5 Lecture 5: Local Search- Simulated Annealing, Genetic Algorithm

5.1 Introduction to Simulated Annealing 47
5.2 How does simulated annealing navigate the solution space: 48
5.2.1 Probability of Accepting a New State 48
5.2.2 Cooling Schedule 49
5.2.3 Random Selection of Neighbors 49
5.2.4 Mathematical Convergence 49
5.3 Example Problem: 49
5.3.1 Traveling Salesman Problem (TSP) 49
5.3.2 Genetic algorithm 50
5.3.3 Algorithm 51
5.3.4 Explanation of the Pseudocode 51
5.4 Mutation 53
5.4.1 Purpose of Mutation 53
5.4.2 How Mutation Works 53
5.4.3 Common Types of Mutation 53
5.4.4 Mutation Rate 54
5.4.5 Advantages and Applications 54
5.5 Examples 54
5.5.1 Traveling Salesman Problem (TSP) 54
5.5.2 0/1 Knapsack Problem 56
5.5.3 8-Queen Problem 57
5.5.4 Graph Coloring Problem: 59
5.5.5 Max-cut Problem 60
5.6 Application of Genetic Algorithm in Machine Learning 61
5.6.1 Problem Setup: Feature Selection for Predictive Modeling 61
5.6.2 Problem Setup: Hyperparameter Optimization for a Neural Network 62
5.6.3 Genetic Algorithm in AI Games: 64
5.6.4 Genetic algorithms in Finance 64

5.1 Introduction to Simulated Annealing

Simulated Annealing (SA) is a probabilistic technique for approximating the global optimum of a given function. It is particularly useful for solving large optimization problems where other methods might be too slow or get stuck in local optima.
The technique is inspired by the physical process of annealing in metallurgy, where metals are heated to a high temperature and then cooled according to a controlled schedule to achieve a more stable crystal structure.
The method was developed by S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi in 1983, based on principles of statistical mechanics.
Simulated Annealing is an optimization technique that simulates the heating and gradual cooling of materials to minimize defects and achieve a stable state with minimum energy.

It starts with a randomly generated initial solution and a high initial temperature, which allows the **acceptance of suboptimal solutions** to explore the solution space widely.
The algorithm iterates by generating small modifications to the current solution, evaluating the cost changes, and **probabilistically deciding** whether to accept the new solution based on the current temperature.
The temperature is gradually reduced according to a predetermined cooling schedule, which **decreases the likelihood of accepting worse solutions** and helps fine-tune towards an optimal solution.
The process concludes once a stopping criterion, like a specified number of iterations, a minimal temperature, or a quality threshold of the solution, is met.

Algorithm 2 Algorithm Simulated_Annealing

```
Input: initial_solution, initial_temperature, cooling_rate, stopping_temperature
Output: best_solution_found
current_solution ← initial_solution
current_temperature ← initial_temperature
best_solution ← current_solution
while:  current_temperature > stopping_temperature
  new_solution ← generate_neighbor(current_solution)
  cost_difference ← cost(new_solution) - cost(current_solution)
  if cost_difference < 0 or exp(-cost_difference / current_temperature) > random(0, 1) current_solution ← new_solution
      current_solution ← new_solution
  if cost(new_solution) < cost(best_solution)
      best_solution ← new_solution
  current_temperature ← current_temperature × cooling_rate
return best_solution
```

Key Parameters:

- **Initial Temperature:** High enough to allow exploration.
- **Cooling Rate:** Determines how quickly the temperature decreases.
- **Stopping Temperature:** Low enough to stop the process once the system is presumed to have stabilized.

5.2 How does simulated annealing navigate the solution space:

5.2.1 Probability of Accepting a New State

The key mathematical concept in simulated annealing is the probability of accepting a new state S' from a current state S . This probability is determined using the Metropolis-Hastings algorithm, which is defined as follows:

$$P(\text{accept } S') = \min\left(1, e^{-\frac{\Delta E}{T}}\right)$$

where:

- $\Delta E = E(S') - E(S)$ is the change in the objective function (cost or energy) from the current state S to the new state S'.

- T is the current temperature.
- e is the base of the natural logarithm.

  The equation $e^{-\frac{\Delta E}{T}}$ is crucial as it controls the acceptance of new solutions:
- If $\Delta E < 0$ (meaning S' is a better solution than S), then $e^{-\frac{\Delta E}{T}} > 1$, and the new solution is always accepted $(\min(1, \text{value}) = 1)$.
- If $\Delta E > 0$ (meaning §' is worse), the new solution is accepted with a probability less than 1. This probability decreases as ΔE increases or as T decreases.

5.2.2 Cooling Schedule

The cooling schedule is a rule or function that determines how the temperature T decreases over time. It is typically a function of the iteration number k. A common choice is the exponential decay given by:

$$T(k) = T_0 \cdot \alpha^k$$

where:
- T_0 is the initial temperature.
- α is a constant such that $0 < \alpha < 1$, often close to 1.
- k is the iteration index.

The choice of α and T_0 influences the convergence of the algorithm. A slower cooling (higher α) allows more thorough exploration of the solution space but takes longer to converge.

5.2.3 Random Selection of Neighbors

The random selection of a neighbor S' is typically governed by a neighborhood function, which defines possible transitions from any given state S. The randomness allows the algorithm to explore the solution space non-deterministically, which is essential for escaping local optima.

5.2.4 Mathematical Convergence

Theoretically, given an infinitely slow cooling (i.e., $\alpha \to 1$ and infinitely many iterations), simulated annealing can converge to a global optimum. This stems from the ability to continue exploring new states with a non-zero probability, provided the cooling schedule allows sufficient time at each temperature level for the system to equilibrate.

Simulated annealing integrates concepts from statistical mechanics with optimization through a controlled random walk. This walk is guided by the probabilistic acceptance of new solutions, which balances between exploiting better solutions and exploring the solution space broadly, moderated by a temperature parameter that systematically decreases over time.

5.3 Example Problem:

5.3.1 Traveling Salesman Problem (TSP)

Let's consider an example of solving a small Traveling Salesman Problem (TSP) using simulated annealing. The TSP is a classic optimization problem where the goal is to find the shortest possible route that visits a set of cities and returns to the origin city, visiting each city exactly once.

Problem Setup

Suppose we have a set of five cities, and the distances between each pair of cities are given by the following symmetric distance matrix:

	A	B	C	D	E
A	0	12	10	19	8
B	12	0	3	5	6
C	10	3	0	6	7
D	19	5	6	0	4
E	8	6	7	4	0

Objective

Find the shortest path that visits each city exactly once and returns to the starting city.

Simulated Annealing Steps

1. **Initialization:** Randomly generate an initial route, say, $A \to B \to C \to D \to E \to A$.
2. **Heating:** Set an initial high temperature to allow significant exploration. For instance, start with a temperature of 100.
3. **Iteration:** - **Generate a Neighbor:** Create a new route by making a small change to the current route, such as swapping two cities. For instance, swap cities B and D to form a new route $A \to D \to C \to B \to E \to A$.

   - **Calculate the Change in Cost:** Determine the total distance of the new route and compare it with the current route.

   - **Acceptance Decision:** Use the Metropolis criterion to decide probabilistically whether to accept the new route based on the change in cost and the current temperature.
4. **Cooling:** Reduce the temperature based on a cooling schedule, e.g., multiply the temperature by 0.95 after each iteration.
5. **Termination:** Repeat the iteration process until the temperature is low enough or a fixed number of iterations is reached. Assume the stopping condition is when the temperature drops below 1 or after 1000 iterations.

Example Calculation

Assuming the first iteration starts with the initial route and the randomly generated neighbor as described:

- Current Route: $A \to B \to C \to D \to E \to A$ (Distance $= 12+3+6+4+8 = 33$) - New Route: $A \to D \to C \to B \to E \to A$ (Distance $= 19+6+3+6+8 = 42$)

The change in cost ΔE is 42 - 33 = 9. The probability of accepting this worse solution at temperature 100 is $e^{-9/100} \approx 0.91$. A random number is generated between 0 and 1; if it is less than 0.91, the new route is accepted, otherwise, the algorithm retains the current route.

This process is repeated, with the temperature decreasing each time, until the termination conditions are met. The result should be a route that approaches the shortest possible loop connecting all five cities.

5.3.2 Genetic algorithm

Genetic algorithms (GAs) are a class of optimization algorithms inspired by the principles of natural selection and genetics. These algorithms are used to solve search and optimization problems and are particularly effective for complex issues that are difficult to solve using traditional methods.

Genetic algorithms mimic the process of natural evolution, embodying the survival of the fittest among possible solutions. The core idea is derived from the biological mechanisms of reproduction, mutation, recombination, and selection. These biological concepts are translated into computational steps that help in finding optimal or near-optimal solutions to problems across a wide range of disciplines including engineering, economics, and artificial intelligence.

5.3.3 Algorithm

Algorithm 3 Genetic Algorithm

```
Input: Population size, fitness function, mutation rate, crossover rate, maximum generations
Output: Best solution found
begin
  Initialize population with random candidates
  Evaluate the fitness of each candidate
  while termination condition not met do
    Select parents from the current population
    Perform crossover on parents to create new offspring
    Perform mutation on offspring
    Evaluate the fitness of new offspring
    Select individuals for the next generation
    best solution in new generation > best solution so far
      Update best solution found
  end while
  return best solution found
end
```

5.3.4 Explanation of the Pseudocode

Initialize Population: A genetic algorithm begins with a population of randomly generated individuals. Each individual, or chromosome, represents a possible solution to the problem. Depending on the problem the chromosomes are encoded.

Evaluate Fitness: Each solution in the population is assessed using the fitness function to determine how well it solves the problem. Depending on the problem the fitness function will be measured.

Selection: This step involves choosing the fitter individuals to reproduce. Selection can be done in various ways, such as Truncation Selection, tournament selection, roulette wheel selection, or rank selection. In this course we are using truncation selection, where we select the fittest 3/4th of the population.

Truncation Selection
Description: Only the top-performing fraction of the population is selected to reproduce.
Procedure: Rank individuals by fitness, then select the top $x\%$ to become parents of the next generation.
Pros and Cons: Very straightforward and ensures high-quality genetic material is passed on, but can quickly reduce genetic diversity.

Crossover (Recombination): Pairs of individuals are crossed over at a randomly chosen point to produce offspring. The crossover rate determines how often crossover will occur. Two common types of crossover techniques are single-point crossover and two-point (or double-point) crossover.

- Single-Point Crossover:

  In single-point crossover, a single crossover point is randomly selected on the parent chromosomes. The genetic material (bits, characters, numbers, depending on the encoding of the solution) beyond that point in the chromosome is swapped between the two parents. This results in two new offspring, each carrying some genetic material from both parents.

  Procedure:

  Select a random point on the chromosome. The segments of the chromosomes after this point are swapped between the two parents.

  Example:

  Suppose we have two binary strings:

  Parent 1: 110011

  Parent 2: 101010

  Assuming the crossover point is after the third bit, the offspring would be:

  Offspring 1: 110010 (first three bits from Parent 1, last three bits from Parent 2)

  Offspring 2: 101011 (first three bits from Parent 2, last three bits from Parent 1)

- Two-Point Crossover:

  Two-point crossover involves two points on the parent chromosomes, and the genetic material located between these two points is swapped between the parents. This can introduce more diversity compared to single-point crossover because it allows the central segment of the chromosome to be exchanged, potentially combining more varied genetic information from both parents.

  Procedure:

  Select two random points on the chromosome, ensuring that the first point is less than the second point.

  Swap the segments between these two points from one parent to the other.

  Example:

  Continuing with the same parent strings:

  Parent 1: 110011

  Parent 2: 101010

  Lets choose two crossover points, between the second and fifth bits. The offspring produced would be:

  Offspring 1: 100010 (first two bits from Parent 1, middle segment from Parent 2, last bit from Parent 1)

  Offspring 2: 111011 (first two bits from Parent 2, middle segment from Parent 1, last bit from Parent 2)

5.4 Mutation

With a certain probability (mutation rate), mutations are introduced to the offspring to maintain genetic diversity within the population.

The purpose of mutation is to maintain and introduce genetic diversity into the population of candidate solutions, helping to prevent the algorithm from becoming too homogeneous and getting stuck in local optima.

5.4.1 Purpose of Mutation

1. **Introduce Variation:** Mutation introduces new genetic variations into the population by altering one or more components of genetic sequences, ensuring a diversity of genes.
2. **Prevent Local Optima:** By altering the genetic makeup of individuals, mutation prevents the population from converging too early on a suboptimal solution.
3. **Explore New Areas:** It enables the algorithm to explore new areas of the solution space that may not be reachable through crossover alone.

5.4.2 How Mutation Works

Mutation operates by making small random changes to the genes of individuals in the population. In the context of genetic algorithms, an individuals genome might be represented as a string of bits, characters, numbers, or other data structures, depending on the problem being solved.

5.4.3 Common Types of Mutation

1. **Bit Flip Mutation (for binary encoding):**
   - **Procedure:** Each bit in a binary string has a small probability of being flipped (0 changes to 1, and vice versa).
   - **Example:** A binary string '110010' might mutate to '110011' if the last bit is flipped.
2. **Random Resetting (for integer or real values):**
   - **Procedure:** A selected gene is reset to a new value within its range.
   - **Example:** In a string of integers $[4, 12, 7, 1]$, the third element 7 might mutate to 9.
3. **Swap Mutation:**
   - **Procedure:** Two genes are selected and their positions are swapped. This is often used in permutation-based encodings.
   - **Example:** In an array $[3, 7, 5, 8]$, swapping the second and fourth elements results in $[3, 8, 5, 7]$.
4. **Scramble Mutation:**
   - **Procedure:** A subset of genes is chosen and their values are scrambled or shuffled randomly.
   - **Example:** In an array $[3, 7, 5, 8, 2]$, scrambling the middle three elements might result in $[3, 5, 8, 7, 2]$.
5. **Uniform Mutation (for real-valued encoding):**

- **Procedure:** Each gene has a fixed probability of being replaced with a uniformly chosen value within a predefined range.
- **Example:** In an array of real numbers $[0.5, 1.3, 0.9]$, the second element 1.3 might mutate to 1.1.

5.4.4 Mutation Rate

The mutation rate is a critical parameter in genetic algorithms. It defines the probability with which a mutation will occur in an individual gene. A higher mutation rate increases diversity but may also disrupt highly fit solutions, whereas a lower mutation rate might not provide enough diversity, leading to premature convergence. The optimal mutation rate often depends on the specific problem and the characteristics of the population.

Evaluate New Offspring: The fitness of each new offspring is calculated.

Generation Update: The algorithm decides which individuals to keep for the next generation. This can be a mix of old individuals (elitism) and new offspring.

Termination Condition: The algorithm repeats until a maximum number of generations is reached, or if another stopping criterion is satisfied (like a satisfactory fitness level).

Return Best Solution: The best solution found during the evolution is returned.

Note: For exam problems if you are asked to simulate, unless otherwise instructed, start with 4 chromosomes in the population, select best 3 at each step, crossover between the best and the other two selected

5.4.5 Advantages and Applications

Genetic algorithms are particularly useful when:

- The search space is large, complex, or poorly understood.
- Traditional optimization and search techniques fail to find acceptable solutions.
- The problem is dynamic and changes over time, requiring adaptive solutions.

5.5 Examples

5.5.1 Traveling Salesman Problem (TSP)

Problem Setup:

We have five cities: A, B, C, D and E. The distance matrix given:

	A	B	C	D	E
A	0	2	9	10	7
B	2	0	6	5	8
C	9	6	0	12	10
D	10	5	12	0	15
E	7	8	10	15	0

Genetic Algorithm Setup

Chromosome Representation: A permutation of $[A, B, C, D, E]$.

Fitness Function: The fitness of each chromosome is calculated as the inverse of the total route distance. The shorter the route, the higher the fitness.

$$\boxed{\text{Fitness} = \frac{1}{\text{Route Distance}}}$$

Example Calculation For the chromosome $[A, B, C, D, E]$:

Distance $(A - -B) : 2$
Distance $(B - -C) : 6$
Distance $(C - -D) : 12$
Distance $(D - -E) : 15$
Distance ($E - -A$ to complete the loop): 7
Total Distance: $2 + 6 + 12 + 15 + 7 = 42$
Fitness Calculation:

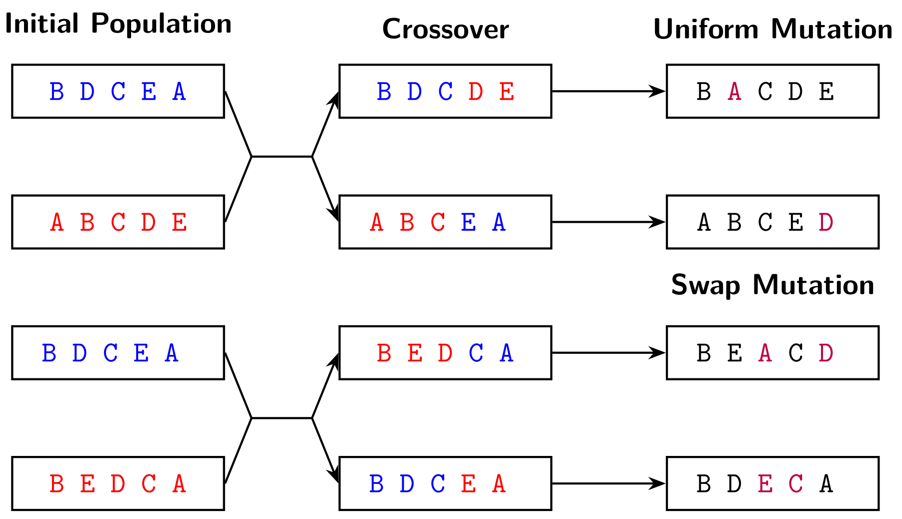
$$\text{Fitness} = \frac{1}{\text{Total Distance}} = \frac{1}{42}$$

Initial Population
Chromosome $1 : [A, B, C, D, E]$
Chromosome $2 : [B, E, D, C, A]$
Chromosome $3 : [E, D, B, A, C]$
Chromosome $4 : [B, D, C, E, A]$

Fitness Calculation
Chromosome 1: Total Distance = 42, Fitness = $\frac{1}{42}$
Chromosome 2: Total Distance = 45, Fitness = $\frac{1}{45}$
Chromosome 3: Total Distance = 49, Fitness = $\frac{1}{49}$
Chromosome 4: Total Distance = 39, Fitness = $\frac{1}{39}$

Iteration 1
Selection:
Select top 3/4th based on fitness: Chromosome 4, Chromosome 1 and Chromosome 2.

Crossover (One-Point):



Parents: Chromosome 4 and Chromosome 1
Crossover Point: After 2nd position
Offspring: $[B, D, C, D, E]$ and $[A, B, C, E, A]$

Parents: Chromosome 4 and Chromosome 2
Crossover Point: After 1st position
Offspring: $[B, E, D, C, A]$ and $[B, D, C, E, A]$

Mutation
$[B, D, C, D, E] \rightarrow [B, A, C, D, E]$ (Uniform mutation)
$[A, B, C, E, A] \rightarrow [A, B, C, E, D]$ (Uniform mutation)
$[B, E, D, C, A] \rightarrow [B, E, A, C, D]$ (Swap mutation)
$[B, D, C, E, A] \rightarrow [B, D, E, C, A]$ (Swap mutation)

Note: It is better to use one specific type of mutation in your simulation. Always mention what type of mutation you are using.
New Population:
Replace previous population with the new sets chromosomes $[B, A, C, D, E]$, $[A, B, C, E, D]$, $[B, E, A, C, D]$, $[B, D, E, C, A]$.

Repeat:
This process is repeated over several generations to find the chromosome with the highest fitness, representing the shortest possible route that visits each city exactly once and returns to the starting point.

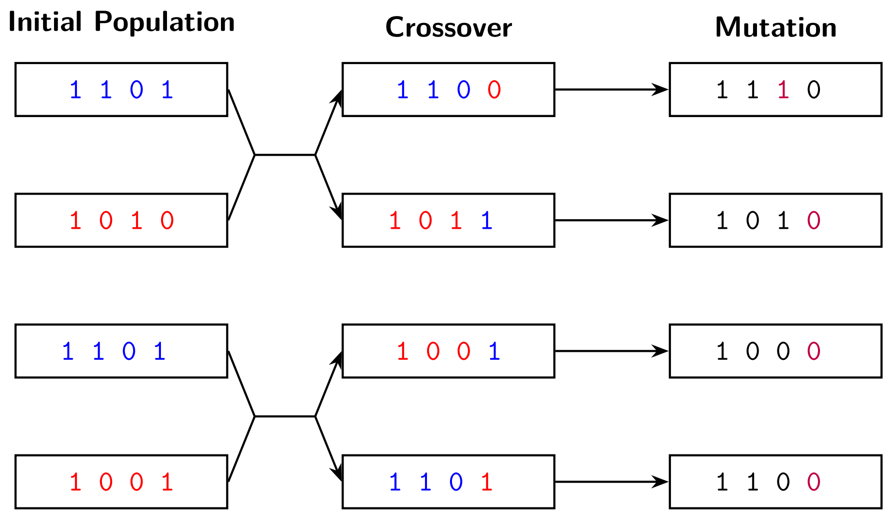
5.5.2 0/1 Knapsack Problem

Objective: Maximize the value of items packed in a knapsack without exceeding the weight limit.

Constraints:
Maximum weight the knapsack can hold: 15 kg Items:
Item 1: Weight = 6 kg, Value = $\$30$
Item 2: Weight = 3 kg, Value = $\$14$
Item 3: Weight = 4 kg, Value = $\$16$
Item 4: Weight = 2 kg, Value = $\$9$

Genetic Algorithm Implementation
Representation: Each chromosome is a string of bits, where each bit represents whether an item is included (1) or not (0). For example, '1010' means Items 1 and 3 are included, while Items 2 and 4 are not.
Step 1: Initialization
Generate a random initial population of solutions (chromosomes).
Population Size: 4
Chromosomes: $101, 1010, 0110, 1001$
Step 2: Fitness Evaluation
Calculate the total value of each chromosome, ensuring the weight does not exceed the capacity.
1101: Value = $\$30 + \$14 + \$9 = \$53$, Weight = 6 + 3 + 2 = 11 kg 1010: Value = $\$30 + \$16 = \$46$, Weight = 6 + 4 = 10 kg 0110: Value = $\$14 + \$16 = \$30$, Weight = 3 + 4 = 7 kg

Step 3: Selection (Truncation Selection)
Select the top 3/4th of chromosomes based on their value.
Selected: $1101(\$53), 1010(\$46), 1001(\$39)$
Highest fitness chromosome: 1101
Step 4: Crossover (One-Point)
Perform a one-point crossover between the selected pairs (1101 and 1010) and (1101 and 1001).
Crossover point: Between the third and fourth bits.
Offspring 1: 1100 (from 1101 and 1010)
Offspring 2: 1011 (from 1010 and 1101)
Offspring 1: 1101 (from 1101 and 1001)
Offspring 2: 1001 (from 1001 and 1101)
Step 5: Mutation
Apply a mutation by flipping a random bit in each offspring.
1100 $\rightarrow$1110 (mutate the third bit)
1011 $\rightarrow$1010 (mutate the last bit)
1101 $\rightarrow$1100 (mutate the last bit)
1001 $\rightarrow$1000 (mutate the last bit)



Step 6: New Population
Replace the previous population with the set of new chromosomes.
New Population: 1110, 1010, 1100, 1000
Step 7: Repeat
This process is repeated over several generations to find the chromosome with the highest fitness, representing the maximum value found.

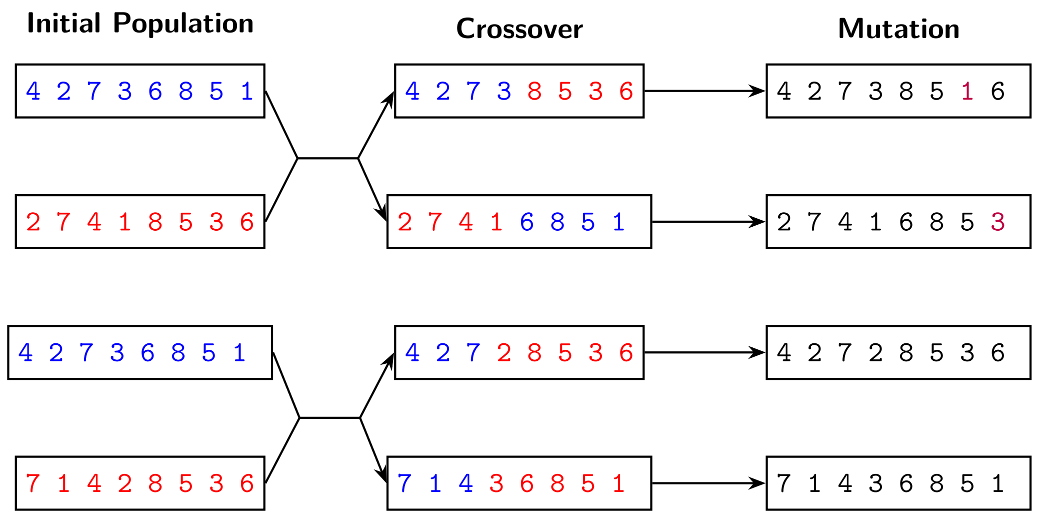
5.5.3 8-Queen Problem

The 8-queens problem involves placing eight queens on an 8x8 chessboard so that no two queens threaten each other. This means no two queens can share the same row, column, or diagonal.

Chromosome Representation: Each chromosome can be represented as a string or array of 8 integers, each between 1 and 8, representing the row position of the queen in each column.

Initial Population:
We randomly generate a small population of 4 individuals for simplicity:
Chromosome A: $[4, 2, 7, 3, 6, 8, 5, 1]$
Chromosome B: $[2, 7, 4, 1, 8, 5, 3, 6]$
Chromosome C: $[7, 1, 4, 2, 8, 5, 3, 6]$
Chromosome D: $[5, 3, 1, 7, 2, 8, 6, 4]$
Fitness Function
Fitness is calculated based on the number of non-attacking pairs of queens. The maximum score for 8 queens is 28 (i.e., no two queens attack each other).

Iteration 1
Step 1: Calculate Fitness
A: 26 (non-attacking pairs)
B: 24
C: 24
D: 23
Step 2: Selection
Select the top 3/4th of chromosomes based on their fitness. This results in selecting:
Chromosome A: $[4, 2, 7, 3, 6, 8, 5, 1], 26$
Chromosome B: $[2, 7, 4, 1, 8, 5, 3, 6], 24$
Chromosome C: $[7, 1, 4, 1, 8, 5, 3, 6], 24$
Step 3: Crossover (Single Point)



Crossover between A and B: Crossover at position 4
Offspring 1: $[4, 2, 7, 3, 8, 5, 3, 6]$
Offspring 2: $[2, 7, 4, 1, 6, 8, 5, 1]$
Crossover between A and C: Crossover at position 3

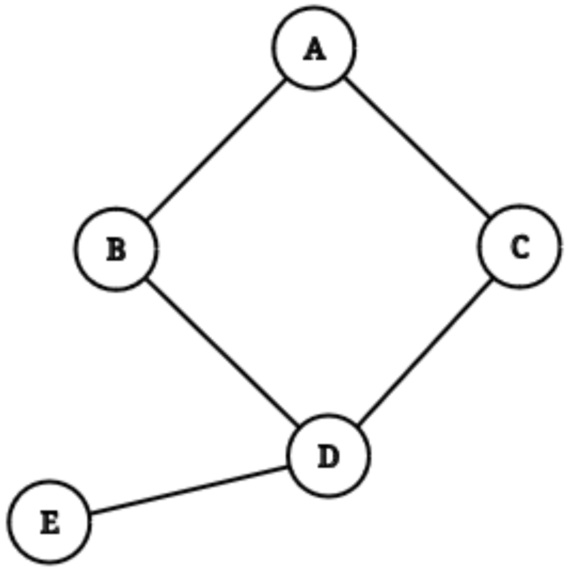
Offspring 3: $[4, 2, 7, 1, 8, 5, 3, 6]$
Offspring 4: $[2, 7, 4, 3, 6, 8, 5, 1]$
Step 4: Mutation
Offspring 1 after mutation: $[4, 2, 7, 3, 8, 5, 1, 6]$ (Uniform mutation at index 7)
Offspring 2 after mutation: $[2, 7, 4, 1, 6, 8, 5, 3]$ (Uniform mutation at index 8)
Offspring 3 after mutation: $[4, 2, 7, 1, 8, 5, 3, 6]$
Offspring 4 after mutation: $[2, 7, 4, 3, 6, 8, 5, 1]$
Step 5: Create new Population replacing the old population
New Population:
$[4, 2, 7, 3, 8, 5, 1, 6]$
$[2, 7, 4, 1, 6, 8, 5, 3]$
$[4, 2, 7, 1, 8, 5, 3, 6]$
$[2, 7, 4, 3, 6, 8, 5, 1]$
Repeat until the configuration with highest fitness (non-attacking pair) is found.

5.5.4 Graph Coloring Problem:

The graph coloring problem involves assigning colors to the vertices of a graph such that no two adjacent vertices share the same color, and the goal is to minimize the number of colors used.

Graph Description

Consider a simple graph with 5 vertices (A, B, C, D, E) and the following edges: AB, AC, BD, CD, DE.



Genetic Algorithm Setup for Graph Coloring

Chromosome Representation: Each chromosome is an array where each position represents a vertex and the value at that position represents the color of that vertex. For simplicity, we'll use numerical values to represent different colors.

Initial Population:

We randomly generate a small population of 4 solutions:
Chromosome A: $[1, 2, 3, 1, 2]$

Chromosome B: $[2,3,1,2,3]$
Chromosome C: $[1,2,1,3,2]$
Chromosome D: $[3,1,2,3,1]$

Fitness Function Fitness is determined by the number of properly colored edges (i.e., edges connecting vertices of different colors). The maximum fitness for this graph is 5 (one for each edge).

Iteration 1

Calculate Fitness

Calculate the fitness based on the coloring rules:
A: Fitness = 5 (all edges correctly colored)
B: Fitness = 5
C: Fitness = 3 (CD edge is incorrectly colored)
D: Fitness = 4 (DE edge is incorrectly colored)

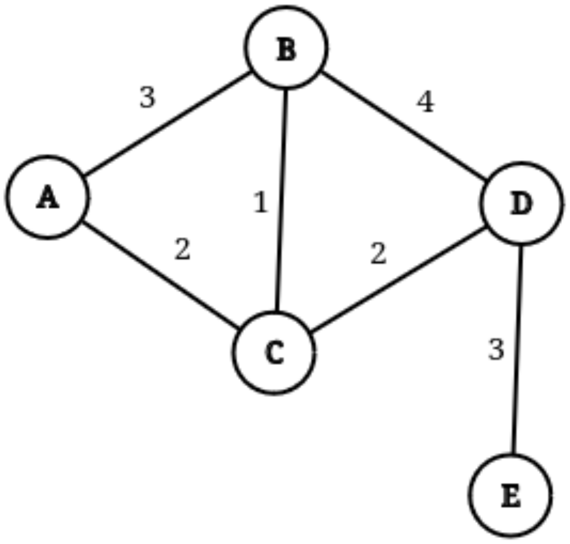
Carry on selection, crossover, mutation and population replacement as before.

5.5.5 Max-cut Problem

The Max-Cut problem is a classic problem in computer science and optimization in which the goal is to divide the vertices of a graph into two disjoint subsets such that the number of edges between the two subsets is maximized. Here, I'll outline how to solve this problem using a genetic algorithm (GA).

Problem Setup

Consider a graph with 5 vertices (A, B, C, D, E) and the following edges with given weights: AB = 3, AC = 2, BC = 1, BD = 4, CD = 2, DE = 3



The goal is to find a division of these vertices into two sets that maximizes the sum of the weights of the edges that have endpoints in each set.

Genetic Algorithm Setup for Max-Cut

Chromosome Representation: Each chromosome is a string of bits where each bit represents a vertex. A bit of '0' might represent the vertex being in set X and '1' in set Y.

Initial Population:

Randomly generate a small population of 4 solutions:

Chromosome A: 10101 (Vertices A, C, E in set Y; B, D in set X)
Chromosome B: 01010 (Vertices B, D in set Y; A, C, E in set X)
Chromosome C: 11001 (Vertices A, B, E in set Y; C, D in set X)
Chromosome D: 00110 (Vertices C, D in set Y; A, B, E in set X)

Fitness Function

Fitness is determined by the sum of the weights of the edges between the two sets. For a chromosome, calculate the sum of weights for edges where one endpoint is '0' and the other is '1'.

Calculate the fitness for each chromosome based on their separation:
A: Edges AC, BD, DE = 2 + 4 + 3 = 9
B: Edges AB, CD = 3 + 2 = 5
C: Edges AC, BD = 2 + 4 = 6
D: Edges BC, DE = 1 + 3 = 4

Carry on selection, crossover, mutation, and replacement as before.

5.6 Application of Genetic Algorithm in Machine Learning

5.6.1 Problem Setup: Feature Selection for Predictive Modeling

Suppose you're working with a medical dataset aimed at predicting the likelihood of patients developing a certain disease. The dataset contains hundreds of features, including patient demographics, laboratory results, and clinical parameters. Not all of these features are relevant for the prediction task, and some may introduce noise or redundancy.

Genetic Algorithm Setup for Feature Selection

Chromosome Representation: Each chromosome in the GA represents a possible solution to the feature selection problem. Specifically, a chromosome can be encoded as a binary string where each bit represents the presence (1) or absence (0) of a corresponding feature in the dataset.

Initial Population: Generate an initial population of chromosomes randomly, where each chromosome has a different combination of features selected (1s) and not selected (0s).

Fitness Function: The fitness of each chromosome (feature subset) is determined by the performance of a predictive model trained using only the selected features. Common performance metrics include accuracy, area under the ROC curve, or F1-score, depending on the problem specifics. Optionally, the fitness function can also penalize the number of features to maintain model simplicity.

Genetic Algorithm Process for Feature Selection

Step 1: Initialization

- Generate an initial population of feature subsets encoded as binary strings.

Step 2: Evaluation

- For each chromosome, train a model using only the features selected by that chromosome. Evaluate the model's performance on a validation set.

Step 3: Selection

- Select chromosomes for reproduction. Techniques like tournament selection or roulette wheel selection can be used, where chromosomes with higher fitness have a higher probability of being selected.

Step 4: Crossover

- Perform crossover between pairs of selected chromosomes to create offspring. A common method is one-point or two-point crossover, where segments of parent chromosomes are swapped to produce new feature subsets.

Step 5: Mutation

- Apply mutation to the offspring with a small probability. This could involve flipping some bits from 0 to 1 or vice versa, thus adding or removing features from the subset.

Step 6: Replacement

- Form a new generation by replacing some of the less fit chromosomes in the population with the new offspring. This could be a generational replacement or a steady-state replacement (where only the worst are replaced).

Iteration

- Repeat the process for a number of generations or until a stopping criterion is met (such as no improvement in fitness for a certain number of generations).

Example Use Case: Feature Selection in Medical Diagnosis You have a dataset with 200 features from various medical tests. You apply a genetic algorithm with an initial population of 50 chromosomes, evolving over 100 generations. Each chromosome dictates which features are used to train a logistic regression model to predict disease occurrence.

The process might reveal that only 30 out of the 200 features significantly contribute to the prediction, eliminating redundant and irrelevant features and thus simplifying the model without sacrificing (or possibly even improving) its performance.

5.6.2 Problem Setup: Hyperparameter Optimization for a Neural Network

Suppose we are developing a neural network to classify images into categories (e.g., for a fashion item classification task). The performance of the neural network can depend heavily on the choice of various hyperparameters such as the number of layers, number of neurons in each layer, learning rate, dropout rate, and activation function.

Genetic Algorithm Setup for Hyperparameter Optimization

Chromosome Representation: Each chromosome represents a set of hyperparameters for the neural network. For instance:

- Number of layers (e.g., 2-5)
- Neurons in each layer (e.g., 64, 128, 256, 512)
- Learning rate (e.g., 0.001, 0.01, 0.1)
- Dropout rate (e.g., 0.1, 0.2, 0.3)
- Activation function (e.g., relu, sigmoid, tanh)

Initial Population: Generate an initial population of chromosomes, each encoding a different combination of these hyperparameters.

Fitness Function: The fitness of each chromosome is evaluated based on the validation accuracy of the neural network configured with the hyperparameters encoded by the chromosome. Optionally, the fitness function can also include terms to penalize overfitting or excessively complex models.

Genetic Algorithm Process for Hyperparameter Optimization

Step 1: Initialization

- Generate an initial population of random but valid hyperparameter sets.

Step 2: Evaluation

- For each chromosome, construct a neural network with the specified hyperparameters, train it on the training data, and then evaluate it on a validation set. The validation set accuracy serves as the fitness score.

Step 3: Selection

- Select chromosomes for reproduction based on their fitness. High-fitness chromosomes have a higher probability of being selected. Techniques like tournament selection or rank-based selection are commonly used.

Step 4: Crossover

- Perform crossover operations between selected pairs of chromosomes to create offspring. Crossover can be one-point, two-point, or uniform (where each gene has an independent probability of coming from either parent).

Step 5: Mutation

- Apply mutation to the offspring chromosomes at a low probability. Mutation might involve changing one of the hyperparameters to another value within its range (e.g., changing the learning rate from 0.01 to 0.001).

Step 6: Replacement

- Replace the least fit chromosomes in the population with the new offspring, or use other replacement strategies like elitism where some of the best individuals from the old population are carried over to the new population.

Iteration

- Repeat the evaluation, selection, crossover, and mutation steps for several generations until the performance converges or a maximum number of generations is reached.

Example Use Case: Image Classification

You are using a dataset of fashion items where the task is to classify images into categories like shirts, shoes, pants, etc. You apply a genetic algorithm to optimize the hyperparameters of a convolutional neural network (CNN). After several generations, the GA might converge to an optimal set of hyperparameters that gives the highest accuracy on the validation dataset.

For instance, the best solution found by the GA could be:

- Number of layers: 3
- Neurons per layer: [512, 256, 128]
- Learning rate: 0.01
- Dropout rate: 0.2
- Activation function: relu

5.6.3 Genetic Algorithm in AI Games:

Example Use Case: Developing a Chess AI

Imagine you're developing an AI for a chess game. You start with 50 different strategies encoded as chromosomes. Each strategy is evaluated based on its performance in 100 games against diverse opponents. The strategies are then evolved over 100 generations, with each generation involving selection, crossover, and mutation to develop more refined and successful game strategies.

By the end of these iterations, the genetic algorithm might produce a strategy that effectively balances aggressive and defensive play, adapts to different opponent moves, and optimizes piece positioning throughout the game.

5.6.4 Genetic algorithms in Finance

Example Use Case: Diversified Investment Portfolio

Assume you manage an investment fund that considers a diverse set of assets, including stocks from various sectors, government and corporate bonds, and commodities like gold and oil. The task is to determine how much to invest in each asset class:

Assets:

Stocks (technology, healthcare, finance), government bonds, corporate bonds, gold, oil. Objective: Maximize the Sharpe ratio, considering historical returns and volatility data for each asset class.

Problem Setup:

Portfolio Optimization for Investment Management Suppose you are an investment manager looking to create a diversified investment portfolio. You want to determine the optimal allocation of funds across a set of available assets (e.g., stocks, bonds, commodities) to maximize returns while controlling risk, subject to various constraints like budget limits or maximum exposure to certain asset types.

Genetic Algorithm Setup for Portfolio Optimization

Chromosome Representation: Each chromosome in the GA represents a potential portfolio, where each gene corresponds to the proportion of the total investment allocated to a specific asset.

Initial Population: Generate an initial population of chromosomes, each encoding a different allocation strategy, ensuring that each portfolio adheres to the budget constraint (i.e., the total allocation sums to 100

Fitness Function: The fitness of each chromosome (portfolio) is typically evaluated based on its expected return and risk (often quantified as variance or standard deviation). A common approach to measure fitness is to use the Sharpe ratio, which is the ratio of the excess expected return of the portfolio over the risk-free rate, divided by the standard deviation of the portfolio returns.

After running the genetic algorithm for several generations, the GA might find an optimal portfolio that, for example, allocates 20% to technology stocks, 15% to healthcare stocks, 10% to finance stocks, 20% to government bonds, 15% to corporate bonds, 10% to gold, and 10% to oil. This portfolio would have the highest Sharpe ratio found within the constraints set by the algorithm.