Topic 10 - Graphs
Lecture 10b - Algorithms

CSCI 240

Data Structures and Algorithms

Prof. Dominick Atanasio

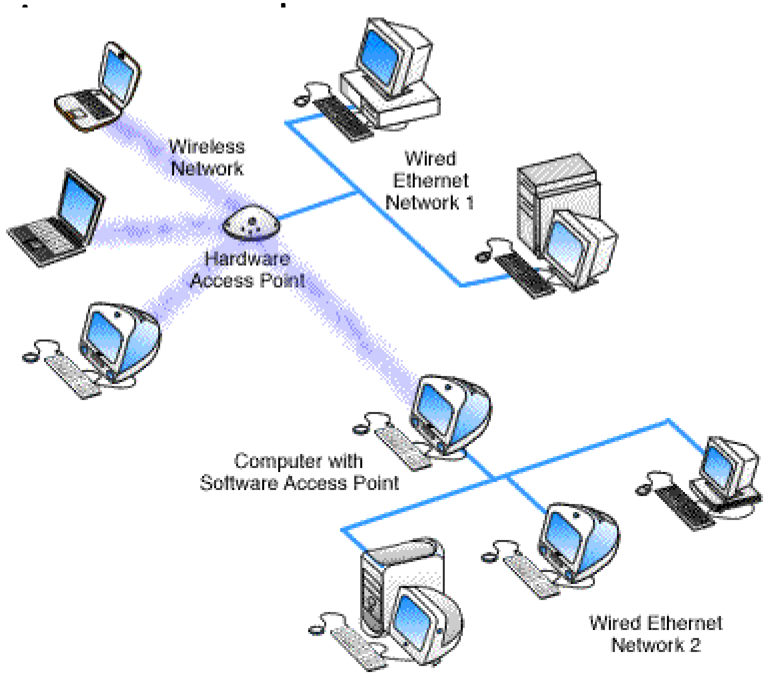
Today

- This Class
  - Single-source shortest path problem
    - Dijkstra's algorithm
    - Uninformed
    - Informed

Finding a Path

- Idea:
  - A problem is often represented by a graph, and the answer to the problem can be found by answering some question about paths in the graph. For example, "Does a path exist?".

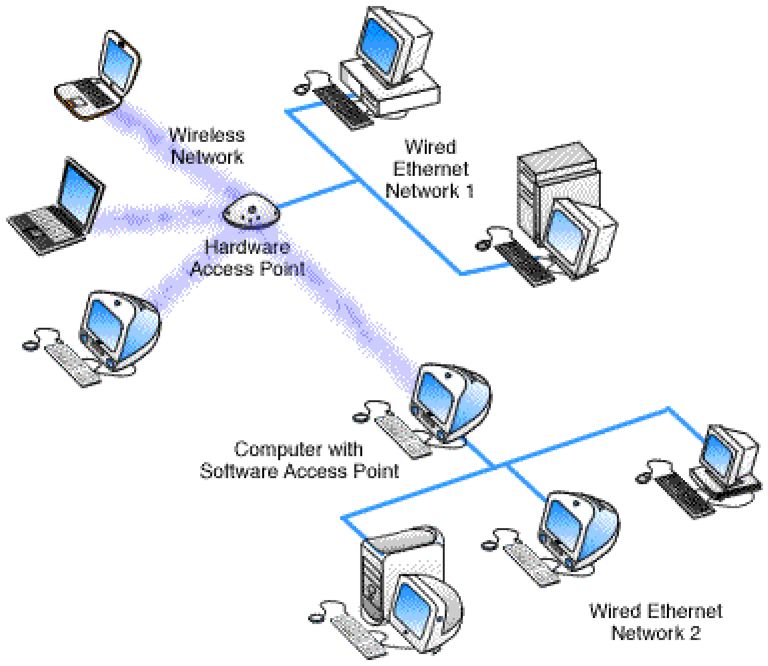
- Problem:
  - We have a network of computers. The question is whether one macl
    another machine.



Finding a Path

- Abstraction:
  - The network of computers can be represented by a graph, with each vertex representing one of the machines in the network and each edge representing a communication wire between two machines.
  - The question now is whether the corresponding vertices are connected by a path.

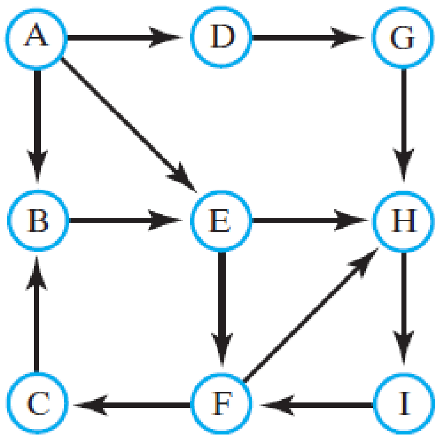
- Solution:
  - Either a BFS or a DFS can be used to determine whether a path exist



The Shortest Path in an Unweighted Graph

- In an unweighted graph, the shortest path between two given vertices has the shortest length—that is, it has the fewest edges.

(a)

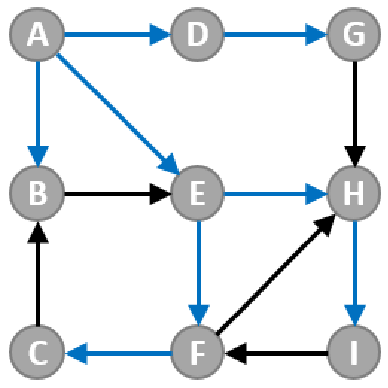


(b)

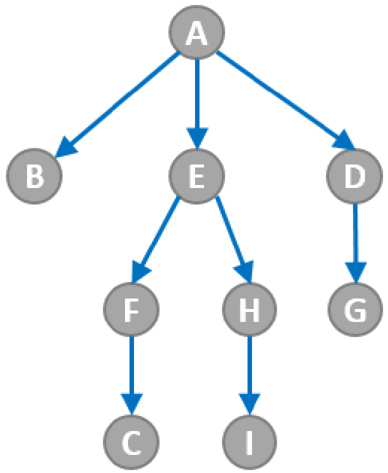
A → B → E → F → H
A → B → E → H
A → D → G → H
A → E → F → H
A → E → H

The Shortest Path in an Unweighted Graph

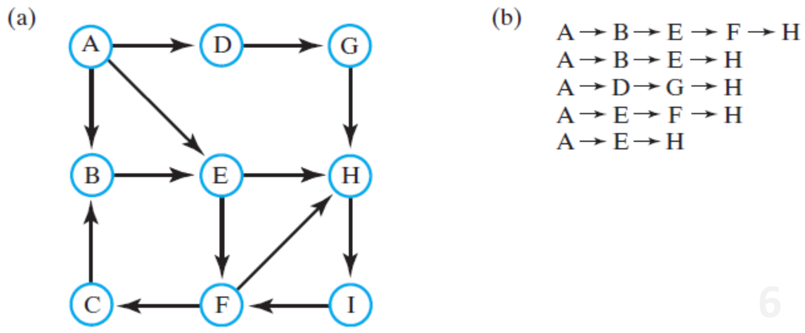
- The algorithm to find this path is based on a breadth-first traversal.



Breadth-First Traversal

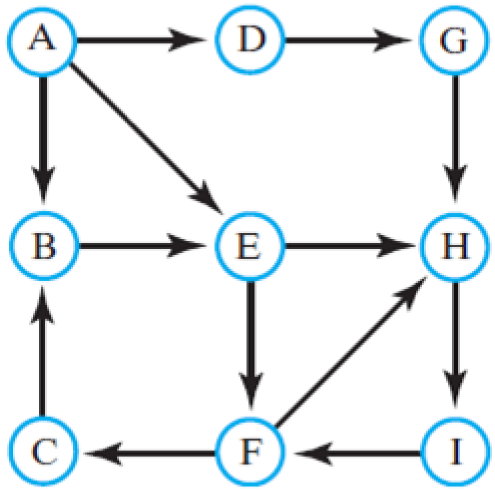


Paths form a breadth-first tree
(Order in which the nodes are expanded)

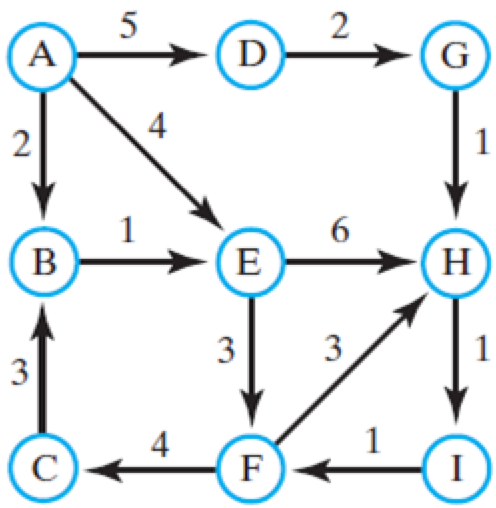


The Shortest Path in a Weighted Graph

- In a weighted graph, each edge has a value attached to it, called the weight or cost of the edge.



An unweighted graph

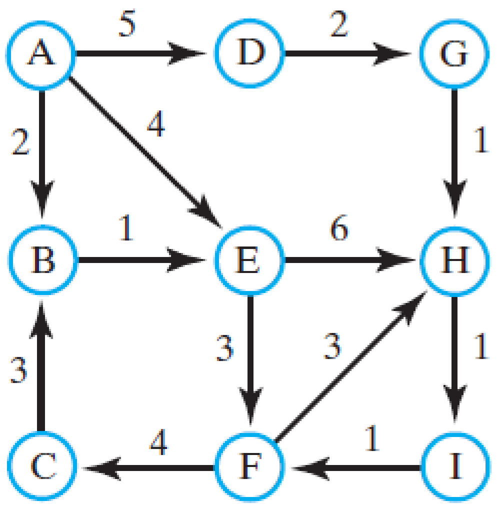


A weighted graph

The Shortest Path in a Weighted Graph

- A weighted edge is an edge together with an integer called the edge's weight.
- The weight of a path is the total sum of the weights of all the edges in the path.
- If two vertices are connected by at least one path, then we can define the shortest path between two vertices, which is the path that has the smallest weight.
  - There may be several paths with equally small weights, in which case each of the paths is called "smallest“.

(a)



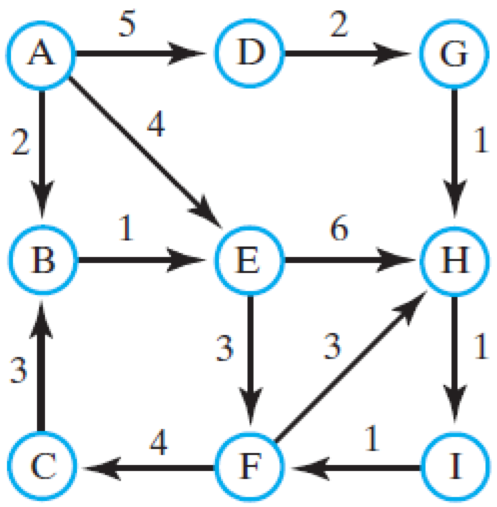
(b)

Path

A → B → E → F → H
A → B → E → H
A → D → G → H
A → E → F → H
A → E → H

- A weighted edge is an edge together with an integer called the edge's weight.
- The weight of a path is the total sum of the weights of all the edges in the path.
- If two vertices are connected by at least one path, then we can define the shortest path between two vertices, which is the path that has the smallest weight.
  - There may be several paths with equally small weights, in which case each of the paths is called "smallest“.

(a)



(b)

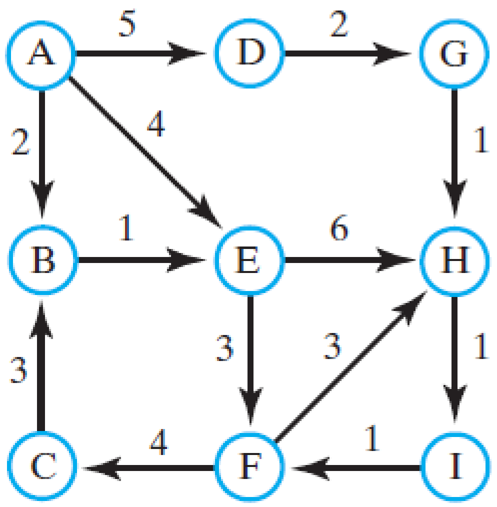
Path	Weight
A → B → E → F → H	
A → B → E → H	
A → D → G → H	
A → E → F → H	
A → E → H	

what is the weight of each path?

The Shortest Path in a Weighted Graph

- A weighted edge is an edge together with an integer called the edge's weight.
- The weight of a path is the total sum of the weights of all the edges in the path.
- If two vertices are connected by at least one path, then we can define the shortest path between two vertices, which is the path that has the smallest weight.
  - There may be several paths with equally small weights, in which case each of the paths is called "smallest“.

(a)



(b)

Path	Weight
A → B → E → F → H	9
A → B → E → H	9
A → D → G → H	8
A → E → F → H	10
A → E → H	10

In a weighted graph, the shortest path is not necessarily the one with the fewest edges.

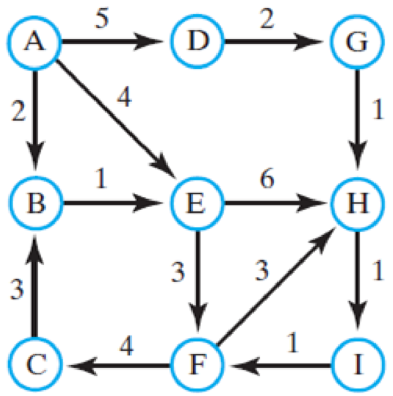
The Shortest Path in a Weighted Graph

- Single-source shortest path problem
  - Bellman-Ford algorithm
    - Allows negative-weight edges
    - Slower than Dijkstra's algorithm
  - Dijkstra's algorithm
    - All edge weights are non-negative

Single-Source Shortest Path

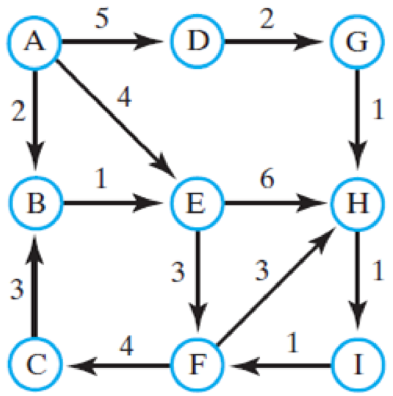
- Given a directed graph $G = (V, E)$ with edge-weight function $w\colon E \rightarrow R$, and a source vertex s.

- From a given source vertex s in V, find the shortest path weights for all vertices in V.
  - Let $\delta(s, v)$ be the shortest path for all v in V.



Find the shortest paths from A to B, C, D, E, F, G, H, and I

- Brute-Force Algorithm:
- Distance(s, t):
- for each path p from s to t:
- compute w(p)
- return p encountered with smallest w(p)



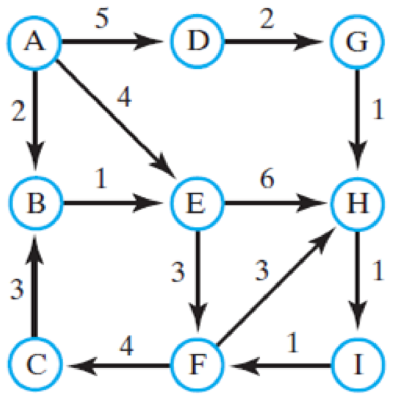
Find the shortest paths from A to B, C, D, E, F, G, H, and I

Single-Source Shortest Path

- Brute-Force Algorithm:
- Distance(s, t):
- for each path p from s to t:
- compute w(p)
- return p encountered with smallest w(p)

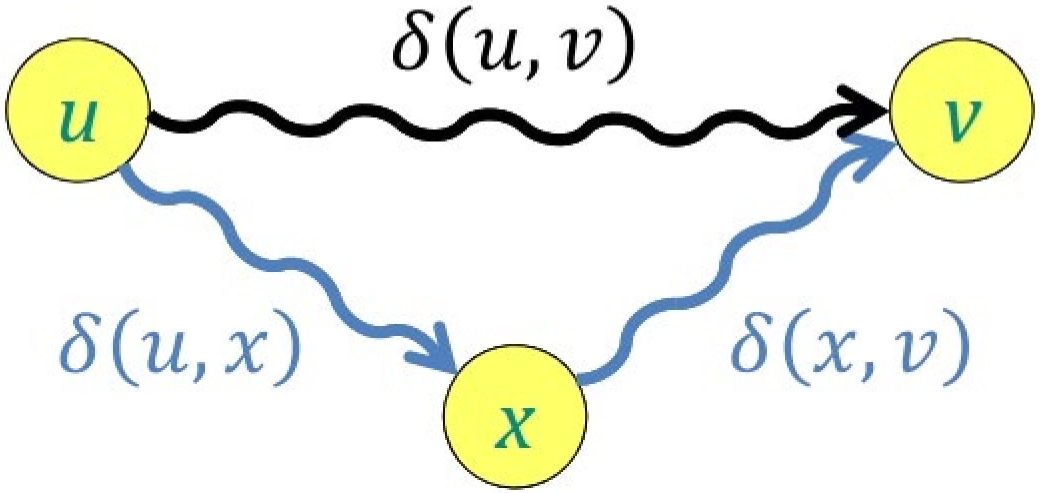
However,

- The number of paths can be infinite when there's negative-weight cycles.
- Assume there's no negative-weight cycles, the number of paths can be exponential.



Find the shortest paths from A to B, C, D, E, F, G, H, and I

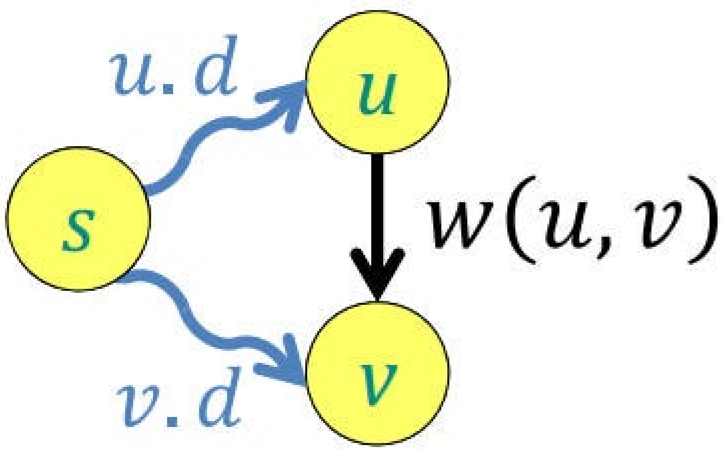
- Some Theorems associated with shortest path
  - A subpath of a shortest path is a shortest path. Proof by contradiction.
  - Triangle inequality. For all u, v, x that belong to V, we have δ(u, v) ≤ δ(u, x) + δ(x, v), where δ(i, j) is the shortest path weight from i to j.



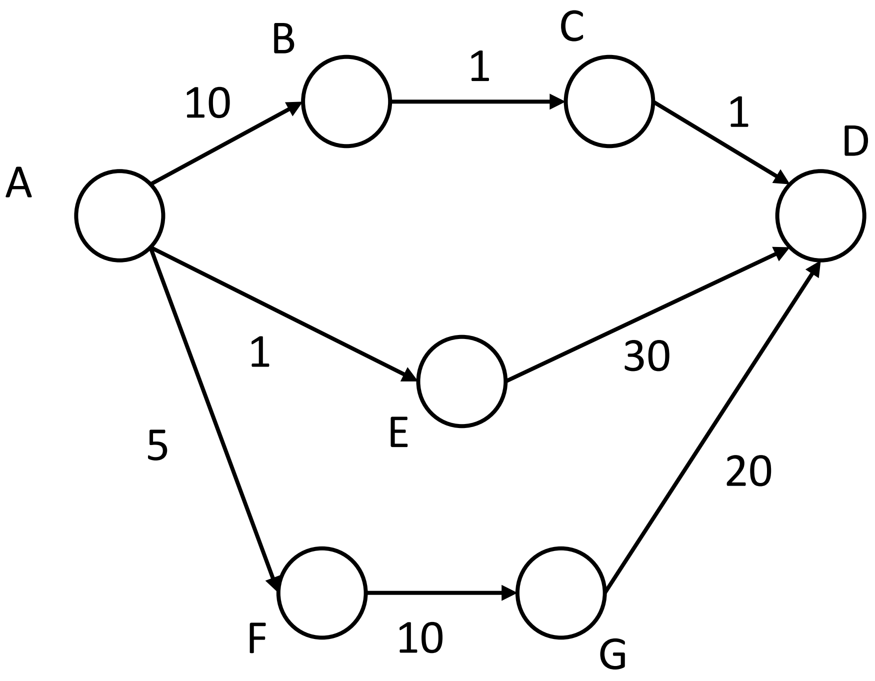
Single-Source Shortest Path

- According to triangle inequality:
- $\delta(s, v) \leq \delta(s, u) + \delta(u, v) \leq \delta(s, u) + w(u, v)$

- Relaxation technique:
  - Initialization: s.d = 0, v.d = ∞ for v ≠ s
  - Goal: v.d = δ(s, v) for all v in V
  - Repeatedly improve estimates toward goal, by aiming to achieve triangle inequality
  - Consider an edge (u, v)
    - relax (u, v): if v.d > u.d + w(u, v), then v.d = u.d + w(u, v)



*Note: **v.d** is called the distance estimate of the path from s to v*



Relaxation technique:

- Initialization: s.d = 0, v.d = ∞ for v ≠ s
- Goal: v.d = δ(s, v) for all v in V
- Repeatedly improve estimates toward goal, by aiming to achieve triangle inequality
- Consider an edge (*u*, *v*)
  - relax (*u*, *v*): if $v.d > u.d + w(u, v)$, then $v.d = u.d + w(u, v)$

*Note: **v.d** is called the distance estimate of the path from s to v*

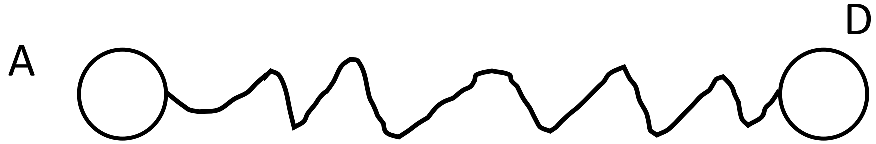
Relaxation Technique

Relaxation technique:

- Initialization: s.d = 0, v.d = ∞ for v ≠ s
- Goal: v.d = δ(s, v) for all v in V
- Repeatedly improve estimates toward goal, by aiming to achieve triangle inequality
- Consider an edge (u, v)
  - relax (u, v): if $v.d > u.d + w(u, v)$, then $v.d = u.d + w(u, v)$

*Note: **v.d** is called the distance estimate of the path from s to v*

Relaxation Technique



Relaxation technique:

- Initialization: s.d = 0, v.d = ∞ for v ≠ s
- Goal: v.d = δ(s, v) for all v in V
- Repeatedly improve estimates toward goal, by aiming to achieve triangle inequality
- Consider an edge (*u*, *v*)
  - relax (*u*, *v*): if *v.d* > *u.d* + *w*(*u*, *v*), then *v.d* = *u.d* + *w*(*u*, *v*)

*Note: **v.d** is called the distance estimate of the path from s to v*

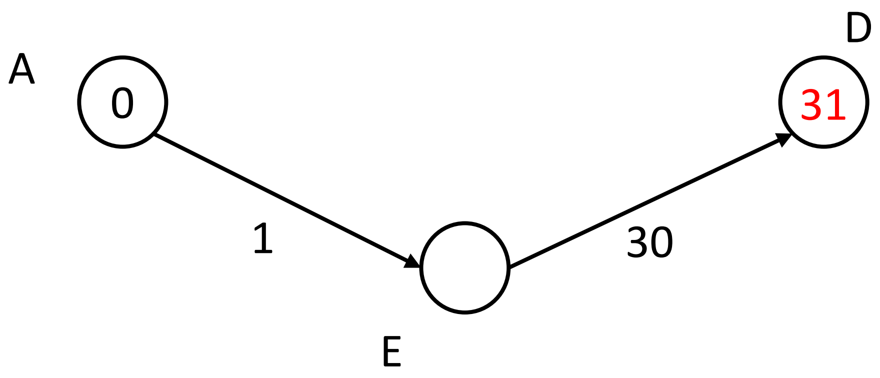
Relaxation Technique

Relaxation technique:

- Initialization: s.d = 0, v.d = ∞ for v ≠ s
- Goal: v.d = δ(s, v) for all v in V
- Repeatedly improve estimates toward goal, by aiming to achieve triangle inequality
- Consider an edge (u, v)
  - relax (u, v): if $v.d > u.d + w(u, v)$, then $v.d = u.d + w(u, v)$

*Note: **v.d** is called the distance estimate of the path from s to v*

Relaxation Technique



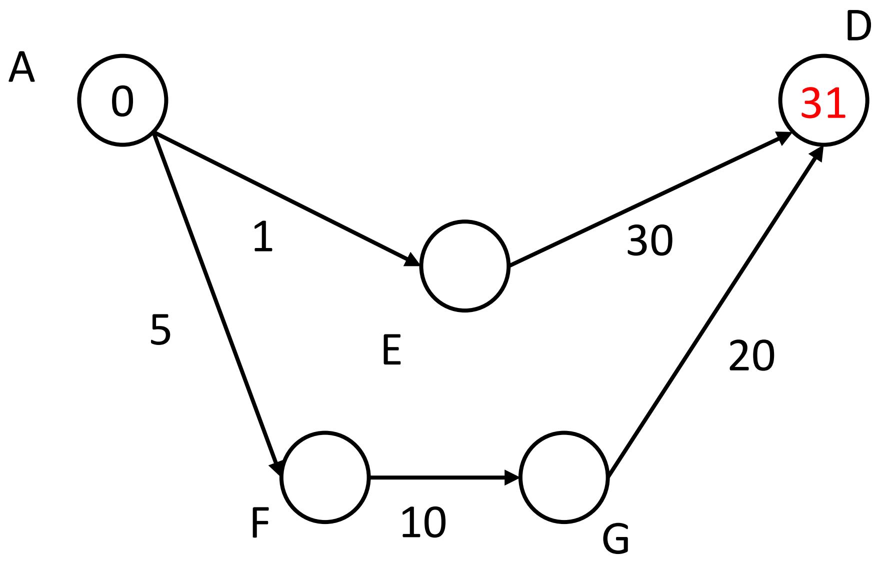
31 < ∞

Relaxation technique:

- Initialization: s.d = 0, v.d = ∞ for v ≠ s
- Goal: v.d = δ(s, v) for all v in V
- Repeatedly improve estimates toward goal, by aiming to achieve triangle inequality
- Consider an edge (*u*, *v*)
  - relax (*u*, *v*): if *v.d* > *u.d* + *w*(*u*, *v*), then *v.d* = *u.d* + *w*(*u*, *v*)

Note: ***v.d*** *is called the distance estimate of the path from s to v*

Relaxation Technique



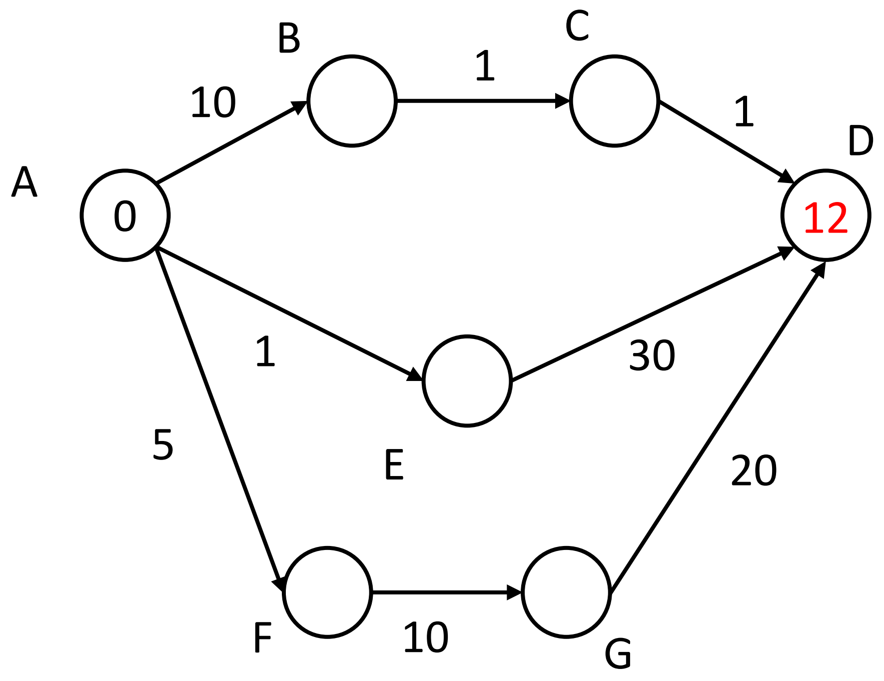
31 < 35 < ∞

Relaxation technique:

- Initialization: s.d = 0, v.d = ∞ for v ≠ s
- Goal: v.d = δ(s, v) for all v in V
- Repeatedly improve estimates toward goal, by aiming to achieve triangle inequality
- Consider an edge (*u*, *v*)
  - relax (*u*, *v*): if *v.d* > *u.d* + *w*(*u*, *v*), then *v.d* = *u.d* + *w*(*u*, *v*)

*Note: **v.d** is called the distance estimate of the path from s to v*

Relaxation Technique



$12 < 31 < 35 < \infty$

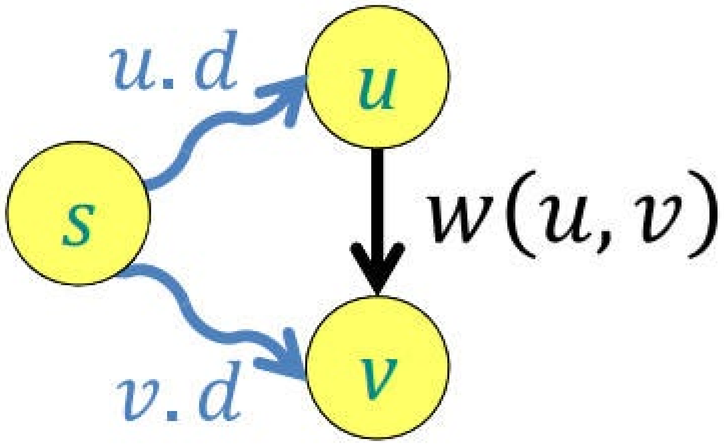
Relaxation technique:

- Initialization: s.d = 0, v.d = ∞ for v ≠ s
- Goal: v.d = δ(s, v) for all v in V
- Repeatedly improve estimates toward goal, by aiming to achieve triangle inequality
- Consider an edge (*u*, *v*)
  - relax (*u*, *v*): if *v.d* > *u.d* + *w*(*u*, *v*), then *v.d* = *u.d* + *w*(*u*, *v*)

*Note: **v.d** is called the distance estimate of the path from s to v*

Relaxation technique:

```
for v in V:
    v.d = infinity
    s.d = 0
while some edege (u, v) has v.d > u.d + w(u,v):
    pick such an edge (u, v)
    relax(u, v):
            if v.d > u.d + w(u,v):
                    v.d = u.d + w(u,v)
```



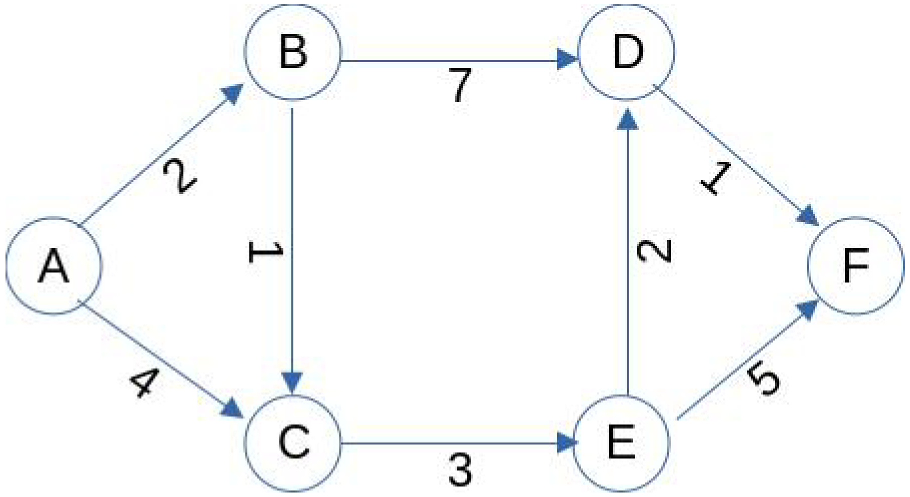
Note: ***v.d*** *is called the distance estimate of the path from s to v*

Dijkstra's Algorithm

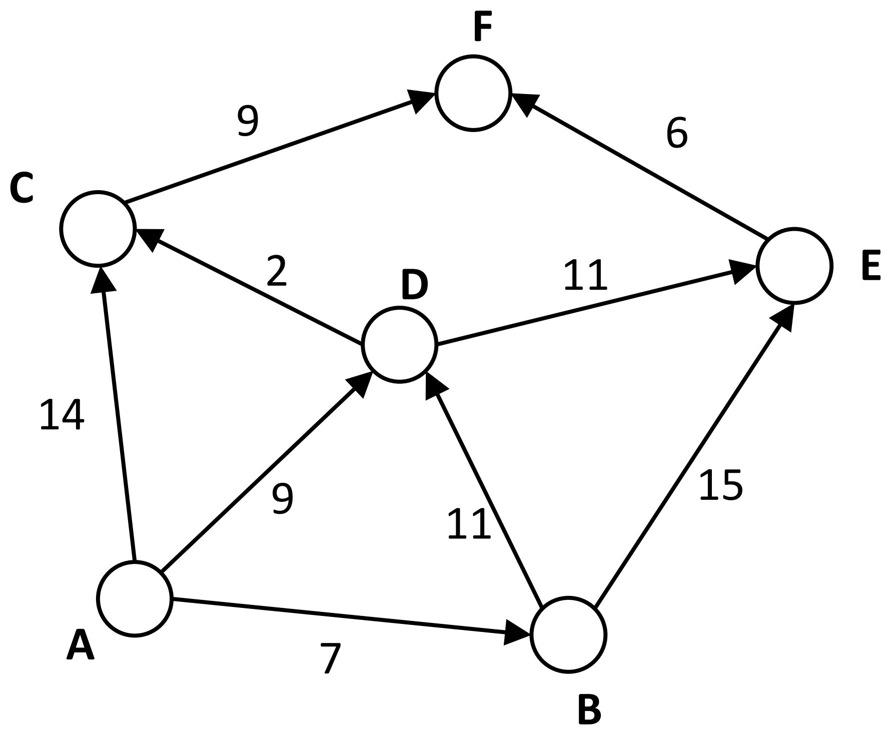
- Dijkstra's algorithm,
  - Conceived by Dutch computer scientist Edsger Dijkstra in 1959,
  - Is a graph search algorithm that solves the single-source shortest path problem for a graph with non-negative edge path costs, producing a shortest path tree.
  - This algorithm is often used in network routing protocols.

- Idea: Greedy
  - Maintain a set S of vertices whose shortest path distance from s are known.
  - At each step add to S the vertex v in (V - S) whose distance estimate from s is minimal.
  - Update the distance estimates of vertices adjacent to v.

- Demonstrated in class



In-Class Exercise



Implementation of Dijkstra's Algorithm

- function Dijkstra(Graph, source):

```
function Dijkstra(Graph, source):
    for each vertex v in Graph:                // Initializations
        dist[v] := infinity                    // Unknown distance function from source to v
        previous[v] := undefined               // Previous node in optimal path from source
    dist[source] := 0                          // Distance from source to source
    Q := the set of all nodes in Graph         // All nodes in the graph are unoptimized thus are in Q (priority queue)
    while Q is not empty:                      // The main loop
        u := vertex in Q with smallest dist[]
        if dist[u] = infinity:
            break                              // all remaining vertices are inaccessible
        remove u from Q
        for each neighbor v of u:              // where v has not yet been removed from Q
            alt := dist[u] + dist_between(u, v)
            if alt < dist[v]
                dist[v] := alt
                previous[v] := u
    return previous[]
```

Time Complexity

```
function Dijkstra(Graph, source):
    for each vertex v in Graph:                 // Initializations
        dist[v] := infinity                     // Unknown distance function from source to v
        previous[v] := undefined                // Previous node in optimal path from source
    dist[source] := 0                           // Distance from source to source
    Q := the set of all nodes in Graph          // All nodes in the graph are unoptimized - thus are in Q (priority queue)
    while Q is not empty:                       // The main loop
        u := vertex in Q with smallest dist[]
        if dist[u] = infinity:
            break                               // all remaining vertices are inaccessible
        remove u from Q
        for each neighbor v of u:               // where v has not yet been removed from Q
            alt := dist[u] + dist_between(u, v)
            if alt < dist[v]
                dist[v] := alt
                previous[v] := u
    return previous[]
```

$|V|$ times

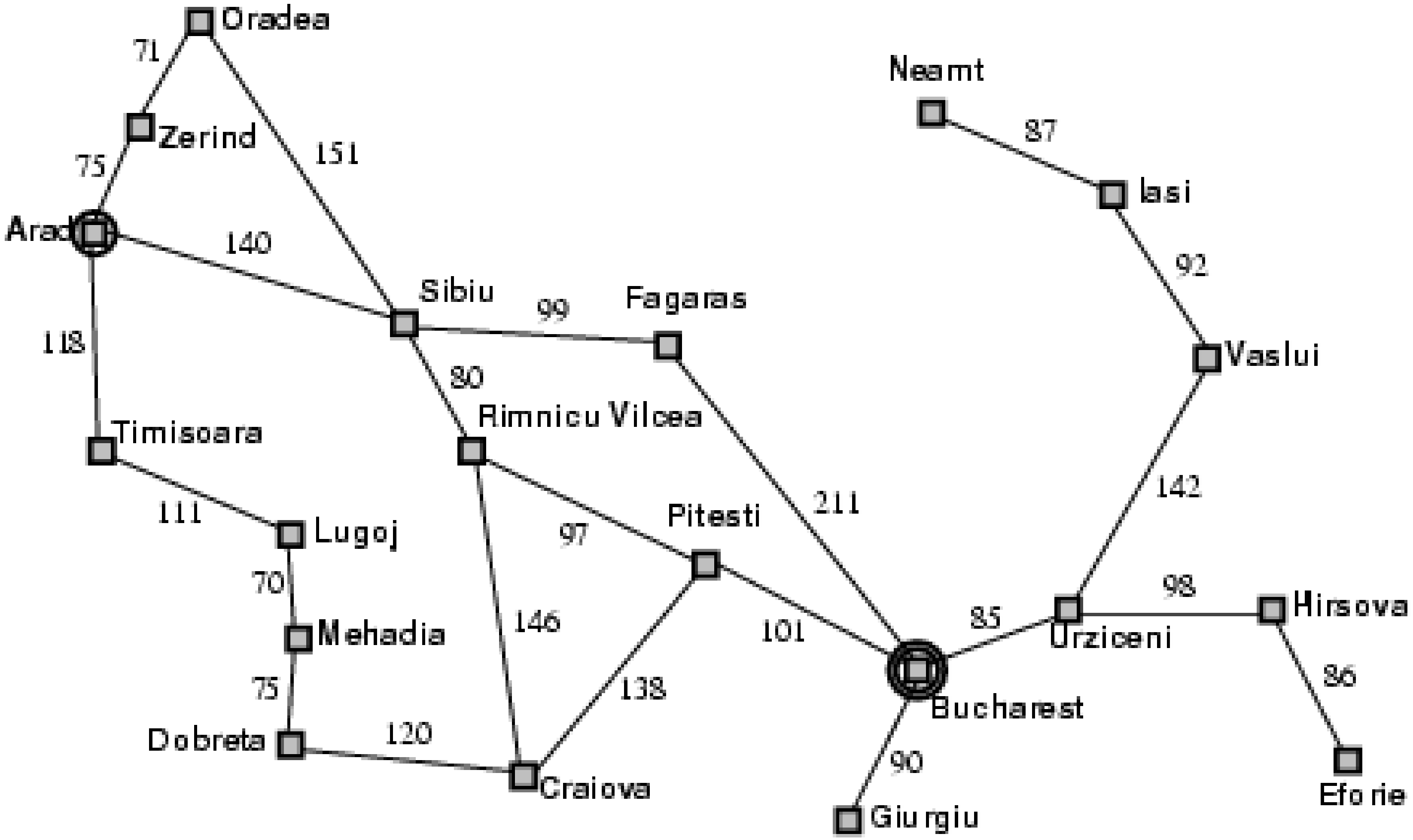
degree(u) times

$$\text{Time complexity} = O(|V| \cdot T_{Extract\_min} + |E| \cdot T_{Decrease\_dist})$$

Uninformed Search Strategies

- Uninformed search (blind search) strategies use only the information available in the problem definition
- Strategies that know whether one non-goal state is expected to be better than another are called informed search or heuristic search
- General uninformed search strategies:
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search

Road Map of Romania



Frontier and Explored Set

- Goal of frontier: Contain the set of discovered node(s) to expand (explore)
  - Possible data structures?
- Goal of explored set: Contains Explored states to prevent for repeated exploration
- Possible data structures?

Uniform-Cost Search

- Expand least-cost unexpanded node
- Implementation:
  - Frontier = priority queue ordered by path cost g(n)
- Example, shown on board, from Sibiu to Bucharest

Uniform-Cost Search

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
     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 ← CHILD-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
```

Uniform-Cost Search is Optimal

- Uniform-cost search expands nodes in order of their optimal path cost
- Hence, the first goal node selected for expansion must be the optimal solution

Informed Search

- Definition:
  - Use problem-specific knowledge beyond the definition of the problem itself
  - Can find solutions more efficiently
- Best-first search strategies
  - Greedy best-first search
  - A*

Best-First Search

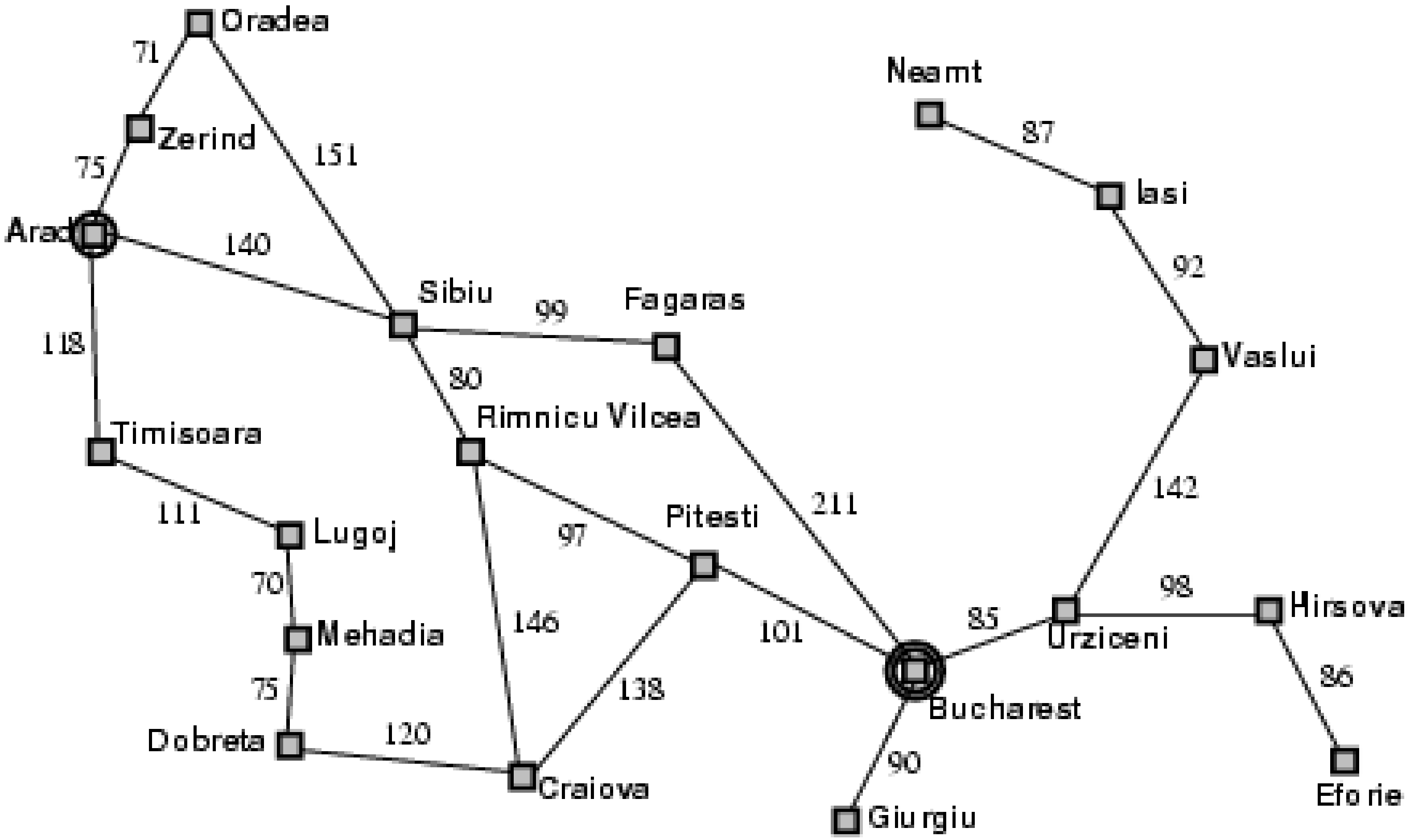
- Idea: use an evaluation function f(n) for each node
  - estimate of "desirability"
  - Expand most desirable unexpanded node
- Implementation: use a data structure that maintains the frontier in a decreasing order of desirability
- Is it really the best?
- Special cases: uniform-cost (Dijkstra’s algorithm), greedy search, A* search
- A key component is a heuristic function h(n):
  - h(n) = estimated cost of the cheapest path from node n to a goal node
  - h(n) = 0 if n is the goal
  - h(n) could be general or problem-specific

37

Best First Search Pseudo-code

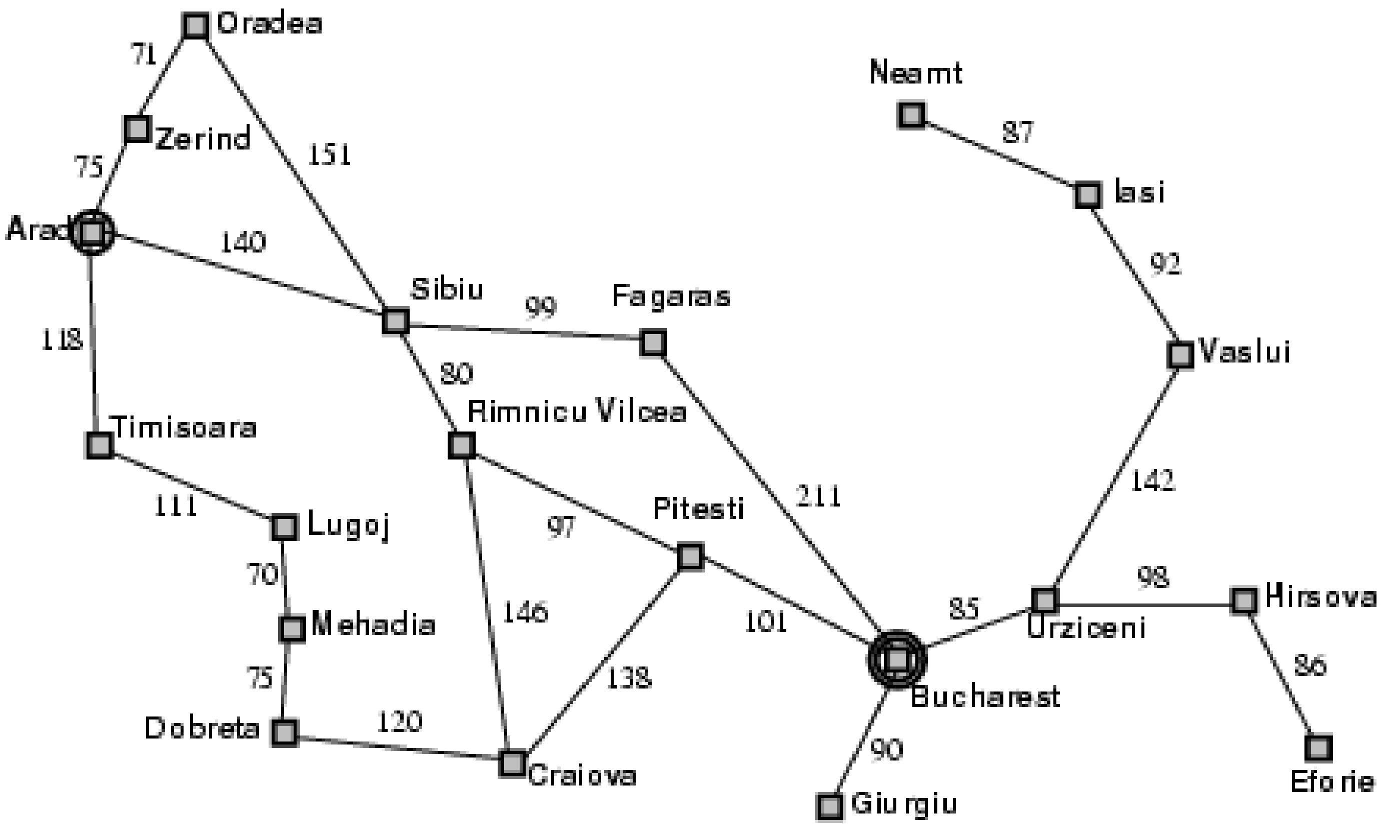
```
Best_First_Search(problem)
{
        initialize the queue, Q, with the initial state (node)
        while Q is not empty, do
                assign the first element of Q to N
                if N is the goal, return SUCCESS
                remove N from Q
                add the children of N to Q
                sort the entire Q by f (n)
        end-while
        return FAILURE
}
```

Recall Romania Map Example



What’s a proper heuristic that measures cheapest path from current node to goal node?

Romania Map with Costs



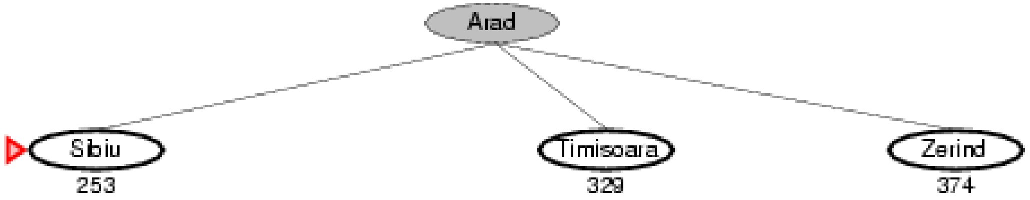
Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy Best-First Search

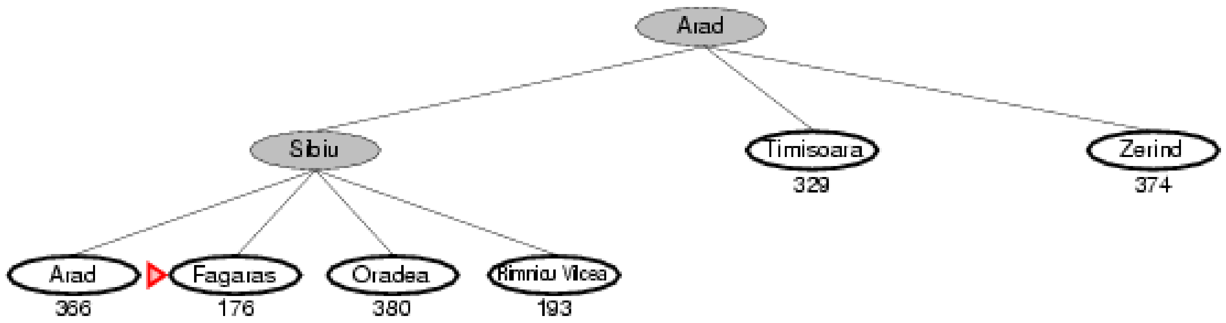
- Evaluation function: f(n) = h(n)
  - estimate the cost from n to goal
- hSLD = straight line distance from n to Bucharest

Example: Arad to Bucharest

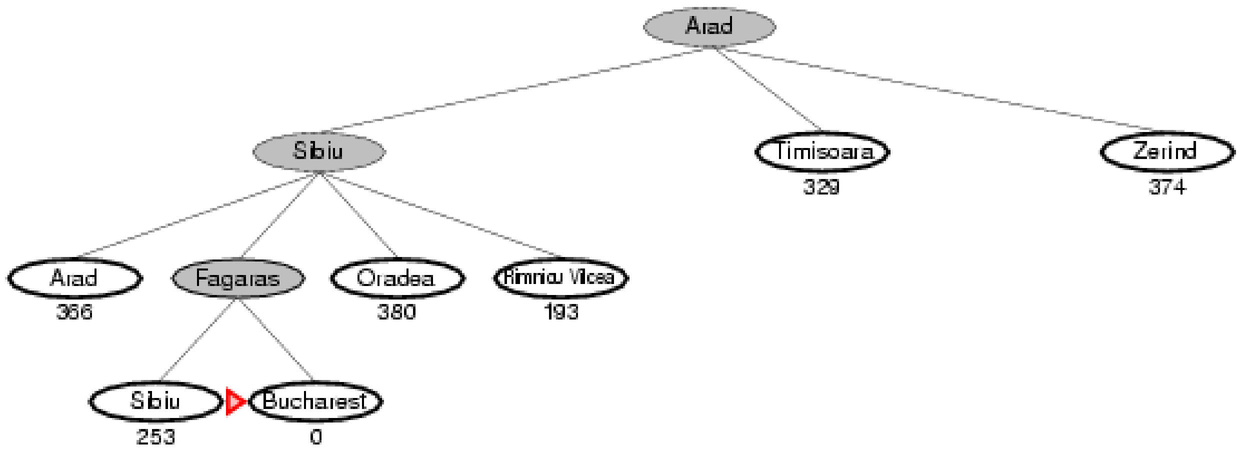
Example: Arad to Bucharest



Example: Arad to Bucharest



Example: Arad to Bucharest



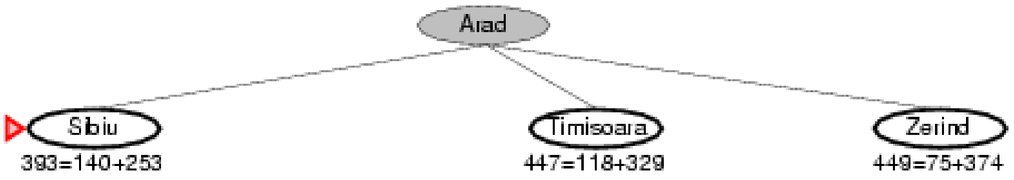
A*: Minimizing Total Est. Cost

- Idea: avoid expanding paths that are already expensive

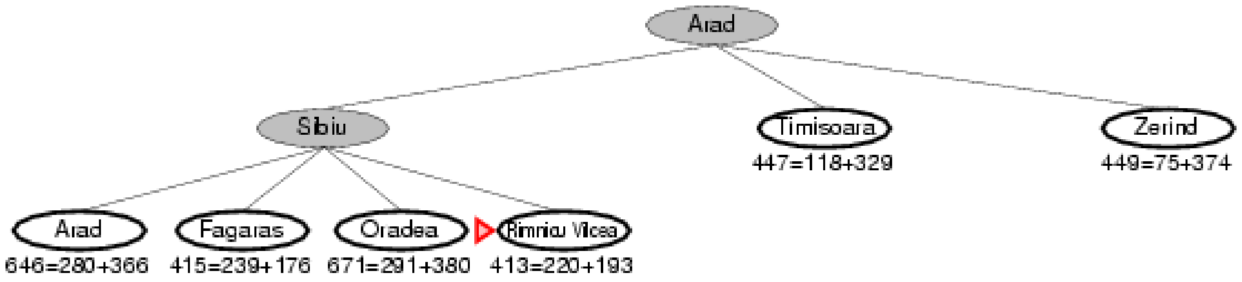
- Evaluation function $f(n) = g(n) + h(n)$
  - $g(n)$ = path cost so far to reach n
  - $h(n)$ = estimated cost from n to goal
  - $f(n)$ = estimated total cost of path from n to goal

A* Search Example

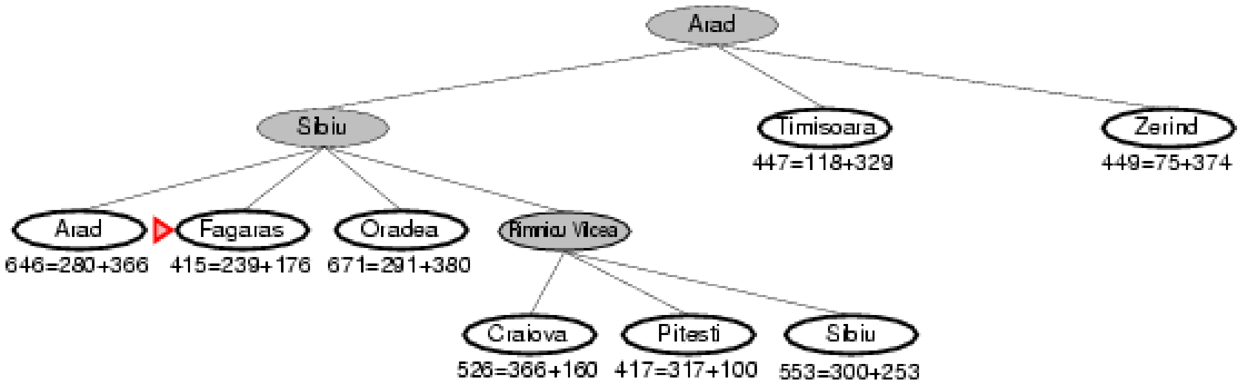
A* Search Example



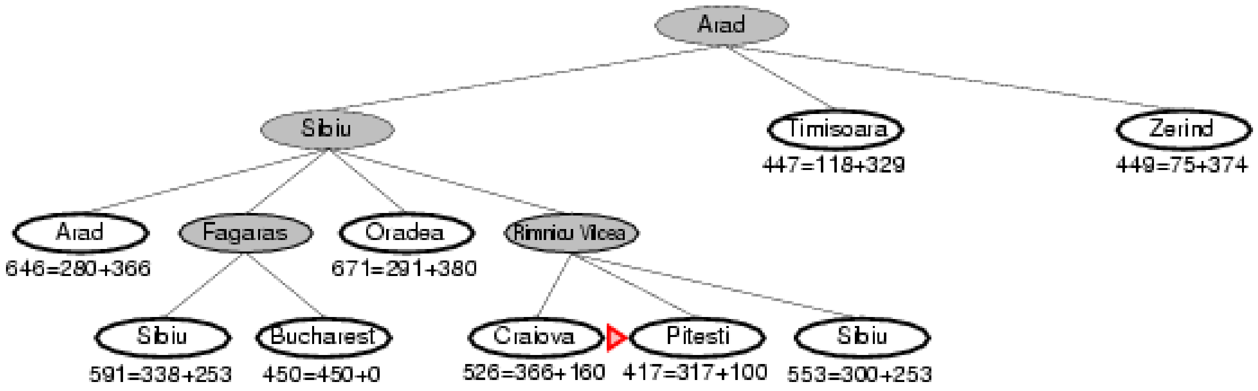
A* Search Example



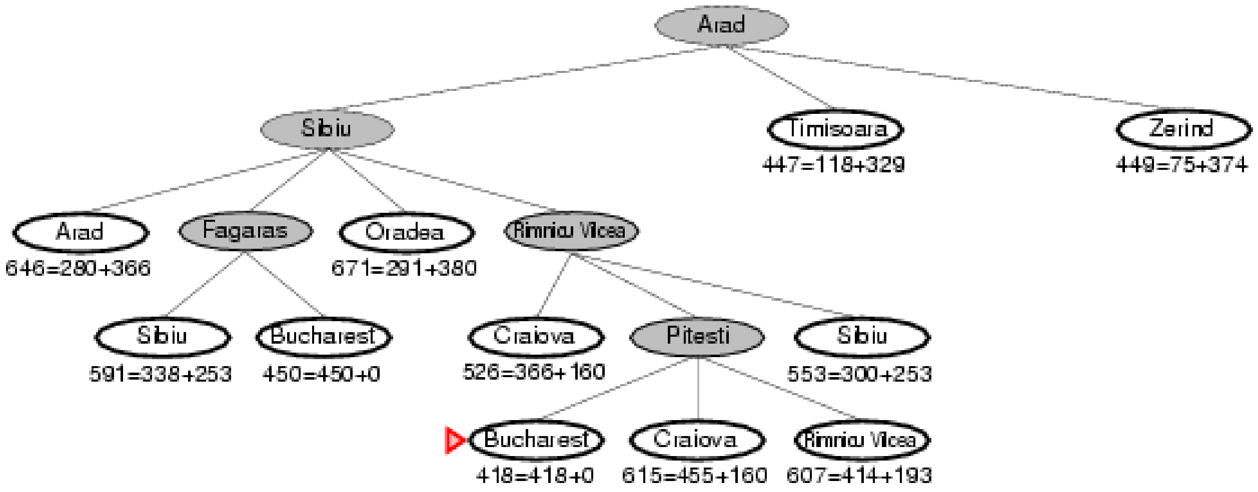
A* Search Example



A* Search Example

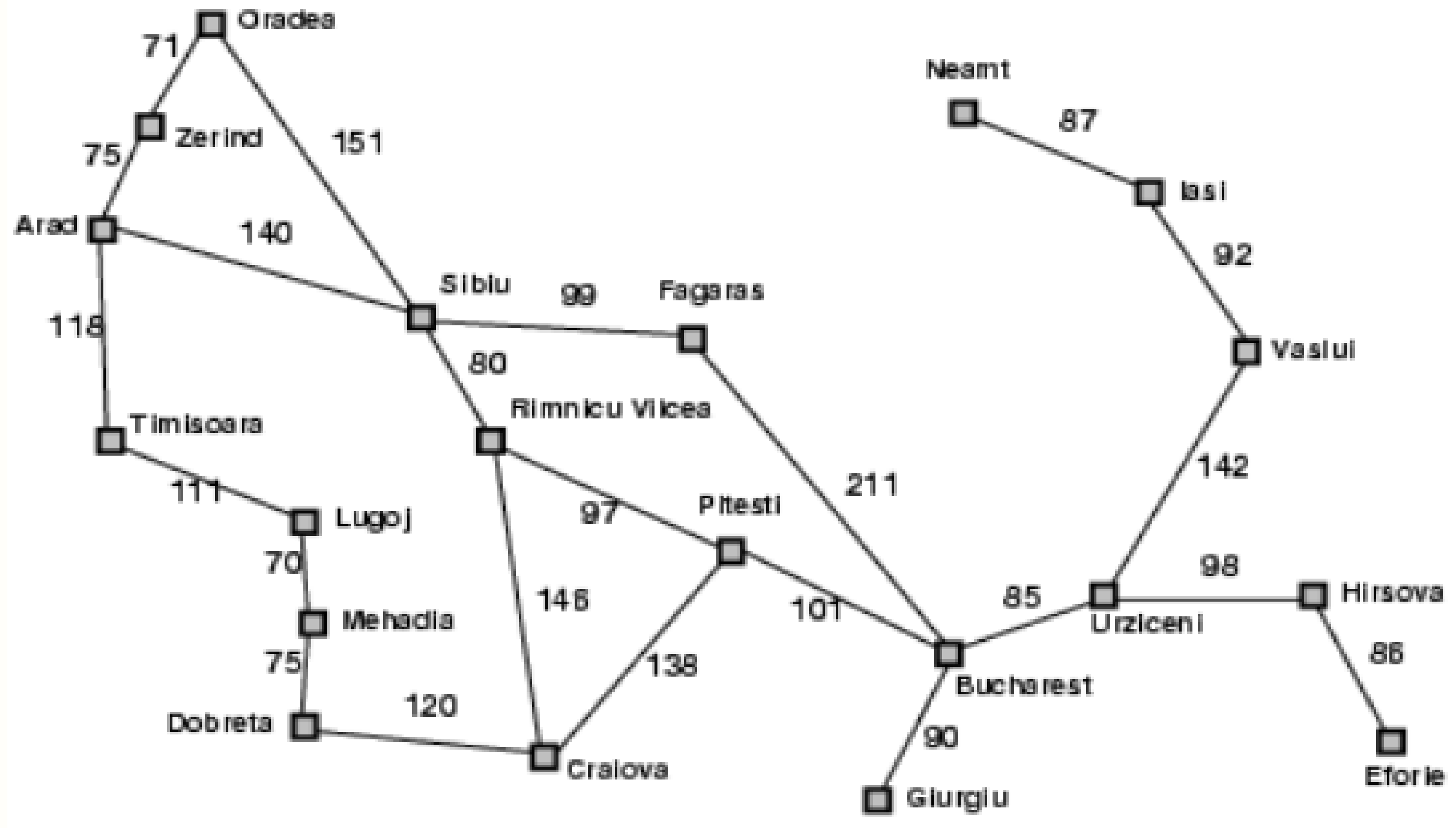


A* Search Example



Exercise

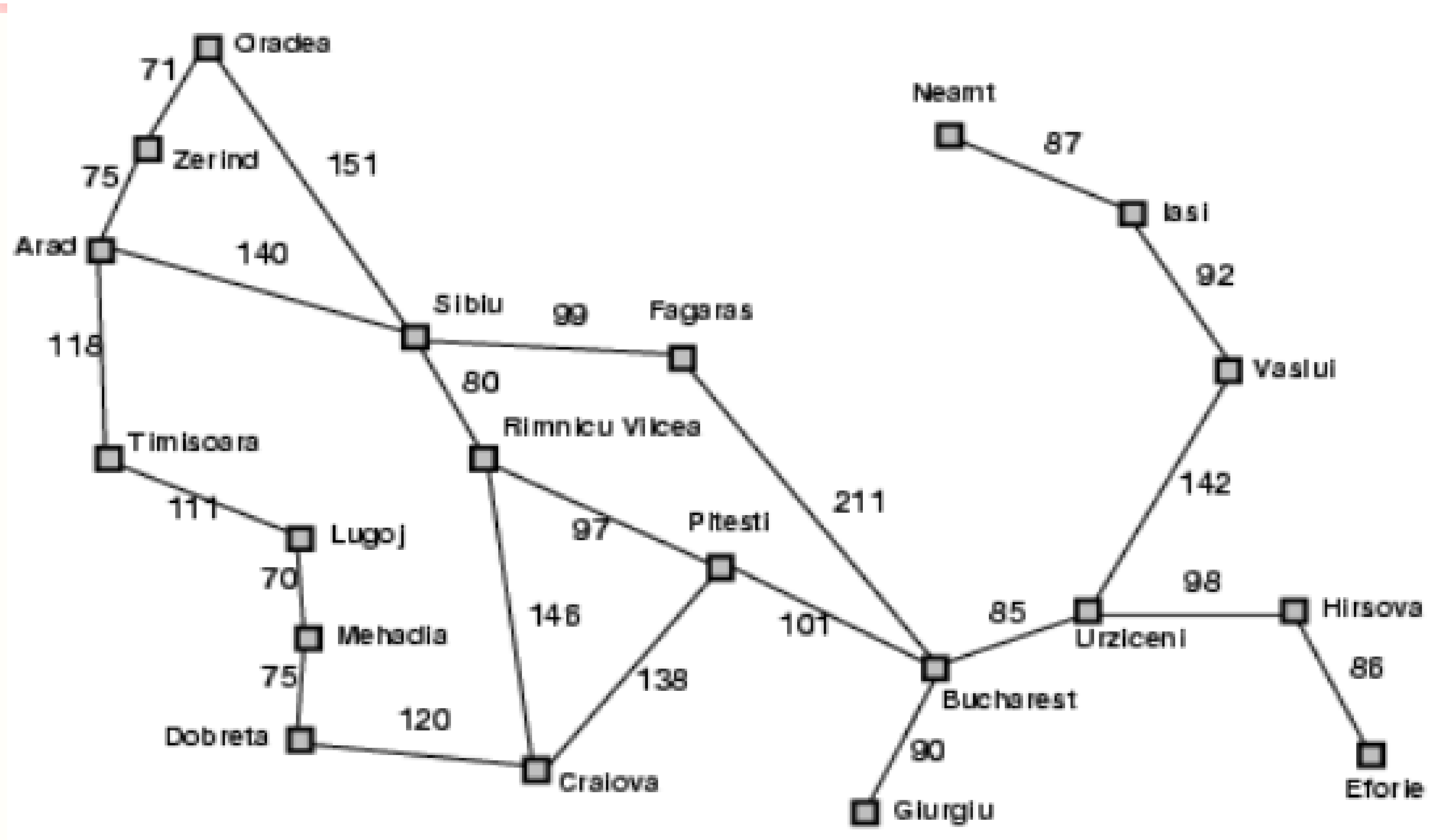
- Use A* graph-search to generate a path from Lugoj to Bucharest using the straight-line distance heuristic.



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Exercise

- Use A* graph-search to generate a path from Lugoj to Bucharest using the straight-line distance heuristic.



Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374