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 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
  - HW2 will go out next week, due 10/17
  - Midterm on 10/19
- TA office hours:
  - Thursday 3:15-4:15PM, ECS 254

Local search

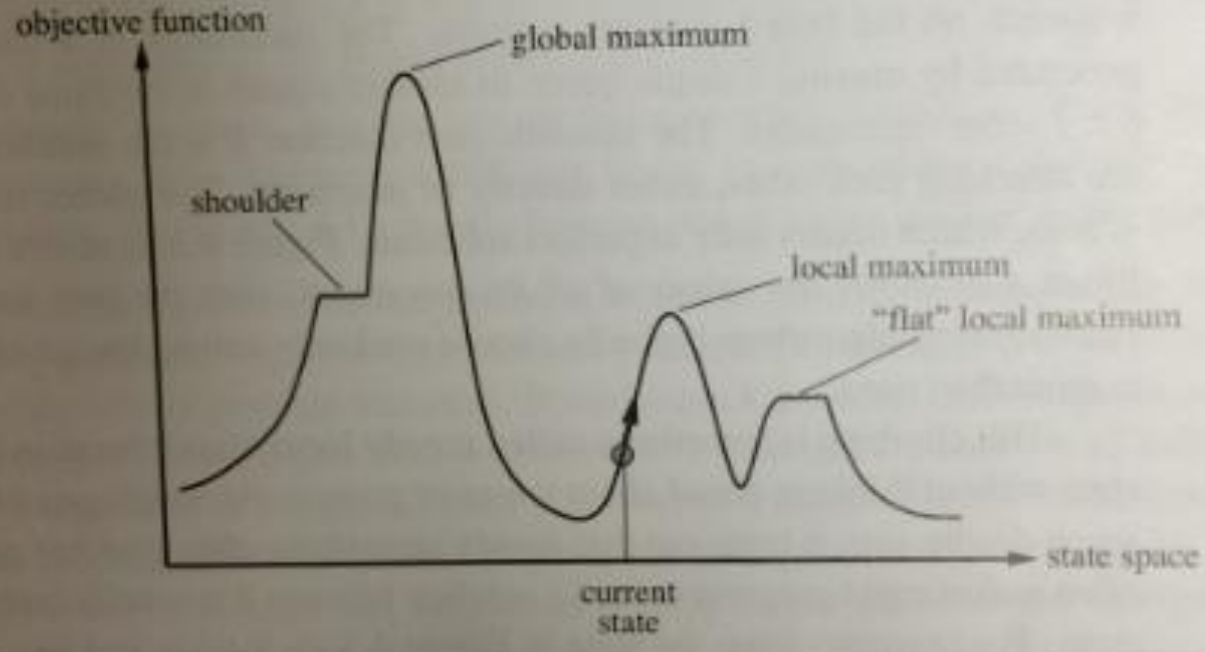


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

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.

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 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

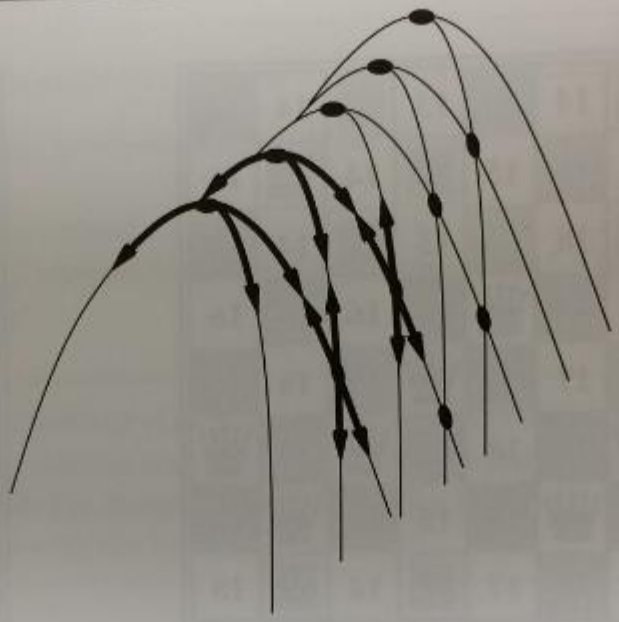


Figure 4.4 Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point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

- 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

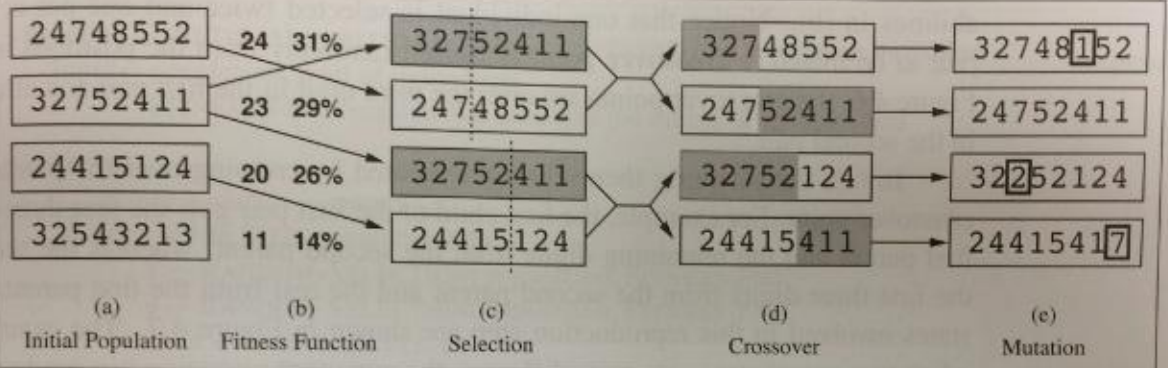


Figure 4.6 The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

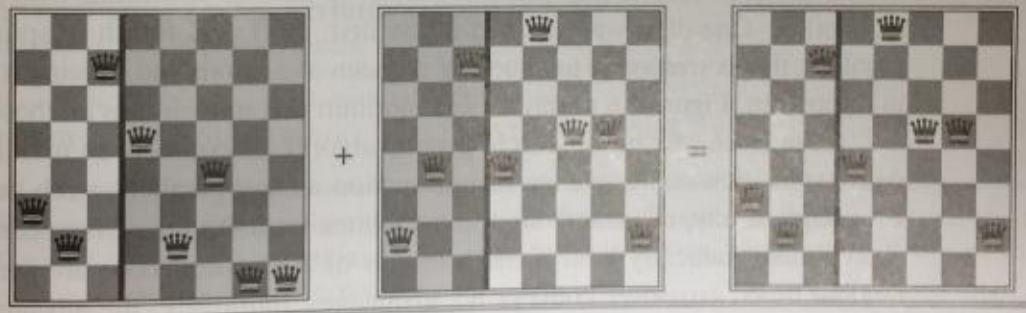


Figure 4.7 The 8-queens states corresponding to the first two parents in Figure 4.6(c) and the first offspring in Figure 4.6(d). The shaded columns are lost in the crossover step and the unshaded columns are retained.

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.

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 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 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

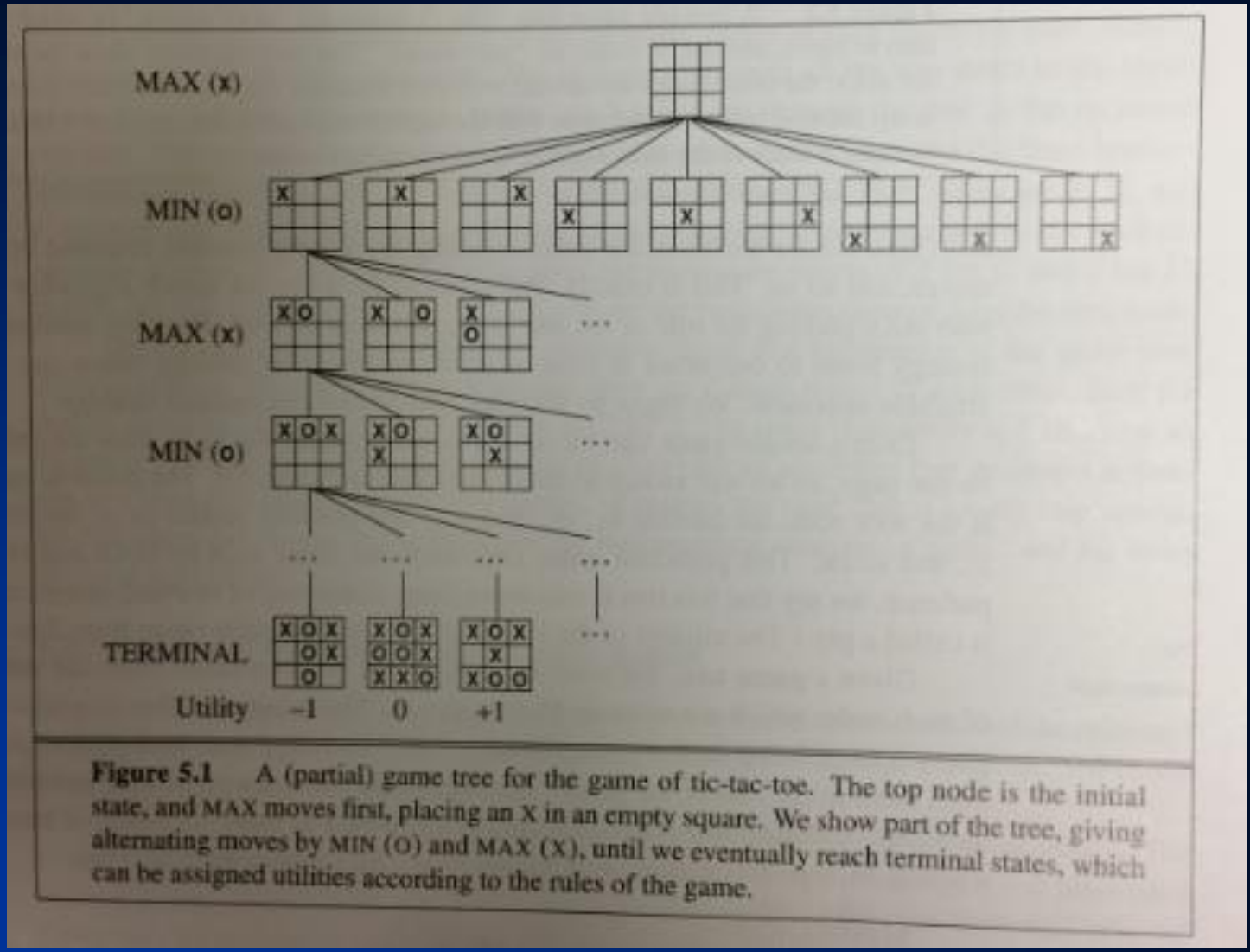


Figure 5.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities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

Game tree

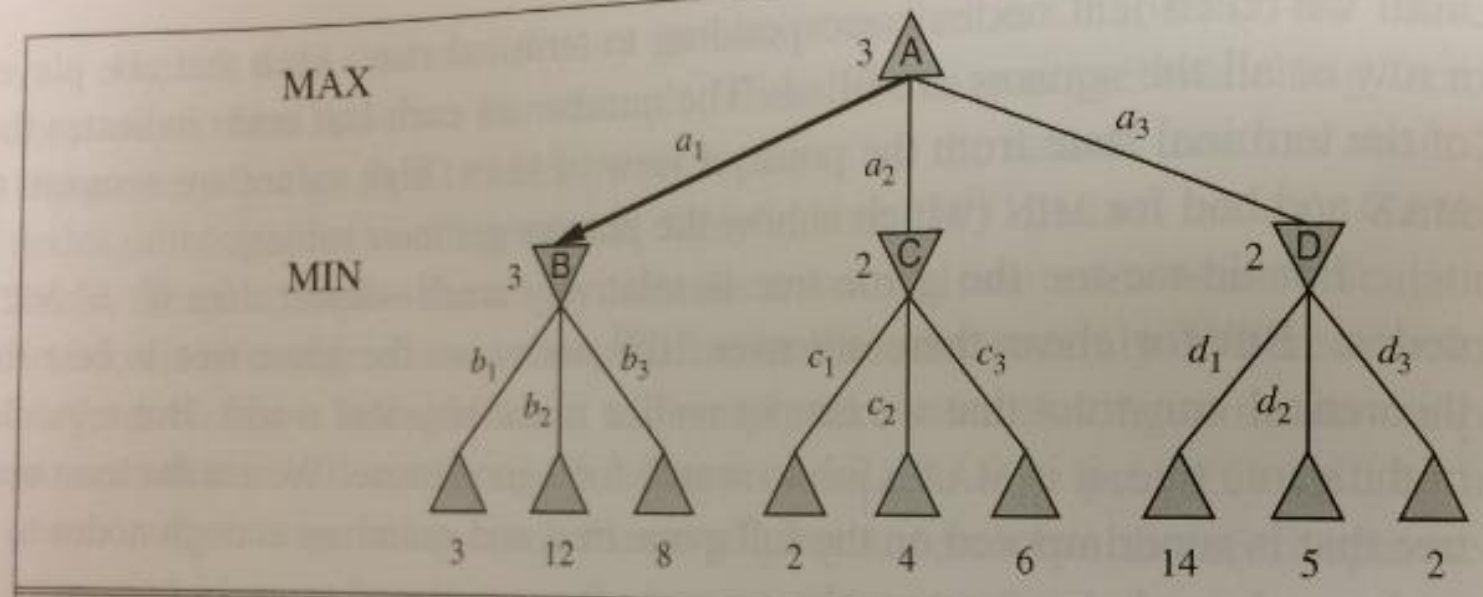


Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state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

Optimal decisions in games

- Even a simple game like tic-tac-toe is too complex for us to draw the entire game tree on one page, so we will instead examine a “trivial” game. The possible moves for MAX at the root node are labeled a1, a2, and a3. The possible replies to a1 for MIN are b1, b2, b3, and so on. This particular game ends after one move each by MAX and MIN. (We say that this tree is one move deep, consisting of two half-moves, each of which is called a **ply**.) The utilities of the terminal states in this game range from 2 to 14.

Optimal decisions in games

- Given a game tree, the optimal strategy can be determined from the **minimax value** of each node, which we write as MINIMAX(n). The minimax value of a node is the utility (for MAX) of being in the corresponding state, *assuming that both players play optimally* from there to the end of the game. Obviously, the minimax value of a terminal state is just its utility. Furthermore, given a choice, MAX prefers to move to a state of maximum value, whereas MIN prefers a state of minimum value. So we have:

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

Optimal decisions in games

- Let us apply these definitions to the game tree considered above. The terminal nodes on the bottom level get their utility values from the game's UTILITY function. The first MIN node, labeled B, has three successor states with values 3, 12, and 8, so its minimax value is 3. Similarly, the other two MIN nodes have minimax value 2. The root node is a MAX node; its successor states have minimax values 3, 2, and 2; so it has a minimax value of 3. We can also identify the **minimax decision** at the root: action a1 is the optimal choice for MAX because it leads to the state with the highest minimax value.

Optimal decisions in games

- This definition of optimal play for MAX assumes that MIN also plays optimally—it maximizes the *worst-case* outcome for MAX. What if MIN does not play optimally? Then it is easy to show (homework exercise) that MAX will do even better. Other strategies against suboptimal opponents may do better than the minimax strategy, but these strategies necessarily do worse against optimal opponents.

The minimax algorithm

- The **minimax algorithm** computes the minimax decision from the current state. It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations. The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are **backed up** through the tree as the recursion unwinds. For example, in the figure the algorithm first recurses down to the three bottom-left nodes and uses the UTILITY function on them to discover that their values are 3, 12, and 8, respectively. Then it takes the minimum of these values, 3, and returns it as the backed-up value of node B. A similar process gives the backed-up values of 2 for C and 2 for D. Finally, we take the maximum of 3, 2, and 2 to get the backed-up value of 3 for the root node.

Minimax algorithm

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

Figure 5.3 An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state. The notation $\text{argmax}_{a \in s} f(a)$ computes the element a of set S that has the maximum value of $f(a)$.

Minimax algorithm

- Does the minimax algorithm resemble any algorithms we have seen previously?

- How does it rate on the “big 4”?
  - Recall that game-tree search is still a form of search.

Minimax algorithm

- The minimax algorithm performs a complete depth-first exploration of the game tree. If the maximum depth of the tree is m and there are b legal moves at each point, then the time complexity of the minimax algorithm is O(b^m). The space complexity is O(bm) for an algorithm that generates all actions at once, or O(m) for an algorithm that generates actions one at a time. For real games, of course, the time cost is totally impractical, but this algorithm serves as the basis for the mathematical analysis of games and for more practical algorithms.

Optimal decisions in multiplayer games

- Many popular games allow more than two players. Let us examine how to extend the minimax idea to multiplayer games. This is straightforward from the technical viewpoint, but raises some interesting conceptual issues.
- First, we need to replace the single value for each node with a *vector* of values. For example, in a three-player game with players A, B, and C, a vector (vA,vB,vC) is associated with each node. For terminal states, this vector gives the utility of the state from each player's viewpoint. (In two-player zero-sum games, the two-element vector can be reduced to a single value because the values are always opposite.) The simplest way to implement this is to have the UTILITY function return a vector of utilities.

Multiplayer minimax algorithm

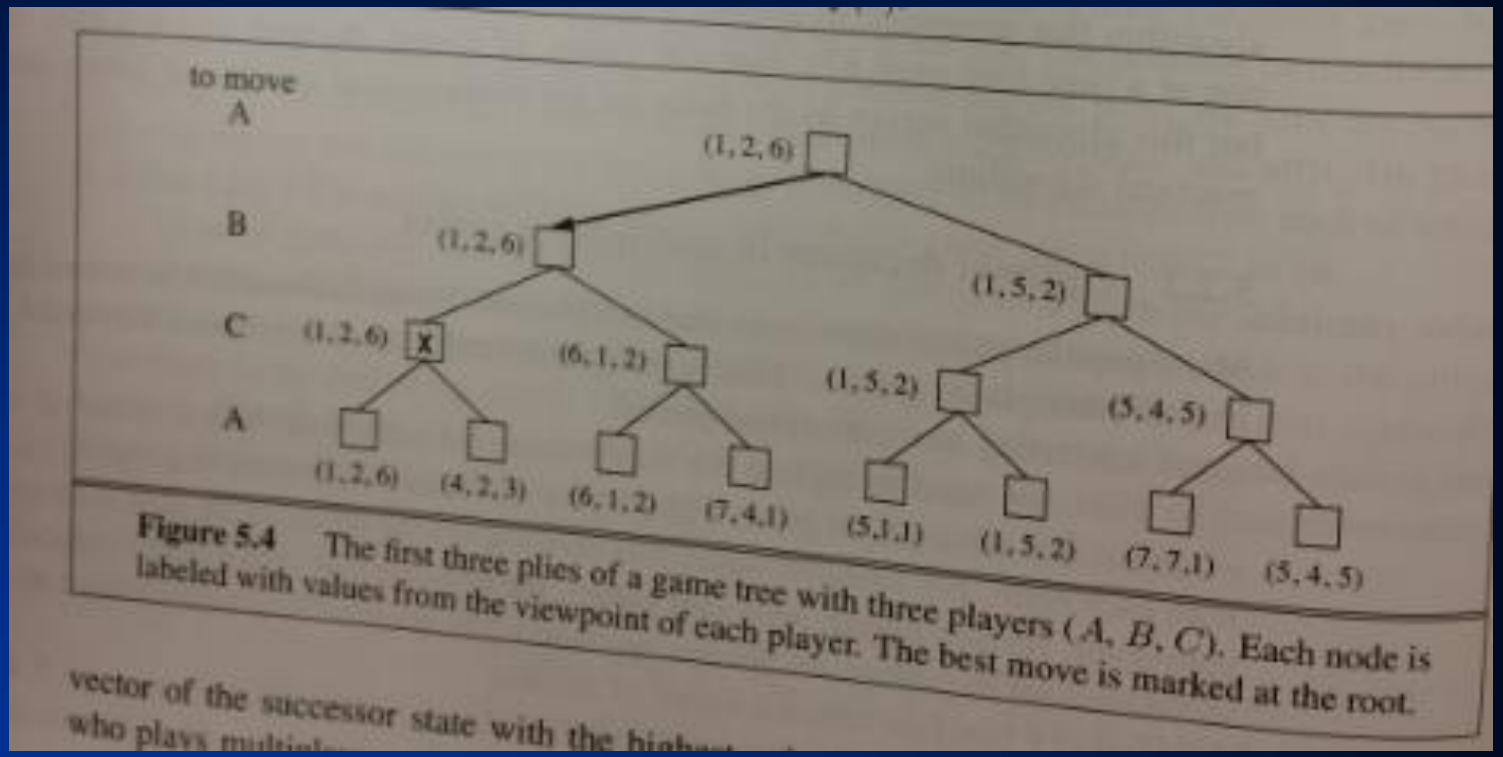


Figure 5.4 The first three plies of a game tree with three players (A, B, C). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

vector of the successor state with the highest

who plays

Multiplayer minimax

- Now we have to consider nonterminal states. Consider the node marked X in the game tree. In that state, player C chooses what to do. The two choices lead to terminal states with utility vectors (vA=1,vB=2,vC=6) and (vA=4,vB=2,vC=3). Since 6 is bigger than 3, C should choose the first move. This means that if state X is reached, subsequent play will lead to a terminal state with utilities (vA=1,vB=2,vC=6). Hence, the backed-up value of X is this vector.

Multiplayer minimax

- The backed-up value of a node n is always the utility vector of the successor state with the highest value for the player choosing at n. Anyone who plays multiplayer games, such as Diplomacy, quickly becomes aware that much more is going on than in two-player games. Multiplayer games usually involve **alliances**, whether formal or informal, among the players. Alliances are made and broken as the game proceeds. How are we to understand such behavior? Are alliances a natural consequence of optimal strategies for each player in a multiplayer game?

Multiplayer minimax

- It turns out that they can be. For example, suppose A and B are in weak positions and C is in a stronger position. Then it is often optimal for both A and B to attach C rather than each other, lest C destroy each of them individually. In this way, collaboration emerges from purely selfish behavior. Of course, as soon as C weakens under the joint onslaught, the alliance loses its value, and either A or B could violate the agreement. In some cases, a social stigma attaches to breaking an alliance, so players must balance the immediate advantage of breaking an alliance against the long-term disadvantage of being perceived as untrustworthy.

Multiplayer minimax

- If the game is not zero-sum, then collaboration can also occur with just two players. Suppose, for example, that there is a terminal state with utilities (vA=1000,vB=1000) and that 1000 is the highest possible utility for each player. Then the optimal strategy is for both players to do everything possible to reach this state—that is, the players will automatically cooperate to achieve a mutually desirable goal.

Game-tree search pruning

- The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree. Unfortunately, we can't eliminate the exponent, but it turns out that we can effectively cut it in half. The trick is that it is possible to compute the correct minimax decision without looking at every node in the game tree. That is, we can borrow the idea of **pruning** from the search section (recall that A* pruned the subtree following below Timisoara) to eliminate large parts of the tree from consideration. The particular technique we consider is **alpha-beta pruning**. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

Game tree

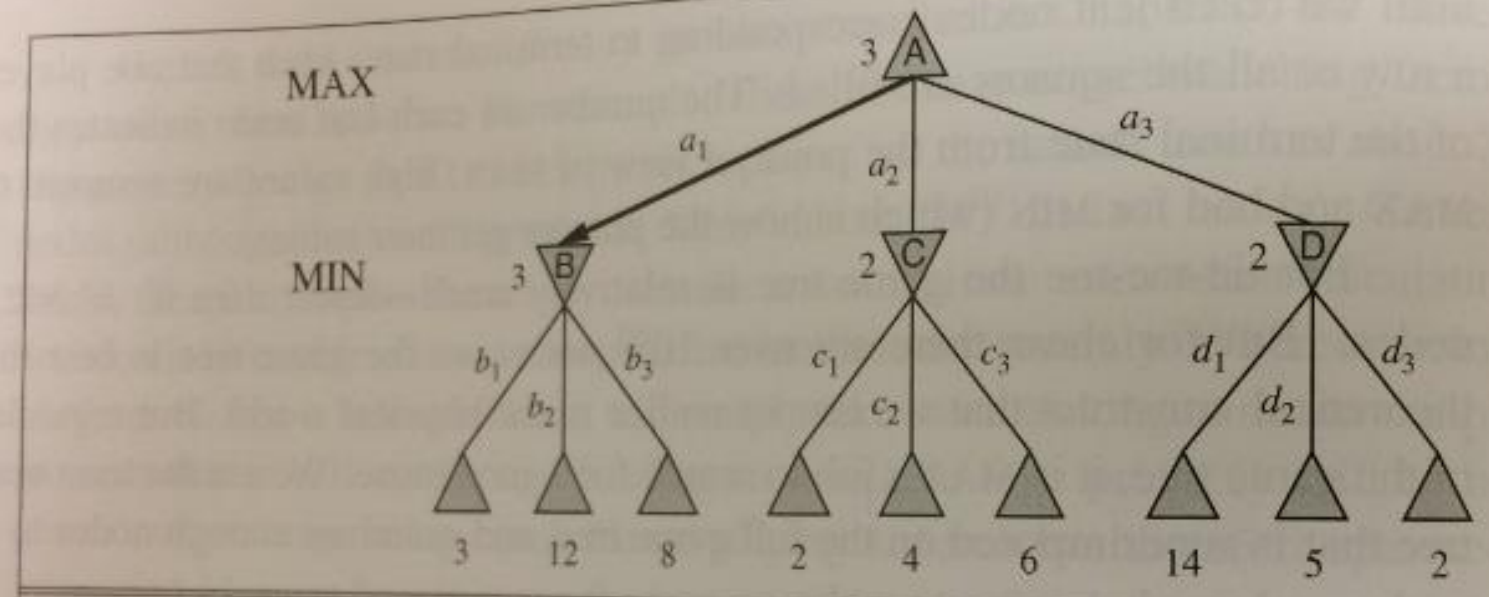


Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state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

Alpha-beta pruning

- Consider again the two-play game tree. Let's go through the calculation of the optimal decision once more, this time paying careful attention to what we know at each point in the process. The steps are explained in the figure on the next page. The outcome is that we can identify the minimax decision without ever evaluating two of the leaf nodes.

Alpha-beta pruning

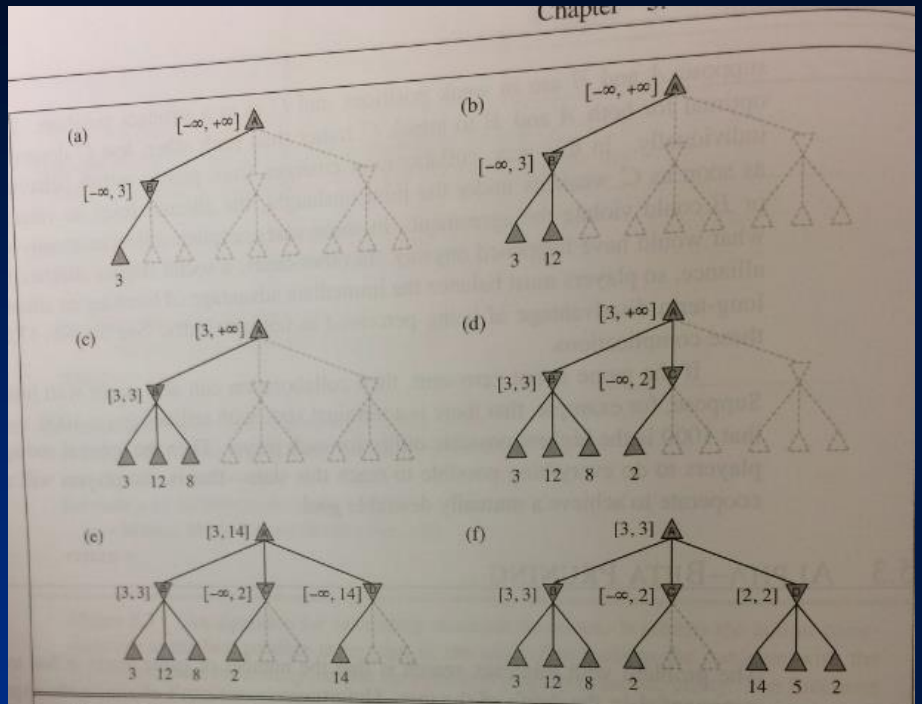


Figure 5.5 Stages in the calculation of the optimal decision for the game tree in Figure 5.2. At each point, we show the range of possible values for each node. (a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of *at most* 3. (b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3. (c) The third leaf below B has a value of 8; we have seen all B's successor states, so the value of B is exactly 3. Now, we can infer that the value of the root is *at least* 3, because MAX has a choice worth 3 at the root. (d) The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of *at most* 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successor states of C. This is an example of alpha–beta pruning. (e) The first leaf below D has the value 14, so D is worth *at most* 14. This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring D's successor states. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14. (f) The second successor of D is worth 5, so again we need to keep exploring. The third successor is worth 2, so now D is worth exactly 2. MAX's decision at the root is to move to B, giving a value of 3.

somewhere in the tree (see Figure 5.6),

Alpha-beta pruning

- Another way to look at this is as a simplification of the formula for MINIMAX. Let the two unevaluated successors of node C in the figure have values x and y. Then the value of the root node is given by:

  MIMIMAX(root)

  = max(min(3,12,8),min(2,x,y),min(14,5,2)

  = max(3,min(2,x,y),2)

  = max(3,z,2) where z = min(2,x,y) <= 2

  = 3.

- In other words, the value of the root and hence the minimax decision are *independent* of the values of the pruned leaves x and y.

Alpha-beta pruning

- Alpha-beta pruning can be applied to trees of any depth, and it is often possible to prune entire subtrees rather than just leaves. The general principle is this: consider a node n somewhere in the tree (see next figure) such that Player has a choice of moving to that node. If Player has a better choice m either at the parent node of n or at any choice point further up, then *n will never be reached in actual play*. So once we have found out enough about n (by examining some of its descendants) to reach this conclusion, we can prune it.

General alpha-beta pruning

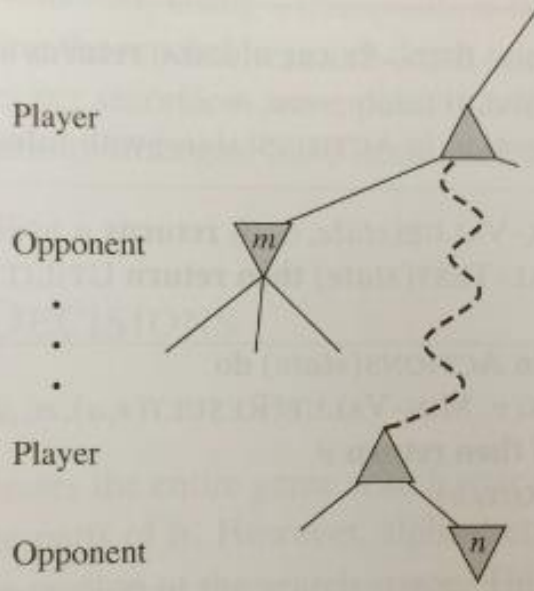


Figure 5.6 The general case for alpha–beta pruning. If m is better than n for Player, we will never get to n in play.

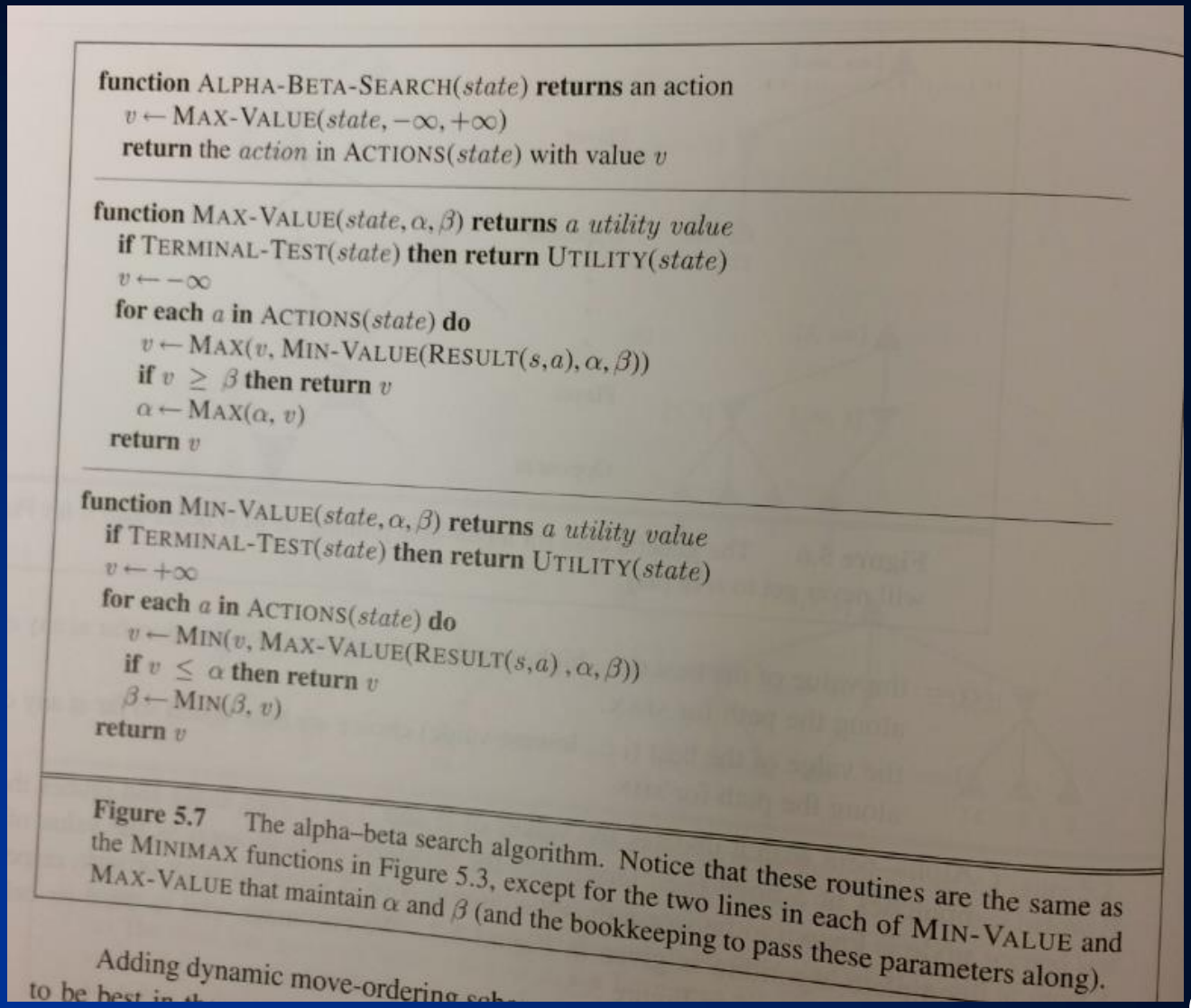
Alpha-beta search

- Remember that minimax search is depth-first, so at any one time we just have to consider the nodes along a single path in the tree. Alpha-beta pruning gets its name from the following two parameters that describe bounds on the backed-up values that appear anywhere along the path:
  - α = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.
  - β = the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.

Alpha-beta search algorithm

- Alpha-beta search updates the values of α and β as it goes along and prunes the remaining branches at a node (i.e., terminates the recursive call) as soon as the values of the current node is known to be worse than the current α or β value for MAX or MIN, respectively. The complete algorithm is given on the next slide. We can trace its behavior when applied to the example.

Alpha-beta search algorithm



```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a), α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure 5.7 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

Adding dynamic move-ordering sch
to be best in th

Move ordering

- The effectiveness of alpha-beta pruning is highly dependent on the order in which the states are examined. For example, in the figure we could not prune any successors of D at all because the worst successors (from the point of view of MIN) were generated first. If the third successor of D had been generated first, we would have been able to prune the other two. This suggests that it might be worthwhile to try to examine firs the successors that are likely to be best.

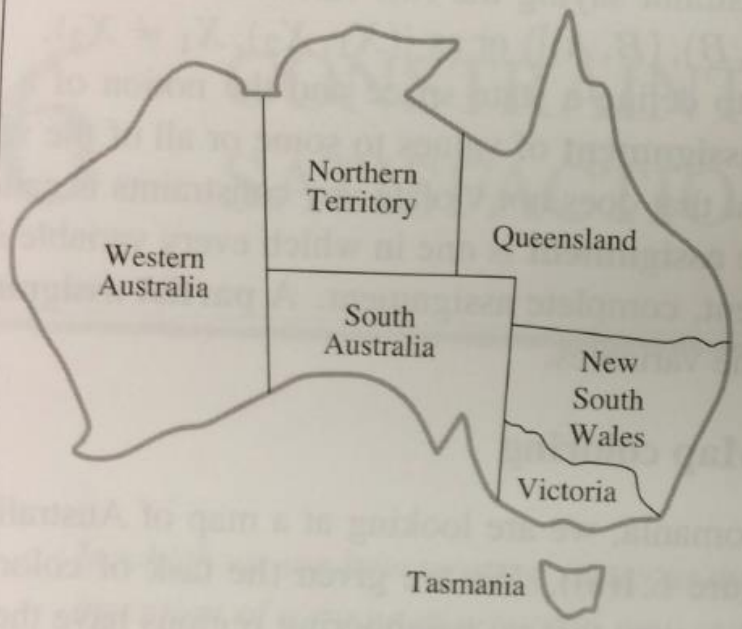
Alpha-beta move ordering

- If this can be done, then it turns out that alpha-beta needs to examine only O(b^(m/2)) nodes to pick the best move, instead of O(b^m) for minimax. This means that the effective branching factor becomes sqrt(b) instead of b – for chess, about 6 instead of 35. Put another way, alpha-beta can solve a tree roughly twice as deep as minimax in the same amount of time. If successors are examined in random order rather than best-first, the total number of nodes examined will be roughly O(b^(3m/4)) for moderate b. For chess, a fairly simple ordering function (such as trying captures first, then threats, then forward moves, and then backward moves) gets to within about a factor of 2 of the best-case O(b^(m/2)) result.

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



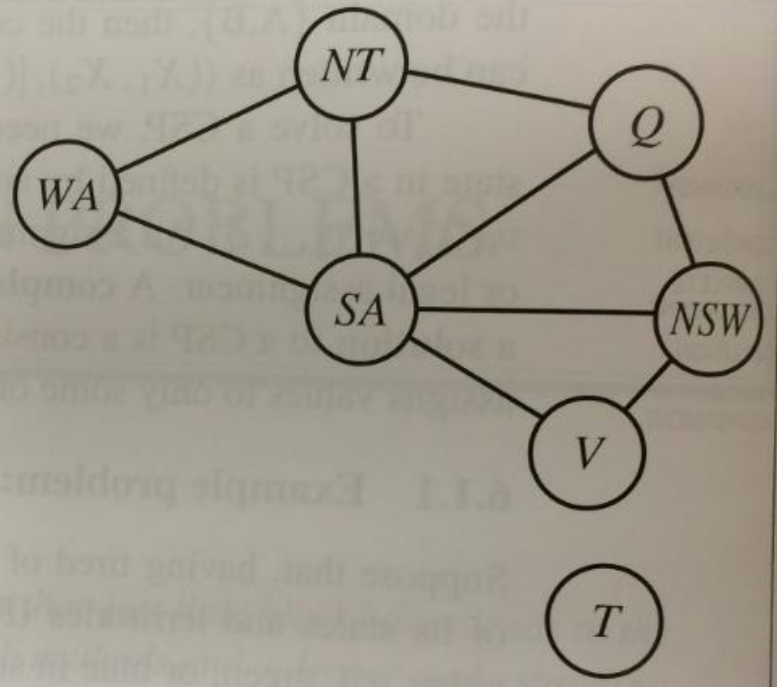


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

immediately discard further

Homework for next class

- Chapter 9 from Russell-Norvig textbook.
- HW1: out 9/5 due 10/3