5. Informed Search Strategies

Outline

- Best-first search
- Greedy best-first search
- A^* search
- Heuristics
- Local search algorithms
- Hill-climbing search
- Simulated annealing search

Informed Search Strategy

- Uninformed search strategies can find the solutions by systematically generating new states and testing them against the goal.
- These strategies are incredibly inefficient in most of the cases.
- Informed Search Strategies use problem specific knowledge beyond the definition of problem itself.

- Greedy Best-First Search

Introduction to Greedy Best-First Search

- Greedy Best-First Search (GBFS) is a **graph traversal algorithm** that is used in artificial intelligence for finding the shortest path between two points or solving problems with multiple possible solutions.
- It is classified as a **heuristic search** algorithm since it relies on an evaluation function to determine the next step, focusing on getting as close to the goal as quickly as possible.
- GBFS falls under the category of **informed search algorithms**, where the decision-making process involves knowledge about the problem to guide the search process.

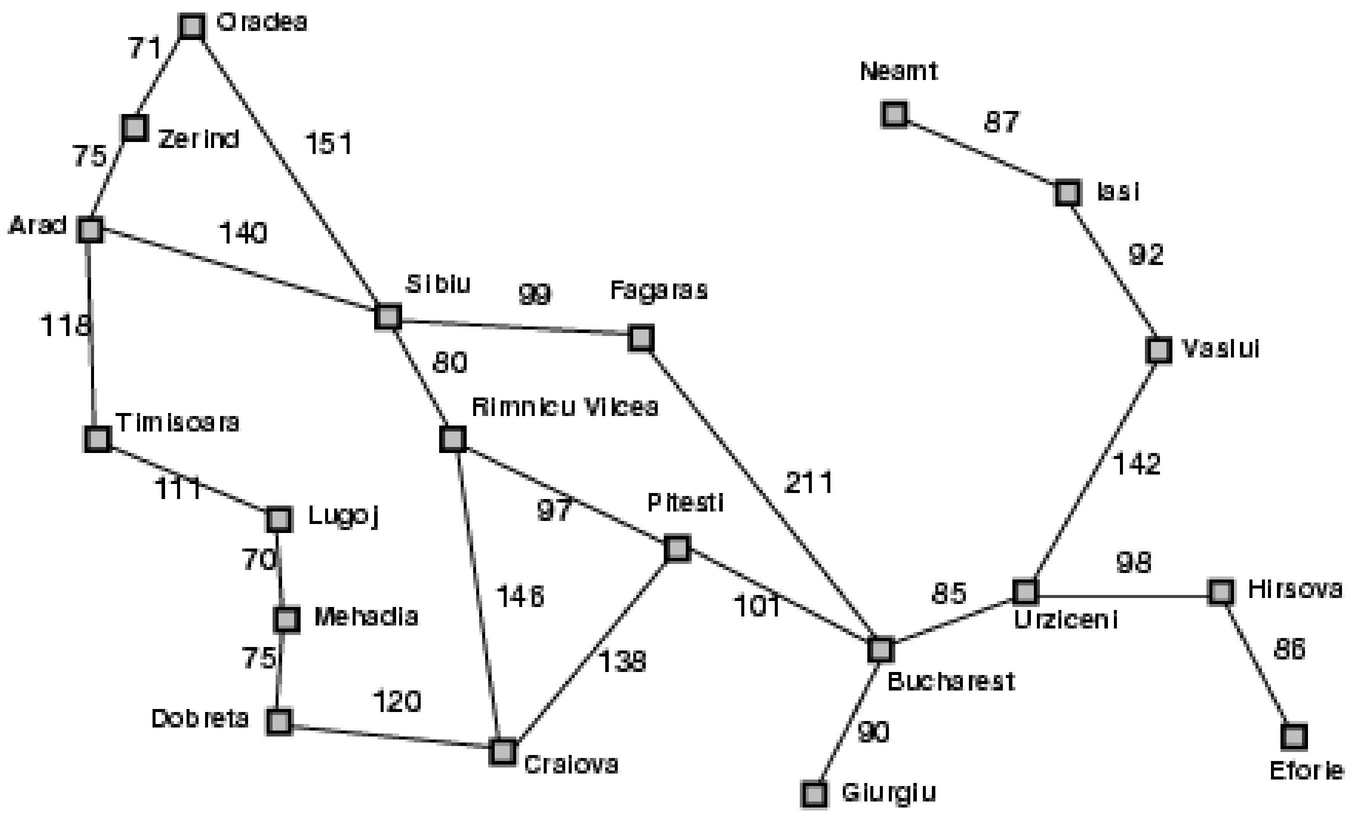
How Greedy Best-First Search Works?

- Greedy Best-First Search works by **evaluating the cost of each possible path** and then expanding the path with the lowest cost. This process is repeated until the goal is reached.
- The algorithm uses a **heuristic function to determine which path is** the most promising.
- The heuristic function takes into account the cost of the current path and the estimated cost of the remaining paths.
- If the cost of the current path is lower than the estimated cost of the remaining paths, then the current path is chosen. This process is repeated until the goal is reached.
- The algorithm does not backtrack or reconsider nodes that seem less optimal, making it fast but potentially incomplete if the path it chooses does not lead to a solution.

Greedy search algorithm may not always produce an optimal solution, but the solution will be locally optimal,as it will be generated in comparatively less amount of time.

Romania with step costs in km

Consider a problem where we need to travel from **Arad** to **Bucharest** in Romania. The cities and their step costs (distances in km) are given in the map below: At each stage visit an unvisited city nearest to the current city



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 (GBFS) is an informed search algorithm that expands the node that appears to be closest to the goal based on a heuristic function.

It uses a heuristic **h(n)**, which estimates the cost from the current node to the goal.

However, it does not consider the cost accumulated from the start node, which can sometimes lead to inefficient paths.

Heuristic Function:

$$f(n) = h(n)$$

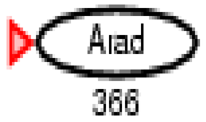
where:

- $h(n)$ is the estimated cost from node n to the goal.

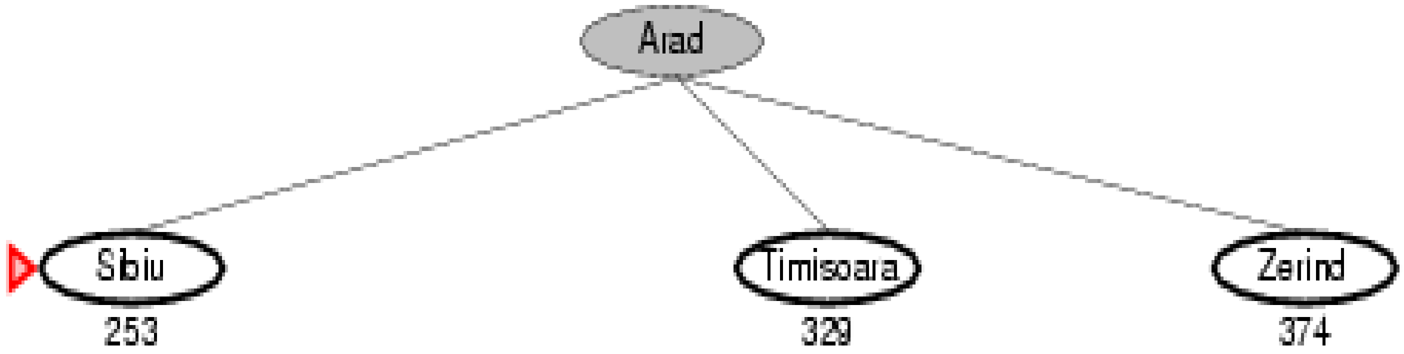
Greedy Best-First Search

- Evaluation function $f(n) = h(n)$ (heuristic)
- $f(n)$ = estimate of cost from n to *goal*
  $h_{SLD}(n)$ = straight-line distance from n to Bucharest
  Greedy best-first search expands the node that appears to be closest to goal

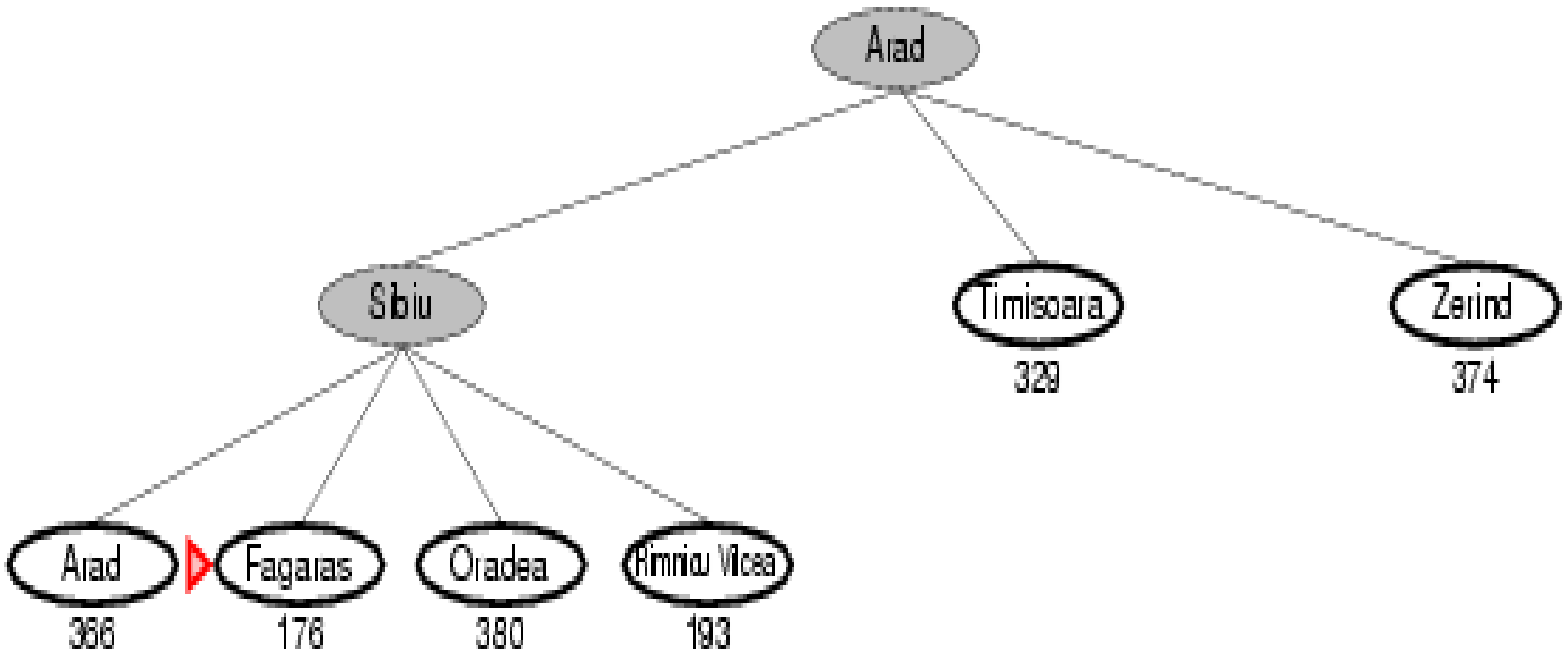
Greedy Best-First Search Example



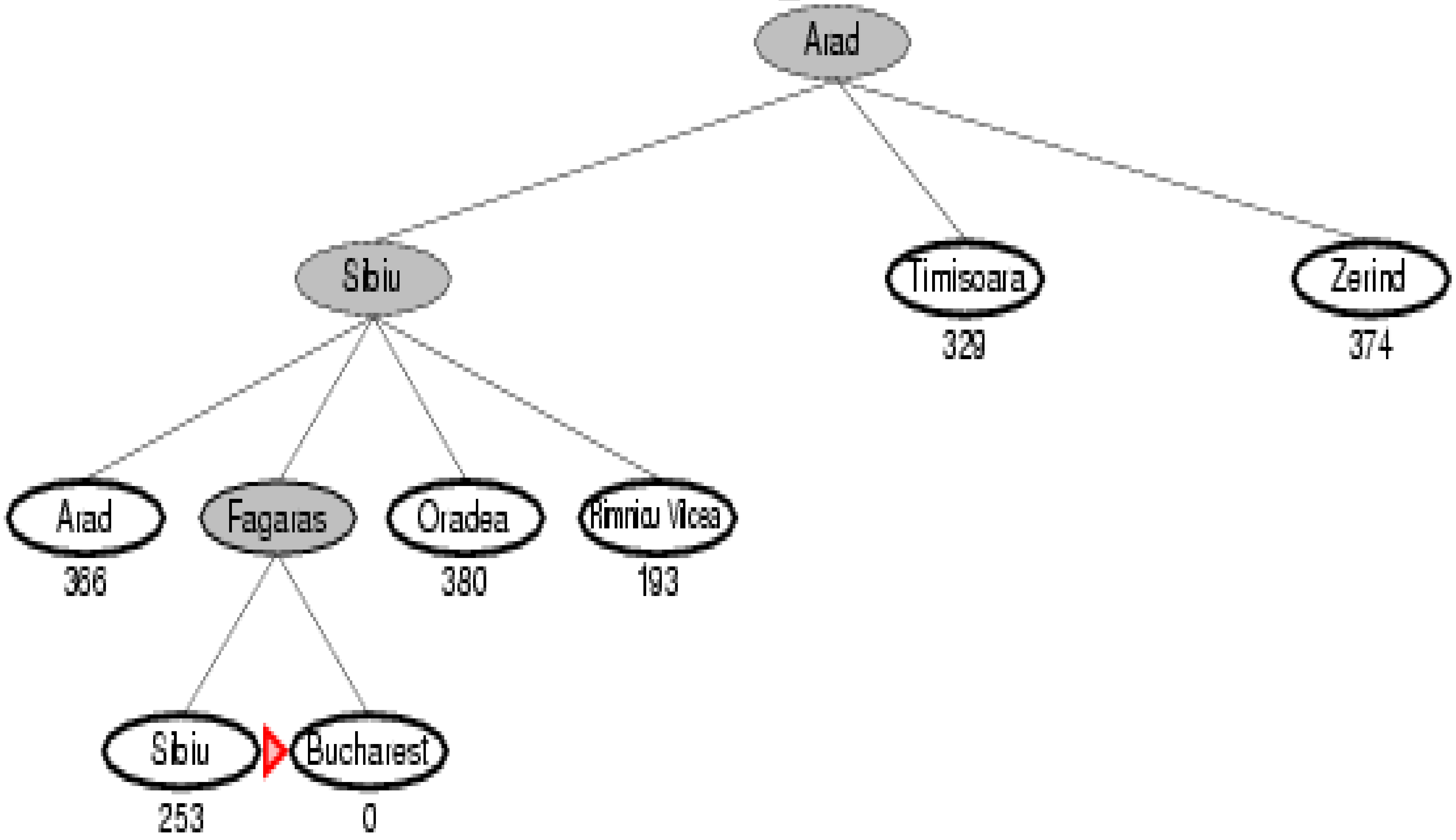
Greedy Best-First Search Example



Greedy Best-First Search Example



Greedy Best-First Search Example



Properties of Greedy Best-First Search

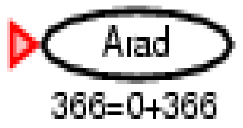
- Complete? No – can get stuck in loops, e.g., Iasi □ Neamt □ Iasi □ Neamt □
- Time? $O(b^m)$, but a good heuristic can give dramatic improvement
- Space? $O(b^m)$ -- keeps all nodes in memory
- Optimal? No

- A* Search

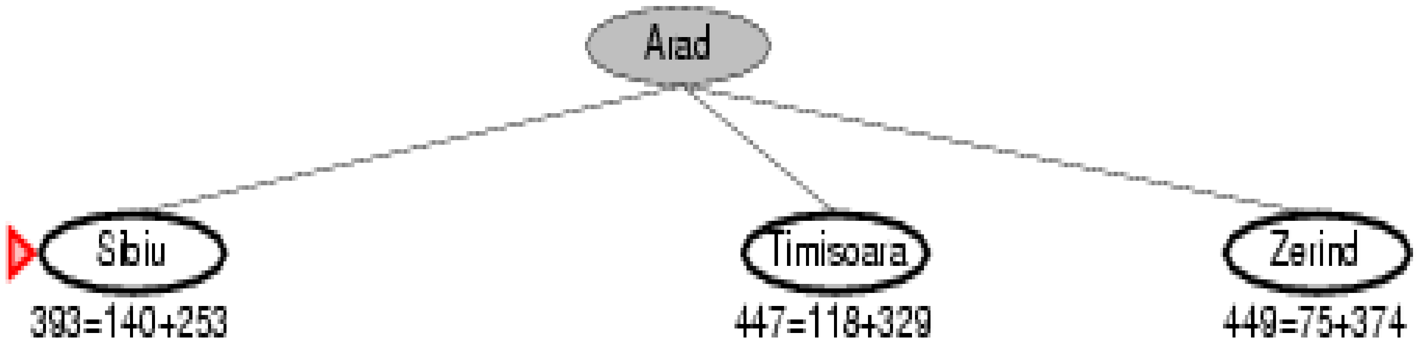
A* search

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

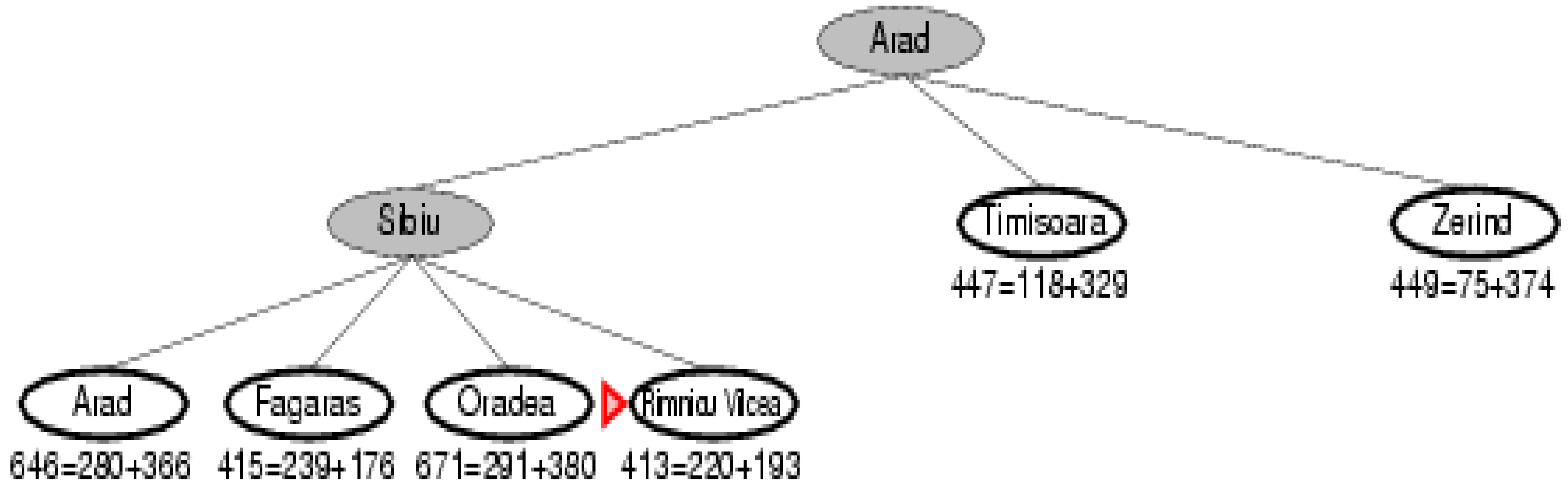
A* search example



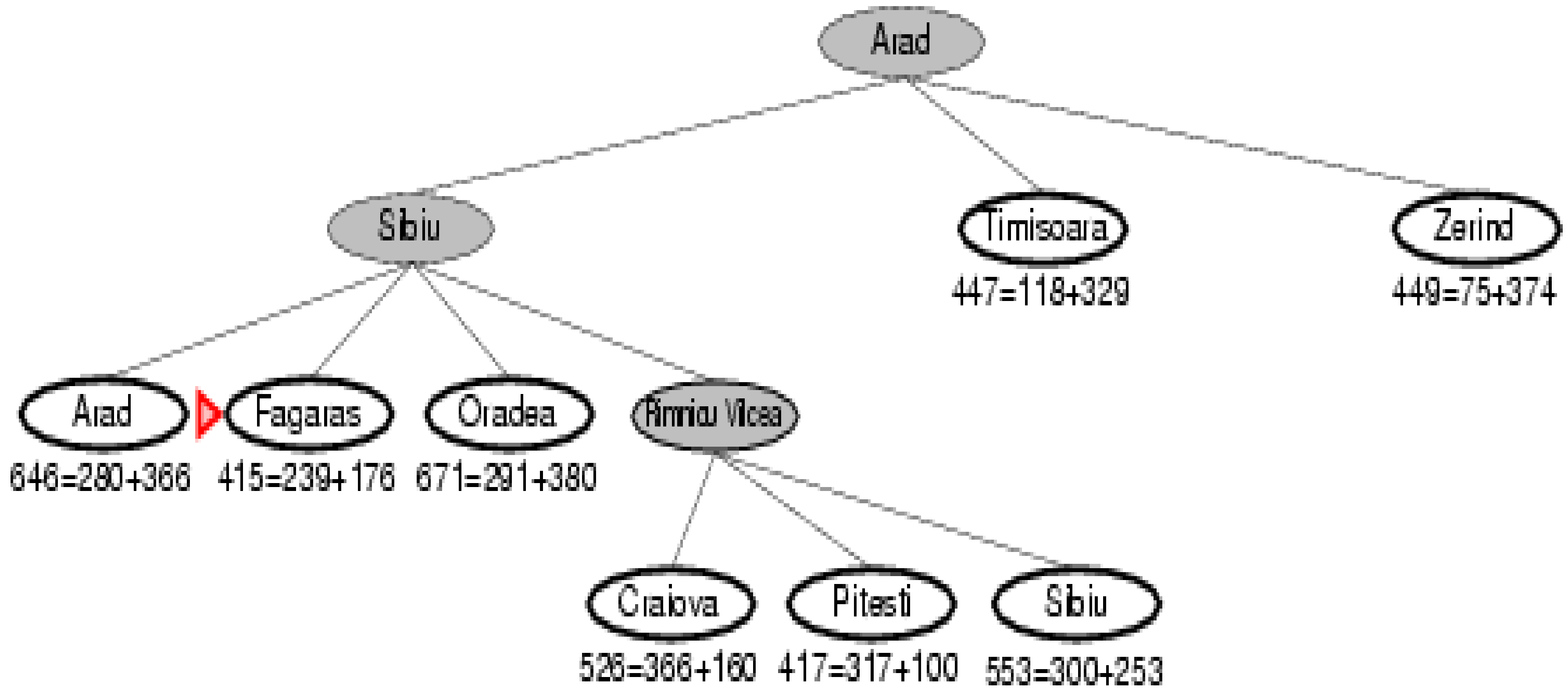
A* search example



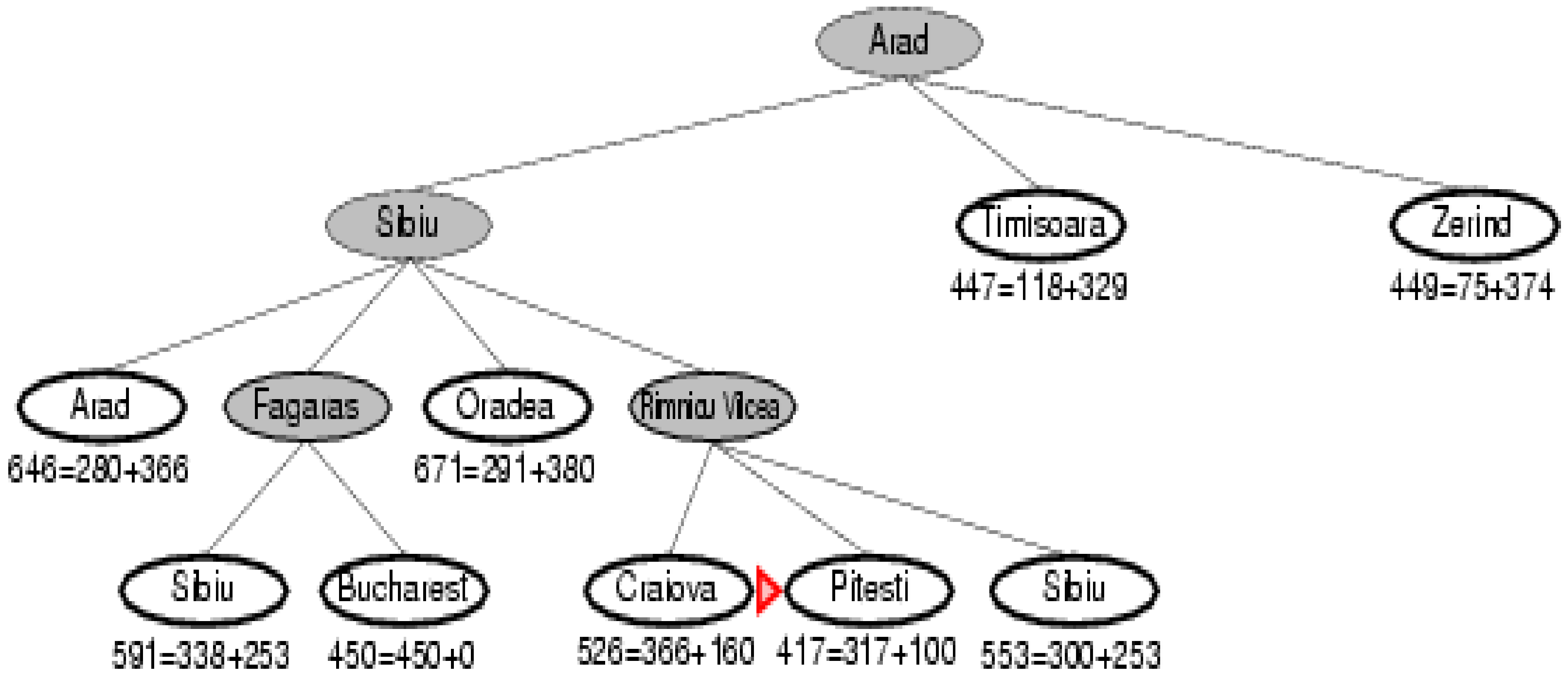
A* search example



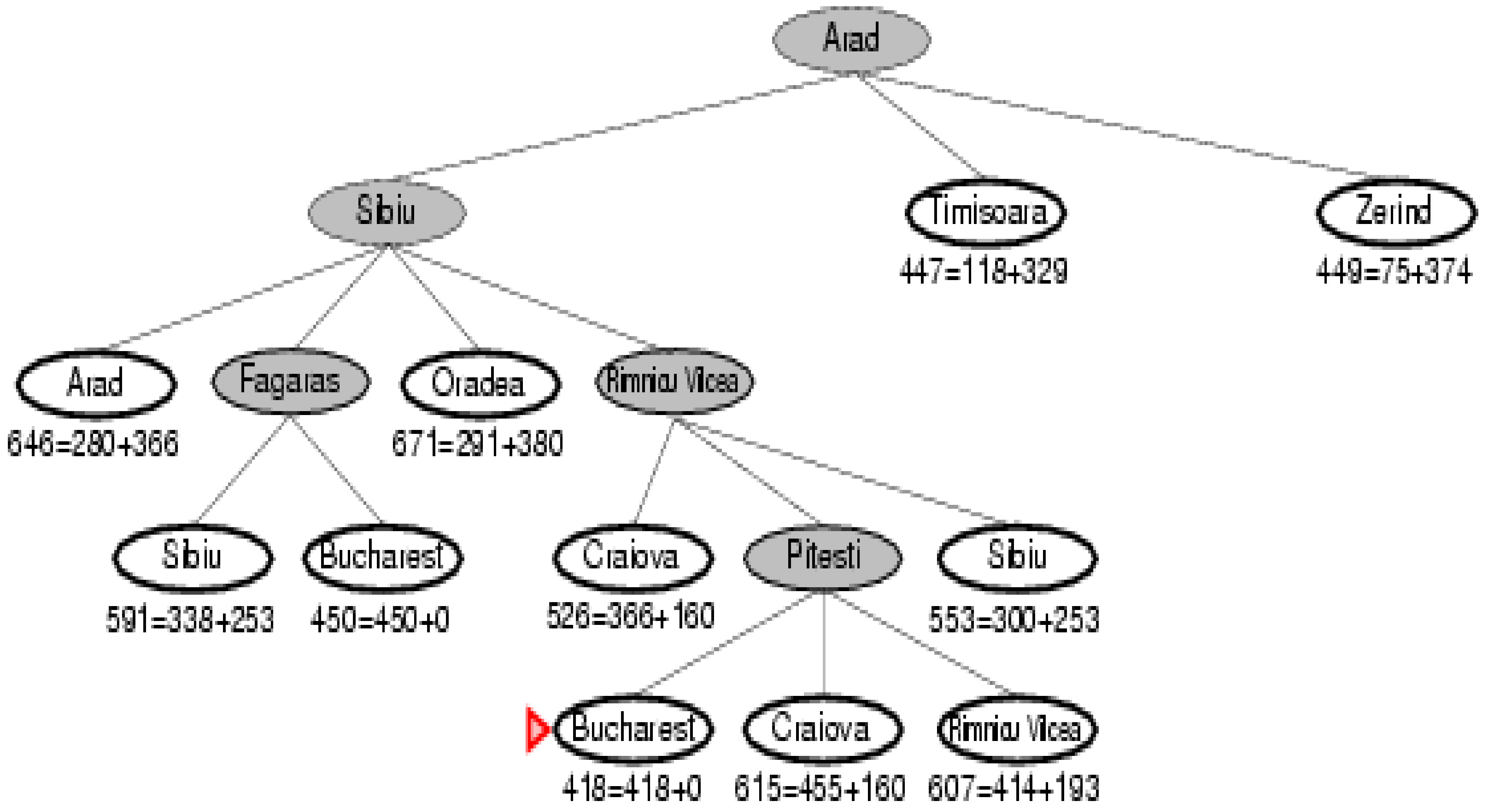
A* search example



A* search example



A* search example



Admissible Heuristics

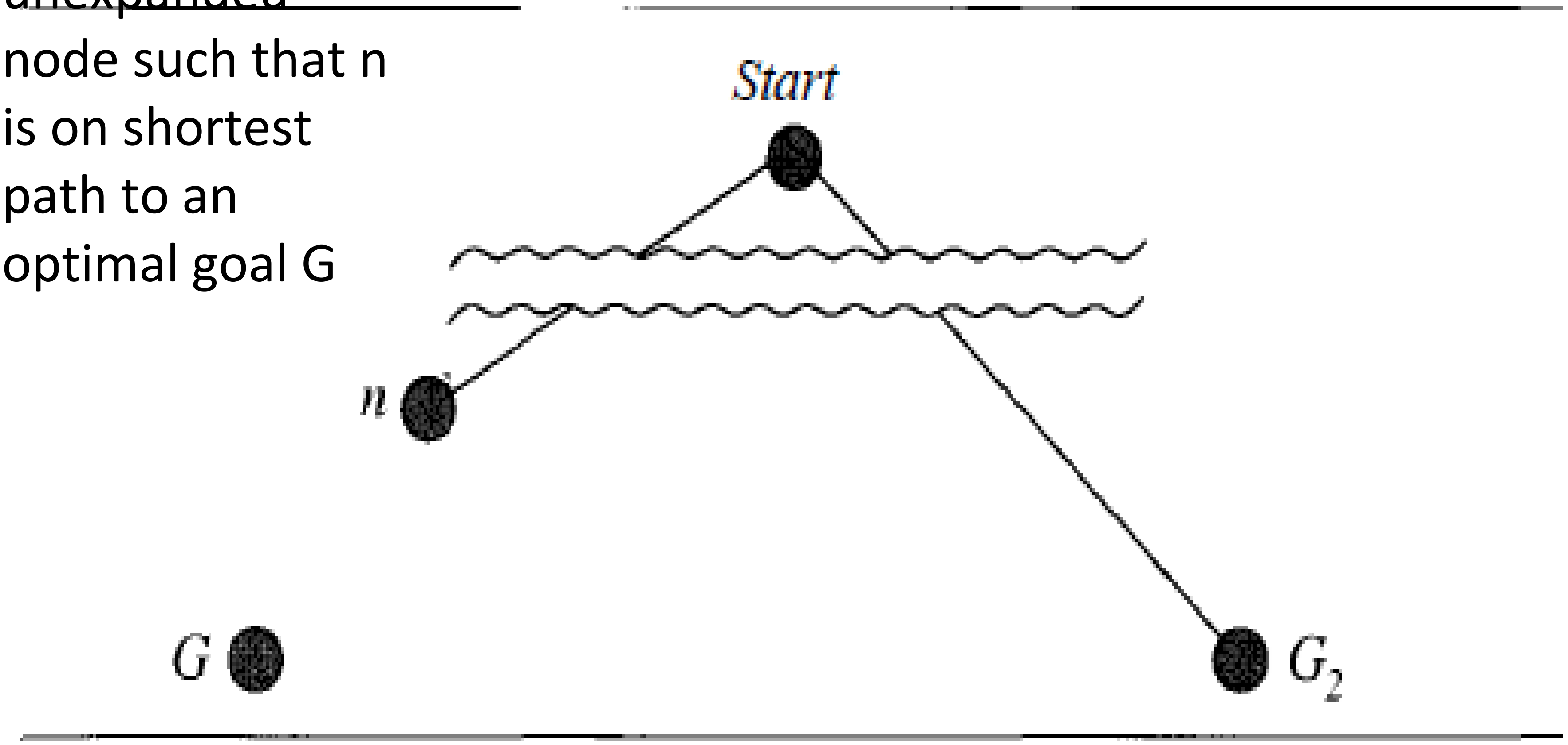
- A heuristic *h(n)* is admissible if for every node *n*, *h(n)* ≤ *h*(n)*, where *h*(n)* is the true cost to reach the goal state from *n*.
- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic
- Example: *hSLD(n)* (never overestimates the actual road distance)

Optimality of A* proof

- Let G be an optimal goal state, with path cost f*.
- Let G2 be a suboptimal goal state : a goal state with path cost g(G2) >f*.
- The situation we imagine is that A* has selected G2 from the queue. Because G2 is a goal state, this would terminate the search with a suboptimal solution

Optimality of A* proof…

n be a unexpanded node such that n is on shortest path to an optimal goal G



Optimality of A* proof..

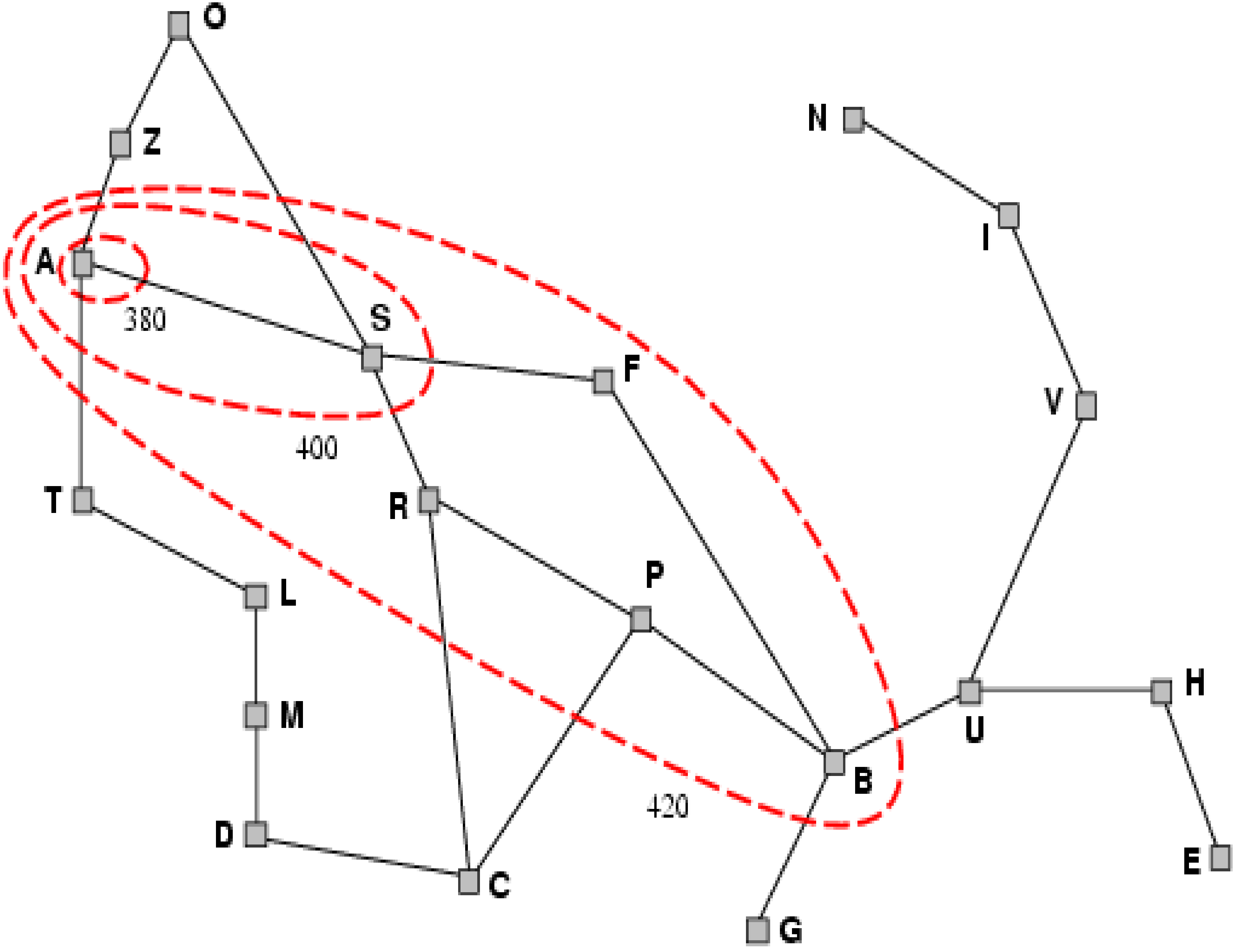
- Consider a node *n* that is currently a leaf node on an optimal path to *G*
- Because *h* is admissible, we must have
- f* > f(n)
- Further more *n* is not chosen for expansion over G2, we must have
- f(n) > f(G2)
- Therefore f* >f(G2)

Optimality of A* proof..

- But G2 is a goal state we have h(n) =0
- Hence f(G2)=g(G2)
- Thus we proved that f* > g(G2)
- This contradicts the assumption that G2 is suboptimal.
- A* never selects a suboptimal goal for expansion.
- A* is a Optimal algorithm.

Contours

- If *f* never decreases along any path out from the root, we can conceptually draw **Contours** in the state space.
- Inside the Contour labeled 400, all nodes have *f(n)* less than or equal to 400
- A* expands the leaf node of lowest *f*, we can see that an A* search fans out from start node, adding nodes in concentric bands of increasing *f-Cost.*

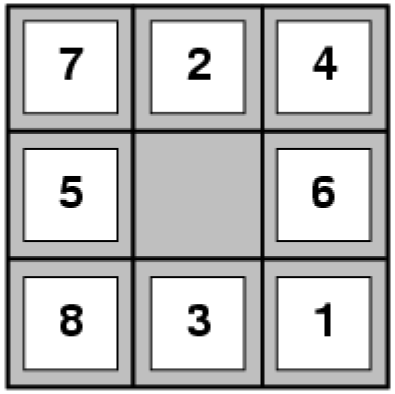

O
Z
A
S
F
T
R
L
P
M
D
C
B
G
U
H
E
N
I
V
380
400
420

Properties of A*

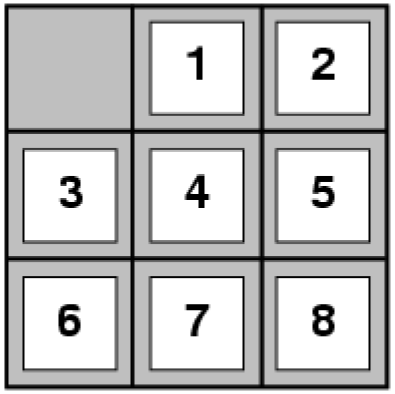
- <u>Complete?</u> Yes (unless there are infinitely many nodes)
- <u>Time?</u> Exponential
- <u>Space?</u> Keeps all nodes in memory
- <u>Optimal?</u> Yes

Admissible Heuristics

- 8-puzzle:
  - *h1(n)* = number of misplaced tiles
  - *h2(n)* = total city block distance /Manhattan distance
  - (i.e., no. of squares from desired location of each tile)
    h1(S) = ? 8
  - h2(S) = ? 3+1+2+2+2+3+3+2 = 18



Start State



Goal State

Simple Hill Climbing

- Use heuristic to move only to states that are *better* than the current state.
- Always move to better state when possible.
- The process ends when all operators have been applied and none of the resulting states are better than the current state.
- Always climb uphill whenever you can, without looking around.
- Climbing a mountain in the dark - test one direction and if it is uphill take a step in that direction. If you find that there is no uphill move in any direction you are done (at the top of the mountain).

Simple Hill Climbing Algorithm

Simple Hill Climbing is a straightforward variant of hill climbing where the algorithm evaluates each neighboring node one by one and selects the first node that offers an improvement over the current one.

Algorithm for Simple Hill Climbing

1. Evaluate the initial state. If it is a goal state, return success.
2. Make the initial state the current state.
3. Loop until a solution is found or no operators can be applied:
    - Select a new state that has not yet been applied to the current state.
    - Evaluate the new state.
    - If the new state is the goal, return success.
    - If the new state improves upon the current state, make it the current state and continue.
    - If it doesn’t improve, continue searching neighboring states.
4. Exit the function if no better state is found.

- Simple with no complex data structures used (efficient on memory)
- No backtracking
- Unselected nodes during the search are immediately discarded
- It performs a local search and chooses the first state that is better than the current state
- Does not guarantee to find a solution (goal)
- Does not guarantee to find the optimal path
- May get trapped by a local maxima or a local minima
- Works well when used with a precise and accurate heuristic function
- Works well when the problem has different paths to different goals

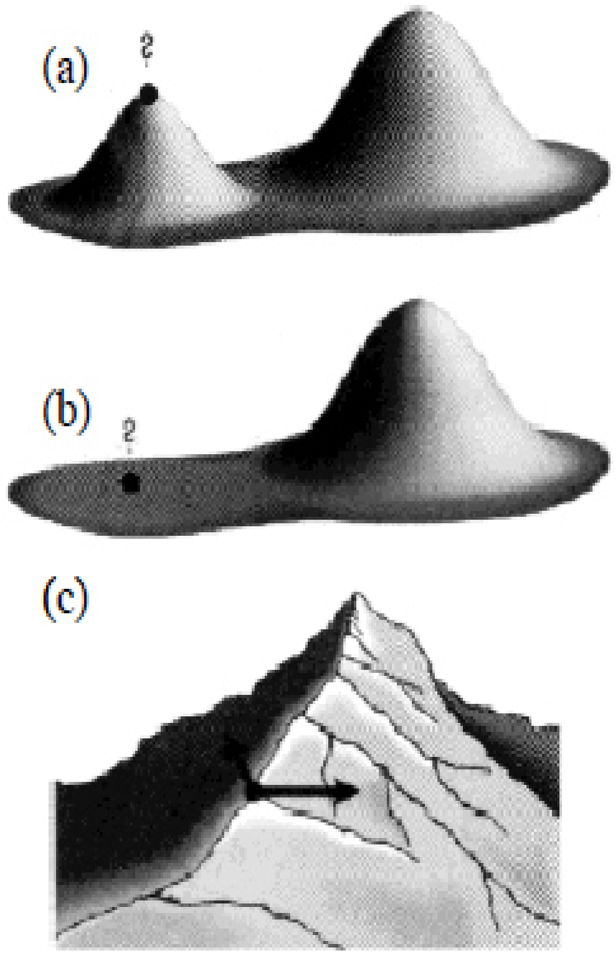
Hill Climbing

This simple policy has three well-known drawbacks:

1. **Local Maxima**: a local maximum as opposed to global maximum.

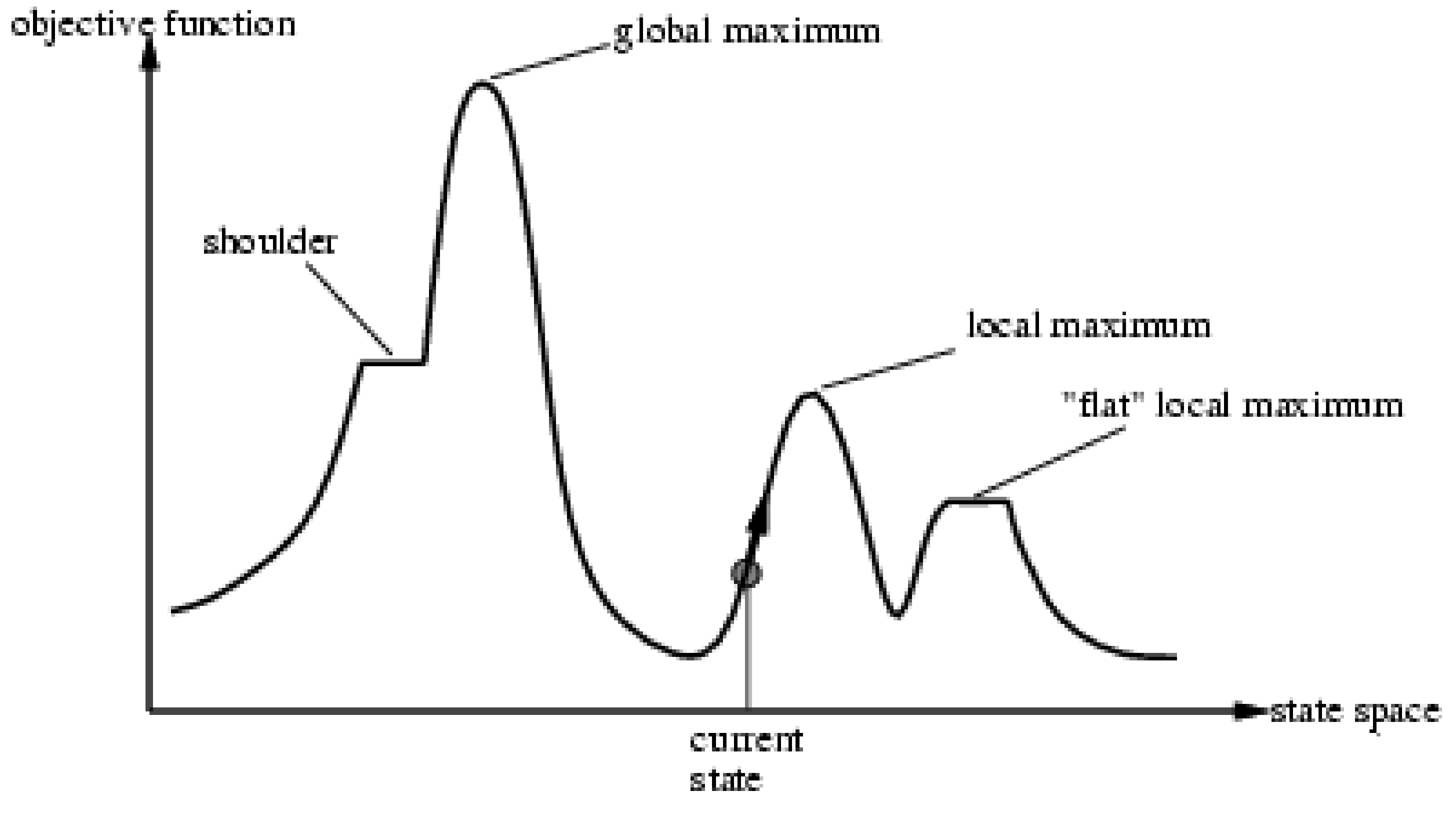
2. **Plateaus**: An area of the search space where evaluation function is flat, thus requiring random walk.

3. **Ridge**: Where there are steep slopes and the search direction is not towards the top but towards the side.



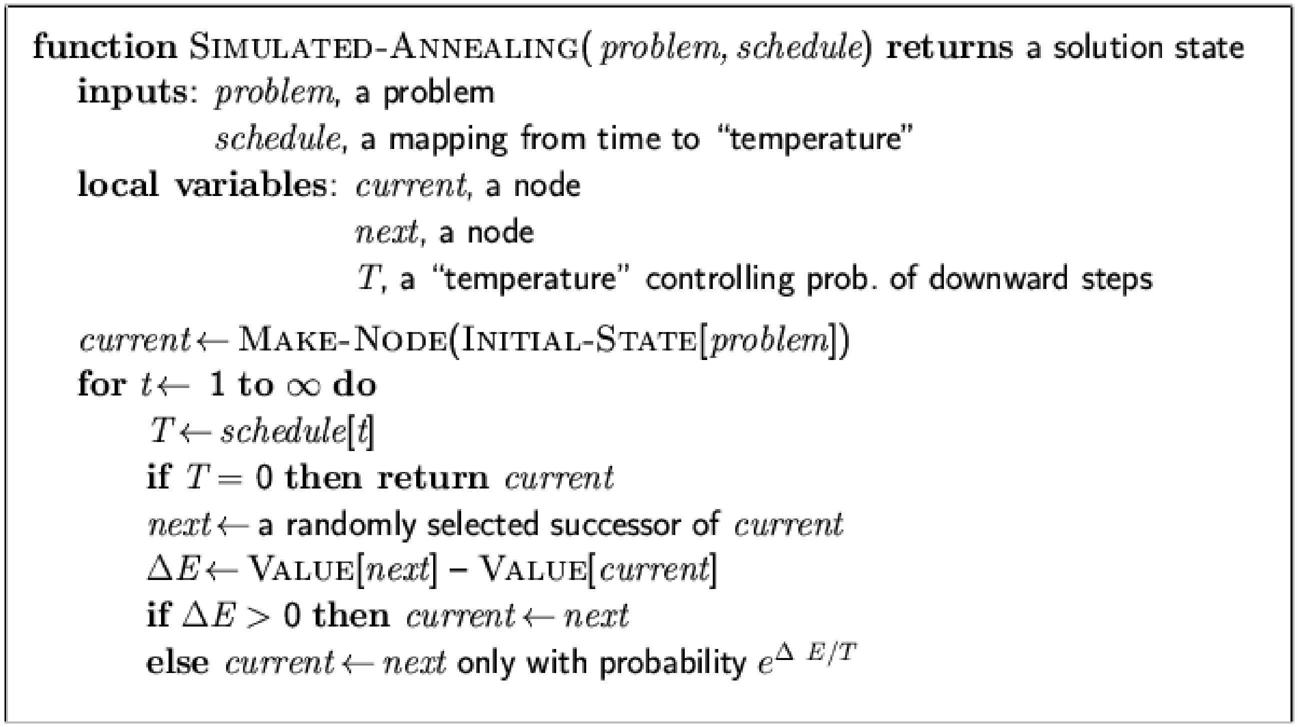
Hill-climbing search

- Problem: depending on initial state, can get stuck in local maxima



Simulated annealing search

- Idea: escape local maxima by allowing some "bad" moves but gradually decrease their frequency



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     T, a "temperature" controlling prob. of downward steps

    current ← MAKE-NODE(INITIAL-STATE[problem])
    for t ← 1 to ∞ do
        T ← schedule[t]
        if T = 0 then return current
        next ← a randomly selected successor of current
        $\Delta E$ ← VALUE[next] – VALUE[current]
        if $\Delta E > 0$ then current ← next
        else current ← next only with probability $e^{\Delta E/T}$
```

Properties of simulated annealing search

- One can prove: If *T* decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1
- Widely used in VLSI layout, airline scheduling, etc