Informed search algorithms

Chapter 3

Outline I

- Informed = use problem-specific knowledge
- Which search strategies?
  - **Best-first search and its variants**
- Heuristic functions?
  - **How to invent them**

Outline II

- Best-first search
- A*
- Heuristics
- Hill climbing

Kinds of Search problems

- Type of solution for a given problem
  (a) Any solution
    - “How do I get to Toronto? Money/Time is no object!”
  (b) Optimal solution (best, “good quality”, cheapest,...).
    - “How do I get to Toronto with $15?”
- Nature of problem obtained
  - (a) Finding a path or sequence to solve a problem.
    - path “transforms” start state to goal state
    - E.g., moves needed to solve a Rubik’s Cube?
  - (b) Finding a configuration that is a solution.
    - this single state is everything you need
    - e.g., where to put 8 queens on a board for 8-queen puzzle?

Overview of search algorithms I

- **Basic idea**:
  – offline, simulated exploration of state space by generating successors of already-explored states (a.k.a.~expanding states)

  – A strategy is defined by picking the order of node expansion

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
   initialize the search tree using the initial state of problem
   loop do
       if there are no candidates for expansion then return failure
       choose a leaf node for expansion according to strategy
       if the node contains a goal state then return the corresponding solution
       else expand the node and add the resulting nodes to the search tree
```

Previously: tree-search

```
function TREE-SEARCH(problem,fringe) return a solution or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if EMPTY?(fringe) then return failure
        node ← REMOVE-FIRST(fringe)
        if GOAL-TEST[problem] applied to STATE[node] succeeds
            then return SOLUTION(node)
        fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
```

A strategy is defined by picking *the order of node expansion*

Overview of search algorithms II

- **Blind search (uninformed)**
  - **Exhaustive search over all configurations**
    - **"ANY" problem: immediately stop when a solution discovered**
    - **"Optimal" problem: stop when you are sure best solution found**
  - **Usually expensive in computation effort!**
- Heuristic Search (informed)
    - Informed = use problem-specific knowledge
  - **<u>"ANY" problems:</u> heuristic helps to find a solution more efficiently**
    - **heuristics will reduce the number of cases to be looked at**
    - **although "good" solutions might arise, the best solution is not guaranteed**
  - **<u>Optimal problems</u>: exhaustive, but can help determine which parts of search tree can be ignored**
    - **again, heuristic reduces number of cases to investigate**
    - **a best solution is guaranteed (but computation effort is an issue)**

Heuristics

- Heuristic: any rule or method that provides some guidance in decision making
  - we use problem-domain specific information in making a decision
  - heuristics vary in the amount of useful information they can lend us
    - e.g., a stronger heuristic: don't make any chess move that results in your losing a piece!
    - e.g., a weaker heuristic (rule of thumb): knights are best moved into central board positions

- heuristics are often denoted by a functional value: high values

denote positive paths, while lower or negative are less promising ones

A heuristic function

- [dictionary] *"A rule of thumb, simplification, or educated guess that reduces or limits the search for solutions in domains that are difficult and poorly understood."*
  - $h(n)$ = **estimated cost of the cheapest path from node *n* to goal node**.
  - If *n* is goal then $h(n)=0$

How to design heuristics? Later..

Heuristics and search

- **Consider a search tree: a given node has a number of children expanded for it (possible all, or just a few)**
  - ideally, we'd like to know which child takes us towards a solution; but this might not be determinable (hence the need for blind search)

- **heuristics permit us to evaluate the children, and select a most promising one**
  - we can even rank them in order of promise
  - this lets us incorporate problem-specific knowledge into the search strategy
  - note: many domains do not admit strong heuristics, so the rating of nodes might be of minimal use (but better than nothing, hopefully!)
- **There are books dedicated to the design of heuristic functions**

Some heuristic search algorithms to consider

- **Optimal (path) search**
  - **Best-first**
  - A*
- **Non-optimal (local) search**
  - **Hill-climbing**
  - **Beam Search**
  - **Simulated annealing**
  - **Genetic algorithm**

Finding the Best Solution

- aka "optimal search"
  - quality of solution is important
- British museum technique: exhaustively find all paths, then pick the best (e.g., least distance)
  - may use depth-first, breadth-first,... any blind search technique you wish
  - well, search itself isn't important - just exhaustively enumerate all solutions!

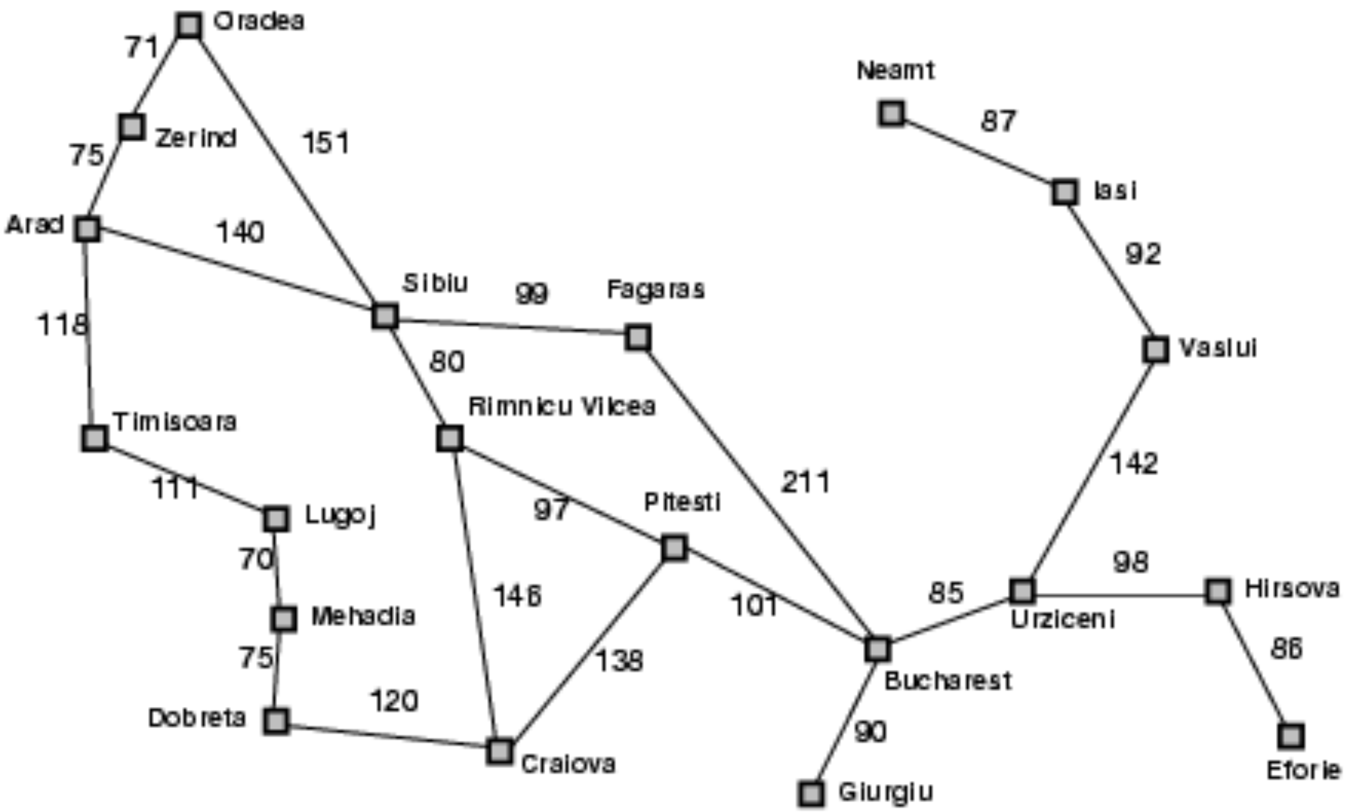
Path problems and Underestimates

- by adding a guesstimate of the distance remaining for a partial path node, the search can be sped up even more
  - if you knew exact distance, no search would be necessary
  - if your guess is an overestimate, the problem is that you can no longer use distance information to terminate searches
    - the excess distance value may say a node is too bad to use, when it isn't at all
    - also, any partial path value can't be compared to exact distances with any accuracy
- how do you make lower-bound estimates?
  - closer to real values - more accurate the search
  - heuristically

Best-first search

- General approach of informed search:
  - **Best-first search: node is selected for expansion based on an _evaluation function $f(n)$_**
- Idea: evaluation function measures distance to the goal.
  - **Choose node which *appears* best**
- Implementation:
  - *fringe* **is queue sorted in decreasing order of desirability.**
  - Special cases: **greedy 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

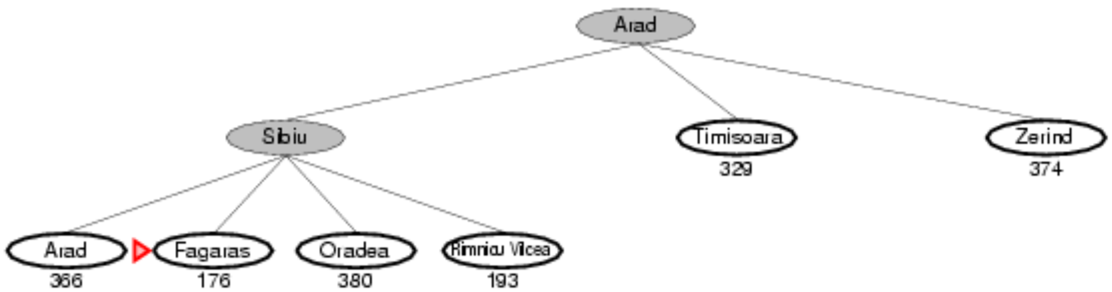
Greedy best-first search

- Evaluation function $f(n) = h(n)$ (heuristic)
- = 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

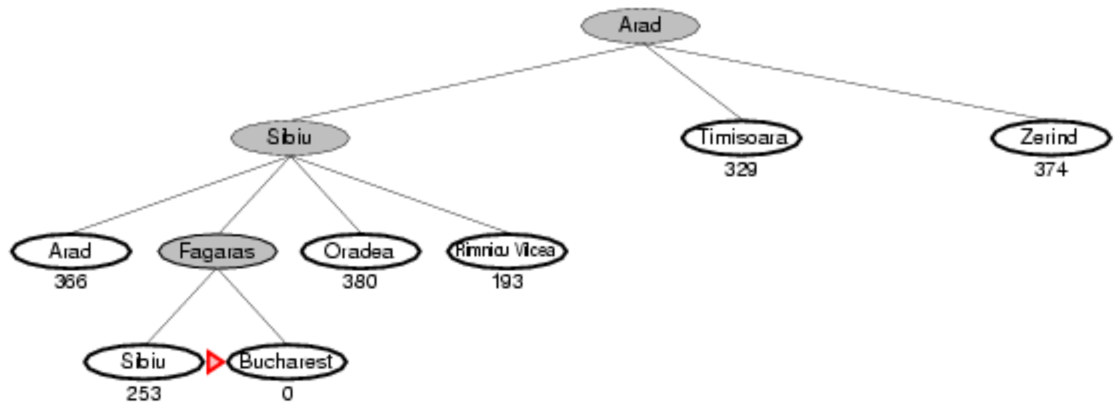
Greedy best-first search example

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, same as DF-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 to reach the goal from n.

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

A* search

- A* search uses an admissible heuristic
  - **A heuristic is admissible if it *never overestimates* the cost to reach the goal**
  - **Are optimistic**

Formally:

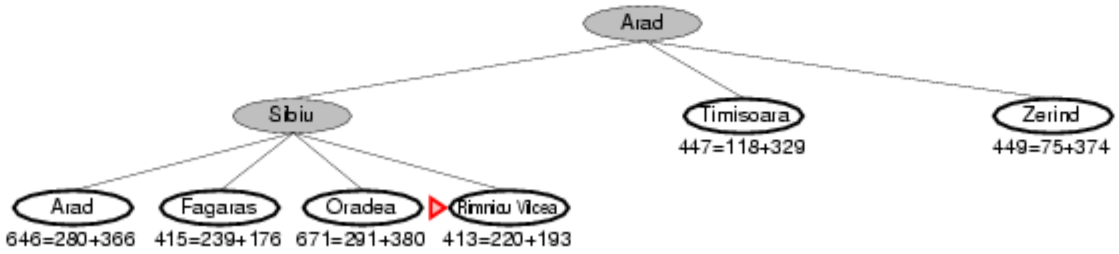
1. $h(n) <= h^*(n)$ where $h^*(n)$ is the true cost from n
2. $h(n) >= 0$ so $h(G)=0$ for any goal G.

e.g. $h_{SLD}(n)$ never overestimates the actual road distance

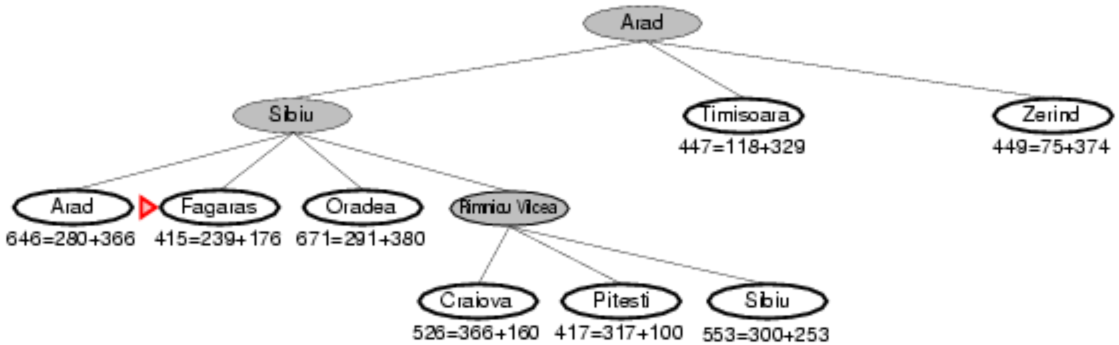
A^* search example

A* search example

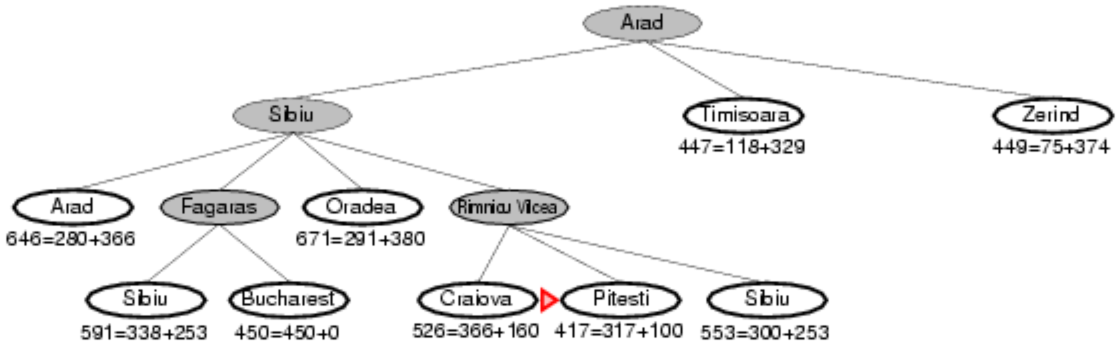
A* search example



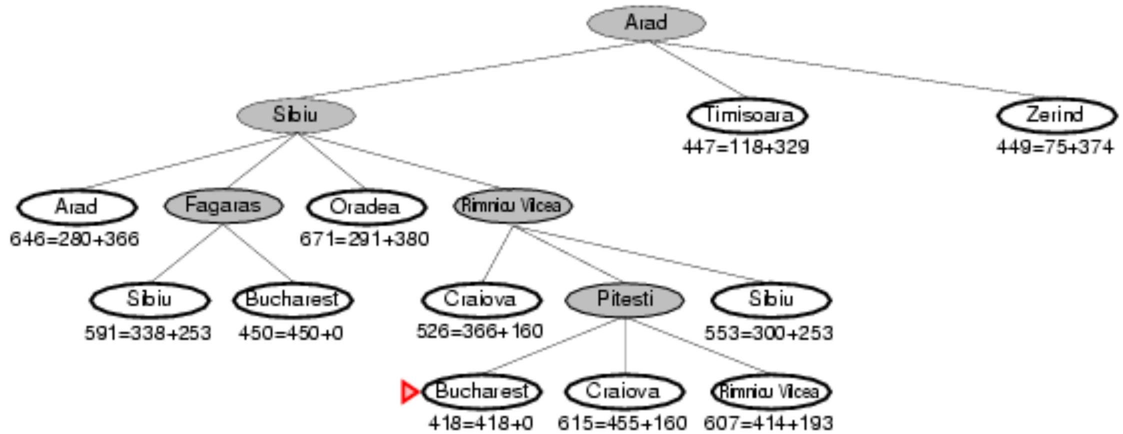
A* search example



A* search example



A* search example



Optimality of A*

- Reading assignment

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

Improving the memory cost for A*

(Further details found in 1st Edition of your text)

- Algorithm;
  - set cutoff, h(start node)
    - i.e., an initial estimate of distance to goal
- Do pure depth-first search, stopping when f(n) > cutoff
    - if succeed, done.
    - if fail, cutoff+minimum-amount-by which cutoff-was-exceed;iterate step
- This is called “**iterative deepening A***” (IDA)

Iterative Deepening A* (IDA)

- Always finds an optimal solution
- Uses space linear in solution depth
- Is asymptotically no slower that A*
- Assuming a tree structure space, so we don't have to check for cycles
- Or at least, that cycles are few and long
- So, IDA* is about as good as we can do, given an (admissible) heuristic function.

IDA* vs A*

- In practice using a heuristic like the manhattan distance, A* can't solve the 15 puzzle because machine run out of memory, IDA*

- Empirically, IDA* generates more nodes than A*, but surprisingly, it often runs faster!
  - it incurs less overhead per node, and it is easier to implement (since it is essentially depth-first as opposed to breadth-first)
  - e.g., IDA* finds 12 step solution to 8-puzzle problem very quickly expanding 39 nodes

- A lot depends on the problem space
  - For example, in a space where there are only a few values for the heuristic function (e.g.,, the n-puzzle), IDA* works well, In a case in which each state has a different value (e.g., finding the paths between locations), IDA* will have to expand the square of of the number of nodes A* will expand

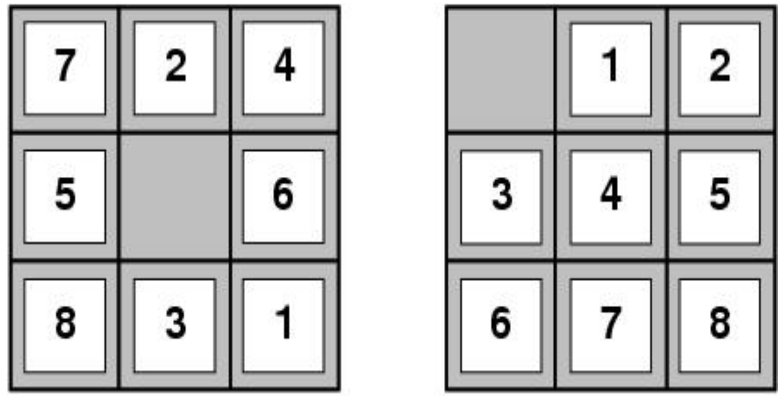
Heuristics: defns

- **Admissible heuristic: the measurement never overestimates the score (distance)**
  - otherwise, overestimates may cause good nodes to be skipped, because they are being ignored when they shouldn't be
  - however, if heuristics are too cautious, no useful information is provided. Extreme case: zero for all evaluations (same as using no heuristic!)
- **Consistency: for distance measurements, a consistent heuristic will give smaller values as you get closer to 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.
- 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

Heuristic functions



Start State Goal State

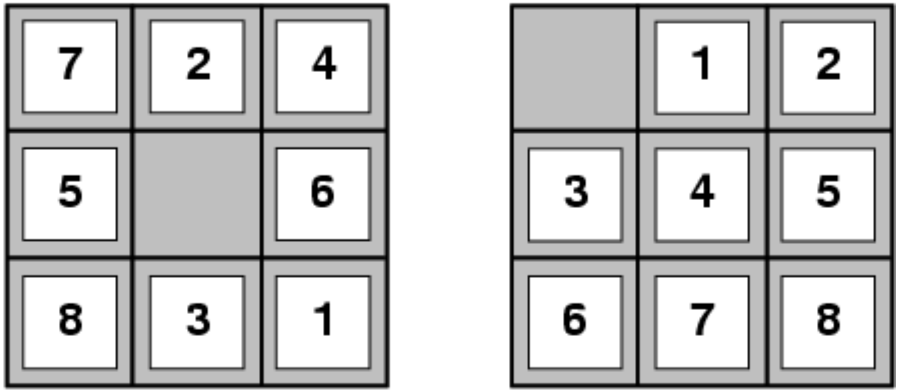
- E.g for the 8-puzzle
  - Avg. solution cost is about 22 steps (branching factor +/- 3)
  - Exhaustive search to depth 22: 3.1 x 10^{10} states.
  - A good heuristic function can reduce the search process.

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

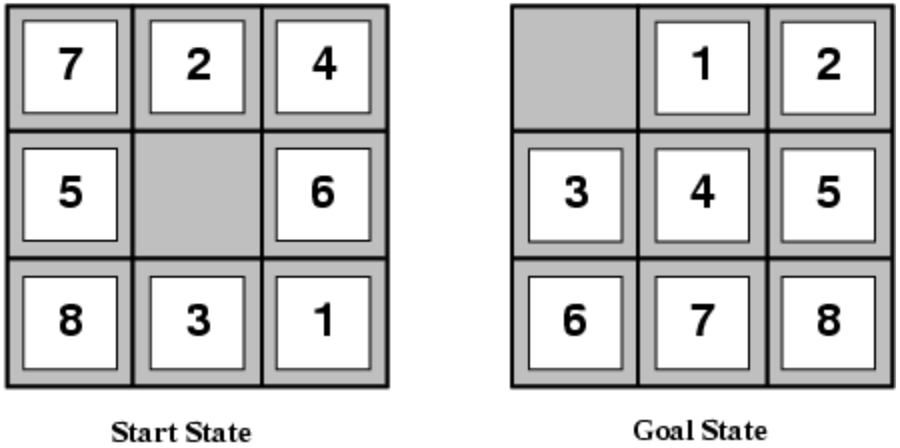
- $h_1(S) = ?$
- $h_2(S) = ?$

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)



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

Inventing admissible heuristics

- Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem:
  - Relaxed 8-puzzle for h_1 : a tile can move anywhere

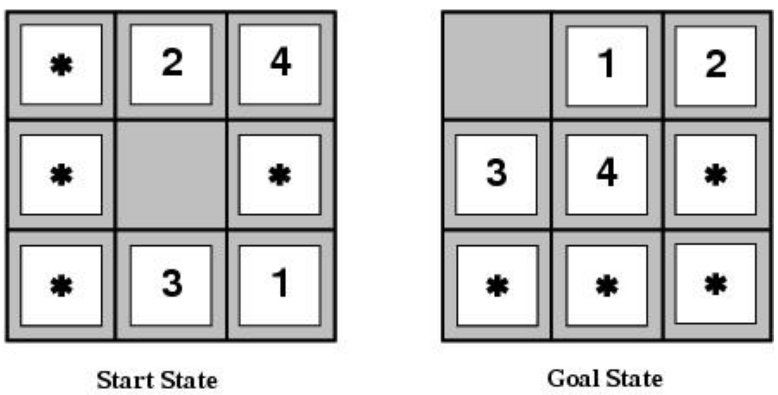
    then, $h_1(n)$ gives the shortest solution
  - Relaxed 8-puzzle for h_2 : a tile can move to any adjacent square.

    As a result, $h_2(n)$ gives the shortest solution.

Key point: The optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem.

Inventing admissible heuristics

- Admissible heuristics can also be derived from the solution cost of a subproblem of a given problem.
- This cost is a lower bound on the cost of the real problem.
- Pattern databases store the exact solution to for every possible subproblem instance.
  - The complete heuristic is constructed using the patterns in the DB



Inventing admissible heuristics

- Another way to find an admissible heuristic is through learning from experience:
  - Experience = solving lots of 8-puzzles
  - An inductive learning algorithm can be used to predict costs for other states that arise during search.

Learning to search better

- All previous algorithms use *fixed strategies*.
- Agents can learn to improve their search by exploiting the *meta-level state space*.
  - Each meta-level state is a internal (computational) state of a program that is searching in *the object-level state space*.
  - In A* such a state consists of the current search tree
- A meta-level learning algorithm from experiences at the meta-level.

Local search and optimization

- Previously: systematic exploration of search space.
  - **Path to goal is solution to problem**
- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution
- Generally, for some problems path is irrelevant.
  - **E.g 8-queens**

- Different algorithms can be used
  - **Local search**

Example: *n*-queens

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

Local search and optimization

- Local search= use single current state and move to neighboring states.
- Advantages:
  - **Use very little memory**
  - **Find often reasonable solutions in large or infinite state spaces.**
- Are also useful for pure optimization problems.
  - **Find best state according to some *objective function*.**
  - **e.g. survival of the fittest as a metaphor for optimization.**

Hill-climbing search

- **It terminates when a peak is reached**.
- Hill climbing does not look ahead of the immediate neighbors of the current state.
- Hill-climbing chooses randomly among the set of best successors, if there is more than one.
- Hill-climbing a.k.a. ***greedy local search***

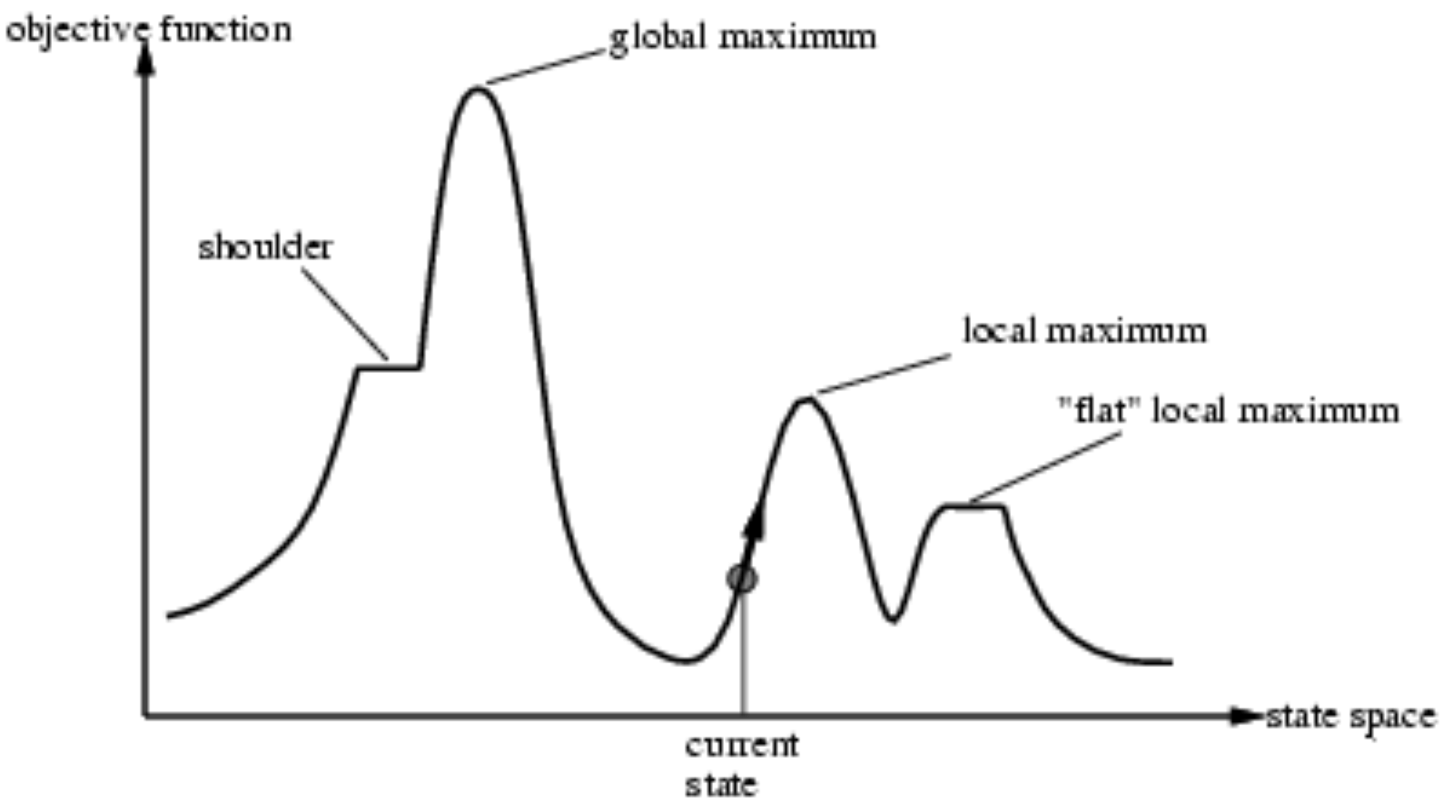
Hill-climbing search

```
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

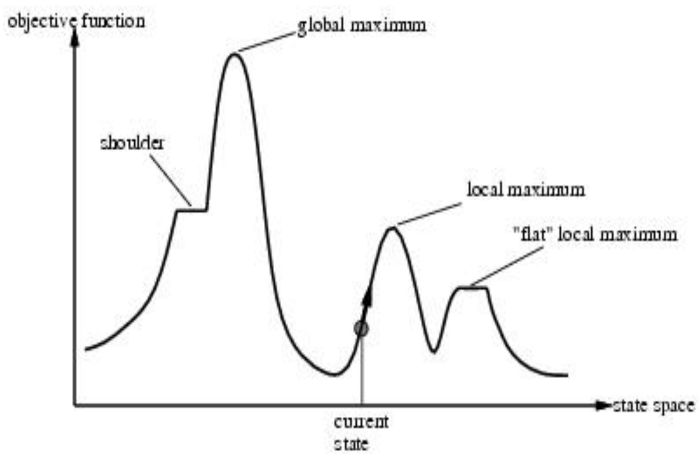
- Problem: depending on initial state, can get stuck in local maxima.

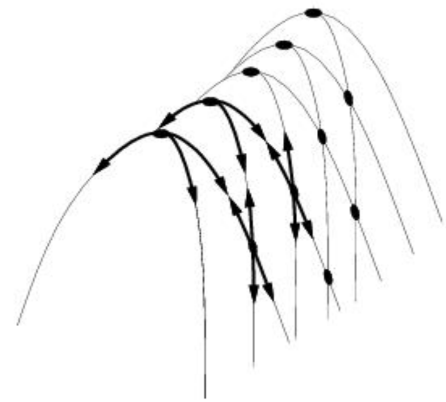


Hill-climbing example

- 8-queens problem (complete-state formulation).
- Successor function: move a single queen to another square in the same column.
- Heuristic function *h(n)*: the number of pairs of queens that are attacking each other (directly or indirectly).

Drawbacks





- **Ridge** = sequence of local maxima difficult for greedy algorithms to navigate
- **Plateaux** = an area of the state space where the evaluation function is flat.
- Gets stuck 86% of the time.

Hill-climbing variations

- Stochastic hill-climbing
  - **Random selection among the uphill moves.**
  - **The selection probability can vary with the steepness of the uphill move**.
- First-choice hill-climbing
  - **stochastic hill climbing by generating successors randomly until a better one is found.**
- Random-restart hill-climbing
  - **Tries to avoid getting stuck in local maxima.**

Simulated annealing

Tries to fix the weakness with hill-climbing methods where the search gets stuck in a local maximum.

Basic Idea: Instead of picking the best move, pick a random move; if the successor state obtained by this move is an improvement over the current state, then do it. Otherwise, make the move with some probability < 1. The probability decreases exponentially with the badness of the move.

Define a temperature function that decreases over time. At each move, compute the current temperature T, and use T to determine the probability with which to allow a move to a worse state. In the limit, T goes to 0 zero at which point the method is doing hill-climbing, hence the probability is proportional to T.

Simulated annealing

- Escape local maxima by allowing “bad” moves.
  - **Idea: but gradually decrease their size and frequency**.
- Origin; metallurgical annealing
- Bouncing ball analogy:
  - **Shaking hard (= high temperature).**
  - **Shaking less (= lower the temperature).**
- If T decreases slowly enough, best state is reached.
- Applied for VLSI layout, airline scheduling, etc.

Simulated annealing

function SIMULATED-ANNEALING(*problem, schedule*) **return** a solution state
  input: *problem*, a problem
    schedule, a mapping from time to temperature
  local variables: *current***,** a node.
    *next***,** a node.
    *T***,** a "temperature" controlling the probability 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 $\Delta E > 0$ **then** *current* ← *next*
    else *current* ← *next* only with probability $e^{\Delta E / T}$

Local beam search (Reading assignment)

- Keep track of *k* states instead of one
  - **Initially: *k* random states**
  - **Next: determine all successors of *k* states**
  - **If any of successors is goal → finished**
  - Else select *k* best from successors and repeat.
- Major difference with random-restart search
  - **Information is shared among *k* search threads**.
- Can suffer from lack of diversity.
  - **Stochastic variant: choose k successors at proportionally to state success.**

Search summary

- best search method is problem specific
- heuristics permit us to evaluate the children, and select a most promising one
  - we can even rank them in order of promise
  - this lets us incorporate problem-specific knowledge into the search strategy
  - note: many domains do not admit strong heuristics, so the rating of nodes might be of minimal use (but better than nothing, hopefully!)
- There are books dedicated to the design of heuristic functions