# Evaluation of various Algorithms for the Linear Sum Assignment Problem

By

**Melisa Akdemir, Ludwig Austermann,**
**Tabea Both & Matthias Jean Théo Personnaz**

A report for our final project in
Scientific Computing

Technical University Berlin
February 14, 2023

# Contents

# 1 Introduction

The linear sum assignment problem (LSAP) is an optimization problem that seeks to find the best allocation of resources to tasks, with the goal of minimizing the overall cost of the assignments. This problem is commonly used to model real-world scenarios where resources must be assigned to tasks efficiently, such as assigning workers to jobs or tasks to machines.

The cost of each assignment is described in a cost matrix $C$, where each element $c_{i,j}$ of the matrix represents the cost of assigning a particular resource $i$ to a particular task $j$. The cost matrix is represented as an $n \times n$ matrix, where $n$ represents the number of tasks.

The LSAP is a special case of the more general transportation problem and can be formulated mathematically to minimize the sum of the elements of the cost matrix that correspond to the chosen assignments. This can be formulated as the following integer linear programming problem:

$$\text{minimize} \quad \sum_{i=1}^{n}\sum_{j=1}^{n} c_{i,j}x_{i,j} \qquad \text{(total cost)} \tag{1}$$

$$\text{subject to} \quad \sum_{j=1}^{n} x_{i,j} = 1 \quad \forall i \in 1,\dots,n \tag{2}$$

$$\sum_{i=1}^{n} x_{i,j} = 1 \quad \forall j \in 1,\dots,n \tag{3}$$

$$x_{i,j} \in \{0,1\} \quad \forall i,j \in 1,\dots,n \tag{4}$$

where $x_{i,j}$ is a binary decision variable that indicates whether a resource $i$ is assigned to a task $j$. The two constraints (2) and (3) ensure that each resource is assigned to exactly one task, and each task is assigned to exactly one resource, respectively. The last constraint ensures that $x_{i,j}$ is binary.

To ensure that the solution is integer-valued and to keep the complexity of the cost matrix reasonable, the cost matrix is represented as an integer-valued matrix in the `.mm` file format.

Different solution techniques will be explored to solve the LSAP, with the aim of finding the most efficient and cost-effective allocation of resources to tasks.

# 2 Various Problem Formulations and Algorithms

There is a multitude of algorithms, to solve the LSAP efficiently. Some of them, like the one in [Kao+01] have a theoretical runtime complexity of only $\mathcal{O}\Big(\sqrt{n}W\log(\frac{n^2C}{W})/\log n\Big)$, where[1] $C \coloneqq \max_{i,j} c_{i,j}$ and $W \coloneqq \sum_{i,j} c_{i,j}$. However, in practice instances of this problem can be quickly solved and the Hungarian method or Shortest Augmenting Path methods are widely used in practice. The Shortest Augmenting Path methods count to the most efficient methods known. ([BDM12] chapter 4.4)

We present some common solution techniques along with our implementation on our repository.

[1] We have assume here for the formulation of $C$ and $W$, that all costs are positive, to shorten the representation. Many algorithms shift the cost matrix anyway, as the assignment is invariant to it.

### 2.1 Greedy approach

In real-life applications, for very small instances of the Assignment Problem, one might want to perform a traversal of every possible permutations and choose the one that best fits some criterion, such as the one featured in the cost matrix. This is particularly relevant for large extent problems with little to no global dependency, such as the TSP on highly clustered graphs featuring scale-variant structures, or some very complex, broader assignment problems involving time-dependency and incomplete rolling-release pieces of information. For these problems, it is interesting to be able to explore the feasible region of some small instances.

As a reference, we implemented a greedy algorithm that aims at simply going through all possible permutations of $n$ elements. This approach is not smart in many regards for medium to large instances. It would normally be qualified as *stupid* and intractable anyway. Firstly, the number of permutations increases exponentially with the number of elements, which is intractable for even small instances ($n < 30$).

However, for smaller instances, possibly on already parallelized graph problems, one might want to solve very small instances of the assignment problem locally, and possibly with limited memory and data structure complexities. Thus, going through all permutations *might* be something to consider. In addition, its memory complexity is only $w \cdot \mathcal{O}(n)$ where $w$ is the number of parallel workers. Thus, *for the sake of experiment* and comparison, we carried it out.

This task of traversing all possible permutations is highly parallelizable. We have therefore implemented it on both the CPU and GPU. To traverse all permutations, it is necessary to be able to generate them safely and without forgetting or repeating any (which would increase the amount of computation for no reason). Several algorithms exist, but we have chosen to encode the permutations using natural numbers. Each permutation will therefore be associated with a number between 0 and $n! - 1$, called encoding, and the algorithm to decode the permutation from its original integer must be able to be performed using elementary data structures supported by GPU cores, such as classic arithmetic operations and arrays of numbers.

We have therefore chosen the **Lehmer bijective encoding** as such:

$$L : \mathfrak{S}_n \longrightarrow \{0\} \times [\![0, 1]\!] \times \cdots \times [\![0, n-1]\!] \tag{5}$$

$$\sigma \longmapsto L(\sigma) = [\#\{j < i, \sigma(i) > \sigma(j)\}]_{1 \leq i \leq n} \tag{6}$$

This counts the number of inversions in the permutation.

Each Lehmer encoding is then transformed into a unique integer between 0 and $n! - 1$ through the **factorial number system**, which is a number system whose bases change at every "order of magnitude", following that of the factorials.

Both steps of this encoding are bijective and the decoding can easily be done on a Nvidia CUDA core for Compute capability $\geq 2.0$ through *dynamic memory allocation* in $\mathcal{O}(n)$ memory size per kernel. That way, each unique thread ID gets to process one permutation.

**Caveats** Of course, several catches remain.

- For one thing, we are information-theoretically doomed to use as many as $\Theta(n \log n)$ bits for encoding one permutation and $\Theta(n^2 \log n)$ for storing it in plain and performing the score evaluation, as the factorials have the asymptotical equivalent lying in $\log(n!) \sim n \log n$. That means that the GPU single-precision integers won't be able to proceed permutations

of 13 elements or more. Likewise, special server-grade GPUs working with high-performance double precision won't be able to process $n \geq 21$.

- The number of threads and blocks are limited, although it is not the first-encountered limit.

But this entire algorithm does not even require some advanced data structures, such as sets, like the Hungarian method does. Our results have proved that on a relatively small-sized GTX 1650, as many as 10 billion permutations a second could be processed, as compared with only a few millions per second on an equally small, consumer-grade Intel i5-10300H chip. We did not include the scaled comparisons in the graphs, though, for the previous, obvious intractability reasons.

### 2.2 Hungarian method

The Hungarian method, also known as the Kuhn-Munkers-algorithm is a way to solve the LSAP. The method is a primal-dual one and some of its variants, utilizing some well-suited data structures, are the fastest-known methods to solve the LSAP.

$$\text{maximize} \quad \sum_{i=1}^{n} u_i + \sum_{j=1}^{n} v_j \tag{7}$$

$$\text{subject to} \quad u_i + v_j \leq c_{i,j} \quad \forall i, j \in 1, \ldots, n \tag{8}$$

$$\tag{9}$$

The idea behind this algorithm, is to use a minimal submatching, to construct a minimal submatching of higher order. More concretely, in each iteration we have feasible dual solution $u, v$, as show in the dual problem **?? ??**, and a partial primal solution, satisfying the *complementary slackness* 10. This ensures us the minimality. To construct a assignment of higher order, we use augmenting paths (as seen in bipartite matching), which are paths along vertices in the bipartite graph, with edges alternating between assigned and unassigned and whose first and last edges are unassigned, on the partial graph only containing zero reduced cost edges. By flipping these paths, that is, transforming unassigned edges to assigned ones and vice versa, we can obtain a new submatching, which has higher order.

$$x_{i,j}(c_{i,j} - u_i - v_j) = 0 \quad \forall i, j \in 1, \ldots, n \tag{10}$$

By then selecting the path optimally, we maintain, at all time, a minimal submatching. Iterating over this procedure yields us then a minimal matching for the initial problem.

We will further distinguish between the simple Version and an optimized version – which integrates a shortest path algorithm.

### 2.3 Parallelized Hungarian method

The Hungarian method is quite difficult to parallelize, since it is mainly a outright serial algorithm. One section, though, is a significant bottleneck for large instances: the preprocessing part of each run, which updates the costs column-wise, then row-wise by substracting the minimum, is prone to parallelization on a GPU since it operates directly on the cost matrix; and the GPUs are well suited for performing intense computations on arrays and matrices. However, provided the rest of

the algorithm makes intense use of set operations, this elementary improvement must be considered with care, as every memory transfer will significantly impact the performance as the matrices must be updated every time if the instance size is small.

For incompatibility reasons between our version of `gcc` and `nvcc` compiler, and the code could not be compiled and integrated properly.[2] We are very sad not to be able to run the code and present the results, but the algorithm is there implemented in our GitHub repository in CUDA.

### 2.3.1 Simple Implementation

The simple Hungarian method is an optimized primal-dual solution technique for solving the LSAP. The simple Hungarian method works by repeatedly reducing the cost matrix and assigning zeros to unique elements in the sets of resources and tasks until all zeros have been assigned and a minimum cost solution is found. The time complexity for the original Implementation was $\mathcal{O}(n^4)$. With optimized data structures it can be optimized to run in $\mathcal{O}(n^3)$.

The matrix formulation of the LSAP can be described as minimizing the trace of the permutation of the cost matrix $C$.

**Require:** squared cost matrix $C$
**Ensure:** Permutation $P$ minimizing the trace of $P \times C$
1: for each row $r_i$ subtract the row minimum ($\min_j r_{i,j}$)
2: for each column $c_j$ subtract the column minimum ($\min_i c_{i,j}$)
3: Cover all zeros in the matrix using the minimum number of columns and rows

4: **while** number of covered zeros is not $n$ **do**
5: Find smallest entry $a_{i,j}$ not marked
6: subtract $a_{i,j}$ from each uncovered row
7: add $a_{i,j}$ to each covered column

### 2.3.2 Shortest Path Implementation

In the shortest augmenting path method, we consider one source at a time and build from there the incremental graph, but unlike the standard Hungarian method we operate on the bipartite graph itself. That this incremental graph does not have to be built, can be seen in [Tom71]. It uses the methods from the Hopcroft-Karp, see chapter 3.3 [BDM12], to find the shortest path from a given vertex. Therefore the method introduces the costs $\pi_j$ from given vertex to a vertex $j$ via shortest path. The optimal solution is now found via Hungarian method or Edmunds-Karp algorithm.

The famous Jonker and Valgenant algorithms, but also many other algorithms are of this kind. The main difference, is how the data is preprozessed, which is also the bottleneck in Jonker and Valgenant, see chapter 4.4.4 in [BDM12]. Here, we did not focus on the preprocessing, so that we can not use these gains in performance.

## 2.4 Signature method

The signature method, proposed in the mid-1980s by Balinski, is one of the most famous dual algorithms for LSAP. Given a set of n resources and a set of m tasks, the goal is to find an assignment of tasks to resources such that the total cost of the assignment is minimized. In this

[2] This is a known problem as we have seen it mentioned in numerous GitHub problem tickets.

problem, the cost of assigning task i to resource j is represented by a cost matrix $C$, where $c_{i,j}$ is the cost of assigning task i to resource j.

The algorithm is based on the concept of duality in linear programming and utilizes a pivoting process to solve the assignment problem.

In the first phase, The dual pivot operation involves identifying two nodes in the tree, a source node s and a destination node t, and then creating two new subtrees. The algorithm then calculates the potential cost reduction delta by finding the lowest cost edge between the two subtrees and adding it to the tree. The dual variables u and v, which represent the slack in the constraints, are then updated to reflect the cost reduction.

The algorithm terminates when no further improvements can be made to the solution. At this point, the optimal assignment is found and the optimal cost can be computed by summing the costs of the assigned tasks.

The time complexity of the Dual Pivoting Algorithm is $\mathcal{O}(n^3)$ where $n$ is the number of tasks.

### 2.5 Successive Shortest Path method

The Successive Shortest Path Algorithm is an algorithm used to solve the minimum Cost Flow Problem [AMO09], it can be used to solve a modified LSAP.

To transform the Problem we add two additional vertices, a source $s$ and a sink $t$, and connect the source with the set of resources, every vertex $i$ gets a directed edge from the source to $i$ with cost 0. The sink will be connected to the set of tasks in the same way. Each edge in the network will have a capacity constraint of 1, meaning that each edge can carry a flow of at most 1 unit. The transformed Problem can be formulated as a linear program.

Let $G = (V, E)$ be a directed graph, where $V$ is the set of vertices and $E$ is the set of edges. Let $c_{i,j}$ be the cost per unit of flow on edge $(i, j) \in E$, and let $f_{i,j}$ be the flow on edge $(i, j)$. The modified constraints can be formulated as follows:

$$\text{minimize} \quad \sum_{(i,j)\in E} c_{i,j} f_{i,j} \qquad \text{(total cost)} \tag{11}$$

$$\text{subject to} \quad \sum_{j\in\delta^+(i)} f_{i,j} = \sum_{j\in\delta^-(i)} f_{j,i} \quad \forall i \in V \setminus \{s, t\} \tag{12}$$

$$\sum_{j\in\delta^+(i)} f_{s,j} = n = \sum_{j\in\delta^-(i)} f_{j,t} \tag{13}$$

$$0 \leq f_{i,j} \leq u_{i,j} = 1 \quad \forall (i, j) \in E \tag{14}$$

where $\delta^+(i)$ and $\delta^-(i)$ are the sets of outgoing and incoming neighbors of vertex $i$, respectively.

In this special case, we want to send the specific amount of flow, the number of resources, from the source to the sink. This way every resource will be chosen, and the capacity constraint ensures a one-to-one matching from resources to tasks. Since the flow value $f_{i,j}$ can just be binary, the minimizing cost function is the minimizing cost function of the original problem.

We can solve the reformulated problem of the Linear Sum Assignment Problem using solution methods for the Minimum Cost Flow problem. Specifically, we use the Successive Shortest Path algorithm . This algorithm is based on the Ford Fulkerson method [CSR13][S.738, Chapter 26], which is typically used to solve the Maximum Flow problem.

The main idea behind the method is to continuously search for augmenting paths in the residual graph, which represent potential paths for increasing the flow. At each step, the algorithm pushes

the minimum amount of residual capacity in the path along the found augmenting path, effectively augmenting the overall flow in the network. The solution to the minimum cost flow can be obtained by examining the flow value of each edge. If the flow between nodes $i$ and $j$ is present, it indicates that resource $i$ is assigned to task $j$. This binary flow representation provides a straightforward and efficient way to reconstruct the optimal solution to the Linear Sum Assignment Problem.

**Require:** Flownetwork $G = (V, E, s, t)$; $n$ number of flowamount;
**Ensure:** minimal cost flow;
1: initialize $e$.`flow` is 0 on all edges;
2: flow amount is 0;

3: **while** flow amount smaller than $n$ **do**
4: Find augmenting path $s - t$ Path $P$ in residual Graph $G_f$;
5: Find minimal residual capacity $\delta$ in Path $P$;
6: push flow amount $\delta$ on path $P$ to augment flow;
7: increase flow amount with $\delta$;
8: update $G_f$;

Since the while-loop in this algorithm runs at most $n$ times, the time complexity depends on how an augmenting path $P$ is found. In our implementation, the augmenting path is found by using an implementation of the shortest path faster algorithm. The shortest path faster algorithm is an optimized version of the Bellmann-Ford implementation, which computes an augmenting shortest Path in $\mathcal{O}(|V| \cdot |E|)$. The average runtime, however, suggests $\mathcal{O}(|E|)$, making it faster than Bellmann-Ford.

With this in mind, we can compute the overall running time of this algorithm for solving the LSAP as $\mathcal{O}(n \cdot (n + m + 2) \cdot 3n) = \mathcal{O}(n^3)$, since our modified flow network has $n + m + 2$ vertices and $3n$ edges.

# 3 Benchmarking

## 3.1 Instance definition and methodology

The test instances for the Linear Sum Assignment Problem can be found in a dedicated folder within the Github repository. These instances are stored in a file format with the extension `.mm` and contain an integer-valued cost matrix that represents the problem to be solved.[3]

We provide various instance test sets, as described in [BDM12] in chapter 4.10.1, which are hereafter called

- *Uniform Easy*, where the elements are uniformly distributed integers between 0 and 9,
- *Uniform*, where the range is 0 to 99,
- *Geometric*, where $c_{i,j} = \lfloor \|X_i - Y_j\|_2 \rfloor$ with $X, Y \sim \text{Unif}(\{0, \ldots, 99\}^{2\times\dim})$, are taken,
- *Two-Cost*, where each element is 1 with probability 0.5 and 10 000 otherwise,
- *Worst-Case*, where $c_{i,j} = (i - 1)(j - 1)$, and
- *Sparse*, as *Uniform*, but with each element set to zero with probability $1 - \frac{2 \log n}{n}$.

[3]Further information is being provided in the repo – in the directory and the `Readme.md`.

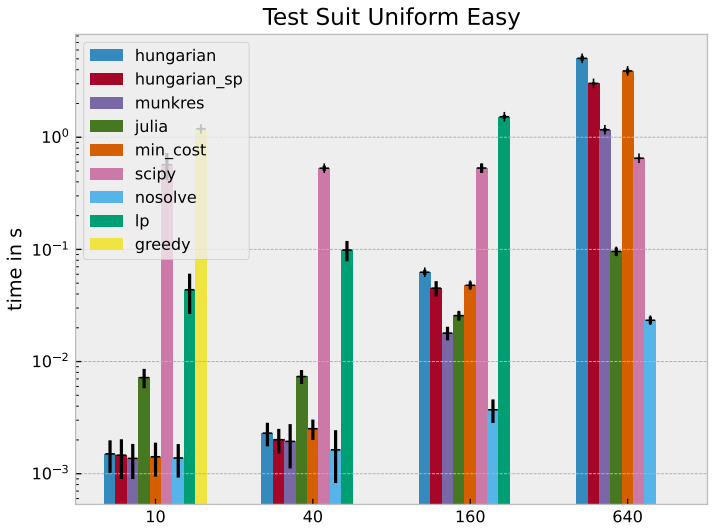


Figure 1: Comparison of different algorithms with the Uniform Easy test suit

Instructions on running the benchmarks are in the `Readme.md` of our repo. Note, that `greedy` is only run for dimension 10. Likewise, `lp` is not run for dimension 640.

### 3.2 Results

We notice, that even for $n = 10$, the combinatorial greedy approach takes the longest. Considering the asymptotic runtime, we did not need to test the other cases. Therefore it will not be displayed in the following data frames.

Next we notice, that external code did result in big performance loss on the smaller test cases, but the julia and scipy methods caught fast up. Scipy was even the fastest solver for the Worst-Case test suit, see 5. The lp method was for all test cases and instance sizes too unspecialized for this problem.

Our hungarian shortest path algorithm is probably the fastest of our provided implementations, but our munkres algorithm was much faster in the Sparse 6 and the Two-Cost (which is also sparse after one reduction) Test Suit 4.

## 4 Conclusion

In this report we presented different approaches to solving the Linear Sum Assignment Problem. Through our implementations and benchmarktesting we could determine the fastest solving method being the Hungarian method, namely the simple Hungarian method, not the Shortest Path Version.

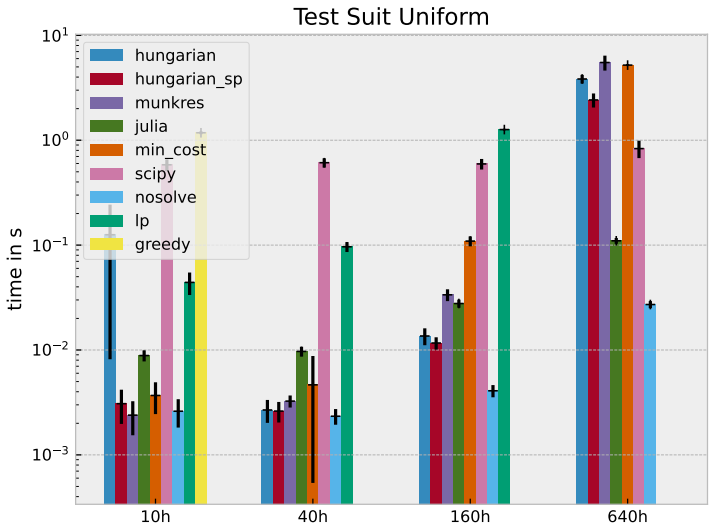


Figure 2: Comparison of different algorithms with the Uniform test suit

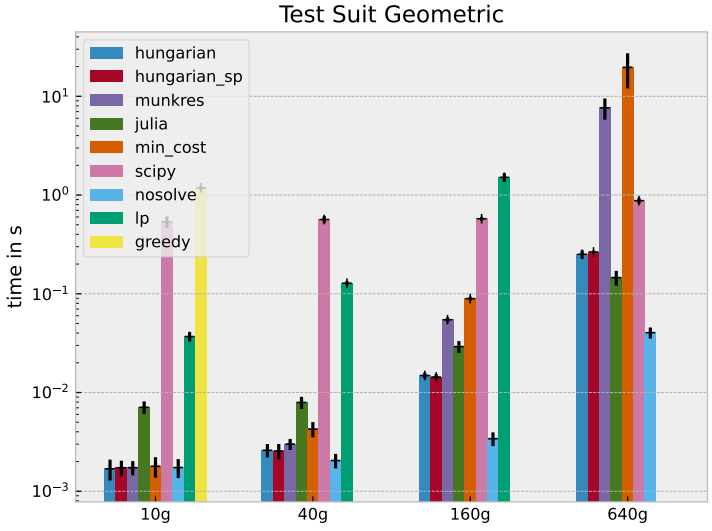


Figure 3: Comparison of different algorithms with the Geometric test suit

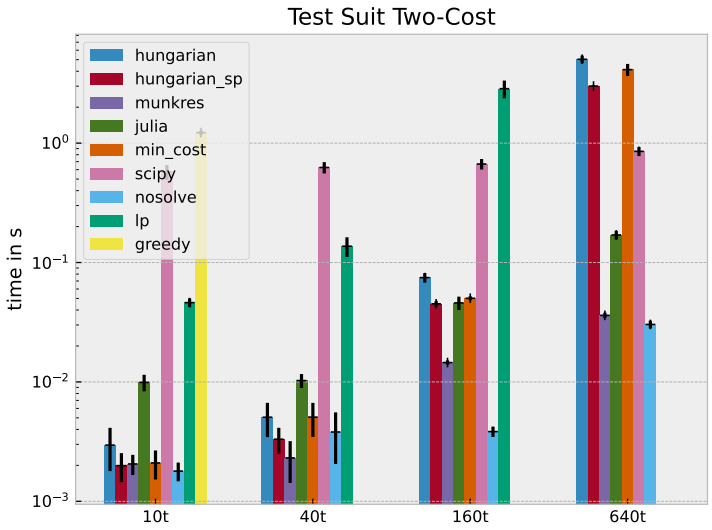


Figure 4: Comparison of different algorithms with the Two-Cost test suit

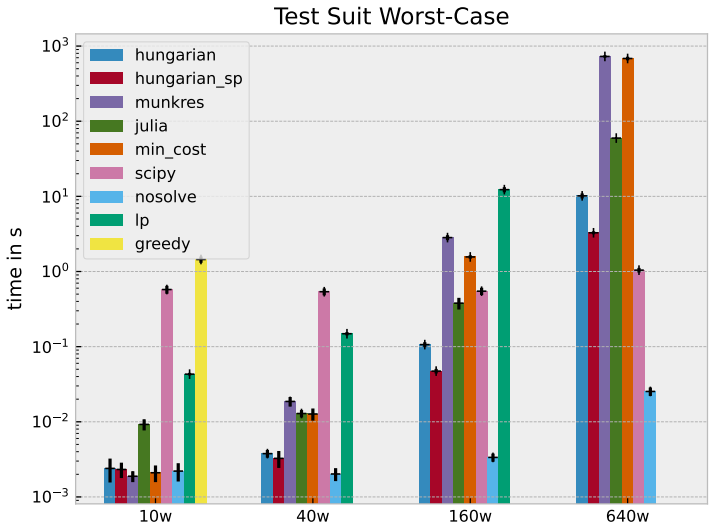


Figure 5: Comparison of different algorithms with the Worst-Case test suit

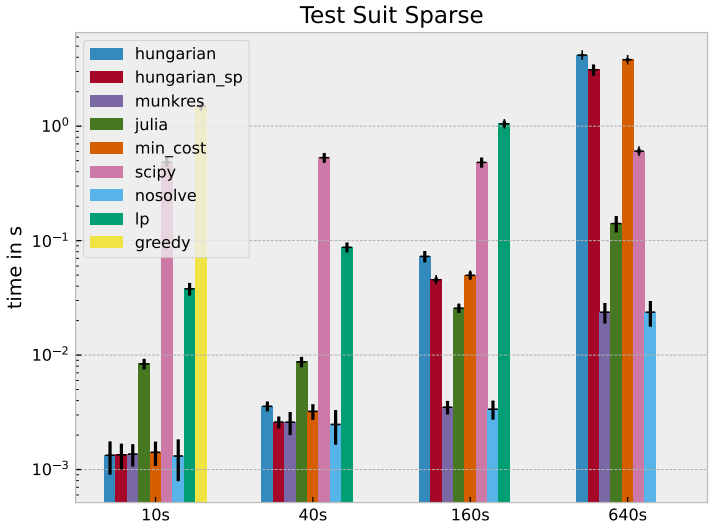


Figure 6: Comparison of different algorithms with the Geometric test suit

## References


[AMO09] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. "Minimum cost flow problemMinimum Cost Flow Problem". In: *Encyclopedia of Optimization*. Ed. by Christodoulos A. Floudas and Panos M. Pardalos. Boston, MA: Springer US, 2009, pp. 2095–2108.

[BDM12] Rainer Burkard, Mauro Dell'Amico, and Silvano Martello. *Assignment Problems*. Society for Industrial and Applied Mathematics, 2012. eprint: `https://epubs.siam.org/doi/pdf/10.1137/1.9781611972238`.

[CSR13] Rivest Cormen Leiserson, Clifford Stein, and Thomas H. Ronald Charles E. *Algorithmen - Eine Einführung*. Oldenbourg Verlag, 2013. eprint: `https://www.degruyter.com/document/doi/10.1515/9783110522013/pdf`.

[Kao+01] Ming-Yang Kao et al. "A Decomposition Theorem for Maximum Weight Bipartite Matchings". In: *SIAM Journal on Computing* 31.1 (2001), pp. 18–26. eprint: `https://doi.org/10.1137/S0097539799361208`.

[Tom71] Nobuaki Tomizawa. "On some techniques useful for solution of transportation network problems". In: *Networks* 1 (1971), pp. 173–194.