UNIT–III: SEARCH IN COMPLEX ENVIRONMENTS (9 periods)

"Local search algorithms and optimization problems – Hill-climbing search, Simulated annealing, Local beam search, Evolutionary algorithms; Optimal decisions in games – The minimax search algorithm, Optimal decisions in multiplayer games, Alpha-Beta pruning, Move ordering; Monte Carlo tree search.

Problem of finding a good state without worrying about the path to get there, covering both discrete and continuous states.

3.1 Local Search and Optimization Problems

3.1.1 Hill-climbing search
3.1.2 Simulated annealing
3.1.3 Local beam search
3.1.4 Evolutionary algorithms

Local search algorithms operate by searching from a start state to neighboring states, without keeping track of the paths, nor the set of states that have been reached.
they have two key advantages:
(1) they use very little memory;
(2) they can often find reasonable solutions in large or infinite state spaces for which systematic algorithms are unsuitable.
Local search algorithms can also solve optimization problems, in which the aim is to find the best state according to an objective function.

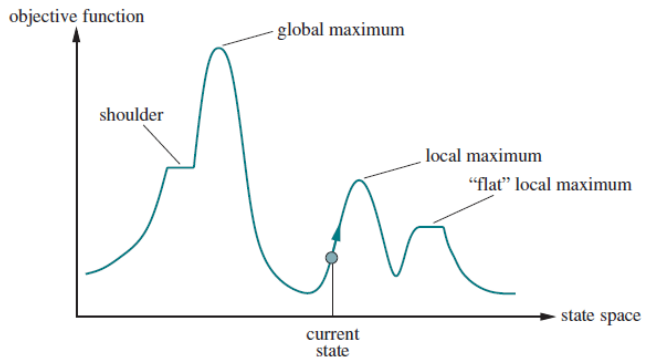


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum.

Local Search and Optimization Problems

- Consider the states of a problem laid out in a state-space landscape, as shown in Figure 4.1
- Each point (state) in the landscape has an "elevation," defined by the value of the objective function.
- If elevation corresponds to an objective function, then the aim is to find the highest peak—a global maximum—the process called as hill climbing.
- If elevation corresponds to cost, then the aim is to find the lowest valley—a global minimum—called as gradient descent.

3.1.1 Hill-climbing search

- It keeps track of one current state and on each iteration moves to the neighboring state with highest value—(i.e) it heads in the direction that provides the steepest ascent

- It terminates when it reaches a "peak" where no neighbor has a higher value.
- Hill climbing does not look ahead beyond the immediate neighbors of the current state

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  current ← problem.INITIAL
  while true do
    neighbor ← a highest-valued successor state of current
    if VALUE(neighbor) ≤ VALUE(current) then return current
    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.

- **Hill Climbing Search illustrated using 8 Queens Problem**

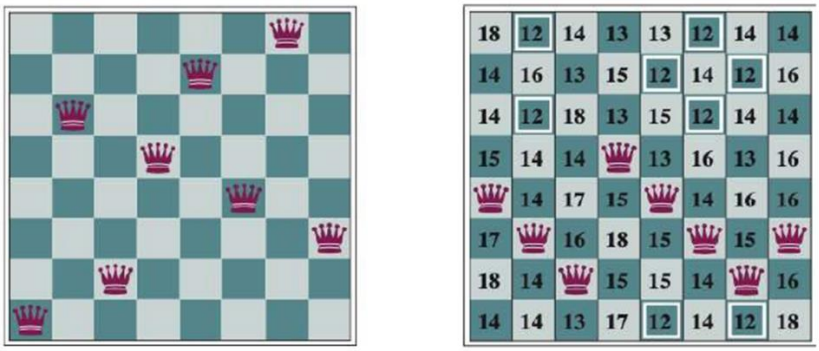


Figure 4.3 (a) The 8-queens problem: place 8 queens on a chess board so that no queen attacks another. (A queen attacks any piece in the same row, column, or diagonal.) This position is almost a solution, except for the two queens in the fourth and seventh columns that attack each other along the diagonal. (b) An 8-queens state with heuristic cost estimate $h = 17$. The board shows the value of h for each possible successor obtained by moving a queen within its column. There are 8 moves that are tied for best, with $h = 12$. The hill-climbing algorithm will pick one of these.

- Every state has 8 queens on the board, one per column. The initial state is chosen at random, and the successors of a state are all possible states generated by moving a single queen to another square in the same column (8X7-56 Successors)
- The heuristic cost function is the number of pairs of queens that are attacking each other; this will be zero only for solutions.
- Hill climbing is sometimes called greedy local search because it grabs a good neighbor state without thinking ahead about where to go next.
- Hill climbing can make rapid progress toward a solution because it is usually quite easy to improve a bad state.
- Unfortunately, hill climbing can get stuck for any of 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**
  - Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.
- **PLATEAUS**
  - Plateau is a flat area of the state-space landscape. It can be a flat local maximum, from which no uphill exit exists, or a shoulder, from which progress is possible

- Hill-climbing search can get lost wandering on the plateau.

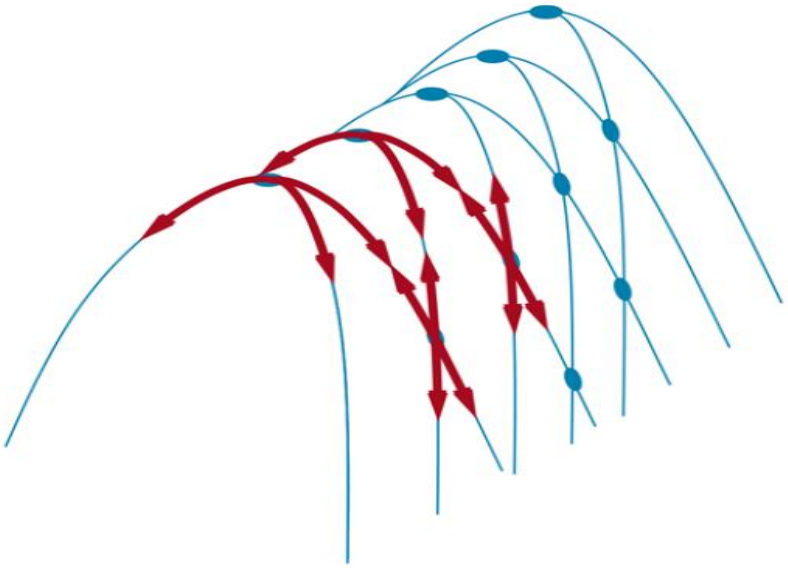


Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill. Topologies like this are common in low-dimensional state spaces, such as points in a two-dimensional plane. But in state spaces with hundreds or thousands of dimensions, this intuitive picture does not hold, and there are usually at least a few dimensions that make it possible to escape from ridges and plateaus.

- **Variants of Hill Climbing Search**
  - Stochastic hill climbing
  - First-choice hill climbing
  - Random-restart hill climbing
- **Stochastic hill climbing**
  - Chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move.
  - Converges more slowly than steepest ascent, but 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.
- **Random-restart hill climbing**
  - It conducts a series of hill-climbing searches from randomly generated initial states, until a goal is found.

3.1.2 Simulated Annealing

- A hill-climbing algorithm that never makes "downhill" moves toward states with lower value (or higher cost) is always vulnerable to getting stuck in a local maximum.
- It seems reasonable to try to combine hill climbing with a random walk in a way that yields both efficiency and completeness.

Simulated annealing is such an algorithm

- The overall structure of the simulated-annealing algorithm is 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 ΔE by which the evaluation is worsened
- The probability also decreases as the "temperature" T goes down
- "bad" moves are more likely to be allowed at the start when T is high, and they become more unlikely as T decreases.
- If the schedule lowers to 0 slowly enough, then a property of the Boltzmann distribution
- If the schedule lowers to 0 slowly enough, then a property of the Boltzmann distribution eΔE/T is that all the probability is concentrated on the global maxima, which the algorithm will find with probability approaching 1.

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  current ← problem.INITIAL
  for t = 1 to ∞ do
      T ← schedule(t)
      if T = 0 then return current
      next ← a randomly selected successor of current
      ΔE ← VALUE(current) – VALUE(next)
      if ΔE > 0 then current ← next
      else current ← next only with probability e^(−ΔE/T)
```

Figure 4.4 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the "temperature" T as a function of time.

3.1.3 Local Beam Search

- The local beam search algorithm keeps track of states rather than just
- one. It begins with randomly generated states.
- At each step, all the successors of all k states are generated. If any one is a goal, the algorithm halts.
- Otherwise, it selects the best successors from the complete list and repeats.
- In a local beam search, useful information is passed among the parallel search threads
- Local beam search can suffer from a lack of diversity among the states—they can become clustered in a small region of the state space, making the search little more than a –times slower version of hill climbing.
- A variant called stochastic beam search, analogous to stochastic hill climbing, helps to overcome this problem.
- Instead of choosing the top successors, stochastic beam search chooses successors with probability proportional to the successor's value, thus increasing diversity

3.1.4 Evolutionary Algorithms

- Evolutionary algorithms can be seen as variants of stochastic beam search that are explicitly motivated by the metaphor of natural selection in biology
- There is a population of individuals (states), in which the fittest (highest value) individuals produce offspring (successor states) that populate the next generation, a process called recombination.
- There are endless forms of evolutionary algorithms, varying in the following ways:
- The size of the population.

- The representation of each individual.
- In genetic algorithms, each individual is a string over a finite alphabet (often a Boolean string), just as DNA is a string over the alphabet ACGT.
- In evolution strategies, an individual is a sequence of real numbers, and in genetic programming an individual is a computer program
- The mixing number, , which is the number of parents that come together to form
- offspring. ρ = 2 – common case ρ = 1 stochastic beam search ρ >2 rare but can be simulated on computers
- The selection process for selecting the individuals who will become the parents of the next generation:
- one possibility is to select from all individuals with probability proportional to their fitness score.
- Another possibility is to randomly select n individuals (n > ρ), and then select the ρ most fit ones as parents
- The recombination procedure. One common approach (assuming), is to randomly
- select a crossover point to split each of the parent strings, and recombine the parts to form two children,
- one with the first part of parent 1 and the second part of parent 2;
- the other with the second part of parent 1 and the first part of parent 2.
- The mutation rate, which determines how often offspring have random mutations to their representation.
- Once an offspring has been generated, every bit in its composition is flipped with probability equal to the mutation rate.
- The makeup of the next generation. This can be just the newly formed offspring, or it can include a few top-scoring parents from the previous generation (a practice called elitism, which guarantees that overall fitness will never decrease over time).
- The practice of culling, in which all individuals below a given threshold are discarded, can lead to a speedup

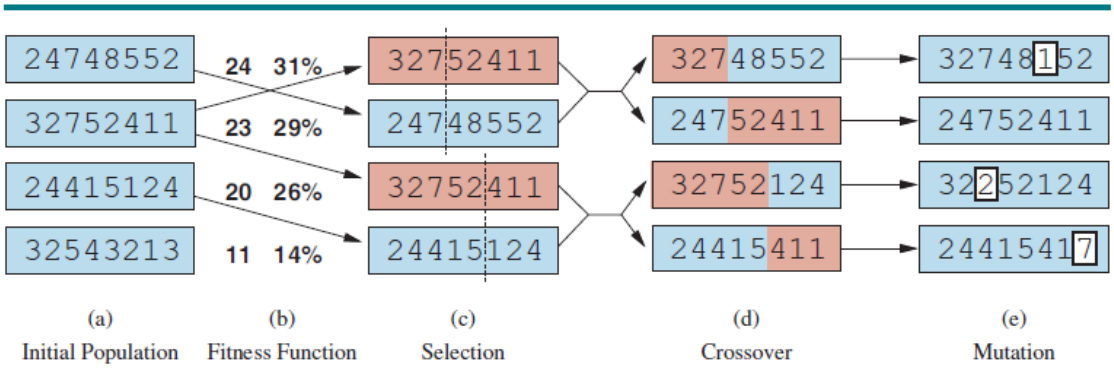


Figure 4.5 A genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by a fitness function in (b) resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

- Fig (a) shows a population of four 8-digit strings, each representing a state of the 8-queens puzzle: the c-th digit represents the row number of the queen in column .
- In (b),each state is rated by the fitness function.
- Higher fitness values are better, so for the 8-queens problem we use the number of nonattacking pairs of queens, which has a value of 28 (8*7/2)
- The values of the four states in (b) are 24, 23, 20, and 11.
- The fitness scores are then normalized to probabilities, and the resulting values are shown

next to the fitness values in (b).

- In (c), two pairs of parents are selected, in accordance with the probabilities in (b).
- Notice that one individual is selected twice and one not at all.
- For each selected pair, a crossover point (dotted line) is chosen randomly.
- In (d), we cross over the parent strings at the crossover points, yielding new offspring.

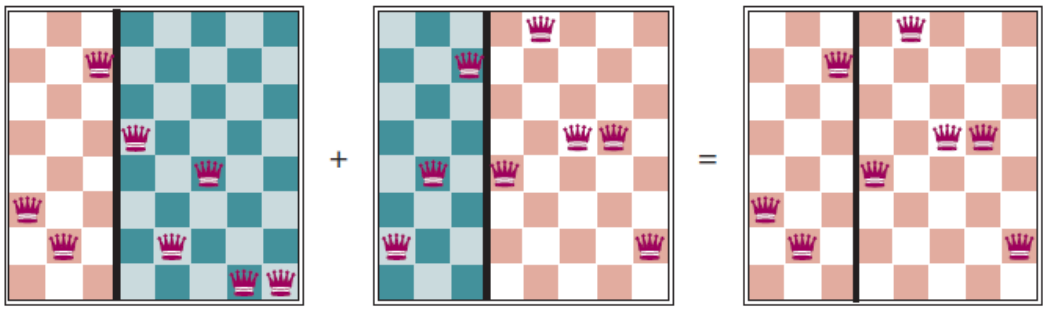


Figure 4.6 The 8-queens states corresponding to the first two parents in Figure ??(c) and the first offspring in Figure ??(d). The green columns are lost in the crossover step and the red columns are retained. (To interpret the numbers in Figure ??: row 1 is the bottom row, and 8 is the top row.)

```
function GENETIC-ALGORITHM(population, fitness) returns an individual
  repeat
    weights ← WEIGHTED-BY(population, fitness)
    population2 ← empty list
    for i = 1 to SIZE(population) do
      parent1, parent2 ← WEIGHTED-RANDOM-CHOICES(population, weights, 2)
      child ← REPRODUCE(parent1, parent2)
      if (small random probability) then child ← MUTATE(child)
      add child to population2
    population ← population2
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to fitness

function REPRODUCE(parent1, parent2) returns an individual
  n ← LENGTH(parent1)
  c ← random number from 1 to n
  return APPEND(SUBSTRING(parent1, 1, c), SUBSTRING(parent2, c + 1, n))
```

Figure 4.7 A genetic algorithm. Within the function, *population* is an ordered list of individuals, *weights* is a list of corresponding fitness values for each individual, and *fitness* is a function to compute these values.

- Genetic algorithms are similar to stochastic beam search, but with the addition of the crossover operation.
- This is advantageous if there are blocks that perform useful functions.
- The theory of genetic algorithms explains how this works using the idea of a schema, which is a substring in which some of the positions can be left unspecified.
- For example, the schema 246***** describes all 8-queens states in which the first three queens are in positions 2, 4, and 6, respectively.
- Strings that match the schema (such as 24613578) are called instances of the schema.

- It can be shown that if the average fitness of the instances of a schema is above the mean, then the number of instances of the schema will grow overtime.
- Genetic algorithms have their place within the broad landscape of optimization
- Methods, particularly for complex structured problems such as circuit layout or job scheduling, and more recently for evolving the architecture of deep neural networks

3.2 Optimal decisions in games

- The minimax search algorithm
- Optimal decisions in multiplayer games
- Alpha-Beta pruning
- Move ordering

- **Optimal decisions in games**
  - MAX wants to find a sequence of actions leading to a win, but MIN has something to say about it.
  - MAX's strategy must be a conditional plan—a contingent strategy specifying a response to each of MIN's possible moves.
  - In games that have a binary outcome (win or lose), we could use AND–OR search to generate the conditional plan
  - For games with multiple outcome scores, we need a slightly more general algorithm called minimax search

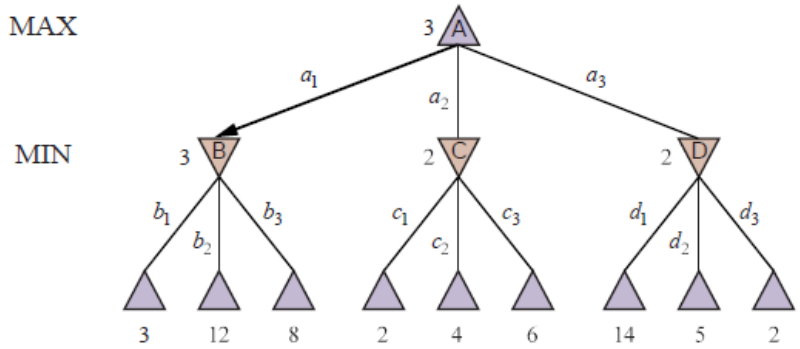


Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state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

- Given a game tree, the optimal strategy can be determined by working out the minimax value of each state in the tree, which we write as MINIMAX(s).
- The minimax value is the utility (for MAX) of being in that state, assuming that both players play optimally from there to the end of the game.
- The minimax value of a terminal state is just its utility.
- In a non-terminal state, MAX prefers to move to a state of maximum value when it is MAX's turn to move, and MIN prefers a state of minimum value (that is, minimum value for MAX and thus maximum value for MIN). So we have:

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

- The terminal nodes on the bottom level get their utility values from the game's UTILITY function.
- The first MIN node, labeled, 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.
- minimax decision at the root: action a1 that gives optimal choice for MAX because it leads to the state with the highest minimax value.

3.2.1 The minimax search algorithm

- It is a recursive algorithm that proceeds all the way down to the leaves of the tree and then backs up the minimax values through the tree as the recursion unwinds.
- 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.
- 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.

```
function MINIMAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Figure 5.3 An algorithm for calculating the optimal move using minimax—the move that leads to a terminal state with maximum 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 and the move to get there.

- 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 legal moves at each point, then the time complexity of the minimax algorithm is O(bm)
- 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
- The exponential complexity makes MINIMAX impractical for complex games.
- By approximating the minimax analysis in various ways, we can derive more practical algorithms.

3.2.2 Optimal decision in multiplayer games

- Many popular games allow more than two players.
- Let us examine how to extend the minimax idea to multiplayer games.
- 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.
- The simplest way to implement this is to have the UTILITY function return a vector of utilities.
- In general, the backed-up value of a node n is the utility vector of the successor state with the highest value for the player choosing at n.

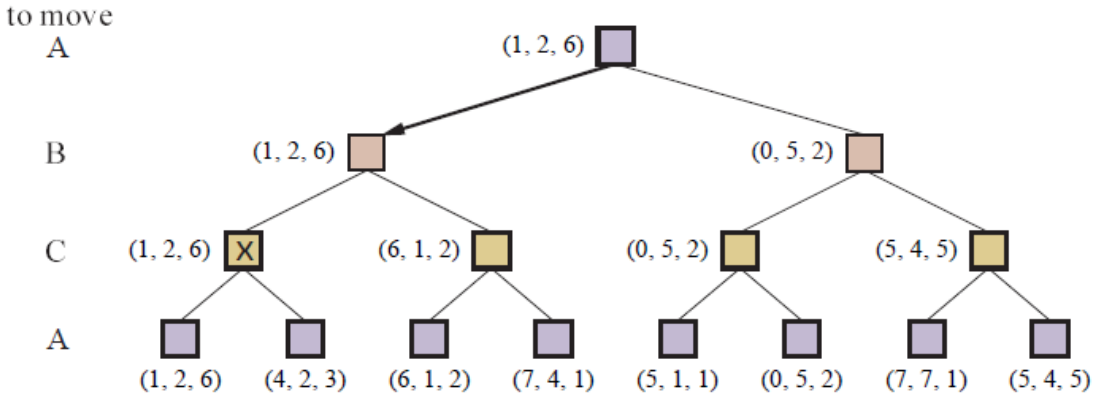
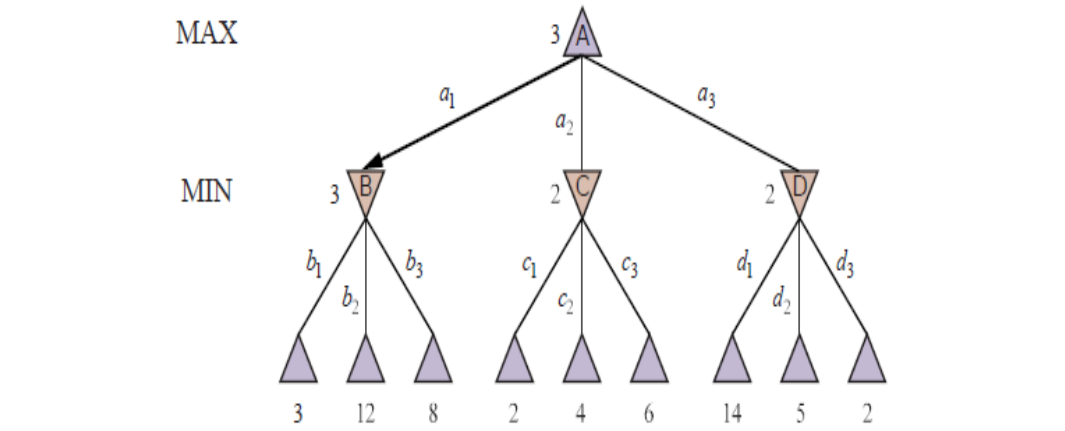


Figure 5.4 The first three ply of a game tree with three players (A, B, C). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

- Multiplayer games usually involve alliances, whether formal or informal, among the players. Alliances are made and broken as the game proceeds.

3.2.3 Alpha-Beta Pruning

- The number of game states is exponential in the depth of the tree
- No algorithm can completely eliminate the exponent, but we can sometimes cut it in half, computing the correct minimax decision without examining every state by pruning large parts of the tree that make no difference to the outcome.
- The particular technique we examine is called alpha–beta pruning.
- Consider again the two-ply game tree
- The outcome of optimal decision is that we can identify the minimax decision without ever evaluating two of the leaf nodes



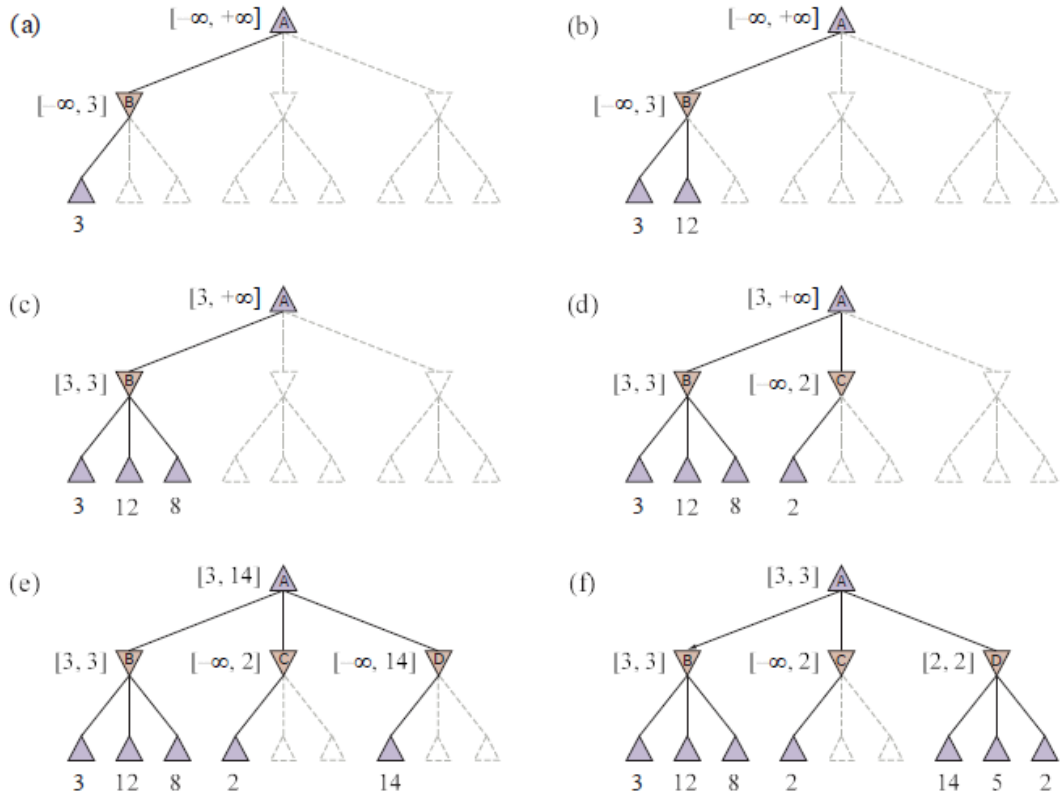


Figure 5.5 Stages in the calculation of the optimal decision for the game tree in Figure ??. At each point, we show the range of possible values for each node. (a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of *at most* 3. (b) The second leaf below B has a value 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.

- Another way to look at this is as a simplification of the formula for MINIMAX.
- Let the two unevaluated successors of node C in Figure 5.5 have values x and y.

Then the value of the root node is given by

$$\begin{aligned}\textsc{Minimax}(root) &= \max(\min(3,12,8), \min(2,x,y), \min(14,5,2)) \\ &= \max(3, \min(2,x,y), 2) \\ &= \max(3,z,2) \qquad \text{where } z = \min(2,x,y) \leq 2 \\ &= 3.\end{aligned}$$

- The value of the root and hence the minimax decision are independent of the
- values of the leaves x and y, and therefore they can be pruned.
- Alpha–beta pruning can be applied to trees of any depth, and it is often possible to prune entire subtrees rather than just leaves.
- Consider a node somewhere in the tree (see Figure 5.6), such that Player has a choice of moving to.
- If Player has a better choice either at the same level or at any point higher up in the tree, then Player will never move to n.
- So once we have found out enough about (by examining some of its descendants) to reach this conclusion,
- we can prune it.

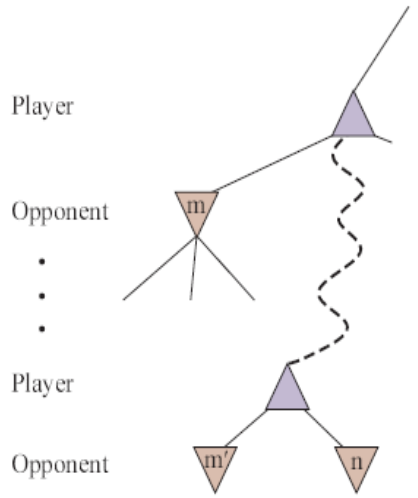


Figure 5.6 The general case for alpha–beta pruning. If m or m' is better than n for Player, we will never get to n in play.

- 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 two extra parameters in MAX-VALUE (state, α, β)
- 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. Think: α = "at least."
- 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. Think: β = "at most."
- 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 value of the current node is known to be worse than the current α or β value for MAX or MIN, respectively

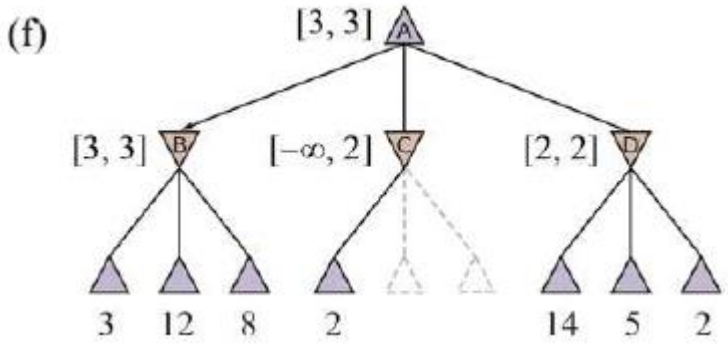
```
function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
    if v2 > v then
      v, move ← v2, a
      α ← MAX(α, v)
    if v ≥ β then return v, move
  return v, move

function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
    if v2 < v then
      v, move ← v2, a
      β ← MIN(β, v)
    if v ≤ α then return v, move
  return v, move
```

Figure 5.7 The alpha–beta search algorithm. Notice that these functions are the same as the MINIMAX-SEARCH functions in Figure ??, except that we maintain bounds in the variables α and β, and use them to cut off search when a value is outside the bounds.

3.2.4 Move ordering

- The effectiveness of alpha–beta pruning is highly dependent on the order in which the states are examined



- we could not prune any successors of at D 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, with value 2, we would have been able to prune the other two successors.
- This suggests that it might be worthwhile to try to first examine the successors that are likely to be best.
- If this could be done perfectly, alpha–beta would need to examine O(bm/2)only
- nodes to pick the best move, instead of O(bm) for minimax
- Alpha–beta with perfect move ordering can solve a tree roughly twice as deep as minimax in the same amount of time.
- With random move ordering, the total number of nodes examined will be roughly for

- O(b3m/4) moderate b.
- Adding dynamic move-ordering schemes, such as trying first the moves that were found to be best in the past, brings us quite close to the theoretical limit.
- The past could be the previous move—often the same threats remain—or it could come from previous exploration of the current move through a process of iterative deepening
- First, search one ply deep and record the ranking of moves based on their evaluations. Then search one ply deeper, using the previous ranking to inform move ordering; and so on.
- The increased search time from iterative deepening can be more than made up from better move ordering.
- The best moves are known as killer moves, and to try them first is called the killer move heuristic.
- In game tree search, repeated states can occur because of transpositions—different permutations of the move sequence that end up in the same position, and the problem can be addressed with a transposition table that caches the heuristic value of states.
- Even with alpha–beta pruning and clever move ordering, minimax won't work for games like chess and Go, because there are still too many states to explore in the time available.
- In the very first paper on computer game-playing, Programming a Computer for Playing Chess (Shannon, 1950), Claude Shannon recognized this problem and proposed two strategies:
- Type A strategy - considers all possible moves to a certain depth in the search tree, and then uses a heuristic evaluation function to estimate the utility of states at that depth. It explores a wide but shallow portion of the tree (chess programs have been Type A)
- Type B strategy - ignores moves that look bad, and follows promising lines "as far as possible." It explores a deep but narrow portion of the tree (Go programs are more often Type B)

3.3 Monte Carlo Tree Search

- The game of Go illustrates two major weaknesses of heuristic alpha–beta tree search:
- First, Go has a branching factor that starts at 361, which means alpha–beta search would be limited to only 4 or 5 ply.
- Second, it is difficult to define a good evaluation function for Go because material value is not a strong indicator and most positions are in flux until the endgame.
- In response to these two challenges, modern Go programs have abandoned alpha–beta search and instead use a strategy called Monte Carlo tree search (MCTS).
- The basic MCTS strategy does not use a heuristic evaluation function.
- The value of a state is estimated as the average utility over a number of simulations of complete games starting from the state.
- A simulation (also called a playout or rollout) chooses moves first for one player, than for the other, repeating until a terminal position is reached.
- At that point the rules of the game (not fallible heuristics) determine who has won or lost, and by what score.
- For games in which the only outcomes are a win or a loss, "average utility" is the same as "win percentage."
- To get useful information from the playout we need a playout policy that biases the moves towards good ones."

- For Go and other games, playout policies have been successfully learned from self-play by using neural networks
- Given a playout policy, we next need to decide two things:
  - From what positions do we start the playouts,
  - How many playouts do we allocate to each position?
- The simplest answer, called pure Monte Carlo search, is to do N simulations starting from the current state of the game, and track which of the possible moves from the current position has the highest win percentage.
- For some stochastic games this converges to optimal play as N increases, but for most games it is not sufficient—we need a selection policy that selectively focuses the computational resources on the important parts of the game treeIt balances two factors:
  - **exploration of states that have had few playouts,**
  - **exploitation of states that have done well in past playouts, to get a more accurate estimate of their value**
- Monte Carlo tree search does that by maintaining a search tree and growing it on each iteration of the following four steps

- **SELECTION:** Starting at the root of the search tree, we choose a move (guided by the selection policy), leading to a successor node, and repeat that process, moving down the tree to a leaf
- **EXPANSION:** We grow the search tree by generating a new child of the selected node; Figure 5.10(b) shows the new node marked with 0/0. (Some versions generate more than one child in this step.)
- **SIMULATION:** We perform a playout from the newly generated child node, choosing moves for both players according to the playout policy. These moves are not recorded in the search tree. In the figure, the simulation results in a win for black.
- **BACK-PROPAGATION:** We now use the result of the simulation to update all the search tree nodes going up to the root. Since black won the playout, black nodes are incremented in both the number of wins and the number of playouts,
- so 27/35 becomes 28/36 and 60/79 becomes 61/80. Since white lost, the white nodes are incremented in the number of playouts only, so 16/53 becomes 16/54 and the root 37/100 becomes 37/101.

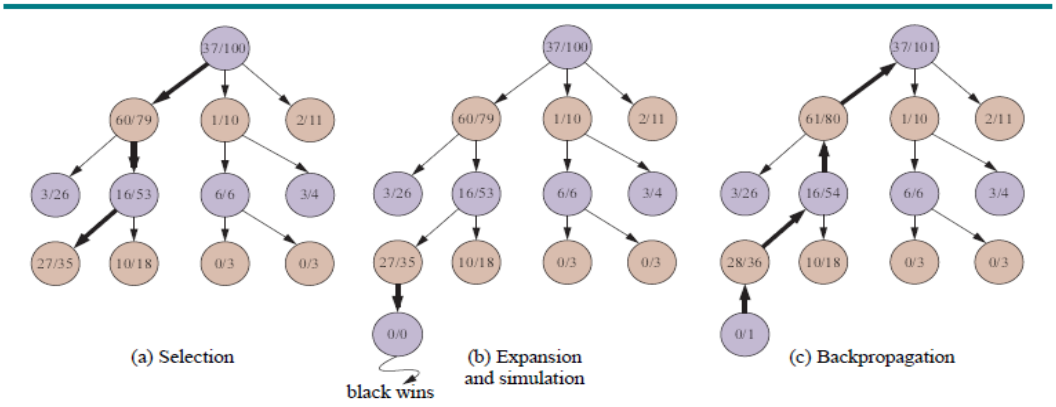


Figure 5.10 One iteration of the process of choosing a move with Monte Carlo tree search (MCTS) using the upper confidence bounds applied to trees (UCT) selection metric, shown after 100 iterations have already been done. In (a) we select moves, all the way down the tree, ending at the leaf node marked 27/35 (for 27 wins for black out of 35 playouts). In (b) we expand the selected node and do a simulation (playout), which ends in a win for black. In (c), the results of the simulation are back-propagated up the tree.

- We repeat these four steps either for a set number of iterations, or until the allotted time has

expired, and then return the move with the highest number of playouts.

- One very effective selection policy is called "upper confidence bounds applied to trees" or UCT.
- **The policy ranks each possible move based on an upper confidence bound formula** called **UCB1. For a node , the formula is:**

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(\text{PARENT}(n))}{N(n)}}$$

- where U(n) is the total utility of all playouts that went through node, N(n) is the number of playouts through node, PARENT(n)is the parent node of n in the tree.
  Term in square root is exploitation term.

The term with the square root is the exploration term: it has the count $N(n)$ in the denominator, which means the term will be high for nodes that have only been explored a few times. In the numerator it has the log of the number of times we have explored the parent of n. This means that if we are selecting n some non-zero percentage of the time, the exploration term goes to zero as the counts increase, and eventually the playouts are given to the node with highest average utility.

C is a constant that balances exploitation and exploration. There is a theoretical argument that C should be $\sqrt{2}$, but in practice, game programmers try multiple values for C and choose the one that performs best. (Some programs use slightly different formulas; for example, AlphaZero adds in a term for move probability, which is calculated by a neural network trained from past self-play.) With $C = 1.4$, the 60/79 node in Figure 5.10 has the highest UCB1 score, but with $C = 1.5$, it would be the 2/11 node.

```
function MONTE-CARLO-TREE-SEARCH(state) returns an action
  tree ← NODE(state)
  while IS-TIME-REMAINING() do
    leaf ← SELECT(tree)
    child ← EXPAND(leaf)
    result ← SIMULATE(child)
    BACK-PROPAGATE(result, child)
  return the move in ACTIONS(state) whose node has highest number of playouts
```

Figure 5.11 The Monte Carlo tree search algorithm. A game tree, *tree*, is initialized, and then we repeat a cycle of SELECT / EXPAND / SIMULATE / BACK-PROPAGATE until we run out of time, and return the move that led to the node with the highest number of playouts.

- The time to compute a playout is linear, not exponential, in the depth of the game tree, because only one move is taken at each choice point. That gives us plenty of time for multiple playouts.
- we have enough computing power to consider a billion game states before we have to make a move, then minimax can search 6 ply deep,
  - alpha–beta with perfect move ordering can search 12 ply,
  - Monte Carlo search can do 10 million playouts.
- Which approach will be better? That depends on the accuracy of the heuristic function versus the selection and playout policies.

- The conventional wisdom has been that Monte Carlo search has an advantage over alpha–beta for games like Go where the branching factor is very high (and thus alpha–beta can't search deep enough), or when it is difficult to define a good evaluation function.
- What alpha–beta does is choose the path to a node that has the highest achievable evaluation function score, given that the opponent will be trying to minimize the score.
- If the evaluation function is inaccurate, alpha–beta will be inaccurate.
- A miscalculation on a single node can lead alpha–beta to erroneously choose (or avoid) a path to that node.

- **Advantages**

- Monte Carlo search relies on the aggregate of many playouts, and thus is not as vulnerable to a single error.
- It is possible to combine MCTS and evaluation functions by doing a playout for
- a certain number of moves, but then truncating the playout and applying an evaluation function.
- Monte Carlo search can be applied to brand-new games, in which there is no body of experience to draw upon to define an evaluation function. As long as we know the rules of the game,
- Monte Carlo search does not need any additional information.
- The selection and playout policies can make good use of hand-crafted expert knowledge when it is available, but good policies can be learned using neural networks trained by self-play alone

- Disdvantages

- It is likely that a single move can change the course of the game, because the stochastic nature of Monte Carlo search means it might fail to consider that move.
- Monte Carlo search also has a disadvantage when there are game states that are "obviously" a win for one side or the other (according to human knowledge and to an evaluation function), but where it will still take many moves in a playout to verify the winner.
- It was long held that alpha–beta search was better suited for games like chess with low branching factor and good evaluation functions, but recently Monte Carlo approaches have demonstrated success in chess and other games.
- The general idea of simulating moves into the future, observing the outcome, and using the outcome to determine which moves are good ones is one kind of reinforcement learning.

- Go Game

- The board is empty at the onset of the game (unless players agree to place a handicap).
- Black makes the first move, after which White and Black alternate.
- A move consists of placing one stone of one's own color on an empty intersection on the board.
- A player may pass their turn at any time.
- A stone or solidly connected group of stones of one color is captured and removed from the board when all the intersections directly adjacent to it are occupied by the enemy. (Capture of the enemy takes precedence over self-capture.)
- No stone may be played so as to recreate a former board position.
- Two consecutive passes end the game.
- A player's area consists of all the points the player has either occupied or surrounded.
- The player with more area wins."