331 – Intro to AI
Week 03
Heuristics
R&N Chapter 3 (Section 3.5, 3.6), Chapter 4.1

T.J. Borrelli

Heuristics

- Heuristic: a rule or other piece of information that is used to make methods such as search more efficient or effective
- In search, often use a heuristic evaluation function, f(n):
  - f(n) tells you the approximate distance of a node, n, from a goal node
- f(n) may not be 100% accurate, but it should give better results than pure guesswork

Heuristics

- A heuristic should reduce the number of nodes that need to be examined
- The more *informed* a heuristic is, the better it will perform
- Heuristics are used for solving *constraint satisfaction problems*
  - Generate a possible solution, and then make small changes to bring it closer to satisfying constraints

Best-First Search

- Pick the *most likely* node (based on some heuristic value) from the partially expanded tree at each stage
- Tends to find a path more quickly/efficiently than depth-first or breadth-first search, but does not guarantee to find the best path
- This is a greedy algorithm – may converge to local optimum

Best-First Search

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

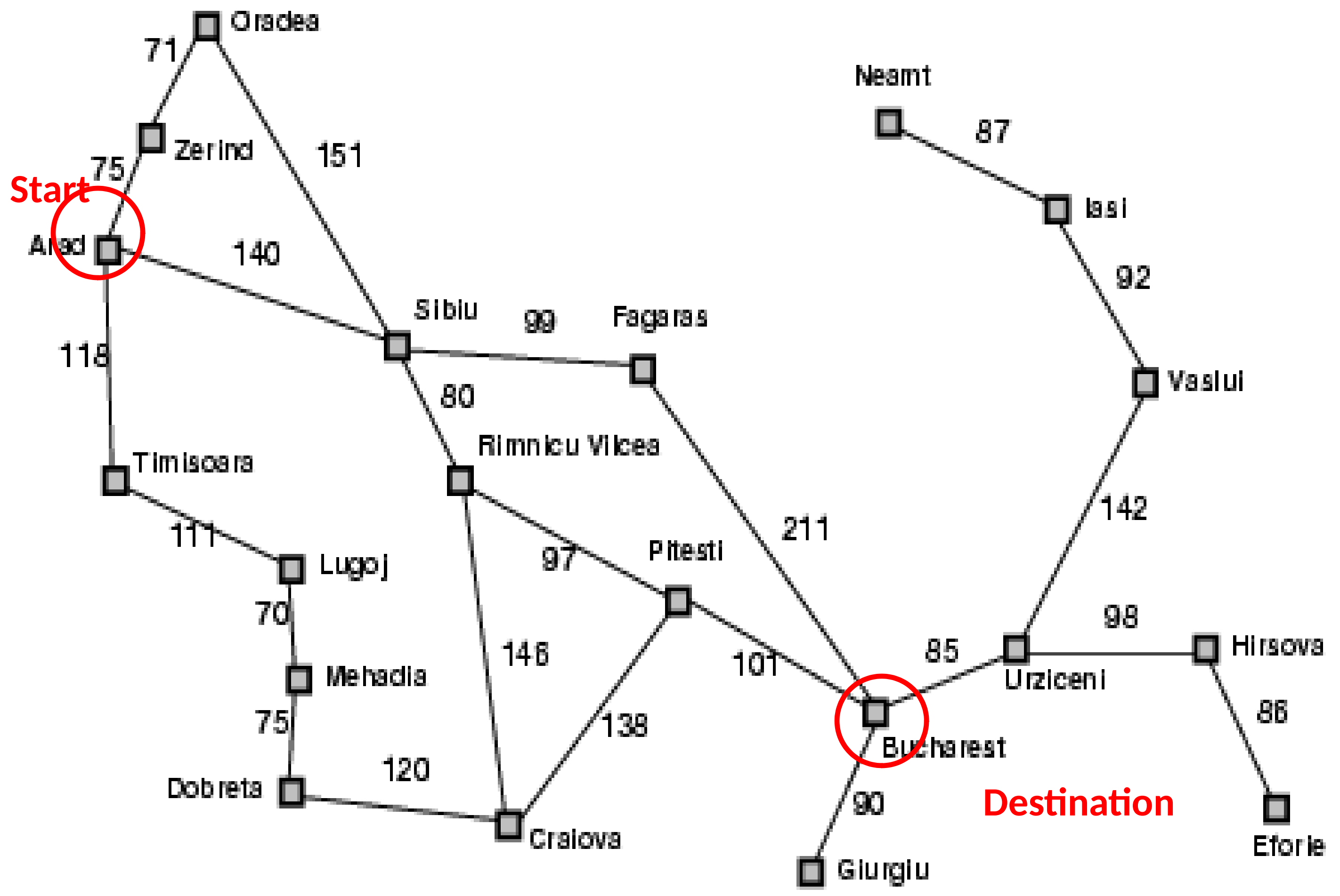
Greedy Best-First Search

- Greedy Best-First Search always expands the node that appears to be closest to the goal
- Not optimal, and not guaranteed to find a solution at all
- Can easily be fooled into taking poor paths

Greedy Best-First Search

- Evaluation function *$f(n) = h(n)$* (heuristic)
- The heuristic is the estimate of cost from node *n* to the goal
  - e.g., $h_{SLD}(n)$ = straight-line distance (“as the crow flies”) from start node to destination node
- Greedy best-first search expands the node that *appears* to be closest to goal

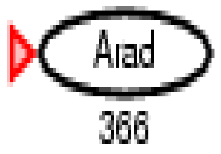
Romania with Step Costs in km



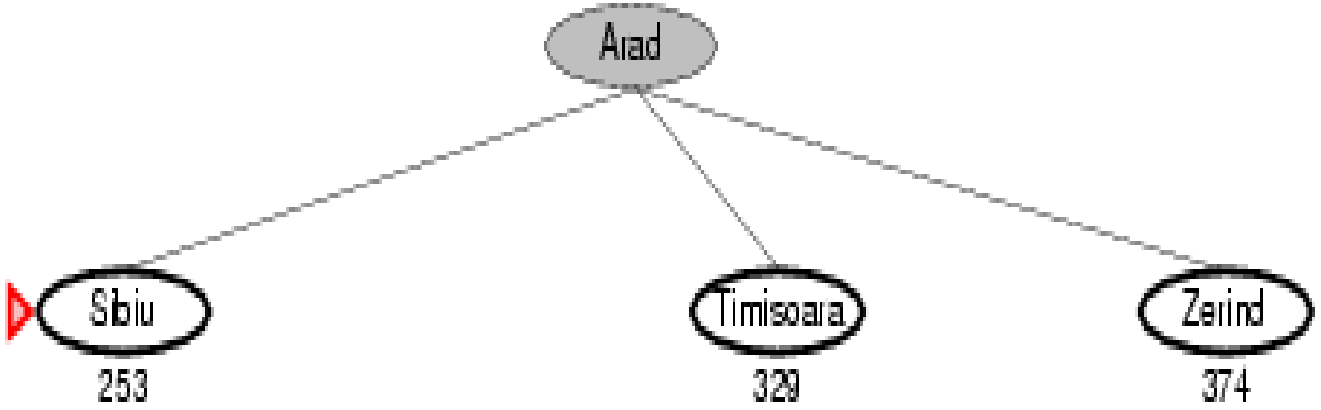
Straight–line distance to Bucharest

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

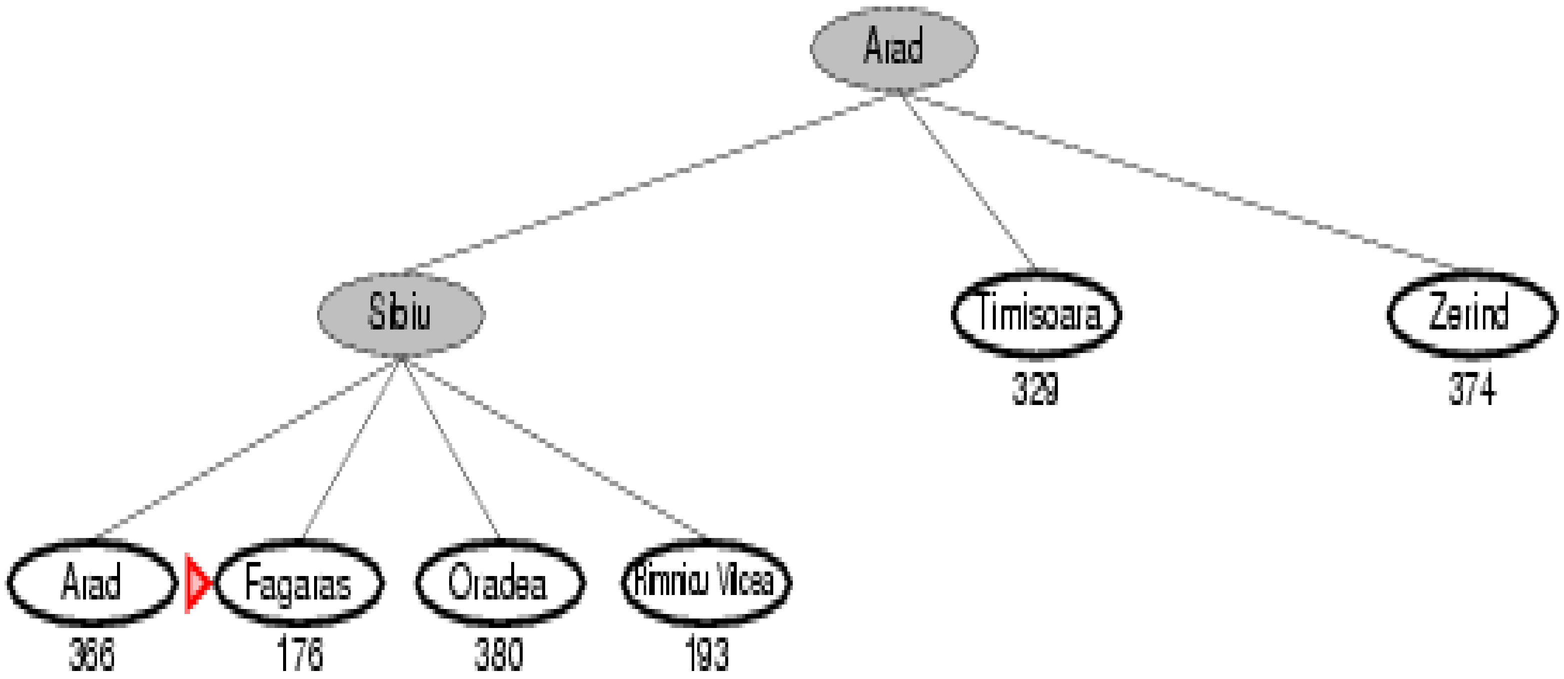
Greedy Best-First Search Example



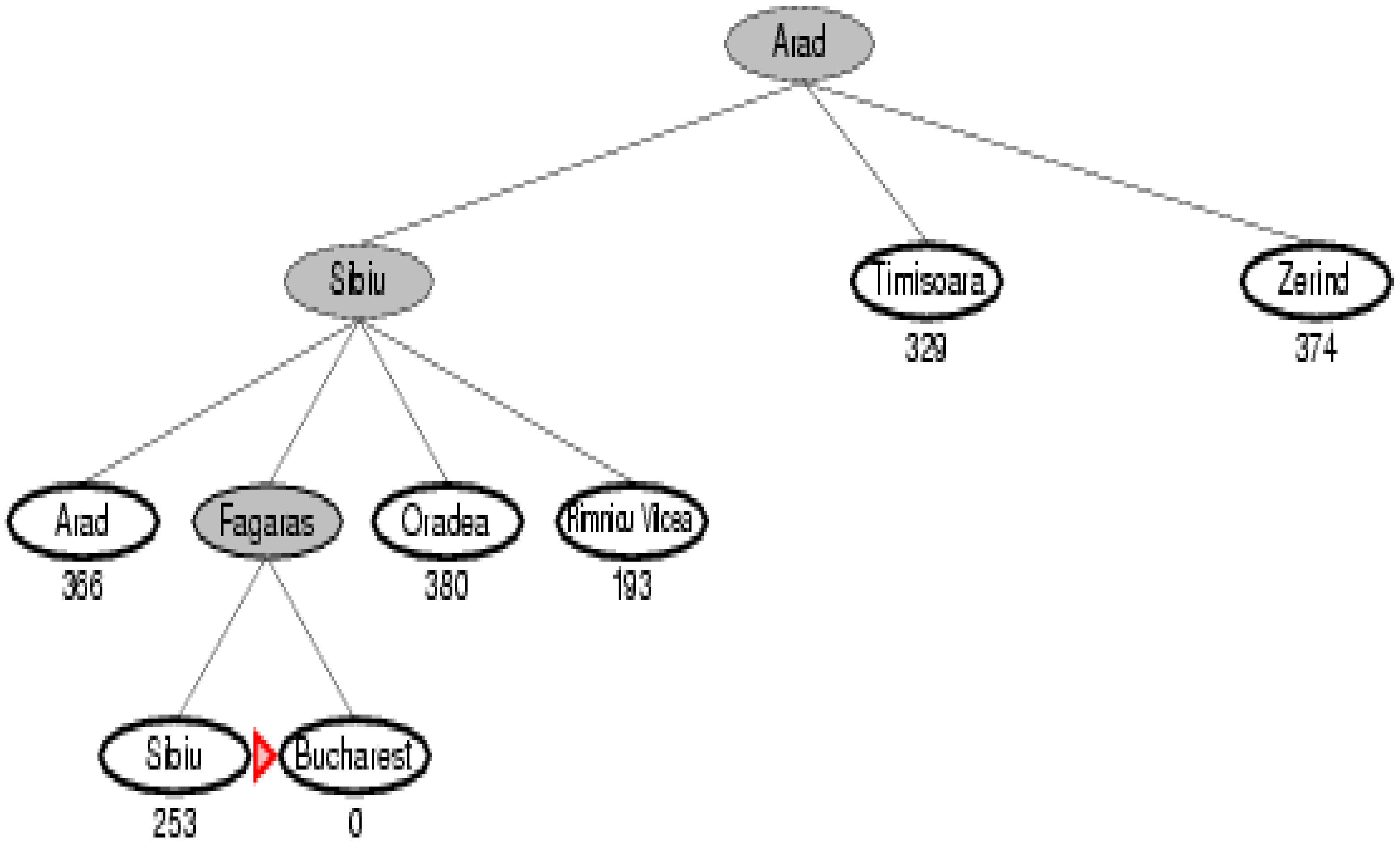
Greedy Best-First Search Example



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 → etc.
- Time?
  - $O(b^m)$, but a good heuristic can give dramatic improvement
- Space?
  - $O(b^m)$ -- keeps all nodes in memory
- Optimal?
  - No

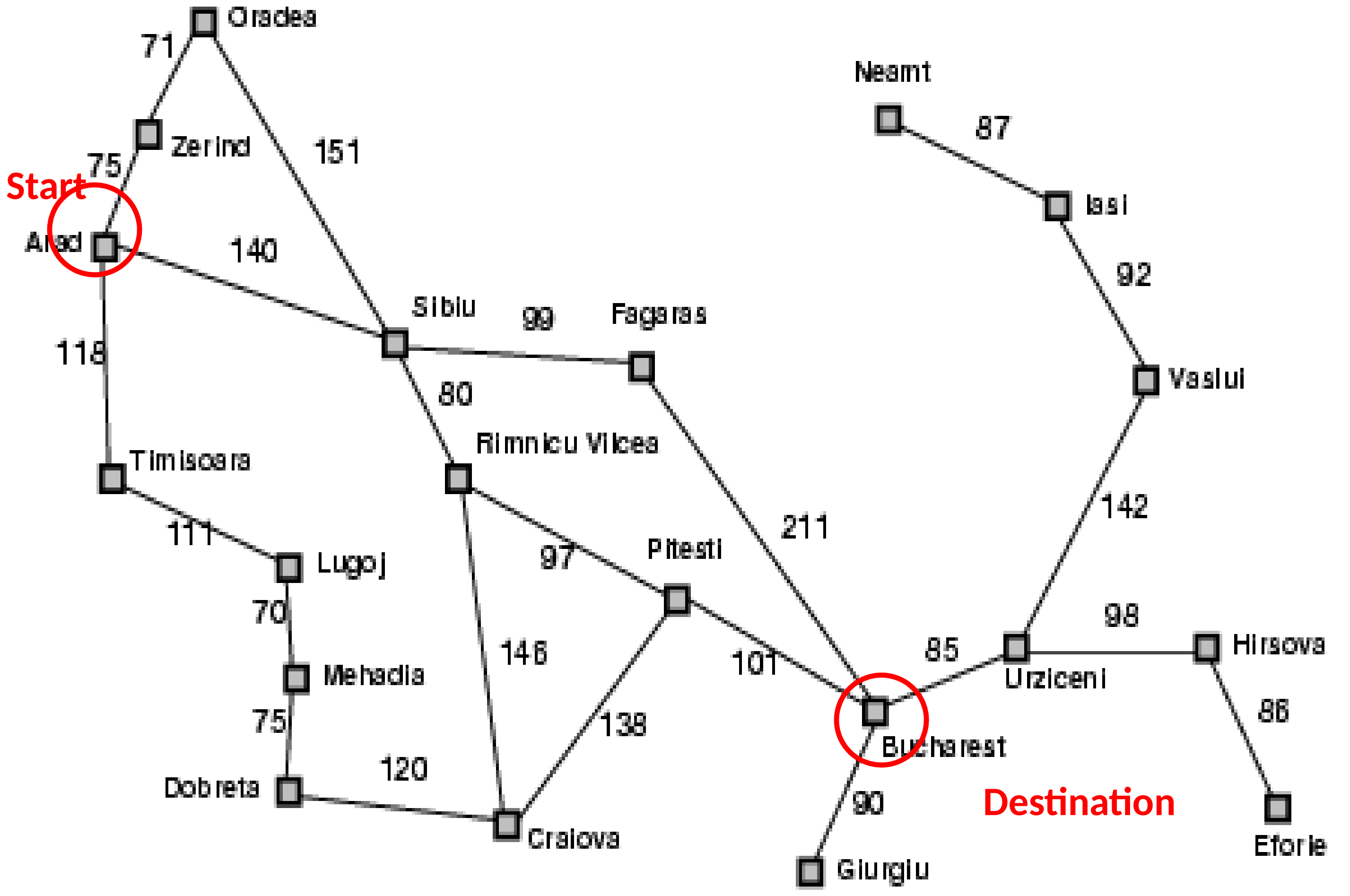
A* Search

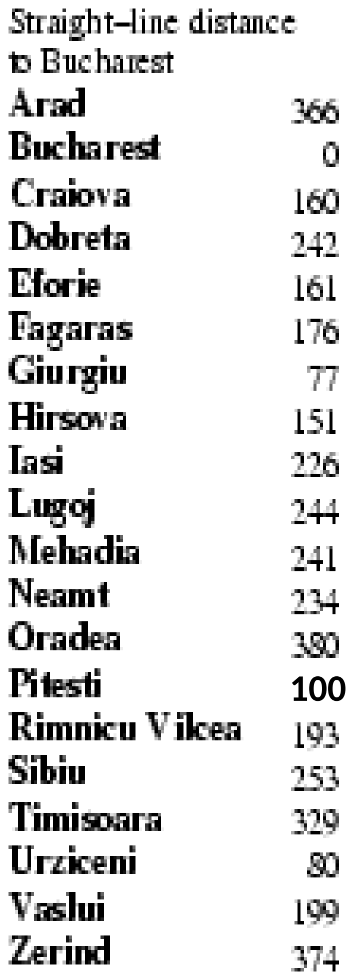
- Idea: Avoid expanding paths that are already expensive
- Evaluation function $f(n) = g(n) + h(n)$
- $f(n)$ = estimated total cost of path through n to goal
- $g(n)$ = cost so far to reach n
- $h(n)$ = estimated cost from n to goal

A* Search Example

- Choose h(n) to be the straight-line distance to the goal(Bucharest), as with Greedy Best-First
- Choose g(n) to be distance from the start node to the current node

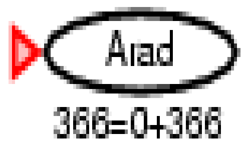
A* Search Example



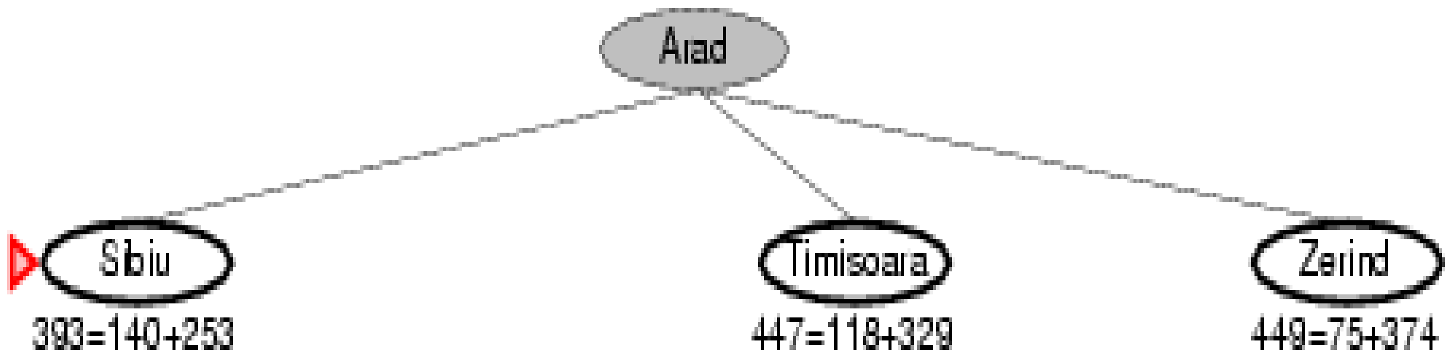


Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

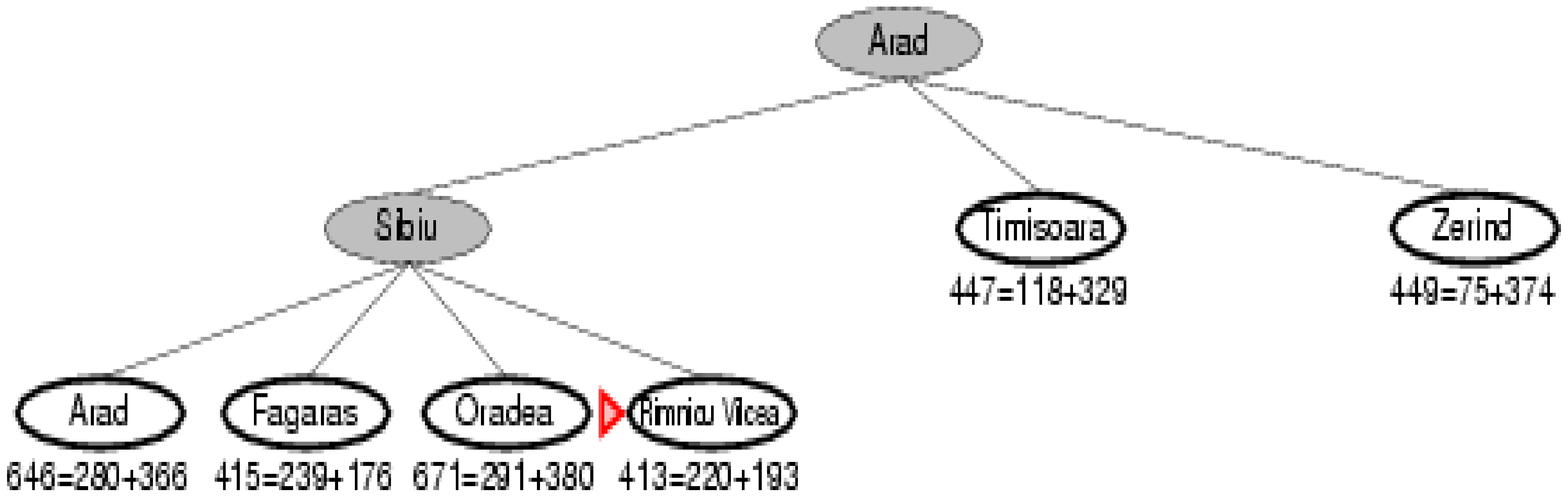
A* Search Example



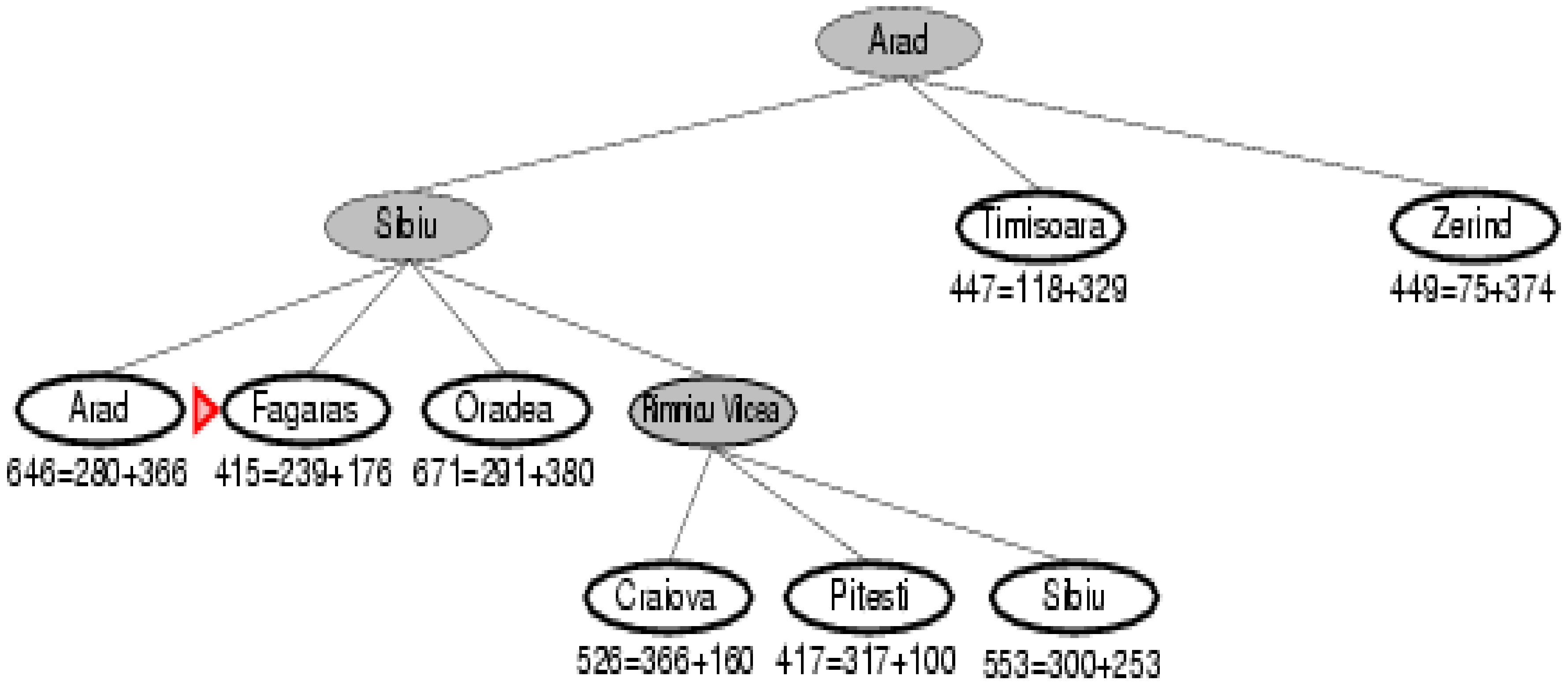
A* Search Example



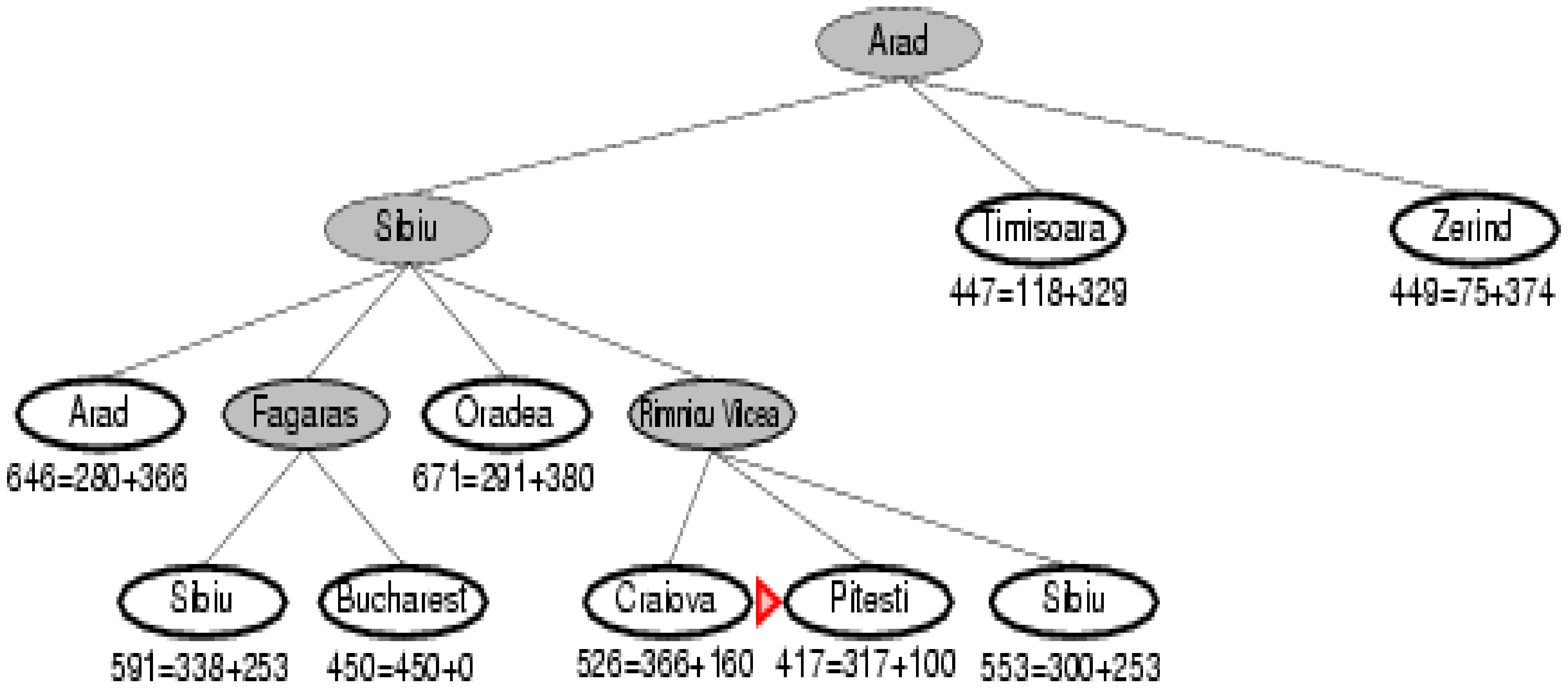
A* Search Example



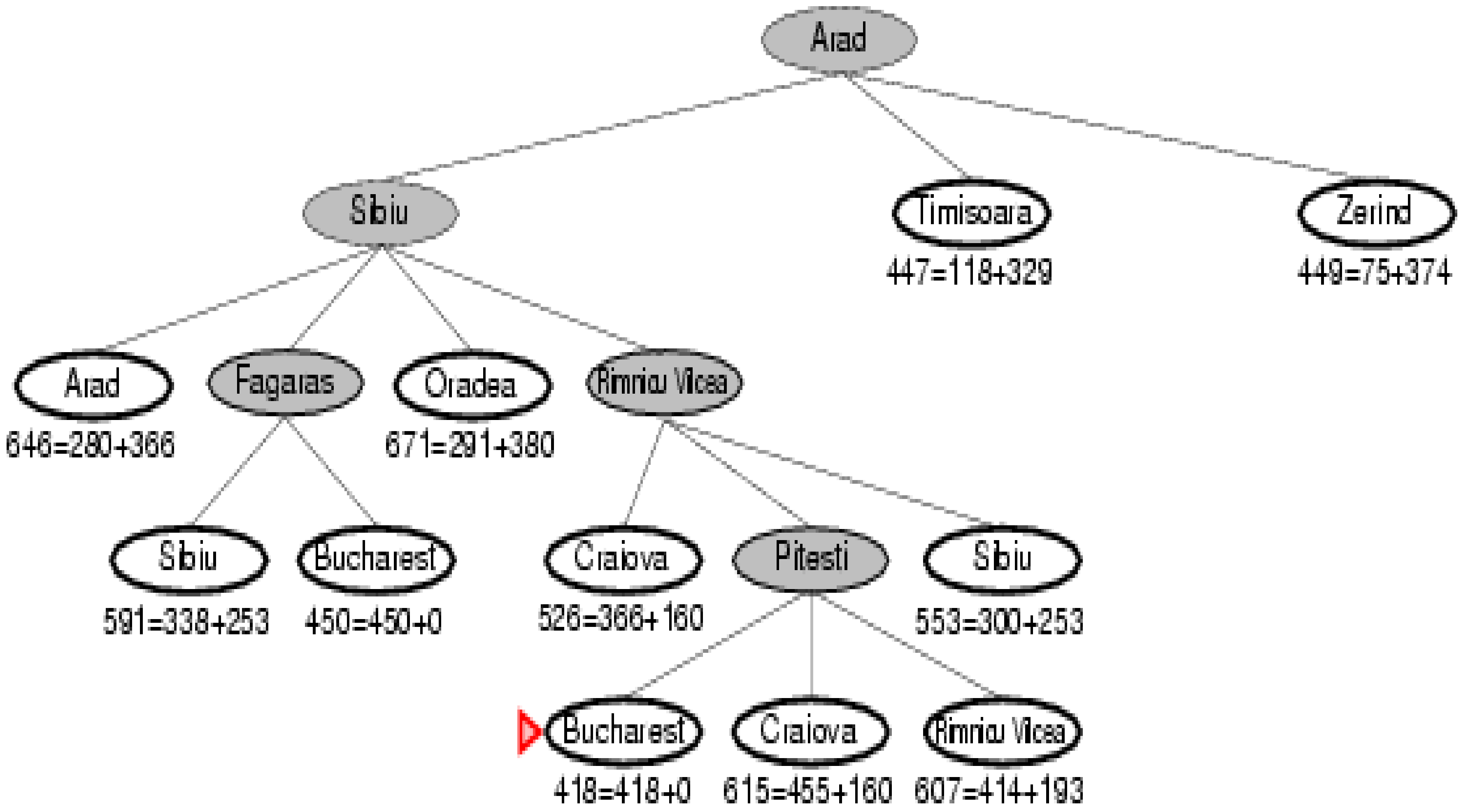
A* Search Example



A* Search Example



A* Search Example



A* Search

- A* algorithms are optimal:
  - They are guaranteed to find the shortest path to a goal node, provided *h* is never an overestimate
    - i.e., *h* is an *admissible heuristic*
- A* methods are also optimally efficient – they expand the fewest possible paths to find the right one
- If *h* is not admissible, the method is called A, rather than A*

Admissible Heuristics

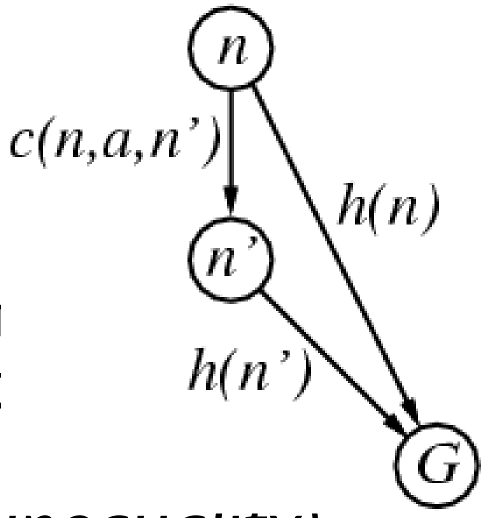
- A heuristic *h(n)* is admissible if for every node *n*, *h(n) ≤ h*(n)*, where *h*(n)* is the true cost to reach the goal state from *n*
- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic
- Example: *h(n) = "straight-line distance"* never overestimates the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

Admissible Heuristics

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

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

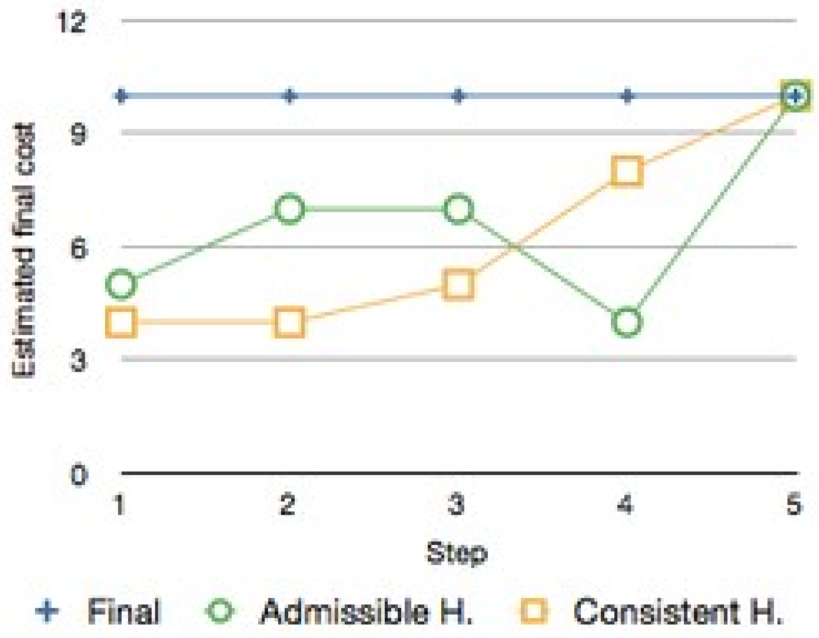
- If *h* is consistent, then for every node *n* and every successor *n'* of *n* generated by any action *a'*, the estimated cost of reaching the goal fron *n* is no greater than the step cost of get to n' plus the estimated cost of reachin goal from n' (also know as the *triangle inequality)*



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

Admissible Heuristics

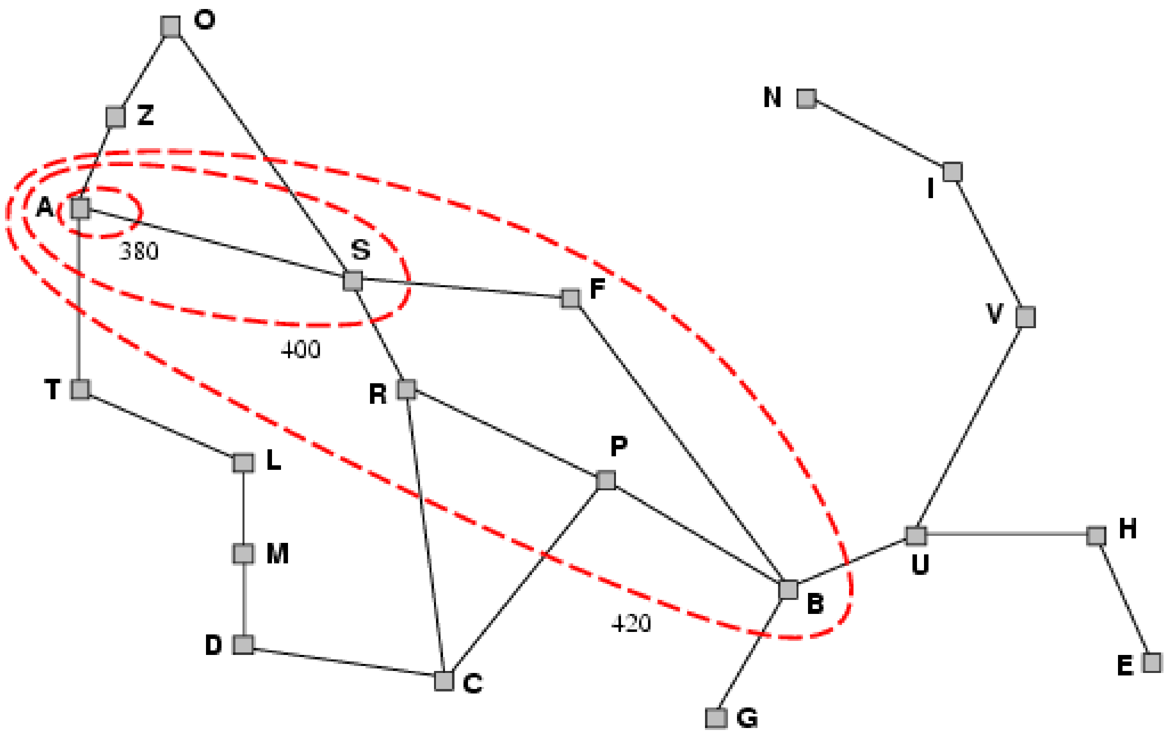
- All consistent heuristics are admissible, but not all admissible heuristics are consistent



http://en.wikipedia.org/wiki/Consistent_heuristic

Admissible Heuristics

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

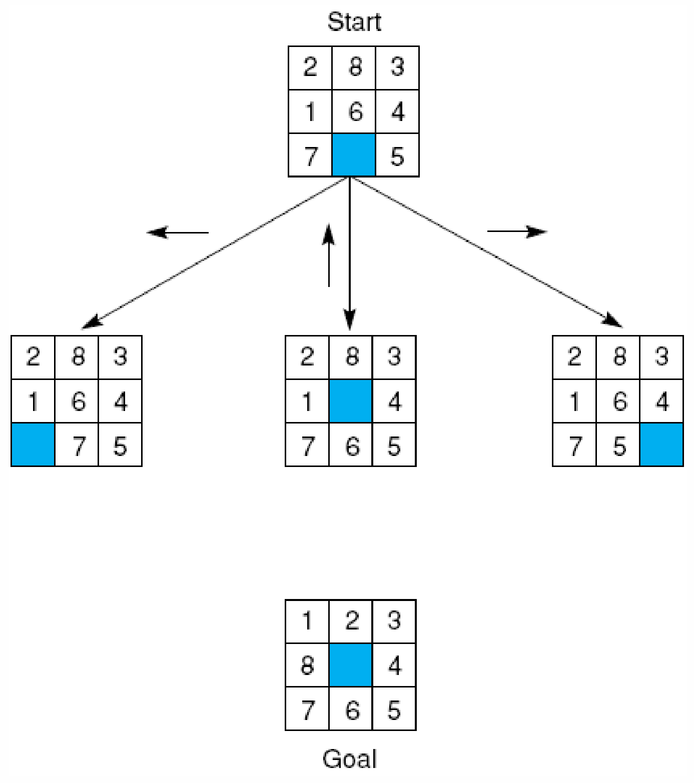


Another note on A*

- The choice of heuristic function h(n) can change the behavior of A*
- We can use different h(n) values or weights
- You may have noticed that h(n) = 0 is admissible (since h(n) <= h*(n) the actual cost)
- This is one extreme in which only g(n) plays a role and reduces A* to Dijkstra's shortest path algorithm, which is guaranteed to find the shortest path, but we may lose some efficiency
- On the other extreme is to use h(n) such that it is very high relative to g(n), in this case A* is reduced to Greedy Best First Search (remember that this is not optimal nor complete)

Other Admissible Heuristics

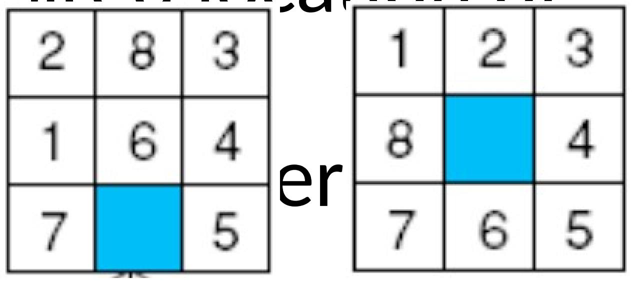
The start state, first move, and goal state for our example 8-puzzle:



Other Admissible Heuristics

For the 8-puzzle:

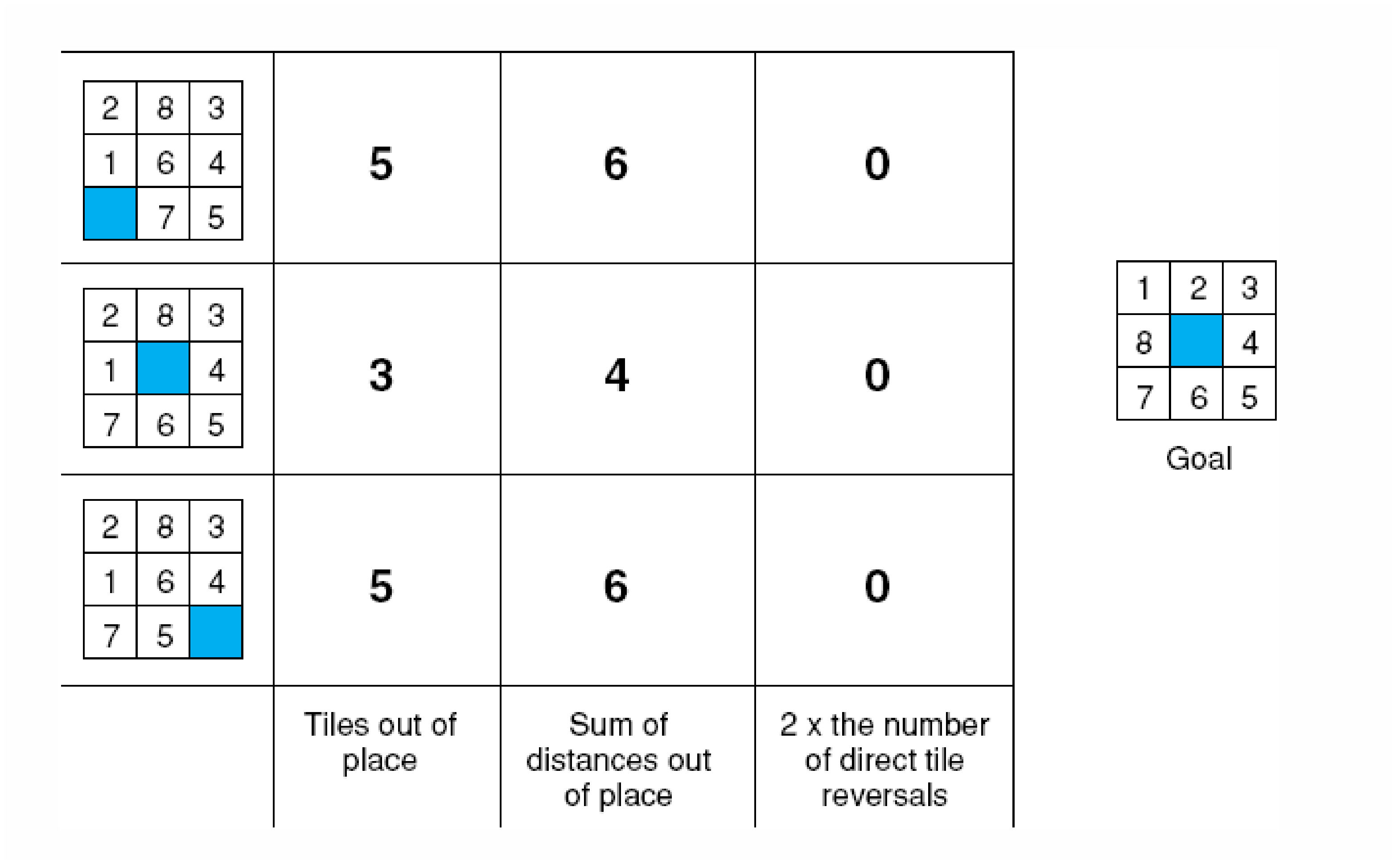
- *$h_1(n)$* = number of misplaced tiles
- *$h_2(n)$* = total Manhattan distance (i.e., the number of squares from desired location of each tile)
- *$h_3(n)$* = 2 * number of direct er



Start Goal

- $h_1(S) = 4$
- $h_2(S) = 1+1+0+0+0+2+0+2 = 6$
- $h_3(S) = 0$

Other Admissible Heuristics

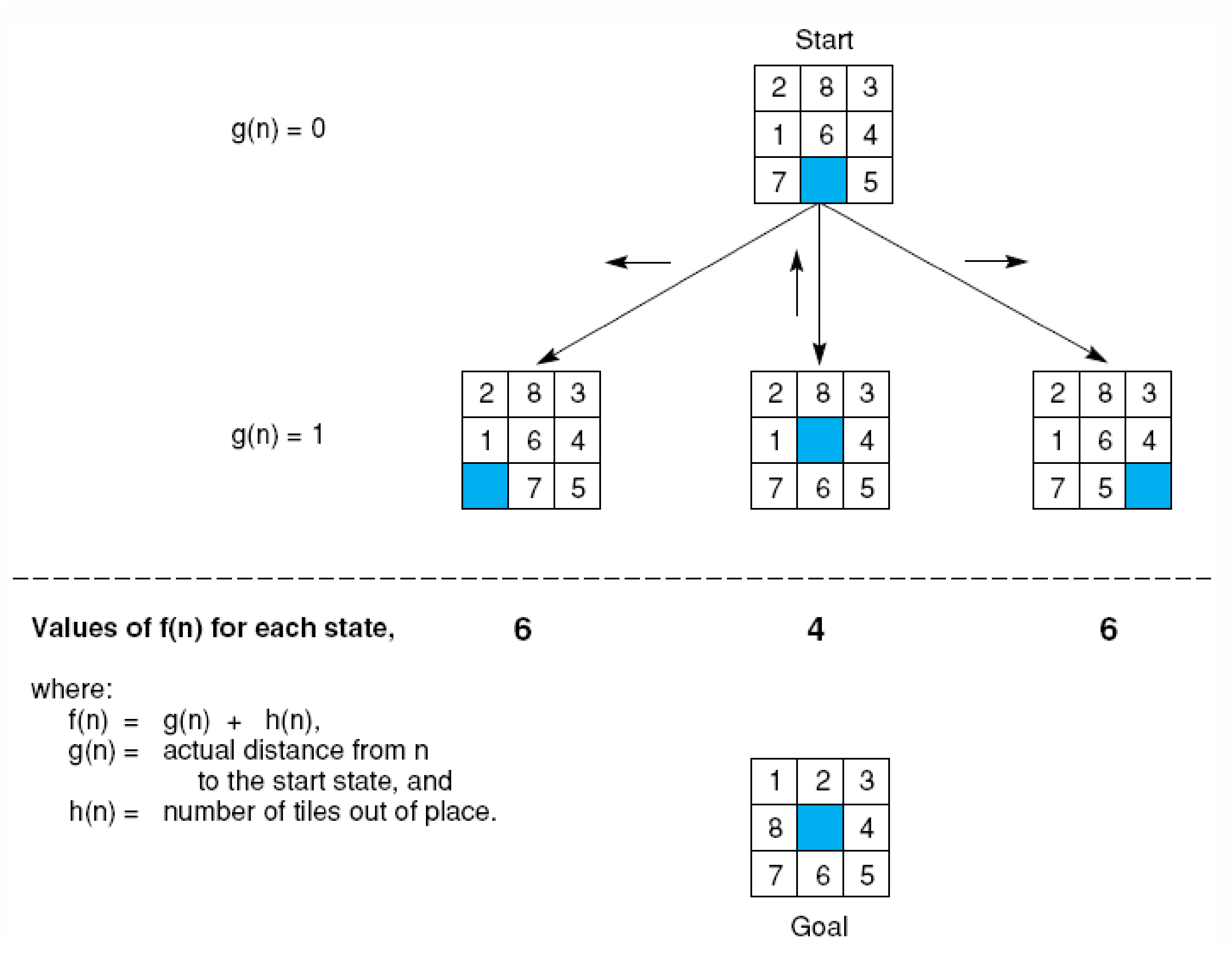


	Tiles out of place	Sum of distances out of place	2 x the number of direct tile reversals
2 8 3 1 6 4 🟦 7 5	5	6	0
2 8 3 1 🟦 4 7 6 5	3	4	0
2 8 3 1 6 4 7 5 🟦	5	6	0

1	2	3
8	🟦	4
7	6	5

Goal

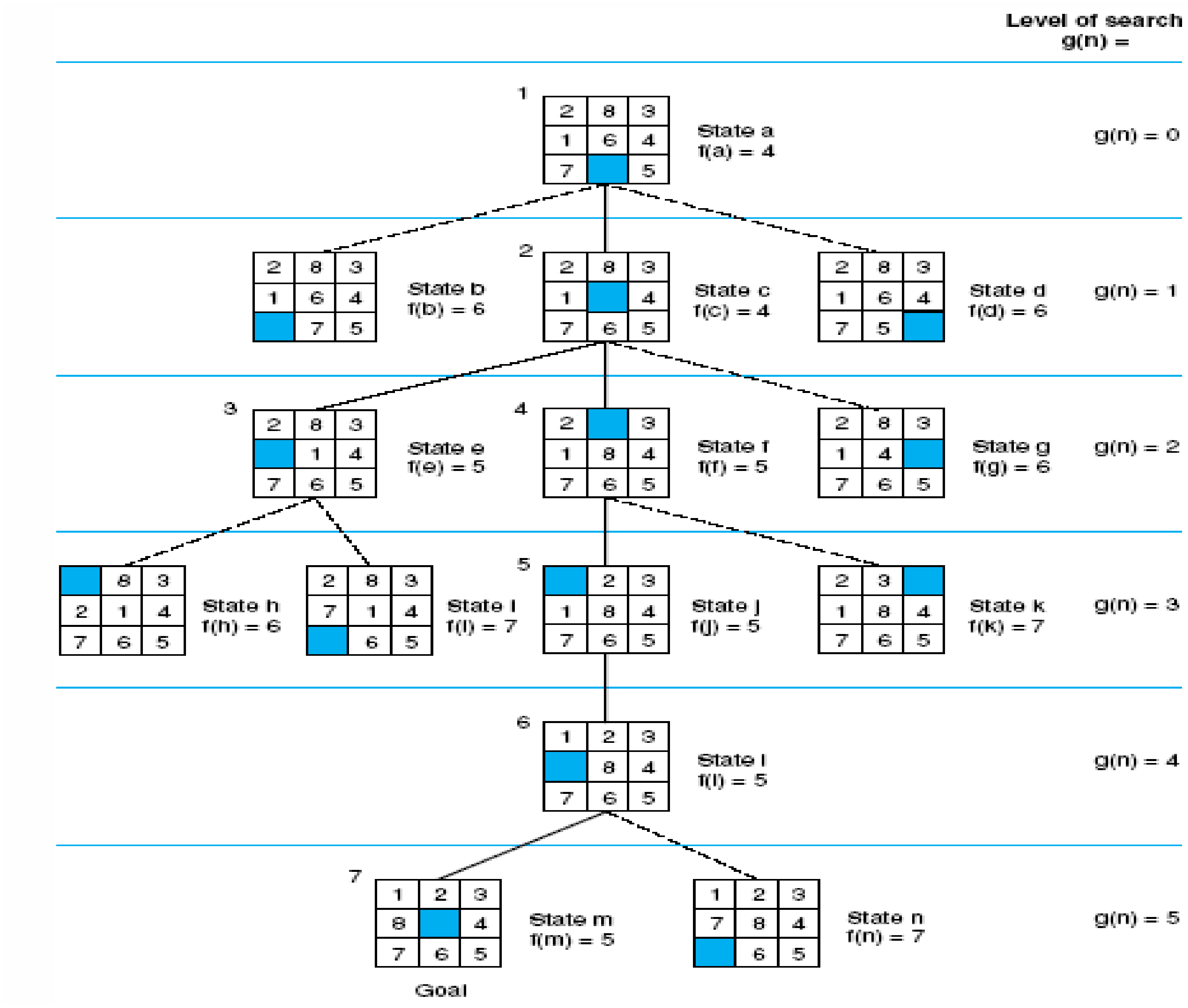
Other Admissible Heuristics



Values of f(n) for each state, **6** **4** **6**

where:
f(n) = g(n) + h(n),
g(n) = actual distance from n to the start state, and
h(n) = number of tiles out of place.

Other Admissible Heuristics



Properties of A*

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

Dominance

- If $h_2(n) \geq h_1(n)$ for all *n* (and both are admissible) then h_2 dominates h_1 (h_2 is better)
- Typical search costs (average number of nodes expanded) for iterative deepening search (IDS), A*(h_1) and A*(h_2) for the 8-puzzle :
- *depth = 12:* IDS = 3,644,035 nodes
- A*(h_1) = 227 nodes
- A*(h_2) = 73 nodes
- *depth = 24:* IDS = too many nodes (intractable!)
- 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 exact solution
- If the rules are relaxed so that a tile can move to any adjacent square, then $h_2(n)$ gives the exact solution

Relaxed Problems

- If a problem is written down in a formal language, it is possible to construct heuristics automatically. Consider the following rule:
  - “A tile can move from square A to square B if A is horizontally or vertically adjacent to B and B is blank”
- We can generate three heuristics by removing one or both of the conditions from the above rule:
  a) “A tile can move from square A to square B”
  b) “A tile can move from square A to square B if A is adjacent to B”
  c) “A tile can move from square A to square B if B is blank”

Constraint Satisfaction Problems

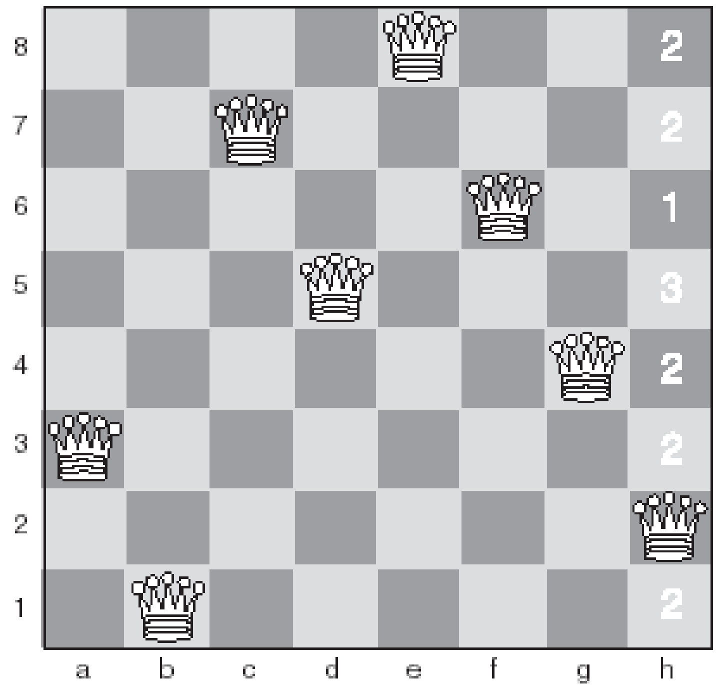
- A constraint satisfaction problem is a combinatorial optimization problem with a set of constraints
- Can be solved using search
- With many variables, it is essential to use heuristics (brute-force is too costly!)

The Eight Queens Problem

- Example of a constraint satisfaction problem:
  - Place eight queens on a chess board so that no two queens are on the same row, column or diagonal
- Can be solved by search, but the search tree is large
- *Heuristic repair* (constraint satisfaction) is very efficient at solving this problem

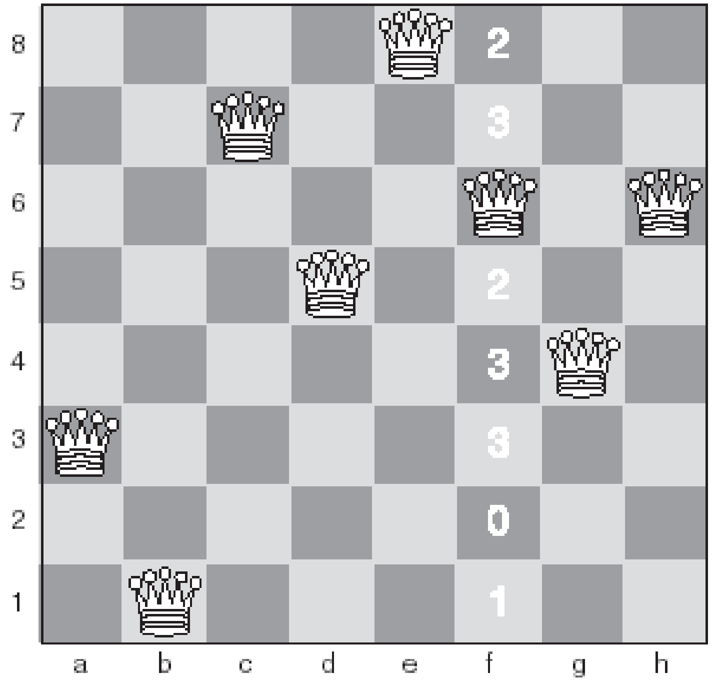
The Eight Queens Problem

- Initial state – one queen is conflicting with another (locations c7 and h2)
- Constrain the problem by only moving a queen in the same column (column h)
- The numbers in the right-most squares indicate how many conflicts will arise if we move the queen at h2 to that row
- Move the queen at h2 to the square with the fewest conflicts (h6)



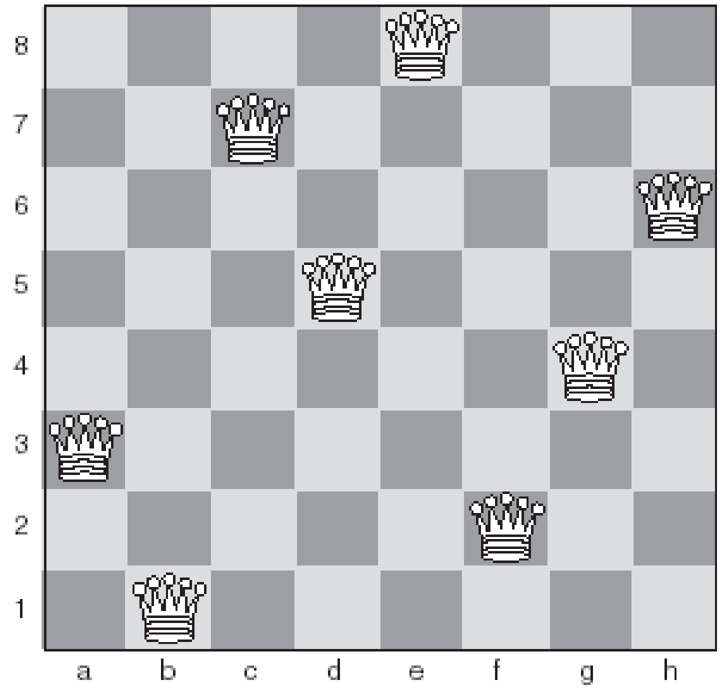
The Eight Queens Problem

- Second state – after moving the queen at h2 to h6
- Now the queen at f6 is conflicting with the queen at h6, so we'll move f6 to the square with the fewest conflicts (f2)



The Eight Queens Problem

- Final state – a solution!



Local Search Algorithms

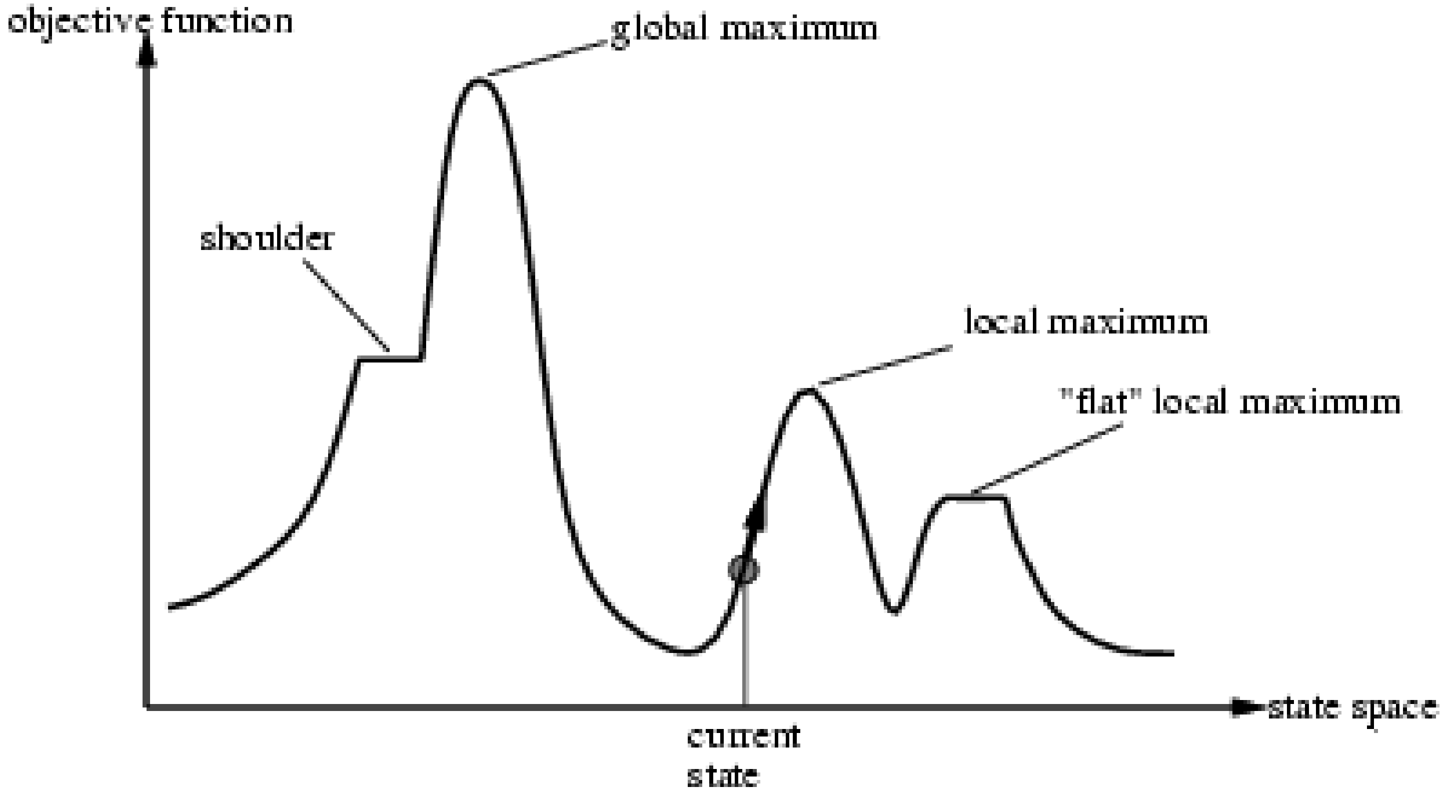
- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution
- Like heuristic repair, local search methods start from a random state, and make small changes until a goal state is achieved
- Most local search methods are susceptible to local maxima, like *hill-climbing*

Hill-Climbing

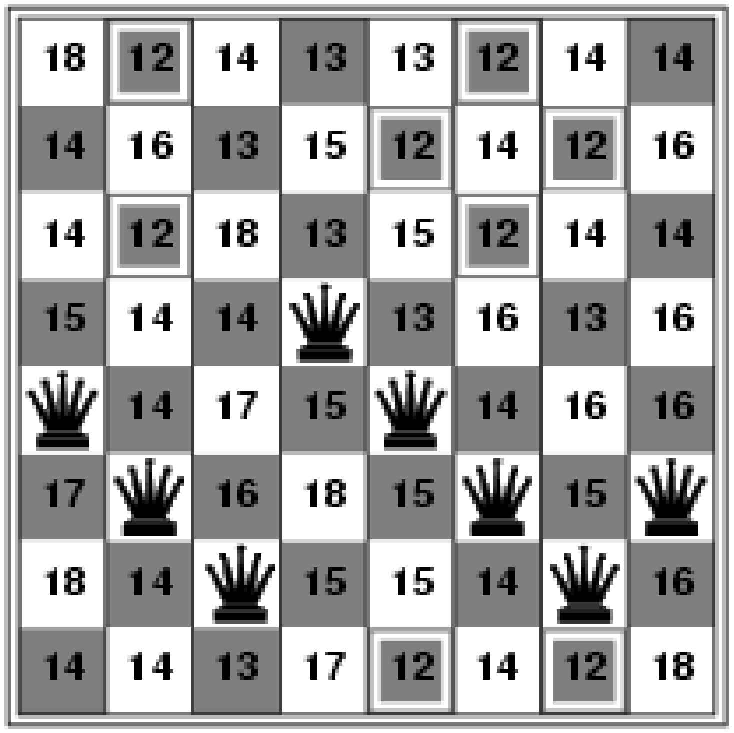
- Hill-climbing is an informed, irrevocable (choices cannot be “un-done”) search method (once you go down a path, there's no turning back)
- Easiest to understand when considered as a method for finding the highest point in a three dimensional search space:
  - Check the height one unit away from your current location in each direction
  - As soon as you find a position whose height is higher than your current position, move to that location, and restart the algorithm

Hill-Climbing

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



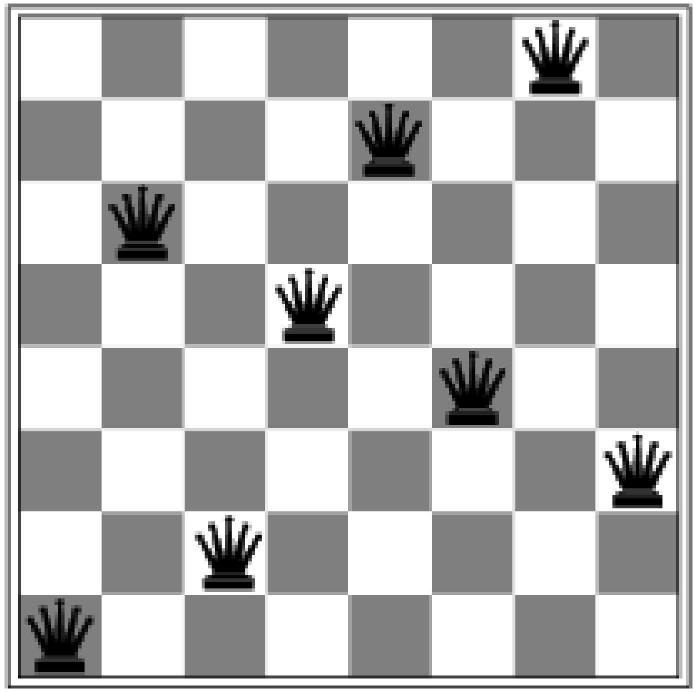
Hill-Climbing: 8-Queens Problem



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 state shown here
- The number in each square indicates how many conflicts will result by moving the queen in that column to the indicated row

Hill-Climbing: 8-Queens Problem


A local minimum with *h = 1*

Improved Heuristics For The Eight Queens Problem

- One way to solve this problem is to have a tree that is 8-ply deep (a *ply* is a level), with a branching factor of 64 for the first level (because there are 64 squares to place the first queen on an empty board), 63 for the next level, and so on, down to 57 for the eighth level

Improved Heuristics For The Eight Queens Problem

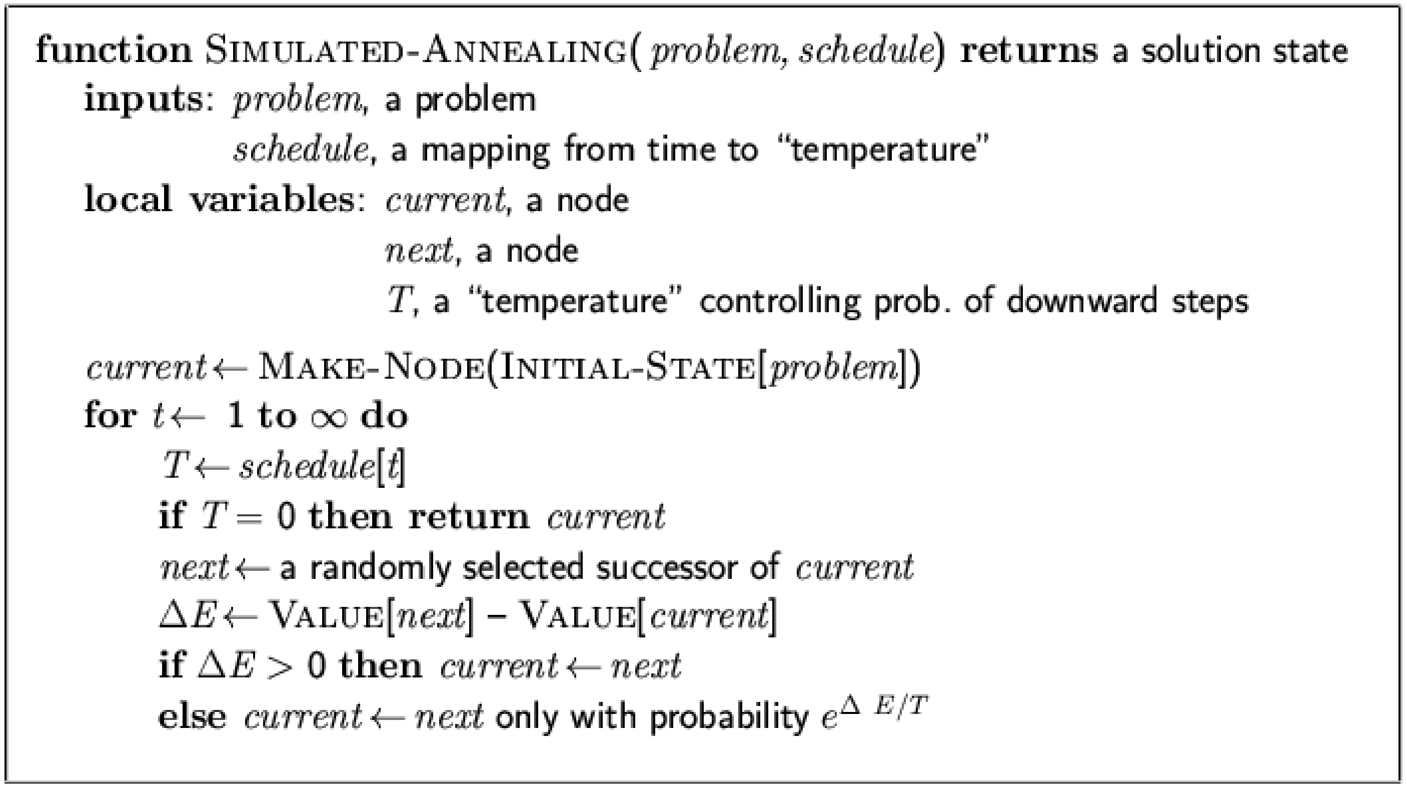
- This can be improved further by noting that each row and column must contain exactly one queen
  - If we assume the first queen is placed in row 1, the second in row 2, etc. the first level will have a branching factor of 8, the next 7, the next 6, etc.
  - In addition, as each queen is placed on the board, it "uses up" a diagonal, meaning that the branching factor is only 5 or 6 after the first choice has been made

Simulated Annealing Search

- Since hill-climbing never moves “downhill” can get stuck at local maximum
- And purely random walk would be inefficient
- What if we tried combining both in a way to yield efficient and complete
- A method borrowed from metallurgy (statistical physics), based on the way in which metal is heated and then cooled very slowly in order to make it extremely strong
- Goal is to obtain a minimum value for some function of a large number of variables
  - This value is known as the *energy of the system*

Simulated Annealing Search

- Simulated annealing escapes local minima by allowing some "bad" moves but gradually decreases the frequency with which they are allowed



```
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
        ΔE ← VALUE[next] – VALUE[current]
        if ΔE > 0 then current ← next
        else current ← next only with probability $e^{\Delta E/T}$
```

Simulated Annealing Search

- A random start state is selected
- A small random change is made to this state
  - If this change lowers the system energy, it is accepted
  - If it increases the energy, it may be accepted, depending on a probability called the Boltzmann acceptance criteria:
    - $e^{(-dE/T)}$

Simulated Annealing Search

- Because the energy of the system is sometimes allowed to increase, simulated annealing is able to escape from local minima
- Simulated annealing is a widely used local search method for solving problems with a very large numbers of variables
  - For example: scheduling problems, traveling salesperson (TSP), placing VLSI (chip) components