Informed Search

Introduction to Artificial Intelligence

G. Lakemeyer

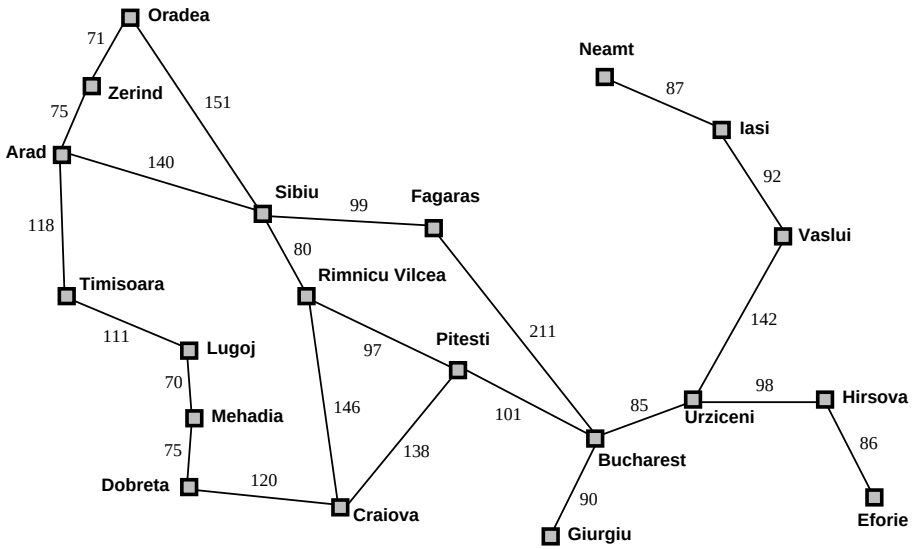
Winter Term 2018/19

Best-First Search

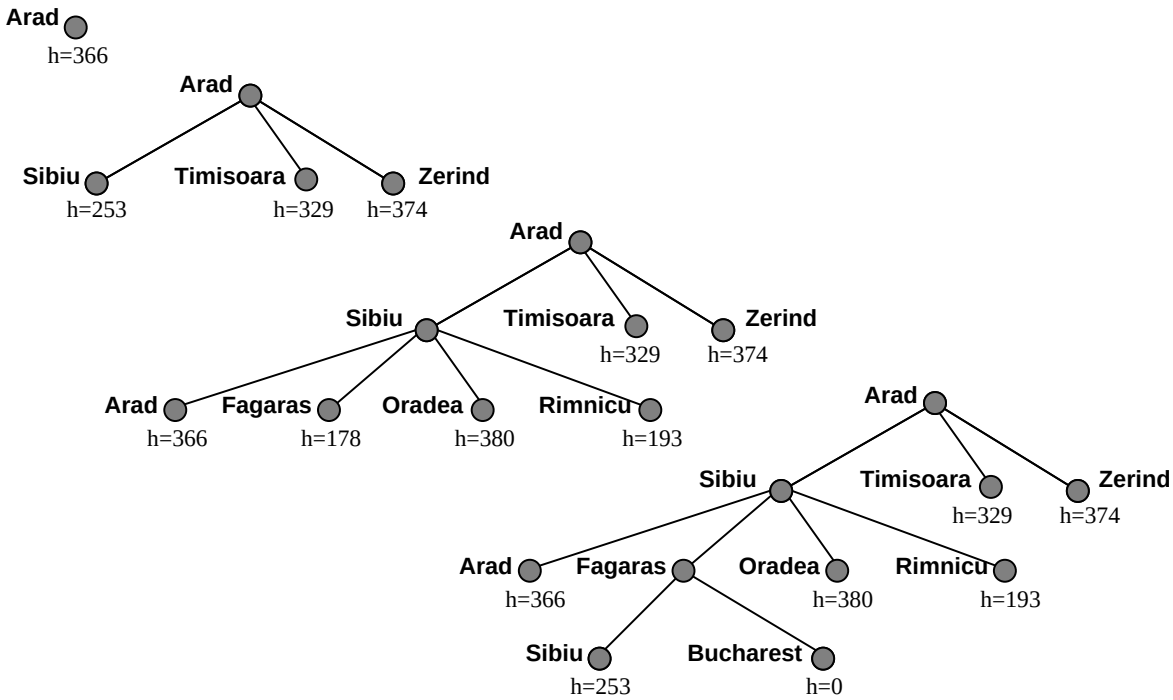
Search methods differ in their strategies which node to expand next

Uninformed search:	fixed strategies without information about the cost from a given node to a goal.
Informed search:	uses information about the cost from a given node to a goal in the form of an evaluation function f, assigning each node a real number.
Best-First Search:	always expand the node with the "best" f-value.
Greedy Search:	$h(n)$ = estimated cost from state at node n to a goal state. Expand node n where $h(n)$ is minimal. Use $f = h$.

Greedy Search Example



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



A*

combines uniform cost search with greedy search.

$g(n)$ = actual cost from the initial state to n.

$h(n)$ = estimated cost from n to the nearest goal.

$f(n) := g(n) + h(n)$.

$f(n)$ = is the estimated cost of the cheapest path which passes through n.

Let $h^*(n)$ be the actual cost of the optimal path from n to the nearest goal.

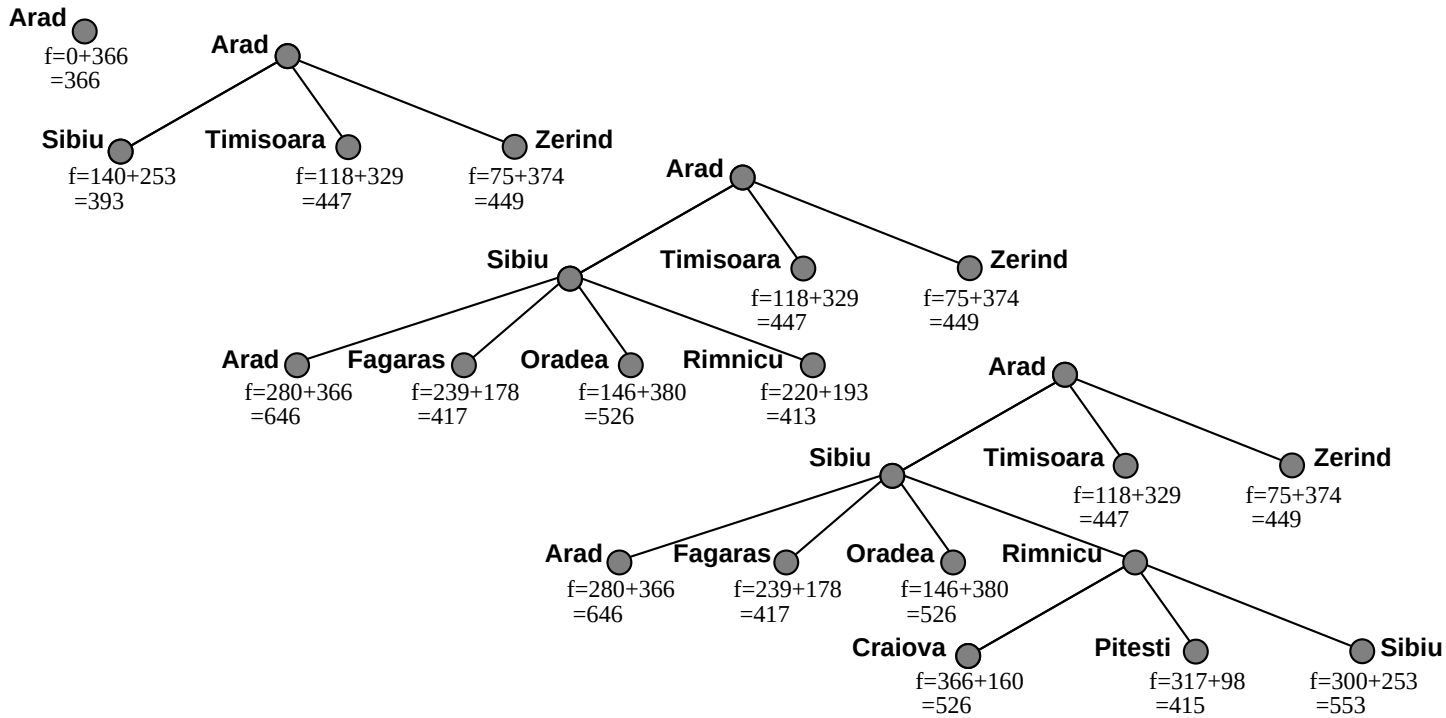
Admissible Heuristic

h is called admissible if we have for all n:

$$h(n) \leq h^*(n) \ .$$

We require for A* that h is admissible.

Example A*



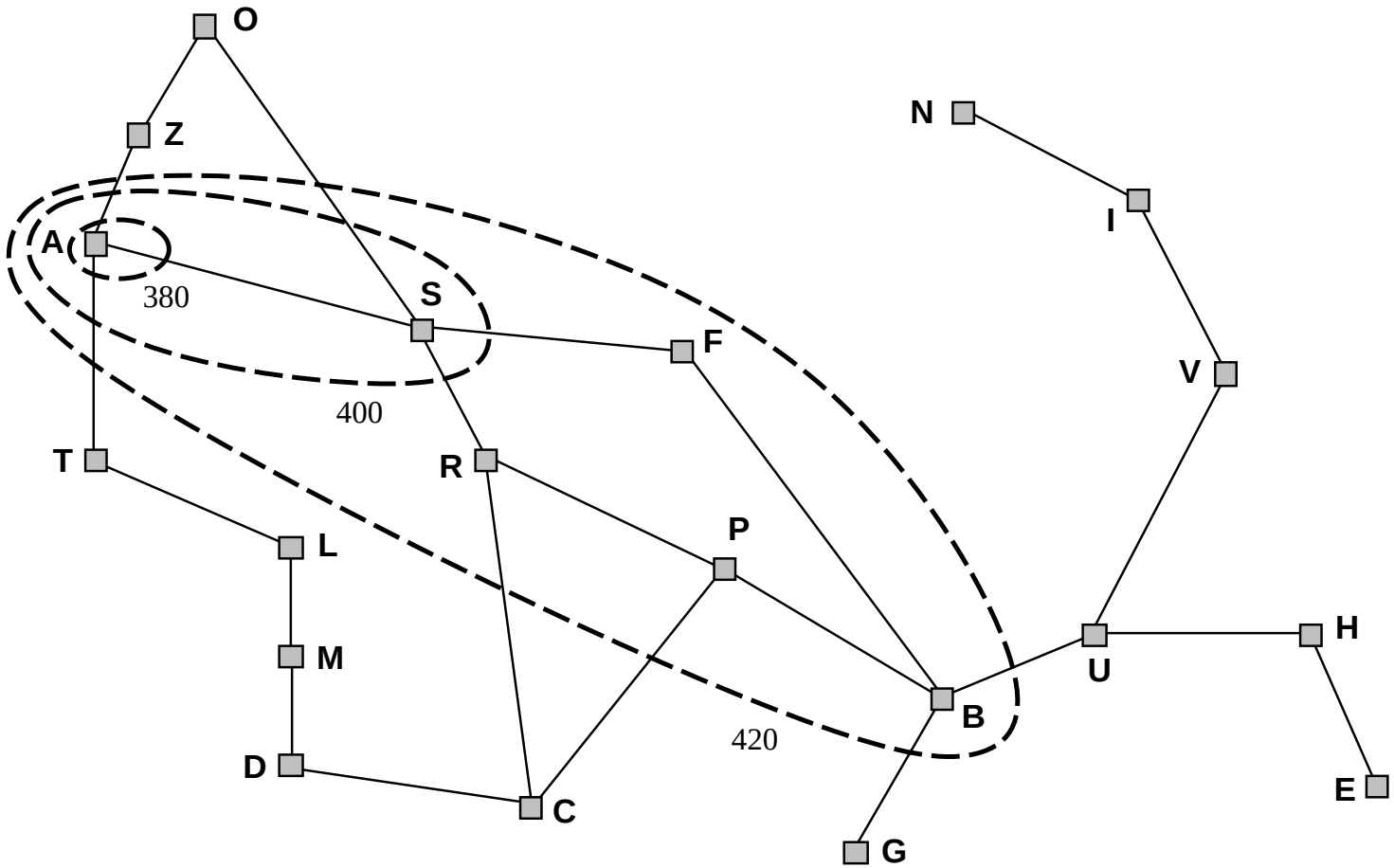
Note: in the example f is monotone nondecreasing.
The following can always be guaranteed:

Path-Max Equation

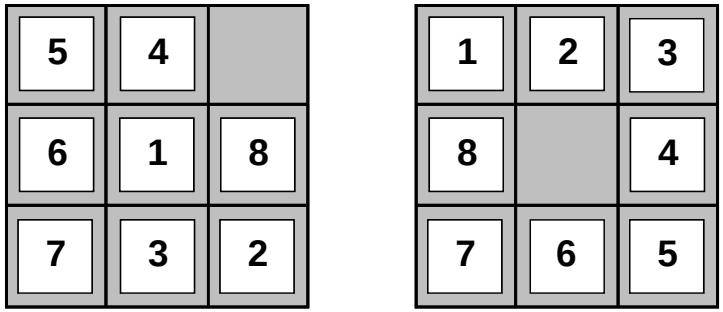
Let n, n' be nodes, where is n parent of n'. Then let

$$f(n') = max(f(n), g(n') + h(n')).$$

Contour Lines in A*



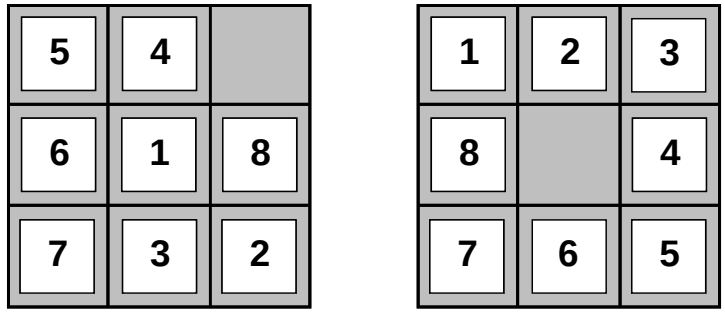
Heuristic Function 1



h_1 = number of tiles in the wrong position.

h_2 = sum of the distances to the goal location for all tiles
(Manhattan Distance)

Heuristic Function 1



h_1 = number of tiles in the wrong position.

h_2 = sum of the distances to the goal location for all tiles
(Manhattan Distance)

Effective branching factor b^*: If A^* generates N nodes with solution depth d, then b^* is the branching factor of a uniform tree of depth d with $N + 1$ nodes, i.e.

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

b^* is a measure for the goodness of h: the closer b^* is to 1 the better.

Heuristic Function 2

	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

How to Find a Heuristic

General Strategy:

- Simplify the problem
- Compute the exact solution for the simplified problem
- Use the solution cost as heuristic

How to Find a Heuristic

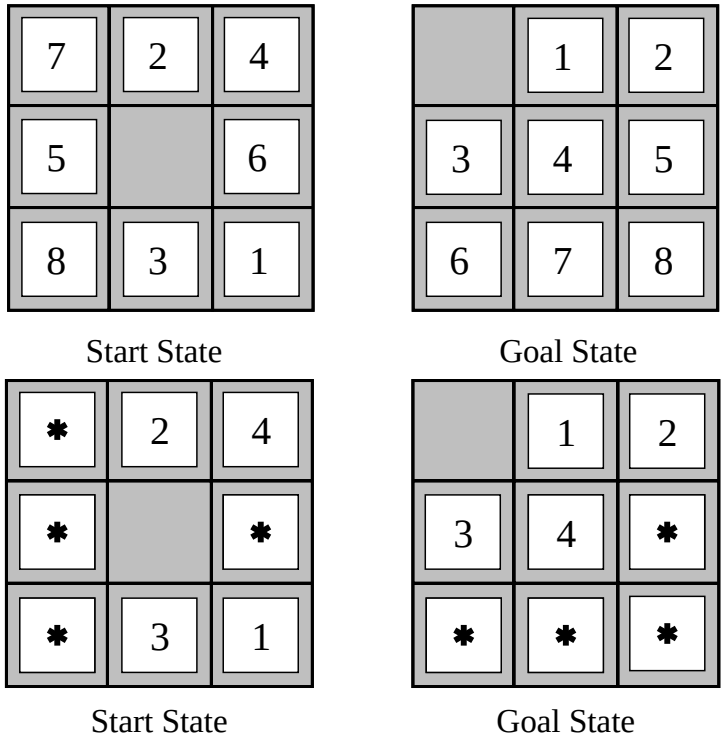
General Strategy:

- Simplify the problem
- Compute the exact solution for the simplified problem
- Use the solution cost as heuristic

For example:

- h_1 is the solution cost for the simplified 8-puzzle where tiles can be placed at an arbitrary position with a single action.
- h_2 corresponds to the exact solution, if tiles can be moved to an arbitrary position but actions are restricted to moving a tile to a neighboring position.

Pattern Databases



Idea: Compute the exact solution for each *pattern* with four numbers and use that value as heuristic. When more than one pattern applies, use the maximum value.
Better than Manhattan!

IDA*

function IDA*(*problem*) **returns** a solution sequence
  inputs: *problem*, a problem
  static: *f-limit*, the current *f*- COST limit
    root, a node

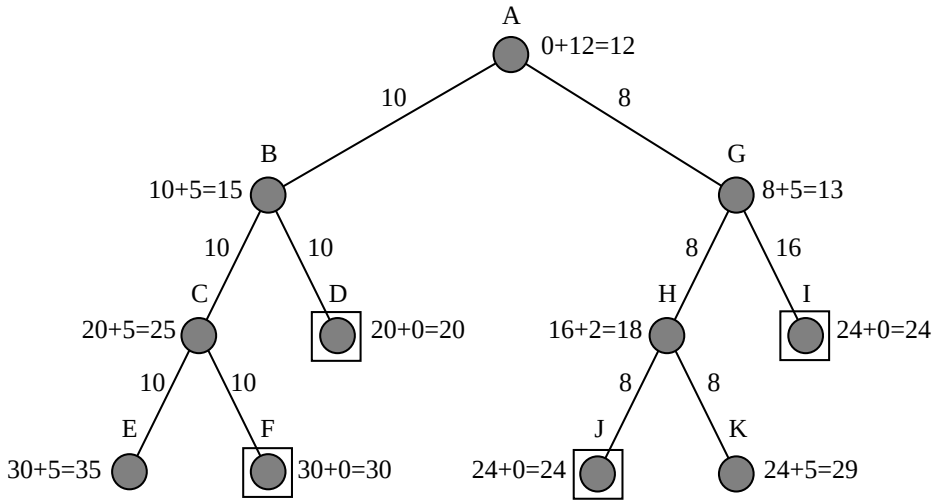
  root ← MAKE-NODE(INITIAL-STATE[*problem*])
  f-limit ← *f*- COST(*root*)
  loop do
    solution, *f-limit* ← DFS-CONTOUR(*root*, *f-limit*)
    if *solution* is non-null **then return** *solution*
    if *f-limit* = ∞ **then return** failure; **end**

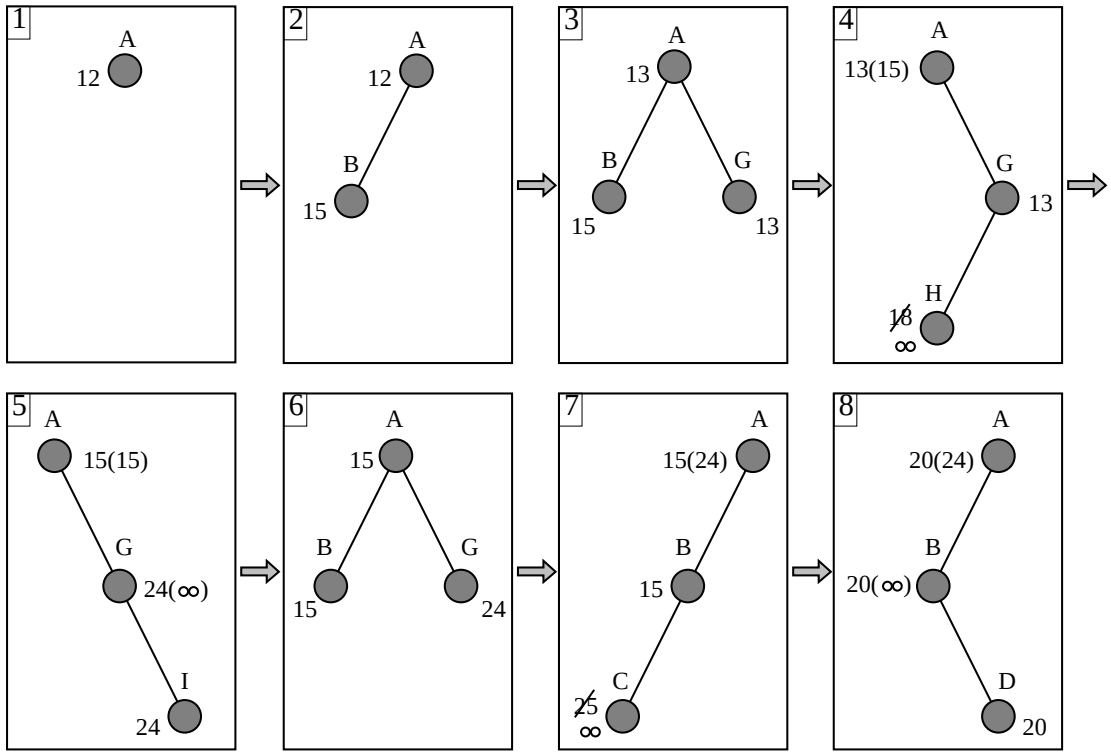
---

function DFS-CONTOUR(*node*, *f-limit*) **returns** a solution sequence and a new *f*- COST limit
  inputs: *node*, a node
    f-limit, the current *f*- COST limit
  static: *next-f*, the *f*- COST limit for the next contour, initially ∞

  if *f*- COST[*node*] > *f-limit* **then return** null, *f*- COST[*node*]
  if GOAL-TEST[*problem*](STATE[node]) **then return** *node*, *f-limit*
  for each node *s* **in** SUCCESSORS(*node*) **do**
    solution, *new-f* ← DFS-CONTOUR(*s*, *f-limit*)
    if *solution* is non-null **then return** *solution*, *f-limit*
    next-f ← MIN(*next-f*, *new-f*); **end**
  return null, *next-f*

SMA* 1



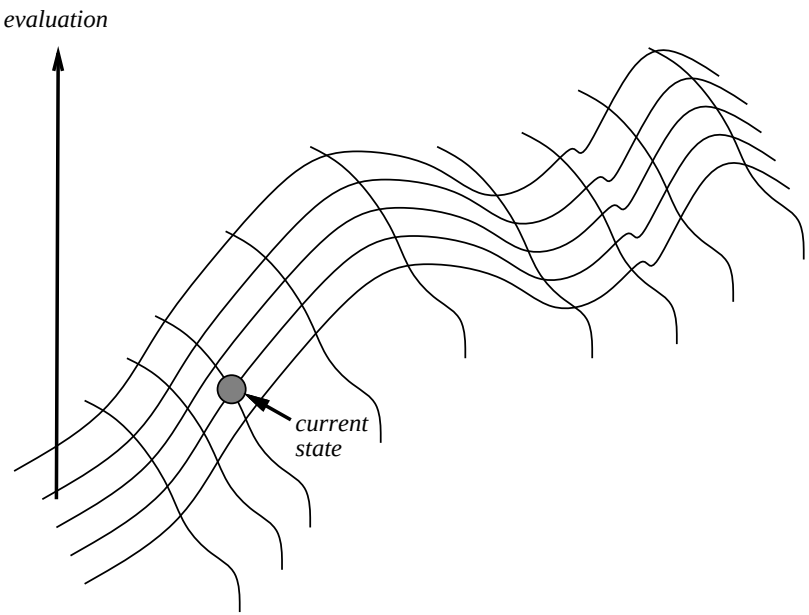


SMA* 2

```
function SMA*(problem) returns a solution sequence
    inputs: problem, a problem
    static: Queue, a queue of nodes ordered by f-cost

    Queue ← MAKE-QUEUE({MAKE-NODE(INITIAL-STATE[problem])})
    loop do
        if Queue is empty then return failure
        n ← deepest least-f-cost node in Queue
        if GOAL-TEST(n) then return success
        s ← NEXT-SUCCESSOR(n)
        if s is not a goal and is at maximum depth then
            f(s) ← ∞
        else
            f(s) ← MAX(f(n), g(s)+h(s))
        if all of n's successors have been generated then
            update n's f-cost and those of its ancestors if necessary
        if SUCCESSORS(n) all in memory then remove n from Queue
        if memory is full then
            delete shallowest, highest-f-cost node in Queue
            remove it from its parent's successor list
            insert its parent on Queue if necessary
        insert s on Queue
    end
```

Hill Climbing



function HILL-CLIMBING(*problem*) **returns** a solution state
  inputs: *problem*, a problem
  static: *current*, a node
    next, a node

  current ← MAKE-NODE(INITIAL-STATE[*problem*])
  loop do
    next ← a highest-valued successor of *current*
    if VALUE[next] < VALUE[current] **then return** *current*
    current ← *next*
  end

Simulated Annealing

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 $e^{\Delta E/T}$

Ridges