Politechnika
Wrocławska

Analysis and Comparison of the Performance of Algorithms Solving Selected Graph Problems and Sorting Algorithms

Kytrysh Andrii

Wrocław 2025

Spis treści

1 Aim of Project

The aim of this project is to become acquainted with sorting algorithms through their implementation and to examine their behavior under conditions approximating real-world scenarios. In particular, the project investigates and analyzes the impact of the distribution and type of input data (e.g., int, float, double, etc.) on sorting time. The project's objectives include not only the implementation of the algorithms themselves but also the development of additional software to support the experiments—measuring algorithm execution time, recording sorting results, reading input data from files, generating input data of various distributions and sizes, and saving sorted output to files.

Furthermore, the project aims to study algorithms that solve the problems of finding a Minimum Spanning Tree (MST) and the shortest path, utilizing different graph representations such as adjacency lists and incidence matrices. The behavior of these algorithms will be examined under conditions approximating real-world scenarios. This includes investigating and analyzing the influence of graph size, density, and representation on algorithm performance and applicability. The objectives also encompass the implementation of these algorithms and the creation of supporting software for measuring execution times, saving results, reading and generating graphs of varying sizes and densities, and recording the algorithmic outputs to files. The results will also be demonstrated in the console, providing either the shortest path with its total weight or the MST with its total weight.

2 Introduction to Sorting Algorithms

In this project, we will study various algorithms on an increasing sequence of input data sizes, which is an approximation of real-world cases of using sorting algorithms in practice. We will perform an analysis of the obtained data and compare it with theory. We will also create visualizations of the results obtained from the experiments. Each experiment contains 100 repetitions for each case and type of algorithm. The result of a series of repetitions is the average, minimum, and maximum execution time of the algorithm, as well as the median time. All of this information will be used to analyze the results and to build charts. At the very beginning, we will discuss the algorithms that we are going to implement.

2.1 Insertion Sort

Insertion sort is a simple algorithm that builds a sorted part of the array element by element. In each iteration, it takes the next element from the unsorted part and inserts it into the appropriate place in the already ordered part, shifting larger elements one position to the right. Its time complexity is $O(n^2)$ in the worst and average case, while in the best case—when the data is already pre-sorted—it runs in linear time $O(n)$. Due to its simplicity, it is efficient for small datasets or nearly sorted data. For large arrays, it becomes impractical.

2.2 Binary Insertion Sort

Binary insertion sort is a variant of classical insertion sort, in which binary search is used to find the insertion point instead of linear scanning. As a result, the number of comparisons decreases to $O(\log n)$ per insertion, giving a total of $O(n \log n)$ comparisons, but the number of element shifts remains $O(n^2)$. The algorithm preserves stability and simplicity of implementation, just like standard insertion sort, but may be somewhat faster.

2.3 Quicksort

Quicksort is a "divide and conquer" algorithm. It selects an element called a pivot, divides the array into two parts: smaller and larger than the pivot, and then recursively sorts both parts. The average time complexity is $O(n \log n)$, whereas in the worst case—e.g., when consistently choosing an extreme element as the pivot—it is $O(n^2)$. In practice, with random pivot selection (a random pivot selection was implemented in this project) or more advanced pivot selection strategies, Quicksort is very fast and memory-efficient. The algorithm performs in-place sorting. It is not stable.

2.4 Heapsort

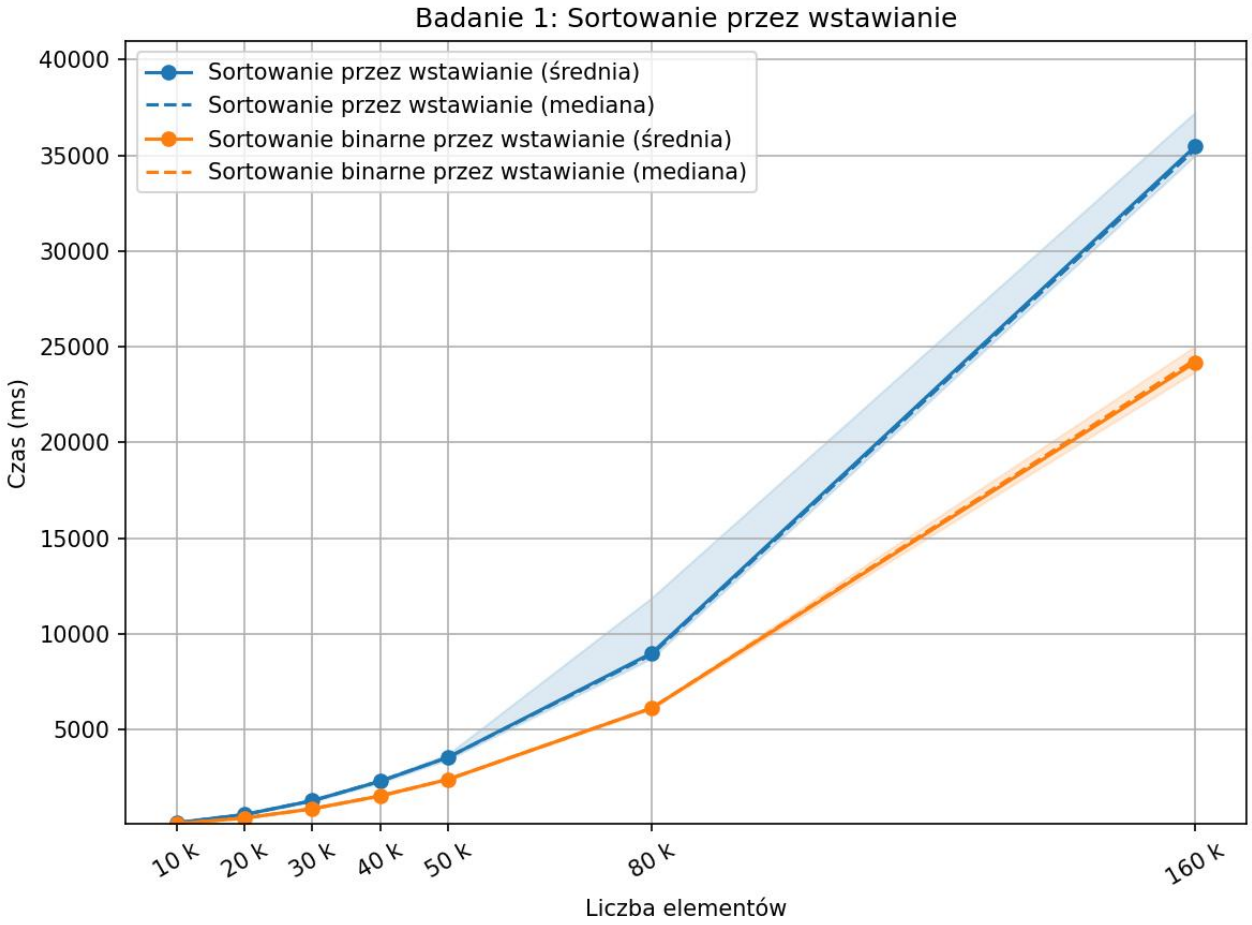
Heapsort also uses the "divide and conquer" idea. First, it builds a heap structure (a complete binary tree) from the array in $O(n)$ time, and then repeatedly removes the root (the largest element) and rebuilds the heap, which takes an additional $O(n \log n)$. The overall complexity is always $O(n \log n)$, regardless of the initial distribution of the data. The algorithm is not stable. Sorting is performed in place. The algorithm is very practical in situations where guaranteed preservation of the same time complexity in the worst case is required.

3 The Impact of Dataset Size on the Performance of Sorting Algorithms

In this chapter, we present the results of Study 1, in which we varied the size of the input dataset and measured the execution time of four sorting algorithms: simple insertion sort, binary insertion sort, heapsort, and quicksort. The measurements included both calculations of individual runs (for each instance size we performed 100 repetitions and recorded each result) as well as aggregate statistics (average, median, and minimum and maximum values).

Insertion sort, heapsort, and quicksort were displayed on two different plots for better readability of the chart. The reason is that the latter two types of sorting have much better computational complexity, which means that when displayed together with insertion sort, we would observe only a continuous line.

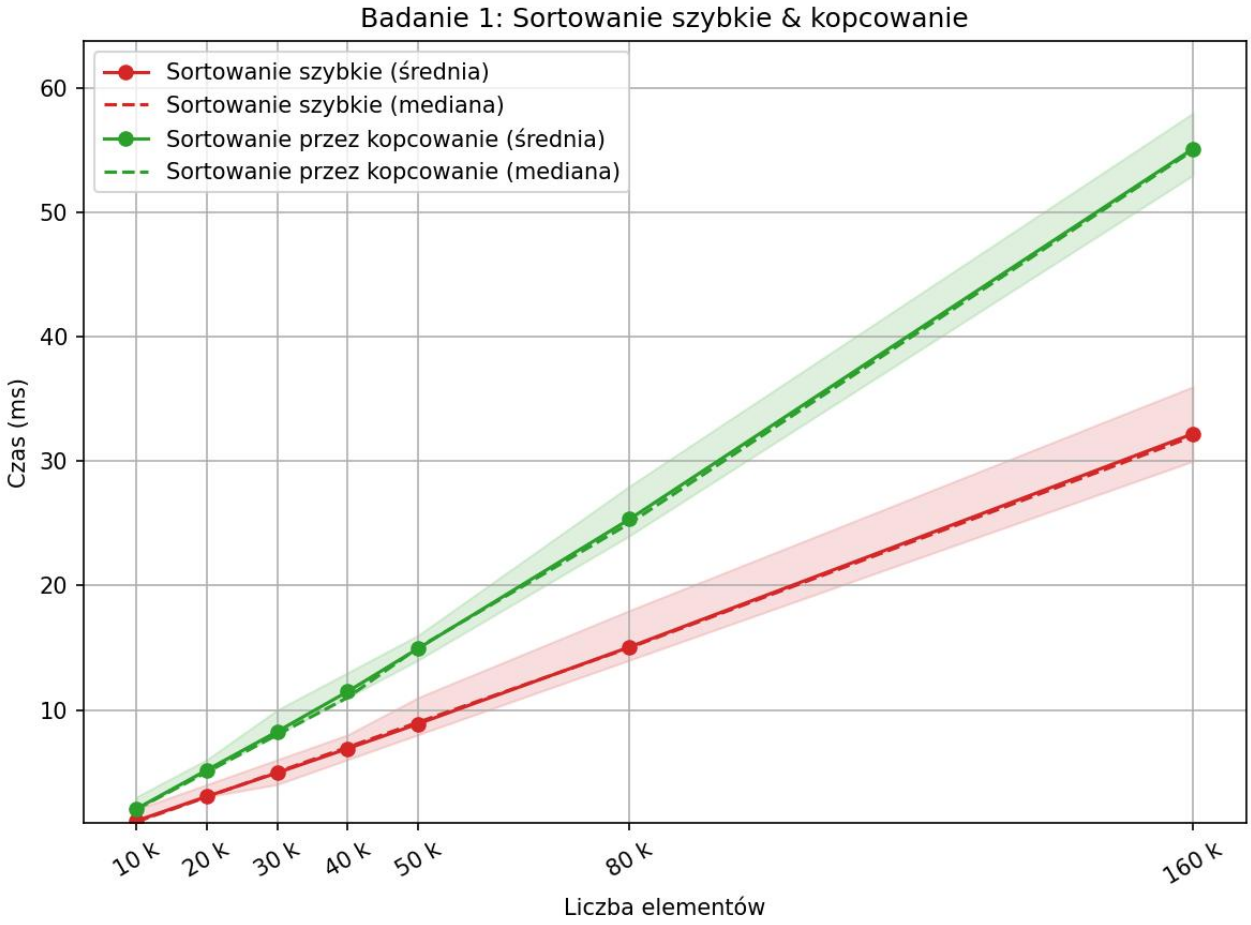
The following chart shows the relationship between time and the number of elements for simple and binary insertion sort. The boundaries of the minimum and maximum execution times are also visible. Additionally, the chart shows the median lines for both algorithms.



A similar chart was also made for the sorting times of quicksort and heapsort.

The charts clearly show the quadratic growth of execution time for both variants of insertion sort. For $N = 10\,\text{k}$ the time is within a few hundred milliseconds, but at $N = 80\,\text{k}$ it reaches the order of 10^4 ms, and at $N = 160\,\text{k}$ – more than 3.5×10^4 ms. The binary variant (orange) is consistently slightly faster than the simple variant (blue), because with the help of binary search, the time of finding the insertion point is reduced to $O(\log N^2)$, although the shifting operation within the array still requires $O(N^2)$ moves.

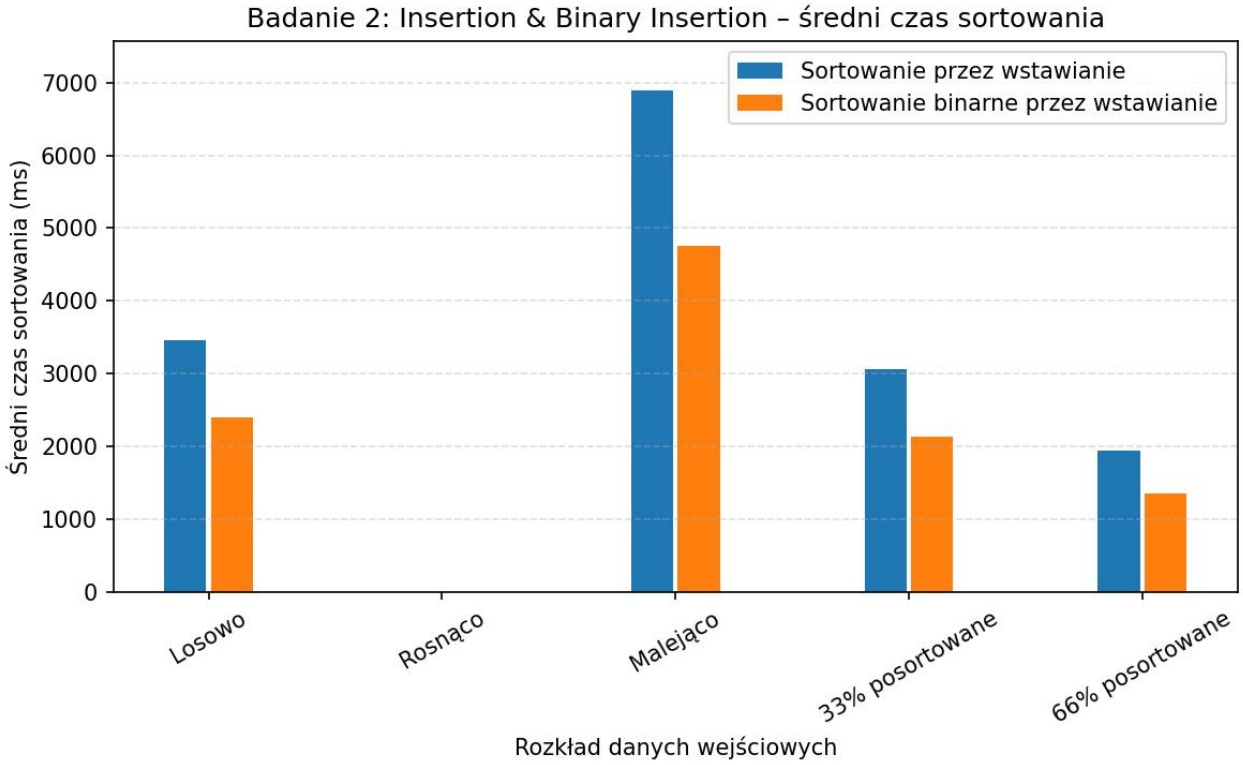
Quicksort (red points) and heapsort (green points) grow much more slowly – almost linearly on semi-logarithmic plots. For $N = 160\,\text{k}$ the average time of Quick Sort is about 32 ms, and Heap Sort – about 55 ms. The difference between them is mainly due to the fact that Quick Sort is practically very fast with a random



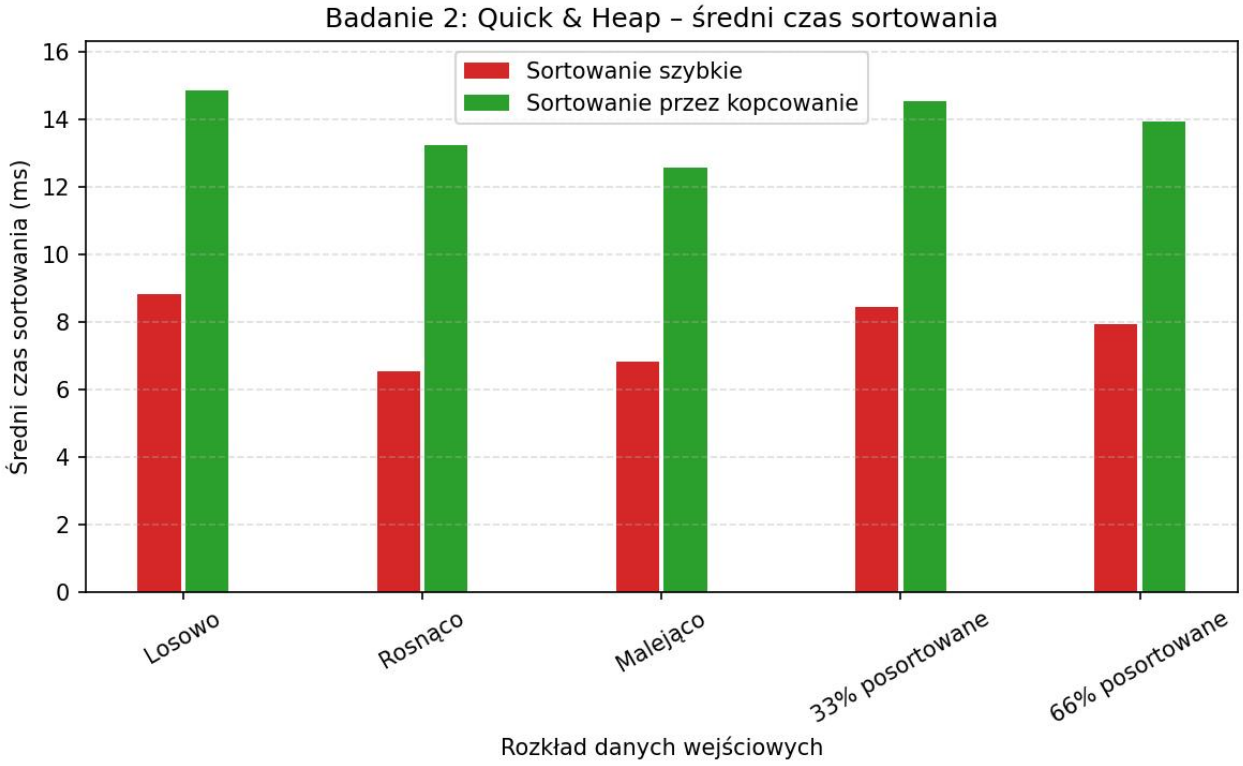
pivot, but may degenerate under unfavorable data distributions. Heapsort in practice provides more stable, although somewhat slower, execution times.

4 The Impact of Initial Data Distribution on Sorting Time

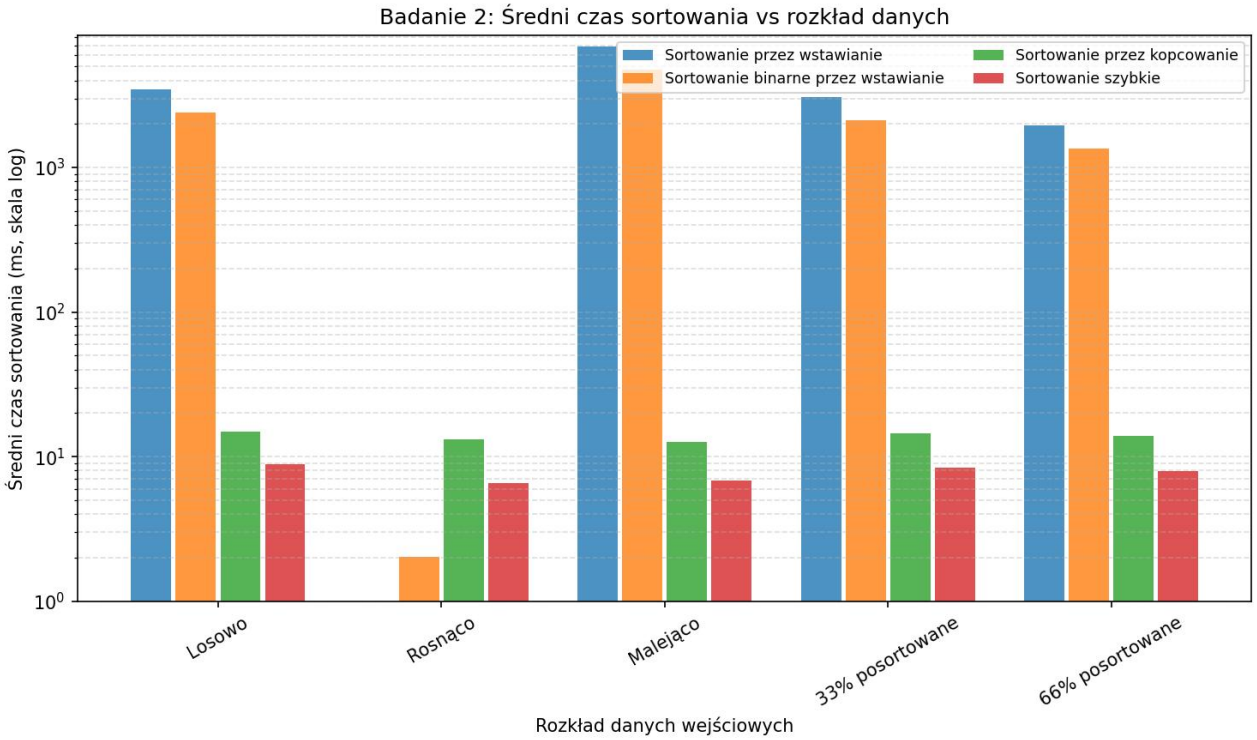
In the second study, the impact of the type of initial dataset ordering on the average sorting time of four algorithms was assessed, performing 100 measurements for each distribution (random, ascending, descending, and partially pre-sorted at 33% and 66%) and calculating the mean values. The input data contained 50,000 elements.



The figure shows that insertion sort achieves the highest time for descendingly sorted data (approx. 6900 ms). In this case, the algorithm literally has to reverse the list of elements, and at each check the following element is smaller than the previous one. The time is significantly lower for partially sorted data (approx. 3050 ms at 33%) and minimal (on the order of single milliseconds) for already ascendingly sorted data. In this case, the algorithm performs n passes through the dataset, and since the dataset is already sorted, it does not move anything. The binary version, however, systematically reduces times by about 30–35% due to binary search of the insertion point, which means that with each check half of the dataset is eliminated.



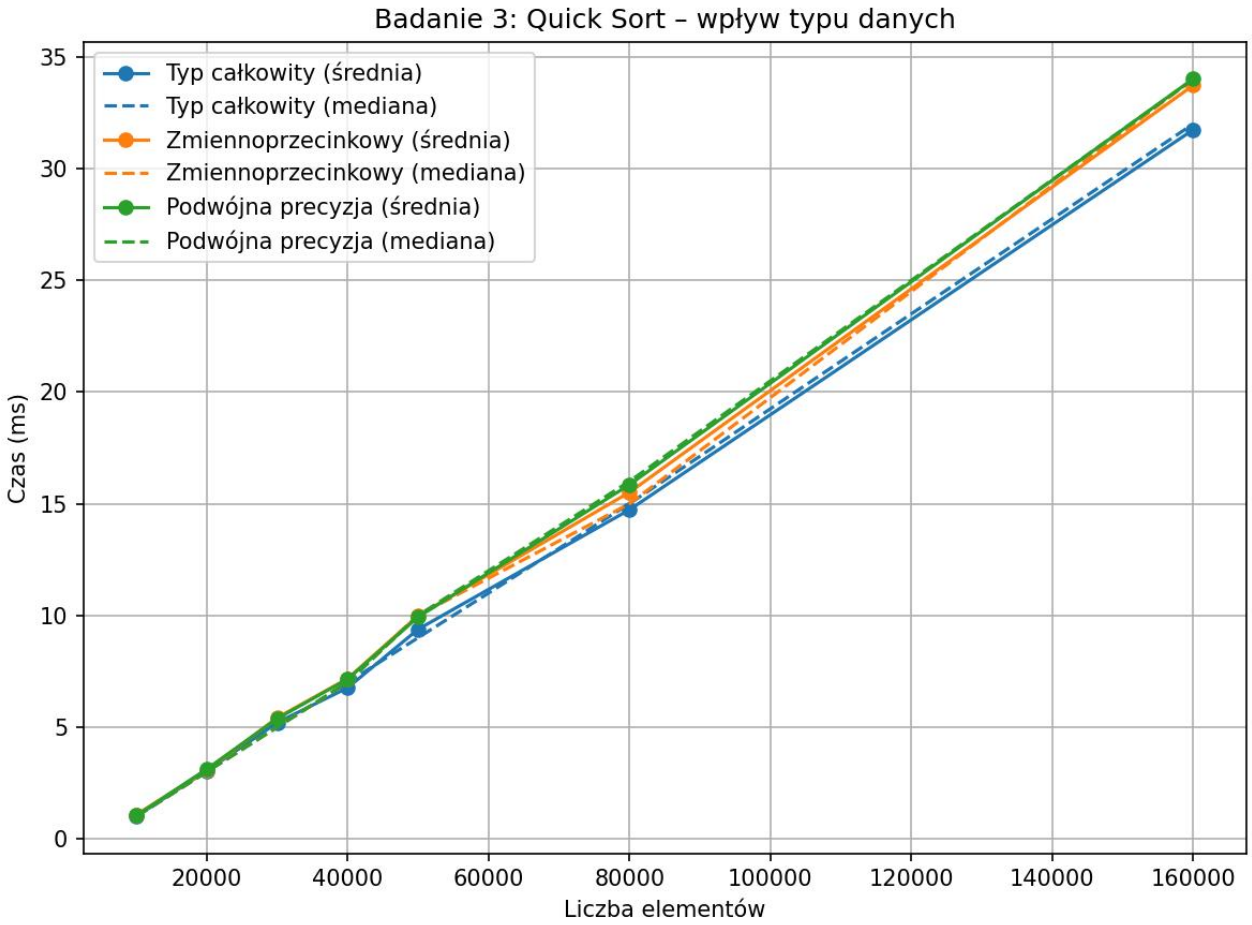
The chart above shows that quicksort is the fastest for ascendingly and descendingly sorted data, but somewhat slower on random and partially sorted data. Heapsort shows relatively stable performance in all cases except for descendingly sorted data. In this case, the array already satisfies the max-heap property (each parent is greater than its children), so the sift-down operations in the heap-building phase do almost nothing. As a result, heap construction, although generally $O(n)$, proceeds exceptionally fast for such data layouts, with a minimal number of comparisons and without swaps.



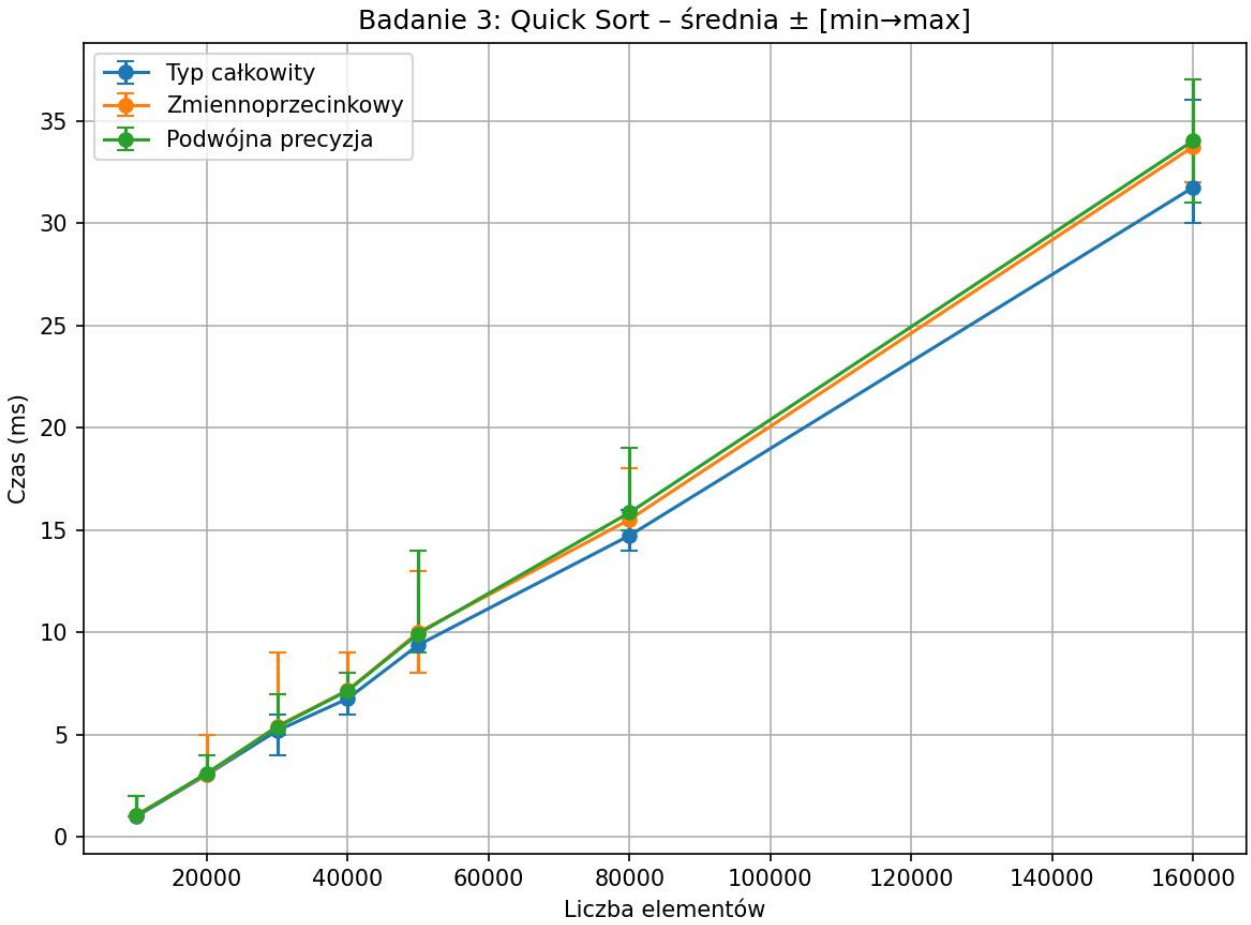
The figure above presents the same results including heapsort and quicksort in logarithmic scale, which emphasizes that quicksort maintains average times in the range of 7–9 ms, while heapsort – 12–15 ms, regardless of data distribution. Both of these algorithms are in-place sorts, which means that they operate with almost no additional auxiliary memory proportional to the input size. Quicksort uses only a recursion stack of depth $O(\log n)$, while heapsort modifies the array without any auxiliary structures, which results in low memory usage and good cache efficiency.

5 The Impact of Data Type on the Efficiency of a Selected Algorithm

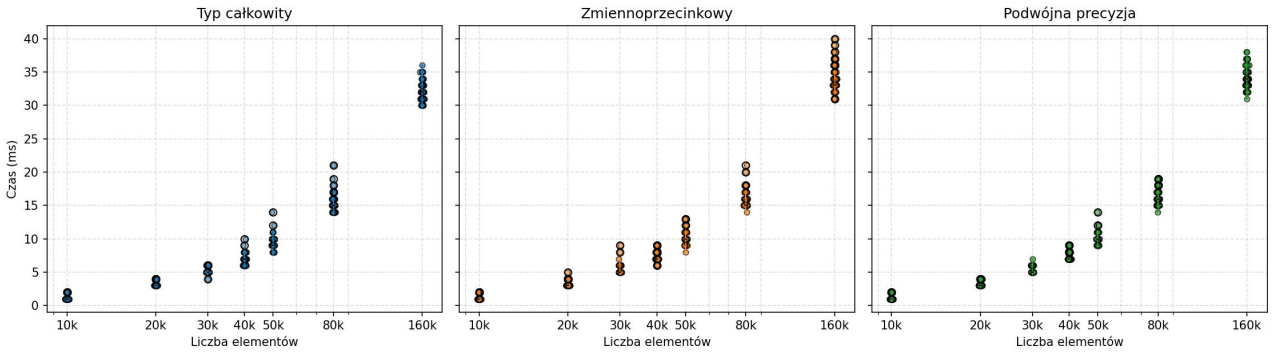
In the third study, the effect of the element storage type (32-bit integer, 32-bit single-precision floating point, and 64-bit double-precision floating point) on the performance of Quick Sort was assessed, performing 100 measurements for each type and each dataset size (10 k, 20 k, 30 k, 40 k, 50 k, 80 k, 160 k), and calculating the mean, median, and min–max range.



The figure above shows an almost linear increase in sorting time with increasing N. Sorting floating-point and double-precision data types is visibly slower than sorting integer types, which results from the additional overhead of operations on floating-point numbers. This is due to the fact that floating-point comparisons, according to the standard, must unpack and compare both the mantissa and the exponent, as well as handle special cases (NaN, infinity), which take more processor cycles than simple integer comparisons.



The figure above presents the same data with error bars representing minimum and maximum values; it shows that not only the mean times, but also the amplitude of deviations remains comparable for all three types, which confirms that the main factor influencing Quick Sort performance is dataset size rather than its representation.



The above chart shows that the spread of times (whiskers and points) is similar in all three cases – which indicates that the stability of the quicksort algorithm does not depend on the data type.

6 Introduction to Graph Algorithms

In this project, we will study various graph algorithms on an increasing sequence of input data sizes and input graph densities. Graphs will be examined within the following range of vertex sizes: 25; 50; 75; 100; 125; 150; 175; 250; 500; 750; 1000. This choice is an approximation to reality. For example, when an algorithm must find the shortest path from point A to B, for instance in a large city such as New York, where there are 472 subway stations and more than 300 bus stops, this already constitutes 772 possible stops to visit. Most often we travel short segments, which is why studies for 500, 750, and 1000 vertices were of technical rather than practical interest. As will become apparent during the experiments, it was not possible to perform all measurements for graphs of size 500+ vertices and density 50%+ due to computer performance limitations and suboptimally implemented data structures and algorithms.

7 The Minimum Spanning Tree Problem

The Minimum Spanning Tree (MST) is a subset of the edges of a directed or undirected graph that connects all vertices so that the sum of edge weights is minimal. MST is applied, among others, in the design of telecommunication networks, roads, or power systems, where we want to connect all points at the lowest possible cost. In this project, we will treat the graph as undirected in order to solve the MST problem.

7.1 Prim's Algorithm

Prim's algorithm starts from any vertex and gradually adds to the growing tree the least costly edge emerging from the already built fragment. In the adjacency list version (PrimList), in each iteration we only scan the edges leaving the vertices already in the tree, which in the worst case (without using a priority queue) gives a complexity of

$$O(V^2).$$

However, if we represent the graph with an incidence matrix (PrimMatrix), we scan all E edges each time, causing the runtime to grow to

$$O(V \cdot E).$$

In practice, PrimList is usually faster on sparse graphs, whereas PrimMatrix loses efficiency precisely due to the "global" scanning of all edges.

7.2 Kruskal's Algorithm

Kruskal operates on a flat list of all edges: first, it sorts them in ascending order by weight, and then adds edges sequentially to the forest being built using the Union–Find structure, thus avoiding the creation of cycles. The sorting stage takes

$$O(E \log E),$$

and thanks to the efficient disjoint-set structure (Union–Find), subsequent union and find operations execute in $\alpha(n)$–amortized time, which altogether yields memory complexity

$$O(V + E)$$

and runtime practically dominated by the sorting of edges. KruskalList and KruskalMatrix differ only in the method of reading the original edges, but the mechanism of sorting and union operations is identical.

8 The Shortest Path Problem

In the shortest path problem, we want for a given starting vertex to find the cheapest (shortest) path to every other vertex in the graph, taking into account edge weights.

8.1 Dijkstra's Algorithm

Dijkstra's algorithm uses a priority queue (min-heap) to extract in each iteration the vertex with the minimal distance from the source and then relax all edges leaving that vertex. It does not work with graphs that contain negative cycles. In the optimal implementation on an adjacency list (with a priority queue), it achieves a complexity of

$$O\big((V + E) \log V\big).$$

If, however, we do not use a queue and "manually" scan all vertices in search of the smallest distance, we obtain in the list version:

$$O(V^2),$$

and in the matrix version (scanning all E edges in each of the V iterations):

$$O(V \cdot E).$$

8.2 Bellman–Ford Algorithm

The Bellman–Ford algorithm does not require the absence of negative cycles in the graph and works by relaxing all edges in successive iterations. In each of the $V - 1$ iterations, it scans the entire list of E edges, thus deterministically achieving a complexity of

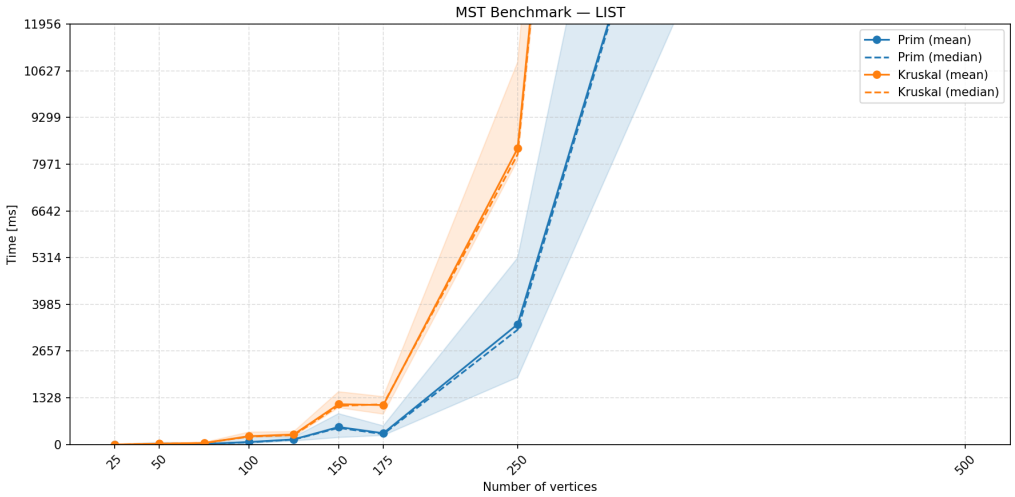
$$O(V \cdot E).$$

Additionally, after these $V - 1$ steps, one more relaxation round can be performed to detect the presence of negative cycles in the graph.

9 The Impact of Graph Size on Algorithm Performance

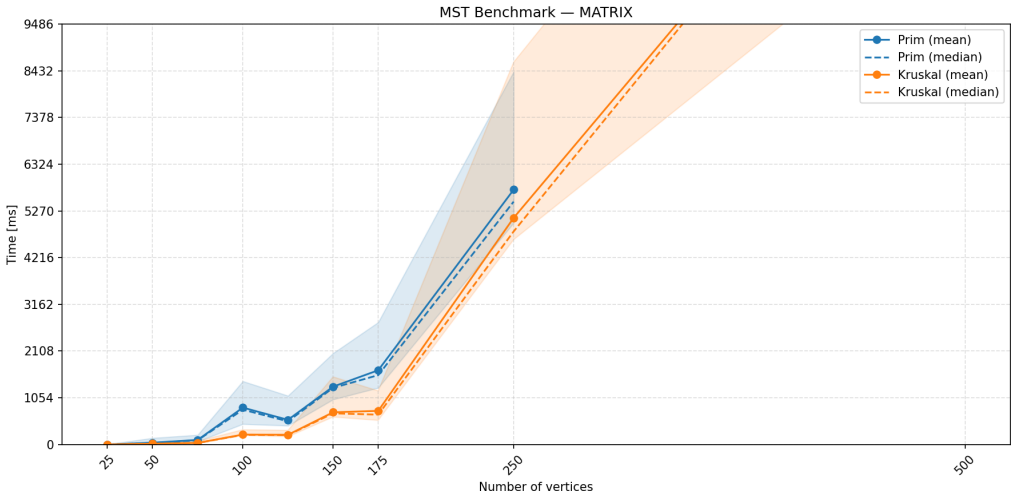
This chapter presents the results of runtime measurements of algorithms solving the MST and shortest path problems on plots of average time versus the number of vertices. Each point on the plots represents an average value from a series of tests on graphs of a given number of vertices and different edge densities 25%, 50%, and 99%.

In Figure [1] it can be seen that for small graphs (n 100) both Prim's and Kruskal's algorithms execute in the order of tens to hundreds of milliseconds. Only when moving to about 150 vertices do we observe a noticeable increase in cost. Both algorithms begin to grow almost exponentially. For n=250 the difference becomes dramatic: KruskalList reaches several thousand milliseconds, while PrimList exceeds 2–3 seconds, reflecting the higher complexity $O(V^2)$ in the case of Prim and the adjacency list representation of the graph.



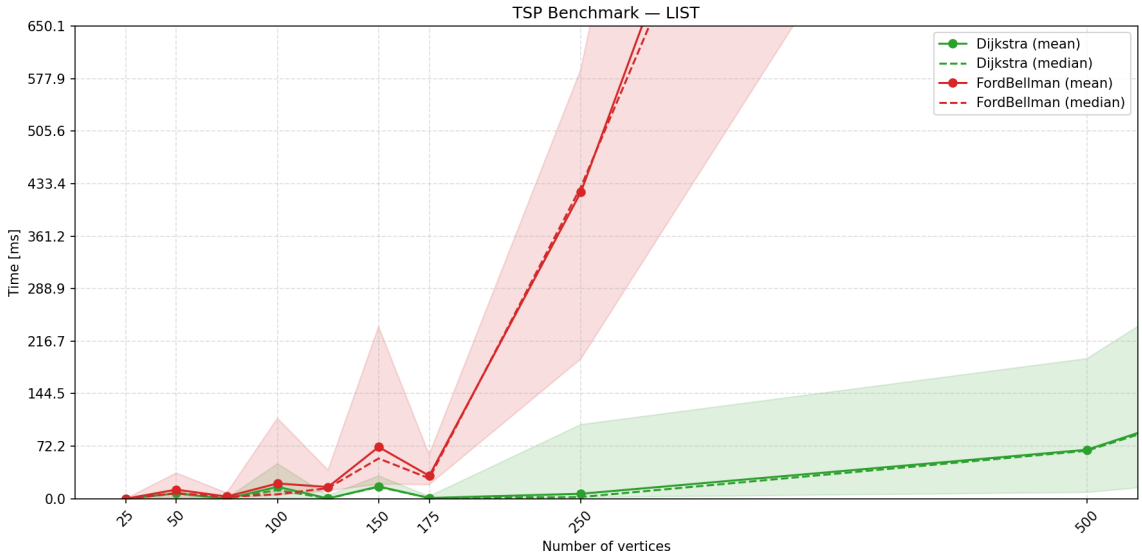
Rysunek 1: The impact of the number of vertices on algorithm performance for graph representation as an adjacency list

In the matrix representation [2] the pattern is similar, but the times are generally higher from around 150 vertices onward. PrimMatrix and KruskalMatrix grow similarly here, but the matrix version of Prim $O(V \cdot E)$ becomes clearly slower than KruskalMatrix $O(V + E)$ at every point.



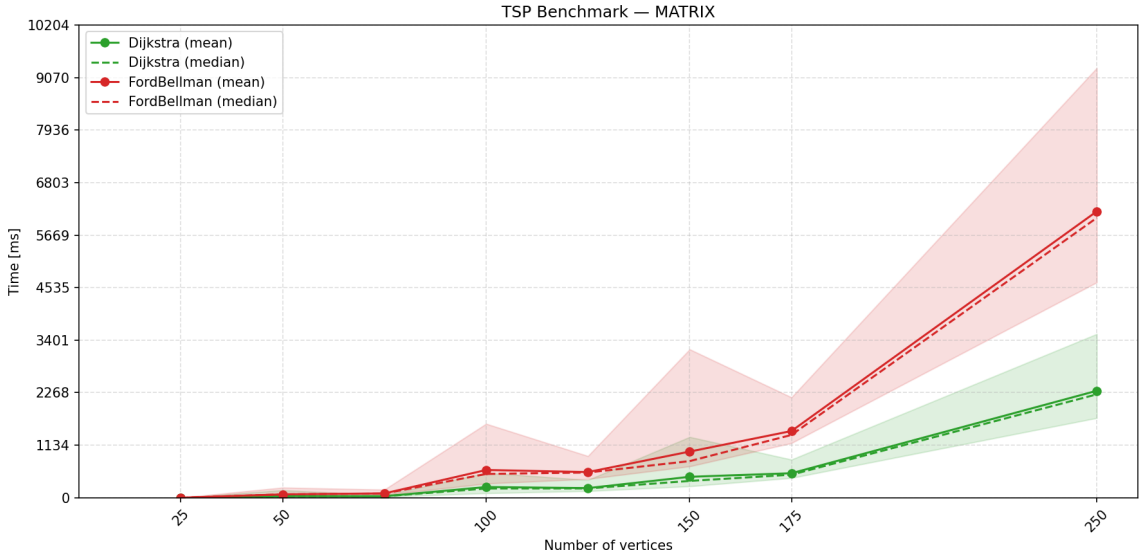
Rysunek 2: The impact of the number of vertices on algorithm performance for graph representation as an incidence matrix

On the plot for the shortest path problem [3] it can be seen that Dijkstra's algorithm (list + priority queue) behaves very efficiently and for most graph sizes maintains times below tens of milliseconds. On the other hand, Bellman–Ford, with complexity $O(V \cdot E)$, starts to visibly lag behind already at n=100, and at n=250 reaches hundreds of milliseconds up to several seconds, which perfectly reflects its linear–product complexity. The priority queue in Dijkstra's algorithm was implemented manually, using a custom implementation of a heap.



Rysunek 3: The impact of the number of vertices on algorithm performance for graph representation as an adjacency list

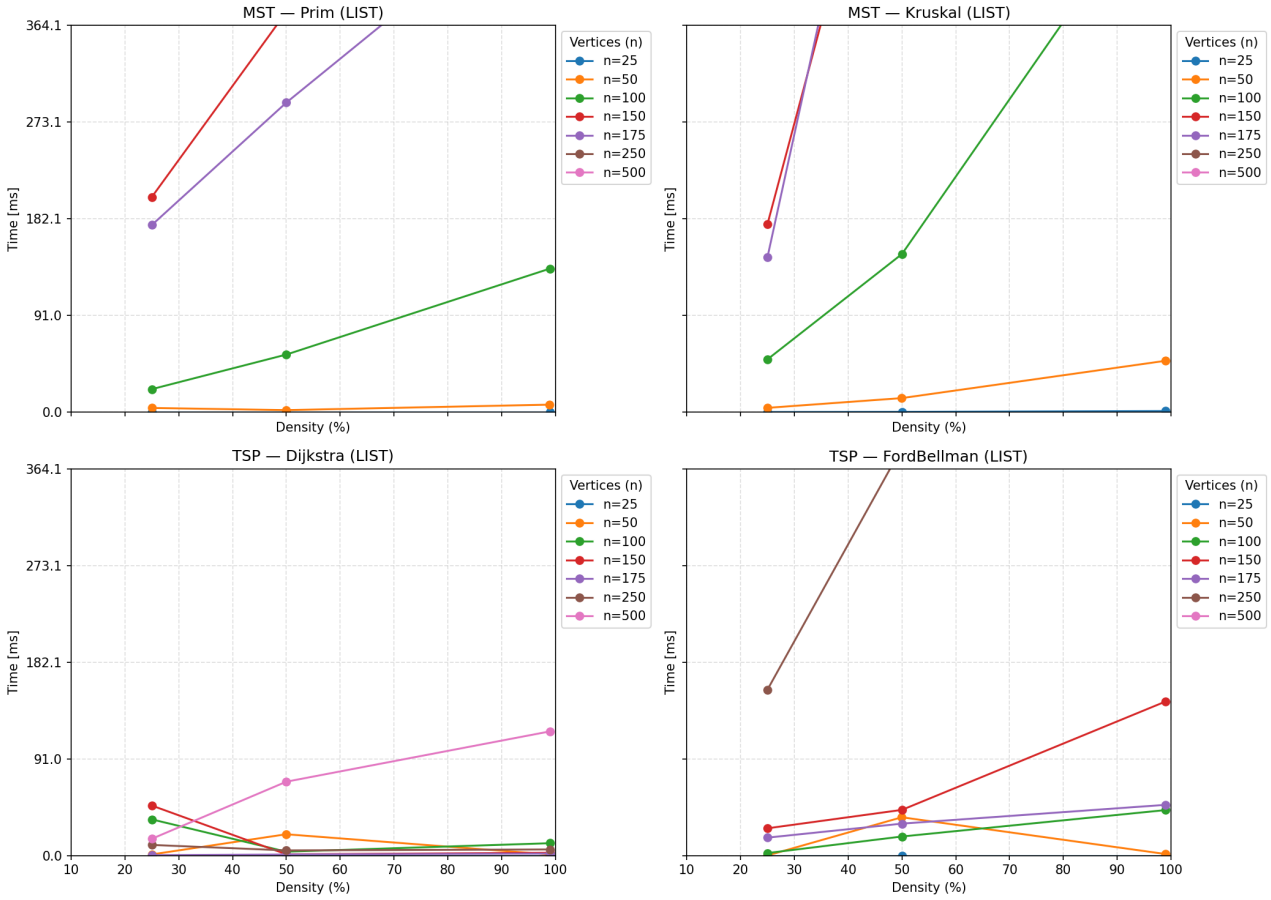
The matrix representation [4] once again highlights the performance drop of Bellman–Ford – its plots are the highest of all, especially at n=250. Dijkstra's algorithm in the matrix version remains at a relatively low level of runtime (up to hundreds of milliseconds), but it is evident that it too loses efficiency with the growing number of edges scanned in each iteration.



Rysunek 4: The impact of the number of vertices on algorithm performance for graph representation as an incidence matrix

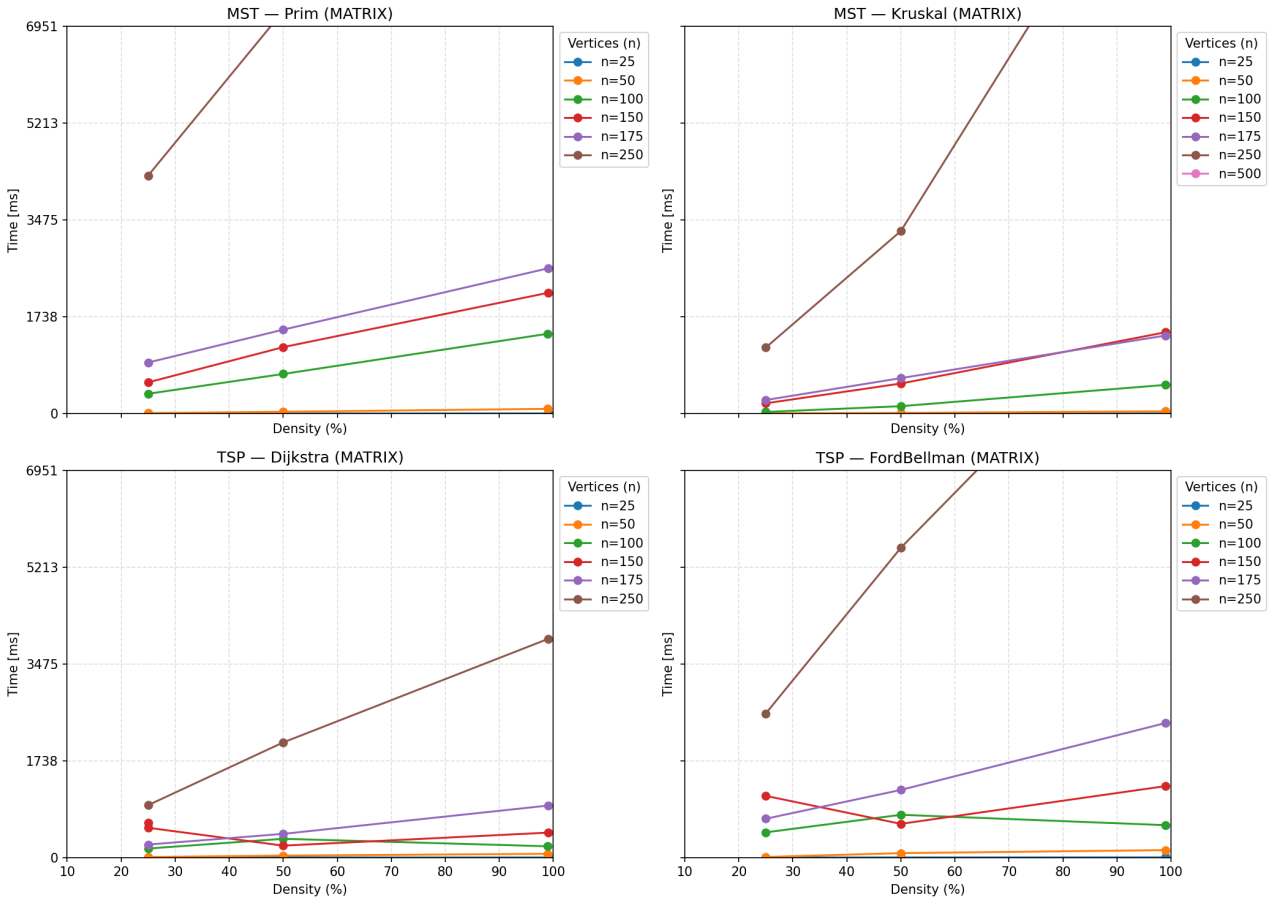
10 The Impact of Graph Density on Algorithm Performance

In this section, we analyze how the average runtime of MST and shortest path (TSP) algorithms changes with increasing edge density, with a fixed number of vertices. The results are presented separately for the list representation [5] and the matrix representation [6]. Each curve corresponds to a different graph size n (number of vertices) and shows how the runtime distribution depends on the percentage of edges in the graph.



Rysunek 5: The impact of density on the runtime of algorithms on the adjacency list

In the plots for the adjacency list (Fig. 5), it is clearly visible that the PrimList and KruskalList algorithms have completely different sensitivities to density. For small graphs (e.g. $n = 25, 50$) the times remain almost constant regardless of the share of edges — this is the effect of small E, which even in a dense graph does not significantly burden the main loop. As n increases (100–175), we observe an almost linear increase in Kruskal's runtime as a function of density, while Prim for the same graph sizes grows much more gently. For $n = 250$ (brown curve) both algorithms increase quickly, but PrimList should grow faster, resulting from the need to search for the minimum edge from each vertex in the tree (complexity $O(V^2)$), while KruskalList sorts the edge list only once ($O(E \log E)$), and density only increases E, not directly affecting the main loop of the algorithm after sorting. Unfortunately, this expected effect of faster growth is not visible in the chart. This may be caused by a faulty implementation of the algorithm or, more credibly, the experiments were run simultaneously on 4 computers for testing several algorithms at once. Each computer had a processor frequency in the range of 3–4.8 GHz. Such variation may have caused a measurement that was poorly comparable to others because it was performed on a physically slower computer.



Rysunek 6: The impact of density on the runtime of algorithms on the incidence matrix

In the matrix representation [6] the runtimes grow even more steeply. PrimMatrix scans all graph edges in each iteration, so its curves accelerate with density — particularly visible for $n \geqslant 175$ and $n \geqslant 250$. KruskalMatrix, on the other hand, although it also sorts the entire edge list, works somewhat slower than the list-based version, because access to the matrix is less local and cache costs are higher. Additionally, the Bellman–Ford algorithm in the matrix version records a sharp increase for larger n and higher density — each additional edge is multiplied by the number of vertices in runtime $O(V \cdot E)$. Dijkstra's algorithm on the matrix also grows linearly with density, but its slope is much smaller than Bellman–Ford, which fits well with its complexity $O(V^2)$ with adjacency list support, and in the matrix version – $O(V \cdot E)$. In summary, from the theoretical perspective we should see almost identical plots for Dijkstra Matrix, which is indeed observed. As can be seen in the plots, the speed of time growth is similar and the plots almost overlap.

11 Wnioski

In this project, the implementation of four sorting algorithms was carried out: insertion sort, binary insertion sort, heap sort, and quicksort. While writing the code for each algorithm, their working mechanisms were studied.

The insertion sort and binary insertion sort algorithms exhibit a rapid increase in execution time as the number of elements grows – for n80k n80k they become impractical. The binary variant reduces the number of comparisons and is approximately 30 faster than the classical version, yet the cost of element shifts still increases as O(n2) O(n 2).

The algorithms with complexity O(nlogn) O(nlogn), namely quicksort and heap sort, are characterized by a linear–logarithmic increase in execution time. Quicksort achieves the lowest average times (7–9 ms for n=160k n=160k), whereas heap sort is slightly slower (12–15 ms), but it guarantees consistent complexity regardless of the data distribution.

The impact of the initial ordering of data is crucial for quadratic algorithms – they sort descendingly ordered data the slowest and ascendingly ordered data the fastest. In the case of O(nlogn) O(nlogn) algorithms, the times remain stable, except for the accelerated heap construction for descendingly ordered data.

Changing the data type (int32, float32, float64) results in a slight shift of the quicksort time curves – operations on floating-point numbers require additional processor cycles and more memory traffic. They do not significantly affect stability or variance of performance.

In summary, for large datasets the best compromise between speed and predictability is quicksort (with random pivot selection), whereas when worst-case guarantees are required, heap sort is the better choice. Quadratic algorithms are suitable for small datasets or ascendingly ordered input.

In this project, the most difficult part was the experiments. In total, the time for all experiments amounted to 12 hours of uninterrupted computer work. Another issue was sorting across the full value range of the double type. This may possibly be related to a non-ideal independent implementation of the vector data structure. Therefore, during the experiments, the range of tested values for float and double types was reduced. The range of sorted float values was mainly lowered by reducing the range for the double type in order to maintain consistency of the experiments.

The simplest part was creating the structure of the program and planning the classes, their private and public methods, and designing how the experiment results would be saved so that the data format would be useful for fast analysis.

For creating the charts and automating the research process, a Python script was written. From the very beginning, it was planned to use the API of the OriginLab Pro program. This program is well known among engineers working daily with large amounts of data. Unfortunately, due to the difficult-to-understand and not very accessible documentation of the program's API, a decision was made to use the Python libraries numpy and matplotlib, which provide similar tools to those of OriginLab Pro. As part of this project, four variants of algorithms for determining the minimum spanning tree (PrimList, PrimMatrix, KruskalList, KruskalMatrix) and two shortest path algorithms (DijkstraList, Bellman–FordList and their matrix equivalents) were also implemented and tested. Each implementation was verified for correctness and freed from memory leaks, as confirmed by debugger tests. A script was also prepared to automate the generation of random graphs with specified sizes and densities, measurement of execution times, and visualization of results.

Analysis of the "Time vs Number of vertices" charts showed that the execution times of the algorithms grow in accordance with theoretical complexities. PrimList and PrimMatrix exhibit quadratic or quasi-quadratic growth (O(V2) O(V 2) and O(VE) O(VE), respectively), which becomes particularly apparent for large n n. KruskalList and KruskalMatrix, on the other hand, scale much more gently thanks to the one-time sorting of edges (O(ElogE) O(ElogE)), followed by union operations.

In the context of the shortest path, the DijkstraList algorithm (implemented with a priority queue) turned out to be more efficient than its matrix version, due to limiting the search to actual vertex neighbors. Bellman–Ford, although reliable even with negative weights, grows linearly with respect to the product VE VE, which in dense or large graphs makes it much slower than Dijkstra. On the charts for the shortest path problem, it is evident that the difference in the slopes of the curves of both algorithms correlates well with the complexities O((V+E)logV) O((V+E)logV) and O(VE) O(VE).

Studies of the effect of edge density confirmed that algorithms which sort edges once (Kruskal) are the least sensitive to an increase in the number of edges — their times grow proportionally to the additional cost of sorting, but are not multiplied across iterations. PrimMatrix and Bellman–FordMatrix, however, "scan" the entire incidence matrix in each step, which makes them practically unusable for dense graphs. The list-based versions of these algorithms behave much better, as they operate only on existing edges.

In summary, the obtained results are consistent with theory: algorithms of complexity O(ElogE) O(ElogE) (Kruskal) or O((V+E)logV) O((V+E)logV) (Dijkstra with PQ) should in practice be preferred over the simple quadratic variants of Prim or Bellman–Ford. Implementations without advanced priority queues and with typical data structures nevertheless allowed verification of their behavior in conditions approximating real applications and to plan further optimizations, e.g. introducing Fibonacci for Dijkstra. The times reported in the documentation represent averaged values from 50 independent measurements for each configuration, which yields a total of 1350 unique graphs and ensures the reliability of the obtained observations.

Difficulties encountered. Time. Generating such a large pool of graphs took a total of 40 hours. And although I wanted to generate graphs of smaller sizes, the question: What if prevailed. The experiments took another 24–25 hours. Moreover, graphs with 500, 750, 1000 vertices were not fully examined, therefore they were partially excluded from the analysis in the report, but they are included in the benchmark records. This "incompleteness" results from the complexity of the algorithms on the matrix representation of the graph. Namely, these graphs in the matrix representation were not fully studied for the above graph sizes, and a single pass of the algorithm on the matrix representation for a graph with 750+ vertices at density 99 took nearly 3 minutes. Another problem that was created was the disturbance of measurements by individual exceptionally slower runs. This happened due to the already mentioned approach of conducting experiments on 4 different computers, 3 of which had the same processor frequency of 4.3 GHz, but only one had 3.2 GHz and calculated the small graphs. This caused distortion of the charts at their beginning for graphs of size up to 150 vertices inclusive.

All sorts required in the implementations of the algorithms were carried out using the previously implemented Quick Sort from part 1 of the project. Additionally, the priority queue was implemented independently, also based on the already existing MAX heap from part 1 of the project.