

Informed search algorithms

Chapter 4

Outline

- Hill-climbing search
- Heuristics
- Best-first search
- Greedy best-first search
- A^* search

Heuristic search techniques

- Blind searches are normally very inefficient. By adding domain knowledge we can improve the search process.
- The idea behind the heuristic search is that we explore the node that is most likely to be nearest to a goal state.
- A heuristic function has some knowledge about the problem so that it can judge how close the current state is to the goal state.
- A heuristic function $h(n)$ = estimated cost of the cheapest path from the state at node n to a goal state.

Hill climbing


- It is often used when a good heuristic function is available for evaluating states but when no other useful knowledge is available.
- This algorithm is simply a **loop that continuously moves in the direction of increasing value** i.e uphill.
- It terminates when it reaches a “**peak**” where no neighbor has a higher value.
- The algorithm doesn't maintain a search tree, **so the current node data structure only records the state** and its objective function value.
- Hill – climbing doesn't look ahead beyond the immediate neighbors of the current state.

Hill climbing algorithm

1. Evaluate the **initial state (IS)**. If it is the **goal state (GS)** , then return it and quit. Else consider IS as the **current state (CS)** and proceed.
2. Loop until a solution is found or there are no **new operator (OP)** to be applied to the **CS**.
 - a) Select an **OP** that has not yet been applied to the CS and apply it to produce a **new state (NS)**.

Hill climbing algorithm

b) Evaluate the **NS**:

- I. If NS is a GS , then return it and quit.
- II. If it is not a GS but better than the CS, then consider it as the current state(i,e CS  NS) and proceed.
- III. If NS is not better than CS then continue in the loop by selecting the next appropriate OP for CS.

Steepest – Ascent Hill climbing algorithm

It considers all the moves from the **CS** and selects the best one as the next state. It is also called **gradient search**.

Algorithm

1. Evaluate the **initial state (IS)**. If it is the **goal state (GS)**, then return it and quit. Else consider **IS** as the current state **(CS)** and proceed.
2. Loop until a solution is found or until a complete iteration produces no change to the **CS**:



Steepest – Ascent Hill climbing algorithm

- a) Let **successor** (**SUCC**) be a state such that any **NS** that can be generated from **CS** is better than **SUCC**.
[i.e setting SUCC to a minimum value at the beginning of an iteration or set **CS** as **SUCC**]
- b) For each **operator OP** that applies to the **CS** do:
 - I. Apply **OP** to **CS** and generate a **NS**.
 - II. Evaluate the **NS**. If it is a **GS** then return it and quit. If not , compare it with **SUCC**. If **NS** is better than **SUCC**, then set **SUCC** to **NS**; else leave SUCC unchanged.
- c) If the SUCC is better than CS, then set CS to SUCC [i.e move to the next best state]



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

- Both basic and steepest-ascent hill climbing may fail to find a solution.
- Either algorithm may terminate not by finding a goal state (**GS**) but by getting to a state from which no better states can be generated.
- This will happen if the program has reached either a **local maximum**, a **plateau** or a **ridge**.

Hill-climbing search

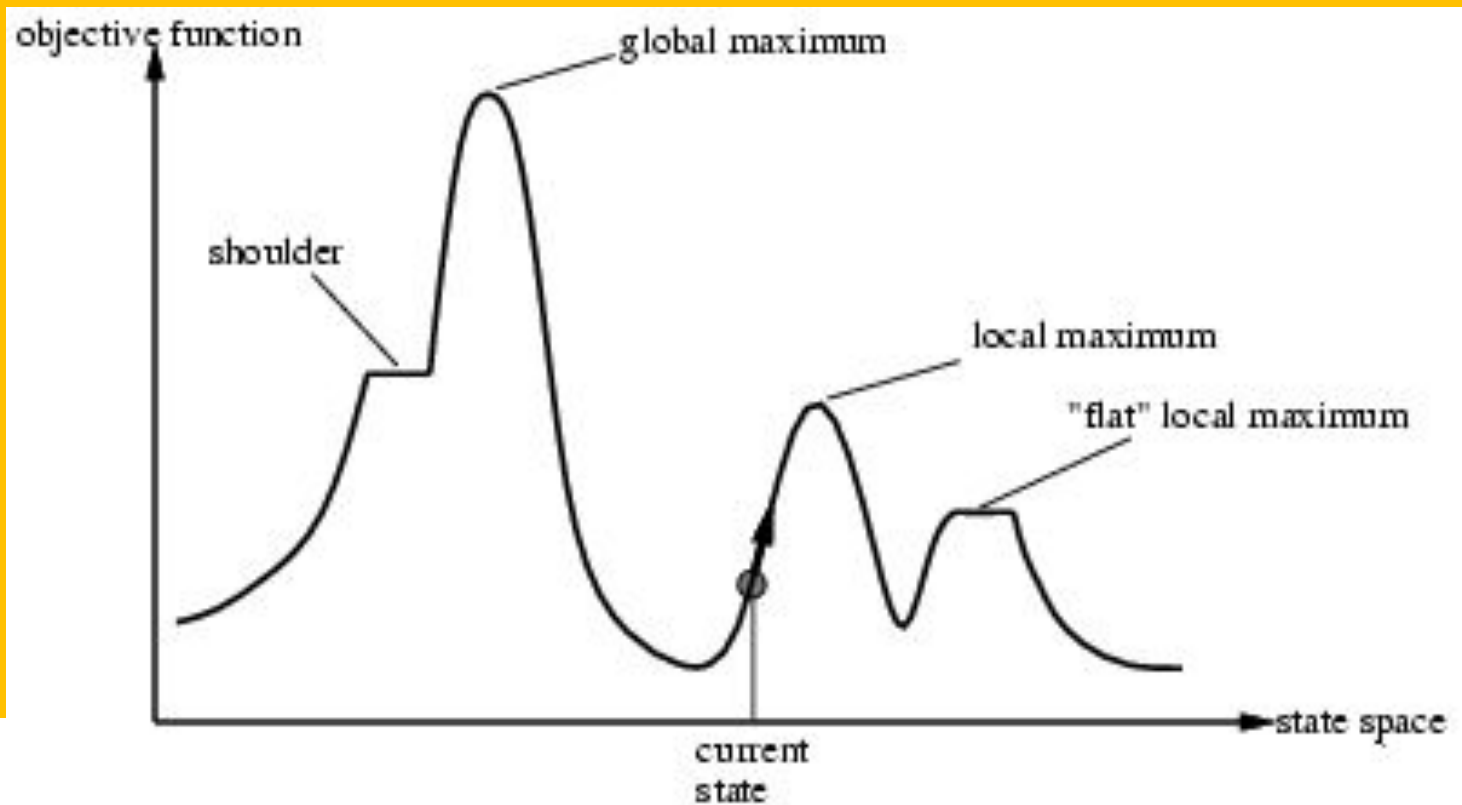
- A **local maximum** is a state that is better than all its neighbors but is not better than some other states farther away.
- Solution of local maxima:-
 - Move in some arbitrary direction
 - Back track to an ancestor and try some other alternatives.
- A plateau is a flat area in the search space in which all the neighboring states have the same heuristic function value.

Hill-climbing search

- Solution of plateau
 - Expand few generation ahead to move to a different section of the search space.
- A ridge is an area in the search space which is higher than its surroundings but itself has slopes. It is not possible to traverse a ridge by a single move i.e no such operator is available.
- Solution of ridges:-
 - Apply several operators before doing the evaluation

Hill-climbing search

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

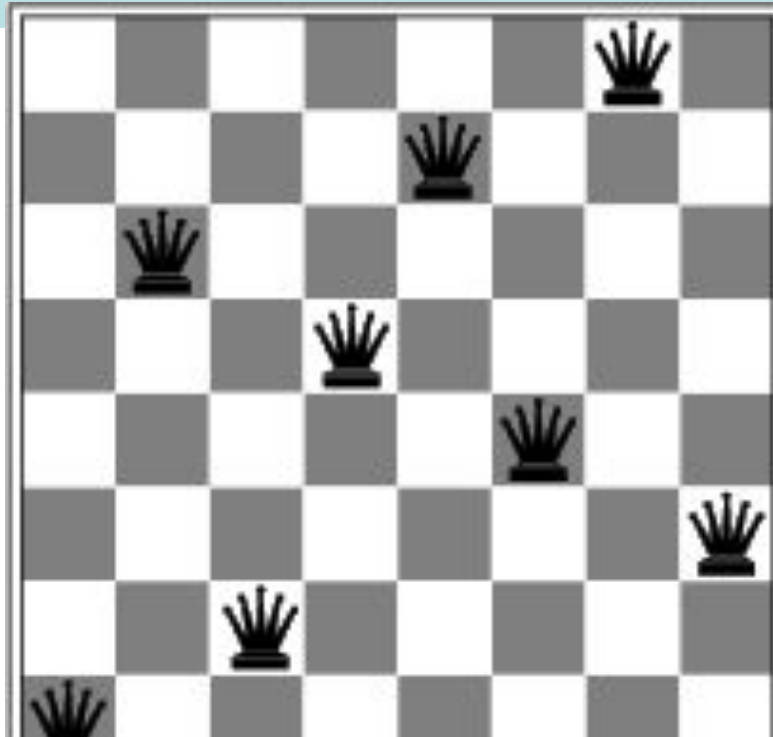


Hill-climbing search: 8-queens problem

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♚	13	16	13	16
♚	14	17	15	♚	14	16	16
17	♚	16	18	15	♚	15	♚
18	14	♚	15	15	14	♚	16
14	14	13	17	12	14	12	18

- 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 $h = 1$

Best-first search

- Idea: use an **evaluation function** $f(n)$ for each node
 - estimate of "desirability"
 - Expand most desirable unexpanded node
- Implementation:
Order the nodes in fringe in decreasing order of desirability
- Special cases:
 - greedy best-first search
 - A* search

Greedy best-first search

- It tries to expand the node that is closest to the goal state.
- It follows a single path but switch over to a different path if it appears to be more promising at any stage.
- A promising node is selected by applying a suitable HF to each competing node.

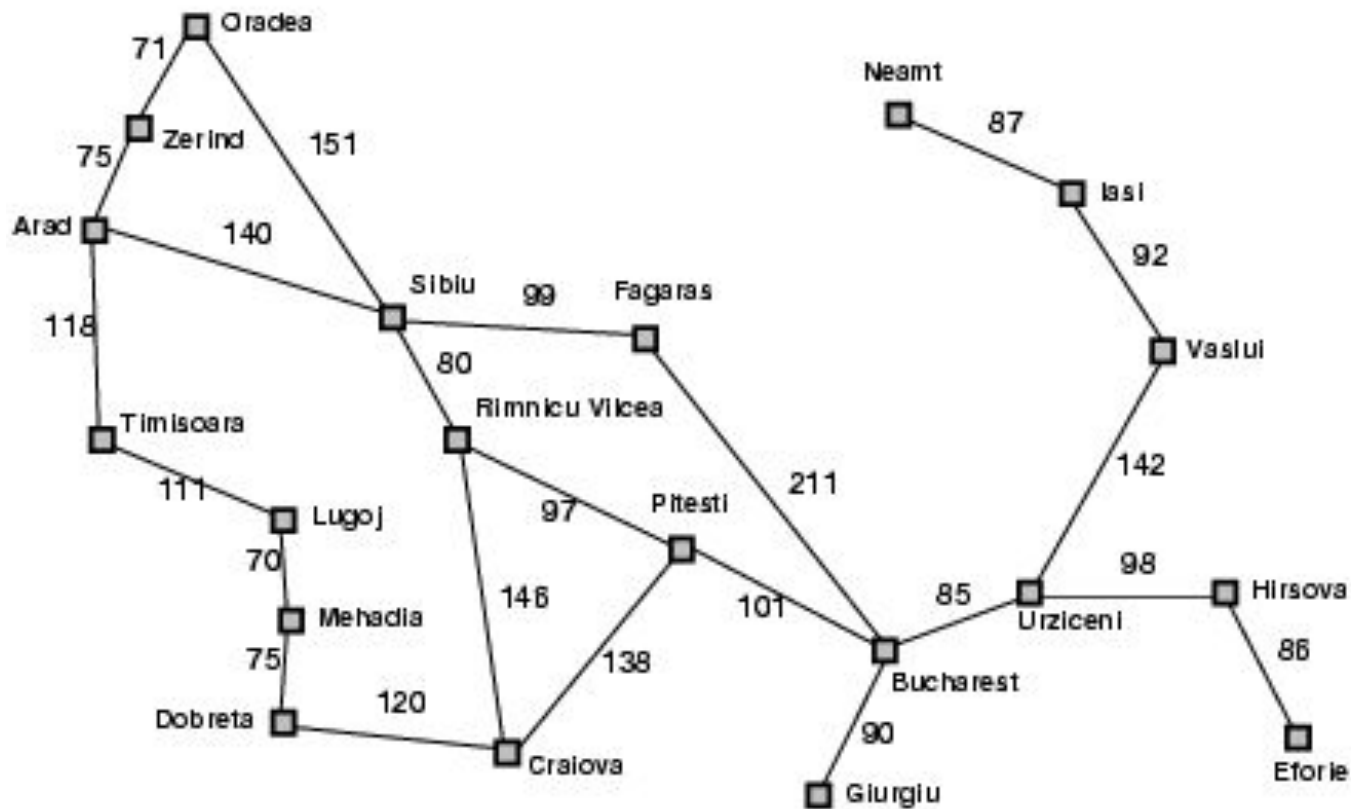
Data structures used for Best-First Search

- OPEN
 - It is a priority queue of nodes which have been generated and have had the heuristic function applied to them but which have not yet been expanded. (priority is evaluated by a HF value).
- CLOSED
 - It contains nodes that have already been expanded. (it is used to ensure that the same node is not expanded more than once.)

Greedy Best-First Search

- Evaluation function $f(n) = h(n)$ (**h**euristic)
- = estimate of cost from 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

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

Best-first search algorithm

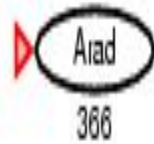
- Start with OPEN containing just the initial state.
1. Until a goal is found or there are no nodes left on OPEN do:
 - a) Pick the best node from OPEN.
 - b) Generate its successors.

Best-first search algorithm

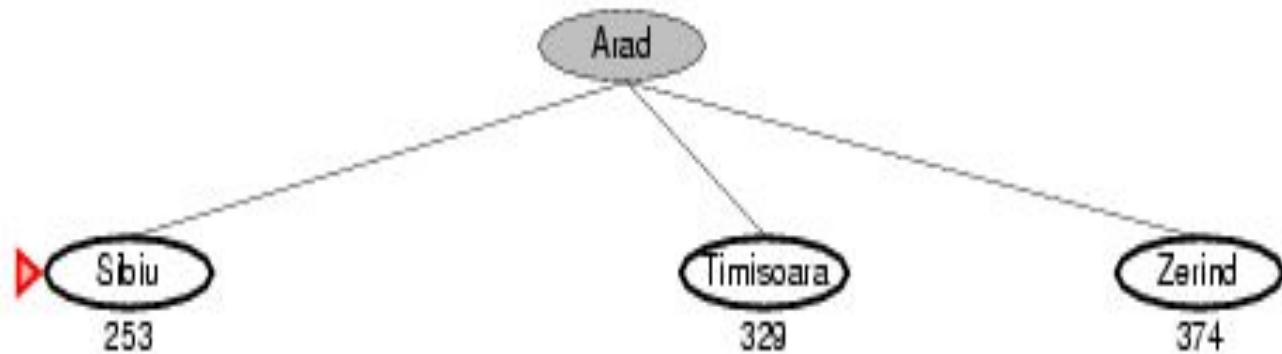
c) For each successor do:

- i) If it has not been generated before , evaluate it, add it to OPEN, and record its parent.
- ii) If it has been generated before, change the parent if this new path is better than the previous one. In that case , update the cost of getting to this node and to any successors that this node may already have.

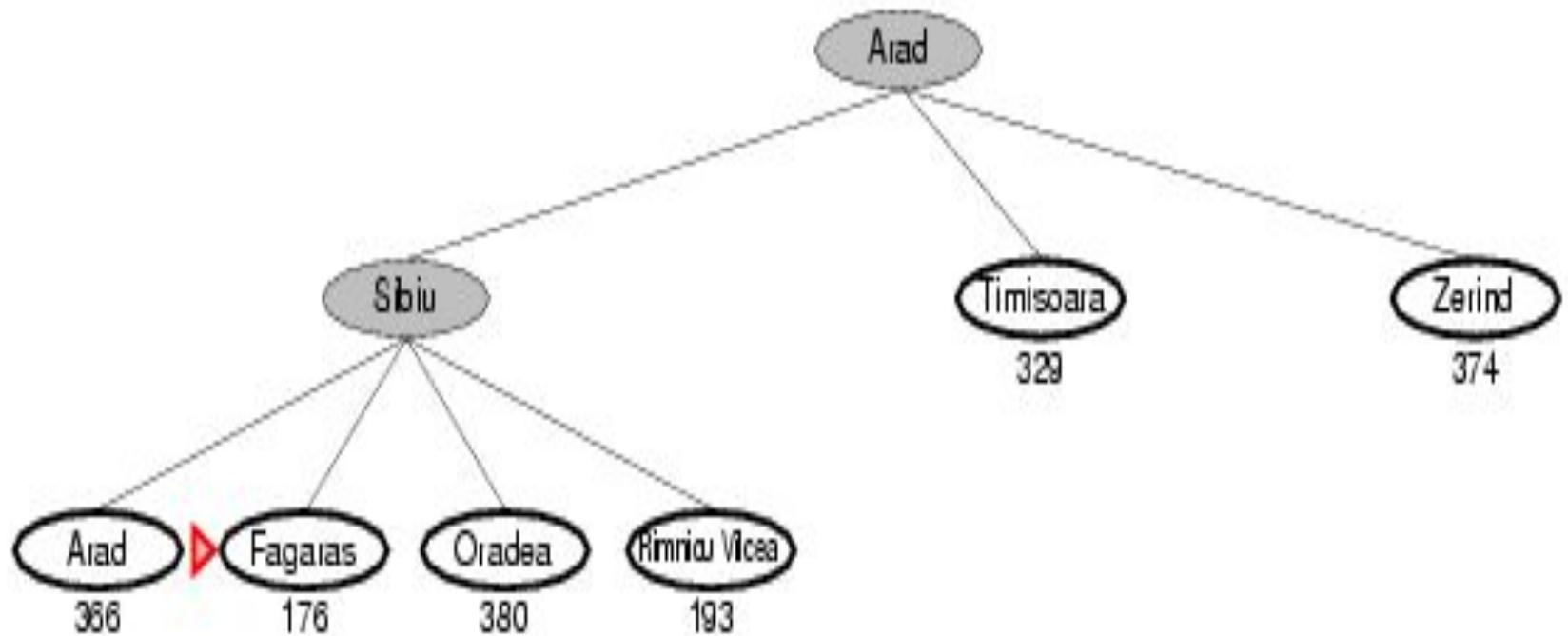
Greedy best-first search example



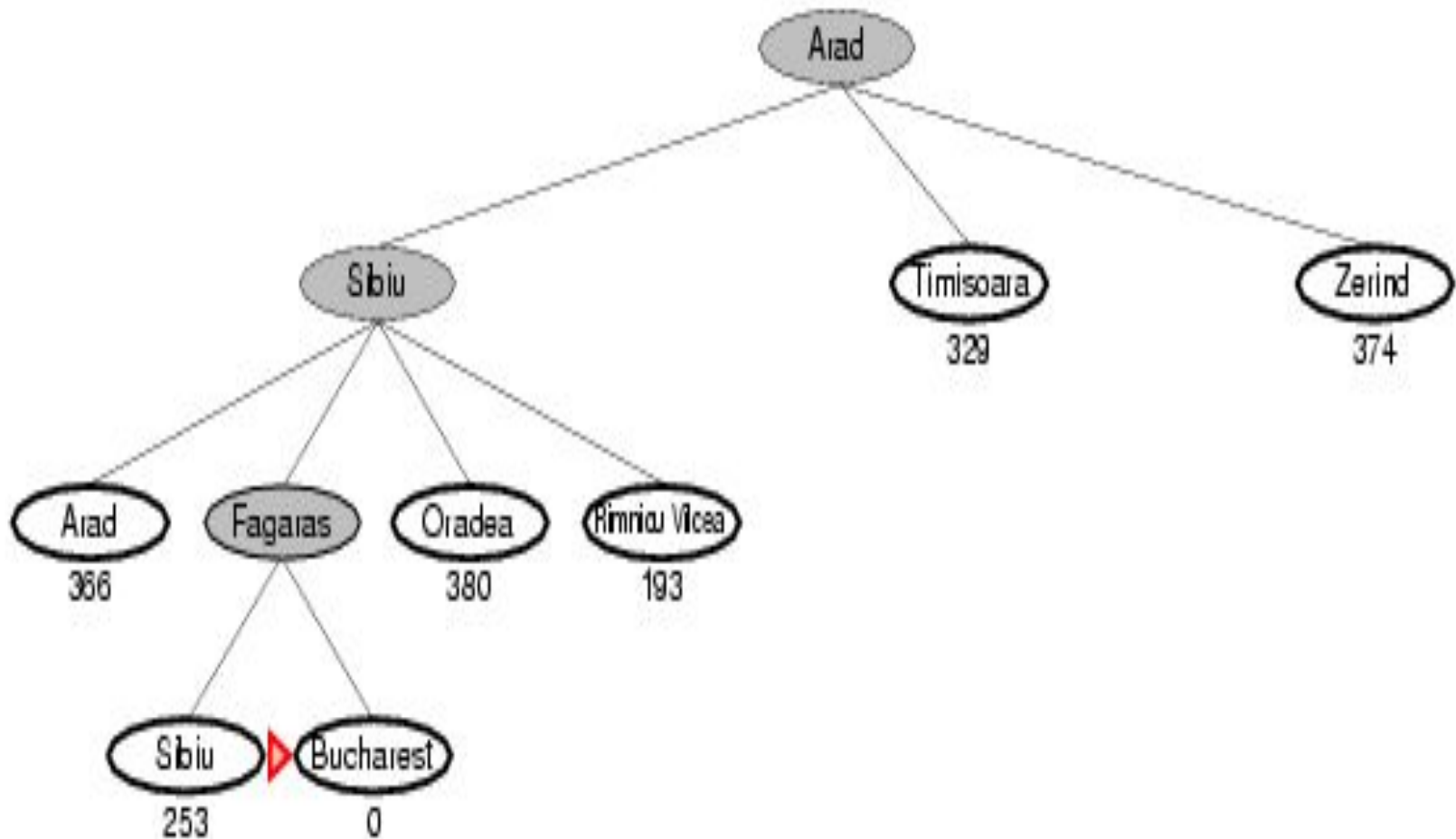
Greedy best-first search example



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., lasi □ Neamt □ lasi □ 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

- 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

- The Best first search algorithm is a simplified version of A* algorithm .
- A* uses the same f, g ,h functions as well as the lists OPEN and CLOSED.
- A* algorithm:-
 - 1) start with OPEN containing only the initial node. Set that node's g value to zero , its h value to whatever it is and its f value to $h + 0 = h$. initially closed is empty.

A* search

2. Until a goal node is found, repeat the procedure:
 1. if there are no nodes on OPEN, report failure. Otherwise, pick the node from OPEN with the lowest f value. Call it BESTNODE. Remove it from OPEN, place it on CLOSED. If BESTNODE is a goal node, then exit and report a solution; else generate the successors of BESTNODE. For each such successor, do the following:

A* search

- a) Set SUCCESSOR to point back to BESTNODE.
- b) Compute $g(\text{SUCCESSOR}) = g(\text{BESTNODE}) +$ the cost of getting from BESTNODE to SUCCESSOR.
- c) See if SUCCESSOR is the same as any node on OPEN. If so, call that node OLD. Since this node already exists in the graph, we can throw SUCCESSOR away and add OLD to the list of BESTNODES successors.

A* search

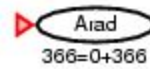
determine the parent link:- If the path just found to SUCCESSOR is cheaper than the current best path to OLD, then reset OLD's parent – link to BESTNODE; else, don't update anything.

d) if SUCCESSOR is not in OPEN but in CLOSED, call it OLD and add OLD to the list of BESTNODE's successors. Check to see if the new path is better as in step (c) . If it is, then set the parent link and g and f value appropriately. Propagate the new cost downward and determine the better path. If required set the parent link accordingly.

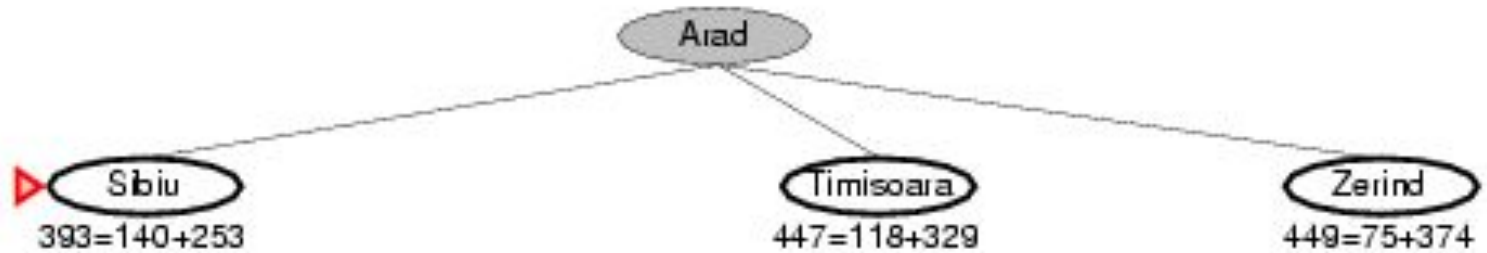
A* search

e) If SUCCESSOR is neither in OPEN nor in CLOSED, then put it on OPEN and add it to the list of BESTNODE's successors. Compute $f(\text{SUCCESSOR}) = g(\text{SUCCESSOR}) + h(\text{SUCCESSOR})$

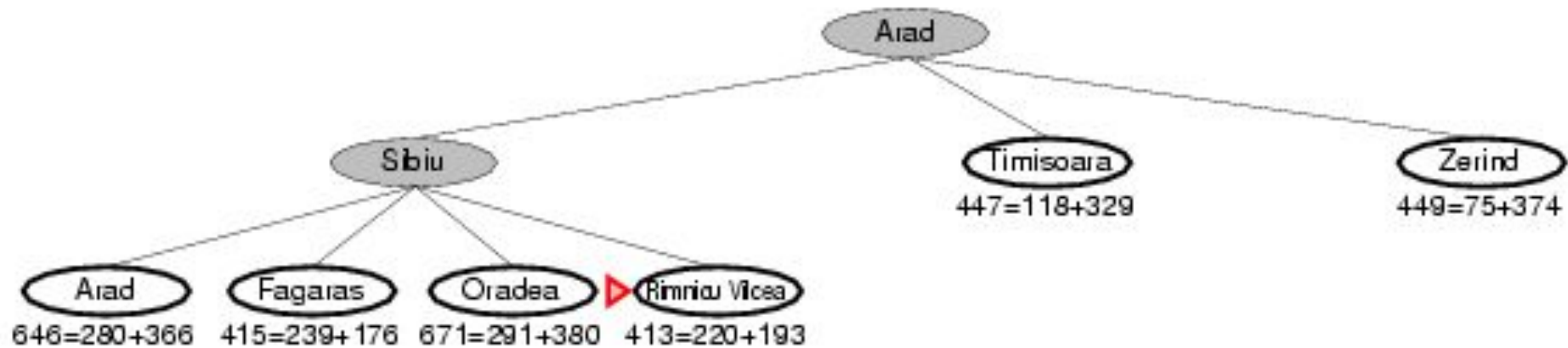
A* search example



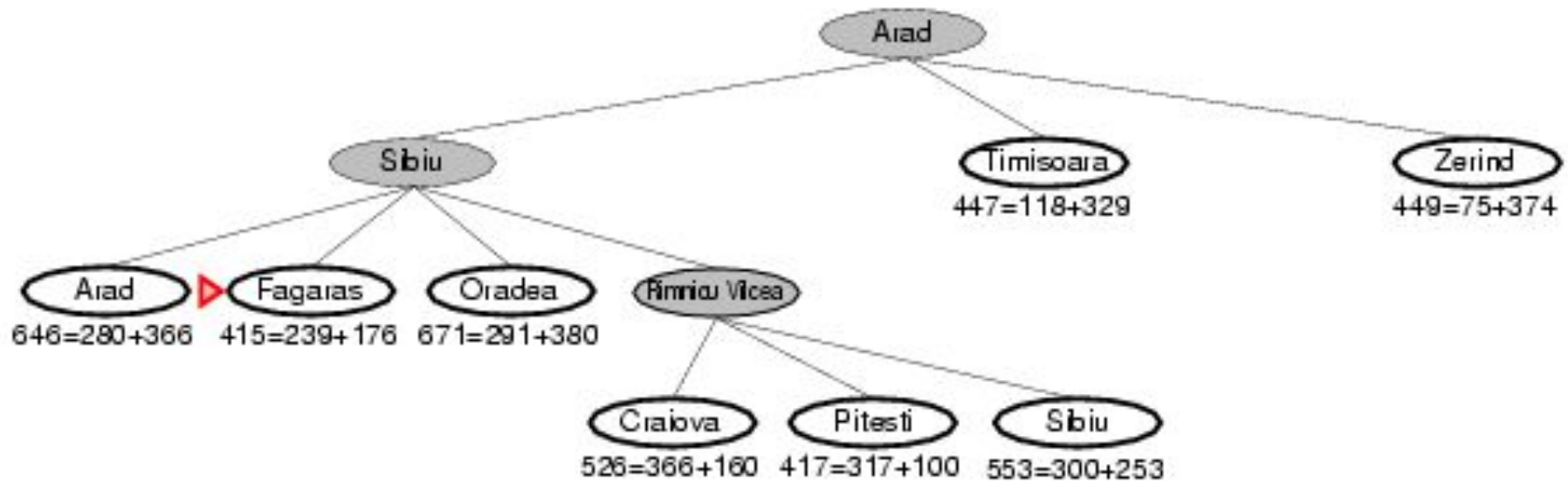
A* search example



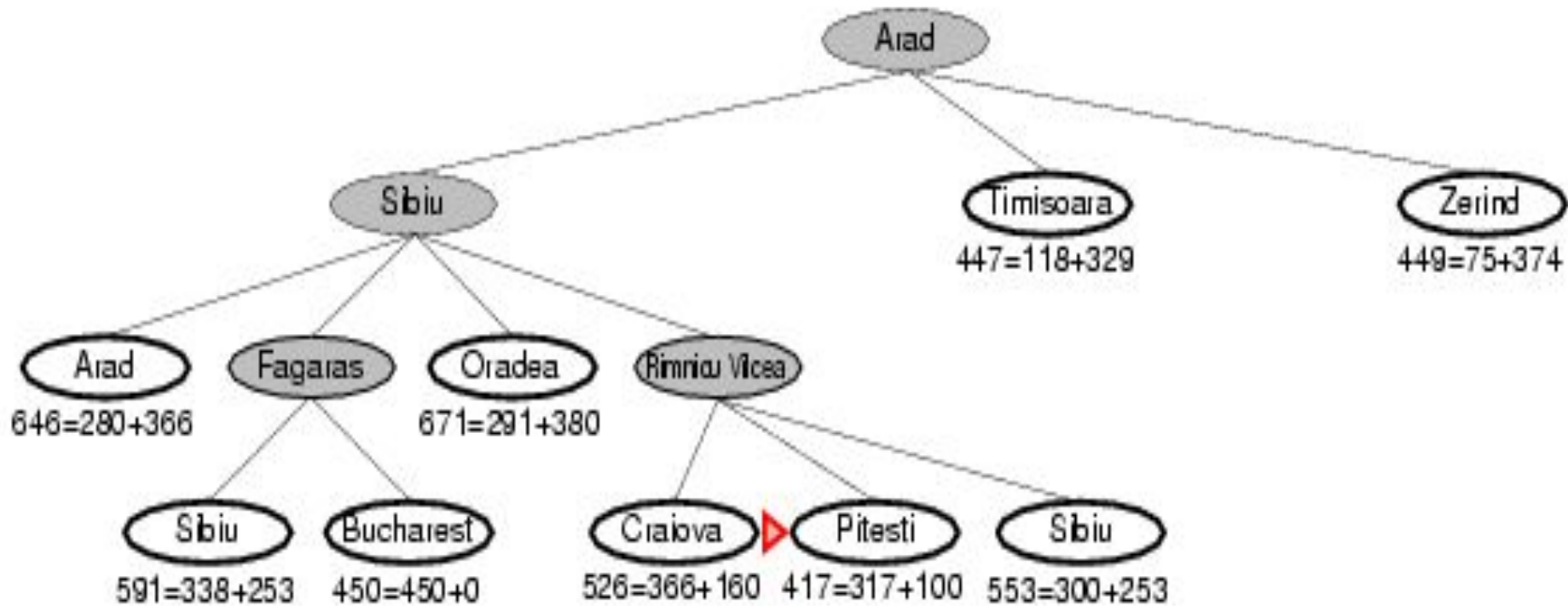
A* search example



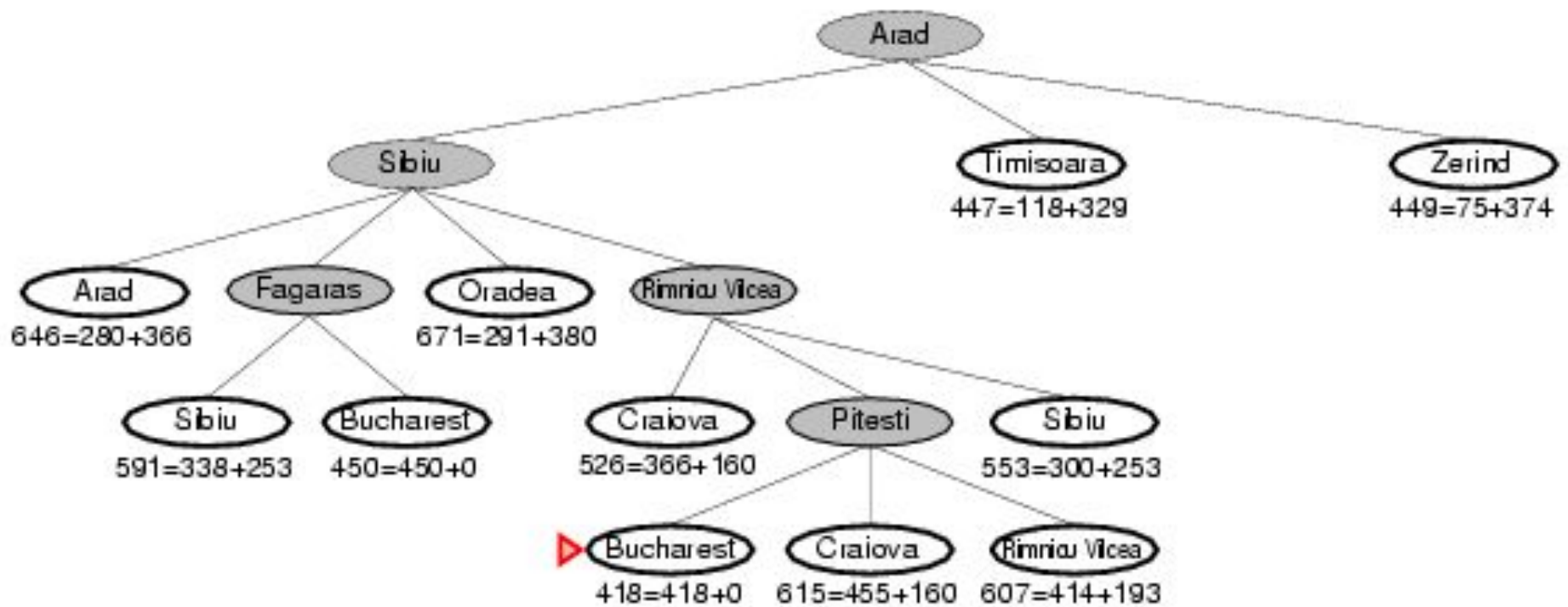
A* search example



A* search example



A* search example



Properties of A*

- Complete? Yes (unless there are infinitely many nodes with $f \leq f(G)$)
- Time? Exponential
- Space? Keeps all nodes in memory
- Optimal? Yes if $h(n)$ is admissible.

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**
- Example: $h_{SLD}(n)$ (never overestimates the actual road distance)
- **Theorem:** If $h(n)$ is admissible, A^* using TREE-SEARCH is optimal

Consistent heuristics

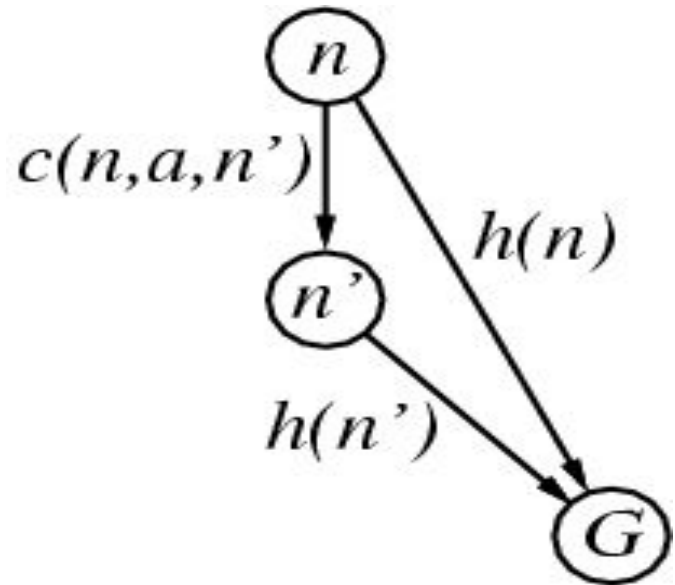
- A heuristic is **consistent** if for every node n , every successor n' of n generated by any action a ,
$$h(n) \leq c(n,a,n') + h(n')$$

- 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



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)

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

- $h_1(S) = ?$
- $h_2(S) = ?$

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)

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

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