بسم الله الرحمن الرحيم

BISMILLAH ARRAHMAN ARRAHEEM

Artificial Intelligence (CS-401)

Lecture 4:
Heuristic Informed Search Algorithms

Dr. Fahad Sherwani (Assistant Professor – FAST NUCES)

PhD in Artificial Intelligence

Universiti Tun Hussein Onn Malaysia

fahad.sherwani@nu.edu.pk

What is meant by Heuristics?

- Derived from a Greek word that means “To Discover"
- Heuristic describes a rule of thumb or a method
- Comes from experience
- Helps you think through things
- Like the process of elimination
- The process of trial and error
- You can think of a heuristic as a shortcut

Best-first search

Idea: use an evaluation function *f(n)* for each node

- family of search methods with various evaluation functions (estimate of "desirability“)
- usually gives an estimate of the distance to the goal
- often referred to as *heuristics* in this context

→Expand most desirable unexpanded node

→A heuristic function ranks alternatives at each branching step based on the available information (heuristically) in order to make a decision about which branch to follow during a search.

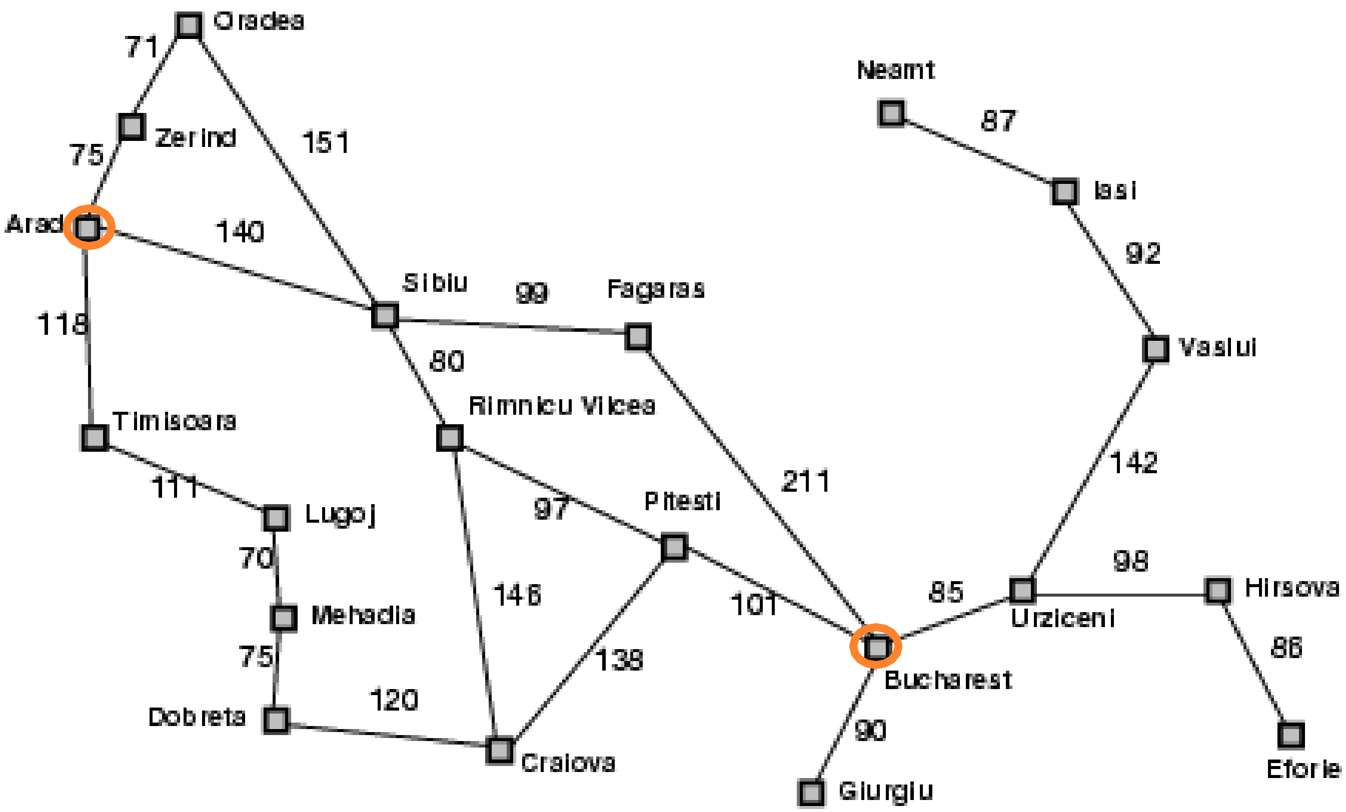
Implementation:

Order the nodes in fringe in decreasing order of desirability.

Special cases:

- Greedy best-first search
- A* search

Romania with step costs in km



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

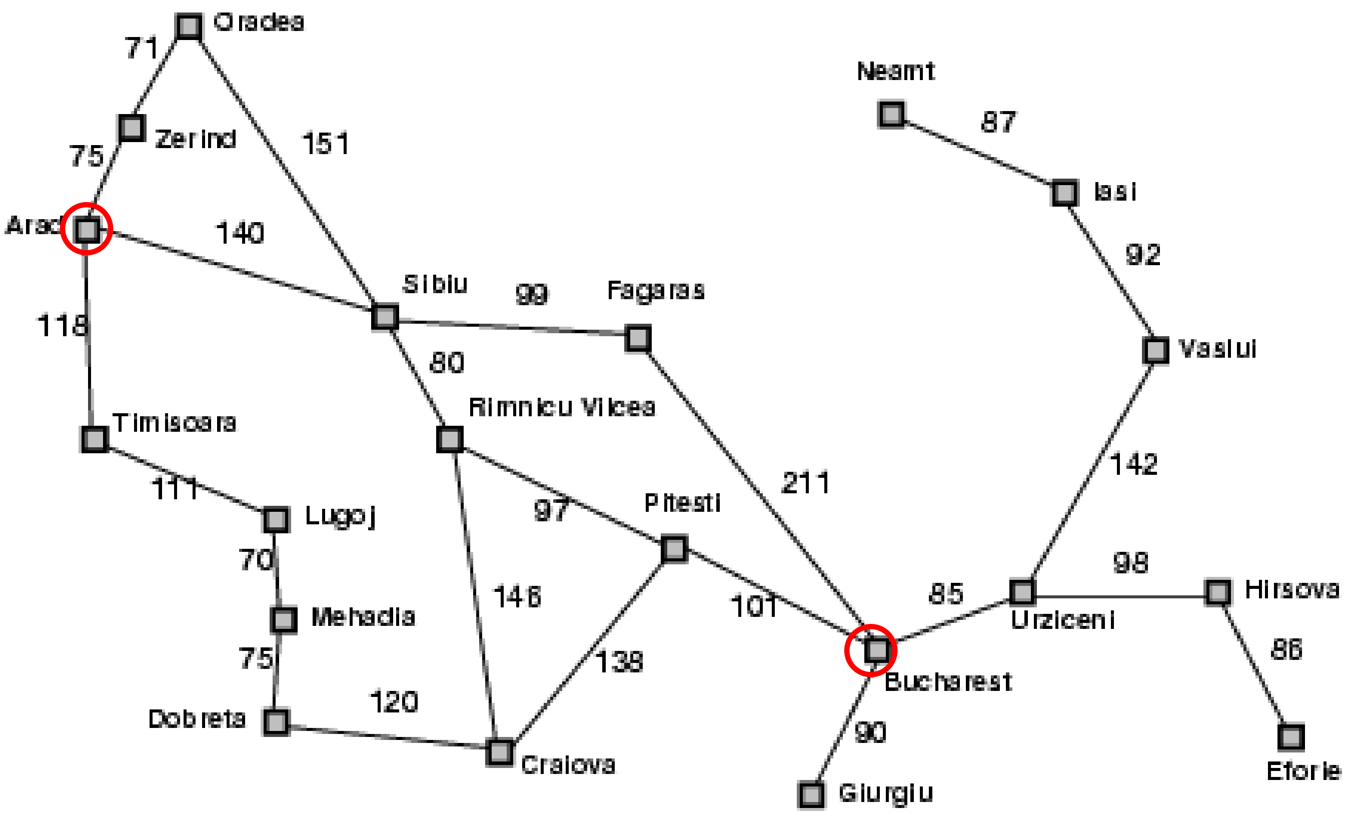
Greedy best-first search

- Greedy best-first search expands the node that appears to be closest to goal heuristically.
- Estimate of cost from *n* to *goal* ,e.g., $h_{SLD}(n)$ = straight-line distance from *n* to Bucharest.

Utilizes a heuristic function as evaluation function

- $f(n) = h(n)$ = estimated cost from the current node to a goal.
- Heuristic functions are problem-specific.
- Often employs straight-line distance for route-finding and similar problems.
- Often better than depth-first, although worst-time complexities are equal or worse (space).

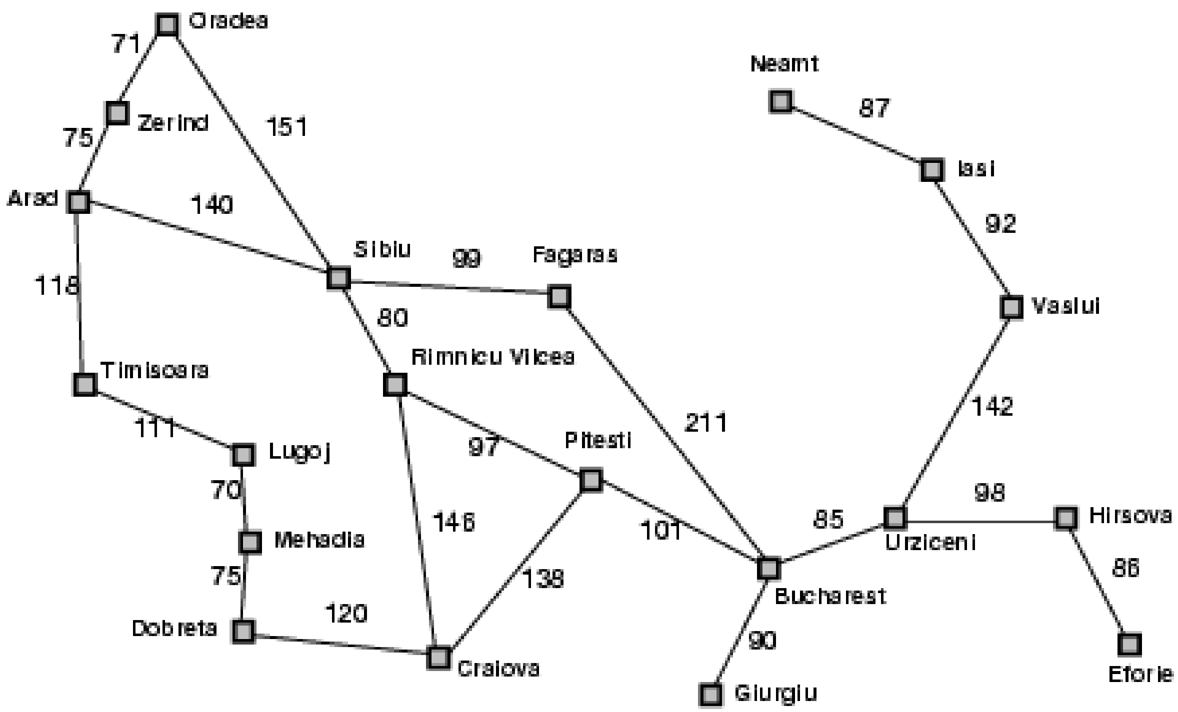
Greedy best-first search example



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

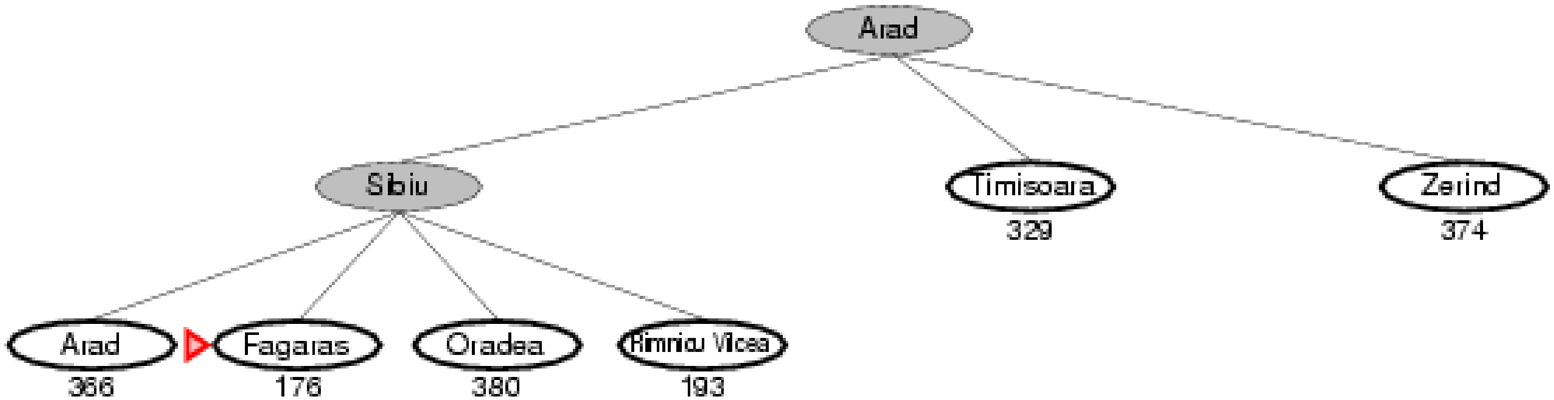
Greedy best-first search example



Straight–line distance
to Bucharest

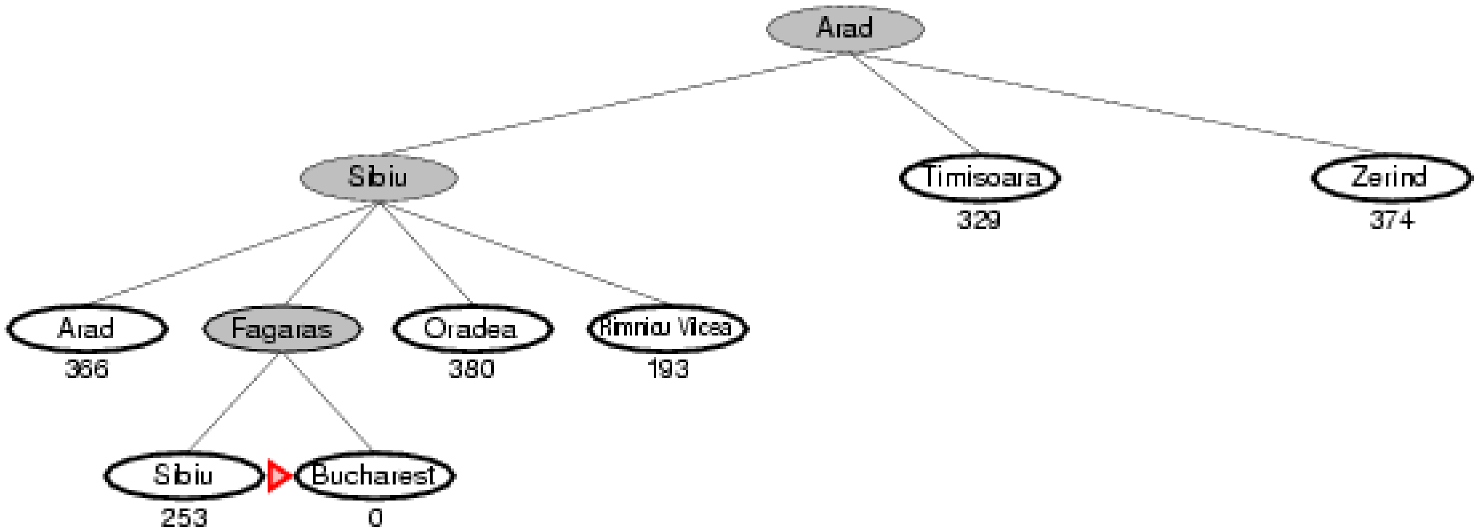
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best-first search example



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

Greedy best-first search example



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

Greedy best-first search

```
function GREEDY-BEST-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE : /* Cost f(n) = h(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

Properties of greedy best-first search

Complete: No – can get stuck in loops (e.g., Iasi → Neamt → Iasi → Neamt → ….)

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

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

Optimal: No

b	branching factor
m	maximum depth of the search tree

A^* search

Idea: avoid expanding paths that are already expensive.

Evaluation function = path cost + estimated cost to the goal

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

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

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

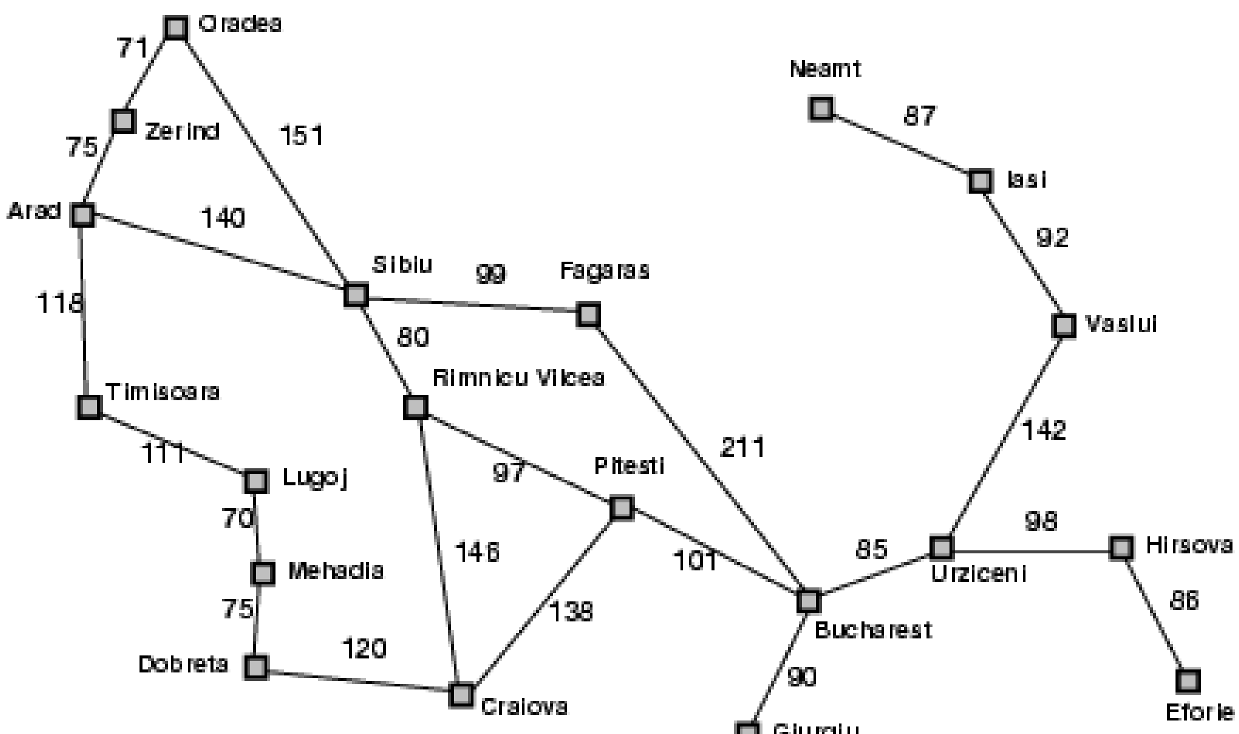
-$f(n)$ = estimated total cost of path through n to goal

Combines greedy and uniform-cost search to find the (estimated) cheapest path through the current node

- Heuristics must be admissible
  - Never overestimate the cost to reach the goal
- Very good search method, but with complexity problems

A* search example

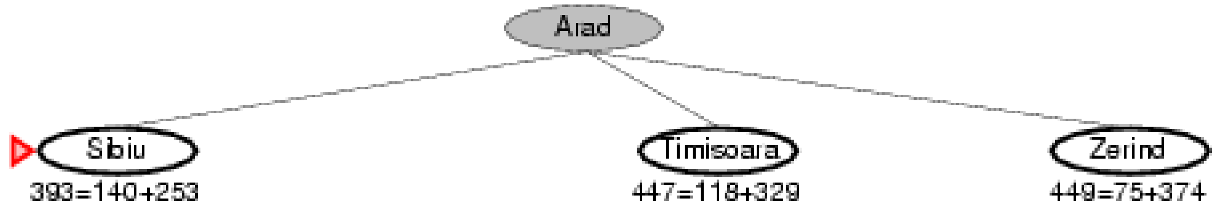
Example from [1]

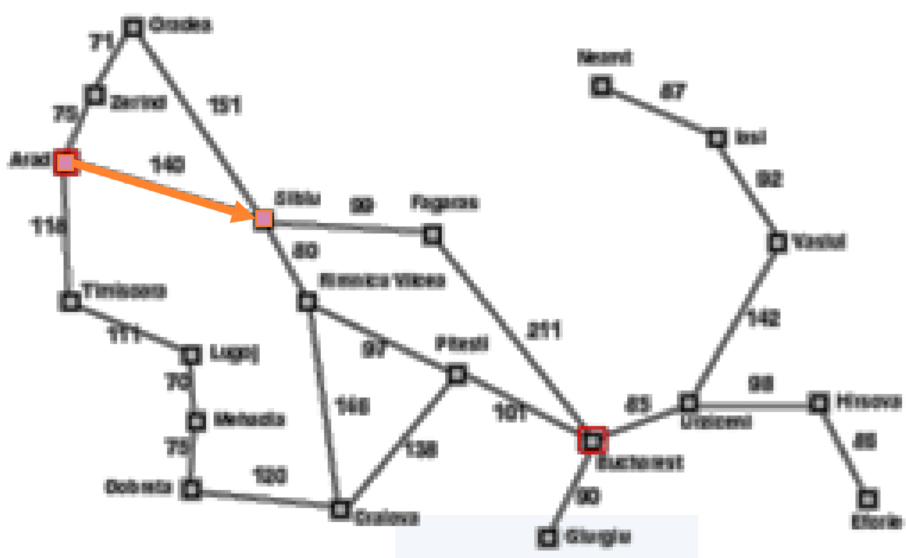


Straight–line distance to Bucharest

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

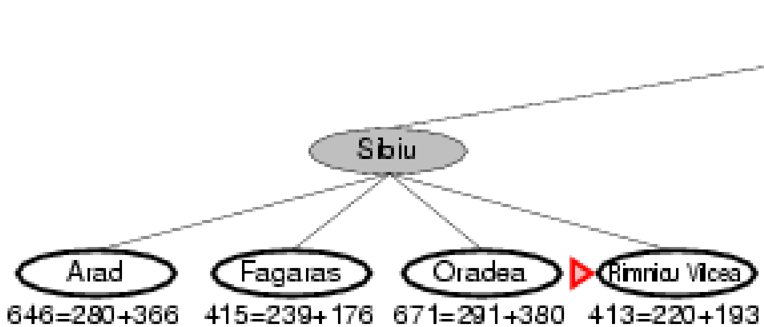
A* search example





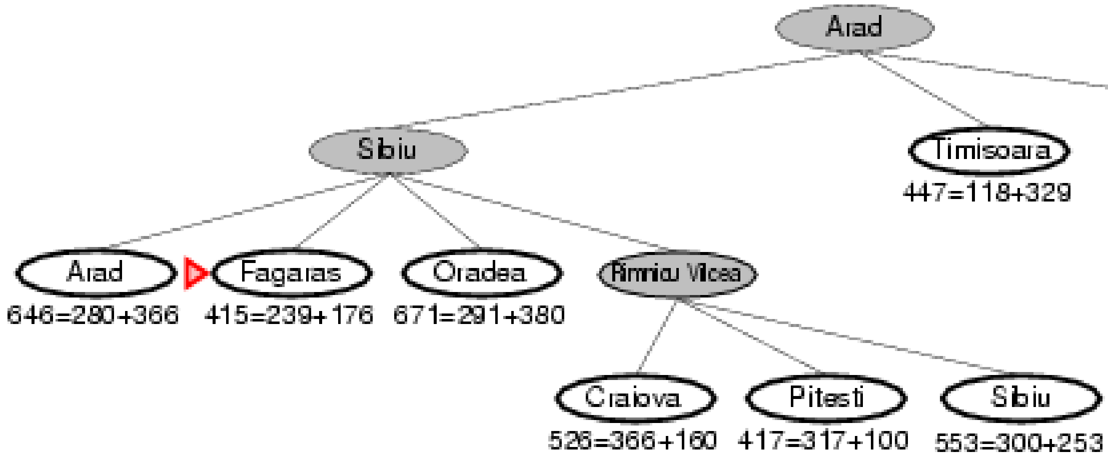
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

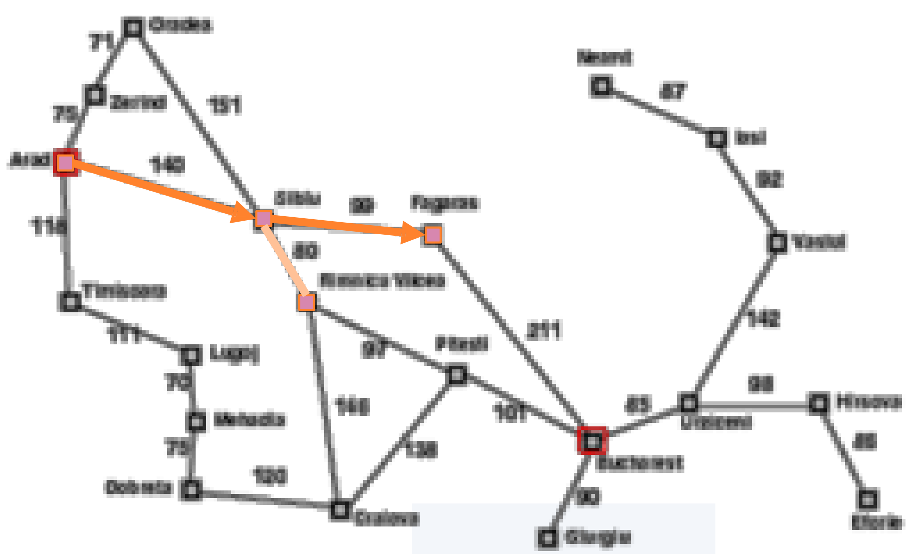
A* search example



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

A* search example

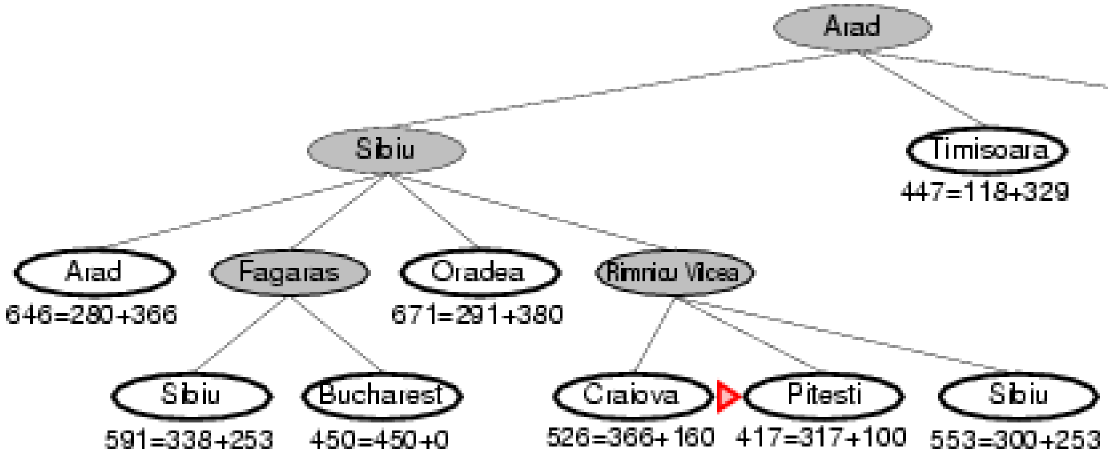




Straight-line distance to Bucharest

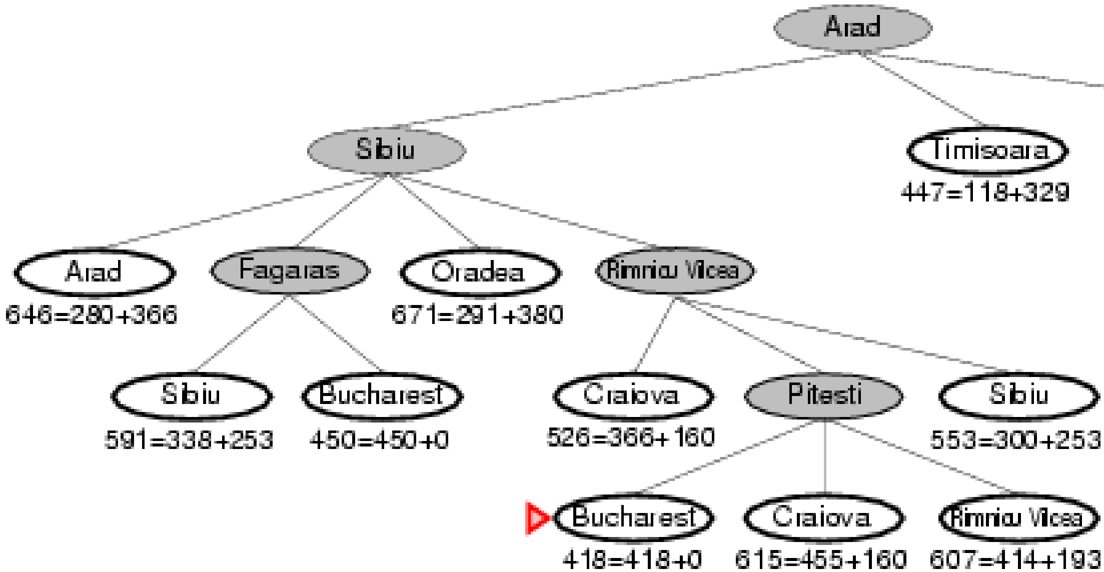
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* search example



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

A* search example



Straight-line distance
to Bucharest

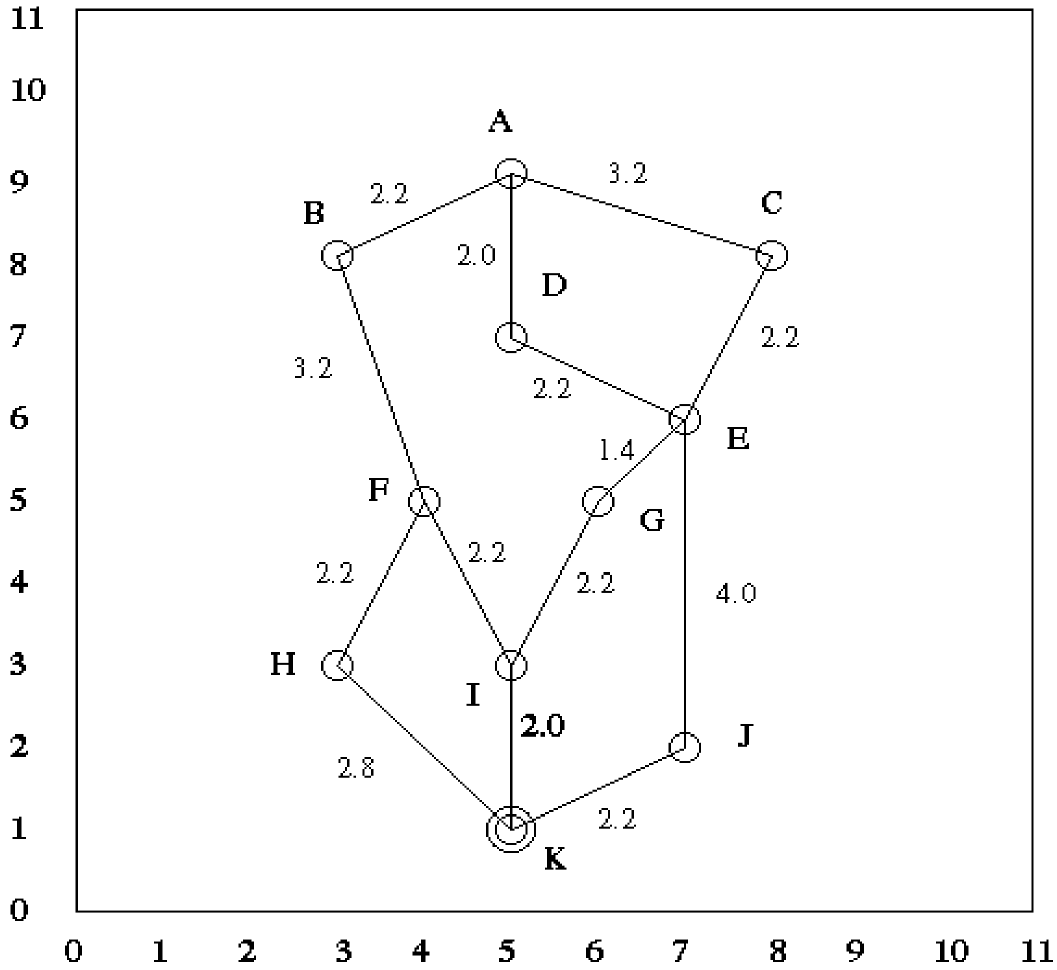
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* Algorithm

```
function A-STAR-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE : /* Cost f(n) = g(n) + h(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

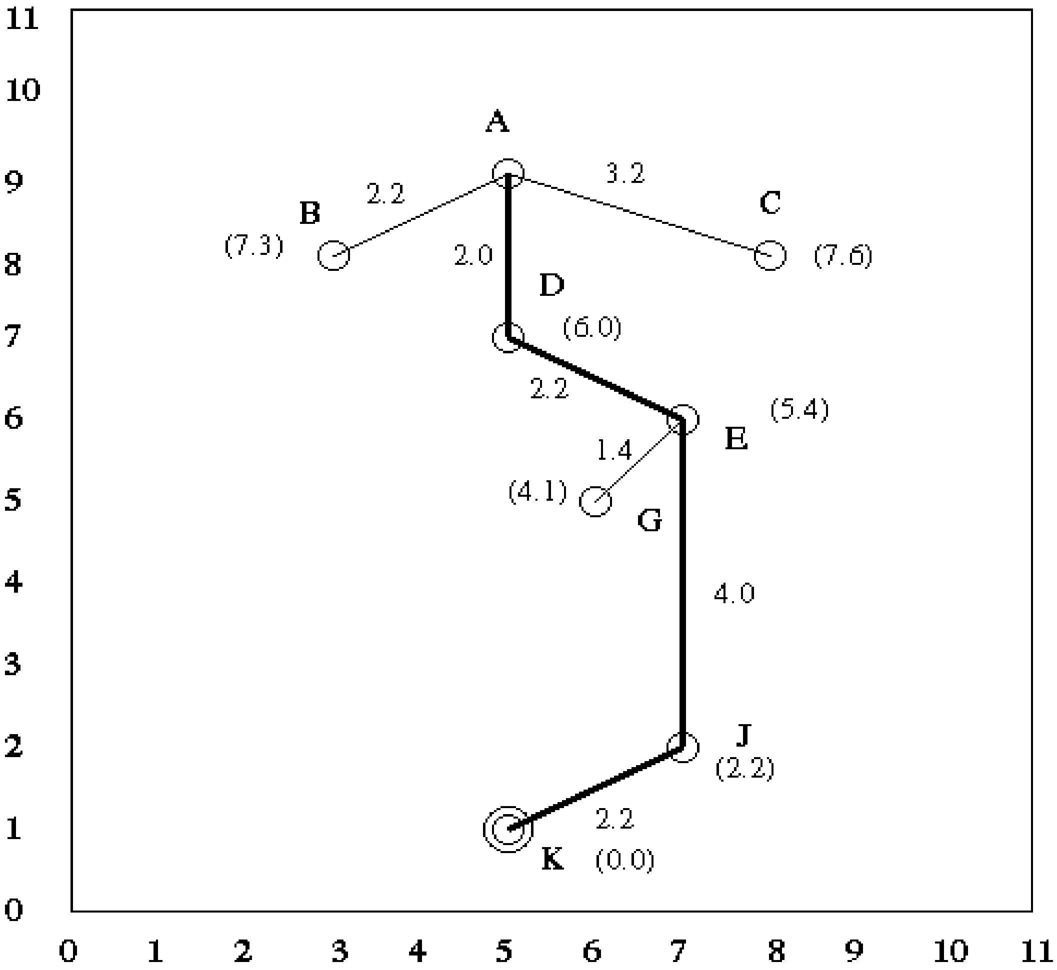
Greedy Best-First Exercise

Node	Coordinates	h(n)
A	(5,9)	8.0
B	(3,8)	7.3
C	(8,8)	7.6
D	(5,7)	6.0
E	(7,6)	5.4
F	(4,5)	4.1
G	(6,5)	4.1
H	(3,3)	2.8
I	(5,3)	2.0
J	(7,2)	2.2
K	(5,1)	0.0

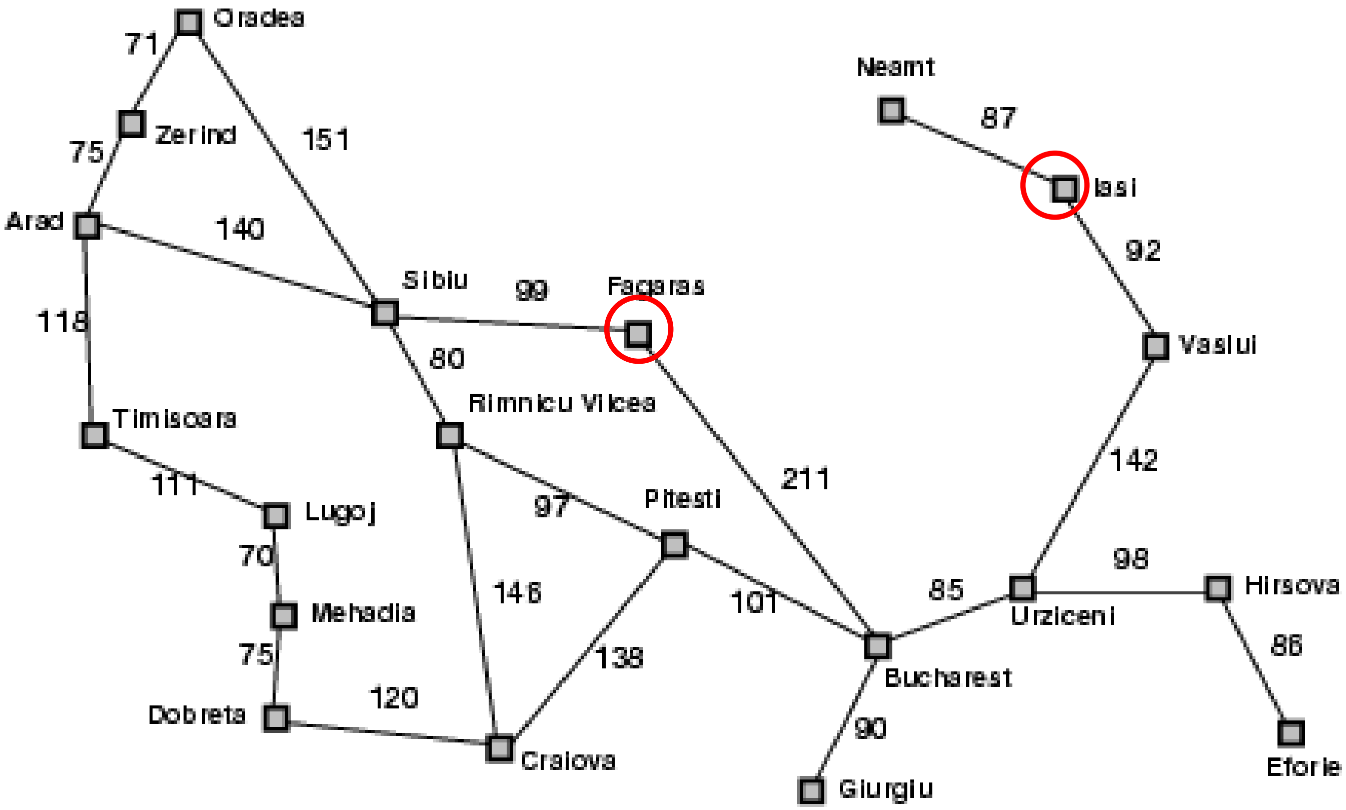


Solution to Greedy Best-First Exercise

Node	Coordinates	h(n)
A	(5,9)	8.0
B	(3,8)	7.3
C	(8,8)	7.6
D	(5,7)	6.0
E	(7,6)	5.4
F	(4,5)	4.1
G	(6,5)	4.1
H	(3,3)	2.8
I	(5,3)	2.0
J	(7,2)	2.2
K	(5,1)	0.0



A* Exercise



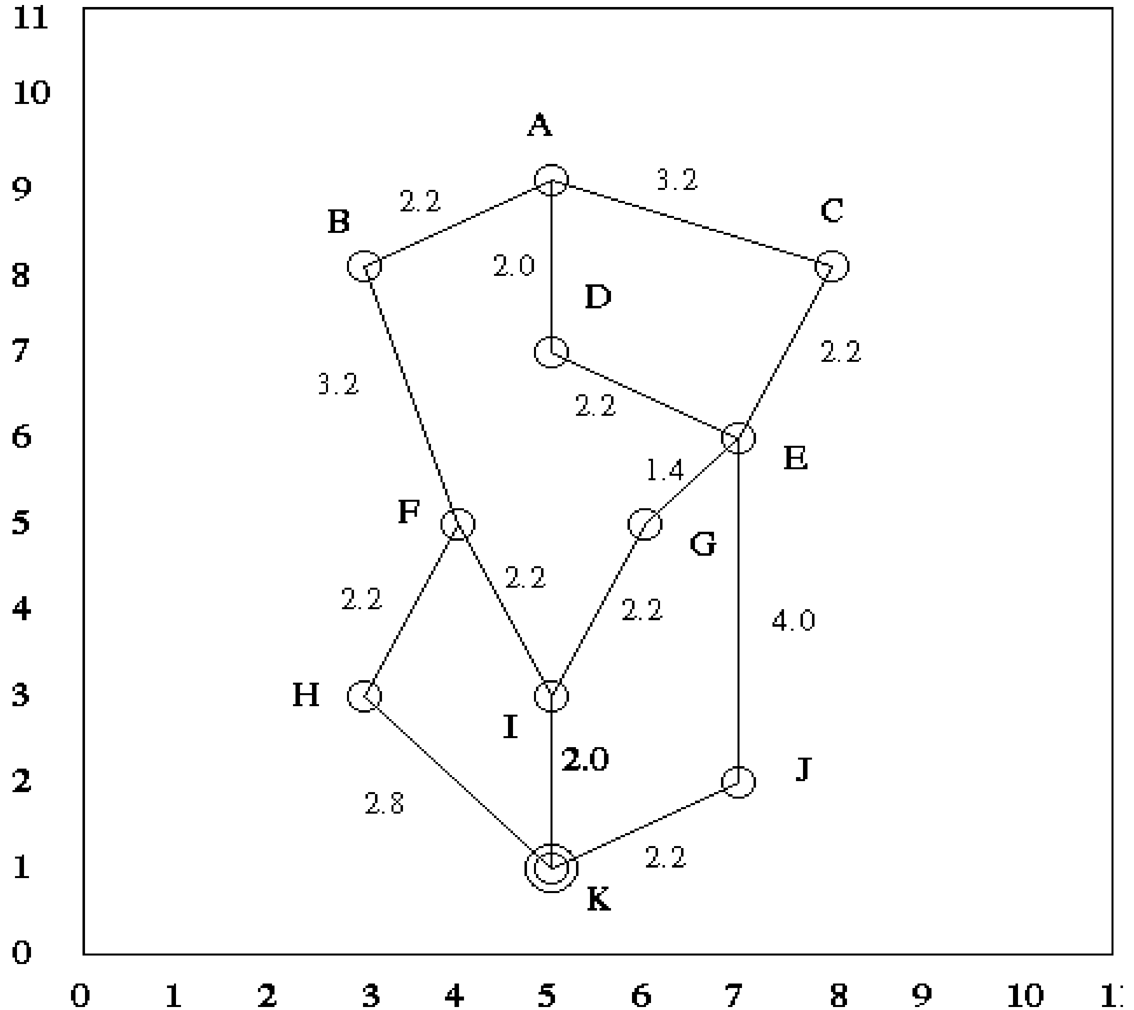
Straight–line distance to Bucharest

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

How will A* get from Iasi to Fagaras?

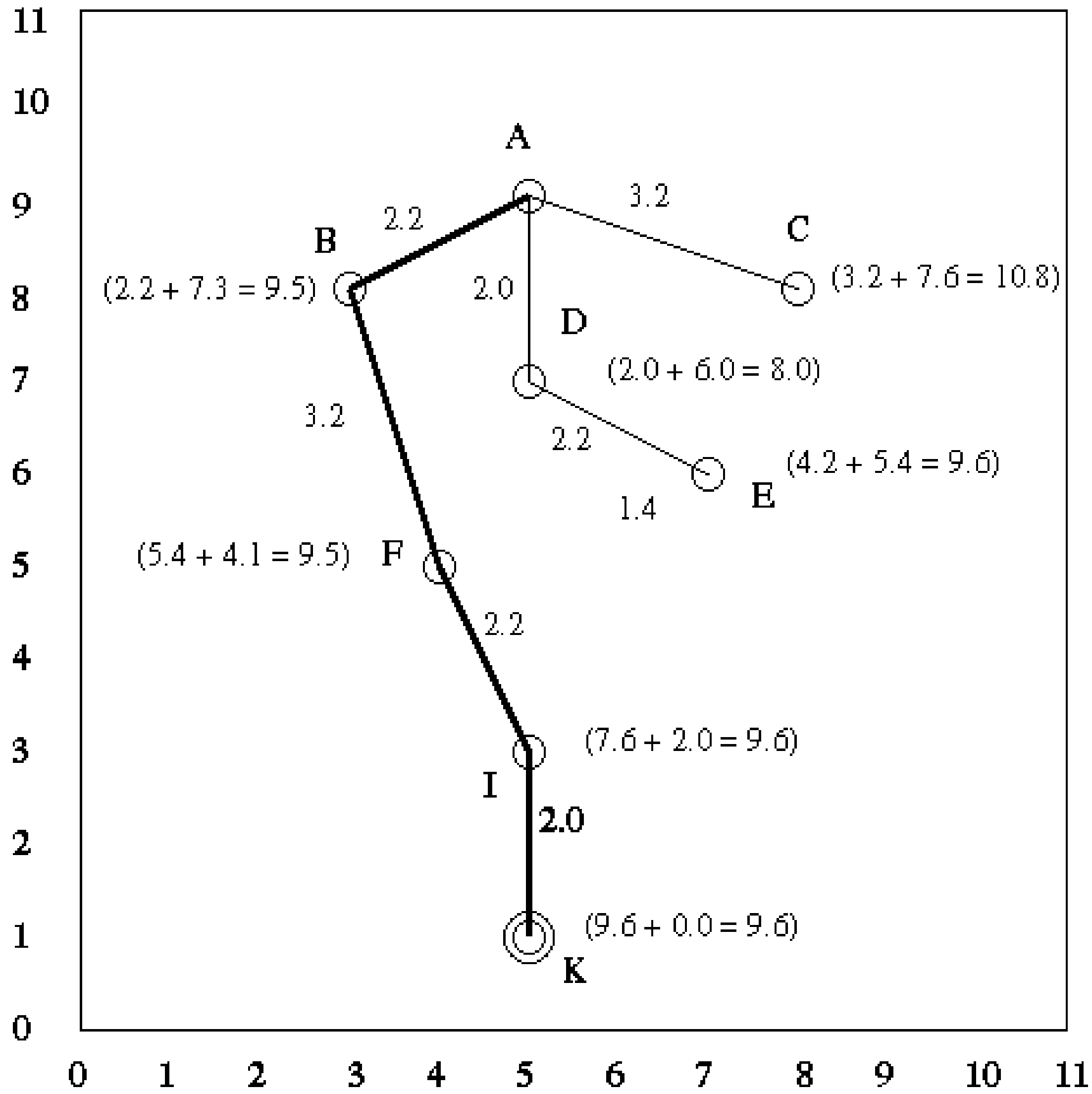
A* Exercise

Node	Coordinates	h(n)
A	(5,9)	8.0
B	(3,8)	7.3
C	(8,8)	7.6
D	(5,7)	6.0
E	(7,6)	5.4
F	(4,5)	4.1
G	(6,5)	4.1
H	(3,3)	2.8
I	(5,3)	2.0
J	(7,2)	2.2
K	(5,1)	0.0



Solution to A* Exercise

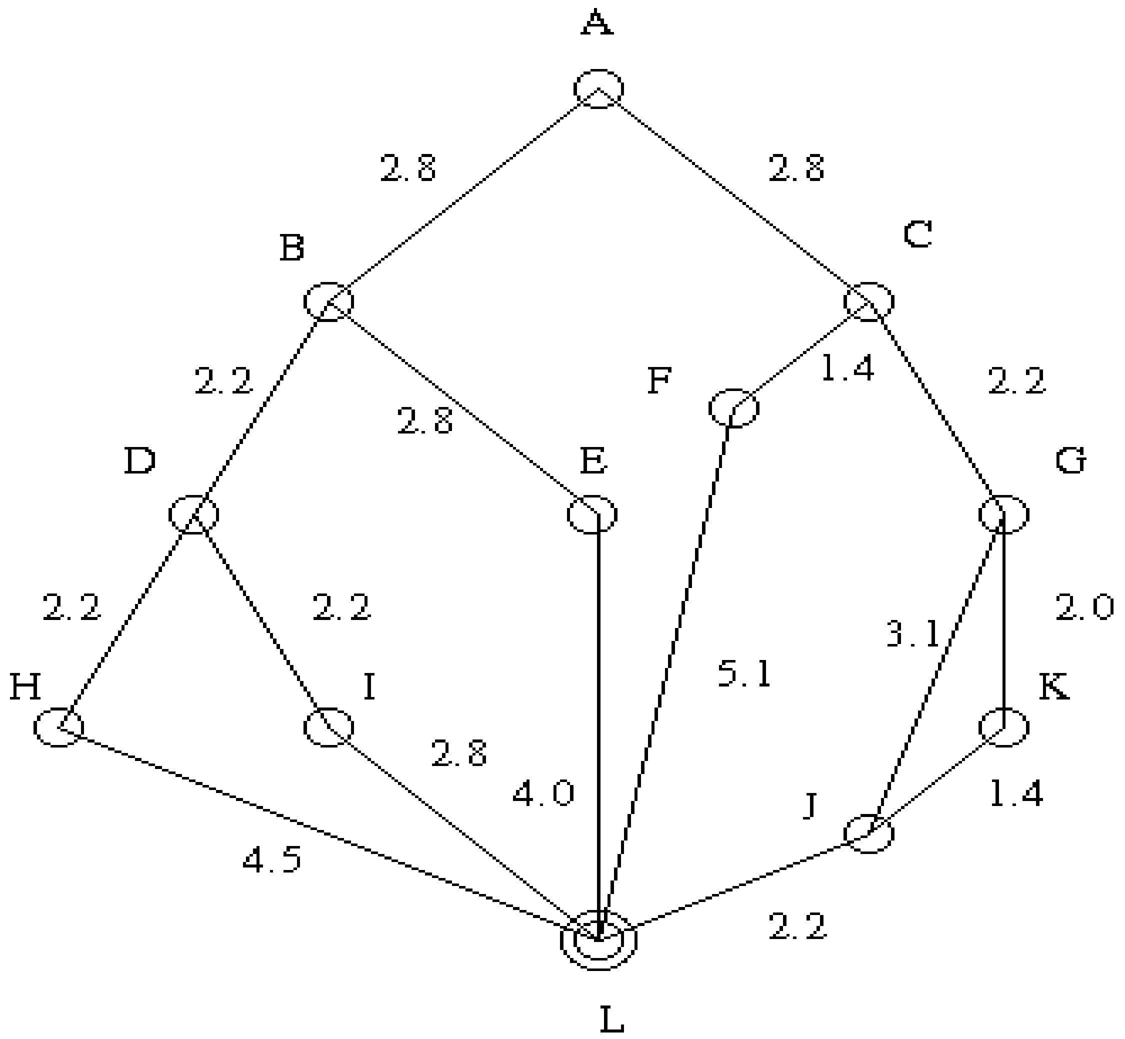
Node	Coordinates	h(n)
A	(5,9)	8.0
B	(3,8)	7.3
C	(8,8)	7.6
D	(5,7)	6.0
E	(7,6)	5.4
F	(4,5)	4.1
G	(6,5)	4.1
H	(3,3)	2.8
I	(5,3)	2.0
J	(7,2)	2.2
K	(5,1)	0.0



Another Exercise

Do 1) A* Search and 2) Greedy Best-Fit Search

Node	C	g(n)	h(n)
A	(5,10)	0.0	8.0
B	(3,8)	2.8	6.3
C	(7,8)	2.8	6.3
D	(2,6)	5.0	5.0
E	(5,6)	5.6	4.0
F	(6,7)	4.2	5.1
G	(8,6)	5.0	5.0
H	(1,4)	7.2	4.5
I	(3,4)	7.2	2.8
J	(7,3)	8.1	2.2
K	(8,4)	7.0	3.6
L	(5,2)	9.6	0.0



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.

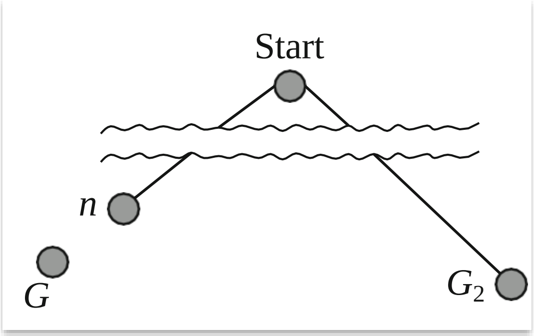
The heuristic function $h_{SLD}(n)$ *is* admissible because it never overestimates the actual road distance)

Theorem-1: If $h(n)$ is admissible, A^* using `TREE-SEARCH` is optimal.

Optimality of A^* (proof)

Recall that $f(n) = g(n) + h(n)$

Now, suppose some suboptimal goal G_2 has been generated and is in the fringe. Let n be an unexpanded node in the fringe such that n is on a shortest path to an optimal goal G.



We want to prove:

$f(n) < f(G_2)$

(then A* will prefer n over G_2)

$f(G_2) = g(G_2)$ since $h(G_2) = 0$

$g(G_2) > g(G)$ since G_2 is suboptimal

$f(G) = g(G)$ since $h(G) = 0$

Then $f(G_2) > f(G)$ from above

$h(n) \leq h^*(n)$ since h is admissible

$g(n) + h(n) \leq g(n) + h^*(n)$

Then $f(n) \leq f(G)$

Thus, A^* will never select G_2 for expansion

Optimality of A* (proof)

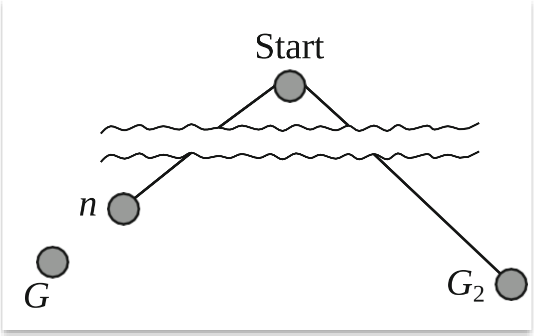
Recall that $f(n) = g(n) + h(n)$

Now, suppose some suboptimal goal G_2 has been generated and is in the fringe. Let n be an unexpanded node in the fringe such that n is on a shortest path to an optimal goal G.

We want to prove:

$\boldsymbol{f(n) < f(G_2)}$

(then A* will prefer $\boldsymbol{n}$ over G_2)



In other words:

$f(G_2) = g(G_2) + h(G_2) = g(G_2) > C^*$,

since G_2 is a goal on a non-optimal path (C^* is the optimal cost)

$f(n) = g(n) + h(n) \leq C^*$, since h is admissible

$f(n) \leq C^* < f(G_2)$, so G_2 will never be expanded

→ A* will not expand goals on sub-optimal paths

Properties of A*

- Complete: Yes

  unless there are infinitely many nodes with $f \leq f(G)$

- Time: Exponential

  because all nodes such that $f(n) \leq C^*$ are expanded!

- Space: Keeps all nodes in memory

  fringe is exponentially large

- Optimal: Yes

Memory Bounded Heuristic Search

How can we solve the memory problem for A* search?

Idea: Try something like iterative deeping search, but the cutoff is f-cost (g+h) at each iteration, rather than depth first.

Two types of memory bounded heuristic searches:

- Recursive BFS
- SMA*

Simple Memory Bounded A* (SMA*)

- This is like A*, but when memory is full we delete the worst node (largest *f*-value).
- Like RBFS, we remember the best descendent in the branch we delete.
- If there is a tie (equal *f*-values) we first delete the oldest nodes first.
- SMA* finds the optimal *reachable* solution given the memory constraint.
- But time can still be exponential.

SMA* pseudocode

Based on [2]

```
function SMA*(problem) returns a solution sequence
  inputs: problem, a problem
  static: Queue, a queue of nodes ordered by f-cost

  Queue ← MAKE-QUEUE({MAKE-NODE(INITIAL-STATE[problem])})
  loop do
      if Queue is empty then return failure
      n ← deepest least-f-cost node in Queue
      if GOAL-TEST(n) then return success
      s ← NEXT-SUCCESSOR(n)
      if s is not a goal and is at maximum depth then
          f(s) ← ∞
      else
          f(s) ← MAX(f(n),g(s)+h(s))
      if all of n's successors have been generated then
          update n's f-cost and those of its ancestors if necessary
      if SUCCESSORS(n) all in memory then remove n from Queue
      if memory is full then
          delete shallowest, highest-f-cost node in Queue
          remove it from its parent's successor list
          insert its parent on Queue if necessary
      insert s in Queue
  end
```

Simple Memory-bounded A* (SMA*)

SMA* is a shortest path algorithm based on the A* algorithm.

The advantage of SMA* is that it uses a bounded memory, while the A* algorithm might need exponential memory.

All other characteristics of SMA* are inherited from A*.

How it works:

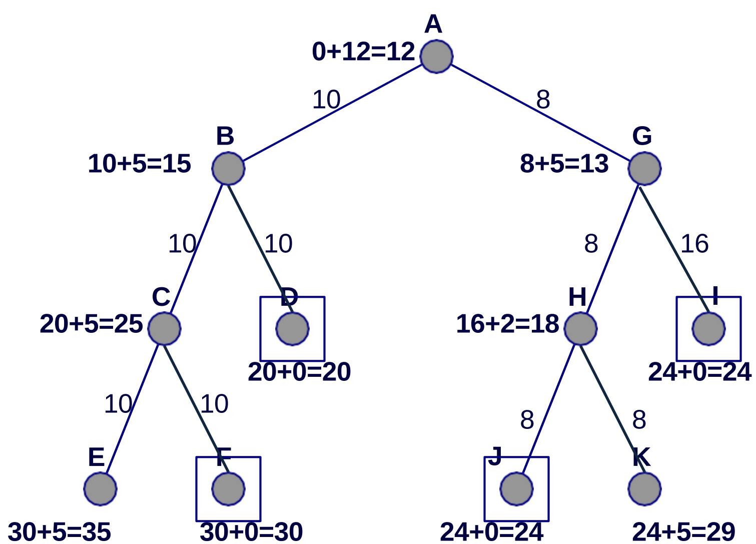
- Like A*, it expands the best leaf until memory is full.
- Drops the worst leaf node- the one with the highest f-value.
- SMA* then backs up the value of the forgotten node to its parent.

Simple Memory-bounded A* (SMA*)

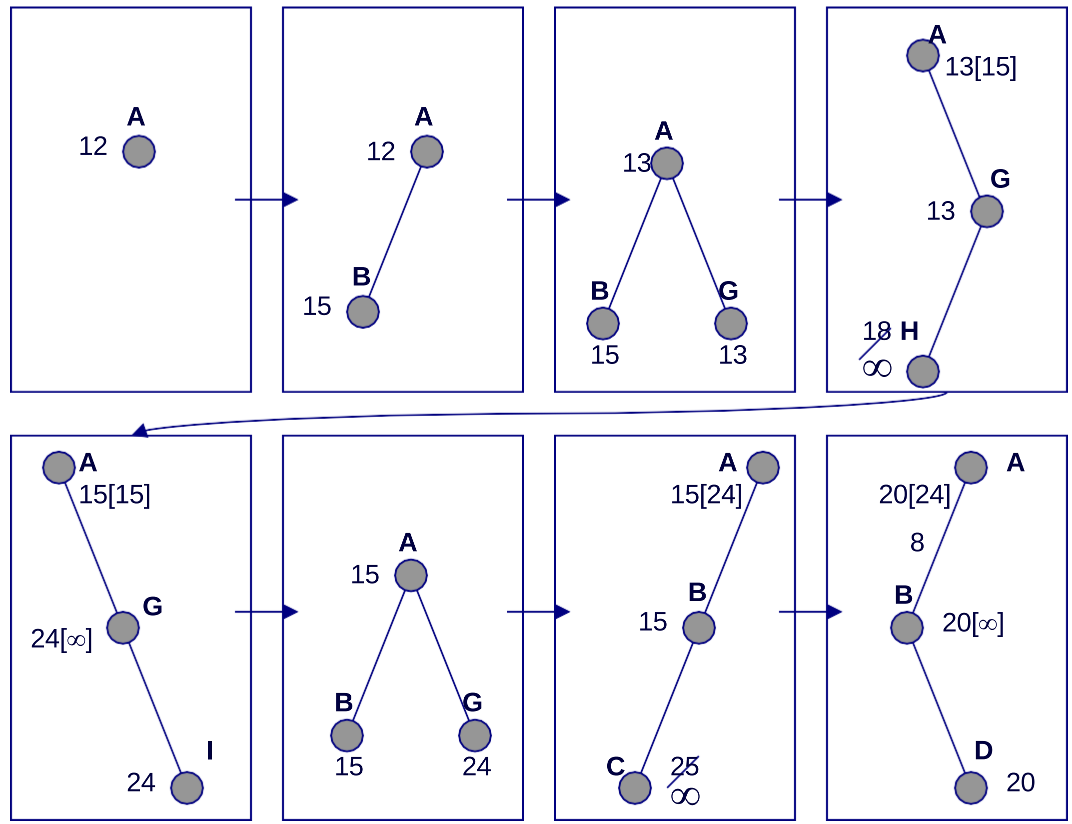
(Example with 3-node memory)

Search space

$f = g+h$ ⌋ = goal

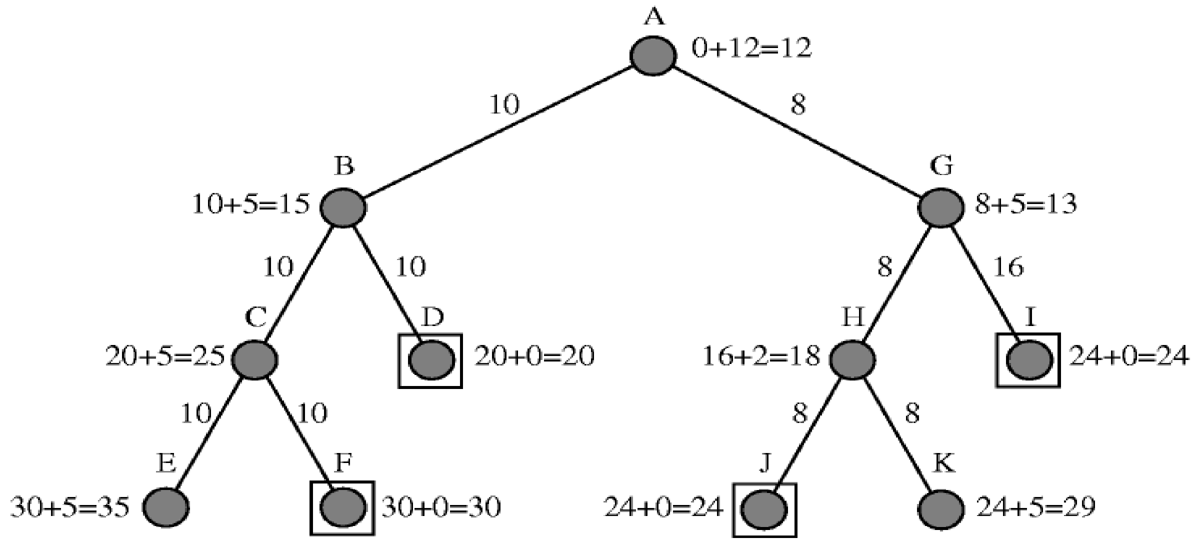


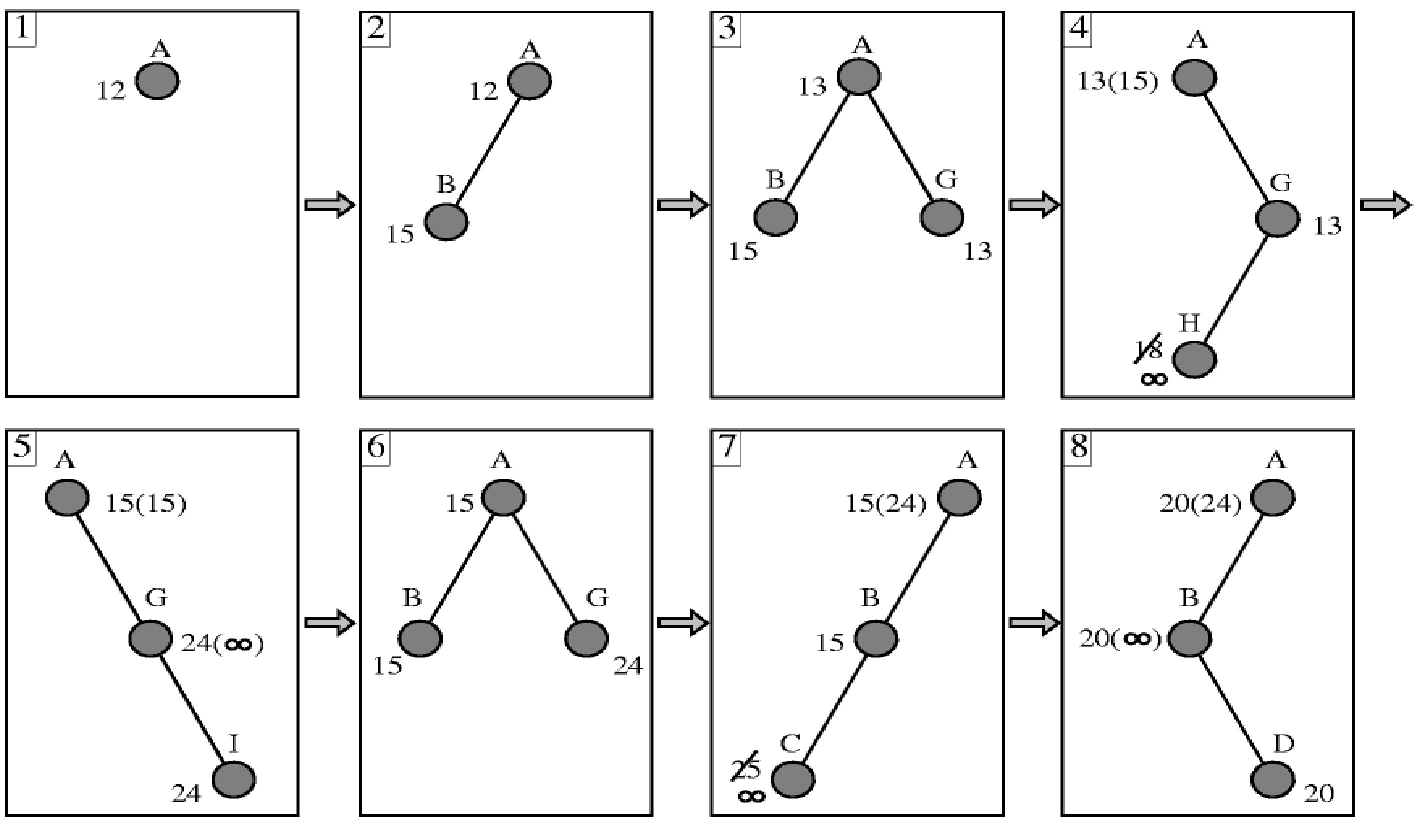
Progress of SMA*. Each node is labeled with its *current f*-cost. Values in parentheses show the value of the best forgotten descendant.



∞ is given to nodes that the path up to it uses all available memory.

Can tell when best solution found within memory constraint is optimal or not.


A
0+12=12
10
8
B
10+5=15
G
8+5=13
10
10
8
16
C
20+5=25
D
20+0=20
H
16+2=18
I
24+0=24
10
10
8
8
E
30+5=35
F
30+0=30
J
24+0=24
K
24+5=29


1
A
12
2
A
12
B
15
3
A
13
B
15
G
13
4
A
13(15)
G
13
H
~~18~~
∞
5
A
15(15)
G
24(∞)
I
24
6
A
15
B
15
G
24
7
A
15(24)
B
15
C
~~25~~
∞
8
A
20(24)
B
20(∞)
D
20

The Algorithm proceeds as follow

Let MaxNodes = 3

- Initially B and G are added to open list, then hit max.
- B is larger f value, so discard but save f(B)=15 at parent A

Add H, but f(H)=18. This is not a goal and we can never go deeper, so set f(H)=infinity and save at G.

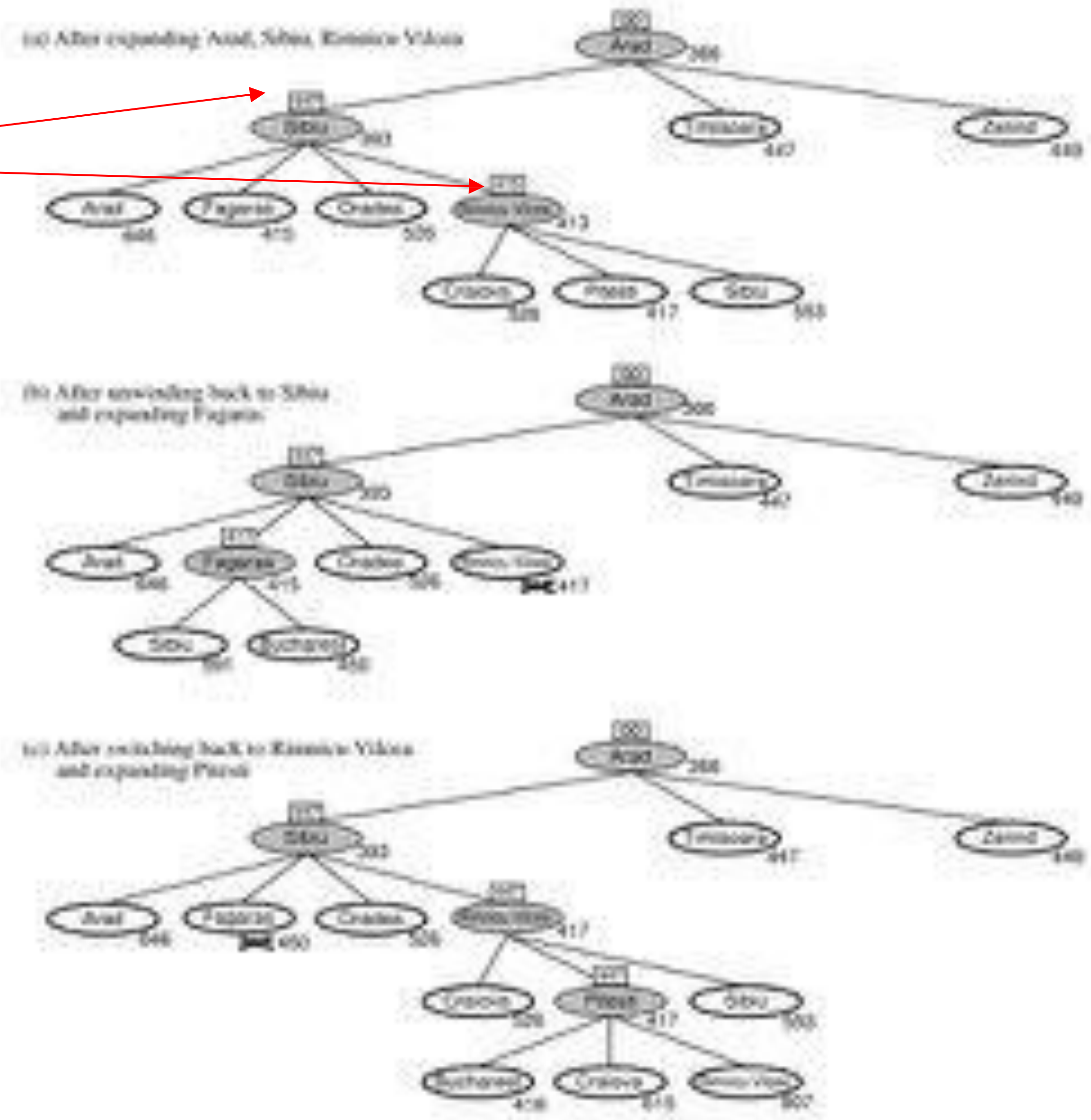
- Generate next child I with f(I)=24, bigger other child of A. Now we have seen all children of G, so reset f(G) to 24.
- Regenerate B and child C. This is not a goal so f(C) is reset to infinity.
- Generate second child D with f(D)=20, backing up value to ancestors.
- D is a goal node, so search terminates.

SMA* Properties [2]

- It works with a heuristic, just as A*
- It is complete if the allowed memory is high enough to store the shallowest solution.
- It is optimal if the allowed memory is high enough to store the shallowest optimal solution, otherwise it will return the best solution that fits in the allowed memory.
- It avoids repeated states as long as the memory bound allows it
- It will use all memory available.
- Enlarging the memory bound of the algorithm will only speed up the calculation.
- When enough memory is available to contain the entire search tree, then calculation has an optimal speed

Recursive Best First Search (RBFS)

best alternative
over fringe nodes,
which are not children:
do I want to back up?



RBFS changes its mind very often in practice.

This is because the $f=g+h$ become more accurate (less optimistic) as we approach the goal. Hence, higher level nodes have smaller f-values and will be explored first.

Problem? If we have more memory we cannot make use of it.

Ay idea to improve this?

Relaxed Problems

A problem with fewer restrictions on the actions is called a relaxed problem.

The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original 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 near square, then $h_2(n)$ gives the shortest solution.