CAP 4630
Artificial Intelligence

Instructor: Sam Ganzfried

sganzfri@cis.fiu.edu

- http://www.ultimateaiclass.com/
- https://moodle.cis.fiu.edu/
- HW1 out 9/5 today, due 9/28 (maybe 10/3)
  - Remember that you have up to 4 late days to use throughout the semester.
  - https://www.cs.cmu.edu/~sganzfri/HW1_AI.pdf
  - http://ai.berkeley.edu/search.html
- Office hours

- https://www.cs.cmu.edu/~sganzfri/Calendar_AI.docx
- Midterm exam: on 10/19
- Final exam pushed back (likely on 12/12 instead of 12/5)
- Extra lecture on NLP on 11/16
- Will likely only cover 1-1.5 lectures on logic and on planning
- Only 4 homework assignments instead of 5

Heuristic functions

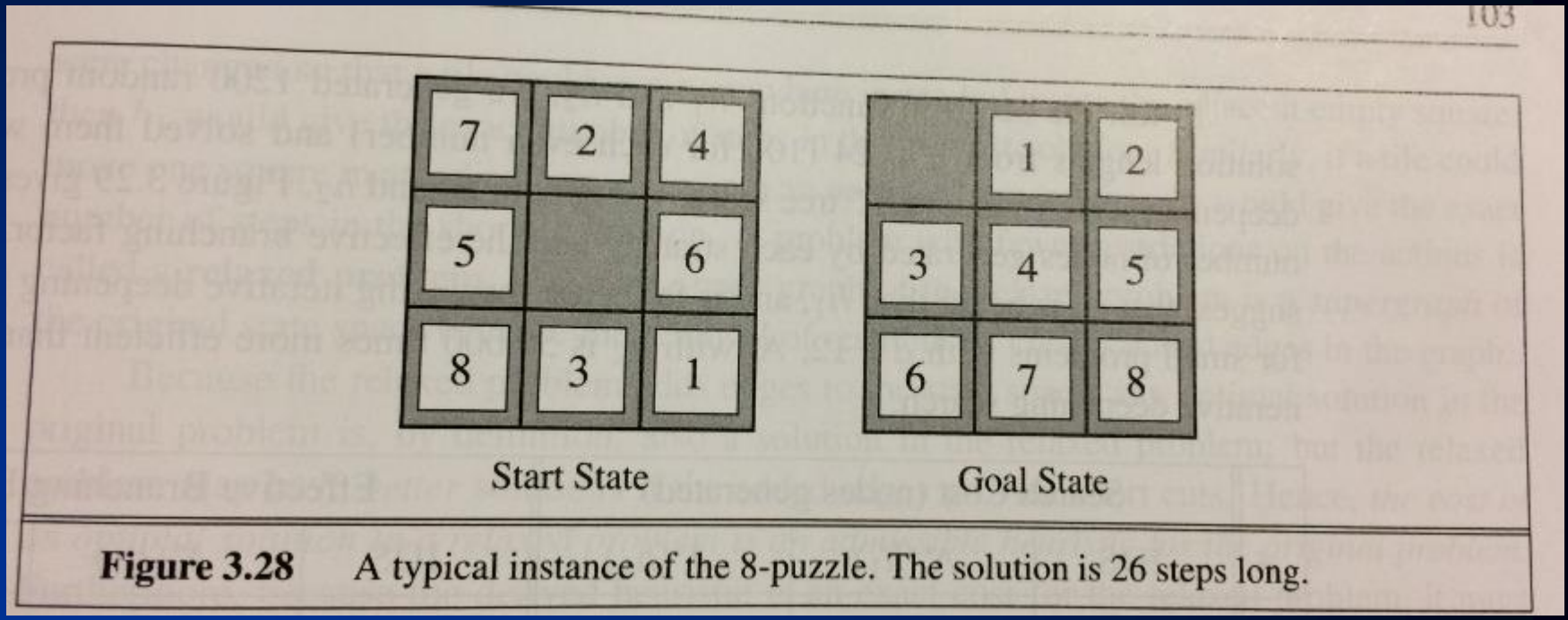

Figure 3.28 A typical instance of the 8-puzzle. The solution is 26 steps long.

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 vs. classical search

- The search algorithms that we have seen so far are designed to explore search spaces systematically. This systematicity is achieved by keeping one or more paths in memory and by recording which alternatives have been explored at each point along the path. When a goal is found, the *path* to that goal also constitutes a *solution* to the problem. In many problems, however, the path to the goal is irrelevant. For example, in the 8-queens problem, what matters is the final configuration of queens, not the order in which they are added. The same general property holds for many important applications such as integrated circuit design, factory-floor layout, job-shop scheduling, automatic programming, telecommunications network optimization, vehicle routing, and portfolio management.

Local search

- If the path to the goal does not matter, we might consider a different class of algorithms, ones that do not worry about paths at all. **Local search** algorithms operate using a single **current node** (rather than multiple paths) and generally move only to neighbors of that node. Typically, the paths followed by the search are not retained. Although local search algorithms are not systematic, they have two key advantages:
  - 1) They use very little memory—usually a constant amount
  - And 2) they can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.

Local search

- In addition to finding goals, local search algorithms are useful for solving pure **optimization problems** (more on this in upcoming lectures), in which the aim is to find the best state according to an **objective function**. Many optimization problems do not fit the "standard" search model introduced before. For example, nature provides an objective function—reproductive fitness—that Darwinian evolution could be seen as attempting to optimize, but there is no "goal test" and no "path cost" for this problem.

Local search

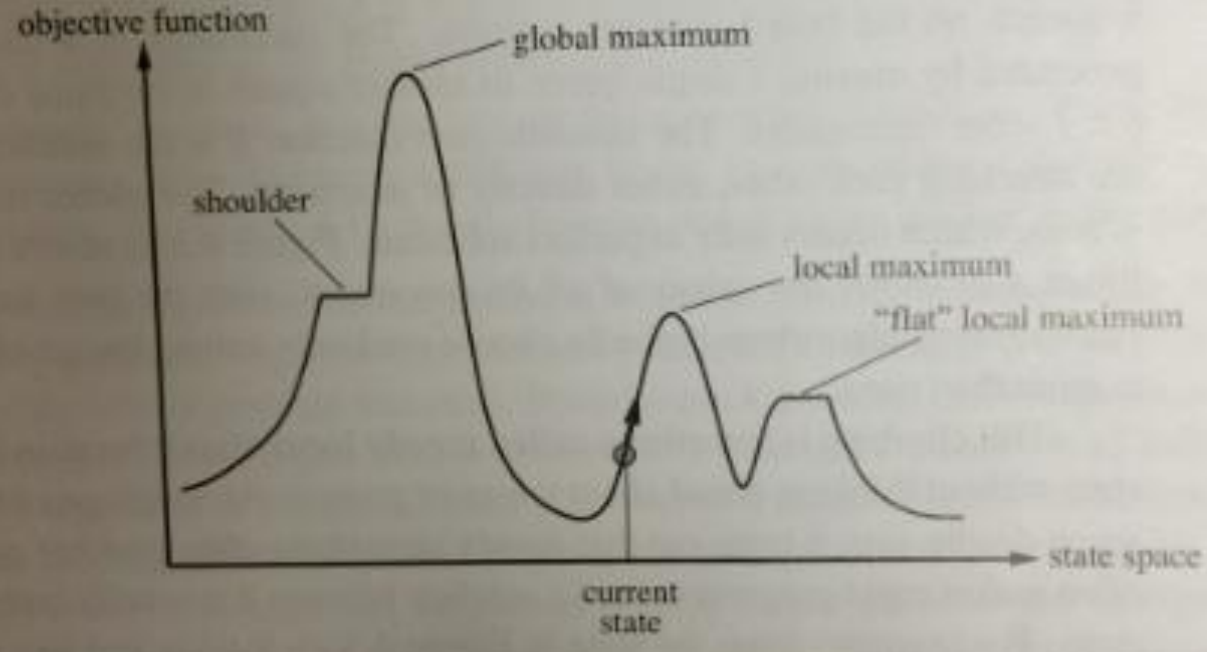

Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

Local search

- To understand local search, we find it useful to consider the **state-space landscape**. A landscape has both “location” (defined by the state) and “elevation” (defined by the value of the heuristic cost function or objective function). 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**. (You can convert between them by inserting a minus sign). Local search algorithms explore this landscape. A **complete** local search algorithm always finds a goal if one exists; an **optimal** algorithm always finds a global minimum/maximum

Hill-climbing search

- The **hill-climbing** search algorithm (**steepest-ascent** version) is simply a loop that continually moves in the direction of increasing value—that is, uphill. It terminates when it reaches a “peak” where no neighbor has a higher value. The algorithm does not maintain a search tree, so the data structure for the current node need only record the state and the value of the objective function. Hill climbing does not look ahead beyond the immediate neighbors of the current state. This resembles trying to find the top of Mount Everest in a thick fog while suffering from amnesia.

8-queens

(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.

Hill-climbing search

```
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.

Hill-climbing search

- To illustrate hill climbing, we will use the **8-queens problem** introduced earlier. Local search algorithms typically use a **complete-state formulation**, where each state has 8 queens on the board, one per column. The successors of a state are all possible states generated by moving a single queen to another square in the same column (so each state has 8x7=56 successors) The heuristic cost function h is the number of pairs of queens that are attacking each other, either directly or indirectly. The global minimum of this function is zero, which occurs only at perfect solutions.

Hill climbing

- The figure shows a state with h = 17. The figure also shows the values of all its successors, with the best figure also shows the values of all its successors, with the best successors having h = 12. Hill-climbing algorithms typically choose randomly among the set of best successors if there is more than one.

Hill climbing

- Hill climbing is sometimes called **greedy local search** because it grabs a good neighbor state without thinking ahead about where to go next. Although greed is considered one of the seven deadly sins, it turns out that greedy algorithms often perform quite well. Hill climbing often makes rapid progress toward a solution because it is usually quite easy to improve a bad state. For example, from the 8-queens game state, it takes just five steps to reach the right state which has h=1 and is very nearly a solution.

Hill climbing can get stuck

- Unfortunately, hill climbing often gets stuck for the following reasons:
  - **Local maxima**: a local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upward toward the peak but will then be stuck with nowhere else to go. More concretely, the right figure for the 8-queens is a local maximum (i.e., a local minimum for the cost h); every move of a single queen makes the situation worse.

Local search

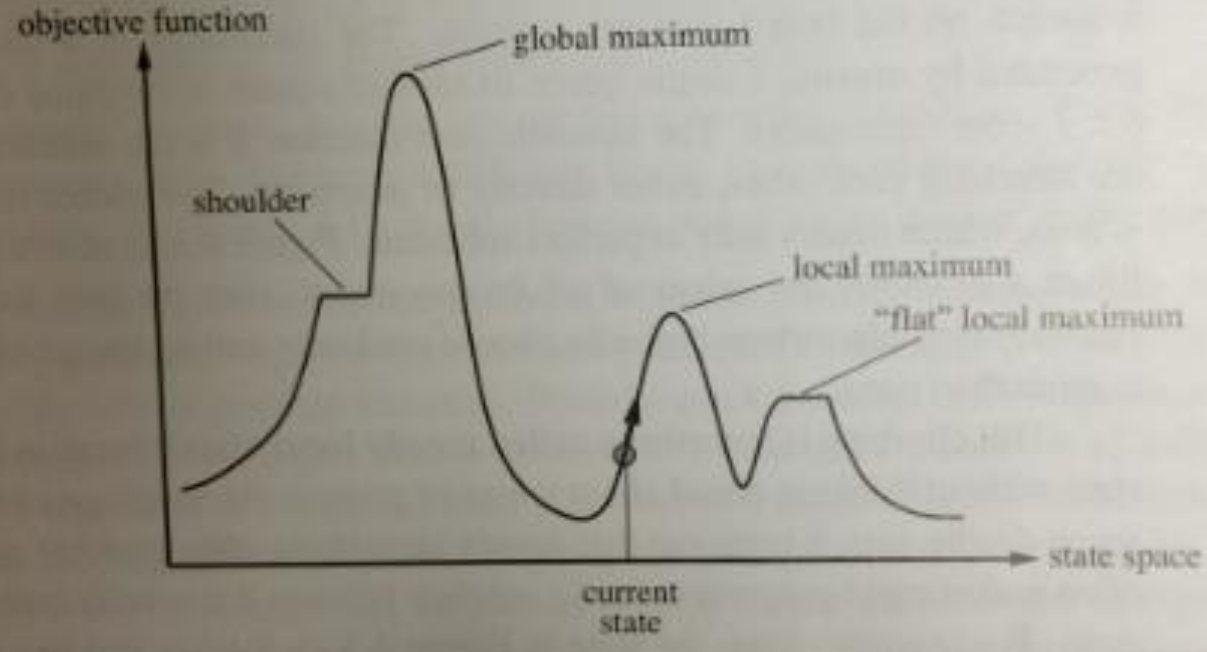

Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

Hill climbing can get stuck

- Unfortunately, hill climbing often gets stuck for the following reasons:
  - **Local maxima**: a local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upward toward the peak but will then be stuck with nowhere else to go.
  - **Ridges**: a ridge is shown in next figure. 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 exist exists, or a **shoulder**, from which progress is possible. A hill-climbing search might get lost on the plateau.

Hill climbing ridge

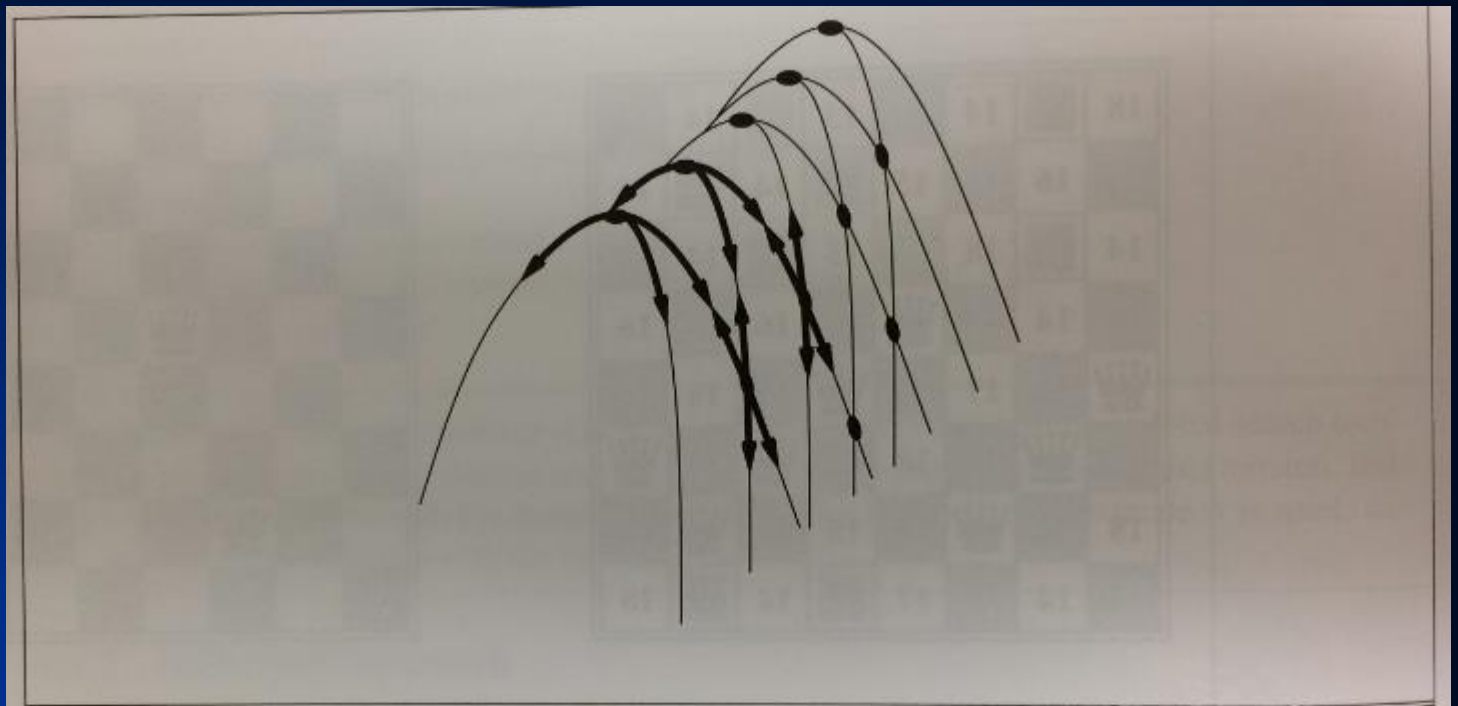

Figure 4.4 Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill.

Hill climbing

- In each case, the algorithm reaches a point at which no progress is being made. Starting from a randomly generated 8-queens state, steepest-ascent hill climbing gets stuck 86% of the time, solving only 14% of problem instances. It works quickly, taking just 4 steps on average when it succeeds and 3 when it gets stuck—not bad for a state space with 8^8 ~= 17 million states.

Hill climbing

- The algorithm halts if it reaches a plateau where the best successor has the same value as the current state. Might it not be a good idea to keep going—to allow a **sideways move** in the hope that the plateau is really a “shoulder?”

Hill climbing

- The answer is usually yes, but we must take care. If we always allow sideways moves when there are no uphill moves, an infinite loop will occur whenever the algorithm reaches a flat local maximum that is not a shoulder. One common solution is to put a limit on the number of consecutive sideways moves allowed. For example, we could allow up to, say, 100 consecutive sideways moves in the 8-queens problem. This raises the percentage of problem instances solved by hill climbing from 14% to 94%. Success comes at a cost: the algorithm averages roughly 21 steps for each successful instance and 64 for each failure.

Hill climbing

- Many variants of hill climbing have been invented.
- **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. This is a good strategy when a state has many (e.g., thousands) of successors).

Hill climbing

- The hill-climbing algorithms described so far are incomplete—they often fail to find a goal when one exists because they can get stuck on local maximum.
- **Random-restart hill climbing** adopts the well-known adage, “If at first you don’t succeed, try, try again.” It conducts a series of hill-climbing searches from randomly generated initial states, until a goal is found. It is trivially complete with probability approaching 1, because it will eventually generate a goal state as the initial state.

Random-restart hill climbing

- If each hill-climbing search has a probability p of success, then the expected number of restarts required is 1/p. For 8-queens instances with no sideways moves allowed, p~=0.14, so we need roughly 7 iterations to find a goal (6 failures and 1 success). The expected number of steps is the cost of one successful iteration plus (1-p)/p times the cost of failure, or roughly 22 steps in all. When we allow sideways moves, 1/0.94 ~= 1.06 iterations are needed on average and (1x21)+(0.06/0.94)x64~=25 steps. For 8-queens, then, random-restart hill climbing is very effective indeed. Even for three million queens, the approach can find solutions in under a minute.

Random-restart hill climbing

- The success of hill climbing depends very much on the shape of the state-space landscape: if there are few local maxima and plateau, random-restart hill climbing will find a good solution very quickly. On the other hand, many real problems have a landscape that looks more like a widely scattered family of balding porcupines on a flat floor, with miniature porcupines living on the tip of each porcupine needle, *ad infinitum*. NP-hard problems typically have an exponential number of local maxima to get stuck on. Despite this, a reasonably good local maximum can often be found after a small number of restarts.

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. Therefore, it seems reasonable to try to combine hill climbing with a random walk in some way that yields both efficiency and completeness.

Simulated annealing

- **Simulated annealing** is such an algorithm. In metallurgy, **annealing** is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to reach a low-energy crystalline state. To explain simulated annealing, we stich our point of view from hill climbing to **gradient descent** (minimizing cost) and imagine the task of getting a ping-pong ball into the deepest crevice in a bumpy surface. If we just let the ball roll, it will come to rest at a local minimum. If we shake the surface, we can bounce the ball out of the local minimum. The trick is to shake just hard enough to bounce the ball out of the LM but not hard enough to dislodge it from the global minimum. The SA solution is to start by shaking hard (i.e., high temperature) and then gradually reduce the intensity.

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.

Simulated annealing

- The innermost loop of the SA algorithm is quite similar to hill climbing. Instead of picking the *best* move, however, it picks a *random* move. If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1. The probability decreases exponentially with the "badness" of the move—the amount DE by which the evaluation is worsened. The probability also decreases as the "temperature" T goes down: "bad" moves are more likely to be allowed a the start when T is high, and they become more unlikely as T decreases. If the *schedule* lowers T slowly enough, the algorithm will find a global optimum with probability approaching 1.

Simulated annealing

- Simulated annealing was first used extensively to solve VLSI layout problems (creating integrated circuit by combining thousands of transistors into a single chip) in the early 1980s. It has been applied widely to factory scheduling and other large-scale optimization tasks.
- Homework exercise (for HW2): compare performance of simulated annealing to that of random-restart hill climbing on the 8-queens puzzle.

Local beam search

- Keeping just one node in memory might seem to be an extreme reaction to the problem of memory limitations. The **local beam search** algorithm keeps track of k states rather than just 1. It begins with k randomly generated states. At each step, all successors of all k states are generated. If any one is a goal, the algorithm halts. Otherwise, it selects the k best successors from the complete list and repeats.

Stochastic beam search

- Improves on normal beam search, analogously to stochastic hill climbing. Instead of choosing best k from the pool of candidate successors, SBS chooses k successors at random, with the probability of choosing a given successor being an increasing function of its value. SBS bears some resemblance to the problem of natural selection, whereby the “successors” (offspring) of a “state” (organism) populate the next generation according to its “value” (fitness).

Genetic algorithms

- Variant of stochastic beam search in which successor states are generated by combining *two* parent states rather than by modifying a single state. The analogy to natural selection is the same as in stochastic beam search, except that now we are dealing with sexual rather than asexual reproduction.

Genetic algorithms

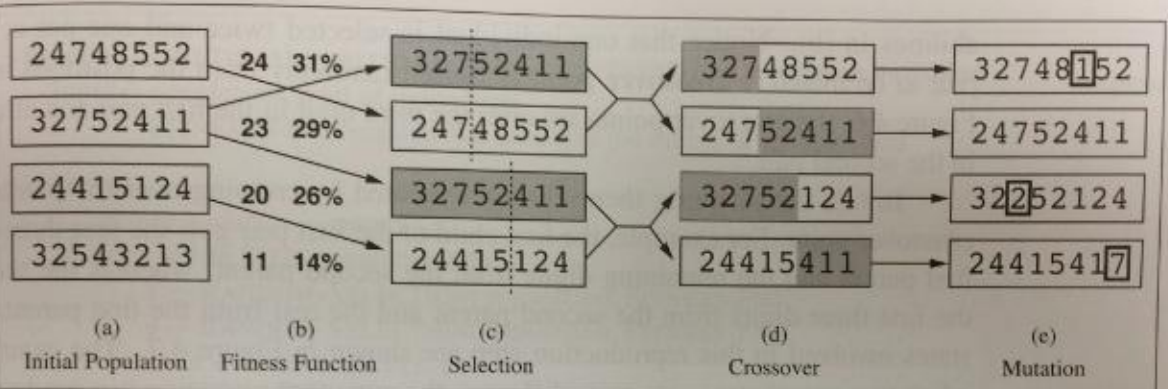

Figure 4.6 The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

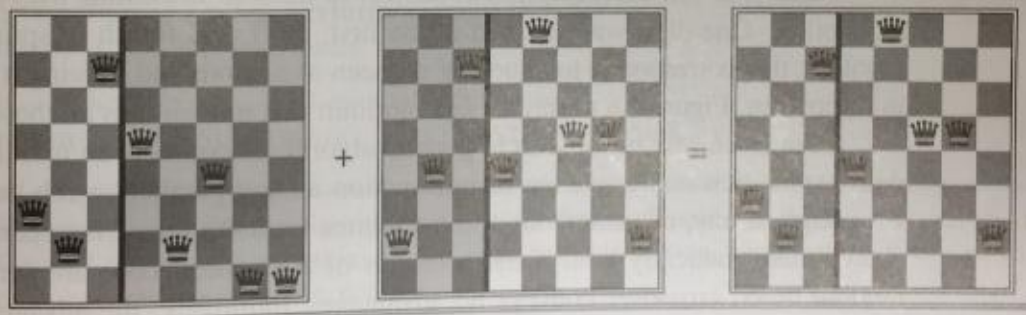

Figure 4.7 The 8-queens states corresponding to the first two parents in Figure 4.6(c) and the first offspring in Figure 4.6(d). The shaded columns are lost in the crossover step and the unshaded columns are retained.

Genetic algorithms

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
    new_population ← empty set
    for i = 1 to SIZE(population) do
      x ← RANDOM-SELECTION(population, FITNESS-FN)
      y ← RANDOM-SELECTION(population, FITNESS-FN)
      child ← REPRODUCE(x, y)
      if (small random probability) then child ← MUTATE(child)
      add child to new_population
    population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN

function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

Figure 4.8 A genetic algorithm. The algorithm is the same as the one diagrammed in Figure 4.6, with one variation: in this more popular version, each mating of two parents produces only one offspring, not two.

Homework for next class

- Chapter 7 from Russell-Norvig textbook.
- HW1: out 9/5 due 9/28