Informed/Heuristics search algorithms

Material

- Ch 3 of Artificial Intelligence A Systems Approach by Tim Jones
  Chapter 4 of Artificial Intelligence a Modern Approach by Russell and Norvig

Informed Search Techniques

- Blind search is not always possible, because they require too much time or Space (memory).
- Heuristics are **rules of thumb**; they do not guarantee for a solution to a problem.
- Heuristic Search is a weak techniques but can be effective if applied correctly; they require domain specific information

Characteristics of Heuristic Search

- Heuristics, are knowledge about domain, which help search and reasoning in its domain.
- Heuristic search incorporates domain knowledge to improve efficiency over blind search.
- Heuristic is a function that, when applied to a state, returns value as estimated merit of state, with respect to goal.
  - Heuristics might (for reasons) under estimate or over estimate the merit of a state with respect to goal.
  - Heuristics that under estimates are desirable and called admissible.
- Heuristic evaluation function estimates likelihood of given state leading to goal state.
- Heuristic search function estimates cost from current state to goal, presuming function is efficient.

Heuristic Search compared with other search

The Heuristic search is compared with Brute force or Blind search techniques

Compare Algorithms

Brute force / Blind search	Heuristic search
Only have knowledge about already explored nodes	Estimates "distance" to goal state
No knowledge about how far a node is from goal state	Guides search process toward goal state
	Prefer states (nodes) that lead close to and not away from goal state

Informed Search Techniques

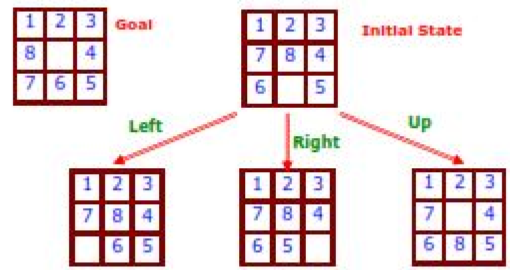
Example : 8 - Puzzle

- State space: Configuration of 8- tiles on the board
- state **Initial**: any configuration **Goal**: tiles in a specific order

- Solution: optimal sequence of operators
- Action: "blank moves"
  - Condition: the move is within the board
  - Directions: Left, Right, Up, Dn
- Problem
  - which 8-puzzle move is best?
  - what heuristic(s) can decide?
  - which move is "best" (worth considering first) ?

- **Actions**

  Figure below shows: three possible moves - left , up, right



Count correct positions **h = 6** **h = 4** **h = 5**

Find Which move is best ?

- Apply the Heuristic : Three different approaches
  - Count correct position of each tile, compare to goal state
  - Count incorrect position of each tile, compare to goal state
  - Count how far away each tile is from it is correct position.

Approaches	Left	Right	Up
1. Count correct position	6	4	5
2. Count incorrect position	2	4	3
3. Count how far away	2	4	4

Each of these three approaches are explained below.

- **Heuristic :**

Three different approaches

- 1st approach :

  Count correct position of each tile, compare to goal state.

  - Higher the number the better it is.
  - Easy to compute (fast and takes little memory).
  - Probably the simplest possible heuristic.

- 2nd approach

  Count incorrect position of each tile, compare to goal state

  - Lower the number the better it is.
  - The "best" move is where lowest number returned by heuristic.

- 3rd approach

  Count how far away each tile is from it's correct position

  - Count how far away (how many tile movements) each tile is from it's correct position.
  - Sum up these count over all the tiles.
  - The "best" move is where lowest number returned by heuristic.

Heuristic Search Algorithms : types

- Generate-And-Test
- Hill climbing
  - Simple
  - Steepest-Ascent Hill climbing
  - Simulated Anealing
- Best First Search
  - OR Graph
  - A* (A-Star) Algorithm
  - Agendas
- Problem Reduction
  - AND-OR_Graph
  - AO* (AO-Star) Algorithm
- Constraint Satisfaction
- Mean-end Analysis

Best-first search

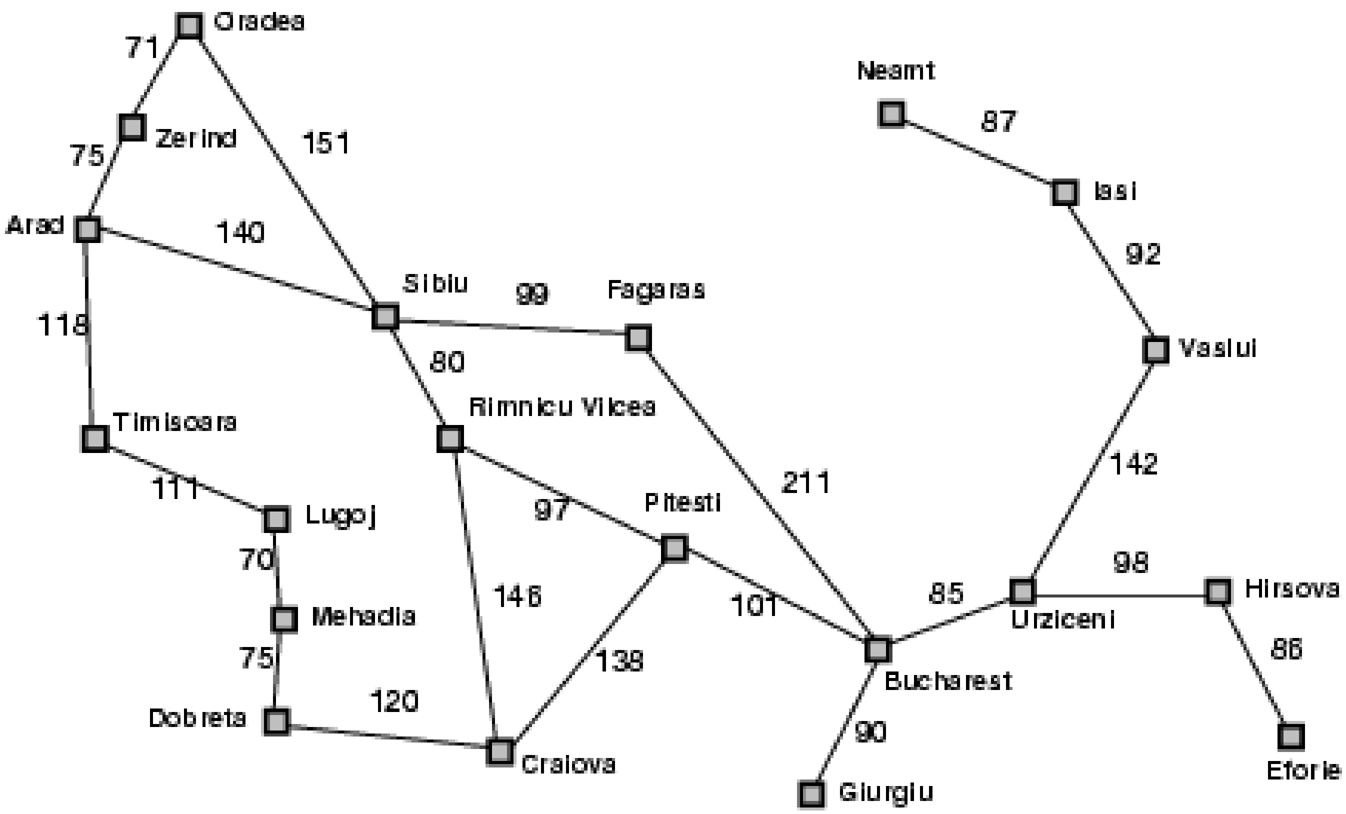
- method of exploring the node with the best “score” first
- Idea: use an evaluation function *f(n)* for each node estimate of "desirability"
  - → The node with the lowest evaluation is considered as best node and selected for expansion
  - → maintains two lists, one containing a list of candidates yet to explore (OPEN), and one containing a list of visited nodes (CLOSED)
  - → algorithm always chooses the best of all unvisited nodes that have been graphed while other search strategies, such as depth-first and breadth-first, have this restriction

  Algorithm:

  1. Define a list, OPEN, consisting solely of a single node, the start node, s.
  2. IF the list is empty, return failure.
  3. Remove from the list the node n with the best score (the node where f is the minimum), and move it to a list, CLOSED.
  4. Expand node n.
  5. IF any successor to n is the goal node, return success and the solution (by tracing the path from the goal node to s).
  6. FOR each successor node:
     a) apply the evaluation function, f, to the node.
     b) IF the node has not been in either list, add it to OPEN.
  7. looping structure by sending the algorithm back to the second step.

- Special cases:
  - greedy best-first search
  - A^* search

Romania with step costs in km



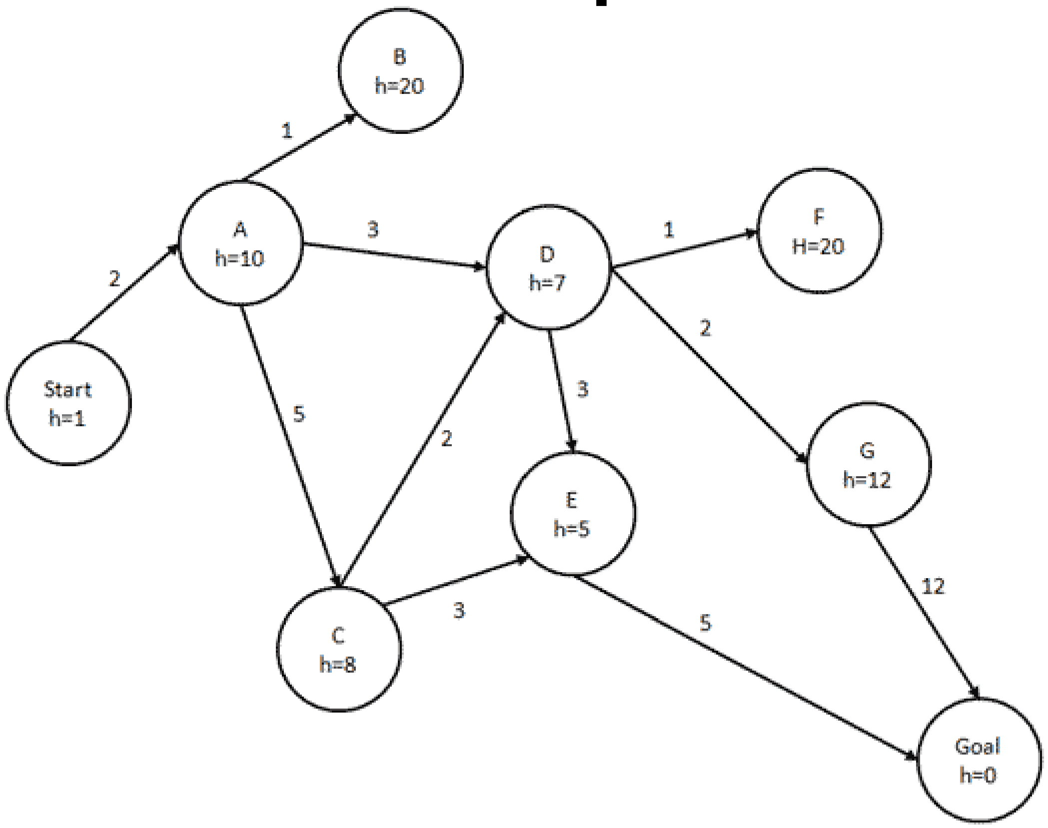
Straight–line distance to Bucharest

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

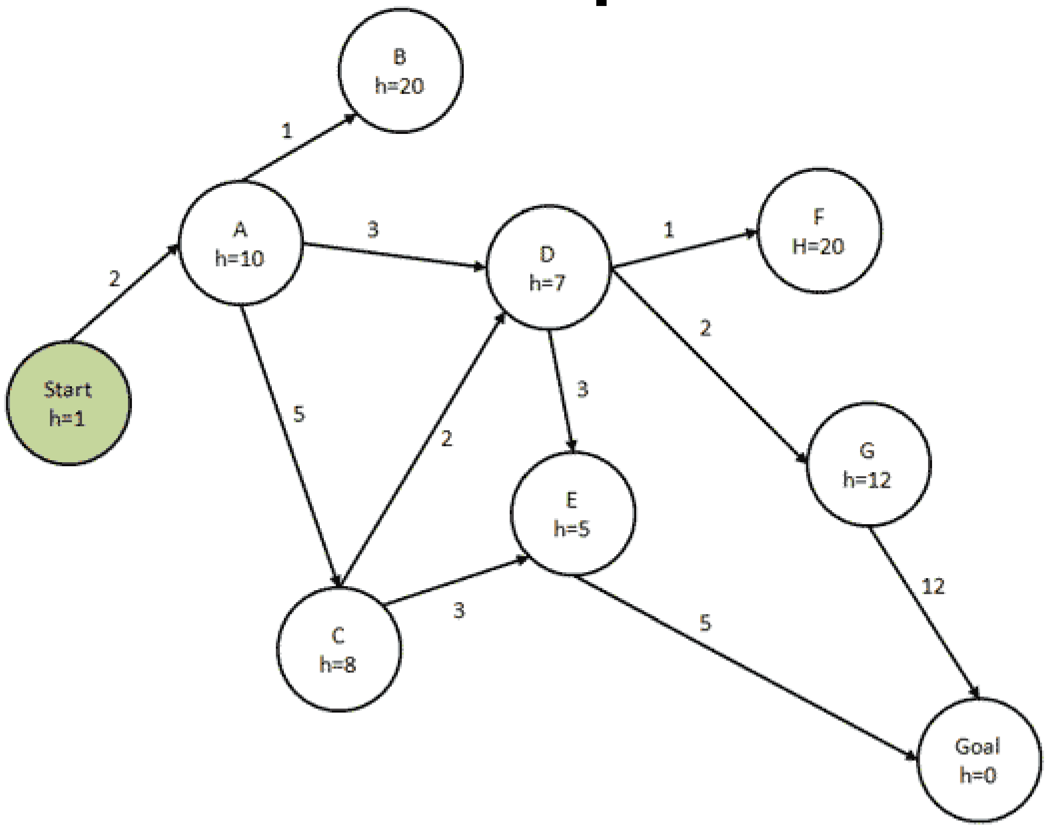
Greedy best-first search

- greedy best-first search uses only the heuristic, and not any link costs to expand the node that appears to be closest to goal
- Evaluation function $f(n) = h(n)$ (heuristic)
- = estimate of cost from n to *goal*
- disadvantage: if the heuristic is not accurate, it can go down paths with high link cost since there might be a low heuristic for the connecting node
- e.g., $h_{SLD}(n)$ = straight-line distance from n to Bucharest

Greedy best-first search example

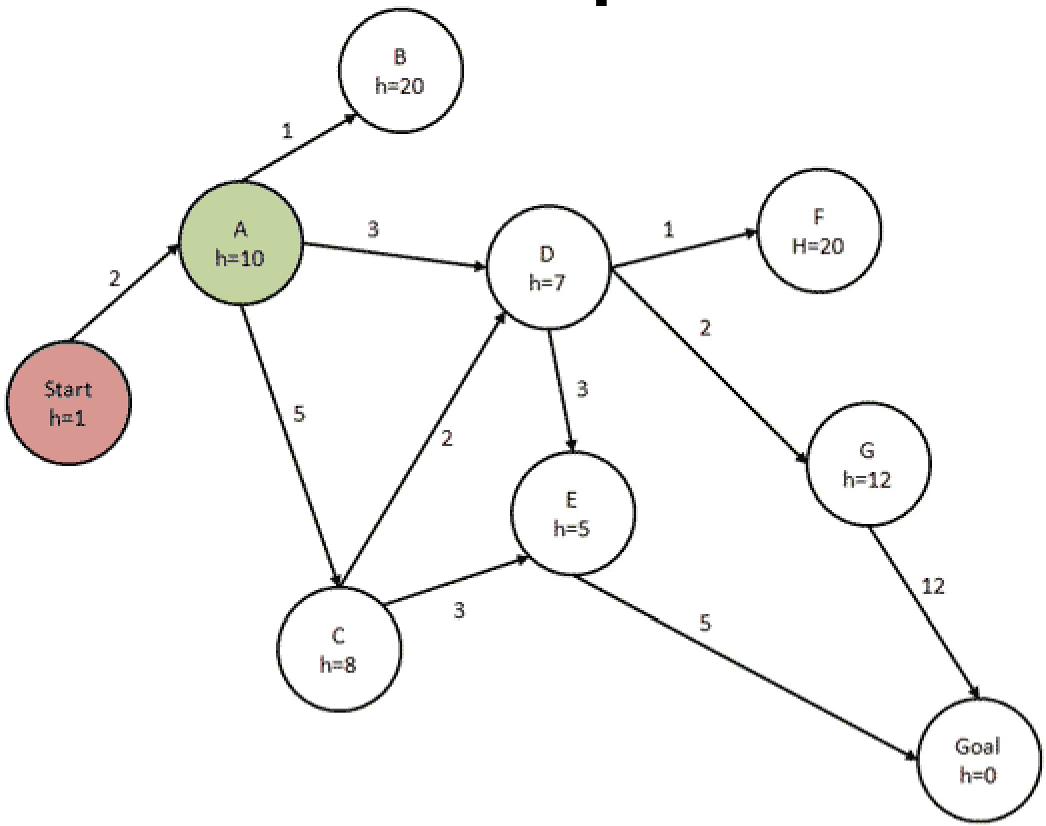


Greedy best-first search example



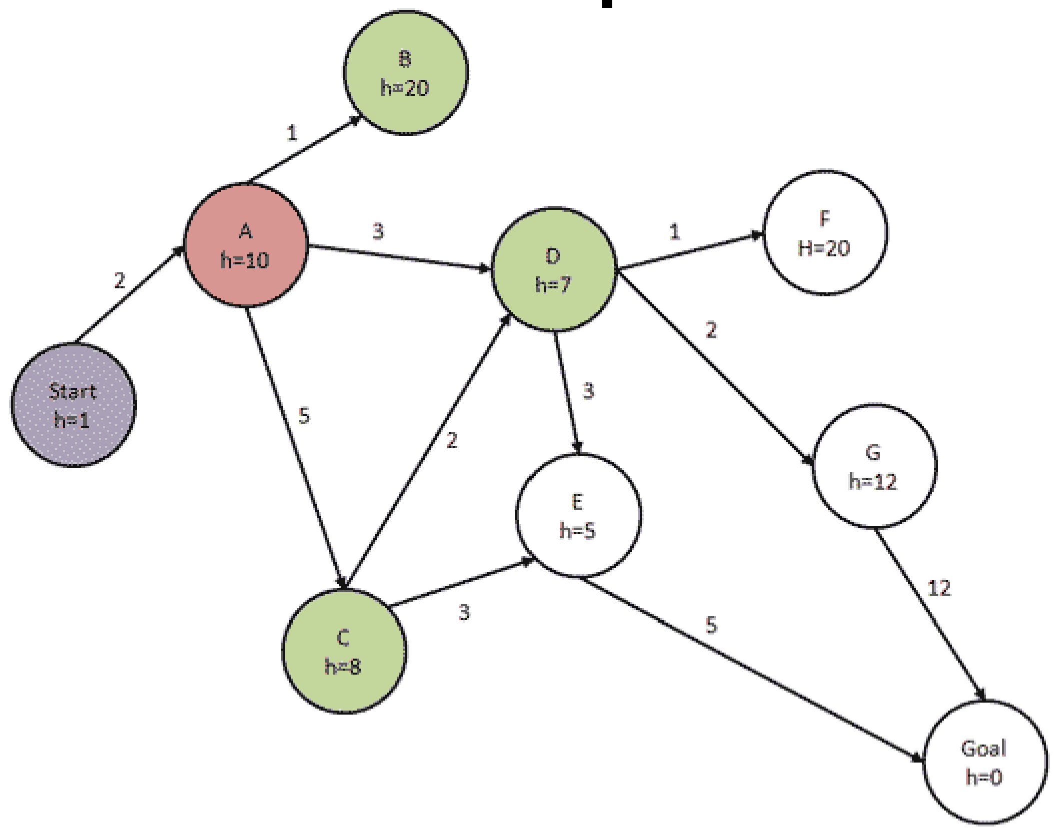
First, add the Start node to the fringe

Greedy best-first search example



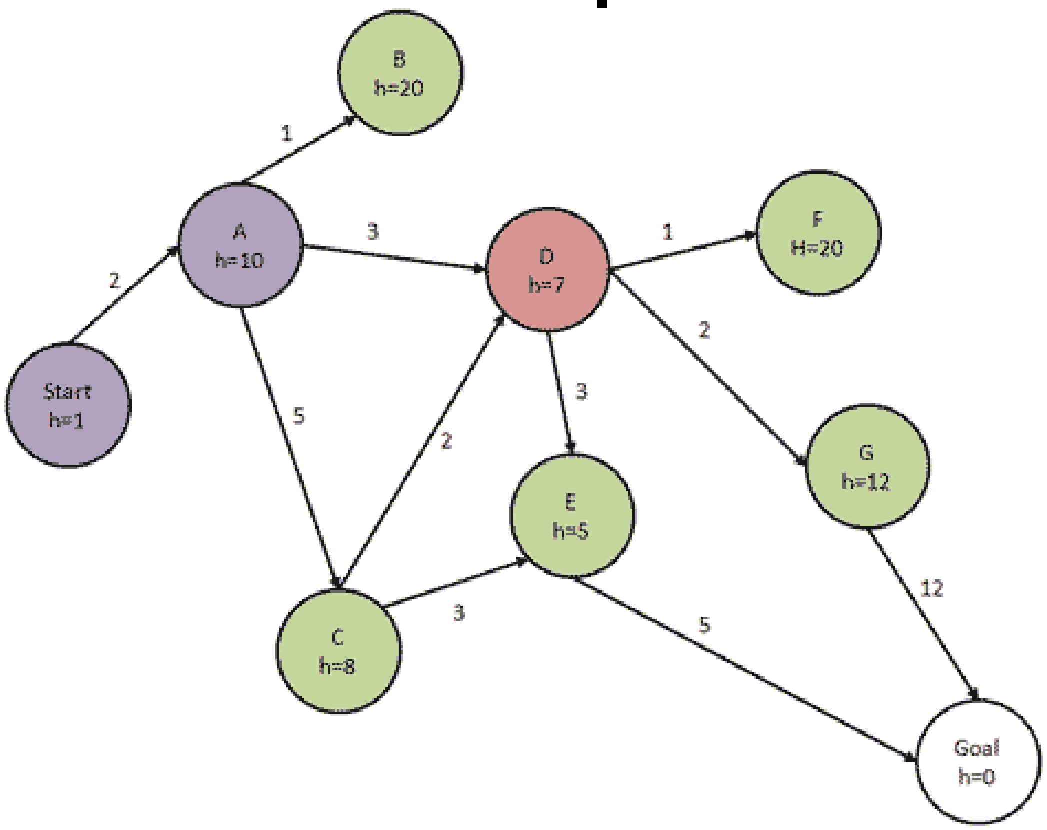
Visit the Start node and add its neighbors to the fringe

Greedy best-first search example



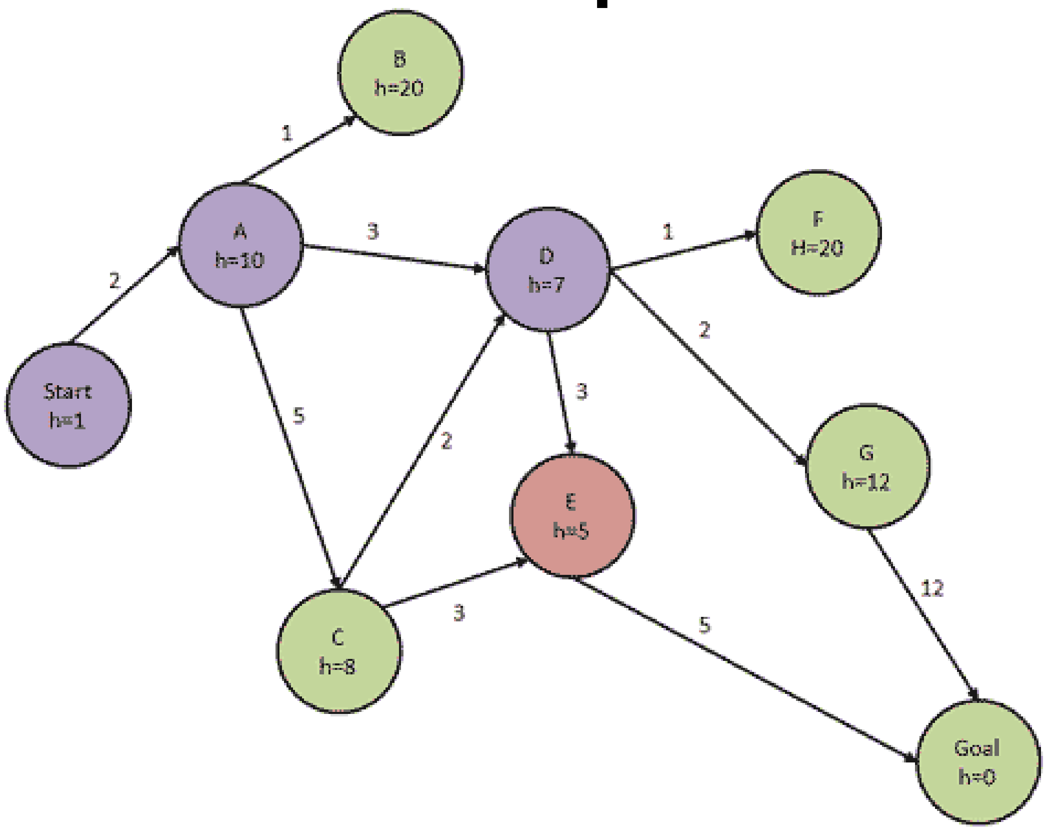
Visit node *A* and add its neighbors to the fringe.

Greedy best-first search example



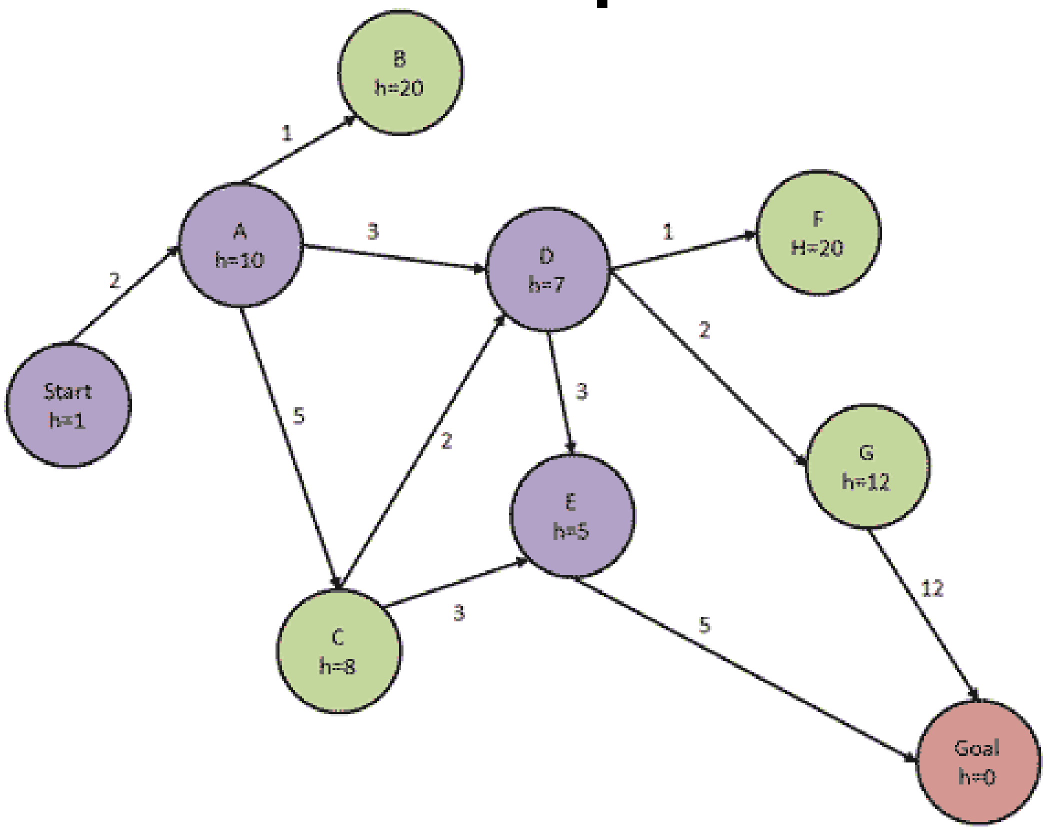
node *D* has the lowest heuristic value, we visit at that node and add its neighbors to the fringe

Greedy best-first search example



node *E* has the lowest heuristic in the fringe, it is visited at and its neighbors are added to the fringe.

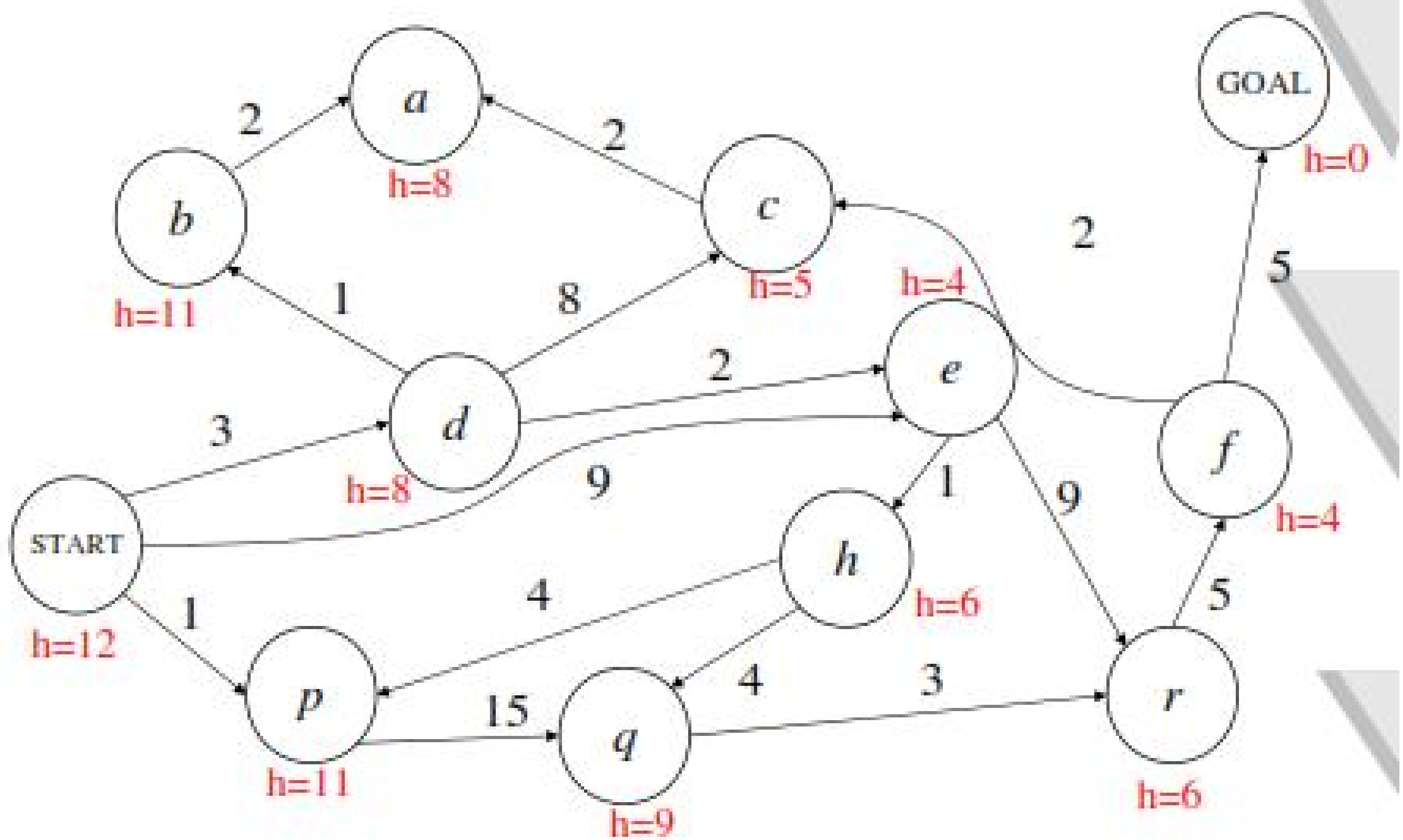
Greedy best-first search example



Goal is in the priority queue with a heuristic of 0, it is visited and a path to the goal is found.

The path found from Start to Goal is: Start -> A -> D -> E -> Goal. In this case, it was the optimal path, but only because the heuristic values were fairly accurate

Greedy best first Search: another illustration



Properties of greedy best-first search

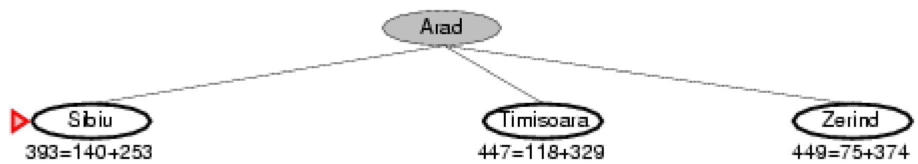
- Complete? No – can get stuck in loops,
- Time? $O(b^m)$, but a good heuristic can give dramatic improvement though all nodes are visited in the worst case
- Space? Also $O(b^m)$ -- keeps all nodes in memory
- *b* is the average branching factor (the average number of successors from a state), and *m* is the maximum depth of the search tree
- Optimal? No

A* search

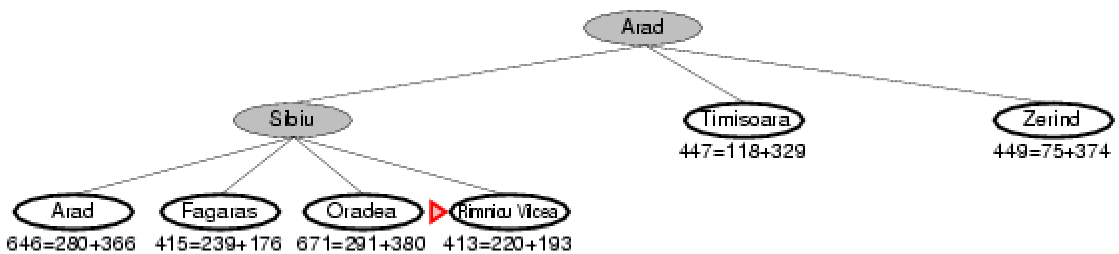
- Idea: avoid expanding paths that are already expensive
-
- Evaluation function $f(n) = g(n) + h(n)$
-
- $g(n)$ = cost so far to reach n
- $h(n)$ = estimated cost from n to goal
- $f(n)$ = estimated total cost of path through n to goal

A* search example

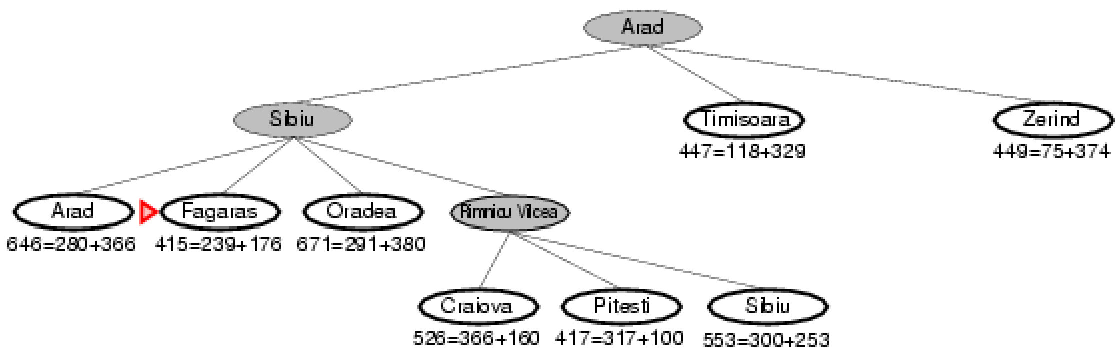
A* search example



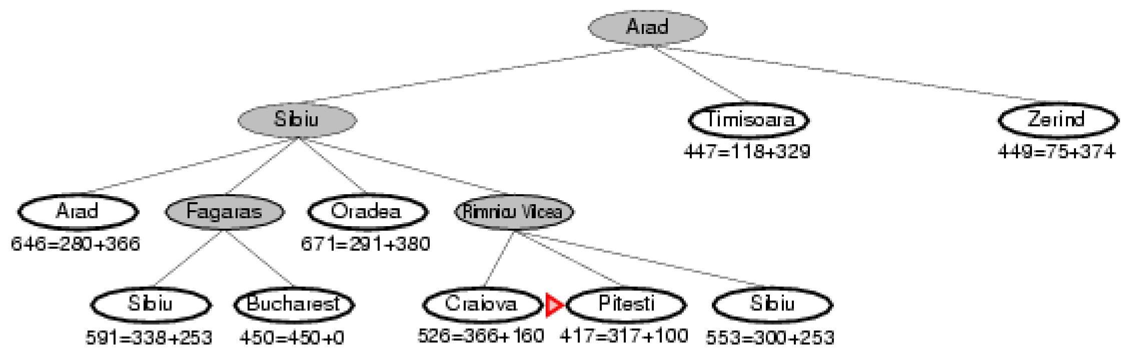
A^* search example



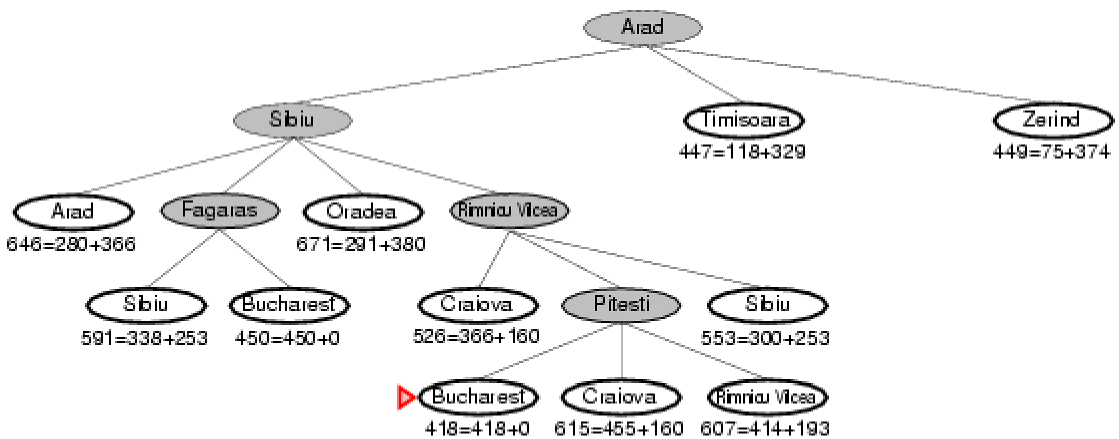
A* search example



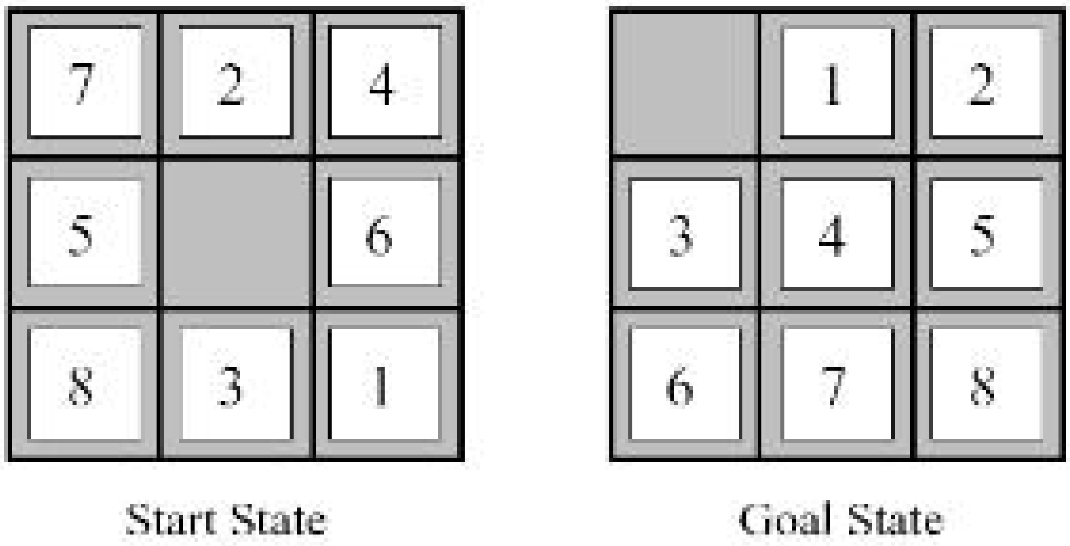
A* search example



A* search example



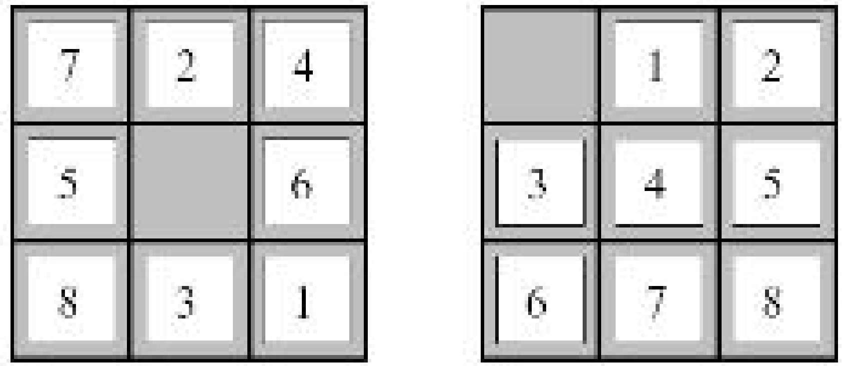
Example: 8 Puzzle



- What are the states?
- How many states?
- What are the actions?
- What states can I reach from the start state?
- What should the costs be?

8 Puzzle I

- Number of tiles misplaced?
- Why is it admissible?
- h(start) = 8
- This is a relaxed-problem heuristic



Start State   Goal State

	Average nodes expanded when optimal path has length…		
	…4 steps	…8 steps	…12 steps
ID	112	6,300	3.6×10^6
TILES	13	39	227

8 Puzzle II

- What if we had an easier 8-puzzle where any tile could slide any direction at any time, ignoring other tiles?
- Total *Manhattan* distance
- Why admissible?

- h(start) =

$$3 + 1 + 2 + \ldots$$

$$= 18$$

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

	Average nodes expanded when optimal path has length…		
	…4 steps	…8 steps	…12 steps
TILES	13	39	227
MAN-HATTAN	12	25	73

8 Puzzle III

- How about using the *actual cost* as a heuristic?
  - Would it be admissible?
  - Would we save on nodes expanded?
  - What's wrong with it?

- With A*: a trade-off between quality of estimate and work per node!

Other A* Applications

- Pathing / routing problems
- Resource planning problems
- Robot motion planning
- Language analysis
- Machine translation
- Speech recognition

Admissible Heuristics

- A heuristic h is admissible (optimistic) if:

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

- where $h^*(n)$ the true cost to a nearest goal

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.
- E.g. Euclidean distance on a map problem
- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic
- Coming up with admissible heuristics is most of what's involved in u sing A* in practice.
- Inadmissible heuristics are often quite effective (especially when you have no choice)
- Theorem: If $h(n)$ is admissible, A^* using `TREE-SEARCH` is optimal

Properties of A*

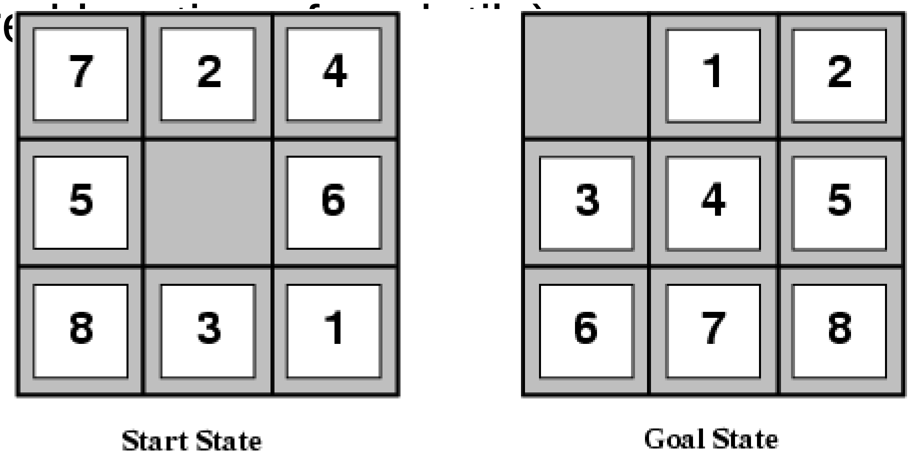
- Complete? Yes (unless there are infinitely many nodes with f ≤ *f(G)*)
-
- Time? Exponential
-
- Space? Keeps all nodes in memory
-
- Optimal? Yes
-

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 desire



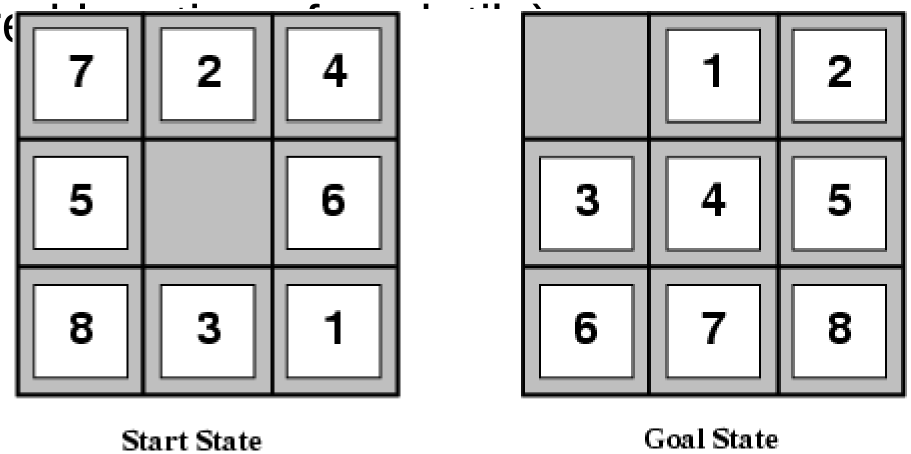
- $h_1(S) = ?$
- $h_2(S) = ?$
-

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 desire



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

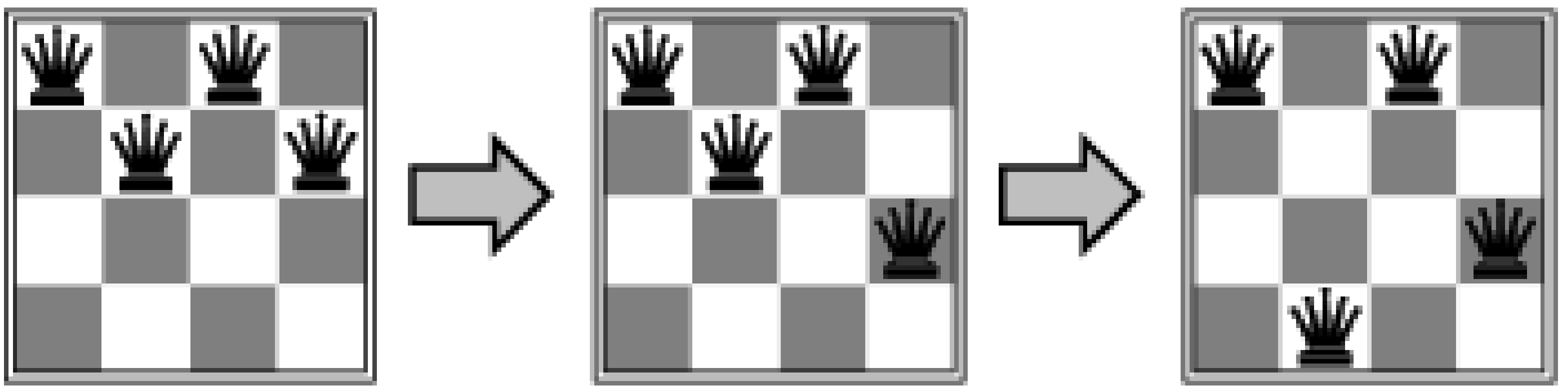
Relaxed problems

- 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 using Hill climbing algorithm

Example: *n*-queens

- Put *n* queens on an *n* × *n* board with no two queens on the same row, column, or diagonal
- 

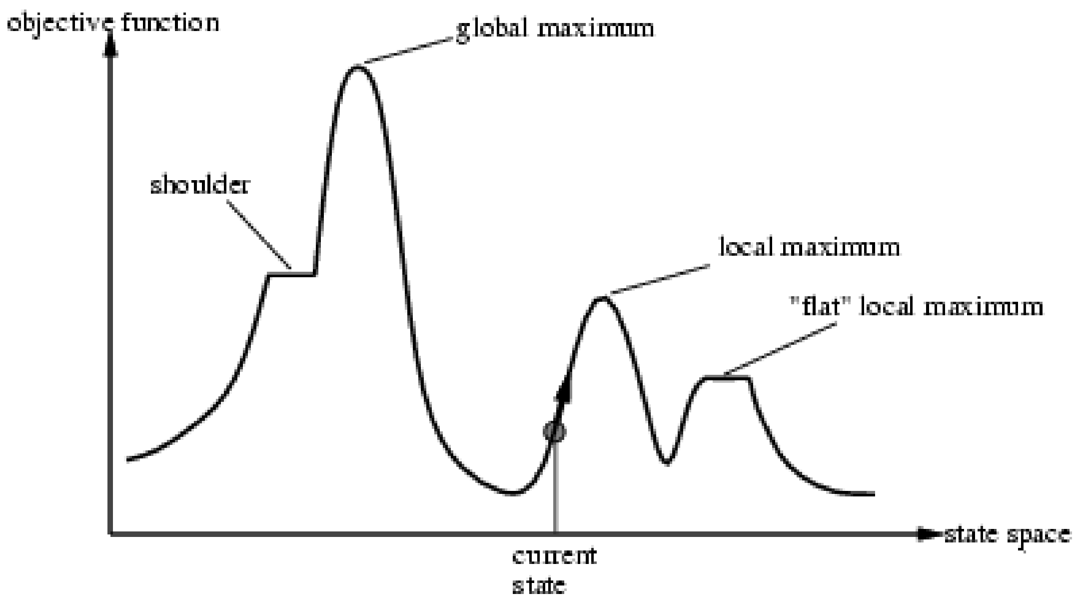
Hill-climbing search

- "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
```

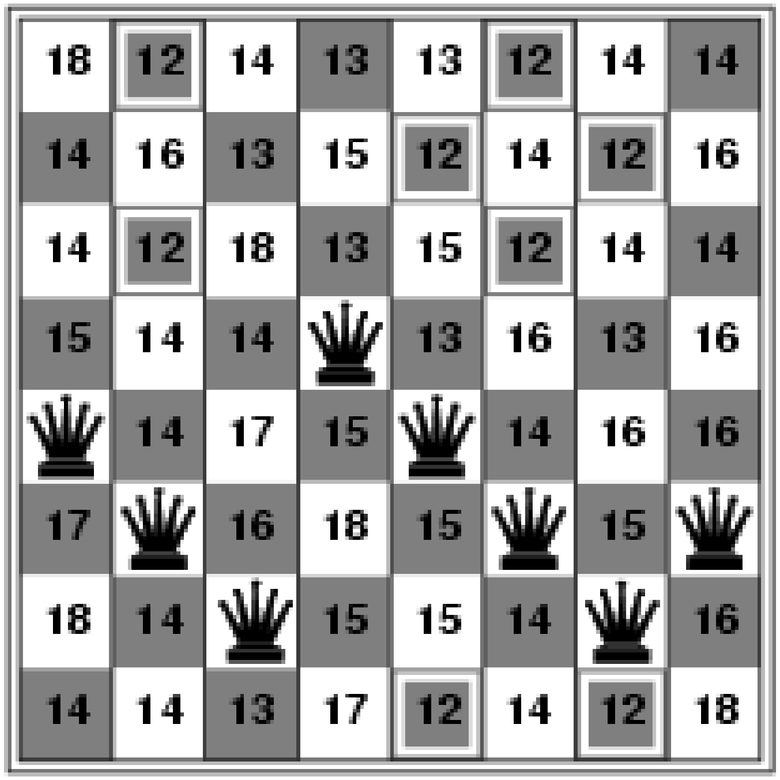
Hill-climbing search

- iterative algorithm that starts with an arbitrary solution to a problem, then attempts to find a better solution by incrementally changing a single element of the solution. If the change produces a better solution, an incremental change is made to the new solution, repeating until no further improvements can be found.
- Simple, general idea:
  - Start wherever
  - Always choose the best neighbor
  - If no neighbors have better scores than current, quit
- Problem: depending on initial state, can get stuck in local maxima
- 

Hill climbing

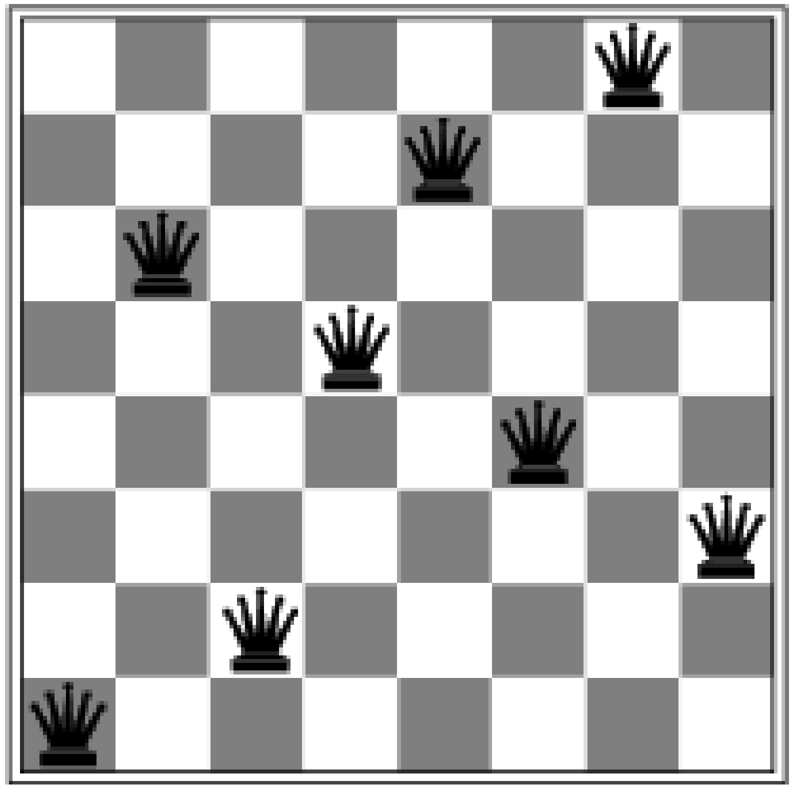
- Application in the traveling salesman problem:
    - It is easy to find an initial solution that visits all the cities but will be very poor compared to the optimal solution. The algorithm starts with such a solution and makes small improvements to it, such as switching the order in which two cities are visited. Eventually, a much shorter route is likely to be obtained.
- good for finding a local optimum (a good solution that lies relatively near the initial solution)
- Not guaranteed to find the best possible solution (the global optimum) out of all possible solutions (the search space).
- Its relative simplicity makes it a popular first choice amongst optimizing algorithms
- more advanced algorithms such as simulated annealing or tabu search may give better results, in some situations hill climbing works just as well
- can often produce a better result than other algorithms when time is limited

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 ← 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^(Δ 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

- Widely used in VLSI layout, airline scheduling, etc
-

Tabu search

- A metaheuristic algorithm that can be used for solving combinatorial optimization problems, such as the traveling salesman problem (TSP). Tabu search uses a local or neighborhood search procedure to iteratively move from a solution x to a solution x' in the neighborhood of x, until some stopping criterion has been satisfied. To explore regions of the search space that would be left unexplored by the local search procedure
- Tabu search modifies the neighborhood structure of each solution as the search progresses
- Example: Traveling salesman problem