Chapter 10
Biogeography-Based Optimization for Large Scale Combinatorial Problems

Dawei Du
Cleveland State University, USA

Dan Simon
Cleveland State University, USA

ABSTRACT

Biogeography-based optimization (BBO) is a recently-developed heuristic algorithm that has shown impressive performance and efficiency over many standard benchmarks. The application of BBO is still limited because it was only developed four years ago. The objective of this chapter is to expand the application of BBO to large scale combinatorial problems. This chapter addresses the solution of combinatorial problems based on BBO combined with five techniques: (1) nearest neighbor algorithm (NNA), (2) crossover methods designed for traveling salesman problems (TSPs), (3) local optimization methods, (4) greedy methods, and (5) density-based spatial clustering of applications with noise (DBSCAN). This chapter also provides a discussion about the advantages and disadvantages for each of these five techniques when used with BBO, and describes the construction of a combinatorial solver based on BBO. In the end, a framework is proposed for large scale combinatorial problems based on hybrid BBO. Based on four benchmark problems, the experimental results demonstrate the quality and efficiency of our framework. On average, the algorithm reduces costs by over 69% for a 2152-city TSP compared to other methods: genetic algorithm (GA), ant colony optimization (ACO), nearest neighbor algorithm (NNA), and simulated annealing (SA). Convergence time for the algorithm is only 28.56 sec on a 1.73-GHz quad core PC with 6 GB of RAM. The algorithm also demonstrated good results for small and medium sized problems such as alysses16 (16-city TSP, where we obtained the best performance), st70 (70-city TSP, where the second best performance was obtained), and rat575 (575-city TSP, where the second best performance was obtained).

DOI: 10.4018/978-1-4666-3942-3.ch010
INTRODUCTION

Heuristic algorithms are well known for their robustness and easy application. With the sustainable development of modern computer hardware, the long computation time is no longer a critical bottleneck for heuristic algorithms. In contrast, researchers benefit from using heuristic algorithms because it is not necessary to have a good understanding of the problem’s structure. This is extremely helpful for industries which may have very complex systems.

Biogeography-based optimization (BBO) was first introduced in 2008 (Simon) making it a relatively young algorithm compared to others. But the performance of BBO on benchmark problems is better than many classical algorithms which have had many years of development. In BBO, the population is analogous to an archipelago, and each island in this archipelago is a possible solution to the optimization problem. From here on we refer to candidate solutions as solutions or individuals. The implementation of the algorithm is based on the following four terms - habitat suitability index (HSI), suitability index variable (SIV), immigration rate, and emigration rate. The HSI represents the goodness of the island, where a high HSI means that the solution represented by the island has relatively good performance on the optimization problem, and a low HSI means poor performance. Each SIV represents a solution feature (that is, an independent variable of an optimization solution) in an island. The immigration rate and emigration rate are important solution characteristics for migration, and are the features of BBO that distinguish it from other evolutionary algorithms. A high performing island has a high emigration rate and low immigration rate. Conversely, a low performing island has a low emigration rate and high immigration rate. The emigration rate indicates how likely a solution is to share its features with other solutions. The immigration rate indicates how likely a solution is to accept features from other solutions. For BBO, the method to create the next generation is to share the individuals’ information with other individuals in the population. In BBO, this information sharing is called immigration and emigration, which involves updating the population by migrations between islands. The basic procedure of BBO is as follows in Box 1.

Combinatorial problems are confirmed as NP-hard problems, and their huge search space determines their incompatibility with traditional mathematic methods. This makes them a perfect benchmark for heuristic algorithms. For the demonstration and simulation purposes of this chapter, the traveling salesman problem (TSP) will be used. Assume there exists a 100-city TSP. The total number of candidate solutions are 100! =

**Box 1.**

For each solution \( Hi \) in the population

For each SIV in the solution

Select solution \( Hi \) for immigration with probability proportional to immigration rate

If \( Hi \) is selected for immigration

Select \( Hj \) for emigration with probability proportional to emigration rate

Randomly select an SIV \( a \) from \( Hj \)

Replace a random SIV \( b \) in \( Hi \) with SIV \( a \)

end

end
9.3326×10^{27}. Therefore new methods must be developed for TSPs without using exhaustive search methods.

The reason we abandon exhaustive search is its high computational expense. But even with heuristic algorithms, the TSP is still very time consuming, and computational expense is still the top concern. Two general directions are proposed to increase the efficiency of BBO to decrease its computation time.

First is the modification of BBO algorithm. Four types of techniques are added to BBO: nearest neighbor algorithm, crossovers designed for TSPs, local optimization methods, and greedy methods. The idea is to search for the best combination of techniques to create a hybrid BBO in order to achieve the best balance between its computation time and performance.

The second direction modifies the problems by proposing a new framework for combinatorial problems. We will generate a framework based on a clustering algorithm, nearest neighbor algorithm, and parallel computing. The goal here is the same as in the previous goal, which is to achieve the best balance between computation time and performance for combinatorial problems.

The organization of this chapter is as follows. In the second section, we introduce the background of combinatorial problems and heuristic algorithms. In the third section, we propose all the modifications for BBO. The fourth section talk about increasing the efficiency of modified BBO for large scale problems. In the fifth section, we simulate all the proposed techniques to test the performance of the modified BBOs. Finally the results and topics covered are summarized and ideas for future efforts are presented.

**BACKGROUND**

Combinatorial problems are not new to heuristic algorithms. As a matter of fact, they are considered as standard benchmarks for heuristic algorithms. For example, TSP, a famous combinatorial problem, is an ancient problem whose origins have been lost in the mists of history. But we know that the TSP was first formulated as a mathematical problem by Karl Menger in 1930 (Mitchell, 1998). There are three major reasons that the TSP has become a standard benchmark for heuristic algorithms. First, the TSP is an easily stated problem and it is similar to many practical problems such as sensor selection (Boilot, 2003), the mailman problem (Desrochers, 1990), robotic path planning (Lozovyy, 2011), and many others. Second, the TSP can easily be modified to become a multi-objective problem (Jaszkiwicz, 2002) and solving multi-objective problems is a practical challenge in many areas of engineering and industry. Third, the optimal TSP solution is extremely hard to find using analytical methods. Even using numerical methods, it is still quite a challenge.

Because of the reasons above, many TSPs are formed to challenge the performance of existing heuristic algorithms, and many new heuristic algorithms are specially formed to conquer TSPs. Almost all the famous algorithms have already been tested with TSP benchmarks. For example, the genetic algorithm (GA) is the most famous and widely used heuristic algorithm. P.W. Poon (1995) invented cycle crossover for GAs which allows GAs to solve the TSP. Ant colony optimization is another widely used heuristic algorithm which has also been tested and confirmed as a good TSP solver by M. Dorigo (1997). Simulated annealing, another well known heuristic algorithm, also uses the TSP as a benchmark for its performance test in (Aarts, 1989).

But among all the algorithms, which one is the most powerful algorithm in this area is always a question. But there is no conclusive answer because of the diversity of TSPs. No one can achieve full domination in this area. However, in a general sense one might still outperform the other. As
mentioned in the introduction, the performance of BBO has been tested in (Simon, 2008). In that paper, the performance of BBO was tested against seven popular evolutionary algorithms on 14 benchmarks. Based on the simulation results, BBO outperformed most of the algorithms. This gives us the confidence to apply it to combinatorial problems.

In (Mo, 2010), BBO has already been applied to TSPs. The new algorithm is called biogeography migration algorithm for traveling salesman problem (TSPBMA) which is a specially modified version of BBO for combinatorial problems. BBO is tested on four TSP benchmark problems against five popular algorithms: ant colony optimization (ACO), genetic algorithm (GA), immune algorithm (IA), fish swarm (FS), and particle swarm optimization (PSO). Based on the simulation results in Table 1, BBO provides promising results. This gives us reason to continue the development of BBO for combinatorial problems.

In 2011, M. Ergezer and D. Simon published a paper regarding BBO application to combinatorial problems (Ergezer, 2011). BBO with Circular Opposition (BBO/CO) was introduced as a modified version of BBO which can achieve promising results. Two techniques are implemented to create BBO/CO: CW, which is clockwise circular opposition; and combinatorial BBO migration, which is also called the simple version of inver-over crossover that will be detailed in the following sections. The performance of BBO/CO compared with original BBO was tested based on 16 benchmarks which are shown in Table 2. Based on the performance comparison, it indicates that appropriate modifications of BBO can have a positive impact.

Although BBO has already been modified for combinatorial problems, the modifications have been relatively simple. For example, in Mo (2010), only the migration component has been modified. Cycle crossover is used to replace the original migration method in order to process combinatorial problems. In Ergezer (2011), the migration component becomes inver-over crossover combined with circular opposition, which is the only changed compared to the original BBO. But the modification should not be restricted to just one component. In this chapter, modification of migration is just a small part of our work. In order to achieve the best solution, we modify all three major components of BBO: population initialization, migration, and mutation, and we explore their modifications with multiple methods. Also, we propose a technique to significantly increase the efficiency of BBO for combinatorial problems.

Table 1. TSP cost values achieved by various evolutionary algorithms (Mo, 2010). The best performance for each benchmark is shown in bold font.

<table>
<thead>
<tr>
<th></th>
<th>TSPBMA</th>
<th>ACO</th>
<th>GA</th>
<th>IA</th>
<th>FS</th>
<th>PSO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Otter30 Problem</td>
<td>420</td>
<td>420</td>
<td>425</td>
<td>442</td>
<td>430</td>
<td>520</td>
</tr>
<tr>
<td>Eil50 Problem</td>
<td>425</td>
<td>424</td>
<td>428</td>
<td>464</td>
<td>451</td>
<td>554</td>
</tr>
<tr>
<td>Eil75 Problem</td>
<td>535</td>
<td>535</td>
<td>545</td>
<td>583</td>
<td>572</td>
<td>684</td>
</tr>
<tr>
<td>Kna100 Problem</td>
<td>21282</td>
<td>21282</td>
<td>21761</td>
<td>22435</td>
<td>22067</td>
<td>66635</td>
</tr>
</tbody>
</table>
Table 2. Mean of the best solutions obtained by BBO and BBO/CO on TSP benchmarks (Ergezer, 2011)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>BBO</th>
<th>BBO/CO</th>
</tr>
</thead>
<tbody>
<tr>
<td>an532</td>
<td>1,154,304</td>
<td>1,140,103</td>
</tr>
<tr>
<td>berlin52</td>
<td>9,795</td>
<td>9,811</td>
</tr>
<tr>
<td>bkc127</td>
<td>302,056</td>
<td>298,700</td>
</tr>
<tr>
<td>ch130</td>
<td>20,552</td>
<td>20,304</td>
</tr>
<tr>
<td>d15512</td>
<td>58,521,418</td>
<td>58,369,040</td>
</tr>
<tr>
<td>kroA150</td>
<td>109,793</td>
<td>108,651</td>
</tr>
<tr>
<td>kroA200</td>
<td>169,256</td>
<td>165,191</td>
</tr>
<tr>
<td>kroC100</td>
<td>57,509</td>
<td>57,799</td>
</tr>
<tr>
<td>lns105</td>
<td>42,005</td>
<td>41,661</td>
</tr>
<tr>
<td>lns318</td>
<td>375,896</td>
<td>374,011</td>
</tr>
<tr>
<td>p654</td>
<td>1,440,864</td>
<td>1,422,779</td>
</tr>
<tr>
<td>rat575</td>
<td>83,835</td>
<td>82,699</td>
</tr>
<tr>
<td>rl11849</td>
<td>85,134,513</td>
<td>84,926,068</td>
</tr>
<tr>
<td>sl70</td>
<td>1,162</td>
<td>1,147</td>
</tr>
<tr>
<td>us13509</td>
<td>2,105,421,221</td>
<td>2,098,349,568</td>
</tr>
<tr>
<td>vm1084</td>
<td>7,208,117</td>
<td>7,142,633</td>
</tr>
</tbody>
</table>

**Population Initialization of BBO**

Population initialization is usually the first step for all heuristic algorithms. But for most of the problems to which we apply heuristic algorithms, we do not have a good understanding of the effect of each independent variable. That means we do not know how to create a good population based on our expertise, so we randomly create it. It is no doubt the simplest way for population initialization. Lacking expertise, it is also the most inefficient way to create a population. But random population initialization is not the only way for population initialization. For certain problems like the TSP, there exist certain ways of creating an initial population which can provide a great benefit to the algorithm.

For TSPs, the most commonly used technique is called nearest neighbor algorithm (NNA) (Cover & Hart, 1967). The detailed procedure is as follows.

1. Randomly select a city as the ending point of the trip, which is also the starting point.
2. Calculate the distance between the ending point city and the cities which are not included in the trip.
3. Based on the distances calculated in step 2, find the nearest city to the ending point city. Link them and name the most recently added city as the ending point city.
4. If all the cities are included in the trip, terminate; otherwise, go to step 2.

The procedure is fairly easy to operate, and clearly not time consuming even for a large scale problem. The most time consuming part is the calculation of Euclidean distances between cities. For a TSP with \( n \) cities, the total number of calculations of Euclidean distance is

\[
\text{number of calculations} = \frac{n(n - 1)}{2}
\]
For a 1000-city problem, the total number of calculations is only 499,500 which is an acceptable number when considering the problem size.

Migration of BBO

Migration is the method to combine features or modify features based on parent individuals to create offspring. It is also the most important component in BBO. But as we know, combinatorial problems are coded differently compared to other types of problems. Each element in the individual contains no information, but the order of the elements in that individual is what contains information. Since the original BBO is coded for continuous problems rather than combinatorial problems, in order to migrate order information more efficiently and validly between individuals, we need new types of migrations. Three types of migration methods are discussed in this book chapter - matrix crossover, cycle crossover, and inver-over crossover.

Matrix Crossover

Matrix crossover is introduced by Fox and McMahon (1991). The advantage of matrix crossover is that the application is very straightforward, and easy to operate for any TSP. Based on this method, the offspring can inherit information from both parents, and also generate random information to create a child not identical to either parent. But the drawback is also obvious for matrix crossover. As we see from the name of the method, all ordering information is represented by matrices, which requires heavy calculations for transformation between the standard TSP array expression and the matrix expression. Matrix crossover is thus contrary to our goal of reducing the total computation time for BBO.

The detailed procedure of matrix crossover is as follows.

• First, for an $n$-city problem, we need to convert the ordering information of all individuals to an $n$ by $n$ matrix. Each row in the matrix expression provides the position information of a city in the trip. For example, the $k$-th row represents the position information of city $k$. Each column in each row represents a certain city. The number in each column represents the ordering relationship between the column city and row city. For example, if city $g$ is before city $k$, the number in the $g$-th column in the $k$-th row is 1. In other words, if city $g$ is after city $k$, the number in the $g$-th column in the $k$-th row is 0. Based on this method, we convert all the individuals in the population (that is, all candidate solutions) to the matrix expression.

• Second, based on the selection methods - for example, roulette wheel - we select individuals to perform migration. Once the parents are selected, we perform AND logic on two matrices and we obtain one child matrix.

• Third, we will find that the child matrix is incomplete; that is, it does not completely represent a TSP tour. In this next step, we randomly fill in necessary information to create a valid child.

• In the last step, we transform the child from matrix expression to sequential representation.

An example is provided in Figure 1 to illustrate how to apply matrix crossover.

Cycle Crossover

Cycle crossover has already been tested in Oliver (1987) and obtained superior performance against competitors. It also achieves satisfying results in Mo (2010). Cycle crossover is first introduced in
Figure 1. Example of matrix crossover with a 5-city TSP

Oliver (1987). The application of cycle crossover is fairly easy. In contrast to matrix crossover, no expression transformation is needed, and it guarantees that every child generated is valid and complete. That is also the reason cycle crossover has been widely used for combinatorial problems.

The basic procedure of cycle crossover is as follows.

1. We randomly select a city as the starting point in parent 1, and record its position.
2. In parent 2, find the city at the position we recorded in parent 1 and then record this city. Go back to parent 1, search for the city we found in parent 2 and then record its position in parent 1.
3. Repeat step 2 until we obtain a closed cycle, which means we have returned to the starting city. Then we copy the cities from the closed cycle in parent 2, and the cities that are not in the closed cycle in parent 1, to obtain child 1. Similarly, we copy the cities from the closed cycle in parent 1, and the cities that are not in the closed cycle in parent 2, to obtain child 2.

We provide an example in Figure 2 to illustrate the application of cycle crossover.
**Biogeography-Based Optimization for Large Scale Combinatorial Problems**

Figure 2. Example of cycle crossover with a 9-city TSP

Inver-Over Crossover

The third migration method is called inver-over crossover. It was invented by G. Tao and Z. Michalewicz in 1998 (Tao, 1998). According to its experimental performance, it is a powerful tool for the TSP. Like cycle crossover, all the children generated by inver-over crossover are valid and complete. Inver-over crossover does not require any additional expression transformation.

The basic procedure of inver-over crossover is as follows.

1. Two parents are used to generate a child. Randomly select a city in parent 1 as the starting point, city s.
2. Find s in parent 2 and choose the city next to it as the ending point, city t. Then find this ending point city in parent 1.
3. Reverse the cities between s+1 (the city next to the starting point city) and t in parent 1. That is the child created by inver-over crossover.

An example is provided in Figure 3 to illustrate how to operate inver-over crossover.

**Let's Talk about Local Optimization**

Combinatorial problems have some special characteristics that are different from continuous or other discrete benchmarks. For example, candidate solutions for most benchmarks are composed of variables and each variable has its own domain. In that case, heuristic algorithms need to search each variable in each domain for the optimal solution. But combinatorial problems are different. In the TSP, for example, the coordinates of each

Figure 3. Example of inver-over crossover with a 5-city TSP
Biogeography-Based Optimization for Large Scale Combinatorial Problems

city are fixed. The task of heuristic algorithms is to rearrange the order of the cities for the optimal solution. In other words, each individual in the population has enough information to create an optimal solution.

Local search optimization is a kind of method that can find the optimal solutions by modifying the candidate solutions. Although the number of unique candidate TSP solutions is \( n! \), there are only \( n \) cities in the TSP. All the necessary city indices to create an optimal solution are contained in every individual. Since the combination of techniques can be more powerful than techniques that are used on their own, in BBO we intend to use local search as a complement to migration. In the remainder of this section, we introduce three local optimization methods which have been successfully implemented in TSPs: 2-opt, 3-opt, and \( k \)-opt. These methods are applied after migration as a complement to our migration strategy.

2-opt is a simple but effective local research method invented by G. Croes in 1958 (Johnson, 1997). The operation of 2-opt is as follows:

1. Find a random individual in a sequence-based problem.
2. Break two links in this individual.
3. Connect the cities which only have one link connected, with the constraint that the resulting path includes all cities.

In Figure 4, we apply 2-opt to an 8-city TSP as an example.

3-opt is an updated technique based on 2-opt (Johnson, 1997). Instead of replacing two links in the individual as in 2-opt, the 3-opt technique breaks three links and then randomly reconnects the cities that have broken links. Even though 2-opt and 3-opt have good performance in sequence-based problems, their limitation is that the number of links to break and reconnect is predefined, and is difficult to adapt to the current situation.

In order to improve 2-opt and 3-opt, \( k \)-opt is introduced by Shen Lin in 1965 and is discussed in Johnson (1997). \( k \)-opt is a method for adaptively choosing the number of links to break and reconnect. According to experimental results, when the number of the replaced links increases, the performance of the \( k \)-opt increases too. But the computation burden also increases. So we need to find a balance between the expected performance and the computation burden. In heuristic algorithms, at the beginning of the generations, the population improvement speed is very fast. As the complement to migration, we do not need very intense \( k \)-opt. So the \( k \) value should be a small number. But as time progresses, the convergence speed of the algorithm slows down. In this situation, we need to increase the intensity of \( k \)-opt to increase the population improvement speed.

Figure 4. Example of 2-opt with an 8-city TSP
so we should use a bigger $k$ value. Here we can conclude that $k$ should increase as the generation count increases:

$$k = \frac{nc}{2m}$$

where $n$ is the number of cities, $m$ is the maximum generation number, $c$ is the current generation number, and $[x]$ is the greatest integer that is less than or equal to $x$.

**How About Being Greedy?**

Greedy methods have a long history as an effective technique in heuristic algorithms, and many algorithms use them as a basic component. The definition of a greedy method is just as its name implies: always choose the immediate benefit, and refuse to take any losses (Gutin, 2002). But it may not be the best choice for all situations. For example, when playing chess, the players can plan 10 to 20 steps ahead and don’t focus on instant benefits. But in some problems, like the TSP and the network routing problem, a greedy method can still be helpful as a complement to the optimization algorithm.

For BBO, we can use the greedy method in three places – migration, local optimization, and mutation. As we know, migration is a function for an individual to share and receive information with other individuals in order to generate offspring. Although individuals with better fitness have higher probabilities to share features, and individuals with worse fitness have higher probabilities to receive features, there is no guarantee the child will outperform its parents. During mutation we introduce random information to the population. The performance of new individuals is unpredictable in this case. So should we keep an offspring with worse performance? The answer is different according to different situations. If the algorithm really needs fresh blood in the population, then even though the offspring has worse performance than either of the parents, we still keep it. But if we want to make sure the performance of the entire population improves, then we might want to apply a greedy method and abandon offspring with worse performance.

**SOLUTION FOR LARGE SCALE PROBLEMS**

Combinatorial problems are not necessary large scale problems, but usually they are. As mentioned in Section 1, the number of combinations is $9.3326 \times 10^{157}$ for a 100-city TSP, but a 100-city TSP is not even considered a large problem in the TSP family. The computational time is always the top concern for a large scale problem. It is also the top priority when we design an algorithm. In this section, a framework is designed for large scale problems, which will look for the best balance of computational expense and performance. In order to achieve our goal, two advanced techniques are introduced in the framework: problem decomposition and parallel computing.

**Problem Decomposition**

Problem decomposition is a classic method because of the lack of powerful computation methods and machines in the past. A common way to treat a large scale problem is to divide it into small pieces, then solve them individually. The combination of the solution of each small piece then becomes the solution to the original problem. Based on today’s technology, the processing ability of CPUs is incredible. But it still has limitations, and the calculation speed is still not sufficient for many problems like TSPs. That is the reason we turn to problem decomposition in order to reduce the time expense.
Biogeography-Based Optimization for Large Scale Combinatorial Problems

How can we decompose a TSP? Recall in the statistics area that when performing data analysis, we prefer applying clustering algorithms before the analysis. This is because when all the data inside a group is more similar to each other than to the data outside the group, it is easier to conduct an analysis to determine its characteristics.

Although there is no need to perform data analysis for combinatorial problems, clustering algorithms are still an inspiration and can be used directly in TSP decomposition. In 1996, density-based spatial clustering of applications with noise (DBSCAN) was proposed by Martin Ester et al. (Ester, 1996).

Here we need to define four terms: eps-neighbor, minPts, direct density-reachable, and density-reachable. We assume that the space under discussion is two dimensional.

- **EPS-Neighborhood**: Choose an arbitrary point; its eps-neighborhood is the area with radius eps whose center is the given point.
- **MinPts**: The minimum number of points in a cluster.
- **Direct Density-Reachable**: Two points are direct density-reachable if the distance between them is less than eps.
- **Density-Reachable**: Two points are density-reachable if there exists a sequence of points between them such that each adjacent pair in the sequence is direct density-reachable.

The concept of DBSCAN is as follows.

1. Start with an arbitrary point that has not been labeled.
2. Determine its eps-neighborhood. If it contains at least minPts points, then label these points as visited and go to step 3; otherwise, label this point as noise, then go to step 1.
3. Find all the density-reachable points of the arbitrary point chosen in step 1, then label them as visited. A cluster is formed by these points. If all the point are labeled, then terminate; otherwise, go to step 1.

This results in clusters of points that are labeled as visited, and also isolated points that are labeled as noise. DBSCAN is a good fit for TSP. In many traveling salesman problems, the cities we plan to visit have certain patterns. In many TSPs, people intend to go to a major city at first and then explore the cities around it. After that, we head to another major city. We can treat each major city as the center of a cluster, and classify this major city along with its satellite cities as a cluster. This type of density based problem is a perfect match for DBSCAN.

Parallel Computing

Parallel computing has a long history, but for a long time it could not perform well due to the restrictions of hardware (Kumar, 1994). But with today’s technology, CPUs with multiple cores are very common. However, we cannot effectively use the power of most multi-core CPUs, as most software operates based on one core. This is the reason we plan to introduce parallel computing to our framework, because it is an effective way to reduce our long computation time for TSPs. But still, parallel computing can only significantly benefit for large scale problems because of the parallel computing setup time and master to slave communication time. For smaller problems, the setup time and communication time may comprise a large portion of the simulation time.

As mentioned in Section 4.1, we can take advantage of problem decomposition for a large scale problem based on DBSCAN. Solving one large scale problem can be transferred into solving several smaller ones. This solution also raises another question: how should we process all of these small problems? That is why we introduce parallel computing. In our framework, all the
decomposed problems are handled by parallel computing to achieve the maximum CPU usage. Combining all the ideas in this section and previous ones, we propose a framework designed for large scale TSPS, which is called BBO for TSP based on DBSCAN (BBO/DBTSP). The detailed scheme is shown in Figure 5.

Figure 5 describes a framework based on parallel computing for the TSP. In this framework, the large size problem is decomposed to $n$ smaller sub-problems. The number $n$ is determined by DBSCAN. In the following steps, each CPU core acts as a slave in parallel computing, and applies BBO to the assigned sub-problem independently. In order to balance the load of each slave, we equally partition the work each slave receives based on the size of each sub-problem. Once the BBO simulations are finished in all cores, we combine the outputs from each core using NNA to obtain the near-optimal result. In the end, we perform BBO, including the near-optimal result obtained in the previous step, to search for the globally optimal solution. One benefit of this structure is that there is no communication between cores during the parallel computing process, so it significantly reduces the complexity of the process while still attaining 100% CPU usage. But problem decomposition and parallel computation both require time consuming overhead. Although we can benefit a lot from BBO/DBTSP when the problem size is large enough, the run time for small problems may be slower than non-decomposition-based algorithms.

**Experimental Results**

In this section, all the techniques mentioned in Section 3 and the framework we proposed in Section 4 are tested on four TSP benchmarks. All the benchmarks are selected from TSPlib (Reinelt, 1991) - ulysses16, st70, rat575, and u2152. Ulysses16 is a 16-city TSP; st70 is a 70-city TSP; rat575 is a 575-city TSP; and u2152 is a 2152-city TSP. In order to obtain a broad comparison between techniques, benchmark sizes vary from small to extra large problems. The definition of the size of a problem is calculated based on the following rule.

Assume the calculation of the distance between two cities only needs one instruction in a CPU.
The operating frequency of a top line CPU (intel i7-2600, quad core) is 3.4 GHz per core. The total operating frequency is therefore 13.6 GHz.

For an \( n \) city problem, the total time \( T \) needed to calculate the distances of all possible trips is given as follows:

\[
T = \frac{n(n-1)}{1.36 \times 10^{10}} \text{ sec}
\]

Based on the calculation time, TSPs are divided into four categories:

1. **For Problems Less than 16 Cities (Small Problems):** \( T \leq 2.46 \times 10^4 \) sec. Brute force methods are suitable.
2. **For Problems between 17 to 50 (Normal Problems):** \( 2.46 \times 10^4 \text{ sec} < T \leq 1.12 \times 10^{10} \) sec. In this case, for a 50 city problem, it will take \( 8.51 \times 10^9 \) years to find the solution based on a brute force method.
3. **For Problems between 50 to 100 (Large Problems):** \( 1.12 \times 10^{10} \text{ sec} < T \leq 6.68 \times 10^{14} \) sec.
4. **For Problems Larger than 100 (Extra Large Problems):** \( T > 6.68 \times 10^{14} \)

In this book chapter, the aim is not only to find the best modification for BBO, it is also to compare BBO with other popular competitors. Five popular competitors are selected: GA (Poon, 1995), NNA (Cover & Hart, 1967), ant colony optimization (ACO) (Dorigo, 1997), simulated annealing (SA) (Aarts, 1989) and BBO/CO (Ermeg, 2011). In order to guarantee fairness, we set two termination criteria for each algorithm. The algorithm will terminate when either of them is met.

1. **Number of Evaluations of Cost Function:** 10,000
2. **CPU Time:** 300 sec

Also, since the performance of heuristic algorithms varies from one simulation to the next, a single comparison between algorithms may not reflect their true performances. To guarantee a fair comparison, Monte Carlo simulations are performed. We conduct each simulation 20 times, and take the average performance as the overall performance metric.

In order to compare the performance of techniques without affecting other factors, each modified BBO uses the following parameters.

**Default BBO setup:**

- **Population Size:** 100
- **Number of Elite Individuals per Generation:** 1
- **Population Initialization:** Random
- **Migration:** Cycle crossover
- **Local Optimization Method:** None
- **Greedy Method:** None

In the following sections, we study four major modifications - population initialization, different types of migration, local optimization methods, and greedy methods.

**Initialization Methods**

The first test is the performance of different population initialization methods. We designed six population initialization methods: no NNA; NNA for 1 individual, 5 individuals, 50 individuals, 75 individuals, and 100 individuals (the entire population). The simulation results are shown in Table 3.

On the basis of computation time, no NNA is the quickest. But the performance difference between no NNA and NNA is large, especially for larger scale problems. But when we apply NNA to the algorithm, the performance between different setups is very similar to each other. Based on the simulation results, the best setup of NNA is to perform 1 NNA, which means NNA only needs to be performed on one individual in the initial population.
Table 3. Performance of NNA initialization in BBO, averaged over 20 Monte Carlo simulations. The best results are shown in bold font in each row.

<table>
<thead>
<tr>
<th>TSP</th>
<th>Best distance and CPU time per simulation (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No NNA</td>
</tr>
<tr>
<td></td>
<td>Distance</td>
</tr>
<tr>
<td>ulysses16</td>
<td>75.68</td>
</tr>
<tr>
<td>st70</td>
<td>1432</td>
</tr>
<tr>
<td>rat575</td>
<td>128090</td>
</tr>
<tr>
<td>u2152</td>
<td>241745</td>
</tr>
</tbody>
</table>

Crossover Methods

The second test is the performance of different crossover methods in the migration of BBO. In this book chapter, three crossover methods were discussed: matrix crossover, cycle crossover, and inver-over crossover. Their performances are shown in Table 4.

The simulation results show that both in performance and computation, inver-over crossover dominates the other methods. Also, the calculation speed of matrix crossover becomes very slow when the problem size increases, and therefore it is not a good method for large scale problems.

<table>
<thead>
<tr>
<th>TSP</th>
<th>Best distance and CPU time per simulation (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Matrix</td>
</tr>
<tr>
<td>ulysses16</td>
<td>74.22</td>
</tr>
<tr>
<td>CPU Time</td>
<td>0.64</td>
</tr>
<tr>
<td>st70</td>
<td>2725</td>
</tr>
<tr>
<td>CPU Time</td>
<td>2.22</td>
</tr>
<tr>
<td>rat575</td>
<td>102763</td>
</tr>
<tr>
<td>CPU Time</td>
<td>300.00</td>
</tr>
<tr>
<td>u2152</td>
<td>434209</td>
</tr>
<tr>
<td>CPU Time</td>
<td>300.00</td>
</tr>
</tbody>
</table>

Local Optimization

Next is the evaluation of local optimization methods. Three methods were proposed in this book chapter: 2-opt, 3-opt, and k-opt. When using local optimization, we optimized each individual in the population at the end of each generation. The performances of different local optimization methods are shown in Table 5.

The setup with the best computation time is no local optimization. But with small increases of computation time, the improvement is significant when using local optimization, especially for large scale problems. For a small size problem,
Table 5. Performance of No-opt, 2-opt, 3-opt and k-opt, averaged over 20 Monte Carlo simulations. The best results are shown in bold font in each row.

<table>
<thead>
<tr>
<th>TSP</th>
<th>Best distance and CPU time per simulation (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No-opt</td>
</tr>
<tr>
<td>ulysses16</td>
<td></td>
</tr>
<tr>
<td>Distance</td>
<td>75.68</td>
</tr>
<tr>
<td>Time</td>
<td>3.11</td>
</tr>
<tr>
<td>st70</td>
<td></td>
</tr>
<tr>
<td>Distance</td>
<td>1432</td>
</tr>
<tr>
<td>CPU Time</td>
<td>4.23</td>
</tr>
<tr>
<td>rat575</td>
<td></td>
</tr>
<tr>
<td>Distance</td>
<td>128090</td>
</tr>
<tr>
<td>CPU Time</td>
<td>19.23</td>
</tr>
<tr>
<td>u2152</td>
<td></td>
</tr>
<tr>
<td>Distance</td>
<td>241745</td>
</tr>
<tr>
<td>CPU Time</td>
<td>40.23</td>
</tr>
</tbody>
</table>

2-opt and 3-opt outperform k-opt. But with large scale problems, k-opt is the best choice.

Greedy Methods

Next we test different greedy method setups. Three setups are introduced: no greedy method, half of the population uses a greedy method (the individuals that use greedy methods in this approach are randomly selected), and the entire population uses a greedy method. The greedy method is applied at three steps of the BBO algorithm each generation: first we apply it after migration, then we apply it after local optimization, and finally we apply it after mutation. The performances of different greedy method setups are shown in Table 6.

The simulation results reflect that the best strategy from the perspective of computational effort is not to apply a greedy method. Also, when the population is small, the performance is better without the use of a greedy method. But for large scale problems, Table 6 shows the advantage of greedy methods.

Table 6. Performance of different greedy method setups, averaged over 20 Monte Carlo simulations. The best results are shown in bold font in each row.

<table>
<thead>
<tr>
<th>TSP</th>
<th>Best distance and CPU time per simulation (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No Greedy</td>
</tr>
<tr>
<td>ulysses16</td>
<td></td>
</tr>
<tr>
<td>Distance</td>
<td>75.68</td>
</tr>
<tr>
<td>CPU Time</td>
<td>3.11</td>
</tr>
<tr>
<td>st70</td>
<td></td>
</tr>
<tr>
<td>Distance</td>
<td>1432</td>
</tr>
<tr>
<td>CPU Time</td>
<td>4.23</td>
</tr>
<tr>
<td>rat575</td>
<td></td>
</tr>
<tr>
<td>Distance</td>
<td>128090</td>
</tr>
<tr>
<td>CPU Time</td>
<td>19.23</td>
</tr>
<tr>
<td>u2152</td>
<td></td>
</tr>
<tr>
<td>Distance</td>
<td>241745</td>
</tr>
<tr>
<td>CPU Time</td>
<td>40.23</td>
</tr>
</tbody>
</table>


**BBO/DBTSP**

Next we test the BBO/DBTSP framework shown in Figure 5, which is especially designed for large scale problems. Based on the previous simulation results, the best setup for large scale BBO is the following: 1 NNA for population initialization; inver-over crossover; k-opt for local optimization method; and all greedy for the greedy method setup. We use these options in BBO/DBTSP. Then we conduct simulations on the same four TSP benchmarks.

Here, we compare the results between BBO and GA (Kirk, 2007); NNA (Kirk, 2008); ACO (Wang, 2007); SA (Seshadri, 2006) and BBO/CO (Ergezen, 2011). The setups of these algorithms are as follows.

- **GA**: Population size is 100; Crossover is a combination of flip crossover; swap crossover and slide crossover; Crossover rate is 0.5; Mutation rate is 0.01.
- **NNA**: It is not a heuristic algorithm, so no tuning parameters are needed.
- **ACO**: Population size is 20 ants; Initial pheromone value is 10-6; Pheromone update constant is 20; Exploration constant is 1; Global pheromone decay rate is 0.9; Local pheromone is decay rate 0.1; Pheromone sensitivity is 1; Visibility sensitivity is 1.
- **SA**: Initial temperature is 2000; Maximum trails at a temperature are 10 times the population size.
- **Default BBO**: Population size is 100; Number of elite individuals per generation is 1; Population initialization is random initialization; Migration method is cycle crossover; No local optimization method; No greedy method.
- **BBO/CO**: Population size is 100; Number of elite individuals per generation is 1; Population initialization is random initialization; Migration method is inver-over crossover combined with CW circular opposition; No local optimization method; No greedy method.
- **BBO/DBTSP**: Population size is 100; Number of elite individuals per generation is 1; Population initialization is 1 NNA; Migration method is inver-over crossover; Local optimization method is k-opt; Greedy method is all greedy. The CPU contains four cores (considered as four workers in the parallel computation).

Based on the simulation results in Table 7, in ulysses 16, BBO/DBTSP achieved the best solution among all. Although the computation time is slightly longer than the others, it is still within the tolerable range. In st70, SA has the best performance. BBO/DBTSP has the second best which is close to the results from SA, and far better than others. In rat75, ACO is the best choice in the solution aspect, but it is very time consuming compared to the others. This can be a major drawback especially when dealing with large scale problems. BBO/DBTSP is still the second best algorithm, with similar results to ACO but far better computation time. With the largest benchmark u2152, BBO/DBTSP achieved the best performance and fastest convergence speed among all heuristic algorithms. According to these results, BBO/DBTSP has the best overall performance.

### CONCLUSION AND FUTURE RESEARCH DIRECTIONS

In this book chapter, we introduced a BBO algorithm especially designed for combinatorial problems. We presented two fundamental improvements to BBO. First, we introduced additional search techniques to BBO to create hybrid BBO for better performance and convergence speed. Second, we created a framework designed...
Biogeography-Based Optimization for Large Scale Combinatorial Problems

Table 7. Performance of GA, NNA, ACO, SA, default BBO and BBO/DBTSP, averaged over 20 Monte Carlo simulations. The best results are shown in bold font in each row.

<table>
<thead>
<tr>
<th>TSP</th>
<th>Best distance and CPU time per simulation (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GA</td>
</tr>
<tr>
<td>u200</td>
<td>Distance</td>
</tr>
<tr>
<td></td>
<td>CPU Time</td>
</tr>
<tr>
<td>st70</td>
<td>Distance</td>
</tr>
<tr>
<td></td>
<td>CPU Time</td>
</tr>
<tr>
<td>rat575</td>
<td>Distance</td>
</tr>
<tr>
<td></td>
<td>CPU Time</td>
</tr>
<tr>
<td>u2152</td>
<td>Distance</td>
</tr>
<tr>
<td></td>
<td>CPU Time</td>
</tr>
</tbody>
</table>

for large scale combinatorial problems based on a combination of clustering, parallel computing, and BBO.

In the first part of this chapter, we focused on four improvements to BBO: population initialization, migration, local optimization methods, and greedy methods. For population initialization, we introduced the nearest neighbor algorithm (NNA). With the increase of the problem size, we receive more benefit from NNA. Based on the simulation results, I NNA is a good complement to the original BBO. For migration, we used three crossover methods especially designed for TSP: matrix crossover, cycle crossover, and inver-over crossover. With the combination of BBO and these crossover methods, BBO gains the ability to provide solutions to TSPs. According to the performance results in Section 5, inver-over crossover achieves the best overall performance.

Local optimization methods are introduced into BBO to complement the global optimization. The advantage of BBO is that it is designed for global optimization, and it doesn't easily get stuck in locally optimal solutions. However, convergence speed is relatively slow. In contrast, local optimization methods are designed to modify a single individual to search for the local optimum. So we can benefit from local optimization and improve the performance of the BBO. The simulation shows 2-opt and 3-opt give BBO better performance with small problems. k-opt achieves better performance with large problems.

The last BBO enhancement was greedy methods, which always accept changes with better performance, and never takes a step back. It is a common strategy for heuristic algorithms. When combined with BBO, the simulation results show greedy methods can increase performance when the problem size is relatively large.

In the second part, our focus was on large scale problems. A framework was introduced in Section 4. In this framework, we first decompose a large scale problem to smaller sub-problems based on DBSCAN. After successfully decomposing the problems, we apply the concept of parallel computing. Every sub-problem is solved individually with BBO in parallel. The BBO used here benefits from the previous studies by combining the techniques with the best performance - I NNA for population initialization; inver-over crossover; k-opt for local optimization; and all greedy for the greedy method setup. After obtaining the sub-problem solutions from the parallel BBO algorithms, we use NNA to group them and form an overall solution to the problem, which is also an individual in the overall BBO algorithm (each individual in
the population in BBO is a candidate solution to the problem. The last step is to perform BBO for global optimization, and include this newly formed individual in the population. Based on comparisons with other algorithms, BBO/DBTSP has the best overall performance.

FUTURE DIRECTIONS

Combinatorial problems are good benchmarks for heuristic algorithms. Based on the work provided here, BBO has shown its potential in this field. For future work, we suggest four different directions. First, since information sharing techniques are the key component in evolution, we suggest continued study of new techniques to achieve better performance. Second, we note that operations in BBO such as population initialization, mutation, and elitism, affect the performance of the algorithm. We therefore suggest continued study of new techniques for these operations. Third, in order to further test the potential of the scalability of BBO/DBTSP, we suggest that it be tested on larger scale benchmarks like TSPs with over 10,000 cities. Fourth, we note that combinatorial problems represent many real-world problems (for example, scheduling). We therefore suggest studying real-world applications of BBO and the TSP.

ACKNOWLEDGMENT

This work was supported by the National Science Foundation under Grant No. 0826124. Technical writing assistance was provided by Ron Davis from Cleveland State University.

REFERENCES


214


**ADDITIONAL READING**


Biogeography-Based Optimization for Large Scale Combinatorial Problems


**KEY TERMS AND DEFINITIONS**

**BBO:** Biogeography-Based Optimization is an evolutionary algorithm designed by Dan Simon.

**DBSCAN:** A density-based algorithm for discovering clusters in large spatial databases with noise.

**Greedy Method:** It is a kind of methods which always choose the immediate benefit, and refuse to take any losses.

**Migration:** Component of BBO, a technique to share information between candidate solutions.

**NNA:** Nearest neighbor algorithm is designed to connect nearest cities to form a valid TSP solution.

**Parallel Computing:** A form of computation in which many calculations are carried out simultaneously.

**TSP:** Traveling Salesman Problem is a classic type of combinatorial problem in which the goal is to find a route which allows a traveler to visit a set of cities while covering the minimum distance.