4 Lecture 4:

4.1 Local Search

So far, we have utilized search strategies like A* search and the Greedy Best Search algorithm to navigate vast solution spaces and find sequences of actions that lead to optimal solutions. These strategies use heuristics to estimate costs from any node to the goal, improving efficiency by focusing on promising areas of the state space. However, the effectiveness of these methods depends heavily on the quality of the heuristic used. Informed search strategies are particularly useful for finding paths to the goal state. For example, solving the 8-puzzle requires a series of consecutive actions (such as moving the empty space up, down, left, or right) to reach the solution; similarly, traveling from Arad to Bucharest involves a sequence of actions moving from one connected city to another.

On the other hand, some problems focus solely on generating a goal state or a good state, regardless of the specific actions taken. For instance, in a simplified knapsack problem, the solution involves selecting a set of items that maximizes reward without exceeding the weight limit. The process does not concern itself with the order in which items are checked or selected to achieve the maximum reward. Local search strategies can be used to solve such problems. It is especially useful in problems with very large search spaces.

Local search algorithms offer a practical solution for tackling large and complex problems. Unlike global search methods that attempt to cover the entire state space, local search begins with an initial setup and gradually refines it. It makes incremental adjustments to the solution, exploring nearby possibilities through methods like Hill Climbing, Simulated Annealing, and Genetic Algorithms. These techniques adjust parts of the current solution to systematically explore the immediate area, known as the "neighborhood," for better solutions. This approach is particularly effective in environments with numerous potential solutions (local optima), helping to find an optimal solution more efficiently.

4.1.1 The Need for Local Search

Local search is particularly useful in:

Large or poorly understood search spaces, where exhaustive global search is impractical. For example, in large scheduling problems where the number of potential schedules increases exponentially with the number of tasks and resources.

Optimization tasks, focusing on improving a measure rather than finding a path. This is seen in machine learning hyperparameter tuning, where algorithms like Gradient Descent and its variants iteratively adjust parameters to minimize a loss function.

Dynamic problems, where solutions must adapt to changes without restarting the search. A common example is in real-time strategy games, where AI opponents must continuously adjust their strategies based on the player's actions.

Local search also complements hybrid algorithms, combining its strengths with other strategies for enhanced performance across various problems. For instance, Genetic Algorithms use local search within their crossover and mutation phases to fine-tune solutions.

In summary, local search provides a flexible and efficient approach for refining solutions incrementally, making it a critical tool in modern computational problem-solving. These strategies bridge the gap between the theoretical optimality of informed search and the practical needs of real-world applications.

4.1.2 Examples of problems that can be solved by local search:

Local search algorithms are versatile tools for tackling various optimization problems across many fields. Here are some example problems where local search algorithms are particularly effective:

1. **Traveling Salesman Problem (TSP)**

   In the TSP, the goal is to find the shortest possible route that visits a list of cities and returns to the origin city. A local search algorithm like Simulated Annealing or 2-opt (a simple local search used in TSP) can iteratively improve a given route by swapping the order of visits to reduce the overall travel distance.

2. **Knapsack Problem**

   As previously mentioned, the goal here is to maximize the value of items placed in a knapsack without exceeding its weight capacity. Local search techniques can be used to iteratively add or remove items from the knapsack to find a combination that offers the highest value without breaching the weight limit.

3. **Max Cut Problem**

   This is a problem in which the vertices of a graph need to be divided into two disjoint subsets to maximize the number of edges between the subsets. Local search can adjust the placement of vertices in subsets to try and maximize the number of edges that cross between them.

4. **Protein Folding**

   In computational biology, protein folding simulations involve finding low-energy configurations of a chain of amino acids. Local search can be used to tweak the configuration of the protein to find the structure with the lowest possible energy state.

5. **Layout Design**

   In manufacturing and architectural design, layout problems involve the placement of equipment or rooms to minimize the cost of moving materials or to maximize accessibility. Local search can rearrange the placements iteratively to improve the overall layout efficiency.

6. **Parameter Tuning in Machine Learning**

   Tuning hyperparameters of a machine learning model to minimize prediction error or maximize model performance can also be approached as an optimization problem. Techniques like Grid Search, Random Search, or more sophisticated local search methods can be applied to find optimal parameter settings.

4.1.3 Some Constraint Satisfaction Problems can also be solved using local search strategies:

1. **Graph Coloring**

   This problem requires assigning colors to the vertices of a graph so that no two adjacent vertices share the same color, using the minimum number of colors. Local search can explore solutions by changing the colors of certain vertices to reduce conflicts or the number of colors used.

2. **Job Scheduling Problems**

   In job scheduling, the task is to assign jobs to resources (like machines or workstations) in a way that minimizes the total time to complete all jobs or maximizes throughput.

   Job scheduling can be viewed as a CSP when the task is to assign start times to various jobs subject to constraints such as job dependencies (certain jobs must be completed before others can start), resource availability (jobs requiring the same resource cannot overlap), and deadlines.

   Local search can be used to iteratively shift jobs between resources or reorder jobs to find a more efficient schedule.

3. **Vehicle Routing Problem**

   Similar to TSP but more complex, this problem involves multiple vehicles and aims to optimize the delivery routes from a central depot to various customers.

   This problem can also be modeled as a CSP, where constraints might include vehicle capacity limits, delivery time windows, and the requirement that each route must start and end at a depot.

   Local search can adjust routes by reassigning customers to different vehicles or changing the order of stops to minimize total distance or cost.

Local search algorithms are ideal for these and many other problems because they can provide high-quality solutions efficiently, even when the search space is extremely large and complex. They are particularly valuable when exact methods are computationally infeasible, and an approximate solution is acceptable. 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. That means they are **not systematicthey might never explore a portion of the search space where a solution actually resides.** However, they have **two key advantages:**

1. **they use very little memory; and**
2. **they can often find reasonable solutions in large or infinite state spaces for which systematic algorithms are unsuitable.**

4.2 Local Search Algorithms:

In this course we are going to learn about:

1. Hill-Climbing Algorithm / Gradient Descent
2. Problems of Hill Climbing Algorithms and Remedies
3. Simulated Annealing
4. Genetic Algorithm

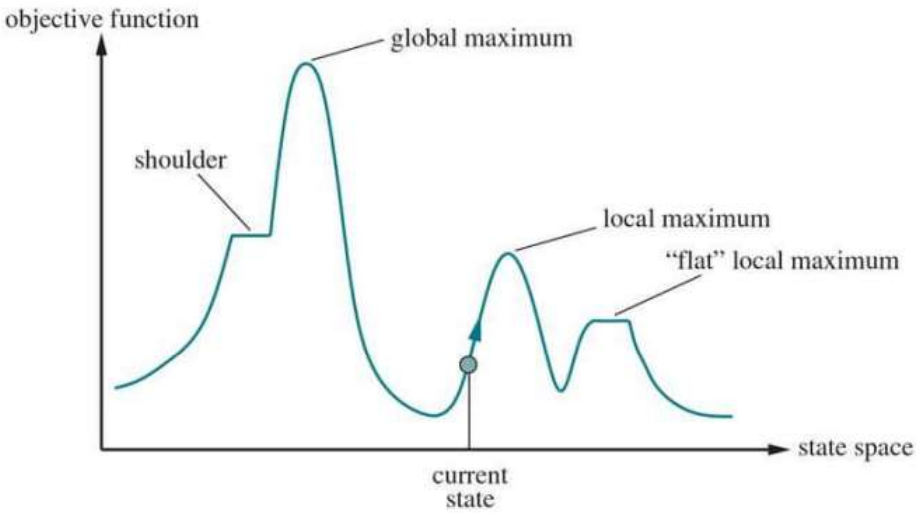
4.2.1 State-Space and Objective Function:

Local search algorithms are powerful tools for addressing optimization problems, where the objective is to find an optimal state that maximizes or minimizes a given objective function. These algorithms navigate through a metaphorical "state-space landscape," where each point or state in this landscape represents a possible solution with a specific "elevation" defined by the objective function. The concept can be more clearly understood through the following points, illustrated with examples:

1. **Understanding the State-Space Landscape**

   Concept: Imagine the state-space of a problem as a geographical landscape where every point represents a possible solution. The elevation at each point is determined by the value of the objective function at that state.

   Elevation as Objective Function: In optimization terminology, elevation could represent a value or a cost associated with each state. High elevations indicate better values in maximization problems, and lower elevations indicate lower costs in minimization problems.



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

2. **Objective of Local Search**

   Maximization (Hill Climbing): Here, the goal is to find the highest point in the landscape, akin to climbing to the peak of a hill. The algorithm iteratively moves to higher elevations, seeking to locate the highest peak, which represents the optimal solution.

   Example: Maximizing sales in a retail chain by adjusting variables such as pricing, marketing spend, and store layout. Each adjustment is a "step" in the landscape, seeking higher profits (higher elevations).

   Minimization (Gradient Descent): In contrast, this approach aims to find the lowest point or valley in the landscape. This is suitable for cost reduction problems where each step attempts to move to lower elevations, minimizing the objective function.

   Example: Minimizing production costs in a manufacturing process by altering materials, labor, and energy usage. Each configuration change is a step toward lower costs.

3. **Examples of Local Search Algorithms in Action**

   i. **Traveling Salesman Problem (TSP):**

   Objective function: Minimize the total travel distance for a salesman needing to visit multiple cities and return to the starting point.

   Local Search Strategy: Start with a random route and iteratively improve it by swapping the order of cities if it results in a shorter route (seeking lower valleys in terms of distance).

   ii. **Knapsack Problem**

   Objective function: Maximize the value of items placed in a knapsack without exceeding its weight capacity.

Local search Strategy: Begin with a random combination of items. Iteratively add or remove items from the knapsack to find a combination that offers the highest value without breaching the weight limit.

iii. **8-queen problem**

Objective function: Minimize the number of pairs of queens that are attacking each other either horizontally, vertically, or diagonally. The ideal goal is to reduce this number to zero, indicating that no queens are threatening each other.

Local search strategy: Start with a random instance of the board and iteratively improving the placement of queens on the board.

4.3 Hill-Climbing Search:

The Hill Climbing search algorithm maintains one current state and iteratively moves to the neighboring state with the highest value, effectively following the steepest path upward. It stops when it reaches a "peak," meaning no adjacent state offers a better value. This algorithm only considers the immediate neighbors and does not plan beyond them.

Algorithm of Hill-Climbing Search:

Algorithm 1 Local Maximum Search

```
Input: initial_state, objective_function()
Output: state that is a local maximum
begin
  current_state ← initial_state
  loop do
    neighbor ← a state that is a neighbor of current_state
if objective_function(neighbor) ≤ objective_function(current_state) then
  return current_state
end if
    current_state ← neighbor
```

Gradient descent algorithm: When the goal of the hill-climbing algorithm is to find the state with the minimum value rather than maximizing the objective, it is known as gradient descent.

4.3.1 Examples:

Simplified Knapsack Problem Setup:

Items (value, weight):
Item 1: Value = $\$10$, Weight = 2 kg
Item 2: Value = $\$15$, Weight = 3 kg
Item 3: Value = $\$7$, Weight = 1 kg
Item 4: Value = $\$20$, Weight = 4 kg
Item 5: Value = $\$8$, Weight = 1 kg

Knapsack Capacity: 7 kg

Objective: Maximize the total value of the items in the knapsack such that the total weight does not exceed 7 kg.

Hill Climbing Algorithm:

Initial Solution Start with a randomly selected set of items that do not exceed the capacity. For this example, let's start with the 1st, 2nd and 3rd item in the knapsack.

The knapsack can be represented as a string of 1s and 0s, where a '1' at ith position indicates that the corresponding (ith) item has been included, and a '0' means the item has not been taken.

Initial Solution: 11100, weight: 6kg, value: 32

Neighbor Generation:

Generate neighboring solutions by toggling the inclusion of each item. For instance, if an item is not in the knapsack, consider adding it; if it is in the knapsack, consider removing it or replace an item with another item not in the solution.

Evaluate and Select:

Calculate the total value and weight for each neighbor. If a neighbor exceeds the knapsack's capacity, discard it.

Choose the neighbor with the highest value that does not exceed the weight capacity. Iteration:

Repeat the process of generating and evaluating neighbors from the current solution. If no neighbors have a higher value than the current solution, terminate the algorithm.

Termination:

The algorithm stops when it finds a solution where no neighboring configurations offer an improvement.

Demonstration:

Step 1: Initial Solution
Knapsack: 11100
Total Value: $\$32$
Total Weight: 6 kg

Step 2: Generate Neighbors (first iteration)
Add Item 4: 11110, Total Value $= \$52$, Total Weight $= 10$ kg
Add Item 5: 11101, Total Value $= \$40$, Total Weight $= 7$ kg
Replace Item 2 with Item 4: 10110, Total Value $= \$37$, Total weight: 7kg/
.. There can be other possible neighbors.

Step 3: Evaluate and Select
Discard Neighbor: 11110 as it exceeds knapsack weight limit.
Best neighbor: 11101 Total Value $= 40$, Total Weight $= 7$ kg

Step 4: Iteration (next steps)
Current configuration: 11101
Add Item 4: 11111, Total Value $= \$60$, Total Weight $= 11$ kg Replace items with Item 4:
All will exceed the weight

Evaluation:

All neighbors are discarded. No better valued neighbor.
Terminates as no better valued neighbor is found.

4.3.2 Key Characteristic:

Hill-Climbing Search is essentially a greedy approach, meaning it always looks for the best immediate improvement at each step, without considering long-term consequences. This can sometimes lead to suboptimal solutions.

4.3.3 Drawbacks:

Hill climbing algorithms may encounter several challenges that can cause them to get stuck before reaching the global maximum. These challenges include:

1. **Local Maxima:** A local maximum is a peak that is higher than each of its neighboring states but is lower than the global maximum. Hill climbing algorithms that reach a local maximum are drawn upward toward the peak but then have nowhere else to go because all nearby moves lead to lower values.

   Example:

   8-queen problem:

   Background: In a 8-queens problem, our goal is to maximize the number of non-attacking pairs of queens. The maximum number of pairs possible from 8 queens is 8*7/2 =28. So, we want all 28 pairs to be in non-attacking positions.

   Let us represent the instances as an array of 8 numbers, ranging from 1 to 8 denoting the columns and the index, ranging from 1 to 8 (sorry, programmers!) denoting the rows.

   Scenario: For an instance of the 8-puzzle, 13528647, there are 5 attacking pairs or in other words, 23 non-attacking pairs. This configuration is a local maximum because it's better than all immediate neighboring configurations, but it is not the global solution since it's not conflict-free.

   Problem: The Hill Climbing algorithm would stop here because all single-move alternatives lead to worse configurations, increasing the number of threats. Despite the presence of better configurations (global maxima with zero threats), the algorithm gets stuck.

2. **Plateaus:** A plateau is an area of the state-space landscape where the elevation (or the value of the objective function) remains constant. Hill climbing can become stuck on a plateau because there is no upward direction to follow. Plateaus can be particularly challenging when they are flat local maxima, with no higher neighboring states to escape to, or when they are shoulders, which are flat but eventually lead to higher areas. On a plateau, the algorithm may wander aimlessly without making any progress.

   Scenario: Imagine a large part of the chessboard setup where several queens are placed in such a manner that moving any one of them doesn't change the number of conflictsit remains constant. This flat area in the search landscape is a plateau.

For Example, the instance, 13572864 has 3 attacking pairs. Swapping the last two queens might not immediately lead to an increase in non-attacking pairs, resulting in a plateau where many configurations have an equal number of non-attacking pairs.

Problem: The Hill Climbing algorithm would find it difficult to detect any better move since all look equally non-promising.

On a plateau, every move neither improves nor worsens the situation, causing Hill Climbing to wander aimlessly without clear direction toward improvement. This lack of gradient (change in the number of conflicts) can trap the algorithm in non-productive cycles, preventing it from reaching configurations that might lead to the global maximum.

3. **Ridges:** Ridges are sequences of local maxima that make it very difficult for greedy algorithms like hill climbing to navigate effectively. Because the algorithm typically makes decisions based on immediate local gains, it struggles to cross over ridges that require a temporary decrease in value to ultimately reach a higher peak.

   These challenges highlight the limitations of hill climbing algorithms in exploring complex landscapes with multiple peaks and flat areas, making them susceptible to getting stuck without reaching the best possible solution.

   Scenario: Consider a situation where moving from one configuration to another better configuration involves moving through a worse one. For example, the instance 13131313 creates a ridge because small moves from this configuration typically result in fewer non-attacking pairs, requiring several coordinated moves to escape this pattern, which hill climbing does not facilitate well.

   Problem: Hill Climbing may fail to navigate such transitions because it does not allow for a temporary increase in the objective function (number of conflicts in this case). Ridges in the landscape can make it very challenging for the algorithm to find a path to the optimal solution since each step must immediately provide an improvement.

4.3.4 Remedies to problems of Hill-Climbing Search Algorithm:

Hill climbing has several variations that address its basic version's limitations, such as getting stuck at local maxima, navigating ridges ineffectively, or wandering on plateaus.

4.3.5 Variants of Hill Climbing:

- **Stochastic Hill Climbing:**

  How it works: This method randomly chooses among uphill moves rather than always selecting the steepest ascent. The probability of choosing a move can depend on the steepness of the ascent.

  Performance: It typically converges more slowly than the standard hill climbing because it might not take the steepest path. However, it can sometimes find better solutions in complex landscapes where the steepest ascent might lead straight to a local maximum.

  Example in 8-Queens: In a landscape where queens are close to forming a solution but are stuck due to subtle needed adjustments, stochastic hill climbing can randomly explore different moves that gradually lead out of a local maximum.

- **First-Choice Hill Climbing:**

  How it works: A variant of stochastic hill climbing, this strategy generates successors randomly and moves to the first one that is better than the current state.

  Advantages: This is particularly effective when a state has a vast number of successors, as it reduces the computational overhead of generating and evaluating all possible moves.

  Example in Knapsack Problem: With thousands of possible item combinations, generating all potential successors to evaluate the best move is impractical. First-choice hill climbing can quickly find a better solution without exhaustive comparisons.

- **Random-Restart Hill Climbing:**

  How it works: This approach involves performing multiple hill climbing searches from different randomly generated initial states. This process repeats until a satisfactory solution is found.

  Completeness: It is "complete with probability 1" because it is guaranteed to eventually find a goal state, assuming there is a non-zero probability of any single run succeeding. If each hill-climbing search has a probability of success, p then the expected number of restarts required is $\frac{1}{p}$.

  Example in Knapsack Problem: If the algorithm gets stuck in a sub-optimal configuration due to local maxima, restarting with a new random set of items can lead to discovering better solutions.

  Example in 8-Queens: Due to the complex landscape of the 8-Queens problem, random restarts can help escape from sub-optimal arrangements by exploring new configurations that might be closer to the global maximum (i.e., a solution with zero attacking pairs).

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

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.

5.3.5 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 Diversity in Genetic Algorithm

It is often the case that the population is diverse early on in the process, so crossover frequently takes large steps in the state space early in the search process (as in simulated annealing). After many generations of selection towards higher fitness, the population becomes less diverse, and smaller steps are typical.

It is important for the initial population to be diversed. Otherwise, similar type of chromosomes will crossover to and produce offsprings with little change in their fitness. This will lead to quick convergence of the algorithm and the chances of finding a solution with maximum fitness will be very low.

5.4.6 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 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 represented by the index of the string or array.

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).

Calculating the number of attacking pairs: The 8-queen problem has the following conditions.

- No pairs share the same column.
- No pairs share the same row.

- No pairs share the same diagonal.

Column-wise conflict Now, due to the representation of the configuration where we have ensured that no pairs can share the same column as the indices of the array are unique.

Row-wise conflict Now, the values in each index represent the row where the queen is placed. Now, if we find the same value in multiple indices that means there are queens sharing that row.

Take for example, the configuration: [8, 7, 7, 3, 7, 1, 4, 4]. Here, the value 7 is repeated thrice and the value 4 is repeated twice. This means, there are 3 queens on the 7th row and 2 queens in the 4th row.

Counting the conflicts in 7th row: 3 queens will form ${}^3C_2 = 3(3-1)/2 = 3$ pairs.

Counting the conflicts in 4th row: 2 queens will form ${}^2C_2 = 2(2-1)/2 = 1$ pairs.

Total 4 row-wise conflicting pairs.

Diagonal Conflicts Two types of diagonals are formed in a square board we call them Major ("/" shaped) diagonal, and Minor ("\" shaped) diagonal. **Major diagonal conflict** If two queens, $Q1$ and $Q2$ share the same major diagonal conflict abs($Q1$[column] $-$ $Q1$[row]) = abs($Q2$[column] $-$ $Q2$[row]).

Going back to the configuration: [8, 7, 7, 3, 7, 1, 4, 4], the queen in 7th row, 2nd column is in the same diagonal as the queen in 1st row, 6th column and the queen in 7th row, 3rd column is in conflict with the queen in 4th row, 8th column. Or in other words 2 queens share the 5th (|7-2| = |1-6|) Major diagonal making (${}^2C_2 = 2(2-1)/2 =)1$ conflicting pair and 2 queens share the 4th (|7-3| = |4-8|) Major diagonal making another (1) conflicting pair.

So, there are two attacking pairs along the major diagonal.

Minor diagonal conflict If two queens, $Q1$ and $Q2$ share the same major diagonal conflict $Q1$[column]+ $Q1$[row] = $Q2$[column] + $Q2$[row].

Going back to the configuration: [8, 7, 7, 3, 7, 1, 4, 4], the queen in 8th row, 1st column is in the same diagonal as the queen in 7th row, 2nd column, the queen in 3rd row, 4th column is in conflict with the queen in 1st row, 6th column and the queen in 7th row, 5th column is in conflict with the queen in 4th row, 8th column. Or in other words 2 queens share the 9th (|8+1|=|7+2|) minor diagonal making 1 (${}^2C_2 = 2(2-1)/2$) conflicting pair, 2 queens share the 7th (|3+4| = |1+6|) minor diagonal making another (${}^2C_2 = 2(2-1)/2$) conflicting pair and 2 queens share the 12th (|7+5| = |4+8|) minor diagonal making another (${}^2C_2 = 2(2-1)/2$) conflicting pair.

So, there are three attacking pairs along the minor diagonal.

Summing up the number of attacking pairs:

- 4 row-wise attacking pairs.
- 2 attacking pairs on the major diagonal.
- 3 attacking pairs on the minor diagonal.

- In total $(4 + 2 + 3 = 9)$ attacking pairs.

Calculating the number of non-attacking pairs: The maximum number of pairs formed from n number of queens is ${}^nC_2 = \frac{n(n-1)}{2}$. The maximum number of pairs that can be formed by 8 queens is $\frac{8(8-1)}{2} = 28$. In the ideal configuration with zero attacking pairs, we can have all 28 pairs in non-attacking mode. Thus, in any configuration,
No. of non-attacking pairs = No. of max possible non-attacking pairs - No. of attacking pairs.

For the example of [8, 7, 7, 3, 7, 1, 4, 4] configuration of the 8-queen problem, the max possible non-attacking pairs is 28 and the total no. of attack we calculated is 9. So the number of non-attacking pairs in this configuration is $28 - 9 = 19$.

So, the Fitness value of this configuration is 19.

A better way to calculate fitness is to converting the value within a range of $(0 to 1)$ or in a $x\%$.

$$\text{fitness} = \frac{\text{no. of non-attacking pairs}}{28}$$

However, for simplicity, we are going to take the no. of non-attacking pairs as our fitness value.

Iteration 1

Initialization of Population:

We randomly generate a small population of 4 individuals for simplicity. To keep track of the individual with the best fitness so far we initially set best-so-far = null and best-fitness-so-far = 0 and update when we find better individuals.

```
best-so-far = [];
best-fitness-so-far = 0;
max-fitness = 28;
```

Step 1: Population

Initial Population

Chromosome 1:	4 2 7 3 6 8 5 1
Chromosome 2:	2 7 4 1 8 5 3 6
Chromosome 3:	4 2 7 3 6 8 5 1
Chromosome 4:	7 1 4 2 8 5 3 6

Step 2: Calculate Fitness

	Chromosomes	Fitness
Chromosome 1:	4 2 7 3 6 8 5 1	26
Chromosome 2:	2 7 4 1 8 5 3 6	24
Chromosome 3:	4 2 7 3 6 8 5 1	24
Chromosome 4:	7 1 4 2 8 5 3 6	23

As we have found a chromosome with better fitness value we have saved before, we update,

```
best-so-far = [4 2 7 3 6 8 5 1];
best-fitness-so-far = 26;
```

However, the fitness is not the highest possible value of 28, so we continue to the next step.

Step 3: Selection

Select the top 3/4th of chromosomes based on their fitness. This results in selecting Chromosome 1, 2 and 3.

	Chromosomes	Fitness	Selection
Chromosome 1:	4 2 7 3 6 8 5 1	26	selected
Chromosome 2:	2 7 4 1 8 5 3 6	24	selected
Chromosome 3:	4 2 7 3 6 8 5 1	24	selected
Chromosome 4:	7 1 4 2 8 5 3 6	23	not selected

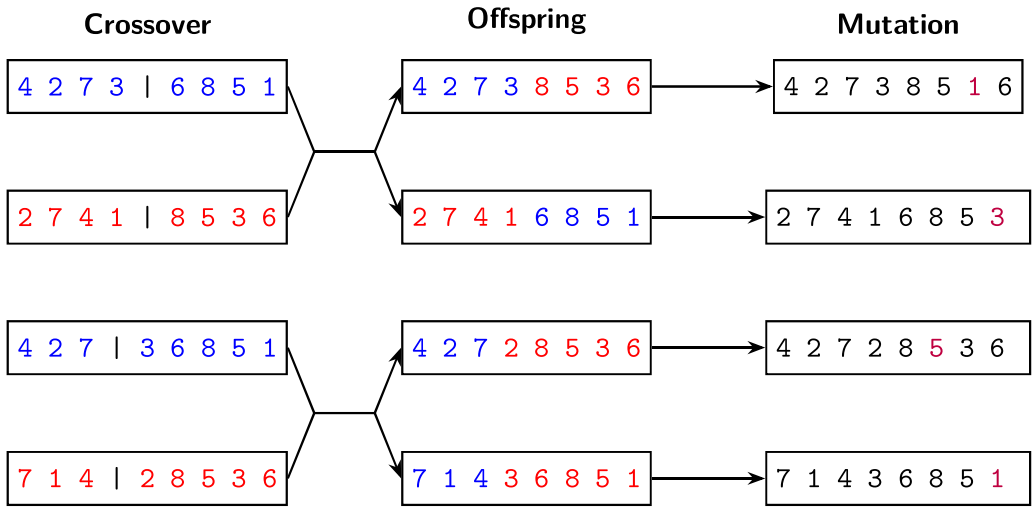
	Selected Chromosomes	Fitness
Chromosome 1:	4 2 7 3 6 8 5 1	26
Chromosome 2:	2 7 4 1 8 5 3 6	24
Chromosome 3:	4 2 7 3 6 8 5 1	24

Step 4: Crossover (Single Point)

We pair the best chromosome [4, 2, 7, 3, 8, 5, 1, 6] with the other two selected chromosomes to create new offspring.

Step 5: Mutation

For each offspring, We randomly choose an index and change the value of the row to a number chosen randomly between 1 to 8. As the number is chosen randomly it may happen that the value remains same as we see for offspring 3 and 4.



Step 6: Create new Population replacing the old population

After the mutation is complete we replace the previous population with the set of new chromosomes and repeat from step 2 with the new population until the configuration with highest fitness (non-attacking pair) is found.

	New Population
Chromosome 1:	4 2 7 3 8 5 1 6
Chromosome 2:	2 7 4 1 6 8 5 3
Chromosome 3:	4 2 7 1 8 5 3 6
Chromosome 4:	2 7 4 3 6 8 5 1

5.5.2 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 the cities $[A, B, C, D, E]$.

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: $2 + 6 + 12 + 15 + 7 = 42$

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

Iteration 1

Initialization of Population:

We start with a population of four randomly generated routes as combination of the cities. We keep track of the best found route and its fitness.

```
best-so-far = [];
best-fitness-so-far = 0;
```

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

Step 2: Fitness Calculation

Lower distance means higher fitness in this problem. The fitness for each chromosome in the population is calculated.

	Chromosomes	Fitness	Selection
Chromosome 1:	A B C D E	$\frac{1}{42}$	selected
Chromosome 2:	B E D C A	$\frac{1}{45}$	selected
Chromosome 3:	E D B A C	$\frac{1}{49}$	not selected
Chromosome 4:	B D C E A	$\frac{1}{39}$	selected

Step 3: Selection

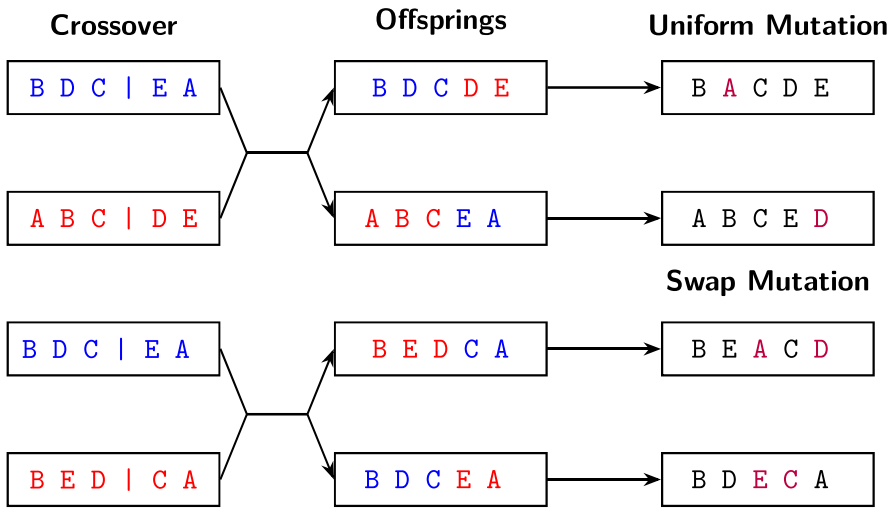
Select top 3/4th based on fitness: Chromosome 4, Chromosome 1 and Chromosome 2. Chromosome 4 has a higher fitness value than the `best-fitness-so-far` we have. So, we make an update.

```
best-so-far = [B D C E A];
best-fitness-so-far = 1/39;
```

	Selected Chromosomes	Fitness
Chromosome 1:	A B C D E	$\frac{1}{42}$
Chromosome 2:	B E D C A	$\frac{1}{45}$
Chromosome 3:	E D B A C	$\frac{1}{49}$

Step 4: Crossover (One-Point)

We make pairs between chromosome with the highest fitness, [B D C E A] with the other two chromosomes that are selected. The crossover point for the pairs are randomly generated. The crossover point in this case for both the pairs were randomly chosen to be 3.



Step 5: Mutation

Offspring 1 is mutated by randomly changing the second city from D to A. Offspring 2 is mutated by changing the fifth city from A to D.
On the other hand, Offspring 3 and Offspring 4 are mutated by swapping cities. In Offspring 3, the third and fifth cities are swapped. In Offspring 4, the third and fourth cities are swapped.
Note: It is better to use one specific type of mutation in your simulation. Always mention what type of mutation you are using.

Step 6: New Population

Replace previous population with the newly generated chromosomes.

New Population

Chromosome 1: B A C D E

Chromosome 2: A B C E D

Chromosome 3: B E A C D

Chromosome 4: B D E C A

Step 7: Repeat

This process is repeated from step 2 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.

5.5.3 0/1 Knapsack Problem

Problem Setup

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	Weight (Kg)	Value ($)
1	6	30
2	3	14
3	4	16
4	2	9

Genetic Algorithm Setup

Chromosome Representation:

Each chromosome is a string of four 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 four random chromosomes (combination of the items) making the initial population. We keep track of the best found route and its fitness.

```
best-so-far = [];
best-fitness-so-far = 0;
```

Initial Population

Chromosome 1: 1 1 0 1

Chromosome 2: 1 0 1 0

Chromosome 3: 0 1 1 0

Chromosome 4: 1 0 0 1

Step 2: Fitness Evaluation

Calculate the total value of each chromosome, ensuring the weight does not exceed the capacity.

	Chromosomes	Fitness
Chromosome 1:	1 1 0 1	53
Chromosome 2:	1 0 1 0	46
Chromosome 3:	0 1 1 0	30
Chromosome 4:	1 0 0 1	39

Step 3: Selection

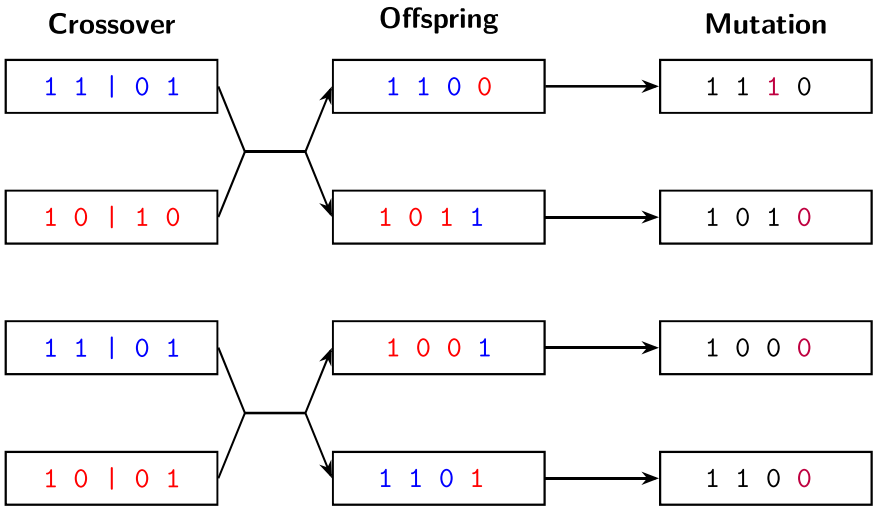
Select the top 3/4th of chromosomes based on their value.

```
best-so-far = [1101];
best-fitness-so-far = 53;
```

	Chromosomes	Fitness
Chromosome 1:	1 1 0 1	53
Chromosome 2:	1 0 1 0	46
Chromosome 4:	0 1 1 0	39

Step 4: Crossover

Perform crossover between the chromosome with the highest fitness value with the two other chromosomes from the selected chromosomes at the third bit.



Step 5: Mutation

Apply a mutation by flipping a random bit in each offspring.

Step 6: New Population

Replace the previous population with the set of new chromosomes.

New Population

Chromosome 1: 1 1 1 0

Chromosome 2: 1 0 1 0

Chromosome 3: 1 0 0 0

Chromosome 4: 1 1 0 0

Step 7: Repeat

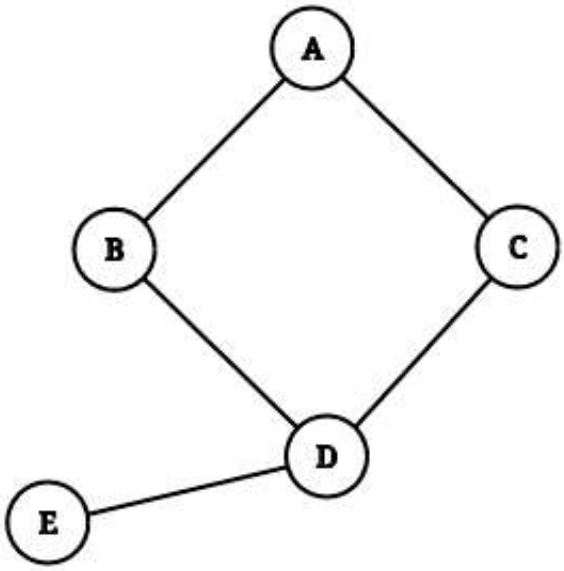
This process is repeated over several generations to find the chromosome with the highest fitness, representing the maximum value found.

5.5.4 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. We keep track of the best found route and its fitness.

```
best-so-far = [];
best-fitness-so-far = 0;
```

Initial Population

Chromosome 1: | 1 2 3 1 2 |

Chromosome 2: | 2 3 1 2 3 |

Chromosome 3: | 1 2 1 3 2 |

Chromosome 4: | 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).

Calculate the fitness based on the coloring rules.
Chromosome 1: Fitness = 5 (all edges correctly colored).
Chromosome 2: Fitness = 5.
Chromosome 3: Fitness = 3 (CD edge is incorrectly colored).
Chromosome 4: Fitness = 4 (DE edge is incorrectly colored).

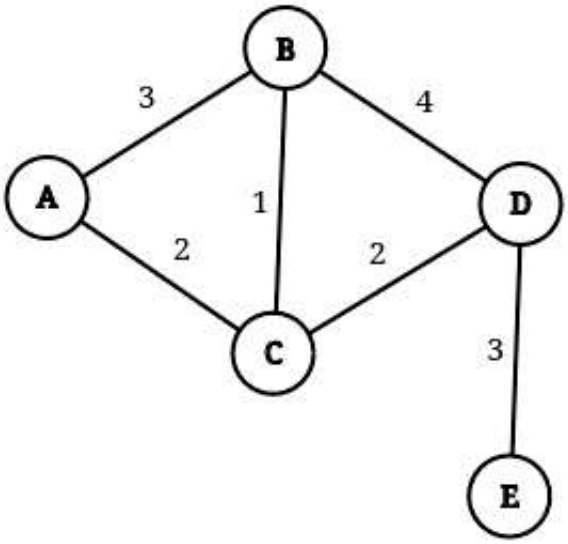
Carry on selection, crossover, mutation and population replacement as before.

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 will 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.

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'.

For example, consider a configuration 10101 where the Vertices A, C, E in set Y; B, D in set X.

Fitness = Sum of Edges (AB, BC, CD, DE) − Sum of Edges(AC) $= 3 + 1 + 4 - 2 = 9$.

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.

5.6.2 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.

5.6.3 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

5.6.4 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.

5.6.5 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.