Informed Search Algorithm (Chapter 3)

Dr. Nasima Begum
Assistant Professor
Dept. of CSE, UAP

Material

- Chapter 3 Section 3.5~
- Exclude memory-bounded heuristic search

Outline

- Heuristics
- Best-first Search
- Greedy Best-first Search
- A^* Search
- Local Search Algorithms
- Hill-climbing Search
- Simulated Annealing Search
- Local Beam Search
- Genetic Algorithms (GA)

Review: Tree Search

- \input{\file{algorithms}{tree-search-short-algorithm}}
- A search strategy is defined by picking the order of node expansion

Review: Uninformed/Informed Search

- Uninformed search algorithms looked through search space for all possible solutions of the problem without having any additional knowledge about search space.

- On the other hand, informed search algorithm contains an array of knowledge such as <u>how far we are from the goal</u>, <u>path cost</u>, <u>how to reach to goal node</u>, etc.

- This knowledge help agents to explore less to the search space and find more efficiently the goal node.

- The informed search algorithm is more useful for large search space.

- Informed search algorithm uses the idea of heuristic, so it is also called Heuristic Search.

Heuristics Function

- Heuristic is a function which is used in Informed Search, and **it finds the most promising path**.

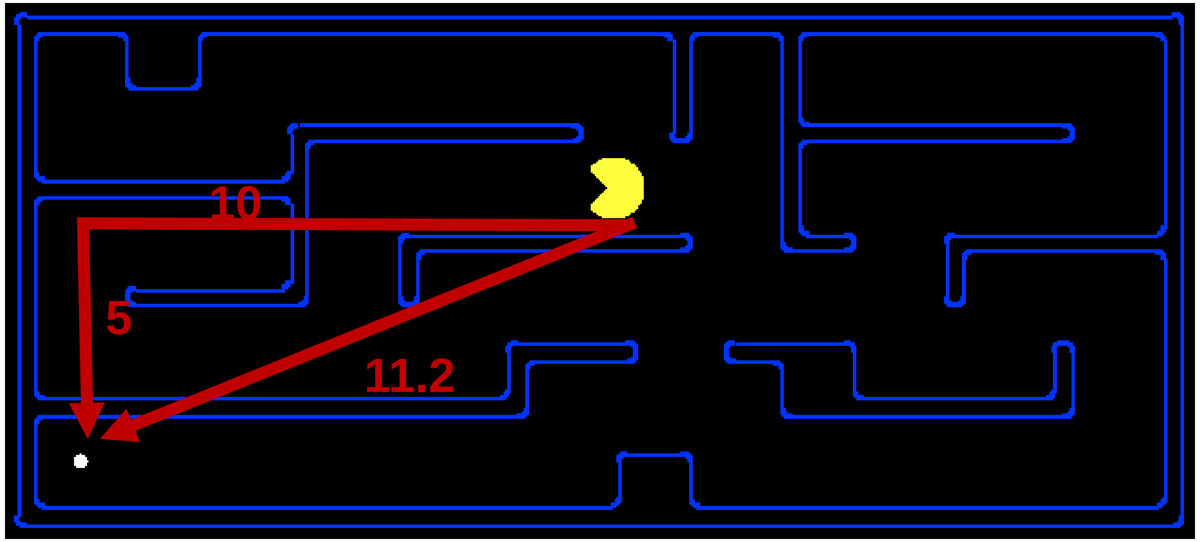
- It takes the current state of the agent as its input and produces the estimation of **how close the agent** is to the goal.

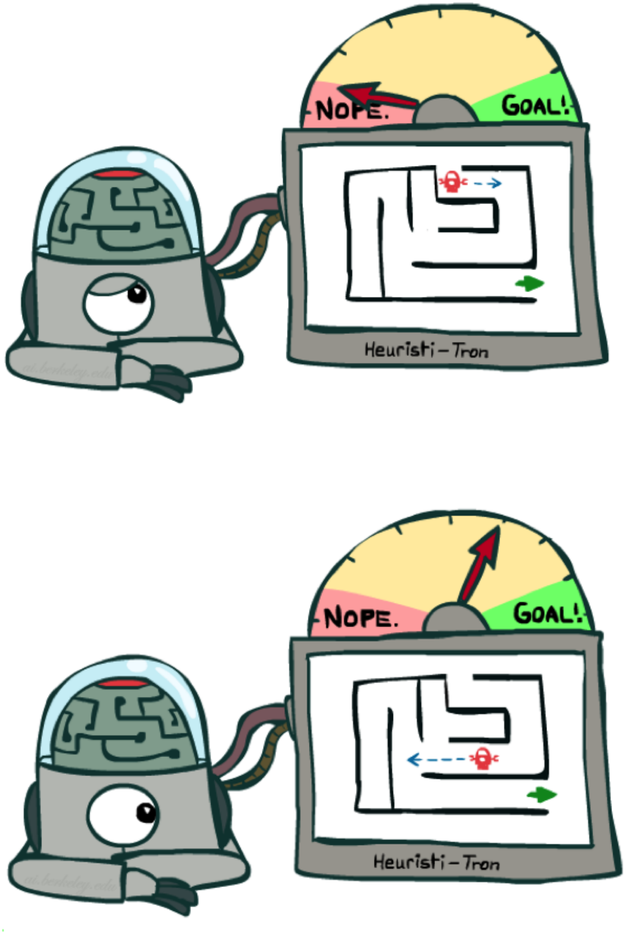
- Heuristic function estimates how close a state is to the goal state.

- It is represented by **h(n)**, and it calculates the cost of an optimal path between the pair of states. The value of the heuristic function is always positive.

Search Heuristics

- A heuristic is:
  - A function that *estimates* how close a state is to a goal
  - Designed for a particular search problem
  - Examples: Manhattan distance, Euclidean distance for pathing





Heuristics Function

- Search Heuristics: In an informed search, a heuristic is a function that estimates how close a state is to the goal state.

- For examples – Manhattan distance, Euclidean distance, etc. (lesser the distance, closer the goal).

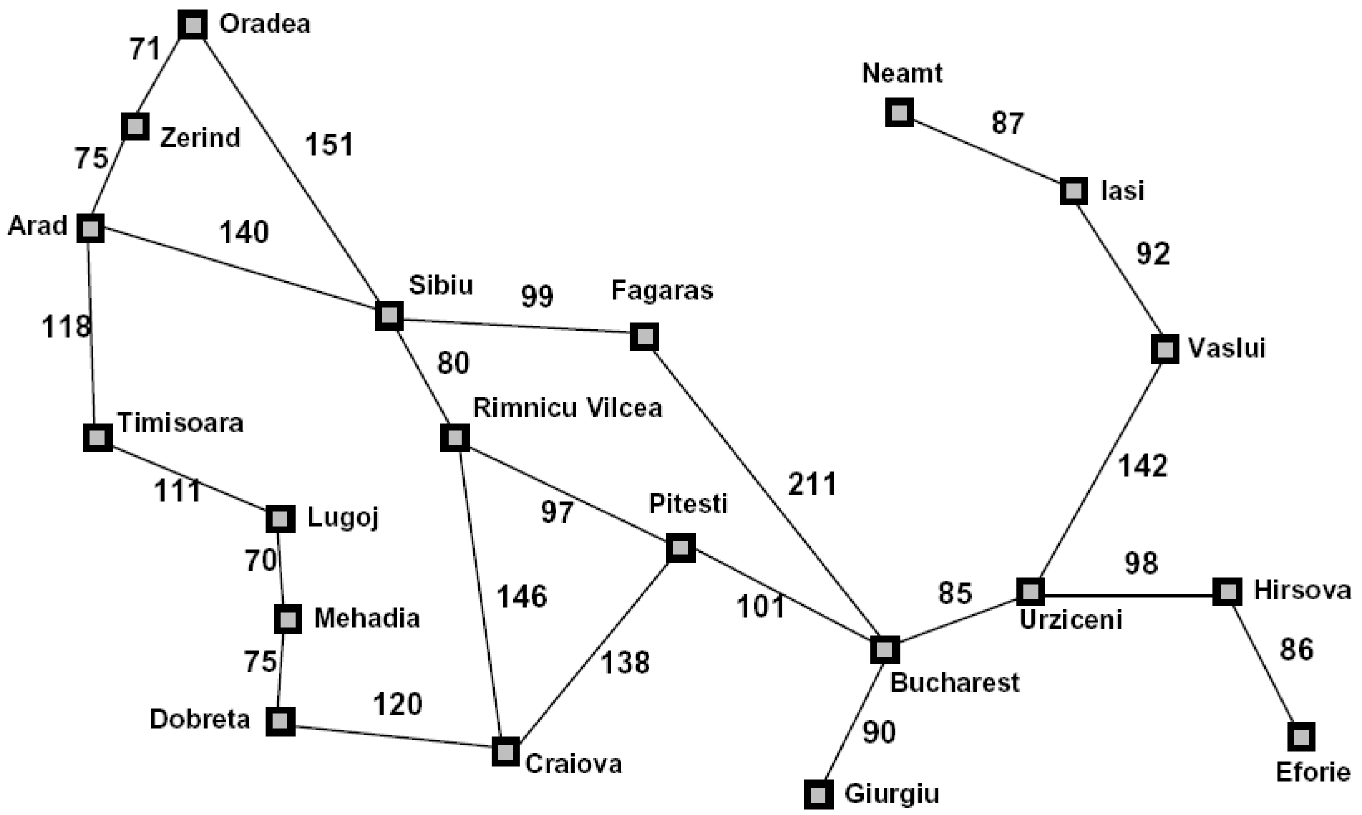
- Admissibility of the heuristic function is given as: $\mathbf{0 <= h(n) <= h^*(n)}$

- Here **h(n) is heuristic cost**, and **h*(n) is the estimated cost**. Hence heuristic cost should be less than or equal to the estimated cost (details later).

Pure Heuristic Search

- Is the simplest form of heuristic search algorithms.
- It expands nodes based on their heuristic value h(n).
- It maintains two lists, OPEN and CLOSED list.
- In the CLOSED list, it places those nodes which have already expanded and in the OPEN list, it places nodes which have yet not been expanded.
- On each iteration, each node **n** with the **lowest heuristic** value is expanded and generates all its successors and **n is placed to the closed list**. The algorithm continues unit a goal state is found.

Example: Heuristic Function



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

h(x)

Best-first Search

- **Idea:** Use an evaluation function *f(n)* for each node
  - *f(n)* provides an estimate for the total cost
  - Expand most desirable unexpanded node first
  - Expand the node n with smallest f(n)
  - Consider the lowest path cost

- Implementation:

  Order the nodes in fringe in decreasing order of desirability (priority queue)

- Special Cases:
  - Greedy Best-first Search
  - A^* Search

Greedy Best-first Search

- Always selects the path which appears best at that moment.
- It uses the heuristic function and search and totally ignores the path cost.
- At each step, we can choose the most promising node.
- It expand the node which is closest to the goal node and the closest cost is estimated by heuristic function,

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

- Where, $h(n)$= estimated cost from node n to the goal.
- The greedy best first algorithm is implemented by the priority queue.

Greedy Best-first Search

- Evaluation function $f(n) = h(n)$ (heuristic)
  = estimate of cost from node n to *goal*

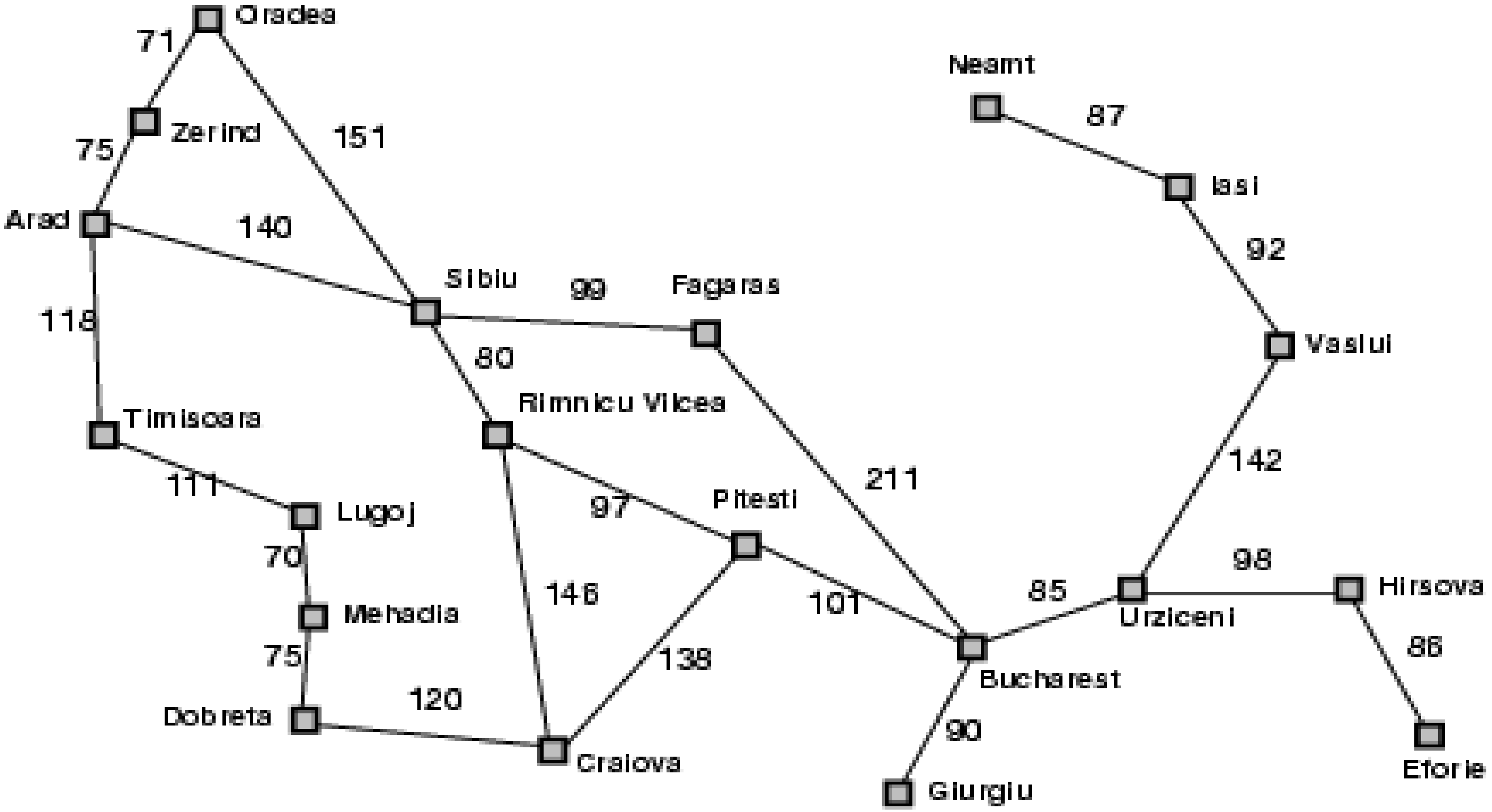
- E.g., $h_{SLD}(n)$ = straight-line distance from n to Bucharest

- Greedy best-first search expands the node that appears to be closest to goal.

Algorithm of Best-first Search

- Step 1: Place the starting node into the OPEN list.
- Step 2: If the OPEN list is empty, Stop and return failure.
- Step 3: Remove the node n, from the OPEN list which has the lowest value of h(n), and places it in the CLOSED list.
- Step 4: Expand the node n, and generate the successors of node n.
- Step 5: Check each successor of node n, and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.
- Step 6: For each successor node, algorithm checks for evaluation function f(n), and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both list, then add it to the OPEN list.
- Step 7: Return to Step 2.

Romania with Step Costs in km

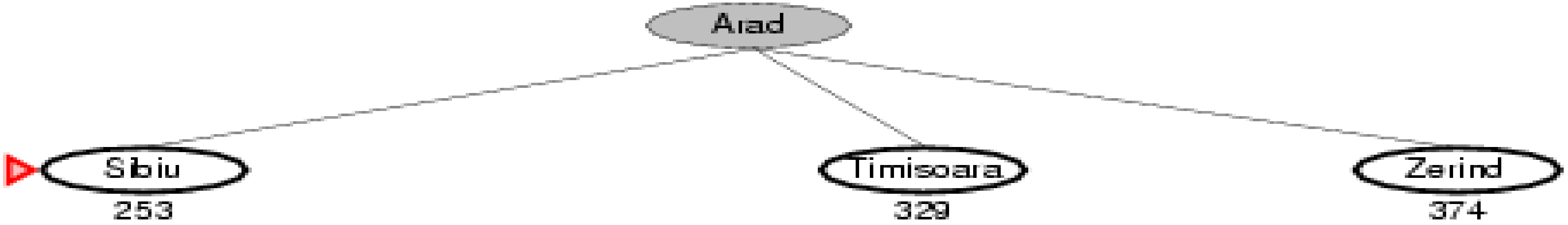


h(x)

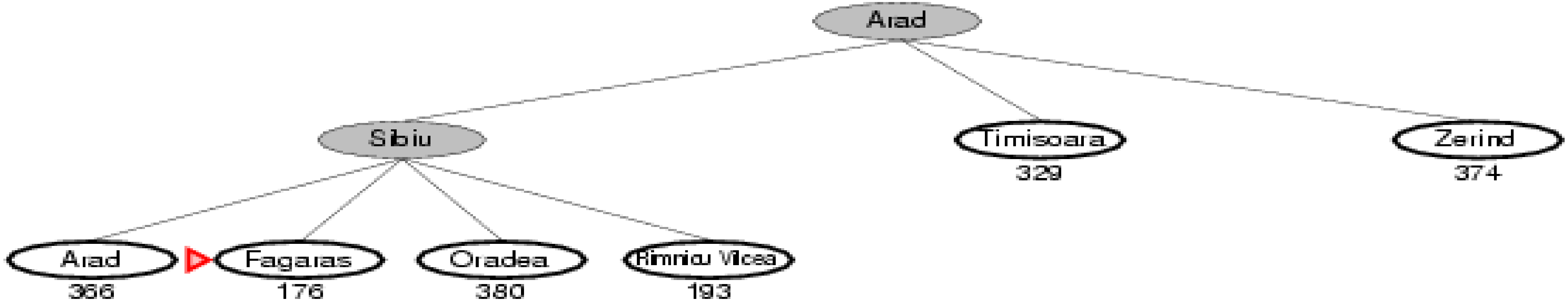
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Example: Greedy Best-first Search

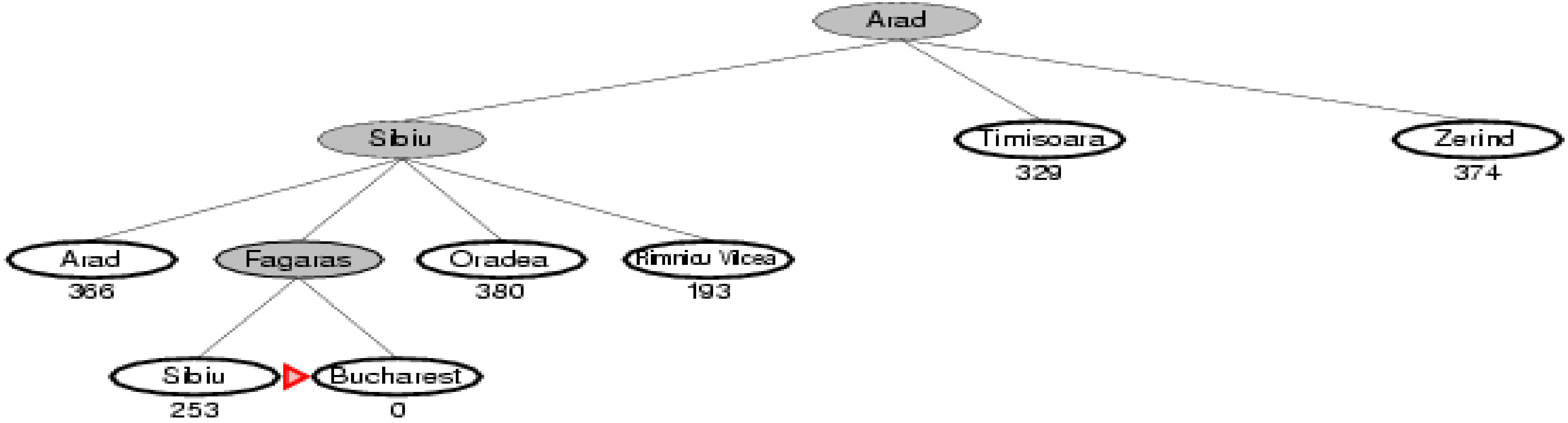
Example: Greedy Best-first Search



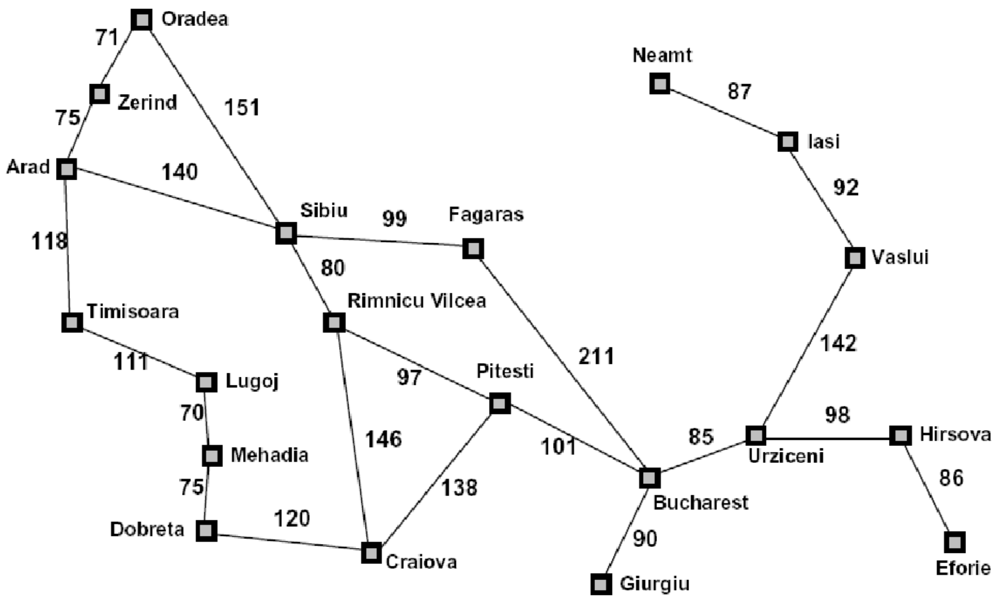
Example: Greedy Best-first Search



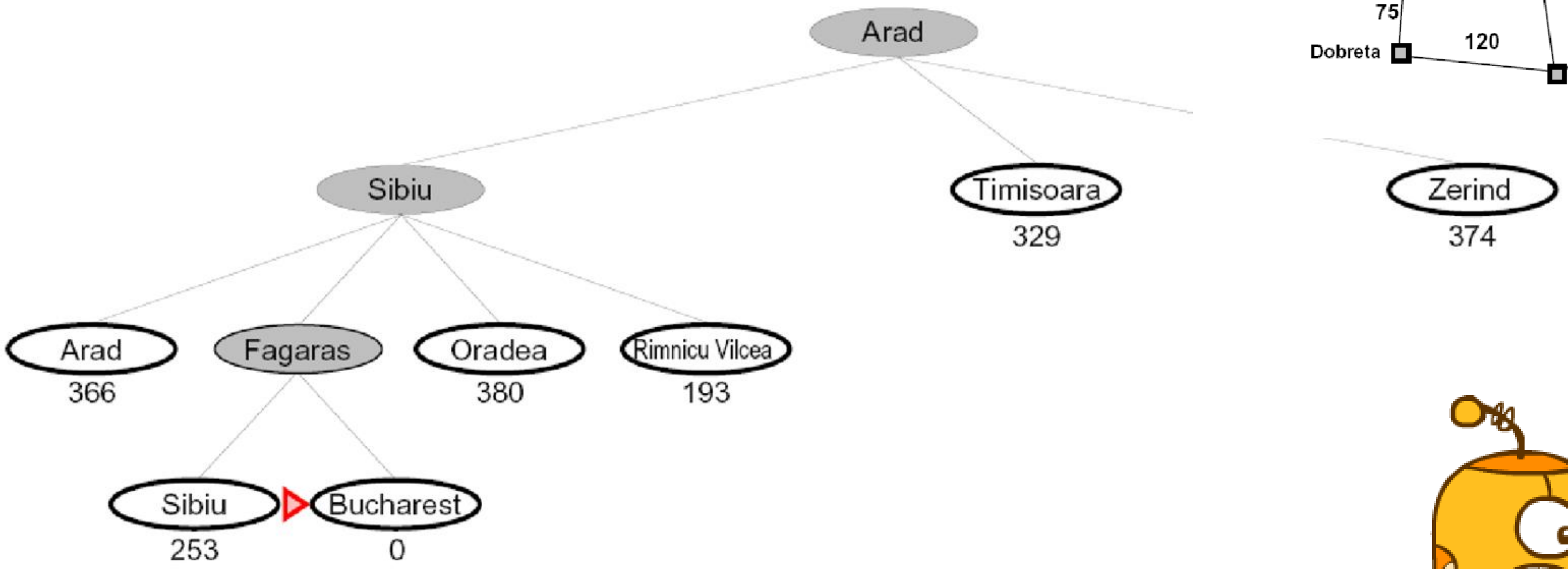
Greedy Best-first Search Example

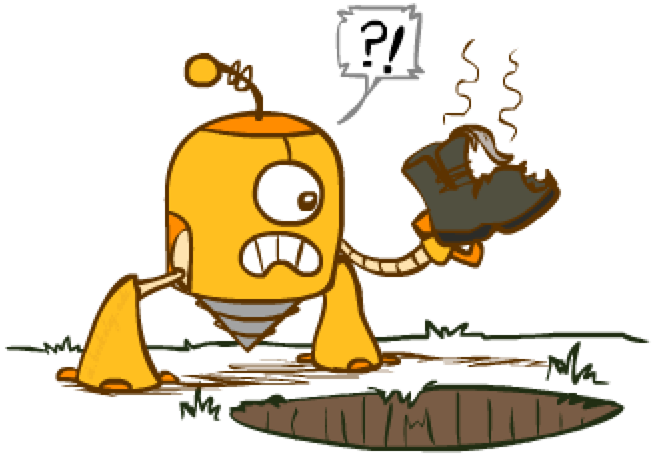


Greedy Search



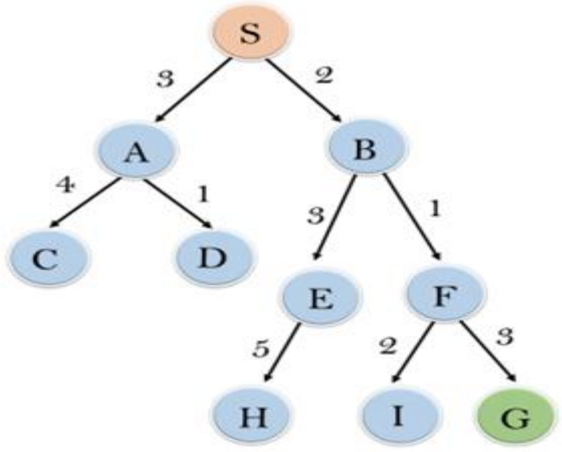
- Expand the node that seems closest…



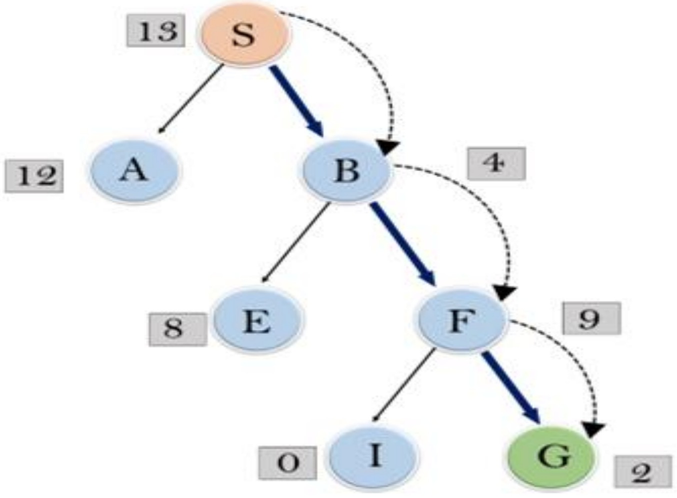


- Is it optimal?
  - No. Resulting path to Bucharest is not the short

Example: Greedy Best-first Search



node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0



Expand the nodes of S and put in the CLOSED list

Initialization: Open [A, B], Closed [S]
Iteration 1: Open [A], Closed [S, B]
Iteration 2: Open [E, F, A], Closed [S, B]
: Open [E, A], Closed [S, B, F]
Iteration 3: Open [I, G, E, A], Closed [S, B, F]
: Open [I, E, A], Closed [S, B, F, G]
Hence the final solution path will be: **S----> B----->F----> G**

Properties of Greedy Best-first Search

- Complete? No – can get stuck in loops,

  e.g., Iasi → Neamt → Iasi → Neamt →

- Time? The worst case is $O(b^m)$, but a good heuristic can give dramatic improvement

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

- Optimal? No (do not consider all the data.)
  - Choice made by a greedy algorithm may depend on choices it has made so far, but it is not aware of future choices it could make.)

Greedy Best-first Search

- **Advantages:**
- Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
- This algorithm is more efficient than BFS and DFS algorithms.

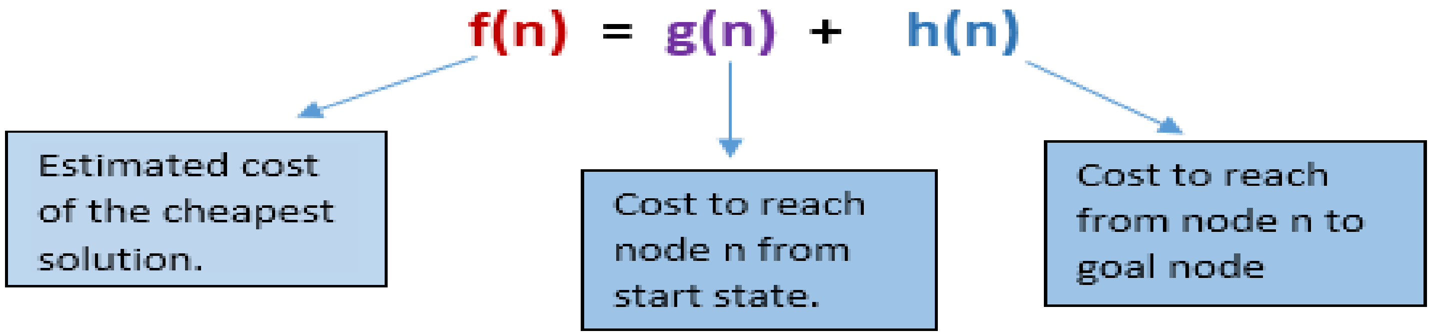
- **Disadvantages:**
- It can behave as an unguided depth-first search in the worst case scenario.
- It can get stuck in a loop as DFS.
- This algorithm is not optimal.

A* Search

- It combines the strengths of UCS and greedy best-first search, by which it solve the problem efficiently.

- Here, the heuristic is the summation of the cost in UCS, denoted by g(n), and the cost in greedy search, denoted by h(n). The summed cost is denoted by f(n).

- Hence we can combine both costs as following, and this sum is called as a fitness number.



- A* search algorithm expands less search tree and provides optimal result faster.

A* Search

- **Idea:** avoid expanding paths that are already expensive, but expands most promising paths first.

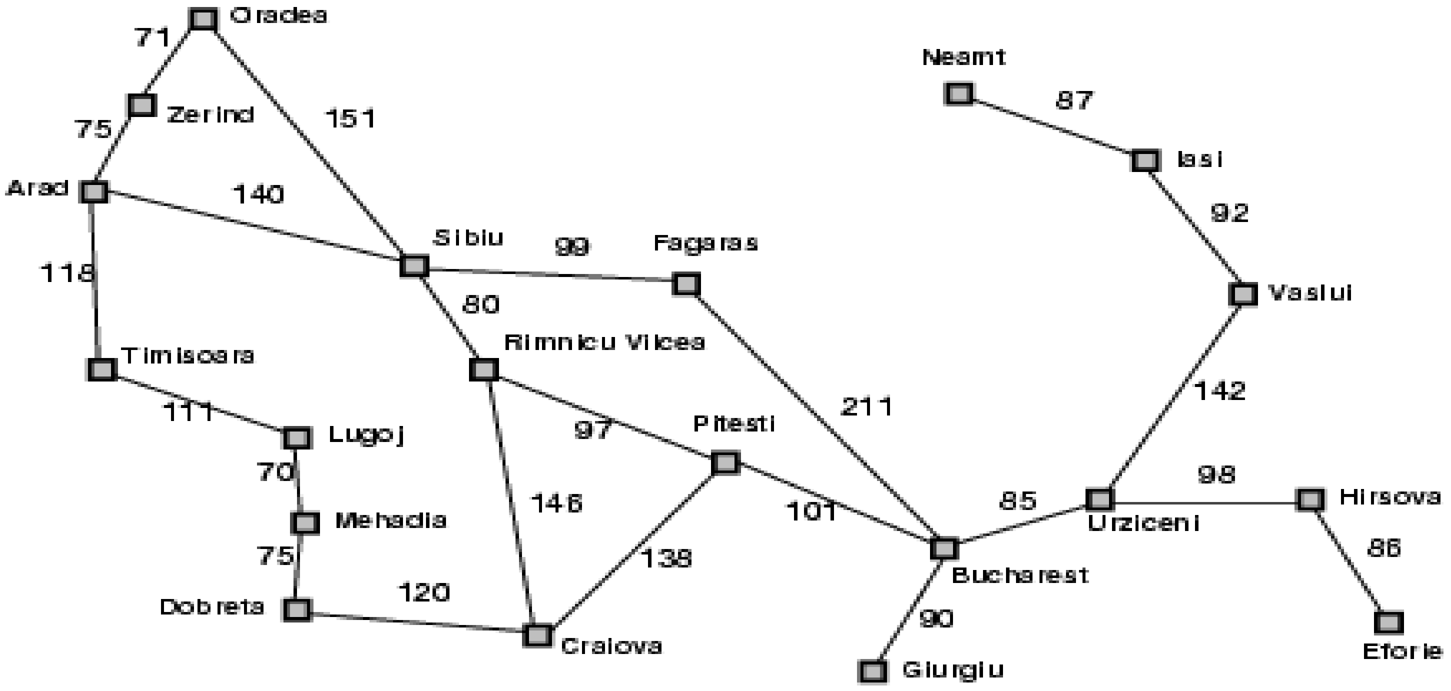
- Evaluation function $f(n) = g(n) + h(n)$

- $g(n)$ = Actual cost to reach n
- $h(n)$ = Estimated cost from n to goal
- $f(n)$ = Estimated total cost of path through n to goal

Algorithm of A* Search

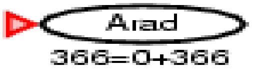
- Step1: Place the starting node in the OPEN list.
- Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stops.
- Step 3: Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise
- Step 4: Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.
- Step 5: Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.
- Step 6: Return to Step 2.

A* Search Example

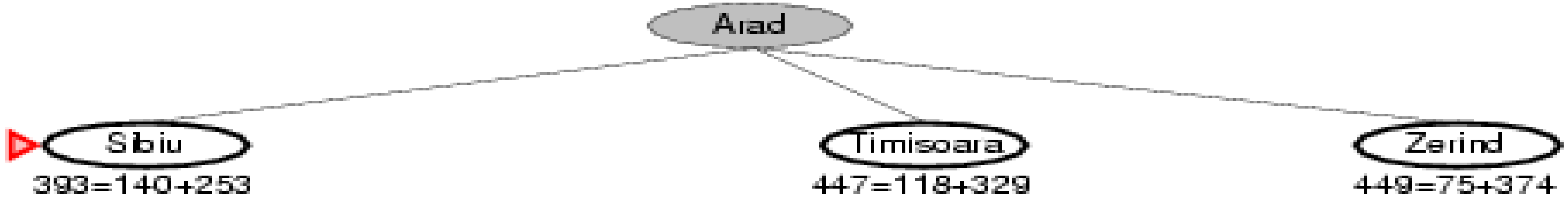


Straight–line distance to Bucharest

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

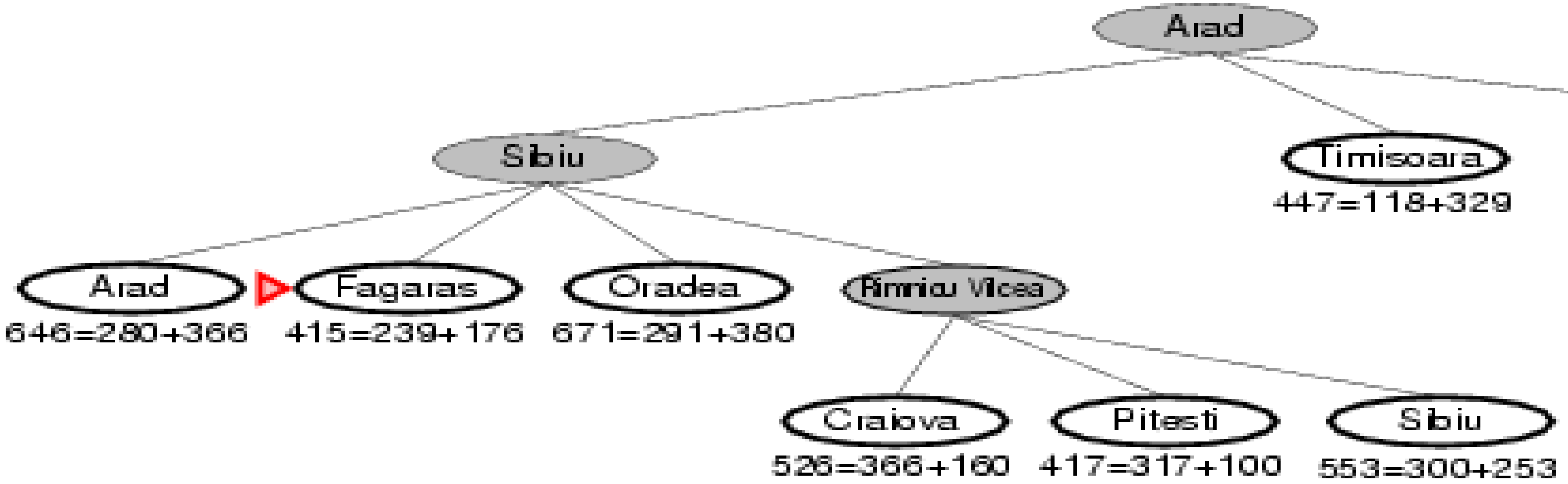


A^* Search Example

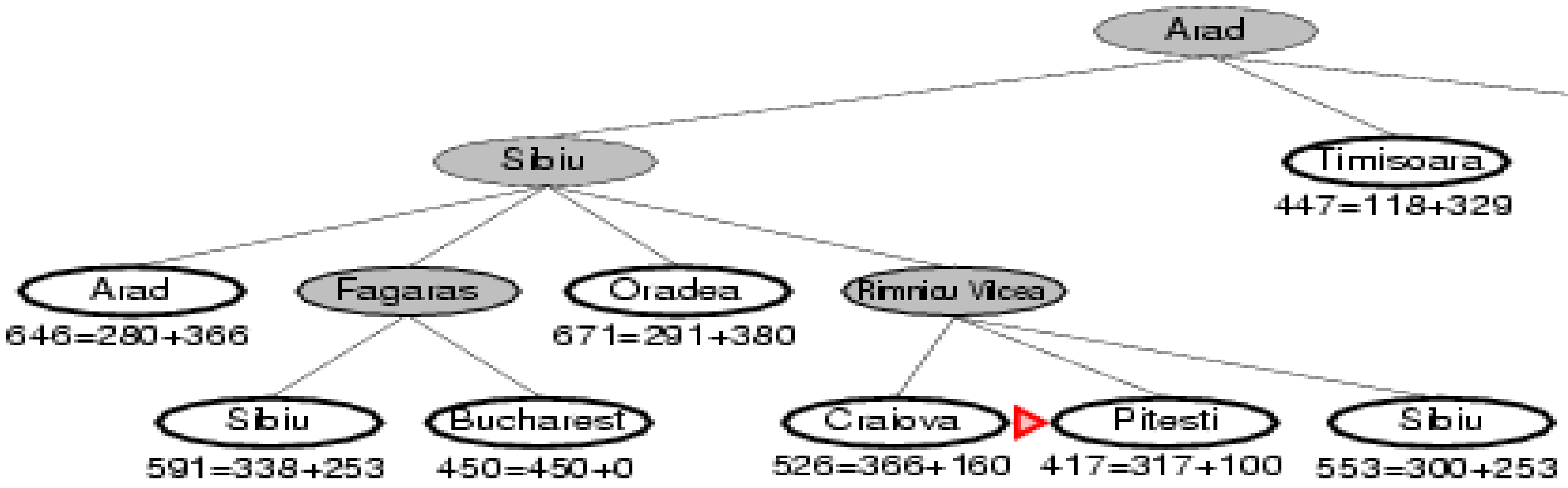


A* Search Example

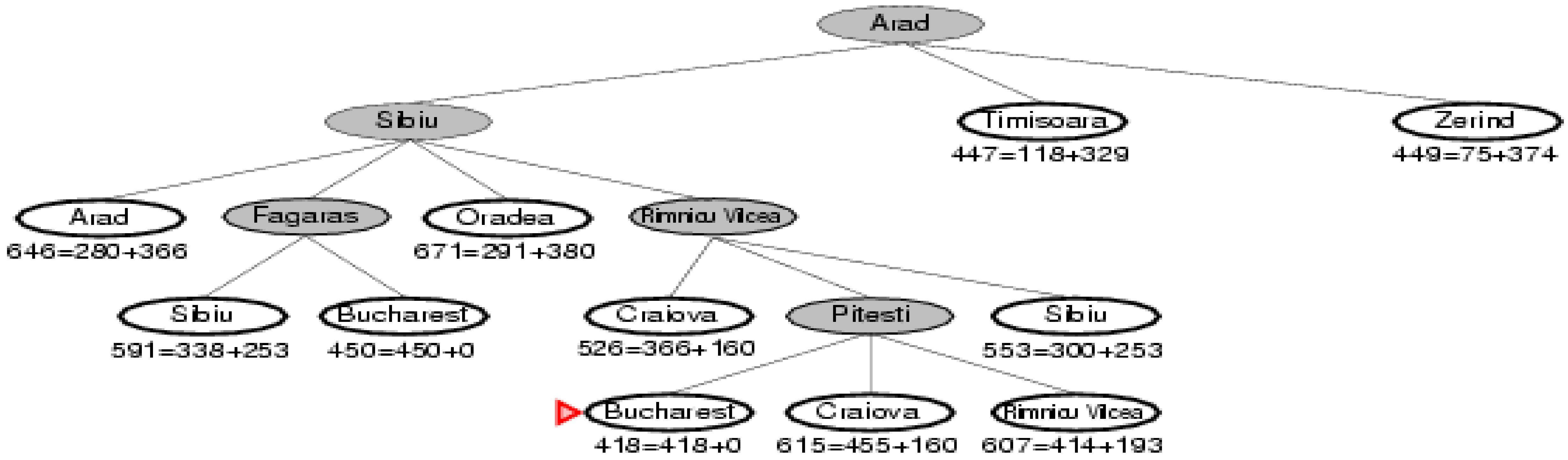
A* Search Example



A* Search Example

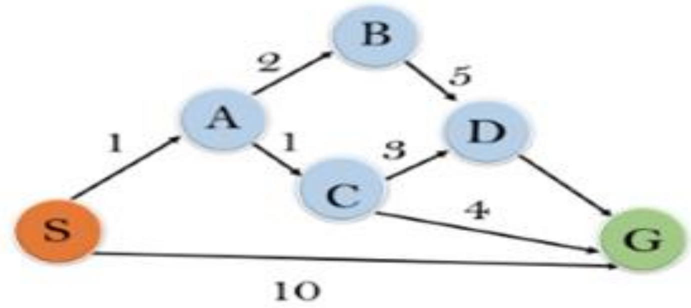


A* Search Example

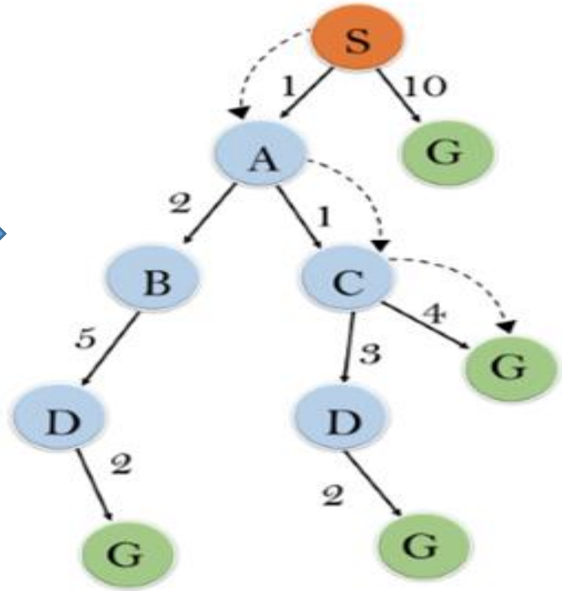


Finally return the path **A--->S--->R--->P--->B**
It provides the optimal path with shortest cost 418.

A* Search Example



State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0



Initialization: {(S, 5)}
Iteration1: {(S--> A, 4), (S-->G, 10)}
Iteration2: {(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}
Iteration3: {(S--> A-->C--->G, 6), (S--> A-->C--->D, 11), (S--> A-->B, 7),
(S-->G, 10)}
Iteration 4 will give the final result, as **S--->A--->C--->G**, it provides the optimal path with cost 6.

Properties of A*

Points to remember:

-A* algorithm returns the path which occurred first, and it does not search for all remaining paths.

-The **efficiency of A*** algorithm depends on the **quality of heuristic**.

-A* algorithm expands all nodes which satisfy the condition f(n).

- Complete? A* algorithm is complete as long as:

  -Branching factor is finite.

  -Cost at every action is fixed.

- Optimal? Yes if it follows below two conditions:

  -Admissible: h(n) should be an admissible heuristic for A* tree search. An admissible heuristic is optimistic in nature.

  -Consistency: Second required condition is consistency for only A* graph-search.

Properties of A*

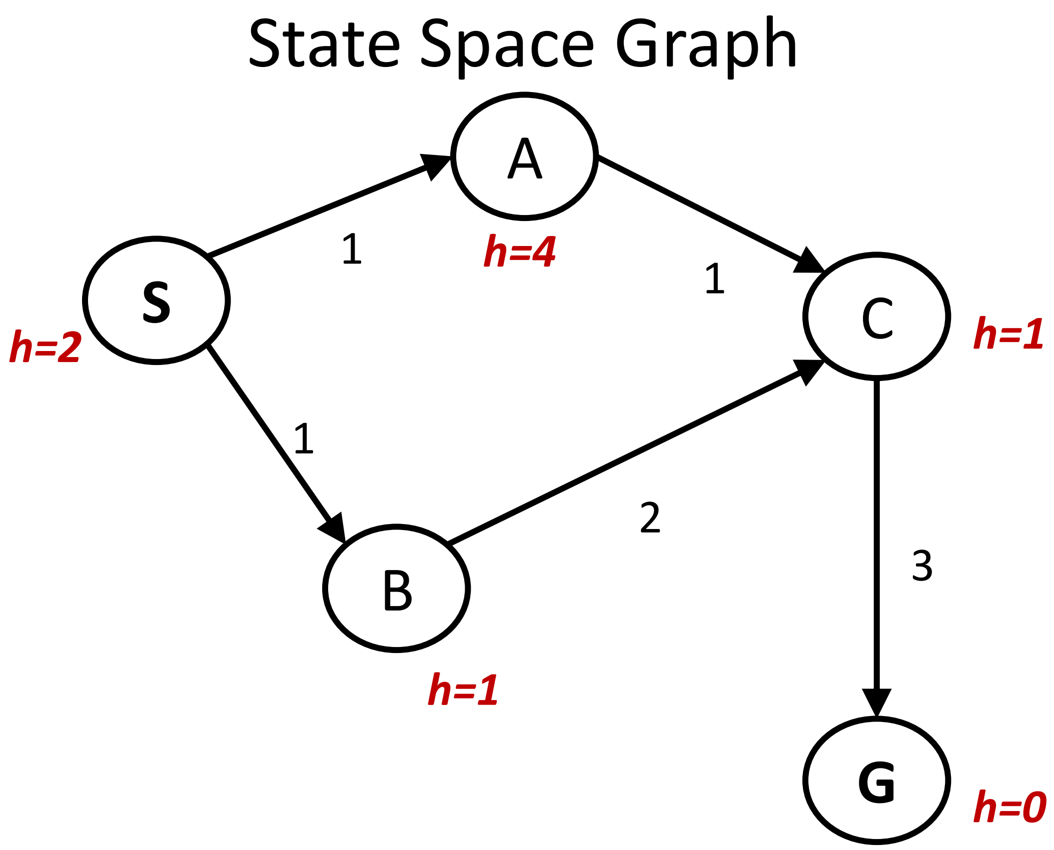
- If the heuristic function is admissible, then A* tree search will always find the least cost path.

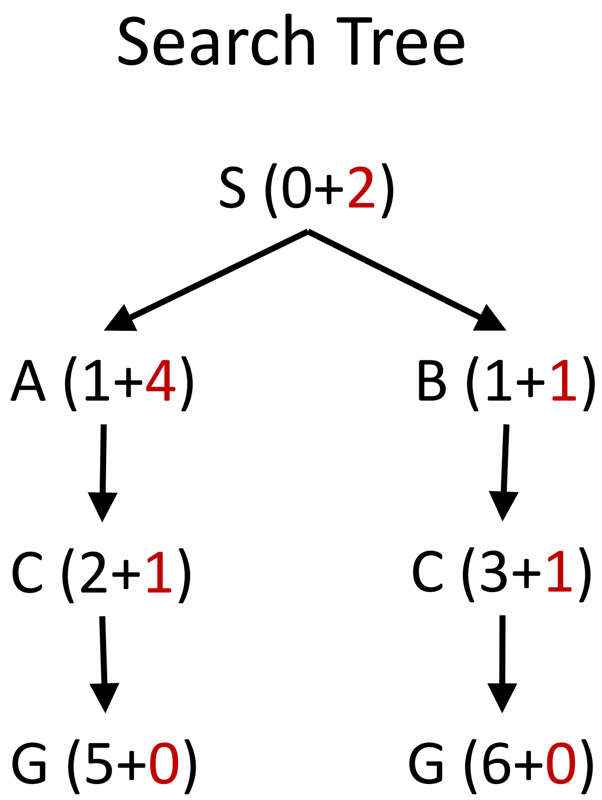
❑<u>Time Complexity?</u> **Exponential (O(b^d))**

➢The time complexity of A* search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution **d**. So the time complexity is **O(b^d)**, where **b** is the branching factor

❑<u>Space Complexity?</u> **Keeps all nodes in memory**. The space complexity of A* search algorithm is **O(b^d)**.

A* Graph Search Gone Wrong?





- **Admissibility is not enough to maintain completeness and optimality under A* graph search.**

A* Graph Search Gone Wrong?

- In the above example, the optimal route is to follow ***S -> A -> C -> G***, yielding a total path cost of 1+1+3 = 5. The other path to the goal, ***S -> B -> C -> G*** has a path cost of 1+2+3 = 6.

- However, as the heuristic value of node ***A*** is so much larger than the heuristic value of node ***B***, node ***C*** is first expanded along the second, suboptimal path as a child of node *B*.

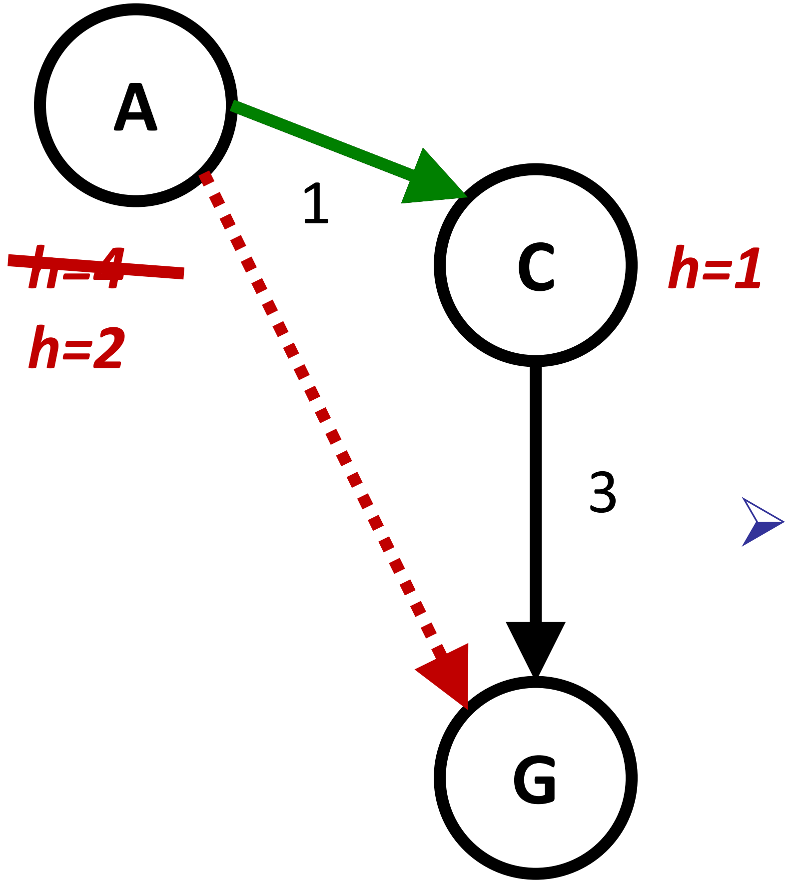
- It's (node C) then placed into the **"closed"** set, and so A* graph search fails to re-expand it when it visits it as a child of *A*, so it never finds the optimal solution.

A* Graph Search Gone Wrong?

- Hence, **to maintain completeness** and **optimality** under A* graph search, we need an even stronger property than admissibility, which is **consistency**.
- The central idea of consistency is that we enforce not only that a heuristic underestimates the *total* distance to a goal from any given node, but also the cost/weight of each edge in the graph.
- The cost of an edge as measured by the heuristic function is simply the difference in heuristic values for two connected nodes.
- Mathematically, the consistency constraint can be expressed as follows:

  - $\forall A,C;\ h(A) - h(C) \leq cost(A,C)$

Consistency of Heuristics



- Main idea: estimated heuristic costs ≤ actual costs
  - **Admissibility: heuristic cost ≤ actual cost to goal**

    h(A) ≤ actual cost from A to G

  - **Consistency: heuristic "arc" cost ≤ actual cost for each arc**

    h(A) – h(C) ≤ cost(A to C) = ?

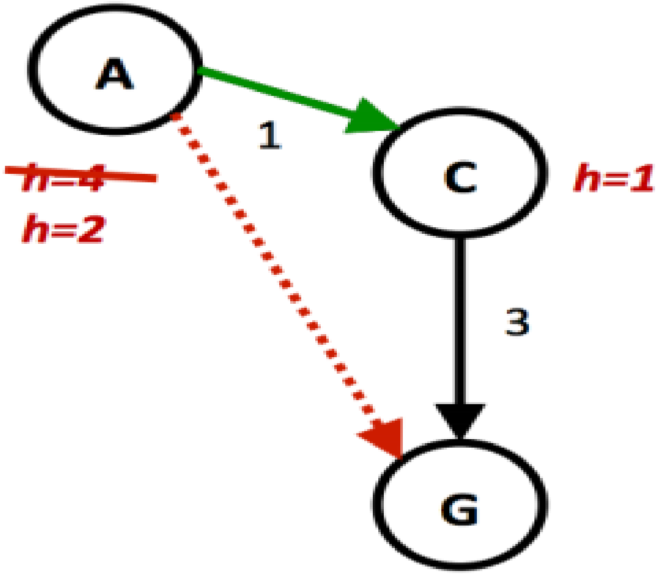
- Consequences of consistency:
  - The **f** value [f(n)] along a path never decreases

    h(A) ≤ cost(A to C) + h(C)

  - A* graph search is optimal

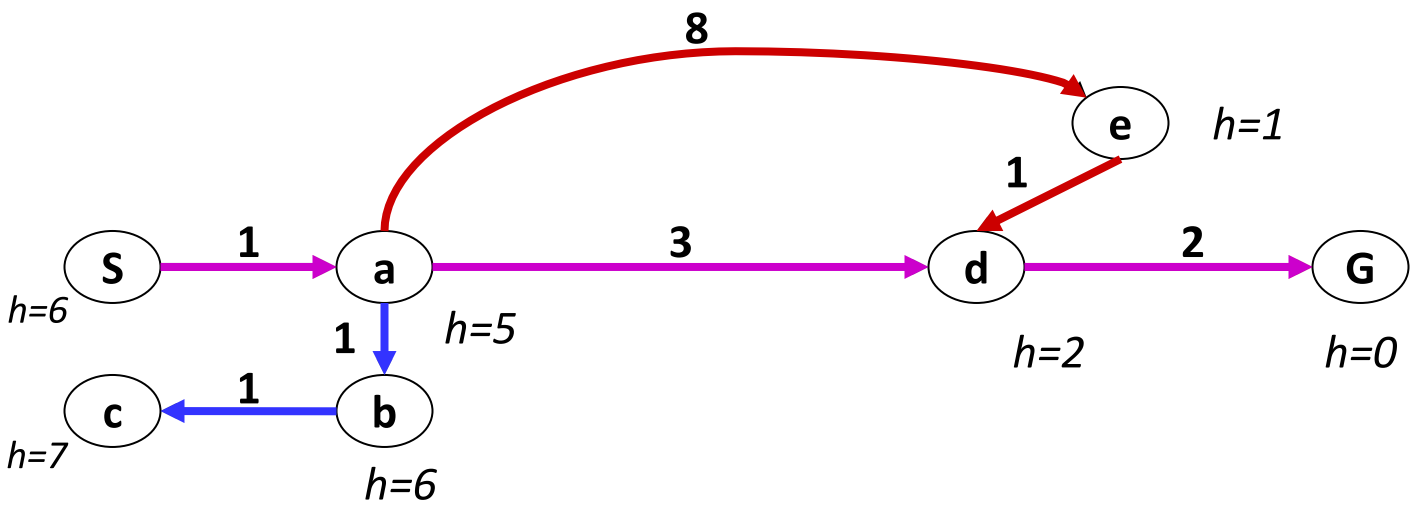
Consistency of Heuristics

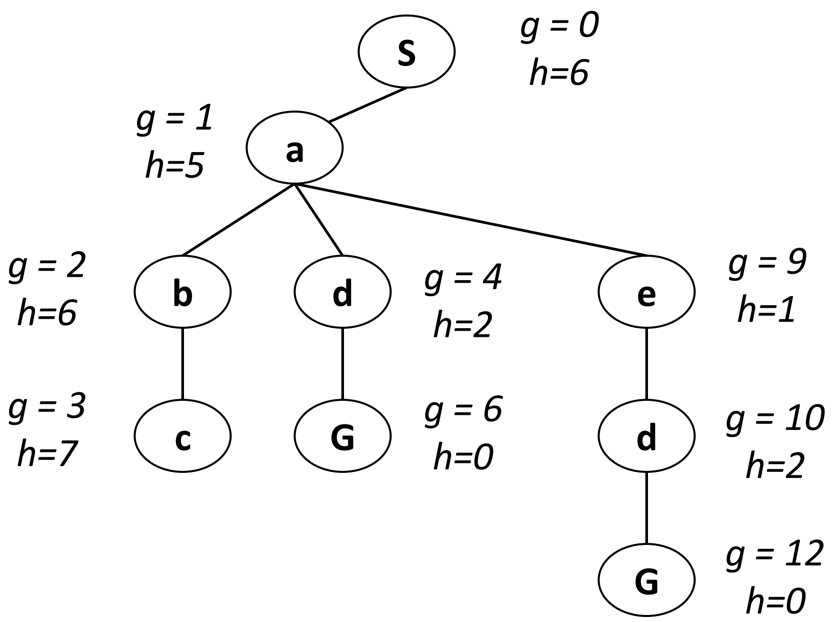
- The red dotted line corresponds to the **total estimated goal distance**.
- If h(A) = 4, then the heuristic is admissible, as the distance from A to the goal is 4 ≥ h(A), and same for h(C) = 1 ≤ 3. As admissibility means: $\mathbf{0 <= h(n) <= h^*(n)}$
- However, the heuristic cost from A to C is h(A) - h(C) = 4 - 1 = 3. Our heuristic estimates the cost of the edge between A and C to be 3 while the true value is cost(A, C) = 1, a smaller value.
- Since **h(A) - h(C) !≤ cost(A, C)**, this heuristic is **not consistent**. Running the same computation for h(A) = 2, however, yields h(A)-h(C) = 2 - 1 = 1 ≤ cost(A,C).
- Thus, using **h(A) = 2 makes our heuristic consistent.**



Combining UCS and Greedy

- Uniform-cost orders by path cost, or *backward cost* g(n)
- Greedy orders by goal proximity, or *forward cost* h(n)
- A* Search orders by the sum: f(n) = g(n) + h(n)





Admissible Heuristics

- A heuristic $h(n)$ is admissible if for every node n,
  $0 \leq h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from n.

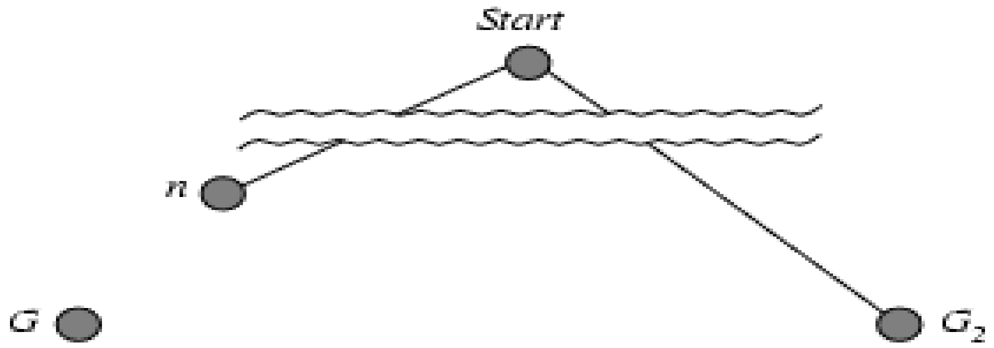
- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic

- Example: $h_{SLD}(n)$ (never overestimates the actual road distance)

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

Optimality of A* (proof)

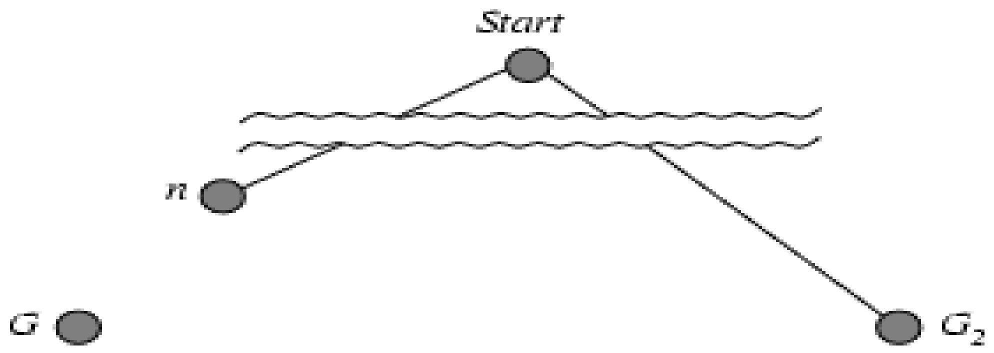
- Suppose some suboptimal goal G_2 has been generated and is in the fringe. Let *n* be an unexpanded node in the fringe such that *n* is on a shortest path to an optimal goal *G*.



- $f(G_2) = g(G_2)$ since $h(G_2) = 0$
- $g(G_2) > g(G)$ since G_2 is suboptimal
- $f(G) = g(G)$ since $h(G) = 0$
- $f(G_2) > f(G)$ from above

Optimality of A^* (proof)

- 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*.



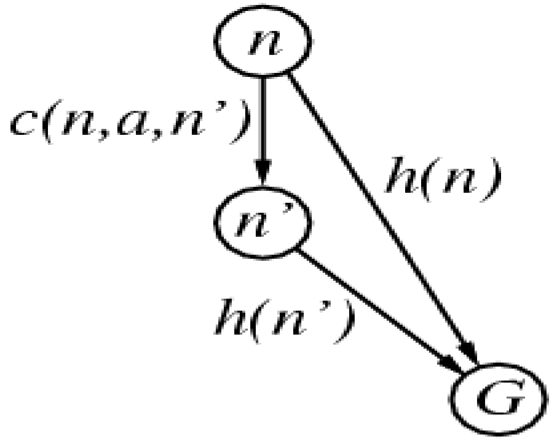
- $f(G_2)$ $> f(G)$ from above
- $h(n)$ $\leq$ h^*(n) since h is admissible
- $g(n) + h(n)$ $\leq g(n) + h^*(n)$
- $f(n)$ $\leq f(G)$

Hence $f(G_2) > f(n)$, and A^* will never select G_2 for expansion

Consistent Heuristics

- A heuristic is consistent if for every node n, every successor n' of n generated by any action a,

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



- If h is consistent, we have

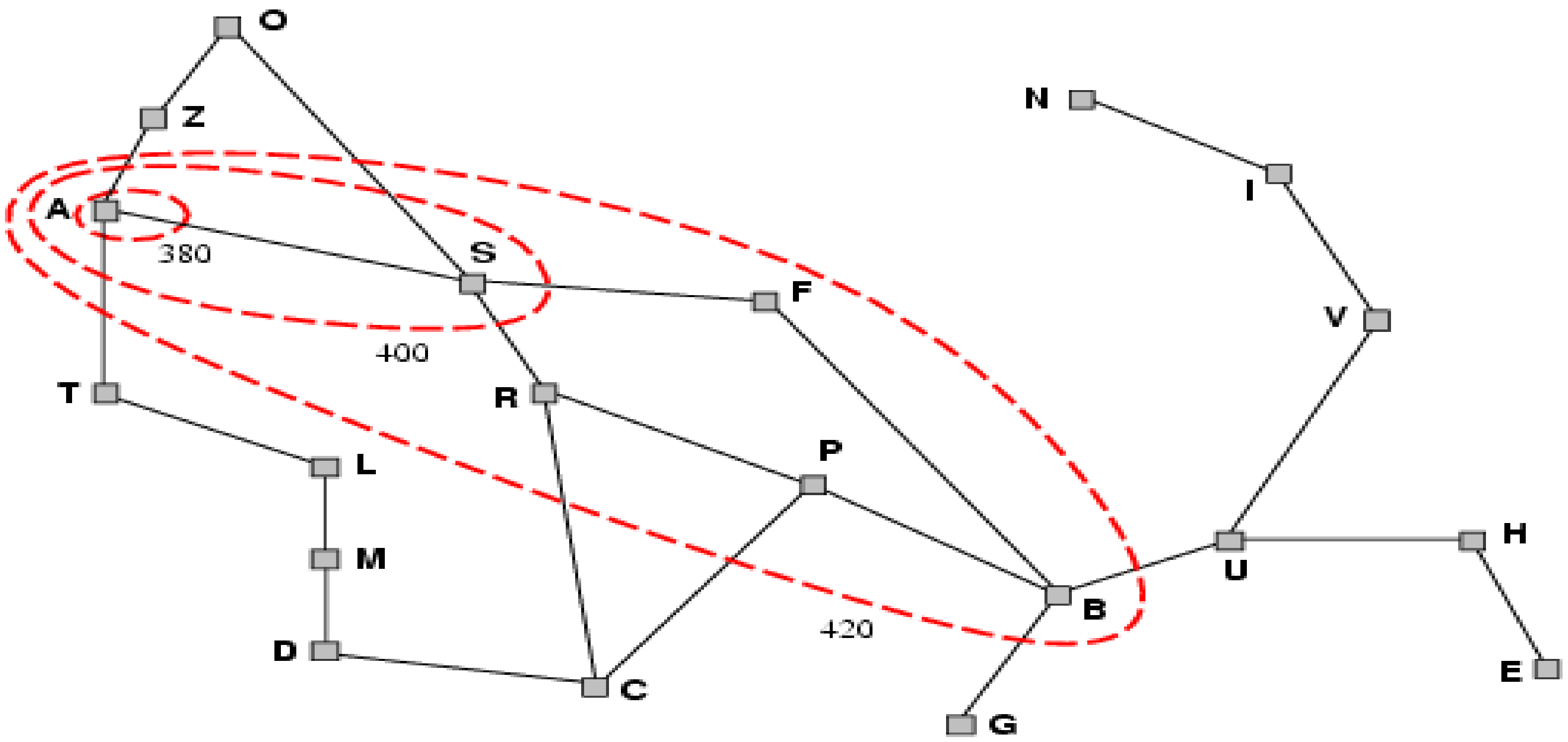
$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n,a,n') + h(n') \\ &\geq g(n) + h(n) \\ &= f(n) \end{aligned}$$

i.e., $f(n)$ is non-decreasing along any path.

- Theorem: If $h(n)$ is consistent, A* using GRAPH-SEARCH is optimal

Optimality of A*

- A* expands nodes in order of increasing *f* value
- Gradually adds "*f*-contours" of nodes
- Contour *i* has all nodes with $f=f_i$, where $f_i < f_{i+1}$

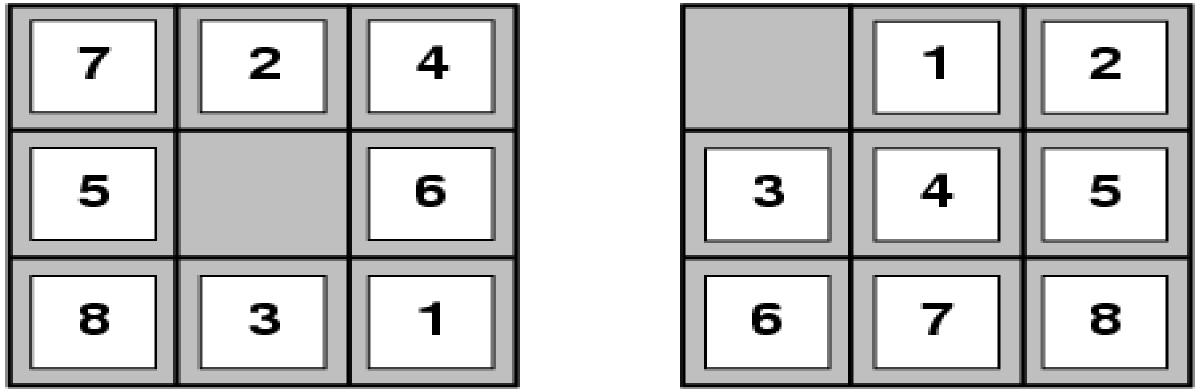


Admissible Heuristics

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)



Start State Goal State

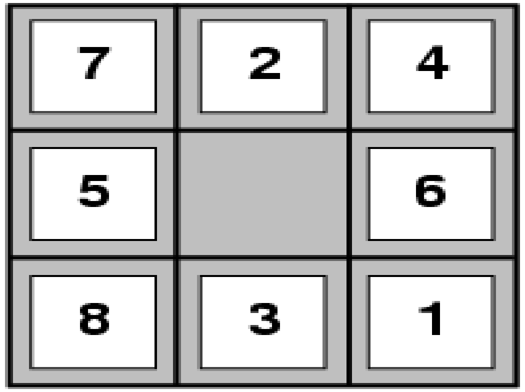
- <u>$h_1(S)$ = ?</u>
- <u>$h_2(S)$ = ?</u>

Admissible Heuristics

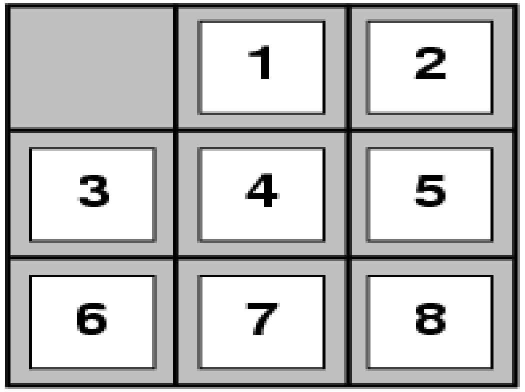
E.g., for the 8-puzzle:

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

(i.e., no. of squares from desired location of each tile)



Start State



Goal State

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

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
-
- Typical search costs (average number of nodes expanded):
-
- $d=12$ IDS = 3,644,035 nodes
  $A^*(h_1)$ = 227 nodes
  $A^*(h_2)$ = 73 nodes
- $d=24$ IDS = too many nodes
  $A^*(h_1)$ = 39,135 nodes
  $A^*(h_2)$ = 1,641 nodes
-

Relaxed Problems

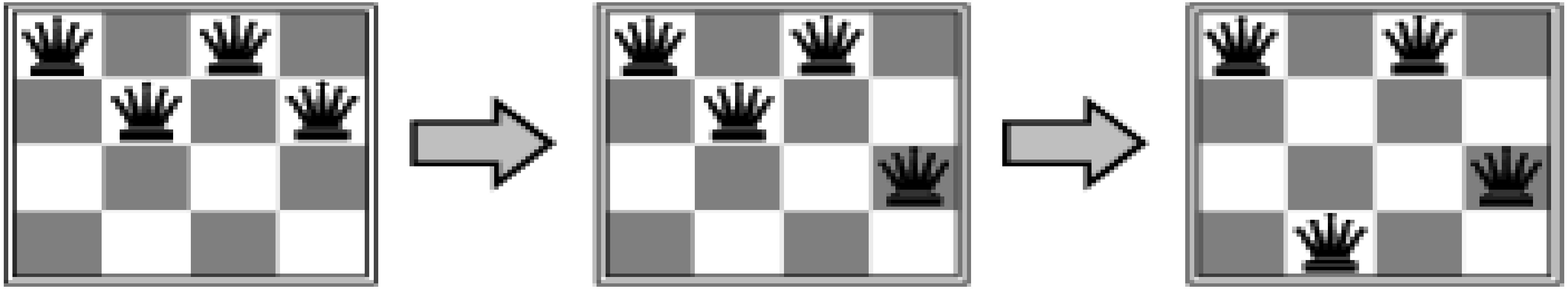
- A problem with fewer restrictions on the actions is called a relaxed problem
-
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
-
- If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then $h_1(n)$ gives the shortest solution
-
- If the rules are relaxed so that a tile can move to any adjacent square, then $h_2(n)$ gives the shortest solution
-

Local Search Algorithms

- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution
-
- State space = set of "complete" configurations
- Find configuration satisfying constraints, e.g., n-queens
- In such cases, we can use local search algorithms
- keep a single "current" state, try to improve it

Example: *n*-queens

- Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal



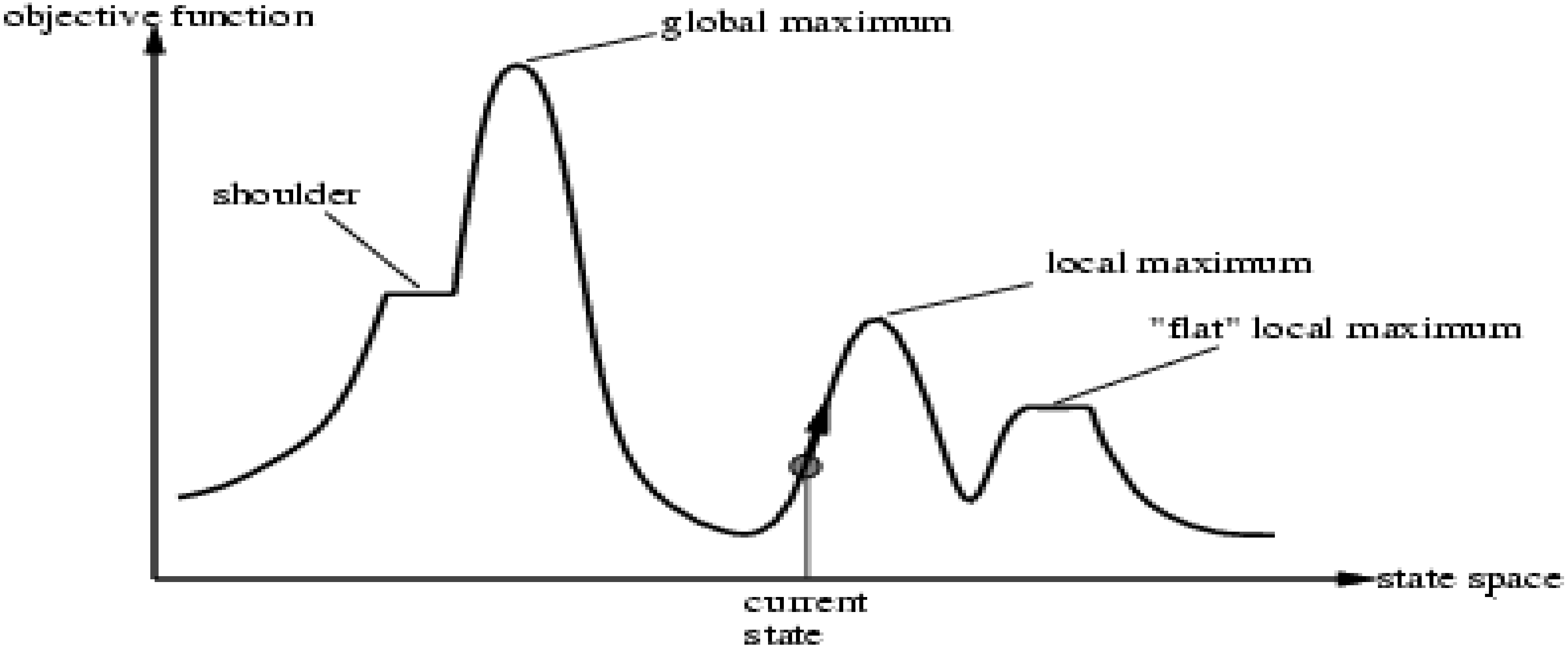
Hill-Climbing Search

➢ "Like climbing Everest in thick fog with amnesia"

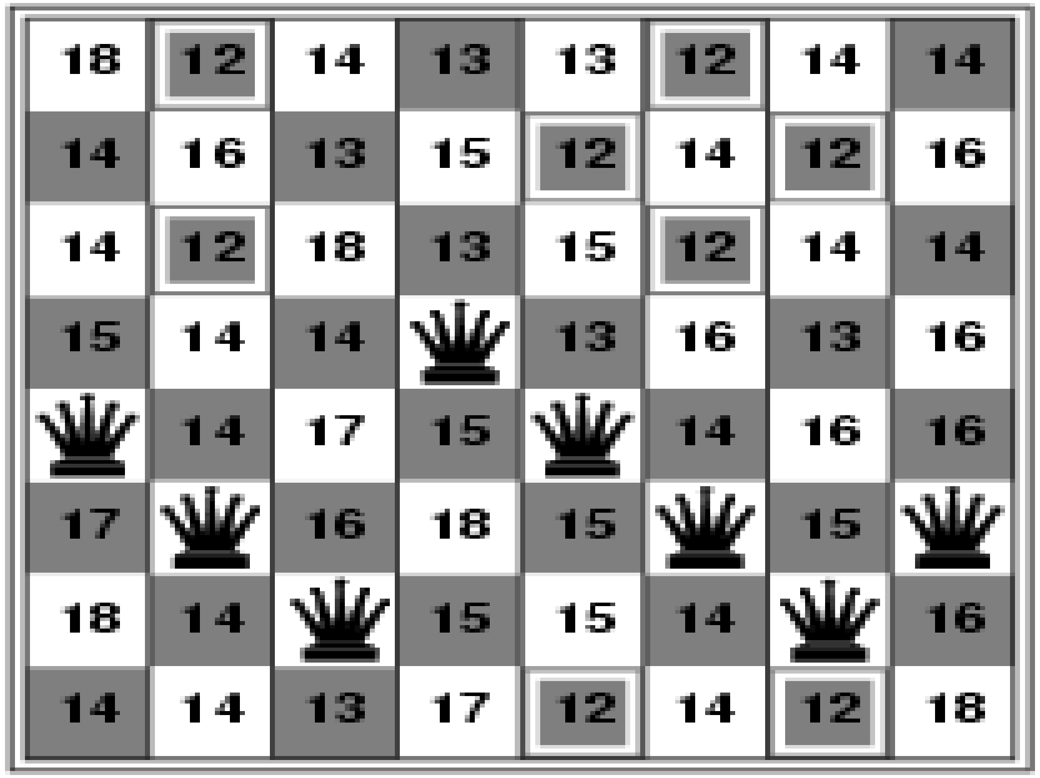
```
function HILL-CLIMBING(problem) returns a state that is a local maximum
   inputs: problem, a problem
   local variables: current, a node
                    neighbor, a node

   current ← MAKE-NODE(INITIAL-STATE[problem])
   loop do
       neighbor ← a highest-valued successor of current
       if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
       current ← neighbor
```

Hill-Climbing Search

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

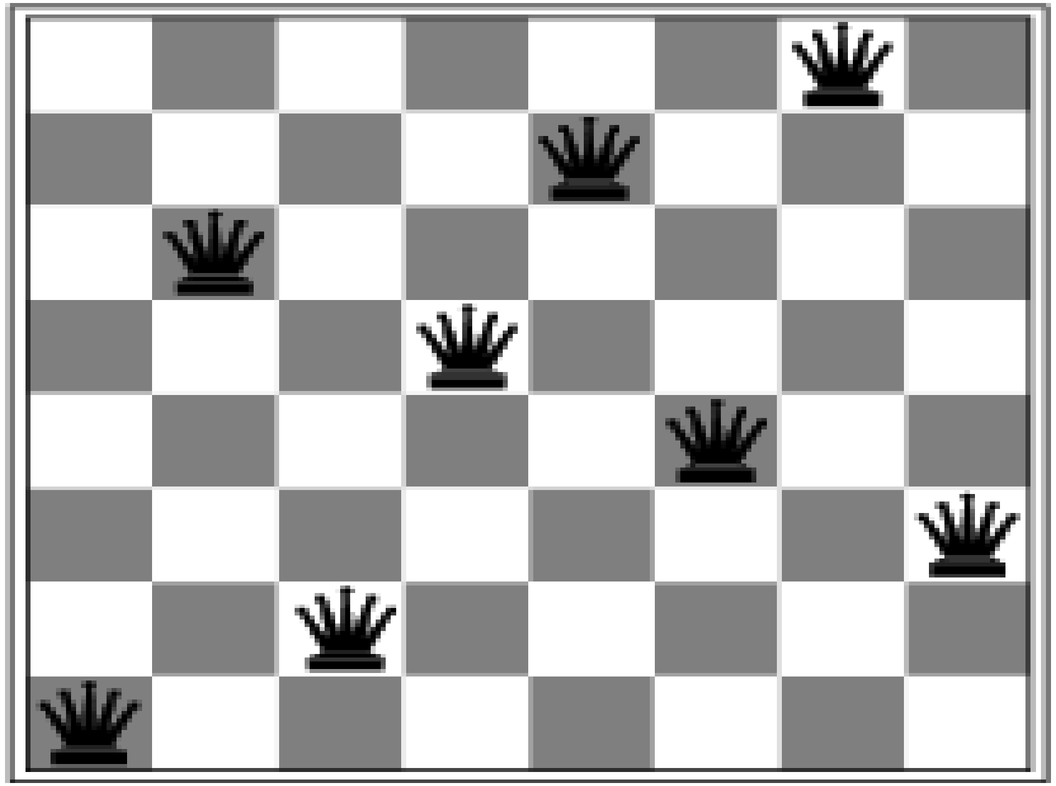


Hill-Climbing Search: 8-queens Problem



- h = number of pairs of queens that are attacking each other, either directly or indirectly
- $h = 17$ for the above state

Hill-climbing Search: 8-queens Problem



- A local minimum with *h = 1*

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 $t \leftarrow 1$ **to** ∞ **do**
    $T \leftarrow$ *schedule*[t]
    if $T = 0$ **then return** *current*
    next ← a randomly selected successor of *current*
    $\Delta E \leftarrow$ 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

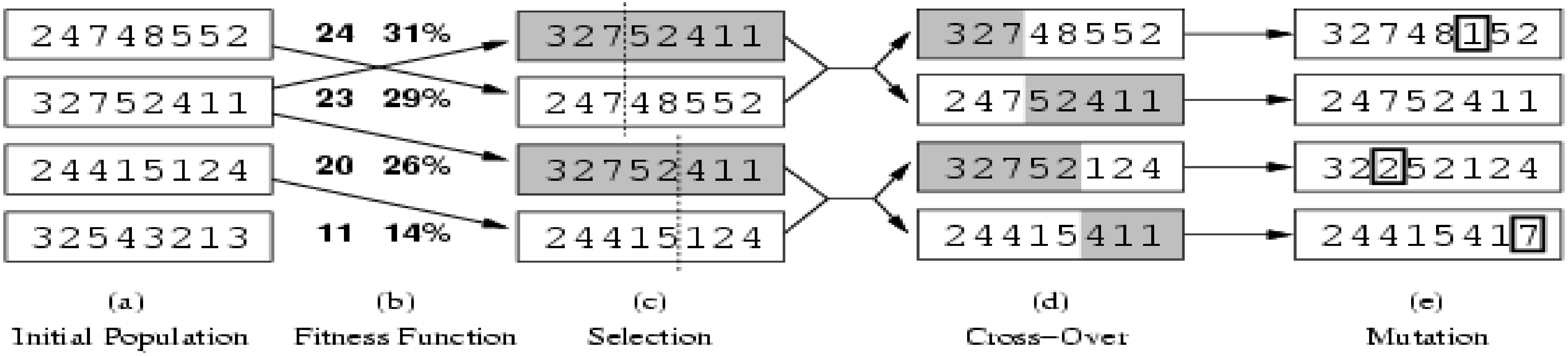
Local Beam Search

- Keep track of *k* states rather than just one
- Start with *k* randomly generated states
- At each iteration, all the successors of all *k* states are generated
- If any one is a goal state, stop; else select the *k* best successors from the complete list and repeat.

Genetic Algorithms

- A successor state is generated by combining two parent states
- Start with *k* randomly generated states (population)
- A state is represented as a string over a finite alphabet (often a string of 0s and 1s)
- Evaluation function (fitness function). Higher values for better states.
- Produce the **next generation** of states by selection, crossover, and mutation

Genetic Algorithms



- Fitness function: number of non-attacking pairs of queens (min = 0, max = 8 × 7/2 = 28)

- 24/(24+23+20+11) = 31%

- 23/(24+23+20+11) = 29% etc

Acknowledgement

- AIMA = Artificial Intelligence: A Modern Approach by Stuart Russell and Peter Norving (3rd edition)
- UC Berkeley (Some slides were created by Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley)
- U of toronto
- Other online resources

Thank You