CS 581 – Advanced Artificial Intelligence

Topic: Search

Mustafa Bilgic

http://www.cs.iit.edu/~mbilgic

https://twitter.com/bilgicm

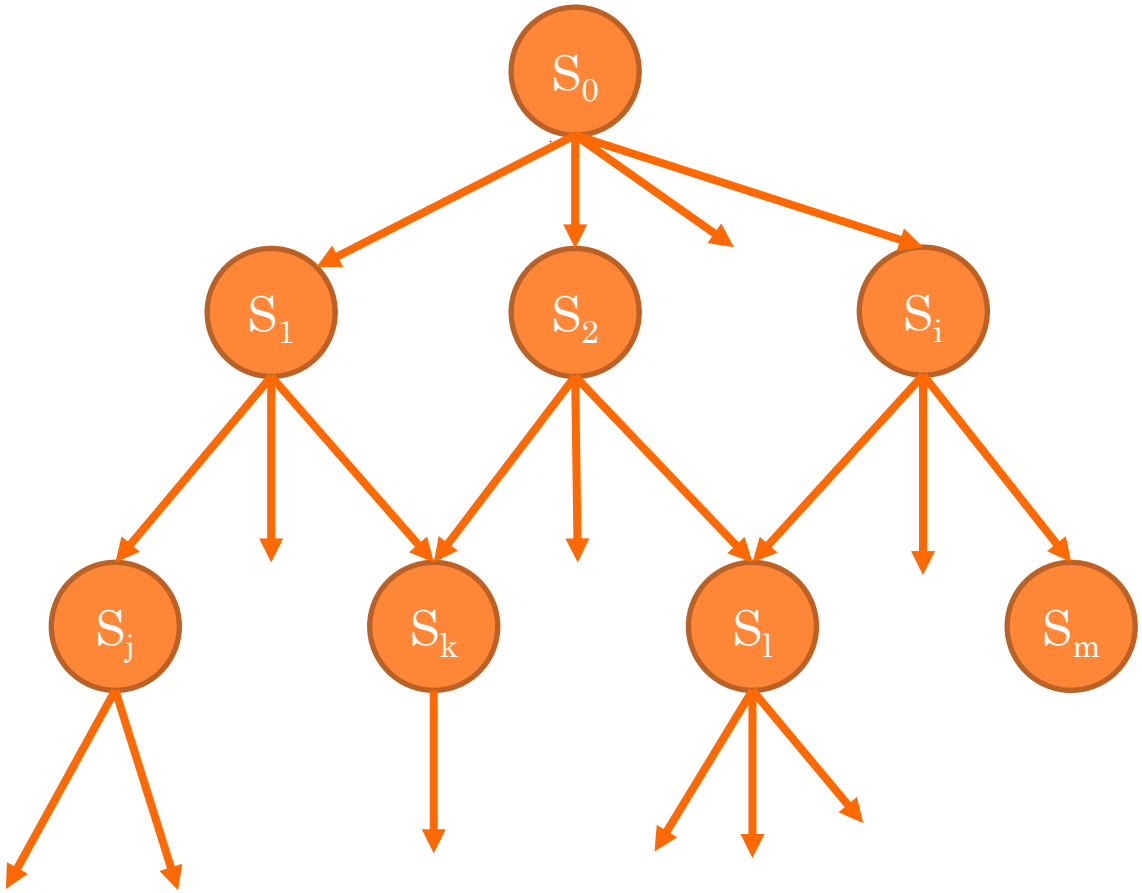
Motivation

- AI is powered by search
- Consider multiple possibilities and find the "optimal" one
- Just a few examples
  - Travel from one location to another
    - Find the "optimal" route
  - Game playing
    - Find the "optimal" move
  - Machine learning
    - Find the "optimal" set of parameters
  - Machine translation
    - Find the "optimal" sequence of words
  - Constraint satisfaction
    - Find the "optimal" assignment
  - …

We will Study

- Hill climbing, A*, genetic algorithms, simulated annealing, …
- Maximum likelihood estimation, Bayesian estimation, expectation maximization, gradient optimization, Lagrange multipliers, policy search, …
- Minimax, Alpha-beta search, Monte-Carlo tree search, …

Abstract Representation



Given

- An initial state
- Available actions at each state
- A transition model
- Costs
  - A goal state and the cost of each action, OR
  - "Goodness" of each state

Find

- A sequence of actions that take you from the initial state to the goal state, where the total cost of the sequence is minimized, OR
- A maximum state

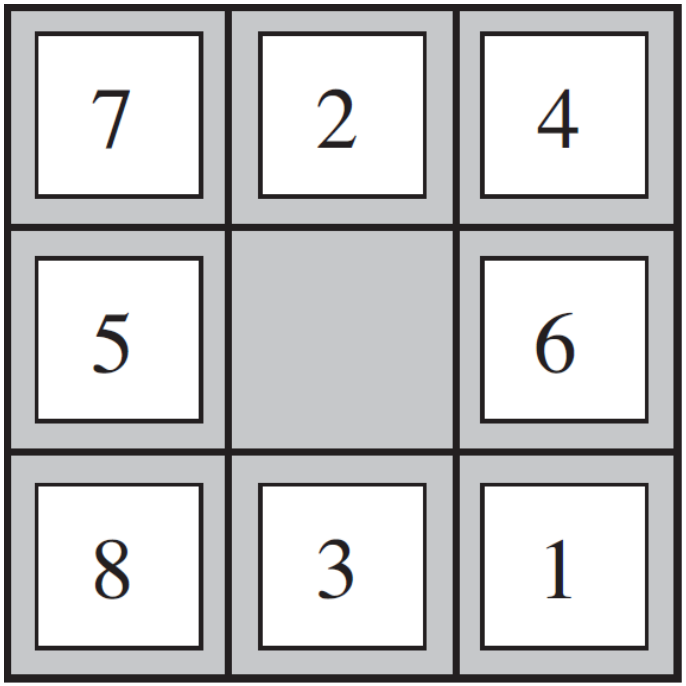
Route Finding

- The world representation
  - In(City)
- Initial state
  - In(Madison, WI)
- Goal state
  - In(Detroit, MI)
- Actions
  - Travel to the neighboring city
- Transition model
  - If you are in city A and travel to city B, the state changes from In(A) to In(B)
- Cost
  - Distance traveled

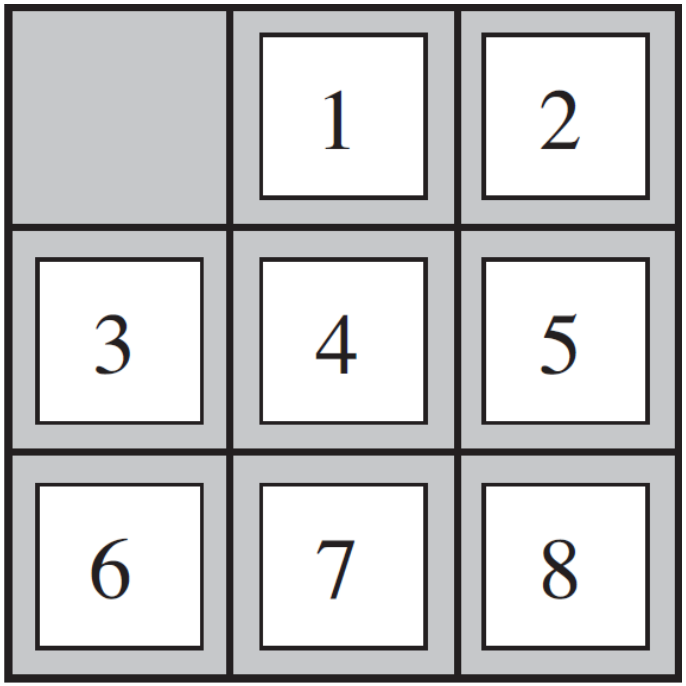
Traveling Salesman Problem

- Given
  - A list of N cities
  - Distances between each pair of cities
- Find
  - A minimum-cost travel plan where the agent visits each city exactly once and returns to the city of origin

8 Puzzle

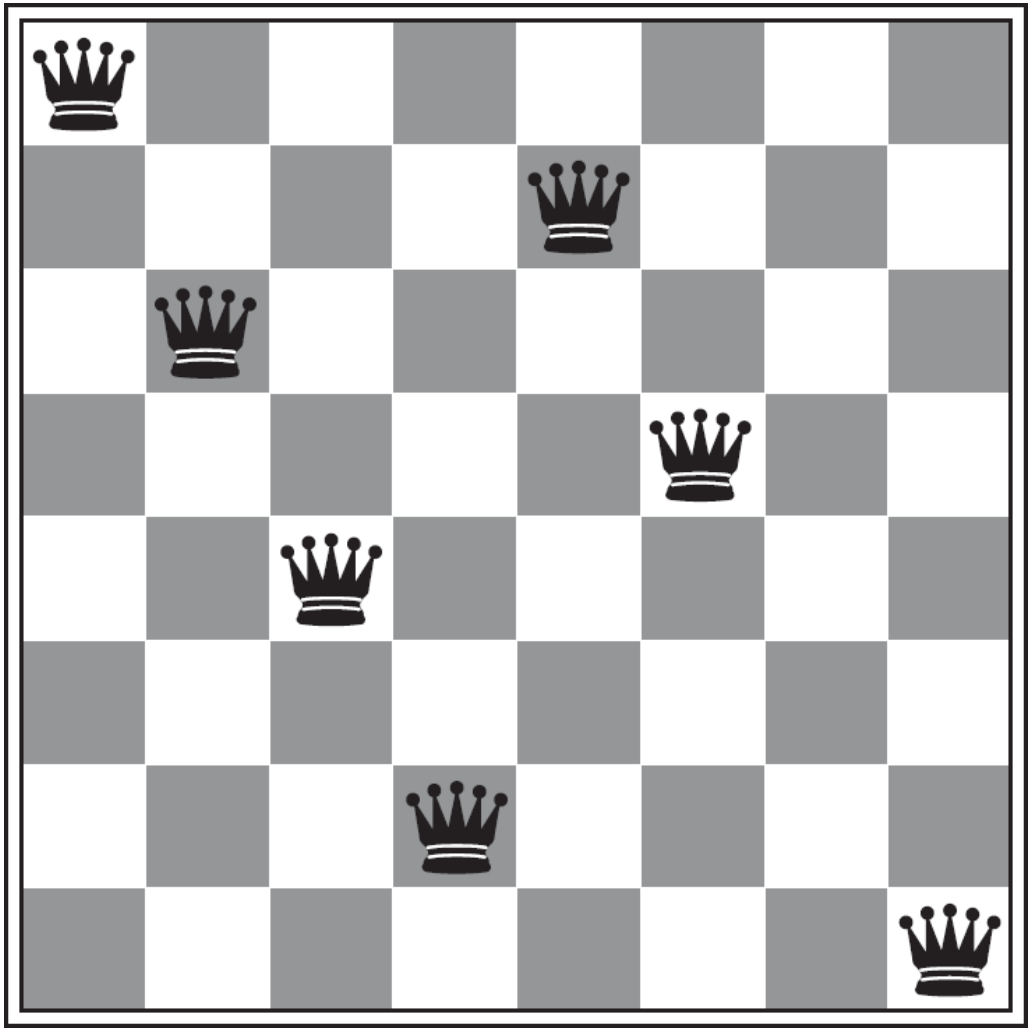


Start State



Goal State

8 Queens



Game Playing

- Single-player games
  - E.g., sudoku
- Two-player games
  - E.g., tic-tac-toe, chess, Go
- Multi-player games
  - E.g., many video games
- Team games
  - E.g., robotic soccer

Outline

- Route finding
  - A sequence of actions to reach from an initial state to a goal state
  - Best-first search: heuristic search, uniform cost search, A*
  - Chapter 3
- Local search
  - Start at an initial state; reach/find a maximum utility state
  - Hill climbing, simulated annealing, genetic algorithms
  - Chapter 4
- Games
  - Two-player games
  - Minimax, alpha-beta pruning, Monte-Carlo tree search
  - Chapter 5

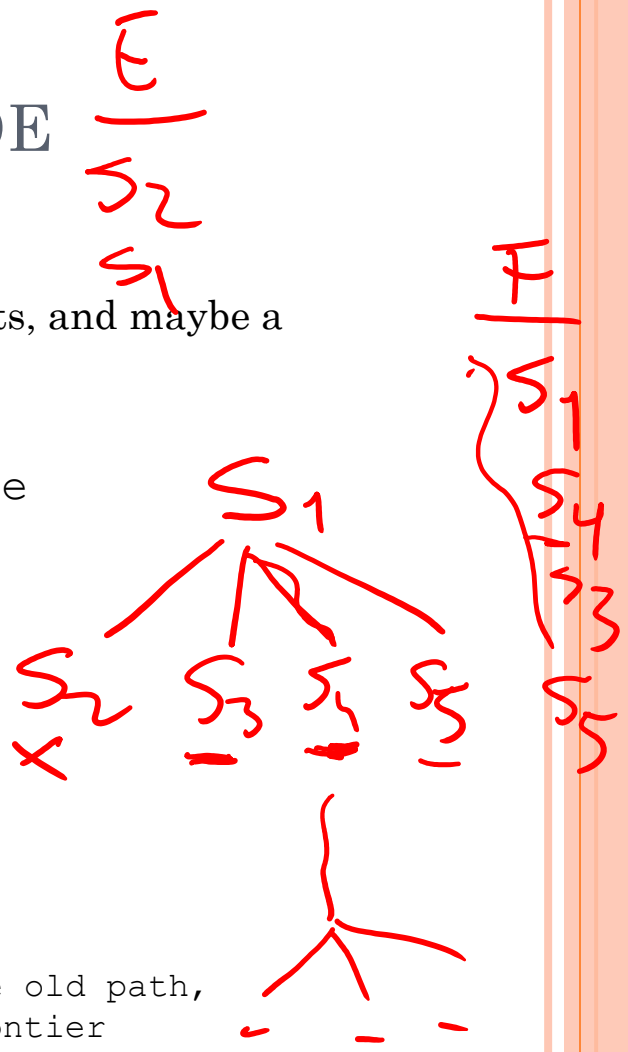
Route Finding
Chapter 3

Exercise

- You are at an initial city D
- You are trying to reach to the city M
- The cities are not fully connected; you can travel from one city to only its neighbor cities
- You don't have access to the full map; you know only the parts you explored or discovered
- Each time you pick a city that you know and ask where you can go from there
- You want to find the shortest path from D to M, and you want to do it as fast as possible
- Notation:
  - $c(n, n')$: the cost of traveling from n to n'
  - $h(n)$: the estimate of the distance from n to the goal
- Three settings:
  - Setting 1: You are given the c function but not h
  - Setting 2: You are given the h function but not c
  - Setting 3: You are given both c and h functions

Best-first Search Pseudocode

- Given
  - The initial state, goal state, available actions, action costs, and maybe a heuristic function
  - `frontier` – a priority queue; `explored`: a set
- `Initialize frontier with the initial state`
- `While frontier is not empty`
  - `Pop the top node n in the frontier`
  - `Add n to explored`
  - `If n is the goal state, return the solution`
  - `For each child n' of n:`
    - `If n' is not in explored`
      - `If n' is not in frontier`
        - `Add it to frontier`
      - `Else if the new path to n' is better than the old path, replace the old n' with the new n' in the frontier`
- `Return failure // frontier is empty; goal was not found`



Best-First Search Algorithms

- Define $f(n) = h(n) + g(n)$
  - $h(n)$: Cost estimate from n to the goal
  - $g(n)$: Cost from the initial state to n

1. Uniform-cost search
   - `frontier` is sorted using $g(n)$
2. Greedy heuristic search
   - `frontier` is sorted using $h(n)$
3. A* search
   - `frontier` is sorted using $f(n)$

Route Finding Example

- See OneNote

Optimal

- Assume the optimal path cost is p^*
- Uniform cost search is guaranteed to return the optimal path
  - Expands all nodes where $g(n) \leq p^*$
- Greedy heuristic search is not optimal; ignores path costs
- A* is optimal only if h is admissible and consistent
  - Expands all nodes where $g(n) + h(n) \leq p^*$

Admissible? Consistent?

- **Admissible** if
  - $h(n)$ never overestimates the optimal cost
  - That is $h(n)$ is always optimistic
  - E.g., straight line distance between two cities
- **Consistent** if
  - If $h(n) \leq c(n,n') + h(n')$
  - n' is the successor of n
  - Triangle inequality
- If a heuristic is consistent, it is also admissible (the other way around is not guaranteed)

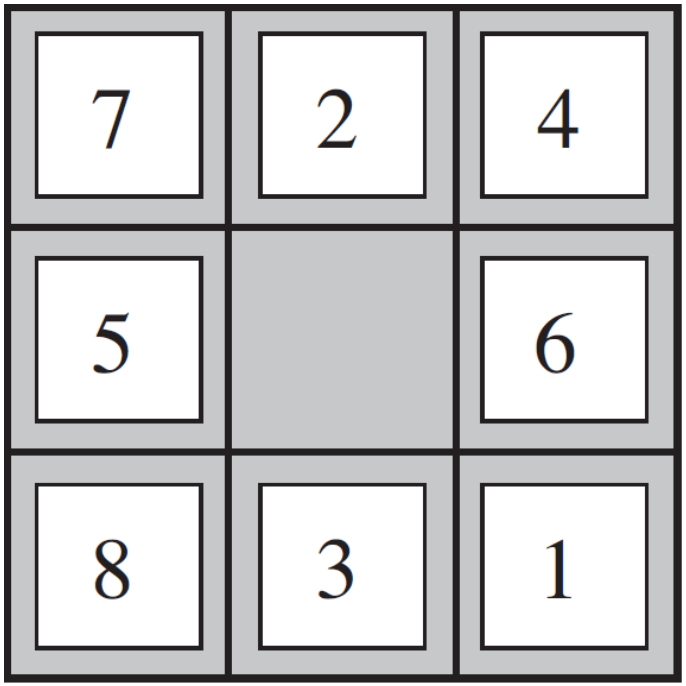
A* Optimality – Proof Sketch

1. If $h(n)$ is consistent, then $f(n)$ along any path is non-decreasing
2. When n is expanded, the optimal path to it has been found

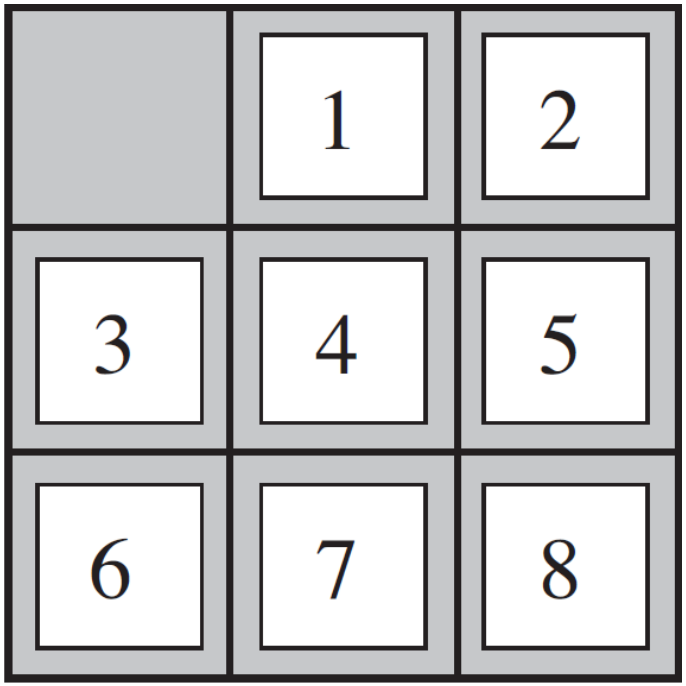
UCS vs A*

- A* = UCS if $h(n) = 0$ for all n
- UCS searches in circles whereas A* searches in ellipses

8 Puzzle



Start State



Goal State

Heuristics

- How to design *h(n)*?
  - The answer is "depends on the domain" but some general principles exist
  - For example, relax the problem constraints a bit
- For travel problems
  - Flight distance
- For 8-Puzzle
  - Number of misplaced tiles
  - Manhattan distance

A 3-Puzzle Problem Example

- See OneNote

Other Search Algorithms

- Depth-first search
- Breadth-first search
- Iterative deepening search
- Bidirectional search

Comparing Algorithms

- Time complexity
- Space complexity
- Completeness
  - If there is a solution, can it find it?
- Optimality
  - Is the found solution the optimal one?

Comparing Heuristics

- h_i **dominates** h_j iff
  - $h_i(n) \geq h_j(n)\ \forall n$
- For the 8-Puzzles problem
  - Manhattan distance dominates # of misplaced tiles
- If you have multiple admissible heuristics where none dominates the other
  - Let $h(n) = \max(h_1(n), h_2(n), \ldots, h_m(n))$
  - $h(n)$ is admissible and it dominates all $h_i(n)$
  - Of course, computing a heuristic is not free, and hence we need to consider the computational cost of $h(n)$ compared to each $h_i(n)$

Effective Branching Factor

- If the total number of nodes generated is N and the solution depth is d, then
  - b^* is the branching factor that a uniform tree of depth d would need to have in order to contain $N+1$ nodes
- $N+1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$
- If A* finds a solution at depth 4 using 40 nodes, what is b^*?
  - ≈ 2.182
- A good heuristic function achieves $b^* \approx 1$

Empirical Comparisons

	Search Cost (nodes generated)			Effective Branching Factor		
d	BFS	A*(h_1)	A*(h_2)	BFS	A*(h_1)	A*(h_2)
6	128	24	19	2.01	1.42	1.34
8	368	48	31	1.91	1.40	1.30
10	1033	116	48	1.85	1.43	1.27
12	2672	279	84	1.80	1.45	1.28
14	6783	678	174	1.77	1.47	1.31
16	17270	1683	364	1.74	1.48	1.32
18	41558	4102	751	1.72	1.49	1.34
20	91493	9905	1318	1.69	1.50	1.34
22	175921	22955	2548	1.66	1.50	1.34
24	290082	53039	5733	1.62	1.50	1.36
26	395355	110372	10080	1.58	1.50	1.35
28	463234	202565	22055	1.53	1.49	1.36

Figure 3.26 Comparison of the search costs and effective branching factors for 8-puzzle problems using breadth-first search, A* with h_1 (misplaced tiles), and A* with h_2 (Manhattan distance). Data are averaged over 100 puzzles for each solution length d from 6 to 28.

Figure from http://aima.cs.berkeley.edu/figures.pdf

Empirical Comparisons

- https://github.com/aimacode/aima-python/blob/master/search4e.ipynb

Find the Maximum Utility State Chapter 4

Discrete State Space Search

- Each state
  - Is discrete
  - Represents a complete representation of a possible solution
  - Has a "desirability" score
  - Might have neighbors reachable through actions/moves
- The goal is to find the maximum/minimum scoring state
- Some examples
  - Bayesian network structure search
    - A state is a candidate structure; the score is BIC; actions are to "add/remove/reverse" an edge
  - 8-Queens
    - A state is placement of 8 queens on the board; the score is the number of pairs of queens attacking each other; the actions are to move a queen to another empty square
  - Traveling salesman
    - A state is a travel plan; the score is the cost of the plan; the actions are to swap the order of two neighboring cities
- The actions and state representations are not written on stone; you can define them in a different way
  - How you define the states and actions can have a profound impact on the shape of the search space

OneNote Illustration

- You are at state A
- Actions are: Left or Right
- Every state has a value
- You want to find the state with the maximum value

HILL CLIMBING

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.

Figure from http://aima.cs.berkeley.edu/algorithms.pdf

The Search Landscape

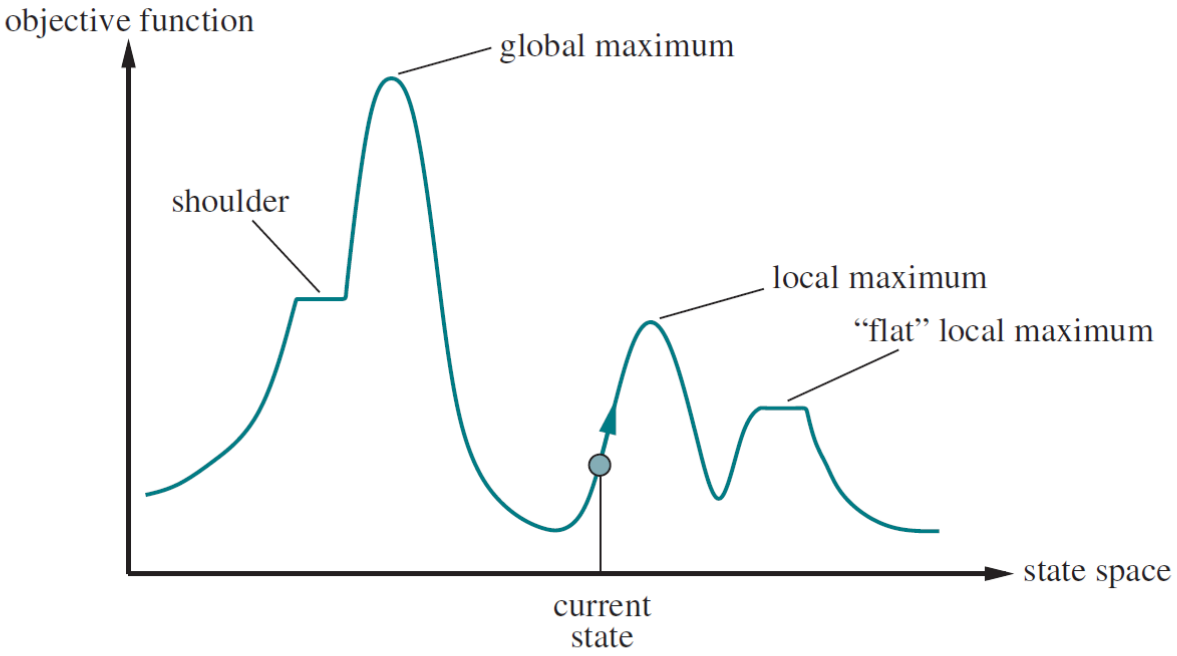


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum.

Figure from http://aima.cs.berkeley.edu/figures.pdf

8-Queens Example

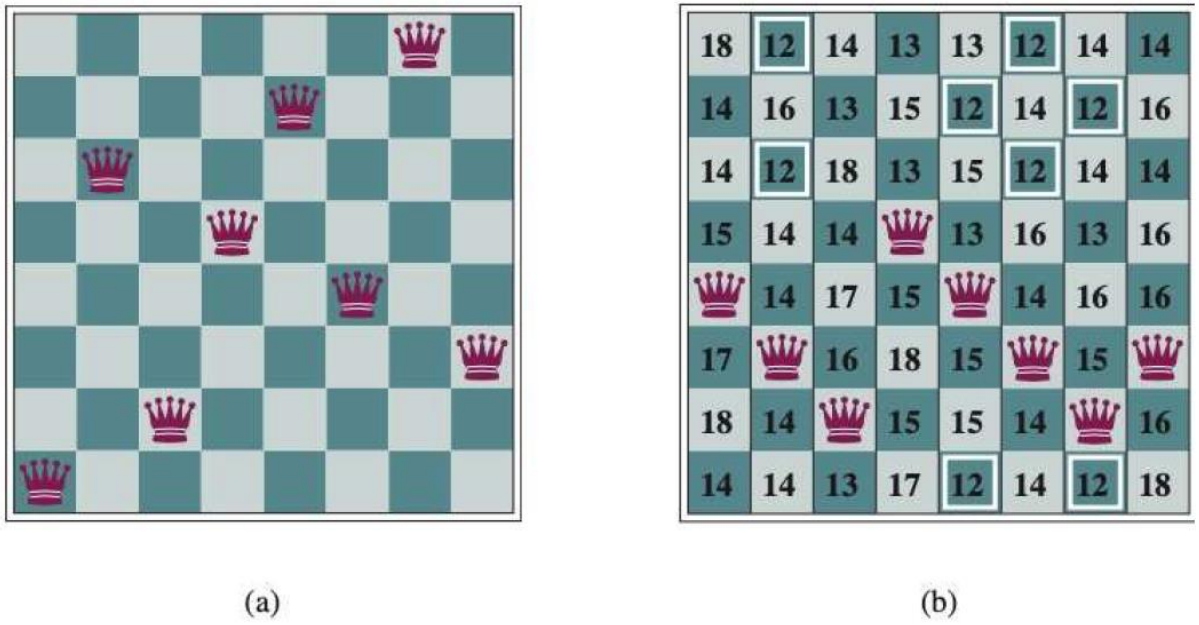


Figure 4.3 (a) The 8-queens problem: place 8 queens on a chess board so that no queen attacks another. (A queen attacks any piece in the same row, column, or diagonal.) This position is almost a solution, except for the two queens in the fourth and seventh columns that attack each other along the diagonal. (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.

Figure from http://aima.cs.berkeley.edu/figures.pdf

Simulated Annealing

- Optimization by Simulated Annealing
  - S. Kirkpatrick, C. D. Gelatt Jr., M. P. Vecchi
  - *Science* 13 May 1983:
  - Vol. 220, Issue 4598, pp. 671-680
  - https://science.sciencemag.org/content/220/4598/671

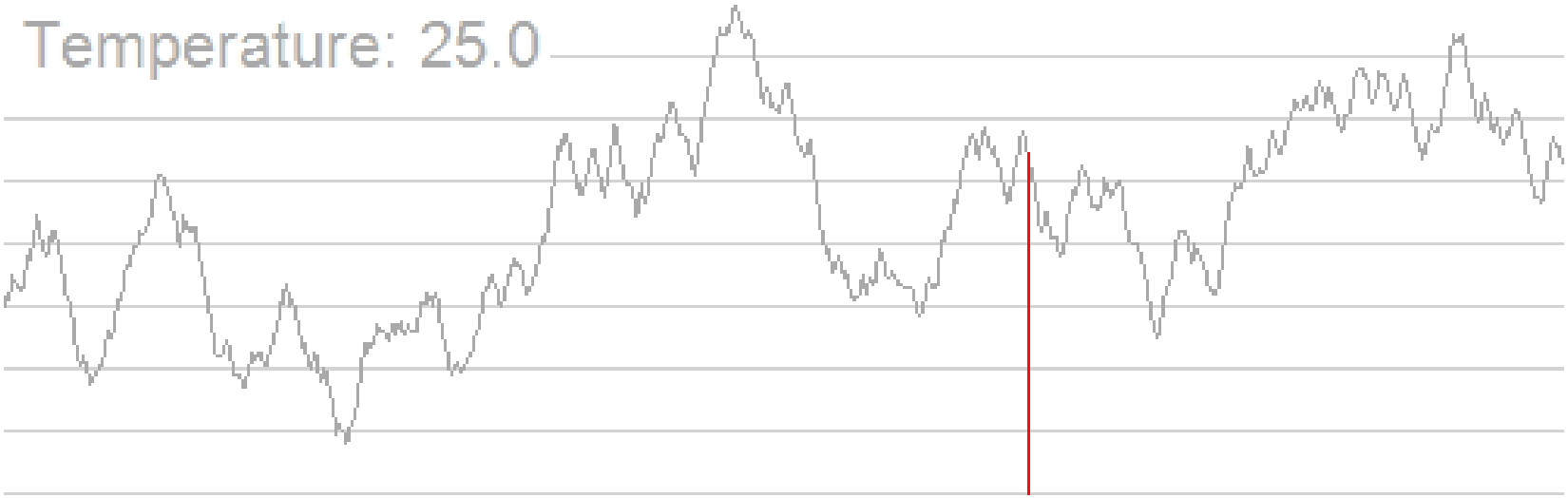
Simulated Annealing

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

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.

Figure from http://aima.cs.berkeley.edu/algorithms.pdf

Simulated Annealing Example



GIF from https://en.wikipedia.org/wiki/Simulated_annealing

Evolutionary Algorithms

- A population of states
- Each state has a "fitness" score
- Select candidate states from the population
- "Recombine" them to create an "offspring"
- Optionally, mutate the "offspring"

- Reading: https://doi.org/10.1007/s11042-020-10139-6

Evolutionary Algorithms – Genetic Algorithm

```
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.

Figure from http://aima.cs.berkeley.edu/algorithms.pdf

8-Queens Example

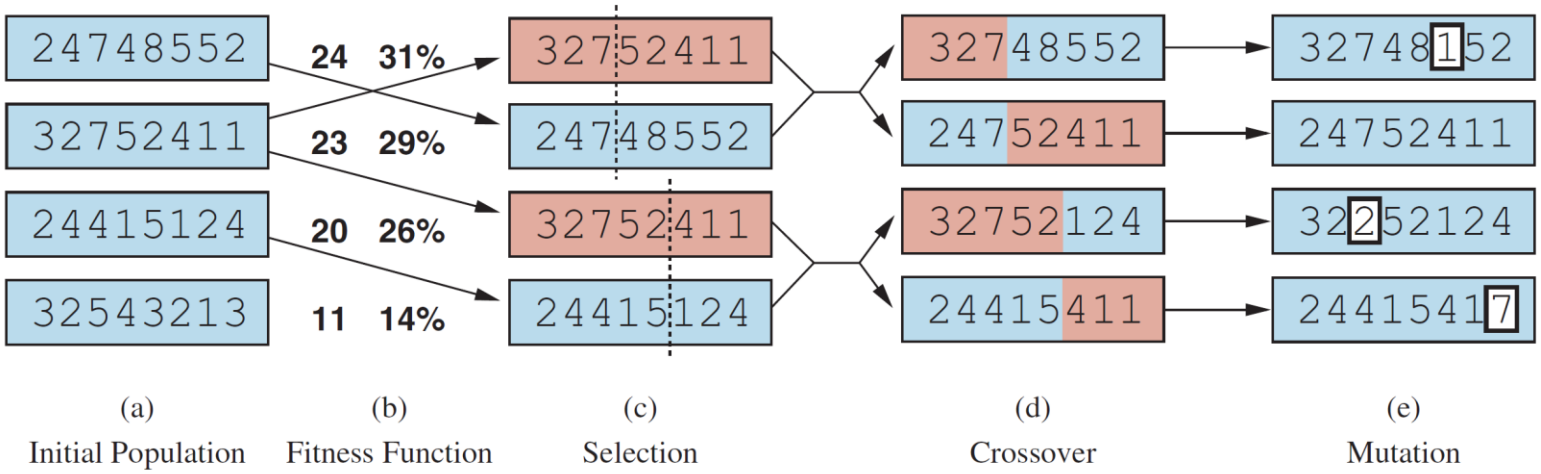


Figure 4.5 A genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by a fitness function in (b) resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

Figure from http://aima.cs.berkeley.edu/figures.pdf

8-Queens Example

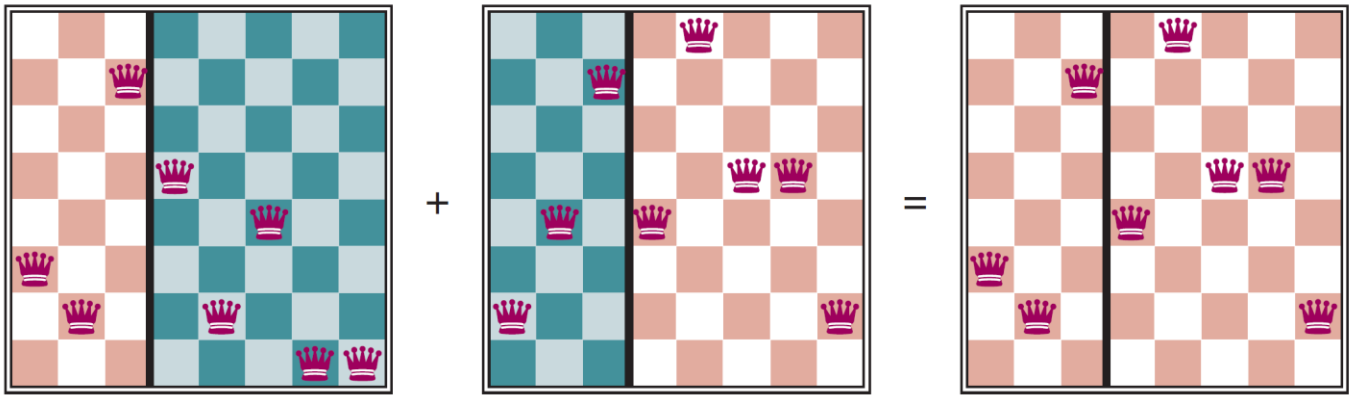


Figure 4.6 The 8-queens states corresponding to the first two parents in Figure **??**(c) and the first offspring in Figure **??**(d). The green columns are lost in the crossover step and the red columns are retained. (To interpret the numbers in Figure **??**: row 1 is the bottom row, and 8 is the top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

Figure from http://aima.cs.berkeley.edu/figures.pdf

Game Playing
Chapter 5

Games

- A **multiagent** and **competitive** environment
- Agents are trying to maximize their utilities at the expense of other agents' utilities
- Zero-sum games, where the sum of the utilities in the end is constant
  - E.g., win = +1, loss = -1, tie = 0
  - E.g., win = +1, loss = 0, draw = 0.5

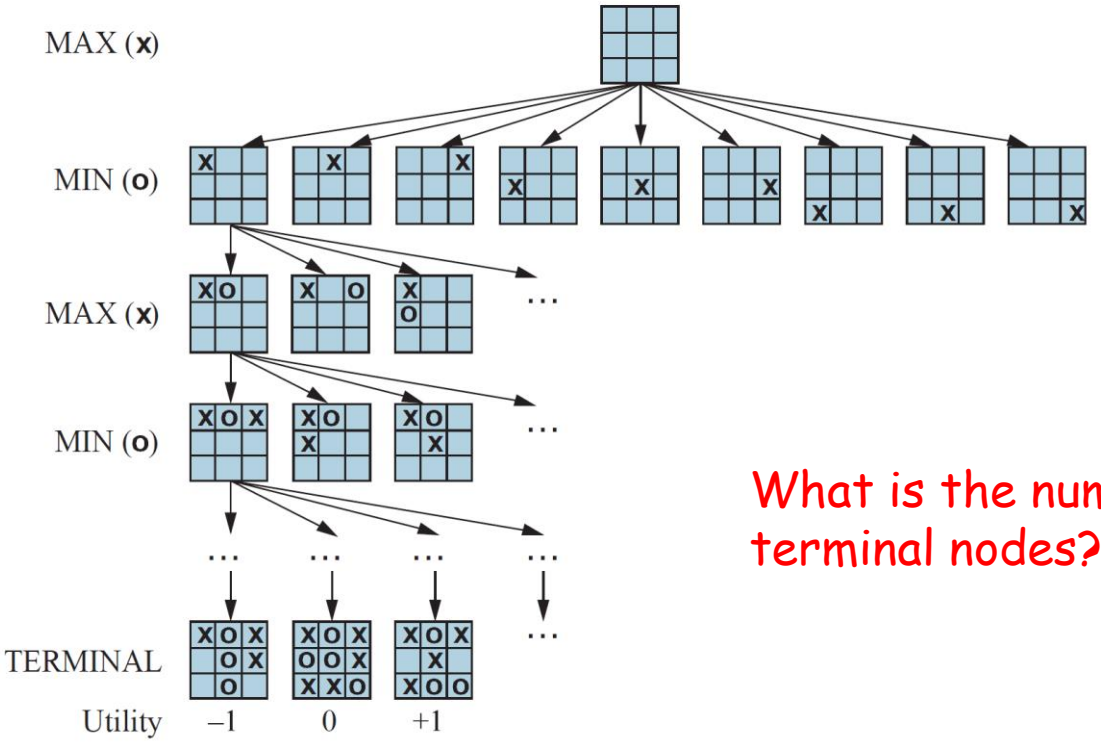
Games are hard to solve

- Chess
  - Average branching factor: ~35
  - If a game lasts 40 moves per player
    - Search tree depth is 80
    - Number of nodes: $35^{80} \approx 10^{123}$
    - Number of estimated atoms in the universe: 10^{80}
  - You still must make a decision before you can calculate the optimal move

Game Tree

- In a two-player game, there are two players: MAX and MIN
- MAX moves first
- Both are maximizing their own utility
- In a zero-sum game search tree
  - MAX node chooses a “max” move
  - MIN node choose a “min” move

TIC-TAC-TOE



What is the number of terminal nodes?

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.

Figure from http://aima.cs.berkeley.edu/figures.pdf

Optimal strategy

- An optimal strategy leads to outcomes at least as good as any other strategy when one is playing on *infallible* opponent
  - What if the opponent is not playing optimally?
- The **minimax** algorithm

MiniMax Example

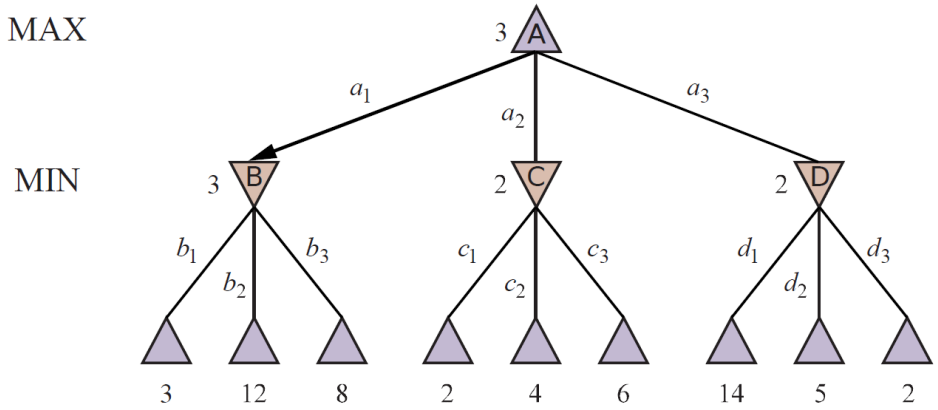
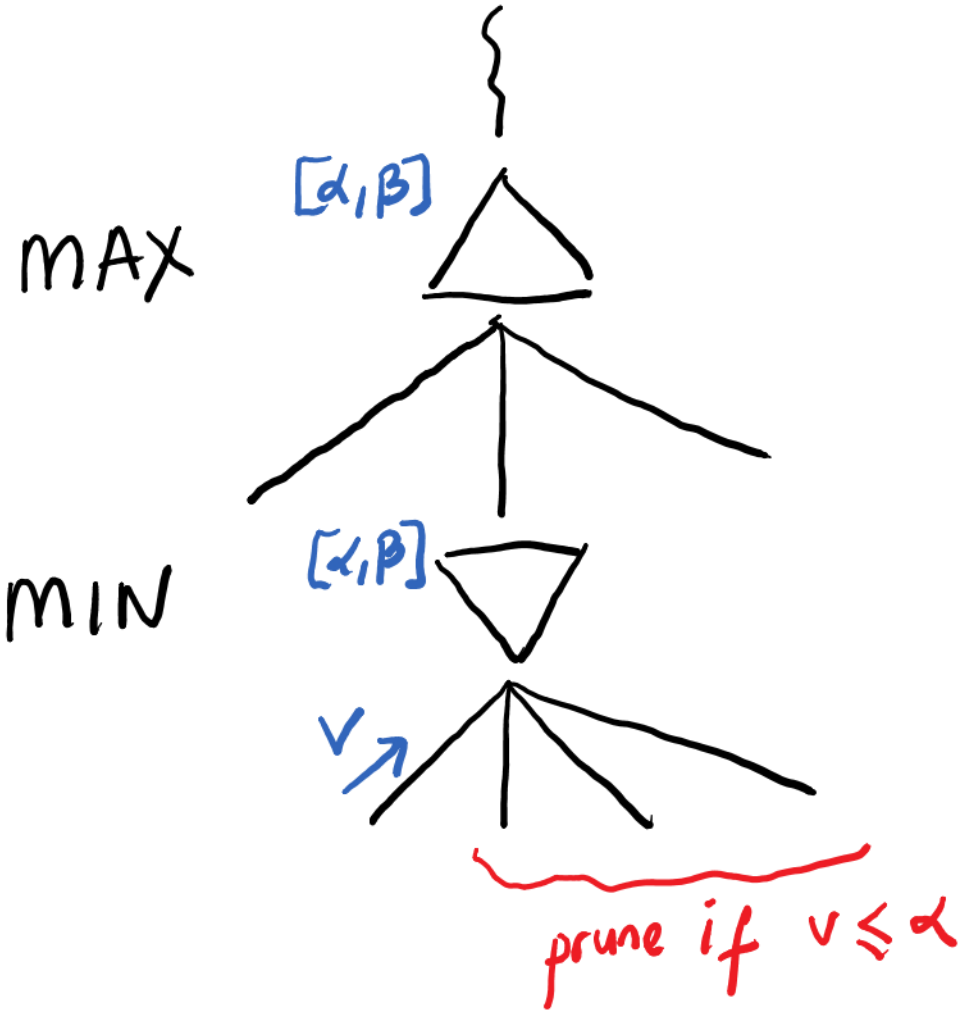


Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state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

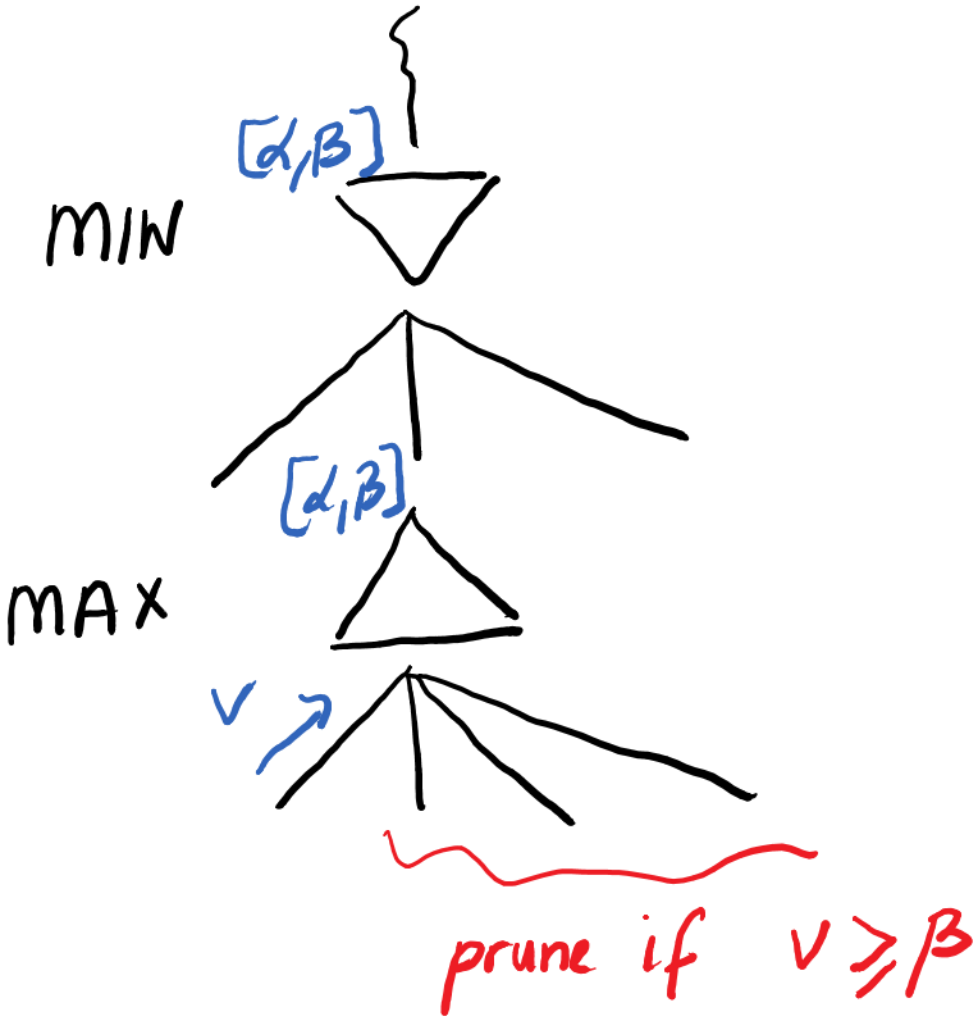
Alpha-beta Pruning

- Keep lower and upper bounds (α, β)
- MAX updates the lower bound α
- MIN updates the upper bound β
- Both pass the current bounds to their children
- MIN(α, β) prunes if $v \leq \alpha$; MAX already has a better choice α
- MAX(α, β) prunes if $v \geq \beta$; MIN already has a better choice β

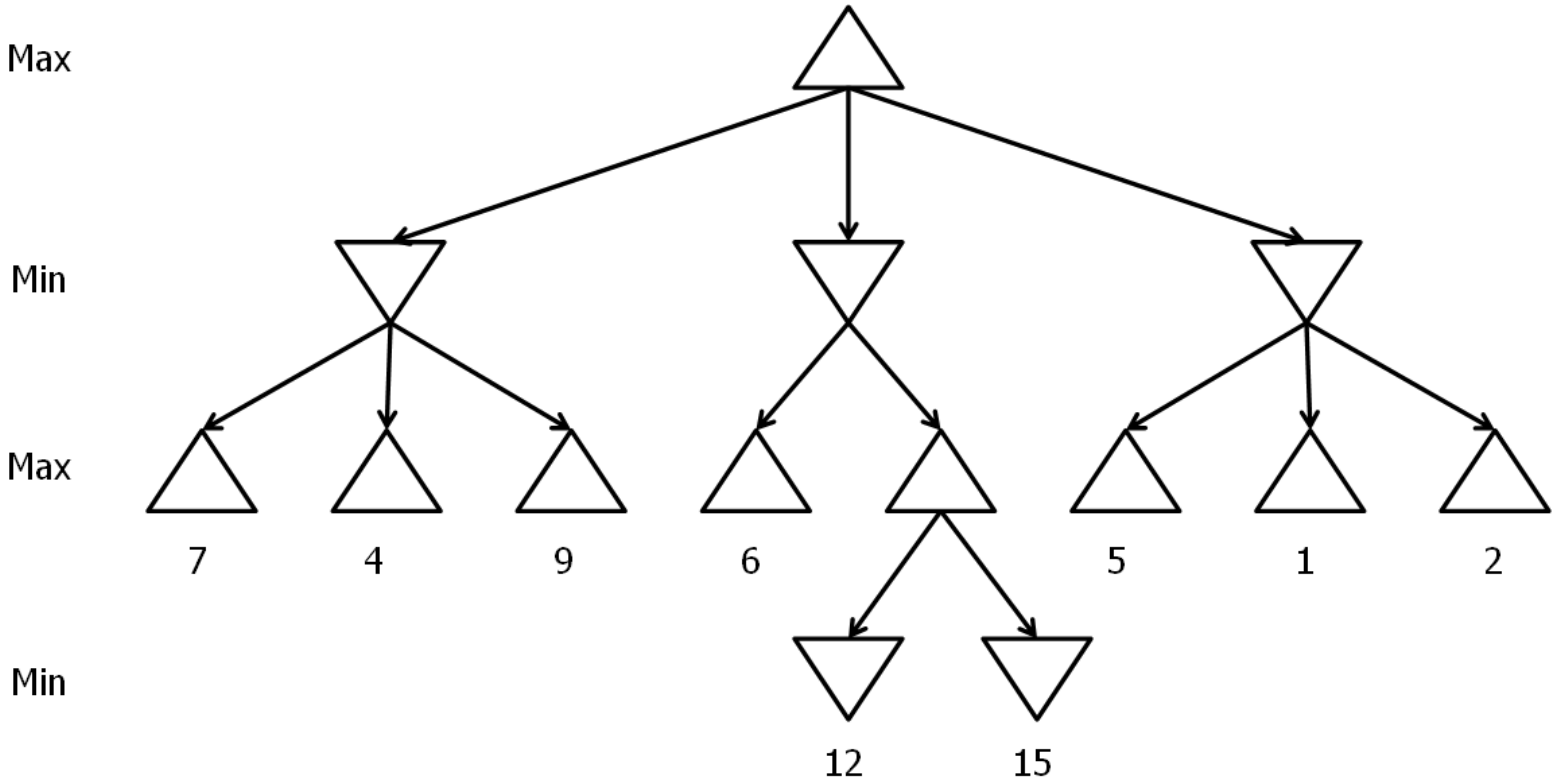
EXAMPLE: MIN PRUNES



EXAMPLE: MAX PRUNES



Alpha-Beta Pruning Example



Early Cut-Off

- To be able to prune, the alpha-beta algorithm has to visit some of the leaf nodes
  - Of course, this is not practical for most games
- A solution: early cut-off
  - How can we do this? Remember, the utility function is defined for the terminal states.

Heuristic Evaluation Functions

- An estimate of the utility is returned rather than the actual utility
  - Define features and a weighted feature combination function
    - e.g., EVAL(s) = $w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$
    - Chess
      - Pawn: 1, Knight or bishop: 3, rook: 5, queen: 9
  - Use machine learning to learn and predict the values of the states

MONTE-CARLO TREE SEARCH

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.

Monte-Carlo Tree Search

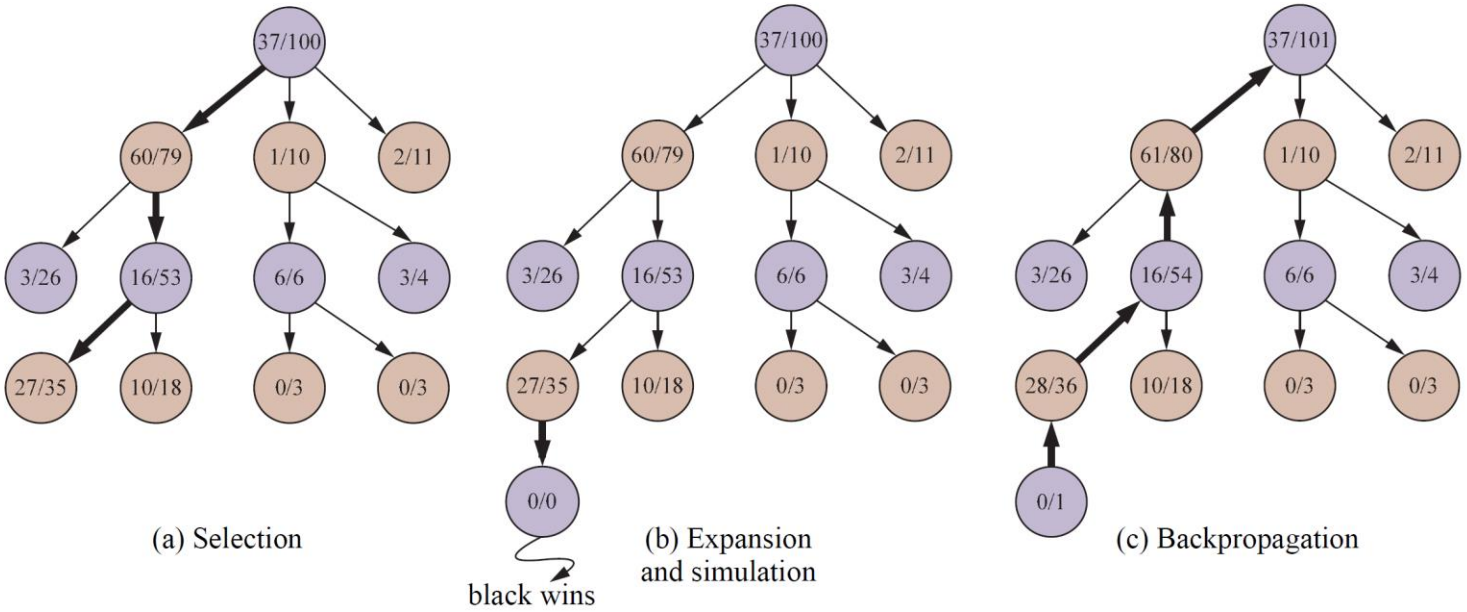


Figure 5.10 One iteration of the process of choosing a move with Monte Carlo tree search (MCTS) using the upper confidence bounds applied to trees (UCT) selection metric, shown after 100 iterations have already been done. In (a) we select moves, all the way down the tree, ending at the leaf node marked 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.