DOI: 10.15514/ISPRAS-2026-38(4)-17

On the Efficiency of Bounded Multi-Source Shortest Path Algorithm

R.S. Gromov, ORCID: 0009-0009-6622-7359 <rsgromov@edu.hse.ru>
R.A. Nesterov, ORCID: 0000-0002-4162-09070 <rnesterov@hse.ru>

National Research University Higher School of Economics,
11, Pokrovsky Boulevard, Moscow, 109028, Russia.

Abstract. This paper explores the performance criteria of the newest algorithm for solving the problem of finding shortest paths on a graph from a given vertex – Bounded Multi-Source Shortest Path Algorithm (BM-SSP). The algorithm was published in 2025 and, as its creators claim, it is asymptotically superior to Dijkstra's deterministic algorithm. However, in the publication devoted to this algorithm, only a theoretical asymptotic analysis of the execution time was given, and not a single benchmark was provided that would prove its practical effectiveness. This study should identify and substantiate the conditions under which the BM-SSP algorithm demonstrates superior efficiency (in terms of execution time and resource consumption) over classical algorithms for finding shortest paths on graphs of various structures. These hypotheses should be proved or disproved by the results of benchmarks that will be conducted on the test infrastructure using a built-in load framework for testing various graphs.

Keywords: Graphs; Shortest paths; Bounded Multi-Source Shortest Path algorithm; Dijkstra algorithm; Time complexity analysis.

For citation: Gromov R.S., Nesterov R.A. On the Efficiency of Bounded Multi-Source Shortest Path Algorithm. Trudy ISP RAN/Proc. ISP RAS, 2026, vol. 38, issue 4, part 2, pp. 23-44. DOI: 10.15514/ISPRAS-2026-38(4)-17.

Acknowledgements. This article is an output of a research project (HSE-BR-2025-024) implemented as part of the Basic Research Program at HSE University.

Об эффективности алгоритма поиска кратчайших путей на графе из множества стартовых вершин

Р.С. Громов, ORCID: 0009-0009-6622-7359 <rsgromov@edu.hse.ru>
Р.А. Нестеров, ORCID: 0000-0002-4162-09070 <rnesterov@hse.ru>

Национальный исследовательский университет "Высшая школа экономики",
Россия, 109028, г. Москва, Покровский бульвар, д. 11.

Аннотация. В статье исследуются критерии эффективности новейшего алгоритма для решения задачи поиска кратчайших путей на графе из заданной вершины – BM-SSP. Алгоритм был опубликован в 2025 году и, как утверждают его создатели, асимптотически превосходит детерминированный алгоритм Дейкстры. Однако в публикации, посвященной этому алгоритму, был дан только теоретический асимптотический анализ времени выполнения, и не было приведено ни одного бенчмарка, который доказал бы его практическую эффективность. Это исследование должно выявить и обосновать условия, при которых алгоритм BM-SSP демонстрирует превосходящую эффективность (с точки зрения времени выполнения и потребления ресурсов) по сравнению с классическими алгоритмами поиска кратчайших путей на графах различной структуры. Эти гипотезы должны быть подтверждены или опровергнуты результатами бенчмарков, которые будут проводиться на тестовой инфраструктуре с использованием разработанного фреймворка нагрузки для тестирования различных графиков.

Ключевые слова: графы; кратчайшие пути; поиск кратчайших путей из множества стартовых вершин; алгоритм Дейкстры; анализ временной сложности.

Для цитирования: Громов Р.С., Нестеров Р.А. Об эффективности алгоритма поиска кратчайшего пути на графе из множества стартовых вершин. Труды ИСП РАН, 2026, том 38, вып. 4, часть 2, стр. 23–44 (на английском языке). DOI: 10.15514/ISPRAS–2026–38(4)–17.

Благодарности. Исследование осуществлено в рамках Программы фундаментальных исследований НИУ ВШЭ (HSE-BR-2025-024).

1. Introduction

The search for shortest paths on graphs is characterized by mature classical approaches and an active quest for superior solutions. For over half a century, Dijkstra and Bellman-Ford algorithms have remained fundamental in academic and industrial systems due to their efficiency and ease of implementation across various IT fields [1–4]. However, as data scales increase in modern distributed systems, even minor asymptotic improvements yield significant performance gains, driving the development of new algorithms that challenge long-standing barriers.

This study's relevance stems from both practical needs and a theoretical breakthrough in late 2025. Duan et al.'s work, "Breaking the Sorting Barrier for Directed Single-Source Shortest Paths" [5], introduced a new deterministic "Bounded Multi-Source Shortest Paths" (BM-SSP) algorithm. It reportedly improves asymptotic complexity for directed graphs with non-negative weights, challenging the assumption that Dijkstra's algorithm is near-optimal for general graphs.

Existing reviews are partial and fragmented, testing BM-SSP only on limited graph sets using desktop environments [6-7]. This study aims to formulate and verify hypotheses regarding the algorithm's effectiveness through comprehensive benchmarks on a specialized load-testing framework. We also provide an analysis and detailed illustrations to enhance understanding of the BM-SSP algorithm [5]. It is important to note that we are not trying to challenge the theoretical effectiveness of the BM-SSP algorithm, we try to understand on which graphs it is more efficient than its analogues and in which cases it is advisable to use it.

This article is structured into five sections. Section 1 outlines the historical context, substantiates the study's relevance, and defines the goal: theoretical analysis and experimental verification of BM-SSP. Section 2 systematizes deterministic approaches, including heap optimizations (Binary,

Fibonacci, Radix), and analyzes Duan et al.'s key article. Section 3 explains the algorithm's hybrid nature, combining Bellman-Ford and Dijkstra ideas via recursive partitioning, detailing its divide-and-conquer technique, FindPivots procedure, and Blocking-based heap. Section 4 formulates assumptions about graph classes where BM-SSP should outperform classical solutions, describes graph types and generation parameters, and presents benchmark results with conclusions. "Section 5" summarizes findings, confirms or refutes hypotheses, and outlines future optimization directions.

2. Related works

The history of shortest path algorithms begins with three fundamental methods released between 1958 and 1962: Bellman-Ford, BFS, and Floyd-Warshall. E. F. Moore's "The Shortest Path Through a Maze" [8] introduced BFS, the first algorithm for traversing unweighted graphs in linear time $O(V+E)$. While efficient, BFS cannot handle weighted directed graphs, which are critical for most graph theory problems. To address this, R. Bellman's "On a Routing Problem" [9] introduced the Bellman-Ford algorithm, using dynamic programming to find paths in graphs with potentially negative weights in $O(V \cdot E)$ steps, while also detecting negative cycles. However, Bellman-Ford does not solve the All-Pairs Shortest Path (APSP) problem. For this, R. W. Floyd developed the Floyd-Warshall algorithm in 1962 [10]. Also based on dynamic programming, it finds shortest paths between all vertex pairs with $O(V^3)$ complexity by iteratively improving distance estimates via intermediate vertices.

But the most fundamental and basic deterministic algorithm is Dijkstra's algorithm, which was published in 1959 in an article [11]. In this paper, the method opposite to dynamic programming is used - the greedy algorithm, thanks to which Dijkstra finds the shortest paths from one vertex to all the others in graphs with non-negative edge weights. The standard implementation of Dijkstra's algorithm uses an adjacency matrix, which is why the asymptotic complexity of the algorithm reaches $\boldsymbol{O(V^2)}$ and it is considered slow. In order to improve the time complexity of the standard Dijkstra implementation, an article - *"Efficient Algorithms for Shortest Paths in Sparse Networks"* by Johnson D.B. was published 18 years later. The author showed that using Binary heap (complete binary tree that satisfies the ordering property) reduces the complexity of the algorithm from $\boldsymbol{O(V^2)}$ (typical for implementation on an adjacency matrix or array) to $\boldsymbol{O\big(E * \log(V)\big)}$, which makes the algorithm efficient for sparse graphs. Another stronger optimization was presented in the article by Fredman, R. E. Tarjan [12], which introduces a new data structure, the Fibonacci heap (a data structure with amortized complexity that performs the operation of adding and decreasing a key in $\boldsymbol{O(1)}$, and extracting the minimum in $\boldsymbol{O\big(\log(n)\big)}$). Thanks to this structure it is possible to find edges with the lowest weight more efficiently. This optimization helped reduce Dijkstra's asymptotics to $\boldsymbol{O\big(E + V * \log(V)\big)}$. Later, another article [13] was released, which suggested another optimization of Dijkstra's algorithm. A more universal structure was introduced – the Radix Heap (uses a bitwise organization of elements to achieve amortized complexity of O(1) for the key reduction operation and $\boldsymbol{O(log(C))}$ for extracting the minimum) and the range of integer weights C, thanks to which the asymptotic can be further reduced to $\boldsymbol{O\big(E + V * \sqrt{log(C)}\big)}$.

A new paper [5] introduced a BM-SSP. It consists of 3 main parts. The "introduction" introduces existing algorithms that solve the problem of objects; a review of the literature; a brief overview of the BM-SSP. The "preliminary" sections describe the constraints that are imposed on graphs so that the BM-SSP algorithm can most effectively solve the problem of objects. The "Main Result" displays the shared files associated with the recursive bmssp program, the FindPivots program, and the lock-based heap. In addition, this chapter provides an analysis of the correctness of the algorithm and an asymptotic analysis of time complexity. At the moment, there are only two papers leading benchmarks of the BM-SSP algorithm [6-7]. In these articles, benchmarks are conducted mainly on randomized sparse graphs and triangular and quadratic tiling graphs.

3. BM-SSP overview

3.1 Basis of algorithm

Before starting an overview of the new BM-SSP algorithm for solving the SSSP problem, it is worth introducing basic definitions and concepts that underlie the theoretical basis of this algorithm. Let us say we have a directed graph with non-negative real edges $G = (V, E)$, where V is the set of vertices and E is the set of edges. BM-SSP, like other deterministic algorithms solving the SSSP problem, must find the shortest paths from some vertex s, called the source, to all other vertices of the graph. The main result of these algorithms is an array of shortest distances $dist = \{d(v) | v \in V \wedge d(v) \in \mathbb{R}\}$. If the vertex v is reachable from the source s, then $d(v) < \inf$, otherwise $d(v) = \inf$. It is also important to note that for algorithms for solving the SSSP problem, the key metric is the calculated (estimated) distance to the vertex - $\hat{d}(v) \geq d(v)$, which is initially equal to inf and which tends to the true distance $d(v)$, considered unknown. Since we initially have no information about $d(v)$, some algorithms (for example, Dijkstra, BM-SSP) are based on an invariant, when we can precisely determine whether we have obtained an accurate estimate of the distance to the vertex or not. In other words, we can call the vertex complete or incomplete. By definition $v \in V_{complete} \Leftrightarrow \hat{d}(v) = d(v)$ and $v \in V_{incomplete} \Leftrightarrow \hat{d}(v) \geq d(v)$.

To maintain an invariant in the shortest path finding algorithms, including the classical Dijkstra algorithm and the modern BM-SSP, two important sets are conceptually distinguished. The set of completed vertices is the kernel of U and the set of vertices under active processing is the frontier of S. In BM-SSP notation, they are formalized as follows: $U = \{u \in V | \hat{d}(v) = d(v)\}$ and $S = \{v \in V | b \leq \hat{d}(v) \leq B\}$, where $b = \max_{u \in U} \hat{d}(u)$, and B is the upper bound of processing at this iteration of the algorithm. The U kernel contains complete vertices; they are excluded from further relaxation. The frontier S forms a set of vertices with intermediate estimates, which are processed and, as the algorithm runs, pass into U. The algorithm is implicitly considered completed when $|U| = V_{reach}$ for a given graph G. As a result, we can present the basic definitions and sets in the form of Table 1.

Table 1. Basic definitions and concepts.

Definition	Description
V	The set of all vertices of the graph $G = (V, E)$
E	The set of all edges of the graph $G = (V, E)$
V_{reach}	The set of reachable vertices from the source s
$d(v)$	The true shortest distance from s to v
$\hat{d}(v)$	Current distance estimate from s to v
U (Kernel)	$\{v\, V \mid \hat{d}(v) = d(v)\}$
S (Frontier)	$\{v \in V \mid b \leq \hat{d}(v) < B\}$
b	$\max_{u \in U} \hat{d}(u)$ - the upper limit of the kernel
B	the upper bound for processing graph vertices $G = (V, E)$

Now that we have introduced the basic definitions and terms, we can begin to overview the BM-SSP algorithm directly. As stated by the authors of this algorithm: *"There exists a deterministic algorithm that takes* $\boldsymbol{O\left(m * log^{2/3}(n)\right)}$ *time to solve the single source shortest path problem on directed graphs with real non-negative edge weights"*, where $\boldsymbol{m = |E|}$ – the number of edges, and $\boldsymbol{n = |V|}$ – the number of vertices in a graph $\boldsymbol{G = (V, E)}$. The main problem that the BM-SSP algorithm solves is the presence of the sorting barrier $\boldsymbol{\Omega\big(n * \log(n)\big)}$ in the most efficient implementations of the Dijkstra algorithm, which is due to the fact that most of them use a priority

queue and maintain the full order of the relaxed vertices in it. The authors of the BM-SSP algorithm show that in order to effectively solve the SSSP problem, it is not necessary to maintain the full order of vertices, it is enough to process vertices in batches in limited intervals, each time recursively reducing the size of the frontier.

In order to solve the sorting barrier problem in practice, the BM-SSP algorithm is based on the Divide-and-Conquer strategy. The algorithm splits the initial set $\boldsymbol{V}$ into no more than $\boldsymbol{2^t}$ pieces of a similar size: $\boldsymbol{V = U_1 \cup U_2 \cup \ldots \cup U_{2^t}}$, where any true distance is $\boldsymbol{d(u_i)}$ of an element from a disjoint set $\boldsymbol{U_i}$ is less than any true distance $\boldsymbol{d(u_j)}$ of an element from a disjoint set $\boldsymbol{U_j}$ for $\boldsymbol{i < j}$. As a part of each new recursive call to the BM-SSP algorithm, it is planned to reduce the frontier $\boldsymbol{S}$ - the set of nearest relaxed vertices, thanks to the *FindPivots* procedure and the *Pull* method of the Block-based heap. Using these approaches together with the Block-based heap, as well as specially selected parameters $\boldsymbol{k = \log^{1/3}(n)}$ and $\boldsymbol{t = \log^{2/3}(n)}$, which regulate the depth of recursion and the number of recursive calls, gives the BM-SSP algorithm an asymptotic gain compared to Dijkstra, as discussed in detail in Section 3.4 of the original article [5].

3.2 Kernel and Pivots

The basis of the BM-SSP algorithm is set by Lemma 3.1 in the original paper [5]: *"We are given an integer level* $l \in \left[0, \frac{\lceil log(n) \rceil}{t}\right]$*, a set of vertices S of size* $\leq 2^{lt}$*, and an upper bound* $B > max_{x \in S}\hat{d}[x]$*. Suppose that for every incomplete vertex* v *with* $d(v) < B$*, the shortest path to* v *visits some complete vertex* $u \in S$*. Then we have an sub-routine BMSSP(l, B, S) (Algorithm 3) in* $O\left(\left(k * l + \frac{t*l}{k} + t\right) * |U|\right)$ *time that outputs a new boundary* $B' \leq B$ *and a vertex set U that contains every vertex v with* $d(v) < B'$ *and the shortest path to v visits some vertex of S. At the end of the sub-routine, U is complete. Moreover, one of the following is true: Successful execution* $B' = B$ *or Partial execution due to large workload* $B' < B$*, and* $|U| = \Theta(k * 2^{lt})$*"*. In other words, it tells us that that there exists an algorithm, asymptotically faster than Dijkstra, that solves the shortest path problem.

For clarity, the generalized BM-SSP algorithm can be represented by the flowchart in Fig. 1.The BMSSP procedure has two possible execution paths - recursive when $l > 0$ and basic when $l = 0$. When $l = 0$, the frontier consists of only one complete vertex, $\{x\}$, and the task boils down to a modified Dijkstra algorithm, which differs from the standard one only in that it monitors the number of vertices that have been removed from the priority queue. If the number of deleted vertices exceeds $k + 1$, the algorithm terminates its work. This is necessary so that, in the basic case, the BMSSP does not degrade asymptotically to the usual Dijkstra and has an asymptotic complexity of $O(k * \log(k))$. In the recursive execution of the BMSSP procedure, when $l > 0$, the *FindPivots* procedure is called first. It is needed in order to reduce the size of the frontier transferred to the BMSSP, avoid working with the full set of S and, as a result, avoid the sorting barrier of $\Omega(n * \log(n))$. At the exit from *FindPivtos*, we get a set of "pivots", which are those vertices of the frontier whose shortest path trees maximally cover all other vertices for the current BMSSP call limited by the parameter B. This is a key place in the algorithm, thanks to which an asymptotic gain is possible. After that, the algorithm initializes a specialized Block-based data structure D with the block size parameter $M = 2^{(l-1)*t}$ and inserts all pivots' estimated distances into it.

This data structure replaces the usual priority queue of Dijkstra's algorithm so that we can efficiently extract minimal elements without completely sorting the entire set. After that, the main while loop of the algorithm is executed until the size of the kernel U exceeds the value of $k * 2^{lt}$ for asymptotic reasons or until the Block-based heap is empty (the shortest distances to all vertices are found).

At the beginning of the cycle, we extract the batch of pivots with the smallest estimated distances from the Blocking-based heap. This helps to implicitly divide our frontier into two areas - the area of the nearest vertices and the rest, the more distant part.

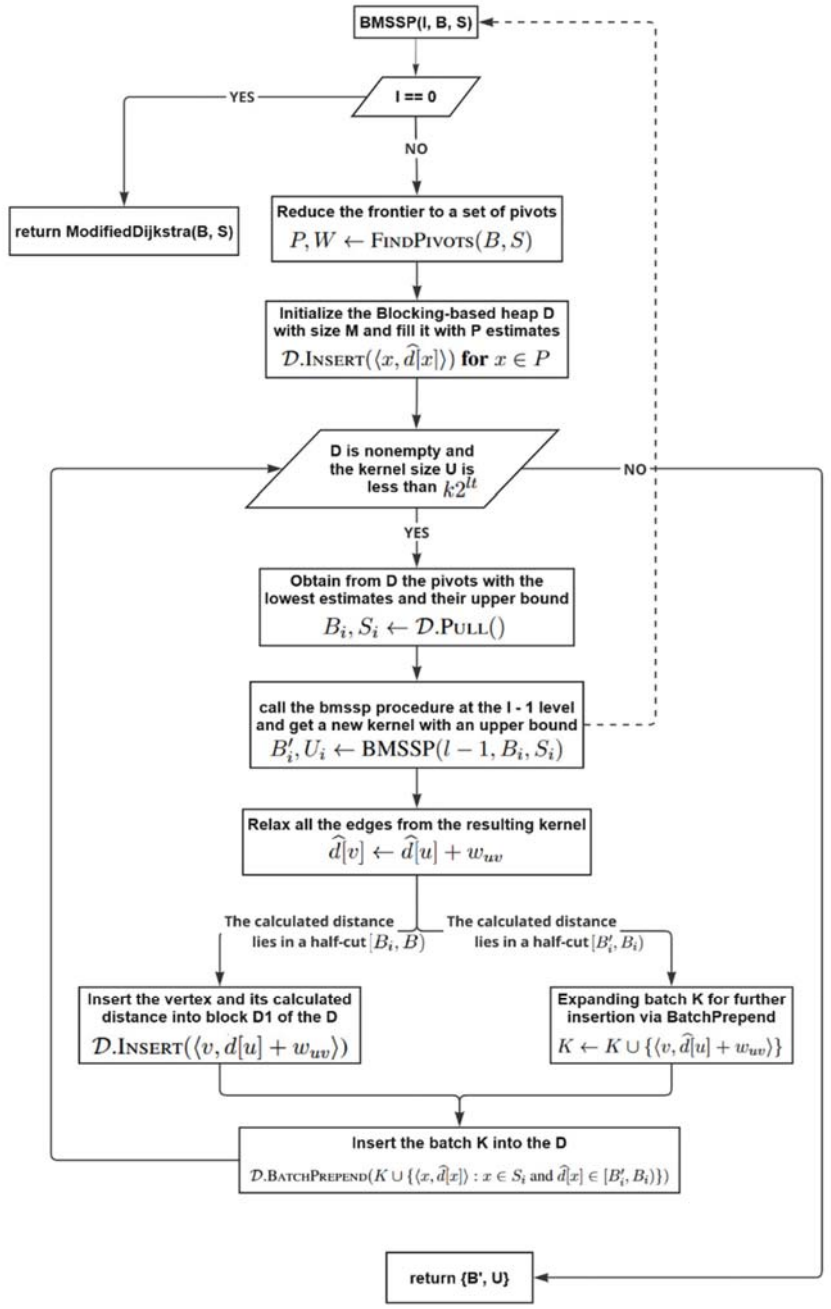

Fig. 1. Generalized scheme of the algorithm.

The most important thing for the algorithm is the nearest vertices, since their use in deeper calls to the BMSSP procedure will contribute to a faster increase in the kernel. The two figures (see Fig. 2 and Fig. 3) below show examples of how graph sets that differ in semantics change before and after extracting pivots from the block heap.

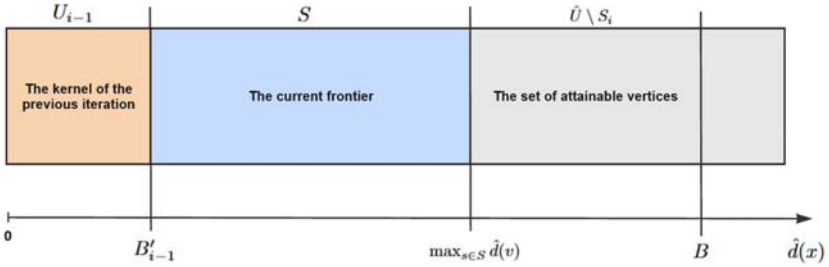

Fig. 2. The i-th iteration of the BMSSP loop, before the Pull operation.

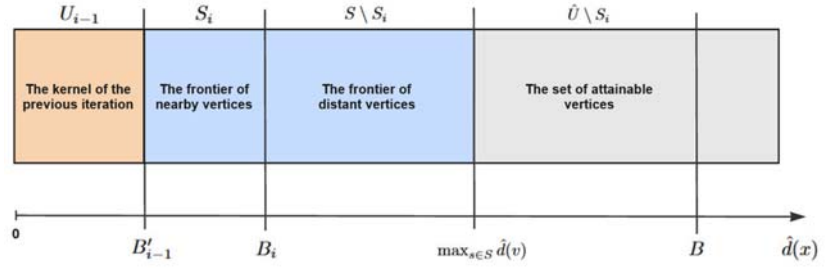

Fig. 3. The i-th iteration of the BMSSP loop, after the Pull operation.

At each $\boldsymbol{i}$-th iteration, we have an existing set of complete vertices, the kernel, which grows both with increasing recursion. Each new call to the BMSSP procedure adds a certain number of vertices to our common kernel $\boldsymbol{U}$. Based on Lemma 3.6 and Lemma 3.7 in [5], the algorithm guarantees that it will eventually process all reachable vertices and that $\boldsymbol{U}$ will contain all vertices from $\boldsymbol{V}$. Below is an illustration (see Fig. 4) of what the graph's vertex sets look like after the $\boldsymbol{l-1}$ level BMSSP procedure is completed. After the new $\boldsymbol{U_i}$ kernel has been attached to the $\boldsymbol{U}$ kernel, we begin to relax all those vertices that are connected to the $\boldsymbol{U_i}$ kernel (see Fig. 4). This is necessary in order to increase the content of the Blocking-based heap, adding to it those vertices that in the future may be most effective for subsequent calls to the BMSSP procedure. If the new relaxed distance at the vertex falls into the segment of the nearest vertices - $[\boldsymbol{B'_i}, \boldsymbol{B_i})$, this means that we can put this distance in the set $\boldsymbol{K}$ for further batch insertion. This set contains all nearest vertices, which are most advantageous to use in subsequent BMSSP. The remaining relaxed vertices are placed individually in the area of the Block-based heap that is designated for the farthest vertices of the frontier.

Separately, we need to consider the key procedure that contributes to the greatest performance gain compared to Dijkstra - the *FindPivots* procedure, the foundations of which are laid in Lemma 3.2 in [5]: *"Suppose we are given a bound B and a set of vertices S. Suppose that for every incomplete vertex v such that $\hat{d}(v) < B$, the shortest path to v visits some complete vertex $u \in S$. Denote by* Ũ *the set that contains every vertex v such that $\hat{d}(v) < B$ and the shortest path to v passes through some vertex in S. The sub-routine $FindPivots(B,S)$ (Algorithm 1) finds a set $W \subseteq$ Ũ of size $O(k|S|)$ and a set of pivots $P \subseteq S$ of size at most $|W|/k$ such that for every vertex $x \in$ Ũ, at least one of the following two conditions holds: 1) At the end of the sub-routine, $x \in W$ and x is complete; 2) The shortest path to x visits some complete vertex $y \in P$".*

In this procedure, a new set W is introduced, which is necessary to accumulate all the vertices processed in k relaxation steps in order to divide them into two categories: those whose shortest paths have already been found and which are excluded from further recursion, and the roots of large subtrees (pivots), used for guaranteed size reduction the front of active vertices.

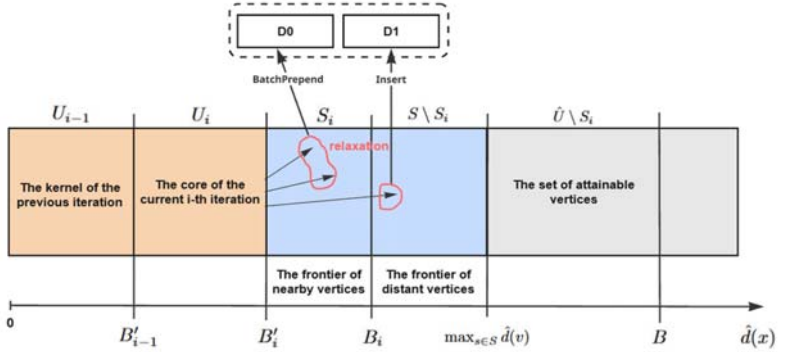

Fig. 4. The i-th iteration of the BMSSP loop, after recursively calling the l - 1 level BMSSP.

At the beginning of the procedure, the sets W and W_0 are initialized as the initial frontier, after which a cycle is started that performs exactly k steps of Bellman-Ford relaxation. At each iteration of i, the algorithm looks at all outgoing edges from the vertices added in the previous step W_{i-1}. If, upon relaxation of the edge (u, v), the new estimate of the distance $d[u] + w_{uv}$ turns out to be no greater than the current $\hat{d}[v]$ and at the same time strictly less than the boundary of B, then the vertex v is considered achievable within the current range and is added to the set W_i and the total set W. Also, during the iteration, the algorithm checks the size of W; if $W > k|S|$, the algorithm is interrupted. This means that the number of processed vertices has grown disproportionately fast, and further allocation of pivots will not give an asymptotic gain. Below is an illustration, shown in Fig. 5 and Fig. 6, of how the procedure works on the graph for early exit.

If the loop has completed all $\boldsymbol{k}$ iterations, the algorithm builds a directed forest $\boldsymbol{F}$ consisting of edges $(\boldsymbol{u}, \boldsymbol{v})$ with both ends lying in $\boldsymbol{W}$ and for which the exact relaxation condition $\widehat{\boldsymbol{d}}(\boldsymbol{v}) = \widehat{\boldsymbol{d}}(\boldsymbol{u}) + \boldsymbol{w_{uv}}$. In this forest, each tree corresponds to a part of the shortest path starting from some vertex of $\boldsymbol{S}$. The algorithm looks at the roots of these trees belonging to the original set $\boldsymbol{S}$, and selects only those roots in the set of pivots $\boldsymbol{P}$ whose size of the corresponding subtree in $\boldsymbol{F}$ is at least $\boldsymbol{k}$ vertices. Thus, only those sources of paths that cover significant areas of the graph fall into $\boldsymbol{P}$. The peculiarity of this procedure is that any vertex $\boldsymbol{x}$ whose shortest path from the source passes through $\boldsymbol{S}$ and has a length of $< \boldsymbol{B}$, either it has already become completed and got into $\boldsymbol{W}$, or its path necessarily passes through some completed vertex from the selected set of pivots $\boldsymbol{P}$.

The whole procedure is performed in time $\boldsymbol{O(k * \min\{k|S|, Ũ\})}$, where Ũ is the number of all vertices of interest with $\boldsymbol{d(v) < B}$ whose paths pass through $\boldsymbol{S}$. This allows the BMSSP algorithm to significantly reduce the size of the active part of the frontier, which is the foundation for bypassing the sorting barrier.

3.3 Block-Based Heap

The basis of the Block-based Heap data structure (see Fig. 7) is laid in Lemma 3.3 in [5]: "*Given at most N key/value pairs to be inserted, an integer parameter M, and an upper bound B on all the values involved, there exists a data structure that supports the following operations:* $Insert\left(v, \hat{d}(v)\right)$, $BatchPrepend\left(\left\{\left(v_1, \hat{d}(v_1)\right), \ldots, \left(v_k, \hat{d}(v_k)\right)\right\}\right)$ *and Pull*".

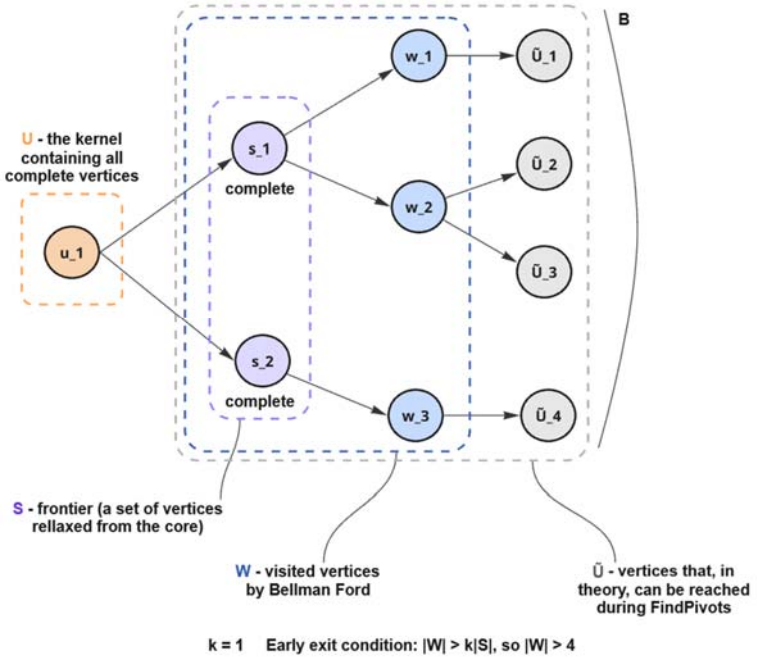

Fig. 5. FindPivots procedure with early exit.

The key task of this structure is to avoid maintaining the complete order of all elements of the frontier. Instead of a huge sorted set, vertices are stored as a sequence of blocks, each of which is a doubly linked list containing no more than M pairs of $< v, \hat{d}(v) >$. Blocking-based heap are divided into two unordered sequences: D_0 and D_1. D_0 accumulates vertices added in a batch manner through the $BatchPrepend$ operation, whereas D_1 contains vertices inserted singly through the $Insert$ operation. The blocks within each sequence are strictly ordered in ascending order of the maximum values of the elements, which allows the algorithm to find areas of the graph with the smallest current estimated distances.

For efficient navigation through the sequence D_1, a balanced binary search tree (for example, a red-black tree) is used, which stores the upper bounds of the values of each block. When calling $Insert$, the algorithm finds a suitable block using a tree search in amortized time $O\left(\max\left(1, log\left(\frac{N}{M}\right)\right)\right)$, adds a new pair to the end of the linked list, and checks the block size. If the number of elements exceeds the M limit, the $Split$ procedure is started, which finds the median of the elements of the current block in linear time $O(M)$, splits the list into two new blocks of approximately equal size, and updates the boundary tree (see Fig. 8). Thanks to this mechanism, the number of blocks in D_1 always remains proportional to $O(N/M)$, where N is the total number of inserts.

The $\boldsymbol{BatchPrepend}$ operation is designed to quickly insert a large number of vertices whose estimated distances are obviously less than all the current elements in the structure. If the number of inserted elements $\boldsymbol{L}$ does not exceed $\boldsymbol{M}$, one new block is created, which is placed at the very beginning of the sequence $\boldsymbol{D_0}$. If $\boldsymbol{L > M}$, the elements are pre-split into $\boldsymbol{O(L/M)}$ blocks by recursively finding the medians, which takes $\boldsymbol{O\left(L * \log(L/M)\right)}$ of time. All formed blocks are added to the beginning of $\boldsymbol{D_0}$, preserving the global monotonicity of the structure without the need to re-sort existing data and without interacting with the boundary tree $\boldsymbol{D_1}$.

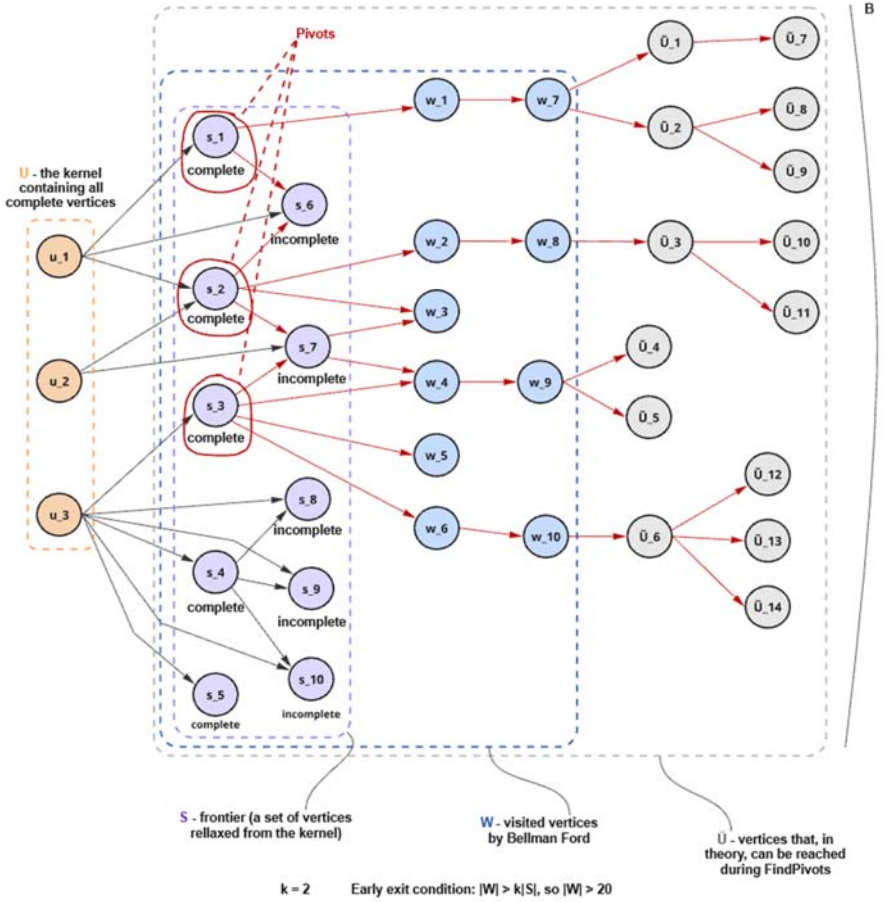

Fig. 6. FindPivots procedure and the directed forest of Pivots.

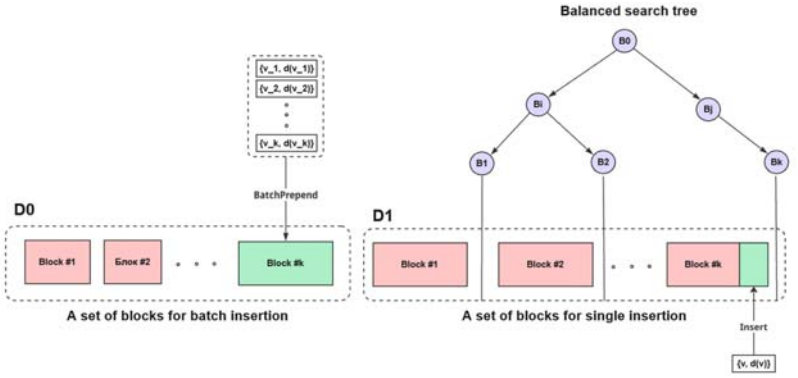

Fig. 7 Generalized structure of Blocking-based heap

The extraction of minimal elements is implemented by the ***Pull*** operation (see Fig. 8). The algorithm sequentially collects block prefixes from $\boldsymbol{D_0}$ and $\boldsymbol{D_1}$ until the total number of vertices reaches $\boldsymbol{M}$ or until all blocks are exhausted. Since the interblock order is guaranteed, assembling the desired subset takes linear time $\boldsymbol{O(M)}$. The extracted vertices are removed from the linked lists in constant time per element, and the remaining data is used to calculate a new dividing boundary $\boldsymbol{x}$ that satisfies the condition $\boldsymbol{max(extracted) < x \leq min(remaining)}$. Clearing empty blocks in $\boldsymbol{D_1}$ is synchronized with the search tree and amortized by previous insertion operations without adding asymptotic overhead.

To sum up, Table 2 shows the list of main operations in the interface of Blocking-based heap and their basic time complexity.

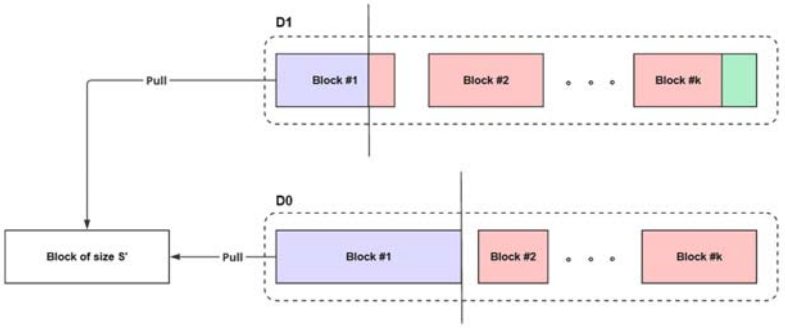

Fig. 8. Pull operation in Block-based heap.

Table 2. Base operations of Block-based heap

Method	Time complexity
$Insert$	$O\left(\max\{1, \log\left(\frac{N}{M}\right)\}\right)$
$BatchPrepend$	$O\left(\mathrm{L} * \max\{1, \log\left(\frac{N}{M}\right)\}\right)$
$Pull$	$O(\lvert S'\rvert)$
$Split$	$O(M)$
$Delete$	$O(1)$

3.4 Implementation details

Our implementation of the BM-SSP algorithm is built upon the foundational C++ codebase available in the repository by Liu et al. [14], which provides a reference implementation of the Duan et al. algorithm [5]. We have adapted and integrated this core logic into our experimental framework to ensure correctness while enabling rigorous benchmarking against Dijkstra's and Bellman-Ford algorithms. The implementation is written using modern C++ (C++17 / 20 standard) and relies exclusively on the Standard Template Library (STL) for core data structures, ensuring portability and ease of compilation across different environments. Key STL containers utilized include std::vector for adjacency lists and distance arrays, std::unordered_map and std::unordered_set for efficient lookups in the heap management, and std::list for the Blocking-based structure of the priority queue.

A key component of the BM-SSP algorithm is its specialized priority queue, referred to as the Blocking-Based Heap. In our implementation, this is realized through the BlockingBasedHeap class template. This structure organizes elements into blocks (D0 and D1) bounded by a parameter $\boldsymbol{M}$. It

utilizes a std::set with a custom comparator (CompareUB) to maintain upper bounds (UBs) for these blocks, allowing for efficient $\boldsymbol{O(\log B)}$ insertion and batch operations. The heap supports BatchPrepend operations, which are crucial for the recursive nature of the BM-SSP, allowing multiple distance updates to be processed efficiently without immediate re-balancing of the entire structure.

To handle the selection of pivots and median calculations within the heap's split and selectKth methods, we implemented the Floyd-Rivest selection algorithm. This deterministic linear-time selection algorithm is preferred over quickselect for its robustness and average-case performance, particularly when dealing with the large vectors of distance elements generated during the pull and batchPrepend operations.

4. Experimental evaluation

4.1 Hypotheses of BM-SSP efficiency

Based on the theoretical analysis of [5] and the developed implementation of *BMSSPSolver*, the following hypotheses about the effectiveness of the algorithm can be formulated. These hypotheses relate asymptotic properties to practical behavior on various classes of graphs. Next, we will formulate the main hypotheses of the effectiveness of the BM-SSP algorithm in comparison with the classical algorithms for finding shortest paths - Dijkstra and Bellman-Ford:

1. The BM-SSP algorithm can show its asymptotic superiority over the classical deterministic algorithm only on very large graphs in which the number of vertices exceeds the value of *N-treshold* (approximately > 10^5). On small and medium-sized graphs, the BM-SSP algorithm loses to its counterparts due to the high overhead associated with maintaining a Block-based heap, which is initialized at each level of recursion and the recursion itself.
2. The BM-SSP algorithm can show performance gains when the graph has high local connectivity (for example, k-partite graphs or randomized graphs with high density parameter). This may be due to the fact that the key procedure of the algorithm, *FindPivtos*, which breaks the Dijkstra sorting barrier, is tied to reducing the frontier to pivots, which with their subtrees maximally cover the studied set of graph vertices. That is, the more branchy the tree is, built from pivots, the more efficient the algorithm works. The branching of a directed forest of pivots is directly related to the density of the graph.
3. On graphs with many cycles, BM-SSP can show superiority on Dijkstra. This hypothesis holds because BM-SSP relaxes and maintains the order of estimated distances (frontier) locally within each recursive call, rather than globally as in Dijkstra (in the case of a priority queue). The presence of cycles leads to frequent re-formatting of the priority queue due to the operations *DecreaseKey* and *Insert*, and as a result, maintaining the global order of the relaxed vertices in the presence of cycles can seriously damage the performance of the Dijkstra algorithm, which cannot be said about BM-SSP.

4.2 Graphs classification

For the correct choice of an algorithm for solving the problem of finding shortest paths, it is necessary to consider various properties of the graph. Depending on its properties the effectiveness of a particular method can vary significantly. In this paper, a four-level classification of graphs is considered, which is based on the following criteria: structure, connectivity, cyclicity, and density. Each of these aspects has a direct impact on the choice of approach, the asymptotic complexity, and the practical performance of algorithms.

The testing framework is able to generate the following types of graphs depending on the parameters passed to it:

- A graph-path (see Fig. 9) is a graph that is a linear structure in which the vertices are arranged in a sequence, and each vertex is connected only to the previous and next. It is a special case of a tree, specifically a list (singly linked / doubly linked). It is set by a single integer parameter n, which is the number of nodes. A graph is created by sequentially adding one new vertex to the last vertex until the vertex limit is reached.
- A graph-cycle (see Fig. 10) is a graph consisting of a single cycle passing through all vertices. It is defined by a single integer parameter n, which is the number of nodes, similar to a path graph. The only difference from a graph-path is the connection of the first and last vertices to each other.
- A tree is a connected acyclic graph with exactly one path between any two vertices (see Fig. 11). It's generated using two parameters: *n* (number of nodes) and *branching_range* (maximum degree). Two vectors track generation: *current_children_count* monitors each node's children, while *available_parents* stores nodes below the limit. The algorithm randomly selects a parent from available nodes, adds an edge, and updates the counter. When a parent reaches *branching_range*, it's removed from available parents, and the new vertex becomes a potential parent, ensuring no vertex exceeds the branching limit.
- A grid graph (triangular / square tiling, see Fig. 12 and Fig. 13) is a graph whose structure resembles a lattice consisting of subgraphs with 3 (4) nodes, where each vertex is connected to its neighbors horizontally and vertically. It is set by two integer parameters: integer $rows$ - the number of rows of subgraphs and integer $cols$ - the number of columns of subgraphs. The generation algorithm iterates over all (r, c) slots, adds an edge between the current and right vertex if there is a neighbor on the right ($c + 1 < cols + 1$), and adds an edge between the current and bottom vertex if there is a neighbor on the bottom ($r + 1 < rows + 1$).
- A K-partite graph (see Fig. 14) divides vertices into k disjoint sets with edges only between different sets. It uses two parameters: *partition_sizes* (list of set sizes) and *edge_probability* (probability of adding edges between partitions). The algorithm distributes N vertices across k non-empty sets, then connects vertex pairs from different partitions (partition[u] ≠ partition[v]) with random weights. This ensures no internal triangles within partitions and guarantees connectivity for k ≥ 2, creating a complete k-partite structure where edge count equals the sum of products of all partition pair sizes.
- A Generalized randomized graph (see Fig. 15, Fig. 16) is defined by parameters: *n* (nodes), *density* (connectivity), *num_components*, *cycle_type* (cyclic/acyclic), and *connectivity_type* (weak/strong). The algorithm distributes *N* vertices across components, then builds a connected framework for each: a random spanning tree for weakly connected graphs or a simple cycle for strongly connected ones. To achieve target density and cycle count, it pools all valid edges (excluding loops and topological violations for acyclic graphs), partially shuffles them using Fisher-Yates for efficient sampling, and adds the required edges with random weights.

4.3 Experimental Environment

4.3.1 Hardware

Since we are testing algorithms for finding shortest paths not only on ordinary graphs, but also on extremely large ones with hundreds of thousands of vertices, it is not enough for us to use the power of conventional PCs. For these purposes, we will use a dedicated high-capacity physical server (with a large amount of RAM) in order to conduct experiments on large graphs most efficiently. Table 3 provides the characteristics of the dedicated server that will be used in the experiments.

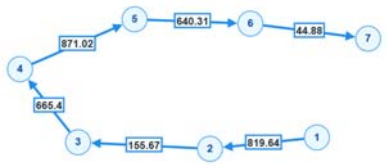

Fig. 9. Graph-path with 7 vertices.

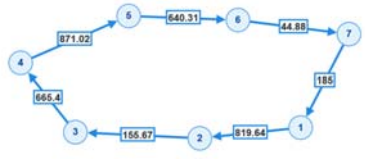

Fig. 10. Graph-cycle with 7 vertices.

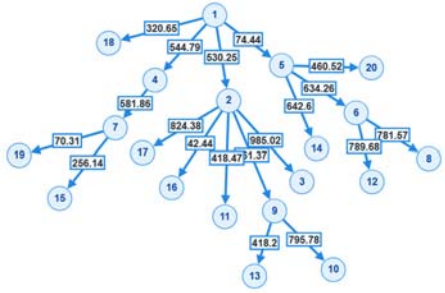

Fig. 11. Graph-tree with 20 vertices and high branching-range.

Fig. 12. Grid graph (triangle lattice) with 3 rows and 3 columns.

Fig. 13. Grid graph (triangle lattice) with 2 rows and 2 columns.

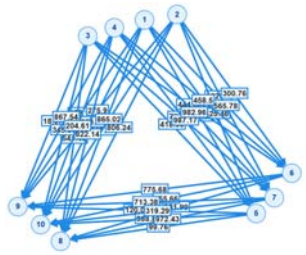

Fig. 14. K-partite graph with 10 vertices and 3 partitions.

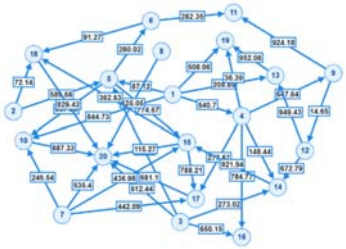

Fig. 15. Random acyclic graph with 20 vertices and low density.

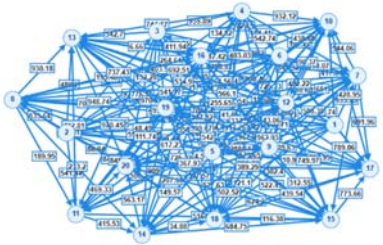

Fig. 16. Random acyclic graph with 20 vertices and high density.

Table 3. Characteristics of dedicated server.

Parameter	Value
Number of CPU cores	80
Processor model	Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz
Processor architecture	x86_64
RAM size	376 Gb
Disk size	520 Gb

4.3.2 Software

The testing framework is a console application written in C++ and ported to the Linux operating system. This application is built using the Cmake and Make utilities and can be compiled using the g++ compiler. Below is a table with the main software (application / system), thanks to which experiments were conducted on a dedicated server (see Table 4).

As artifacts, the testing framework outputs a file containing the graph adjacency matrix so that we can further verify the correctness of the generated graph and benchmark results. The graph visualizer graphonline.top [15] is used as an auxiliary cloud software for the article, which takes an adjacency matrix of a graph and renders it with vertex numbers, edge weights, and their directionality. An online service, Google Tables, is also used to systematize and visualize the benchmarks obtained.

Table 4. System & application software.

Software type	Minimal required version
Operating System	CentOS Linux 7 (Core)
Build tools	Cmake > 3.2 & Make > 4.2.1
Compiler	g++ > 8.3.1
Language	C++ (17 standard)
Libraries	STL & STD

4.3.3 Methodology

As part of the current testing, it is planned to conduct experiments on graphs of various sizes, structures, and parameters. In order for readers to reproduce some of the experiments on various devices (both home PCs and dedicated servers), it was decided to divide the graphs into two groups: small graphs and large graphs. This was done so that with an increase in the number of vertices and the density parameter (the proportion of current graph edges from the maximum number of edges), the number of edges increases at least quadratically from the number of vertices. This means that very large graphs may require a large amount of RAM, and most of the computer's time and resources will be spent on generating the graph, rather than conducting the experiments themselves. For each type of graph, the testing framework conducts experiments on the following graph sizes:

- Small graphs (the experiments are reproducible on a home PC): *10*, *25*, *50*, *75*, *100*, *250*, *500*, *750*, *1000*, *2500*, *5000*, *7500*, *10000* vertices.
- Large graphs (the experiments are reproducible only on a dedicated server): *15000*, *20000*, *25000*, *30000*, *35000*, *40000*, *45000*, *50000*, *60000*, *70000*, *80000*, *90000*, *100000* vertices.

Non-negative real numbers in the range [0.0; 1000.0] are used as edge weights for graphs. Random edge weights are generated using the Mersenne pseudorandom number engine (std::mt19937), initialized with a random value from std::random_device. Uniform distribution (std::uniform_real_distribution) is used to assign weights, which generates real numbers in the user-defined range [0.0; 1000.0].

In order to ensure that deterministic shortest path algorithms can be tested as representatively as possible, the testing framework tests graphs of various structures and parameters: *graph-path*, *graph-cycle*, *graph-tree*, *k-partite-graph*, *planar graph*, *chordal graph*, *triangle lattice graph*, *square lattice graph*, *acyclic randomized graph with low density*, *acyclic randomized graph with high density*, *randomized graph with low density and positive cycles*, *randomized graph with high density and positive cycles*.

Features of conducting benchmarks:

- 3 algorithms are measured: Dijkstra based on binary heap, Bellman-Ford and BM-SSP.
- The algorithm execution time is measured in milliseconds.
- In the benchmark tables, the horizontal axis shows the graph dimensions (number of nodes), and the vertical axis shows the logarithm of the algorithm execution time.
- Each benchmark value (for a fixed algorithm, size, and graph type) is the arithmetic average of 10 runs on variously generated graphs.

The Graphene framework uses the YAML format, which provides a description of graphs generation. The YAML file defines a set of experiments, each of which includes a specification of the graph generator and corresponding parameters for graphs. Examples of typical configurations, including benchmarks of Dijkstra, Bellman-Ford, and BM-SSP algorithms on various types of graphs, are available in the configs directory of the project repository [16].

4.4 Benchmarks and results

- **Graph-path**

  The benchmark results (see Fig. 17, Fig.18) show that due to the simple structure of the graph-path and the huge overhead of maintaining the block heap on each recursive call to the bmssp procedure and the recursion itself, the BM-SSP algorithm loses to Dijkstra and Bellman-Ford. The longer the graph-path becomes, the more BM-SSP lags behind its competitors (especially compared to Bellman-Ford for this type of graph).

Fig. 17. Graph-path (small sizes).

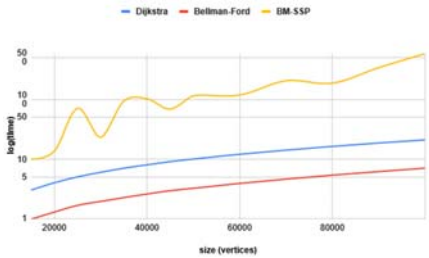

Fig. 18. Graph-path (large sizes).

- **Graph-cycle**

  The situation with a graph-cycle is almost similar to a path graph, since their structure is identical, with the only difference being that the first and last vertices of the cycle graph are closed on top of each other (see Fig. 19, Fig. 20). Because of this, we may see slightly stronger fluctuations in the BM-SSP execution schedule, but the trend towards a deterioration in the execution time of the BM-SSP algorithm with an increase in the number of vertices on such simple graphs still persists (especially compared to Bellman-Ford for this type of graph).

- **Graph-tree**

  When using graph-trees (see Fig. 21-24), the BM-SSP algorithm has stronger fluctuations due to a more complex graph structure (a larger number of edges leads to a more variable behavior of the Block-based heap). Also, on some graph sizes, it can be seen that BM-SSP performs faster than Dijkstra, but it is impossible to conclude about its full advantage on this type of graph, since the line graph of execution time of BM-SSP algorithm is very volatile and it stabilizes only on super-large graphs (size from 1e5).

- **K-partite graph**

  K-partite graphs (see Fig. 25-28) are an example of those graphs where BM-SSP shows absolute superiority over other algorithms due to the efficient operation of the Block-based heap on such a structure. During testing of any size of a K-partite graph, strong fluctuations in execution time are observed, however, the larger the graph becomes, the faster BM-SSP performs. Thus, these benchmarks fully confirm hypothesis No. 2 that BM-SSP performs best on graphs with high local connectivity and high density.

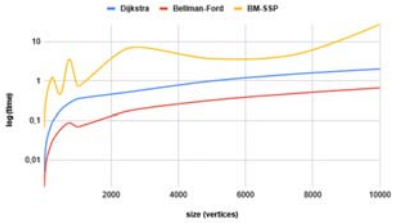

Fig. 19. Graph-cycle (small sizes).

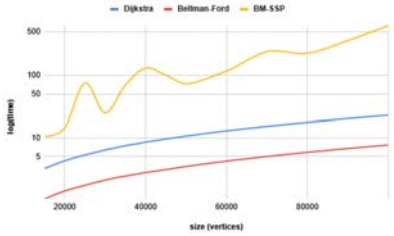

Fig. 20. Graph-cycle (large sizes).

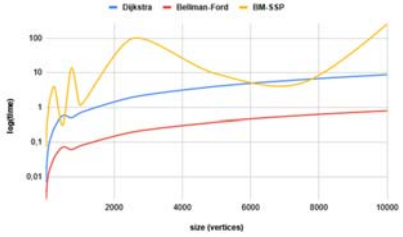

Fig. 21. Graph-tree (small sizes, branching-range = size × 0.2).

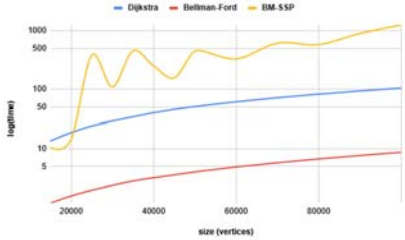

Fig. 22. Graph-tree (large sizes, branching-range = size × 0.2).

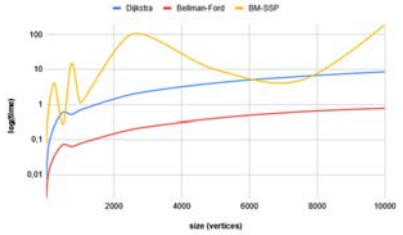

Fig. 23. Graph-tree (small sizes, branching-range = size × 0.8).

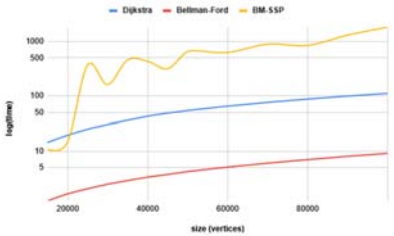

Fig. 24. Graph-tree (large sizes, branching-range = size × 0.8).

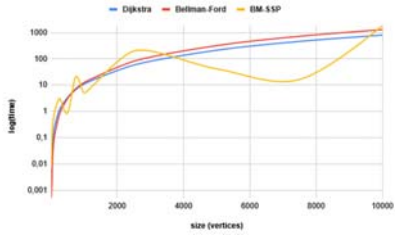

Fig. 25. K-partite graph (small sizes, k = size × 0.1).

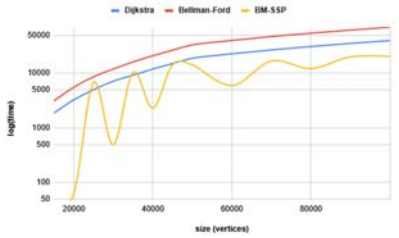

Fig. 26. K-partite graph (large sizes, k = size × 0.1).

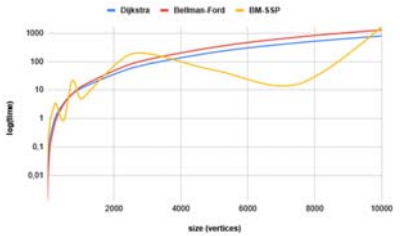

Fig. 27. K-partite graph (small sizes, k = size × 0.3).

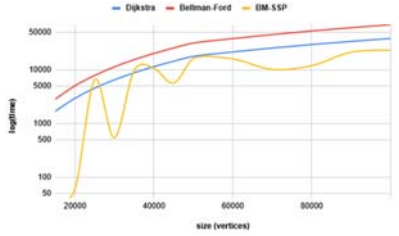

Fig. 28. K-partite graph (large sizes, k = size × 0.3).

- **Acyclic random graph**

  Based on the benchmarks of acyclic graphs (see Fig. 29-32), it can be seen that BM-SSP is most efficiently performed on dense graphs based on hypothesis No. 2, formulated earlier. On graphs with small dimensions and the parameter density = 0.1, BM-SSP was only rarely superior to its competitors. As the number of vertices increased, the performance of the algorithm decreased and it was mostly inferior to Dijkstra and even Bellman-Ford. However, the situation changes in the opposite direction with an increase in the density parameter (up to 0.7).

- **Positive cycled random graph**

  Benchmarks on graphs with positive cycles (see Fig. 33-36) confirm that BM-SSP efficiency depends on density and performs better on cyclic than acyclic graphs of similar size. Random graphs with positive cycles and high density are the most favorable for BM-SSP.

- **Lattice graphs (triangle / square)**

  Benchmarks on grid graphs (triangular and quadratic paving) confirm hypothesis No. 3 that the number of positive cycles of a graph correlates well with the efficiency of the BM-SSP algorithm (see Fig. 37, Fig. 38). These examples show that it has better execution time than Dijkstra, but Bellman-Ford surpasses both options.

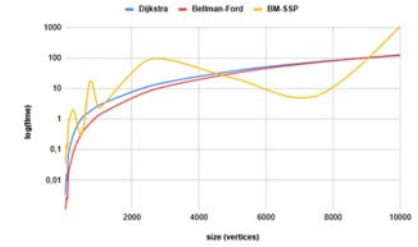

Fig. 29. Acyclic random graph (small sizes, density = 0.1, num-components = 1).

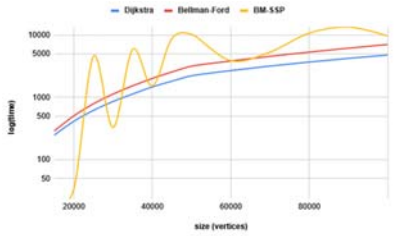

Fig. 30. Acyclic random graph (large sizes, density = 0.1, num-components = 1).

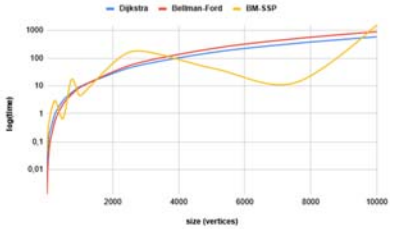

Fig. 31. Acyclic random graph (small sizes, density = 0.7, num-components = 1).

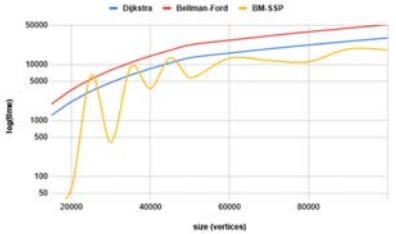

Fig. 32. Acyclic random graph (large sizes, density = 0.7, num-components = 1).

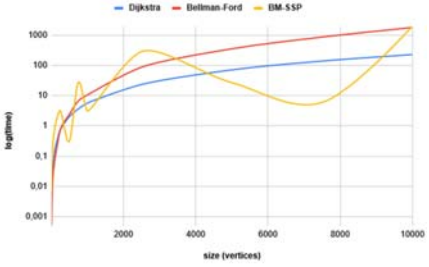

Fig. 33. Positive cycled random graph (small sizes, density = 0.1, num-components = 1)

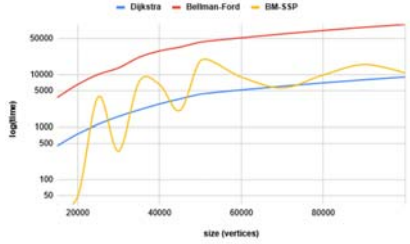

Fig. 34. Positive cycled random graph (large sizes, density = 0.1, num-components = 1)

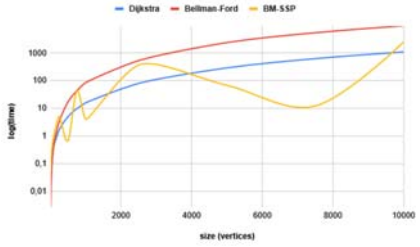

Fig. 35. Positive cycled random graph (small sizes, density = 0.7, num-components = 1)

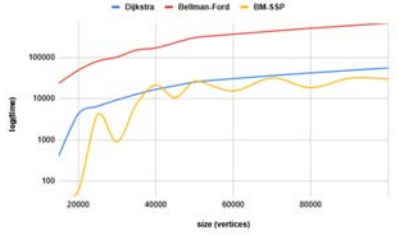

Fig. 36. Positive cycled random graph (large sizes, density = 0.7, num-components = 1)

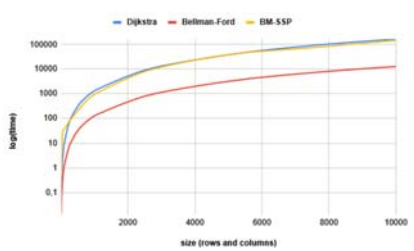

Fig. 37. Triangle-lattice graph

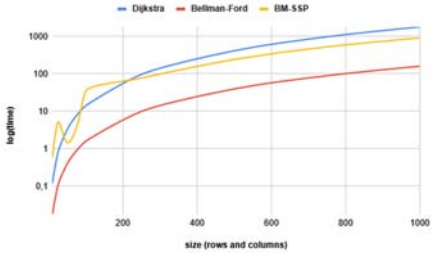

Fig. 38. Square-lattice graph

5. Conclusion

To summarize, we would like to note that the latest Bounded Multi-Source Shortest Path algorithm, having in theory a good asymptotic estimate of time complexity compared to its counterparts such as Dijkstra or Bellman-Ford, in practice it does not always justify its theoretical superiority. On some types of graphs that have an extreme structure (graph-path / graph-cycle/graph-tree), BM-SSP shows worse execution time compared to other deterministic algorithms due to the huge overhead associated with maintaining a Block-based heap and recursion. Due to the same features of the internal structure of BM-SSP, it makes no sense to use this algorithm on small graphs (up to 10,000 nodes). More lightweight and standard algorithms are suitable for them. On the other hand, there are many tasks in which the size of graph data reaches huge values (hundreds of thousands and millions of nodes). For such tasks, the best choice is to use BM–SSP, since Bellman-Ford is inferior to it in performance due to the worst asymptotic formula, which especially manifests itself on large graphs, and Dijkstra begins to degrade in performance due to storing the global state of vertex estimates. BM-SSP also shows its absolute superiority on some types of graphs that have a structure that is most favorable for the operation of Block-based heap (K-partite graph, Random graph with high density and positive cycles, triangle-lattice graph). On such graphs, the BM-SSP can be safely used, even if the graph is small.

Particular attention should be paid to the nuances of the Block-based heap implementation, since it plays a critical role in the asymptotic complexity of the algorithm execution time and the amount of memory consumed. Since the original Block-based heap is a complex structure with a huge number of hash maps, doubly linked lists and a search tree, it was decided to use a regular hash table in this article. But it is important to keep in mind that using the original implementation of the Block-based heap can lead to huge memory costs for maintaining this structure.

Thus, BM-SSP is not a silver bullet, thanks to which it is possible to solve the SSSP problem more efficiently than any other deterministic algorithm. However, these theoretical breakthroughs in the field of graph algorithms show that the generally accepted Dijkstra algorithm is not the most efficient, and perhaps in the future new deterministic algorithms will appear that can solve the SSSP problem more efficiently on more diverse graphs.

References

[1]. Bast H., Delling D., Goldberg A., Müller-Hannemann Ma., Pajor T., Sanders P., Wagner D., Werneck R., Route planning in transportation networks. In Algorithm Engineering: Selected Results and Surveys, Springer, 2016, 19-80. DOI: 10.1007/978-3-319-49487-6.

[2]. Retvari G., Tapolcai J., Enyedi G., Csaszar, A. IP fast reroute: Loop-free alternates revisited. IEEE INFOCOM 2011, pp. 294–298. DOI: 10.1109/INFCOM.2011.5935135.

[3]. Cowen L., Ideker T., Raphael B. J., Sharan, R. Network propagation: a universal amplifier of genetic associations. Nature Reviews Genetics, 2017, vol. 18(9), pp. 551–562. DOI: 10.1038/nrg.2017.38.

[4]. Angles R., Gutierrez C. Survey of graph database models. ACM Computing Surveys (CSUR), 2008, vol. 40, issue 1, pp. 1-39. DOI:10.1145/1322432.1322433.
[5]. Duan R., Mao J., Mao X., Shu X., Yin L. Breaking the sorting barrier for directed single-source shortest paths. Proceedings of the 57th Annual ACM Symposium on Theory of Computing, 2025, pp. 36-44. Also available at: https://arxiv.org/pdf/2504.17033, accessed 01.08.2026.
[6]. Makowski C., Guter W., Russell T., Saragih A., Castro L. BMSSPy: A Python Package and Empirical Comparison of Bounded Multi-Source Shortest Path Algorithm. MIT Center for Transportation & Logistics, 2025 Research Paper No. 2025/034. DOI: 10.2139/ssrn.5777186. Also available at: https://ssrn.com/abstract=577718, accessed 01.08.20266.
[7]. Castro L., Clementino T., de Freitas R. Implementation and Experimental Analysis of the Duan et al., 2025. Algorithm for Single-Source Shortest Paths. DOI: 10.48550/arXiv.2511.03007.
[8]. Moore E.F. The shortest path through a maze. Proc. of the International Symposium on the Theory of Switching. Harvard University Press, 1959, pp. 285-292.
[9]. Bellman R., On a routing problem. Quarterly of Applied Mathematics, vol. 16, no. 1, pp. 87-90, Apr. 1958. DOI: 10.1090/qam/102435.
[10]. Floyd R.W. Algorithm 97: Shortest path. Communications of the ACM, Jun. 1962, vol. 5, no. 6, p. 345. DOI: 10.1145/367766.368168.
[11]. Dijkstra E. W. A note on two problems in connexion with graphs. Numerische Mathematik, Dec. 1959, vol. 1, no. 1, pp. 269-271. DOI: 10.1007/BF01386390.
[12]. Fredman M.L., Tarjan R.E. Fibonacci heaps and their uses in improved network optimization algorithms. Journal of the ACM, Jul. 1987, vol. 34, no. 3, pp. 596-615. DOI: 10.1145/28869.28874.
[13]. Ahuja R.K., Mehlhorn K., Orlin J.B., Tarjan R.E. Faster algorithms for the shortest path problem. Journal of the ACM, Apr. 1990, vol. 37, no. 2, pp. 213-223. DOI: 10.1145/77600.77605.
[14]. Liu et al. bmssp: Implementation of the Duan et al. BM-SSP algorithm. GitHub repository, 2024. Available at: https://github.com/lcs147/bmssp.
[15]. GraphOnline. GraphOnline: Online Graph Visualization and Analysis Tool. Available at: https://graphonline.top, accessed: 14.04.2026.
[16]. mrForza, Graphene. GitHub repository. Available at: https://github.com/mrForza/graphene, accessed: 15.04.2026.

Информация об авторах / Information about authors

Роман Сергеевич ГРОМОВ – студент 4-го курса бакалаврской программы «Программная инженерия» (НИУ Высшая школа экономики, Москва). Его исследовательские интересы лежат в области разработки надежных высоконагруженных распределенных информационных систем, систем управления базами данных, а также в области сетевых протоколов передачи данных.

Roman Sergeevich GROMOV is a 4th-year bachelor's degree student in Software Engineering (Higher School of Economics, Moscow). His research interests lie in the development of reliable high-load distributed information systems, database management systems, as well as in the field of network data transmission protocols.

Роман Александрович НЕСТЕРОВ – доцент департамента программной инженерии факультета компьютерных наук, заведующий научно-учебной лабораторией процессно-ориентированных информационных систем НИУ Высшая школа экономики, кандидат компьютерных наук НИУ Высшая школа экономики с 2022 г. Его научные интересы включают подходы к моделированию и анализу поведения сложно организованных информационных систем, с помощью сетей Петри, теорию категорий и общую теорию параллелизма.

Roman Alexandrovich NESTEROV – Cand. Sci. (Comp. Sci.), Associate Professor at the Department of Software Engineering at the Faculty of Computer Science, Head of the Laboratory of Process-Aware Information Systems (PAIS Lab), HSE University (Moscow, Russia). His research interests include approaches to modeling and analyzing the behavior of complex information systems using Petri nets, category theory, and the general theory of parallelism.