CAP 4630
Artificial Intelligence

Instructor: Sam Ganzfried

sganzfri@cis.fiu.edu

- http://www.ultimateaiclass.com/
- https://moodle.cis.fiu.edu/
- HW1 out later today, due 9/14
  - Remember that you have up to 4 late days to use throughout the semester.
  - Will involve a Python programming problem http://ai.berkeley.edu/search.html, and a few exercises from the textbook.
- Office hours

Comparing uninformed search strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

Informed (heuristic) search strategies

- General approach we consider is called **best-first search**.
- For the uninformed graph and tree search, the order in which a node is selected for evaluation was based on being first or last in.
- For informed best-first search, we assume we have an **evaluation function** f(n). This can be construed as a cost estimate, so the node with the *lowest* evaluation is expanded first. The implementation of best-first search is the same as for UCS, except for the use of f instead of g to order the priority queue.

Best-first search

- The choice of f determines the search strategy.
  - Exercise 3.21 shows that best-first tree search includes DFS as a special case.
- Most best-first search 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.

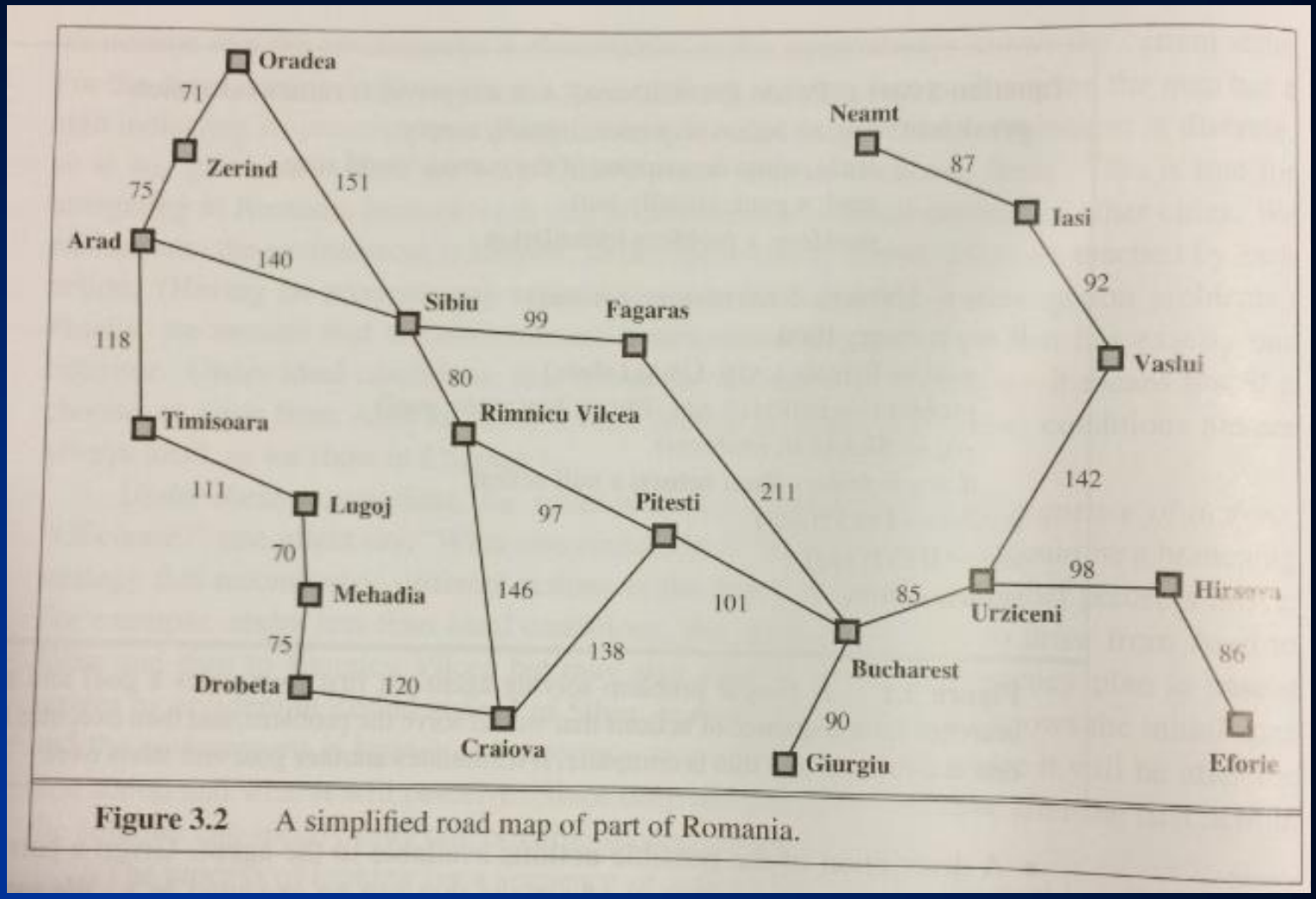
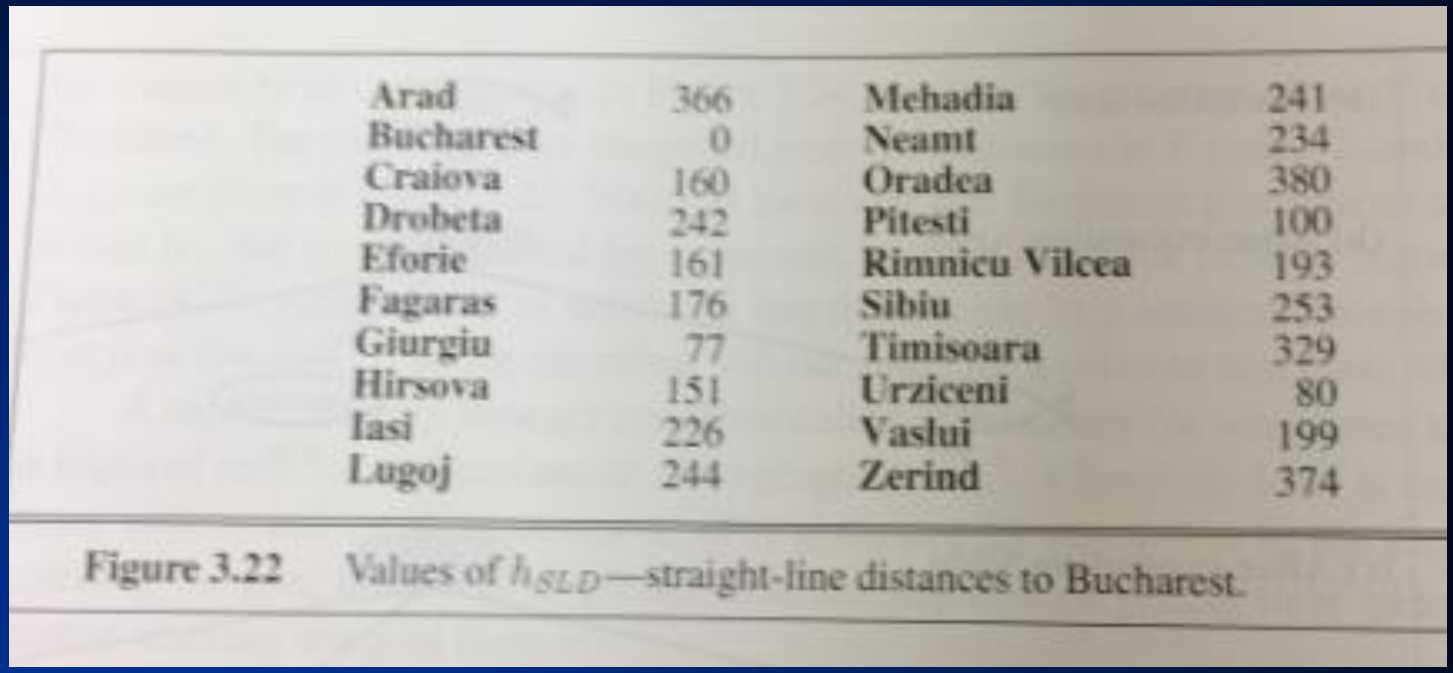


Figure 3.2 A simplified road map of part of Romania.

Best-first search

- What are some possible heuristic functions for the Romania map problem?

Straight-line distance



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

Figure 3.22 Values of h_{SLD}—straight-line distances to Bucharest.

Best-first search

- Heuristic functions are the most common form in which additional knowledge of the problem is imparted to the search algorithm. We consider them to be arbitrary, nonnegative, problem-specific functions, with one constraint: if n is a goal node, then h(n) = 0.

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).
- Romania using SLD with goal Bucharest.
- E.g., h-SLD(In(Arad)) = 366.
- Search cost is minimal, but it is not optimal; the path via Sibiu and Faragas to Bucharest is 32 km longer than the path through RV and Pitesti. This shows why the algorithm is “greedy” – at each step it tries to get as close to the goal as it can.

Greedy best-first search

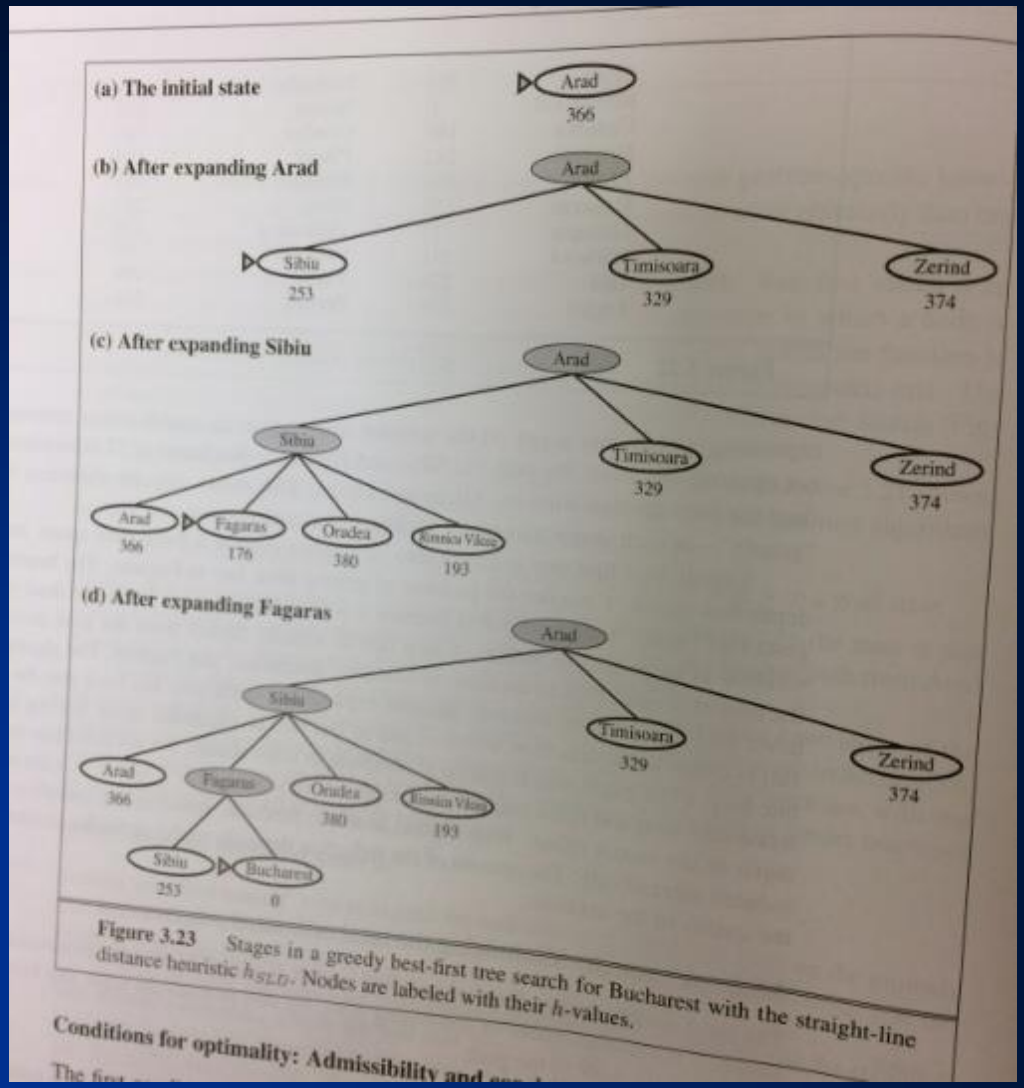


Figure 3.23 Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

Conditions for optimality: Admissibility and

The

Greedy best-first search

- GBFS is also incomplete even in a finite state space, much like DFS. Consider the problem of getting from Iasi to Faragas. The heuristic suggests that Neamt be expanded first because it is closest to Faragas, but it is a dead end. The solution is to first go to Vaslui—a step that is actually farther from the goal according to the heuristic—and then to continue to Urziceni, Bucharest, and Fagaras. The algorithm will never find this solution, however, because expanding Neamt puts Iasi back into the frontier. Iasi is closer to Fagaras than Vaslui is, and so Iasi will be expanded again, leading to an infinite loop.

Greedy best-first search

- The graph search version *is* complete in finite spaces, but not in infinite ones.
- The worst-case time and space complexity for the tree version is O(b^m), where m is the maximum depth of the search space
  - This was the running time of DFS
- With a good heuristic function, however, the complexity can be reduced substantially. The amount of the reduction depends on the particular problem and on the quality of the heuristic.

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 (recall UCS), 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

A* search

- Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of g(n) + h(n). It turns out that his strategy is more than just reasonable: provided that the heuristic function h(n) satisfies certain conditions, A* search is both complete and optimal. The algorithm is identical to Uniform-Cost-Search except that A* uses g+h instead of g.

A* search

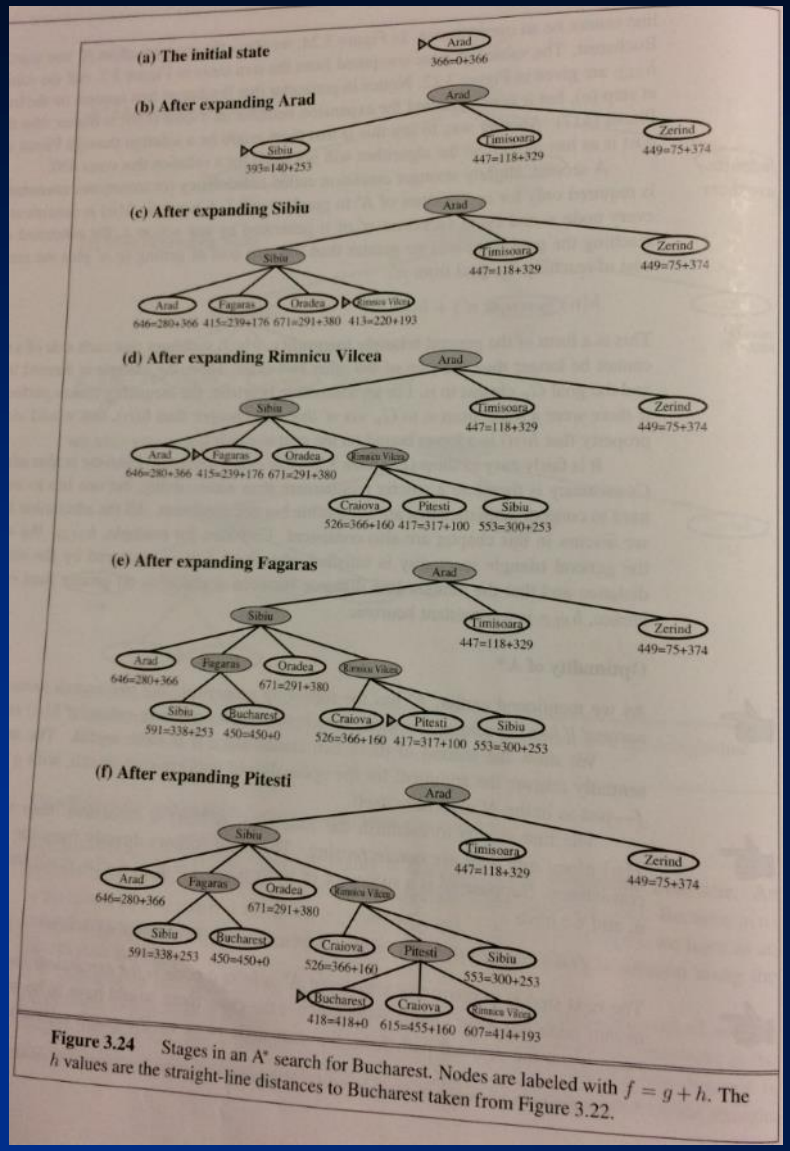


Figure 3.24 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.22.

Conditions for optimality

- The first condition we require for optimality is that h(n) be an **admissible heuristic**. An admissible heuristic is one that *never overestimates* the cost to reach the goal. Because g(n) is the actual cost to reach n along the current path, and f(n) = g(n) + h(n), we have as an immediate consequence that f(n) never overestimates the true cost of a solution along the current path through n.
- Admissible heuristics are *optimistic*, because they think the cost of solving the problem is less than it actually is. Straight-line distance heuristic is admissible.

Conditions for A* optimality

- A second, slightly stronger condition called consistency is required only for applications of A* to graph search. A heuristic h(n) is consistent if, for every node n and every successor 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) <= c(n,a,n’) + h(n’)

Conditions for optimality: Consistency

- 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.
- For an admissible heuristic the inequality makes perfect sense: if there were a route from n to G_n via n' that was cheaper than h(n), that would violate the property that h(n) is a lower bound on the cost to reach G_n.

Optimality of A*

- It can be shown that every consistent heuristic is also admissible (homework problem). Consistency is therefore a stricter requirement than admissibility, but one has to work hard to concoct heuristics that are admissible but not consistent.
- Straight-line distance is consistent. The general triangle inequality is satisfied when each side is measured by the straight-line distance and that the straight-line distance between n and n’ is no greater than c(n,a,n’).

Optimality of A*

- As mentioned earlier, A* has the following properties: *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.*
- Argument for the second claim mirrors the argument for optimality of UCS.
  1. If h(n) is consistent, then the values of f(n) along any path are nondecreasing.
     - Follows from definition of consistency
  2. Whenever A* selects a node n for expansion, the optimal path to that node has been found.
     - Were this not the case, there would have to be another frontier node n' on the optimal path from the start node to n, …

Optimality of A*

- From the two observations, it follows that the sequence of nodes expanded by A* using GRAPH-SEARCH is in nondecreasing order of f(n). Hence, the first goal node selected for expansion must be an optimal solution because f is the true cost for goal nodes (which have h = 0) and all later goal nodes will be at least as expensive.
- The fact that f-costs are nondecreasing along any path also means that we can draw **contours** in the state space, just like the contours of a topographic map.

Optimality of A*

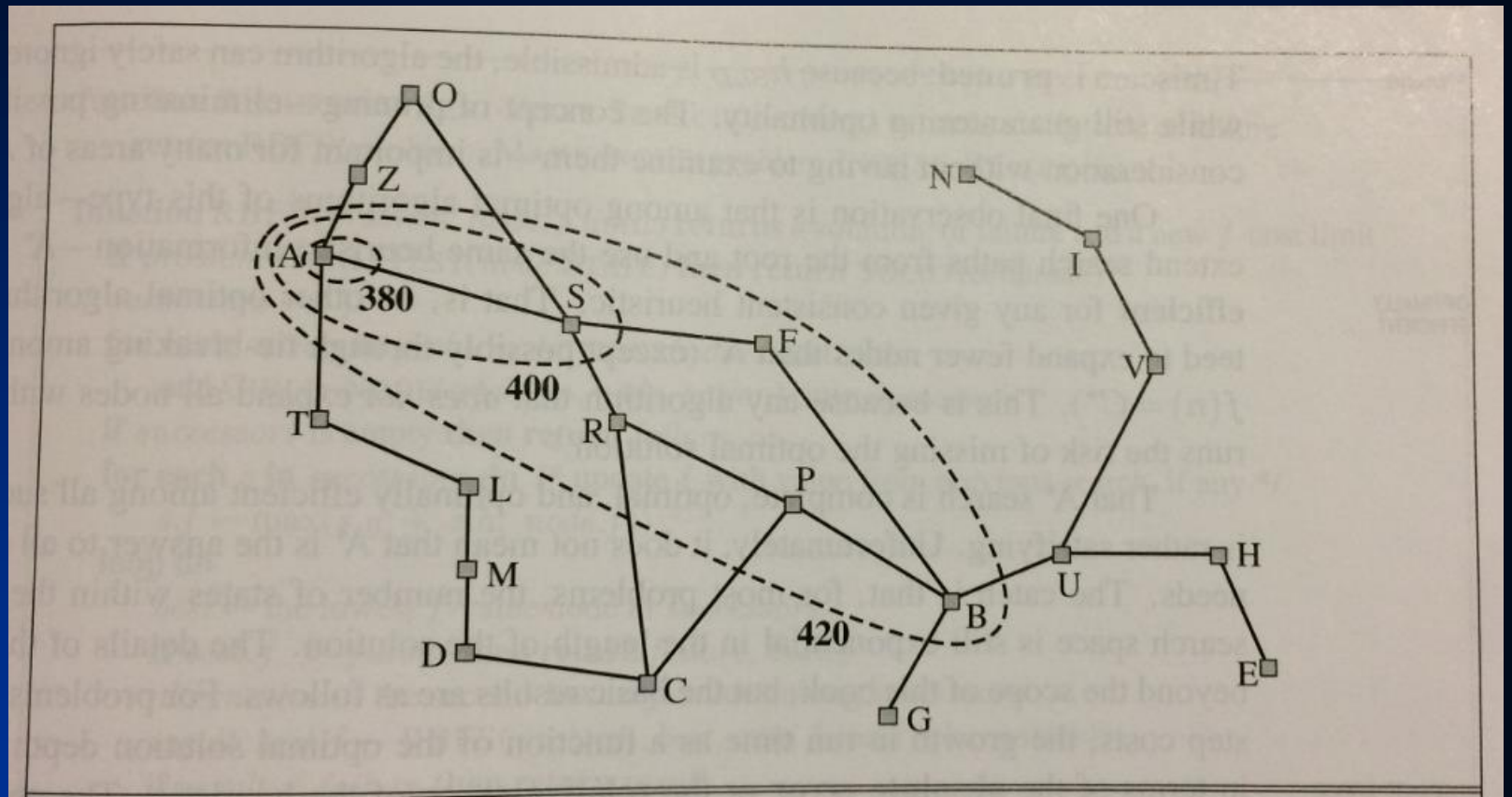


Figure 3.25 Map of Romania showing contours at $f = 380$, $f = 400$, and $f = 420$, with Arad as the start state. Nodes inside a given contour have f-costs less than or equal to the contour value.

Optimality of A*

- 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 counter" (where f(n) = C*) before selecting a goal node.
- 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.

Optimality of A*

- 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 h-SLD is admissible, the algorithm can safely ignore this subtree while still guaranteeing optimality.
- A final observation 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 heuristic. That is, no other optimal algorithm is guaranteed to expand fewer nodes than A* (except possibly through tie-breaking among nodes with f(n) = C*). This is because any algorithm that *does not* expand all nodes with f(n)<C* runs the risk of missing the optimal solution.

Optimality of A*

- Thus, 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 solutions. 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 AE=h*-h, where h* is the actual cost of getting from the root to the goal. And the relative error is defined as RE = (h*-h)/h*.

Optimality of A*

- The complexity results depend very strongly 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 (The 8-puzzle we saw satisfies first and third assumptions). In this case, the time complexity of A* is exponential in the maximum absolute error, that is, O(b^AE). For constant step costs, we can write this as O(b^(RE * 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 RE 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^(RE * d))
- = O((b^RE)^d), so the *effective branching factor* is b^RE.

Optimality of A*

- The complexity of A* often makes it impractical to insist on finding an optimal solution. One can use variants of A* that find suboptimal solutions quickly, or one can sometime design heuristics that are more accurate but not strictly admissible. In any case, the use of a good heuristic still provides enormous savings compared to the use of an uninformed search.
- 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. There are, however, algorithms that overcome the space problem without sacrificing optimality or completeness, at a small cost in execution time.

Memory-bounded heuristic search

- IDA*: Adapt the idea of iterative deepening to the heuristic search context. 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 (homework exercise).

Memory-bounded heuristic search

- **Recursive best-first search**: recursive algorithm that attempts to mimic operation of standard best-first search, using only linear space. As recursion unwinds, RBFS replaces f-value of each node along the path with a **backed-up value**—the best f-value of its children.
- RBFS is somewhat more efficient than IDA*, but still suffers from excessive node generation. Like A*, RBFS is optimal if the heuristic is admissible. Its space is linear in the depth of the deepest optimal solution, but time is difficult to analyze.

Memory-bounded heuristic search

- IDA* and RBFS suffer from using *too little* memory. Between iterations, IDA* retains only a single number: the current f-cost limit. RBFSW 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 reexpanding the same states many times over. Furthermore, they suffer the potentially exponential increase in complexity associated with redundant paths in graphs.
- Seems more sensible therefore to use all available memory. Two algorithms do this: MA (memory-bounded A*) and SMA* (simplified MA*).

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 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. Another way of saying this is that, 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.

SMA*

- SMA* is complete if there is any reachable solution—that is, if d, the depth of the shallowest goal node, is less than the memory size (expressed in nodes). It is optimal if any optimal solution is reachable; otherwise, it returns the best reachable solution. In practical terms, SMA* is a fairly robust choice for finding solutions, particularly when the state space is a graph, step costs are not uniform, and node generation is expensive compared to the overhead of maintaining the frontier and the explored set.

Heuristic functions

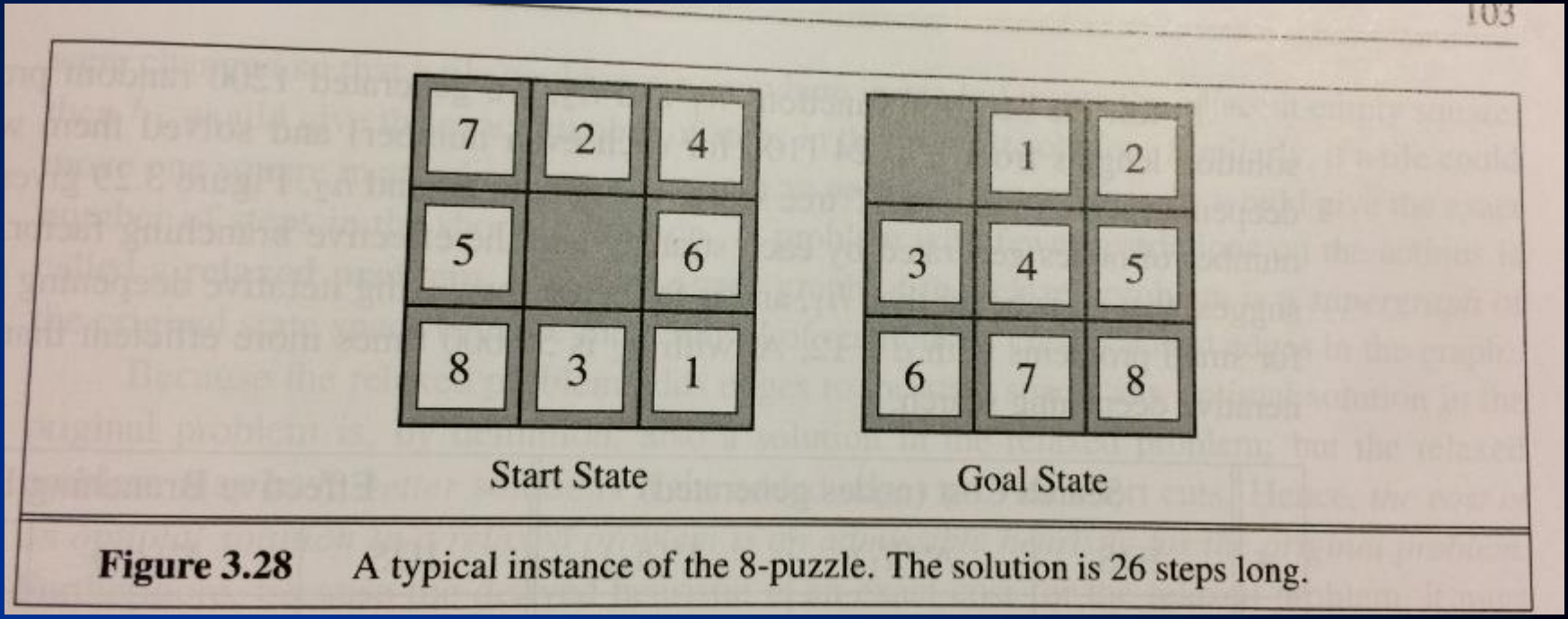


Figure 3.28 A typical instance of the 8-puzzle. The solution is 26 steps long.

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.
- 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). This means that an exhaustive tree search to depth 22 would look at about 3^22 = 3.1*10^10 states. A graph search would cut this down by a factor of about 170,000 because only 181,440 distinct states are reachable (homework exercise). This is a manageable number, but for a 15-puzzle, it would be 10^13, so we will need a good heuristic function.

8-puzzle

- If we want to find the shortest solutions by using A*, we need a heuristic function that never overestimates the number of steps to the goal. There is a long history of such heuristics for the 15-puzzle; here are two commonly used candidates:

8-puzzle

- h1 = the number of misplaced tiles. For the example, all of the 8 tiles are out of position, so the start state would have h1 = 8.

  Is h1 admissible?

8-puzzle heuristic functions

- Yes h1 is admissible, because it is clear that 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 sometimes called the **city block distance** or **Manhattan distance**. Is h2 admissible?

8-puzzle heuristic functions

- Yes 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.

Heuristic functions

- 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,

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

- For example, if A* finds a solution at depth 5 using 52 nodes, then the effective branching factor is 1.92. The effective branching factor can vary across problem instances, but usually it is fairly constant for sufficiently hard problems. Therefore, experimental measurements of b* on a small set of problems can provide a good guide to the heuristic's overall usefulness. A well-designed heuristic would have a value of b* close to 1, allowing fairly large problems to be solved at a reasonable cost.

Heuristic functions

To test the heuristic functions h_1 and h_2, 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 h_1 and h_2. Figure 3.29 gives the average number of nodes generated by each strategy and the effective branching factor. The results suggest that h_2 is better than h_1, and is far better than using iterative deepening search. Even for small problems with $d = 12$, A* with h_2 is 50,000 times more efficient than uninformed iterative deepening search.

	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.

Heuristic functions

- 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 IDS and with A* tree search using both h1 and h2.
- The results suggest that h2 is better than h1, and is far better than using IDS. Even for small problems with d=12, A* with h2 is 50,000 times more efficient than uninformed IDS.

Heuristic functions

- One might ask whether h2 is *always* better than h1. The answer is "Essentially yes." It is easy to see from the definitions of the two that, for any node n, h2(n) >= h1(n). We thus say that h2 **dominates** h1. Domination translates directly into efficiency: A* using h2 will never expand more nodes than A* using h1 (except possibly for some nodes with f(n) = C*).
- Hence, it is generally better to use a heuristic function with higher values, provided it is consistent and that the computation time for the heuristic is not too long.

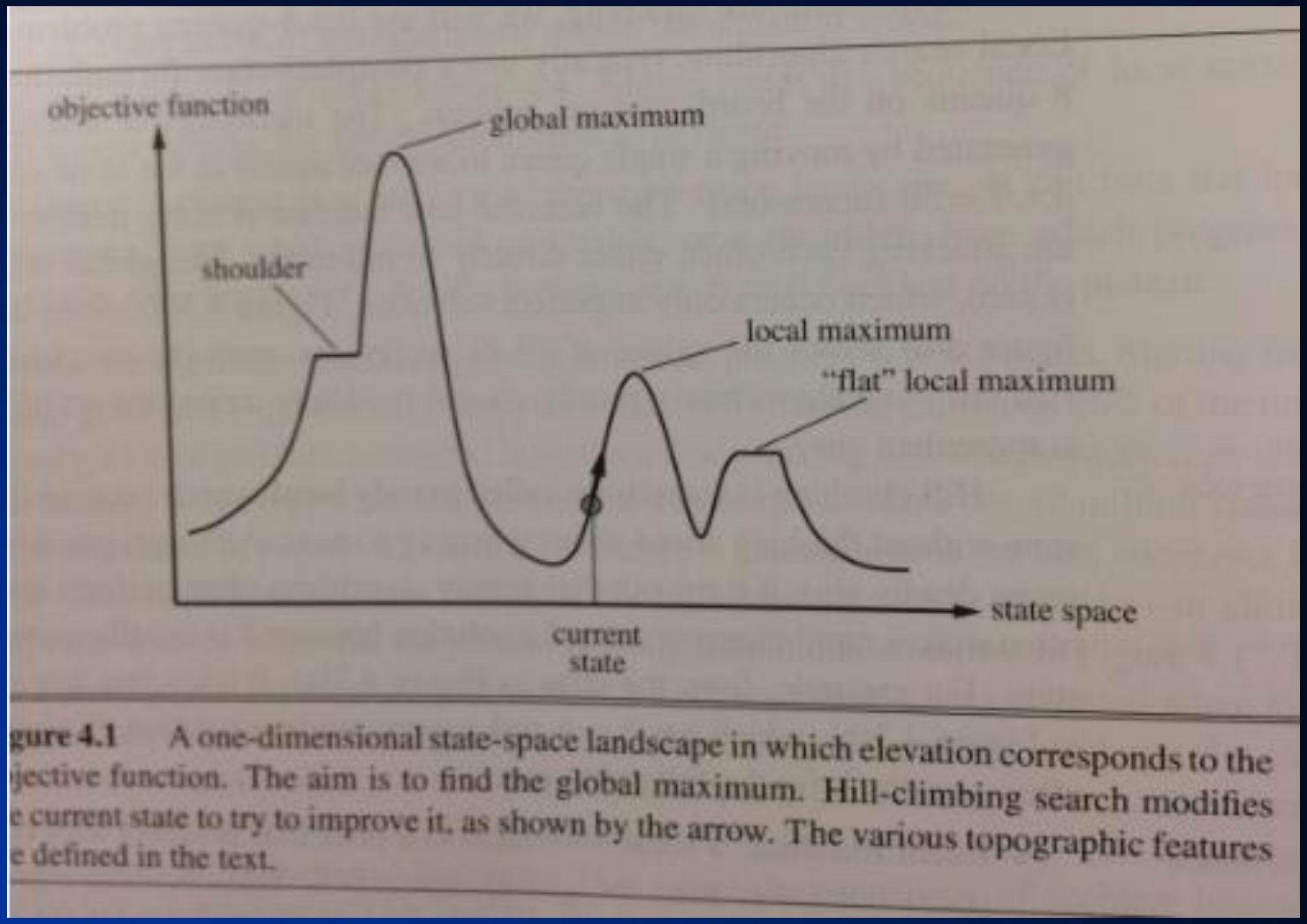
Search wrap-up

- Search-problem definition: Initial state, actions, transition model, path cost, state space, path, solution.
- General TREE-SEARCH and GRAPH-SEARCH algorithm. Tree-search considers all possible paths to find a solution, while graph-search avoids consideration of redundant paths.
- “Big 4” criteria: completeness, optimality, time complexity, space complexity. Often depends on branching factor b and d, depth of the shallowest solution.
- Uninformed search algorithms have access only to the problem definition: BFS, UCS, DFS, DLS, IDS, BDS.
- Informed search may have access to a heuristic function h(n) that estimates the cost of a solution from n: generic best-first search, greedy best-first search, A* search, RBFS, SMA*

Upcoming search paradigms

- Next time: quick introduction to alternative search methodologies.
- Local search: evaluates and modifies one or more current states, rather than systematically exploring paths from an initial state.
  - Global vs. local minimum/maximum, hill-climbing, simulated annealing, local beam search, genetic algorithms
- Adversarial search: search with multiple agents, where our optimal action depends on the cost/"utilities" of other agents and not just our own.
  - E.g., robot soccer, computer chess, etc.
  - Zero-sum games, perfect vs. imperfect information, minimax search, alpha-beta pruning
- Constraint satisfaction: assign a value to each variable that satisfies certain constraints. E.g., map coloring.

Local search



gure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the jective function. The aim is to find the global maximum. Hill-climbing search modifies e current state to try to improve it, as shown by the arrow. The various topographic features e defined in the text.

Adversarial search

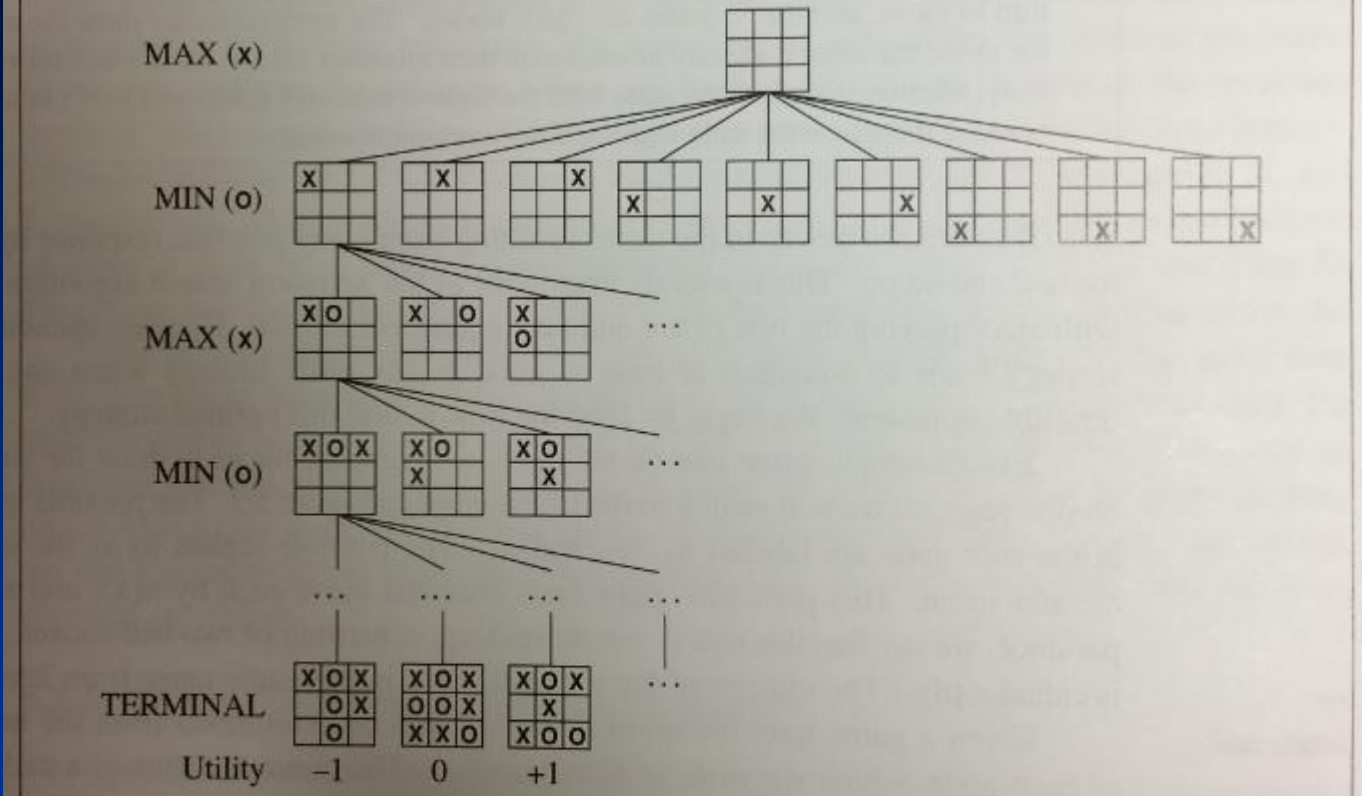
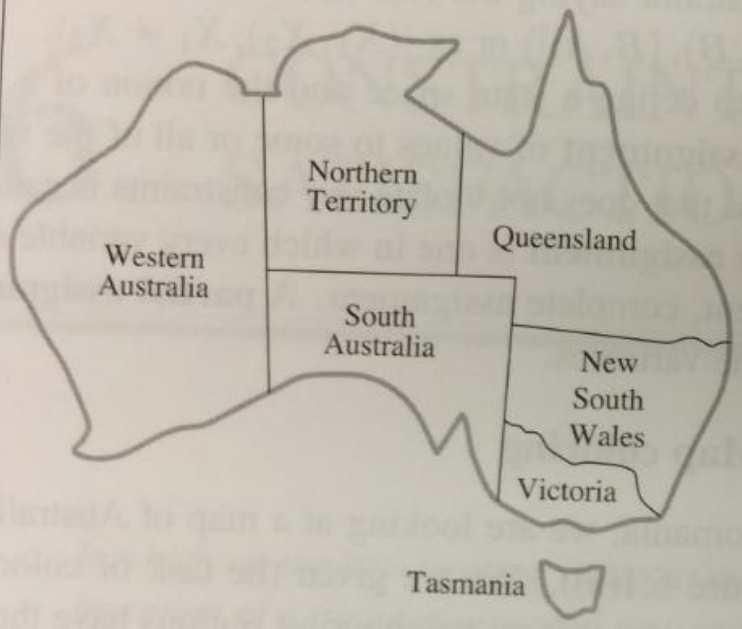


Figure 5.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

Constraint satisfaction



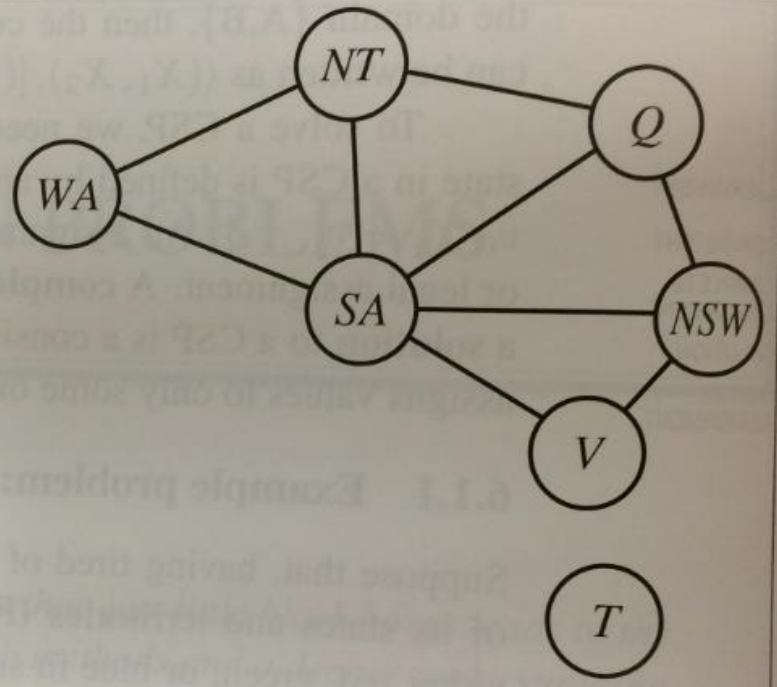


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

immediately discard further

Homework for next class

- Chapter 6 from Russell-Norvig textbook.
- HW1: out today due 9/14