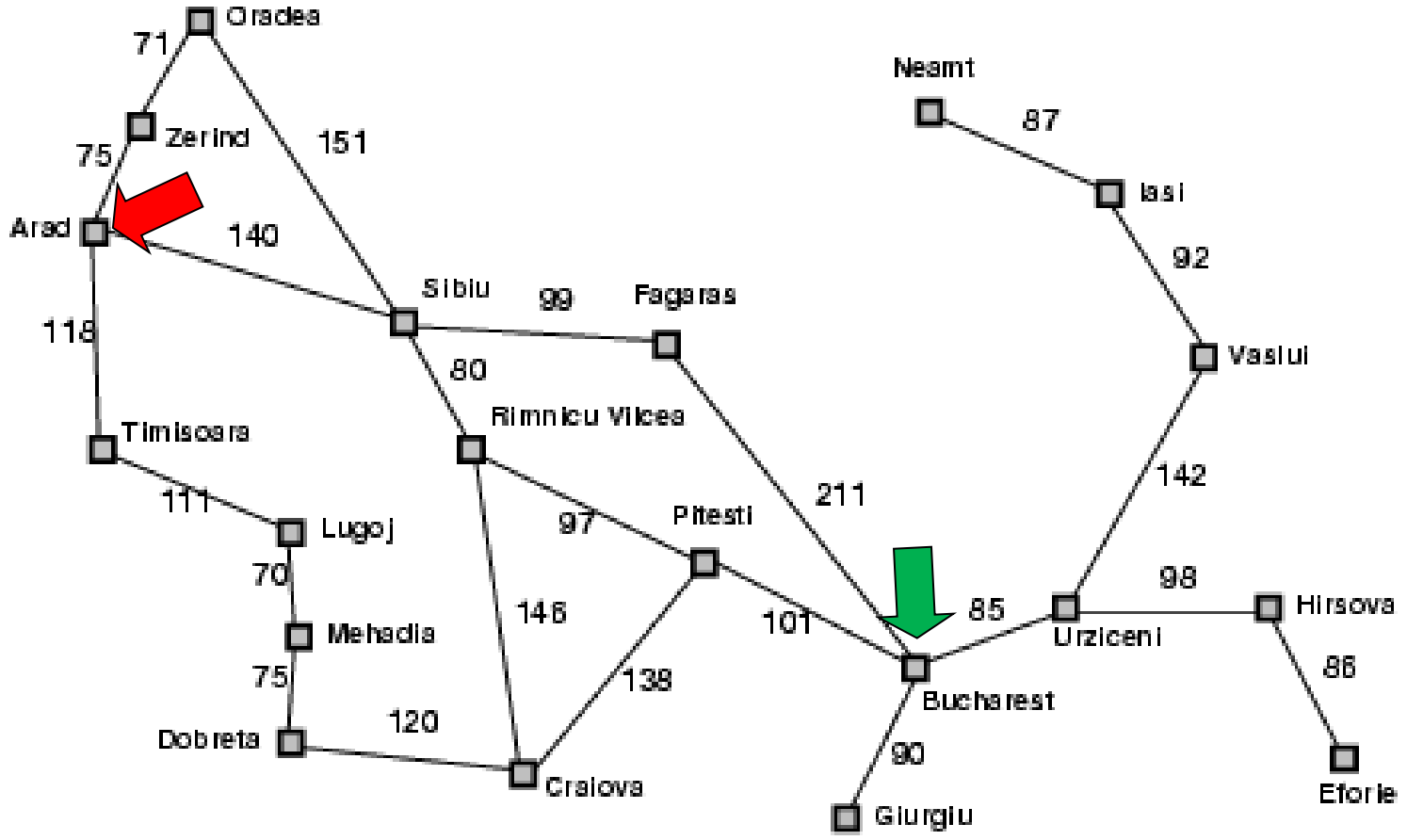
A* Search

Start Node

Goal Node



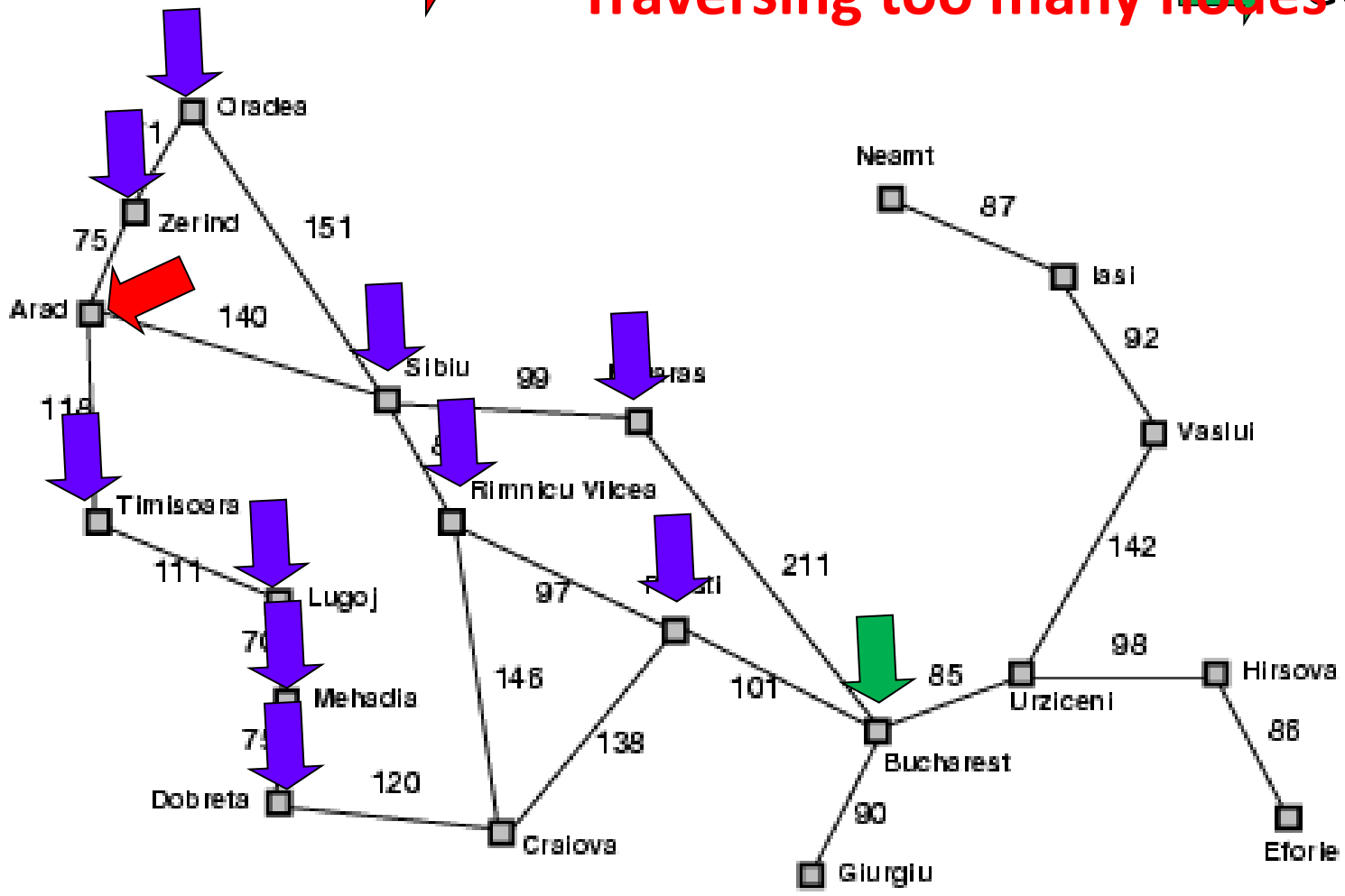
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Dijkstra's Algorithm

Start Node

Traversing too many nodes

Goal Node



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

Dijkstra's Algorithm

- Dijkstra's algorithm has one cost function, which is real cost value from source to each node:

$$f(n)=g(n).$$

IDEA!!!

- Use **heuristics** to guide the search.

  Heuristic: estimation of how to search for a solution

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

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

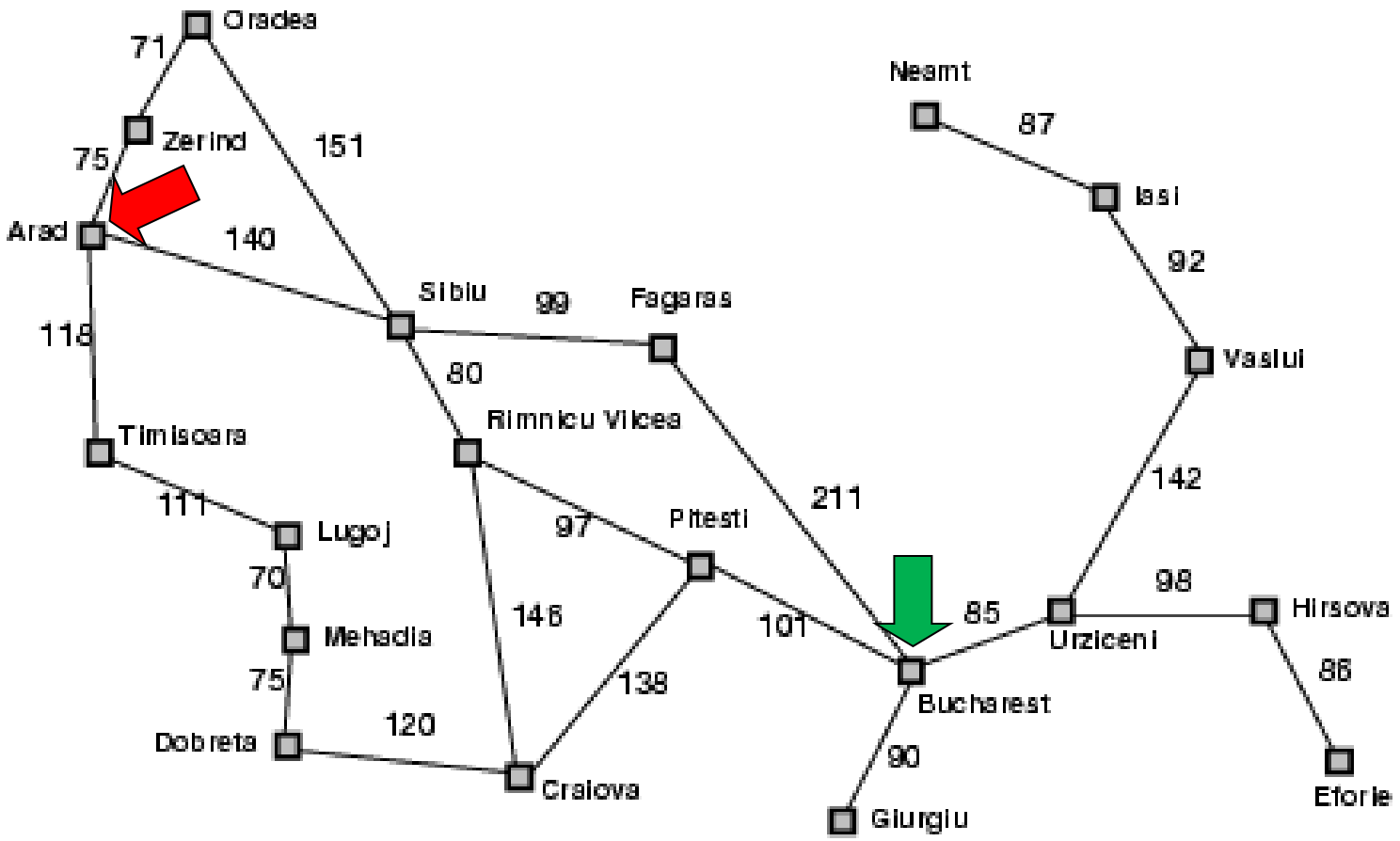
  $h(n)$ = estimated cost from n to goal

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

A* Search

Start Node

Goal Node



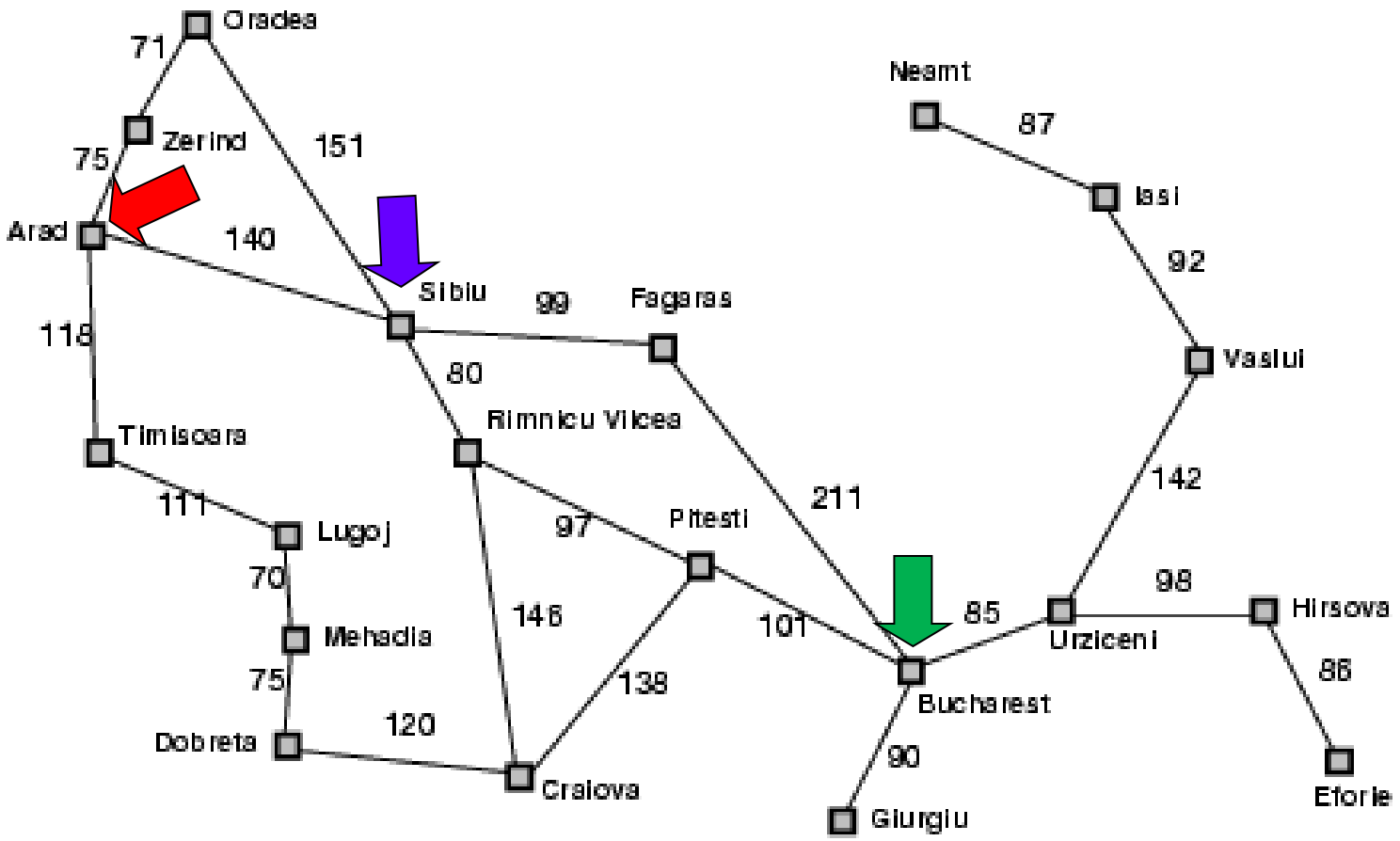
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* Search

Zerind: f(n) = g(n) + h(n) = 75 + 374 = 449

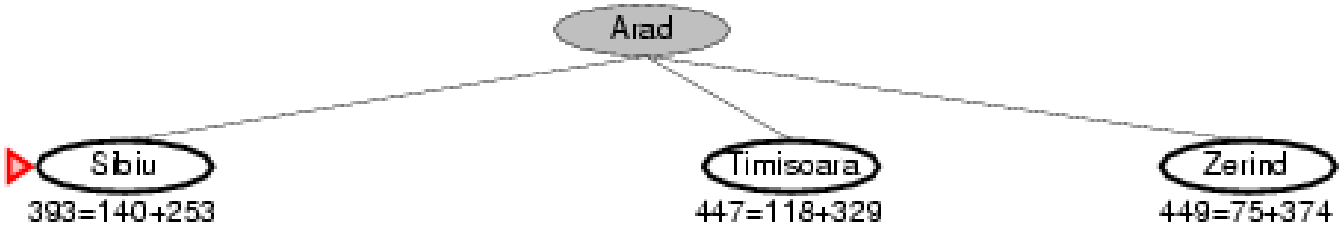
Timisoara: f(n) = g(n) + h(n) = 118 + 329 = 447

Sibiu: f(n) = g(n) + h(n) = 140 + 253 = 393



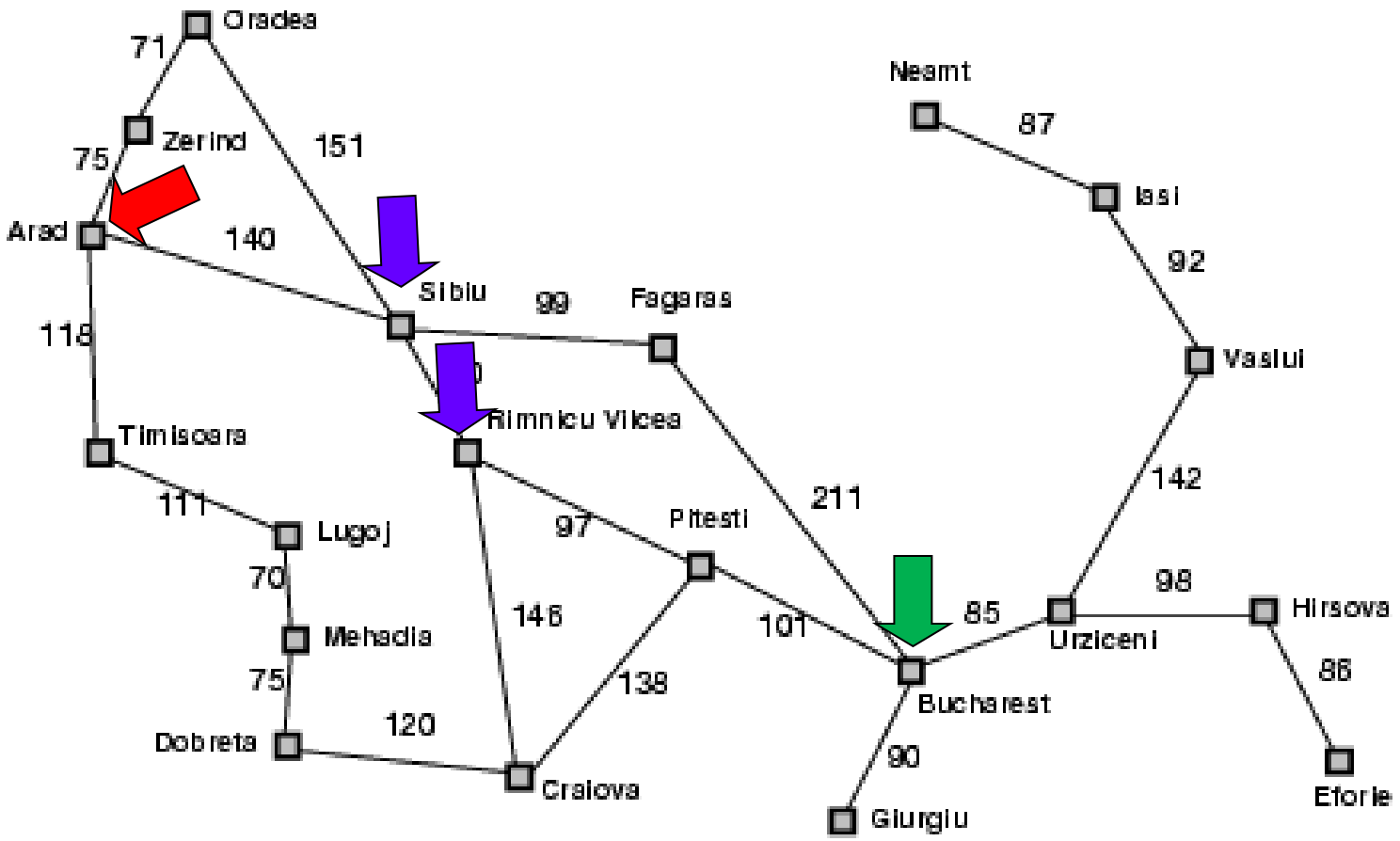
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* Search


Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374

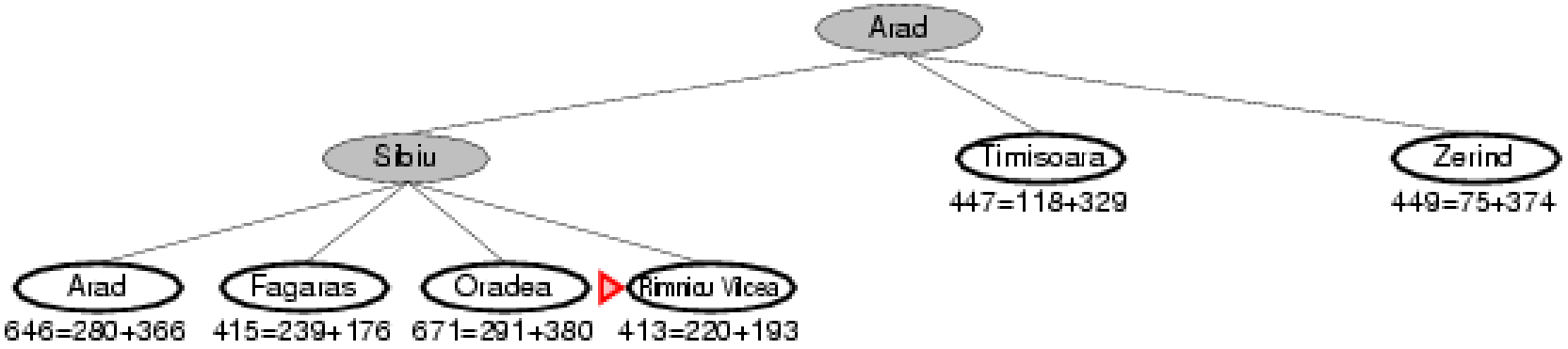
A* Search

Fagaras : f(n) = g(n) + h(n) = (140+99) + 176 = 415
Rimnicu Vilcea : f(n) = g(n) + h(n) = (140+80) + 193 = 413
Oradea : f(n) = g(n) + h(n) = (140+151) + 380 = 671



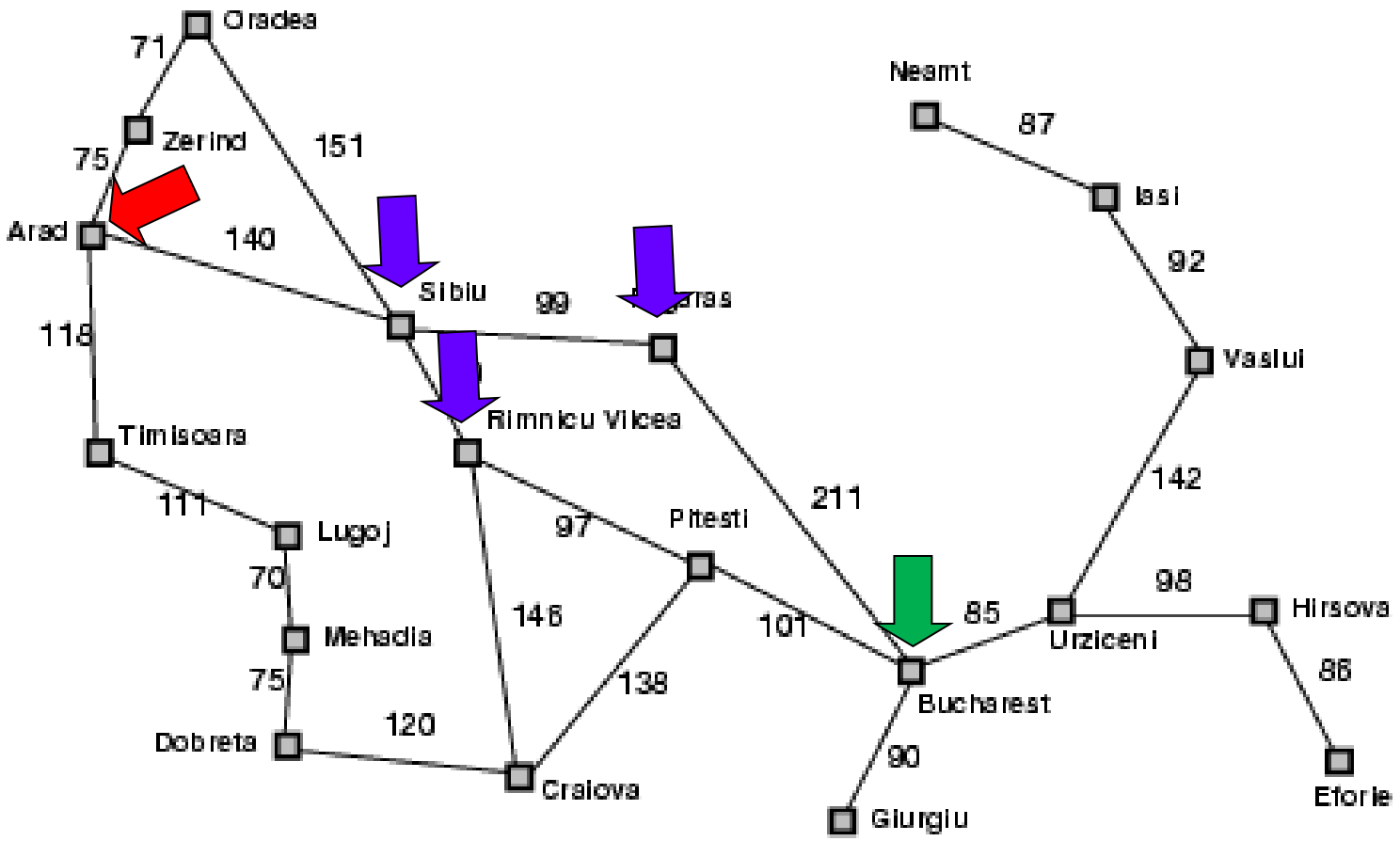
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* Search


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193

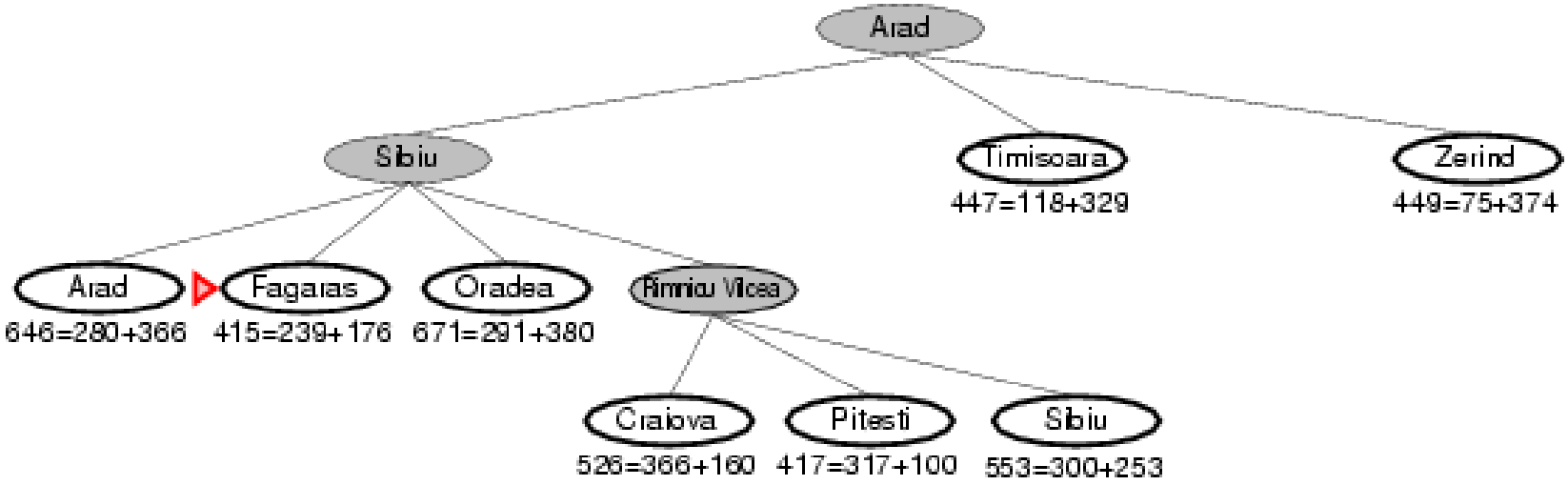
A* Search

Fagaras : f(n) = g(n) + h(n) = (140+99) + 176 = 415
Pitesti : f(n) = g(n) + h(n) = (140+80+97) + 100 = 417



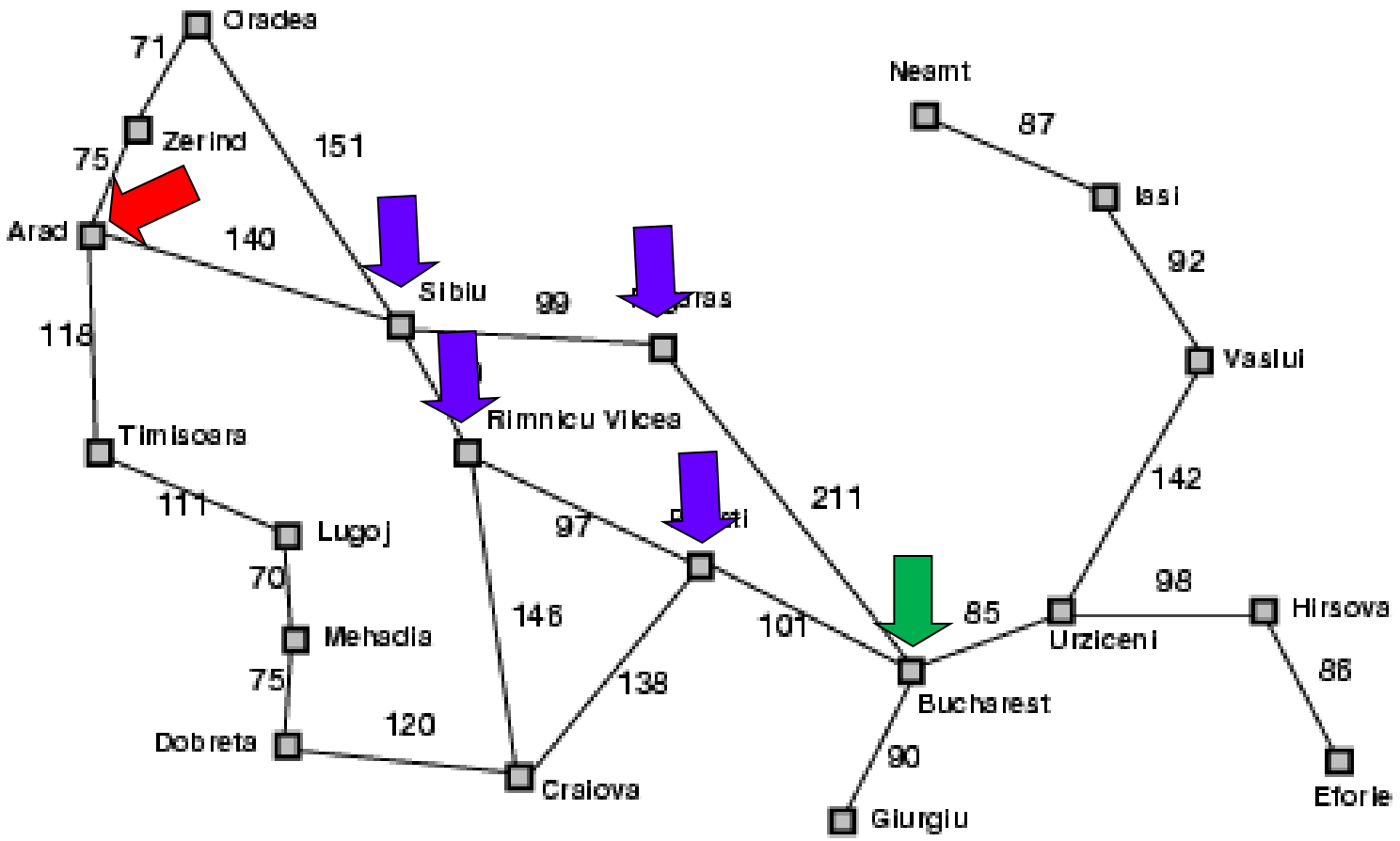
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* Search


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253

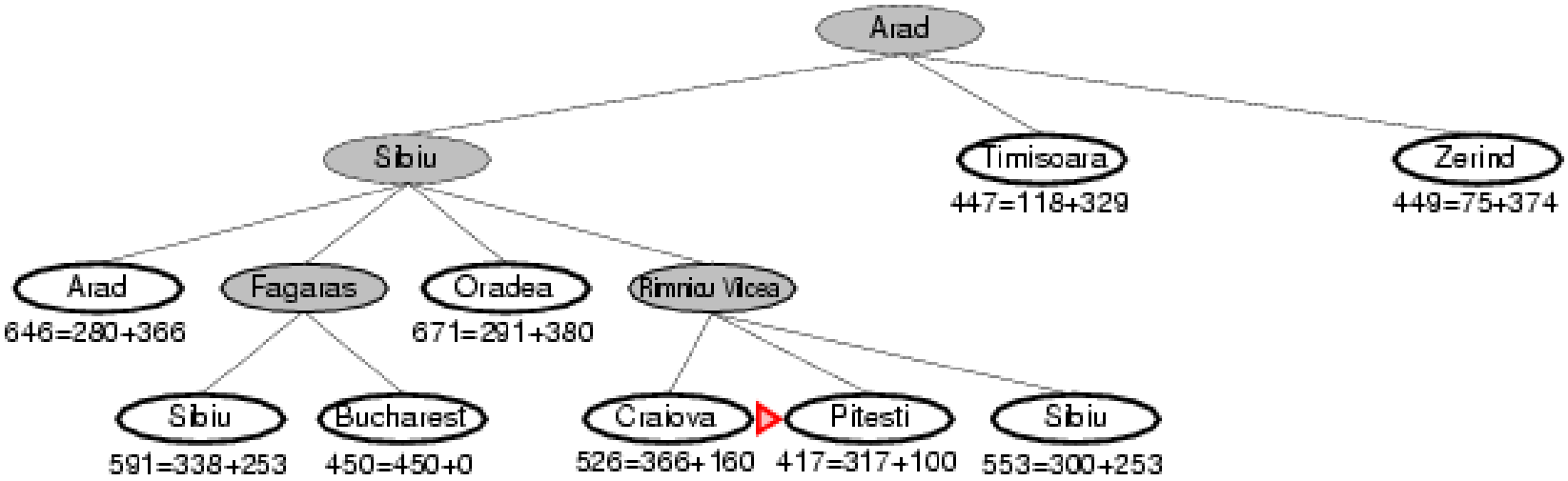
A* Search

Fagaras : f(n) = g(n) + h(n) = (140+99) + 176 = 415
Pitesti : f(n) = g(n) + h(n) = (140+80+97) + 100 = 417
Bucharest (via fagaras) : f(n) = g(n) + h(n) = (140+99+211) + 0 = 450



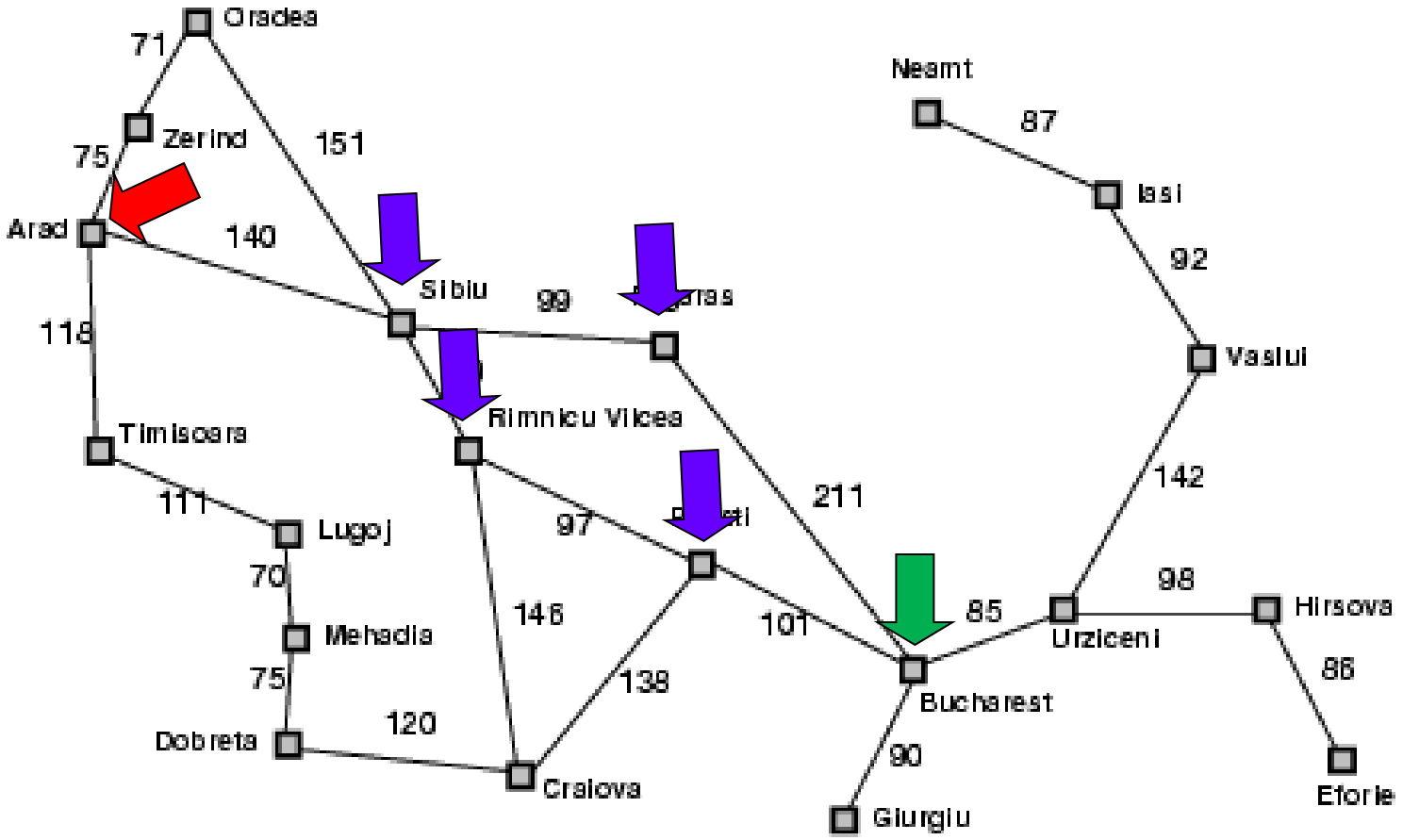
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* Search


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253

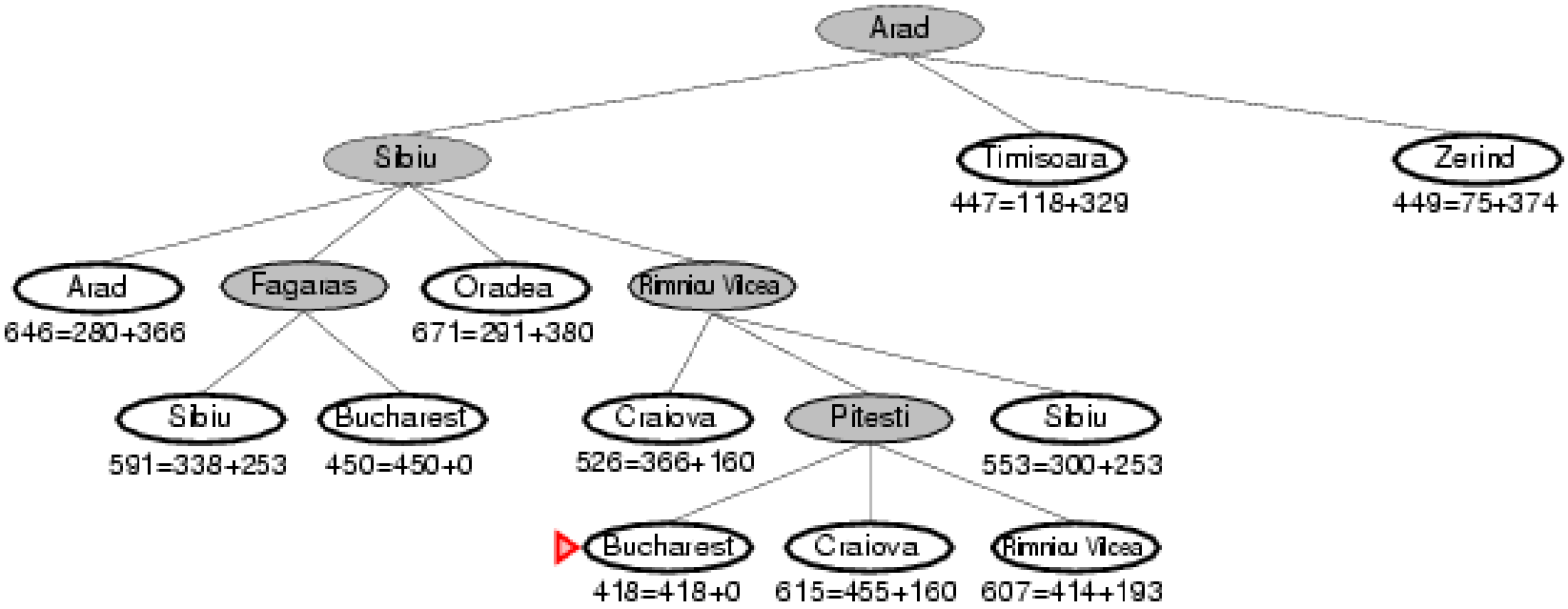
A* Search

Fagaras : f(n) = g(n) + h(n) = (140+99) + 176 = 415
Pitesti : f(n) = g(n) + h(n) = (140+80+97) + 100 = 417
Bucharest (via fagaras) : f(n) = g(n) + h(n) = (140+99+211) + 0 = 450
Bucharest (via Pitesti) : f(n) = g(n) + h(n) = (140+80+97+101) + 0 = 418



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	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* Search


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
Sibiu
553=300+253
Bucharest
418=418+0
Craiova
615=455+160
Rimnicu Vilcea
607=414+193

A* Search

```
function A*(start,goal)
    closedset := the empty set    // The set of nodes already evaluated.
    openset := {start}    // The set of tentative nodes to be evaluated, initially containing the start node
    came_from := the empty map    // The map of navigated nodes.

    g_score[start] := 0    // Cost from start along best known path.
    f_score[start] := g_score[start] + heuristic_cost_estimate(start, goal) // Estimated total cost

    while openset is not empty
        current := the node in openset having the lowest f_score[] value
        if current = goal
            return reconstruct_path(came_from, goal)

        remove current from openset
        add current to closedset
        for each neighbor in neighbor_nodes(current)
            if neighbor in closedset
                continue
            tentative_g_score := g_score[current] + dist_between(current,neighbor)

            if neighbor not in openset or tentative_g_score < g_score[neighbor]
                came_from[neighbor] := current
                g_score[neighbor] := tentative_g_score
                f_score[neighbor] := g_score[neighbor] + heuristic_cost_estimate(neighbor, goal)
                if neighbor not in openset
                    add neighbor to openset
    return failure

function reconstruct_path(came_from,current)
    total_path := [current]
    while current in came_from:
        current := came_from[current]
        total_path.append(current)
    return total_path
```

A* Search

- When h(n) = actual cost to goal
  - Only nodes in the correct path are expanded
  - Optimal solution is found
- When h(n) < actual cost to goal
  - Additional nodes are expanded
  - Optimal solution is found
- When h(n) > actual cost to goal
  - Optimal solution can be overlooked

Admissible heuristics

- A heuristic *h(n)* is admissible if for every node *n*, $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from *n*.
- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic.
- If *h(n)* is admissible, A* using **TREE-SEARCH** is optimal

Dominance

- If $h_2(n) \geq h_1(n)$ for all n (both admissible)

  then h_2 dominates h_1

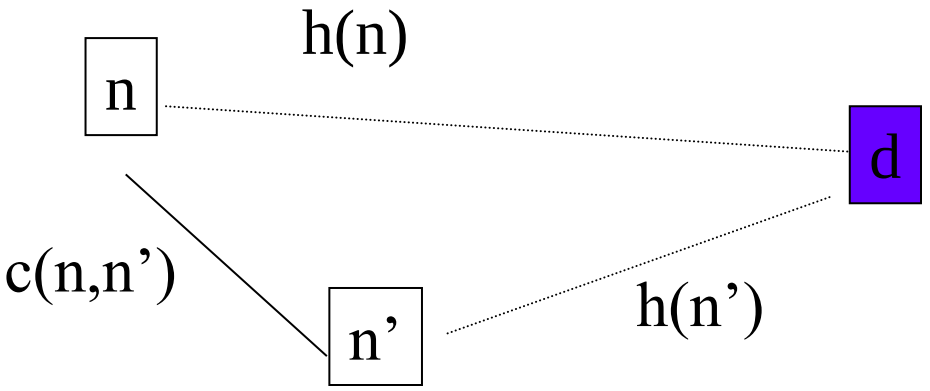
- h_2 is better for search: it is guaranteed to expand

  less or equal number of nodes.

Consistent heuristics

- h(n) is **consistent** if for every node n and for every successor node n' of n:

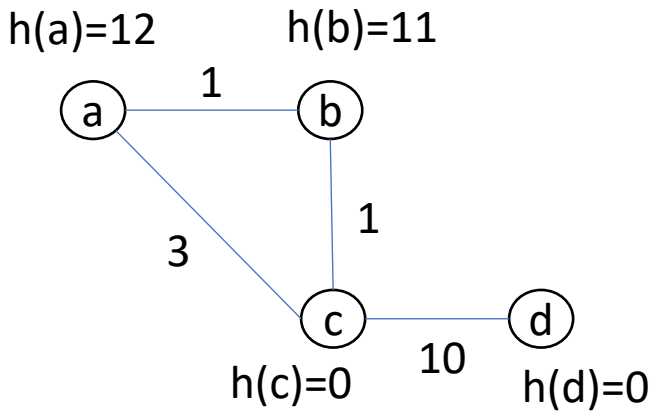
$$h(n) \leq c(n, n') + h(n')$$



Consistent heuristics

- If h(n) is consistent then h(n) is admissible
- Frequently when h(n) is admissible, it is also consistent.
- If h(n) is a consistent heuristic, A* using graph search is optimal.

Consistent heuristics



- H is admissible but not consistent heuristic
- Graph search will find path a-c-d.
  - c will be expanded after a, and there will be no option to expand c again after expanding b (Since c already in closed set)

THANK YOU!!!