Chapter 6
Simulated Annealing, FCM, Partition Coefficients and Tabu Search

6.1 Introduction

In 1945 the construction of the first computer caused a revolution in the world. It was aimed to modify the interactions between Russia and the West. In the academic and research fields it brought back a mathematical technique known as statistical sampling, now referred to as the Monte Carlo method. S. Frankel and N. Metropolis created a model of a thermonuclear reaction for the Electronic Numerical Integrator and Computer (ENIAC), persuaded by the curiosity and interest of John von Neumann, a prominent scientist in that field.

The results of the model where obtained after the end of the World War I, and among the reviewers was Stan Ulam, who had an extensive background in mathematics and the use of statistical methods. He knew these techniques were no longer in use because of the length and tediousness of calculations. His research interest included pattern development in 2D games played with very simple rules. These techniques are now used in various industrial applications known as cellular automata.

Ulam and Neumann sent a proposal of the Monte Carlo method to the theoretical division leader of the Los Alamos Laboratory in New Mexico in 1947, which included a detailed outline of a possible statistical approach to solve the problem of neutron diffusion in fissionable material. The basic idea of the method was to generate a genealogical history of different variables in a process until a statistically valid picture of each variable was created.

The next step was to generate random numbers; here Neumann proposed a method called the *middle-square digits*. Once the random numbers are generated, they must be transformed into a non-uniform distribution desired for the property of interest. Solving problems using this method is easier than other approaches like differential equations, because one needs only to mirror the probability distribution into the search space of the problem at hand.

In 1947 the ENIAC was moved to the Ballistic Research Laboratory in Maryland, its permanent home. After the movement, there was an explosion in the use of the

P. Ponce-Cruz, F. D. Ramirez-Figueroa, *Intelligent Control Systems with LabVIEW™*

Monte Carlo method with this computer. The applications solved several questions of different branches of physics, and by 1949 there was a special symposium held on the method.

The Monte Carlo method gave birth to modern computational optimization problems. We can now see, as a natural consequence of electronic computers, the quick evolution of experimental mathematics, with the Monte Carlo method key to this achievement. It was at this point that mathematics achieved the twofold aspect of experiment and theory, which all other sciences enjoy.

As an example of the method, we can imagine a coconut shy. We want to know the probability of taking 10 shots at the coconut shy and obtain an even number of hits. The only information that we know is that there is a 0.2 probability of having a hit with a single shot. Using the Monte Carlo method we can perform a large number of simulations of taking 10 shots at the coconut shy. Next we can count the simulations with even number of hits and divide that number over the total number of simulations. By doing this we will get an approximation of the probability that we are looking for.

6.1.1 Introduction to Simulated Annealing

A combinatorial optimization problem strives to find the best or optimal solution, among a finite or infinite number of solutions. A wide variety of combinatorial problems have emerged from different areas such as physical sciences, computer science, and engineering, among others. Considerable effort has been devoted to construct and research methods for solving the performance of the techniques. Integer, linear, and non-linear programming have been the major breakthroughs in recent times.

Over the years it has been shown that many theoretical and practical problems belong to the class of *NP-complete problems*. A large number of these problems are still unsolved; there are two main options for solving them. On the one hand, if we strive for optimality the computation time will be very large; these methods are called *optimization methods*. On the other hand, we can search quick solutions with suboptimal performance, called *heuristic algorithms*. However, the difference between these methods is not very strict, because some types of algorithm can be used for both purposes.

Another way to classify algorithms is between general and tailored. While *general algorithms* are applicable to a wide range of problems, *tailored algorithms* use problem-specific information, restricting their applicability. The intrinsic problem is that for the former ones, for each type of combinatorial optimization problem, a new algorithm must be constructed.

6.1.2 Pattern Recognition

Recognizing and classifying patterns is a fundamental characteristic of human intelligence. It plays a key role in human perception as well as other levels of cognition. The field of study has evolved since the 1950s. *Pattern recognition* can be defined as a process by which we search for structures in data and classify them into categories such that the degree of association is high among structures of the same kind. Prototypical categories are usually characterized from past experience, and can be done by more than one structure.

Classification of objects falls in the category of *cluster analysis*, which plays a key role in pattern recognition. Cluster analysis it is not restricted to only pattern recognition, but is applicable to the taxonomies in biology and other areas, classification of information, and social groupings.

Fuzzy set theory has been used in pattern recognition since the mid-1960s. We can find three fundamental problems in pattern recognition, where most categories have vague boundaries. In general, objects are represented by a vector of measured values of r variables: $a = [a_1 \ldots a_r]$.

This vector is called a *pattern vector,* where a_i (for each $i \in \mathrm{N}_r$) is a particular characteristic of interest. The first problem is concerned with representation of input data, which is obtained from the objects to be recognized, known as *sensing problems*. The second problem concerns the extraction of different features from the input data, in terms of the dimension of the pattern vector; they are called *feature extraction problems*. These features should characterize attributes, which determine the pattern classes.

The third problem involves the determination of optimal decision procedures for the classification of given patterns. Most of the time this is done by defining an *appropriate discrimination function* of patterns by assigning a real number to a pattern vector. Then, individual pattern vectors are evaluated by discrimination functions, and the classification is designed by the resulting number.

6.1.3 Introduction to Tabu Search

There are many problems that need to be solved by optimization procedures. Genetic algorithms (GA) or simulated annealing is used for that purpose. Tabu search (TS) is among the methods found in this field of optimization solutions. As its name suggests, tabu search is an algorithm performing the search in a region for the minimum or maximum solution of a given problem.

Searching is quite complicated because it uses a lot of memory and spends too much time in the process. For this reason, tabu search is implemented as an intelligent algorithm to take advantage of memory and to search more efficiently.

6.1.4 Industrial Applications of Simulated Annealing

We will briefly describe some industrial applications of the simulated annealing. Scheduling is always a difficult problem in the industry. Processes and logistics must be carefully combined to harmonize and increase production of plants. In 1997 A.P. Reynolds [1] and others presented a paper on simulated annealing for industrial applications. They optimized the scheduling process in order to optimize the resources of a manufacturing plant to meet the demand of different products.

S. Saika and other researchers [2] from Matsushita at the Advanced LSI Technology Development Center introduced a high-performance simulated annealing application to transistor placement. Called widely stepping simulated annealing, they applied it to the 1D transistor placement optimizations used in several industrial cells. They claim to have solutions as good as the standard algorithm and better, with a processing time one-thirtieth that of the normal simulated annealing.

R.N. Bailey, K.M. Garner, and M.F. Hobbs published a paper [3] showing the application of simulated annealing and GAs to solve staff scheduling problems. They use the algorithms to solve the scheduling of the work of staff with different skill levels, which is difficult to achieve because there is a large number of solutions. The results show that both simulated annealing and GAs can produce optimal and near-optimal solutions in a relatively short time for the nurse scheduling problem.

6.1.5 Industrial Applications of Fuzzy Clustering

Manufacturing firms have increased the use of industrial robots over the years. There has also been an increase in the number of robot manufacturers, offering a wide range of products. This is how M. Khouja and D.E. Booth [4] used a fuzzy clustering technique for the evaluation and selection of industrial robots given a specific application. They take into consideration real-world data instead to create the model.

B. Moshiri and S. Chaychi [5] use fuzzy logic and fuzzy clustering to model complex systems and identify non-linear industrial processes. They claim that their proposed advantage is simple, flexible and of high accuracy, easy to use and automatic. They applied this system to a heat exchanger.

6.1.6 Industrial Applications of Tabu Search

Tabu search has been widely used to optimize several industrial applications. For example, L. Zhang [6] and his team proposed a tabu search scheme to optimize the vehicle routing problem, with the objective of finding a schedule that will guarantee

the safety of all vehicles. Their algorithm proved to be good enough compared with other more mature algorithms specially designed for the vehicle problem.

Artificial neural networks (ANNs) based on the tabu search algorithm have also been used by H. Shuang [7] to create a wind speed prediction model. A backpropagation neural network has its weights optimized using tabu search. Then the neural network is used as a model to predict the wind speed 1 hour ahead. It improved the prediction compared with a simple backpropagation neural network.

In 2007 J. Brigitte and S. Sebbah presented a paper [8] in which 3G networks are optimized. The location of primary bases and the core network link capacity is optimized. The dimensioning problem is modeled as a mixed-integer program and solved by a tabu search algorithm; the search criteria includes the signal-to-noise plus interference ratio. Primary bases are randomly located and after a few iterations their location is changed and the dimensioning optimized.

This base optimization problem was previously addressed by C.Y. Lee and published in a paper in 2000 [9]. He also aimed to minimize the number of base stations used and its location in an area covered by cellular communications. The results presented show that a 10 % in cost reduction is achieved, and between a 10 and 20% of cost reduction in problems with 2500 traffic demand areas with code division multiple access (CDMA) systems.

6.2 Simulated Annealing

It was in 1982 and 1983 that Kirkpatrick, Gelatt and Vecchi introduced the concepts of annealing in combinatorial optimization. It was also independently presented in 1985 by Černy. The concepts are based on the physical annealing process of solids and the problem of solving large optimization problems.

Annealing is a physical process where a substance is heated and cooled in a controlled manner. The results obtained by this process are strong crystalline structures, compared to structures obtained by fast untempered cooling, which result in brittle and defective structures. For the optimization process the structure is our encoded solution, and the temperature is used to determine how and when new solutions are accepted. The process contains two steps [4, 10]:

1. *Increase* temperature of the heat bath to a maximum value at which the solid melts.
2. *Carefully* decrease the temperature of the heat bath until particles arrange themselves in the ground state of the solid.

When the structure is in the liquid phase all the particles of the solid arrange themselves in a random way. In the ground state the particles are arranged in a highly structured lattice, leaving the energy of the system at its minimum. This ground state of the solid is only obtained if the maximum temperature is sufficiently high and the cooling is sufficiently low, otherwise, the solid will be frozen into a metastable state rather than the ground state.

Computer simulation methods from condensed matter physics are used to model the physical annealing process. Metropolis and others introduced a simple algorithm to simulate the evolution of a solid in a heat bath at thermal equilibrium. This algorithm is based on Monte Carlo techniques, which generate a sequence of states of the solid.

These states act as the following: given the actual state i of the solid that has energy E_i, the subsequent state j is generated by applying a perturbation mechanism which transforms the present state into the next state causing a small distortion, like displacing a particle. For the next state E_j, if the *energy difference* $E_j - E_i$ is less than or equal to zero, then the j is accepted as the current state. If the energy difference is greater than zero, then the j state is accepted with a certain probability, given by: $\mathrm{e}^{\left(\frac{E_i - E_j}{k_\mathrm{B} T}\right)}$.

Here, T denotes the temperature of the heat bath, and k_B is a constant known as the Boltzmann constant. We will now describe the *Metropolis criterion* used as the acceptance rule. The algorithm that goes with it is known as the *Metropolis algorithm*.

If the temperature is lowered sufficiently slowly, then the solid will reach thermal equilibrium at each temperature. In the Metropolis algorithm this is achieved by generating a large number of transitions at a given temperature value. The thermal equilibrium is characterized by a Boltzmann distribution, which gives the probability of the solid being in the state i with an energy E_i at temperature T:

$$P_T\{X = i\} = \frac{1}{Z(T)} \mathrm{e}^{\left(-\frac{E_i}{k_\mathrm{B} T}\right)}, \tag{6.1}$$

where X is a stochastic variable that denotes the state of the solid in its current form, and $Z(T)$ is a *partition function*, defined by:

$$Z(T) = \sum_j \mathrm{e}^{\left(-\frac{E_j}{k_\mathrm{B} T}\right)}. \tag{6.2}$$

The sum will extend over all the possible states. The simulated annealing algorithm is very simple and can be defined in six steps [11], as shown in Fig. 6.1.

1. *Initial Solution*
   The initial solution will be mostly a random one and gives the algorithm a base from which to search for a more optimal solution.
2. *Assess Solution*
   Consists of decoding the current solution and performing whatever action is necessary to evaluate it against the given problem.
3. *Randomly Tweak Solution*
   Randomly modify the working solution, which depends upon the encoding.
4. *Acceptance Criteria*
   The working solution is compared to the current solution, if the working one has less energy than the current solution (a better solution) then the working solution is copied to the current solution and the temperature is reduced. If the working

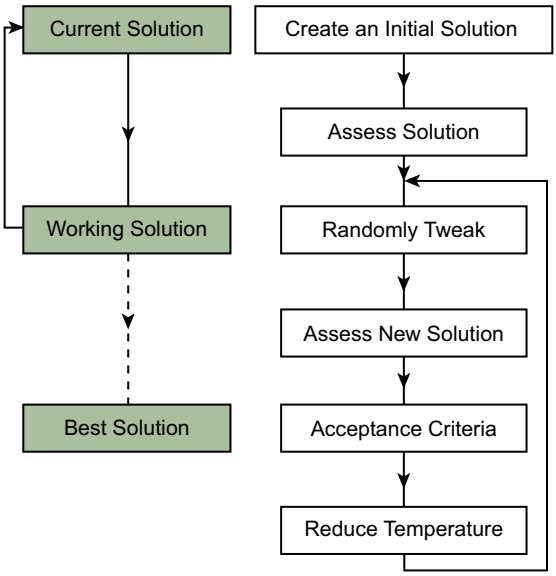


Fig. 6.1 Simulated annealing algorithm

solution is worse than the current one, the acceptance criteria is evaluated to determine what to do with the current solution. The probability is based on (6.3):

$$P(\delta E) = \mathrm{e}^{-\frac{\delta E}{T}}, \tag{6.3}$$

which means that at higher temperatures poorer solutions are accepted in order to search in a wider range of solutions.

5. *Reduce Temperature*
   After a certain number of iterations the temperature is decreased. The simplest way is by means of a geometric function $T_{i+1} = \alpha T_i$, where the constant α is less than one.
6. *Repeat*
   A number of operations are repeated at a single temperature. When that set is reduced the temperature is reduced and the process continues until the temperature reaches zero.

6.2.1 Simulated Annealing Algorithm

We need to assume an analogy between the physical system and a combinatorial optimization problem, based on the following equivalences:

- Solutions in a combinatorial optimization problem are equivalent to states of a physical system.
- The energy of a state is the cost of a solution.

The *control parameter* is the temperature, and with all these features the simulated annealing algorithm can now be viewed as an iteration of the Metropolis algorithm evaluated at decreasing values of the control parameters. We will assume the ex-

istence of a neighborhood structure and a generation mechanism; some definitions will be introduced.

We will denote an instance of a combinatorial optimization problem by (S, f), and i and j as two solutions with their respective costs $f(i)$ and $f(j)$. Thus, the acceptance criterion will determine if j is accepted by i by applying the following *acceptance probability*:

$$P_c(\text{accepted } j) = \begin{cases} 1 & \text{if } f(j) \leq f(i) \\ e^{\left(\frac{f(i)-f(j)}{c}\right)} & \text{if } f(j) > f(i) \end{cases} . \tag{6.4}$$

where here $c \in R^+$ denotes the control parameter. The generation mechanism corresponds to the perturbation mechanism equivalent at the Metropolis algorithm and the acceptance criterion is the Metropolis criterion.

Another definition to be introduced is the one of *transition*, which is a combined action resulting in the transformation of a current solution into a subsequent one. For this action we have to follow the next two steps: (1) application of the generation mechanism, and (2) application of the acceptance criterion.

We will denote c_k as the value of the control parameter and L_k as the number of transitions generated at the kth iteration of the Metropolis algorithm. A formal version of the simulated annealing algorithm [5] can be written in pseudo code as shown in Algorithm 6.1.

Algorithm 6.1

```
SIMULATED ANNEALING
init:
k = 0
i = i_start
repeat
 for l = 1 to L_k do
 GENERATE j from S_i :
 if f(j) ≤ f(i) then i = j
   else
    if e^((f(i)−f(j))/c_k) > rand[0,1) then i = j
   k = k + 1
   CALCULATE LENGTH (L_k)
   CALCULATE CONTROL (L_k)
until stopcriterion
end
```

The probability of accepting perturbations is implemented by comparing the value of $e^{f(i)-f(j)}/c$ with random numbers generated in $(0, 1)$. It should also be obvious that the speed of convergence is determined by the parameters L_k and c_k.

A feature of simulated annealing is that apart from accepting improvements in cost, it also accepts, to a limited extent, deteriorations in cost. With large values of c large deteriorations or changes will be accepted. As the value of c decreases, only smaller deteriorations will be accepted. Finally, as the value approaches zero, no perturbations will be accepted at all. This means that the simulated annealing algorithm can escape from local minima, while it still is simple and applicable.

6.2.2 Sample Iteration Example

Let us say that the current environment temperature is 50 and the current solution has an energy of 10. The current solution is perturbed, and after calculating the energy the new solution has an energy of 20. In this case the energy is larger, thus worse, and we must therefore use the acceptance criteria. The delta energy of this sample is 10. Calculating the probability we will have:

$$P = \mathrm{e}^{\left(-\frac{10}{50}\right)} = 0.818731\,. \tag{6.5}$$

So for this solution it will be very probable that the less ideal solution will be propagated forward. Now taking our schedule at the end of the cycle, the temperature will be now 2 and the energies of 3 for the current solution, and 7 for the working one. The delta energy of the sample is 4. Therefore, the probability will be:

$$P = \mathrm{e}^{\left(-\frac{4}{2}\right)} = 0.135335\,. \tag{6.6}$$

In this case, it is very unlikely that the working solution will be propagated in the subsequent iterations.

6.2.3 Example of Simulated Annealing Using the Intelligent Control Toolkit for LabVIEW

We will try to solve the N-queens problem (NQP) [3], which is defined as the placement of N queens on an $N \times N$ board such that no queen threatens another queen using the standard chess rules. It will be solved in a 30×30 board.
Encoding the solution. Since each column contains only one queen, an N-element array will be used to represent the solution.
Energy. The energy of the solution is defined as the number of conflicts that arise, given the encoding. The goal is to find an encoding with zero energy or no conflicts on the board.
Temperature schedule. The temperature will start at 30 and will be slowly decreased with a coefficient of 0.99. At each temperature change 100 steps will be performed.

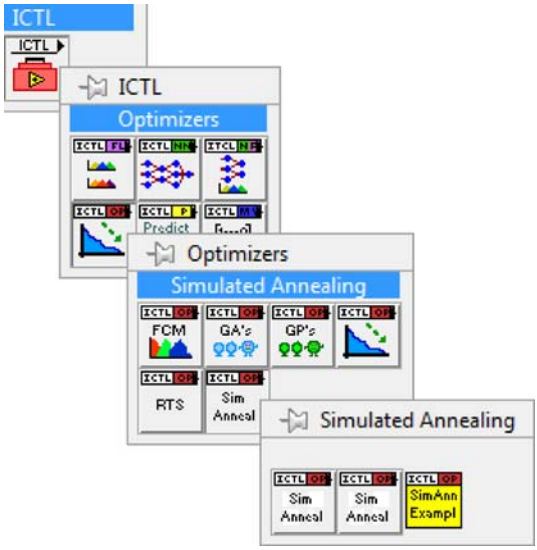


Fig. 6.2 Simulated annealing VIs

The initial values are: initial temperature of 30, final temperature of 0.5, alpha of 0.99, and steps per change equal to 100. The VIs for the simulated annealing are found at: Optimizers ≫ Simulated Annealing, as shown in Fig. 6.2.

The front panel is like the one shown in Fig. 6.3. We can choose the size of the board with the *MAX_LENGTH* constant. Once a solution is found the green LED *Solution* will turn on. The initial constants that are key for the process are introduced in the cluster *Constants*. We will display the queens in a 2D array of bits. The *Current*, *Working* and *Best* solutions have their own indicators contained in clusters.

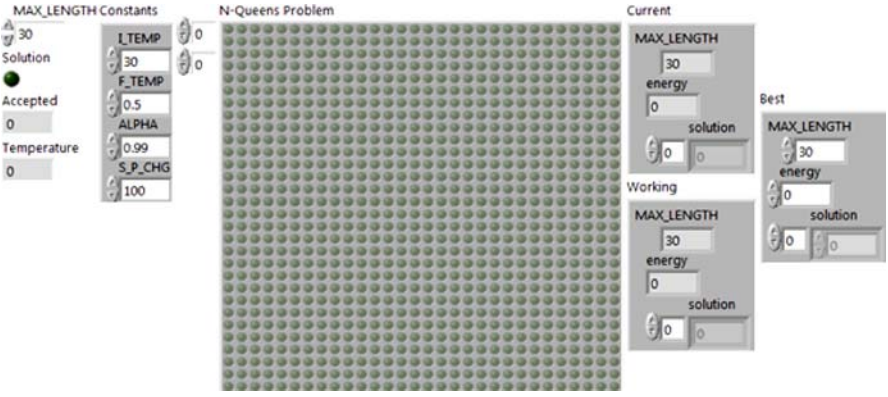


Fig. 6.3 Front panel for the simulated annealing example

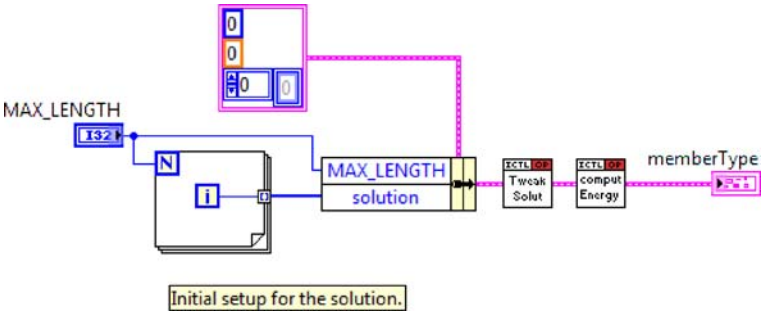


Fig. 6.4 Block diagram for the generation of the initial solution

Our initial solution can be created very simply; each queen is initialized occupying the same row as its column. Then for each queen the column will be varied randomly. The solution will be tweaked and the energy computed. Figure 6.4 shows the block diagram of this process.

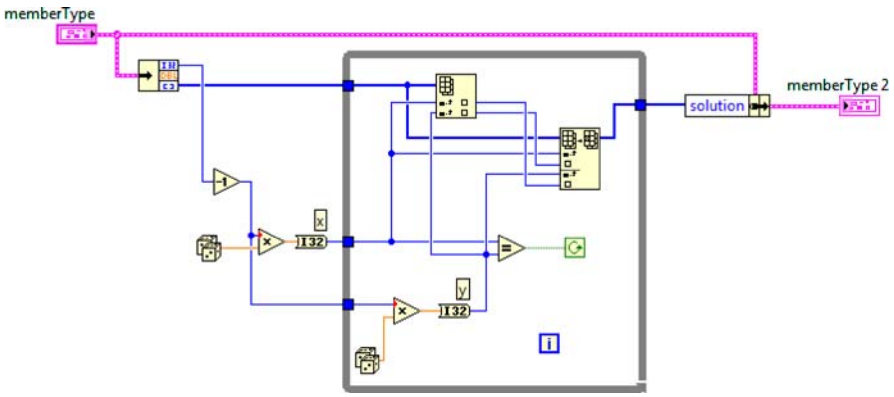


Fig. 6.5 Code for the tweaking process of the solution

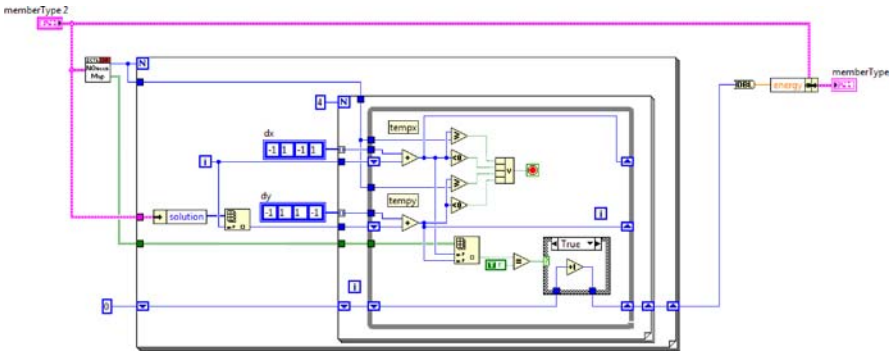


Fig. 6.6 Code for the computation of energy

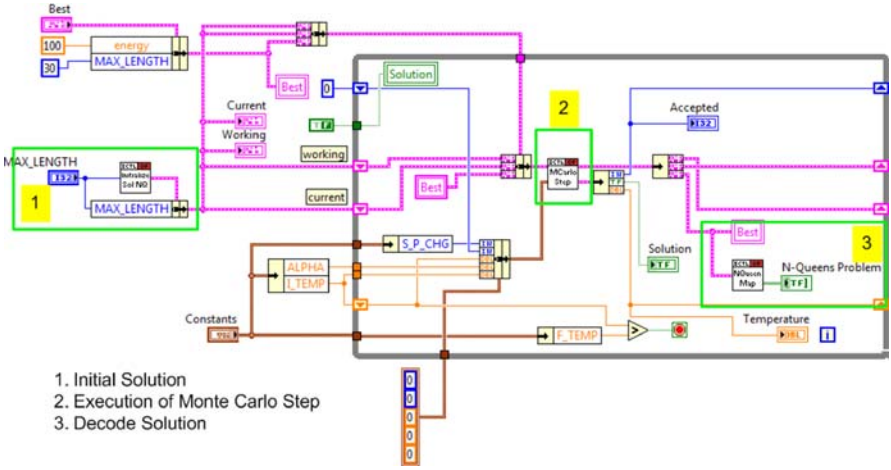


Fig. 6.7 Block diagram of the simulated annealing example for the N-queen problem

The tweaking is done by the code shown in Fig. 6.5; basically it randomizes the position of the queens. The energy is computed with the following code. It will try to find any conflict in the solution and assess it. It will select each queen on the board, and then on each of the four diagonals looking for conflicts, which are other queens in the path. Each time one is found the conflict variable is increased. In Fig. 6.6 we can see the block diagram. The final block diagram is shown in Fig. 6.7.

6.3 Fuzzy Clustering Means

In the field of optimization, fuzzy logic has many beneficial properties. In this case, *fuzzy clustering means* (FCM), known also as *fuzzy c-means* or *fuzzy k-means*, is a method used to find an optimal clustering of data.

Suppose, we have some collection of data $\mathbf{X} = \{x_1, \ldots, x_n\}$, where every element is a vector point in the form of $x_i = (x_i^1, \ldots, x_i^p) \in \Re^p$. However, data is spread in the space and we are not able to find a clustering. Then, the purpose of FCM is to find clusters represented by their own centers, in which each center has a maximum separation from the others. Actually, every element that is referred to as clusters must have the minimum distance between the cluster center and itself. Figure 6.8 shows the representation of data and the FCM action.

At first, we have to make a partition of the input data into c subsets written as $P(X) = \{U_1, \ldots, U_c\}$, where c is the number of partitions or the number of clusters that we need. The partition is supposed to have fuzzy subsets U_i. These subsets must satisfy the conditions in (6.7) to (6.8):

$$\sum_{i=1}^{c} U_i(x_k) = 1, \quad \forall x_k \in X \tag{6.7}$$

$$0 < \sum_{k=1}^{n} U_i(x_k) < n\,. \tag{6.8}$$

The first condition says that any element x_k has a fuzzy value to every subset. Then, the sum of membership values in each subset must be equal to one. This condition suggests to elements that it has some membership relation to all clusters, no matter how far away the element to any cluster. The second condition implies that every cluster must have at least one element and every cluster cannot contain all elements in the data collection. This condition is essential because on the one hand, if there are no elements in a cluster, then the cluster vanishes.

On the other hand, if one cluster has all the elements, then this clustering is trivial because it represents all the data collection. Thus, the number of clusters that FCM can return is $c = [2, n-1]$. FCM need to find the centers of the fuzzy clusters. Let $v_i \in \Re^p$ be the vector point representing the center of the ith cluster, then

$$v_i = \frac{\sum_{k=1}^{n} [U_i(x_k)]^m \, x_k}{\sum_{k=1}^{n} [U_i(x_k)]^m}, \quad \forall i = 1, \ldots, c\,, \tag{6.9}$$

where $m > 1$ is the *fuzzy parameter* that influences the grade of the membership in each fuzzy set. If we look at (6.9), we can see that it is the weighted average of the data in U_i. This expression tells us that centers may or may not be any point in the data collection.

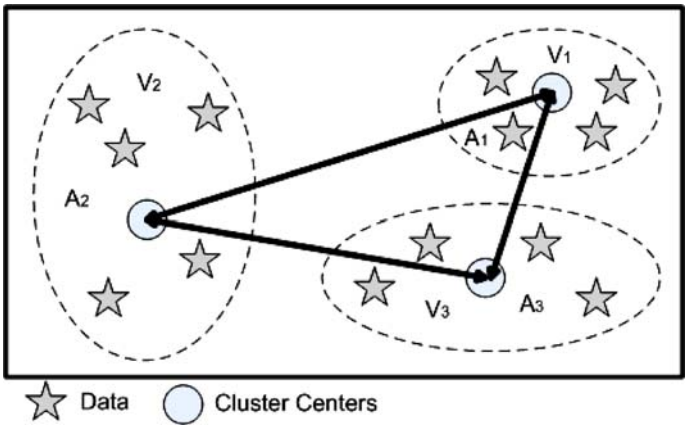


Fig. 6.8 Representation of the FCM algorithm

Actually, FCM is a recursive algorithm, and therefore needs an objective function that estimates the optimization process. We may say that the objective function $J_m(P)$ with grade m of the partition $P(X)$ is shown in (6.10):

$$J_m(P) = \sum_{k=1}^{n} \sum_{i=1}^{c} [U_i(x_k)]^m \, \| x_k - v_i \|^2 . \tag{6.10}$$

This objective function represents a measure of how far the centers are from each other, and how close the elements in each center are. For instance, the smaller the value of $J_m(P)$, the better the partition $P(X)$. In these terms, the goal of FCM is to minimize the objective function.

We present the FCM algorithm developed by J. Bezdek for solving the clustering data. At first, we have to select a value $c = [2,\ n-1]$ knowing the data collection X. Then, we have to select the fuzzy parameter $m = (1,\ \infty)$. In the initial step, we select a partition $P(X)$ randomly and propose that $J_m(P) \to \infty$. Then, the algorithm calculates all cluster centers by (6.9). Then, it updates the partition by the following procedure: for each $x_k \in X$ calculate

$$U_i(x_k) = \left[\sum_{j=1}^{c} \left(\frac{\| x_k - v_i \|^2}{\| x_k - v_j \|^2} \right)^{\frac{1}{m-1}} \right]^{-1} , \quad \forall i = 1, \ldots, c . \tag{6.11}$$

Finally, the algorithm derives the objective function with values found by (6.9) and (6.11), and it is compared with the previous objective function. If the difference between the last and current objective functions is close to zero (we say $\varepsilon \geq 0$ is a small number called the *stop criterion*), then the algorithm stops. In another case, the algorithm recalculates cluster centers and so on. Algorithm 6.2 reviews this discussion. Here $n = [2, \infty)$, $m = [1, \infty)$, U are matrixes with the membership functions from every sample of the data set to each cluster center. P are the partition functions.

Algorithm 6.2	FCM procedure
Step 1	Initialize time $t = 0$. Select numbers $c = [2,\ n-1]$ and $m = (1,\ \infty)$. Initialize the partition $P(X) = \{U_1, \ldots, U_c\}$ randomly. Set $J_m(P)^{(0)} \to \infty$.
Step 2	Determine cluster centers by (6.9) and $P(X)$.
Step 3	Update the partition by (6.11).
Step 4	Calculate the objective function $J_m(P)^{(t+1)}$.
Step 5	If $J_m(P)^{(t)} - J_m(P)^{(t+1)} > \varepsilon$ then update $t = t + 1$ and go to *Step 2*. Else, *STOP*.

Example 6.1. For the data collection shown in Table 6.1 with 20 samples. Cluster in three subsets with a FCM algorithm taking $m = 2$.

Table 6.1 Data used in Example 6.1

Number	X data	Number	X data	Number	X data	Number	X data
1	255	6	64	11	58	16	80
2	67	7	64	12	96	17	80
3	67	8	71	13	96	18	71
4	74	9	71	14	87	19	71
5	74	10	58	15	87	20	62

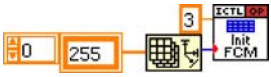


Fig. 6.9 Block diagram of the initialization process

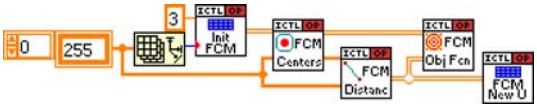


Fig. 6.10 Block diagram of partial FCM algorithm

Solution. The FCM algorithm is implemented in LabVIEW in several steps. First, following the path ICTL ≫ Optimizers ≫ FCM ≫ FCM methods ≫ **init_fcm.vi**. This VI initializes the partition. In particular, it needs the number of clusters (for this example 3) and the size of the data (20). The output pin is the partition in matrix form. Figure 6.9 shows the block diagram. The 1D array is the vector in which the twenty elements are located.

Then, we need to calculate the cluster centers using the VI at the path ICTL ≫ Optimizers ≫ FCM ≫ FCM methods ≫ **centros_fcm.vi**. One of the input pins is the matrix **U** and the other is the data. The output connections are referred to as U^2 and the cluster centers *Centers*. Then, we have to calculate the objective function. The VI is in ICTL ≫ Optimizers ≫ FCM ≫ FCM methods ≫ **fun_obj_fcm.vi**. This VI needs two inputs, the U^2 and the distances between elements and centers. The last procedure is performed by the VI found in the path ICTL ≫ Optimizers ≫ FCM ≫ FCM methods ≫ **dist_fcm.vi**. It needs the cluster centers and the data. Thus, **fun_obj_fcm.vi** can calculate the objective function with the distance and the partition matrix powered by two coming from the previous two VIs. In the same way, the partition matrix must be updated by the VI at the path ICTL ≫ Optimizers ≫ FCM ≫ FCM methods ≫ **new_U_fcm.vi**. It only needs the distance between elements and cluster centers. Figure 6.10 shows the block diagram of the algorithm.

Of course, the recursive procedure can be implemented with either a *while-loop* or a *for-loop* cycle. Figure 6.11 represents the recursive algorithm. In Fig. 6.11 we create a *Max Iterations* control for number of maximum iterations that FCM could reach. The *Error* indicator is used to look over the evaluation of the objective function and *FCM Clusters* represents graphically the fuzzy sets of the partition matrix found. We see at the bottom of the while-loop, the comparison between the last error and the current one evaluated by the objective function. □

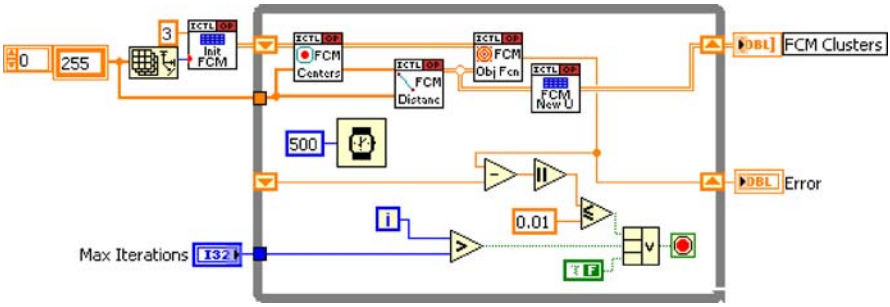


Fig. 6.11 Block diagram of the complete FCM algorithm

6.4 FCM Example

This example will use previously gathered data and classify it with the FCM algorithm; then we will use T-ANNs to approximate each cluster. The front panel is shown in Fig. 6.12.

We will display the normal FCM clusters in a graph, the approximated clusters in another graph and the error obtained by the algorithm in an *XY* graph. We also need to feed the program with the number of neurons to be used for the approximation, the number of clusters and the maximum allowed iterations. Other information can be displayed like the centers of the generated clusters and the error between the ap-

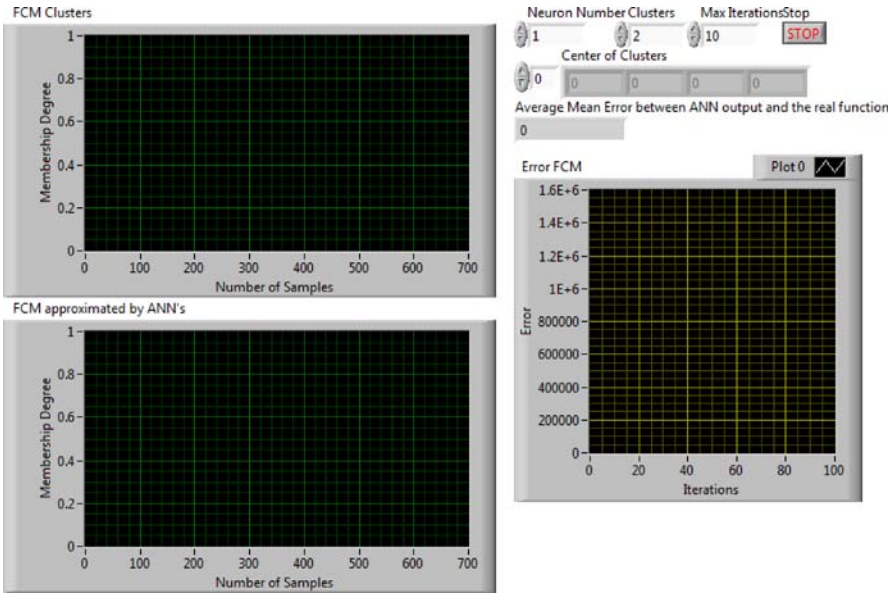


Fig. 6.12 Front panel of the FCM example

proximated version of the clusters and the normal one. This example can be located at Optimizers ≫ FCM ≫ **Example_FCM.vi** where the block diagram can be fully inspected, as seen in Fig. 6.13 (with the results shown in Fig. 6.14).

This program takes information previously gathered, then initializes and executes the FCM algorithm. It then orders the obtained clusters and trains a T-ANNs with

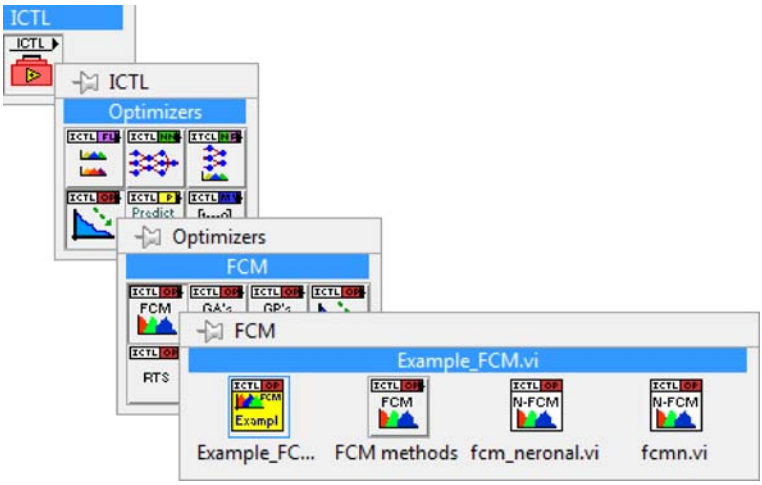


Fig. 6.13 VIs for the FCM technique

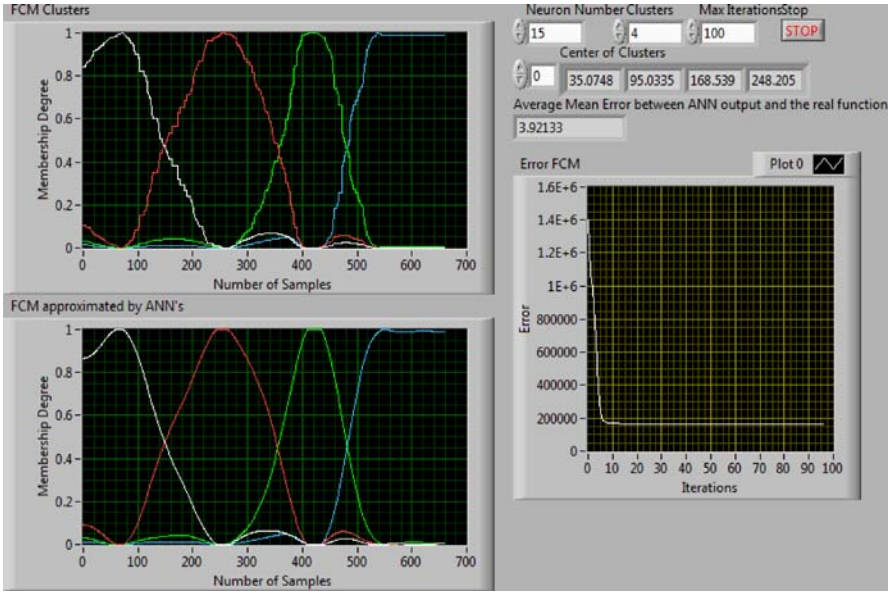


Fig. 6.14 The FCM program in execution

the information of each cluster. After that the T-ANNs are evaluated, and the average mean error between the approximated and the real clusters are calculated.

6.5 Partition Coefficients

FCM described in Algorithm 6.1 is very useful in pattern recognition techniques. However, no matter which application is being developed, FCM has a problem: what could be the value for the number of clusters? The answer is the *partition coefficient*.

Partition coefficient (PC) is a method used to validate how well a clustering algorithm has identified the structure presented in the data, and how it represents it into clusters. This small algorithm is based on the following:

$$PC\ (\mathbf{U};c) = \frac{\sum_{j=1}^{n}\sum_{i=1}^{c}\left(u_{ij}\right)^2}{n}, \tag{6.12}$$

where $\mathbf{U}$ is the partition matrix and u_{ij} is the membership value of the jth element of the data related to the ith cluster, c is the number of clusters and n is the number of elements in the data collection. From this equation, it can be noted that the closer the PC is to 1, the better classified the data is considered to be. The optimal number of clusters can be denoted at each c by Ω_c using (6.13):

$$\max_{c}\left[\max_{\Omega_c \in U}\{PC\ (U;c)\}\right]. \tag{6.13}$$

Algorithm 6.3 shows the above procedure.

Algorithm 6.3	Partition coefficient
Step 1	Initialize $c = 2$. Run FCM or any other clustering algorithm.
Step 2	Calculate the partition coefficient by (6.12).
Step 3	Update the value of clusters $c = c + 1$.
Step 4	Run until no variations at PC are found and obtain the optimal value of clusters by (6.13).
Step 5	Return the optimal value c and *STOP*.

Example 6.2. Assume the same data as in Example 6.1. Run the PC algorithm and obtain the optimal number of clusters.

Solution. The partition coefficient algorithm is implemented in LabVIEW at ICTL ≫ Optimizers ≫ Partition Coeff. ≫ **PartitionCoefficients.vi**. On the inside of this VI, the FCM algorithm is implemented. So, the only thing we have to do is to connect the array of data and the number of clusters at the current iteration. Figure 6.15

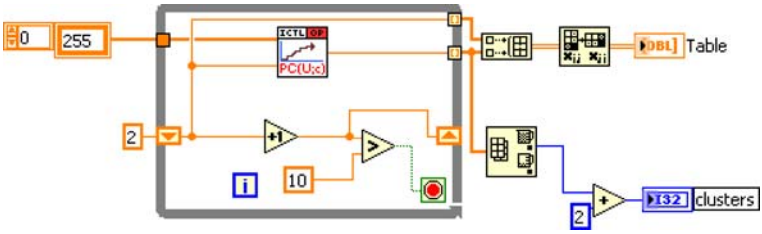


Fig. 6.15 Block diagram finding the optimal number of clusters

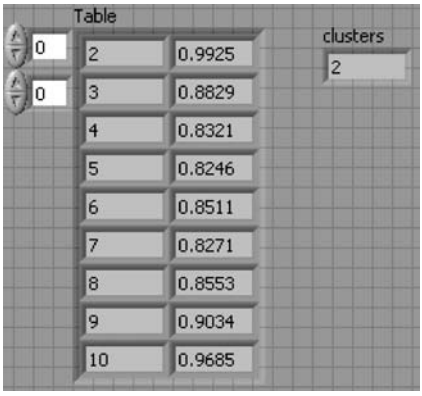


Fig. 6.16 Front panel of Example 6.2 showing the optimal value for clusters

is the block diagram of the complete solution of this example. In this way, we initialize the number of clusters in 2 and at each iteration, this number is increased. The number 10 is just for stopping the process when the number of clusters is larger than this. Finally, in *Table* we find the evaluated PC at each number of clusters and *clusters* indicates the number of optimal clusters for this particular data collection. Figure 6.16 shows the front panel of this example. The solution for this data collection is 2 clusters. □

6.6 Reactive Tabu Search

6.6.1 Introduction to Reactive Tabu Search

The word *tabu* means that something is dangerous, and taking it into account involves a risk. This is not used to avoid certain circumstances, but instead is used in order to prohibit features, for example, until the circumstances change. As a result, tabu search is the implementation of intelligent decisions or the responsive exploration in the search space.

The two main properties of tabu search are *adaptive memory* and *responsive exploration*. The first term refers to an adaptation of the memory. Not everything

is worth remembering, but not everything is worth forgetting either. This property is frequently used to make some subregion of the search space tabu. Responsive exploration is a mature decision in what the algorithm already knows, and can be used to find a better solution. The latter, is related to the rule by which tabu search is inspired: a bad strategic choice can offer more information than a good random choice. In other words, sometimes is better to make a choice that does not qualify as the best one at that time, but it can be used to gather more information than the better solution at this time.

More precisely, tabu search can be described as a method designed to search in not so feasible regions and is used to intensify the search in the neighborhood of some possible optimal location.

Tabu search uses memory structures that can operate in a distinct kind of region, which are recency, frequency, quality, and influence. The first and the second models are *how recent* and *at what frequency* one possible solution is performed. Thus, we need to record the data or some special characteristic of that data in order to count the frequency and the time since the same event last occurred. The third is *quality*, which measures how attractive a solution is. The measurement is performed by features or characteristics extracted from data already memorized. The last structure is *influence*, or the impact of the current choice compared to older choices, looking at how it reaches the goal or solves the problem. When we are dealing with the direct data information stored, memory is *explicit*. If we are storing characteristics of the data we may say that memory is *attributive*.

Of course, by the adaptive memory feature, tabu search has the possibility of storing relevant information during the procedure and forgetting the data that are not yet of interest. This adaptation is known as *short term memory* when data is located in memory for a few iterations; *long term memory* is when data is collected for a long period of time.

Other properties of tabu search are the *intensification* and *diversification* procedures. For example, if we have a large search region, the algorithm focuses on one possible solution and then the intensification procedure explores the vicinity of that solution in order to find a better one. If in the exploration no more solutions are optimally found, then the algorithm diversifies the solution. In other words, it leaves the vicinity currently explored and goes to another region in the search space. That is, tabu search explores large regions choosing small regions in certain moments.

6.6.2 Memory

Tabu search has two types of memory: short term and long term-based memories. In this section we will explain in more detail how these memories are used in the process of optimizing a given problem.

To understand this classification of memory, it is necessary to begin with a mathematical description. Suppose that the search space is V so $x \in V$ is an element of

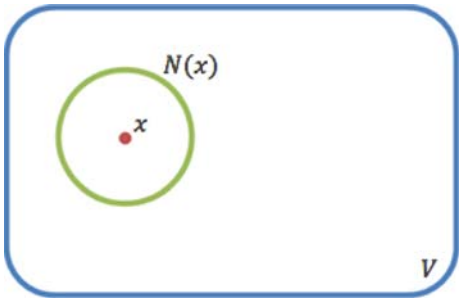


Fig. 6.17 Search space of the tabu search

the solution in the search space. The algorithm searches in the vicinity of x known as $N(x)$ where $N(x) \subset V$. Figure 6.17 shows these terms.

6.6.2.1 Short Term Memory

The first memory used in tabu search is the short term. When we are searching in the vicinity $N(x)$, and we try to do it as fast as possible. Another method is the so-called *steepest descent method* that can be explained as follows: we pick up an element x, then we take an objective function $f(x)$ and store this value. In the next iteration we look for an element $x' \in N(x)$ and evaluate the function $f(x')$. If $f(x') < f(x)$ then x' is the new optimal solution. We repeat until the condition is not true. Therefore, the solution of that method is known as the local optimum, because the solution is the optimal one in the vicinity but not in the entire search space.

This process is very expensive computationally. Therefore, a short term memory is used. We try to delimit the search space by the vicinity of an element of that space. Then, we try to minimize as much as possible the search space. How can we do it? If some features of the solution are really known, then it is easy to deduce that some solutions are prohibited. For example, the last element evaluated is prohibited from being selected. This is a simple example, but some other characteristics might be applied to avoid the selection of these possible solutions. So, there exists a subspace of the vicinity $N(x)$ that is a tabu list, namely T. The new vicinity is characterized as $N^*(x) = N(x)/T$. In this way, tabu search is an algorithm that explores a dynamic vicinity of the solution.

The tabu list or the tabu space is stored in short term memory. One of the simplest uses of this memory is in the recent process. If some solution x is evaluated then the next few iterations are prohibited.

In the same manner, when we talk about iterations of selecting new elements to evaluate, we are trying to say that the change or the *move* between the current solution and the following is evaluated to know if this change is really useful or not. The dynamic vicinity of movement is distinguished by two classifications: the vicinity of permissible moves and the tabu moves. Then, by the attributes of the

elements x we can determine if some move can be added to the permissible vicinity $N^*(x)$ or if the move might be dropped to the T subspace.

In order to assign the time of prohibition of some special element, the tabu tenure is created. The tabu tenure is a value t in the interval $[t_{\min},\ t_{\max}]$ that describes the number of prohibited iterations remaining until the element will be reused. This value can be assigned randomly or in a more systematic form.

Example 6.3. Let $V = \{9, 4, 6, 1, 8, 2\}$ be the values of the search space and the vicinity $N(x) = \{4,\ 8,\ 9\}$ with value $x = 6$, considering the vicinity with a radius of 3. Then, assume the tabu list of the entire domain as $T(V) = \{0,\ 0,\ 5,\ 0,\ 4,\ 0\}$. Suppose that $t \in [0,\ 5]$ and when some element is selected, the tabu tenure is 5 and all the other components of T are decreased by one. (a) What is the permissible vicinity $N^*(6)$? (b) What was the last value selected before $x = 6$? (c) If we are looking around $x = 4$ what could be the entire vicinity $N(4)$?
Solution. (a) Looking at the tabu set, we know that all 0 values mean that elements in that position are permitted. So, the possible elements that can be picked up are $N^*(V) = \{9,\ 4,\ 1,\ 2\}$. But, we are searching in the vicinity $N(6) = \{4,\ 8,\ 9\}$. Then, $N^*(6) = N^*(V) \cap N(6) = \{9,\ 4,\ 1,\ 2\} \cap \{4,\ 8,\ 9\}$. Finally, the permissible set around the element 6 is: $N^*(6) = \{4,\ 9\}$.

(b) As we can see, the current element is 6, then in the tabu list this element has a value of $t = 5$. This matches with the procedure defined in the example. Actually, in this way all other values were decreased by one. Therefore, if we are trying to look before the current value, the tabu list must be $T(V)_{\text{before\_6}} = T(V)_{\text{current}} + 1$. In other words, $T(V)_{\text{before\_6}} = \{d,\ d,\ 0,\ d,\ 5,\ d\}$, where d is a possible value different or equal to zero because we are uncertain. Actually, if 5 is the tenure assigned for the current selection, then the current selection before the 6-element is 8. This is the reason why in the $N^*(6)$, this element does not appear.

(c) Obviously, we just need to look around 4 with a radius of 3. So, the vicinity is $N(4) = \{1,\ 2,\ 6\}$.

With the quality property, some moves can be reinforced. For example, if some element is selected, then the tabu tenure is fired and the element will be in the tabu list. But, if the element has a high quality, then the element could be promoted to a permissible value. This property of the short term memory is then useful in the process of tabu search. □

6.6.2.2 Long Term Memory

In the same way as the short term, this type of memory is used in order to obtain more attributes for the search procedure. However, long term memory is focused on storing information about the past history.

The first implementation is the frequency dimension of memory. In other words, we can address some data information (explicitly or attributively) and after some iterations or movements, try to analyze the frequency with which this data has appeared. In this way, there exists an array of elements named *transition measure*, which counts the number of iterations that some element was modified. Another

array linked with the frequency domain is the *residence measure* and it stores the number of iterations that some element has taken part in the solution.

What is the main purpose of long term memory? It is easy to answer: frequency or what remains the same, transition and residence are measures of how attractive some element is to be included in the vicinity when the frequency is high. On the other hand, if the frequency is low, the element related to that measure has to be removed or placed in the tabu list. Then, long term memory is used to assign attributes to elements in order to discard or accept those in the dynamic vicinity.

Example 6.4. Let $V = \{9, 4, 6, 1, 8, 2\}$ be the values of the search space. Suppose that $tr = \{0,\ 2,\ 5,\ 2,\ 1,\ 4\}$ is the transition array and $r = \{3,\ 1,\ 0,\ 0,\ 2,\ 0\}$ is the residence array. (a) Have any of the elements in V ever been in the best solution thus far? Which one? (b) How many iterations was the element 8 taken into account in the best solution thus far? (c) Can you know at which iteration the best solution was found?

Solution. (a) Yes. There are three elements that have been in the best solution. These elements are 9, 4 and 8 because its residence elements are distinct from zero. In fact, element 9 has been in the solution three times, element 4 has been one time, and element 8 has been two times.

(b) By the transition array we know that element 1 associated to element 8 refers to the fact that this element has been in the best solution only one time. Element 6 has been five times in the best solution so far.

(c) Yes, we can know at which iteration the best solution was found. Suppose that the current iteration is t. The transition array updates if some element has been in the best solution, but if the best solution is modified by the best solution so far, obviously the transition array is initialized. Then, the maximum number of counts that the current transition array has is 5. So, the best solution so far was found at iteration $t_{\text{best}} = t - 5$. □

6.6.2.3 Intensification and Diversification

In the introduction, we said that tabu search is a method that tries to find the best solution in short periods of time without searching exhaustively. These characteristics are offered in some way by the two types of memory described above and by the two following procedures.

The first one is *intensification*. Suppose that we have a search space V, which has n subspaces $W_i, \forall i = 1, \ldots, n$, and that these have no intersection and the two characteristics apply: $(W_i \cap W_j) = \emptyset, \forall i \neq j$ and $W_i \subset V$. So, $V = \{W_i \cup W_j), \forall i \neq j$. This means that the search space can be divided by n regions. We can think of one of these subspaces as in the vicinity of an element of V, so-called $W_i = N(x_i)$. We also know that tabu search looks in this vicinity in order to find the local optimum x_i'. Then, this local optimum is stored in the memory and the process is repeated in another vicinity $N(x_j)$. Suppose now that the algorithm recorded k local optimum elements $\{x_1', \ldots, x_j', \ldots, x_k'\}$. The intensification is thus the process

in which the algorithm visits the vicinity of each of the local optimum elements recorded by means, and looks at the vicinity $N(x_i'), \forall i = 1, \ldots, k$.

In other words, the intensification procedure is a subroutine of the tabu search that tries to find a better solution in the vicinity of the best solutions (local optimum values) so far. This means that it intensifies the exploration.

The second procedure is *diversification*. Let us suppose that we have the same environment as in the intensification procedure. The question is how the algorithm explores distinct places in order to record local optimum values. Diversification is the answer. When some stop searching threshold is fired, the local optimum is recorded and then the algorithm accepts the option to search in a different region, because there may be some other solutions (either better or just as good as the local optimum found so far) placed in other regions.

To make this possible, the algorithm records local optimum values and evaluates permissible movements in the entire region. Therefore, the tabu list has the local optimum elements found so far and all the vicinities of these values. In this way, more local optimum elements mean fewer regions in which other solutions could be found. Or, in the same way, the set of permissible movements comes to be small.

As we can see, the intensification procedure explores regions in which good solutions were found and the diversification procedure permits the exploration into unknown regions. Thus, the algorithm searches as much as it can and restricts all movements when possible to use less time.

6.6.2.4 Tabu Search Algorithm

Tabu search has several modifications in order to get the best solution as fast as possible. In this way, we explain first the general methodology and then we explain in more detail the modification known as reactive tabu search.

6.6.2.5 Simple Tabu Search

First, we need a function that describes the solution. This function is constructed by elements. If the solution has n-dimension size, then the function must have n elements of the form $f = \{f_1, \ldots, f_n\}$. We refer to the function configuration at time t with $f^{(t)} = \{f_1^{(t)}, \ldots, f_n^{(t)}\}$. Then, we aggregate some terminology shown in Table 6.2.

We associate to each of the elements in $f^{(t)}$, a permissible move referred to as $\mu_i, \forall i = 1, \ldots, n$; they are well defined in a set of permissible moves A. All other moves are known as tabu elements displayed in the set τ. Actually, the complement of A is τ.

First, all movements are permitted, so A is all the search space and $\tau = \emptyset$. A configuration is selected randomly. In this case, we need a criterion in order to either intensify the search or to diversify it. The criterion selected here is to know if the current configuration has been selected before. If the frequency of this configura-

Table 6.2 Terminology for simple tabu search

Symbol	Description
t	Iteration counter
f	Configuration at time t
μ	Set of elementary moves
L	Last iteration when the move was applied
Π	Last iteration when the configuration was used
Φ	Number of times the configuration has been visited during the search
f_b	Best configuration known
E_b	Best energy known
A	Set of admissible moves
C	Chaotic moves
S	Subset of A
τ	Set of tabu moves, non-admissible moves
T	Prohibition period

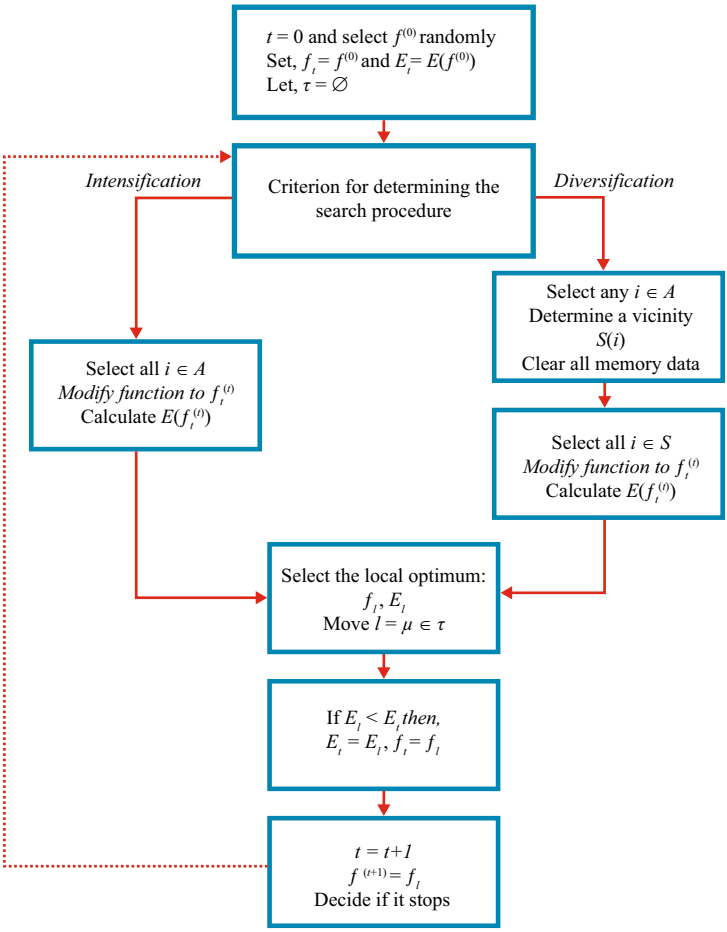


Fig. 6.18 Flow chart of the simple tabu search

tion is high, then the algorithm needs to diversify it; otherwise, the intensification procedure is defined as the following.

Using the current configuration, we need to evaluate all possible elements. An energy function $E(f_i^{(t)}), \forall i = 1, \ldots, |S|$ is evaluated for each of the configurations, done by selecting all the possible moves ($|S|$ means the cardinality or the number of elements in the subset of permissible moves $S \subset A$). The criterion is to select the configuration that minimizes the energy function. After that, this configuration is labeled as the best configuration f_b and the energy is stored in a variable named the best energy known so far E_b.

If we need a diversification procedure, the algorithm erases all memory data and makes a searching process in a distinct region of the search space, which tries to find a local optimum f_b. The best energy and the best configuration is then actualized with this process.

Finally, the iteration time is incremented and the procedure is done until some stop criterion is defined. The algorithm is presented in Algorithm 6.4 and shown in Fig. 6.18.

6.6.2.6 Reactive Tabu Search

In the simple tabu search, the tabu list and permissible moves sets are a function of the last configuration found (local optimum). This is not a good method for finding the solution faster because the tabu list must have more elements than the permissible set when t is large. That is, permissible moves are not enough to be in the current vicinity, and the local optimum might not be the best one in that place.

An alternative to modifying the tabu tenure T is to use the reactive tabu search, which is a modification that gives the possibility of intensifying the search in regions, depending on the historical values of energy in that place. Thus, this method reacts with respect to the intensification/diversification procedures.

In this way, the simple tabu search will be the basis of the process. Then, another subroutine is defined in order to satisfy two main principles: (1) select the intensification/diversification procedure, and (2) modify the tabu tenure.

As in Table 6.1, Φ is a function that returns the number of iterations that some configuration has been visited during the searching procedure, the value is $\Phi(f)$. When the configuration $f^{(t)}$ is used, the function Π stores the actual iteration at which this configuration is evaluated, so the value recorded is $\Pi(f^{(t)}) = t$. The last function is used to store the last iteration at which the current configuration was evaluated. Suppose, that the configuration $f = f_0$ and this configuration are evaluated at time $t = t_0$. Then, five iterations later the same configuration $f^{(t+5)} = f_0$ is evaluated. The last iteration at which this configuration appeared in the searching process is then $\Pi(f^{(t+5)} = f_0) = t_0$.

Finally, this configuration comes from some move μ_i. As with the configuration, the moves have an associated function L that returns the last iteration at which the move was applied in a local optimum and the value recorded is $L(\mu_i)^{(t)} = t$. Let us suppose the same environment as the previous example. Suppose now

Algorithm 6.4	Simple tabu search
Step 1	Let the iteration time be $t = 0$. Initialize the current configuration $f^{(0)}$ randomly. The best configuration is assigned as $f_b = f^{(0)}$ and the best energy as $E_b = E(f^{(0)})$. Actually, the set of permissible moves A is all the search space and the tabu set is $\tau = \emptyset$.
Step 2	Select a criterion to decide between *intensification* (go to *Step 3*) or *diversification* (go to *Step 6*) procedures.
Step 3	With the current configuration $f^{(t)}$ do a modification in one element, the so-called $f_i^{(t)}$ running i for all permissible elements of A. If it is permitted, evaluate the energy $E(f_i^{(t)})$.
Step 4	Select the local optimum configuration temporarily known as f_l and its proper energy E_l. Then, the move μ that produces f_l must be stored in the tabu list τ.
Step 5	Compare the temporary energy with respect to the best energy. If the temporary energy is less than the best energy, then $E_b = E_l$ and $f_b = f_l$. Go to *Step 10*.
Step 6	Select another region in the permissible moves and store the elements in a subset $S \subset A$. Clear memory data used to select the intensification/diversification process.
Step 7	With the current configuration $f^{(t)}$ do a modification in one element, so-called $f_i^{(t)}$ running i for all permissible elements of S. If it is permitted, evaluate the energy $E(f_i^{(t)})$.
Step 8	Select the local optimum configuration temporarily known as f_l and its proper energy E_l. Then, the move μ that produces f_l must be stored in the tabu list τ by a period of T iterations.
Step 9	Compare the temporary energy with respect to the best energy. If the temporary energy is less than the best energy, then $E_b = E_l$ and $f_b = f_l$. Go to *Step 10*.
Step 10	Increment the iteration $t = t + 1$. Set $f^{(t+1)} = f_l$ and go to *Step 2* until the stop criterion is fired, then *STOP*.

that the configuration $f_0^{(t_0)}$ comes from the modification $\mu_x(t_0)$, then we record $L(\mu_x)^{(t_0)} = t_0$. Seven iterations later, some other configuration $f_1^{(t_0+7)}$ comes from the move μ_x, too. Then, if we need to know the last iteration at which this move was in some configuration in the searching procedure, we need to apply $L(\mu_x)^{(t_0+7)} = t_0$.

Now, the subroutine does the next few steps. At first, we evaluate the last iteration $\Pi(f^{(t)})$ at which the current configuration was evaluated and this value is assigned to a variable R. Of course, the number of iterations that this configuration has been in the searching process is updated by the rule $\Phi(f) = \Phi(f) + 1$. If the configuration is greater than some value, i.e., REP_MAX, then we store this configuration in a set of configurations named chaos C by the rule $C = C \cup f$. With this method we are able to know if we have to make a diversification procedure because the number of elements in the set C must be less than some value of threshold $CHAOS$. Otherwise, the algorithm can be in the intensification stage.

On the other hand, we need to be sure that the number of tabu elements is less than the number of permissible values. This action can be derived with the condition $R < 2(L-1)$, which means that the number of the last iteration at which the configuration was in the searching procedure is at least double the number of the last iteration at which the move was in the process. Then, we assume that the variable R can be averaged with the equation $R_{\text{ave}} = 0.1R + 0.9R_{\text{ave}}$. This value controls the tabu tenure as shown in the Algorithm 6.5. Figure 6.19 shows the flow chart of the reactive tabu search.

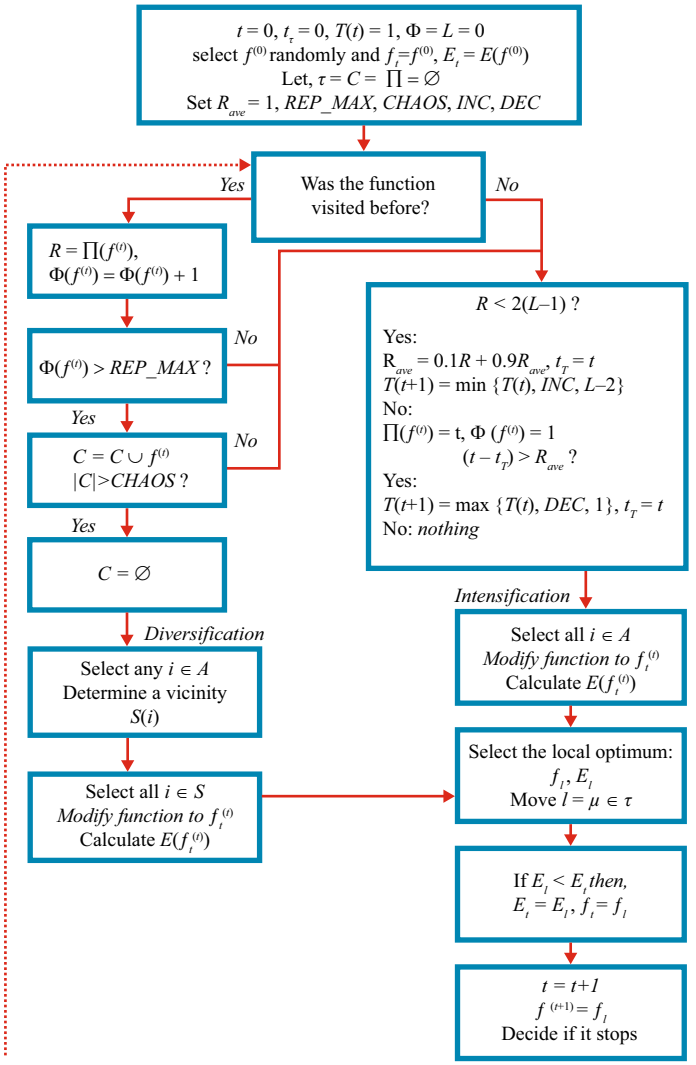


Fig. 6.19 Flow chart of the reactive tabu search

Algorithm 6.5	Reactive tabu search
Step 1	Let the iteration time be $t = 0$. Initialize the current configuration $f^{(0)}$ randomly. The best configuration is assigned as $f_b = f^{(0)}$ and the best energy as $E_b = E(f^{(0)})$. Actually, the set of permissible moves A is all the search space, and the tabu set is $\tau = \emptyset$. Let the set of chaotic configurations be $C = \emptyset$. Set the tabu tenure as $T(t) = 1$ and the last iteration at that point changed to $t_T = 0$. Set $\Pi = \emptyset$ and $\Phi = L = 0$. Initialize the variable $R_{ave} = 1$, REP_MAX, $CHAOS$, INC, DEC.
Step 2	Evaluate $\Pi(f^{(t)})$. If there is any value set the rules: $R = \Pi(f^{(t)})$ and $\Phi(f^{(t)}) = \Phi(f^{(t)}) + 1$. Else, go to *Step 5*.
Step 3	If $\Phi(f^{(t)}) > REP\_MAX$ then update the chaotic set $C = C \cup f^{(t)}$. Else, go to *Step 5*.
Step 4	If $\lvert C \rvert > CHAOS$, then reinitialize $C = \emptyset$ and make a *diversification* procedure as done in *Step 10*. Else, go to *Step 5*.
Step 5	If $R < 2(L-1)$ then do the following: $R_{ave} = 0.1R + 0.9R_{ave}$ $T(t+1) = \min\{T(t) \cdot INC, L-2\}$ $t_T = t$. Else, follow the next instructions: $\Pi(f^{(t)}) = t$ $\Phi(f^{(t)}) = 1$ Go to *Step 6*.
Step 6	Evaluate $(t - t_T) > R_{ave}$. If this is true then update the value of the tabu tenure $T(t+1) = \max\{T(t) \cdot DEC, 1\}$ and record the iteration of this modification $t_T = t$ and make an *intensification* procedure as done in *Step 7*.
Step 7	With the current configuration $f^{(t)}$ do a modification in one element, so-called $f_i^{(t)}$ running i for all permissible elements of A. If it is permitted, evaluate the energy $E(f_i^{(t)})$.
Step 8	Select the local optimum configuration temporarily known as f_l and its proper energy E_l. Then, the move μ that produces f_l must be stored in the tabu list τ.
Step 9	Compare the temporary energy with respect to the best energy. If the temporary energy is less than the best energy, then $E_b = E_l$ and $f_b = f_l$. Go to *Step 14*.
Step 10	Select another region in the permissible moves and store the elements in a subset $S \subset A$. Clear memory data used to select the intensification/diversification process.
Step 11	With the current configuration $f^{(t)}$ do a modification in one element, the so-called $f_i^{(t)}$ running i for all permissible elements of S. If it is permitted, evaluate the energy $E(f_i^{(t)})$.
Step 12	Select the local optimum configuration temporarily known as f_l and its proper energy E_l. Then, the move μ that produces f_l must be stored in the tabu list τ by a period of T iterations.
Step 13	Compare the temporary energy with respect to the best energy. If temporary energy is less than the best energy, then $E_b = E_l$ and $f_b = f_l$. Go to *Step 14*.
Step 14	Increment the iteration $t = t + 1$. Set $f^{(t+1)} = f_l$ and go to *Step 2* until the stop criterion is fired, then *STOP*.

Table 6.3 Data for Example 6.3, tabu search for fuzzy associated matrices

Left	Center	Right	Motor 1	Motor 2
1	1	1	20	1
1	1	151	1	20
1	1	226	1	17
1	226	151	1	12
1	226	226	1	15
76	1	1	20	1
76	1	76	20	1
76	1	151	1	15
76	226	151	1	15
76	226	226	1	12
151	1	1	20	1
151	1	76	20	1
151	1	151	12	12
151	226	151	1	1
151	226	226	1	4
226	1	1	20	1
226	151	226	3	1
226	226	1	10	1
226	226	151	1	1
226	226	226	1	1

Example 6.5. Tabu search can be implemented in order to optimize the fuzzy associated matrix or the membership functions in fuzzy controllers. Take for example an application on robotics in which we have to optimize four input membership functions. These functions may represent the distance between the robot and some object measured by an ultrasonic sensor. We have three ultrasonic sensors measuring three distinct regions in front of the robot.

These sensors are labeled as *left*, *center* and *right*. Assume that the membership functions have the same shape for all sensors. In addition, we have experimental results in which we find values of each measure at the first three columns and the last two columns are the desired values for moving the wheels. Data is shown in Table 6.3. Use the reactive tabu search to find a good solution for the membership functions. A prototyping of those functions are shown in Fig. 6.20.

Solution. This example is implemented in LabVIEW following the path ICTL $\gg$ Optimizers $\gg$ RTS $\gg$ **Example RTS.vi**. The desired inputs and outputs of the fuzzy controller are already programmed inside this VI. Then, it is not necessary to copy Table 6.2. □

We first explain the front panel. On the top-left, are the control variables. *Max Iters* is the number of times that the algorithm will be reproduced. If this number is exceeded, then the algorithm stops. The next one is *Execution Delay* that refers to a timer delay between iterations. It is just here if we want to visualize the process slowly. The *Cts* cluster has the *INC* value that controls the increment of the tabu tenure when the algorithm is evaluating if it needs an intensification or diversification process.

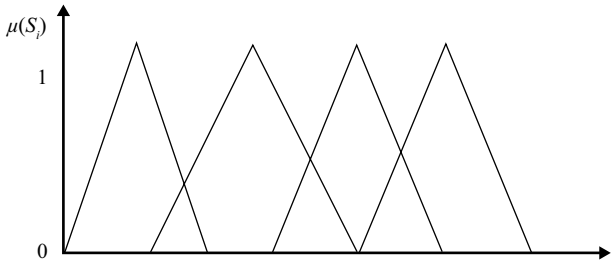


Fig. 6.20 Prototyping of membership functions

DEC is the decrement value of the tabu tenure in the same way as the last one. *CHS* is the chaos value. When the cardinality of the chaos set is greater than this value, then the algorithm makes a diversification process. Finally, the number of repetitions *REP* is known as *REP_MAX* in the algorithm previously described.

The graphs on the left side of the window are *Current f* and *Best f*. The first one shows the actual position of the membership functions. The other one shows the best configuration of the membership functions found thus far. In the middle of the window is all the information used to analyze the procedure. This cluster is called the *Information Cluster* and it is divided into three clusters.

The first is *Arr In* that shows the function f in Boolean terms (zeros and ones). *Pi, Phi* and A are the sets of the time at which some function was evaluated, the number of times that the function was evaluated, and the permissible moves, respectively. The *Cts* is the modified values of the control values.

Finally, on the right side of the window the *Error* graph history is shown. In this case, we have a function that determines the square of the error measured by the difference of the desired outputs and the actual outputs that the fuzzy controller returns with the actual configuration of the membership functions.

We will explain the basic steps of the reactive tabu search. First, we initialize a characterization of the configuration with 32 bits selected randomly. Of course, these bits can only have values of 0 or 1 (Fig. 6.21a). Then, we evaluate this configuration and obtain the best error thus far (Fig. 6.21b). Actually, all other values and sets explained at *Step 1* are initialized, as seen in Fig. 6.22.

Steps 2–6 in Algorithm 6.5 are known as the reaction procedure. These steps are implemented in LabVIEW in the path ICTL ≫ Optimizers ≫ RTS ≫ **rts_mbr.vi**. This VI receives three clusters: *Var In* is the cluster of the initialization values in Fig. 6.22 on the left side, *Arr In* is the cluster of the initialization values in Fig. 6.22 on the right side and *Cts In* is the cluster with the control values (*INC, DEC, CHS, REP*). Finally, it needs the configuration f. In addition, this VI returns all modified values in clusters as *Arr Out* and *Var Out*, and it determines if it needs a diversifying search procedure by the pin *Diversify?* This can be seen in Fig. 6.23.

If a diversification procedure is selected, it is implemented in the VI located at ICTL ≫ Optimizers ≫ RTS ≫ **rts_dvsm.vi**. The input connections are two clusters (*Arr In* and *Var In* explained before) and the actual configuration is known

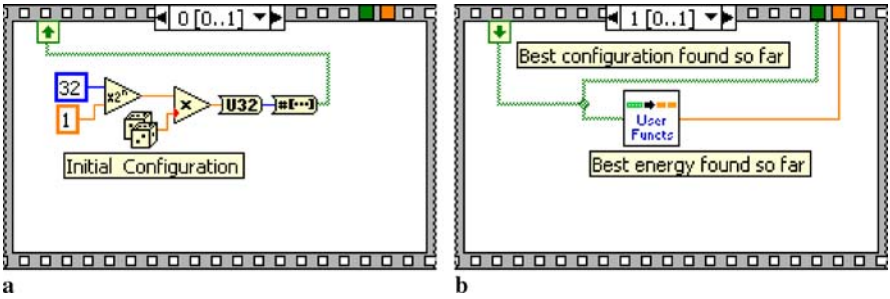


Fig. 6.21a,b Reactive tabu search implementation. **a** Initialization of the configuration with 32 bits. **b** Calculation of best error

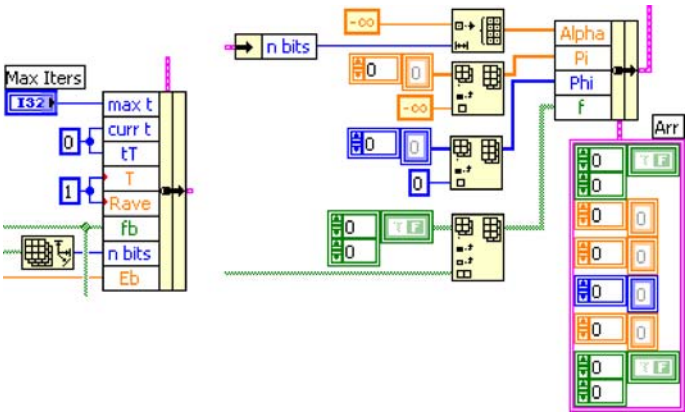


Fig. 6.22 Initialization of the parameters in tabu search

as *current f*. The output connectors are the two clusters updated (*Var Out* and *Arr Out*) and the new configuration by the pin *new f*. Figure 6.24 shows this VI.

Fig. 6.23 Determining connections of the reaction procedure

Fig. 6.24 Determining connections of the diversifying procedure

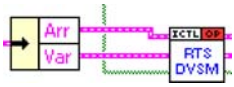
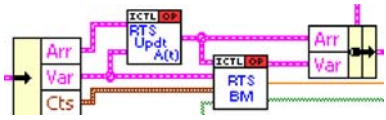


Fig. 6.25 Block diagram of the intensification procedure



If there is no diversifying procedure, then the intensification procedure is running. This can be implemented with two VIs following the path ICTL ≫ Optimizers ≫ RTS ≫ **rts_updt-A.vi**. This VI updates the permissible moves with the information of *Arr In* and *Var In*. Then, *Arr Out* is the update of the values inside this cluster, but in fact the A set updating is the main purpose of this VI. The function then looks for a configuration with this permissible moves and then evaluates the best move with the VI at the path ICTL ≫ Optimizers ≫ RTS ≫ **rts_bm.vi**. This VI takes *Arr In*, *Var In* and the actual configuration *current f*. This returns *Var Out*

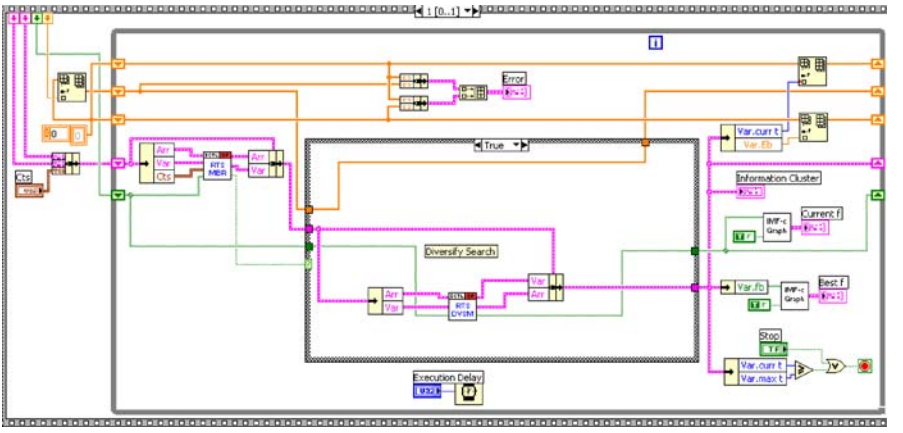


Fig. 6.26 Block diagram of the RTS example

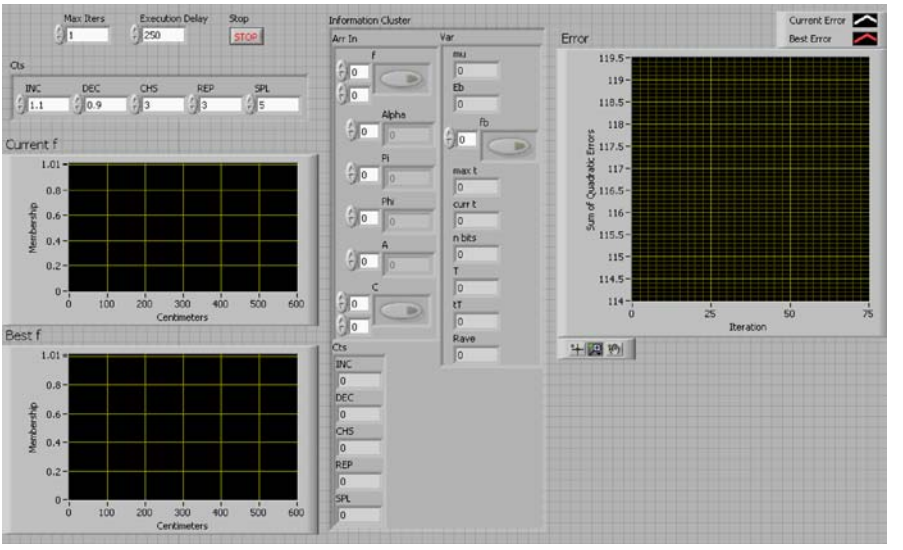


Fig. 6.27 Front panel of the RTS example. This is the initialization step

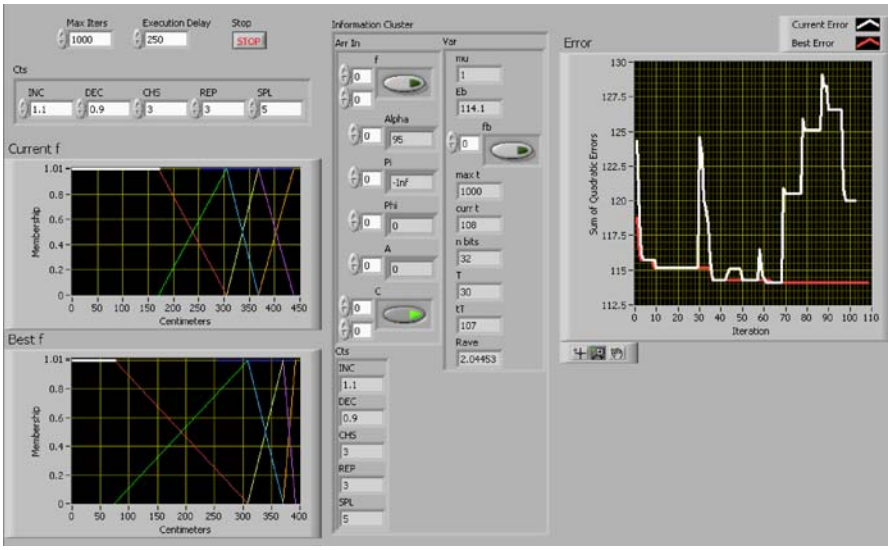


Fig. 6.28 Front panel of the RTS example at 100 iterations

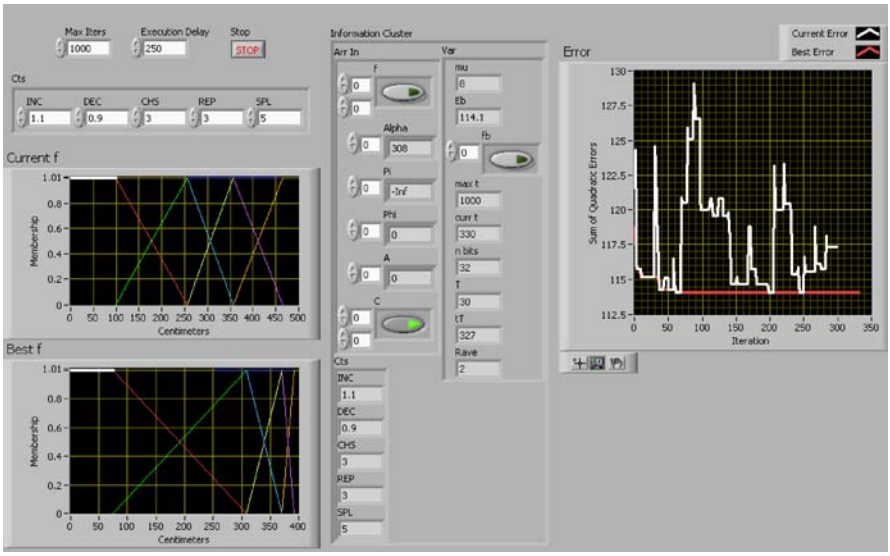


Fig. 6.29 Front panel of the RTS example at 300 iterations

(as a modification procedure of these values), *Energy* that is the energy evaluated at the current configuration, and the *new f* configuration. This block diagram can be viewed in Fig. 6.25.

After either intensification or diversification procedures, we have to choose if the configuration is better than the best configuration found thus far. It is easy to com-

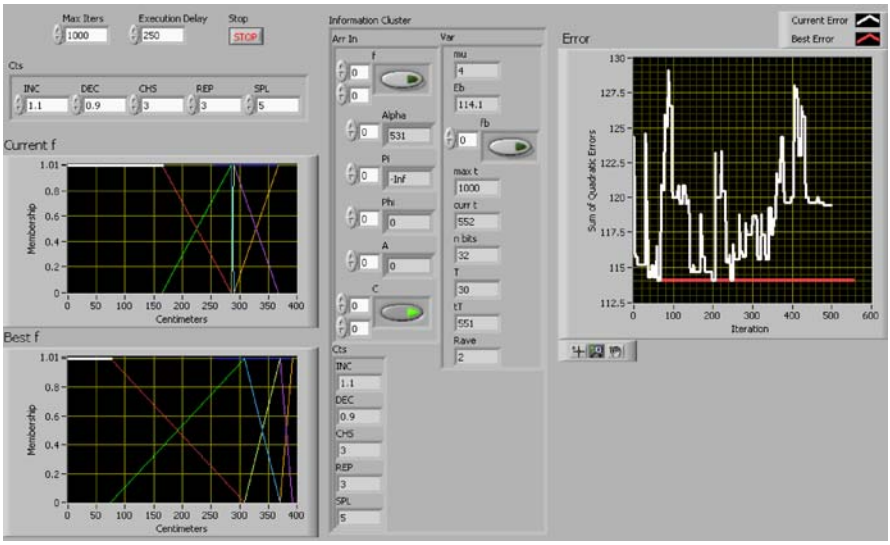


Fig. 6.30 Front panel of the RTS example at 500 iterations

pare the best configuration up to this point that comes from the *Var In* cluster and the actual configuration coming from any of the searching procedures. Figure 6.26 shows a global visualization of the block diagram of this example.

Continuing with the example, let $INC = 1.1$, $DEC = 0.9$, $CHS = 3$ and $REP = 3$. The maximum number of iterations $Max\ Iters = 1000$ and $Execution\ Delay = 250$. Finally, we can look at the behavior of this optimization procedure in Fig. 6.26 (initialization) and Figs. 6.27–6.30. As we can see, at around 80 iterations, the best solution was found.

References

1. Khouja M, Booth DE (1995) Fuzzy clustering procedure for evaluation and selection of industrial robots. J Manuf Syst 14(4):244–251
2. Saika S, et al. (2000) WSSA: a high performance simulated annealing and its application to transistor placement. Special Section on VLSI Design and CAD Algorithms. IEICE Trans Fundam Electron Commun Comput Sci E83-A(12):2584–2591
3. Bailey RN, Garner KM, Hobbs MF (1997) Using simulated annealing and genetic algorithms to solve staff-scheduling problems. Asia Pacific J Oper Res Nov 1 (1997)
4. Baker JA, Henderson D (1976) What is liquid? Understanding the states of matter. Rev Mod Phys 48:587–671
5. Moshiri B, Chaychi S (1998) Identification of a nonlinear industrial process via fuzzy clustering. Lect Notes Comput Sci 1415:768–775
6. Zhang L, Guo S, Zhu Y, Lim A (2005) A tabu search algorithm for the safe transportation of hazardous materials. Symposium on Applied Computing, Proceedings of the 2005 ACM Symposium on Applied Computing, Santa Fe, NM, pp 940–946

7. Shuang H, Liu Y, Yang Y (2007) Taboo search algorithm based ANN model for wind speed prediction. Proceedings of 2nd IEEE Conference on Industrial Electronics and Applications (ICIEA 2007), 23–25 May 2007, pp 2599–2602
8. Brigitte J, Sebbah S (2007) Multi-level tabu search for 3G network dimensioning. Proceedings of IEEE Wireless Communications and Networking Conference (WCNC 2007), 11–15 March 2007, pp 4411–4416
9. Lee CY, Kang HG (2000) Cell planning with capacity expansion in mobile communications: a tabu search approach. IEEE Trans Vehic Technol 49(5):1678–1691
10. Kirkpatrick S, Gelatt CD, Vecchi MP (1983) Optimization by simulated annealing. Science 220:671–680
11. Jones MT (2003) AI application programming. Simulated Annealing. Charles River Media, Boston, pp 14–33

Futher Reading

Aarts E, Korst J (1989) Simulated annealing and Boltzmann machines: a stochastic approach to combinatorial optimization and neural computing. Wiley, New York

Battiti R, Tecchiolli G (1994) The reactive tabu search. ORSA J Comput 6(2):126–140

Dumitrescu D, Lazzerini B, Jain L (2000) Fuzzy sets and their application to clustering and training. CRC, Boca Raton, FL

Glover F, Laguna M (2002) Tabu search. Kluwer Academic, Dordrecht

Klir G, Yuan B (1995) Fuzzy sets and fuzzy logic. Prentice Hall, New York

Reynolds AP, et al. (1997) The application of simulated annealing to an industrial scheduling problem. Proceedings on Genetic Algorithms in Engineering Systems: Innovations and Applications, 2–4 Sept 1997, No 446, pp 345–350

Windham MP (1981) Cluster validity for fuzzy clustering algorithms. Fuzzy Sets Syst 5:177–185