6 Tabu search

This chapter covers

- Understanding local search
- Understanding how tabu search extends local search
- Solving constraint-satisfaction problems
- Solving continuous problems
- Solving routing problems
- Solving assembly line balancing problems

In the previous chapter, you were introduced to trajectory-based metaheuristics, and you learned about simulated annealing (SA) as an example of these metaheuristic algorithms. The actual first use of a metaheuristic is probably Fred Glover's *tabu search* (TS) in 1986, although his seminal article on tabu search was published later, in 1997 [1]. The word "tabu" (also spelled "taboo") originated from the Polynesian languages of the South Pacific. It is a term used to describe something that is prohibited, forbidden, or considered socially unacceptable within a particular culture or society. Tabu search is called "tabu" because it uses a memory structure to keep track of solutions that have been recently explored so it can avoid returning to them, especially in the early stage of the search, in order to avoid getting stuck in local optima.

TS is a powerful trajectory-based optimization technique that has been successfully applied to solve different optimization problems in different areas, such as scheduling, design, allocation, routing, production, inventory and investment, telecommunications, logic and artificial intelligence, technology, graph optimization, and general combinatorial optimization. TS can be considered a combination of local search and memory structures.

This chapter presents tabu search as a trajectory-based metaheuristic optimization technique, discusses its pros and cons, and looks at its applications in different domains. To illustrate how this algorithm can be used to solve optimization problems, a variety of case studies and exercises will be presented. Let's start by closely exploring local search.

6.1 Local search

Imagine yourself enjoying a vacation at a resort that features multiple restaurants, each offering a diverse selection of dishes to satisfy your every craving. During the initial day of your stay, you might choose a restaurant randomly or select the nearest one to your room if you are exhausted from your journey. You may continue dining at that particular restaurant or explore other options within the resort. In this case, you are applying local search by limiting your options to those found within the resort, without considering the possibility of ordering food online or leaving the resort to dine elsewhere.

Local search (LS) is a search technique that iteratively explores a subset of a search space in the neighborhood of the current solution or state in order to improve this solution with local changes. The type of local changes that may be applied to a solution is defined by a *neighborhood structure*. For a finite set of candidate solutions S, a neighborhood structure represents a set of neighboring solutions $N(s) \subseteq S$ that can be generated by making a small change to the current solution $s \in S$. The horizon of $N(s)$ as a neighborhood of s varies from exploring all the possible neighbors of the current solutions (random search) to only considering one neighbor (local search). The former can be computationally demanding, while the latter has a very limited horizon or search space and is highly vulnerable to getting trapped in a local minimum.

As shown in algorithm 6.1, a local search algorithm starts from an initial feasible solution and iteratively moves to a neighboring solution as long as the new neighboring solution is better than the old one.

Algorithm 6.1 Local search

```
Input: an initial feasible solution
Output: optimal solution
Begin
    While termination criteria not met do
        Generate a neighboring solution by applying a series of local
modifications (or moves)
        if the new solution is better then
           Replace the old one
```

Typically, every feasible solution has more than one neighboring solution. The name "local search" implies that the algorithm searches for a new solution in the neighborhood of the current one. For example, hill climbing can be considered a local search technique where a new neighboring solution that is locally maximizing the criterion or objective function is considered in each iteration. The hill climbing algorithm is a greedy algorithm, as it accepts only improving solutions. This sometimes makes it converge to local optima, which are usually average solutions unless the search is extremely lucky. The solution quality and the computation time are usually dependent on the chosen local moves.

Local search algorithms have been successfully applied to solve many hard combinatorial optimization problems in reasonable time. Application domains include areas such as operations research, management science, engineering, and bioinformatics. The performance of LS-based approaches can be further enhanced by introducing mechanisms for escaping from local minima in the search space. These mechanisms include, but are not limited to, simulated annealing, random noise, mixed random walk, and tabu search. Tabu search was originally proposed to allow LS to overcome the difficulty of local optima and prevent cycling by allowing non-improving moves and memorizing the recent history of the search.

Let's now discuss the various components of TS.

6.2 Tabu search algorithm

Going back to our resort example, even if you enjoyed your first meal at a particular restaurant within the resort, you may opt to dine at a different one on the following day to explore other options and to avoid becoming trapped in a local optimum. Suppose you promise yourself not to dine at the same restaurant for several days so you can explore the other dining options at the resort. Once you have sampled various restaurants, you might opt to return to one of the restaurants you previously visited and dine there for the remainder of your stay. You apply tabu search by memorizing your impressions of each meal at each restaurant you try, and you can search for alternatives, taking into consideration your previously memorized favorites. This allows you to enhance your local search by using memory to explore the search space more flexibly and responsively beyond local optimality.

This example demonstrates that tabu search incorporates adaptive memory and responsive exploration. *Adaptive memory* involves remembering information that is relevant or useful during the search process, such as recent moves made by the algorithm and the promising solutions found. *Responsive exploration* is a problem-solving approach that adapts and adjusts the behavior of the solver based on new information and the search history to find superior solutions faster.

Tabu search

"Tabu search is based on the premise that problem solving, in order to qualify as intelligent, must incorporate adaptive memory and responsive exploration. The adaptive memory feature of TS allows the implementation of procedures that are capable of searching the solution space economically and effectively. Since local choices are guided by information collected during the search, TS contrasts with memoryless designs that heavily rely on semi-random processes that implement a form of sampling. The emphasis on responsive exploration in tabu search, whether in a deterministic or probabilistic implementation, derives from the supposition that a bad strategic choice can often yield more information than a good random choice." (From Glover, Laguna, and Marti, "Principles of tabu search" [2].)

Tabu search is an iterative neighborhood search algorithm where the neighborhood changes dynamically. This algorithm was originally proposed to allow local search to overcome local optima. TS enhances local search by actively avoiding points in the search space already visited. By avoiding already visited points, loops in search trajectories are avoided and local optima can be escaped. Tabu search employs memory through a tabu list, which prohibits revisiting recently explored neighborhoods. This is done to avoid getting stuck in local optima. This combination can substantially increase the efficiency of solving some problems. The main feature of TS is the use of an explicit memory, which has two purposes: to avoid revisiting previously explored solutions and to explore unvisited regions of the solution space. The TS process starts with an initial randomized solution and then finds neighboring solutions. The best solution is then chosen and added to a tabu list. In subsequent iterations, tabu-active items are excluded as potential candidates unless enough time has elapsed and they can be reconsidered. This method helps prevent TS from getting stuck in local optima. Furthermore, to mitigate the effect of a tabu list excluding some good solutions, an aspiration criterion $A(s)$ can be employed, which allows previously tabu moves to be reconsidered if they result in a better solution than the current best-known solution.

Algorithm 6.2 shows how tabu search combines local search and memory structures.

Algorithm 6.2 Tabu search

```
Input: an initial feasible solution
Output: optimal solution
Begin
    While termination criteria not met do
        Choose the best: s'∈N(s) ← N(s)-T(s)+A(s)
        Memorize s' if it improves the best known solution
        s←s'
        Update Tab list T(S) and Aspiration criterion A(s)
```

As the algorithm shows, tabu search starts by using an initial feasible solution s and explores the search space iteratively to generate an optimal or near-optimal solution. At each iteration, and while the termination criteria are not met, the algorithm creates a candidate list of moves that lead to new solutions from the current solution within

the neighborhood $N(s)$. If the new solution s' is an improving solution that is not listed as tabu-active $T(s)$ or is an admissible solution considering the aspiration criteria $A(s)$, the obtained solution is designated as the new current solution. Admissibility is then revised by updating the tabu restrictions and aspiration criteria.

Figure 6.1 summarizes the steps of TS in a flowchart. We start by obtaining a solution from initialization or from an intermediate or long-term memory component. We then create a candidate list of moves by applying an operator on the current solution, such as swapping, deleting and inserting, etc., depending on the nature of the problem at hand. These candidate neighboring solutions are evaluated, and the best admissible candidate is chosen. We keep updating the admissibility conditions, tabu restrictions, and aspiration criteria if the stopping criteria are not satisfied.

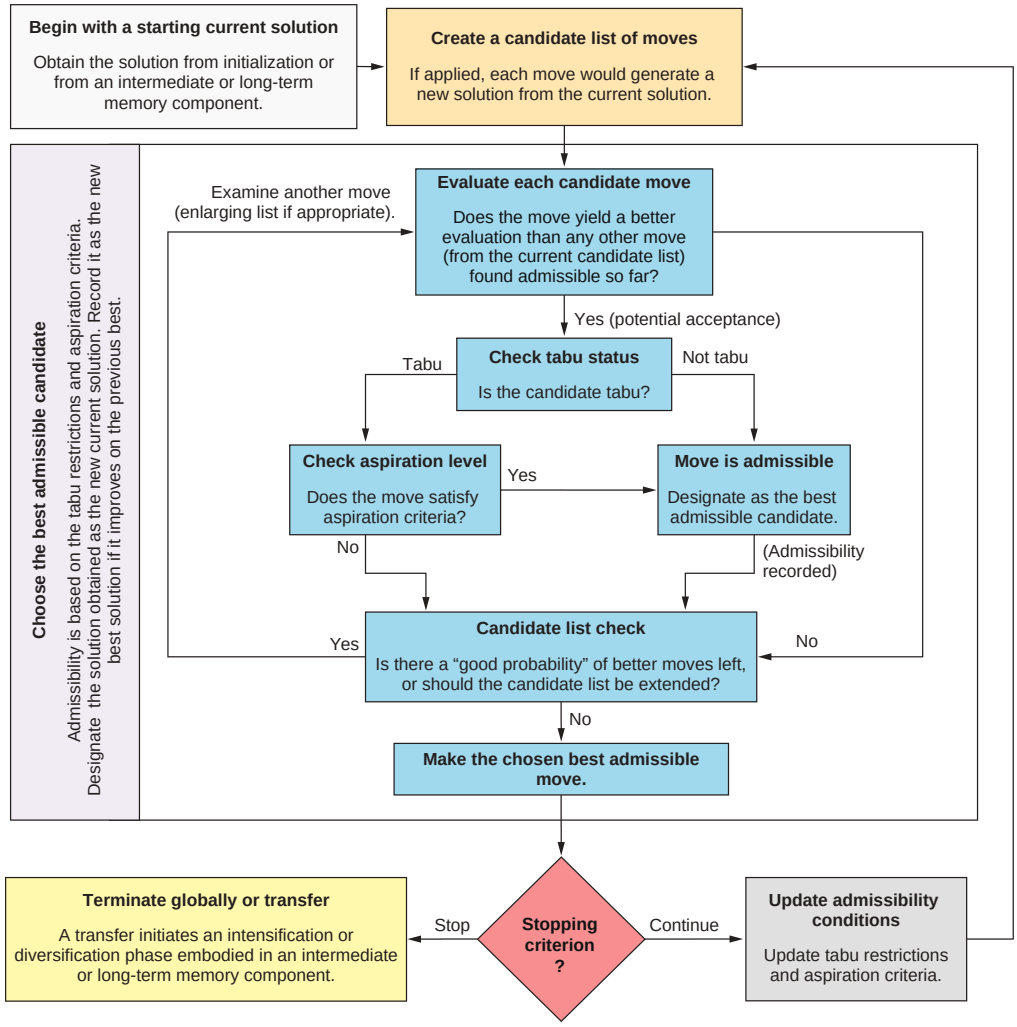


Figure 6.1 Tabu search steps (based on F. Glover's "Tabu search and adaptive memory programming—advances, applications and challenges" [1])

The following criteria may be used to terminate TS:

- The neighborhood is empty, meaning that all possible neighboring solutions have already been explored.
- The number of iterations performed since the last improvement exceeds a specified limit.
- There is external evidence that an optimal or a near-optimal solution has been reached.

To gain a better understanding of the TS algorithm, let's consider a simplified version of a symmetric traveling salesman problem (TSP) with only four cities, as illustrated in figure 6.2.

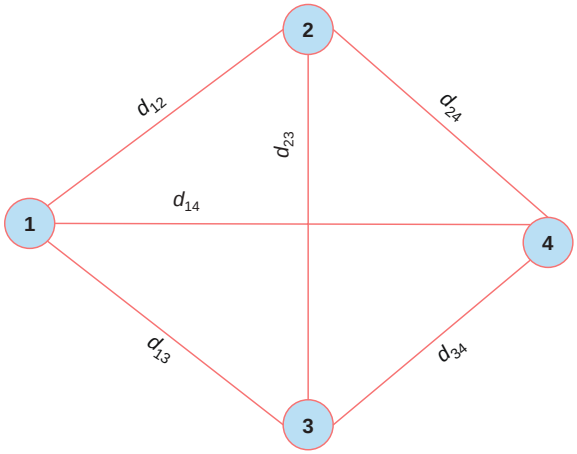


Figure 6.2 A 4-city TSP. The weights on the edges of the graph represent the travel distances between the cities.

A feasible solution can be represented as a sequence of cities or nodes where each city is visited exactly once. Assuming that the home city is city 1, an initial feasible solution can be selected randomly or using a greedy approach. A possible greedy approach is to choose the unvisited node closest to the current node and to continue this process until all nodes have been visited, resulting in a complete feasible tour that covers all nodes. This initial solution can be represented using permutation, such as {1,2,4,3}.

To generate a neighboring solution, we can apply a swapping operator. The neighborhood represents a set of neighboring solutions that can be generated by a pairwise exchange of any two cities in the solution. For this 4-city TSP, and fixing node 1 as the starting node or home city, the number of neighbors is the number of combinations without repetition $C(n,k)$ or n-choose-k:

$$C(n,k) = \frac{n!}{k!(n-k)!} = \frac{3!}{2!(3-2)!} = 3 \text{ neighbors}$$

Given the initial solution is {1,2,4,3}, the following three feasible neighboring solutions can be generated by applying the swapping operator:

- {1,2,3,4} by swapping 3 and 4
- {1,3,4,2} by swapping 2 and 3
- {1,4,2,3} by swapping 2 and 4

At each iteration, the neighboring solution with the best objective value (minimum total distance) is selected.

6.2.1 Memory structure

Local search strategies are often memoryless ones that keep no record of their past moves or solutions. The main feature of TS is the use of an explicit memory. *Explicit memory* refers to a mechanism that remembers the moves that have been previously visited during the search process. A simple TS usually implements the following two forms of adaptive memory mechanisms:

- *A recency-based or short-term memory*—This is a mechanism that keeps track of recently visited moves during the search process. It plays a role in preventing the algorithm from revisiting moves that have been explored recently.
- *A frequency-based or long-term memory*—This is a mechanism that tracks the historical frequency of specific moves throughout the entire search process and penalizes moves that have been visited frequently without success or that have proven to be less promising.

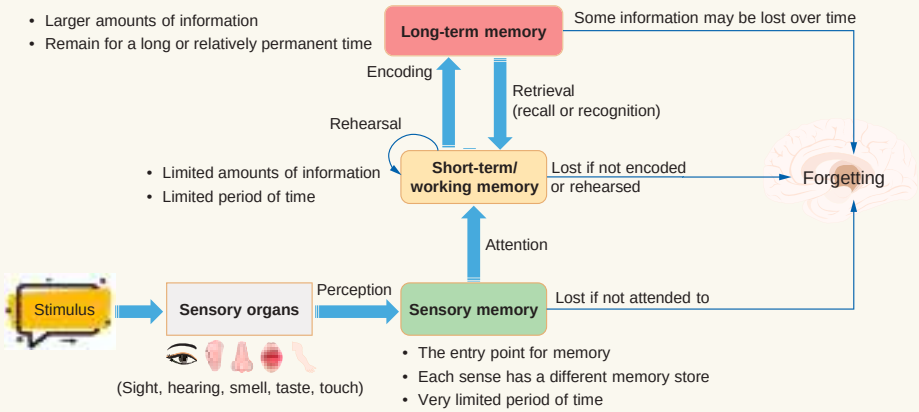
Memory types

According to the Atkinson–Shiffrin model (also known as the multi-store model or modal model), human memory has three components: sensory memory, working memory (sometimes called short-term memory), and long-term memory, as shown in the following figure.

Sensory memory is a very brief memory that automatically results from our perceptions and generally disappears after the original stimulus has ceased. Each of our five senses has a different memory store. For example, visual info is stored in iconic memory, while auditory info is stored in echoic memory.

The amount of information stored in short-term memory depends on the attention paid to the elements of sensory memory. Working memory is a more recent extension of the concept of short-term memory. This memory allows you to store and use the temporary information required to execute specific tasks. Rehearsal and repetition can help in increasing the duration of short-term memory. For example, imagine yourself as a customer-service associate in a fast food or beverage drive-thru, taking orders from customers and ensuring those orders are fulfilled. The order information provided by the customers is stored in your short-term or working memory, and once the order is fulfilled, this information is not kept in your memory.

(continued)



Memory types

Long-term memory holds your lifelong memories and a vast amount of information, such as your birthday, your address, work skills you've learned, etc. Some important information captured by working memory can be encoded and stored in long-term memory. The purpose of encoding is to assign a meaning to the information being memorized. For example, you might encode the word "omelet" as "egg, beaten, fried." If you could not recall the word "omelet" spontaneously, you can still retrieve it by invoking one of the indexes that you used to encode it, such as "egg." This is similar to encoding information using a lookup table for quick information retrieval.

Figure 6.3 shows a knapsack problem as an example. In this problem, each item has a *utility* and a *weight*, and we want to maximize the utility of the contents of the knapsack without exceeding the maximum weight. The problem is constrained by the capacity of the knapsack. Neighboring solutions can be generated by swapping items in and out of the knapsack.

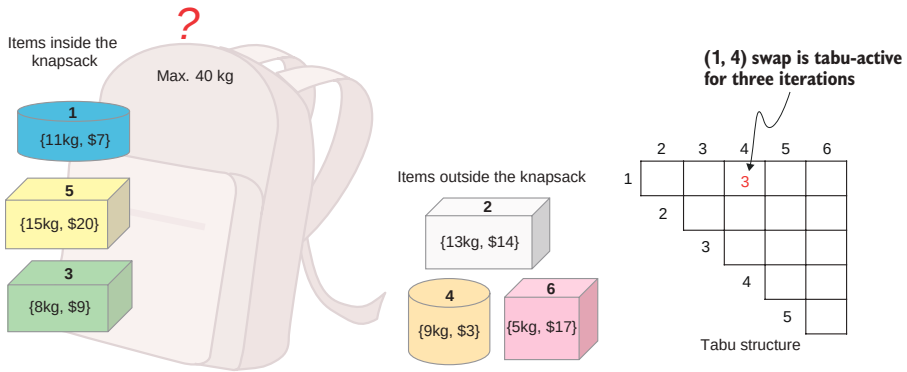


Figure 6.3 A knapsack problem

As illustrated in figure 6.3, a new candidate solution can be generated by swapping items 1 and 4. In this case, this swap will be tabu-active for the next three iterations, as shown in the tabu structure in the figure. *Tabu-active* moves are currently on the tabu list and cannot be selected for exploration in the current iteration. We could also generate neighboring solutions by adding or removing different items. If a neighborhood structure considers "add" and "remove" as separate moves, it might be a good idea to keep separate tabu lists for each type of move. Frequency-based memory keeps track of the frequency of the different swaps performed during a specified time interval. The idea is to penalize swaps that have been visited frequently.

The use of recency and frequency memory in TS serves primarily to prevent the searching process from cycling, which involves endlessly repeating the same sequence of moves or revisiting identical sets of solutions. Moreover, these two memory mechanisms play a role in achieving a trade-off between exploration and exploitation, as illustrated in figure 6.4.

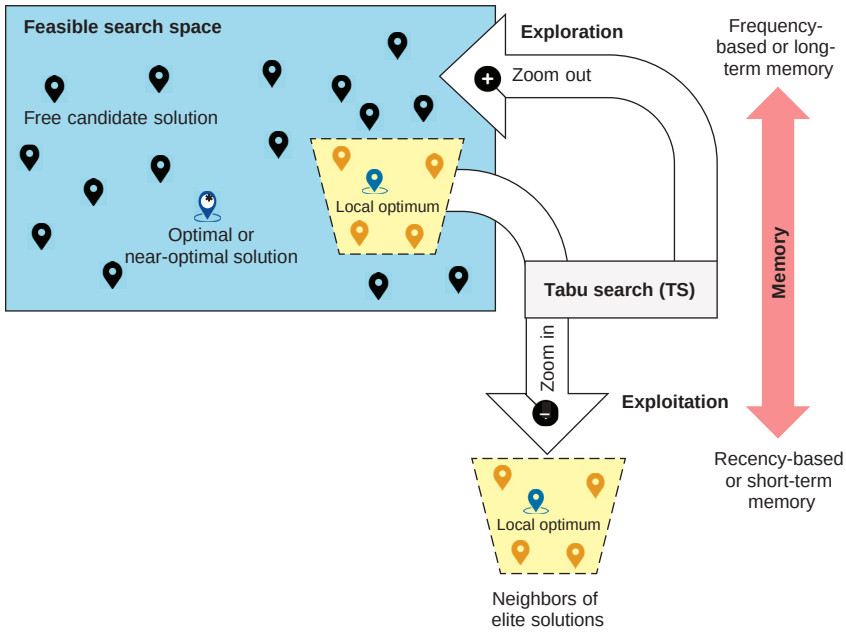


Figure 6.4 TS short-term memory and long-term memory and the search dilemma

The recency-based memory restricts the search to within a set of potentially prosperous or elite solutions to intensify the search while avoiding repetition or reversal of previously visited solutions. Frequency-based memory emphasizes the frequency of different moves to guide the algorithm toward new regions in the search space that might have not been explored. By discouraging the repetition of recent moves, recency-based memory contributes to exploration to a certain extent, but the primary reinforcement for exploration comes from frequency-based memory. The interplay between these

memory mechanisms maintains a balance, allowing the algorithm to efficiently navigate the feasible search space.

For the 4-city TSP, a tabu structure can be used to represent both forms of memory, as shown in figure 6.5. In recency-based memory, the tabu structure stores the number of iterations for which a given swap is prohibited.

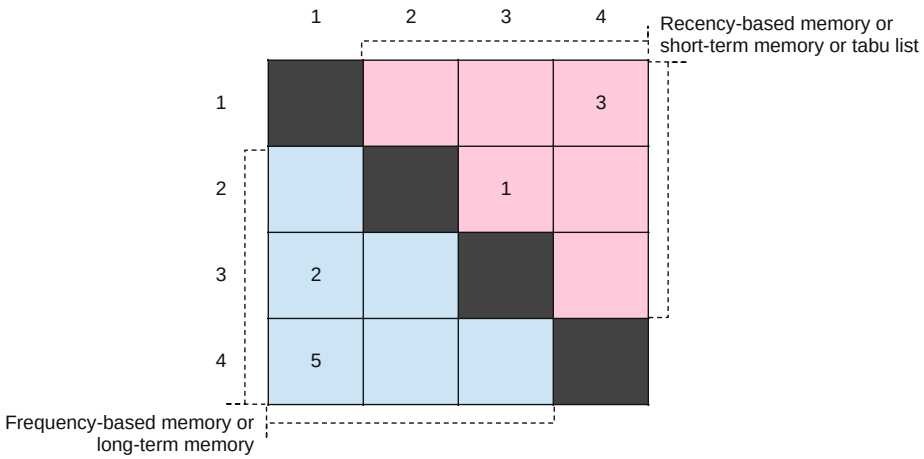


Figure 6.5 Tabu structure for the 4-city TSP. The numbers in recency-based memory represent the number of iterations remaining for tabu-active moves; the numbers in long-term memory represent the frequency count of using the move.

This recency-based memory mechanism is implemented using a tabu list as a data structure to keep track of the forbidden or tabu-active moves, preventing the algorithm from revisiting them for a specified number of iterations, called the *tabu tenure*. At each iteration, the tenure of each move already in the tabu list is decreased by 1, and those moves with zero tenure are dropped from the tabu list. The tabu tenure T can be chosen using different methods:

- *Static*—Choose T to be a constant, which may depend on the problem size, such as using guidelines like $\sqrt{N}$ or $N/10$ iterations where N is the problem size. It has been shown that a static tabu tenure cannot always prevent cycling [3].
- *Dynamic*—Choose T to vary randomly between a specific range T_{min} and T_{max} following the search progress. The threshold T_{min} and T_{max} can vary based on how the solution is improving during a certain number of iterations.

In the previous 4-city TSP example (figure 6.2), let's assume the tabu tenure is set as 3 iterations. If a solution is generated based on swap (1,4), this swap will be tabu-active for three iterations, meaning that it cannot be performed for the next three iterations.

Frequency-based memory, shown in the lower-left corner of figure 6.5, contains values that correspond to the frequency count of the swap. Whenever a swap occurs between two cities, the frequency counters of the respective swap values in the frequency

table will increase by 1. When searching for the optimal solution, the values in the frequency counter are taken into account as a penalty for solutions which have been visited frequently. A penalized value directly proportional to the frequency count can be added to the cost or the fitness function of the solution.

6.2.2 Aspiration criteria

Avoiding tabu-active moves is necessary, but some of these moves may possess significant potential. In such instances, the tabu restrictions may hinder promising solutions, even in the absence of cycling risks. This problem is known as *stagnation*. In tabu search, stagnation can occur when the algorithm keeps rejecting candidate moves because they are tabu-active, and all tabu-inactive moves have already been explored or are non-improving moves. This can result in the algorithm revisiting the same solutions repeatedly without making any significant progress toward better solutions.

Aspiration criteria can mitigate this stagnation by allowing the algorithm to consider moves that are tabu-active but that lead to better solutions than the current best solution. By temporarily lifting tabu conditions for certain attributes of the solution, the algorithm can explore new regions of the search space and potentially discover better solutions. A commonly used aspiration criterion in almost all tabu search implementations is to allow the tabu activation rule to be overridden if the move yields a solution better than the best obtained so far (the incumbent solution) and when few iterations are left before this tabu-active move will get out of the tabu list.

6.2.3 Adaptation in TS

TS is applicable in both discrete and continuous solution spaces. For some complex problems, such as scheduling, quadratic assignment, and vehicle routing, tabu search obtains solutions that often surpass the best solutions previously found by other approaches. However, to achieve the best results, many parameters need to be carefully tuned, and the number of iterations required may also be large.

As is the case for all metaheuristics algorithms, a global optimum may not be found, depending on the parameter settings. TS parameters include the initial solution generation method (random, greedy, heuristic, etc.), tabu tenure, neighborhood structure, aspiration criteria, stopping criteria, and penalized value of the frequency count. These parameters can be pretuned or autotuned to improve the performance of TS. Parameter tuning refers to finding suitable values for the different algorithm parameters before the algorithm is run, but adaptation can also be done on the fly while the algorithm is running, following deterministic, adaptive, or self-adaptive approaches to balance exploration and exploitation:

- *Deterministic tuning* is when the control parameter is changed according to some deterministic update rule without taking into account any information from the search algorithm.

- *Adaptive tuning* is when the update rule takes information from the search algorithm and changes the control parameter accordingly.
- *Self-adaptive tuning* is when the update rule itself is adapted.

One of the most important parameters of TS is the tabu tenure. Figure 6.6 illustrates the effect of tabu tenure on the performance of TS. A tabu tenure that is too short may result in frequent cycling, where the algorithm performs the same moves or revisits the same solutions in a repetitive manner. This hinders the exploration of diverse areas in the solution space and may prevent the discovery of optimal or near-optimal solutions. Moreover, tabu tenures that are too short may lift restrictions on moves quickly, potentially causing the algorithm to overlook promising solutions that were temporarily deemed unfavorable. In contrast, a tabu tenure that is excessively long may lead to stagnation, where certain moves remain prohibited for an extended period. This can prevent the algorithm from exploring new regions of the solution space, potentially hindering the discovery of better solutions. Moreover, long tabu tenures increase the memory footprint of the algorithm, potentially leading to inefficiency and increased computational demands. This can be particularly problematic for large-scale problems.

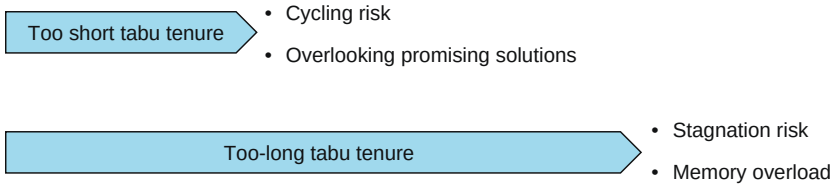


Figure 6.6 Effect of tabu tenure

One of the adaptive approaches incorporated into TS is to allow the length of the short-term memory (the tabu tenure) to vary dynamically and intensify the search when indicators identify promising regions or to promote diversification if the improvements seem to be minimal or a local optimum is detected. For example, you can set a lower bound $L_{\min}$ and an upper bound $L_{\max}$ for the tabu tenure. You can then decrement the tabu tenure by 1 if the solution has improved over the last iteration so the search will focus in a region of potential improvement. If the solution has deteriorated over the last iteration, you can increment the tabu tenure by 1 to guide the search away from an apparently bad region, as illustrated in figure 6.7. The values of $L_{\min}$ and $L_{\max}$ can be randomly changed every specific number of iterations.

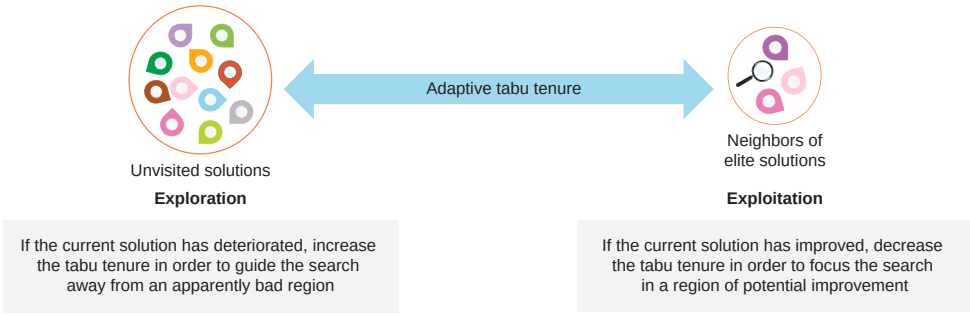


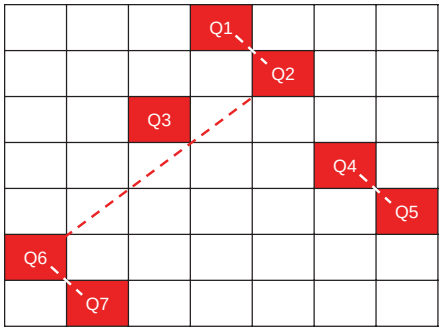
Figure 6.7 Dynamically controlling the tabu tenure

Reactive TS prevents the cycle occurrence by automatically learning the optimal tabu tenure [4]. In this approach, two possible reaction mechanisms are considered. An *immediate reaction mechanism* increases the tabu tenure to discourage additional repetitions. After a number of R immediate reactions, the geometric increase is sufficient to break any limit cycle. A second mechanism, called an *escape mechanism*, counts the number of moves that are repeated many times (more than REP times). When this number is greater than a predefined threshold REP, a diversifying escape movement is enforced. Other algorithm parameters, such as applying frequency-based memory or aspiration criterion, can be also considered when creating an adaptive version of tabu search.

Now that you have a good understanding of the various components of tabu search, let's explore how this algorithm can be used to solve a variety of optimization problems.

6.3 Solving constraint satisfaction problems

The n-queens problem is a classic puzzle that involves placing n chess queens on an $n \times n$ chessboard in such a way that no two queens threaten each other. In other words, no two queens should share the same row, column, or diagonal. This is a constraint-satisfaction problem (CSP) that does not define an explicit objective function. Let's suppose we are attempting to solve a 7-queens problem using tabu search. In this problem, the number of collisions in the initial random configuration shown in figure 6.8a is 4: {Q1 – Q2}, {Q2 – Q6}, {Q4 – Q5}, and {Q6 – Q7}.



a) 7-queens problem

Q	1	2	3	4	5	6	7
C	4	5	3	6	7	1	2

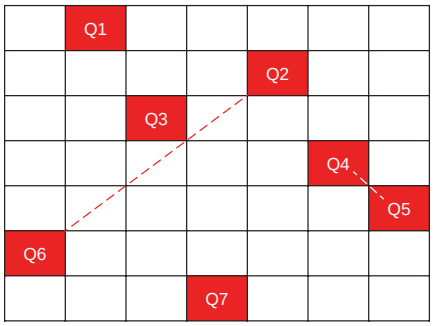
b) Solution representation

Queen swap	Δ value	
1,7	-2	*
2,3	-2	
2,6	-2	
5,6	-2	
1,5	-1	

c) Neighboring solutions

Figure 6.8 TS initialization for a 7-queens problem. At the left, the dotted lines show the 4 collisions between the queens. In the middle, C represents the column where a queen Q is placed. At the right, * denotes the swap that gives the best neighboring solution.

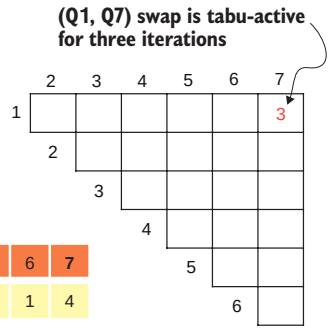
The initial solution in figure 6.8a can be represented as the ordering shown in figure 6.8b. A number of candidate neighboring solutions can be generated by swapping as shown in figure 6.8c. Swaps (Q1,Q7), (Q2,Q3), (Q2,Q6), and (Q5,Q6) give the same value, so let's assume that (Q1,Q7) is arbitrarily selected as a move that gives a new solution, which is shown in figure 6.9. In the initial iteration, Q1 was placed in column 4, and Q7 was placed in column 2. Swapping Q1 and Q7 means placing Q1 in column 2 and Q7 in column 4.



a) New solution after swapping Q1 and Q7

Q	1	2	3	4	5	6	7
C	2	5	3	6	7	1	4

b) Solution representation

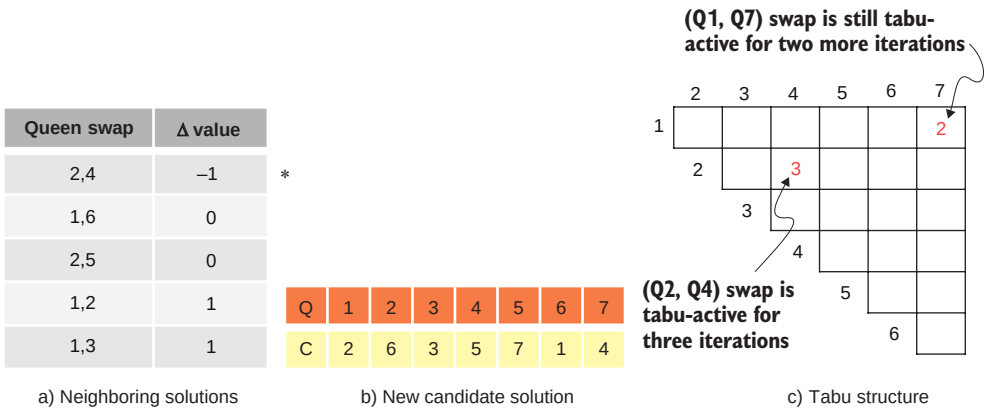


c) Tabu structure

Figure 6.9 A 7-queens problem—TS iteration 1

The number of collisions is now reduced to 2, which are {Q2-Q6} and {Q4-Q5}. The tabu structure is updated as shown in figure 6.9c, forbidding the recently performed swap (Q1,Q7) for three iterations, assuming that the tabu tenure is 3.

In the next iteration, other neighboring solutions can be generated by swapping Q2 and Q4, as illustrated in figure 6.10. Swap (Q2,Q4) gives a new candidate solution, as it reduces the collisions by 1. The associated number of collisions for this solution is 1. The tabu structure is updated, and the search continues.

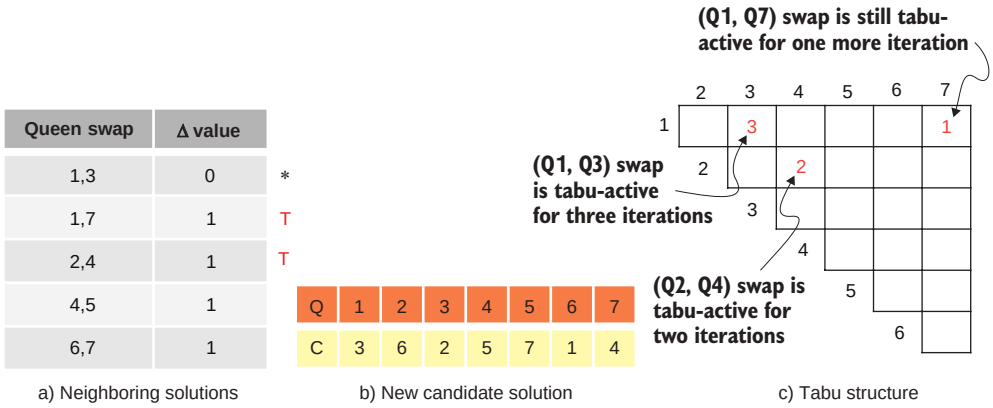


Queen swap	Δ value	
2,4	–1	*
1,6	0	
2,5	0	
1,2	1	
1,3	1	

Q	1	2	3	4	5	6	7
C	2	6	3	5	7	1	4

Figure 6.10 A 7-queens problem—TS iteration 2

In the next iteration (figure 6.11), swap (Q1,Q3) is selected as a move that gives a new solution.



Queen swap	Δ value	
1,3	0	*
1,7	1	T
2,4	1	T
4,5	1	
6,7	1	

Q	1	2	3	4	5	6	7
C	3	6	2	5	7	1	4

Figure 6.11 A 7-queens problem—TS iteration 3

In the new iteration (figure 6.12), swap (Q5,Q7) is selected. The *T* in figure 6.12a denotes the tabu-active moves.

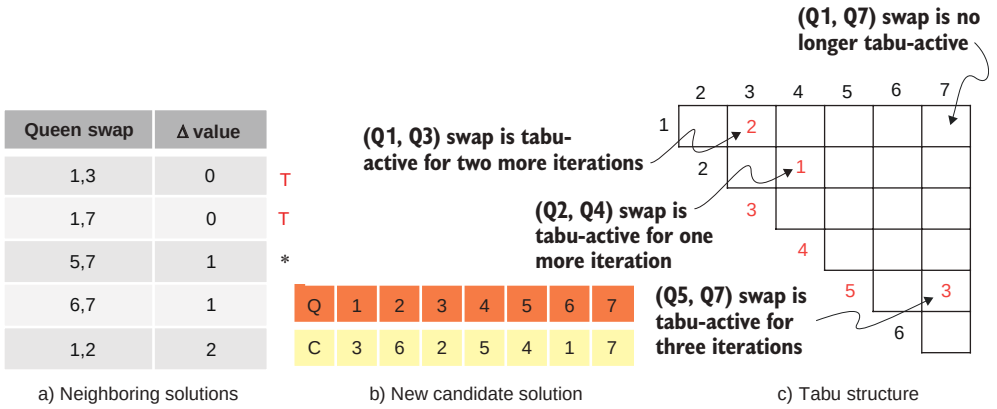


Figure 6.12 A 7-queens problem—TS iteration 4

In the next iteration, the (Q4,Q7) swap is selected (figure 6.13).

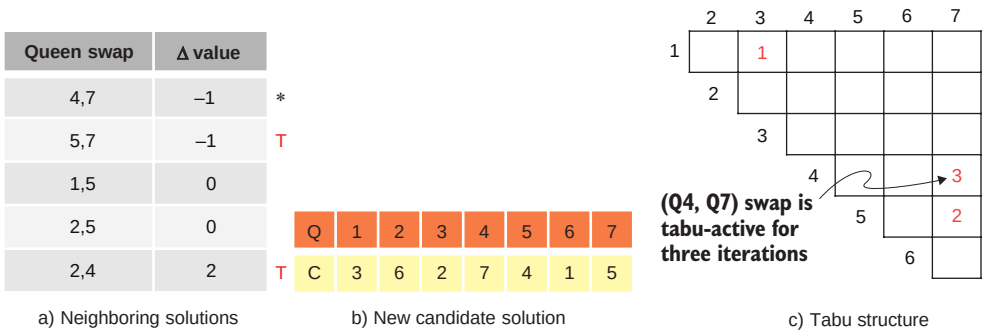


Figure 6.13 A 7-queens problem—TS iteration 5

In the next iteration, as the improving swaps are tabu-active, we can apply aspiration criteria to select swap (Q1,Q3) because there is only one iteration left before this swap is out of the tabu list (figure 6.14).

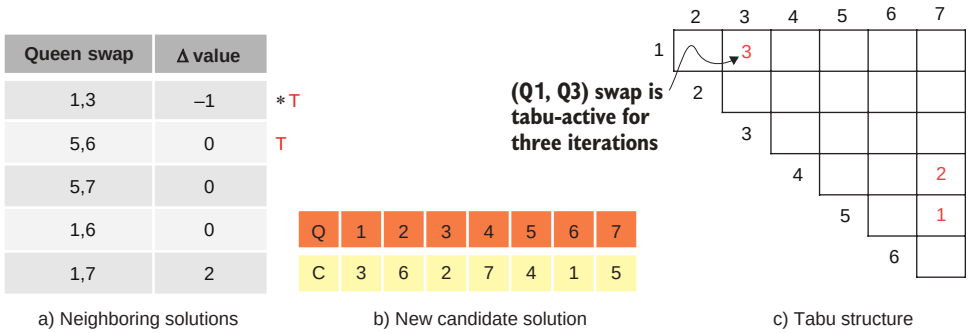


Figure 6.14 A 7-queens problem—TS iteration 6

Based on this solution, the board configuration will be as shown in figure 6.15. This is one of various possible solutions.

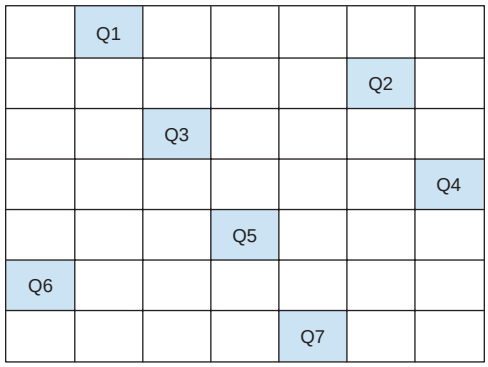


Figure 6.15 A 7-queens solution generated by hand-iteration

Let's explore how we can use Python to solve this problem using tabu search. To begin, we'll import the following Python libraries for random number generation and multi-dimensional arrays and plotting. Then we'll define a function to generate a random configuration for a *n*-queens board, based on a predefined board size.

Listing 6.1 Solving the 7-queens problem

```
import random
import numpy as np
import matplotlib.pyplot as plt
def get_initial_state(board_size):
    queens = list(range(board_size))
    random.shuffle(queens)
    return queens
```

Assuming that the board size is 7, calling this function returns a random board configuration such as [0, 4, 1, 5, 6, 2, 3]. This means that Q1, Q2, Q3, Q4, Q5, Q6, and Q7 are placed in columns 1, 5, 2, 6, 7, 3, and 4 respectively.

We can then define a function to compute the number of queens that are attacking each other on the board. This function is defined as follows:

```
def num_attacking_queens(queens):
    board_size = len(queens)
    num_attacks = 0
    for i in range(board_size):
        for j in range(i + 1, board_size):
            if queens[i]==queens[j] or abs(queens[i] - queens[j]) == j - i:
                num_attacks += 1
    return num_attacks
```

Next, we can create a function to determine the best possible move that decreases the number of attacks on the board, while ensuring that the move is not currently on the tabu list (i.e., not tabu-active). This function is defined as follows:

```
def get_best_move(queens, tabu_list):
    board_size = len(queens)
    best_move = None
    best_num_attacks = board_size * (board_size - 1) // 2
    for i in range(board_size):
        for j in range(board_size):
            if queens[i] != j:
                new_queens = queens.copy()
                new_queens[i] = j
                if str(new_queens) not in tabu_list:
                    num_attacks = num_attacking_queens(new_queens)
                    if num_attacks < best_num_attacks:
                        best_move = (i, j)
                        best_num_attacks = num_attacks
    return best_move
```

As you may have noticed, the best number of attacks is initialized as the maximum number of attacks, which is $n * (n - 1) / 2$. In a 7-queens problem, this number is 7 * 6 / 2 = 21.

We also need to implement a function that updates the tabu list based on a predefined tabu tenure. Here is the definition of this function:

```
def update_tabu_list(tabu_list, tabu_tenure, move):
    tabu_list.append(str(move))
    if len(tabu_list) > tabu_tenure:
        tabu_list.pop(0)
```

The following function executes the steps of the tabu search, taking input parameters such as the maximum number of iterations, the tabu tenure, and the maximum number of moves without improvement before concluding that the solution is stuck, and the initial solution:

```
def tabu_search(num_iterations, tabu_tenure, max_non_improvement, queens):
    num_non_improvement = 0
    best_queens = queens
    best_num_attacks = num_attacking_queens(queens)
    tabu_list = []

    for i in range(num_iterations):
        move = get_best_move(queens, tabu_list)
        if move is not None:
            queens[move[0]] = move[1]
            update_tabu_list(tabu_list, tabu_tennure, move)
            num_attacks = num_attacking_queens(queens)
            if num_attacks < best_num_attacks:
                best_queens = queens
                best_num_attacks = num_attacks
                num_non_improvement = 0
        else:
```

```
            num_non_improvement += 1
            if num_non_improvement >= max_non_improvement:
                break

    return best_queens, num_attacks
```

For a board size of 7, the maximum number of iterations is 2,000, the tabu tenure is 10, and the maximum number of moves without improvement before considering the solution to be stuck is 50. Calling the tabu search gives the solution [5, 1, 4, 0, 3, 6, 2], which is shown in figure 6.16.

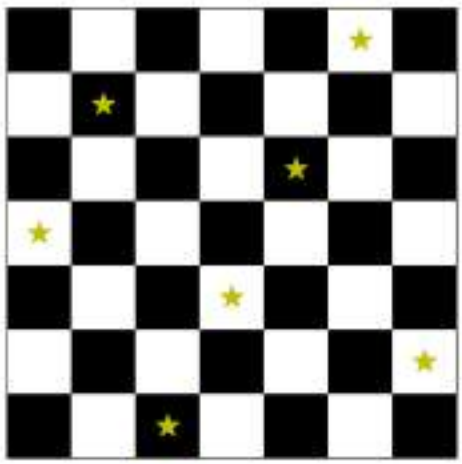


Figure 6.16 A 7-queens solution generated by Python code

The full code for this implementation can be found in listing 6.1 in the book's GitHub repository. The number of iterations is used in the code as a stopping criterion. As an exercise, you can modify the code to add a stopping criterion that terminates the search once a solution with zero attacks has been found.

The *n*-queens problem is a discrete problem, as it involves finding a feasible configuration of chess queens on a discrete chessboard. In the following section, we'll explore how tabu search can be applied to continuous problems in the form of function optimization.

6.4 Solving continuous problems

As an illustration of continuous problems, let's begin with function optimization. The Himmelblau function ($f(x,y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$), named after David Mautner Himmelblau (1924–2011), is a multimodal function that is often used as a test problem for optimization algorithms. It is a nonconvex function with four identical local minima at (3.0, 2.0), (–2.805118, 3.131312), (–3.779310, –3.283186), and (3.584428, –1.848126), as shown in figure 6.17.

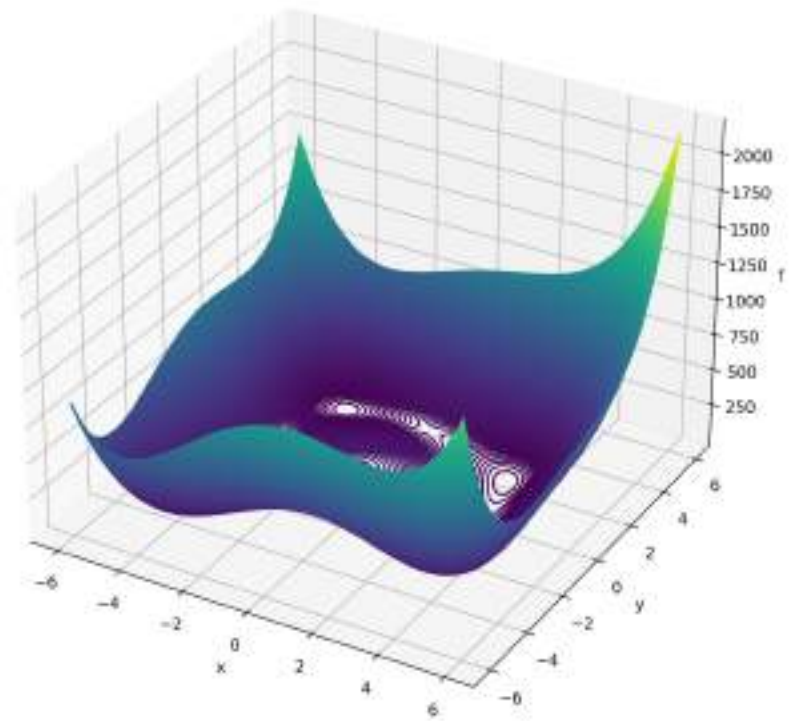


Figure 6.17 Himmelblau's function has four identical local minima at (3.0, 2.0), (–2.805118, 3.131312), (–3.779310, –3.283186), and (3.584428, –1.848126).

A generic Python implementation of tabu search is available as part of our optalgotools package. In this implementation, a hash table or dictionary as an indexed data structure is used to implement the tabu structure. A hashmap is a set of key–value pairs with no duplicate keys. It can be used to quickly retrieve data no matter how much data there is, as it has a big $O(1)$ for add, get, and delete functions.

The generic TS solver takes the following arguments:

- Maximum number of iterations (default `max_iter=1000`)
- Tabu tenure (default `tabu_tenure=1000`)
- Neighborhood size (default `neighbor_size=10`)
- Aspiration criteria (default `use_aspiration=True`)
- Remaining number of iterations to get out of tabu (default `aspiration_limit=None`)
- Incorporating frequency-based memory (default `use_longterm=False`)

The next listing shows how we can solve the minimization problem of Himmelblau's function using the generic tabu search solver implemented in optalgotools.

Listing 6.2 Solving Himmelblau's function using tabu search

```
import numpy as np
from optalgotools.algorithms import TabuSearch
from optalgotools.problems import ProblemBase, ContinuousFunctionBase

def Himmelblau(x,y):
       return (((x**2+y-11)**2) + (((x+y**2-7)**2)))

Himmelblau_bounds = np.asarray([[-6, 6], [-6, 6]])

Himmelblau_obj = ContinuousFunctionBase(Himmelblau, Himmelblau_bounds)

ts = TabuSearch(max_iter=100, tabu_tenure=5, neighbor_size=50, use_
aspiration=True,
➥ aspiration_limit=2, use_longterm=False, debug=1)
ts.run(Himmelblau_obj)
```

Import the generic tabu search solver from optalgotools.

Import the continuous problem base.

Define the objective function.

Define the bounds.

Create a continuous function object.

Define the TS solver. Add debug = 1 to print the initial and final solution.

Run the solver.

Running this code gives a potential solution for Himmelblau's function:

```
Tabu search is initialized:
current value = 148.322
Tabu search is done:
curr iter: 100, curr best value: 0.005569730862620958, curr best: sol:
[3.00736837 1.98045825], found at iter: 21
```

Proper tuning of the various algorithm parameters allows you to find an optimal or near-optimal solution. Several other optimization test functions are available in appendix B. You may consider trying different functions by modifying listing 6.2.

Next, let's examine how tabu search can address the traveling salesman problem.

6.5 Solving TSP and routing problems

Let's look at using tabu search, implemented in Google OR-Tools, to solve the Berlin52 instance of TSP. This dataset contains 52 locations in the city of Berlin (http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/STSP.html). The objective of the problem is to find the shortest possible tour that visits each location exactly once and then return to the starting location. The shortest route obtained for the Berlin52 dataset is 7,542, as explained in the previous chapter.

We'll start by importing the TSP problem class, the OR-Tools constraint programming solver, and the protocol buffer module that defines various enumerations (`enums`) used in the routing library of OR-Tools. We'll then create a `tsp` object from our generic `tsp` class implemented in `optalgotools`. We'll extract points of interest, nodes, or cities and calculate pairwise distances. The pairwise distances will be converted into integers as required by OR-Tools. Then we'll store the problem data in the form of a dictionary. In this dictionary, `distance_matrix` will represent the pairwise distances between the points of interest in the dataset.

Listing 6.3 Solving Belin52 TSP using OR-Tools tabu search

```
import numpy as np
from optalgotools.problems import TSP
from ortools.constraint_solver import pywrapcp
from ortools.constraint_solver import routing_enums_pb2
import matplotlib.pyplot as plt

berlin52_tsp_url = 'https://raw.githubusercontent.com/coin-or/jorlib/
b3a41ce773e9b3b5b73c149d4c06097ea1511680/jorlib-core/src/test/resources/
tspLib/tsp/berlin52.tsp'
berlin52_tsp = TSP(load_tsp_url=berlin52_tsp_url, gen_method='mutate',
➥ init_method='random')

cities = berlin52_tsp.cities
tsp_dist=berlin52_tsp.eval_distances_from_cities(cities)
tsp_dist_int=list(np.array(tsp_dist).astype(int))
```

Import the TSP problem class from optalgotools.

Import the protocol buffer module.

Import the Python wrapper for the C++ constraint programming solver in OR-Tools.

Get berlin52 from a permalink.

Create a different tsp object from the problem class.

Define the problem parameters.

We need to create a routing model by defining `data`, an index manager (`manager`), and a routing model (`routing`). The pairwise distances between any two nodes will be returned by the `distance_callback` function, which also converts from the routing variable `Index` to the distance matrix `NodeIndex`. The cost of the edge joining any two points of interest in the dataset is computed using an arc cost evaluator that tells the solver how to calculate the cost of travel between any two locations.

The data model is where the distance matrix, number of vehicles, and home city or initial depot are defined:

```
def create_data_model():
    data = {}
    data['distance_matrix'] = tsp_dist_int
    data['num_vehicles'] = 1
    data['depot'] = 0
    return data
```

The following function returns the pair-wise distance between any two nodes:

```
def distance_callback(from_index, to_index):
    from_node = manager.IndexToNode(from_index)
    to_node = manager.IndexToNode(to_index)
    return data['distance_matrix'][from_node][to_node]
```

The obtained route and its cost, or length, can be printed using the following function:

```
def print_solution(manager, routing, solution):
    print('Objective: {} meters'.format(solution.ObjectiveValue()))
    index = routing.Start(0)
    plan_output = 'Route for vehicle 0:\n'
    route_distance = 0
```

```
    while not routing.IsEnd(index):
        plan_output += ' {} ->'.format(manager.IndexToNode(index))
        previous_index = index
        index = solution.Value(routing.NextVar(index))
        route_distance += routing.GetArcCostForVehicle(previous_index, index, 0)
    plan_output += ' {}\n'.format(manager.IndexToNode(index))
    plan_output += 'Route distance: {}meters\n'.format(route_distance)
```

Let's now look at how we can actually solve the TSP using the tabu search implemented in OR-Tools. We'll start by using the preceding functions to create a data model, which generates the data necessary for the TSP, such as the distance matrix between the cities, the number of vehicles, and the home city or the initial depot.

Next, we'll define a manager to manage the indices of the routing problem. For this, we'll use the `RoutingIndexManager` class from the `pywrapcp` module in the OR-Tools. This module provides a Python wrapper for the CP-SAT solver, a constraint programming solver developed by Google.

Then we'll create the `RoutingModel` object by using the `RoutingIndexManager` object. This `RoutingModel` object is used to define the constraints and objectives of the capacitated vehicle routing problem (CVRP), which is considered a generalization of TSP. The `RegisterTransitCallback()` method will register a callback function that calculates the distance between two cities. This callback function is defined in the `distance_callback` function.

The `SetArcCostEvaluatorOfAllVehicles()` method will set the arc cost evaluator of all the vehicles to the transit callback index, which calculates the distance between two nodes. In our case, we have a single traveling salesman or a single vehicle (so `num_vehicles=1`), but this code can also handle multiple TSP (mTSP) or multiple vehicles as well.

The `DefaultRoutingSearchParameters()` method will create an object of the `RoutingSearchParameters` class, which specifies the search parameters for solving the routing problem. In this case, the local search metaheuristic is set to tabu search, and the time limit is set to 30 seconds. Other available methods include `GREEDY_DESCENT`, `SIMULATED_ANNEALING`, and `GENERIC_TABU_SEARCH`. The main difference between `TABU_SEARCH` and `GENERIC_TABU_SEARCH` is in the way they handle the tabu list. `TABU_SEARCH` maintains a tabu list for each variable and applies tabu constraints to the current assignment. `GENERIC_TABU_SEARCH`, on the other hand, maintains a single tabu list for the entire search and applies tabu constraints to the moves made by the search.

The `SolveWithParameters()` method solves the routing problem with the specified search parameters. If a solution is found, it calls the `print_solution()` function to print the solution:

Create the model.

```
data = create_data_model()
manager = pywrapcp.RoutingIndexManager(len(data['distance_matrix']),
➥                                      data['num_vehicles'], data['depot'])
```

```
routing = pywrapcp.RoutingModel(manager)

transit_callback_index = routing.RegisterTransitCallback(distance_callback)

routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)

search_parameters = pywrapcp.DefaultRoutingSearchParameters()
search_parameters.local_search_metaheuristic = (
➥     routing_enums_pb2.LocalSearchMetaheuristic.TABU_SEARCH)
search_parameters.time_limit.seconds = 30
search_parameters.log_search = True

solution = routing.SolveWithParameters(search_parameters)
if solution:
    print_solution(manager, routing, solution)
```

Set TABU_SEARCH as the solver.

Find the solution.

The following `get_routes()` function can then be called to extract the routes for each vehicle from the solution. This function iterates through each vehicle, starting with the start node, and adds the nodes visited by the vehicle until it reaches the end node. It then returns a list of routes for each vehicle:

```
def get_routes(solution, routing, manager):
  routes = []
  for route_nbr in range(routing.vehicles()):
    index = routing.Start(route_nbr)
    route = [manager.IndexToNode(index)]
    while not routing.IsEnd(index):
      index = solution.Value(routing.NextVar(index))
      route.append(manager.IndexToNode(index))
    routes.append(route)
  return routes

routes = get_routes(solution, routing, manager)

for i, route in enumerate(routes):
  print('Route', i, route)
berlin52_tsp.plot(route)
```

Print the route.

Visualize the route.

Running this code produces the following results and the route shown in figure 6.18:

```
Objective: 7884 meters
Route for vehicle 0:
 0 -> 21 -> 31 -> 44 -> 18 -> 40 -> 7 -> 8 -> 9 -> 42 -> 32 -> 50 -> 11 -> 10
-> 51 -> 13 -> 12 -> 26 -> 27 -> 25 -> 46 -> 28 -> 29 -> 1 -> 6 -> 41 -> 20
-> 16 -> 2 -> 17 -> 30 -> 22 -> 19 -> 49 -> 15 -> 43 -> 45 -> 24 -> 3 -> 5 ->
14 -> 4 -> 23 -> 47 -> 37 -> 36 -> 39 -> 38 -> 33 -> 34 -> 35 -> 48 -> 0
```

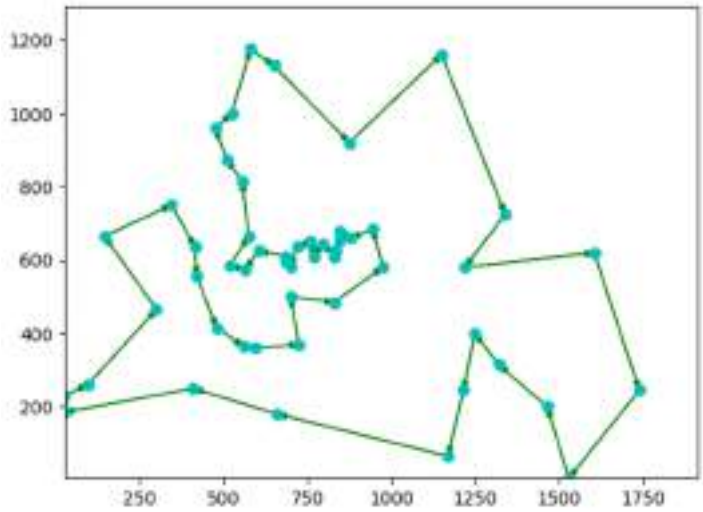


Figure 6.18 A TSP solution using the tabu search in OR-Tools. The graph shows the x and y locations of the points of interest included in the dataset in km.

The preceding implementation applies a simple aspiration criterion, where a solution is accepted if it is better than any other solution encountered so far. OR-Tools is very efficient in solving this problem (the obtained route length is 7,884, while the optimal solution is 7,542). However, the implemented tabu search is mainly used to solve routing problems.

As a continuation of listing 6.3, the following code snippet shows a generic tabu search solver in optalgotools that can be used to solve the same problem:

Create a TSP object for the problem.

Create a TS object to help in solving the TSP problem.

```
from optalgotools.algorithms import TabuSearch

ts = TabuSearch(max_iter=100, tabu_tenure=5, neighbor_size=10000,
➥ use_aspiration=True, aspiration_limit=2, use_longterm=False, debug=1)

ts.init_ts(berlin52_tsp,'random')
ts.val_cur

ts.run(berlin52_tsp, repetition=1)

print(ts.s_best)
print(ts.val_best)

berlin52_tsp.plot(ts.s_best)
```

Get an initial random solution, and check its length.

Run TS, and evaluate the best solution distance.

Print the best route.

Print the route length.

Visualize the best route.

Running this code produces the following results:

```
sol: [0, 21, 17, 2, 16, 20, 41, 6, 1, 29, 28, 15, 45, 47, 23, 36, 33, 43, 49,
19, 22, 30, 44, 18, 40, 7, 8, 9, 42, 32, 50, 10, 51, 13, 12, 46, 25, 26, 27,
11, 24, 3, 5, 14, 4, 37, 39, 38, 35, 34, 48, 31, 0], found at iter: 51
7982.79
```

As you can see, the obtained route length using our tabu search solver is 7,982.79, while the tabu search implemented in OR-Tools provides 7,884, and the optimal solution is 7,542. The tabu search algorithm implemented in optalgotools is also slower than the optimized tabu search implemented in Google's OR-Tools.

Let's revisit the delivery semi-truck routing problem discussed in section 5.6. In this problem, we need to find the optimal route for a delivery semi-truck to visit 18 Walmart Supercenters in a selected part of the Greater Toronto Area (GTA) starting from Walmart Supercenter number 3001, located at 270 Kingston Rd. E in Ajax, Ontario. The next listing shows how we can use the generic tabu search solver to handle this problem. A complete listing is available in the book's GitHub repo.

Listing 6.4 Solving the delivery semi-truck problem using tabu search

```
from optalgotools.algorithms import TabuSearch
from optalgotools.problems import TSP

gta_part_tsp = TSP(dists=gta_part_dists, gen_method='mutate')

ts = TabuSearch(max_iter=1000, tabu_tenure=5, neighbor_size=100,
➥ use_aspiration=True, aspiration_limit=2, use_longterm=False, debug=1)

ts.init_ts(gta_part_tsp,'random')

draw_map_path(G, ts.s_cur, gta_part_loc, gta_part_pathes)

ts.run(gta_part_tsp, repetition=5)

print(ts.s_allbest)
print(ts.val_allbest)

draw_map_path(G, ts.s_allbest, gta_part_loc, gta_part_pathes)
```

Get an initial random solution, and check its length.

Create a TS object to help solve the TSP problem.

Create a TSP object for the problem.

Draw the path of the random initial solution.

Run tabu search five times, and return the best solution.

Print the best solution.

Print the best route length.

Visualize the obtained route.

The generated route for the delivery semi-truck problem is shown in figure 6.19.

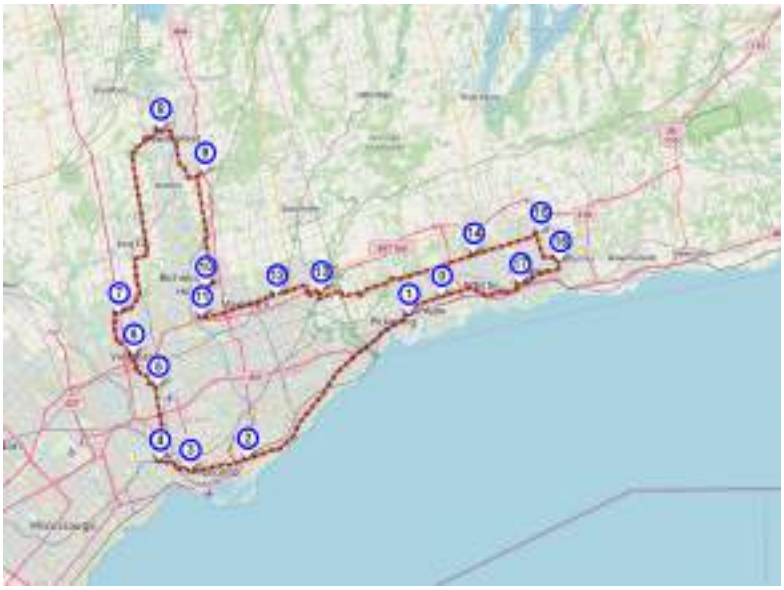


Figure 6.19 The TS solution for the Walmart delivery semi-truck route with a total distance of 223.53 km

Tabu search generates a slightly shorter route (223.53 km) than simulated annealing (227.17 km). Compared to the tabu search algorithm implemented in OR-Tools, the tabu search algorithm in optalgotools gives you more freedom to tune more parameters and to handle different types of discrete and continuous problems.

In the next section, we will delve into another notable challenge that the manufacturing sector faces.

6.6 Assembly line balancing problem

Henry Ford designed and installed an assembly line for car mass production in 1913. This development of assembly line manufacturing enabled mass production during the second industrial revolution and beyond. An *assembly line* is a flow-oriented production system where the productive units performing the operations, referred to as *workstations* or simply *stations*, are aligned sequentially. The work pieces visit the stations successively as they are moved along the line, usually by some kind of transportation system, such as a conveyor belt. At each workstation, new parts are added or new assemblies take place, resulting in a finished product at the end.

For example, figure 6.20 shows an example of a bike assembly line with five workstations. Beginning at the initial workstation WS-1, workers focus on assembling the frame, laying the foundation for subsequent tasks. Moving along the line, WS-2 takes charge of installing the forks and handlebars, while WS-3 attaches the wheels. Following this, at WS-4, workers undertake the intricate assembly of crankset, chain, derailleurs, gears, and pedals. Finally, at WS-5, the seat is securely affixed and other accessories are added, completing the assembly process. Three lamps are used to indicate the status of operation of each workstation: emergency, finish, and work in progress (WIP).

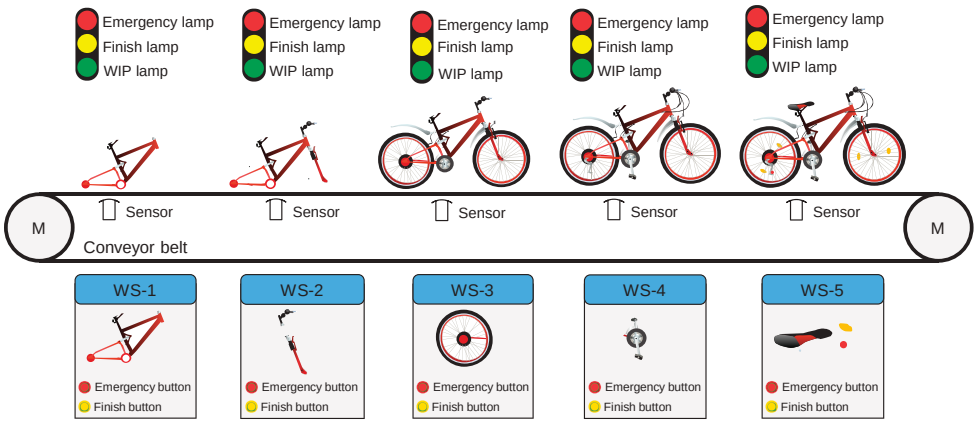


Figure 6.20 Assembly line balancing problem

It is crucial to optimize the design of an assembly line before implementing it, as assembly lines are designed to ensure high production efficiency, and reconfiguring them can result in significant investment costs. The assembly line balancing problem (ALBP) addresses the assignment of tasks (work elements) to workstations in order to minimize the amount of idle time of the line, while satisfying specific constraints. ALBP generally comprises all tasks and decisions related to equipping and aligning the productive units for a given production process before the actual assembly process can start. This encompasses setting the system capacity, which includes cycle time, number of stations, and station equipment, as well as assigning work content to productive units, which includes task assignment and determining the sequence of operations. This *balancing* of assembly lines is a difficult combinatorial optimization problem arising frequently in manufacturing.

Assembly line balancing problems can be categorized into two main groups: simple assembly line balancing problems (SALBPs) and generalized assembly line balancing problems (GALBPs). An SALBP involves the production of a single product in a serial line on one-sided workstations, while a GALBP considers different assembly line objectives, such as mixed model assembly lines, parallel lines, U-shaped lines, and two-sided lines.

In SALBP, we have a number of tasks that need to be completed by a number of workstations. Each task i has a time requirement t_i, and we are given a maximum number of workstations. Each workstation has a cycle time C, which refers to the time allocated for each station in the assembly line to complete its assigned tasks and pass the product to the next station. The goal is to minimize the number of workstations needed.

To capture more realistic conditions for ALBPs in industry, the time and space assembly line balancing problem (TSALBP) incorporates additional space constraints. A TSALBP involves assigning a set of n tasks with temporal and spatial attributes and a precedence graph. Each task must be assigned to only one station, provided that

- All precedence constraints are met
- The workload time for each station does not exceed the cycle time
- The required space for each station does not exceed the global available space

Different variations on TSALBP with different levels of complexity are shown in table 6.1.

Table 6.1 TSALBP variations: F (feasibility problem), OP (mono-objective optimization problem), MOP (multi-objective optimization problem)

Problem	# of stations	Cycle time	Space or layout of the stations	Type
TSALBP-F	Given	Given	Given	F
TSALBP-1	Minimize	Given	Given	OP
TSALBP-2	Given	Minimize	Given	OP
TSALBP-3	Given	Given	Minimize	OP
TSALBP-1/2	Minimize	Minimize	Given	MOP
TSALBP-1/3	Minimize	Given	Minimize	MOP
TSALBP-2/3	Given	Minimize	Minimize	MOP
TSALBP-1/2/3	Minimize	Minimize	Minimize	MOP

In the bike assembly line illustrated in figure 6.20, installing the forks and handlebar depends on the availability of an assembled frame. Similarly, attaching the wheels depends on the frame and forks assembly being completed. This dependency is defined by a precedence diagram, which shows the relationships between tasks, indicating which tasks must be completed before others can begin. For example, task 2 should be performed before starting tasks 3 and 4, as per the precedence diagram depicted in figure 6.21. In ALBPs, the sequence of the tasks should not violate the specified precedence due to the dependency relations between them.

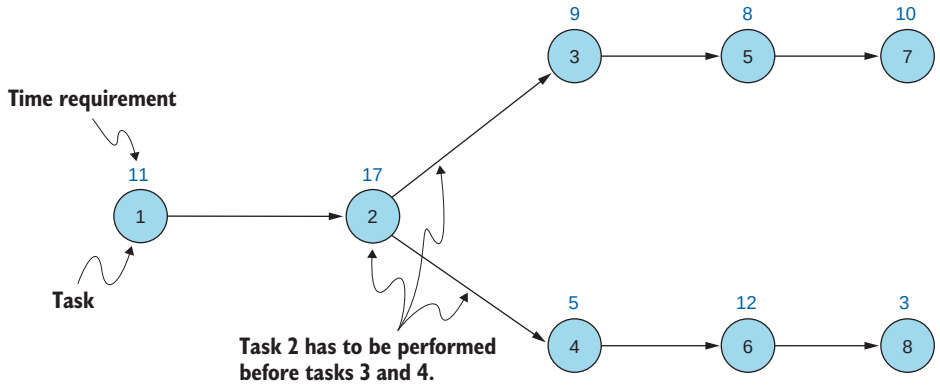


Figure 6.21 A precedence diagram

Simple assembly line balancing problems can be classified into two types: type 1 (SALBP-1) and type 2 (SALBP-2). Under type 1 (SALBP-1), the objective is to minimize the number of stations for a given cycle time. Conversely, under type 2 (SALBP-2), the goal is to minimize the cycle time for a given number of stations. Let's consider a type 1 (SALBP-1) problem that consists of minimizing the number of stations *NS* given fixed values of the cycle time *CT* and of the available area per station *A*. TSALBP-1 is equivalent to SALBP-1 if $A \rightarrow \infty$. We'll use smoothing index (SI) as a quantitative measure to evaluate the uniformity of workload distribution among the workstations. Each neighboring solution will be quantitatively evaluated using this SI. The SI aims to get the optimal task assignment for each station to minimize the idle time between stations, taking into account that the constraints imposed on the station's workload cannot exceed the cycle time.

SI is calculated as in equation 6.1:

$$SI = \sqrt{\frac{\sum_{i=1}^{NS} (WL_{max} - WL_i)^2}{NS}} \qquad \textbf{6.1}$$

where

- WL_i is the workload of workstation i
- WL_{max} is the maximum workload
- *NS* is the number of stations

Tasks are assigned to the stations such that the workload doesn't exceed the cycle time and without violating their precedence. Assume that the cycle time *CT* is 4 minutes and the number of tasks *n* is 6, with the precedence diagram given in figure 6.22.

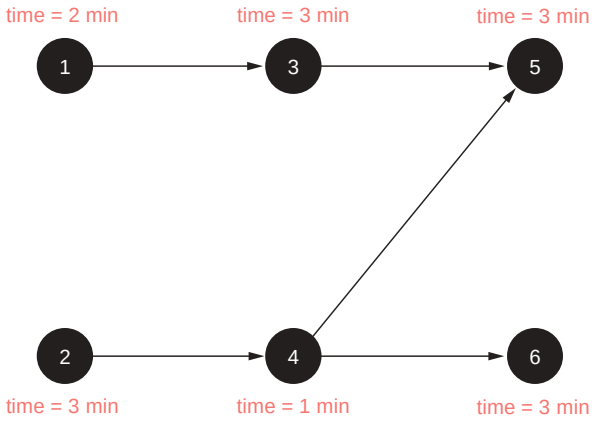


Figure 6.22 Precedence diagram example for six tasks

Let's perform hand iterations to understand how TS can be used to solve this problem, considering a tabu tenure of 3. A random initial solution is generated, as shown in figure 6.23, and its SI is evaluated using equation 6.1. The tabu structure or neighborhood can be defined as any other solution that is obtained by a pair-wise exchange of any two tasks in the solution. In our case, we have six tasks (i.e., $n = 6$) and a pairwise exchange (i.e., $k = 2$). So the maximum number of neighbors is the number of combinations without repetition $C(n,k)$, or n-choose-k, or $n! \,/\, k!(n-k)! = 6! \,/\, 2!4! = 15$ neighbors. The solution is presented as a permutation of tasks. For example, the initial solution [1 2 3 4 5 6] reflects the order of execution of the six tasks, taking into consideration the precedence constraint.

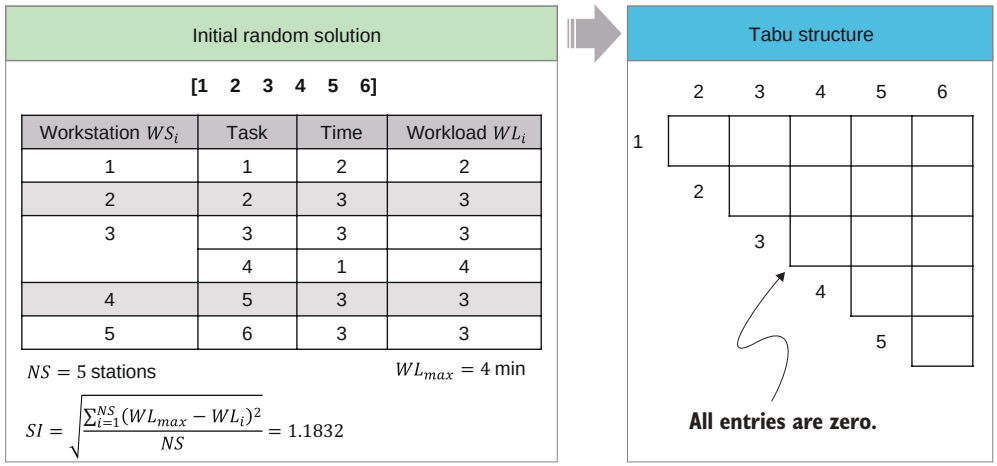


Figure 6.23 TS initialization for SALBP

Figure 6.24 shows the first iteration of TS for solving the SALBP. To generate a neighboring solution, we have to check the precedence diagram (figure 6.22). For example, for task 5 to start, both predecessor tasks 3 and 4 must have finished. Following this precedence diagram, when task 4 finishes, then both tasks 5 and 6 can start.

Let's use the swap method to find a neighboring solution. For this iteration, the neighboring feasible solutions are (1-2), (2-3), (3-4), and (5-6). As the three swaps lead to the same SI, we can arbitrarily pick one, such as (1-2), that results in a new order of task execution (i.e., a new candidate solution). This solution is [2 1 3 4 5 6]. The (1-2) swap should be added to the tabu structure for three iterations, assuming that the tabu tenure is 3.

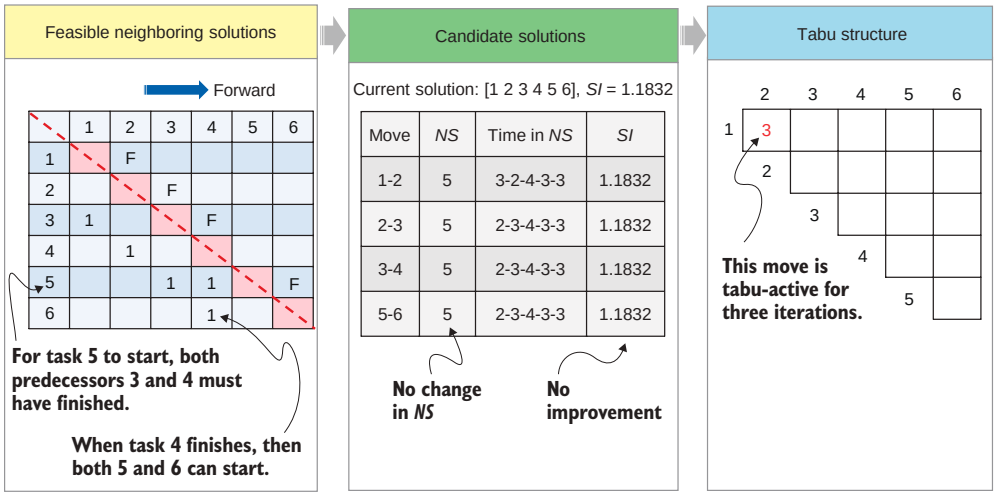


Figure 6.24 TS iteration 1 for SALBP

Moving forward, figure 6.25 shows the second iteration of tabu search. For this iteration, the neighboring feasible solutions are (2-1), (2-3), (3-4), and (5-6). Note that the move (2-1) is tabu-active. The (3-4) swap is selected because it has the smallest SI. The new solution is [2 1 4 3 5 6] with SI = 0, calculated with equation 6.1.

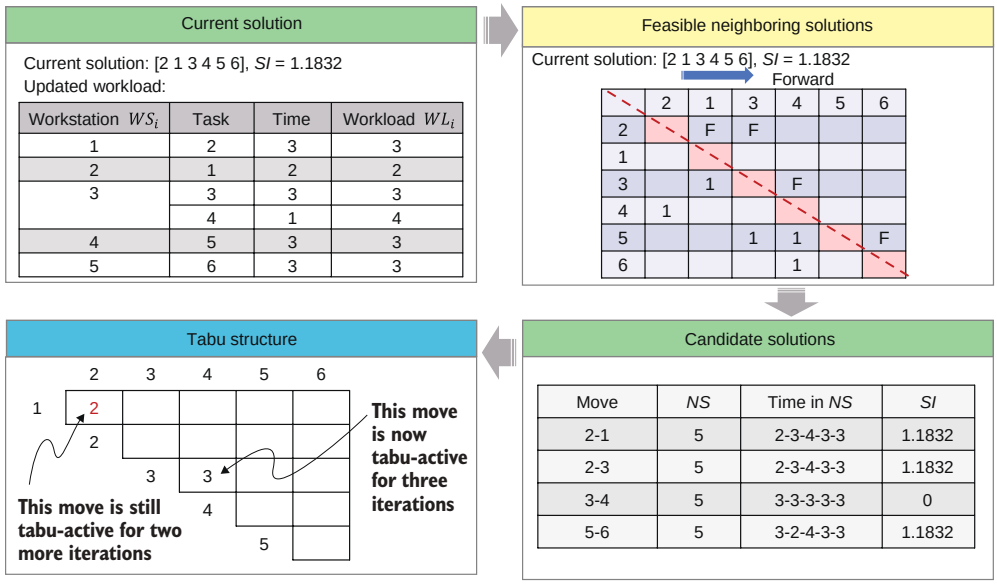


Figure 6.25 TS iteration 2 for SALBP

The tabu list is updated before we start the next iteration, as shown in the figure. The next listing shows a snippet of the tabu search implementation for solving SALBP. A complete listing is available in the book's GitHub repo.

Listing 6.5 Solving SALBP using tabu search

```
import pandas as pd
import numpy as np
import random as rd
import math
import matplotlib.pyplot as plt

tasks = pd.DataFrame(columns=['Task', 'Duration'])
tasks= pd.read_csv("https://raw.githubusercontent.com/Optimization-Algorithms
➥ -Book/Code-Listings/main/Appendix%20B/data/ALBP/ALB_TS_DATA.txt", sep
=",")
Prec= pd.read_csv("https://raw.githubusercontent.com/Optimization-Algorithms
➥ -Book/Code-Listings/main/Appendix%20B/data/ALBP/ALB_TS_PRECEDENCE.txt",
➥ sep =",")
Prec.columns=['TASK', 'IMMEDIATE_PRECEDESSOR']
```

Read data from appendix B directly.

```
Cycle_time = 4
```
Define the cycle time.

```
tenure = 3
max_itr=100
```
Get an initial solution.

Ensure the feasibility of the solution, considering the task precedence constraint.

```
solution = Initial_Solution(len(tasks))
soln_init = Make_Solution_Feasible(solution, Prec)

sol_best, SI_best=tabu_search(max_itr, soln_init, SI_init, tenure, WS, tasks,
➥ Prec_Matrix, Cycle_time)
```
Run the tabu search.

```
Smoothing_index(sol_best, WS, tasks, Cycle_time, True)
```
Calculate the SI of the best solution.

```
plt = Make_Solution_to_plot(sol_best, WS, tasks, Cycle_time)
plt.show()
```
Visualize the solution.

Running this code produces the following output:

```
The Smoothing Index value for ['T3', 'T5', 'T6', 'T1', 'T4', 'T2'] solution
sequence is: 0.0
The number of workstations for ['T3', 'T5', 'T6', 'T1', 'T4', 'T2'] solution
sequence is: 5
The workloads of workstation for ['T3', 'T5', 'T6', 'T1', 'T4', 'T2']
solution sequence are: [3. 3. 3. 3. 3.]
```

Figure 6.26 shows the initial and the final solution found by tabu search with a fair load balance between the workstations.

Figure 6.26 SALBP initial and final solutions

Let's now use the generic tabu search solver that's implemented as part of our optalgotools package. There are several benchmark datasets for ALBPs. These datasets are available in appendix B of the book's GitHub repo, and you can access them directly by using the URL to the raw content of the file, which can be obtained by using the "Raw" view in GitHub. Precedence graphs are provided in files with an .IN2 extension.

The next listing shows how to use the generic solver to solve the MANSOOR benchmark SALBP (best *NS* for a given *CT* = 48 is 4). The solution shows both the minimum number of workstations and the SI.

Listing 6.6 Assembly line balancing problem benchmarking

```
from optalgotools.algorithms import TabuSearch
from optalgotools.problems import ALBP

data_url="https://raw.githubusercontent.com/Optimization-Algorithms-Book/
Code-Listings/main/Appendix%20B/data/ALBP/SALBP-data-sets/precedence%20
graphs/"

albp_instance= ALBP(data_url, "MANSOOR.IN2", 48.0)

ts = TabuSearch(max_iter=20, tabu_tenure=4, neighbor_size=5, use_
aspiration=True,
➥ aspiration_limit=None, use_longterm=False)
ts.init_ts(albp_instance)
ts.run(albp_instance, repetition=5)

SI = albp_instance.Smoothing_index(list(ts.s_best), ts.val_best,
➥ albp_instance.tasks, True)
print(SI)
```

Import the tabu search solver from optalgotools.

Import the ALBP class from the generic problem class.

Define the URL of the datasets.

Create an ALBP instance.

Create an instance of the tabu search solver.

Solve the problem using tabu search.

Calculate the SI of the solution.

Print the results.

Running this code gives the following results:

```
The Smoothing Index value for ['T1', 'T2', 'T4', 'T5', 'T6', 'T7', 'T9',
'T8', 'T10', 'T3', 'T11'] solution sequence is: 12.296340919151518
The number of workstations for ['T1', 'T2', 'T4', 'T5', 'T6', 'T7', 'T9',
'T8', 'T10', 'T3', 'T11'] solution sequence is: 5
The workloads of workstation for ['T1', 'T2', 'T4', 'T5', 'T6', 'T7', 'T9',
'T8', 'T10', 'T3', 'T11'] solution sequence are: [42. 44. 20. 45. 34.]
```

The complete listing in the book's GitHub repo shows several different datasets, including the following:

- MITCHELL (best NS for a given CT = 26 is 5)
- SAWYER30 (best NS for a given CT = 26 is 10)
- HAHN (best NS for a given CT = 2338 is 7)
- GUNTHER (best NS for a given CT = 44 is 12)
- BUXEY (best NS for a given CT = 47 is 7)
- LUTZ2 (best NS for a given CT = 11 is 49)
- BARTHOL2 (best NS for a given CT = 104 is 41)
- JACKSON (best NS for a given CT = 9 is 6)
- TONGE70 (best NS for a given CT = 293 is 13)

That concludes the second part of this book. We'll now shift our focus to evolutionary computation algorithms like genetic algorithms. These algorithms feature inherent parallelism and the capability to adapt their search for optimal solutions dynamically.

Summary

- Local search iteratively explores a subset of the search space in the neighborhood of the current solution or state in order to improve the solution by making local changes.
- Tabu search extends local search by combining it with adaptive memory structures. It guides a local search procedure to explore the solution space beyond any local optimality.
- Adaptive memory structures are used to remember recent algorithm moves and capture promising solutions.
- A tabu list is a data structure that keeps track of tabu-active moves.
- Tabu tenure refers to the specified number of iterations for which certain moves or solutions are marked as tabu-active.
- A too-short tabu tenure can result in cycling and the neglect of promising solutions, while a too-long tabu tenure may lead to stagnation and memory overload.
- As a way to avoid search stagnation, aspiration criteria allow tabu-active moves to be accepted by relaxing or temporarily lifting the tabu condition.
- A crucial aspect of adaptive tabu search involves striking a balance between exploiting search and exploration.