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
  - 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: ECS 254 today after lecture

- 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

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

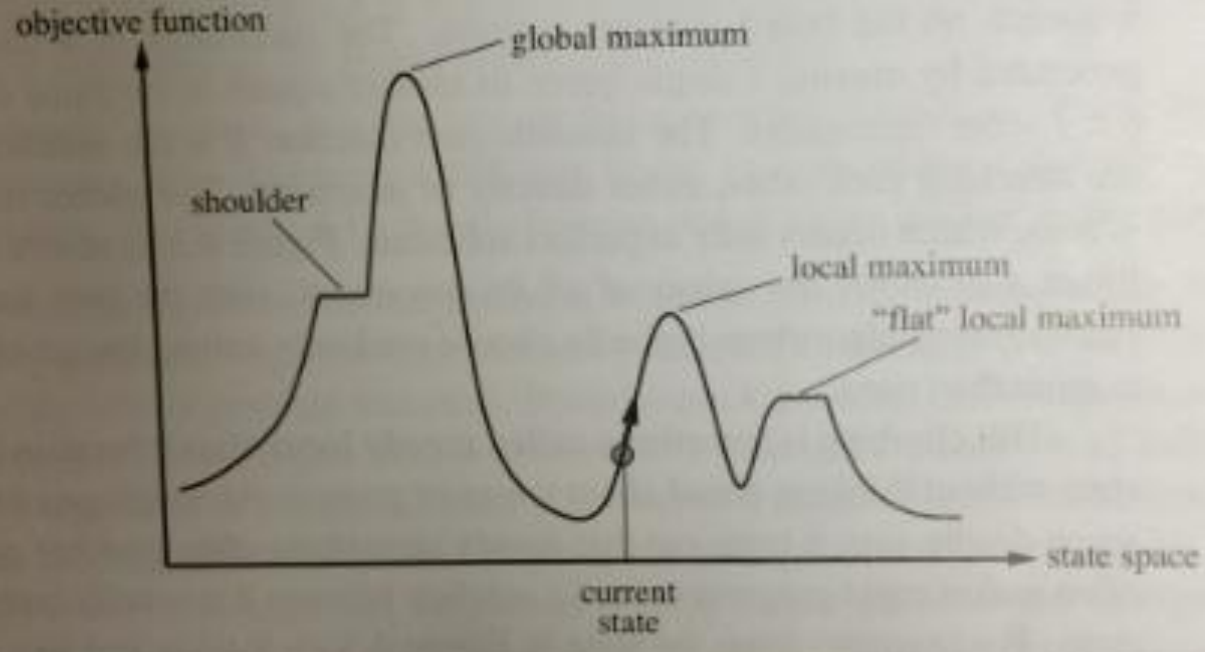


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

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

- 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

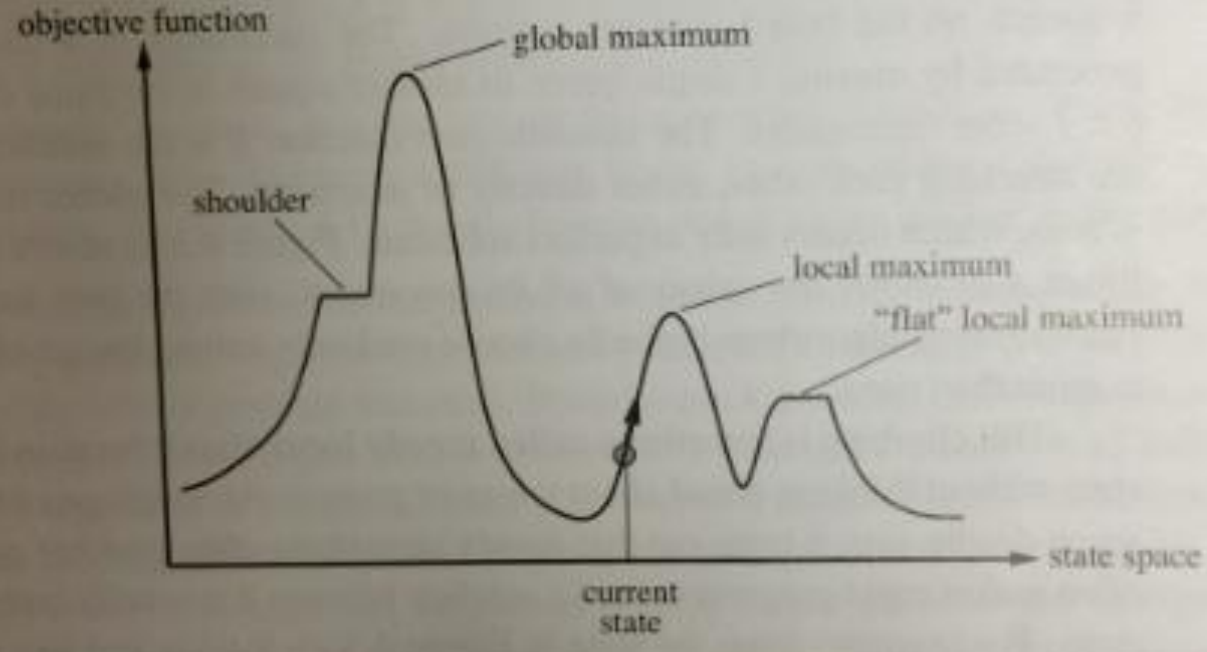


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

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

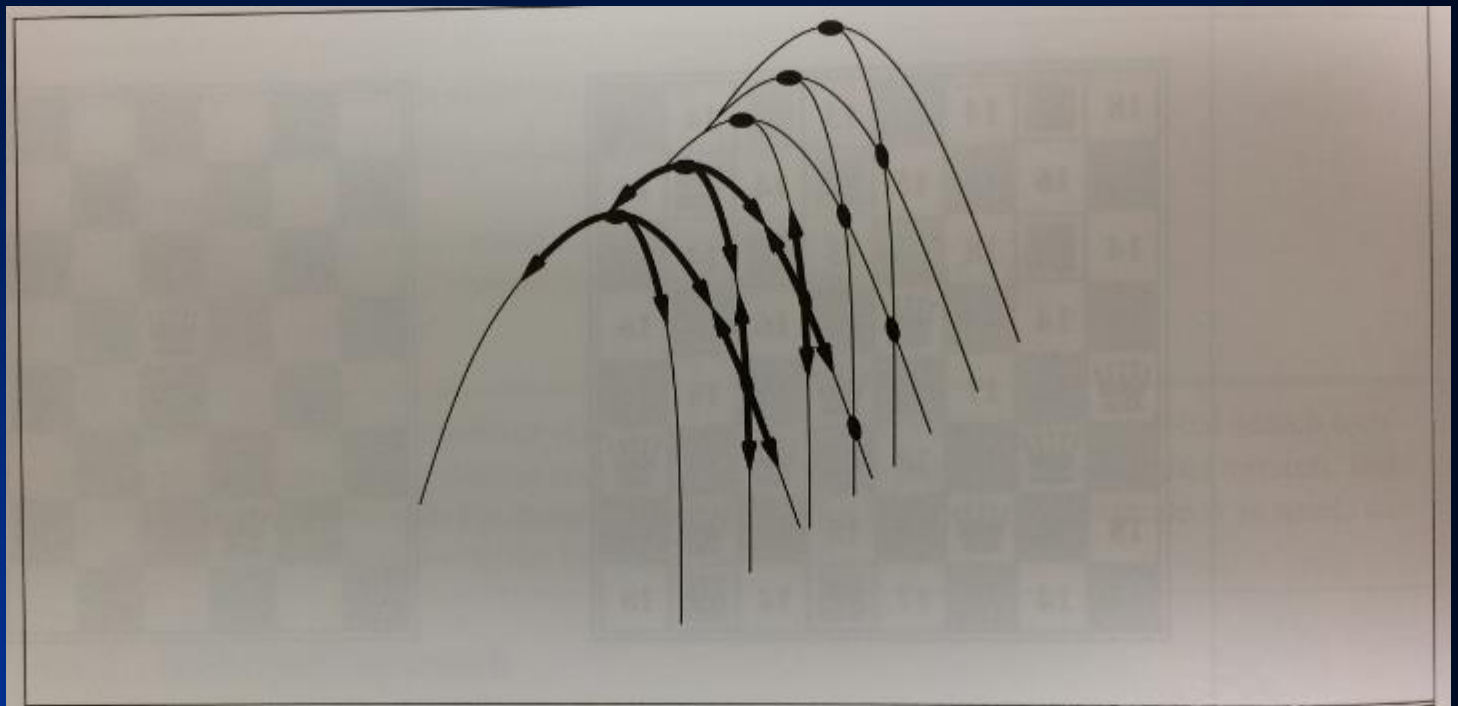


Figure 4.4 Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point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

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.

Local beam search

- At first sight, a local beam search with k states might seem to be nothing more than running k random restarts in parallel instead of in sequence. In fact, the two algorithms are quite different. In a random-restart search, each search process runs independently of the others. *In a local beam search, useful information is passed among the parallel search threads*. In effect, the states that generate the best successors say to the others, “Come over here, the grass is greener!” The algorithm quickly abandons unfruitful searches and moves its resources to where toe most progress is being made.

Local beam search

- In its simplest form, local beam search can suffer from a lack of diversity among the k states—they can quickly become concentrated in a small region of the state space, making the search little more than an expensive version of hill climbing. A variant called **stochastic beam search**, analogous to stochastic hill climbing, helps alleviate this problem. Instead of choosing the 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

- A **genetic algorithm** (**GA**) is a variant of stochastic beam search in which successor states are generated by combining *two* parent states rather than by modifying a single state.
- Like beam searches, GAs begin with a set of k randomly generated states, called the **population**. Each state, or **individual**, is represented a s a string over a finite alphabet—most commonly, a string of 0s and 1s. For example, an 8-queens state must specify the positions of 8 queens, each in a column of 8 squares, and so requires 8 x log(8) = 24 bits. Alternatively, the state could be represented as 8 digits, each in the range from 1 to 8. The figure shows a population of four 8-digit strings representing 8-queens states.

Genetic algorithms

- The production of the next generation of states is shown in Figure 4.6b-e. In (b), each state is rated by the objective function, or (in GA terminology) the **fitness function**. A fitness function should return higher values for better states, so, for the 8-queens problem we use the number of *nonattacking* pairs of queens, which has a value of 28 for a solution. The values of the four states are 24, 23, 20, and 11. In this particular variant of the genetic algorithm, the probability of being chosen for reproducing is directly proportional to the fitness score, and the percentages are shown next to the raw scores.

Genetic algorithms

- In (c), two pairs are selected at random for reproduction, in accordance with the probabilities in (b). Notice that one individual is selected twice and one not at all. For each pair to be mated, a **crossover** point is chosen randomly from the positions in the string. In the figure, the crossover points are after the third digit in the first pair and after the fifth digit in the second pair.
- In (d), the offspring themselves are created by crossing over the parent strings at the crossover point. For example, the first child of the first pair gets the first three digits from the first parent and the remaining digits from the second parent, whereas the second child gets the first three digits from the second parent and the rest from the first parent. The 8-queens states involved in the reproduction step are shown in Figure 4.7.

Genetic algorithms

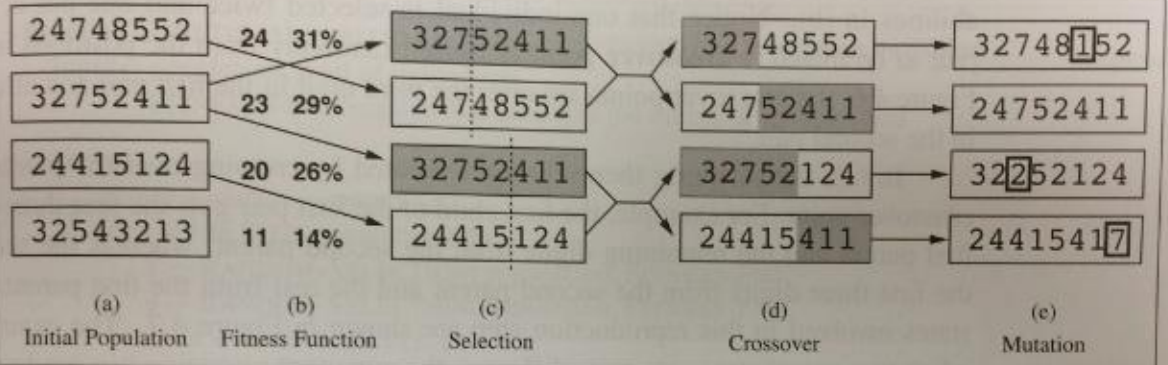


Figure 4.6 The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

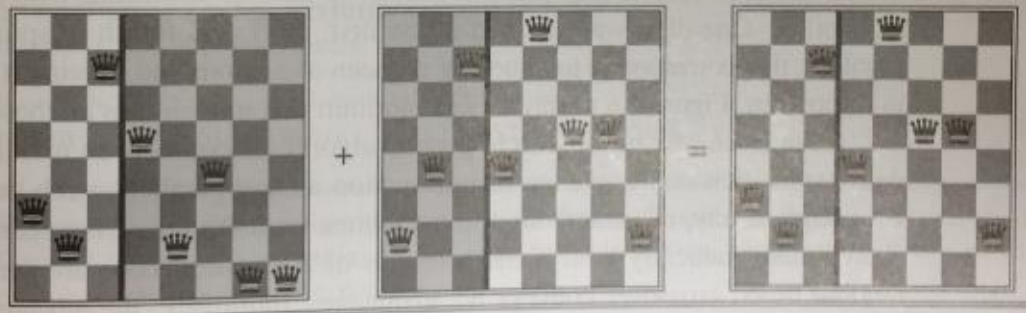


Figure 4.7 The 8-queens states corresponding to the first two parents in Figure 4.6(c) and the first offspring in Figure 4.6(d). The shaded columns are lost in the crossover step and the unshaded columns are retained.

Genetic algorithms

- The example shows that when two parent states are quite different, the crossover operation can produce a state that is a long way from either parent state. It is often the case that the population is quite diverse early on in the process, so crossover (like simulated annealing) frequently takes large steps in the state space early in the search process and smaller steps later on when most individuals are quite similar.

Genetic algorithms

- Finally, in (e), each location is subject to random **mutation** with small independent probability. One digit was mutated in the first, third, and fourth offspring. In the 8-queens problem, this corresponds to choosing a queen at random and moving it to a random square in its column.

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.

Genetic algorithms

- Like stochastic beam search, genetic algorithms combine uphill tendency with random exploration and exchange of information among parallel search threads. The primary advantage, if any, of genetic algorithms comes from the crossover operation. Yet it can be shown mathematically that, if the positions of the genetic code are permuted initially in a random order, crossover conveys no advantage. Intuitively, the advantage comes from the ability of crossover to combine large blocks of letters that have evolved independently to perform useful functions, thus raising the level of granularity at which the search operates. For example, it could be that putting the first three queens in positions 2, 4, and 6 (where they do not attack each other) constitutes a useful block that can be combined with other blocks to construct a solution.

Genetic algorithms

- In practice, genetic algorithms have had a widespread impact on optimization problems, such as circuit layout and job-shop scheduling. At present, it is not clear whether the appeal of genetic algorithms arises from their performance or from their aesthetically pleasing origins in the theory of evolution. Much work remains to be done to identify the conditions under which genetic algorithms perform well.

Local search for continuous spaces

- Earlier we explained the distinction between discrete and continuous environments, pointing out that most real-world environments are continuous. Yet none of the algorithms we have described (except for first-choice hill climbing and simulated annealing) can handle continuous state and action spaces, because they have infinite branching factors. There exist other local search techniques for finding optimal solutions in continuous spaces. Many of the basic techniques originated in the 17th century after the development of calculus by Newton and Leibniz. We find use for these techniques at several places, including for learning, vision, and robotics.

Local search

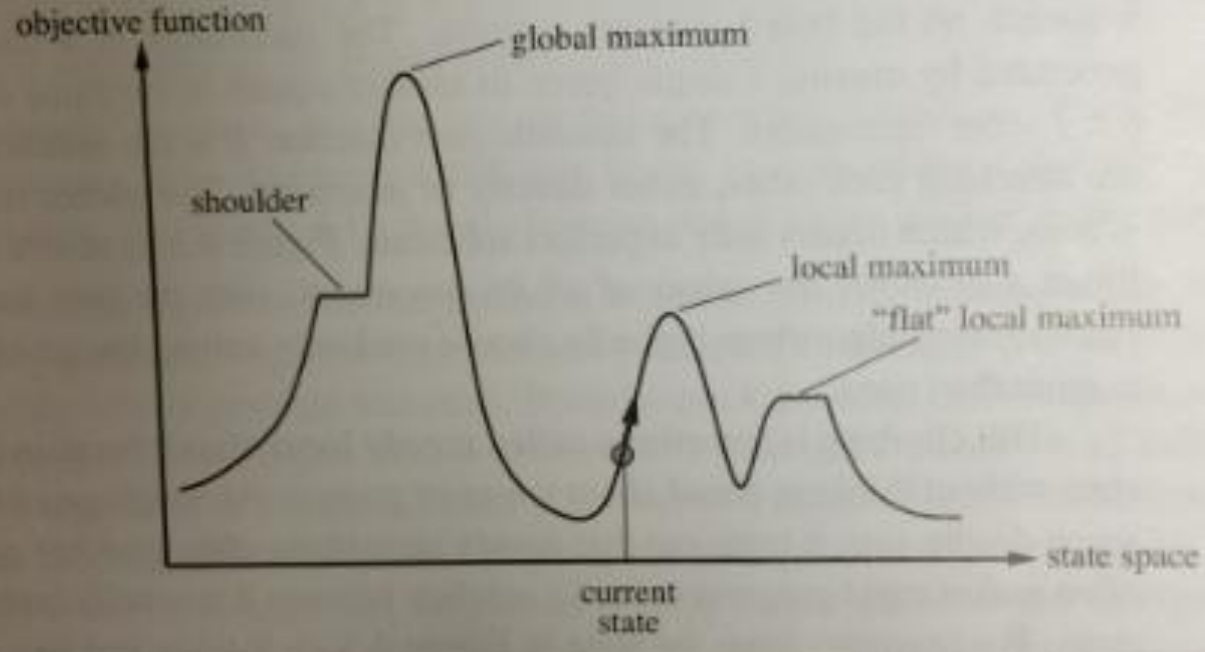


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

Local search for continuous spaces

- Suppose we want to place three new airports anywhere in Romania, such that the sum of squared distances form each city on the map to its nearest airport is minimized. The state space is then defined by the coordinates of the airports: (x1,y1), (x2,y2), and (x3,y3). This is a *six-dimensional* space; we also say that the states are defined by six **variables**. Moving around in this space corresponds to moving one or more of the airports on the map. The objective function f(x1,y1,x2,y2,x3,y3) is relatively easy to compute for any particular state once we compute the closest cities.

Local search for continuous spaces

- Let Ci be the set of cities whose closest airport (in the current state) is airport i. Then, *in the neighborhood of the current state*, where the Ci's remain constant, we have
- $f(x1,y1,x2,y2,x3,y3) = \sum_{i=1}^{3} \sum_{c \text{ in } Ci} (xi-xc)^2 + (yi-yc)^2$
- This expression is correct *locally*, but not globally because the sets Ci are (discontinuous) functions of the state.
- One way to avoid continuous problems is simply to **discretize** the neighborhood of each state. For example, we can move only one airport at a time in either the x or y direction by a fixed amount +- K. With 6 variables, this gives 12 possible successors for each state. We can then apply any of the local search algorithms described previously. We could also apply stochastic hill climbing and simulated annealing directly, without discretizing the space. These algorithms choose successors randomly, which can be done by generating random vectors of length K.

Local search for continuous spaces

- Many methods attempt to use the **gradient** of the landscape to find a maximum. The gradient of the objective function is a vector Df that gives the magnitude and direction of the steepest slope (vector of the derivatives of the objective with respect to each variable).
- In some cases we can find a maximum by solving the equation Df = 0 (this could be done, for example, if we were placing just one airport; the solution is the arithmetic mean of all the cities' coordinates). In many cases, however, this equation cannot be solved in closed form. For example, with three airports, the expression for the gradient depends on what cities are closest to each airport in the current state. This means we can compute the gradient *locally* (but not *globally*). Given a locally correct expression for the gradient, we can perform steepest-ascent hill climbing by updating the current state according to $x \leftarrow x + a\, Df(x)$.

Local search extensions

- More advanced approaches for continuous spaces: empirical gradient, line search, Newton-Raphson method, Hessian matrix, etc. We will see some of these in the optimization module of the class.
- Can also apply local search for nondeterministic actions (And-Or search trees), for partial observation (belief-state search), and for online search in real-time for unknown environments (all of the algorithms we have seen produce agents for **offline** search).

Local search wrap-up

- Local search methods such as **hill climbing** operate on complete-state formulations, keeping only a small number of nodes in memory. Several stochastic algorithms have been developed, including **simulated annealing**, which returns optimal solutions when given an appropriate cooling schedule.
- Many local search methods apply also to problems in continuous spaces. **Linear programming** and **convex optimization** problems obey certain restrictions on the shape of the state space and the nature of the objective function, and admit polynomial-time algorithms that are often extremely efficient in practice.
- A **genetic algorithm** is a stochastic hill-climbing search in which a large population of states is maintained. New states are generated by **mutation** and **crossover**, which combines pairs of states from the population.

Adversarial search

- We first consider games with two players, whom we call MAX and MIN. MAX moves first, and then they take turns moving until the game is over. At the end of the game, points are awarded to the winning player, and penalties given to the loser. A game can be formally defined as a kind of search problem with the following elements:

Search problem definition

- States
- Initial state
- Actions
- Transition model
- Goal test
- Path cost

Definition for 8-queens problem

- **States**: Any arrangement of 0 to 8 queens on the board is a state.
- **Initial state**: No queens on the board.
- **Actions**: Add a queen to any empty square.
- **Transition model**: Returns the board with a queen added to the specified square
- **Goal test**: 8 queens are on the board, none attacked
- **Path cost**: (Not applicable)

Game definition

- S_0: the **initial state**, which specifies how the game starts
- PLAYER(s): defines which player has the move in a state
- ACTIONS(s): Returns the set of legal moves in a state
- RESULT(s,a): The **transition model**, which defines the result of a move.
- TERMINAL-TEST(s): A **terminal** test, which is true when the game is over and false otherwise. States where the game has ended are called **terminal states**.
- UTILITY(s,p): A **utility function** (also called an objective function or payoff function), defines the final numeric value for a game that ends in terminal state s for a player p. In chess, the outcome is a win, loss, or draw, with values +1, 0, or ½. Some games have a wider variety of possible outcomes; the payoffs in backgammon range from 0 to +192.

Zero-sum games

- A **zero-sum** game is (confusingly) defined as one where the total payoff to all players is the same for every instance of the game.
- Is chess zero-sum?
- Checkers?
- Poker?

Zero-sum games

- Chess is zero-sum because every game has payoff of either 0 +1, 1+0, or ½ + ½
- "Constant-sum" would have been a better term, but zero-sum is traditional and makes sense if you imagine that each player is charged an entry fee of ½.

Game tree

- The initial state, ACTIONS function, and RESULT function define the **game tree** for the game—a tree where the nodes are game states and the edges are moves. The figure shows part of the game tree for tic-tac-toe. From the initial state, MAX has nine possible moves. Play alternates between MAX's placing an X and MIN's placing an O until we reach leaf nodes corresponding to terminal states such that one player has three in a row or all the squares are filled. The number on each leaf node indicates the utility value of the terminal state from the point of view of MAX; high values are assumed to be good for MAX and bad for MIN (which is how the players get their names).

Game trees

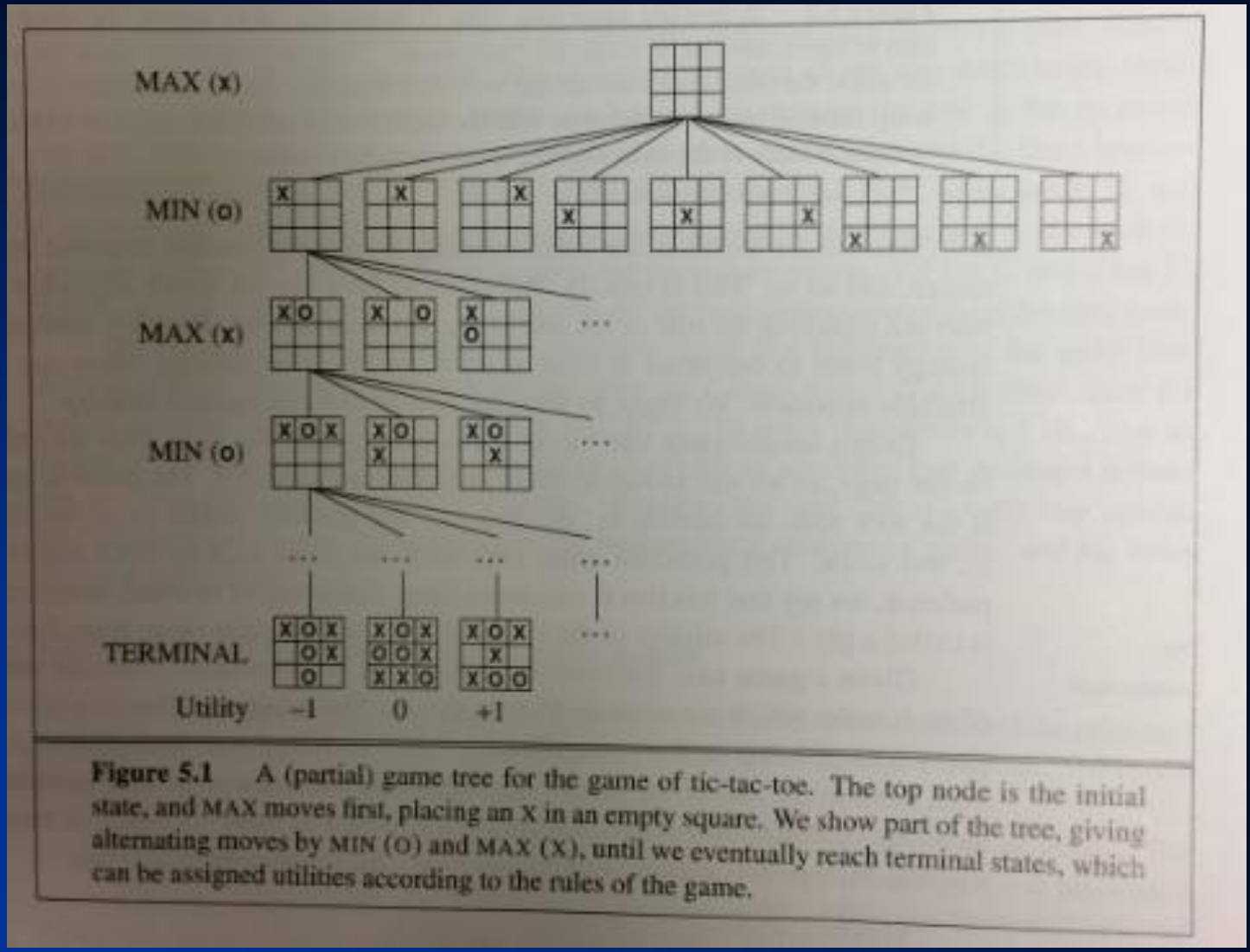


Figure 5.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities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

Game trees

- For tic-tac-toe the game tree is relatively small—fewer than 9! = 362,880 terminal nodes. But for chess there are over 10^40 nodes, so the game tree is best thought of as a theoretical construct that we cannot realize in the physical world. But regardless of the game tree, it is MAX's job to search for a good move. We use the term **search tree** for a tree that is superimposed on the full game tree, and examines enough nodes to allow a player to determine what move to make.

Optimal decisions in games

- In a normal search problem, the optimal solution would be a sequence of actions leading to a goal state—a terminal state that is a win. In adversarial search, MIN has something to say about it. MAX therefore must find a contingent **strategy**, which specifies MAX's move in the initial state, then MAX's moves in the states resulting from every possible response by MIN, then MAX's moves in the states resulting by every possible response by MIN to *those* moves, and so on. Roughly speaking, an optimal strategy leads to outcomes at least as good as any other strategy when one is playing an infallible opponent.

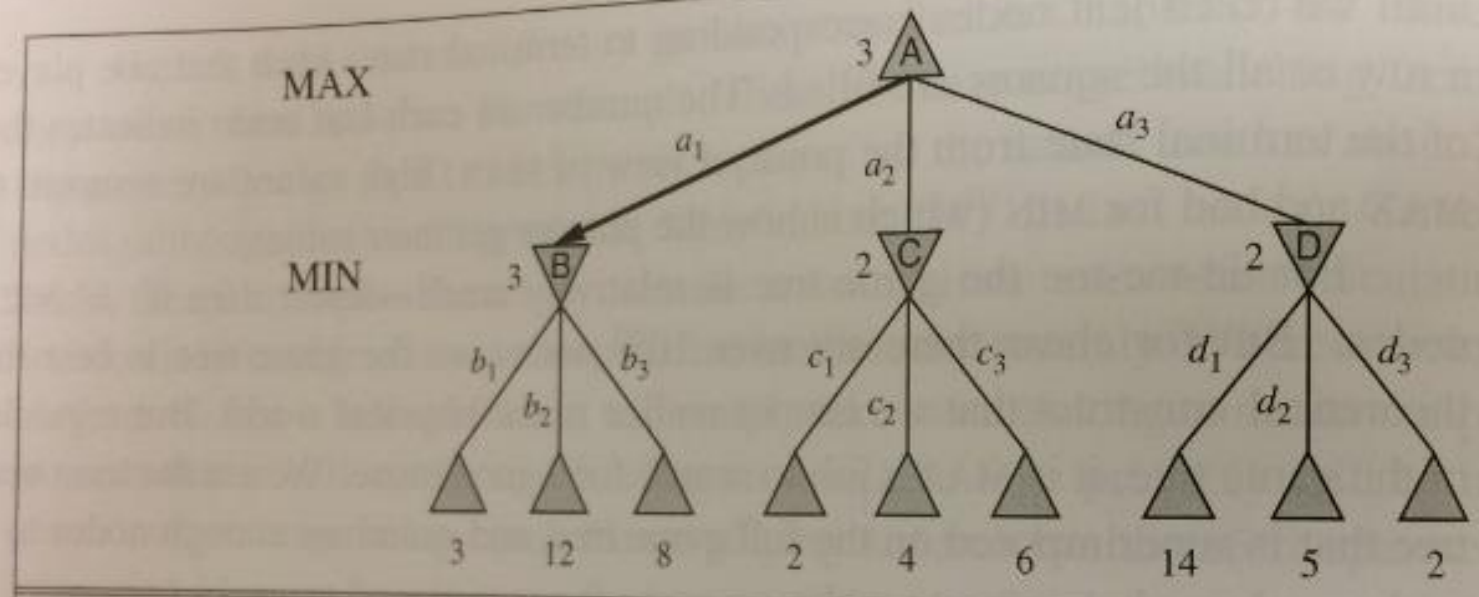
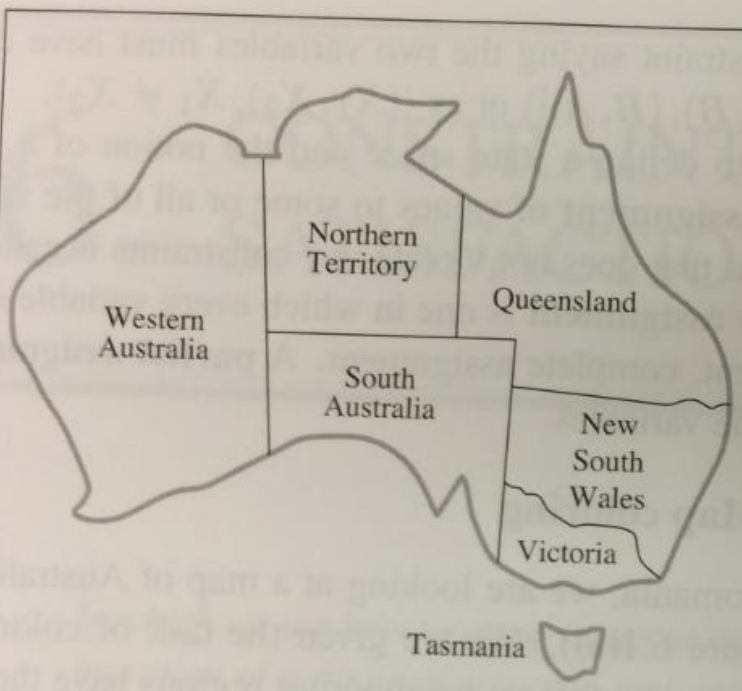


Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

Alternative search paradigms

- 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.

Constraint satisfaction



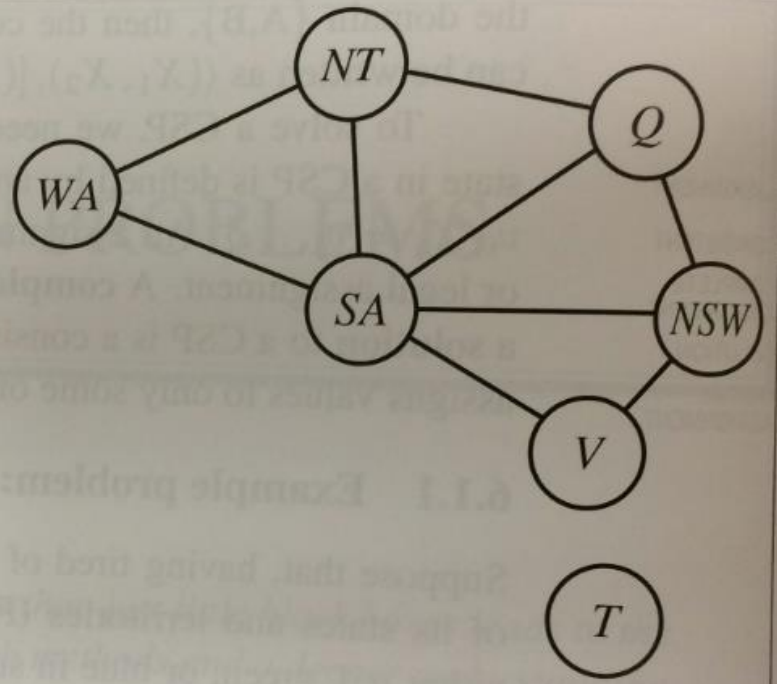


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

immediately discard further

Homework for next class

- Chapter 8 from Russell-Norvig textbook.
- HW1: out 9/5 due 9/28