Informed Search Algorithms

Ways of Incorporating Knowledge and Constraints into Search Algorithms

- **Problem Formulation**

  Choice of relevant states

  Choice of operators

  Definition of goal

- **Heuristic Evaluation Function**

  $f(s)$ gives a "value" for each state.

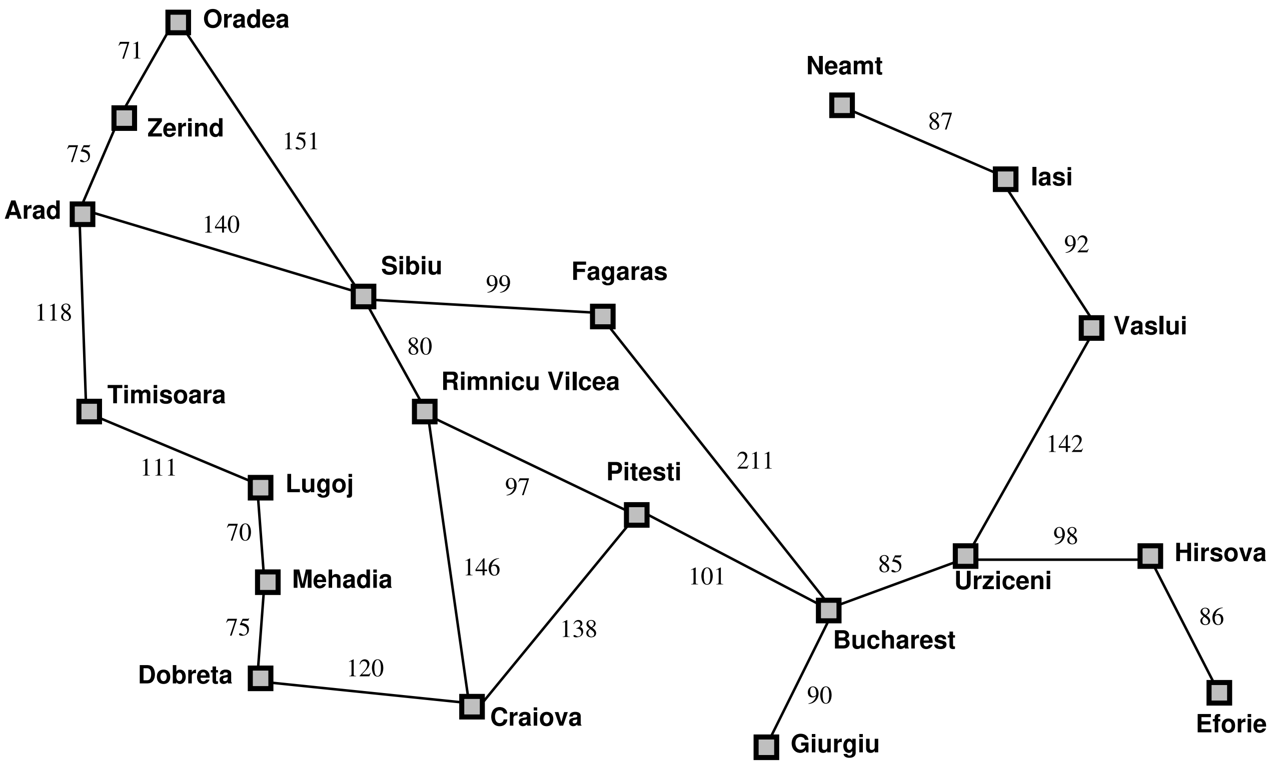
```
function BEST-FIRST-SEARCH(problem, EVAL-FN) returns a solution sequence
   inputs: problem, a problem
           Eval-Fn, an evaluation function

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

Cost-to-Go Functions

Common kind of evaluation function: $f(s) = h(s)$ estimates the distance remaining to a goal state

Example: $h(s)$ = straight-line distance to goal

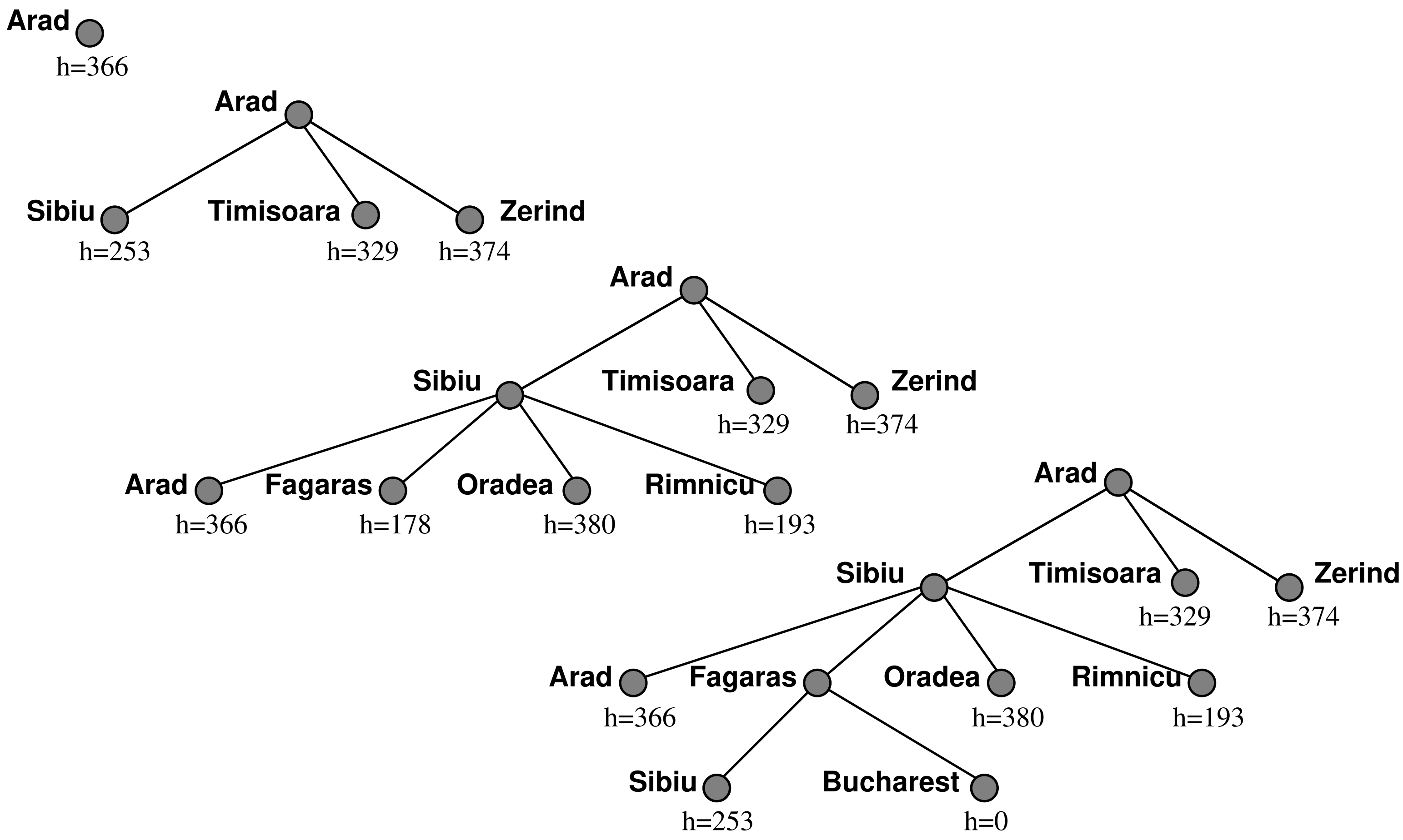


Straight–line distance
to Bucharest

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

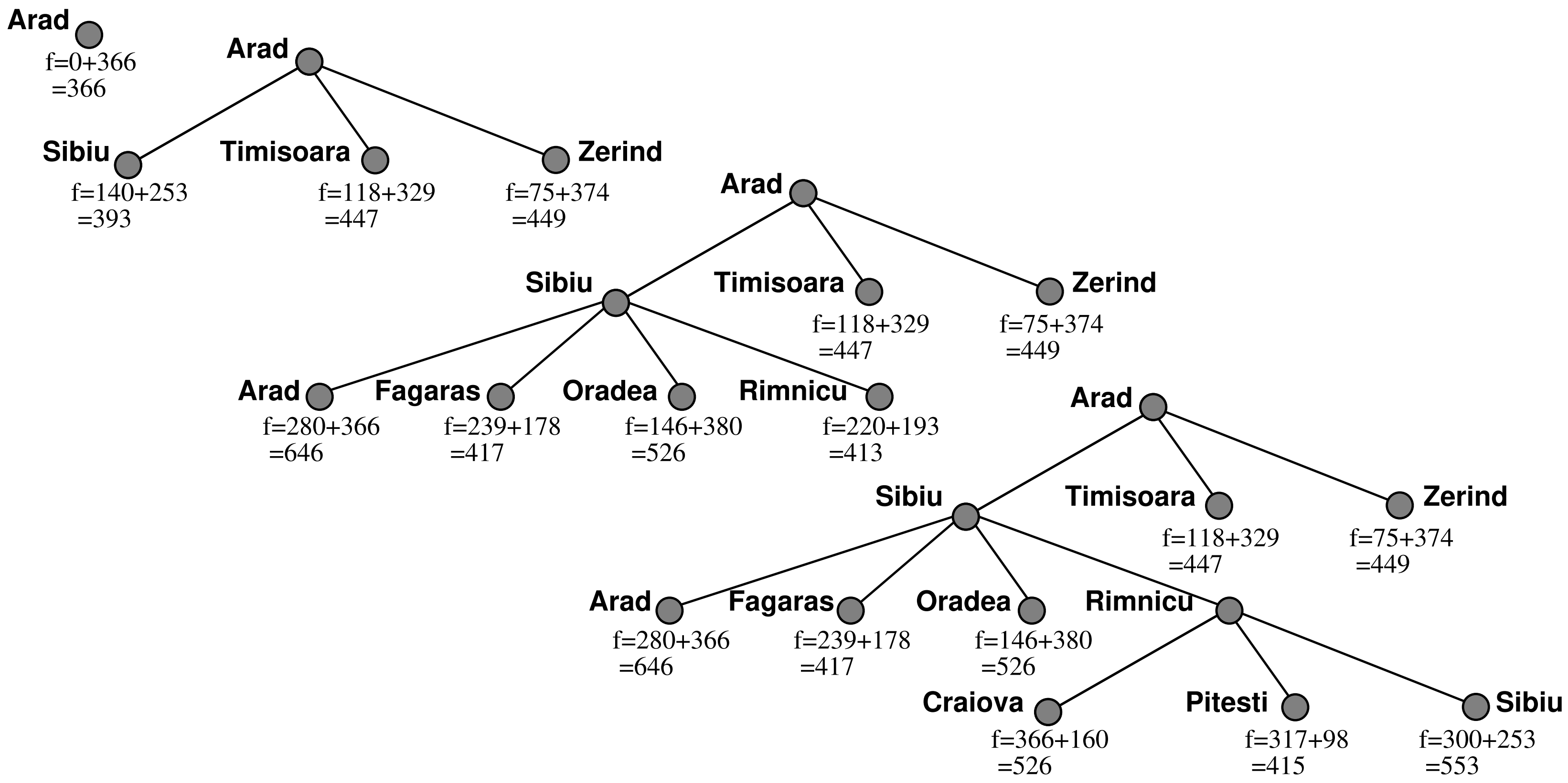
The Greedy Algorithm

Expand the node that has the smallest value for $h(s)$

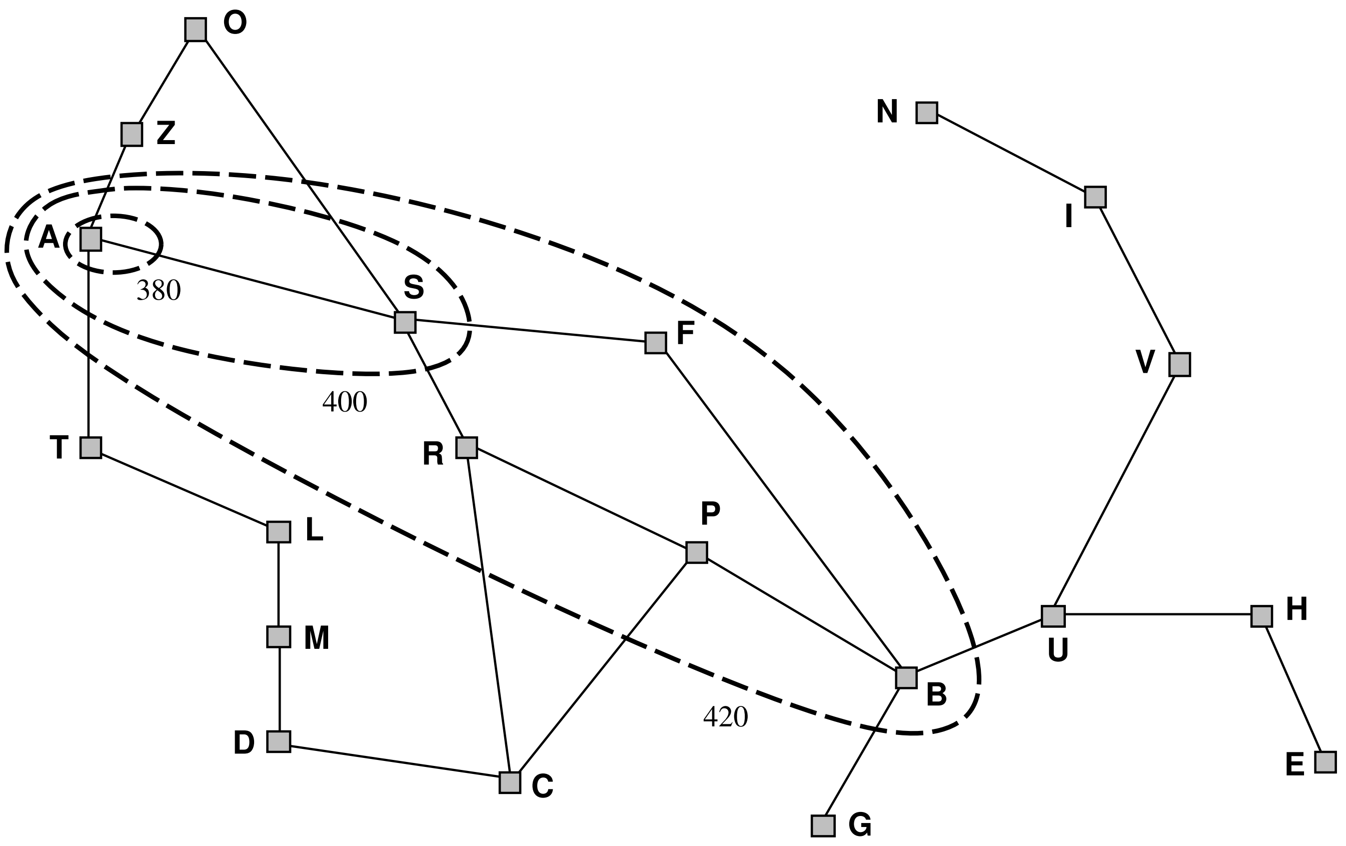


The A^* Algorithm

$f(s) = g(s) + h(s)$ **total estimated cost of getting to the goal via** s



Total Cost Contours



Assumption: $h(s)$ is monotonically decreasing.

Optimality of A^*

If $h(s)$ underestimates the true distance to the goal, then A^* will find the optimal solution.

Proof: The cost contour that includes the optimal path will be expanded first.

Effective Branching Factor for Two Heuristics for 8 Puzzle

- h_1: Number of tiles out-of-place.
- h_2: Total manhattan distance for out-of-place tiles.

	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

Inventing Good h Functions by Problem Relaxation

- **8-puzzle:**

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

- **Remove blank requirement**

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

  (manhattan distance heuristic)

- **Remove adjacency requirement**

  A tile can move from square A to square B if B is blank.

  (number of tiles out of place)

- **Remove both requirements**

  A tile can move from square A to square B

  (number of tiles out of place)

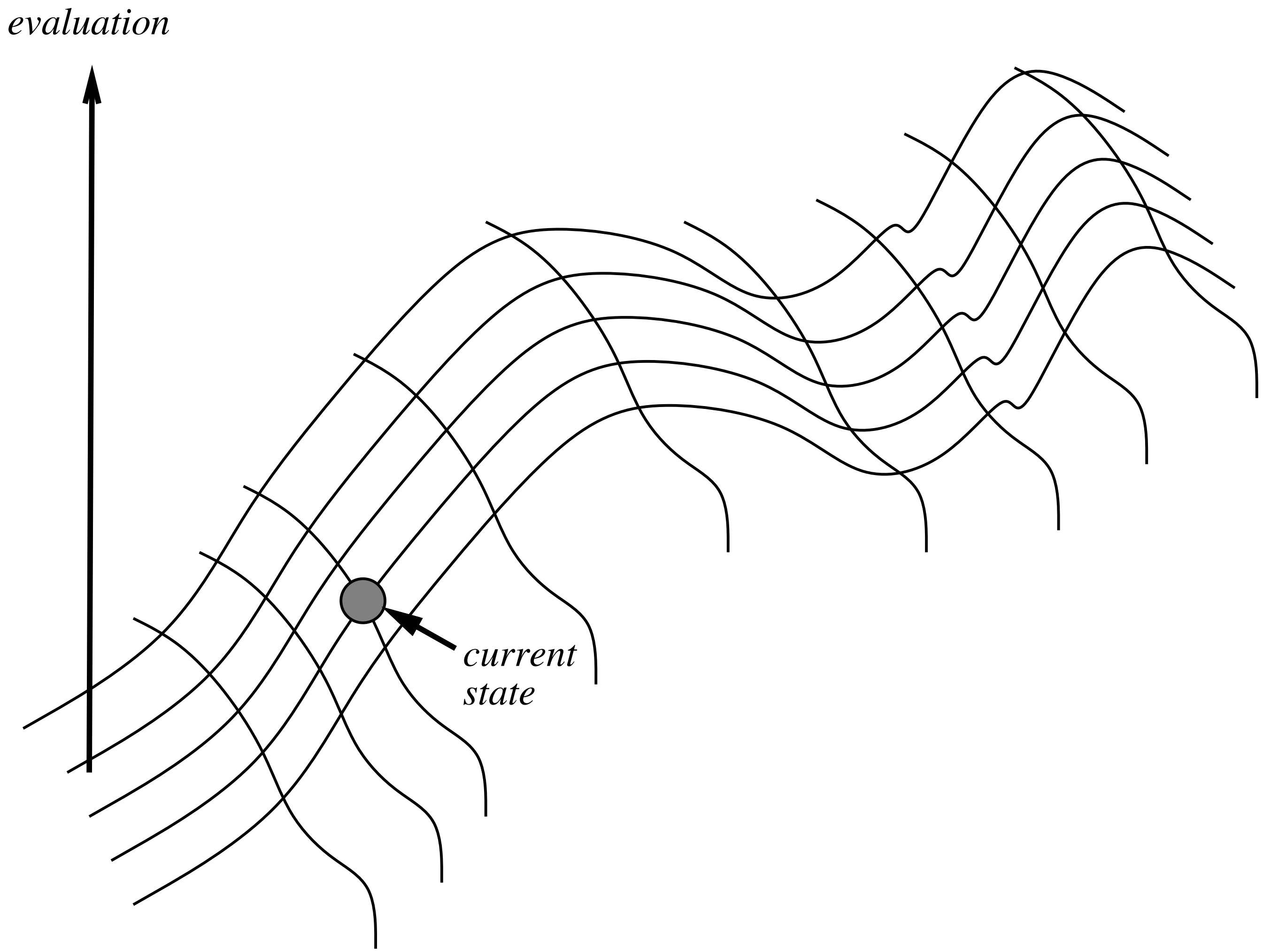
Local Search

```
Procedure LocalSearch(problem, SelectionMethod, StoppingRule)

state = initial state
while true do
  children = Expand(state)
  if (StoppingRule(state, children)) return state
  state = SelectionMethod(state, children);
```

Hill-Climbing Algorithm

- **SelectionMethod:** Choose child with largest value of $f(s)$.
- **StoppingRule:** Stop if all children have $f(s) \leq f(\text{state})$. ("local maximum")


evaluation
current
state

Gradient Ascent (or Descent)

Suppose the state space is continuous (and real-valued or vector-valued).

Each state has infinitely many possible children!

If $f(s)$ is differentiable, then we can compute the direction of "steepest ascent" and take a step in that direction.

```
Procedure GradientAscent(problem, f, f', StepSize)

state = initial state
while |f'(state)| > 0 do
  state := state + StepSize * f'(state)

return state
```

Simulated Annealing

Escape Local Minima via Probabilistic Hill-Climbing

function SIMULATED-ANNEALING(*problem, schedule*) **returns** a solution state
    inputs: *problem*, a problem
            schedule, a mapping from time to "temperature"
    static: *current*, a node
            next, a node
            T, a "temperature" controlling the probability of downward steps

    current ← MAKE-NODE(INITIAL-STATE[*problem*])
    for $t \leftarrow 1$ **to** ∞ **do**
        $T \leftarrow$ *schedule*[t]
        if T=0 **then return** *current*
        next ← a randomly selected successor of *current*
        $\Delta E \leftarrow$ VALUE[*next*] – VALUE[*current*]
        if $\Delta E > 0$ **then** *current* ← *next*
        else *current* ← *next* only with probability $e^{\Delta E/T}$