Structural Adaptation of Sorting Algorithms Based on Constructive Fragments

Viktor Shynkarenko [1,*,†], Oleksii Makarov [1,*,†]

[1] *Ukrainian State University of Science and Technologies, Lazaryana str. 2, Dnipro, 49010, Ukraine*

Abstract

Modern information technologies are based on processing large volumes of data. At the same time, the task of developing and applying effective algorithms for data processing, in particular sorting, remains relevant. Constructive-synthesizing modeling was applied to form the sorting algorithm code. The meta-algorithm of program code generation is presented. Parts of existing sorting algorithms and auxiliary utilities are used for generation. A genetic algorithm was used to select the algorithm with the maximum time efficiency under the given conditions of use. The use of a standard genetic algorithm faces a problem caused by a different number of elementary sorting operations, which leads to the use of chromosomes of different lengths. To solve the problem, a representation of the chromosome in the form of a binary tree is proposed. To form an algorithm that is guaranteed to sort the array, all leaf nodes include the final sorting gene at the end of the initial sequence of genes. This gene is decoded by calling the existing sorting algorithm, which is guaranteed to perform the sorting. Mechanisms of coding and decoding of the sorting algorithm from chromosome have been implemented. Linearization is performed for decoding and formation of the appropriate sorting algorithm: formation of a textual representation using a depth-first tree traversal algorithm. The fitness function is defined as the median sorting time of randomly generated sorting arrays (the same arrays for all chromosomes) in some stable environment, taking into account certain features of these arrays. The use of other fitness functions related to the number of calculations, comparisons or permutations is foreseen. The developed software should be applied in adapting sorting algorithms to stable input data streams and usage environments. An experiment was performed to verify the ability of the developed method and the corresponding software to form time-efficient sorting algorithms in different hardware and software environments. In the performed experiments, one component of the hardware and software environment was varied, namely the features of the data to be sorted.

Keywords

Information Technology, Software, Sorting, Constructive-synthesizing Modeling, Genetic Algorithm, Binary Tree

1. Introduction

With the development of the Internet and information technology in general, the volumes of data generated and stored are growing rapidly [1]. With the digitization of most areas of life - from business and science to personal communications - there is a need for effective management of this data. The growing volume of data presents us with challenges related to storage, processing, analysis and interpretation.

In this context, the relevance of efficient sorting algorithms becomes critical. To effectively work with large volumes of data, it is necessary to quickly and efficiently handle their processing. Even the simplest operations, such as sorting, can become time- and resource-consuming if inefficient methods are used.

Efficient sorting algorithms [2] make it possible to quickly process large amounts of data, which is critical for many applications. In some network devices, data sorting can be used to optimize the

14th International Scientific and Practical Conference from Programming UkrPROG'2024, May 14-15, 2024, Kyiv, Ukraine
[*] Corresponding author.
[†] These authors contributed equally.

shinkarenko_vi@ua.fm (V. Shynkarenko); makarovov@hotmail.com (O. Makarov)
0000-0001-8738-7225 (V. Shynkarenko); 0009-0003-0921-155X (O. Makarov)

processing of data packets and manage network traffic. Database management systems (DBMS) use sorting algorithms to perform queries, merge data and other operations [3]. In distributed data storage systems, such as Hadoop or Apache Spark [4], sorting algorithms are used to process large amounts of information and ensure speed.

The evolution of sorting algorithms is a fascinating journey in the history of computer science that began with simple but effective methods. Those algorithms were studied with theoretical [5] and experimental [6] methods. For example, Bubble sort or Insertion sort with computational complexity $O(n^2)$ [7, 8]. Later, more complex and optimized algorithms appeared, more suitable for sorting large volumes of data. Such as Quick sort or Merge sort, with an average computational complexity of O(n*log(n)). Numerous attempts have been made to improve and optimize existing algorithms [9, 10, 11]. In the subsequent growth of requirements for stability [12] and speed of sorting led to the appearance of combined algorithms. The most famous representatives of which are Timsort [13] – a symbiosis of Insertion sort and Merge sort. Introsort [14] starts with a Quick sort, then, under certain conditions, switches to a Heapsort, and calls an Insertion sort for small sequences [15].

Combined algorithms include the advantages of the components to increase efficiency. In [16, 17], the newest approach to the formation, transformation, and analysis of structures using the operations of linking, substitution, inference, etc. is considered.

For the formation of structures (components and their mutual arrangement) of sorting algorithms, the approach of constructive-synthesizing modeling is applied. This ensures the following tasks are solved:

- creation of new sorting algorithms from parts of existing ones;
- adaptation of sorting algorithms to sorted data;
- adaptation of data structures in RAM.

This work does not consider a theoretical model, but only its practical application based on this model.

The purpose of this work is to develop a genetic algorithm [18, 19] for structural adaptation of sorting algorithms. Structural adaptation of sorting algorithms consists in forming an adapted algorithm from some parts of known algorithms in such a way that it will be no worse in time indicators than other sorting algorithms in some stable usage environment.

A feature of the genetic algorithm for this task is the formation of chromosomes of undetermined length and composition with the possibility of both encoding and decoding into a sorting algorithm.

2. Construction of sorting algorithms

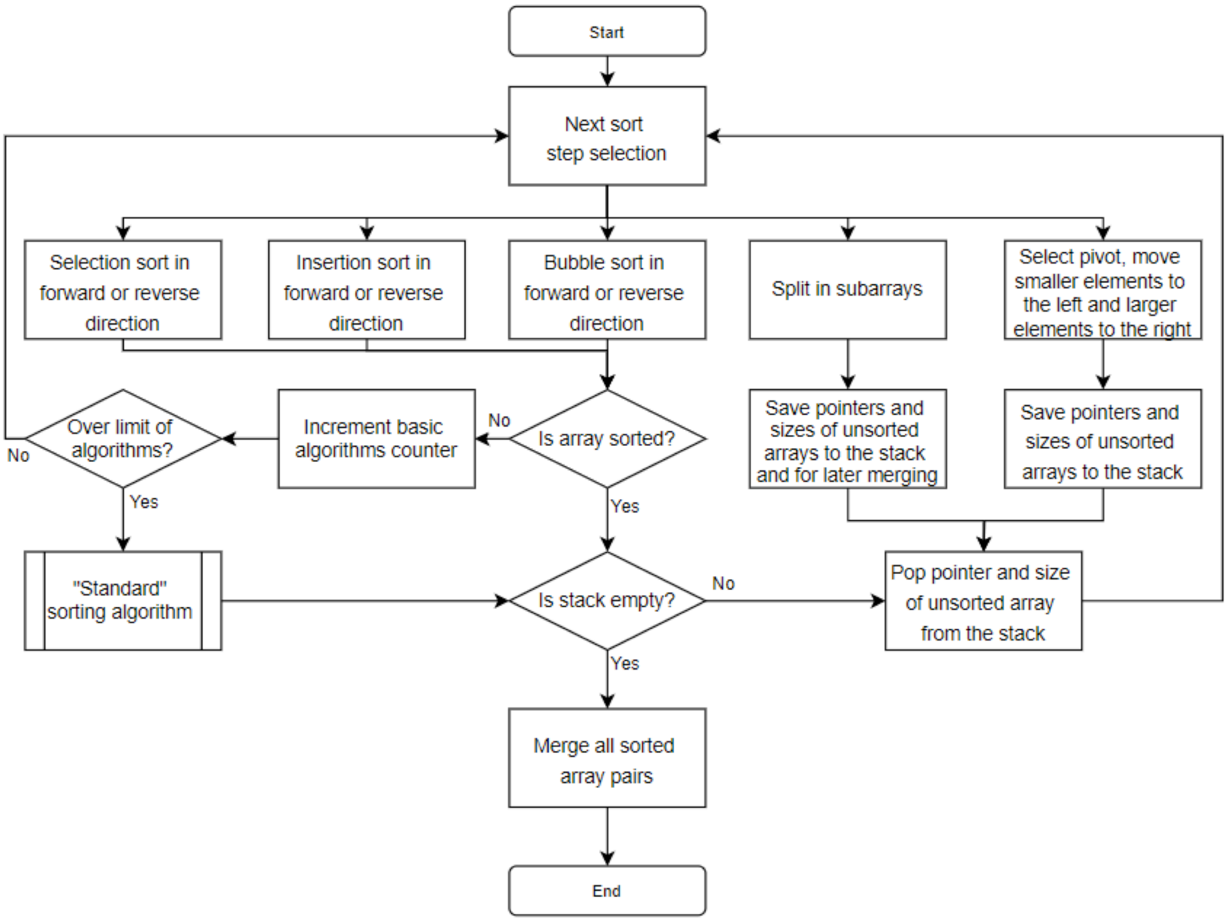
To construct sorting algorithms, we will use parts of existing well-known sorting algorithms. The following basic (atomic) operations from these algorithms were used in the current version of the program:

- Quick sort. Splitting an array of data into two parts relative to a selected element, called a pivot. All elements smaller than the pivot element are moved to the left of it, and all larger elements are to the right;
- Insertion sort. Insertion of one element from the unsorted part of the array into the sorted subarray;
- Selection sort. Search for the minimum or maximum element in an unsorted subarray and permutation;
- Cocktail shaker sort. Passing through the array in the forward or reverse direction and permuting all pairs of elements standing in the reverse order;

- Merge sort. Merging two sorted arrays into one.

Also, a possible operation is the conditional division of the array into two parts and the sorting of each separately. After each part is sorted, it is necessary to merge it into a single sorted array.

The model involves the addition of other basic operations. These can be parts of existing sorting algorithms. For example, the insertion of an element with a certain step, used in the Shell sort. It is also advisable to use deterministic and stochastic preprocessing algorithms [16], or their component operations. This can improve the efficiency of the final algorithm.



Start
Next sort step selection
Selection sort in forward or reverse direction
Insertion sort in forward or reverse direction
Bubble sort in forward or reverse direction
Split in subarrays
Select pivot, move smaller elements to the left and larger elements to the right
Over limit of algorithms?
No
Increment basic algorithms counter
No
Is array sorted?
Save pointers and sizes of unsorted arrays to the stack and for later merging
Save pointers and sizes of unsorted arrays to the stack
Yes
Yes
"Standard" sorting algorithm
Is stack empty?
No
Pop pointer and size of unsorted array from the stack
Yes
Merge all sorted array pairs
End

Figure 1: Block diagram of the meta-algorithm of program code generation/

2.1. Algorithm formation rules

To form the algorithm (Figure . 1), the component basic operations are chosen randomly. Thus, any combinations of atomic algorithms are possible and, as a result, many unique constructed sorting algorithms.

Some basic algorithms have special requirements for their use. If the requirements are not met, the generation will be impossible or the constructed algorithm will not complete the array sorting. For example, when dividing an array into two parts and sorting each part separately, the merge operation into one sorted array should be called.

Consider problems with a combination of sorting algorithms. The atomic parts of the Selection and Bubble sort algorithms divide the array into an unsorted (or not yet processed) part and a sorted part. If forward and reverse passes are performed, then the array is divided into two sorted parts, at the beginning and end of the array, and unsorted. Due to the peculiarities of the logic of the algorithms, the sorted parts have elements that are strictly smaller (at the beginning) or larger (at the end of the array) than those remaining in the unsorted middle part. If we pass through array only in one direction, then we will get classic Selection or Bubble sort. In this case, the array will be completely sorted. If passes in different directions are used, then the sorted parts at the beginning and at the end will grow until they meet and form a single sorted array.

For classic Insertion sort - when all insertions are performed only in the left (or only in the right) part – the result will be a fully sorted array. But if we perform insertions at the beginning and end of the array, we will form two sorted arrays, respectively. Moreover, there are no guarantees that the elements of the right array are strictly greater than or equal to the elements on the left. To merge two sorted subarrays into one, you need to call the Merge operation.

Additional complexity arises combining the algorithms described above. Suppose that an array of length N is being sorted. After performing M operations of selection to the beginning or passes through with the bubble sort in the reverse direction, we are guaranteed to have a sorted subarray [0, M-1] on the left, all elements of which are strictly less than or equal to the remaining elements in the unsorted part. If we now insert an element under index M in the left part, we will have a sorted array [0, M], but now its elements will not be less than or equal to those remaining in the unsorted part.

Subsequent calls of selection operations may not be efficient because the smallest element from the unsorted part of the array may be smaller than the one added by the Insertion sort. And the left sorted part after adding this element becomes unsorted, which means the inefficiency of the created algorithm.

For Bubble sort, one insert operation is not critical. If the element added by insertion under index M is greater than the next (M+1) that will be added by Bubble sort, then at the first step they will be swapped and the subarray will be sorted. However, if an insert and at least one selection operation were performed, then the Bubble sort loses its effectiveness and the array may remain in an unsorted state (Figure . 2).

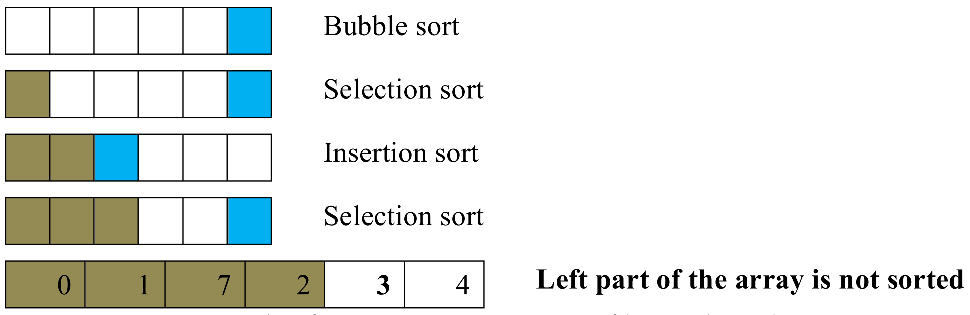


Figure 2: An example of an incorrect sequence of basic algorithms.

To solve the above-mentioned problem, we will introduce new atomic operations – merge from the left (ML) and merge from the right (MR). Additional indicators will be used to track calls to insert operations. If an insert to the beginning (left) operation was used, the appropriate indicator will be set. Before you next call the Bubble sort or select operations on the current array, you will need to call the merge from the left operation first. That is, save the pointers of the sorted part on the left and the rest of the array for further merging. For a similar situation, the merge operation on the right will be called at the end of the array.

3. Application of the genetic algorithm to select the most effective sorting algorithm

A chromosome consists of a sequence of genes – basic sorting algorithms and auxiliary utilities. To represent a chromosome in text form, we set a text representation for each gene. Thus, the chromosome will have a sequence of genes separated by the symbol ",". Since each gene represents a function, we will use its abbreviation for the text representation. Consider existing genes:

- BSB (Bubble sort backward) – one pass through the array from the end to the beginning with the permutation of pairs of elements in the reverse order;
- BSF (Bubble sort forward) – the same as BSB, only passing from the beginning to the end of the array;
- FSB (Find swap biggest element) – search for the largest element in the unsorted part of the array and swap it to the current place;
- FSS (Find swap smallest element) – the same as FSB, only the search for the smallest element is performed;
- SEEI (Single element end insertion) – insertion of the current element into the sorted part at the end of the array;
- SEI (Single element insertion) – insertion of the current element into the sorted part at the beginning of the array;
- SIS (Split in subarrays) – dividing the unsorted part of the array into two equal parts for further sorting of each of them separately. Also storage of pointers and sizes of arrays for further execution of the merge operation;
- FS (Final sort) – one of the well-known standard sorting algorithms, which is guaranteed to perform sorting;
- PUA (Pop unsorted array) – get the pointer and size of the next unsorted part for sorting. Executed for each part saved by the SIS operation;
- ESS (End subarray sort) – an auxiliary operation that closes curly brackets opened by previous operations to check the sorting indicator. Executed for each PUA operation after the sort gene sequence;
- ML (Merge left) – saving pointers to the sorted part at the beginning and the rest of the array for further merging into one sorted array;
- MR (Merge right) – the same as ML only for the sorted part at the end of the array;
- P (Partition) – choosing the pivot element, moving elements smaller than the pivot to the left and larger elements to the right.
- An example of a formed chromosome in text form:
- SEEI, MR, FSB, SIS, PUA, BSF, FSS, FS, ESS, PUA, SEI, ML, FSS, FS, ESS, ESS.
- To limit the length of the chromosome, we will introduce certain restrictions.

Initially, when sorting the input array, the depth is equal to one. When dividing the array and proceeding to sorting any of its parts, the depth increases by one. In this way, the number of partitioning of the array into parts is limited.

A separate parameter limits the number of basic sorting operations for each subarray. When the maximum value is reached, a split or final sort is called.

3.1. A genetic algorithm for generating sorting algorithms

Each individual represents a sorting algorithm. Atomic operations that are components of existing sorting algorithms and auxiliary utilities are represented by genes.

The fitness function that evaluates the quality of each individual is the time efficiency. But there can also be the following: the number of comparisons, the number of permutations or a weighted combination of those factors.

Generation of the next population is carried out with the help of crossover and mutations. Crossover exchanges parts between two parents to create a new individual.

A mutation randomly changes some elements of the genetic sequence. Some parts of the chromosome are generated anew and atomic operations are randomly selected.

Individuals are selected for the next generation based on their fitness. Individuals with greater fitness are more likely to survive and participate in the creation of new individuals.

A certain number of individuals, with the best indicators, are transferred to the next generation without changes. Others are formed by crossing individuals of the existing population. Individuals for crossing can be chosen randomly or according to certain rules. It is also possible to crossover the best individuals according to the rules and the rest randomly. To diversify the population, a certain number of randomly generated individuals are added, just as in the first population.

Genetic algorithm parameters such as population size, mutation probability, number of generations, etc. are set. A stopping condition is defined, for example, the maximum number of generations or reaching a certain level of fitness.

Several iterations of the genetic search are performed, parameters and the fitness function are optimized to improve the results.

The application of the genetic algorithm to the generation of sorting algorithms allows for the automatic evolution of solutions intended for different scenarios and their adaptation to constantly changing conditions.

3.2. Representation of a chromosome in the form of a tree

The input array can have any percentage of sorting. Once the data in the array has been sorted, it is a good idea to terminate the execution to avoid degradation of time efficiency. We implement tracking of such a situation by using the array sorting flag, which will be checked after each basic sorting operation.

Constant updating and checking of the array sorting flag introduces additional rules and restrictions into the process of chromosome generation. In the code, each subsequent check adds a new level of nesting and scope, delimited by curly braces "{" and "}". Checking the sorting flag and opening the scope has each gene representing the basic sorting operation. Accordingly, the final section of the chromosome should have curly braces that close in the number equal to the number of open braces. When dividing the array into subarrays and sorting each separately, we will have a sequence of basic sorting operations and the corresponding number of closed brackets. The optimal data structure for representing the above-described approach is a binary tree (Figure . 3)

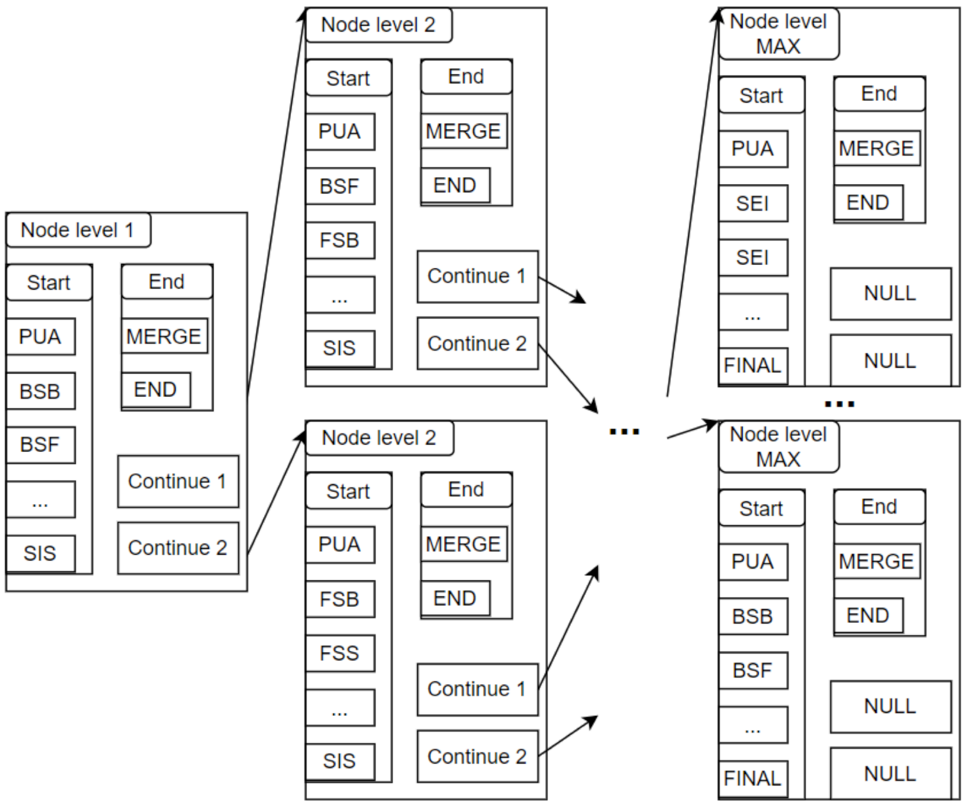


Figure 3: Chromosome in tree shape diagram.

Each node represents a part of the chromosome (sequence of genes) that corresponds to the sorting of a certain part of the array. Start and end gene sequences are stored separately. Child nodes are created when the array is partitioned and the parts of the array are sorted separately. For example, when an array is split into two equal parts, two descendant nodes are created. Each created node will have a part of the chromosome that implements the sorting of the corresponding part of the array. The end gene array will include a merge of sorted arrays and a current array sort finish gene, which includes closing curly braces and zeroing variables.

The generation of the algorithm from the chromosome should be based on the principle of the Depth-first traversal of the tree [20]. That is, for each node, the genes are in the following sequence:

1. start genes;
2. genes of the first child node;
3. genes of the second child node;
4. end genes.

The tree chromosome crossover operation is performed as a node exchange. To preserve the depth of the chromosome tree, the exchange can be carried out only between nodes of the corresponding level. However, due to the fact that the leaf nodes at the last level have a final sorting gene, which is guaranteed to sort the array, it is possible to exchange between nodes of different levels.

4. Experiments

The purpose of the experiment is to check the ability of the developed method and the corresponding software to form time-efficient sorting algorithms in different hardware and software environments. In the performed experiments, one component of the hardware and software environment was varied, namely the features of the data to be sorted.

4.1. Software instrumental means of the experiment

An algorithm using different combinations of fragments of basic algorithms can have a fairly large number of their calls. If they are called as separate functions in the program, overhead costs can significantly affect the overall time of the algorithm. An alternative option for creating a combined algorithm was applied – the generation of the program text. This approach minimizes the number of function calls.

The program developed for experiments consists of modules:

- implementation of the genetic algorithm;
- formation of the text of the sorting program by the coded chromosome;
- compilation of the formed combined sorting algorithm;
- sorting of data arrays with execution time measurement.

Compilation of the formed sorting algorithm is performed using an external script – a .bat file which:

- copies the .cpp file to the appropriate directory;
- performs compilation and assembly, resulting in the formation of an .exe file. Starts the execution of the generated combined sorting algorithm.

Since the median sorting time of one algorithm for many data is measured, the same set of input arrays for all combined and control standard algorithms is used for sorting.

The results of each experiment are recorded by the auxiliary program in a text file

4.1. Types of test data

Arrays of integers of various structures and volumes were created for conducting experiments.

Structure 1 – an array of random numbers. It was filled with random numbers in the range [0, N], (N is the length of the array).

Structure 2 – the array is fully sorted.

Structure 3 – the array is partially sorted. Sorting percentage. In a fully sorted array of length N, the unsorted percentage m is specified. The number of unsorted elements will be M=N*m. M is divided into k random numbers greater than or equal to 2, which will be written into the array K. Then, for each K[i], a random index j in the initial array is selected, such that j < N-K[i]. Next, within the subarray N[j, j+K[i]], K[i] permutations of randomly selected elements are performed.

Structure 4 – the array is partially sorted. Several permutations. In a fully sorted array of length N, two indices (i and j) are selected, and the elements located at these indices are swapped. The operation is repeated k times. The indicator k is calculated by the formula k=std::min(int(arrayLength/1e5), 3).

Structure 5 – the array is sorted in reverse order.

Experiments were performed on arrays with volumes ranging from one hundred thousand to one million elements in steps of one hundred thousand.

For each population, 51 arrays of sorted data of the appropriate structure were generated. They were stored in 51 binary files. Each individual of the population, which is a specific sorting algorithm, read the array of data from the file. Performed the sorting, checked that the sorting was completed successfully and saved the result (execution time) in an array. After sorting all input arrays, the median time was calculated and stored in a separate binary file. Thus, identical data were used for each individual.

4.3. Features of the genetic algorithm

The experiments were performed with the following parameters of the genetic algorithm:

- number of populations – 10;
- number of individuals in each population – 10;
- percentage of the best chromosomes that are transferred to the next population without changes – 30;
- the number of standard tested algorithms included in each population is 4: Quick sort, Merge sort, Insertion sort and Heapsort;
- final sorting – Quick sort.

In order to limit the length of the chromosome, and accordingly the length of the generated sorting function, the chromosome generation process was controlled using the following parameters:

- the maximum number of algorithms before splitting. This parameter determined the maximum number of genes (algorithms) upon reaching which the gene for the final sorting or partitioning of the array was guaranteed to be selected depending on the current depth of the chromosome tree. In the case of selecting a gene for the array partition, two child nodes were added to the current node of the chromosome, and the gene selection process continued for them with a depth increased by one;
- the maximum depth of the chromosome tree. At the beginning of the chromosome generation, the depth for the root node was zero. When selecting the array partitioning gene, two descendant nodes with a depth greater than that of the current node by one were added to the current node. If the depth is less than the maximum, the final sorting gene cannot be selected. Instead, the array partitioning gene is chosen. When the maximum depth

is reached, instead of the array partitioning gene, the final sorting gene is selected and the child nodes are not added. In this way, a balanced chromosome tree with a given depth is constructed.

5. Results of experimental research

In the course of the experimental study, the six best chromosomes were determined, which showed the smallest sorting time under the given conditions. The execution time of the sorting algorithms corresponding to the selected chromosomes was compared with the execution time of the "standard" sorting algorithms: Quick sort, Heapsort, Insertion sort and Merge sort.

Figure . 4 shows the results of the experiment for arrays of structure 1 with a length of one million elements. Quick sort has the best sort time with a median time of 56 ms. The next are algorithms based on the best chromosomes with execution time of 60-70 ms. The next time has a Heapsort – 96-97 ms. The penultimate result has Merge sort – 139-142 ms. And the worst result, as expected, was shown by Insertion sort, which has a quadratic complexity – 59-60 s. Due to the extremely large distance from the indicators of the rest of the algorithms, Insertion sort was not added to the graph. The same situation is observed on arrays of tested data with a length of 100,000-900,000. Quick sort always finishes first. Next, with indicators higher by 2-10%, there are algorithms formed by the best chromosomes. At the end of the list is a Heapsort, Merge sort and Insertion sort.

The chromosome with the best performance in the last population is SEI_ML_SIS_PUA_BSB_SIS_PUA_FS_ESS_PUA_SEEI_FSS_MR_FS_ESS_ESS_PUA_P_PUA_SEEI_MR_FSB_SEI_ML_FS_ESS_PUA_BSB_FS_ESS_ESS_ESS.

Chromosome decoding (each next level is an indentation – genes performed for the subarray after partitioning):

```
SingleElementSelection;
MergeLeft
SplitInSubarrays
    BubbleSortBackwards
    SplitInSubarrays
        // First part
        {FinalSort}
        // Second part
        {SingleElementEndInsertion
        FindSwapSmallest
        MergeRight
        FinalSort}
    Partition
        // First part
        {SingleElementEndInsertion
        MergeRight
        FindSwapBiggest
        SingleElementInsertion
        MergeLeft
        FinalSort}
        // Second part
        {BubbleSortBackwards
```

FinalSort}

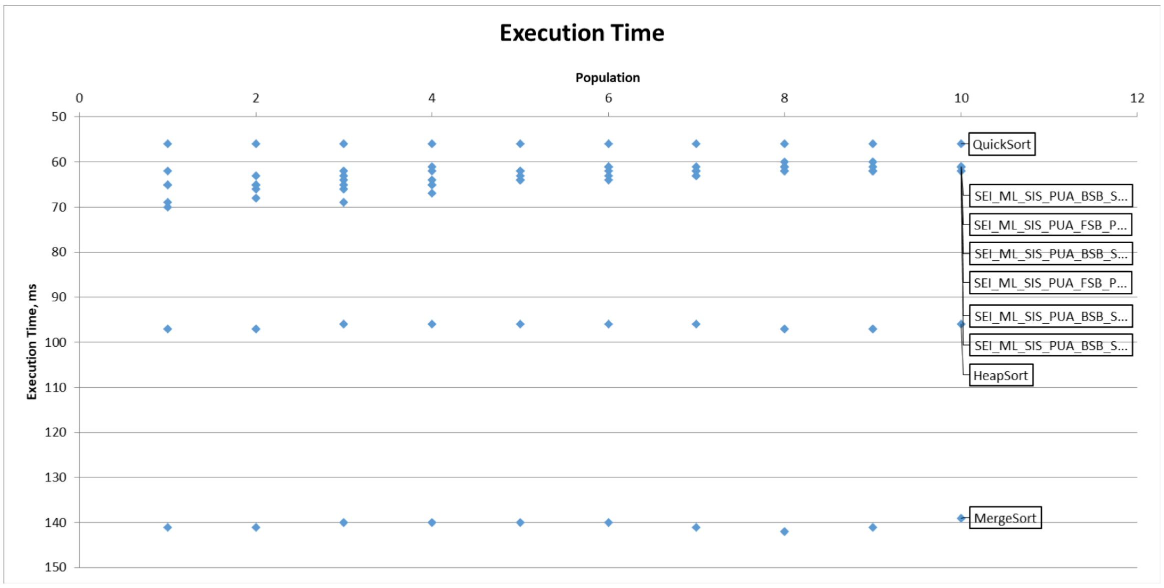


Figure 4: Median sorting time by structure 1 with different sorting algorithms.

The results of the experiment with data according to structure 2.

Some algorithms execution time is less than one millisecond, so for this data type all results are presented in microseconds. Algorithms based on the best chromosomes starting with BSB (BubbleSortBackwards) and BSF (BubbleSortForward) genes turned out to be the fastest on small amounts of tested data. The speed is achieved by the fact that the algorithm makes one pass through the array in the forward (BSF) or reverse (BSB) direction, checking all pairs of elements. Since the array is sorted, no permutation is performed and the execution of the function ends immediately. Next, with the worst time indicators, there are standard algorithms in the following order: Insertion sort, Quick sort, Heapsort, and Merge sort.

Table 1
Median sorting time by structure 2 with a volume of 100000 elements in microseconds

Algorithms for the best chromosomes						Control standard algorithms			
Chromosome 1	Chromosome 2	Chromosome 3	Chromosome 4	Chromosome 5	Chromosome 6	Quick sort	Heap sort	Merge sort	Insertion sort
23	23	24	25	29	34	752	4112	7867	44

We can observe the same results on arrays of medium length.

Table 2
Median sorting time by structure 2 with a volume of 500000 elements in microseconds

Algorithms for the best chromosomes						Control standard algorithms			
Chromosome 1	Chromosome 2	Chromosome 3	Chromosome 4	Chromosome 5	Chromosome 6	Quick sort	Heap sort	Merge sort	Insertion sort
117	138	140	151	152	154	3962	22970	41716	229

However, a slightly different picture is observed on large arrays.

Table 3
Median sorting time by structure 2 with a volume of 1000000 elements in microseconds

Algorithms for the best chromosomes						Control standard algorithms			
Chromosome 1	Chromosome 2	Chromosome 3	Chromosome 4	Chromosome 5	Chromosome 6	Quick sort	Heap sort	Merge sort	Insertion sort
584	603	607	620	621	651	8322	46539	82075	484

The best time shows the Insertion sort. Then there are algorithms that are formed according to the best chromosomes. They all start with the SEI_BSF_ML gene sequence. That is, first an attempt is made to insert the second element into the sorted subarray on the left. Because of this, the indices increase and BSF will not process the entire array. The merge left (ML) helper function will then be called. In this way, redundant operations will be performed, which will increase the total time of the algorithm. Because the initial population did not have chromosomes starting with the BSB or BSF genes, and they were not generated when each new population was constructed, the best-chromosome-based algorithms performed worse in time than the standard Insertion sort.

For arrays according to structure 3, the results are quite similar for all array volumes. Quick sort is always better than others. The next by time efficiency are algorithms formed according to the best chromosomes, in which genes for split into subarrays (SIS) or Partition (P) are located at the beginning. Standard sorting algorithms have the largest time indicators: Heapsort, Merge sort and Insertion sort.

Table 4
Median sorting time by structure 3 with a volume of 100000 elements in milliseconds

Algorithms for the best chromosomes						Control standard algorithms			
Chromosome 1	Chromosome 2	Chromosome 3	Chromosome 4	Chromosome 5	Chromosome 6	Quick sort	Heap sort	Merge sort	Insertion sort
17	17	17	17	17	17	15	51	82	105

For arrays according to structure 4 with a length of one million, 10 permutations were performed. The results of the experiment are shown in Table 5

Table 5
Median sorting time by structure 4 with a volume of 1000000 elements in milliseconds

Algorithms for the best chromosomes						Control standard algorithms			
Chromosome 1	Chromosome 2	Chromosome 3	Chromosome 4	Chromosome 5	Chromosome 6	Quick sort	Heap sort	Merge sort	Insertion sort
15	15	16	16	17	18	13	50	83	2

Insertion sort has the best result among all populations. Next comes Quick sorting. A little less efficiency (by 10-20%) was shown by constructed algorithms that were formed according to the best chromosomes. Heapsort and Mergesort turned out to be the worst.

On the different volumes of tested data arrays by structure 5 different chromosomes were selected. However, the general picture is the same for all data volumes

Table 6
Median sorting time by structure 5 with a volume of 1000000 elements in milliseconds

Algorithms for the best chromosomes						Control standard algorithms			
Chromosome 1	Chromosome 2	Chromosome 3	Chromosome 4	Chromosome 5	Chromosome 6	Quick sort	Heap sort	Merge sort	Insertion sort

15	16	16	17	17	19	10	49	91	121438

The best time has the Quick sort. The next by time efficiency, with a significant lag (50-90%), there are algorithms based on the best chromosomes. And Heapsort, Merge sort and Insertion sort have significantly worse time efficiency under the given conditions.

6. Conclusions

Modern information technologies are based on processing large volumes of data. At the same time, the task of developing and applying effective algorithms for data processing, in particular sorting, remains relevant.

A genetic algorithm for structural adaptation of sorting algorithms was developed. Its use allows to carry out structural adaptation and choose the most effective and adapted to stable usage environments.

The presented approach to the formation of a chromosome in the form of a tree allows solving existing problems. It allows you to efficiently form chromosomes, the result of decoding of which will be an algorithm that is guaranteed to perform sorting.

The formalized model of the formation of sorting algorithms by means of constructive-synthesizing modeling is quite voluminous and will be presented in a separate article.

The developed software should be applied to adaptation of sorting algorithms to stable input data streams and usage environments.

The ability of the developed method and the corresponding software to form time-efficient sorting algorithms in various hardware and software environments has been experimentally confirmed.

References

[1] 20 Issues Tech Companies Are Facing Now (And How To Address Them). URL: https://www.forbes.com/sites/forbestechcouncil/2023/09/07/20-issues-tech-companies-are-facing-now-and-how-to-address-them/

[2] V. Shynkarenko, P. Ilchenko, H. Zabula Tools of investigation of time and functional efficiency of bionic algorithms for function optimization problems, in Proceedings of the 11th International Conference of Programming (UkrPROG 2018), Kyiv, 2018, CEUR-WS Team, 2139, 270–280.

[3] Garcia-Molina, H., Ullman, J. D., & Widom, J. (2008). Database Systems: The Complete Book (2nd ed.). Pearson.

[4] O. Mendelevitch, C. Stella, D. Eadline, Practical Data Science with Hadoop and Spark, Addison-Wesley, 2016.

[5] H.H. Aung Analysis and Comparative of Sorting Algorithms, International Journal of Trend in Scientific Research and Development 3(5), 2019, 1049-1053. doi:10.31142/ijtsrd26575

[6] M. R.Choudhury, A. Dutta, (2022). Establishing pertinence between Sorting Algorithms prevailing in n log (n) time, J Robot Auto Res 3 (2), 2022, 220-226. doi: 10.21203/rs.3.rs-1754555/v1

[7] D. Knuth, The Art Of Computer Programming, vol. 3: Sorting And Searching, Addison-Wesley, 1973.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein Introduction to algorithms (3rd ed.). Cambridge, MA: The MIT Press. 2009.

[9] A. S. Mohammed, Ş. E. Amrahov, F. V. Çelebi, (2017). Bidirectional Conditional Insertion Sort algorithm; An efficient progress on the classical insertion sort. Future Generation Computer Systems, 71, 2017,102-112. doi: 10.1016/j.future.2017.01.034

[10] S. Kushagra, A. López-Ortiz, A. Qiao, J. I. Munro Multi-pivot quicksort: theory and experiments. Proc. Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX), 2014, 47–60.
[11] R. Mansi Enhanced quicksort algorithm. Int. Arab J. Inf. Technol 7(2), 2010, 161–166.
[12] R. Yadav, R. Yadav, S. B. Gupta Comparative study of various stable and unstable sorting algorithms. in Artificial Intelligence and Speech Technology. CRC Press, 2021, 463–477.
[13] Timsort. URL: https://svn.python.org/projects/python/trunk/Objects/listsort.txt.
[14] D.R. Musser, Introspective Sorting and Selection Algorithms, Software: Practice and Experience, 1997, № 27 (8), pp.983–993. doi: 10.5555/261387.261395.
[15] A. N Frak. et al. Comparison study of sorting techniques in static data structure, International Journal of Integrated Engineering: Special Issue Data Information Engineering 10(6), 2018, 106–112. doi: 10.30880/ijie.2018.10.06.014
[16] V.I. Shinkarenko, A.Yu. Doroshenko, O.A. Yatsenko, V.V. Raznosilin, K.K. Galanin, Data Stochastic Preprocessing for Sorting Algorithms, CEUR in: Proceedings of the 13th International Scientific and Practical Programming Conference UkrPROG 2022, vol. 3501, 29-38.
[17] A. Doroshenko, O. Yatsenko Formal and adaptive methods for automation of parallel programs construction: emerging research and opportunities. Hershey: IGI Global, 2021.
[18] J.H. Holland, Adaptation in Natural and Artificial Systems, The MIT Press, 1992.
[19] D. Whitley, An Overview of Evolutionary Algorithms: Practical Issues and Common Pitfalls, Journal of Information and Software Technology, vol. 43, no. 14, pp. 817-831, 2001.
[20] R. Tarjan, Depth-First Search and Linear Graph Algorithms, SIAM Journal on Computing, 1972, № 1(2). doi: 10.1137/0201010.