Comparative Study of Population-based Metaheuristic Algorithms in Case Study of DNA Sequence Assembly

Lala Septem Riza[1*], Yudi Prasetyo[1], Muhammad Iqbal Zain[1], Herbert Siregar[1], Rani Megasari[1], Topik Hidayat[2], Diah Kusumawaty[2], Miftahurrahma Rosyda[3]

[1]*Department of Computer Science Education*
Universitas Pendidikan Indonesia
Dr. Setiabudi Str. No. 229, Bandung 40154, Indonesia
E-mails: lala.s.riza@upi.edu, yudiprasetyo@upi.edu, iqbalzain99@upi.edu, herbert@upi.edu, megasari@upi.edu

[2]*Department of Biology Education*
Universitas Pendidikan Indonesia
Dr. Setiabudi Str. No. 229, Bandung 40154, Indonesia
E-mail: topikhidayat@upi.edu, diah.kusumawaty@upi.edu

[3]*Informatics Department*
Universitas Ahmad Dahlan
Kapas 9 Str. Yogyakarta 55166, Indonesia
E-mail: miftahurrahma.rosyda@tif.uad.ac.id

[*]*Corresponding author*

Received: December 22, 2023 **Accepted: May 21, 2024**

Published: September 30, 2024

Abstract: *Modern technology encounters difficulties performing DNA sequencing on long DNA sequences. Therefore, longer DNA sequences must be cut into smaller fragments. DNA sequence assembly is the process of combining several short genome sequences to create a longer DNA sequence. This study aims to compare the performance of several population-based metaheuristic algorithms in handling the DNA sequence assembly problem based on computation time, number of contigs, and overlap value. The algorithms used in this study include the Honey Badger Algorithm (HBA), Lévy Flight Distribution (LFD), African Vultures Optimization Algorithm (AVOA), and Particle Swarm Optimization (PSO). Overall, AVOA has the best results where it can produce the most total overlap, where the most overlap is 49952 in the dataset with length 750 and coverage 25. AVOA also has the best efficiency because it has a faster computation time than other algorithms in all datasets. Besides AVOA, PSO produces total overlap and computation time that is not far from AVOA. However, based on the number of contigs, HBA is able to create the least number of contigs, especially on datasets with a length of 750 and coverage of 15, with a total of 6 contigs.*

Keywords: *DNA sequence assembly, Optimization, Population-based metaheuristic, R programming language, String matching.*

Introduction

DNA sequencing is a fundamental technique in molecular biology that enables the determination of the exact sequence of nucleotides in a DNA molecule. It involves methods such as Sanger sequencing and next-generation sequencing (NGS), which have revolutionized the field of genomics and been instrumental in numerous scientific breakthroughs. Sanger sequencing, also known as chain termination sequencing, was the first method developed to sequence DNA and was the method used to sequence the first human genome [39]. On the other hand, NGS technologies, such as Illumina sequencing, have enabled high-speed

sequencing, drastically increasing the speed and volume of data generation [30].

With current technology, long DNA sequences are complex to sequence using an accurate sequencing process. For example, human DNA is 3.2 billion nucleotides long and cannot be read in one process. Because of this problem, long DNA sequences must be broken down into smaller fragments in a sequencing process called shotgun sequencing. This approach breaks large pieces of DNA into small enough pieces that machines can automatically process them. *De novo* genome assembly assembles a series of short genomic sequences, also called reads, into longer DNA sequences, called contigs. *De novo* genome assembly is used for new genomes, i.e., when there is no existing reference genome. There is an excellent need for *de novo* assembly because the number of fully sequenced species is minimal compared to the estimated number of organisms present [26].

Several factors complicate the assembly process. First, repetitive sequences in the genome can lead to ambiguity in assembly because it is difficult to determine where these repeats occur in the genome [3]. Second, sequencing errors can lead to errors in assembly [33]. Finally, the size and complexity of the assembled genome can also present difficulties. For example, making a human genome, which is approximately 3 billion base pairs in size, is a much more complex task than assembling a small bacterial genome. Despite these challenges, advances in sequencing technology and computational methods continue to improve the accuracy and efficiency of DNA sequence assembly.

Traditionally, DNA sequence assembly has been implemented using three different approaches. The first is the Overlap-Layout-Consensus (OLC) method. Examples of tools using this method are PASQUAL [28] and MAP [27]. The OLC approach is based on finding overlaps between reads and constructing a graph where each vertex represents reads, and each edge represents the similarity of overlaps between reads. The graph is then traversed and transformed to extract long contigs. The second approach to implementing DNA sequence assembly is based on greedy algorithms such as SSAKE [44] and SHARCGS [9]. In this case, reads are progressively overlapped using seed reads to generate longer contigs. The third approach to implement DNA sequence assembly uses the De Bruijn graph (DBG) as a data structure, such as Velvet [48], SPAdes [4], and ABySS [40]. DBG-based DNA sequence assembly [36] cuts reads into consecutive sub-sequences of length k, called k-mers. A vertex represents each k-mer, and the edge between two vertices represents $k-1$ overlaps.

Optimization is a fundamental concept in computer science and mathematics that involves finding the best solution among a set of possible solutions to a given problem. The "best" solution is usually defined in terms of an objective function that measures the quality of the solution. Optimization problems can be found in various fields, including engineering, economics, data analysis, and bioinformatics [25]. Different optimization problems exist, such as linear and nonlinear, discrete and continuous, and deterministic and stochastic, each with its own characteristics and solution methods [6]. However, many real-world optimization problems are complex and difficult, often involving large solution spaces, multiple conflicting objectives, and dynamic environments [8].

Metaheuristic algorithms have been developed to address these challenges. Metaheuristics are problem-independent, high-level algorithmic frameworks that provide guidelines or strategies for developing heuristic optimization algorithms [5]. Metaheuristics are often used to solve complex optimization problems where traditional methods are neither effective nor efficient. Metaheuristics can be divided into two main categories: single-solution metaheuristics, such as

Simulated Annealing [23] and Tabu Search [13], and population-based metaheuristics, such as Genetic Algorithms (GA) [15], Particle Swarm Optimization [22], Ant Colony Optimization [10], and Differential Evolution [42]. These kinds of algorithms have been widely used in various fields due to their ability to effectively and efficiently explore the solution space and their flexibility to adapt to different problems [25, 46]. However, each algorithm has advantages and disadvantages, and the selection of an algorithm often depends on the specific characteristics of the issue at hand [2, 21].

Many metaheuristic algorithms that follow the OLC approach and aim to compute Hamiltonian paths on overlap graphs have been developed to tackle the DNA sequence assembly problem, in addition to methods based on graph theory. Several studies have been conducted to design and test GA-based DNA sequence assembly techniques [7, 19, 34]. The potential of algorithms based on swarm intelligence has also been explored, with examples including ant colony optimization [31], Particle Swarm Optimization (PSO) [18, 29, 37, 43], Bee Algorithms [47], Cuckoo Search Algorithm [20], and Penguin Search Optimization Algorithm [12].

This study aims to compare the performance of population-based metaheuristic algorithms on the DNA sequence assembly problem. The algorithms used are PSO [22], Honey Badger Algorithm (HBA) [14], Lévy Flight Distribution (LFD) [16] and African Vultures Optimization Algorithm (AVOA) [1]. We explore the potential of recently developed metaheuristic algorithms. While PSO has demonstrably addressed DNA assembly challenges [18], we aim to evaluate some novel approaches that hold promise for surpassing PSO's performance. LFD [16], HBA [14], and AVOA [1] all possess intriguing characteristics that make them well-suited for this task. These algorithms have shown advancements in various aspects compared to PSO in tackling complex problems, suggesting their potential to yield superior results in DNA assembly. This study proposes a computational model to solve the DNA sequence assembly problem and compares the performance of each algorithm based on computation time, number of contigs, and overlap value.

Materials and methods

Computational model

The computational model is shown in Fig. 1. The model created in this study is used to reconstruct the DNA sequence (short pieces of DNA). This is because the DNA sequence from the sequencing results is complicated to get whole pieces of DNA, fragments of DNA scattered in the environment, or the ability of sequencing technology still needs to be improved. After all, the entire DNA is very long. We use the OLC model approach to reconstruct the DNA sequence, which is explained as follows:

1. Overlap (O): Search for the same sequence among fragments.
2. Layout (L): Uses alignment strategy to sort fragments based on high overlap values.
3. Consensus (C): Calculates the consensus sequences from the layout phase.

An example of the OLC method can be seen in Fig. 2, which illustrates the whole sequence: (A) fragments based on the sequence, (B) overlap search stage, (C) layout stage, and (D) until the fragments are reorganized into consensus.

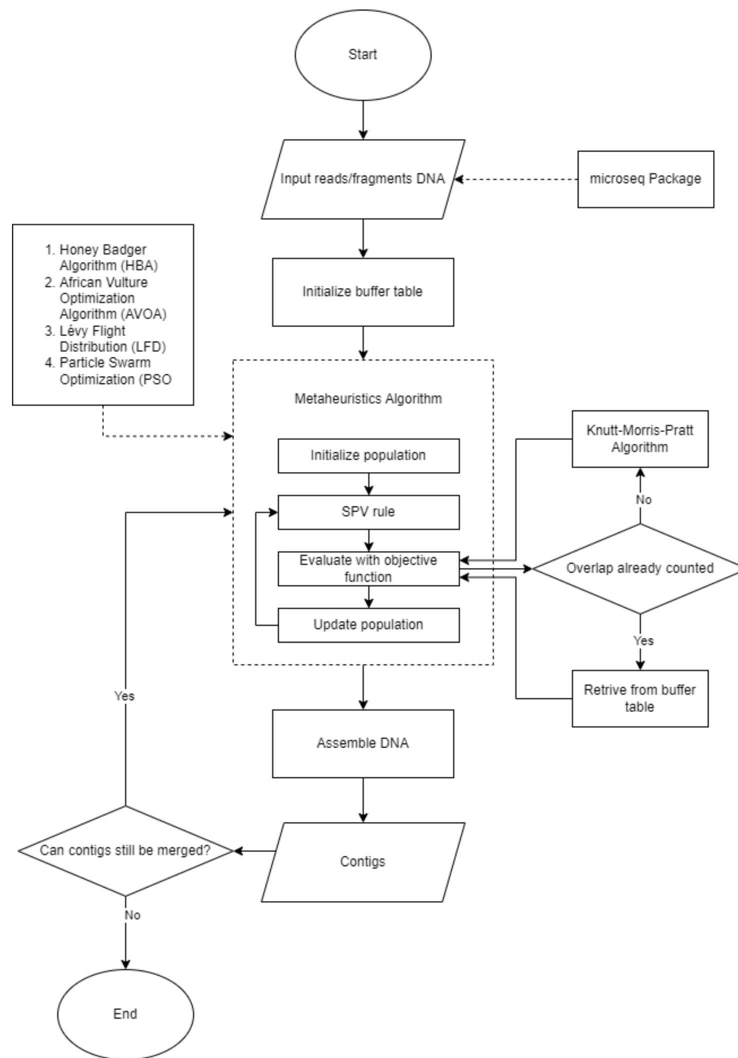


Fig. 1 Computational model diagram of metaheuristic algorithm on DNA sequence assembly problem

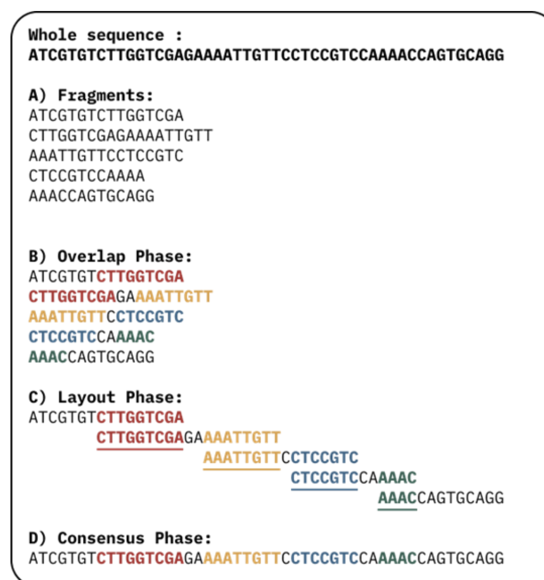


Fig. 2 An illustration of DNA sequence assembly method

Generally, the steps performed in this study can be divided into: (i) data input; (ii) buffer table initialization; (iii) population-based metaheuristic implementation; (iv) DNA assembly; and (v) DNA reassembly to generate contigs. The explanation of each step is as follows.

Input DNA reads

In the computational model created, before users can use the application, they are prompted to enter DNA read files or pieces of DNA that they want to combine in ".fasta" format. The program will process the following data after the DNA reads are successfully entered.

```
seq <- read_fasta("data.fasta")
reads <- to_data_frame(seq)
```

Buffer table

The buffer table is used to store the calculated overlap value. The buffer table is a $n×n$ dimensional table (n is the number of fragments), initially initiated with a value of -1 (Fig. 3); if ever accessed, the value in specific rows and columns changes to the overlap value that the program has calculated. For example, f_1 with f_3 will be calculated for the overlap value. The value in the buffer table at coordinates (1, 3) (assume the index starts from 1) will be changed to the overlap value obtained. Fig. 3 illustrates the buffer table value that is updated to the value of 8 obtained from the search result of the overlap value of f_1 with f_3.

```
        Before                              After

[[-1, -1, -1, -1, -1],           [[-1, -1, 8, -1, -1],
 [-1, -1, -1, -1, -1],            [-1, -1, -1, -1, -1],
 [-1, -1, -1, -1, -1],     →      [-1, -1, -1, -1, -1],
 [-1, -1, -1, -1, -1],            [-1, -1, -1, -1, -1],
 [-1, -1, -1, -1, -1]]            [-1, -1, -1, -1, -1]]
```

Fig. 3 An example of an updated buffer table

If in another combination f_1 is again adjacent to f_3, the program only needs to retrieve the value from the buffer table at coordinates (1, 3) instead of recalculating the overlap value. This is quite effective, considering the number of fragments and combinations in the metaheuristic algorithm and the length of the fragments to be compared. When the algorithm is run, a value at a particular coordinate in the buffer table will likely be retrieved multiple times.

```
overlap_table <- matrix(-1, nrow(reads), nrow(reads))
```

Population-based metaheuristic

At this stage, the initial value of the population of the metaheuristic algorithm is initialized. The population in the metaheuristic represents a group of n different individuals, where the user can set the number of populations (n). The individuals in the metaheuristic represent a combination of fragment sequences that can form many/one contig.

In Fig. 4, we can see the representation of the individual in the population of the metaheuristic algorithm are continuous values, which later will be converted to permutation of a fragment. One way to convert continuous values in the matrix into a permutation form is to use the SPV rule. The way the SPV rule works is to take the smallest value from a list (i), then store it as the smallest value, then the following order is the second smallest value in the list, and so on. The SPV rule uses an example of a value created earlier, as shown in Fig. 4.

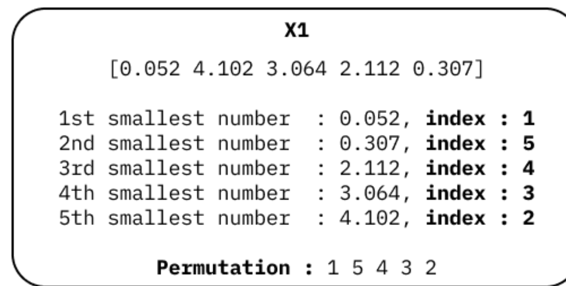


Fig. 4 SPV rule representation of the first index of *X*

The objective function of this metaheuristics algorithm is as follows.

$$total_{overlap_n} = \sum_{j=0}^{n-1} overlap_{value_{j,j+1}}. \tag{1}$$

The overlap value function is calculated based on the number of characters that overlap the suffix of one fragment with the prefix of the following fragment. The combination of all overlap values in the order of the fragments is called the total overlap. This can be expressed as follows.

$$overlap_value_{i,i+1} = score(i,\ i+1). \tag{2}$$

The scoring mechanism is performed using the Knuth-Morris-Pratt (KMP) algorithm [24]. As for the pseudocode to run scoring using the KMP algorithm, it is as follows:

```
FUNCTION kmp_table(s):
table <- empty array of size s.length
j <- 0
FOR i <- 1 TO s.length - 1:
WHILE j > 0 AND s[j] != s[i]:
j <- table[j – 1]
IF s[j] == s[i]:
j <- j + 1
table[i] <- j
END FOR
RETURN table
END FUNCTION

FUNCTION overlap_kmp(s1, s2):
IF s2 is a substring of s1 THEN
RETURN length of s2
ELSE IF s1 is a substring of s2 THEN
RETURN length of s1
ELSE
max_overlap <- minimum of length of s1 and length of s2
IF max_overlap == 0 THEN
RETURN 0
END IF
s <- concatenate s2, '#', s1
table <- kmp_table(s)
overlap_value <- last element of table
IF overlap_value >= max_overlap THEN
RETURN max_overlap
ELSE
# Check for prefix/suffix overlap
FOR i <- overlap_value DOWNTO 1:
IF s2 ends with first i characters of s1 OR s1 ends with first i
```

```
characters of s2 THEN
RETURN i
END IF
END FOR
RETURN 0
END IF
END IF
END FUNCTION
```

The above pseudocode is used to compare the suffix of the first string (s_1) with the second string (s_2) prefix. To do this, it considers the possibility of the first string being inside the second string and vice versa; if this happens, the algorithm will immediately return the value of the shortest string. If this does not occur, it will check for overlapping using the KMP algorithm in the pseudocode snippet above.

```
FUNCTION find_total_overlap(X):
overlap <- 0
FOR i <- 1 TO (length of X) - 1:
left_index <- find_indices_of_i_in_X(X, i)
right_index <- find_indices_of_i_plus_1_in_X(X, i)
IF overlap_table[left_index, right_index] == -1 THEN
overlap_table[left_index, right_index] <-
overlap_kmp(reads[left_index,], reads[right_index,])
END IF
overlap <- overlap + overlap_table[left_index, right_index]
END FOR
RETURN overlap
END FUNCTION
```

The pseudocode above shows the find *total_overlap* function, which aims to find the total overlap value of the sequence permutations stored in an individual. The input parameters given to this function are the list of fragments and the sequence permutation of these fragments. It is repeated for each pair in the fragment sequence; if the data is already in the buffer table, then the data is retrieved directly, but if it is not yet in the buffer table, then the calculation is performed using the *overlap_kmp* function explained earlier. The results of all calculations are summed and stored in the result variable and returned by the function, which represents the individual's score.

The algorithm enters the iteration process after the initial population is formed and the best individual is obtained from the initial population. At each iteration, the solution is evaluated based on the objective function. Next, the update process is applied according to each algorithm. These steps are repeated for a predetermined number of iterations or until a satisfactory solution is found. Fig. 5 illustrates the general process of the population-based metaheuristic algorithm.

```
INPUT: fungsi tujuan f(x)
OUTPUT: solusi terbaik
Bangkitkan populasi awal P0
Evaluasi setiap kandidat solusi pada P0
WHILE t < maxIteration
Perbarui posisi setiap kandidat solusi P0
evaluasi kandidat solusi baru
t = t + 1
END WHILE
```

Fig. 5 Metaheuristic population-based algorithm template [38]

Assemble DNA

After the entire set of metaheuristic algorithm processes has been run, a permutation of the DNA fragment sequence is obtained, and this sequence is then used as the basis for assembling the fragments into a whole. The assembly process shown in Fig. 6 illustrates how the fragments are recombined in the assembly process. However, in this process, the overlap between two fragments must exceed the threshold; this threshold is the minimum percentage of overlap value needed for the fragments to be recombined.

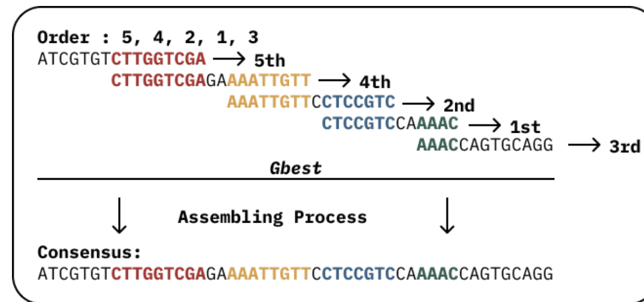


Fig. 6 DNA assembly process

Suppose two fragments are to be merged, for example, *fragA* with *fragB*. In that case, for the two fragments to be merged, the overlap value between the two fragments must be more than the $threshold \times min(len(fragA, fragB))$ or the threshold value multiplied by the length of the shortest fragment between the two fragments. This is done to prevent miss assembly or a condition where the merged fragments are not their partners.

If the two fragments do not match, the mismatched fragment will be added to the second, third, etc. contig list. This also causes the contig produced by the assembly process to consist of one or more contig sequences. The pseudocode to illustrate the DNA assembly process is as follows:

```
FUNCTION assemble_dna(strings):
assembled_overlap_count <- 0
IF length of strings == 0 THEN
RETURN empty string
ELSE IF length of strings == 1 THEN
RETURN strings[1]
ELSE
result <- array containing the first element of strings

FOR i <- 2 TO length of strings:
max_overlap <- 0
max_index <- -1

FOR j <- 1 TO length of result:
overlap <- find_overlap(result[j], strings[i])
IF overlap > max_overlap THEN
max_overlap <- overlap
max_index <- j
END IF
END FOR

IF max_overlap < min(length of strings[i], length of
result[max_index]) * threshold THEN
result <- concatenate result, strings[i]
ELSE
assembled_overlap_count <- assembled_overlap_count +
```

```
max_overlap

IF length of result[max_index] == max_overlap THEN
result[max_index] <- strings[i]
ELSE
result[max_index] <- concatenate result[max_index], substring
of strings[i] from index (max_overlap + 1) to the end
END IF
END IF
END FOR

RETURN [result, assembled_overlap_count]
END IF
END FUNCTION
```

Reassemble DNA

Because the metaheuristic algorithm is an optimization algorithm, likely, the permutation found is not the global best. Still, the local best, and there is a possibility that fragments that should be close together are separated. Therefore, the contig results of the DNA assembly process are re-entered into the model of the metaheuristic algorithm as new input data. Reassembling DNA is performed by rerunning the process from stage 2 to stage 4. This process is repeated until the contig can no longer be split. The characteristics of contigs that can no longer be split are contigs that have gone through the reassembly process, but the length of the contig has not decreased, indicating that there are no pairs that match the predetermined threshold. After this process is complete, the program will obtain contigs that are considered optimal enough. The final contig obtained is output as the result of the program.

Datasets

This study used data from previous studies [2, 17, 18, 32, 43]. We downloaded data from NCBI [49] based on the ID of the DNA sequences to be used for experiments. The DNA sequences used in this study have the NCBI IDs M15421, NC001453, and X60189. The downloaded data will then be cut as in the DNA sequencing process. Details of the data sets are shown in Table 1. Each dataset combines three sequences downloaded from NCBI and then slices them to simulate DNA sequencing. We use different coverage combinations to determine the performance of each algorithm. Length is the length of each read, read numbers are the number of reads (fragments) in the data set, and coverage is the number of times a given nucleotide is sequenced or "covered" by reads.

Table 1. Details of the datasets used

Dataset	Length	Read numbers	Coverage
500_7	500	413	7
750_7	750	275	7
500_15	500	886	15
750_15	750	590	15
500_25	500	1477	25
750_25	750	984	25

Experimental setup
The program in this experiment was written using *R* language version 4.2.1 and *R* Studio version 2023.06.1. The packages used in the research are as follows:

1. *microseq* [41]: used to read ".fasta" files.
2. *dplyr* [45]: used to process data frame in *R*.
3. *Biostrings* [35]: used to convert frame data containing sequences to ".fasta" files.
4. *Rcpp* [11]: used to run C++ code in the *R* environment, in this study, the KMP algorithm is written in C++ so that the KMP process runs faster.

The experimental scenario was performed by running four population-based metaheuristic algorithms on the six datasets shown in Table 1. To optimise our metaheuristic approach for DNA assembly, we employed a grid search strategy. This involved evaluating a range of candidate parameter values and selecting the combination that yielded the most favourable outcomes. Specifically, we prioritised settings that minimised the number of contigs (fragmented DNA segments), reduced computational time, and maximised the number of overlaps (alignments between fragments). The best parameters obtained from the grid search used for each algorithm are shown in Table 2. The experiments were performed by running each algorithm for 1000 iterations, and the population was 25.

Table 2. Used parameters in the experiment

Algorithm	Parameters combination
HBA	$beta = 2, C = 1.5$
LFD	$threshold = 20$
AVOA	$p_1 = 0.4, p_2 = 0.6, p_3 = 0.4$
PSO	$V_{max} = 2, c_i = 2, c_g = 2, w = 0.2$

Results and discussion

Below are the results and discussion of each experiment performed. Each experiment performed is compared based on the number of contigs, the number of overlaps, and the computation time. The lower the number of contigs produced, the better the algorithm because it can combine each fragment with a given threshold. The greater the number of overlaps, the better the value of the objective function, indicating that the algorithm can find a better solution.

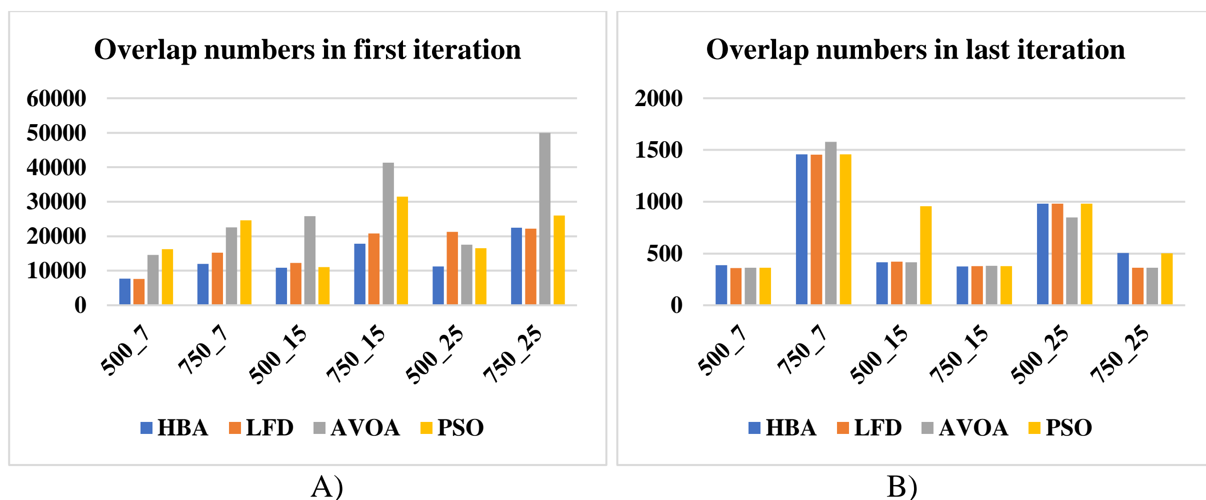
From the results obtained, AVOA has the highest total overlap in almost all datasets at the first run of the algorithm. However, if, based on the number of contigs, HBA is better at producing contigs, it can create a small number of contigs at the last iteration. Based on computational time, AVOA has the fastest computational time in all dataset scenarios on the first run, followed by PSO.

Table 3 compares the number of overlaps from the first and the last iteration. The first iteration is a large computational iteration because there is still a lot of data. The last iteration is the iteration where the contig is no longer divisible. Overall, AVOA can produce the most overlap in the first iteration, followed by PSO. It can also be seen that the longer the reads of the dataset and the greater the dataset coverage, both AVOA and PSO can produce good overlap in the first and the last iteration. HBA also showed good performance in terms of the number of overlaps in the 500 length with 7 coverage and 750 length and 25 coverage. It can be seen where the four algorithms can produce high overlap on a dataset with a read length of 750 and coverage of 25.

Table 3. Comparison of the number of overlaps from the first and last iteration

Iteration	Algorithm	Dataset					
		500_7	**750_7**	**500_15**	**750_15**	**500_25**	**750_25**
First	**HBA**	7743	11993	10894	17817	11221	22451
	LFD	7666	15211	12293	20764	**21241**	22192
	AVOA	14551	22573	**25777**	**41322**	17536	**49952**
	PSO	**16227**	**24660**	11039	31473	16493	25963
Last	**HBA**	**386**	1457	414	375	980	**504**
	LFD	360	1455	420	379	**981**	363
	AVOA	362	**1578**	416	**380**	847	364
	PSO	364	1458	**956**	377	**981**	502

The AVOA algorithm also shows superior performance in terms of total overlap at the first iteration, as shown in Fig. 7A. However, at the last iteration (Fig. 7B), PSO achieves a higher total overlap in most datasets. A higher total overlap indicates that the algorithm can perform optimization well since the objective function, in this case, is to maximize the amount of overlap. In addition to AVOA, PSO achieves high total overlap in some data.



A) B)

Fig. 7 Comparison of overlap numbers in the first iteration (A) and the last iteration (B) of all algorithms

Table 4 compares the number of contigs from both the first and last iteration. It can be seen that HBA can produce fewer contigs, especially on datasets with a read length of 750 and a coverage of 15, with six contigs. It also can be seen that the AVOA algorithm excels on the first iteration but performs poorly at the last iteration in terms of the number of contigs.

Based on the number of contigs, AVOA and PSO can achieve the lower number of contigs at the first iteration (Fig. 8A). Meanwhile, HBA has the lowest number of contigs in some data sets at the last iteration (Fig. 8B). The low number of contigs indicates that the algorithm can sequence DNA sequences correctly, resulting in DNA sequences with overlaps that exceed the threshold.

Table 4. Comparison of the number of contigs from the first and last iteration

Iteration	Algorithm	Dataset					
		500_7	750_7	500_15	750_15	500_25	750_25
First	HBA	86	61	83	62	89	59
	LFD	80	58	85	62	92	66
	AVOA	**70**	**45**	**77**	**52**	**82**	58
	PSO	75	46	84	55	93	**52**
Last	HBA	**15**	11	**12**	**6**	**13**	10
	LFD	16	**10**	13	10	**13**	**8**
	AVOA	**15**	12	14	9	14	9
	PSO	17	11	18	8	**13**	10

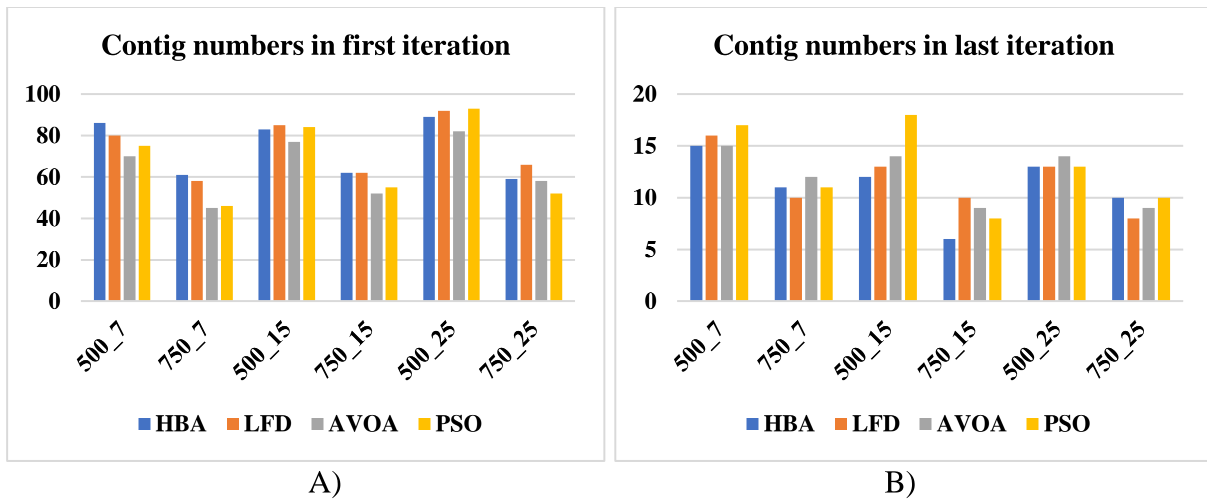


Fig. 8 Comparison of contig numbers in the first iteration (A) and last iteration (B)

One of the standout results from the data is the consistently lower computation time of the AVOA algorithm across different datasets, as shown in Fig. 9. The total computation time data is shown in Table 5. This indicates that AVOA is more efficient regarding computational resources than the other three algorithms. Computational efficiency is important in DNA sequence assembly, which has a vast solution space.

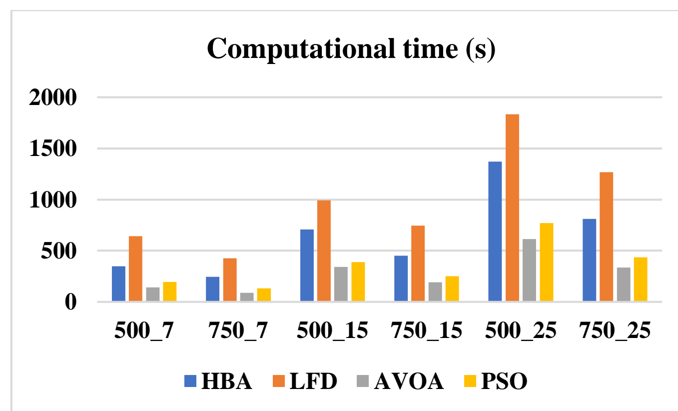


Fig. 9 Comparison of computational time (*s*) for all algorithms

Table 5. Comparison of total computational time

Algorithm	Dataset					
	500_7	**750_7**	**500_15**	**750_15**	**500_25**	**750_25**
HBA	347.26	244.57	707.89	450.51	**1371.9**	811.24
LFD	640.89	425.98	991.49	743.73	1832.41	1266.71
AVOA	**141.51**	**89.8**	**340.08**	**191.79**	613.29	**333.9**
PSO	195.85	133.15	389.56	252.14	771.48	435.13

Overall, AVOA has the best results where it can produce the most total overlaps and also has the best efficiency because it has a faster computation time than other algorithms. Besides AVOA, PSO also produces total overlap and computation time that are not far from AVOA. However, based on the number of contigs, HBA is able to produce the least number of contigs, especially on data sets of length 750.

Conclusion

In this study, we investigate and compare the performance of population-based metaheuristic algorithms in solving the problem of DNA sequence assembly. DNA sequence assembly is an essential and complex computational challenge in genomics, which aims to combine short pieces of DNA sequences into longer ones. The experiments conducted on benchmark data showed that AVOA performed best by producing the highest number of overlaps, namely 49,952, on a data set with a length of 750 and a coverage of 25. In addition, AVOA was the most efficient in terms of computational time compared to other algorithms on all datasets tested. While PSO had almost comparable results to AVOA in overlap and time efficiency, HBA generated the least number of contigs, especially on the 750 length and 15 coverage datasets with only six contigs. To push the boundaries of DNA assembly, we can explore hybridizing metaheuristics with complementary methods, incorporating biological knowledge into the algorithms, and parallelization for large datasets. The big data approach with parallelization is also suitable for this research. All of the approaches should be followed by real-world validation. The use of this approach can be utilized for larger datasets in order to solve real-world problems such as diet analysis, whole genome sequencing, and metagenomic sequencing.

References

1. Abdollahzadeh B., F. S. Gharehchopogh, S. Mirjalili (2021). African Vultures Optimization Algorithm: A New Nature-inspired Metaheuristic Algorithm for Global Optimization Problems, Computers & Industrial Engineering, 158, 107408.
2. Ali A. B., G. Luque, E. Alba (2020). An Efficient Discrete PSO Coupled with a Fast Local Search Heuristic for the DNA Fragment Assembly Problem, Information Sciences, 512, 880-908.
3. Alkan C., S. Sajjadian, E. E. Eichler (2011). Limitations of Next-generation Genome Sequence Assembly, Nature Methods, 8(1), 61-65.
4. Bankevich A., S. Nurk, D. Antipov, A. A. Gurevich, et al. (2012). SPAdes: A New Genome Assembly Algorithm and Its Applications to Single-cell Sequencing, Journal of Computational Biology, 19(5), 455-477.
5. Blum C., A. Roli (2003). Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison, ACM Computing Surveys (CSUR), 35(3), 268-308.
6. Boyd D., K. Crawford (2012). Critical Questions for Big Data: Provocations for a Cultural,

Technological, and Scholarly Phenomenon, Information, Communication & Society, 15(5), 662-679.
7. Bucur D. (2017). A Stochastic *de novo* Assembly Algorithm for Viral-sized Genomes Obtains Correct Genomes and Builds Consensus, Information Sciences, 420, 184-199.
8. Deb K. (2011). Multi-objective Optimisation Using Evolutionary Algorithms: An Introduction, In Multi-objective Evolutionary Optimisation for Product Design and Manufacturing, 3-34, London, Springer London.
9. Dohm J. C., C. Lottaz, T. Borodina, H. Himmelbauer (2007). SHARCGS, a Fast and Highly Accurate Short-read Assembly Algorithm for *de novo* Genomic Sequencing, Genome Research, 17(11), 1697-1706.
10. Dorigo M., V. Maniezzo, A. Colorni (1996). Ant System: Optimization by a Colony of Cooperating Agents, IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), 26(1), 29-41.
11. Eddelbuettel D., R. François (2011). Rcpp: Seamless R and C++ Integration, Journal of Statistical Software, 40, 1-18.
12. Gheraibia Y., A. Moussaoui, S. Kabir, S. Mazouzi (2016). Pe-DFA: Penguins Search Optimisation Algorithm for DNA Fragment Assembly, International Journal of Applied Metaheuristic Computing, 7(2), 58-70.
13. Glover F. (1986). Future Paths for Integer Programming and Links to Artificial Intelligence, Computers & Operations Research, 13(5), 533-549.
14. Hashim F. A., E. H. Houssein, K. Hussain, M. S. Mabrouk, et al. (2022). Honey Badger Algorithm: New Metaheuristic Algorithm for Solving Optimization Problems, Mathematics and Computers in Simulation, 192, 84-110.
15. Holland J. H. (1992). Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence, MIT Press, Cambridge, Massachusetts.
16. Houssein E. H., M. R. Saad, F. A. Hashim, H. Shaban, et al. (2020). Lévy Flight Distribution: A New Metaheuristic Algorithm for Solving Engineering Optimization Problems, Engineering Applications of Artificial Intelligence, 94, 103731.
17. Huang K. W., J. L. Chen, C. S. Yang (2012). A Hybrid PSO-based Algorithm for Solving DNA Fragment Assembly Problem, Third International Conference on Innovations in Bio-inspired Computing and Applications, IEEE, 223-228.
18. Huang K. W., J. L. Chen, C. S. Yang, C. W. Tsai (2015). A Memetic Particle Swarm Optimization Algorithm for Solving the DNA Fragment Assembly Problem, Neural Computing and Applications, 26, 495-506.
19. Hughes J. A., S. Houghten, D. Ashlock (2016). Restarting and Recentering Genetic Algorithm Variations for DNA Fragment Assembly: The Necessity of a Multi-strategy Approach, Biosystems, 150, 35-45.
20. Indumathy R., S. Uma Maheswari, G. Subashini (2015). Nature-inspired Novel Cuckoo Search Algorithm for Genome Sequence Assembly, Sadhana, 40(1), 1-14.
21. Karaboga D., B. Akay (2009). A Comparative Study of Artificial Bee Colony Algorithm, Applied Mathematics and Computation, 214(1), 108-132.
22. Kennedy J., R. Eberhart (1995). Particle Swarm Optimization, Proceedings of the International Conference on Neural Networks (ICNN'95), 4, 1942-1948.
23. Kirkpatrick S., C. D. Gelatt Jr., M. P. Vecchi (1983). Optimization by Simulated Annealing, Science, 220(4598), 671-680.
24. Knuth D. E., J. H. Morris Jr., V. R. Pratt (1977). Fast Pattern Matching in Strings, SIAM Journal on Computing, 6(2), 323-350.
25. Kochenderfer M. J., T. A. Wheeler (2019). Algorithms for Optimization, MIT Press, Cambridge, Massachusetts.

26. Kunin V., A. Copeland, A. Lapidus, K. Mavromatis, et al. (2008). A Bioinformatician's Guide to Metagenomics, Microbiology and Molecular Biology Reviews, 72(4), 557-578.
27. Lai B., R. Ding, Y. Li, L. Duan, et al. (2012). A *de novo* Metagenomic Assembly Program for Shotgun DNA Reads, Bioinformatics, 28(11), 1455-1462.
28. Liu X., P. R. Pande, H. Meyerhenke, D. A. Bader (2012). PASQUAL: Parallel Techniques for Next Generation Genome Sequence Assembly, IEEE Transactions on Parallel and Distributed Systems, 24(5), 977-986.
29. Mallen-Fullerton G. M., G. Fernandez-Anaya (2013). DNA Fragment Assembly Using Optimization, IEEE Congress on Evolutionary Computation, 1570–1577.
30. Mardis E. R. (2008). Next-generation DNA Sequencing Methods, Annual Review of Genomics and Human Genetics, 9(1), 387-402.
31. Meksangsouy P., N. Chaiyaratana (2003). DNA Fragment Assembly Using an Ant Colony System Algorithm, Congress on Evolutionary Computation, 3, 1756-1763.
32. Minetti G., E. Alba (2010). Metaheuristic Assemblers of DNA Strands: Noiseless and Noisy Cases, IEEE Congress on Evolutionary Computation, 1-8.
33. Nagarajan N., M. Pop (2013). Sequence Assembly Demystified, Nature Reviews Genetics, 14(3), 157-167.
34. Nebro A. J., G. Luque, F. Luna, E. Alba (2008). DNA Fragment Assembly Using a Grid-based Genetic Algorithm, Computers & Operations Research, 35(9), 2776-2790.
35. Pagès H., P. Aboyoun, R. Gentleman, S. DebRoy (2022). Biostrings: Efficient Manipulation of Biological Strings, http://bioconductor.riken.jp/packages/3.14/bioc/manuals/Biostrings/man/Biostrings.pdf.
36. Pevzner P. A., H. Tang, M. S. Waterman (2001). An Eulerian Path Approach to DNA Fragment Assembly, Proceedings of the National Academy of Sciences, 98(17), 9748-9753.
37. Rajagopal I., U. M. Sankareswaran (2015). An Adaptive Particle Swarm Optimization Algorithm for Solving DNA Fragment Assembly Problem, Current Bioinformatics, 10(1), 97-105.
38. Riza L. S., E. P. Nugroho (2018). MetaheuristicOpt: An R Package for Optimisation Based on Meta-Heuristics Algorithms, Pertanika Journal of Science & Technology, 26(3).
39. Sanger F., S. Nicklen, A. R. Coulson (1977). DNA Sequencing with Chain-terminating Inhibitors, Proceedings of the National Academy of Sciences, 74(12), 5463-5467.
40. Simpson J. T., K. Wong, S. D. Jackman, J. E. Schein, et al. (2009). ABySS: A Parallel Assembler for Short Read Sequence Data, Genome Research, 19(6), 1117-1123.
41. Snipen L., K. H. Liland (2018). Microseq: Basic Biological Sequence Handling, R package version, 2(5), https://CRAN.R-project.org/package=microseq.
42. Storn R., K. Price (1997). Differential Evolution – A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces, Journal of Global Optimization, 11, 341-359.
43. Verma R. S. (2012). DSAPSO: DNA Sequence Assembly using Continuous Particle Swarm Optimization with Smallest Position Value Rule, First International Conference on Recent Advances in Information Technology (RAIT), 410-415.
44. Warren R. L., G. G. Sutton, S. J. M. Jones, R. A. Holt (2007). Assembling Millions of Short DNA Sequences Using SSAKE, Bioinformatics, 23(4), 500-501.
45. Wickham H., R. François, L. Henry, K. Müller, D. Vaughan (2023). Dplyr: A Grammar of Data Manipulation, https://CRAN.R-project.org/package=dplyr.
46. Yang X. S., S. Deb (2009). Cuckoo Search via Lévy Flights, World Congress on Nature & Biologically Inspired Computing (NaBIC), 210-214.
47. Zemali E., A. Boukra (2018). CS-ABC: A Cooperative System Based on Artificial Bee Colony to Resolve the DNA Fragment Assembly Problem, International Journal of Data Mining and Bioinformatics, 21(2), 145-168.

48. Zerbino D. R., E. Birney (2008). Velvet: Algorithms for *de novo* Short Read Assembly using De Bruijn Graphs, Genome Research, 18(5), 821-829.
49. https://www.ncbi.nlm.nih.gov NCBI Database (Access date 19 September 2024).

Prof. Lala Septem Riza, Ph.D.
E-mail: lala.s.riza@upi.edu

Lala Septem Riza is a Professor and Computer Science Lecturer at the Department of Computer Science Education, Universitas Pendidikan, Indonesia. His research interests are in the fields of artificial intelligence, machine learning, soft computing, and big data analysis.

Yudi Prasetyo, M.Sc.
E-mail: yudiprasetyo@upi.edu

Yudi Prasetyo graduated from the Department of Computer Science Education, Universitas Pendidikan, Indonesia. His research interests are in the fields of R programming language, DNA sequence assembly, optimization, population-based metaheuristic, string matching.

Muhammad Iqbal Zain, M.Sc.
E-mail: iqbalzain99@upi.edu

Muhammad Iqbal Zain is a Fresh Graduate of Computer Science at Indonesian Education University. He is interested in data science and analytics, and has some research background and certifications in those subjects.

Herbert Siregar, M.Sc.
E-mail: herbert@upi.edu

Herbert Siregar is a Lecturer at the Department of Computer Science Education, Universitas Pendidikan, Indonesia. His research interests are in the fields of computer science, education, management, and information systems.

Rani Megasari, M.Sc.
E-mail: megasari@upi.edu

Rani Megasari is a Lecturer at the Department of Computer Science Education, Universitas Pendidikan, Indonesia. Her research interests are in the fields of meeting scheduling negotiation and graph colouring.

Topik Hidayat, Ph.D.
E-mail: topikhidayat@upi.edu

Topik Hidayat is a Doctor and Lecturer at the Department of Biology Education Universitas Pendidikan, Indonesia. His research interests are in the fields of botany, molecular plant systematics, evolutionary biology, environmental biotechnology, and biology education.

Diah Kusumawaty, Ph.D.
E-mail: diah.kusumawaty@upi.edu

Diah Kusumawaty is a Doctor and Lecturer at the Department of Biology Education Universitas Pendidikan, Indonesia. Her research interests are in the fields of genetics, molecular biology, disease and immunity systems in fish.

Miftahurrahma Rosyda, M.Sc.
E-mail: miftahurrahma.rosyda@tif.uad.ac.id

Miftahurrahma Rosyda is a Lecturer of Informatic Departement of Universitas Ahmad Dahlan. Her research interests are in the fields of bioinformatics and artificial intelligence.

© 2024 by the authors. Licensee Institute of Biophysics and Biomedical Engineering, Bulgarian Academy of Sciences. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).