Informed Search Method

Use information about the state space to prevent algorithms form blundering about in the dark.

Best-First Search

Uses the queuing function to order the nodes in the queue according to an **evaluation function,** and choose to expand a node that seems to have the best **desirability**.

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*)

A common aspect of Best-First Search strategies is that, *in order to focus the search*, they use some ***estimated measure*** *which incorporates some estimate of the cost of the path from a state to the closest goal*

Minimize estimated cost to reach a goal : Greedy search

The node whose state is judged to be the closest to the goal state is always expanded first.

✧The strategy is to minimize the **remaining path cost**, and is called the **greedy search**.

A **heuristic function** is used to calculate the path cost:

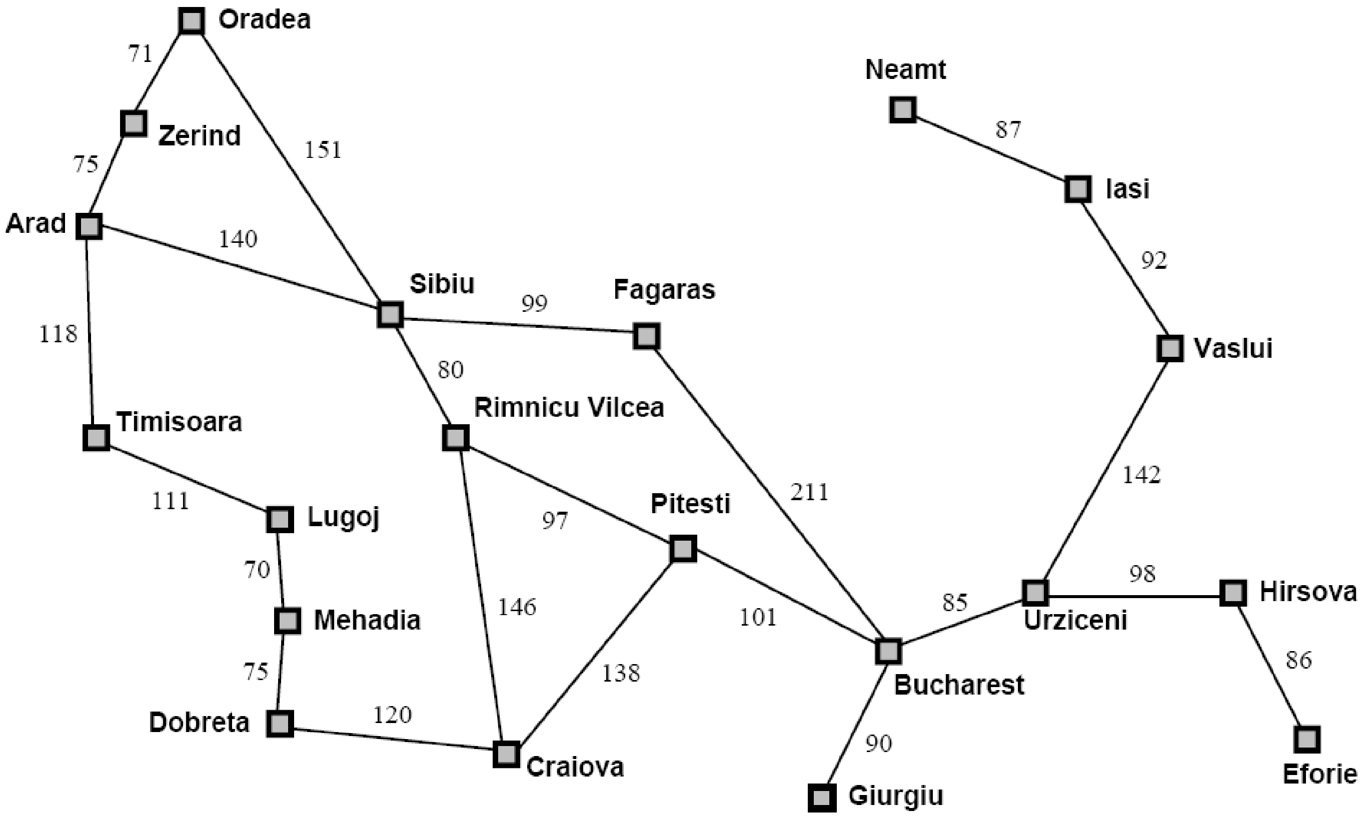
h(n) = estimated cost of the cheapest path from the state at node ***n*** to ***a goal state***.

h(n) must be **0** for goal states.

✧The code is :

function GREEDY-SEARCH(*problem*)
returns a solution or failure.
return BEST-FIRST-SEARCH(*problem*, *h*)

✧An example :

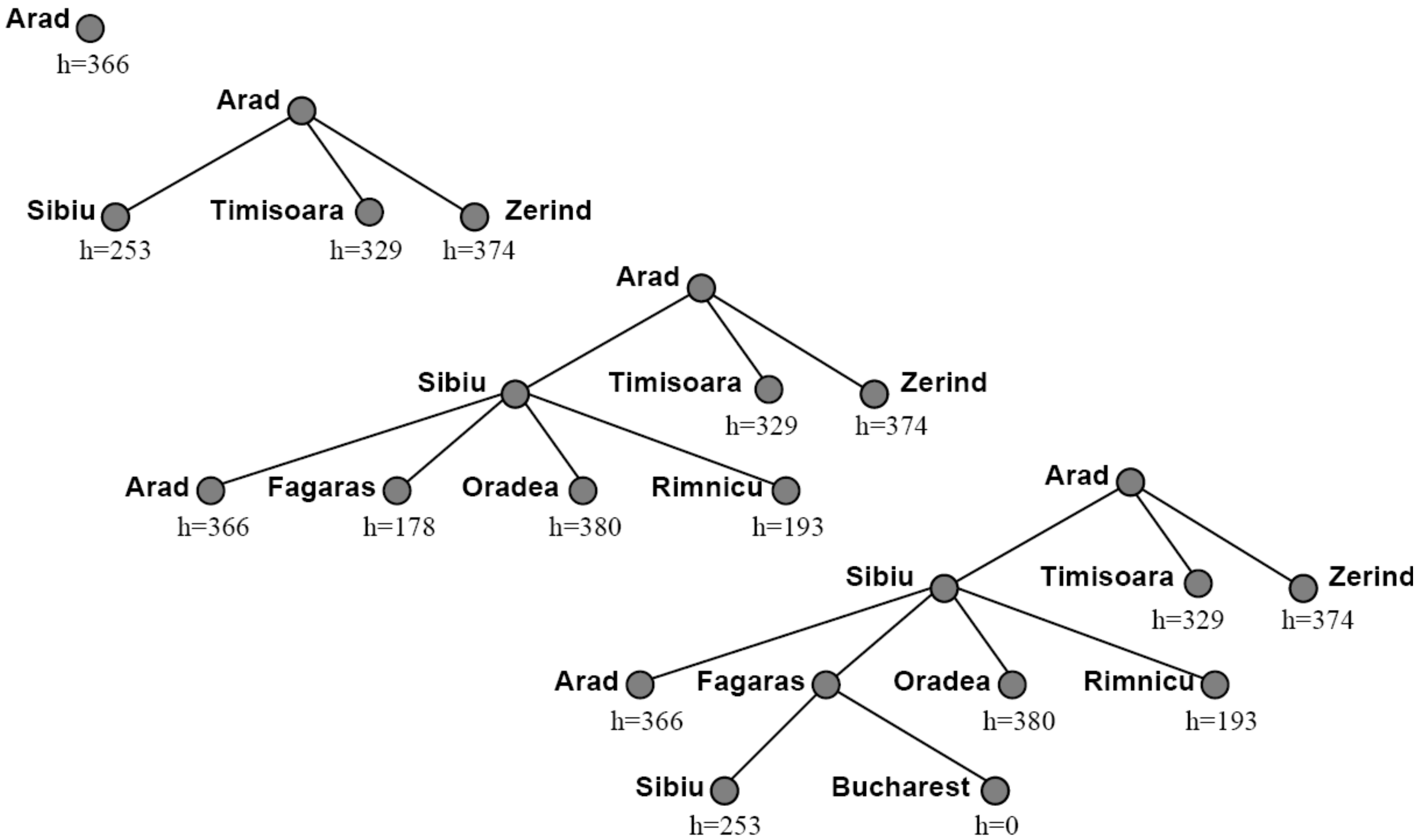


Straight−line distance to Bucharest

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

Define the heuristic function as:

h**$_{SLD}$**(n) = straight-line distance between ***n*** and the ***goal*** location.



Arad
h=366
Arad
Sibiu
Timisoara
Zerind
h=253
h=329
h=374
Arad
Sibiu
Timisoara
Zerind
h=329
h=374
Arad
Fagaras
Oradea
Rimnicu
h=366
h=178
h=380
h=193
Arad
Sibiu
Timisoara
Zerind
h=329
h=374
Arad
Fagaras
Oradea
Rimnicu
h=366
h=380
h=193
Sibiu
Bucharest
h=253
h=0

✧Incomplete and not optimal:

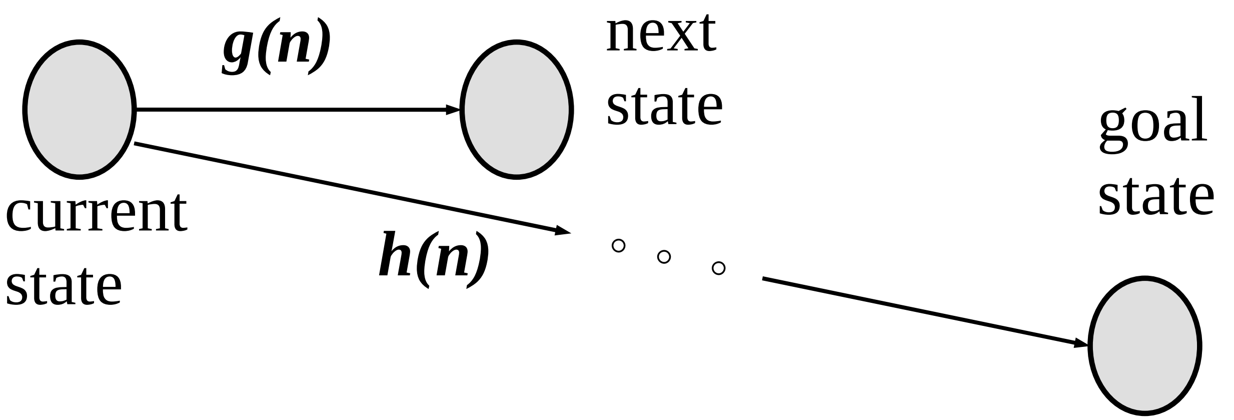
1. greedy approaches may be **quick**. It may have the minimum search cost because of using *immediate best choice* instead of *long term options*.
2. greedy approaches may **not be optimal**. The solution found may not be the best one. (Arad-Rimnicu-Vilcea-Pitesti-Bucharest is the best, for example.)
3. greedy approaches may **need to backtrack** (consider Iasi to Fagaras).

✧Problem: holds all nodes in memory

✧ **time complexity == space complexity ==** $O(b^m)$**.**

Minimize the total path cost : A* search

Combines the Greedy search (quick) and Uniform-cost search (complete and optimal) to gain the benefits.



✧The strategy is to minimize the **total path cost**.

An **evaluation function** ***f(n)*** is used to estimate the total path cost :

f(n) **= estimated cost of the cheapest solution through *n***
= *g(n)* + *h(n)* ***where***

g(n) = (known) cost of getting to ***n***
h(n) = estimated cheapest cost to get from ***n*** to ***goal***

In A* search, ***h*** must be restricted to **admissible heuristic**, which is *inherently optimistic*, and *never overestimates the cost to reach the goal*.

The optimism is transferred to the ***f*** function : *if* ***h*** *is admissible,* ***f(n)*** *never overestimates the actual cost of the best solution through n.*

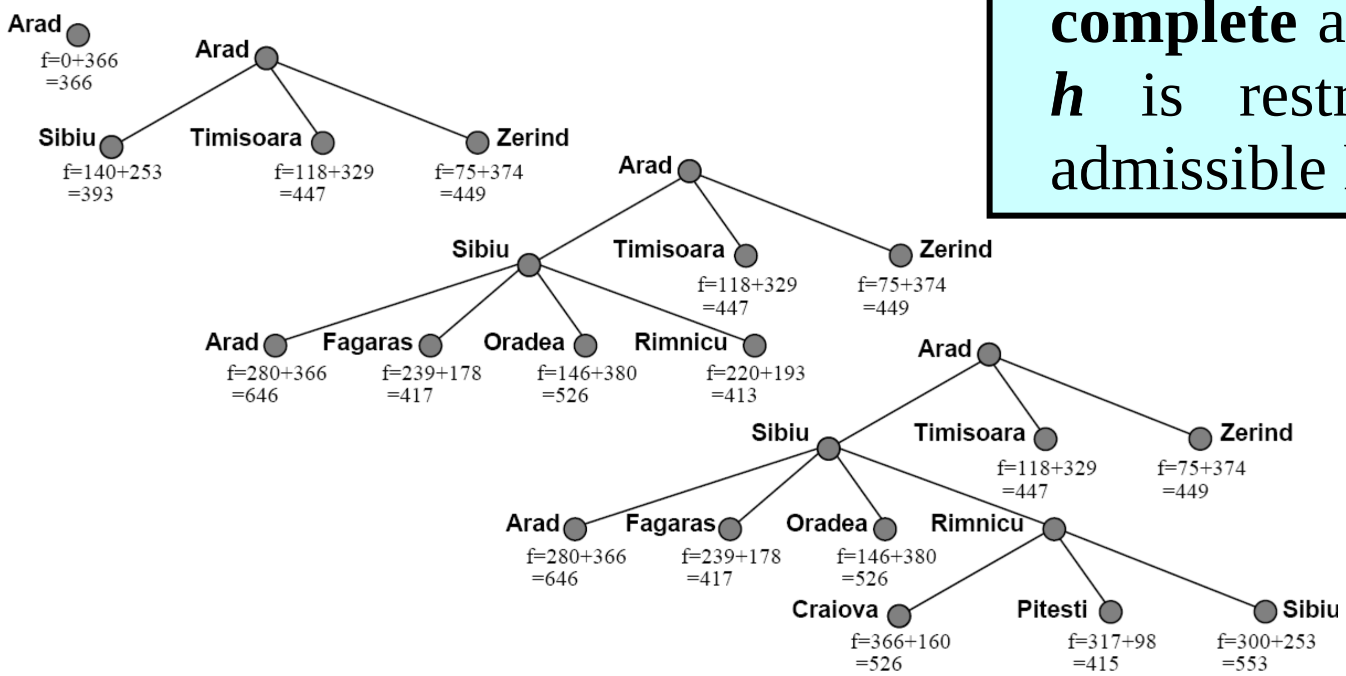
✧The code is :

function A* SEARCH(*problem*) **returns** a solution or failure.
return BEST-FIRST-SEARCH(*problem*, *g*+*h*)

✧An example :

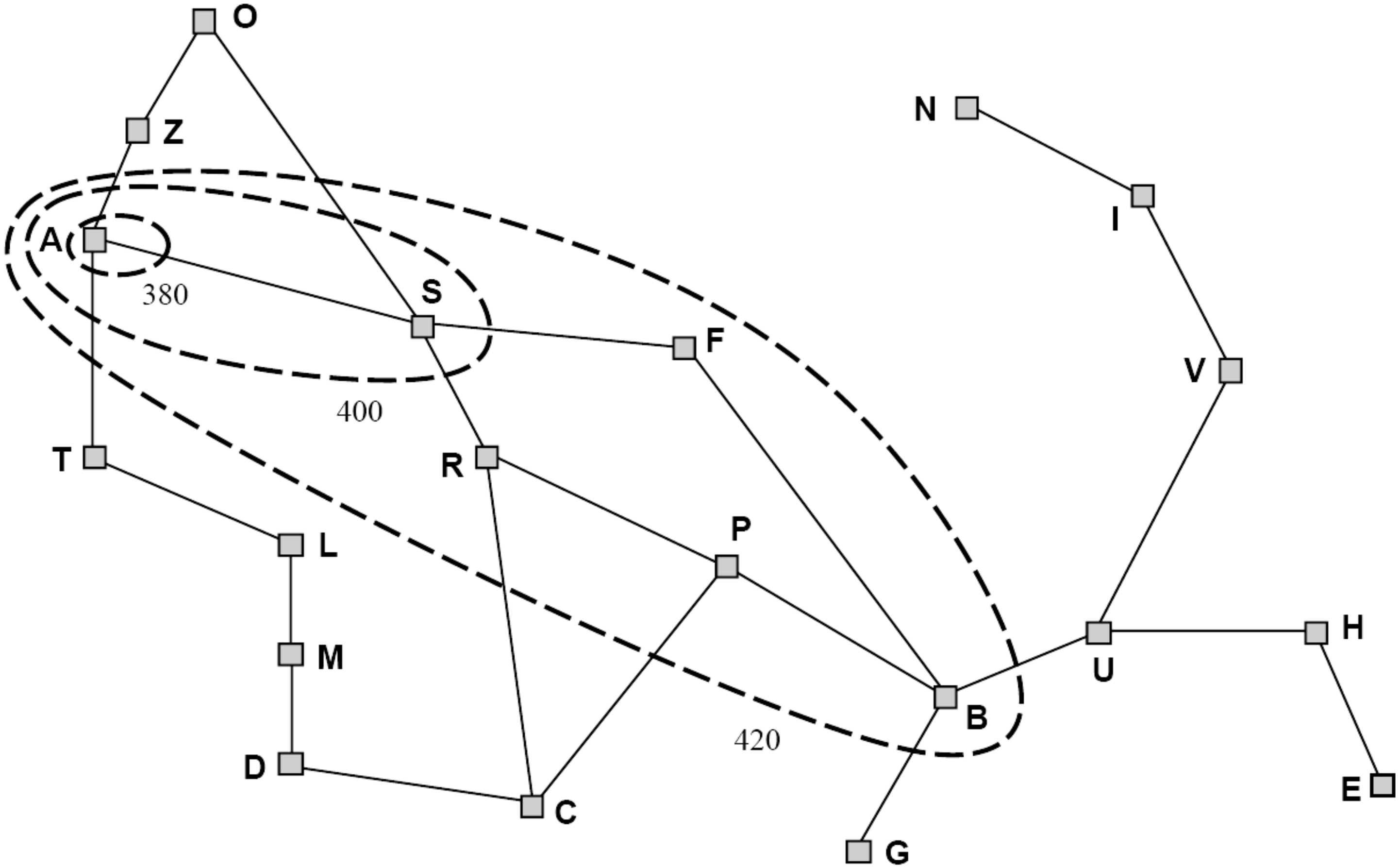
✧**Complete and Optimal**:

A* search approaches is **complete** and **optimal** if ***h*** is restricted to an admissible heuristic.



※Conceptually prove A* is optimal and complete

- If ***f*** never decreases along any path out from root, conceptual **contours** can be drawn.



- IF **f*** is cost of optimal solution THEN:

  A* expands all nodes **f(n) < f***
  A* expands some nodes **f(n) = f***
  A* expands no nodes **f(n) > f***

- A* search is **optimally efficient** for any given heuristic function, i.e. no other optimal algorithm is guaranteed to expand fewer nodes than A*

※**Time complexity *O(b^m)*** (bad news)

- Time has exponential growth unless **error** in heuristic function grows no faster than the *log* of the **actual path cost**:

$$| h(n) - h^*(n) | \le O(\log h^*(n))$$

- For almost all heuristics in practical use, the error is at least proportional to path cost, and therefore results the exponential growth in the time complexity.

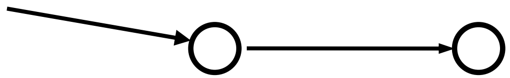
※Space complexity $O(b^m)$

Because A* search keeps all generated nodes in memory, empirically, it runs out of space before time.

※The behavior of A* search

- If ***h*** is monotonically increasing, i.e. if it exhibits **monotonicity**, ***f*-cost** will never decrease along any path.
- Almost all **admissible heuristics** are **monotonic**.
- When ***f*** has a non-monotonically increasing ***h(n)***, it can be fixed by using a **pathmax** equation:

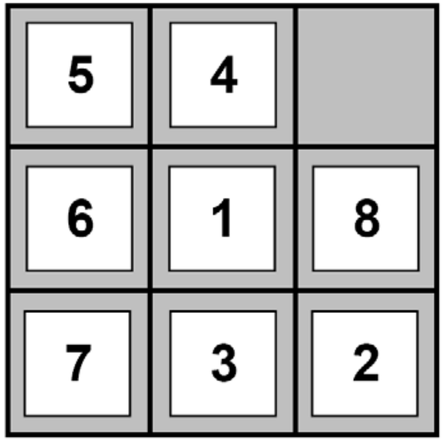
f(n') = max(f(parent(n')), g(n') + h(n'))



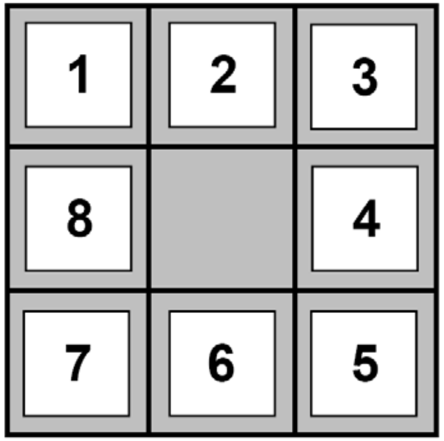
maybe f(n) >= f(n')

Heuristic Functions

Using the 8-puzzle problem as an example to show how a heuristic function can be established.



Start State



Goal State

✧Typical solution is **20** steps.

✧Average branching factor is **3**.

✧Blind search would look at $\mathbf{3^{20} = 3.5\times10^{9}}$ states.

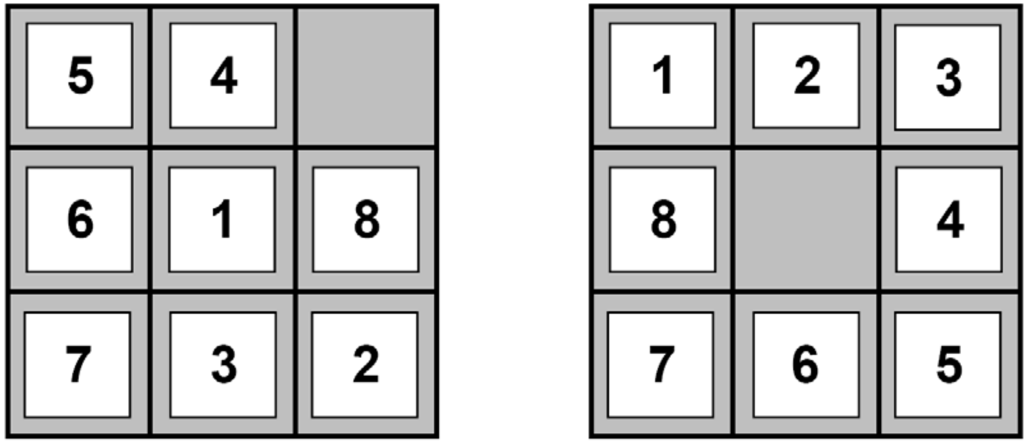
✧By checking repeated states would reduce to still **9!=362,880**, Not to mention the memory needed.

✧Needs an admissible heuristic function to find the shortest solutions. Two candidates:

1. **h_1** = the number of tiles that are in the wrong position.
2. **h_2** = the sum of the distances of the tiles from their goal position. This must be the **city block distance** or **Manhattan distances**, which is the sum of the *horizontal* and *vertical* distances.

✧For figure 7, the ***h*** values for the start states are:

$$\mathbf{h_1} = 8 \quad \mathbf{h_2} = 2+3+3+2+4+2+0+2$$



Start State **Goal State**

The effect of heuristic accuracy on performance

The quality of a heuristic can be judged by the **effective branching factor b***

b*: the **branching factor** for an uniform tree of depth **d** to contain **N** nodes -- the total number of nodes expanded by A*.
$\mathbf{N = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d}$.

✧A well designed heuristic would have a value of b* close to **1**.

✧Compare the result

	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

Figure 8: Comparison of the search costs and effective branching factors for the ITERATIVE-DEPEENING-SEARCH and A* algorithms with h_1,h_2. Data are averaged over 100 instances of the 8-puzzle, for various solution length.

✧Will h2 be always better than h1? Yes

If for any node $\boldsymbol{n}$, $\boldsymbol{h_2(n) \geq h_1(n)}$, then $\boldsymbol{h_2}$ **dominate** $\boldsymbol{h_1}$, and A* uses $\boldsymbol{h_2}$ will expand *fewer* nodes.

Every node $\boldsymbol{n}$ is expanded if

$$\boldsymbol{f(n) \leq f^*} \quad \text{i.e.} \quad \boldsymbol{h(n) < f^* - g(n)}$$

This also means that every node expanded by $\boldsymbol{h_2}$ will be expanded by $\boldsymbol{h_1}$ too.

✧It is always better to use a heuristic function with higher values, as long as it does not overestimate.

◆ Inventing heuristic functions

※**Relaxed problem**: A problem with fewer restrictions on the operators is called a **relaxed problem**.

The cost of an exact solution to a relaxed problem is sometimes a good heuristic for the original problem.

- For example, the 8-puzzle operators:

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

can be reduced to :

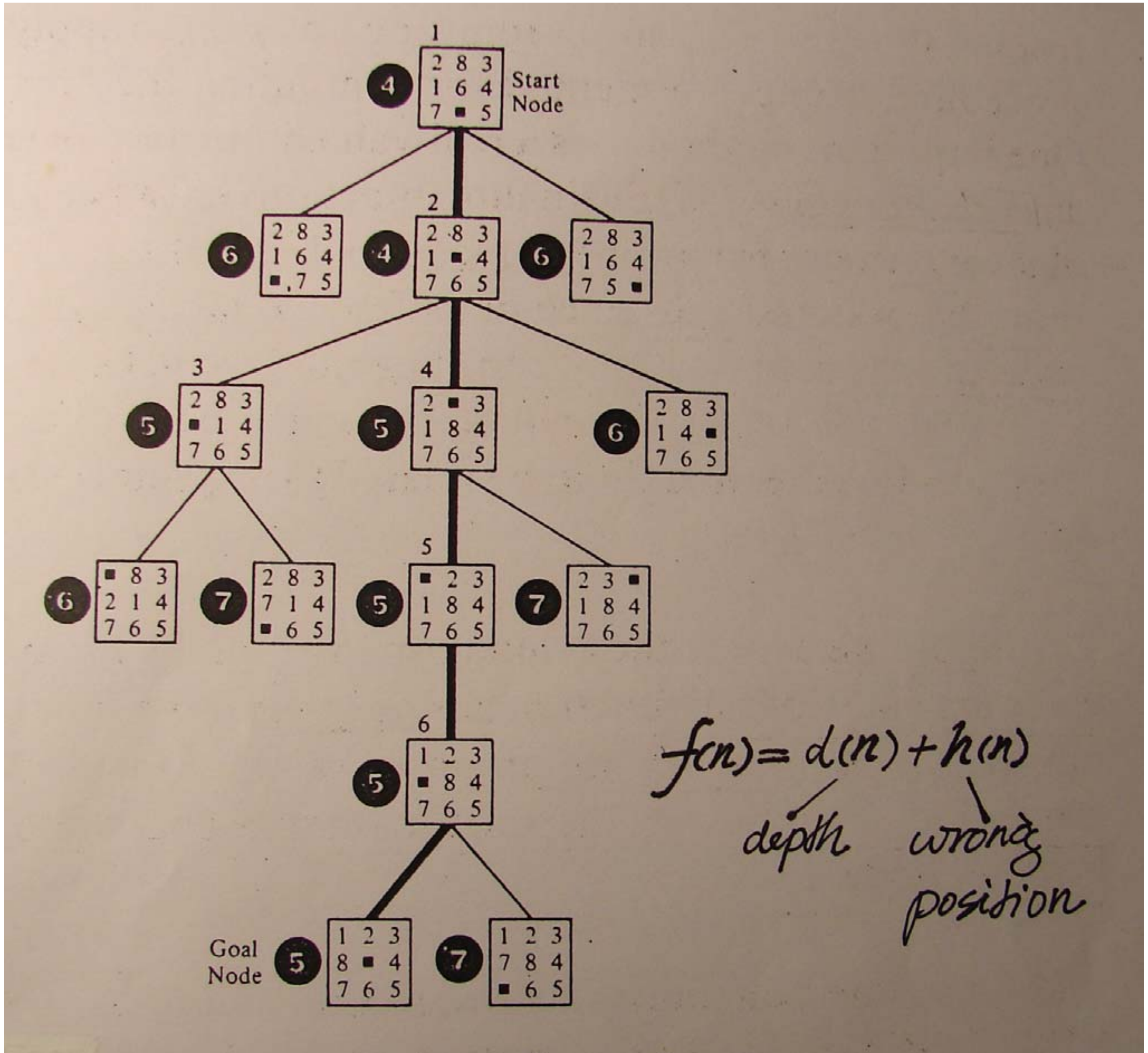
(a) A tile can move from square A to square B if A is adjacent to B.
(b) A tile can move from square A to square B if B is blank.
(c) A tile can move from square A to square B.

※May use maximum of a set of heuristics:

If a set of admissible heuristics $h_1, h_2, ..., h_m$ is available for a problem, and none of them dominates any of others, then just use:

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

h is **admissible**, and **dominate** all of the individual heuristics.



A search tree using an evaluation function.

Memory Bounded Search

Available memory space is the first thing to consider when facing complex problem.

Iterative deepening A* search (IDA*)

Each iteration is a **Depth-first search** that expands all nodes of cost below some ***f***-cost limit.

✧Each iteration expands all nodes inside the **contour** for the current ***f***-cost, generating a ***new f***-cost for the next iteration.

✧**Complete:** Yes

✧**Optimal:** Yes

✧**Space: *O*(branching factor*depth)**

✧**Time:** similar to A*, but depends on the number of different values the heuristic function can take.

1. Typically ***f*** value only increases **two** or **three** times along any solution path -- thus, IDA* only goes through **two** or **three** iterations
2. Less overhead than A* because no priority queue.

```
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
```

✧Problems in more complex domains:

1. When each contour expands very few (maybe just 1) state.

Example domain: In the Traveling Salesman problem, the heuristic value is different for every state. Thus, each iteration will expand exactly **one more** state**.**

If A* expands **N** nodes, IDA* will expand

1+2+......+N = *O*(N^2) nodes**.**

A possible solution: ε-admissible algorithm Increase the ***f***-cost limit by at least some **ε** on each iteration.
This, however, finds solutions that can be worse than the true optimal solution by at most **ε**.

2. No memory between searches: may repeat searching from an already seen node (solution: see SMA*).

Simplified Memory Bounded A* (SMA*)

Try to use **any available memory** to carry out the search, especially to avoid repeated states.

- ✧SMA* avoids repeated states if record of them fits in memory.
- ✧In generating a successor, if there is no memory left, the node with the high ***f***-cost in the queue is dropped.
- ✧The quality of the best path in a forgotten sub-tree is remembered in a node. In case of regenerating, only the path is generated.
- ✧Simplified SMA* algorithm is in ***Figure 12.***

- ✧**Complete:** Yes, if the available memory can hold the shallowest solution path.
- ✧**Optimal:** Yes, if the available memory can hold the shallowest optimal solution path.
- ✧Search is **optimally efficient** for the entire search tree that fits in the available memory.

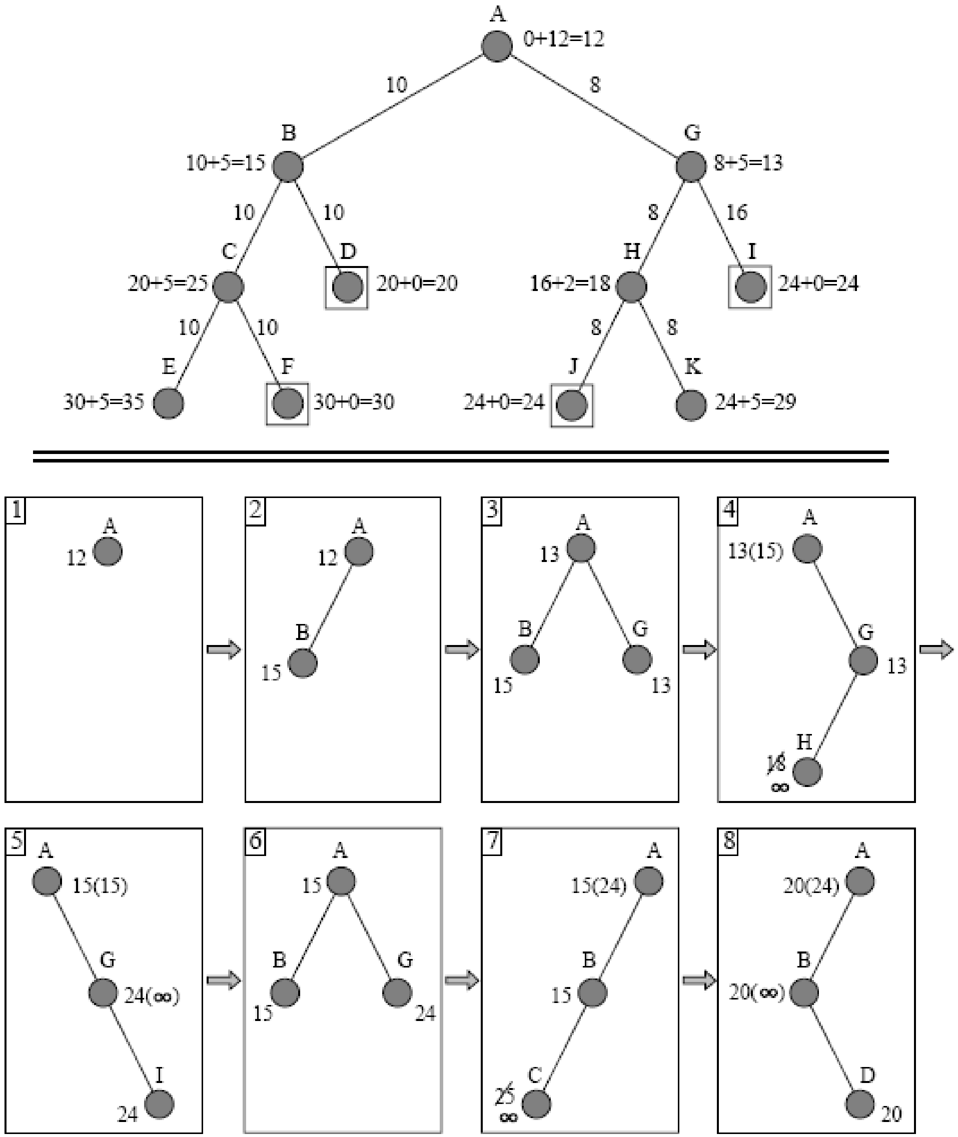


Figure 11: Progress of an SMA* search with a memory size of three nodes, on the state space shown at the top. Each node is labelled with its *current* f-cost. Values in parentheses show the value of the best forgotten descendant.

```
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
```

Figure 12: Sketch of the SMA* algorithm. Note that numerous details have been omitted in the interests of clarity.

Iterative Improvement Algorithms

For problems in which the *state description itself contains all the information needed for a solution*, the iterative improvement algorithms provide the most practical approach.

- ✧The path by which the solution is researched is irrelevant.
- ✧The general idea is to start with a complete configuration and to make modifications to improve its quality.
- ✧Consider to lay out all the states on the surface of a landscape as in (***Figure 13***)
- ✧The idea is to move around the landscape trying to find the highest peeks, which are the optimal solution.
- ✧Iterative improvement algorithms usually keep track of only the current state.

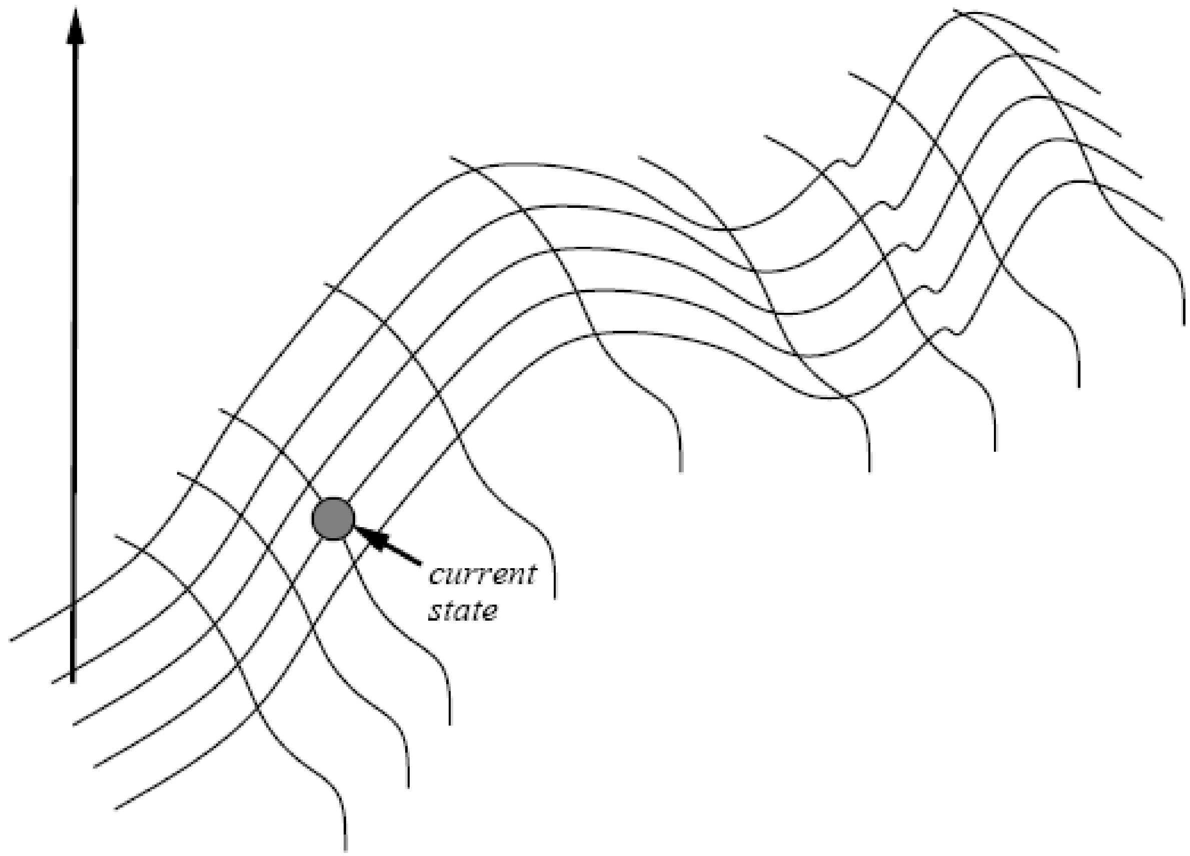


Figure 13: Iterative improvement algorithms try to find peaks on a surface of states where height is defined by the evaluation function.

Hill-climbing (or gradient descent) search

Always moves in the direction of increasing value. The algorithm does not maintain a search tree.

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

✧Problems:

1. **local maxima**: if the algorithm is trapped on a local maxima, it will halt even though the solution is far from satisfactory.
2. **Plateaux**: If the algorithm hits an area that is basically flat, it will do random walk.
3. **Ridge**: (suppose the peak is on the same ridge) there may not be an operator that allows one to change to an adjacent higher state.

✧**Random-restart hill-climbing**, which starts a series of hill-climbing searches from randomly generated initial state, may solve the problem and find the optimal solution given enough iterations.

Simulated annealing

Jump randomly and allow to take some **downhill** steps (to escape the local maximum) with decreasing frequency (probability).

```
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 ← 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)
```

✧The probability to move to a worse state decreases exponentially with the "**badness**" of the move--the amount **ΔE** by which the evaluation is worsened.

✧The probability is also controlled by the **T**, the "**temperature**", which is determined by a ***schedule*** as a function of how many cycles already have been completed.

✧Higher T, higher probability to make "**bad**" moves.

✧If the schedule lowers T slowly enough, the algorithm will find a global optimum.