A user-friendly Bees Algorithm for continuous and combinatorial optimisation

Asrul Harun Ismail, Wegie Ruslan & Duc Truong Pham

To cite this article: Asrul Harun Ismail, Wegie Ruslan & Duc Truong Pham (2023) A user-friendly Bees Algorithm for continuous and combinatorial optimisation, Cogent Engineering, 10:2, 2278257, DOI: 10.1080/23311916.2023.2278257

To link to this article: https://doi.org/10.1080/23311916.2023.2278257

© 2023 The Author(s). Published by Informa UK Limited, trading as Taylor & Francis Group.

Published online: 20 Nov 2023.

Submit your article to this journal

Article views: 82

View related articles

View Crossmark data
A user-friendly Bees Algorithm for continuous and combinatorial optimisation

Asrul Harun Ismail¹,²,³*, Wegie Ruslan³,⁴ and Duc Truong Pham²

Abstract: This paper introduces a new variant of the Bees Algorithm (BA) called Bees Algorithm with 2-parameter (BA₂), which is a population-based metaheuristic algorithm designed to solve continuous and combinatorial optimisation problems. The proposed algorithm simplified the BA’s parameters by combining exploration and exploitation strategies while preserving the algorithm’s core principles to efficiently search for optimal solutions. The paper provides a detailed description of the algorithm’s core principles and its application to two engineering problems, the air-cooling system design (ACSD) and the printed circuit board assembly sequence optimisation (PASO). The results show that BA₂ outperforms previous versions of the basic BA in terms of convergence speed and solution quality. However, the authors acknowledge that further research is needed to test the scalability and generalisability of the algorithm to larger and more diverse optimisation problems. Overall, this paper provides valuable insights into the potential of metaheuristics for solving real-world optimisation problems.

Keywords: bees Algorithm; nature-inspired algorithm; bee-inspired algorithm; metaheuristics; continuous optimisation problem; combinatorial optimisation problem

1. Introduction

Metaheuristics have become increasingly important in today’s complex optimisation landscape due to their ability to handle large-scale, nonlinear, nonconvex, and NP-hard problems more efficiently than exact methods. As the size and complexity of real-world problems grow, exact methods often become computationally infeasible or less effective, whereas metaheuristics can provide good-quality solutions within a reasonable time frame. Additionally, metaheuristics can adapt better to noisy or dynamic environments, tackle multi-objective optimisation problems, and work effectively with limited information or computational resources. These qualities make metaheuristics an essential tool for addressing the increasingly complex and diverse optimisation challenges that arise in modern applications across various domains.

While metaheuristics offer several advantages in solving complex optimisation problems, they also come with certain disadvantages. These include the lack of optimality guarantees in a very complex problem, as metaheuristics typically find near-optimal solutions rather than exact ones. Additionally, they may suffer from premature convergence, where the algorithm becomes trapped...
in local optima and fails to explore the entire search space effectively. Metaheuristic solutions heavily rely on parameter settings, making it challenging to find the optimal configuration for a specific problem, particularly for users with limited domain knowledge (Riff & Montero, 2013). This initial parameter setting issue, which is a major weakness, can lead to suboptimal performance and slow convergence if not addressed properly.

Many metaheuristic algorithms, including the Bees Algorithm, involve tuning several initial parameters, such as population size, exploration, exploitation, and termination criteria. Determining the ideal combination of these parameters can be a daunting task, particularly for individuals new to the algorithm. In the basic Bees Algorithm, there are at least seven initial parameters (Pham et al., 2005, 2006) that need to be configured before optimising a problem. These include the scout population size \( (n) \), the numbers of elite \( (e) \) and selected \( (m) \) sites, the numbers of elite \( (nep) \) and non-elite selected \( (nsp) \) workers, and the neighbourhood size \( (ngh) \). Additionally, in the standard BA version, users must also set other initial parameters: the neighbourhood search shrinking rate \( (shrink) \) and stagnation limit \( (stlim) \) of the abandonment strategy, which helps in escaping local optimum traps. Balancing exploration and exploitation requires adjusting at least four parameters, which can be difficult for novice users who are unfamiliar with the intricacies of the Bees Algorithm. To further complicate matters, users also need to consider the termination parameters.

This paper will examine the principles of \( BA_2 \) to address the challenge of complex parameter settings and explore its applications in both the continuous and combinatorial domains. \( BA_2 \) introduces a novel search procedure that reduces the initial parameters of the basic Bees Algorithm without fundamentally changing its nature. This updated algorithm only requires two parameters for exploration and exploitation in the solution space, making it more efficient and easier to implement than the basic and standard BA.

The next section will provide a literature review of the Basic Bees Algorithm, featuring six parameters, and the Standard Bees Algorithm, which includes eight parameters. This review will encompass 15 years of development and a wide range of applications from engineering to bioinformatics. Section three will focus on the fundamental concepts of the proposed \( BA_2 \) algorithm, highlighting the inspiration drawn from bee foraging behaviour, which simultaneously encompasses exploration and exploitation.

Section four will focus on computational examples of \( BA_2 \) applied to continuous and combinatorial problems, specifically optimising air-cooling system design (ACSD) and printed circuit board (PCB) assembly sequence optimisation (PASO) problems. The ACSD problem involves optimising systems to fulfil specific requirements, while the PASO problem is a well-known combinatorial routing problem aimed at finding the shortest assembly route for components on a PCB. The results of the proposed \( BA_2 \) algorithm will be presented and compared with outcomes from previous versions to highlight the improvements achieved.

The fifth section concludes the paper by summarising the findings obtained, highlighting the enhanced efficiency and applicability of \( BA_2 \) to various real-world challenges and proposing future research directions.

The main contribution of this work is the reduction of the BA’s initial parameters while preserving the algorithm’s core principles and the demonstration of the successful application of \( BA_2 \) to two engineering optimisation problems, representing both continuous and combinatorial problems.

2. The Bees Algorithm
The Bees Algorithm is a straightforward population-based metaheuristic algorithm used for addressing continuous and combinatorial optimisation problems (Lones, 2020) and one of the bee-inspired algorithms. Drawing inspiration from the foraging behaviour of honeybees (Pham et al., 2005), this intelligent algorithm emulates the process of bees locating nectar and sharing
information about food sources within their hive. Initially, the Bees Algorithm was designed for searching in continuous space and was assessed using numerical functions. In contrast, other popular algorithms, such as the genetic algorithm (GA), ant colony optimisation (ACO), tabu search (TS), and simulated annealing (SA), were initially developed to tackle combinatorial problems in discrete space. The performance of these algorithms was evaluated using the travelling salesman problem (TSP) as the standard test function.

BA has gained popularity in a wide range of applications, such as engineering design, machine learning, and scheduling, due to its ability to efficiently solve complex optimisation problems. However, one of the main challenges of BA is the need to tune its fixed set of parameters manually, which can be a time-consuming process and may not always result in the best possible solution. The basic and standard versions of the Bees Algorithm require six and eight initial parameters, as mentioned above, and initial inputs such as: the error threshold and the maximum number of function evaluations (NFE) or iterations (nIt) are often used as stopping criteria.

### 2.1. Bee-inspired algorithms

There are at least eight prominent bee-inspired metaheuristic algorithms, including Mating Bee Optimization (MBO) proposed by Abbass (2001), Bee Colony Optimisation (BCO) by Lucic (2002) as an extension idea from the Bees System by Lucic and Teodorovic (2001), Bee Evolution for Genetic Algorithms (BEGA) by Jung (2003), Honey Bees Algorithm (HBA) by Nakrani and Tovey (2004), Bees Algorithm (BA) by Pham et al. (2005), Artificial Bee Colony (ABC) by Karaboga (2005), Bee Swarm Optimisation (BSO) by Akbari et al. (2010) and Generic Bee Colony Optimisation (GBCO) by Wong (2012). These algorithms draw inspiration from two aspects of bee behaviour: foraging and mating. Mostly, the algorithms that mirror the foraging actions of bees (BA, ABC, HBA, BSO, BCO, GBCO) use the roles of worker and scout bees to search for solutions. The solutions are depicted as N-dimensional vectors that represent the position of a bee agent, all striving to locate the best food source, or the global optimum. MBO and BEGA, on the other hand, are inspired by the mating and reproduction of bees. Here, every solution embodies a bee’s genotype with the queen bee signifying the best solution, drones indicating trial or random solutions, and worker bees relating to unique heuristic functions.

In general, these bee-inspired algorithms begin by generating initial solutions at random. The process of selection and creation of new solutions continues until the algorithm fulfils a termination condition, usually leading to a solution that is near-optimal. The convergence rate and the degree of accuracy can vary between algorithms, resulting in a broad range of algorithms, each with a distinct level of search efficiency. Achieving a balance between exploration and exploitation, which are critical components of population-based metaheuristics, is vital to prevent premature convergence or stagnation in the search process (Darbandi, 2017). Probabilistic functions largely drive the selection and creation of new solutions, and the main differences between algorithms lie in the specific functions they use (Solgi & Lodiciga, 2021).

Regarding initial parameters, ABC and HBA stand out with only five parameters, the least among the group. BSO uses the most, with eight parameters. BCO, GBCO and MBO have six parameters, while BA and BEGA use seven (these numbers are based on the original versions of the algorithms). To address the complexity of choosing appropriate values for all these parameters, researchers have investigated automatic parameter tuning (Phan et al., 2020). An example is the study by Tan et al. (2021) that used machine learning to tune two of the GBCO parameters on the TSP. In parallel, as reported here for the BA, efforts have been invested in reducing the number of parameters that require tuning.

### 2.2. Core principles of basic BA

As a foraging-inspired algorithm, Basic BA uses scout bees and worker bees to do exploration and exploitation. The algorithm unfolds in four primary phases: Parameter Setting, Initialisation, Local Search, Global Search, and Population Ranking.
2.2.1. Parameters setting

Before running the Basic BA, its initial parameters must be established. These six parameters include: the number of scout bees (n), the number of selected potential patches with high honey resource (m), the number of the highest quality patches deemed as elite patches (e), the number of worker bees on the non-elite selected patches (nsp), the number of worker bees on the elite patches (nep), and the range of neighbourhood search in the solution space (ngh). For the number of patches, \( e(n|m|e, m; n) \) as the n scout bees are representing all visited patches.

The relationship between the parameters is as follows: \( e < m < n \), representing the natural progressive selection from all visited patches to the elite ones. Ideally, “n” should be significantly larger than the problem dimension, with a minimum of three scout bees. Similarly, \( nsp < nep \), indicating that more worker bees are assigned to the more potential patches. The range of neighbourhood search parameter should \( 0 < ngh < 0.25 \), as it governs the search radius during the local search. Additionally, it needs to set two initial inputs value, the maximum number of iterations (nlt) and the solution space boundaries \( (x_{\text{max}}, x_{\text{min}}) \), which is not impact the algorithm performance.

2.2.2. Initialisation

Basic BA starts by dispatching the scout bees (n) across the field to identify potential patches. These patches are uniformly random distributed \( (U(\text{min, max})) \) within the predefined solution space, which is set according to the upper \( (x_{\text{max}}) \) and lower \( (x_{\text{min}}) \) boundaries of the solution space (see Eq.1). The greater the value of “n”, the more comprehensive the initial solutions that represented space equally.

The positions of all scouts \( (x_{\text{scout}}(i)) \) are calculated as follows:

\[
x_{\text{scout}}(i) = U[x_{\text{min}}, x_{\text{max}}], \quad \forall i = 1, \cdots, n
\]  

(1)

2.2.3. Population Ranking-1

After finding the patches, scout bees return to the hive and communicate their discoveries via a “waggle dance”. Patches are evaluated based on their quality, with those demonstrating superior fitness values earning a higher ranking. Patches with better fitness values (represented by a scout) attract more worker bees.

2.2.4. Local Search

Scouts with followers (workers) revisit their patches to exploit them further. For the elite patches, “nep” worker bees accompany their scout to extract resources from the associated patch, with “nsp” workers doing the same for non-elite selected patches. All worker bees exploit the area around their scout, using the scout’s position as the centre and \( ngh \cdot (x_{\text{max}} - x_{\text{min}}) \) as the search radius (see Eq.2). If a worker discovers a patch with a better fitness value within the neighbourhood of a patch, the scout bee moves to that worker’s location. The positions of all workers \( (x_{\text{worker}}(i,j)) \) are calculated as follows:

\[
r = ngh \cdot (x_{\text{max}} - x_{\text{min}})
\]

(2)

\[
x_{\text{worker}}(i,j) = \begin{cases} x_{\text{scout}}(i) + U[-r, r], & \forall i = 1, \cdots, m; \ j = \begin{cases} 1, \cdots, \text{nep} & \text{if } i \leq e \\ 1, \cdots, \text{nsp} & \text{if } e < i \leq m \end{cases} \end{cases}
\]

(3)

2.2.5. Global Search

The remaining scout bees, those without followers, re-exploring the field for more potential patches. These scouts re-initialise a new random solution rather than modifying the current solution. The search procedure mirrors the initialisation phase. If a scout identifies a patch with a superior fitness value within the field (solution space), the scout’s location will be updated.
2.2.6. Population Ranking-2

Post search completion, all the bees return to the hive for a comprehensive evaluation of all the patches. This step also employs a greedy ranking approach, where the best fitness value across all scouts is selected as the optimal solution provided by the Basic BA.

It should be noted that in the continuous formulation, the movements or modifications (Eqs. 1 to 3) are applicable to all variables, whereas in the combinatorial version, only a selected subset of variables undergo modification.

This process continues until a stopping condition is met. In summary, Algorithm 1 presents the Basic BA pseudocode in its most simple form.

<table>
<thead>
<tr>
<th>Algorithm 1: Basic Bees Algorithm (Ismail, 2021)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Start</td>
</tr>
<tr>
<td>2: Set the parameters of $n, e, m, nep, nsp, ngh, nIt$;</td>
</tr>
<tr>
<td>3: Generate $n$ initial sites/solutions from the $n$ scout bees (using Eq.1);</td>
</tr>
<tr>
<td>4: Evaluate the fitness of the $n$ initial solutions;</td>
</tr>
<tr>
<td>5: Select $m$ sites/solutions with the best fitness scores for neighbourhood search;</td>
</tr>
<tr>
<td>6: while iteration $&lt; nIt$ do</td>
</tr>
<tr>
<td>7: for each site $i (i = 1, \ldots, e)$</td>
</tr>
<tr>
<td>8: Exploit inside radii $ngh$ of the site $i$ by $nep$ worker bees (using Eq.3), and replace the site $i$, if a new site is found with better fitness scores within the $ngh$ of site $i$;</td>
</tr>
<tr>
<td>9: end for</td>
</tr>
<tr>
<td>10: for each site $j (j = e+1, \ldots, m)$</td>
</tr>
<tr>
<td>11: Exploit inside radii $ngh$ of the site $j$ by $nsp$ worker bees (using Eq.3), and replace the site $j$, if a new site is found with the better fitness scores within the $ngh$ of site $j$;</td>
</tr>
<tr>
<td>12: end for</td>
</tr>
<tr>
<td>13: for each site $k (k = m+1, \ldots, n)$</td>
</tr>
<tr>
<td>14: Assign remaining $(n - m)$ scout bees to search randomly the new sites/solutions and evaluate their fitness scores;</td>
</tr>
<tr>
<td>15: end for</td>
</tr>
<tr>
<td>16: Evaluate the fitness of the $n$ current solutions;</td>
</tr>
<tr>
<td>17: end while</td>
</tr>
<tr>
<td>18: Report the site with the best fitness score from all sites;</td>
</tr>
<tr>
<td>19: End</td>
</tr>
</tbody>
</table>

In the standard BA (continues) version, two more initial parameters are included, which determine the approach for abandoning unproductive sites and reducing the neighbourhood search scope. The shrinking procedure was first introduced by Pham and Ghanbarzadeh (2007), followed by Pham and Darwish (2008). In their 2008 work, the shrinking procedure was employed alongside fuzzy logic control to ascertain the number of worker bees assigned to each patch. This represented the sole attempt at simplifying the initial parameters of the BA during that period.

The standard Bees Algorithm, introduced by Pham and Castellani (2009), builds upon the basic version by incorporating an additional initial parameter, the stagnation limit ($stlim$), to establish a site abandonment process aimed at enhancing efficiency and effectiveness. It also retains the shrinking procedure proposed by Pham and Ghanbarzadeh (2007). When $stlim$ reaches its
maximum value, worker bees abandon sites that have exhausted their honey resources. These refinements allow the Bees Algorithm to avoid local optima more effectively.

Two years after it was introduced, the Bees Algorithm was used to solve combinatorial problems such as scheduling (Pham et al., 2007) and PCB assembly sequences (Pham et al., 2007). More applications include timetabling, circuit design, gene sequencing, supply chain management, vehicle routing, disassembly planning, and path planning. For combinatorial problems, the Bees Algorithm uses a different search procedure than for continuous problems. Combinatorial problems have special features such as discrete search spaces, limited constraints, ordered solutions, and cost functions (Koc, 2010; Otri, 2011).

The Basic Bees Algorithm for combinatorial optimisation problems (CBA) is similar to the continuous version but with different global and local search procedures. In the combinatorial version, the global search usually employs a constructive heuristic to create the initial solution using random permutation generator, while the local search uses a special random generator instead of moving inside a given radius. This random generator produces a random number and modifies the initial solution by making swapping, reversing, and insertion operations on it. While the continuous version of the algorithm uses “ngh” within the solution space range \( (ngh \cdot (x_{max} - x_{min})) \) for searching movement, the combinatorial version of the Basic Bees Algorithm employs the problem’s dimension \( (ngh \cdot dim) \). Algorithm 2 presents combinatorial search operator in its most basic form (Ismail et al., 2020).

Algorithm 2: Combinatorial Neighbour Search (operator)

Input: \( Tour_{init} \) = initial solution from previous process

Output: \( Tour \) = modified solution

1: Start;
2: \( i \leftarrow \text{random integer (1,3)} \);
3: \( a^{(i)} \leftarrow \text{random integer (1, dim)} \); \# dim = dimension of the problem
4: \( b^{(i)} \leftarrow a + ngh \cdot dim \);
5: if \( i == 1 \) then
6: \( Tour \leftarrow \text{Swap (Tour}_{init\text{, } a^{(i)}, b^{(i)})} \);
7: endif
8: else if \( i == 2 \) then
9: \( Tour \leftarrow \text{Insert (Tour}_{init\text{, } a^{(i)}, b^{(i)})} \);
10: else
11: \( Tour \leftarrow \text{Reverse (Tour}_{init\text{, } a^{(i)}, b^{(i)})} \);
12: endif
13: End;
14: End;

Figure 1 illustrates a movement within a combinatorial problem. This is because of the inherent difference between continuous and combinatorial domains, which originate from the structure of their respective solution spaces. The search movement in combinatorial optimisation involves integer movements within a discrete space.

3. Bees algorithm with two parameters (BA2)
This section discusses into the fundamental principles of BA2, highlighting the key distinctions from the basic BA, while still preserving the core principles of the basic BA and investigate the
exploration and exploitation balance in both algorithms. We undertake a visualisation-based analysis utilising “peaks” function for exploration and exploitation balance analysis.

3.1. Core principles of BA₂

As a version of BA, it is critical for BA₂ to uphold the algorithm’s fundamental principles. BA₂ holds on to the main phases and rules of BA, such as giving more worker bees to good patches for a full search. The main change in BA₂ combines its two types of searching, exploration and exploitation. It does this by making a new function that tells each worker on a patch to either stay close to the scout or move far away while keeps assigning more worker bees to good patches.

The concept of merging these two search processes is derived from Woodgate et al. (2017) assertion that worker bees both exploit and explore a patch. His research, which involved the use of a heat map sensor, revealed that worker bees would explore or exploit a field corresponding to the distribution of resources in that patch. Furthermore, they would extend their exploration when the honey supply within their current patch gets depleted. As depicted in Figure 2, one bee is dedicated to exploitation (Figure 2a), while the other allocates equal efforts to both exploitation and exploration (Figure 2b).

The BA₂ algorithm streamlines the search process by including a particular function and blending the exploration and exploitation procedures. These functions generate the number of worker bees and the radius of the most populated areas for each patch based on their rank. The assigned workers then carry out both exploration and exploitation tasks on each patch. In this basic form of BA₂, the recruited workers are generated once by processing the information on the quality of a patch defined by its rank relative to other patches.

The number of workers assigned to each patch ($w_k$) is determined by a linear function based on their rank ($k$) among the total number of patches ($n$), as shown in Eq.4. The number of workers for the most promising patch ($w_{k-1}$) is equal to the total maximum number of employed workers ($W_{max} = n_p$), while the minimum number of workers ($W_{min}$) is equal to one (see Eq.5). It is important to note that the number of workers assigned to each patch must be an integer. Additionally, the most likely swarming area for the patch with the highest potential will be near ($a_{k-1} = 0$) the centre of the patch or scout position, while the patch with the lowest potential will have a larger coverage ($a_{k-n} = 1$) that tend to explore (see Eqs.6-7). The location of each worker bee will depend on the neighbourhood radius, which is determined by the value of $ngh_k$ (see Eq.8).
The balance (exploration and exploitation) of BA$_2$ can be credited to the triangular random number generator (TRNG) (Kotz & Van Dorp, 2004), which controls the locations where the worker bees swarm. In Eqs. 6, 7 and 8, $a_k$ refers to the most likely value within the triangular distribution $T[0, a_k, 1]$ which ranges between 0 and 1, where 0 corresponds to the smallest coverage or closest proximity to the centre of the flower patch and 1 signifies the largest coverage or distance from the centre. Note that even when $a_k$ is set to 0, the value of $T [0, 0, 1]$ will not be exactly 0, but rather a small value close to 0 within the range 0 to 1.

The triangular random number can be obtained from Eq. 9:

\[
\text{triangular random number } t = \begin{cases} 
  a + \sqrt{U(c-a)(b-a)} & \text{for } 0 < U < F \\
  c - \sqrt{(1-U)(c-a)(c-b)} & \text{for } F \leq U < 1
\end{cases}
\]

In Eq. 9, random variable $U$ is sampled from a uniform distribution within the range (0, 1), and $F = (b-a)/(c-a)$, where $a$ denotes the minimum possible value of $t \in T [a, b, c]$, $b$ represents the most likely value, and $c$ indicates the maximum possible value.

The power of the triangular random distribution lies in its ability to accurately represent random conditions when information is limited. By adjusting the most likely value as the swarming area based on the potential rank ($a_k$), the algorithm can adapt to various random conditions and establish the exploration-exploitation integration search. The minimum and maximum ranges of $a_k$ are set to 0 and 1,
respectively, allowing each patch to incorporate both exploitation ($0 < a_k < 0.5$) and exploration ($0.5 < a_k < 1$) aspects during the search process ($ng_h_k = [0, a_k, 1]$). As a result, the $BA_2$ search procedure is as follows:

$$r(k,j) = ng_h_k \cdot (x_{max} - x_{min}) = [0, a_k, 1] \cdot (x_{max} - x_{min})$$

$$x_{worker}(k,j(k)) = x_{scout}(k) + U[-r(k,j(k)), r(k,j(k))],$$

$$\forall k = 1, \ldots, n, j(k) = \begin{cases} 
1, \ldots, w_1 & \text{if } k = 1 \\
\vdots & \\
1 & \text{if } k = n 
\end{cases}$$

Figure 3 illustrates the role of the triangular distribution in managing the worker bees’ swarm around a scout bee. As observable in both diagrams, exploration and exploitation are present, but they exhibit different ratios. For the $[0,0.25,1]$ distribution, the majority, 80%, of the worker bees are searching in the area near the scout, whereas the opposite scenario applies to $[0,0.75,1]$. Through this function, $BA_2$ successfully blends exploration and exploitation across all patches, but the ratio varies depending on the quality or rank of the patches.
Algorithm 3 provides the pseudocode of BA₂ in its simplest form.

**Algorithm 3: Bees Algorithm 2 Parameter**

1: Start
2: Generate n initial sites/solutions from the nscout bees (using Eq.1);
3: Evaluate the fitness of the initial solutions;
4: Determine the number of worker bees for all sites (using Eq.5), nep worker will exploit/explore on the best site and 1 bee on the worst site;
5: while iteration < nlt do
6: for all sites (k = 1, …, n) do
7: Exploit and explore the area inside radi nghₖ of a site by the worker bees (Eq.11), and replace the scout if a worker is found with the better fitness scores within the nghₖ;
8: end for
9: Evaluate the fitness of the nmodified solutions;
10: end while
11: Report the site with the best fitness score from all scouts;
12: End

For combinatorial optimisation, a different concept of neighbourhood is needed. The radius movement of nghₖ in BA₂ translates into sequence length or range movement in the two-parameter combinatorial BA (BA₂₂). In the combinatorial version, a short-range subsequence holds a similar significance as a small neighbourhood radius does in the continuous version. The BA₂₂ uses the same combinatorial operator as the basic BA version, which includes the swap, reverse, and insert procedures (refer to Algorithm 2 for more information).

Figure 4 presents comparison of functions inside each phase between BA and BA₂ based on their simple initial parameters setting scenario. The parameters for BA are set as n = 3, e = 1, m = 2, nep = 20, nsp = 10, ngh = 0.1 while for BA₂ with n = 3, nep = 20.

It could be seen from the figures that basic BA separates local and global search using parameters of elite (e), and selected patches (m). The local search in BA takes place on m patches utilising a random number generator with a uniform distribution inside a neighbourhood defined by radius ngh. The search is conducted by nep and nsp workers. The global search in BA occurs over the entire solution space. In contrast, in BA₂, both the local and global searches perturb the current solutions within a dynamic radius nghₖ generated by a triangular random number regulator. The searches are conducted by w₁, …, wₖ workers (w₁ = nep). The global search of BA₂ has a larger radius, nghₖ, compared to the local search (nghₖ is greater than 0.5). The local and global searches in BA₂ depend on the ranks of the workers following the fitness ranking of all n scouts. Therefore, in BA₂, the global and local searches operate without requiring parameters e, m, nsp, and ngh.

The Bees Algorithm with two parameters for combinatorial optimisation problems (CBA₂) shares similarities with its continuous counterpart (BA₂), though the search procedures differ. In the combinatorial version, the search procedure tweaks the current solution by executing swapping, reversing, and insertion operations, as outlined in Algorithm 2.

### 3.2. Comparison of exploitation-exploration balance between BA and BA₂

As we can see in the previous section, in the basic BA, the search method has separate exploitation and exploration phases. Exploitation and exploration are carried out independently at m and n – mllocations. The scouts do not recruit or have no worker at the (n – m) sites, as these sites
or patches are less attractive to exploit. The exploitation process uses a uniform distribution number (within the span of ngh projection) to manipulate honey at potential patches. It is worth emphasising that in the BA, all workers present in the selected sites (m) will use the same neighbourhood (ngh) for exploration, unlike in the BA2 method, where the neighbourhood (nghk) is determined based on the rank of the site.

The basic BA also tends more to exploit than explore (Pham & Darwish, 2008) (also can be seen in Figure 5(a-c). This focus on exploitation helps the algorithm look carefully at potential areas and find the best solutions quickly. However, exploration is also important, so the algorithm does not get stuck in local optima solutions and looks at the whole space of solutions. In the basic BA, the remaining scout bees as the main exploration operator use a uniform random number re-initialisation to look at the whole search space well. In this way, scout bees with the lowest rank can visit many different areas, even if they do not bring more bees. The balance between exploitation and exploration is important for the best results because it helps the algorithm search well and focus on the best areas in the space of solutions.

Figure 5a-c display the search pattern on “peaks” function of the basic BA (n = 40, m = 20, e = 5, nep = 20, nsp = 15, ngh = 0.1), while (b) and (d) showcase those of BA2 (n = 40, nep = 20) in these scenarios with more scout bees, making the differences more easily discernible. The “peaks” function shown in Eq.12 can be found in the MATLAB documentation at the following link: [https://uk.mathworks.com/help/matlab/ref/peaks.html].
\[ z = 3(1 - x)^2 e^{-x^2-(y+1)^2} - 10\left(\frac{x}{5} - x^3 - y^2\right) e^{-x^2-y^2} - \frac{1}{3} e^{-(x+1)^2-y^2} \] (12)

These figures support the earlier statement that the basic BA generally intends more towards exploitation than exploration. In contrast, \( BA_2 \) achieves a more balanced approach between these two processes. In BA's pattern, there are 20 remaining bees (purple dots) that have no followers to forage the field. These bees are responsible for exploring new areas to find food sources, while the other worker bees exploit the already known food sources. In the next pattern of \( BA_2 \), Figure 5b-d, all worker bees simultaneously explore new areas and exploit known food sources, except for the two scout bees with the lowest potential (purple dots). These two bees will primarily be exploring the area with only one assigned worker. These scouts, labelled \( k = 39 \) and \( k = 40 \), are considered to have the lowest quality, so they only have one assigned worker bee, with \( w_{k-39} = 1 \), \( 48 \approx 1 \) and \( w_{k-40} = 1 \). In this benchmark function, both algorithms, BA and \( BA_2 \), were able to find the highest point for the “peaks” function problem after 200 iterations, despite these two having different exploration-exploitation balances.

4. \( BA_2 \) and CBA\(_2\) implementation for real problems
To evaluate the performance of the proposed \( BA_2 \) algorithm, two engineering problems were selected. Engineering problems are often challenging due to their complexity and nonlinearity, which involve multiple variables and constraints that must be satisfied to achieve an optimised solution (Gandomi et al., 2013).

The first engineering problem was the design of an air-cooling system, which is a continuous problem. The second problem was the optimisation of a printed circuit board assembly sequence plan, which is a combinatorial problem. By applying the \( BA_2 \) algorithm to these two engineering problems, the effectiveness of the algorithm could be demonstrated and evaluated. As mentioned above, before using BA to solve an optimisation problem, there is a need to set the initial parameters of the algorithm. To compare these two algorithms, BA and \( BA_2 \), a fair scenario must be arranged.

Table 1 provides the initial population size for both algorithms, which in this case the total population is set as 425 bees. The population of the basic BA is computed as follows: \((e \cdot nsp) + ((m - e) \cdot nsp) + (n - m)\). On the other hand, the population of \( BA_2 \) is calculated as the sum of \( w_k \) for \( k = 1 \) to \( n \).

4.1. Case study-1: Air-cooling system design (ACSD)
In this continuous problem, the air needs to be cooled from 95 to 10°C between two stages of air compression, and a facility is shown in Figure 6 to perform this cooling. The process involves cooling the air first in a precooler and then in a refrigeration unit. Water is used to cool the air, and it passes through the condenser of the refrigeration unit, then into the precooler, and finally to a cooling tower where it releases heat to the atmosphere. The flow rate and specific heat of both air and water are given, and the water leaves the cooling tower at a temperature of 24°C.

The system is designed for minimum cost, which includes the cost of the refrigeration unit, precooler, and cooling tower, designated as \( x_1, x_2, \) and \( x_3 \), respectively, in pounds. The cost of the refrigeration unit or the compressor power is designed to be 48\( q_1 \) and is expressed as \( x_1 = 48q_1 \), the precooler design has a capacity of 50\( q_2 \) from the coil pipe and is expressed as \( x_2 = 50q_2 \), and the cooling tower unit is expressed as \( x_3 = 25q_3 \), where \( q_1, q_2, q_3 \) is the cooling capacity of the units. The compression power P kW required by the refrigeration unit in 0.25\( q_1 \), and both \( q_1 \) and the compression power must be absorbed by the condenser cooling water passing through the refrigeration unit. The complete set of constraint equations is:
Table 1. Parameter settings for BA and BA₂

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Symbol</th>
<th>Value</th>
<th>BA</th>
<th>BA₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scout n</td>
<td></td>
<td>50</td>
<td></td>
<td>50</td>
</tr>
<tr>
<td>Elite sites e</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-elite selected site m</td>
<td></td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Elite worker bees nep</td>
<td></td>
<td>20</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>Non-elite worker bees nsp</td>
<td></td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Neighbour search size ngh</td>
<td></td>
<td>0.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Colony size:</td>
<td></td>
<td>425</td>
<td>425</td>
<td></td>
</tr>
</tbody>
</table>

\[ q_1 = (1.2)(1.0)(t_3 - 10) \] (13)

\[ q_2 = (1.2)(1.0)(95 - t_3) \] (14)

\[ P = 0.25q_1 \] (15)

\[ x_1 = 48q_1 \] (16)

\[ x_2 = \frac{50q_2}{t_3 - t_1} \] (17)

\[ x_3 = 25q_3 \] (18)

\[ q_1 + P = (2.3)(4.19)(t_1 - 24) \] (19)

\[ (1.2)(1.0)(95 - t_3) = (2.3)(4.19)(t_2 - t_1) \] (20)
\[(2.3)(4.19)(t_2 - 24) = q_3 \quad (21)\]

The set of Eqs.13-21 includes nine equations, and there are ten variables, namely, \(q_1, q_2, q_3, P, x_1, x_2, x_3, t_1, t_2, \) and \(t_3\). The objective, Eq.22, is to eliminate all variables and equations except for the three optimisation variables, \(x_1, x_2, \) and \(x_3\). As the elimination process proceeds, the number of unknowns will always be one more than the number of equations. Therefore, when all variables except the three \(x\)'s are eliminated, there should be two constraints remaining, which are Eqs.23–24. These two equations represent the constraints of the optimisation problem. Thus, the complete mathematical statement of this optimisation problem is as follows:

\[\text{Min} f_1(x) = x_1 + x_2 + x_3 \quad (22)\]

\[0.01466x_1x_2 - 14x_2 + 1.042x_1 = 5100 \quad (23)\]

\[7.69x_3 - x_1 = 19615 \quad (24)\]

Table 2 was employed to record and compare the solutions obtained by BA, BA\(_2\), and exact method (Gurobipy). The table includes the best obtained, average, worst, standard deviation from 30 independent running experiments, and their (BA and BA\(_2\)) convergence graphics. The results indicated that the solution found by BA\(_2\) was comparable to the best solution found by the basic BA algorithm; although it may be slightly better, the difference was not statistically significant when comparing the average values of the experiments (see Figure 7).

Experiments were also conducted using an exact optimisation method using the Gurobipy software \((f_1(x) = 4683.8251)\) and the results were very close to those obtained by BA and BA\(_2\). This finding suggests that BA\(_2\) can be as effective, or even more effective, than its predecessor in solving continuous optimisation problems even though utilising fewer initial parameter settings.

### 4.2. Case study-2: PCB assembly sequence optimisation (PASO) problem

In addressing this combinatorial problem utilising CBA (Ismail et al., 2020) and CBA\(_2\), we seek to optimise the assembly sequence of a printed circuit board (PCB) by determining the best sequence for placing 50 components on the board, with the aim to minimise assembly time. The optimisation of the PCB assembly

| Table 2. Comparison of results for BA and BA\(_2\) in ACSD |
|-----------------|-----------------|-----------------|-----------------|-----------------|
|                | \(f_1(x)\)      | \(x_1\)         | \(x_2\)         | \(x_3\)         |
| Gurobi         | Exact           | 4683.8251       | 1452.76         | 491.43          | 2739.63         |
| BA\(_2\)       | Best            | 4683.8252       | 1452.65         | 491.56          | 2739.62         |
|                | Worst           | 4683.8259       | 1453.33         | 490.79          | 2739.70         |
|                | Average         | 4683.8253       |                 |                 |                 |
|                | Standard deviation | 0.0003713      |                 |                 |                 |
| BA             | Min             | 4683.8252       | 1452.62         | 491.60          | 2739.61         |
|                | Max             | 4683.9686       | 1460.78         | 482.52          | 2740.68         |
|                | Mean            | 4683.86854      |                 |                 |                 |
|                | Standard deviation | 0.06904884     |                 |                 |                 |
process (a type-1 machine problem) resembles to the well-known travelling salesman problems (TSPs). TSPs are essential examples of combinatorial optimisation problems and are classified within the complexity class of NP-complete.

Figure 8 (a) provides a general representation of a PCB assembly machine, which is made up of three components: a feeder F1 (or an array of feeders F2) that supplies the necessary components, an assembly head H that picks up the components from F1 or F2 and places them onto the PCB, and a table T that carries the PCB. Figures 8 (b) and 8 (c) illustrate the location and coordinates of the components that are being placed onto the PCB. In a type-1 machine, a single feeder F1 supplies components to a single assembly head H (without a turret) at a fixed location in the horizontal X-Y plane. Head H also deposits components onto the PCB at a fixed point in the X-Y plane. Table T moves in the X-Y plane to position

<table>
<thead>
<tr>
<th>i</th>
<th>x</th>
<th>y</th>
<th>i</th>
<th>x</th>
<th>y</th>
<th>i</th>
<th>x</th>
<th>y</th>
<th>i</th>
<th>x</th>
<th>y</th>
<th>i</th>
<th>x</th>
<th>y</th>
<th>i</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>90</td>
<td>11</td>
<td>120</td>
<td>190</td>
<td>21</td>
<td>220</td>
<td>200</td>
<td>31</td>
<td>220</td>
<td>41</td>
<td>40</td>
<td>240</td>
<td>140</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>120</td>
<td>130</td>
<td>12</td>
<td>140</td>
<td>180</td>
<td>22</td>
<td>200</td>
<td>220</td>
<td>32</td>
<td>200</td>
<td>60</td>
<td>42</td>
<td>240</td>
<td>120</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>130</td>
<td>13</td>
<td>160</td>
<td>220</td>
<td>160</td>
<td>23</td>
<td>160</td>
<td>180</td>
<td>33</td>
<td>180</td>
<td>60</td>
<td>43</td>
<td>240</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>140</td>
<td>140</td>
<td>14</td>
<td>180</td>
<td>220</td>
<td>24</td>
<td>180</td>
<td>180</td>
<td>34</td>
<td>160</td>
<td>60</td>
<td>44</td>
<td>220</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>160</td>
<td>140</td>
<td>15</td>
<td>200</td>
<td>180</td>
<td>25</td>
<td>200</td>
<td>170</td>
<td>35</td>
<td>140</td>
<td>40</td>
<td>45</td>
<td>200</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>120</td>
<td>150</td>
<td>16</td>
<td>220</td>
<td>220</td>
<td>26</td>
<td>200</td>
<td>140</td>
<td>36</td>
<td>120</td>
<td>90</td>
<td>46</td>
<td>180</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>100</td>
<td>180</td>
<td>17</td>
<td>240</td>
<td>220</td>
<td>27</td>
<td>240</td>
<td>80</td>
<td>37</td>
<td>140</td>
<td>100</td>
<td>47</td>
<td>160</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>120</td>
<td>230</td>
<td>18</td>
<td>240</td>
<td>210</td>
<td>28</td>
<td>240</td>
<td>60</td>
<td>38</td>
<td>180</td>
<td>140</td>
<td>48</td>
<td>140</td>
<td>80</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>100</td>
<td>230</td>
<td>19</td>
<td>240</td>
<td>200</td>
<td>29</td>
<td>220</td>
<td>60</td>
<td>39</td>
<td>200</td>
<td>130</td>
<td>49</td>
<td>120</td>
<td>50</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>140</td>
<td>220</td>
<td>20</td>
<td>240</td>
<td>180</td>
<td>30</td>
<td>240</td>
<td>40</td>
<td>40</td>
<td>220</td>
<td>160</td>
<td>50</td>
<td>100</td>
<td>60</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
components at the desired locations on the PCB. This type of assembly machine is the simplest and is typically employed when only one type of component needs to be placed.

Process optimisation for an assembly machine typically involves a component sequencing problem. The objective of component placement sequencing is to optimise the movements of the table. This problem can be viewed as a symmetrical TSP, for which the objective function (Pham et al., 2007) can be expressed as:

\[
M \text{Inf}_2(x) = D_{\text{total}} = \sum_{i=1}^{N} D(C(i), C(i + 1))
\]  

where \( N \) is the total number of components to be placed onto a board (PCB), and \( D(C(i), C(i + 1)) \) represents the distance travelled by the table or the assembly head when moving from the placement position for component \( C(i) \) to the placement position for component \( C(i + 1) \). (Note that the placement position for \( C(N + 1) \) is usually considered to be the same as that for \( C(1) \), the starting position, since \( C(N) \) is the last component to be placed onto a given PCB.) As the cycle time is the critical factor of interest in this problem, the objective function is typically rewritten as:

\[
M \text{Inf}_2(x) = T_{\text{total}} = \frac{D_{\text{total}}}{v}
\]  

where \( v \) is the average speed of movement of the table or assembly head (60 mm/s).

Table 3 presents the results’ comparison between two algorithms, namely, CBA and CBA\(_2\), used for optimising the PASO problem. The results showed that CBA\(_2\) outperformed the basic CBA, with a difference of one second between their best and average values. Moreover, the CBA\(_2\) algorithm provided significantly better results compared to previous research on CBA for this PASO problem, reducing assembly time by more than five seconds (from 24.08 seconds to 18.5677) compared to the best solution found previously (Pham et al., 2007).

<table>
<thead>
<tr>
<th>Route/sequence (Best)</th>
<th>CBA(_2)</th>
<th>CBA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst</td>
<td>18.88</td>
<td>20.1</td>
</tr>
<tr>
<td>Average</td>
<td>18.76831</td>
<td>19.71</td>
</tr>
<tr>
<td>Standard deviation</td>
<td>0.149163337</td>
<td>0.335658557</td>
</tr>
</tbody>
</table>

Table 3. Comparison of results for CBA\(_2\) and CBA in PASO

![Graph showing comparison between CBA and CBA\(_2\) for assembly time](image-url)
The study assessed the robustness of the CBA\textsubscript{2} algorithm by running the optimisation program independently 30 times, and the results showed that the algorithm had better accuracy and convergence performance compared to the CBA. Specifically, the CBA\textsubscript{2} algorithm achieved convergence of under 1,200 mm (total distance) or 20 seconds of assembly time at the ±1000\textsuperscript{th} iteration, while the CBA took until the ±1800\textsuperscript{th} iteration to achieve convergence in under 20 seconds of assembly time (see Figure 9).

It is important to mention that the CBA algorithm also exhibited improved performance in this study compared to CBA in previous research (Pham et al., 2007). The superior solution obtained by the current CBA algorithm can be attributed to the application of the combinatorial neighbour search operator, which utilises a combination of reversion, swap, and insertion (Ismail et al., 2020). In contrast, previous research has only employed reversion or 2-opt moves.

5. Discussion

BA\textsubscript{2} shares the strengths of BA without its main limitation associated with the large number of parameters that need to be set by the user. There have been many studies of the types of problems for which BA is most suitable. As BA\textsubscript{2} is based on the same core principles as BA, BA\textsubscript{2} is by design able to handle the same kinds of problems.

From the results presented, it could be seen that the performance of BA\textsubscript{2} is similar to that of BA on the ACSD problem (case study 1) which has relatively few variables. It is recalled that the local search operator in BA\textsubscript{2} does not select which solution variables to modify. Although this could quickly yield reasonably accurate results for problems of small dimensions like ACSD, this approach could give poor accuracy for higher dimensions. In contrast, CBA\textsubscript{2} is selective about the variables that it modifies. This makes CBA\textsubscript{2} more suitable for handling high-dimensional problems and is why it shows a marked increase in accuracy compared to CBA in the PASO problem (case study 2).

There are some potential limitations in this work that should be considered. One possible limitation is the generalisability of the results. The experimental outcomes discussed in the previous section are specific to two distinct optimisation problems, with the problem dimension being relatively small. Hence, we recommend extending the investigation to a broader range of applications and exploring (C)BA\textsubscript{2}’s scalability in managing large-scale optimisation problems.

A third potential limitation is the selection of parameter settings utilised in the experiments. We employed specific parameter settings for (C)BA\textsubscript{2}, acknowledging that different settings may lead to different results. Therefore, the results may not be generalisable to other parameter settings.
However, our work so far has revealed that the algorithm is relatively insensitive to \( n \) and \( \text{nep} \) and, in any case, it is much simpler to decide the values of those two parameters than having to set the seven parameters of the original BA. Nevertheless, in future investigations, it would be valuable to conduct parameter settings analyses to understand the impact of \( n \) and \( \text{nep} \) on the algorithm’s performance.

Lastly, in this paper we have focused on comparing \( \text{BA}_2 \) and its predecessor \( \text{BA} \) to demonstrate that the new algorithm is just as effective despite having a much reduced number of parameters. We have not yet evaluated the performance of \( \text{BA}_2 \) against that of other state-of-the-art optimisation algorithms, which could be the subject of future investigations.

6. Conclusion
In conclusion, this paper introduces new variants of the Bees Algorithm (BA) called \( \text{BA}_2 \) and \( \text{CBA}_2 \), which integrate exploration and exploitation strategies to efficiently solve complex optimisation problems. The introduced algorithms are tested on two engineering problems, namely the air-cooling system design (ACSD) and the printed circuit board assembly sequence optimisation (PASO) and compared with previous versions of the basic BA and CBA. The results demonstrate that \( \text{BA}_2 \) and \( \text{CBA}_2 \) outperform their predecessors in terms of convergence speed and solution accuracy.

This study reveals a significant benefit of \( \text{BA}_2 \), a superior balance between exploration and exploitation, reflecting its capacity to distribute bees evenly across the solution space. This marks a substantial improvement over its predecessor. This attribute of \( \text{BA}_2 \) allows it to investigate the solution space more thoroughly and efficiently, thereby increasing the likelihood of identifying the global optimum solution and avoiding local optima traps.

The codes for \( \text{BA} \), \( \text{CBA} \), \( \text{BA}_2 \), and \( \text{CBA}_2 \) are available in Appendix 1 and 2, or they can be accessed on the website www.github.com/asrulharunismail.

Further research is needed to test the scalability and generalisability of the algorithm to more diverse optimisation problems. Additionally, future studies should also develop a local search operator with variable selection to improve accuracy and convergence quality for larger dimension problems.

Author details
Asrul Harun Ismail 1,2,3
E-mail: asrulharunismail@gmail.com
ORCID ID: http://orcid.org/0000-0002-6580-4654
Duc Truong Pham 4,5
1 Centre of Operational Research and Logistics, School of Mathematics and Physics, University of Portsmouth, Portsmouth, UK.
2 Bees Algorithm Research Group, School of Engineering, University of Birmingham, Birmingham, UK.
3 Department of Industrial Engineering, University of Pancasila, Jakarta, Indonesia.
4 Department of Mechanical Engineering, Catholic University of Indonesia Atma Jaya, Jakarta, Indonesia.

Disclosure statement
No potential conflict of interest was reported by the author(s).

Data availability statement
The Authors confirm that the data supporting the findings of this study are available within this article.

Authorship statement
We are submitting a manuscript entitled: “A User-Friendly Bees Algorithm for Continuous and Combinatorial Optimisation” which will be published in Cogent Engineering. We certify that we have participated sufficiently in the work to take public responsibility for the content, including participation in the concept, design, analysis, writing, or revision of the manuscript. The final manuscript before submission was checked and approved by all the authors.

Citation information
Cite this article as: A user-friendly Bees Algorithm for continuous and combinatorial optimisation, Asrul Harun Ismail, Duc Truong Pham, Cogent Engineering (2023), 10: 2278257.

References


Appendix 1

%%% BA2 for Air Cooling System
clc;
clear;
close all;
%%% Problem Definition
ObjFunction = @(x) objective(x); Dims = 3;
VarMin = [0 0 0]; VarMax = [10000 10000 10000];
%%% Bees Algorithm Parameters
n = 20; nep = 50; MaxIt = 1000;
%%% Initialisation
Bees.Position = []; Bees.Cost = []; Bees = repmat(Bees, n, 1); Shrink = 1;
for i = 1:n
    Bees(i).Position = unifrnd(VarMin, VarMax, [1, Dims]);
    Bees(i).Cost = ObjFunction(Bees(i).Position);
end
 [~, RankOrder] = sort([Bees.Cost]); Bees = Bees(RankOrder);
%% Evaluation
recruitment = ceil(linspace(nep,1,n)); %recruitment
%%% Bees Algorithm Local and Global Search
for it = 1:MaxIt
    for i = 1:n
        bestNewBee.Cost = inf;
        assignment = linspace(0,1,recruitment(i));
        for j = 1:recruitment(i)
            newBee.Position = foraging(Bees(i).Position,assignment(j), ...
                                       VarMin,VarMax,Shrink);
            newBee.Cost = ObjFunction(newBee.Position);
            if newBee.Cost < bestNewBee.Cost
                bestNewBee = newBee;
            end
        end
        if bestNewBee.Cost < Bees(i).Cost
            Bees(i) = bestNewBee;
        end
    end
    [~, RankOrder] = sort([Bees.Cost]); %Evaluation
    Bees = Bees(RankOrder);
    %Update Best Solution
    OptSol = Bees(1);
    disp(['Iteration ' num2str(it) ': BestCost = ' num2str(OptSol.Cost) ...
                ': BestPos = ' num2str(OptSol.Position)]);
    Shrink = Shrink * 0.99;
end
%%% Objective Function and Constraints
function z = objective(x)
    z = x(1) + x(2) + x(3) + penalty(x);
end
function p = penalty(x)
    constraint1 = @(x) 0.01466*x(1)*x(2) - 14*x(2) + 1.042*x(1) - 5100;
    constraint2 = @(x) 7.69*x(3) - x(1) - 19615;
    c1 = constraint1(x); c2 = constraint2(x);
    penaltyFactor = 1e4;
    p = penaltyFactor * min(0, c1)^2 + penaltyFactor * min(0, c2)^2;
end
%% Neighbourhood Search Mechanism
function y = foraging(x, assign, VMn, VMx, Shrink)
    nVar = numel(x);
    ngh = triangular(0, assign, 1); % Triangular RNG
    r = ngh * (VMx - VMn) * Shrink;
    y = x + unifrnd(-r, r, size(x));
    for i = 1:nVar
        y(i) = max(y(i), VMn(i));
    end
end

%% Triangular random number generator (TRNG)
function random_number = triangular(a, b, c)
    U = rand();
    F = (b - a)/(c - a);
    if U <= F
        random_number = a + sqrt(U * (c - a) * (b - a));
    else
        random_number = c - sqrt((1 - U) * (c - a) * (c - b));
    end
end
Appendix 2

%%% BA$_{2C}$ for PCB Assembly Sequence
clc;
clear;
close all;
%%% Problem Definition
Dims=50; VarSize=[1 Dims];
 x = [100 120 100 140 160 120 100 140 120 140 160
  200 200 220 240 240 240 240 220 200 160 180 200
  200 240 240 220 240 220 200 180 160 140 120 140
  180 200 220 240 240 240 220 200 180 160 140 120
  100];
 y = [90 130 130 140 140 150 180 230 230 220 190 180
  220 220 180 220 220 210 200 180 200 200 220 180
  170 140 80 60 60 40 40 60 60 60 40 90
  100 140 130 160 140 120 100 100 100 100 100 80
  50 60];
D=Mat(Dims,x,y); %Matrix Distance
ObjFunction=@(tour) TourLength(tour,D); %Distance Calculation
%%% Bees Algorithm Parameters
n=20; nep=50; MaxIt=2000;
%%% Initialisation
Empty_Bees.Position=[]; Empty_Bees.Cost=[];
Bees=repmat(Empty_Bees,n,1);
for i=1:n
    Bees(i).Position=permperm(Dims);
    Bees(i).Cost=ObjFunction(Bees(i).Position);
end
 [~, RankOrder]=sort([Bees.Cost]); Bees=Bees(RankOrder); %Evaluation
recruitment = round(linspace(nep,1,n)); %recruitment
%%% Bees Algorithm Local and Global Search
for it=1:MaxIt
 for i=1:n
    bestnewbee.Cost=inf;
    assignment= linspace(0,1,recruitment(i)); %assignment
    for j=1:recruitment(i)
        newbee.Position= Foraging_Combi(assignment(i)* Dims);
        newbee.Cost=ObjFunction(newbee.Position);
        if newbee.Cost<bestnewbee.Cost
            bestnewbee=newbee;
        end
    end
    if bestnewbee.Cost<Bees(i).Cost
        Bees(i)=bestnewbee;
    end
 end
 [~, RankOrder]=sort([Bees.Cost]); %Evaluation
Bees=Bees(RankOrder); %Update Best Solution
OptSol=Bees(1);
disp(['Iteration ' num2str(it) ': Best Cost = ';
     num2str(OptSol.Cost/60)]); % time = distance/speed; speed= 60 mm/s
end
figure(1); PlotSolution(OptSol.Position,x,y); %Plot
%%% Matrix Distance
function [ D ] = MatD( n, x,y )
    D=zeros(n,n);
    for i=1:n-1
        for j=i+1:n
            D(i,j)=round(sqrt((x(i)-x(j))^2+(y(i)-y(j))^2)); D(j,i)=D(i,j);
        end
    end
end

%%% Distance calculation
function L=TourLength(tour,D)
    n=numel(tour);
    tour=[tour tour(1)]; L=0;
    for i=1:n
        L=L+D(tour(i),tour(i+1));
    end
end

%%% Combinatorial Neighbourhood Search Mechanism
function qnew=Foraging_Combi(q,sz)
    nVar=numel(q);
    ngh=ceil(triangular(0,sz,1)*nVar); m=randi([1 3]); %Triangular RNG
    switch m
        case 1 % Do Swap
            qnew=Swap(q, ngh);
        case 2 % Do Reversion
            qnew=Reversion(q, ngh);
        case 3 % Do Insertion
            qnew=Insertion(q, ngh);
    end
end

function qnew=Swap(q, ngh)
    n=numel(q);
    ii=randi([1 n]); i2=ii + randi([1 ngh]); i2(2>n)=n;
    qnew=q; qnew([ii i2])=q([i2 ii]);
end

function qnew=Reversion(q, ngh)
    n=numel(q);
    ii=randi([1 n]); i2=ii + randi([1 ngh]); i2(2>n)=n;
    qnew=q; qnew(i1:i2)=q(i2:-1:i1);
end

function qnew=Insertion(q, ngh)
    n=numel(q); a=randi(2);
    switch a
        case 1
            ii=randi([1 n]);
        case 2
            ii=randi([1 n-1]);
            ii=[ii ii+1];
        end
    i2=ii(end) + randi([-ngh ngh]);
    i2(2>n)=n; i2(2<n)=1;
    if ii<i2
        qnew=[q(1:ii-1) q(ii+1:i2) q(iii) q(i2+1:end)];
    else
        qnew=[q(1:i2) q(ii) q(i2+1:ii-1) q(ii+1:end)];
    end
end
%% Triangular random number generator (TRNG)
function random_number = triangular(min, mode, max)
    r = rand();
    prop_mode = (mode - min)/(max - min);
    if r <= prop_mode
        random_number = min + sqrt(r * (max - min) * (mode - min));
    else
        random_number = max - sqrt((1 - r) * (max - min) * (max - mode));
    end
end

%% Plot
function PlotSolution(tour,x,y)
    tour=[tour tour(1)];
    plot(x(tour),y(tour),'k-o','MarkerSize',3,'MarkerFaceColor','b',...
         'LineWidth',.5);
    xlabel('x'); ylabel('y');
    axis equal;
    grid off;
    alpha = 0.1;
    xmin = min(x); xmax = max(x);
    dx = xmax - xmin;
    xmin = floor((xmin - alpha*dx)/10)*10;
    xmax = ceil((xmax + alpha*dx)/10)*10;
    xlim([xmin xmax]);
    ymin = min(y); ymax = max(y);
    dy = ymax - ymin;
    ymin = floor((ymin - alpha*dy)/10)*10;
    ymax = ceil((ymax + alpha*dy)/10)*10;
    ylim([ymin ymax]);
end