Knowledge Representation & Reasoning
COMP9016

Dr Ruairí O'Reilly
ruairi.oreilly@cit.ie

Informed Search Strategies & Beyond Classical Search

CIT CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Informed (heuristic) search strategies

>> **Informed search strategy -** one that uses problem-specific knowledge beyond the definition of the problem itself.

>> General approach considered: **best-first search**

> Instance of the general TREE-SEARCH or GRAPH-SEARCH.

> Expansion based on an evaluation function, *f(n).*

> Implementation of best-first graph search identical to that for uniform-cost search

> Use *f* instead of *g* to order the priority queue

>> Most best-first algorithms include as a component of *f* a heuristic function, denoted *h(n)*:

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

CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Greedy best-first search

- **Greedy best-first search** tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function; that is, *f(n) = h(n).*

- *Let us see how this works for route-finding problems in Romania,* we use the straight-line distance heuristic, which we will call *hSLD*. If the goal is Bucharest, we need to know the straight-line distances to Bucharest

- For example, *hSLD(In(Arad))=366*. Notice that the values of *hSLD* cannot be computed from the problem description itself. Moreover, it takes a certain amount of experience to know that *hSLD* is correlated with actual road distances and is, therefore, a useful heuristic.

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Values of h_{SLD}—straight-line distances to Bucharest.

CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Greedy best-first search

- *hSLD* finding a path from Arad to Bucharest.
- Is it optimal? Why is it called “Greedy”?

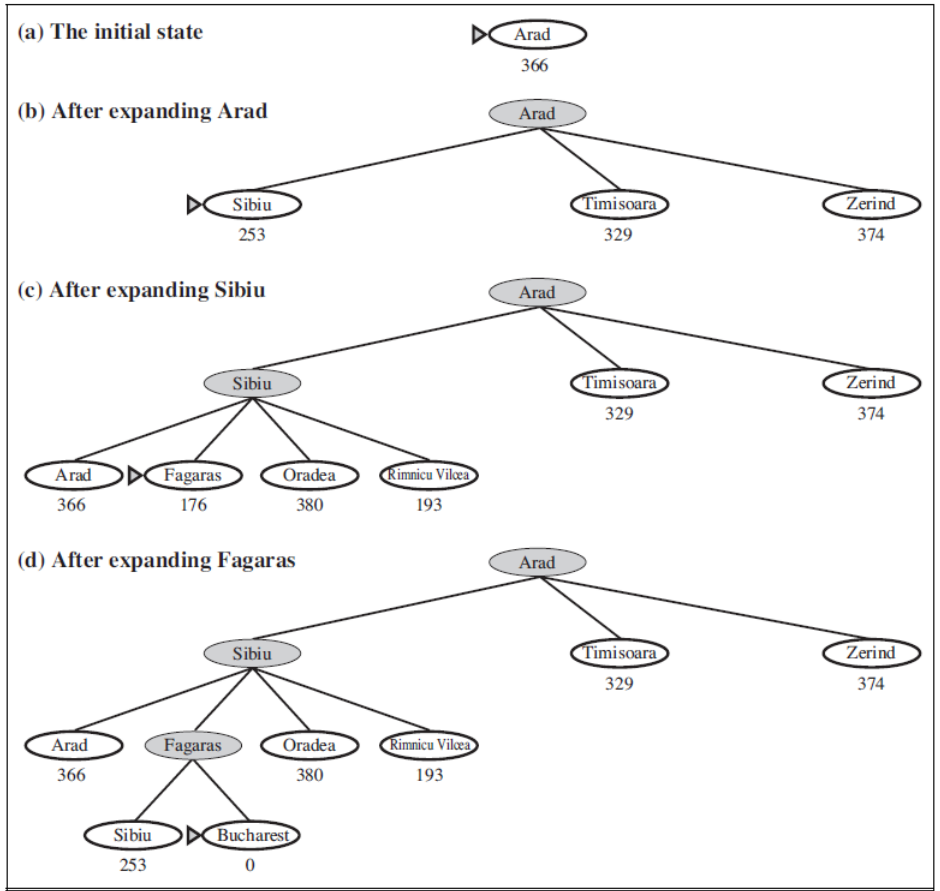


Figure 3.23 Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

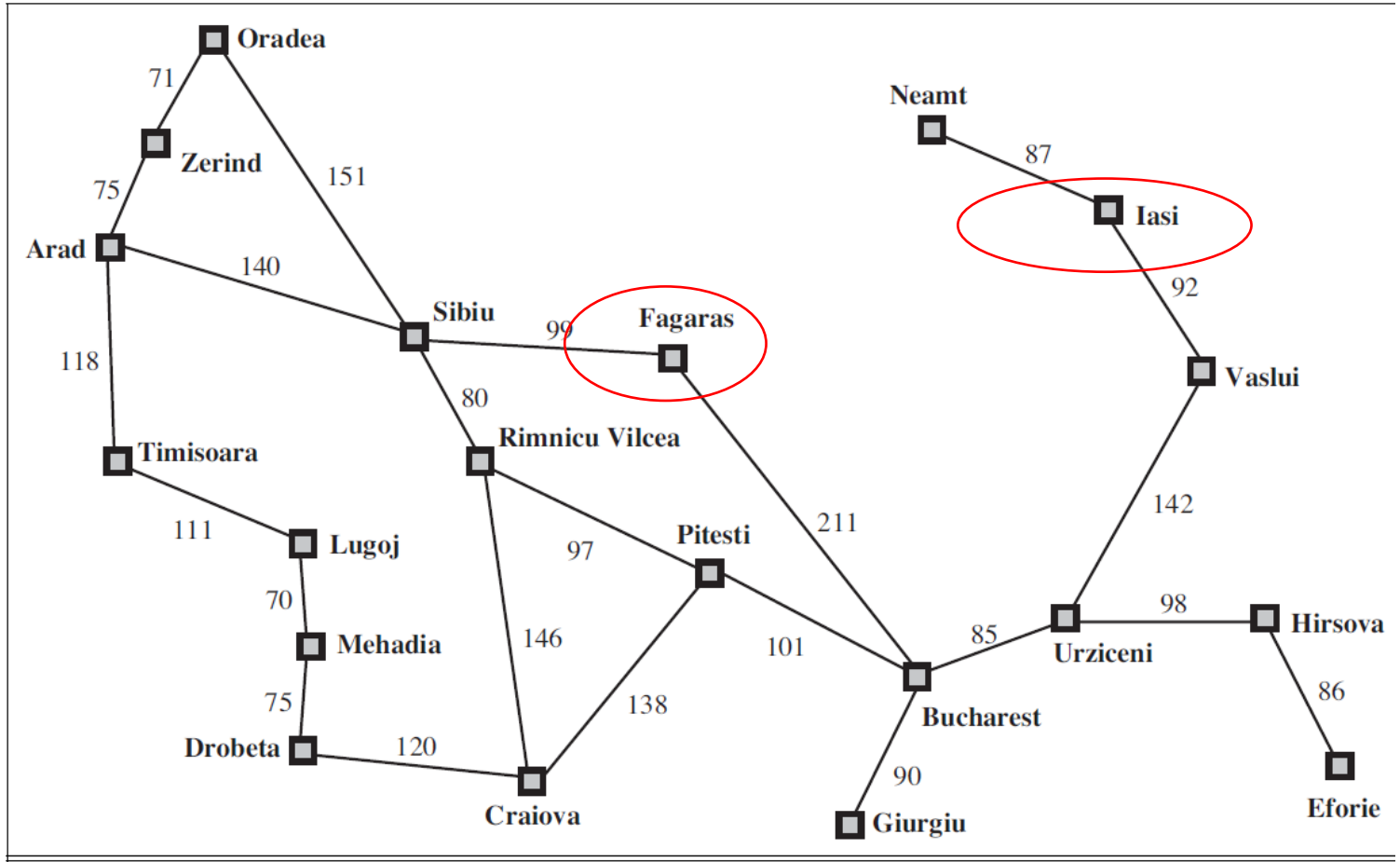


Figure 3.2 A simplified road map of part of Romania.

Example – Getting from Iasi to Fagaras

Time and space complexity for the tree version is O(b^m)

CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

A* search: Minimizing total estimated solution cost

>> The most widely known form of best-first search is called A∗ search. It evaluates nodes by combining *g(n)*, the cost to reach the node, and *h(n)*, the cost to get from the node to the goal:

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

>> Since *g(n)* gives the path cost from the start node to node *n*, and *h(n)* is the estimated cost of the cheapest path from n to the goal, we have

f(n) = estimated cost of the cheapest solution through n .

>> How to find the cheapest solution? a reasonable thing to try first is the node with the lowest value of *g(n) + h(n)*. *A∗ search is both complete and optimal.*

>> The algorithm is identical to UNIFORM-COST-SEARCH except that A∗ uses *g + h* instead of *g*.

CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Admissibility

>> Condition 1: *h(n)* be an **admissible heuristic –** one that never overestimates the cost to reach the goal. *f(n)=g(n) + h(n)*

>> Admissible heuristics by their nature are **"optimistic"** *e.g. hSLD (see Fig)*

>> *The progress of an A∗ tree search for Bucharest*

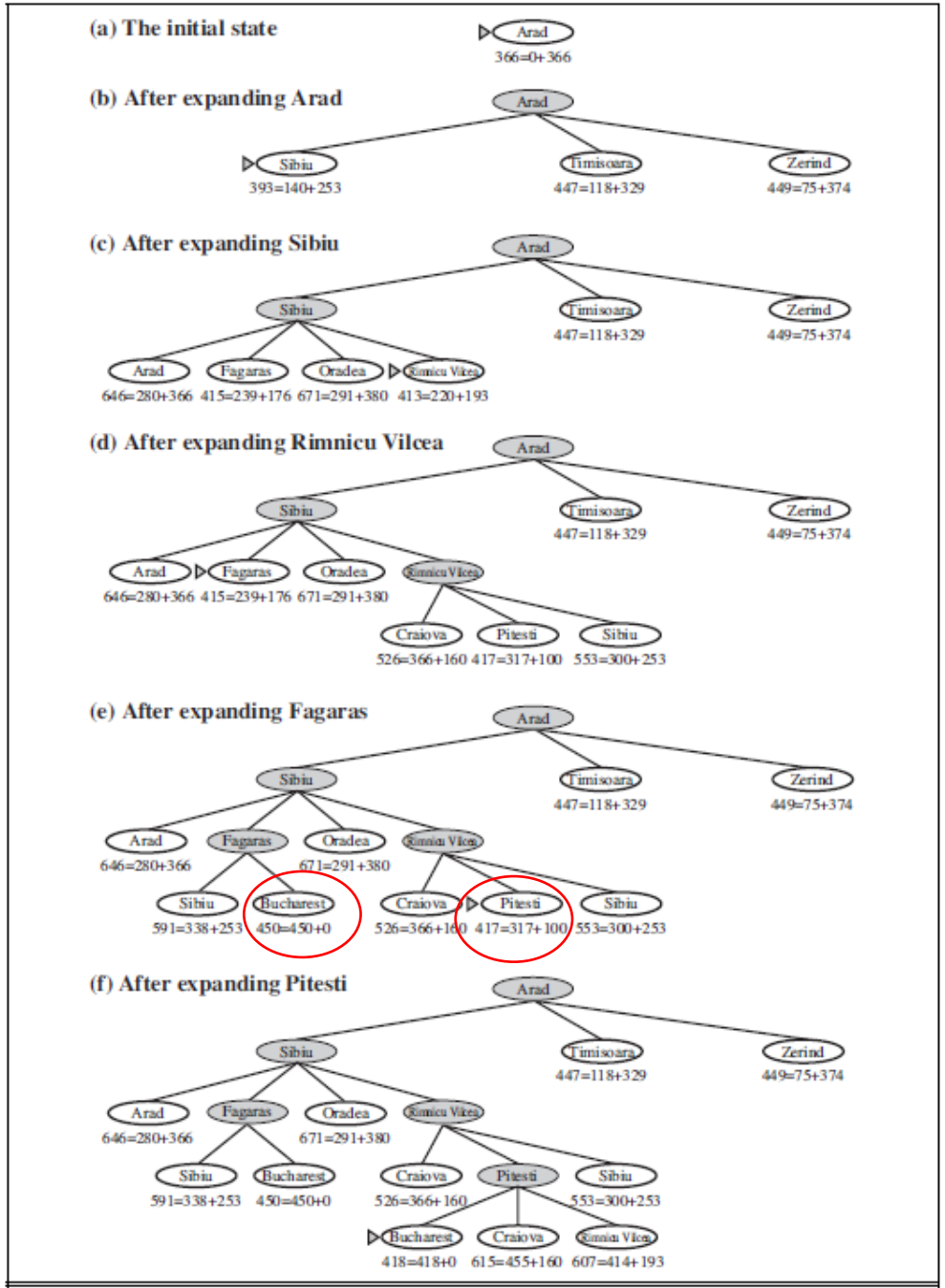


Figure 3.24 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.22.

Consistency

>> Condition 2: A heuristic *h(n)* is **consistent** if, for every node *n* and every successor *n'* of *n* generated by any action *a*, the estimated cost of reaching the goal from *n* is no greater than the step cost of getting to *n'* plus the estimated cost of reaching the goal from *n'*: $h(n) \leq c(n, a, n') + h(n')$

>> This is a form of the general triangle inequality which stipulates that each side of a triangle cannot be longer than the sum of the other two sides. Here, the triangle is formed by n, n', and the goal G_n closest to n.

>> Every consistent heuristic is also admissible. e.g. *hSLD*

CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Optimality of A*

>> *The tree-search version of A∗ is optimal if h(n) is admissible, while the graph-search version is optimal if h(n) is consistent.*

>> Let us establish: *if h(n) is consistent, then the values of f(n) along any path are nondecreasing.*

> Suppose *n'* is a successor of *n*; then *g(n')=g(n) + c(n, a, n')* for some action *a*, and we have

$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n)$$

> The next step is to prove that whenever A∗ selects a node *n* for expansion, the optimal path to that node has been found.

Optimality of A* - contours

- The fact that f-costs are nondecreasing along any path also means that we can draw contours in the state space.
- Then, because A∗ expands the frontier node of lowest f-cost, we can see that an A∗ search fans out from the start node, adding nodes in concentric bands of increasing f-cost.
- With uniform-cost search (A∗ search using h(n) = 0), the bands will be “circular” around the start state. With more accurate heuristics, the bands will stretch toward the goal state and become more narrowly focused around the optimal path.
- If C∗ is the cost of the optimal solution path, then we can say the following:

- A∗ expands all nodes with f(n) < C

- A∗ might then expand some of the nodes right on the “goal contour” (where f(n) = C∗) before selecting a goal node.

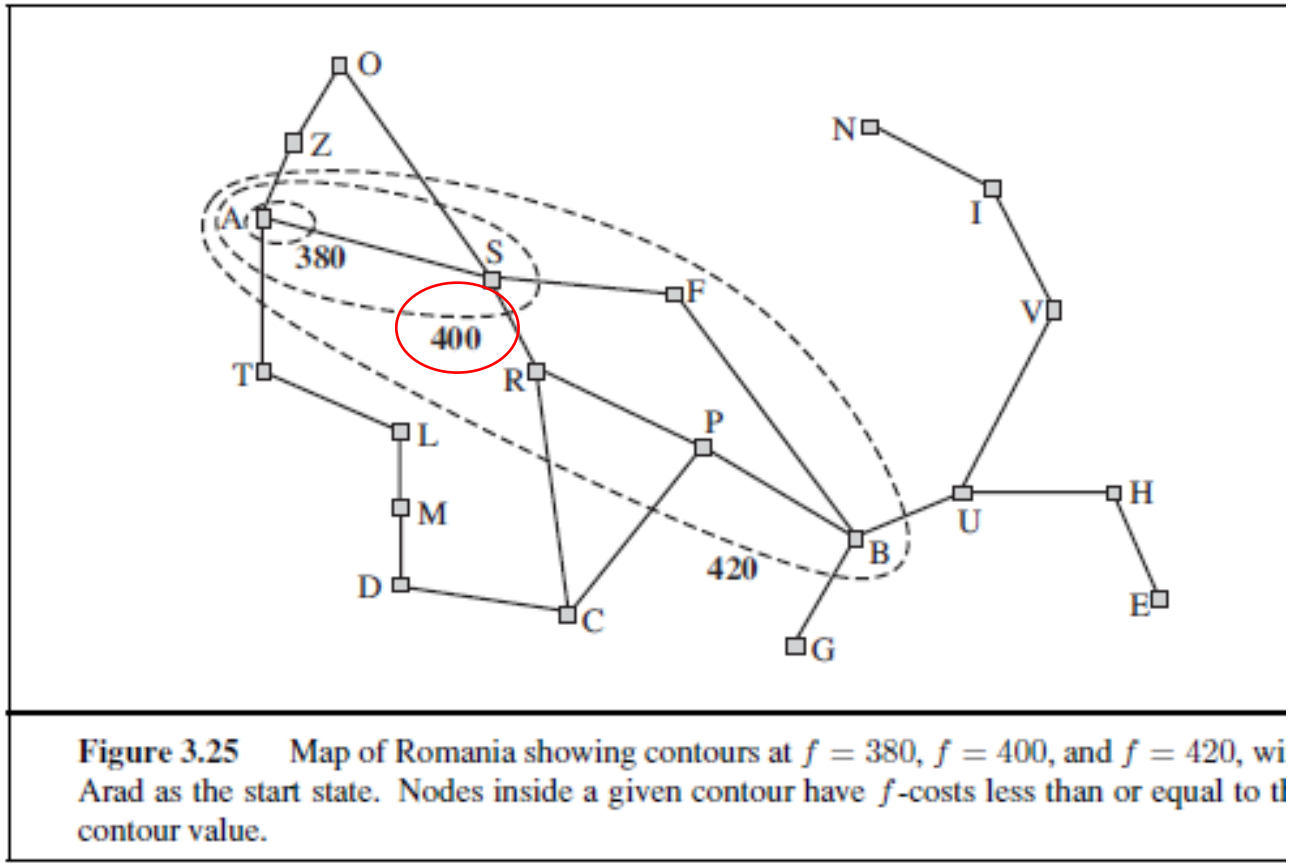


Figure 3.25 Map of Romania showing contours at $f = 380$, $f = 400$, and $f = 420$, wi Arad as the start state. Nodes inside a given contour have f-costs less than or equal to tl contour value.

Optimality of A* - completeness

- Completeness requires that there be only finitely many nodes with cost less than or equal to $C*$, a condition that is true if all step costs exceed some finite ϵ and if b is finite.

- Notice that A∗ expands no nodes with f(n) > C∗—for example, Timisoara is not expanded even though it is a child of the root. We say that the subtree below Timisoara is **pruned**; because *hSLD* is admissible, the algorithm can safely ignore this subtree while still guaranteeing optimality

- Among optimal algorithms of this type, algorithms that extend search paths from the root and use the same heuristic information, A∗ is optimally efficient for any given consistent heuristic.

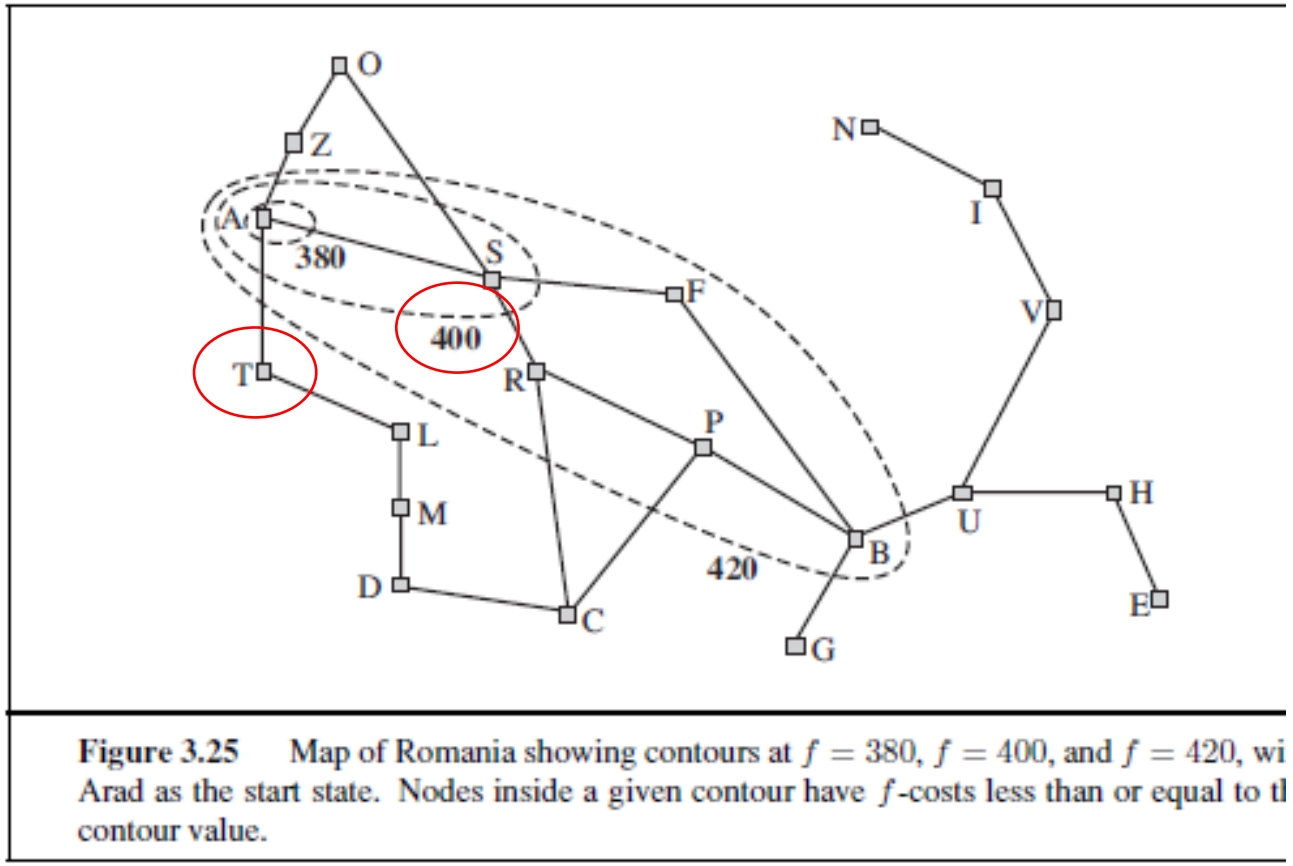


Figure 3.25 Map of Romania showing contours at $f = 380$, $f = 400$, and $f = 420$, wi Arad as the start state. Nodes inside a given contour have f-costs less than or equal to tl contour value.

CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

A* search

>> A∗ search is complete, optimal, and optimally efficient among all such algorithms.

>> The catch is that, for most problems, the number of states within the goal contour search space is still exponential in the length of the solution.

>> For problems with constant step costs, the growth in run time as a function of the optimal solution depth *d* is analyzed in terms of the **absolute error** or the **relative error** of the heuristic. The **absolute error** is defined as $\Delta \equiv h* - h$, where $h*$ is the actual cost of getting from the root to the goal, and the **relative error** is defined as $\in \equiv (h* - h)/h*$.

CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

A* search - complexity

>> Complexity results depend on the assumptions made about the state space. The simplest model studied is a state space that has a single goal and is essentially a tree with reversible actions. In this case, the time complexity of A$*$ is exponential in the maximum absolute error, that is, O(b^{Δ}). For constant step costs, we can write this as O($b^{\epsilon d}$), where *d* is the solution depth.

>> For almost all heuristics in practical use, the absolute error is at least proportional to the path cost h$*$, so ϵ is constant or growing and the time complexity is exponential in d. We can also see the effect of a more accurate heuristic: O($b^{\epsilon d}$)=O((b^{ϵ})^{d}), so the effective branching factor is b^{ϵ}.

>> Computation time is not, however, A$*$'s main drawback. Because it keeps all generated nodes in memory (as do all GRAPH-SEARCH algorithms), A$*$ usually runs out of space long before it runs out of time. For this reason, A$*$ is not practical for many large-scale problems.

CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Memory-bounded heuristic search

>> The simplest way to reduce memory requirements for A∗ is to adapt the idea of iterative deepening to the heuristic search context, resulting in the iterative-deepening A∗ (IDA∗) algorithm.

>> The main difference between IDA∗ and standard iterative deepening is that the cutoff used is the *f* cost *(g+h)* rather than the depth; at each iteration, the cutoff value is the smallest *f* cost of any node that exceeded the cutoff on the previous iteration.

>> IDA∗ is practical for many problems with unit step costs and avoids the substantial overhead associated with keeping a sorted queue of nodes. Unfortunately, it suffers from the same difficulties with real valued costs as does the iterative version of uniform-cost search. We briefly examine two other memory-bounded algorithms, called RBFS and MA∗

CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Recursive best-first search (RBFS)

>> RBFS is a simple recursive algorithm that attempts to mimic the operation of standard best-first search but using only linear space. Its structure is like that of a recursive depth-first search, but rather than continuing indefinitely down the current path, it uses the *f* limit variable to keep track of the *f*-value of the best alternative path available from any ancestor of the current node.

>> If the current node exceeds this limit, the recursion unwinds back to the alternative path. As the recursion unwinds, RBFS replaces the f-value of each node along the path with a backed-up value—the best f-value of its children.

>> In this way, RBFS remembers the f-value of the best leaf in the forgotten subtree and can therefore decide whether it's worth re-expanding the subtree at some later time. We will show how RBFS reaches Bucharest.

RBFS -> Bucharest

- RBFS is somewhat more efficient than IDA∗, but still suffers from excessive node regeneration.
- Like A∗ tree search, RBFS is an optimal algorithm if the heuristic function *h(n)* is admissible. Its space complexity is linear in the depth of the deepest optimal solution, but its time complexity is rather difficult to characterise: it depends both on the accuracy of the heuristic function and on how often the best path changes as nodes are expanded.
- IDA∗ and RBFS suffer from using too little memory. Between iterations, IDA∗ retains only a single number: the current *f*-cost limit. RBFS retains more information in memory, but it uses only linear space: even if more memory were available, RBFS has no way to make use of it. Because they forget most of what they have done, both algorithms may end up re-expanding the same states many times over.
- Furthermore, they suffer the potentially exponential increase in complexity associated with redundant paths in graphs

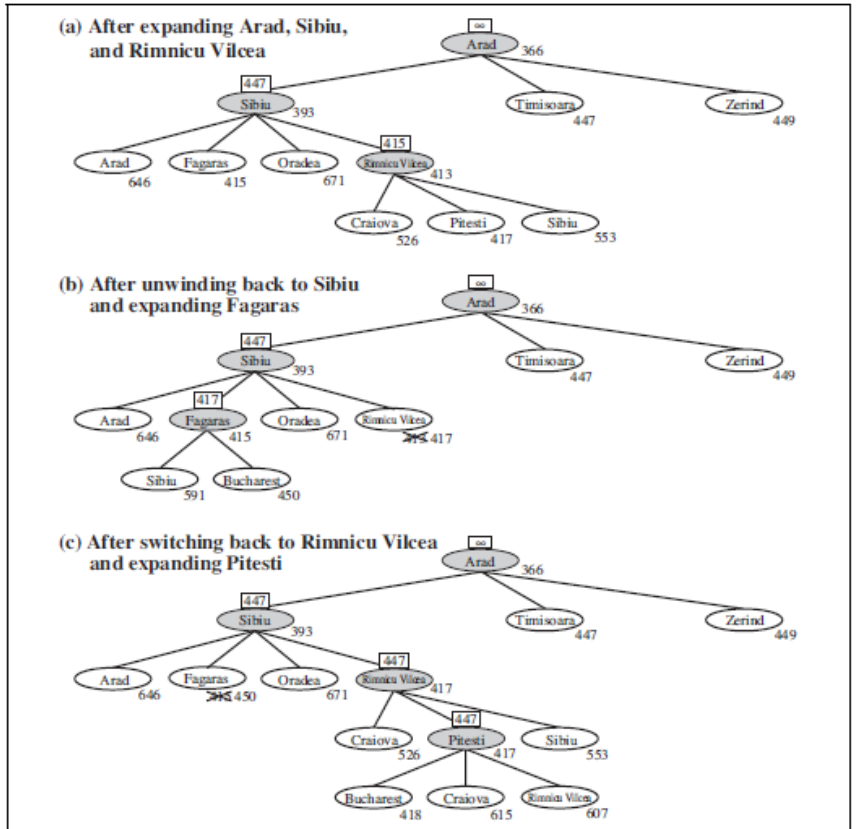


Figure 3.27 Stages in an RBFS search for the shortest route to Bucharest. The *f*-limit value for each recursive call is shown on top of each current node, and every node is labeled with its *f*-cost. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras). (b) The recursion unwinds and the best leaf value of the forgotten subtree (417) is backed up to Rimnicu Vilcea; then Fagaras is expanded, revealing a best leaf value of 450. (c) The recursion unwinds and the best leaf value of the forgotten subtree (450) is backed up to Fagaras; then Rimnicu Vilcea is expanded. This time, because the best alternative path (through Timisoara) costs at least 447, the expansion continues to Bucharest.

CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Simplified Memory Bounded A* (SMA*)

>> SMA∗ proceeds just like A∗, expanding the best leaf until memory is full. At this point, it cannot add a new node to the search tree without dropping an old one. SMA∗ always drops the worst leaf node—the one with the highest f-value.

>> Like RBFS, SMA∗ then backs up the value of the forgotten node to its parent. In this way, the ancestor of a forgotten subtree knows the quality of the best path in that subtree. With this information, SMA∗ regenerates the subtree only when all other paths have been shown to look worse than the path it has forgotten. If all the descendants of a node *n* are forgotten, then we will not know which way to go from *n*, but we will still have an idea of how worthwhile it is to go anywhere from *n*.

>> The complete algorithm is too complicated to reproduce here, but there is one subtlety worth mentioning.

CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Learning to search better

>> At this stage, several fixed strategies—breadth-first, greedy best-first, and so on—have been presented. Could an agent learn how to search better? **metalevel state space**.

>> Each state in a metalevel state space captures the internal (computational) state of a program that is searching in an **object-level state space** such as Romania.

>> Each action in the metalevel state space is a computation step that alters the internal state; for example, each computation step in A∗ expands a leaf node and adds its successors to the tree.

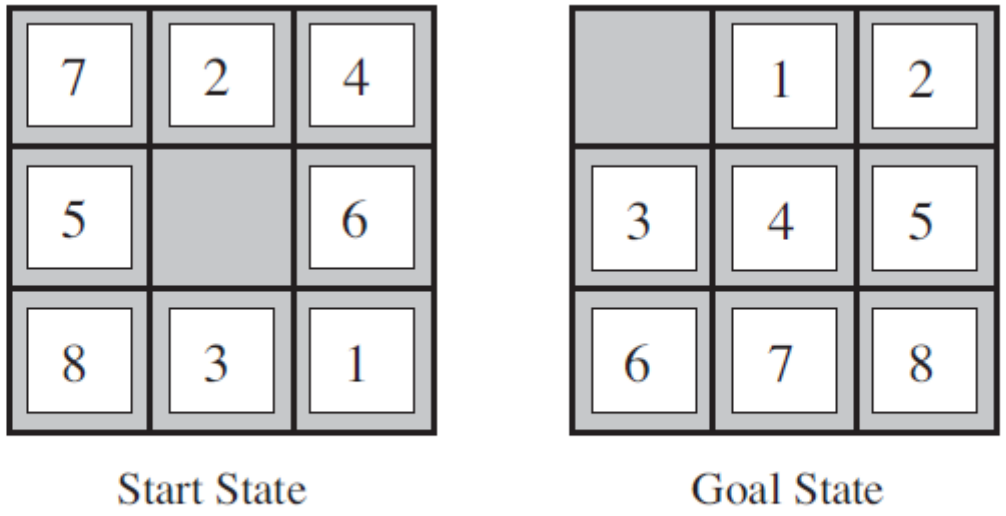
>> Fig 3.24 five steps including an unhelpful step (expansion of Fagaras). Harder problems = > no. of missteps - a **metalevel learning** algorithm can learn from these experiences to avoid exploring unpromising subtrees.

Heuristic functions

>> Heuristics for the 8-puzzle: The object of the puzzle is to slide the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration

Exhaustive Tree-Search: Depth 22, would look at about 3^22 ≈ 3.1×10^10

Graph-Search:170,000 because only 9!/2 = 181, 440 distinct states are reachable



15-puzzle is roughly 10^13 – so we need a good heuristic function.

A need heuristic function that overestimates no. steps to the goal*

Figure 3.28 A typical instance of the 8-puzzle. The solution is 26 steps long.

>> The average solution cost for a randomly generated 8-puzzle instance is about 22 steps. The branching factor is about 3. (When the empty tile is in the middle, four moves are possible; when it is in a corner, two; and when it is along an edge, three.)

CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

HEURISTIC FUNCTIONS

>> There is a long history of such heuristics for the 15-puzzle; here are two commonly used candidates:

h1 = the number of misplaced tiles. For the puzzle on the previous slide, all the eight tiles are out of position, so the start state would have *h1 = 8*. *h1* is an admissible heuristic because any tile that is out of place must be moved at least once.

h2 = the sum of the distances of the tiles from their goal positions. Because tiles cannot move along diagonals, the distance we will count is the sum of the horizontal and vertical distances. This is sometimes called the **city block distance** or **Manhattan distance**. *h2* is also admissible because all any move can do is move one tile one step closer to the goal. Tiles 1 to 8 in the start state give a Manhattan distance of

h2 = 3+1 + 2 + 2+ 2 + 3+ 3 + 2 = 18 .

As expected, neither of these overestimates the true solution cost, which is 26.

CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

The effect of heuristic accuracy on performance

>> One way to characterize the quality of a heuristic is the **effective branching factor** $b*$. If the total number of nodes generated by A∗ for a particular problem is N and the solution depth is d, then $b*$ is the branching factor that a uniform tree of depth d would have to have in order to contain **N + 1** nodes. Thus,

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

e.g. if A∗ finds a solution at depth 5 using 52 nodes, then the effective branching factor is 1.92.

>> To test the heuristic functions *h1* and *h2*, we generated 1200 random problems with solution lengths from 2 to 24 (100 for each even number) and solved them with iterative deepening search and with A∗ tree search using both *h1* and *h2*. Figure 3.29 gives the average number of nodes generated by each strategy and the effective branching factor. The results suggest that *h2* is better than *h1*, and is far better than using iterative deepening search. Even for small problems with d=12, A∗ with *h2* is 50,000 times more efficient than uninformed iterative deepening search.

CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

The effect of heuristic accuracy on performance

	Search Cost (nodes generated)			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	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	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 3.29 Comparison of the search costs and effective branching factors for the ITERATIVE-DEEPENING-SEARCH and A* algorithms with h_1, h_2. Data are averaged over 100 instances of the 8-puzzle for each of various solution lengths d.

>> Will *h2* always be better than *h1?* Yes, It is easy to see from the definitions of the two heuristics that, for any node *n, h2(n) ≥ h1(n).* We thus say that *h2* **dominates** *h1*.

Beyond Classical Search

Relaxing Simplifying Assumptions – getting closer to the real world

Local Search in the state space:

- simulated annealing
- genetic algorithms

Relaxing the assumptions of determinism and observability:

- agent cannot predict percepts
- Contingency?
- Tracking potential states

Online search

CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Local Search and optimization problems

>> Search algorithms to date - Explore search spaces **systematically**, goal found return path.

>> What if the path to the goal is irrelevant? e.g. n-queens

>> If the path to the goal does not matter, we can consider a different class of algorithm:

> **Local search**

- does not concern itself with paths
- Uses a single **current node** and generally moves only to neighbors of that node.
- Typically paths followed are not retained.
- Pros: Low memory usage & can find reasonable solutions in large/infinite state spaces (where systematic algorithms would be unsuitable)

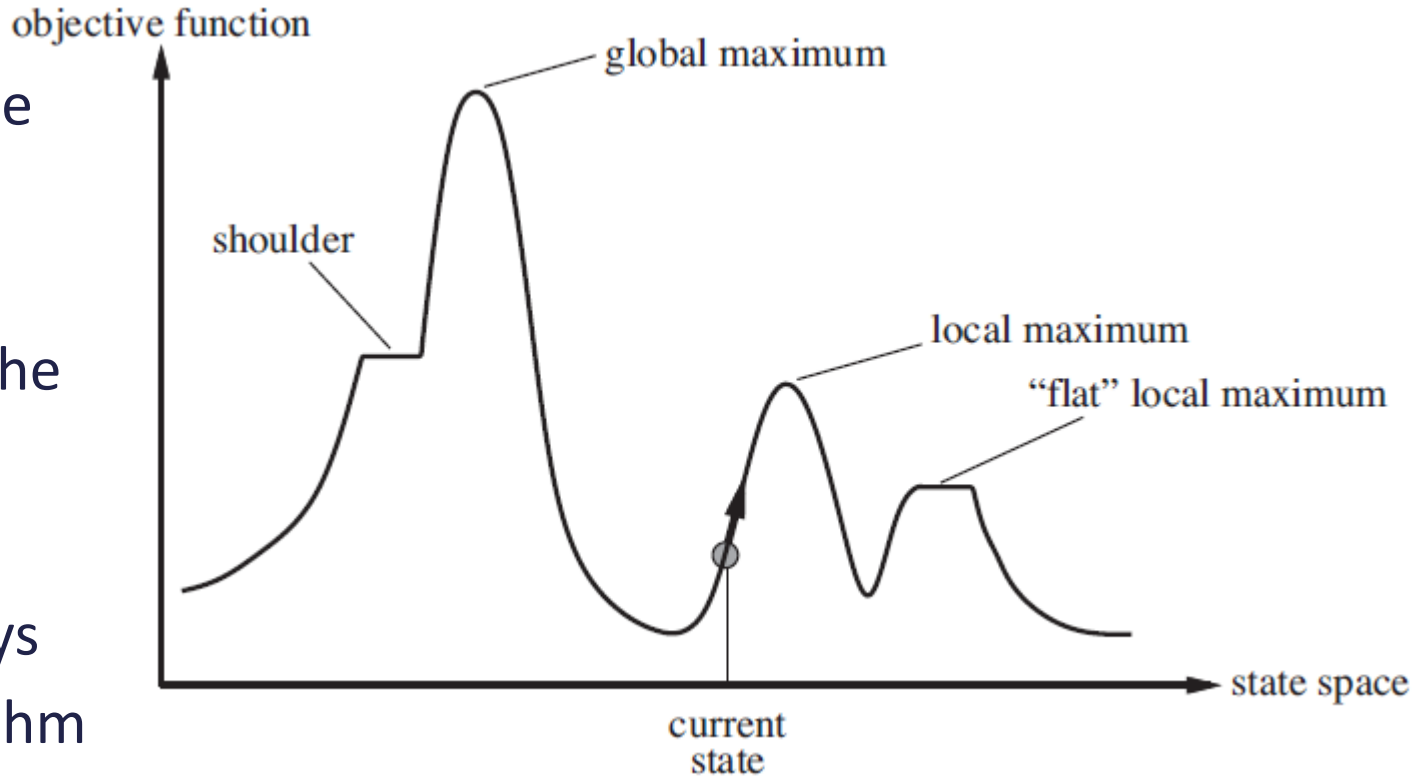
>> In addition to finding goals, local search algorithms are useful for solving pure **optimisation problems**, in which the aim is to find the best state according to an **objective function**.

State-space landscape

>> A landscape has both “location” and “elevation”.

>> If elevation corresponds to cost, then the aim is to find the lowest valley—a **global minimum**; if elevation corresponds to an objective function, then the aim is to find the highest peak—a **global maximum**.

>> A **complete** local search algorithm always finds a goal if one exists; an **optimal** algorithm always finds a global minimum/maximum.



CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Hill-climbing search

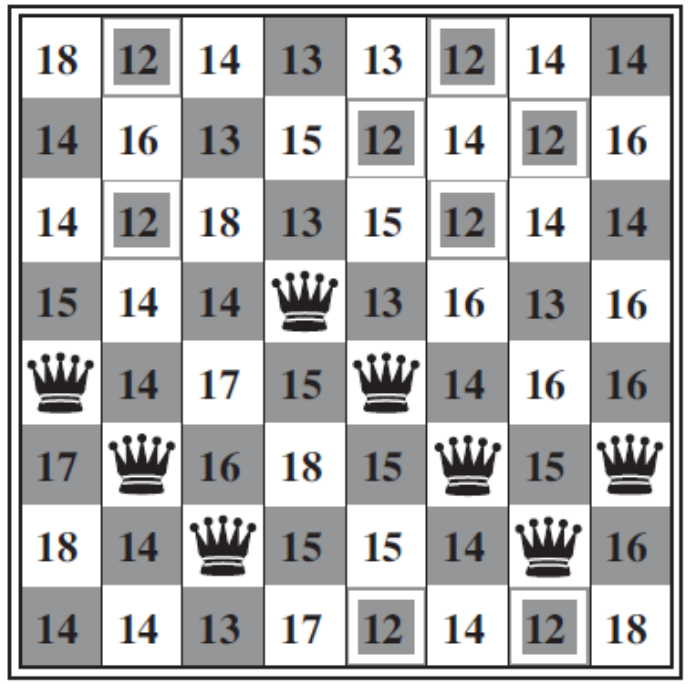
>> The **hill-climbing** search algorithm (**steepest-ascent** version) is shown. It is simply a loop that continually moves in the direction of increasing value—that is, uphill

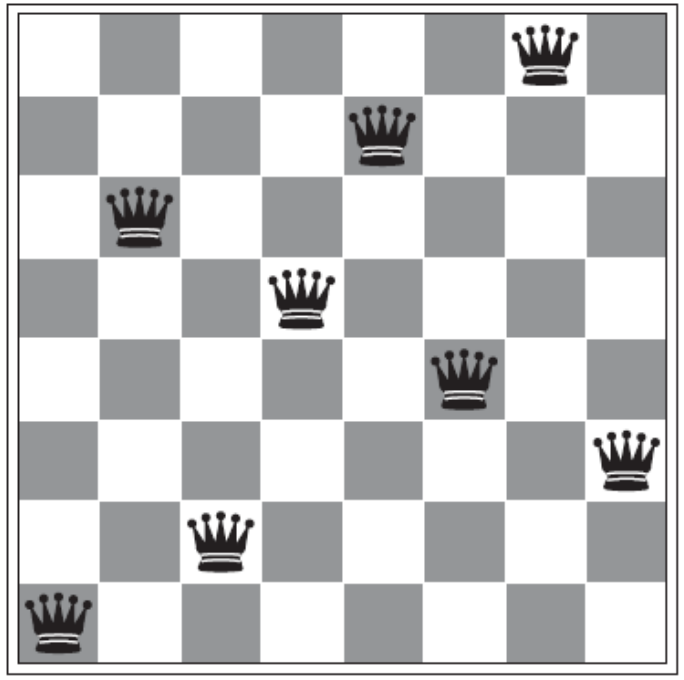
```
function HILL-CLIMBING(problem) returns a state that is a local maximum

    current ← MAKE-NODE(problem.INITIAL-STATE)
    loop do
        neighbor ← a highest-valued successor of current
        if neighbor.VALUE ≤ current.VALUE then return current.STATE
        current ← neighbor
```

Figure 4.2 The hill-climbing search algorithm, which is the most basic local search technique. At each step the current node is replaced by the best neighbor; in this version, that means the neighbor with the highest VALUE, but if a heuristic cost estimate h is used, we would find the neighbor with the lowest h.

CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

E.g. **8-queens problem**





(a) (b)

Figure 4.3 (a) An 8-queens state with heuristic cost estimate $h = 17$, showing the value of h for each possible successor obtained by moving a queen within its column. The best moves are marked. (b) A local minimum in the 8-queens state space; the state has $h = 1$ but every successor has a higher cost.

CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Hill-climbing gets stuck

>> **Local maxima**: a local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum.

>> **Ridges**: result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.

>> **Plateaux**: a plateau is a flat area of the state-space landscape. It can be a flat local maximum, from which no uphill exit exists, or a shoulder, from which progress is possible. A hill-climbing search might get lost on the plateau.

E.g. Random 8-queesn: Stuck 86% (avg. 3 steps) Success 14% (avg. 4 steps) 8^8 states ~ 17m

>> Problem: halting on a plateau? Solution: allow a **sideways move** in the hope that the plateau is really a shoulder (see Fig 4.1).

E.g. 100 sideways moves: Failure 6% (avg. 64 steps) Success 94% (avg. 21 steps)

CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Hill-climbing variants

>> **Stochastic hill climbing** chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move. This usually converges more slowly than steepest ascent, but in some state landscapes, it finds better solutions.

>> **First-choice hill climbing** implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state.

>> **Random-restart hill climbing** adopts the well-known adage, "If at first you don't succeed, try, try again."

8-queens instances with no sideways moves allowed, p ≈ 0.14, so we need roughly 7 iterations

>> Probability success: *p => 1/p = no. of restarts req.*

8-queens instances with sideways moves, p = 1/0.94 ≈ 1.06

CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Simulated annealing

>> A hill-climbing algorithm that never makes “downhill” moves toward states with lower value (or higher cost) is guaranteed to be incomplete, because it can get stuck on a local maximum. In contrast, a purely random walk—that is, moving to a successor chosen uniformly at random from the set of successors—is complete but extremely inefficient.

>> Simulated annealing addresses these issues by combining hill climbing with a random walk in a manner that yields both efficiency and completeness.

>> To explain simulated annealing, we switch our point of view from hill climbing to gradient descent (i.e., minimizing cost)

CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Simulated annealing

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"

  current ← MAKE-NODE(problem.INITIAL-STATE)
  for t = 1 to ∞ do
      T ← schedule(t)
      if T = 0 then return current
      next ← a randomly selected successor of current
      ΔE ← next.VALUE – current.VALUE
      if ΔE > 0 then current ← next
      else current ← next only with probability e^(ΔE/T)
```

Figure 4.5 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. Downhill moves are accepted readily early in the annealing schedule and then less often as time goes on. The *schedule* input determines the value of the temperature T as a function of time.

Summary

- TO BE COMPLETED

CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

References

[1] Russell, S. and Norvig, P., 2002. Artificial intelligence: a modern approach – Solving Problems by Searching, Chapter 3 and Beyond Classical Search, Chapter 4.