Lecture Plan

Blind vs. informed search strategies

Blind search methods

- You already know them: BFS, DFS, UCS *et al*.
- They don't analyse the nodes in the *fringe*; they simply pick the first one (going down or sideways).

Informed search methods

- They try to "guess" which node in the fringe is most *promising*....
- ...by using an *evaluation function*, also known as the *heuristic*.
- The heuristic is based on additional *knowledge* about the search space.

Best-first search algorithm family

- General approach: picks node for expansion based on *evaluation function*, $f(n)$.
- We don't *know* goal is best; if we knew, it wouldn't be a search at all.
- *Heuristic function* $h(n)$ is the key component of $f(n)$.

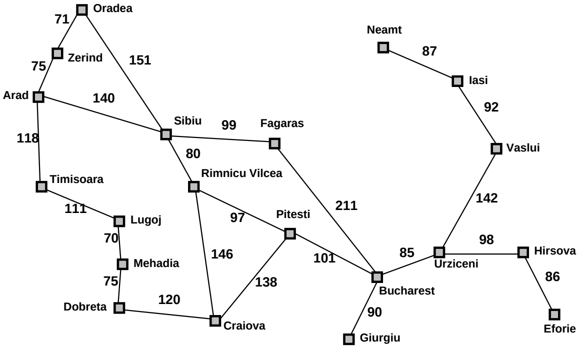
Heuristic functions

- $h(n)$ = estimated cost of the cheapest path from node n to the goal node.
- If n is the goal node, $h(n) = 0$.

Heuristic functions

Choosing an evaluation function

- If we know the locations of cities, we know the straight-line distances between them.
- h_{SLD} – straight-line distance from n to Bucharest.



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

Heuristic functions

Admissible heuristic

- Never overestimates the cost to reach the goal, e.g. $h(n) \leq h^*(n)$ (where $h^*(n)$ is the *real* cost to get from n to goal).
- Is by nature *optimistic*.

Monotonic (consistent) heuristic

- For every node n and every successor n' of n (generated by any action a), estimated cost of reaching the goal from n is no greater than the stop cost of getting to n' plus estimated cost of reaching the goal from n':

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

- This is a form of general *triangle inequality*: each side of a triangle cannot be longer than sum of two other sides.

Greedy search

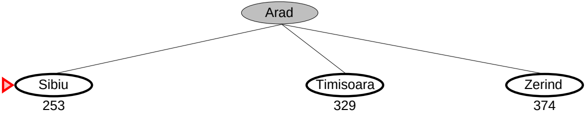
Greedy best-first search

Only takes into account the heuristic, e.g. $f(n) = h(n)$.

Greedy search

Greedy best-first search

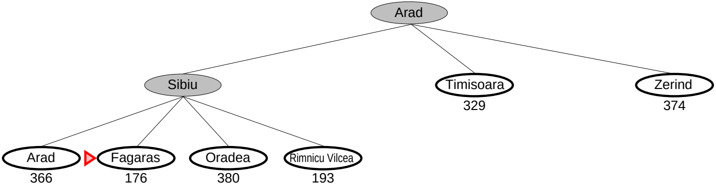
Only takes into account the heuristic, e.g. $f(n) = h(n)$.



Greedy search

Greedy best-first search

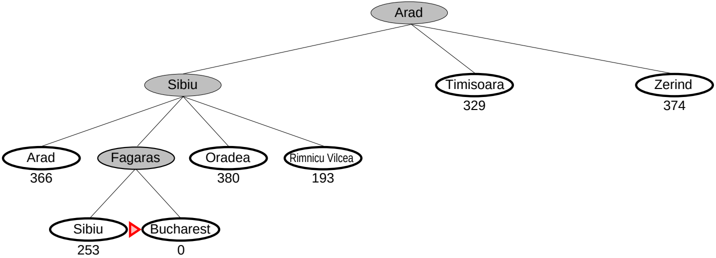
Only takes into account the heuristic, e.g. $f(n) = h(n)$.



Greedy search

Greedy best-first search

Only takes into account the heuristic, e.g. $f(n) = h(n)$.



Greedy search

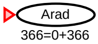
Greedy search properties

- **Completeness:** No – can get stuck in loops.
- **Time complexity:** $O(b^m)$, but depends on quality of heuristic.
- **Space complexity:** $O(b^m)$ – keeps all nodes in memory.
- **Optimality:** No.

A* search

A* search algorithm

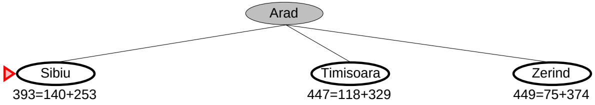
- Idea: avoid expanding paths that are already expensive.
- $f(n) = g(n) + h(n)$, where:
  - $g(n)$ – cost so far to reach n,
  - $h(n)$ – estimated cost from n to goal (heuristic), therefore
  - $f(n)$ – estimated total cost of path through n to goal.



A* search

A* search algorithm

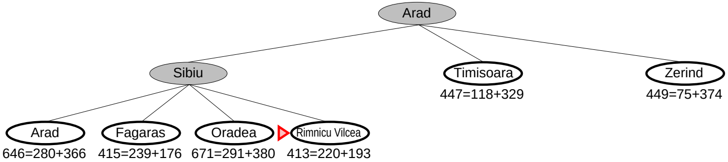
- Idea: avoid expanding paths that are already expensive.
- $f(n) = g(n) + h(n)$, where:
  - $g(n)$ – cost so far to reach n,
  - $h(n)$ – estimated cost from n to goal (heuristic), therefore
  - $f(n)$ – estimated total cost of path through n to goal.



A* search

A* search algorithm

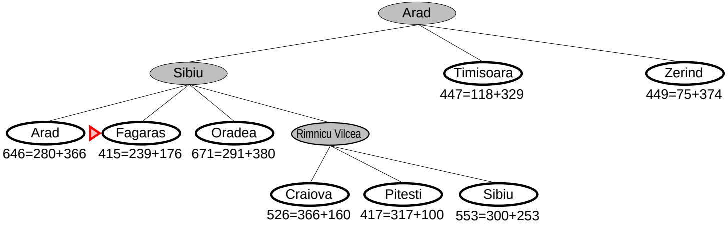
- Idea: avoid expanding paths that are already expensive.
- $f(n) = g(n) + h(n)$, where:
  - $g(n)$ – cost so far to reach n,
  - $h(n)$ – estimated cost from n to goal (heuristic), therefore
  - $f(n)$ – estimated total cost of path through n to goal.



A* search

A* search algorithm

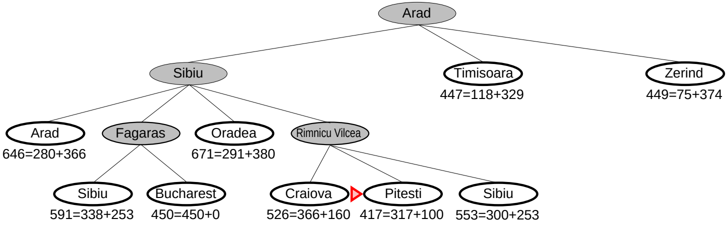
- Idea: avoid expanding paths that are already expensive.
- $f(n) = g(n) + h(n)$, where:
  - $g(n)$ – cost so far to reach n,
  - $h(n)$ – estimated cost from n to goal (heuristic), therefore
  - $f(n)$ – estimated total cost of path through n to goal.



A* search

A* search algorithm

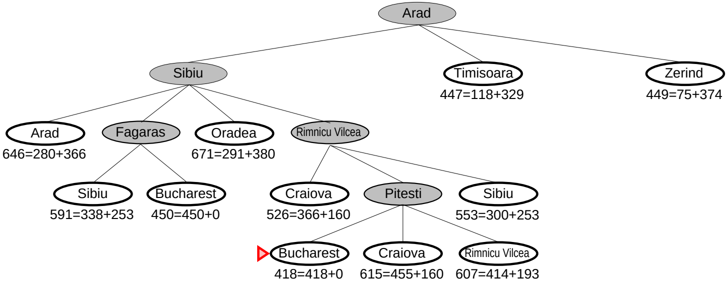
- Idea: avoid expanding paths that are already expensive.
- $f(n) = g(n) + h(n)$, where:
  - $g(n)$ – cost so far to reach n,
  - $h(n)$ – estimated cost from n to goal (heuristic), therefore
  - $f(n)$ – estimated total cost of path through n to goal.



A* search

A* search algorithm

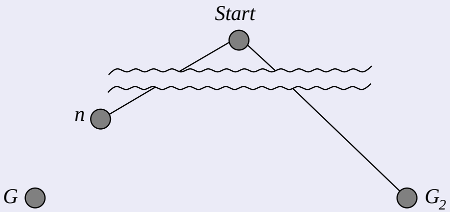
- Idea: avoid expanding paths that are already expensive.
- $f(n) = g(n) + h(n)$, where:
  - $g(n)$ – cost so far to reach n,
  - $h(n)$ – estimated cost from n to goal (heuristic), therefore
  - $f(n)$ – estimated total cost of path through n to goal.



A* optimality

Let's suppose that...

...a suboptimal goal G_2 has been generated. Let n be an unexpanded node on a shortest path to the optimal goal G.



Proof

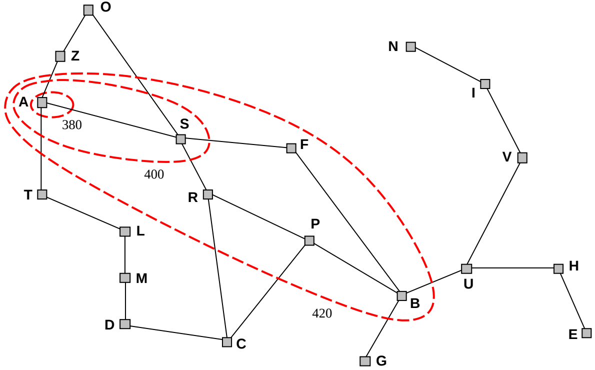
- $f(G_2) = g(G_2)$, since $h(G_2) = 0$
- $g(G_2) > g(G)$, since G_2 is suboptimal
- $g(G) \geq f(n)$, since h is admissible

Since $f(G_2) > f(n)$, A* will never select G_2 for expansion

A* with consistent heuristic

Node expansion order

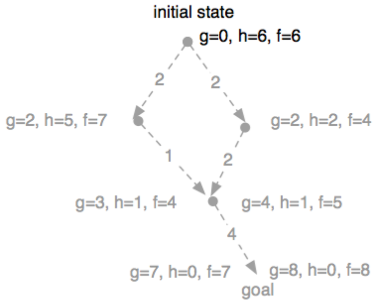
- If the heuristic is consistent, A* expands nodes in order of increasing f value.
- A* gradually adds "f-contours" of nodes; contour i has all nodes with $f = f_i$, where $f_i < f_{i+1}$.



A* with inconsistent heuristic

If *h* is inconsistent:

- As we go along the path, f may sometimes increase.
- A* doesn't always expand nodes in order of increasing f value, as it may find lower-cost paths to nodes it already expanded.
- A* should re-expand those nodes; however, the *Graph-Search* algorithm will not re-expand them.

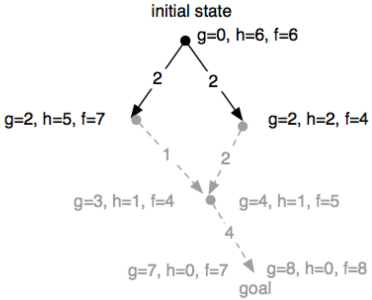


This example is due to Prof. Dana Nau.

A* with inconsistent heuristic

If h is inconsistent:

- As we go along the path, f may sometimes increase.
- A* doesn't always expand nodes in order of increasing f value, as it may find lower-cost paths to nodes it already expanded.
- A* should re-expand those nodes; however, the *Graph-Search* algorithm will not re-expand them.

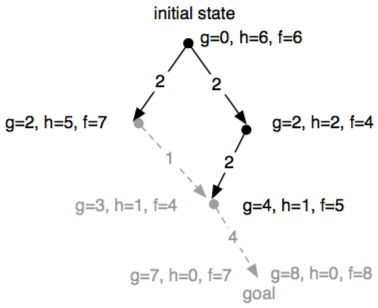


This example is due to Prof. Dana Nau.

A* with inconsistent heuristic

If h is inconsistent:

- As we go along the path, f may sometimes increase.
- A* doesn't always expand nodes in order of increasing f value, as it may find lower-cost paths to nodes it already expanded.
- A* should re-expand those nodes; however, the *Graph-Search* algorithm will not re-expand them.

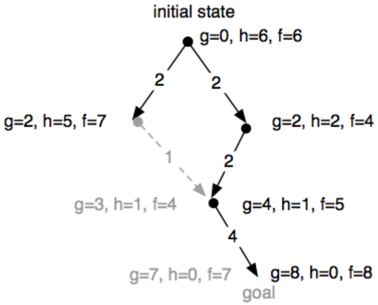


This example is due to Prof. Dana Nau.

A* with inconsistent heuristic

If h is inconsistent:

- As we go along the path, f may sometimes increase.
- A* doesn't always expand nodes in order of increasing f value, as it may find lower-cost paths to nodes it already expanded.
- A* should re-expand those nodes; however, the *Graph-Search* algorithm will not re-expand them.

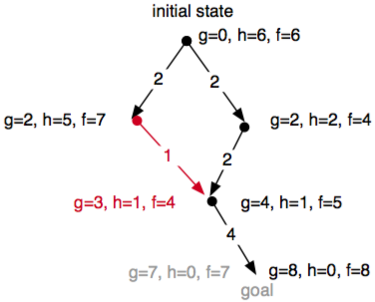


This example is due to Prof. Dana Nau.

A* with inconsistent heuristic

If h is inconsistent:

- As we go along the path, f may sometimes increase.
- A* doesn't always expand nodes in order of increasing f value, as it may find lower-cost paths to nodes it already expanded.
- A* should re-expand those nodes; however, the *Graph-Search* algorithm will not re-expand them.

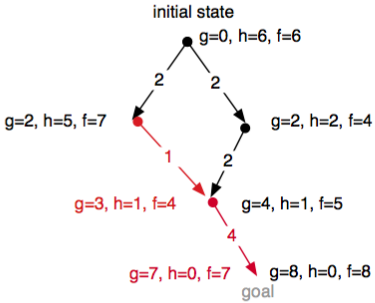


This example is due to Prof. Dana Nau.

A* with inconsistent heuristic

If h is inconsistent:

- As we go along the path, f may sometimes increase.
- A* doesn't always expand nodes in order of increasing f value, as it may find lower-cost paths to nodes it already expanded.
- A* should re-expand those nodes; however, the *Graph-Search* algorithm will not re-expand them.



This example is due to Prof. Dana Nau.

Properties of A*

A* properties

- **Completeness:** Yes, unless there are infinitely many nodes with $f < f(G)$.
- **Time complexity:** Exponential, depends on mean relative error of heuristic.
- **Space complexity:** Keeps all nodes in memory.
- **Optimality:** Yes, if:
  - for *Tree-Search*, the heuristic is admissible,
  - for *Graph-Search*, the heuristic is admissible and consistent.

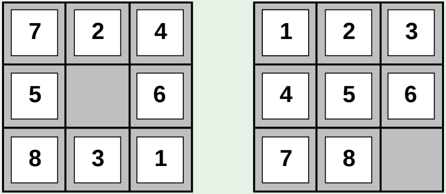
Heuristic functions

Heuristic function quality

- An *admissible* heuristic never overestimates the cost of reaching the goal.
- It usually underestimates – see the SLD heuristic.
- However, a *good* heuristic function tries to minimise the gap between its value and the actual cost.

Example

How would you design a heuristic for the 8-puzzle?

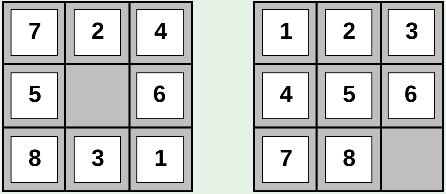


Start State **Goal State**

Heuristic functions

Example

How would you design a heuristic for the 8-puzzle?



Start State   Goal State

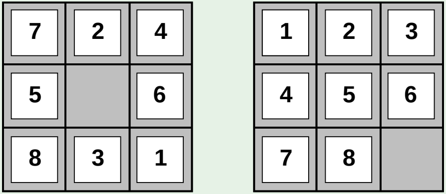
8-puzzle: two possible heuristics

- $h_1(n)$: number of tiles not in their places
- $h_2(n)$: sum of Manhattan distances of all tiles from their places

Heuristic functions

Example

How would you design a heuristic for the 8-puzzle?



Start State **Goal State**

8-puzzle: two possible heuristics

- $h_1(S) = 6$
- $h_2(S) = 4 + 0 + 3 + 3 + 1 + 0 + 2 + 1 = 14$

Heuristic functions

8-puzzle: two possible heuristics

- $h_1(S) = 6$
- $h_2(S) = 4 + 0 + 3 + 3 + 1 + 0 + 2 + 1 = 14$

Heuristic domination

- If $h_2(n) \geq h_1(n)$ for all n, then h_2 *dominates* h_1.
- Therefore, h_2 is *always* better than h_1.

Performance comparison

Number of nodes expanded if solution is at depth d:

d	IDS	A*(h_1)	A*(h_2)
14	3 473 951	539	113
24	54 000 000 000	39 135	1 641

Heuristic functions

Heuristic domination

- If $h_2(n) \geq h_1(n)$ for all n, then h_2 *dominates* h_1.
- Therefore, h_2 is *always* better than h_1.

Hybrid heuristics

If h_a and h_b are admissible heuristics, then

$$h(n) = \max(h_a(n), h_b(n))$$

is admissible and dominates both h_a and h_b.

Deriving admissible heuristics

Admissible heuristics can be derived from the *exact* solution cost of a *relaxed* version of the problem.

Heuristic functions

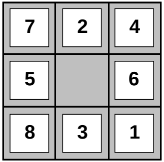
Deriving admissible heuristics

Admissible heuristics can be derived from the *exact* solution cost of a *relaxed* version of the problem.

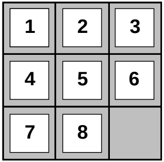
Example: 8-puzzle

For the 8-puzzle problem:

- If the rules are relaxed so that a tile can move *anywhere*, then $h_1(n)$ (number of misplaced tiles) gives the exact cost.
- If the rules are relaxed so that a tile can can move to *any adjacent square* (regardless of whether it's occupied), then $h_2(n)$ (sum of Manhattan distances) gives the exact cost.



Start State



Goal State

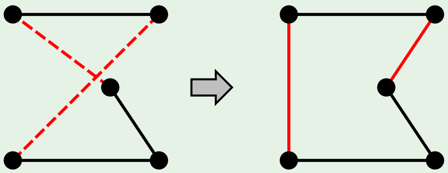
Iterative improvement algorithms

Iterative improvement

- For many problems, the *solution* is important, not how we got there – the path is irrelevant.
- State space: set of "complete" configurations; find *optimal* configuration (e.g., TSP) or configuration satisfying constraints (e.g., timetable).
- Iterative improvement: keep current state, try to improve it.

Example: Travelling Salesman Problem

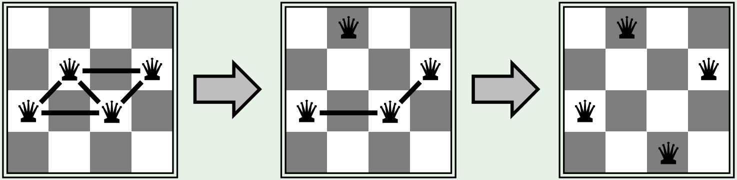
Start with any complete tour, then try to switch pairs.



Iterative improvement algorithms

Example: *n*-queens

- Put n queens on an $n \times n$ board so that no two queens conflict on same row, column or diagonal.
- Move a queen to reduce number of conflicts.



h = 5 **h = 2** **h = 0**

Hill-climbing search

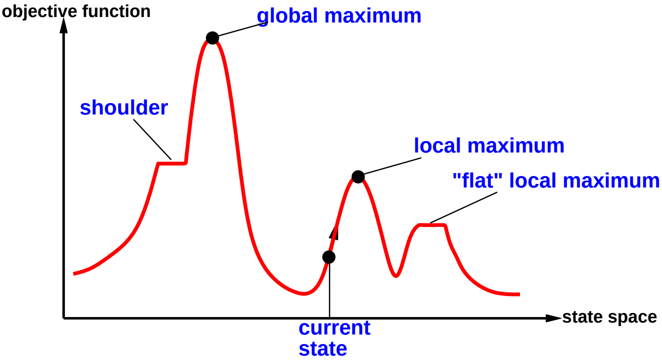
Hill-climbing ascent/descent search

- Also called *local greedy search algorithms*.
- *"Like climbing Mt. 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] $\leq$ VALUE[current] **then return** STATE[*current*]
    current ← *neighbor*
  end

Hill-climbing search



Hill-climbing algorithm can get stuck on:

- local maxima,
- ridges,
- flats.

Hill-climbing search

Variants of hill-climbing search

- If an algorithm (e.g., 'standard' hill-climbing) only selects moves which improve the solution, it is bound to be incomplete. Random-start algorithms are complete, but very ineffective.
- **Stochastic hill-climbing** picks one of the better successors at random.
- **First-choice hill-climbing** generates random successors until it finds one better than the current state.
- **Random-restart hill-climbing** conducts a series of searches, starting with random start states. It is complete with probability close to 1, because it must eventually generate the goal state.

Simulated annealing

Background

In metallurgy, *annealing* is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cool them. This allows the material to coalesce into a low-energy crystalline state.

The algorithm

- Selects move at random.
- If the move improves the solution, it is made every time.
- If the move worsens the solution, the probability if it is made is determined upon:
  - the amount ΔE by which the solution is worsened,
  - the so-called temperature T, which is decreased in every iteration.[a]

[a] That way, the bad moves are more likely to be allowed at the start, and become less likely as the temperature decreases.

Simulated annealing

```
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^(ΔE/T)
```

Local beam search

Local beam search

- Idea: keep k states instead of 1; choose top k of all their successors.
- Not the same as k individual searches run in parallel.
- Searches that find good states invite others to join them.
- Threads that do not provide good results are discarded.

Problem

Often, all k states end up on same local maximum.

Solution: stochastic local beam search

Choose k successors at random, but still be biased towards the good ones.

Genetic algorithms

Genetic algorithms

- Variant of stochastic beam search.
- Successor states generated by combining *two* parent states (rather than modifying a single state).
- We have k randomly generated states, called the *population*.
- Each state is caled an *individual*.
- Each individual is represented as a string over a finite alphabet.

Steps in GA

1. Evaluate *fitness* of each individual in population, as probability of being selected for reproduction.
2. Perform *selection*, taking fitness into account.
3. Perform *crossover* within pairs (at random point).
4. Introduce random *mutation* to allow for stochastic exploration.

Genetic algorithms