)$ to mean the cost of the optimal path from n to the nearest goal, we have:

$$
\begin{aligned}
f(n) &> C^* \quad \text{(otherwise } n \text{ would have been expanded)} \\
f(n) &= g(n) + h(n) \quad \text{(by definition)} \\
f(n) &= g^*(n) + h(n) \quad \text{(because } n \text{ is on an optimal path)} \\
f(n) &\leq g^*(n) + h^*(n) \quad \text{(because of admissibility, } h(n) \leq h^*(n)) \\
f(n) &\leq C^* \quad \text{(by definition, } C^* = g^*(n) + h^*(n))
\end{aligned}
$$

Properties of A∗

Complete??

Properties of A∗

Complete?? Yes

Time??

Properties of A∗

Complete?? Yes

Time?? Exponential $O(b^d)$

Space??

Properties of A∗

Complete?? Yes

Time?? Exponential $O(b^d)$

Space?? Keeps all nodes in memory

Optimal??

Properties of A*

Complete?? Yes

Time?? Exponential $O(b^d)$

Space?? Keeps all nodes in memory

Optimal?? Yes - If the heuristic function *h(n)* is admissible, which in this case means that the shortest path heuristic is always less than or equal to the true shortest path via nodes, then *A** search is also optimal.

Admissible heuristics

E.g., for the 8-puzzle:

$h_1(n)$ = number of misplaced tiles
$h_2(n)$ = total Manhattan distance
(i.e., no. of squares from desired location of each tile)

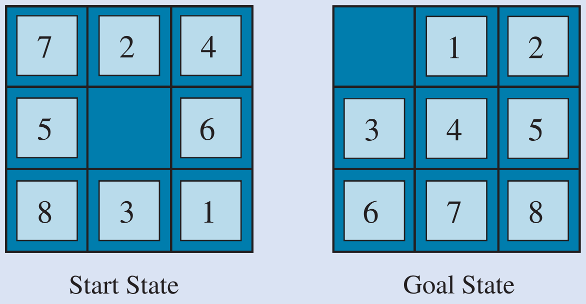


Figure 3.25 A typical instance of the 8-puzzle. The shortest solution is 26 actions long.

$h_1(S)$ =??
$h_2(S)$ =??

Admissible heuristics

E.g., for the 8-puzzle:

$h_1(n)$ = number of misplaced tiles
$h_2(n)$ = total Manhattan distance
(i.e., no. of squares from desired location of each tile)

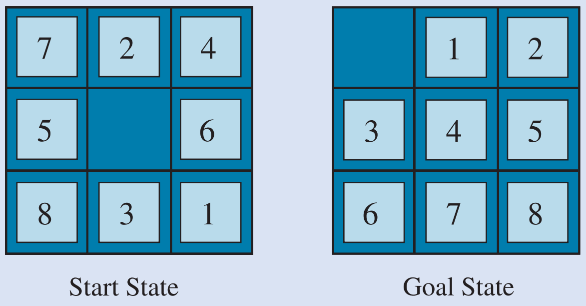


Figure 3.25 A typical instance of the 8-puzzle. The shortest solution is 26 actions long.

$h_1(S)$ =?? 8
$h_2(S)$ =?? 3+1+2+2+2+3+3+2 = 18

As expected, neither of these overestimates the true solution cost, which is 26.

Dominance

If $h_2(n) \geq h_1(n)$ for all n (both admissible), then
h_2 dominates h_1 and is better for search

Typical search costs:

$d = 14$ IDS = 3,473,941 nodes
A*(h_1) = 539 nodes
A*(h_2) = 113 nodes
$d = 24$ IDS $\approx$ 54,000,000,000 nodes
A*(h_1) = 39,135 nodes
A*(h_2) = 1,641 nodes

Given any admissible heuristics h_a, h_b,

$$h(n) = \max(h_a(n), h_b(n))$$

is also admissible and dominates h_a, h_b

Relaxed problems

Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem

If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then $h_1(n)$ gives the shortest solution

If the rules are relaxed so that a tile can move to any adjacent square, then $h_2(n)$ gives the shortest solution

Key point: the optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem

$h_1(n)$ = number of misplaced tiles
$h_2(n)$ = total Manhattan distance

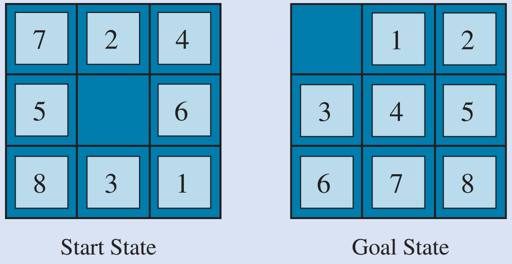


Figure 3.25 A typical instance of the 8-puzzle. The shortest solution is 26 actions long.

$h_1(S)$ =?? 8
$h_2(S)$ =?? 3+1+2+2+2+3+3+2 = 18

The End!