COMP3411/9814: Artificial Intelligence

2c. Heuristic Path Search

Alan Blair

School of Computer Science and Engineering

February 21, 2024

Search Strategies

General Search algorithm:

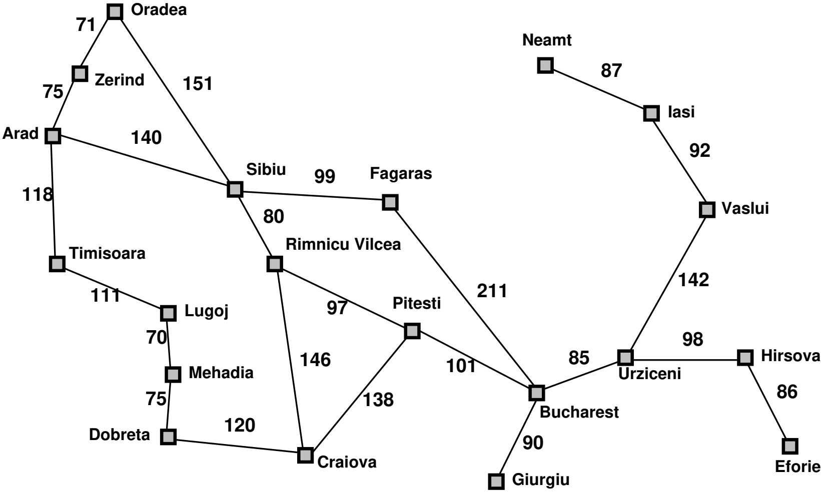
- ➛ add initial state to priority queue
- ➛ repeat:
  - → take selected node from priority queue
  - → test if it is a goal state; if so, terminate
  - → “expand” it, i.e. generate successor nodes and add them to the queue

Search strategies are distinguished by the order in which new nodes are added to (or removed from) the queue of nodes awaiting expansion.

Search Strategies

- BFS and DFS treat all new nodes the same way:
  - BFS add all new nodes to the **back** of the queue
  - DFS add all new nodes to the **front** of the queue
- (Seemingly) **Best First Search** uses an evaluation function $f()$ to order the nodes in the queue; we have seen one example of this:
  - UCS $f(n)$ = cost $g(n)$ of path from root to node n
- **Informed** or **Heuristic** search strategies incorporate into $f()$ an estimate of the distance to goal
  - Greedy Search $f(n) =$ estimate $h(n)$ of cost from node n to goal
  - A* Search $f(n) = g(n) + h(n)$

Romania Street Map



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 Function

There is a whole family of Best First Search algorithms with different evaluation functions $f()$. A key component of these algorithms is a **heuristic function**:

- **Heuristic function** h: **{Set of nodes}** $\longrightarrow$ **R** :
  - $h(n)$ = estimated cost of the cheapest path from current node n to goal node.
  - in the context of search, **heuristic functions** are problem specific functions that provide an estimate of solution cost.

Greedy Best-First Search

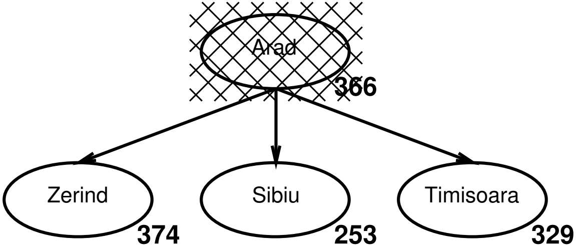
- **Greedy Best-First Search:** Best-First Search that selects the next node for expansion using the heuristic function for its evaluation function, i.e.

  $f(n) = h(n)$
- $h(n) = 0 \iff n$ is a goal state
- i.e. Greedy Search minimises the estimated cost to the goal;
  it expands whichever node n is estimated to be closest to the goal.
- Greedy: tries to “bite off” as big a chunk of the solution as possible, without worrying about long-term consequences.

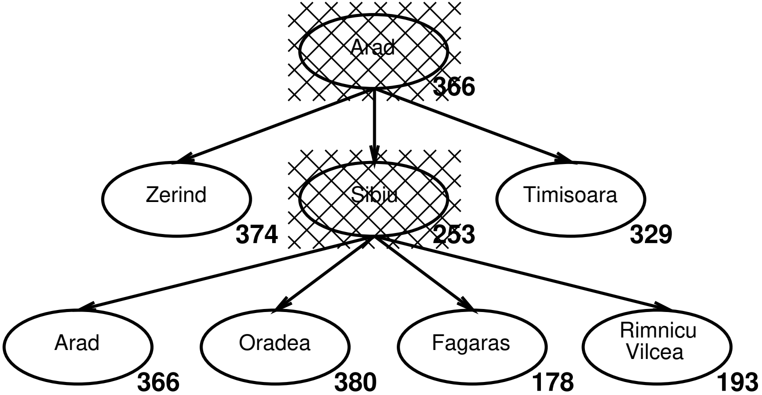
Straight Line Distance as a Heuristic

- $h_{\mathrm{SLD}}(n)$ = straight-line distance between n and the goal location (Bucharest).
- Assume that roads typically tend to approximate the direct connection between two cities.
- Need to know the map coordinates of the cities:
  - $\rightarrow \sqrt{(Sibiu_x - Bucharest_x)^2 + (Sibiu_y - Bucharest_y)^2}$

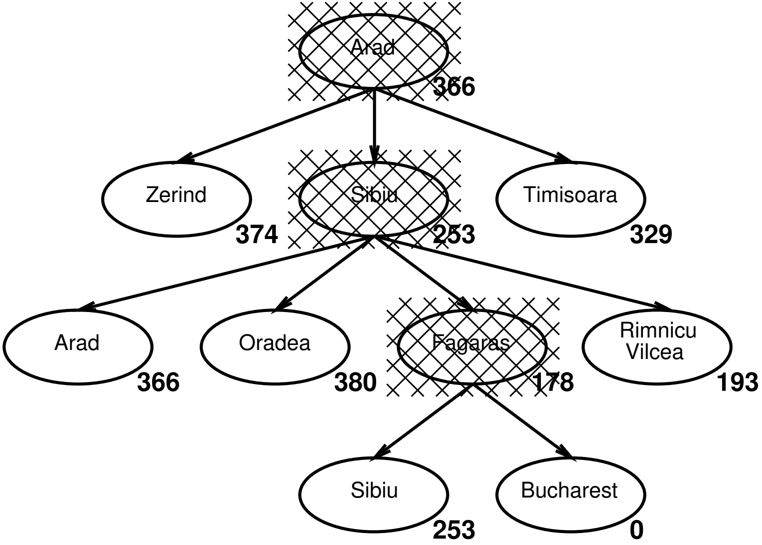
Greedy Best-First Search Example



Greedy Best-First Search Example



Greedy Best-First Search Example



Examples of Greedy Best-First Search

Try:

- Iasi to Fagaras
- Fagaras to Iasi
- Rimnicu Vilcea to Lugoj

Properties of Greedy Best-First Search

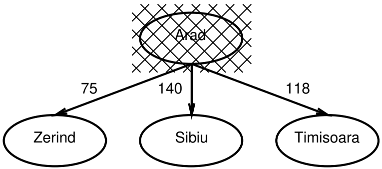
- **Complete:** No! can get stuck in loops, e.g.,
  Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow$ ...
  Complete in finite space with repeated-state checking
- **Time:** $\mathcal{O}(b^m)$, where m is the maximum depth in search space.
- **Space:** $\mathcal{O}(b^m)$ (retains all nodes in memory)
- **Optimal:** No! e.g., the path Sibiu $\rightarrow$ Fagaras $\rightarrow$ Bucharest is 32 km longer than Sibiu $\rightarrow$ Rimnicu Vilcea $\rightarrow$ Pitesti $\rightarrow$ Bucharest.

Therefore Greedy Search has the same deficits as Depth-First Search.
However, a good heuristic can reduce time and memory costs substantially.

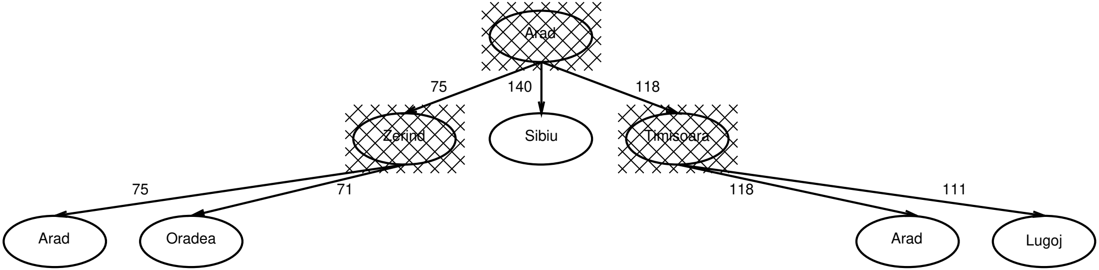
Recall: Uniform-Cost Search

- Expand root first, then expand least-cost unexpanded node
- **Implementation**: QUEUEINGFN = insert nodes in order of increasing path cost.
- Reduces to breadth-first search when all actions have same cost
- Finds the cheapest goal provided path cost is monotonically increasing along each path (i.e. no negative-cost steps)

Uniform Cost Search



Uniform Cost Search



Properties of Uniform Cost Search

- **Complete?** Yes, if b is finite and step costs $\geq \epsilon$ with $\epsilon > 0$.
- **Optimal?** Yes.
- Guaranteed to find optimal solution, but does so by exhaustively expanding all nodes closer to the initial state than the goal.

Q: can we still guarantee optimality but search more efficiently,
by giving priority to more "promising" nodes?

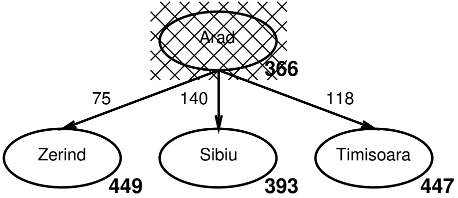
A* Search

- A* Search uses evaluation function $f(n) = g(n) + h(n)$
  - $g(n) =$ cost from initial node to node n
  - $h(n) =$ estimated cost of cheapest path from n to goal
  - $f(n) =$ estimated total cost of cheapest solution through node n
- Greedy Search minimizes $h(n)$
  - efficient but not optimal or complete
- Uniform Cost Search minimizes $g(n)$
  - optimal and complete but not efficient

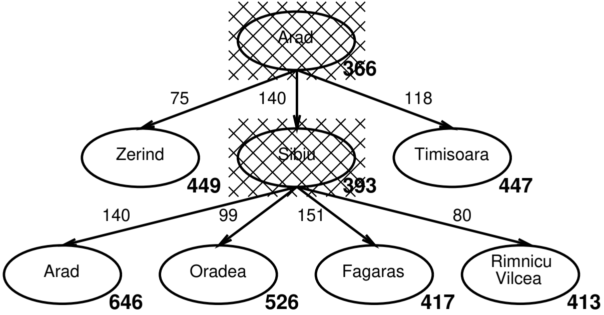
A* Search

- A* Search minimizes $f(n) = g(n) + h(n)$
  - → idea: preserve efficiency of Greedy Search but avoid expanding paths that are already expensive
- **Q**: is A* Search **optimal** and **complete** ?
- **A**: Yes! provided $h()$ is **admissible** in the sense that it never overestimates the cost to reach the goal.

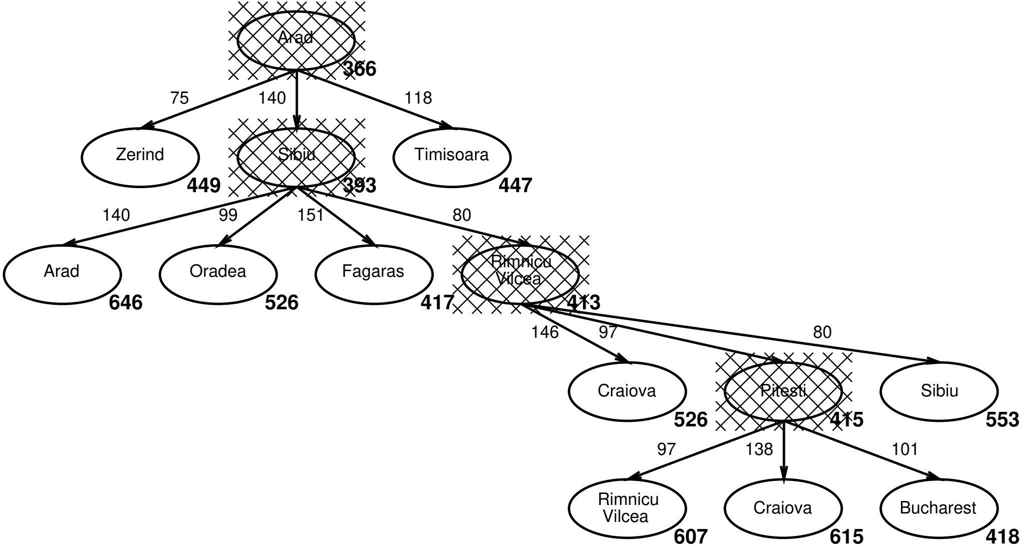
A* Search Example



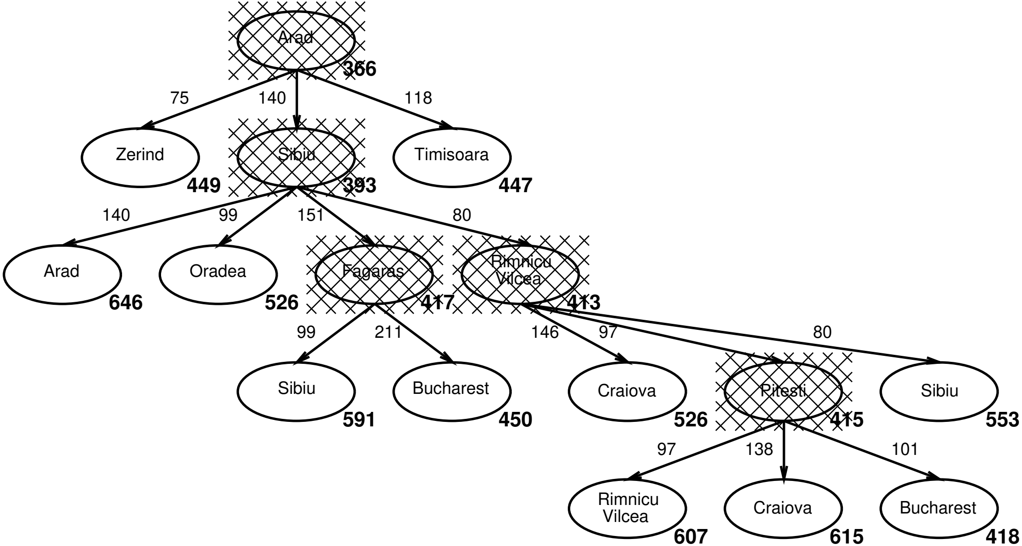
A* Search Example



A* Search Example



A* Search Example

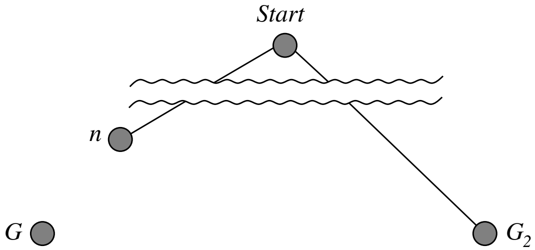


A* Search

- Heuristic $h()$ is called **admissible** if
  $\forall n\ h(n) \leq h^*(n)$ where $h^*(n)$ is **true** cost from n to goal
- If h is admissible then $f(n)$ never overestimates the actual cost of the best solution through n.
- Example: $h_{\text{SLD}}()$ is admissible because the shortest path between any two points is a line.
- Theorem: A* Search is optimal if $h()$ is admissible.

Optimality of A* Search

Suppose a suboptimal goal node G_2 has been generated and is in the queue.
Let n be the last unexpanded node on a shortest path to an optimal goal node G.



$$\begin{aligned} f(G_2) &= g(G_2) && \text{since } h(G_2) = 0 \\ &> g(G) && \text{since } G_2 \text{ is suboptimal} \\ &\geq f(n) && \text{since } h \text{ is admissible.} \end{aligned}$$

Optimality of A* Search

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

Note: suboptimal goal node G_2 may be **generated**, but it will never be **expanded**.

In other words, even after a goal node has been generated, A* will keep searching so long as there is a possibility of finding a shorter solution.

Once a goal node is selected for **expansion**, we know it must be optimal, so we can terminate the search.

Properties of A* Search

- **Complete:** Yes, unless there are infinitely many nodes with $f \leq$ cost of solution.
- **Time:** Exponential in [relative error in $h \times$ length of solution]
- **Space:** Keeps all nodes is memory
- **Optimal:** Yes (assuming $h()$ is admissible).

Iterative Deepening A* Search

- Iterative Deepening A* is a low-memory variant of A* which performs a series of depth-first searches, but cuts off each search when the sum $f() = g() + h()$ exceeds some pre-defined threshold.
- The threshold is steadily increased with each successive search.
- IDA* is asymptotically as efficient as A* for domains where the number of states grows exponentially.

Exercise

What kind of search will Greedy Search emulate if we run it with:

- $h(n) = -g(n)$?
- $h(n) = \ \ g(n)$?
- $h(n) =$ number of steps from initial state to node n ?

Examples of Admissible Heuristics

e.g. for the 8-puzzle:

$h_1(n)$ = total number of misplaced tiles

$h_2(n)$ = total **Manhattan distance** = $\sum$distance from goal position

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

$h_1(S)$ = ?

$h_2(S)$ = ?

➢ Why are h_1, h_2 admissible?

Examples of Admissible Heuristics

e.g. for the 8-puzzle:

$h_1(n)$ = total number of misplaced tiles

$h_2(n)$ = total **Manhattan distance** = $\sum$distance from goal position

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

$h_1(S)$ = 6

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

- ➢ h_1: every tile must be moved at least once.
- ➢ h_2: each action can only move one tile one step closer to the goal.

Dominance

- if $h_2(n) \geq h_1(n)$ for all n (both admissible) then h_2 **dominates** h_1 and is better for search. So the aim is to make the heuristic $h()$ as large as possible, but without exceeding $h^*()$.
- typical search costs:

14-puzzle	IDS	= 3,473,941 nodes
	A*(h_1)	= 539 nodes
	A*(h_2)	= 113 nodes
24-puzzle	IDS	$\approx$ 54 $\times$ 10 ^{9} nodes
	A*(h_1)	= 39,135 nodes
	A*(h_2)	= 1,641 nodes

How to Find Heuristic Functions ?

➛ Admissible heuristics can often be derived from the **exact** solution cost of a simplified or “relaxed” version of the problem.

(i.e. with some of the constraints weakened or removed)

- → 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.

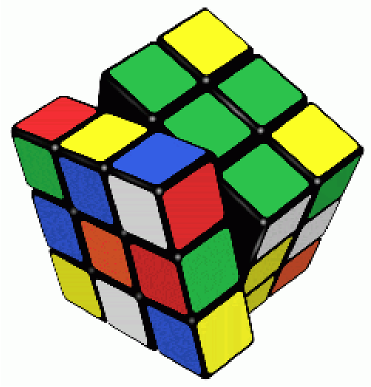
Composite Heuristic Functions

- Let $h_1, h_2, ..., h_m$ be admissible heuristics for a given task.
- Define the **composite heuristic**

$$h(n) = \max(h_1(n), h_2(n), ..., h_m(n))$$

- h is admissible
- h dominates $h_1, h_2, \ldots, h_m$

Rubik's Cube



Heuristics for Rubik's Cube

- 3D Manhattan distance, but to be admissible need to divide by 8.
- better to take 3D Manhattan distance for edges only, divided by 4.
- alternatively, max of 3D Manhattan distance for edges and corners, divided by 4 (but the corners slow down the computation without much additional benefit).
- best approach is to pre-compute **Pattern Databases** which store the minimum number of moves for every combination of the 8 corners, and for two sets of 6 edges.
- to save memory, use IDA*.

"Finding Optimal Solutions to Rubik's Cube using Pattern Databases" (Korf, 1997)

Summary of Informed Search

- Heuristics can be applied to reduce search cost.
- Greedy Search tries to minimize cost from current node n to the goal.
- A* combines the advantages of Uniform-Cost Search and Greedy Search.
- A* is complete, optimal and optimally efficient among all optimal search algorithms.
- Memory usage is still a concern for A*. IDA* is a low-memory variant.