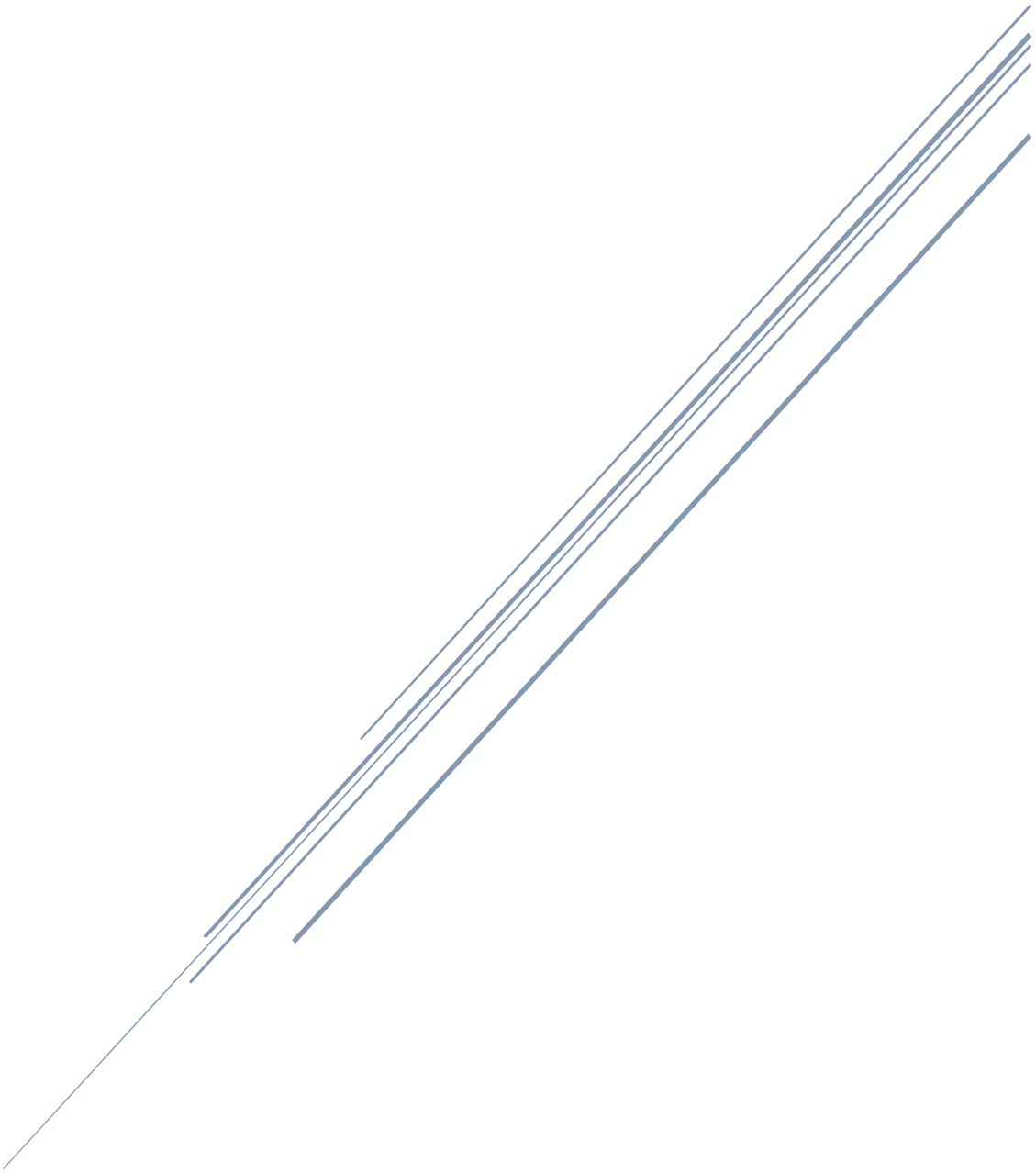
CHAPTER 2

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.

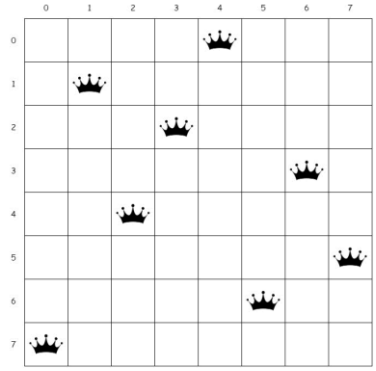


Figure 2.1 Possible optimal solution for the 8-queens problem

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.

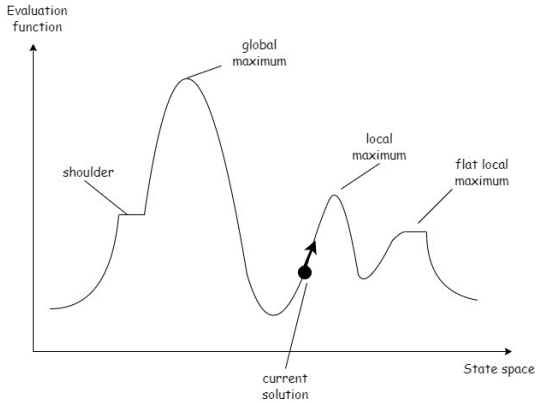


Figure 2.2 A one-dimensional state-space landscape for an optimization problem

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).

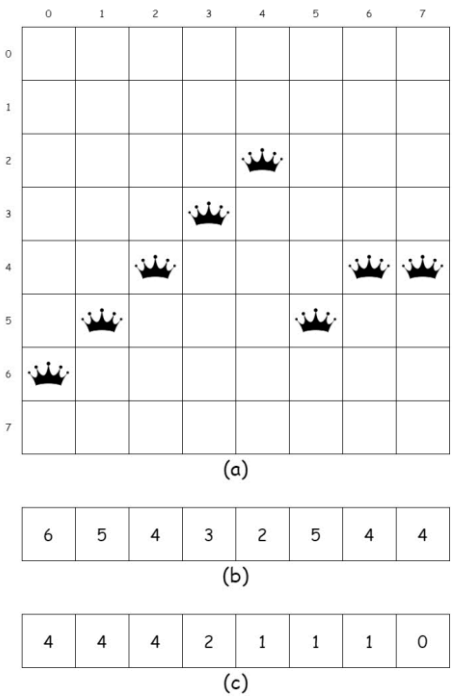


Figure 2.3 Solution representation for the 8-queens problem

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.

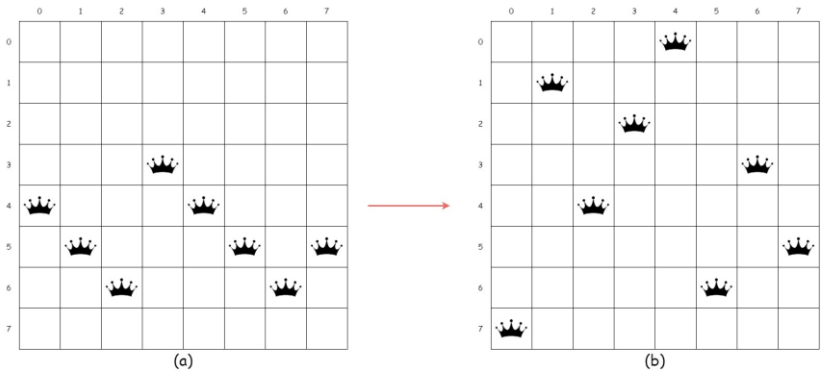


Figure 2.4 8-queens game. (a) An initial solution. (b) A possible optimal solution

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.

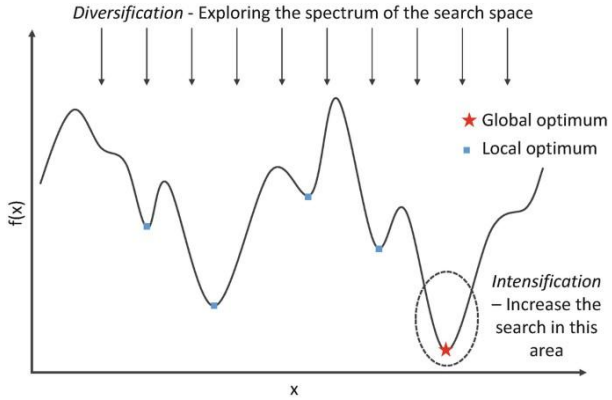


Figure 2.5 Diversification vs. Intensification

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.

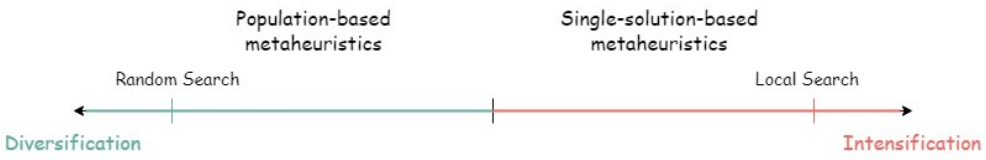


Figure 2.6 Design space of a metaheuristic

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).

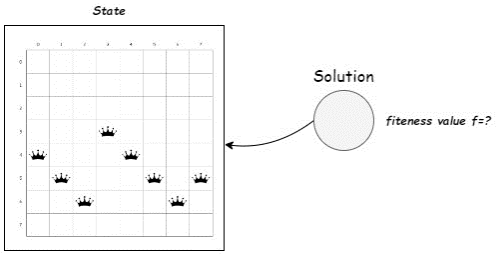


Figure 2.7 Structure of a solution in metaheuristics

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).

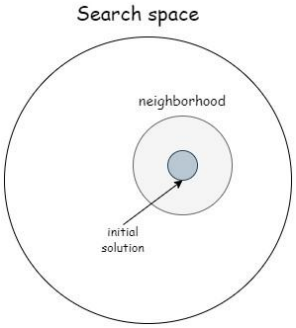


Figure 2.9 Neighborhood in the state space of local search

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.

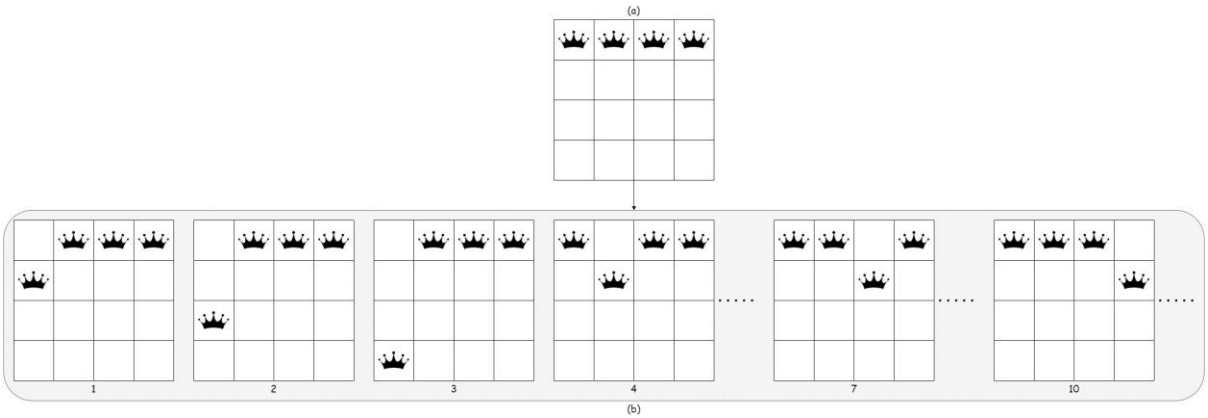


Figure 2.10 Example of a solution neighborhood in 4-queens problem

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.

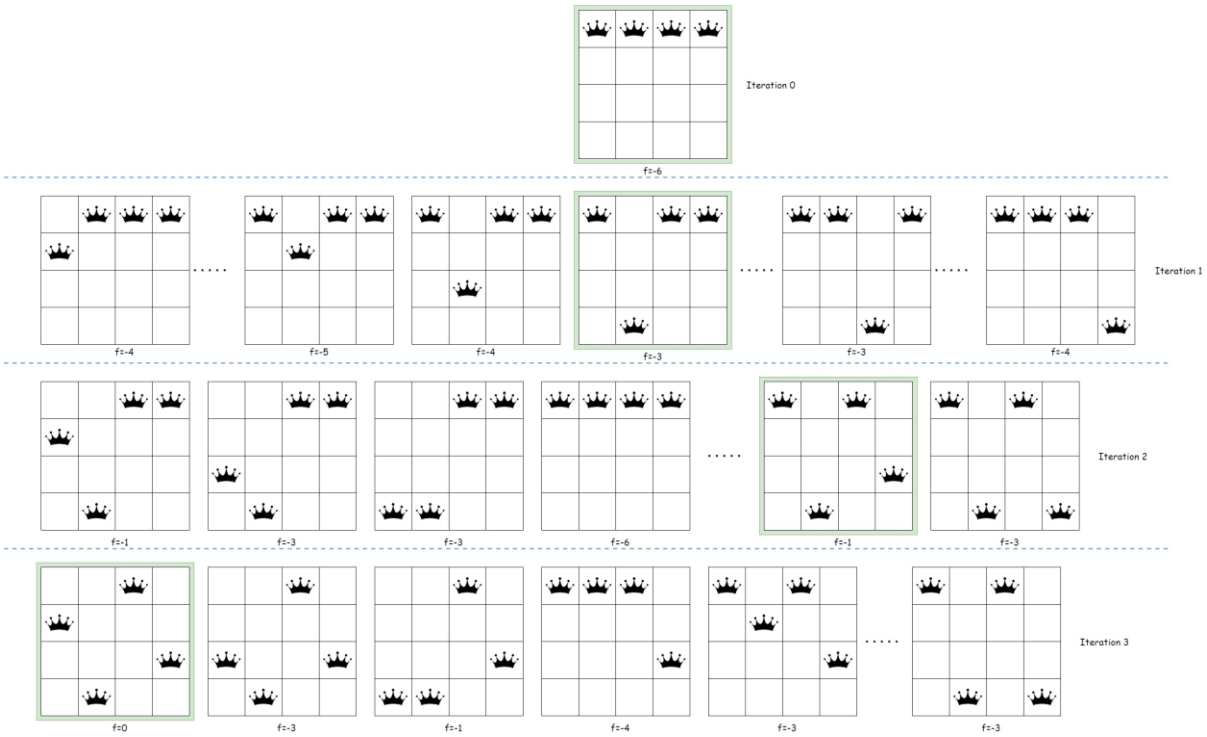


Figure 2.12 Application of Hill Climbing algorithm on the 4-queens problem

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.

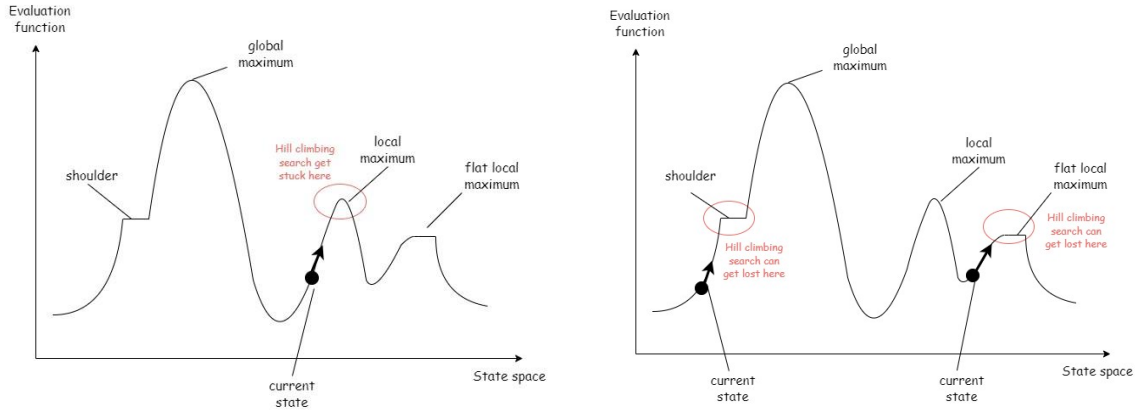
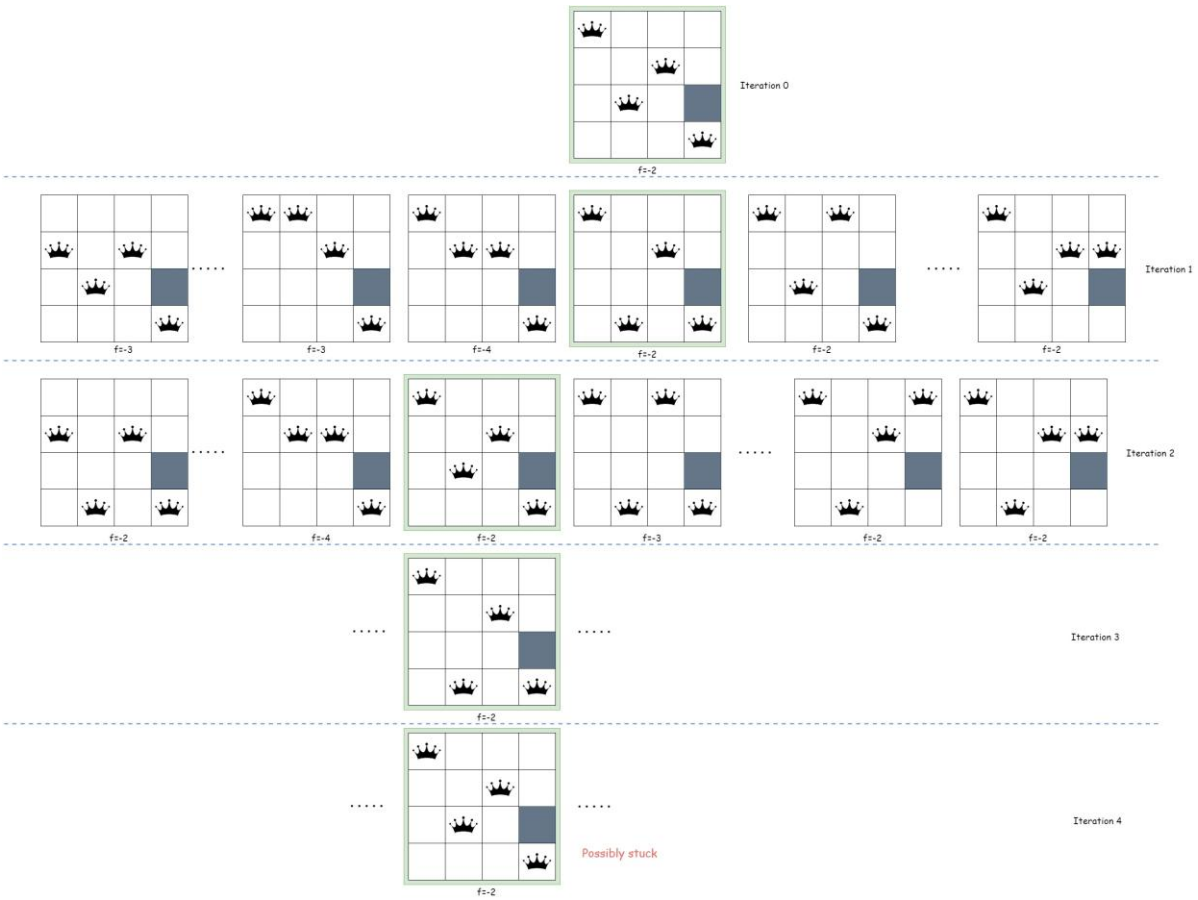


Figure 2.13 Hill Climbing limitations

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.