Simulated Bee Colony (SBC) algorithms model the behavior of honey bees and can be used to find solutions to difficult or impossible combinatorial problems. In this article I’ll explain what exactly SBC algorithms are, describe the types of problems that can be solved using SBC algorithms, and present a complete end-to-end example that uses an SBC algorithm to solve the Traveling Salesman Problem.

A good way for you to get a feel for SBC algorithms and see where I’m headed in this article is to examine the demo program shown running in **Figure 1**. The demo program uses an SBC algorithm to analyze a set of 20 cities (labeled A through T) and find the shortest path that visits every city exactly once. The city data is artificially constructed so that the best path starts at city A and continues through city T, in order, and the best path has a length of 19.0 units.

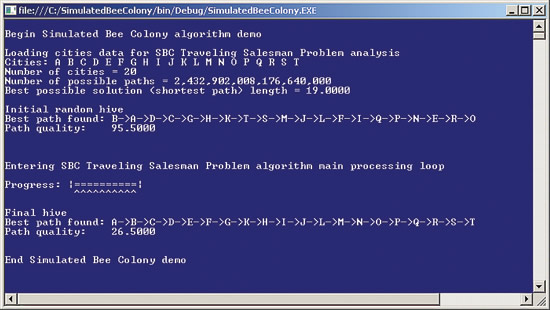


Figure 1 **Simulated Bee Colony Demo**

Behind the scenes the SBC algorithm instantiates a hive of 100 simulated bees, each of which has a random potential solution. Initially, the best of the random solutions has a path length of 95.0 units. The SBC algorithm enters a processing loop, indicated by the text-based progress bar, which simulates the behavior of common honey bees foraging for food. At the end of the SBC processing loop, the best solution found has 16 correct city positions out of 20, and has a path length of 26.5—close to, but not quite, the optimal solution.

SBC algorithms are often called meta-heuristics because they provide a general framework and set of guidelines for creating a problem solution rather than providing a highly detailed solution prescription. This article presents an example of using an SBC to solve a specific problem. Once you understand how an SBC can be used to solve one problem, you can adapt the SBC algorithm presented here to solve your own problems. As this article will demonstrate, SBC algorithms are best suited for solving complex combinatorial problems that have no practical deterministic solutions.

This article assumes you have intermediate-level programming skills. The example in this article is coded using C#, but all the code has been written so that it can be easily refactored to other programming languages. I think you’ll find this article quite interesting and the ability to use SBC algorithms a useful addition to your personal skill set.

## About the Bees

Common honey bees such as Apis mellifera assume different roles within their colony over time. A typical hive may have 5,000 to 20,000 individual bees. Mature bees (20 to 40 days old) usually become foragers. Foraging bees typically occupy one of three roles: active foragers, scout foragers and inactive foragers.

Active foraging bees travel to a food source, examine neighbor food sources, gather food and return to the hive.

Scout bees investigate the area surrounding the hive, often a region of up to 50 square miles, looking for attractive new food sources. Roughly 10 percent of foraging bees in a hive are employed as scouts.

At any given time some of the foraging bees are inactive. These inactive foragers wait near the hive entrance. When active foragers and scouts return to the hive, depending on the quality of the food source they’ve just visited, they may perform a waggle dance to the waiting inactive bees. There’s strong evidence that this waggle dance conveys information to the inactive bees about the location and quality of the food source. Inactive foragers receive this food source information from the waggle dance and may become active foragers.

In general, an active foraging bee continues gathering food from a particular food source until that food source is exhausted, at which time the bee becomes an inactive forager.

## The Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is one of the most widely studied problems in computer science research. There are many variations of the TSP but, informally, the problem is to find the shortest path that visits every city in a given set of cities exactly once.

If you look at **Figure 1** you can see the SBC demo program uses a set of 20 cities that are arbitrarily labeled A through T. A valid path consists of an ordered set of the 20 city labels where each city occurs exactly once. Therefore there are a total of 20! = 2,432,902,008,176,640,000 possible paths.

Behind the scenes there’s a distance value associated with each pair of cities. For simplicity, if city c1 < city c2 the distance between c1 and c2 is just 1.0 times the ordinal distance between the city labels. If c1 > c2 the distance is 1.5 times the ordinal distance between c1 and c2. So the distance from A to B is 1.0 arbitrary units, the distance from B to A is 1.5 units, the distance from A to C is 2.0 units and so on. Therefore, the best path that visits every city exactly once is A -> B-> C -> ... -> T and the best (shortest) path length is (20-1) \* 1.0 = 19.0.

In most TSP scenarios, the distances between cities wouldn’t be artificially computed. Instead, the distances would likely be stored in some sort of lookup data structure. Some variations of the TSP assume that every city is connected to every other city and some problems assume the cities are not fully connected. Additionally, some TSP variations assume that the distance from any city c1 to city c2 is the same as the distance from city c2 to c1, and some problem variations don’t make this bidirectional assumption.

Except in trivial situations, a brute force approach to finding the shortest path is not feasible. For example, with 20 cities, even if you could evaluate 1 million paths per second, examining all 20! possible paths would require more than 77,000 years. This kind of extremely difficult combinatorial optimization problem is the type of problem that SBC algorithms are well suited to handle.

The dummy TSP problem is implemented in a class named CitiesData, shown in **Figure 2**. The complete source code for the SBC demo program is available at [code.msdn.microsoft.com/mag201104BeeColony](http://code.msdn.microsoft.com/mag201104BeeColony).

Figure 2 **The CitiesData Class Definition**

1. **class** CitiesData {
2. **public** **char**[] cities;
4. **public** CitiesData(**int** numberCities) {
5. **this**.cities = **new** **char**[numberCities];
6. **this**.cities[0] = 'A';
7. **for** (**int** i = 1; i < **this**.cities.Length; ++i)
8. **this**.cities[i] = (**char**)(**this**.cities[i - 1] + 1);
9. }
11. **public** **double** Distance(**char** firstCity, **char** secondCity) {
12. **if** (firstCity < secondCity)
13. **return** 1.0 \* ((**int**)secondCity - (**int**)firstCity);
14. **else**
15. **return** 1.5 \* ((**int**)firstCity - (**int**)secondCity);
16. }
18. **public** **double** ShortestPathLength() {
19. **return** 1.0 \* (**this**.cities.Length - 1);
20. }
22. **public** **long** NumberOfPossiblePaths() {
23. **long** n = **this**.cities.Length;
24. **long** answer = 1;
25. **for** (**int** i = 1; i <= n; ++i)
26. **checked** { answer \*= i; }
27. **return** answer;
28. }
30. **public** **override** **string** ToString() {
31. **string** s = "";
32. s += "Cities: ";
33. **for** (**int** i = 0; i < **this**.cities.Length; ++i)
34. s += **this**.cities[i] + " ";
35. **return** s;
36. }
37. }

The definition of some class or data structure that represents your particular problem will be different from the one shown here. However, as a general rule of thumb, problems that are well-suited for solution using an SBC algorithm usually can be represented as a non-numeric array, a non-numeric matrix or a non-numeric jagged array of arrays.

The CitiesData constructor accepts a value for the number of cities and assigns the first city a label of A, the second city a label of B and so on.

The Distance method is defined in an unidirectional way as I previously described and assumes that any city can be reached by any other city.

The ShortestPathLength method returns the optimal path length given the Distance definition. In most SBC cases you won’t have the information necessary to implement a method that returns the optimal solution.

The NumberOfPossiblePaths method just computes n! where n is the number of cities. In some TSP scenarios the number of possible paths is n-1! (if the start city doesn’t matter) and in some scenarios the number of possible paths is n/2! (if the direction of the path doesn’t matter).

The ToString method uses string concatenation rather than the more efficient StringBuilder class to assemble a string with descriptive data. String concatenation is used in order to simplify refactoring to other programming languages.

In this article, to keep the code relatively short and clean, I take shortcuts you wouldn’t use in production code, such as removing most error-checking. For example, method NumberOfPossiblePaths doesn’t deal with numeric overflow if the result is greater than long.MaxValue.

## SBC Program Structure

The SBC algorithm shown in **Figure 1** is implemented as a C# console application. The overall structure of the program is listed in **Figure 3**. Notice that the SBC program structure is relatively simple and uses only the basic System namespace. There are three classes: the Program class, which houses a single Main method; the Hive class, which houses all the SBC algorithm logic; and the CitiesData class presented in the previous section of this article. The Hive class is generically named Hive rather than given a more specific name like TravelingSalesmanHive, even though SBC algorithm implementations are heavily dependent upon the particular problem they’re designed to solve.

Figure 3 **Overall Program Structure**

1. **using** System;
2. **namespace** SimulatedBeeColony {
3. **class** Program {
4. **static** **void** Main(**string**[] args) {
5. Console.WriteLine("\nBegin Simulated Bee Colony algorithm demo\n");
6. . . .
7. Console.WriteLine("End Simulated Bee Colony demo");
8. }
9. }
11. **class** Hive {
12. **public** **class** Bee {
13. . . .
14. }
16. // Hive data fields here
18. **public** **override** **string** ToString() { . . . }
20. **public** Hive(**int** totalNumberBees, **int** numberInactive,
21. **int** numberActive, **int** numberScout, **int** maxNumberVisits,
22. **int** maxNumberCycles, CitiesData citiesData) {
23. . . .
24. }
26. **public** **char**[] GenerateRandomMemoryMatrix() { . . . }
28. **public** **char**[] GenerateNeighborMemoryMatrix(**char**[] memoryMatrix) { . . . }
30. **public** **double** MeasureOfQuality(**char**[] memoryMatrix) { . . . }
32. **public** **void** Solve() { . . . }
34. **private** **void** ProcessActiveBee(**int** i) { . . . }
36. **private** **void** ProcessScoutBee(**int** i) { . . . }
38. **private** **void** ProcessInactiveBee(**int** i) { . . . }
40. **private** **void** DoWaggleDance(**int** i) { . . . }
41. }
43. **class** CitiesData {
44. . . .
45. }
47. } // ns

The Main method is quite simple. After displaying a begin message the CitiesData object is instantiated:

1. Console.WriteLine(
2. "Loading cities data for SBC Traveling Salesman Problem analysis");
3. CitiesData citiesData = **new** CitiesData(20);
4. Console.WriteLine(citiesData.ToString());
5. Console.WriteLine("Number of cities = " + citiesData.cities.Length);
6. Console.WriteLine("Number of possible paths = " +
7. citiesData.NumberOfPossiblePaths().ToString("#,###"));
8. Console.WriteLine("Best possible solution (shortest path) length = " +
9. citiesData.ShortestPathLength().ToString("F4"));

In many SBC scenarios your problem data will reside in external storage such as a text file or a SQL databases, and you’ll load problem data via some load constructor or load method along the lines of myProblemData.Load(dataFile).

Next, the Hive constructor is prepared and called:

1. **int** totalNumberBees = 100;
2. **int** numberInactive = 20;
3. **int** numberActive = 50;
4. **int** numberScout = 30;
5. **int** maxNumberVisits = 100;
6. **int** maxNumberCycles = 3460;
8. Hive hive = **new** TravelingSalesmanHive(totalNumberBees,
9. numberInactive, numberActive, numberScout, maxNumberVisits,
10. maxNumberCycles, citiesData);

As you’ll see in more detail in the following sections of this article, an SBC algorithm uses three types of bees: active, inactive and scout. Although the counts of each of these types of bees can be hard-coded, in most scenarios it’s better to pass these counts in as parameters to the Hive constructor so the algorithm can be more easily tuned for performance.

The value of the totalNumberBees variable could be derived from the other three variables, but the extra variable improves code readability here. The total number of bees will depend on your particular problem. More bees are better but slow the program’s execution. In terms of ratios, there’s some research that suggests the best percentages of active, inactive and scout bees are often roughly 75 percent, 10 percent and 15 percent, respectively.

The 100 value used for the maxNumberVisits variable will be explained shortly, but it’s related to the number of neighbor solutions there are relative to a given solution.

The maxNumberCycles variable is used to control how many times the Solve routine will iterate; this is necessary because in SBC algorithm scenarios you usually don’t know when you’ve reached an optimal solution—if you know the optimal solution you really don’t have a problem to solve. In this case, the value of maxNumberCycles was limited to 3,460 only so that the SBC algorithm did not produce a perfect result. The point here is that although SBC algorithms may produce an optimal result, you usually have no way of knowing this and so you must be willing to accept a “good” result.

When the constructor executes it creates a collection of bees, each of which has a random solution. The Hive object tracks the overall best (shortest) path found by any of the bees in the hive and the best solution’s associated path length.

After calling the Hive constructor, the Main method finishes up by using the ToString method to display the initial, randomly generated Hive values, calling the Solve method with a parameter indicating that a text-based progress bar should be printed, and then displaying the best path and associated path length found:

1. ...
2. Console.WriteLine("\nInitial random hive");
3. Console.WriteLine(hive);
5. **bool** doProgressBar = **true**;
6. hive.Solve(doProgressBar);
8. Console.WriteLine("\nFinal hive");
9. Console.WriteLine(hive);
10. Console.WriteLine("End Simulated Bee Colony demo");
11. }

## The Bee Class

As shown in **Figure 3**, the Hive class contains a nested Bee class definition. The details of the Bee definition are listed in **Figure 4**.

Figure 4 **Bee Class Definition**

1. **public** **class** Bee {
2. **public** **int** status;
3. **public** **char**[] memoryMatrix;
4. **public** **double** measureOfQuality;
5. **public** **int** numberOfVisits;
7. **public** Bee(**int** status, **char**[] memoryMatrix,
8. **double** measureOfQuality, **int** numberOfVisits) {
9. **this**.status = status;
10. **this**.memoryMatrix = **new** **char**[memoryMatrix.Length];
11. Array.Copy(memoryMatrix, **this**.memoryMatrix, memoryMatrix.Length);
12. **this**.measureOfQuality = measureOfQuality;
13. **this**.numberOfVisits = numberOfVisits;
14. }
16. **public** **override** **string** ToString() {
17. **string** s = "";
18. s += "Status = " + **this**.status + "\n";
19. s += " Memory = " + "\n";
20. **for** (**int** i = 0; i < **this**.memoryMatrix.Length-1; ++i)
21. s += **this**.memoryMatrix[i] + "->";
22. s += **this**.memoryMatrix[**this**.memoryMatrix.Length-1] + "\n";
23. s += " Quality = " + **this**.measureOfQuality.ToString("F4");
24. s += " Number visits = " + **this**.numberOfVisits;
25. **return** s;
26. }
27. }

The Bee class has three data fields common to all SBC implementations and one problem-specific data field. The problem-specific field is named memoryMatrix. Every SBC implementation must have some way to represent a solution. In the case of the TSP in this article, a solution can be represented by an array of type char. Let me emphasize that the object that represents a solution is highly problem-dependent and every problem must be analyzed separately to produce a meaningful solution representation.

The field named status holds an int value that indicates the Bee object’s type: 0 for inactive, 1 for active and 2 for scout. If you’re coding in a language that supports enumeration types, you may want to refactor the status field as an enumeration.

The field measureOfQuality holds a double value that represents the goodness of the Bee object’s memoryMatrix. In the case of the TSP, a natural measure of solution quality is the path length represented by the memoryMatrix object. Notice that in this situation, shorter path lengths are better than longer path lengths and so smaller values of the measureOfQuality field represent better solutions than larger values. Every SBC implementation must have some way of computing a measure of solution quality. In many situations this measure is best represented by a value of type double.

The third common data field in the Bee class is named numberOfVisits. This field holds an int value that represents the number of times the Bee object has visited a particular food source/problem solution, without finding a better neighbor solution. This field is used to simulate a bee visiting a food source until that food source is used up. For an active bee, when the value of the numberOfVisits field exceeds a threshold value, the simulated bee will have virtually exhausted the food supply and transition to inactive status (and an inactive bee will transition to active status).

The Bee constructor accepts values for the four data fields, status, memoryMatrix, measureOfQuality, and numberOfVisits. Note that the Bee constructor must accept a value for measureOfQuality because a Bee can’t directly compute this from its memoryMatrix field; determining the measure of quality depends on information stored in the problem-specific CitiesData object.

The Bee class definition contains a ToString method, which exposes the values of the four data fields. The Bee.ToString method is not absolutely necessary but can be quite useful during SBC development to help uncover bugs by using WriteLine statements.

## The Hive Data Fields

The Hive class code is presented in **Figure 5**. There are 14 hive data fields and understanding the purpose of each is the key to understanding how to implement a specific SBC algorithm.

Figure 5 **The 14 Hive Data Fields**

1. **static** Random random = **null**;
3. **public** CitiesData citiesData;
5. **public** **int** totalNumberBees;
6. **public** **int** numberInactive;
7. **public** **int** numberActive;
8. **public** **int** numberScout;
10. **public** **int** maxNumberCycles;
11. **public** **int** maxNumberVisits;
13. **public** **double** probPersuasion = 0.90;
14. **public** **double** probMistake = 0.01;
16. **public** Bee[] bees;
17. **public** **char**[] bestMemoryMatrix;
18. **public** **double** bestMeasureOfQuality;
19. **public** **int**[] indexesOfInactiveBees;

For simplicity and easier debugging with WriteLine statements, all 14 data fields are defined with public scope. You may want to restrict the fields to private scope and create properties for those fields you need to access outside the class definition.

The first field is named random and is type Random. SBC algorithms are probabilistic and the random object is used to generate pseudorandom numbers for several purposes. The random object will be instantiated in the hive constructor.

The second data field is an object of type CitiesData. The SBC implementation needs to know details of the problem being solved. In most cases, such as this one, the problem data is represented as an object. Recall that CitiesData has a list of city labels and a method that returns the distance between any two cities.

The third through sixth data fields are int variables that hold the total number of bees, the number of inactive bees, the number of active bees and the number of scout bees. As mentioned earlier, because each bee represents a potential solution, the more bees in the hive, the better. However, larger numbers of bees degrade program performance.

The seventh data field, maxNumberCycles, is a threshold value used to constrain how long the Solve method runs. One cycle represents processing of each bee in the hive.

The eighth data field, maxNumberVisits, is a threshold value used to prevent a bee from staying too long at a particular solution. In every iteration of the main processing loop in the Solve method, if a bee does not find a neighbor food source with better quality, the bee’s numberOfVisits counter is incremented. If the numberOfVisits counter in a Bee object exceeds the maxNumberVisits threshold value, the bee transitions to an inactive state.

The ninth data field, probPersuasion, is a probabilistic threshold value used to determine whether an inactive bee who observes the waggle dance of a bee that has returned to the hive with a better solution will be persuaded to update its memory with the better solution.

The value of probPersuasion is hard-coded to 0.90, which means that an inactive bee will be persuaded to accept a better solution about 90 percent of the time. The 0.90 value for probPersuasion is based on research findings, but you may want to experiment with other values. Larger values will produce an SBC algorithm which converges to a solution more quickly, at the risk of more likely converging to a non-optimal solution.

The tenth data field, probMistake, is a probabilistic threshold value used to determine whether an active bee will make a mistake—that is, incorrectly reject a neighbor solution that’s better than the bee’s current solution, or incorrectly accept a neighbor solution that’s worse than the bee’s current solution. The value of probMistake is hardcoded to 0.01, which means that an active bee will make a mistake in evaluating a neighbor solution about 1 percent of the time.

The 11th data field, bees, is an array of Bee objects. Recall that each bee has a status (active, inactive, scout), a solution (memoryMatrix), a measure of the solution’s quality (measureOfQuality), and a counter of the number of times a particular virtual food source has been visited without finding a better neighbor food source (numberOfVisits). Because a Bee is defined as a class, each entry in the bees array is a reference to a Bee object.

The 12th data field, bestMemoryMatrix, is an array of char and represents the best solution in the bees array. Recall that because simulated bee colony algorithms are specific implementations of a meta-heuristic, the representation of a problem solution will vary from problem to problem. An alternative design approach to hardcoding a solution type definition is to parameterize this data field as a generic type. When I use an SBC algorithm I’m usually trying to solve a specific problem, so I prefer to recode each new SBC implementation from scratch.

The 13th data field, bestMeasureOfQuality, is the measure of quality that corresponds to the bestMemoryMatrix solution.

The last hive data field, indexesOfInactiveBees, is an array of int. This array holds the indexes of the bees in the hive that are currently inactive. Recall that active bees can transition to an inactive state and inactive bees can transition to an active state. An SBC algorithm implementation must frequently determine which bees are currently inactive when an active bee performs a virtual waggle dance, and storing the indexes of inactive bees improves performance compared to the alternative of iterating through the entire bees array and checking the status data field of each bee.

A visual representation of a possible Hive object is presented in **Figure 6**. The hive shown has 10 bees: 5 active, two scout and three inactive. The currently inactive bees are at indexes 2, 7 and 4 in the bees array. The CitiesData object has five cities. The current best solution is:

**A->B->E->C-D**

This solution has a path length, in distance units, of:

**1.0 + (3 \* 1.0) + (2 \* 1.5) + 1.0 = 8.0**

Note that the citiesData field is a reference to a CitiesData object defined outside of the hive object.

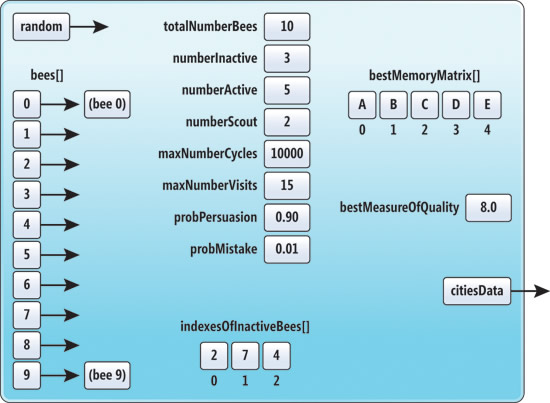


Figure 6 **The Hive Representation**

## The Hive Constructor

The code for the hive constructor is presented in **Figure 7**. The hive constructor accepts seven input parameters. Six of the parameters are scalar and one is an object (citiesData). The totalNumberBees parameter is redundant in the sense that it can be determined from numberInactive, numberActive and numberScout, but I feel the improvement in readability is worth the extra code.

**Figure 7 Hive Constructor**

1. **public** Hive(**int** totalNumberBees, **int** numberInactive,
2. **int** numberActive, **int** numberScout, **int** maxNumberVisits,
3. **int** maxNumberCycles, CitiesData citiesData) {
5. random = **new** Random(0);
7. **this**.totalNumberBees = totalNumberBees;
8. **this**.numberInactive = numberInactive;
9. **this**.numberActive = numberActive;
10. **this**.numberScout = numberScout;
11. **this**.maxNumberVisits = maxNumberVisits;
12. **this**.maxNumberCycles = maxNumberCycles;
14. **this**.citiesData = citiesData;
16. **this**.bees = **new** Bee[totalNumberBees];
17. **this**.bestMemoryMatrix = GenerateRandomMemoryMatrix();
18. **this**.bestMeasureOfQuality =
19. MeasureOfQuality(**this**.bestMemoryMatrix);
21. **this**.indexesOfInactiveBees = **new** **int**[numberInactive];
23. **for** (**int** i = 0; i < totalNumberBees; ++i) {
24. **int** currStatus;
25. **if** (i < numberInactive) {
26. currStatus = 0; // inactive
27. indexesOfInactiveBees[i] = i;
28. }
29. **else** **if** (i < numberInactive + numberScout)
30. currStatus = 2; // scout
31. **else**
32. currStatus = 1; // active
34. **char**[] randomMemoryMatrix = GenerateRandomMemoryMatrix();
35. **double** mq = MeasureOfQuality(randomMemoryMatrix);
36. **int** numberOfVisits = 0;
38. bees[i] = **new** Bee(currStatus,
39. randomMemoryMatrix, mq, numberOfVisits);
41. **if** (bees[i].measureOfQuality < bestMeasureOfQuality) {
42. Array.Copy(bees[i].memoryMatrix, **this**.bestMemoryMatrix,
43. bees[i].memoryMatrix.Length);
44. **this**.bestMeasureOfQuality = bees[i].measureOfQuality;
45. }
46. }
47. }

The class-scope random object is instantiated with a seed value of 0. Supplying a seed value allows you to reproduce results. Next, six input parameter values for the scalar data fields are copied to hive data fields. The hive object has a total of 14 data fields; the threshold values probPersuasion and probMistake are hardcoded.

The Hive constructor takes the input citiesData parameter and assigns the citiesData field to the parameter as a reference. An alternative to this by-reference approach is to make a new copy of the problem data, like so:

1. **int** n = citiesData.cities.Length;
2. **this**.citiesData = **new** CitiesData(n);

This approach uses more memory, but avoids potential side-effect errors. The alternative approach can be used if you’re refactoring the code presented here to a programming language that doesn’t support pointers or references.

At this point in the Hive constructor all entries in the bees array will be null. Next, the constructor initializes the globally best solution (that is, the best solution among all bees in the hive) and corresponding solution quality:

1. **this**.bestMemoryMatrix = GenerateRandomMemoryMatrix();
2. **this**.bestMeasureOfQuality =
3. MeasureOfQuality(**this**.bestMemoryMatrix);

The GenerateRandomMemoryMatrix helper method generates a random solution. The MeasureOfQuality helper method accepts the randomly generated solution and computes its quality. I’ll discuss the code for these two helper methods later in the article.

After initializing the global best solution and its corresponding quality, the hive constructor allocates the indexesOfInactiveBees bees array using the value in the numberInactive field. At this point all values in this indexes array will be 0.

The next part of the constructor code iterates through each Bee object in the bees array and instantiates it using the Bee constructor. The logic in this loop assumes that the first numberInactive cells in the bees array are inactive bees, the next numberScout cells are scout bees and the remaining cells are active bees.

For example, if there are five active bees, two inactive bees and three scout bees, the constructor initializes a bees array of size 10, and instantiates cells 0 and 1 as inactive bees, cells 2-4 as scout bees and cells 5-9 as active bees. Additionally, the indexesOfInactiveBees array will have size 2 and initially hold values 0 and 1.

After the status of the current bee is determined based on the loop index variable, a randomly generated solution is created and its corresponding quality is computed, the number of visits counter is explicitly set to 0, and the Bee constructor is called:

1. **char**[] randomMemoryMatrix = GenerateRandomMemoryMatrix();
2. **double** mq = MeasureOfQuality(randomMemoryMatrix);
3. **int** numberOfVisits = 0;
4. bees[i] = **new** Bee(currStatus, randomMemoryMatrix,
5. mq, numberOfVisits);

After each bee is instantiated, the quality of the bee’s randomly generated solution is checked to see if it’s better than the global best solution. If so, the current bee’s solution and corresponding quality are copied to the bestMemoryMatrix and bestMeasureOfQuality fields. Note in the check for a global best solution quality, a smaller value is better than a larger value because quality value are path lengths and the TSP wishes to minimize the path length.

Instead of explicitly copying a bee’s memory into the bestMemoryMatrix array, an alternative approach is to assign by reference. This approach improves performance at the expense of an increase in complexity.

## Three Essential SBC Methods

Every SBC algorithm implementation must have three problem-specific methods: a method to generate a random solution, a method to generate a neighbor solution relative to a given solution, and a method to compute the quality of a given solution. In this TSP example each method is implemented in a custom and completely problem-dependent way.

A design alternative is to define interfaces and implement these interfaces. Programming via interfaces has several advantages and disadvantages compared to the non-interface approach used, but is mostly a matter of personal preference in my opinion. The method to generate a random solution, GenerateRandomMemoryMatrix, is shown here:

1. **public** **char**[] GenerateRandomMemoryMatrix() {
2. **char**[] result = **new** **char**[**this**.citiesData.cities.Length];
3. Array.Copy(**this**.citiesData.cities, result,
4. **this**.citiesData.cities.Length);
5. **for** (**int** i = 0; i < result.Length; i++) {
6. **int** r = random.Next(i, result.Length);
7. **char** temp = result[r];
8. result[r] = result[i];
9. result[i] = temp;
10. }
11. **return** result;
12. }

Because a solution to the TSP problem is an array of char where each char represents a city label, the GenerateRandomMemoryMatrix method returns a char array. The local result array size is allocated based on the hive’s CitiesData object, and the city IDs stored in the hive’s reference to the CitiesData object are copied into the result array. The order of the values in result array is then randomized using the class scope random object and the Fisher-Yates shuffle algorithm (sometimes called the Knuth shuffle).

At first, it might seem that method GenerateRandomMemoryMatrix conceptually belongs to a Bee object. However, because generating a random solution depends in part on problem-specific data—CitiesData in this case—placing the random solution generation method in the overall hive definition is a better design.

The method to generate a neighbor solution, GenerateNeighborMemoryMatrix, is presented in **Figure 8**.

Figure 8 **Generating a Neighbor Solution**

1. **public** **char**[] GenerateNeighborMemoryMatrix(**char**[] memoryMatrix) {
2. **char**[] result = **new** **char**[memoryMatrix.Length];
3. Array.Copy(memoryMatrix, result, memoryMatrix.Length);
5. **int** ranIndex = random.Next(0, result.Length);
6. **int** adjIndex;
7. **if** (ranIndex == result.Length - 1)
8. adjIndex = 0;
9. **else**
10. adjIndex = ranIndex + 1;
12. **char** tmp = result[ranIndex];
13. result[ranIndex] = result[adjIndex];
14. result[adjIndex] = tmp;  **return** result;
15. }

A key concept in SBC algorithms is the idea that each virtual food source that represents a solution has some sort of neighbor. Without the neighbor concept, the entire idea of an algorithm based on bee behavior collapses. In the case of the TSP, where a solution can be represented as an array of city IDs representing a path from city to city, a natural neighbor solution relative to a current solution is a permutation of the current solution where two adjacent cities have been exchanged.

For example, if a current TSP solution is A,B,C,D,E, then a reasonable neighbor solution is A,C,B,D,E. It’s not so obvious if a permutation where any two arbitrary cities are exchanged (as opposed to two adjacent cities) represents a reasonable neighbor solution. For the previous example, is A,D,C,B,E a reasonable neighbor solution? Deciding on the definition of a neighbor solution for an SBC algorithm is problem-dependent and typically involves subjective criteria.

The neighbor-solution concept also serves to illustrate in part why non-numeric combinatorial problems are especially well suited for solution by SBC algorithms. If a problem is inherently numeric, the idea of a neighbor is often difficult to define satisfactorily. If a problem is inherently combinatorial, the idea of a neighbor can often be nicely defined by some form of mathematical permutation or combination.

The GenerateNeighborMemoryMatrix method accepts a bee’s current memoryMatrix representation of a solution as an input parameter and copies it into a result array. The method selects a single random index into the current result array using the class-scope random object. If the random index points to the last cell then the first and last city IDs are exchanged; otherwise, if the random index points to any non-last cell, the IDs pointed to by the random index and the next index are exchanged.

The neighbor solution concept is related to the maxNumberVisits value. There’s some research that suggests a good value for maxNumberVisits is about five times the number of neighbor solutions possible for any given solution. For example, for three cities (A,B,C), if a neighbor solution is defined as exchanging any pair of adjacent cities, then there are three possible neighbors (exchange A and B, exchange B and C, exchange A and C). So for 20 cities, a reasonable maxNumberVisits value is about 20 \* 5 = 100.

The method to evaluate the quality of a bee’s solution, MeasureOfQuality, is:

1. **public** **double** MeasureOfQuality(**char**[] memoryMatrix) {
2. **double** answer = 0.0;
3. **for** (**int** i = 0; i < memoryMatrix.Length - 1; ++i) {
4. **char** c1 = memoryMatrix[i];
5. **char** c2 = memoryMatrix[i + 1];
6. **double** d = **this**.citiesData.Distance(c1, c2);
7. answer += d;
8. }
9. **return** answer;
10. }

To solve a problem using an SBC algorithm, an essential characteristic of the problem is that any solution must be able to be evaluated to yield a measure of the solution’s quality. In conceptual terms, a real-world combinatorial optimization problem almost always has some inherent and common-sense measure of quality. However, in practical terms, computing the measure of quality of a solution can be difficult and time-consuming.

In this example, the MeasureOfQuality method simply iterates through every pair of consecutive city IDs in the solution represented by parameter memoryMatrix, determines the distance between each pair using the Distance method of the CitiesData object, and accumulates the total distance. Recall that the city data was artificially constructed so that the distance between any two cities could be quickly and easily computed simply by using the ordinal distance between two city IDs. But in a real problem, the distance between two cities would likely have to be looked up in some sort of data structure. In SBC implementations, the MeasureOfQuality method is often the routine that dominates the running time of the program. Therefore, it’s usually worthwhile to make sure that this method is optimized for performance—as well as feasible, given the memory resources of the host system.

## The Solve Method

The Solve method houses all the logic that simulates the behavior of foraging bees to solve a problem. The Solve method is moderately complex and uses three helper methods, ProcessActiveBee, ProcessScoutBee and ProcessInactiveBee. The ProcessActiveBee and ProcessScoutBee methods in turn use a DoWaggleDance helper method. The Solve method is presented in **Figure 9**.

Figure 9 **The Solve Method**

1. **public** **void** Solve(**bool** doProgressBar) {
2. **bool** pb = doProgressBar;
3. **int** numberOfSymbolsToPrint = 10;
4. **int** increment = **this**.maxNumberCycles / numberOfSymbolsToPrint;
5. **if** (pb) Console.WriteLine("\nEntering SBC Traveling Salesman Problem algorithm main processing loop\n");
6. **if** (pb) Console.WriteLine("Progress: |==========|");
7. **if** (pb) Console.Write("           ");
8. **int** cycle = 0;
10. **while** (cycle < **this**.maxNumberCycles) {
11. **for** (**int** i = 0; i < totalNumberBees; ++i) {
12. **if** (**this**.bees[i].status == 1)
13. ProcessActiveBee(i);
14. **else** **if** (**this**.bees[i].status == 2)
15. ProcessScoutBee(i);
16. **else** **if** (**this**.bees[i].status == 0)
17. ProcessInactiveBee(i);
18. }
19. ++cycle;
21. **if** (pb && cycle % increment == 0)
22. Console.Write("^");
23. }
25. **if** (pb) Console.WriteLine("");
26. }

Most of the actual work is farmed out to helper methods ProcessActiveBee, ProcessScoutBee and ProcessInactiveBee. A Boolean input parameter is passed to Solve to indicate whether to print a rudimentary text-based progress bar. This is useful when developing an SBC algorithm to monitor the speed of the implementation and help uncover performance bottlenecks. This approach makes the assumption that the Solve method is part of a console application.

The value of the Boolean parameter value is transferred into a local Boolean variable named pb just to have a short variable name to work with. The numberOfSymbolsToPrint is set to 10 so that each increment in the status bar will represent 10 percent of the total progress, which is determined by the maxNumberCycles value (the increment variable is used to determine how many cycles represent 10 percent progress).

After the loop control variable, cycle, is initialized to 0, a while loop is used to process each bee in the hive. A for loop could just as easily be used. On each cycle, the bees array is iterated using a for loop and each Bee object is processed by the appropriate helper method. After each bee has been processed, if the Boolean doProgressBar parameter is true, the code uses the modulus operator, %, to check if it’s time to print an update to the progress bar using a ^ character.

## The ProcessActiveBee Method

The ProcessActiveBee method is the heart of an SBC algorithm and is the most complex method in terms of code size and branching. The ProcessActiveBee method is presented in **Figure 10**.

Figure 10 **The ProcessActiveBee Method**

1. **private** **void** ProcessActiveBee(**int** i) {
2. **char**[] neighbor = GenerateNeighborMemoryMatrix(bees[i].memoryMatrix);
3. **double** neighborQuality = MeasureOfQuality(neighbor);
4. **double** prob = random.NextDouble();
5. **bool** memoryWasUpdated = **false**;
6. **bool** numberOfVisitsOverLimit = **false**;
8. **if** (neighborQuality < bees[i].measureOfQuality) { // better
9. **if** (prob < probMistake) { // mistake
10. ++bees[i].numberOfVisits;
11. **if** (bees[i].numberOfVisits > maxNumberVisits)
12. numberOfVisitsOverLimit = **true**;
13. }
14. **else** { // No mistake
15. Array.Copy(neighbor, bees[i].memoryMatrix, neighbor.Length);
16. bees[i].measureOfQuality = neighborQuality;
17. bees[i].numberOfVisits = 0;
18. memoryWasUpdated = **true**;
19. }
20. }
21. **else** { // Did not find better neighbor
22. **if** (prob < probMistake) { // Mistake
23. Array.Copy(neighbor, bees[i].memoryMatrix, neighbor.Length);
24. bees[i].measureOfQuality = neighborQuality;
25. bees[i].numberOfVisits = 0;
26. memoryWasUpdated = **true**;
27. }
28. **else** { // No mistake
29. ++bees[i].numberOfVisits;
30. **if** (bees[i].numberOfVisits > maxNumberVisits)
31. numberOfVisitsOverLimit = **true**;
32. }
33. }
35. **if** (numberOfVisitsOverLimit == **true**) {
36. bees[i].status = 0;
37. bees[i].numberOfVisits = 0;
38. **int** x = random.Next(numberInactive);
39. bees[indexesOfInactiveBees[x]].status = 1;
40. indexesOfInactiveBees[x] = i;
41. }
42. **else** **if** (memoryWasUpdated == **true**) {
43. **if** (bees[i].measureOfQuality < **this**.bestMeasureOfQuality) {
44. Array.Copy(bees[i].memoryMatrix, **this**.bestMemoryMatrix,
45. bees[i].memoryMatrix.Length);
46. **this**.bestMeasureOfQuality = bees[i].measureOfQuality
47. }
48. DoWaggleDance(i);
49. }
50. **else** {
51. **return**;
52. }
53. }

The ProcessActiveBee method accepts a single input parameter, i, which is the index of the bee in the bees array. The active bee first obtains a neighbor solution relative to its current solution stored in memoryMatrix, and then determines the quality of that neighbor:

1. **char**[] neighbor =
2. GenerateNeighborMemoryMatrix(bees[i].memoryMatrix);
3. **double** neighborQuality = MeasureOfQuality(neighbor);

Next, the algorithm sets up three local variables that will be used later:

1. **double** prob = random.NextDouble();)
2. **bool** memoryWasUpdated = **false**;
3. **bool** numberOfVisitsOverLimit = **false**;

The prob variable has a value between 0.0 and 1.0 and will be compared against the probMistake field value to determine if the bee makes a mistake in evaluating the neighbor solution—that is, rejects a better neighbor solution or accepts a worse neighbor solution.

The Boolean memoryWasUpdated value will be used to determine if the active bee should perform a waggle dance to the inactive bees (if true) or not (if false). The Boolean numberOfVisitsOverLimit will be compared against the maxNumberVisits field to determine if the bee has exhausted a particular food source without finding a better neighbor solution, and if so should convert from active status to inactive status.

If the current bee finds a better neighbor solution, the algorithm determines if the bee makes a mistake and rejects the better neighbor or if the bee accepts the better neighbor. Similarly, if the current bee did not find a better neighbor solution, the algorithm determines whether the bee makes a mistake and accepts the worse neighbor solution or does not make a mistake and rejects the neighbor.

Notice that there are two different types of mistakes possible, but that both types of mistakes have the same probability, probMistake = 0.01. There’s some research that suggests using two different probabilities for the two different types of mistakes does not improve the effectiveness of SBC algorithms, but you may want to experiment with two different threshold values.

After the current active bee accepts or rejects the neighbor solution, the algorithm checks if the number of visits counter has exceeded the maxNumberVisits threshold. If so, the current bee’s status is converted to inactive, a randomly selected inactive bee is converted to active status and the indexesOfInactiveBees array is updated. Next, the algorithm checks to see if the bee’s memory was updated. If so, the new solution is checked to see if it’s a new global best solution, and then a private helper method, DoWaggleDance, is called to simulate the bee returning to the hive and conveying information about the new food source to the inactive bees.

## The DoWaggleDance Method

The DoWaggleDance helper method simulates an active or scout bee returning to the hive and then performing a waggle dance to inactive bees in order to convey information about the location and quality of a food source. Here’s the DoWaggleDance method:

1. **private** **void** DoWaggleDance(**int** i) {
2. **for** (**int** ii = 0; ii < numberInactive; ++ii) {
3. **int** b = indexesOfInactiveBees[ii];
4. **if** (bees[i].measureOfQuality < bees[b].measureOfQuality) {
5. **double** p = random.NextDouble();
6. **if** (**this**.probPersuasion > p) {
7. Array.Copy(bees[i].memoryMatrix, bees[b].memoryMatrix,
8. bees[i].memoryMatrix.Length);
9. bees[b].measureOfQuality = bees[i].measureOfQuality;
10. }
11. }
12. }
13. }

The input parameter i is the index of the current bee performing the virtual waggle dance. The measure of quality of the current bee’s solution is compared against the measure of quality of each inactive bee. If the current bee’s solution is better and the current inactive bee is persuaded (with probability probPersuasion = 0.90), the current bee’s memory is copied to the inactive bee’s memory.

Note that there are many opportunities to insert error checking into the code presented in this article. For example, inside the for loop in DoWaggleDance, you may want to check the current inactive bee’s status with:

1. **if** (bees[b].status != 0) **throw** **new** Exception(. . .);

Or you may want to verify the inactive bee’s number of visits counter with:

1. **if** (bees[b].numberOfVisits != 0) **throw** **new** Exception(. . .);

## ProcessScoutBee and ProcessInactiveBee

The ProcessScoutBee helper method used by the Solve method simulates the action of a scout bee randomly searching for appealing food sources. The ProcessScoutBee method is presented in **Figure 11**.

Figure 11 **The ProcessScoutBee Method**

1. **private** **void** ProcessScoutBee(**int** i) {
2. **char**[] randomFoodSource = GenerateRandomMemoryMatrix();
3. **double** randomFoodSourceQuality =
4. MeasureOfQuality(randomFoodSource);
5. **if** (randomFoodSourceQuality < bees[i].measureOfQuality {
6. Array.Copy(randomFoodSource, bees[i].memoryMatrix,
7. randomFoodSource.Length);
8. bees[i].measureOfQuality = randomFoodSourceQuality;
9. **if** (bees[i].measureOfQuality < bestMeasureOfQuality) {
10. Array.Copy(bees[i].memoryMatrix, **this**.bestMemoryMatrix,
11. bees[i].memoryMatrix.Length);
12. **this**.bestMeasureOfQuality = bees[i].measureOfQuality;
13. }
14. DoWaggleDance(i);
15. }
16. }

The input parameter i represents the index of a scout bee in the bees array. A scout bee generates a random solution, checks if the random solution is better than the current solution in memory, and, if so, copies the random solution into memory. Recall that smaller quality values are better. If the scout bee has found a better solution, the algorithm checks to see if the new solution is a global best solution.

Note that unlike active bees, in this SBC implementation scout bees never make mistakes evaluating the quality of a food source. There’s no current research on the effect of scout bee mistakes.

The ProcessInactiveBee method is:

1. `**private** **void** ProcessInactiveBee(**int** i) {
2. **return**;
3. }
4. **private** **void** ProcessInactiveBee(**int** i) {
5. **return**;
6. }

In this SBC implementation inactive bees are exactly that—inactive—so the ProcessInactiveBee method is merely a placeholder in case you wish to implement some problem-dependent logic for an inactive bee. One possible modification would be to randomly mutate an inactive bee’s memory with some very small probability.

## Wrapping Up

The overall process of implementing an SBC algorithm starts with problem identification. Complex, non-numerical, combinatorial optimization problems with no practical deterministic solutions are often good candidates for an SBC solution. The target problem must have a way to represent a solution (often as an array or matrix) and each solution must have some sort of a neighbor solution and a measure of solution quality.

For example, consider the problem of dividing a graph into two parts so that the number of connections within each part is maximized and the number of connections between the two parts is minimized. This graph partition problem is combinatorial and there’s no quick algorithm that finds the optimal solution (although there are deterministic algorithms that are good at finding near-optimal solutions). There are many other NP-complete and NP-hard problems that might be tackled using an SBC.

SBC algorithms are based on the behavior of natural systems. There are other such algorithms, including genetic algorithms (GA) based on natural selection and evolution, ant colony optimization (ACO) based on the behavior of ants, and simulated annealing (SA) based on the physical properties of cooling metals.

Algorithms based on natural systems are often easy to implement relative to deterministic approaches. However, algorithms based on natural systems typically require the specification of values for several parameters that tend to be sensitive with regard to their effect on solution convergence speed and solution accuracy. In the case of an SBC, sensitive parameters that must be fine-tuned for each problem include the number of each type of bee, the maximum number of visits before a food source is exhausted, inactive bee persuasion probability threshold and active bee mistake probability.

Although SBC algorithms aren’t applicable to every problem, in some scenarios they can be extremely powerful tools.