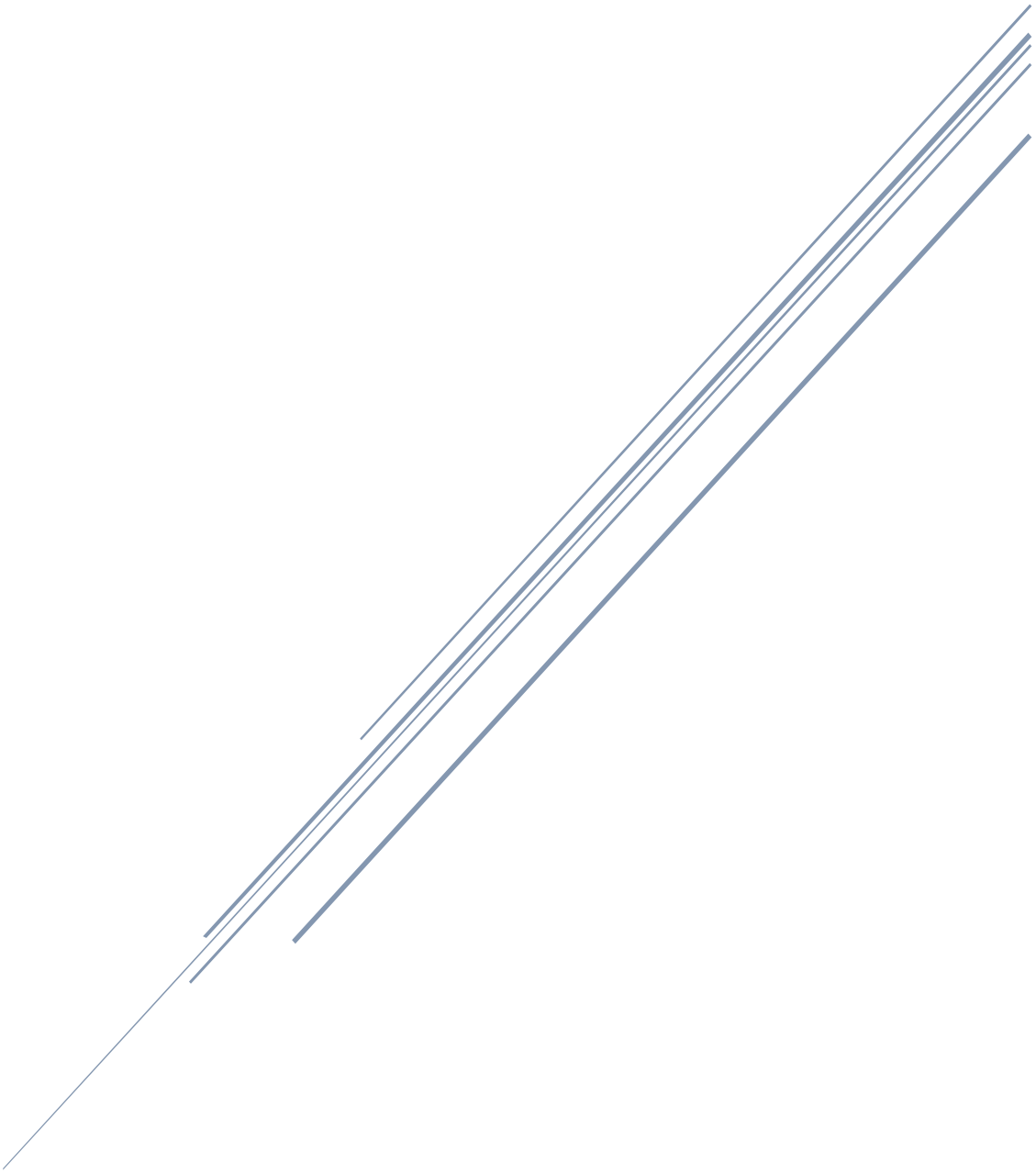
CHAPTER 2

Metaheuristics



2.1 Introduction

The search algorithms discussed in Chapter 1 are designed to explore search spaces by keeping one or more paths in memory and recording the explored alternatives at each juncture along the path. When a goal is found, the path leading to that goal serves as a solution to the problem. However, in many problems, the path to the goal is irrelevant, and the primary focus lies in identifying the unknown goal state. This property holds for many important applications such as integrated-circuit (IC) design, factory-floor layout, job-shop scheduling, ...

Example: 8-queens

The goal of the 8-queens problem is to place eight queens on a chessboard in such a way that no queen attacks any other. A queen can attack any piece in the same row, column, or diagonal. Figure 2.1 shows a possible solution to this problem. what matters is the final configuration of queens, not the order in which they are added.

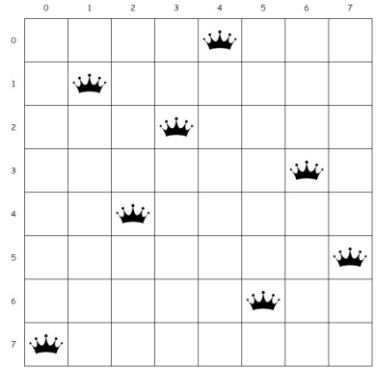


Figure 2.1 Possible optimal solution for the 8-queens problem

If the path to the goal doesn't matter, we might consider a different class of algorithms—those that don't worry about paths at all. These algorithms are referred to as metaheuristics.

In addition to finding goals, metaheuristics are useful for solving pure optimization problems. These problems involve identifying the optimal solution based on an objective function.

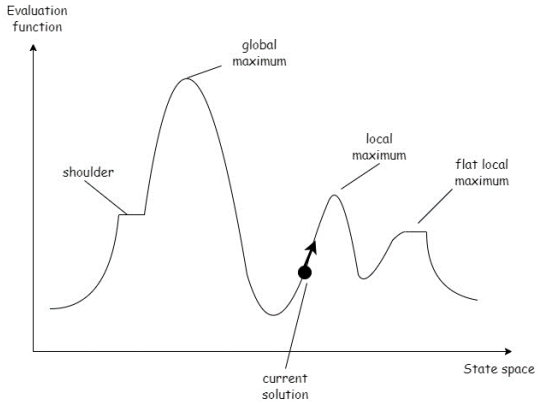


Figure 2.2 A one-dimensional state-space landscape for an optimization problem

To understand an optimization problem, we consider the state-space landscape (depicted in Figure 2.2). This landscape has both "location" (determined by the state) and "elevation" (determined by the value of the objective function). Depending on the nature of the problem, in the case of a maximization problem, the objective is to find the highest valley—a global maximum. Conversely, for a minimization problem, the goal is to identify the lowest peak—a global minimum. If the objective function

corresponds to cost, the objective is to locate a global minimum. It's important to note that any maximization problem can be transformed into a minimization problem by negating the objective function. A metaheuristic explores this landscape with the aim of finding a global minimum/maximum.

2.2 Metaheuristics

2.2.1 Definition

In AI, metaheuristics are algorithms that are designed to find approximate solutions to optimization and search problems. They are often used when the search space is too large or complex to be exhaustively searched, and the specific path to the solution is not as important as finding a reasonably good solution efficiently.

2.2.2 Metaheuristics and combinatorial problems

A combinatorial problem involves counting and arranging discrete elements, often with the goal of finding the number of possible configurations, combinations, or permutations.

Many combinatorial problems are classified as NP-hard. They are often characterized by having a large solution space, and the number of possible solutions grows exponentially with the size of the input. Problems in this category include:

- ✧ **Graph coloring:** Assigning colors to the vertices of a graph such that no two adjacent vertices share the same color.
- ✧ **Knapsack problem:** Selecting a subset of items with given weights and values to maximize the total value without exceeding a given weight limit.
- ✧ **Hamiltonian cycle problem:** Determining whether there exists a cycle that visits each vertex exactly once in a graph.
- ✧ **Traveling salesman problem (TSP):** finding the shortest possible path that visits a set of cities only once and returns to the original city.

Metaheuristics play crucial role in solving combinatorial problems, providing flexible and powerful approaches to find optimal/near-optimal solutions within a reasonable timeframe.

2.2.3 Search Space in metaheuristics

The search space in metaheuristics is the set or domain within which an algorithm searches, representing the range of feasible solutions among which an optimal solution resides. Each point in the search space corresponds to a possible solution, and each of these solutions can be characterized by its value of objective function denoted as f for the given problem. The objective function measures how good the solution is based on the problem constraints and requirements.

The search process can be complicated, making it challenging to determine where to look for an optimal solution or where to initiate the search.

Example: 8-queens problem

Solution state representation: In this problem, the state represents the positions of the 8-queens on the 8×8 chessboard. Each state can be represented as an 8-element array, with each element indicating the column number in which the queen is placed in the corresponding row. For instance, the configuration in Figure 2.3(a) can be represented by the array illustrated in Figure 2.3(b).

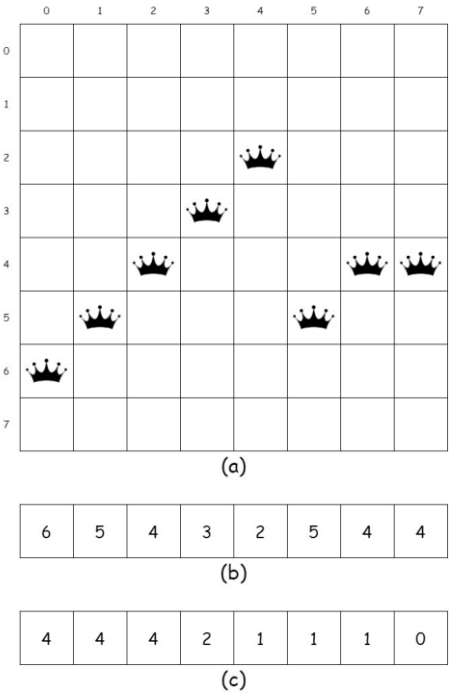


Figure 2.3 Solution representation for the 8-queens problem

Objective function: In the 8-queens problem, the objective is to maximize the number of queens that are not attacking each other. We can define a fitness function that calculates the number of non-attacking pairs of queens. Therefore, the fitness function is inversely proportional to the number of attacks between the queens. Consequently, a solution with a high fitness value will have a minimal number of attacks, ideally zero. The fitness function is as follows (Eq 2.1).

$$fitness(s) = -number\_attacks(s) \tag{2.1}$$

For instance, in the configuration shown in Figure 2.3(a), the number of attacks of each queen is illustrated in Figure 2.3(c), and the fitness value of this solution is -17.

2.2.4 Functioning of metaheuristics

The functioning of metaheuristics involves iteratively improving a solution of a problem. It starts with an initial solution and utilizes a set of rules to modify this solution, aiming to discover a better solution than the current one. This iterative process continues until a satisfactory solution is achieved.

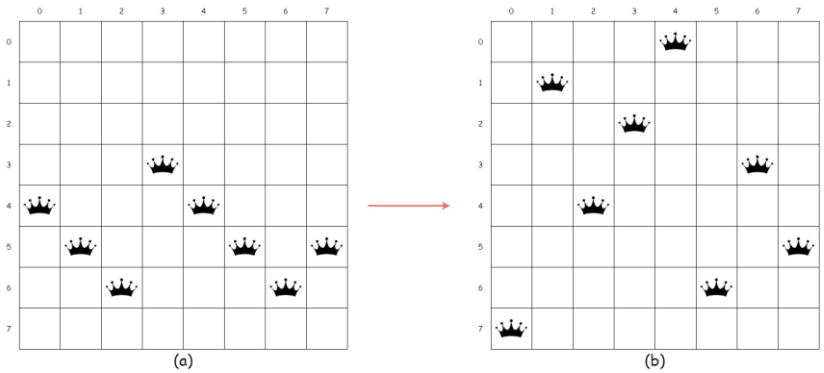


Figure 2.4 8-queens game. (a) An initial solution. (b) A possible optimal solution

Example: 8-queens

A possible initial solution is shown in Figure 2.4(a). This solution is improved with a metaheuristic until we reach the final solution illustrated in Figure 2.4(b).

2.2.5 Classification of metaheuristics

Various types of metaheuristics exist, each with its own strengths and weaknesses. Metaheuristic algorithms are classified based on how they operate over the search space such as nature-inspired vs. non-nature inspired, population-based vs. single point search, memory vs. memory-less usage, deterministic vs. stochastic, iterative vs. greedy …

1) Nature-inspired vs. non-nature inspired

Several metaheuristics are inspired by natural processes:

- ✧ Evolutionary algorithms inspired by biology. For example, genetic algorithm.
- ✧ Particle swarm optimization (simulating the movement of a group of birds), ant colony optimization (modeling the foraging behavior of ants), and bee colony optimization.
- ✧ Simulated annealing inspired by physics.

2) Population-based vs. single-solution-based

Another classification dimension is single solution vs. population-based searches. Single solution approaches focus on modifying and improving a single candidate solution. Metaheuristics falling under this category include local search, hill-climbing, simulated annealing.

On the other hand, population-based approaches focus on maintaining and improving multiple candidate solutions, often utilizing population characteristics to guide the search. Metaheuristics belonging to this category include evolutionary computation like genetic algorithm and particle swarm optimization.

Note: These two families exhibit complementary characteristics:

- ✧ Single-solution-based metaheuristics are exploitation-oriented, possessing the ability to intensify the search in local regions.
- ✧ Population-based metaheuristics are exploration-oriented, enabling better diversification across the entire search space.

3) Memory vs. memory-less usage

Some methods, like tabu search, use information extracted during the search for both short-term and long-term purposes. For others, no information is extracted and used dynamically during the search, such as hill-climbing and simulated annealing.

4) Deterministic vs. stochastic

A deterministic metaheuristic solves the optimization problem by making deterministic decisions, such as hill-climbing and tabu search. Stochastic metaheuristics apply random rules during the search, exemplified by methods like simulated annealing and genetic algorithms.

Note: In deterministic algorithms, using the same initial solution will result in the same final solution, whereas in stochastic algorithms, different final solutions may be obtained from the same initial state.

5) Iterative vs. greedy

Iterative algorithms begin with a complete solution (or a population of solutions) and transform it at each iteration. In contrast, greedy algorithms start with an empty solution, and at each step, a decision variable of the problem is assigned until a complete solution is obtained.

Note: Most metaheuristics are iterative algorithms.

2.2.6 Design of Metaheuristics

In designing a metaheuristic, two contradictory criteria must be taken into account: the exploration of the search space (diversification) and the exploitation of the best solutions found (intensification).

As shown in Figure 2.5, in intensification, promising areas are explored more thoroughly in the hope of refining and improving the solution found. In diversification, unexplored regions must be visited to ensure that all areas of the search space are evenly explored, thus discovering different and potentially better solutions. This helps prevent getting stuck in local optima.

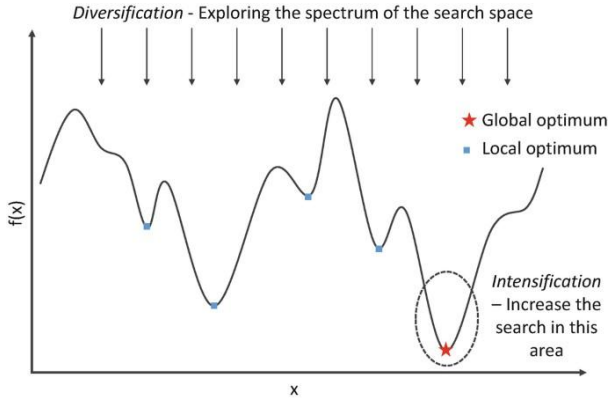


Figure 2.5 Diversification vs. Intensification

Population-based algorithms inherently have a diversification component because they maintain multiple solutions concurrently. These solutions explore various regions of the solution space simultaneously, contributing to diversification. On the other hand, single-solution-based approaches, by focusing on refining the current solution and exploring its neighborhood, are more aligned with intensification (see Figure 2.6).

However, it's essential to note that these concepts are not mutually exclusive. Population-based algorithms can incorporate mechanisms for intensification by selecting and propagating the best solutions. Single-solution-based algorithms may include strategies for diversification, such as introducing randomness or perturbations to explore different regions.

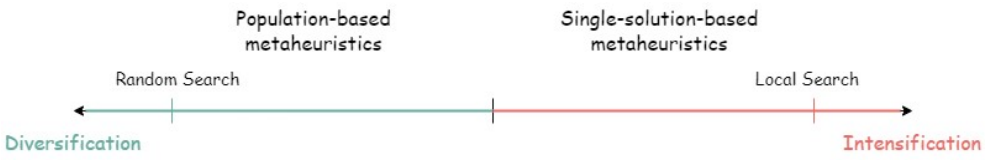


Figure 2.6 Design space of a metaheuristic

2.3 Metaheuristic algorithms

In metaheuristics, the algorithm does not maintain a search tree. Therefore, the data structure for the

current solution only needs to store the *state* (which represents the configuration of the solution) and the value of the objective function, denoted as *f* (see Figure 2.7).

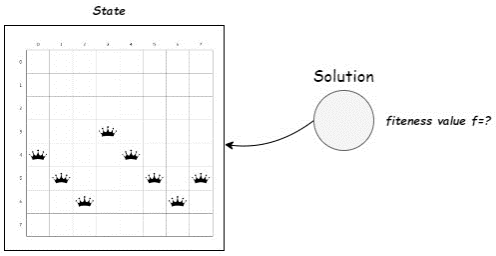


Figure 2.7 Structure of a solution in metaheuristics

2.3.1 Random Search

Random Search is the basic form of a metaheuristic. It explores the search space by randomly generating candidate solutions. It does not rely on any specific information about the structure of the search space, and it is often used as a baseline or a comparison point for more sophisticated algorithms.

Here's a basic explanation of the Random Search algorithm:

- **Initialization:** The algorithm starts by choosing a random solution as the initial solution and evaluates its value of objective function *f*.
- **Search process:** The algorithm generates a new random candidate solution and evaluates its value of objective function *f*. If the *f* value of the newly generated solution is better than the current best solution, the algorithm updates the best solution.
- **Termination:** The algorithm terminates when a stopping criterion is met. This criterion can be a maximum number of iterations, finding a satisfactory solution, a threshold value for the objective function, or a time limit.
- **Solution:** The final solution is the best solution found during the search process.

Random Search algorithm is presented in Figure 2.8.

```
Output: The best solution
Function RandomSearch()
Begin
   bestSolution <- generateRandomSolution()
   bestSolution.f <- evaluateSolution(bestSolution)

   while (stopping condition is not satisfied) do
      candidateSolution <- generateRandomSolution()
      candidateSolution.f <- evaluateSolution(candidateSolution)

      if (candidateSolution.f > bestSolution.f) then
         bestSolution <- candidateSolution
      end if
   end while

   return bestSolution
End
```

Figure 2.8 Random Search algorithm

Random search is a purely exploratory method, and its effectiveness depends on the nature of the optimization problem. While random search may struggle with complex search spaces, it can sometimes surprisingly be effective in high-dimensional spaces.

2.3.2 Local Search

Local search is part of the family of metaheuristics based on a single solution, with a focus on exploitation. The local search algorithm aims to find the best solution to a problem through iterative adjustments to an initial solution. At each iteration, the neighborhood of the current solution is explored, and a new solution is selected from this neighborhood. The current solution is then compared to the new one using the objective function, and if the new one is better, it replaces the old one. This process continues until a satisfactory solution is discovered or a predetermined stopping criterion is satisfied.

The neighborhood of a given solution is the set of feasible solutions that are somehow similar to the given solution, i.e., with some common elements and objective function values not very different (see Figure 2.9).

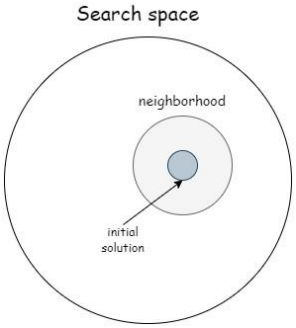


Figure 2.9 Neighborhood in the state space of local search

Example: 4-queens problem

If we consider the example of the 4-queens problem, the neighborhood of the solution shown in Figure 2.10(a) is the set of the 12 solutions shown in Figure 2.10(b). This neighborhood is created by moving each queen at a time, and each queen can move with 1, 2 or 3 steps in the 4×4 chessboard.

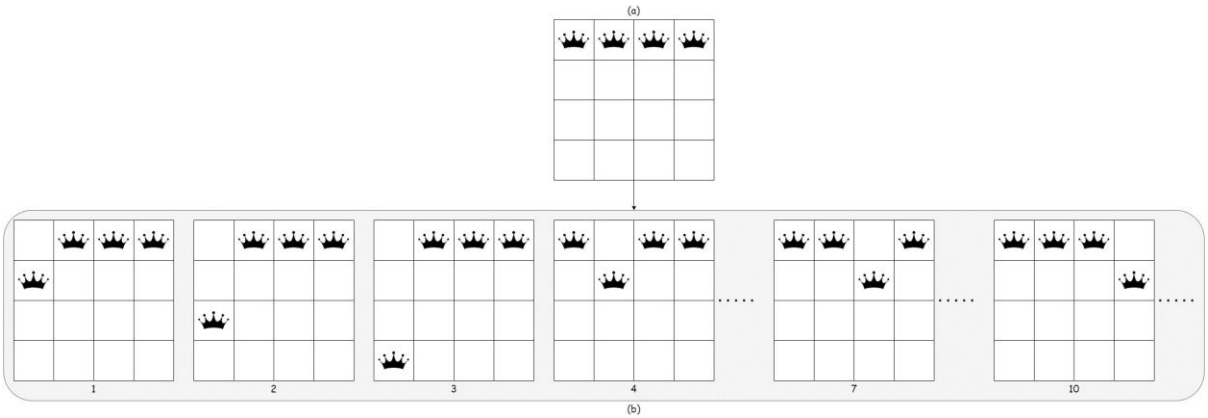


Figure 2.10 Example of a solution neighborhood in 4-queens problem

Here are the general steps of a local search algorithm:

- **Initialization:** The algorithm starts with an initial solution to the problem. This solution can be generated randomly or using a heuristic. The quality of the initial solution is evaluated using the objective function *f*.
- **Neighborhood search:** The algorithm generates neighboring solutions by making small perturbations to the current solution.
- **Selection:** The neighboring solutions are evaluated using the objective function *f*, the best neighboring solution is selected as the new current solution.
- **Termination:** The algorithm terminates when a stopping criterion is met. This criterion can be a maximum number of iterations, a threshold value for the objective function, or a time limit.
- **Solution:** The final solution is the best solution found during the search process.

Several local search algorithms are commonly used in AI and optimization problems. Let's delve into some of the commonly used local search algorithms:

2.3.3 Hill Climbing Search

Hill Climbing is a direct local search algorithm that starts with an initial solution and iteratively moves to the best neighboring solution, aiming to improve the objective function. In other words, the Hill Climbing algorithm keeps moving uphill, striving for higher values until it reaches a peak where no neighbor has a higher value.

Here's how it operates:

- **Initialization:** Start with an initial solution S_0, typically generated randomly or through a heuristic method. Then, assess the quality of the initial solution using an objective function.
- **Neighbor Generation:** Create neighboring solutions by implementing minor changes (moves) to the current solution.
- **Selection:** Choose the neighboring solution that provides the most significant improvement in the objective function. Replace the current solution with the best neighboring solution if it has a better value for the objective function.
- **Termination:** Continue this process until a termination condition is met (e.g., reaching a maximum number of iterations or finding a satisfactory solution).
- **Solution:** The final solution is the best solution found during the search process.

Hill Climbing search algorithm (steepest-ascent version) is presented in Figure 2.11.

```
Input:
- So: Initial solution generated randomly or through a heuristic
Output: The best solution
Function Hill_Climbing (So)
Begin
  bestSolution <- So
  bestSolution.f <- evaluateSolution(bestSolution)

  while (stopping condition is not satisfied) do
    // Generate the best solution's neighbors and compute their f values
    neighbors <- generateNeighbors(bestSolution)
```

```
    for neighbor in neighbors do
      neighbor.f <- evaluateSolution(neighbor)
    end for

    // Select the neighbor with the best f value
    bestNeighbor <- selectBestNeighbor(neighbors)
    if (bestNeighbor.f ⩾ bestSolution.f) then
      bestSolution <- bestNeighbor
    else
      return bestSolution
    end if
  end while

  return bestSolution
End
```

Figure 2.11 Hill Climbing Algorithm

Example: Figure 2.12 illustrates the application of the Hill Climbing algorithm to solve the 4-queens problem.

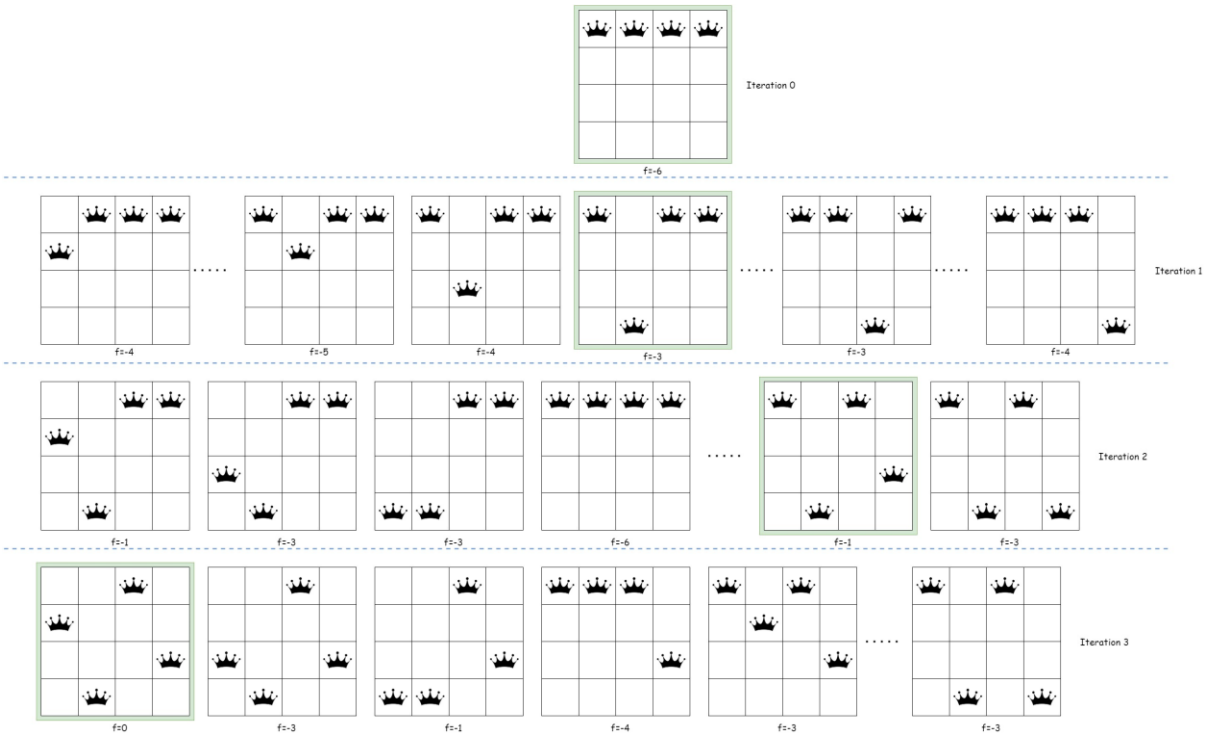


Figure 2.12 Application of Hill Climbing algorithm on the 4-queens problem

Hill Climbing often makes rapid progress toward a solution because it is usually quite easy to improve a bad solution. However, it often gets stuck for the following reasons:

- **Local maxima:** A local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum. When the Hill Climbing algorithm gets close to a local maximum, it goes up but can get stuck because there's no higher ground nearby, as shown in Figure 2.13(a).

✧ **Plateaux:** Plateaux are flat areas in the state-space landscape. They can be flat local maxima with no upward path or shoulders where progress is possible (refer to Figure 2.13(b)). In Hill Climbing searches, getting lost on a plateau is a possibility.

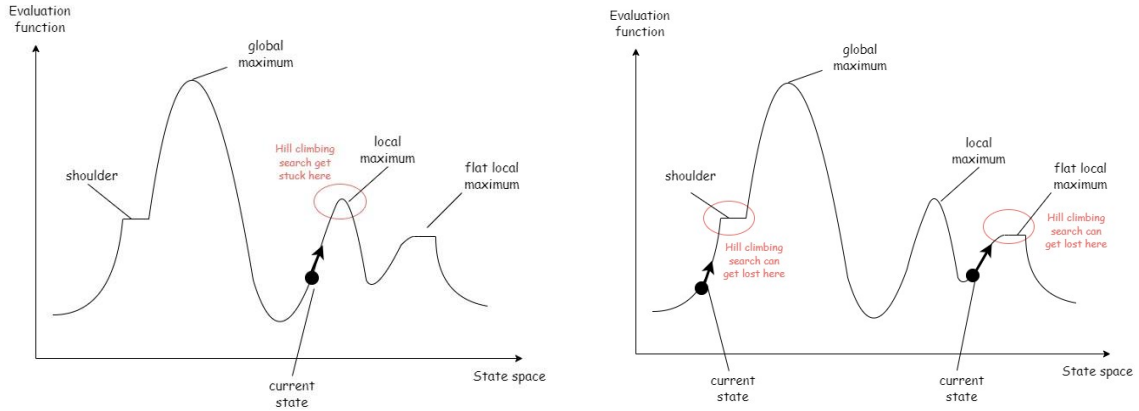
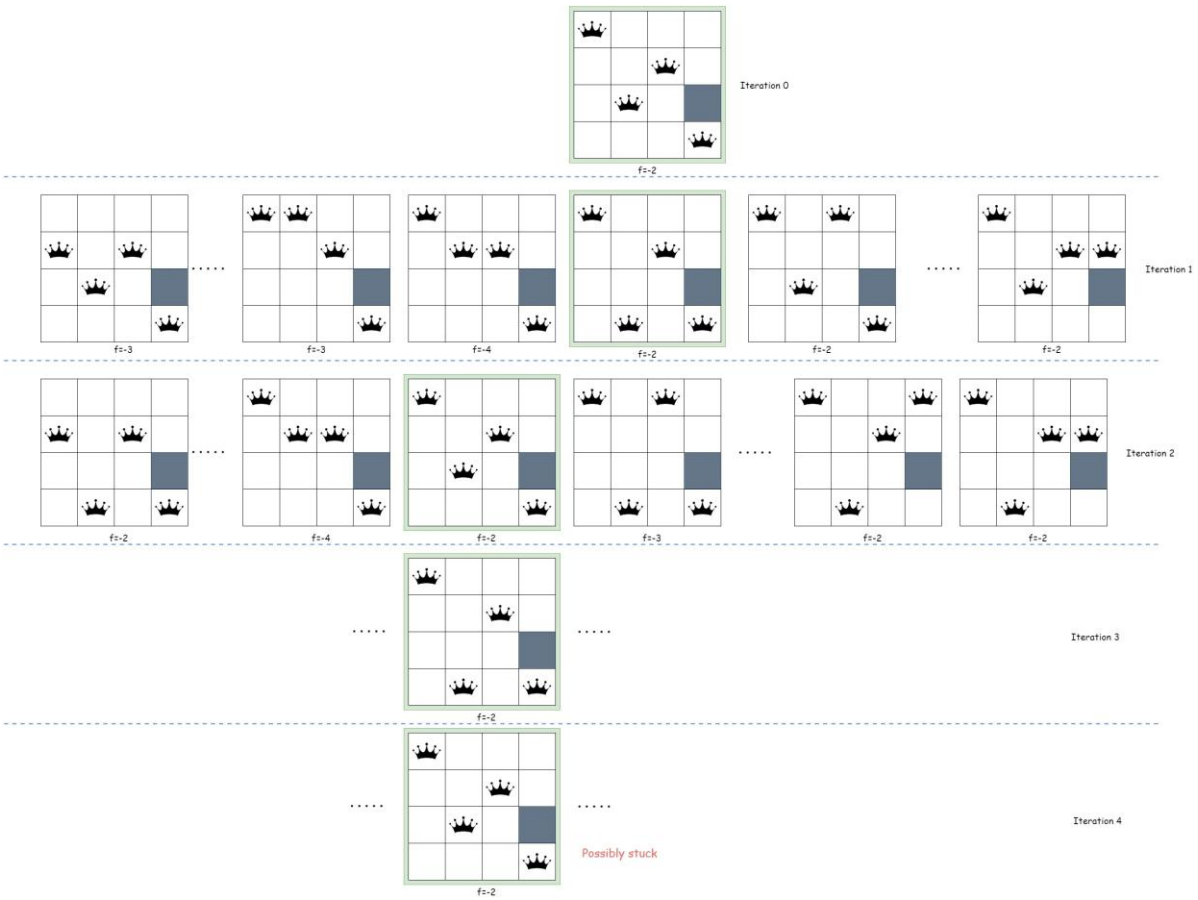


Figure 2.13 Hill Climbing limitations

Example: Figure 2.14 illustrates how the Hill Climbing algorithm can possibly get stuck while solving the 4-queens problem.



To overcome the limitations of standard Hill Climbing, several variations of the algorithm have been developed. These include **Stochastic Hill Climbing**, **First-Choice Hill Climbing** and **Random-Restart Hill Climbing**.

2.3.3.1 Stochastic Hill Climbing

In Stochastic Hill Climbing, rather than strictly choosing the greedy uphill move, we introduce randomness by considering a randomized uphill move. This approach allows exploration of different areas within the search space. While this method typically converges more slowly than steepest ascent version, it can be advantageous in certain state landscapes, potentially leading to the discovery of better solutions. Stochastic Hill Climbing search algorithm is illustrated in Figure 2.14.

```
Input:
- So: Initial solution generated randomly or through a heuristic
Output: The best solution
Function Stochastic_Hill_Climbing(So)
Begin
  bestSolution <- So
  bestSolution.f <- evaluateSolution(bestSolution)

  while (stopping condition is not satisfied) do
    // Generate the best solution's neighbors and compute their f values
    neighbors <- generateNeighbors(bestSolution)
    for neighbor in neighbors do
      neighbor.f <- evaluateSolution(neighbor)
    end for

    // Select all neighbors with f greater than bestSolution.f
    bestNeighbors <- selectAllBestNeighbors(neighbors, bestSolution)
    if (bestNeighbors is empty) than
      return bestSolution
    end if

    // Randomly select a neighbor from the list of best neighbors
    bestNeighbor <- randomlySelectNeighbor(bestNeighbors)
    bestSolution <- bestNeighbor
  end while

  return bestSolution
End
```

Figure 2.14 Stochastic Hill Climbing Algorithm

2.3.3.2 First-Choice Hill Climbing

The First-Choice Hill Climbing algorithm involves the random generation of neighboring solutions until a better one than the current solution is found. This is especially useful in situations where the number of neighboring solutions is too large to compare directly with the current solution. Randomly generating and transitioning to a better solution may lead to a more efficient and effective outcome. First-Choice Hill Climbing search algorithm is illustrated in Figure 2.15.

```
Input:
- So: Initial solution generated randomly or through a heuristic
```

```
Output: The best solution
Function First_Choice_Hill_Climbing(So)
Begin
  bestSolution <-So
  bestSolution.f <- evaluateSolution(bestSolution)

  while (stopping condition is not satisfied) do
    // Generate a random neighbor and compute its f value
    neighbor <- generateRandomNeighbor(bestSolution)
    neighbor.f <- evaluateSolution(neighbor)

    if (neighbor.f > bestSolution.f) then
      bestSolution <- neighbor
    end if
  end while

  return bestSolution
End
```

Figure 2.15 First Choice Hill Climbing Algorithm

2.3.3.3 Random-Restart Hill Climbing

The Random-Restart Hill Climbing algorithm performs a sequence of Hill Climbing searches, initiating each search from randomly generated initial solutions. This approach continues until a goal is reached. Whenever the algorithm becomes stuck at a local maximum or plateau, it initiates a random restart, allowing it to make additional attempts to find an improved solution. Random-Restart Hill Climbing search algorithm is illustrated in Figure 2.16.

```
Input:
- So: Initial solution generated randomly or through a heuristic
Output: The best solution
Function Random_Restart_Hill_Climbing(So)
Begin
  S <- So
  bestSolution <- S

  while stopping condition is not satisfied do
    currentSolution <- Hill_Climbing(S)
    if (isOptimalSolution(currentSolution)) then
      return currentSolution
    end if
    if (currentSolution.f > bestSolution.f) then
      bestSolution <- currentSolution
    end if
    S <- generateRandomSolution()
  end while
```

```
    return bestSolution
End
```

Figure 2.16 Random Restart Hill Climbing Algorithm

The success of Hill Climbing depends on the shape of the state-space landscape. If the landscape has few obstacles, such as local maxima and plateaux, Random-Restart Hill Climbing can quickly find a good solution. However, in many real-world problems, the landscape is often complex. NP-hard problems, in particular, exhibit numerous local maxima, making it challenging to find the global optimum. Despite these challenges, the strategy of multiple restarts often leads to finding a reasonably good local maximum.

2.3.4 Simulated Annealing (SA)

Local search including Hill Climbing often converges to a local optimum, requiring strategies to escape these suboptimal solutions and explore the solution space more comprehensively and effectively.
To enhance exploration, any iterative search or exploration process should consider accepting non-improving moves. This flexibility allows the algorithm to navigate away from local optima.

1) Definition

In metallurgy, annealing is the process used to temper or soften metals and glass by heating them to a high temperature and then gradually cooling them. This gradual cooling allows the material to attain a low-energy crystalline state, affecting its properties such as hardness and ductility.

When the temperature is high, the material exists in a liquid state (see Figure 2.17(a)). During a hardening process, the material transitions to a solid state with non-minimal energy, often referred to as a metastable state (see Figure 2.17(b)). In this state, the atomic structure lacks symmetry. Through a slow annealing process, the material transitions to a solid state where atoms are organized with symmetry, forming a crystalline structure (see Figure 2.17(c)).

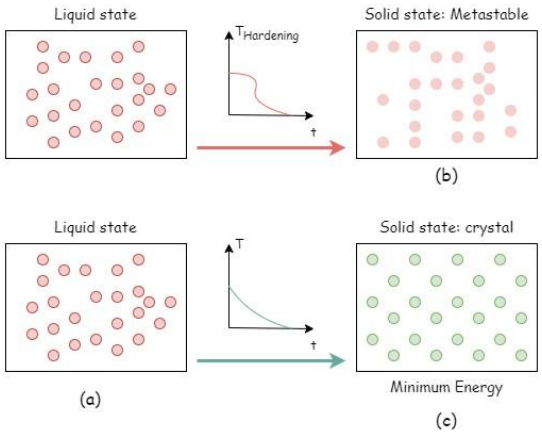


Figure 2.17 hardening vs. annealing process

This process involves bringing a solid to a low-energy state after raising its temperature. It can be summarized by the following two steps:

✧ Elevate the solid to a very high temperature until the structure approaches a melting state.

- ✧ Gradually cool the solid following a specific temperature-decreasing scheme to achieve a solid state with minimized energy.

2) Formulation

In simulated annealing algorithm, an analogy is made between a multi-particle system and the optimization problem by using the following equivalences:

- ✧ The search space states (solutions) represent the possible states of the solid
- ✧ A neighboring solution is generated by modifying the position of one particle
- ✧ The function to be minimized (the objective function) represents the energy of the solid

3) Mechanism

Starting from an initial state i of the metal with energy E_i, a new state j with energy E_j is generated by modifying the position of one particle. If the energy difference $E_j - E_i$ is negative (indicating that the new state features lower energy), then the state j becomes the new current state. However, if the energy difference is greater than or equal to zero, then the probability that the state j becomes the current state is given by the following formula (Eq 2.2):

$$acceptance\_probability(j) = \begin{cases} 1 & if\ E_j < E_i \\ e^{(-(E_j - E_i)/T)} & otherwise \end{cases} \quad (2.2)$$

The probability of accepting a solution depends on a control parameter T, which acts as a temperature.

4) Cooling schedule

The annealing schedule determines the process by which T should be decreased, which influences the performance of the overall algorithm to a great extent.

It is recommended that T is decreased by a factor that is less than one. When T is decreasing exponentially, the rate of decrease becomes slower as the search process nears completion. This gradual reduction allows the system to explore more thoroughly, potentially finding the global minimum.

One common approach is to use the formula $T_i = \alpha\, T_{i-1}$, where $\alpha < 1$. Ideally, α is set between 0.5 and 0.99.

- ✧ **High Temperature (Early Stages):** At the beginning of the process, when the temperature T is high, the acceptance probability for transitions with higher objective degradation is also high. This facilitates exploration of the search space, even accepting solutions that worsen the objective function. This behavior is analogous to the annealing process in metallurgy, where high temperatures allow for greater exploration.
- ✧ **Temperature Decrease:** As the temperature, denoted as T, decreases over time, the acceptance probability decreases as well. This means that transitions with higher objective degradation are less likely to be accepted. The algorithm gradually shifts from exploration to exploitation, focusing on refining the solution.
- ✧ **Low Temperature (Later Stages):** When the temperature becomes very low, only transitions that improve the objective function or have a very low objective deterioration are accepted. The algorithm behaves more like a local search, converging towards the optimal solution.

5) Simulated Annealing algorithm

Simulated Annealing algorithm is illustrated in Figure 2.18. The goal of this algorithm is to find the optimal solution by minimizing the objective function (energy function).

```
Input:
- So: Initial solution generated randomly or through a heuristic
- To: Initial temperature
- NumIterPerT: Number of iterations at the same temperature, usually equal to 1
Output: The best solution
Function Simulated_Annealing(So, To, NumIterPerT)
Begin
  currentSolution <- So
  currentSolution.f <- computeEnergy(currentSolution) // Here, the objective function is the energy
  bestSolution <- currentSolution // Initially the best solution is the current solution

  T <- To // To is the highest temperature

  while (stopping condition is not satisfied) do
    for i<-1 to NumIterPerT do
      // Generate a random neighbor by making perturbation on the current solution
      neighbor <- generateRandomNeighbor(currentSolution)
      // compute the energy of the neighboring solution
      neighbor.f <- computeEnergy(neighbor)

      // Compute the difference of energy between the neighboring solution and the current solution
      ΔE  <- neighbor.f - currentSolution.f
      if (ΔE < 0) then
        currentSolution <- neighbor
      else
        p <- uniform random value  ∈ [0,1)
        if (p ≤ e^(-ΔE/T)) then
          currentSolution <- neighbor
        end if
      end if
      if (currentSolution.f < bestSolution.f) then
        bestSolution <- currentSolution
      end if
    end for
    updateTemperature(T) // T is updated according to a cooling schedule
  end while

  return bestSolution
End
```

Figure 2.18 Simulated Annealing algorithm

Example: Figure 2.19 illustrates the application of the Simulated Annealing algorithm to solve the 4-queens problem. We consider the following input: The initial temperature T_0 is 2.0, the number of iterations per temperature is 1, and the cooling schedule is $T_i=T_{i-1} - 0.4$.

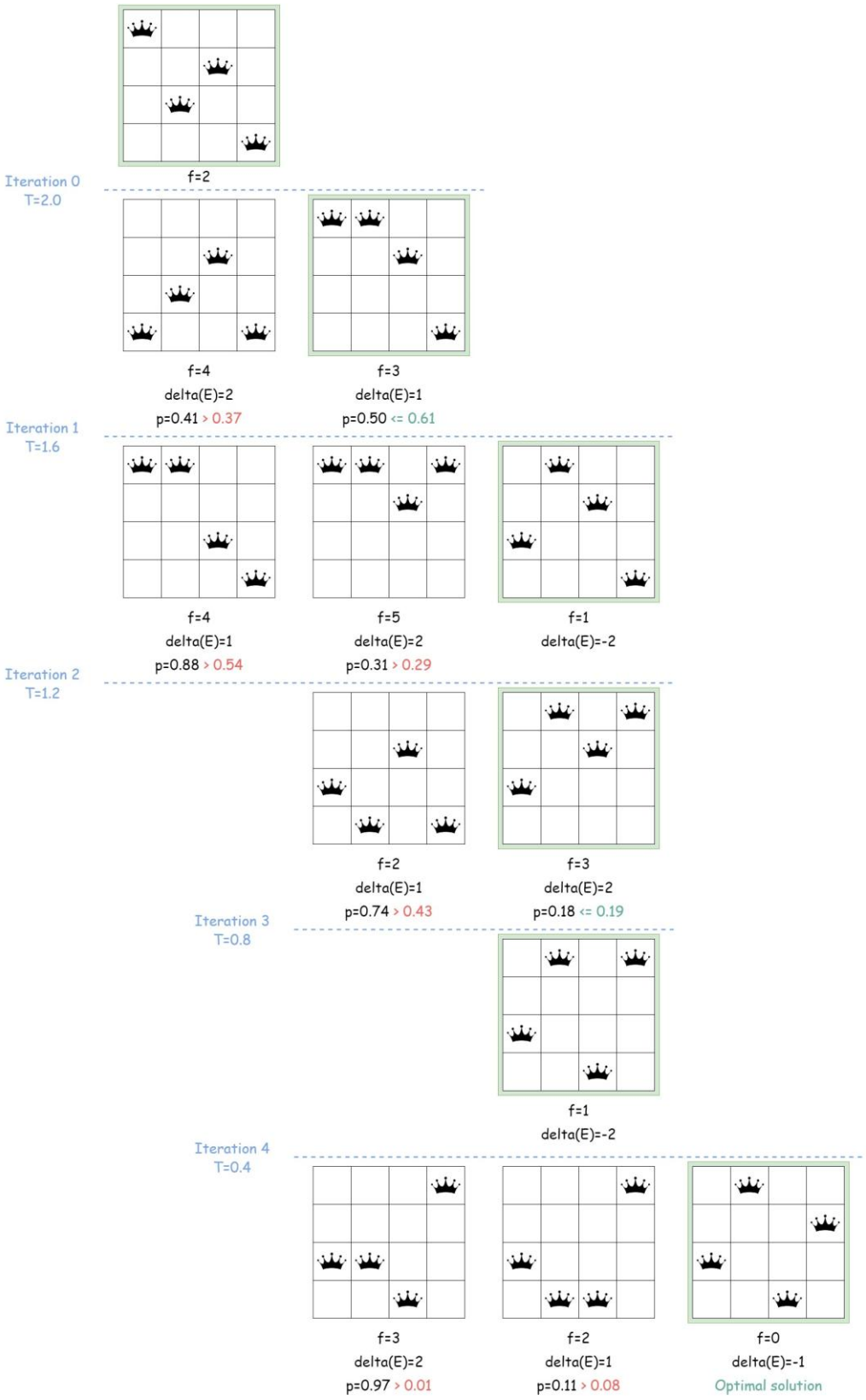


Figure 2.19 Application of Simulated Annealing algorithm on the 4-queens problem

6) Properties

The ability of Simulated Annealing (SA) to accept transitions that degrade the objective function, especially in the early stages of the process, is a key feature of the algorithm. This property allows SA to explore a broader solution space, including regions that might lead to better solutions in the long term and thus avoiding local optimum as shown in Figure 2.20.

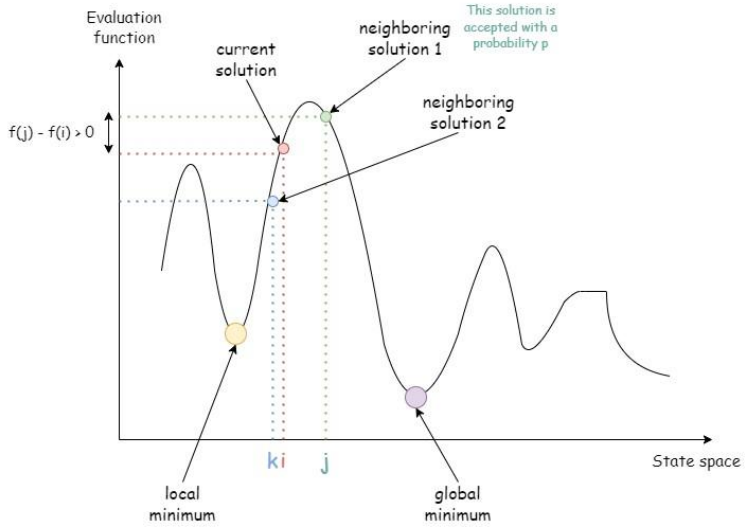


Figure 2.20 Simulated Annealing algorithm key feature

2.3.5 Tabu Search (TS)

When accepting non-improving moves, there is a risk of revisiting previously encountered solutions, regressing to a prior local optimum, or, more broadly, cycling through solutions. This cyclic behavior can result in a wasteful consumption of computational resources. Tabu Search addresses this by allowing the algorithm to move to neighboring solutions that are worse than the current solution if no improving neighbors are available. The algorithm also discourages revisiting explored search space by adding every visited solution to a data structure called the tabu list. The tabu list objects can be anything related to the move, such as the moved entity, move, solution, ...

Tabu Search algorithm is illustrated in Figure 2.21. The goal of this algorithm is to find the optimal solution by maximizing the objective function.

Input:
- So: Initial solution generated randomly or through a heuristic
- TabuSizeLimit: Size of tabu list

Output: The best solution

Function Tabu_Search(So, *TabuSizeLimit*)
Begin
currentSolution <- So
currentSolution.f <- evaluateSolution(*currentSolution*)
bestSolution <- *currentSolution* // Initially the best solution is the current solution

tabuList <- [] // Initialize the Tabu List as an empty list

while (stopping condition is not satisfied) **do**

```
    // Generate the current solution's neighbors and compute their f values
    neighbors <- generateNeighbors(currentSolution)
    neighborsNotTabu = []
    for neighbor in neighbors do
      neighbor.f <- evaluateSolution(neighbor)
      // The tabuObject can be an entity, a move, or a solution
      if (tabuObject(neighbor) not in tabuList) then
        neighborsNotTabu.add(neighbor)
      end if
    end for
    if (neighborsNotTabu is empty) then
      return bestSolution
    else
      bestNeighbor <- selectBestNeighbor(neighborsNotTabu)
      currentSolution <- bestNeighbor
      tabuList.push(tabuObject(currentSolution))
      if (size(tabuList) > TabuSizeLimit) then
        tabuList.removeFirst()
      end if
      if (currentSolution.f > bestSolution.f) then
        bestSolution <- currentSolution
      end if
    end if
  end while

  return bestSolution
End
```

Figure 2.21 Tabu Search algorithm

Example: In Figure 2.22, the application of the Tabu Search algorithm to solve the 4-queens problem is illustrated. Assuming a tabu list size of 2, the tabu object is defined as the entity representing the queen's column.

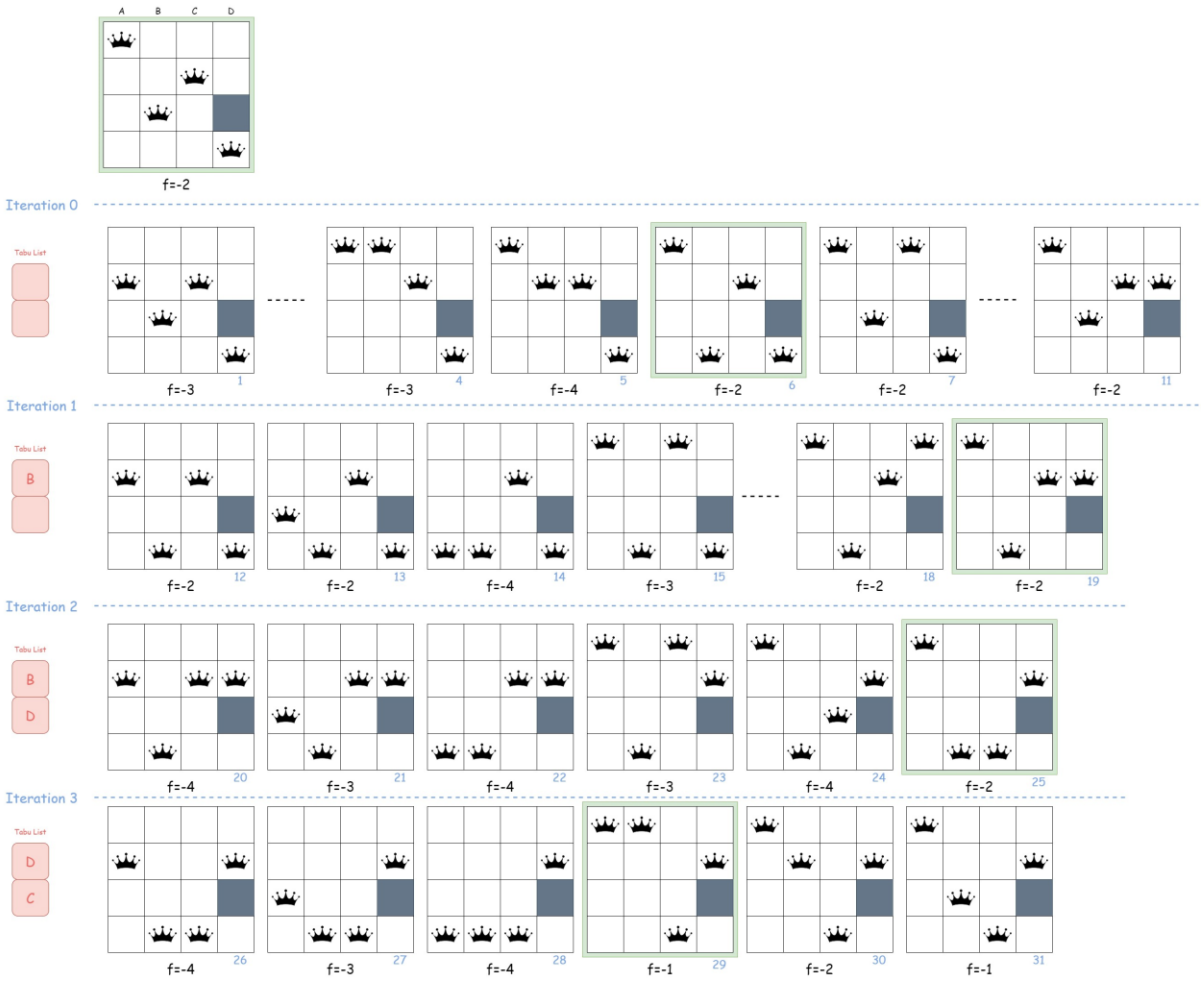


Figure 2.22 Application of Tabu Search algorithm on the 4-queens problem

2.3.6 Genetic Algorithm

Nature has always been a great source of inspiration for mankind. Genetic Algorithm (GA) is a search-based algorithm rooted in the principles of natural selection and genetics. GA belongs to a broader field of computation known as Evolutionary Computation.

In Genetic Algorithm (GA), a pool or population of potential solutions to the given problem is maintained. These solutions undergo recombination and mutation processes in natural genetics, to generate new offspring. The iterative process continues over various generations. Each individual (candidate solution) is assigned a fitness value, typically based on its objective function performance. Fitter individuals have a higher probability of mating, giving rise to potentially more fit individuals in subsequent generations. This concept aligns with the Darwinian Theory of "Survival of the Fittest".

1) Formulation

In Genetic algorithm, an analogy is made between the genetic system and the optimization problem by using the following equivalences:

- ✧ *Individual/chromosome/sequence:* a potential solution to the problem. Each chromosome is composed of a set of genes. Hence, each candidate solution is represented as an array of parameter values. If a problem has N_{par} dimensions (parameters), then each chromosome is encoded as an N_{par} -element array: $Chromosome = [p1, p2, p3, \ldots, p_{Npar}]$

where each element p_i represents a particular value of the i^{th} parameter.

- *Population:* a set of chromosomes or points in the search space.
- *Fitness function:* the function to be optimized. A fitness function should return higher values for better individuals.

2) Mechanism

Here's how it operates:

- **Initialization:** A genetic algorithm starts with a randomly chosen assortment of chromosomes, constituting the first generation (initial population). Subsequently, each chromosome in the population undergoes evaluation by the fitness function to assess how effectively it addresses the problem at hand.
- **Selection:** Now, the selection operator chooses some of the chromosomes for reproduction based on a probability distribution. The fitter a chromosome is, the more likely it is to be selected. Note that the selection operator chooses chromosomes with replacement, allowing the possibility of selecting the same chromosome more than once.
- **Crossover:** The crossover operator resembles the biological crossing over and recombination of chromosomes in cell meiosis. This operator swaps a subsequence of two of the chosen chromosomes to create two offspring. For example, if the parent chromosomes 11010111001000 and 01011101010010 are crossed over after the fourth bit, then 01010111001000 and 11011101010010 will be their offspring. Note that typically, crossover occurs with a probability of 1. However, it is possible that crossover does not occur, in which case the offspring are exact copies of their parents.
- **Mutation:** The mutation operator randomly flips individual elements in the new chromosomes. Typically, mutation occurs with a very low probability, such as 0.001. While the mutation operator may seem unnecessary at first glance, it plays an important role, even if it is secondary to those of selection and crossover. Selection and crossover maintain the genetic information of fitter chromosomes, but these chromosomes are only fitter relative to the current generation. This can cause the algorithm to converge too quickly. In other words, the algorithm can get stuck at a local optimum before finding the global optimum. The mutation operator helps protect against this problem by maintaining diversity in the population, but it can also make the algorithm converge more slowly.
- Typically, the selection, crossover, and mutation process continues until the number of offspring is the same as the initial population, ensuring that the first generation is completely replaced by the newly generated one.
- **Elitism (optional):** Some genetic algorithms may incorporate strategies such as elitism, where highly fit individuals from the current generation are allowed to survive unchanged into the next generation. This helps preserve the best solutions found so far and can enhance the algorithm's efficiency in converging towards optimal solutions.
- Now the second generation is tested by the fitness function, and the cycle repeats. It is a common practice to record the chromosome with the highest fitness from each generation, or the “best-so-far” chromosome.
- **Termination:** Genetic algorithms are typically iterated through multiple generations until the fitness value of the "best-so-far" chromosome stabilizes, indicating convergence to a solution or a set of solutions. The entire process of these iterations is commonly referred to as a "run."
- **Solution:** At the end of each run, there is usually at least one chromosome that represents a highly

fit solution to the original problem. The specifics can vary based on how the algorithm is implemented, and this could be the most fit among all the "best-so-far" chromosomes throughout the run or the most fit within the final generation.

Genetic algorithm is illustrated in Figure 2.23. The goal of this algorithm is to find the optimal solution by maximizing the fitness function.

```
Input:
- populationSize: The size the population
- crossoverProbability: Usually the crossover probability is 1
- mutationProbability: Usually the mutation probability is very small (i.e., 0.001)
Output: The best solution
Function Genetic_Algorithm(populationSize, crossoverProbability, mutationProbability)
Begin
  // Create the initial population and evaluate the fitness of each chromosome in this population
  population <- generateInitialPopulation(populationSize)
  for chromosome in population do
    chromosome.f <- evaluateFitness(chromosome)
  end for

  // Initially the best solution is the best chromosome within the initial population
  bestSolution <- selectBestChromosome(population)

  while (stopping condition is not satisfied) do

    newPopulation <- []
    while (size(newPopulation) < size(population)) do
      parent1, parent2 <- selection(population) // Select the two parents
      offspring <- crossover(parent1, parent2, crossoverProbability)
      mutation(offspring, mutationProbability)
      newPopulation.add(offspring) // add the offspring to the new population
    end for

    population <- newPopulation // Replace the old population with the newly generated population
    currentSolution <- selectBestChromosome(population)
    // Save the best chromosome so far
    if (currentSolution.f > bestSolution.f) then
      bestSolution <- currentSolution
    end if
  end while

  return bestSolution
End
```

```
Function crossover(parent1, parent2, crossoverProbability) // Single-point crossover
Begin
  prob_crossover <- random probability // Generate a random probability
```

```
    if (prob_crossover ≤ crossoverProbability) then
      crossoverPoint <- random value between 1 and chromosome's length
      // Apply a crossover according to the crossover point
      offspring1 <- concatenate (parent1[1, crossoverPoint], parent2[crossoverPoint+1, length(parent2)])
      offspring2 <- concatenate (parent2[1, crossoverPoint], parent1[crossoverPoint+1, length(parent2)])
      return [offspring1, offspring2]
    else
      return [parent1, parent2]
    end if
End
```

```
Function mutation(offspring, mutationProbability) // Flip mutation
Begin
  for chromosome in offspring do
    for gene in chromosome do
      prob_mutate <- random probability // Generate a random probability
      if (prob_mutate ≤ mutationProbability) then
        flip(gene) // Apply the mutation on the gene of the chromosome
      end if
End
```

Figure 2.23 Genetic Algorithm

3) Solution representation

In genetic algorithms, the representation of individuals (chromosomes) plays a crucial role in defining the structure of the search space. Different types of problems require different representations. Here are some common types of representations:

- **Binary Representation:** Genes are represented as binary strings (0s and 1s). Commonly used for problems with binary decision variables.
- **Integer Representation:** Genes are represented as integers. Suitable for problems where the variables are naturally integer values.
- **Real-Valued Representation:** Genes are represented as real numbers. Appropriate for continuous optimization problems.
- **String Representation:** Genes are represented as strings of characters. Useful for problems where the solution can be naturally expressed as a string.
- **Matrix Representation:** Genes are represented as matrices. Suitable for problems where solutions can be expressed in a matrix form.
- **Mixed Representation:** Combining different types of representations within the same individual to address complex problems.

4) Selection strategies

The selection is done proportionally to the fitness function. An individual with a high fitness value is more likely to pass on its traits to the next generations. We have different selection strategies including:

- Random selection
- Roulette wheel selection
- Tournament selection
- Rank selection

- Boltzmann Selection

✧ **Roulette wheel selection:**

The principle of roulette selection involves a linear search through a roulette wheel, where the slots are weighted in proportion to the fitness values of individuals, as shown in Figure 2.24.

Each individual is then assigned a slice on the roulette wheel based on their fitness proportion. The wheel is spun N times (where N is the population size), and on each spin, the individual beneath the wheel's marker is selected to be part of the pool of parents for the next generation.

The expected value of an individual to be selected is determined by dividing its fitness by the total fitness of the population. Thus, we select the chromosomes that will reproduce based on their fitness values, using the following selection probability:

$$SP_i = \frac{f(x_i)}{\sum_{k=1}^{4} f(x_k)}$$

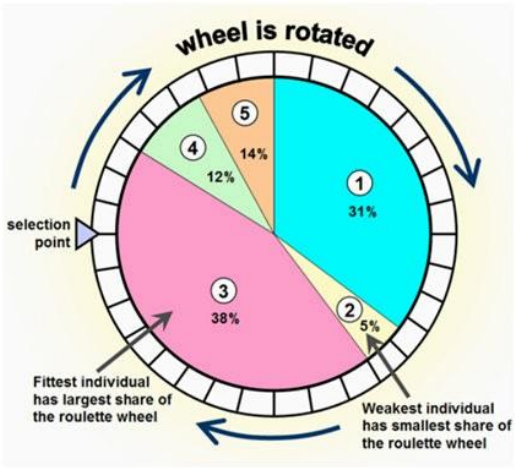


Figure 2.24 Roulette wheel selection

Example 1:

Consider the problem of maximizing the function:

$$\mathrm{f(x)} = -\frac{x^2}{10} + 3x$$

where x is allowed to vary between 0 and 31.

To solve this problem using the genetic algorithm, we must encode the possible values of x as chromosomes. For this example, we will encode x as a binary integer of length 5. Thus, the chromosomes for our genetic algorithm will be sequences of 0's and 1's with a length of 5 bits, and have a range from 0 (00000) to 31 (11111).

To begin the algorithm, we select an initial population of 4 chromosomes at random. The resulting initial population of chromosomes is shown in Table 3.1. Next, we take the x-value that each chromosome represents and evaluate its fitness. The resulting values are recorded in the fourth column of Table 2.1.

Table 2.1 Evaluation of the Initial population

Chromosome N°	Chromosome	x	f(x)	Selection	Expected Count	Actual Count (AC)

	representation			Probability (SP) $f(x)/\sum f(x)$	(EC) $f(x)/Avg\ f(x)$	$round(EC)$
1	1 1 0 1 0	26	10.4	0.19	0.75	1
2	0 0 0 1 0	2	5.6	0.10	0.40	0
3	0 1 1 0 0	12	21.6	0.39	1.57	2
4	1 0 1 1 0	22	17.6	0.32	1.28	1
	Total		55.2	1	4	4
	Average		13.8			

Since our population has 4 chromosomes and each parental combination produces 2 offspring, we need 2 combinations to produce a new generation of 4 chromosomes. We see that if the Roulette wheel is spun four times, we'll get 12 twice and 26 and 22 once. So possible parental combinations are (26, 12) and (12, 22).

The selected chromosomes are displayed in Table 2.2. To create their offspring, a crossover point is chosen at random, which is shown in the table as a vertical line. Here the crossover probability is 1.

Table 2.2 Evaluation of the population after crossover

Parent N°	Parent representation	Offspring	x	f(x)
1	1 1 | 0 1 0	1 1 1 0 0	28	5.6
3	0 1 | 1 0 0	0 1 0 1 0	10	20
3	0 | 1 1 0 0	0 0 1 1 0	6	14.4
4	1 | 0 1 1 0	1 1 1 1 0	30	0

Lastly, each gene (bit) of the new chromosomes mutates with a low probability. For this example, we let the probability of mutation be 0.001. Table 2.3 shows the new offspring after mutation.

Table 2.3 Evaluation of the population after mutation

Offspring N°	Offspring representation	Offspring after mutation	x	f(x)
1	1 1 1 0 0	1 1 1 0 0	28	5.6
2	0 1 0 1 0	0 1 1 1 0	14	22.4
3	0 0 1 1 0	0 0 1 1 0	6	14.4
4	1 1 1 1 0	1 1 1 1 0	30	0

We can see that the maximum f(x) value has increased from 21.6 to 22.4 in the new population.

Example 2: 8-queens problem

We take as a fitness function: the number of pairs of queens that do not attack each other: $fitness(s) = 28 - number\_attacks(s)$, min=0 and max=28. Let's take the following chromosomes as the initial population: (1) 21432102, (2) 00000000, (3) 21641300, (4) 13304013. The chromosomes of the initial population are shown in Figure 2.25.

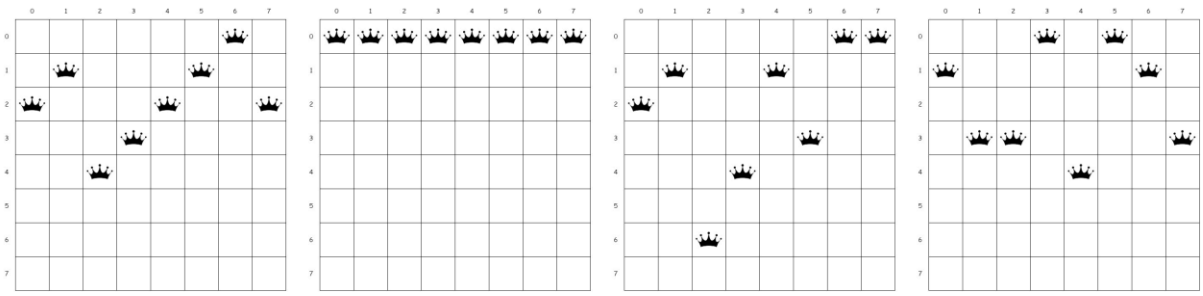


Figure 2.25 Initial population

The evaluation of the initial population is shown in Table 2.4.

Table 2.4 Evaluation of the Initial population

Chromosome N°	Chromosome representation	f(x)	Selection Probability (SP) $f(x)/\sum f(x)$	Expected Count (EC) $f(x)/Avg\ f(x)$	Actual Count (AC) $round(EC)$
1	21432102	11	0.20	0.81	1
2	00000000	0	0	0	0
3	21641300	23	0.43	1.70	2
4	13304013	20	0.37	1.48	1
	Total	54	1	4	4
	Average	13.5			

We observe that after spinning the Roulette wheel four times, the outcomes are 21641300 (twice), 21432102, and 13304013 (once). Consequently, the parental combinations are (21432102, 21641300) and (21641300, 13304013).

The chosen chromosomes are presented in Table 2.5. To generate their offspring, a random crossover point is selected and denoted in the table by a vertical line. The generated offspring are shown in Figure 3.26.

Table 2.5 Evaluation of the population after crossover

Parent N°	Parent representation	Offspring	f(x)
1	2143 | 2102	21431300	19
3	2164 | 1300	21642102	20
3	216 | 41300	21604013	22
4	133 | 04013	13341300	18

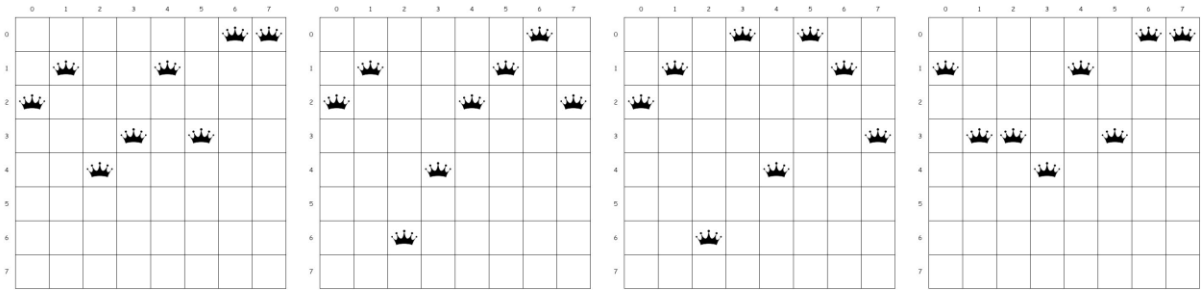


Figure 2.26 Population after crossover

Finally, each gene (digit) of the new chromosomes undergoes mutation with a low probability. In this example, we set the mutation probability to 0.001. Table 3.6 and Figure 2.27 illustrate the resulting offspring after mutation.

Table 2.6 Evaluation of the population after mutation

Offspring N°	Offspring representation	Offspring after mutation	f(x)
1	21431300	21431300	19
2	21642102	21612102	17
3	21604013	22604013	23
4	13341300	13341300	18

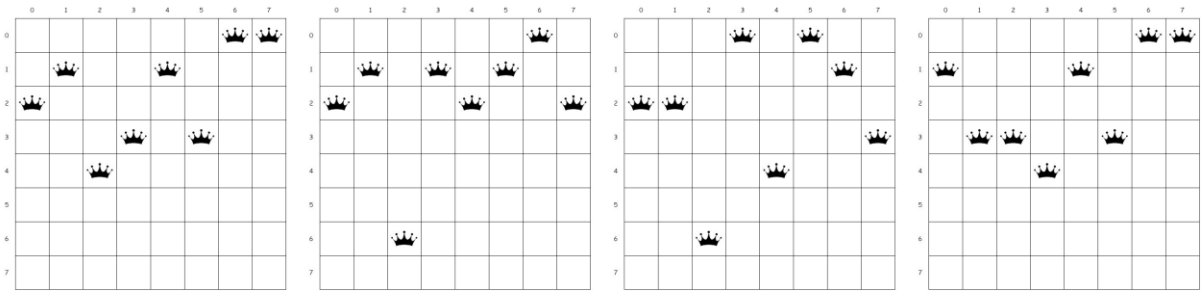


Figure 2.27 Population after mutation

We can see that the maximum f(x) value is still 23 in the new population.

4) Crossover strategies

The traditional genetic algorithm uses single-point crossover, where the two mating chromosomes are cut once at corresponding points, and the sections after the cuts are exchanged. We have other types of crossover strategies, such as:

- Two-Point Crossover
- Multi-Point Crossover
- Partially Mapped Crossover (PMX)
- Order Crossover (OX)
- Cycle Crossover (CX)
- Heuristic Crossover

Example 3: Travelling Salesman Problem (TSP)

The travelling Salesman Problem (TSP) is a classic optimization dilemma. In this scenario, there is an initial city and several other cities to visit. The objective is for a salesman to embark on a journey from the first city, visiting all other cities exactly once.

In summary, the key characteristics of the TSP problem are as follows:

- The journey begins in the initial city
- There are several cities, and each must be visited exactly once
- The travel route must not return to the initial city until all destination cities have been visited

The primary aim of the TSP is to minimize the total distance traveled by the salesperson, achieved by optimizing the order in which the cities are visited.

✧ **Solution representation**

All the cities are sequentially numbered starting from one. The route between the cities is described with an array, where each element represents the number of a city. This array signifies the sequence in which the cities are traversed to form a tour. It is essential that each chromosome contains every city exactly once. For example, the chromosome shown in Figure 3.28 represents a tour starting from city 1, traversing through city 4, and so on, eventually returning to city 1.

Figure 2.28 Solution representation in TSP

✧ **Crossover techniques**

To solve the traveling salesman problem, a simple crossover reproduction scheme does not work as it makes the chromosomes inconsistent i.e., some cities may be repeated while others are missed out. The drawback of the simple crossover mechanism is illustrated in Figure 2.29. As can be seen below, cities 6 and 7 are missing in offspring1 while cities 2 and 4 are visited more than once. Offspring2 too suffers from similar drawbacks. Hence, the need for other crossover techniques.

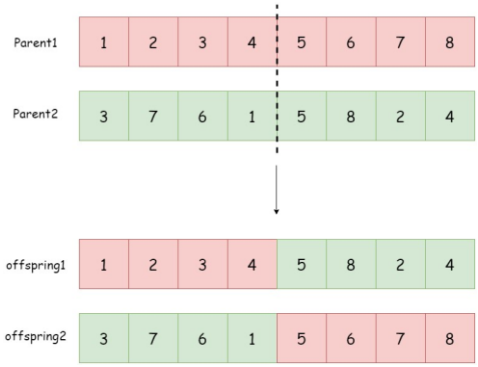


Figure 2.29 Single-point crossover in TSP

a) Partially Mapped Crossover Operator

In the partially mapped crossover (PMX), after choosing two random cut points on parents to build offspring, the portions between the cut points from the two parents are exchanged. For example, consider two parent tours with randomly chosen cut points between the 3rd and 4th bits and between the 6th and 7th bits (the two cut points marked with |):

$$P1 = (348 \mid 271 \mid 65)$$

$$P2= (425 \mid 168 \mid 37)$$

The mapping sections are between the cut points. In this example, the mapping sections are 2↔1,7↔6 and 1↔8. Now, the information from the corresponding mapping sections is exchanged between the parents to create offspring as follows:

$$O1= (\times\times\times \mid 168 \mid \times\times)$$
$$O2= (\times\times\times \mid 271 \mid \times\times)$$

Then, we can fill in the remaining bits (from the original parents) for those positions where no conflict arises, as follows:

$$O1= (34\times \mid 168 \mid \times5)$$
$$O2= (4\times5 \mid 271 \mid 3\times)$$

Hence, the third × in the first offspring is 8, which comes from the first parent. However, since 8 is already present in this offspring, we check the mapping 1↔8 and find that 1 is also present. Further checking the mapping 2↔1, so 2 occupies the third ×. Similarly, the seventh × in the first offspring is filled with 6, inherited from the first parent. However, as 6 is already in this offspring, we refer to the mapping 7↔6 as well, so 7 occupies the seventh ×. Thus, the configuration of the first offspring is determined:

$$O1= (342 \mid 168 \mid 75)$$

Analogously, we proceed to complete the second offspring in a similar fashion:

$$O2= (485 \mid 271 \mid 36)$$

b) Order Crossover Operator

The order crossover (OX) builds offspring by choosing a subtour of one parent while preserving the relative order of bits from the other parent. Consider, for example, two parent tours as follows (with randomly chosen cut points marked with |):

$$P1= (348 \mid 271 \mid 65)$$
$$P2= (425 \mid 168 \mid 37)$$

The offspring are produced in the following way: First, the bits are copied down between the cuts in a similar manner into the offspring, resulting in:

$$O1= (\times\times\times \mid 271 \mid \times\times)$$
$$O2= (\times\times\times \mid 168 \mid \times\times)$$

After this, starting from the second cut point of one parent, the bits from the other parent are copied in the same order, omitting existing bits. The sequence of the bits in the second parent from the second cut point is 3→7→4→2→5→1→6→8. After the removal of bits 2, 7, and 1, which are already in the first offspring, the new sequence is 3→4→5→6→8. This sequence is then placed in the first offspring starting from the second cut point:

$$O1= (568 \mid 271 \mid 34)$$

Analogously, we proceed to complete the second offspring using a similar process:

$$O2= (427 \mid 168 \mid 53)$$

c) Cycle Crossover Operator

The cycle crossover (CX) operator is utilized to create offspring in a manner where each bit, along with its position, comes from one of the parents. For example, consider the tours of two parents:

$$P1= (12345678)$$
$$P2= (85213647)$$

Now, the choice of the first bit for the offspring is ours, and it can either come from the first or second parent. In our example, the first bit of the offspring can be either 1 or 8. Let's choose it to be 1:

$$O1= (1\times\times\times\times\times\times\times)$$

Now, each bit in the offspring must be taken from one of its parents based on the same position. This leaves us with no further choice, and the next bit to be considered is bit 8. The bit from the second parent is just below the selected bit 1. In the first parent, this bit is at the 8^{th} position; thus:

$$O1= (1\times\times\times\times\times\times8)$$

This, in turn, implies bit 7, which is the bit of the second parent just below the selected bit at the 7^{th} position in the first parent. Thus:

$$O1= (1\times\times\times\times\times78)$$

Subsequently, this constraint compels us to place 4 at the 4^{th} position, as:

$$O1= (1\times\times4\times\times78)$$

After this, 1 is encountered, which is already present in the list; thus, we have completed a cycle and proceed to fill the remaining blank positions with the bits from those positions in the second parent:

$$O1= (15243678)$$

Similarly, the second offspring is determined following the same process:

$$O2= (82315647)$$

However, there is a drawback to this technique as it may sometimes result in identical offspring. For instance, consider the following two parents:

$$P1= (34827165)$$
$$P2= (42516837)$$

After applying CX technique, the resultant offspring are as follows:

$$O1= (34827165)$$
$$O2= (42516837)$$

which are the exactly the same as their parents.

5) Mutation strategies

After crossover, the strings undergo mutation. Mutation serves to prevent the algorithm from getting trapped in a local optimum. Its role is twofold: recovering lost genetic materials and introducing random disturbances to genetic information.

There are many different forms of mutation for the different kinds of representation. Different genetic representations (binary, real-valued, permutation, etc.) may require specific mutation operators tailored to their characteristics.

a) Flip mutation

For binary representations, each bit in the chromosome has a small probability of being flipped (changed from 0 to 1 or vice versa). Figure 2.30 shows an example of flip mutation.

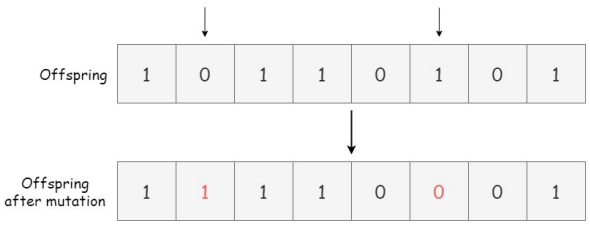


Figure 2.30 Example of flip mutation

b) Swap mutation

For permutation representations, two random positions in the chromosome are selected, and the elements at those positions are swapped. Figure 2.31 shows an example of swap mutation.

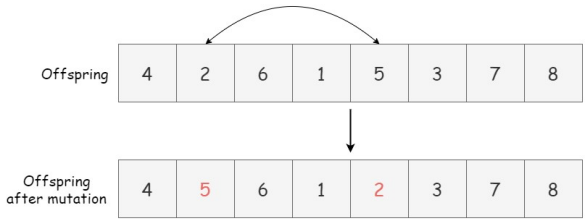


Figure 2.31 Example of swap mutation

c) Reversion mutation

A random position is chosen and the genes next to that position are reversed. This is shown in Figure 2.32.

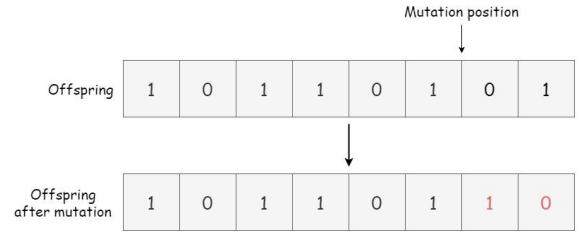


Figure 2.32 Example of reversion mutation

6) Properties:

The performance of the genetic algorithm depends highly on the method used to encode candidate solutions into chromosomes and what the fitness function is actually measuring. Other important details

are the probability of crossover, the probability of mutation, the size of the population, and the number of iterations. These values can be adjusted after assessing the algorithm's performance on a few trial runs.