Heuristic Search

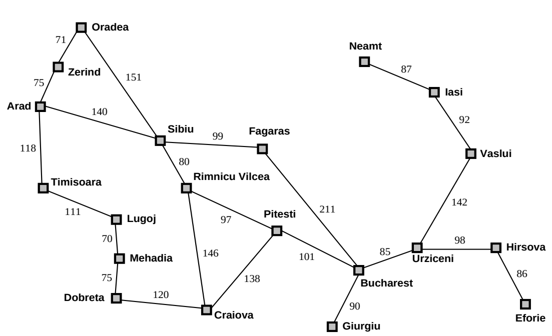
Heuristic Search

- **Heuristic** or **informed search** exploits additional knowledge about the problem that helps direct search to more promising paths.
- A **heuristic function,** *h(n)*, provides an estimate of the cost of the path from a given node to the closest goal state. Must be zero if node represents a goal state.
  - Example: Straight-line distance from current location to the goal location in a road navigation problem.
- Many search problems are NP-complete so in the worst case still have exponential time complexity; however a good heuristic can:
  - Find a solution for an average problem efficiently.
  - Find a reasonably good but not optimal solution efficiently.

Best-First Search

- At each step, best-first search sorts the queue according to a heuristic function.

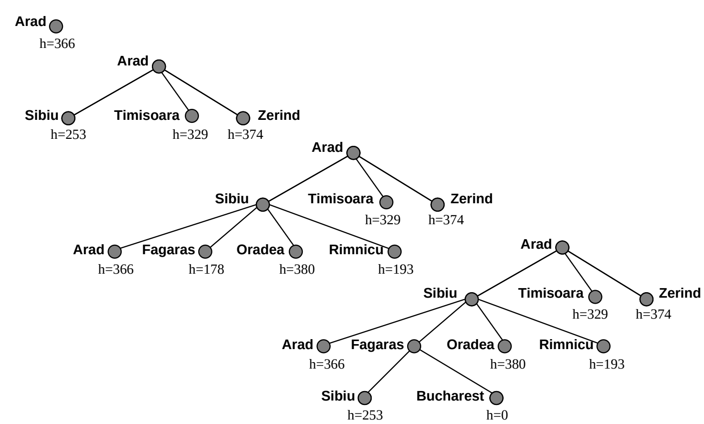
```
function BEST-FIRST-SEARCH( problem, EVAL-FN) returns a solution sequence
   inputs: problem, a problem
           Eval-Fn, an evaluation function

   Queueing-Fn ← a function that orders nodes by EVAL-FN
   return GENERAL-SEARCH( problem, Queueing-Fn)
```



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

Best-First Example



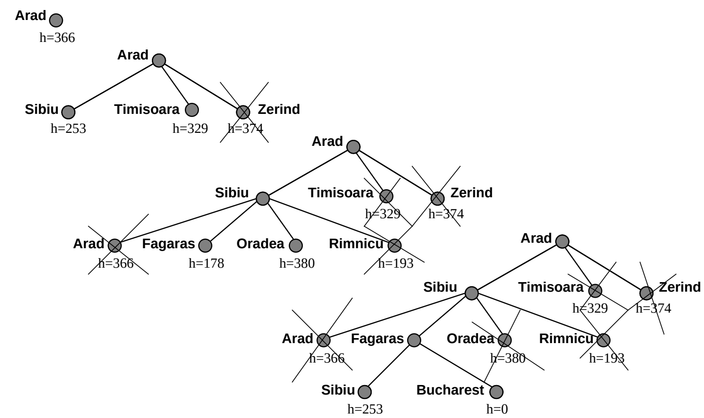
- Does not find shortest path to goal (through Rimnicu) since it is only focused on the cost remaining rather than the total cost.

Best-First Properties

- Not complete, may follow infinite path if heuristic rates each state on such a path as the best option. Most reasonable heuristics will not cause this problem however.
- Worst case time complexity is still O(b^m) where m is the maximum depth.
- Since must maintain a queue of all unexpanded states, space-complexity is also O(b^m).
- However, a good heuristic will avoid this worst-case behavior for most problems.

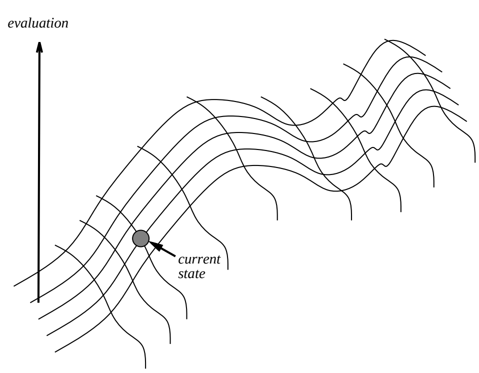
Beam Search

- Space and time complexity of storing and sorting the complete queue can be too inefficient.
- Beam search trims queue to the best *n* options (*n* is called the **beam width**) at each point.
- Focuses search more but may eliminate solution even for finite seach graphs
- Example for *n*=2.



Hill-Climbing

- Beam search with a beam-width of 1 is called **hill-climbing**.
- Pursues locally best option at each point, i.e. the best successor to the current best node.
- Subject to local maxima, plateaux, and ridges.

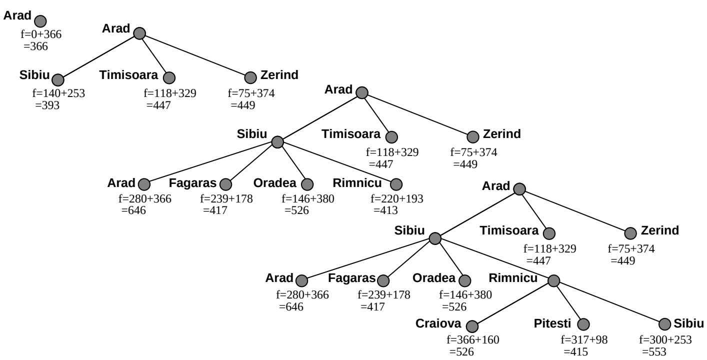


Minimizing Total Path Cost: A* Search

- A* combines features of uniform cost search (complete, optimal, inefficient) with best-first (incomplete, non-optimal, efficient).
- Sort queue by estimated total cost of the completion of a path.

  $f(n) = g(n) + h(n)$
- If the heuristic function always underestimates the distance to the goal, it is said to be **admissible.**
- If *h* is admissible, then *f*(*n*) never overestimates the actual cost of the best solution through *n*.

A* Example



- Finds the optimal path to Bucharest through Rimnicu and Pitesti

Optimality of A*

- If h is admissible, A* will always find a least cost path to the goal.

- Proof by contradiction:

  Let G be an optimal goal state with a path cost f^*
  Let G_2 be a suboptimal goal state supposedly found by A*
  Let n be a current leaf node on an optimal path to G

  Since h is admissible:
  $f^* >= f(n)$

  If G_2 is chosen for expansion over n then:
  $f(n) >= f(G_2)$

  Therefore:
  $f^* >= f(G_2)$

  Since G_2 is a goal state, $h(G_2)$=0, therefore
  $f(G_2) = g(G_2)$
  $f^* >= g(G_2)$

  Therefore G_2 is optimal.
  Contradiction.
  Q.E.D.

Other Properties of A*

- A* is complete as long as
  - Branching factor is always finite
  - Every operator adds cost at least $\delta > 0$

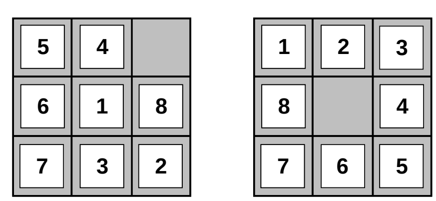
- Time and space complexity still $O(b^m)$ in the worst case since must maintain and sort complete queue of unexplored options.

- However, with a good heuristic can find optimal solutions for many problems in reasonable time.

- Again, space complexity is a worse problem than time.

Heuristic Functions

- 8-puzzle search space
  - Typical solution length: 20 steps
  - Average branching factor: 3
  - Exhaustive search: 3^{20}=3.5 x 10^9
  - Bound on unique states: 9! = 362,880



Start State Goal State

- Admissible Heuristics:
  - Number of tiles out of place (h_1): 7
  - City-block (Manhattan) distance (h_2): 2+3+3+2+4+2+0+2=18

Heuristic Performance

- Experiments on sample problems can determine the number of nodes searched and CPU time for different strategies.

- One other useful measure is **effective branching factor**: If a method expands N nodes to find solution of depth d, and a uniform tree of depth d would require a branching factor of b^* to contain N nodes, the effective branching factor is b^*

  $N = 1 + b^* + (b^*)^2 + ...+ (b^*)^d$

- Experimental Results on 8-puzzle problems

	Search Cost			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Quality of Heuristics

- ISince A* expands all nodes whose f value is less than that of an optimal solution, it is always better to use a heuristic with a higher value as long as it does not over-estimate.

- Therefore h_2 is uniformly better than h_1, or h_2 **dominates** h_1.

- A heuristic should also be easy to compute, otherwise the overhead of computing the heuristic could outweigh the time saved by reducing search (e.g. using full breadth-first search to estimate distance wouldn't help).

Inventing Heuristics

- Many good heuristics can be invented by considering relaxed versions of the problem (abstractions).

- For 8-puzzle:

  A tile can move from square A to B if A is adjacent to B and B is blank

  (a) A tile can move from square A to B if A is adjacent to B.
  (b) A tile can move from square A to B if B is blank.
  (c) A tile can move from square A to B.

- If there are a number of features that indicate a promising or unpromising state, a weighted sum of these features can be useful. Learning methods can be used to set weights.