Algorithm 1015: A Fast Scalable Solver for the Dense Linear (Sum) Assignment Problem

STEFAN GUTHE, TU Darmstadt, Germany and Fraunhofer IGD, Germany
DANIEL THUERCK*, NEC Laboratories, Germany

We present a new algorithm for solving the dense linear (sum) assignment problem and an efficient, parallel implementation that is based on the successive shortest path algorithm. More specifically, we introduce the well known epsilon scaling approach used in the Auction algorithm to approximate the dual variables of the successive shortest path algorithm prior to solving the assignment problem to limit the complexity of the path search. This improves the run time by several orders of magnitude for hard to solve real-world problems, making the run time virtually independent of how hard the assignment is to find. In addition, our approach allows for using accelerators and/or external compute resources to calculate individual rows of the cost matrix. This enables us to solve problems that are larger than what has been reported in the past, including the ability to efficiently solve problems whose cost matrix exceeds the available systems memory. To our knowledge, this is the first implementation that is able to solve problems with more than one trillion arcs in less than 100 hours on a single machine.

CCS Concepts: • **Mathematics of computing → Combinatorial algorithms**; **Combinatorial optimization**.

Additional Key Words and Phrases: Successive Shortest Path Algorithm, Parallel Processing, Epsilon Scaling

ACM Reference Format:
Stefan Guthe and Daniel Thuerck. 2021. Algorithm 1015: A Fast Scalable Solver for the Dense Linear (Sum) Assignment Problem. *ACM Trans. Math. Softw.* 47, 2, Article 18 (April 2021), 27 pages. https://doi.org/10.1145/3442348

1 INTRODUCTION

The linear assignment problem (LAP) is one of the fundamental combinatorial problems. It consists of a number of sources that needs to be assigned to the same number of targets, where each possible assignment carries a certain cost. The desired solution is the assignment of all sources with minimal total costs.

An LAP with assignment variable x_{ij}, $i, j = 1 \ldots n$ and weight (or *cost*) matrix $w \geq 0$ can be expressed as the following linear program:

$$\min \sum_{ij} w_{ij} \cdot x_{ij}$$

*Work done while at TU Darmstadt.

Authors' addresses: Stefan Guthe, stefan.guthe@tu-darmstadt.de, TU Darmstadt, Graphical Interactive Systems Group, Fraunhofer Str. 5, 64283, Darmstadt, Germany, Fraunhofer IGD, Germany; Daniel Thuerck, daniel.thuerck@neclab.eu, NEC Laboratories, Germany.

0098-3500/2021/4-ART18 $15.00
https://doi.org/10.1145/3442348

subject to

$$\begin{aligned}\sum_j x_{ij} &= 1 \qquad (1)\\ \sum_i x_{ij} &= 1\\ x_{ij} &\geq 0,\ x \text{ integer}\end{aligned}$$

The LAP's constraint matrix is famously totally unimodular, i.e. all vertices of the polyhedron are integral. Thus, we drop the integrality requirement and solely work with the linear program relaxation (LP). The dual of the formulation above is:

$$\max \sum_i u_i + \sum_j v_j \tag{2}$$

subject to

$$w_{ij} - u_i - v_j \geq 0 \tag{3}$$

Here, u_i and v_j are the dual variables and $\pi_{ij} := w_{ij} - u_i - v_j \geq 0$ is denoted as *the reduced cost.* By instancing the KKT optimality criteria for the primal-dual pair, we arrive at the following: A pair of feasible primal solution x^* and feasible dual solution (u^*, v^*) is optimal iff

$$x^*_{ij}\left(w_{ij} - u^*_i - v^*_j\right) = 0\ . \tag{4}$$

Essentially, the dual problem is to maximize the sum of the dual variables for non-negative reduced costs. For simplicity, we assume that $w \geq 0$, as this can easily be guaranteed by subtracting the minimum cost value from each w_{ij}. For our implementation, we focus on problems with large, dense cost matrices that so far cannot be solved efficiently [Dell'Amico and Toth 2000]. All theoretical properties of our algorithm, however, also hold for problems with sparse matrices.

We note that the terminology used for variables and input data in the LAP varies from author to author and is often motivated by the application at hand. Similar to Jonker and Volgenant [1987], we refer to the indices i and j as "rows" and "columns", inspired by the form of the cost matrix. The expressions "source" / "target" as well as "worker" / "job" can be used equivalently. The problem may be formulated equivalently as picking a set of disjoint edges with a minimum total weight on a dense bipartite graph over the set of row nodes $R = \cup_i\{r_i\}$ and column nodes $C = \cup_j\{c_j\}$ for $i, j = 1, \ldots, n$ with edge weights w_{ij} for (r_i, c_j).

Notation. In line with the notation used in the LP above, we use w_{ij} to refer to element (i, j) of matrix w; a : serves as wildcard to denote either all columns or all rows, e.g. $w_{:1}$ is the first column of w, whereas $w_{1:}$ represents the first row of w. Similarly, an assignment $a_: = \alpha$ assigns all components of the vector a the value α; comparison operators such as $\leq, \geq$ are applied element- wise in case of a matrix or vector. Corresponding to an assignment variable x, we define a (partial) assignment $A \subseteq R \times C$ and shorthands $A(r_i)$ as well as $A(c_j)$ returning the assigned column resp. row or $\emptyset$ if these are unassigned. Putting $A(r_i)$ in square brackets $[A(r_i)]$ refers to the *index* of the assigned row resp. column, e.g. j for c_j. In general, we try to clearly distinguish between row/column indices and their representative nodes in residual graphs in the remainder of this paper.

Applications. The LAP is used in a variety of different applications. Originally it was used as a means to *assign* workers to jobs suiting their abilities [Munkres 1957]. The method is also frequently used to permute and scale sparse matrices before applying a factorization [Duff and Koster 2001], as a metric for image retrieval [Rubner et al. 2000] and for object tracking [Zhang et al. 2008]. Other areas, such as color and style transfer, try to approximate the solution to the LAP due to its computational costs [Pouli and Reinhard 2011]. When analyzing the performance of our algorithm, we need to take the properties of the applications into account: For a random cost matrix, there is a constant probability that the correct assignment is contained within the lowest $m \leq n$ cost values of any given row. For an application on the other hand, sources and/or targets may form clusters, drastically changing the distribution of cost values in a given row (see for example [Schmitzer 2015, 2016]). Due to this change, the probability that the correct assignment is contained within the lowest m cost values is no longer constant between rows, i.e. the expected performance heavily depends on the structure of the problem.

Assumptions. Our approach is designed to handle a large, dense cost matrix w. While the entries in w can potentially be calculated on-the-fly, we assume that this calculation is costly and storing at least some of the values in a matrix is beneficial. Consequently, our code consumes a function that generates w's entries on the fly. Depending on the amount of memory available, the solver can then decide by itself when to cache entries.

Contributions. We propose a modified successive shortest-path based algorithm [Engquist 1980] for solving large, dense linear assignment problems. We present an implementation of said algorithm in an LAP-solver that is able to handle problem sizes that are several orders of magnitudes larger than published in previous work. We specifically address the computational costs of solving these problems and additionally present an extension to handle problems whose cost matrix can no longer be stored in memory but can be calculated on demand. Our contributions are as follows:

- We propose a dual initialization method that drastically improves the average run time for randomly generated cost matrices as well as real problems (verified against the object tracking and image retrieval task mentioned above).
- We present a parallelization of our shortest-path based method and discuss low-level optimizations w.r.t. access patterns, memory organization and inter-thread communication during the different stages of the computation.
- We describe ways to handle the challenges that come with larger problems and propose caching strategies for saving incomplete cost matrices as well as recomputation guidelines using an additional graphics processing unit (GPU) as coprocessor.
- While the basic algorithm assumes an equal number of rows and columns, we extend our implementation to also solve the unbalanced problem with more columns than rows by including virtual rows with constant cost values. These rows are explicitly handled without requiring additional storage of cost values.

2 RELATED WORK

Fundamentally, the linear assignment problem (first mentioned by von Neumann [Von Neumann 1953] in the context of two-player zero-sum games) is an instance of the minimum cost flow problem where a graph G connects each row with all columns, i.e. forms a dense bipartite graph. Upon including two artificial nodes s, t — s connected solely to all rows, t to all columns — a minimum cost flow satisfying demand n of s, $-n$ of t and 0 of all rows and columns constitutes an optimal assignment. Both linear assignment problems and min-cost flow problems are covered by a large body of work. Focusing on the overarching algorithmic concepts, we give a brief classification and review of the different algorithms for the LAP. Our list is by no means complete; we thus encourage the reader to follow the stream of successively refined and related methods, starting with references given by Dell'Amico and Toth [2000]. Notably, the assignment problem was part of the first DIMACS implementation challenge [Johnson and McGeoch 1993].

Linear programming. The most commonly investigated approaches are based on *linear programming* resp. *flow algorithms.* Clearly, using an off-the-shelf solver for the LAP's LP-formulation works in principle but is not very efficient both in terms of storage and computational complexity. As with other integral LPs, the LAP is known to contain many redundant faces, thus causing pivot operations without progress in the solution value and therefore unnecessary steps in the Simplex algorithm. Replacing the linear algebra in the simplex with symbolic, problem-dependent computations leads to problem-specific variants such as the AB simplex [Barr et al. 1977] or the network simplex [Orlin 1997]. Both approaches restrict the selection of bases to find symbolic representations to avoid unnecessary pivots. For larger-scale problems, interior-point methods were also investigated [Goldberg et al. 1992], but work in this direction was discontinued to our knowledge.

Primal-dual methods are by far the largest class of methods for solving LAPs. Based on the Hungarian algorithm [Kuhn 1955], primal-dual methods output a sequence of primal and/or dual solution pairs that ultimately satisfy the complementary slackness conditions. The original Hungarian method with complexity $O(n^4)$ and its successors start from any given, partial primal solution and a feasible dual solution and proceed in rounds. According to the primal-dual scheme applied to the LAP, each round consists of solving a restricted primal problem given the current dual. In the Hungarian algorithm, this restricted primal is an unweighted bipartite matching on the subset of edges that have reduced costs 0, so each round itself has complexity $O(n^3)$. Over the years, several improvements and algorithmic reorganizations, most notably the advent of *alternating trees*, have resulted in variants of the algorithm with worst-case complexity $O(n^3)$ [Edmonds 1965].

Applying the primal-dual scheme to the min-cost flow instead gives rise to successive shortest path algorithms (for details, see next section) [Edmonds and Karp 1972; Tomizawa 1971]. Starting from a feasible dual solution, these algorithms iteratively find shortest paths from unassigned rows to unassigned columns on the residual graph of reduced costs. Each shortest path adds one more assigned row. Using an efficient implementation of e.g. Dijkstra's shortest path algorithm via Fibonacci heaps [Fredman and Tarjan 1987] can even reduce the theoretical complexity slightly below $O(n^3)$. To improve the run time in practice, several authors have proposed initialization procedures and heuristics to built into shortest path approaches: Jonker and Volgenant [1987] use a 3-stage initialization procedure similar to Bertsekas [1981] of *outpricing* certain columns to avoid repeated long paths late in the computation, a general issue with path and flow-based approaches. Orlin and Lee [1993] even claim that their QuickMatch heuristic cause measured linear behavior in the number of arcs ($O(n^2)$ for dense problems) based on their computational empirical evidence. However, they evaluated their approach only on randomly generated cost matricies. As of today, primal-dual methods are the approaches of choice for most problems.

Relaxation-based. Departing from flow-based approaches, Bertsekas [1988] proposes to solve the LAP by relaxing the complementary slackness conditions: He defines the ϵ - Complementary Slackness (ϵ - CS) by $-\epsilon \leq x_{ij} \left(w_{ij} - u_i - v_j \right)$ for any $\epsilon > 0$. In each iteration of his *Auction algorithm*, values or "bids" for each unassigned row i and a minimum price column j are computed based on the weights w_{ij} and the cost v_j of the row currently assigned to column j. In a second phase, prices (referring to the reduced costs from above) for columns are determined among bids and rows are assignment greedily. While this algorithm bears some resemblance to the Hungarian method with its restriction of the assignable objects, it behaves fundamentally different in practice: Whereas the Hungarian method often traverses long alternating paths in the last iterations and thereby changes a large number of assignments (serializing the computation to the point of being bounded by the size of its strongly connected components), the Auction method is known to rapidly converge to a solution close to an optimum, but then needing a higher amount of time to reach optimality at $\epsilon = 0$. The process of lowering ϵ in subsequent iterations is called *ϵ-Scaling*. From a theoretical point of view, however, its complexity is worse than that of the best augmented path approaches [Dell'Amico and Toth 2000]. The auction approach is specifically suitable for implementations on (massively) parallel architectures [Sathe et al. 2012].

Continuous approaches. Besides directly applying linear programming solvers, the LAP can be interpreted as a special instance of the *transport problem* with all demands and supplies set to 1 and using only discrete quantities. While for the discrete transport problems min-cost flow approaches can lead to efficient algorithms, other approaches are feasible as well: Every integral and non-integral feasible solution of the LAP and, by extension, the transport problem can be interpreted as a doubly-stochastic assignment matrix. An algorithm for finding such a doubly-stochastic matrix is presented by Sinkhorn and Knopp [1967]. As the resulting matrix is not binary (doubly-stochastic matrices only have row- and column sums of 1), an additional step of rounding is necessary. Greedily selecting the largest entry in each row while respecting the LAP constraints usually result in a good approximation to the optimum. Schmitzer [2015] presents an approach to solve the dense LAP by replacing it with a series of sparse problems. Each of the generated subsets of feasible assignments guarantees feasibility of all excluded dual constraints after their solution. Efficient strategies for the subset selection, called shielding neighborhoods, are presented for offset metric (i.e. Euclidean) type cost functions. Further acceleration is provided by a multi-scale scheme [Schmitzer 2016]. However, similar to the Auction algorithm, these solvers need a large number of iterations to converge to the discrete solution. In addition, even if the algorithm converged close to a discrete solution, there is no guarantee that this is indeed the assignment with the minimal cost.

Implementations. For all mentioned algorithmic approaches, there are several implementations available [Duff and Koster 1999; Dufossé et al. 2014; Vasconcelos and Rosenhahn 2009]. Dell'Amico and Toth [2000] discuss computational results for several implementations, however all benchmark problem fall short of $n = 10,000$ due to the computation time requirements or due to memory consumption. Most recently, Date and Nagi [2016] presented a GPU-supported implementation for dense problems with $n = 40,000$. They do, however, not address the issue that larger problems often do not fit into the memory of all available GPUs and/or into the system memory.

Instead of simply presenting another heuristic for decreasing the computational costs of solving a LAP, we aim at delivering an easy to use framework that allows to solve LAPs of arbitrary size using the available resources, both computational and memory, as efficiently as possible.

3 A SUCCESSIVE SHORTEST PATH ALGORITHM WITH APPROXIMATE DUAL INITIALIZATION

Our algorithm for solving the dense LAP to global optimality is of the type *successive shortest path* algorithm with its core structure based on Jonker and Volgenant [1987]. Our improvement, which we call ϵ-Pricing, is based on a re-interpretation of the LAP's optimality conditions. We therefore sketch the main ideas behind successive shortest path algorithms as primal-dual algorithm before discussing the changes for ϵ-Pricing. For brevity, we do not discuss the algorithms' background in regards to min-cost flow problems - although the following derivation is mostly an instance of the corresponding theorems on flow with the formulation changed to reflect the naming conventions of the LAP.

3.1 Deriving successive shortest-path methods

As described in Section 1, we denote the dual variables by $u, v \in \mathbb{R}^n$. Any feasible dual solution u, v satisfies the dual constraints

$$w_{ij} - u_i - v_j \geq 0,\ i, j = 1, \dots, n. \tag{5}$$

For a *partial* assignment $x \in \{0,1\}^{n \times n}$ that satisfies the primal constraints for k rows and columns and is 0 everywhere else, we define the *residual graph* $G_x = (V, E_x)$ that can be used to add an unassigned row to the solution by finding a shortest path from s to t through row r_i, where

$V = R \cup C \cup \{s,t\}$ and

$$\begin{aligned} E_x = \quad & \{(s, r_i) : i = 1, \dots, n\} \\ \cup \quad & \{(c_j, r_i) : x_{ij} = 1\} \\ \cup \quad & \{(r_i, c_j) : x_{ij} = 0\} \\ \cup \quad & \{(c_j, t) : x_{:j} = 0\} \end{aligned} \tag{6}$$

with costs $\pi = 0$ except for

$$\pi_{ij} = w_{ij} - u_i - v_j, \text{ where } x_{ij} = 0. \tag{7}$$

Theorem 3.1. *Let d the distance vector of shortest paths from s to all other nodes in V given a feasible dual solution u, v. Then, the duals u', v' resulting from the update*

$$u'_i := u_i - d_{r_i} \tag{8}$$

$$v'_j := v_j + d_{c_j} \tag{9}$$

are still feasible.

Proof. Since d is the all-shortest paths distance vector, for every $c_j, j = 1, \dots, n$ we have $d_{c_j} \leq d_{r_i} + \pi_{i,j}$ for all $i = 1, \dots, n$ with $x_{i,j} = 0$. Therefore,

$$\begin{aligned} 0 \leq \quad & d_{r_i} - d_{c_j} + (w_{ij} - u_i - v_j) \\ = & \, w_{ij} - (u_i - d_{r_i}) - (v_j + d_{c_j}) \\ = & \qquad w_{ij} - u'_i - v'_j \end{aligned} \tag{10}$$

□

Given a shortest path $(s \longrightarrow t) = (s, r_{i_1}), (r_{i_1}, c_{j_1}), \dots, (c_{j_{k-1}, t})$, this inequality may be sharpened into an optimality criterion, presented as Theorem 3.2.

Theorem 3.2. *Let (r_i, c_j) an edge in the shortest path $(s \longrightarrow t)$ as evaluated by d. Then,*

$$w_{ij} - u'_i - v'_j = 0. \tag{11}$$

Proof. For edges (r_i, c_j) on the shortest path, the path's suboptimality property yields

$$d_{c_j} = d_{r_i} + \pi_{ij} \tag{12}$$

and – using the term rearrangements from above – the claim. □

For a partial assignment x and any feasible duals u, v, finding a shortest path $(s \longrightarrow t)$ and modifying the x by including all edges (r_i, c_j) in the path while excluding assignments (c_j, r_i) accordingly yields a larger assignment x'. By Theorems 3.1 and 3.2, the updated duals u', v' are

- feasible and
- satisfy all *complementary slackness constraints.*

Unless x' is a complete assignment, however, we still lack primal feasibility. This insight then gives rise to successive shortest path methods, a class of dual algorithms. A descriptive pseudocode is given in Algorithm 1.

In each iteration of Algorithm 1 (an *augmentation*), the assignment is extended by one more column (for the concrete mechanism, see Algorithm 2, line 23ff.); thus the loop is executed n times. Since all edge weights are nonnegative, Dijkstra's algorithm can be used for finding a shortest path. As the graph contains a total of n^2 edges, the Dijkstra's algorithm worst-case complexity becomes $O(n^2)$ leading to an overall complexity of $O(n^3)$. Note that there is no priority queue required for dense problems since all path costs have to be checked and updated in each step. In practice,

ALGORITHM 1: Basic dual successive shortest path algorithm.

```
input  : Weight matrix $w \geq 0$
output : Optimal assignment $x^*$
1 $x^0 := 0, u^0 := 0, v^0 := 0$
2 for $k := 1$ to $n$ do
3     Construct $G_{x^{k-1}} = (V, E_{x^{k-1}})$
4     Find a shortest path $(s \longrightarrow t)$ on $G_{x^{k-1}}$
5     Modify assignment $x^{k-1}$ by including all edges $(r_i, c_j)$ and excluding prior assignments $(c_j, r_i)$,
        yielding $x^k$
6     Update duals to $u^k, v^k$ as in Theorem 3.1
7 end
```

several adjustments to the algorithm and extended preprocessing (see e.g. Jonker and Volgenant [1987]) lead to better average run time but do not improve the worst-case complexity. Typically, the final augmentation is the bottleneck as the number of scanned columns can get close to the maximum of n. This is both true for a purely random cost matrix, where the last 10% of assignments empirically show a rapid increase in costs (see Figure 1), as well as for all other tests where we can see a linear increase of the number of scanned columns throughout the entire run (see Figures 1 and 2). Furthermore, traditional implementations of the successive shortest path approach run into issues if the optimal solution w.r.t. $A(r_i)$ and $A(c_j)$ is not unique since all columns with the same cost will be scanned multiple times. We therefore focus on improving the final augmentations and avoid scanning columns with equal costs repeatedly by always preferring unassigned columns in this case which is the weighted equivalent to the lookahead technique of Duff [Duff 1981].

3.2 The implementation of Jonker and Volgenant

After an initial subset of assignments is found using column and row reductions similar to the Hungarian algorithm, Jonker and Volgenant [1987] switch to the augmented path search. In order to achieve a correct solution, this initial subset may be empty but it has to be optimal given the unassigned rows. In fact, the initial assignment can lead to numerical instabilities for non-integer costs. We thus omit it in our approach.

Our augmented path search, as given in Algorithm 2, consists of three separate stages for each row i. First the shortest augmenting path is found based on the reduced cost using a Dijkstra search. Next, the assignments along the shortest path are updated in order to add the new row i to the solution. Finally, the dual variable v is updated based on the minimal distance to an unassigned column for each column that was scanned.

3.3 ϵ-Pricing

We now present our remedy to the bottleneck of scanning almost all columns during the shortest path search in the later augmentations. In the following, we assume that, as in our implementation, Dijkstra's algorithm is used for path finding. Following the dynamic programming principle, Dijkstra's algorithm exploits the suboptimality principle, which transfers to Algorithm 2.

LEMMA 3.3. *In all augmentations $i \geq 0$ in Algorithm 2, the following invariant is maintained: Let $\Pi : 1, \ldots, n \longrightarrow 1, \ldots, n$ be a permutation such that $\mathcal{S}^{k+1} = \mathcal{S}^k \cup \{\Pi_k\}$ for all $k \leq i$. Then $0 \leq d^1_{\Pi_1} \leq d^2_{\Pi_2} \leq \ldots \leq d^k_{\Pi_k}$, i.e. the Dijkstra algorithm sorts all visited columns by their distance from r_i in $G_{x^{k-1}}$.*

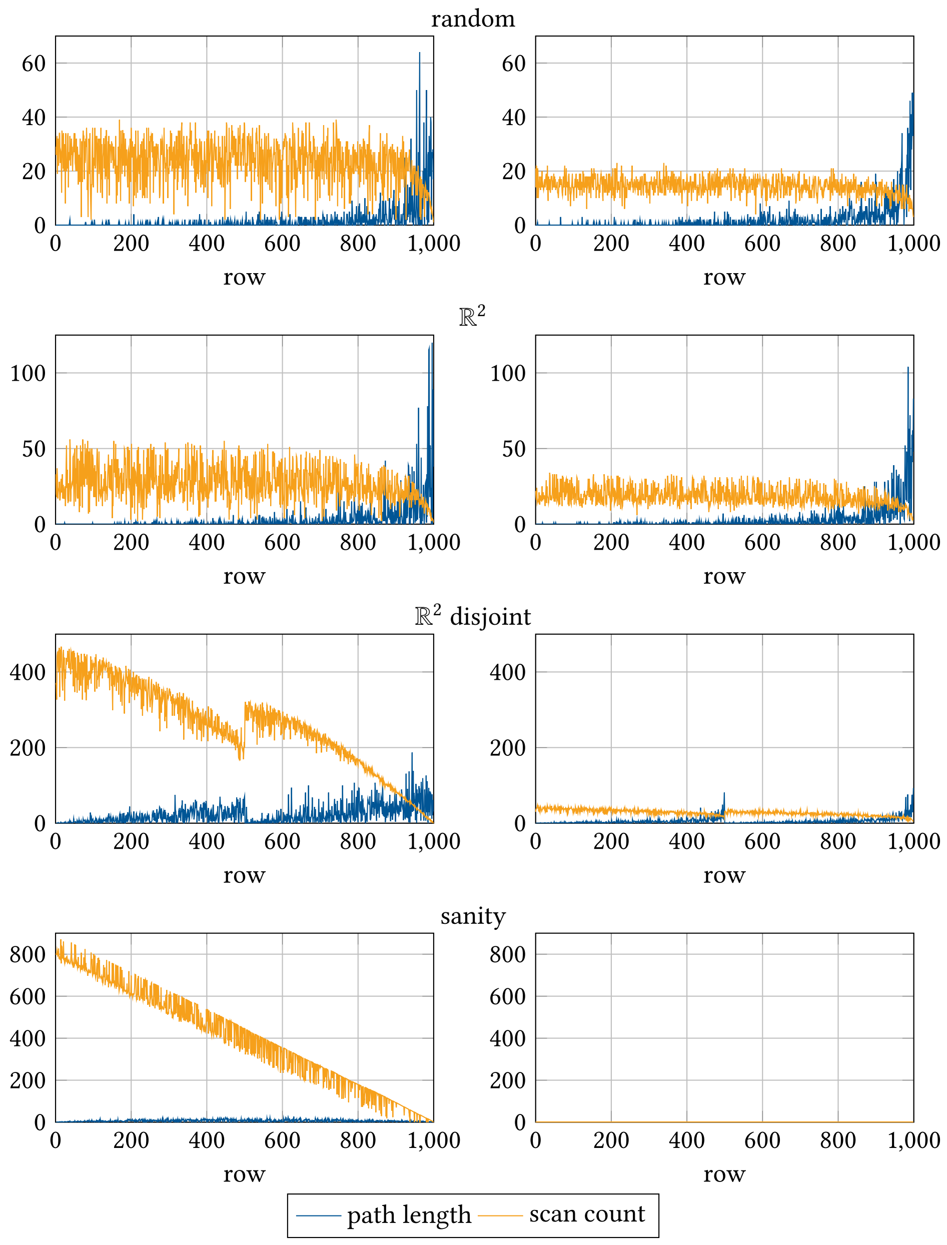

Fig. 1. Path length and scan count during augmented path search using the straight forward successive shortest path search (left) compared against our approach (right) on random cost matrices of size 1000×1000. The random seed was fixed for all matrices in this figure.

PROOF. Having omitted the dual variable u (since, given a feasible solution x inducing an assignment A, we can recover u by setting $u_i = w_{iA(i)} - v_{A(i)}$) from the reduced costs, we continue using the shorthand $\tau_{ij} := \pi_{ij} + u_i \geq 0$ from Algorithm 2.

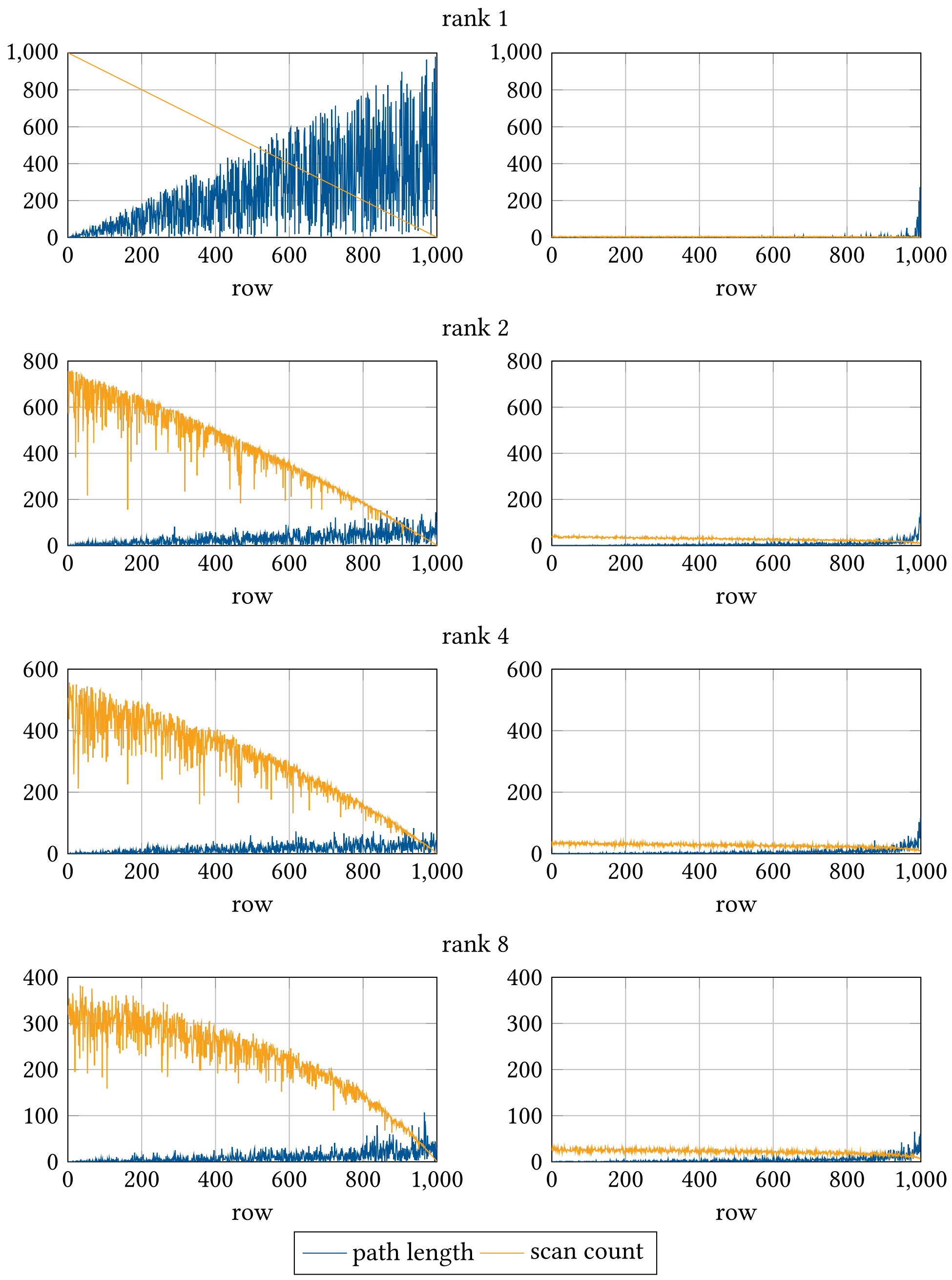

Fig. 2. Path length and scan count during augmented path search using the straight forward successive shortest path search (left) compared against our approach (right) on low rank cost matrices of size 1000×1000. The random seed was fixed for all matrices in this figure.

ALGORITHM 2: Basic successive shortest path algorithm consisting of a Dijkstra path search for the closest unassigned column, followed by an update to the assignment and the dual variable v. For convenience, we use the shorthand $\tau_{\mu\lambda} = w_{\mu\lambda} - v_{\lambda}^{i-1}$.

```
input  : Weight matrix $w \geq 0$
output : Optimal solution to the linear assignment problem in $A$
$v_{\cdot}^{0} := 0; A := \emptyset$;
for $i := 1$ to $n$ do                                         // Iterate over all rows
    $p_{\cdot} := i$;                                          // Dijkstra path search
    $d^0 := \tau_{i:}$;
    $\lambda^0 := \operatorname{argmin}_j d_j^0$;
    $\mathcal{S}^0 := \{\lambda^0\}$;
    $k := 0$;
    while $A(c_\lambda) \neq \emptyset$ do                     // Search until unassigned column found
        $k := k + 1$;
        $d^k := d^{k-1}$;
        $\mu^k := [A(c_{\lambda^{k-1}})]$;
        $\Delta := d_{\lambda^{k-1}}^k - \tau_{\mu^k \lambda^{k-1}}$;
        foreach $j \notin \mathcal{S}$ do
            $\gamma := \Delta + \tau_{\mu^k j}$;
            if $\gamma < d_j^{k-1}$ then                        // Path through $r_{\mu^k}$ is shorter
                $p_j^i := \mu^k$;
                $d_j^k := \gamma$;
            end
        end
        $\lambda^k := \operatorname{argmin}_{j \notin \mathcal{S}} d_j^k$;   // Continue search with column $c_{\lambda^k}$
        $\mathcal{S}^k := \mathcal{S}^{k-1} \cup \{\lambda^k\}$;
    end
    $j := \lambda^k$;                                          // Dijkstra search complete
    repeat                                                     // Update assignment based on shortest path
        $\alpha := j$;
        $\beta := p_j$;
        $j := [A(r_\beta)]$;
        $A(c_\alpha) := r_\beta$;
        $A(r_\beta) := c_\alpha$;
    until $\beta = i$;
    foreach $j \in \mathcal{S}^k$ do                           // Update $v$ based on minimal costs found during search
        $v_j^i := v_j^{i-1} + d_j^k - d_{\lambda^k}^k$;
    end
end
```

Let i be a fixed augmentation. Dual feasibility implies $d^0 := \tau_{i:} \geq 0$. In each iteration of the inner while-loop (line 8ff.), $\lambda^k := \operatorname{argmin}_{j \notin \mathcal{S}} d^k$ where

$$d_j^k = \min(d_j^{k-1}, d_j^{k-1} - \tau_{\mu^k \lambda^{k-1}} + \tau_{\mu^k j}), j \notin \mathcal{S}.$$

Plainspoken, this loop probes the distances to all not-yet-visited column nodes via r_{μ^k}.

The permutation Π describes the order of c_λ's visited during iteration k. Adding up the costs for the path $(r_i, c_{\lambda^1}), (c_{\lambda^1}, A(\lambda^1)), \ldots, (c_{\lambda^k}, A(\lambda^{k-1}))$ leads to

$$d^k_{\lambda^k} = d^{k-1}_{\lambda^{k-1}} + (\tau_{\mu^k \lambda^{k-1}} + \tau_{\mu^k \lambda^k}).$$

Since $\Pi_k = \lambda^k, \Pi_{k-1} = \lambda^{k-1}$, we have

$$\begin{aligned} d^k_{\Pi_k} - d^{k-1}_{\Pi_{k-1}} &= d^k_{\lambda^k} - d^{k-1}_{\lambda^{k-1}} \\ &= \tau_{\mu^k \lambda^{k-1}} + \tau_{\mu^k \lambda^k} \\ &\geq 0 \end{aligned}$$

By induction, this argument holds for all $k \leq i$. □

Lemma 3.3 states that Dijkstra's algorithm sorts the (column-) nodes in the residual graph by their distance from the augmentation row i (resp. node r_i) and in extension also from the source s. Thus, all c_j closer to s than λ^k must have been scanned already; otherwise, $c_j = c_{\lambda^k}$. Each augmentations' final λ^k therefore partitions the set of columns into two – those closer to the source and those further away (or *unscanned*).

Our ϵ-Pricing leverages that idea in combination with a slight reinterpretation of the optimality constraints (primal / dual feasibility and complementary slackness) on a graph $O_{(x*,u*,v*)} = (G, V_O)$ with $V_O = \{(r_i, c_j) : i, j = 1, \ldots, n\}$.

LEMMA 3.4. *Assume that the duals before augmentation $i < n$ are optimal, i.e. $u^{k-1} = u^*, v^{k-1} = v^*$. Let x^* be the unique optimal assignment for u^*, v^*. Then, $(r_i, A(c_i))$ is the shortest path found in the first Dijkstra search ($k = 1$) on G_{x*}.*

PROOF. Optimality conditions for x^*, u^*, v^* imply that

$$w_{ij} - u_i - v_j = 0 \iff x_{i,j} = 1.$$

Following our notation, $x_{i[A^*(c_i)]} = 1$, where A^* represents the optimal assignment x^*.

By feasibility, $w_{ij} - u_i - v_j \geq 0$, hence for $i = 1, \ldots, n$ we have

$$w_{iA(c_i)} - u_i - v_{a^c_i} = 0 \leq w_{ij} - u_i - v_j \text{ for } j = 1, \ldots, n$$

and thus $(r_i, A(c_i))$ is the shortest path from r_i to any unassigned column node in the residual graph $G_{x^{k-1}}$.

It remains to be shown that the edge $(r_i, A(c_i))$ is actually a part of the residual graph $G_{x^{k-1}}$. By inductively applying the argument made above, we see that all assignments in happening steps $i' < i$ of the form $(r_{i'}, A(c_{i'}))$, reconstructing the assignment x^*. Thus, $A(c_i)$ is unassigned before augmentation i and, in turn, $(r_i, A(c_i))$ is an edge in $G_{x^{k-1}}$. □

With Lemma 3.4, we conclude: *given optimal duals, each shortest path in each augmentation is exactly one edge long* and scanning is kept minimal. This, in turn, means that each augmentation only adds one column to the list of scanned columns: the (optimally) assigned one.

ϵ-Pricing approximates the optimal dual by forcing the duals faster toward optimality: In each augmentation phase, every scanned column's dual is decreased by an $\epsilon \geq 0$ (for details, see Algorithm 3). This, in turn, attempts to slowly push all nodes on the path from r_i to the newly assigned column node $c_{A(r_i)}$ further away from s, making them more and more unlikely to be scanned in the subsequent augmentations. Since edge weights are only ever increased, Theorem 3.1 is still valid; Theorem 3.2, however, is violated - the complementary slackness constraints end up potentially being violated.

ALGORITHM 3: Modified version using ϵ to avoid continued scanning of the same set of columns. v can be at most $-n\epsilon$ too small (see Equation 13). Again, for convenience, we use the shorthand $\tau_{\mu\lambda} = w_{\mu\lambda} - v_{\lambda}^{i-1}$.

```
input  : Weight matrix $w \geq 0$
output : Optimal solution to the linear assignment problem in $A$
[$\epsilon_0, \epsilon_{min}, v^0$] := $estimateEpsilonAndV(w)$;         // estimate initial epsilon and v
$\epsilon := \epsilon_0$;                                                  // assign initial epsilon
while $\epsilon \geq 0$ do
    $A := \emptyset$;
    for $i := 1$ to $n$ do                                                 // Iterate over all rows
        ...
        $\bar{d}^0 := d^0_{\lambda^0}$;
        while $A(c_\lambda) \neq \emptyset$ do                            // Search until unassigned column found
            ...
            $\Delta := \bar{d}^{k-1} - \tau_{\mu^k \lambda^{k-1}}$;
            ...
        end
        $j := \lambda^k$;                                                  // Dijkstra search complete
        $\bar{d}^k := \max \bar{d}^{k-1}, d^k_{\lambda^k}$;
        ...
        foreach $j \in \mathcal{S}^k$ do                                   // Update $v$ based on minimal costs and $\epsilon$
            $v^i_j := v^{i-1}_j + d^k_j - \bar{d}^k - \epsilon$;
        end
    end
    $\epsilon := getNextEpsilon(\epsilon, \epsilon_{min})$;               // Get next $\epsilon$ or $-1$ if $\epsilon$ was 0
end
```

Relation to the auction algorithm. Setting $\epsilon > 0$ and performing n augmentations leads to a feasible, but not an optimal assignment w.r.t the original weights w. In fact, such a solution only satisfies relaxed slackness conditions

$$x_{ij}(w_{ij} - u_i - v_j) \geq -n\epsilon \tag{13}$$

which draws a close connection to Bertsekas' auction algorithm [Bertsekas 1988] and his *outpricing* of scanned columns [Bertsekas 1981]. In fact, our ϵ-Pricing resembles the (unnamed) procedure of adding ϵ to bets in the auction algorithm - both lead to a solution that only satisfy $(n)\epsilon-$ resp. ϵ-complementary slackness. Yet, there are important differences between the two approaches:

- In the auction algorithm, all $\epsilon-$augmented bids are collected and the prices are modified; during the next bidding rounds, due to the way values and bids are computed, an increment of ϵ is added to all bids. This, in effect, only allows a dual modification of ϵ in each auction; our pricing strategy allows at max an offset of $n\epsilon$ (depending on the scaling strategy).
- A bidding round does not guarantee an increase in the number of assigned columns - an augmentation does.
- The number of bidding rounds in the auction algorithm required to converge to a solution satisfying ϵ-CS roughly corresponds to $\max w_{ij}/\epsilon$ [Bertsekas 1988] - each successive shortest path iteration requires exactly n augmentations.

To avoid any confusion, we would like to note that both in the auction algorithm as well in our implementation, ϵ is decreased after finding each new assignment - this process is called ϵ-scaling.

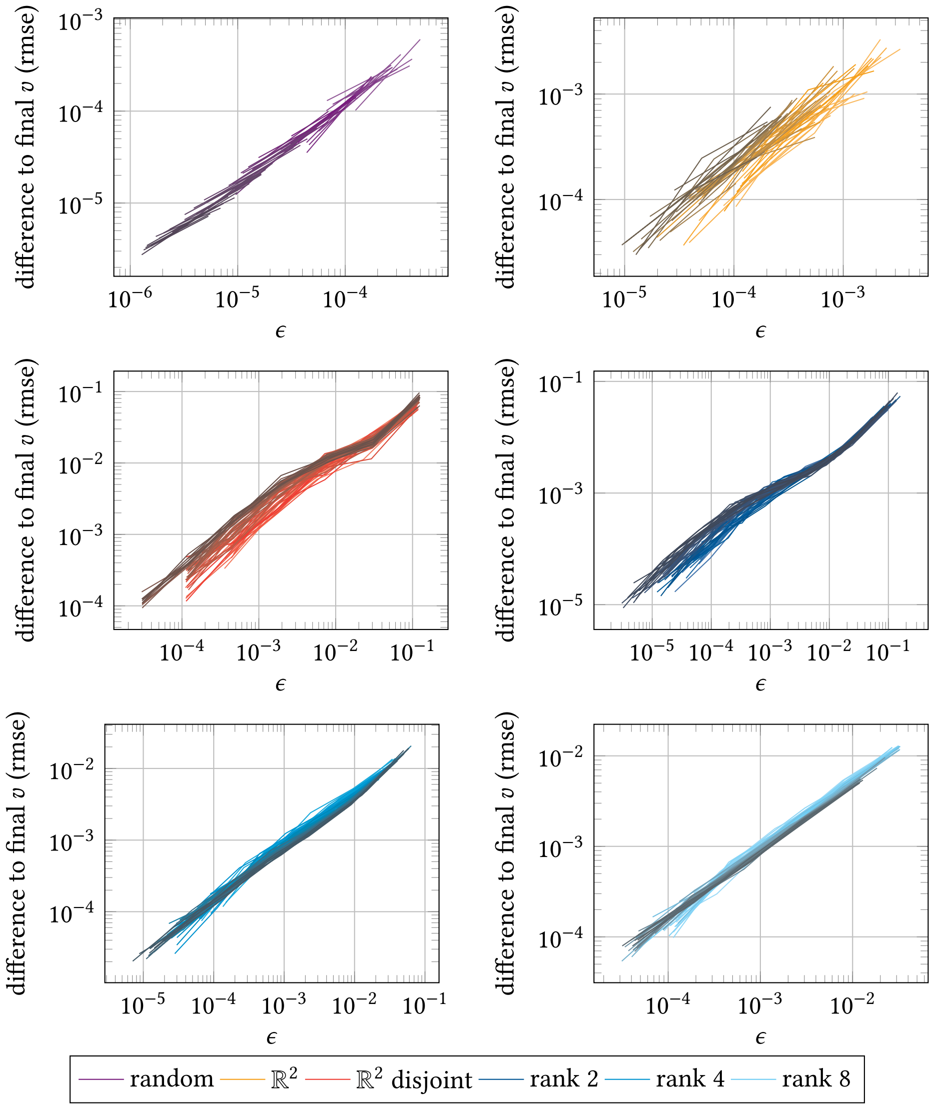

Fig. 3. Relation between ϵ decay and difference to final values of the dual variable v for various problem sizes and classes. Matrix sizes range from 1000×1000 (light) to 32000×32000 (dark) with 5 different matrices per size. Note that sanity and rank 1 matrices do not require ϵ-Pricing due to the initial estimate of v, i.e. ϵ_0 was set to 0.

Lastly, we would like to point out that for a fixed ϵ, Algorithm 3 basically performs one dual SSP pass over the problem. The resulting primal-dual pair $(x^\epsilon, u^\epsilon, v^\epsilon)$ – recovering u from v as pointed out for the feasible primal solution x^ϵ – is feasible. Hence, our algorithm could also be interpreted as a primal-dual analogue to the auction algorithm.

3.4 Setting ϵ

In order to complete Algorithm 3, we need to give details on *estimateEpsilonAndV* and *getNextEpsilon*, which means

- an initial dual solution v_0,
- and initial ϵ^0 as well as
- an ϵ-decay algorithm.

For a general primal-dual linear program (LP) pair

$$\min c^\top x \text{ s.t. } Ax = b, x \geq 0$$

$$\max b^\top y \text{ s.t. } A^\top y + s = c, s \geq 0$$

it holds that

$$c^\top x - b^\top y = c^\top x - x^\top A^\top y = x^\top (c - A^\top y) = x^\top s, \tag{14}$$

i.e. the dual gap for any feasible primal-dual pair (x, y, s) is equal to the complementary slackness. In the case of the matching linear assignment problem, this translates to

$$\sum_{i,j} w_{ij} x_{ij} - \sum_i u_i - \sum_j v_j = \sum_{i,j} x_{ij}(w_{ij} - u_i - v_j).$$

Since Algorithm 3 follows the primal-dual principle, for each ϵ, the resulting primal-dual pair (x^ϵ, v^ϵ) is feasible. Comparing v^ϵ and the optimal dual v^* for the original problem delivers

$$\begin{aligned} 0 = & \sum_{i,j} x^*_{ij}(w_{ij} - u^*_i - v^*_j) \\ = & \sum_{i,j} x^*_{ij}(w_{ij} - u^*_i - (v^\epsilon_j + \nabla_j)) \\ = & \sum_{i,j} x^*_{ij}(w_{ij} - u^*_i - v^\epsilon_j) - \sum_{i,j} x^*_{ij} \nabla_j \\ = & \sum_{i,j} x^*_{ij}(w_{ij} - u^*_i - v^\epsilon_j) - \sum_j \nabla_j \\ \leq & \sum_{i,j} x^\epsilon_{ij}(w_{ij} - u^\epsilon_i - v^\epsilon_j) - \sum_j \nabla_j \end{aligned}$$

where $\nabla_j = v^*_j - v^\epsilon_j$. This, in turn, implies

$$\sum_j \nabla_j \leq \sum_{i,j} x^\epsilon_{ij}(w_{ij} - u^\epsilon_i - v^\epsilon_j) = \sum_{i,j} w_{ij} x^\epsilon_{ij} - \sum_i u^\epsilon_i - \sum_j v^\epsilon_j \tag{15}$$

Thus, the average difference of v^ϵ to v^* is a lower bound for the complementary slackness as well as the duality gap for the primal-dual pair (x^ϵ, v^ϵ). Since Algorithm 3 permits up to n ϵ-modifications per dual variable, we can extract the bound $\nabla_j \leq n\epsilon$, i.e. $\sum_j \nabla_j \leq n^2\epsilon$. In Figure 3, we evaluate experimentally the relation between a given ϵ and $\frac{1}{n}||\nabla||$. Clearly, we notice a high correlation over various problem classes, leading to an average dual approximation of roughly $O(\epsilon)$.

Hence, ϵ's decay determines the ratio of scanned columns vs. the dual approximation and, by the derivation above, the duality gap. Increasing ϵ's rate of decay leads to a faster decrease in the duality gap at the cost of more scanned columns. Since we can measure the duality gap after each iteration of Algorithm 3, we use it as the driving force to determine matching ϵ. This strategy resembles the process of following the central path in interior point method for LPs where each iteration reduces the complementary slackness (and, in turn, the duality gap) such that $x_i s_i = \mu, \mu > 0$ and decay μ.

Initial ϵ and dual solution. In order to generate ϵ_0 and v_0, we use fast heuristics to produce lower and upper bounds on the optimal solution. Potential candidates for the upper bounds are any feasible assignments, e.g.:

- $c^u_{\text{max}} = \sum_i \max_j w_{ij}$
- $c^u_{\text{id}} = \sum_i w_{ii}$
- c^u_{greedy}, the solution generated by greedily picking the lightest leftover edge and including that into the matching or following various orders of iterating rows and assigning their cheapest free column.

Lower bounds can come from

- $c^l_{\text{min}} = \sum_i \min_j w_{ij}$
- c^l_x, the dual solution generated by using any feasible primal x (inducing an assignment A) and setting $v_{A(i)} = w_{iA(i)}, u_i = 0$.

We note that for any modified cost function $\tau_{ij} = w_{ij} - u_i - v_j$, $\sum_i \min_j \tau_{ij} + \sum_i u_i + \sum_j v_j$ is also a lower bound if (u, v) are feasible duals to the original problem.

In our implementation, we use the lower bound $cost^l = c^l_{\text{min}}$ as well as the upper bound $cost^u = c^u_{\text{id}}$. A simple lower bound on ϵ_0 can then be calculated as

$$\epsilon_l := \frac{cost^u - cost^l}{16n^2} \tag{16}$$

where n is the nuber of rows or columns and the 16 comes from fitting curves in Figure 3. At the same time, we can calculate an approximation of the dual variables $u^0 := \min_j w_{:j}$ and $v^0 := \min_i w_{i:} - u_0$. Given the approximated dual and $\tau^0_{ij} := w_{ij} - u^0_i - v^0_j$, we update the estimates for both the lower bound $cost^l_{\tau^0}$ and $cost^{\tau^0}_u := \sum_i w_{i\,\text{argmin}_{j \neq A'(k)|k<i}\, \tau^0_{ij}}$. In case $cost^l_{\tau^0}$ is equal to $cost^{\tau^0}_u$, u^0 and v^0 represent the correct dual variables, immediately solving the LAP.

Otherwise, we check if the reduction of the gap between upper and lower bound was sufficient for u^0 and v^0. If the ratio ${cost^u - cost^l}/{cost^u_{\tau^0} - cost^l_{\tau^0}}$ is smaller than 4, a different order for the greedy approach is selected. For this, the rows are sorted by the difference $\min_{k \neq \text{argmin}_l \tau^0_{lj}} \tau^0_{kj} - \min_k \tau^0_{kj}$ in decreasing order. This order is given as a permutation $p(i)$. Assuming that a greedy selection based on this order produces the correct solution, we calculate a new set of dual values v^1 and u^1 as follows: Every time we pick a new row $p(i)$, we update all $\{v^1_j \mid j \in A'(p(1) \ \ldots \ p(i-1))\}$ so that no $j \in A'(p(1) \ \ldots \ p(i-1))$ will be selected. We also take prior rows $p(l)$ into account so that they will not select any column that was selected by rows $p(k)$ with $k < l$. Finally, we update the estimates for the upper and lower bounds with $\tau^1_{ij} := w_{ij} - u^1_i - v^1_j$ again as $cost^{\tau^1}_l$ and $cost^{\tau^1}_u := \sum_p (i) w_{p(i)\,\text{argmin}_{j \neq A'(p(k))|k<i}\, \tau^1_{p(i)j}}$. Using the maximum lower bound $cost^l_{max}$ and minimum upper bound $cost^u_{min}$ found during this process, we can now calculate

$$\epsilon_0 := \frac{\left(cost^u_{min} - cost^l_{max}\right)^{\frac{3}{2}}}{n\sqrt{cost^u - cost_l}} \tag{17}$$

where the coefficients have been, again, deduced by fitting against the curves in Figure 3. If ϵ_0 is larger ϵ_l, we set both to 0, reverting to the original SSP algorithm. Note that even though this is only a heuristic for calculating these bounds, we did not encounter a single cost matrix where this approach did not produce reasonable bounds.

ϵ-scaling. After each iteration, we update ϵ based on the total modification to v that based on epsilon v_{eps} and the modification that was not based on epsilon v_d. If v_d is larger than v_{eps}, the

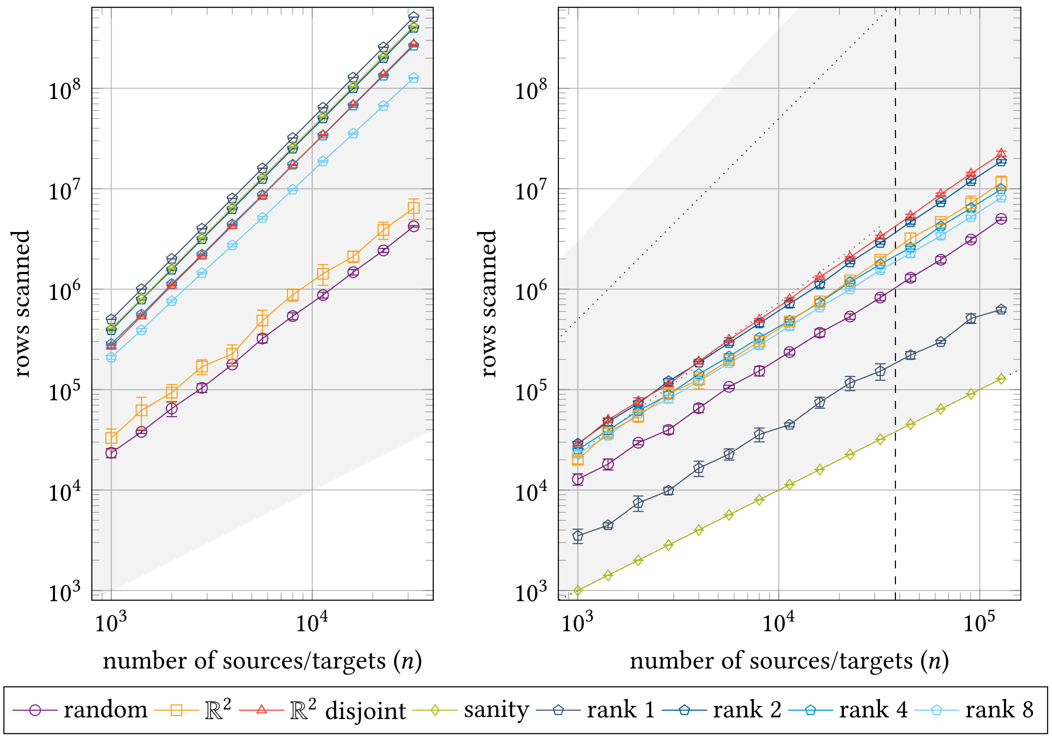

Fig. 4. Comparison of the numbers of rows scanned in the Dijkstra search between the original SSP algorithm (left) and our ϵ-Pricingapproach (right). Upper and lower theoretical bounds of each approach are given by the grey area and the dotted line on the right. Eventually all cost matrices require fewer rows to be scanned with our ϵ-Pricingthan a random cost matrix with the original SSP algorithm (dotted cyan line on the right).

current value of ϵ was is already low enough to switch to $\epsilon = 0$ in the next iteration. Otherwise we calculate the next

$$\epsilon = \min\left(\frac{\epsilon}{4}, \frac{v_{eps}}{8n}\right). \tag{18}$$

If ϵ becomes less than ϵ_l, we also set ϵ to 0 and continue with the next iteration.

Following this approach, we plot the resulting number of scanned columns, compared to a vanilla SSP approach in Figure 4. The raw data for various test cases is presented in Table 1. As can easily be seen, the worst case for our approach (disjoint 2D) requires less rows to be evaluated than the best case (random) for the original SSP algorithm. The data in Table 1 also shows, that rate the comparisons grow in is muss less for our implementation with at most $n^{1.360}$ than for the original SSP with at least $n^{1.494}$.

4 SOLVING LARGE SCALE PROBLEMS

So far, we discussed algorithmic optimizations but in order to efficiently solve large, dense linear assignment problems we have to look into the actual implementation as well. Initially, we can assume that the cost matrix fits into the main memory of a single machine that we will be using. In this case, the matrix can be assumed as given and each access requires roughly the same amount of time.

n	sanity	random	geometric	disjoint	rank 1	rank 2	rank 4	rank 8
1000	405,299	23,465	32,925	268,628	500,500	391,298	284,688	209,141
1414	801,834	37,925	62,201	537,996	1,000,405	789,793	558,041	390,779
2000	1,606,422	64,818	93,318	1,089,861	2,001,000	1,560,268	1,124,012	761,315
2828	3,237,781	104,142	169,431	2,153,271	4,000,206	3,146,486	2,205,867	1,445,345
4000	6,488,536	177,543	226,641	4,289,967	8,002,000	6,261,715	4,407,709	2,730,315
5656	12,973,835	322,430	487,586	8,521,279	15,997,996	12,508,083	8,652,154	5,128,737
8000	25,965,151	539,169	880,954	17,193,386	32,004,000	25,083,512	17,353,291	9,787,226
11313	51,850,375	873,044	1,424,202	34,265,945	63,997,641	50,100,160	33,909,688	18,738,841
16000	103,580,440	1,477,456	2,114,829	68,509,768	128,008,000	100,042,613	67,468,112	35,431,447
22627	207,775,917	2,430,741	3,885,168	136,929,178	256,001,878	199,663,773	133,567,508	66,828,331
32000	415,100,633	4,226,292	6,428,107	274,579,141	512,016,000	400,280,323	265,590,769	126,553,143
$\approx xn^y$	$0.405n^{2.000}$	$0.631n^{1.515}$	$1.195n^{1.494}$	$0.260n^{2.003}$	$0.500n^{2.000}$	$0.386n^{2.001}$	$0.327n^{1.978}$	$0.648n^{1.840}$
nrmse	0.12%	2.19%	4.40%	0.14%	0.00%	0.17%	0.21%	0.21%

n	sanity	random	geometric	disjoint	rank 1	rank 2	rank 4	rank 8
1000	1,000	12,824	19,998	28,229	3,503	28,969	25,239	22,989
1414	1,414	18,119	37,525	50,054	4,462	47,748	39,428	34,751
2000	2,000	29,606	54,741	76,140	7,435	72,742	61,461	57,539
2828	2,828	39,757	90,586	112,897	9,863	121,158	88,949	80,895
4000	4,000	65,389	125,907	190,391	16,519	185,804	142,926	120,681
5656	5,656	106,533	195,722	313,917	22,807	291,242	213,662	183,655
8000	8,000	153,469	301,640	496,800	35,822	456,406	328,156	276,196
11313	11,313	236,650	476,453	791,253	44,641	716,047	487,708	427,577
16000	16,000	368,018	754,908	1,324,769	74,643	1,113,684	736,156	658,319
22627	22,627	534,621	1,220,753	2,077,130	116,496	1,849,133	1,173,760	991,459
32000	32,000	824,494	1,966,770	3,343,990	152,379	2,886,527	1,774,836	1,548,766
45254	45,254	1,297,589	3,206,336	5,366,221	220,592	4,664,148	2,596,731	2,270,752
64000	64,000	1,966,346	4,673,699	8,810,688	298,101	7,333,891	4,225,643	3,405,828
90509	90,509	3,117,990	7,144,123	14,116,517	512,031	11,919,558	6,443,633	5,229,444
128000	128,000	5,013,111	11,537,205	22,315,125	626,420	18,746,959	9,897,628	8,154,602
$\approx xn^y$	$1.000n^{1.000}$	$1.147n^{1.300}$	$3.200n^{1.283}$	$2.541n^{1.360}$	$3.698n^{1.028}$	$2.593n^{1.343}$	$4.304n^{1.246}$	$4.572n^{1.223}$
nrmse	0.00%	2.83%	3.61%	1.40%	12.03%	1.22%	1.82%	2.16%

Table 1. Rows evaluated with and without epsilon pricing while solving the LAP (average of 5 random matrices using the CPU based implementation). The data is fitted against the function nN^y with the normalized root mean square error given below. Note that rank 1 is the worst case matrix for SSP.

Since today's computers typically consist of multiple cores (Uniform Memory Architecture: UMA) or even multiple CPUs with multiple cores (Non-Uniform Memory Architecture: NUMA), the most obvious way to increase performance is to parallelize as many parts of the algorithm as possible. As observed by Dell'Amico and Toth [2000], there are two options for splitting the computationally most expensive part of the algorithm, i.e. the Dijkstra search, between multiple cores. They found it to be more efficient to scan multiple columns at once instead of scanning a single column with multiple cores. However, in our case, we rarely need to scan more than a couple of columns per search and therefore have to resort to the latter, potentially less efficient approach of scanning a single column in parallel. Fortunately, the efficiency of this approach gets better the larger the problem becomes as we will see in Section 5. In our implementation, we chose to fix the amount of columns each thread is responsible for during the Dijkstra search as this allows us to partition the cost matrix between the cores as well. This reduces synchronization overhead while increasing the total available memory bandwidth, especially for NUMA systems with multiple partitions. Another observation that was made in the parallel implementation is that critical sections guarded by locks are very expensive. If we store per thread data instead, use barriers for synchronization and the master thread to collect and distribute all results, the performance is vastly increased. Again this affects NUMA systems more than UMA systems and is much more visible when solving smaller problems. When implementing our algorithm on the GPU, we also noticed that a single GPU is sufficient to solve problems with up to 256000×256000 arcs, as long as the cost matrix can be calculate on the GPU. Larger problems benefit from using multiple GPUs.

If the cost matrix does not fit into memory anymore, we have to be able to calculate the costs on demand. Looking at our algorithm, we can see that we only ever need to access one row of the matrix at a time. In order to not be bound by the computation time of calculating large parts of the matrix again and again, we employ a cache holding as many rows as we can fit into the memory that has been reserved for the solver. Since individual rows will be accessed more often over time than others, our cache replacement scheme should take this into account. We implemented two different caching schemes that can be used for different ratios between cache and matrix size: Segmented least recently used (SLRU) [Karedla et al. 1994] and least frequently used (LFU). In practice, we found that LFU performs better for caches that can fit at least 1/4th of the entire cost matrix, while SLRU performs better for smaller caches. The most extreme case, i.e. storing only a single row of the cost matrix at a time, also works but the computational requirement heavily depend on calculating the cost matrix in that case. If the parallel algorithm is used, the cost matrix is calculated in parallel as well. Besides using the CPU to calculate the cost matrix on demand, it can also be loaded from disk, be transferred over the network, or calculated by an accelerator card such as a GPU. For efficiently handling all these cases, a cost function can be supplied by the user either as a lambda function that calculates a single value or as a lambda function that calculates a (partial) row. As an alternative, the user can also directly specify a cost matrix to be used. In case of the GPU solver, the user only has to implement a device kernel that calculates a part of a given row on the GPU and stores the result in device memory. Using this approach, we can solve dense problems with more than $1,000,000 \times 1,000,000$ arcs as we will see in Section 5.

5 RESULTS

As already mentioned in Section 3, we use the following classes of cost matricies for performance evaluation:

random A cost matrix with random double precision floating point values in range $[0.0 \ldots 1.0]$ that follow a uniform distribution

geometric Cost based on the euclidean squared distance between two sets of random point in $\mathbb{R}^2$. The points are uniformly distributed in $[(0.0, 0.0) \dots (1.0, 1.0)]$ and the cost values are stored in double precision floating point.

disjoint Cost based on the euclidean squared distance between two sets of points in $\mathbb{R}^2$. The points are split into four groups of equal size and evenly distributed in $[(0.0, 0.0) \dots (1.0, 1.0)]$, $[(0.0, 1.0) \dots (1.0, 2.0)]$, $[(1.0, 0.0) \dots (2.0, 1.0)]$ and $[(1.0, 1.0) \dots (2.0, 2.0)]$ with the first and fourth groups forming the source points while the second and third group form the targets. Again, the cost values are stored in double precision floating point.

sanity Matrix $C = 0.1\,(J - I) + j \otimes a + b \otimes j$ (identity matrix I, unity matrix J and unity vector j) produced by using two random vectors a and b with $0 \geq a_i \geq 1 \wedge 0 \geq b_i \geq 1$. The solution $X = I$ for this problem is known by construction can therefore used to verify that the algorithm produces the correct solution.

low rank The cost matrix $C = \sum_{i=1}^{n} a_i \otimes a_i$ is based on the sum of outer vector products which produces a symmetric matrix of rank n for n linear independent vectors a_i with $a_{ij} \geq 0$.

images Cost based on the image metric used by Rubner et al. [2000] which is a Euclidean squared distance in $\mathbb{R}^5$. However, instead of operating on a very limited number of quantized bins (18.5 on average), we use the full precision color and position information in single precision floating point.

All performance measurements have been generated using the following hard and software:

CPU performance: Dual slot (NUMA; 2× Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz) 16 core (32 virtual cores with hyperthreading) machine with 256GB (128GB per slot) of main memory running Ubuntu 16.04.4 LTS

GPU performance GTX980: Dual slot (NUMA; 2× Intel(R) Xeon(R) CPU E5-2687W v3 @ 3.10GHz) 20 core (40 virtual cores with hyperthreading) machine with 128GB (64GB per slot) of main memory running Ubuntu 16.04.4 LTS with 8× GeForce GTX 980, each with 4GB memory

GPU performance T4: Dual slot (NUMA; 2× Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz) 20 core (40 virtual cores with hyperthreading) machine with 256GB (128GB per slot) of main memory running Ubuntu 18.04.4 LTS with 8× Tesla T4, each with 15GB memory

CPU compiler: GNU C++ compiler (g++ 7.4.0)

GPU compiler: NVCC compiler from CUDA 10.2 toolkit (V10.2.89)

Random numbers were generated using the Mersenne twister and uniform distribution implementations of the STL. For the cached performance measurements on the CPU, the memory consumption was limited to use 240GB for the cached rows (a total of roughly 242GB). For the GPU tests, the cache size was limited to 3GB of GPU memory (a total of 24GB in case of using all 8 GPUs). The total memory consumption on each GPU was roughly 3.5GB for the largest test cases. It was also verified that all solutions, with/without ϵ and with/without parallel computations on both the CPU and GPU, produced the same solution.

The initial set of performance measurements compares the run time for the different random based test cases. As seen in Figure 5, the run time heavily depends on how hard the LAP is to solve. The rank 1 matrix starts out at roughly 1/20th of the performance of the purely random cost matrix at 1000×1000 arcs. However, when looking at a problem size of 32000×32000 arcs, the rank 1 matrix required roughly 180 times longer, an additional factor of about 9.

When switching to our ϵ-Pricing algorithm, this behavior changes and the run time becomes less dependent on how hard the LAP is to solve. It stays roughly within a factor of about 30 for a 320000×32000 arc problem between easiest and hardest problem tested (see right side of Figure 5). In addition, the run time of the disjoint cost matrix with ϵ-Pricing is less than the same as the run

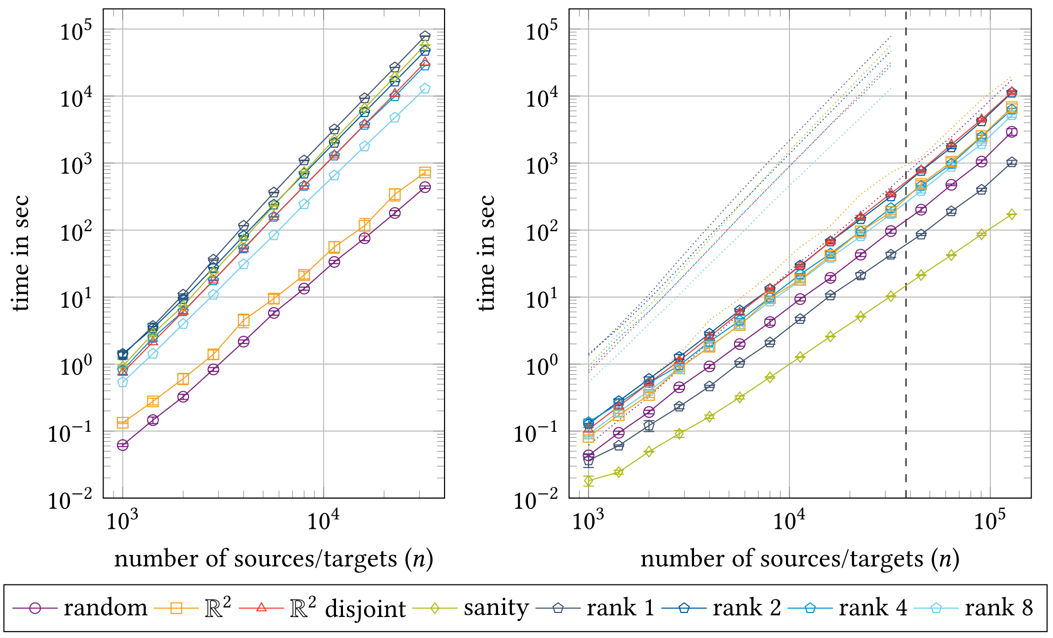

Fig. 5. Run time of solving random, geometric (regular and disjoint) and sanity cost matrices of various sizes with double precision floating point arithmetic. Regular SSP algorithm (left) compared against our ϵ-Pricing approach (right). As can be seen on the right, the time complexity of hard to solve LAPs is reduced towards the time complexity of easy to solve ones.

time of the random cost matrix without ϵ-Pricing. Note that Figure 5 does not contain the setup time of the matrix but does contain the initial estimate of v and ϵ_0.

When further analyzing the performance of our implementation up to this point, we can see that it is mainly limited by the Dijkstra path search. While the overhead for managing the workload of individual threads and inter-thread communication is quite significant for smaller problems, a problem size of 2000×2000 can already see some benefit of running multiple threads. For larger problem sizes, we see a speedup of up to 12.6 with our 32 core NUMA machine which translates into an efficiency of almost 40% (see Figure 6). The reason for this limit is both the amount of work that needs to be carried out sequentially as well as synchronization overhead and memory bandwidth limits. Note that the memory bandwidth increase reported by the STREAM test between a single thread (12, 792.7 MB/s) and 32 threads (61, 045.2 MB/s) is only a factor of 4.77. In addition, these speedups can only be realized when preventing threads from switching between cores and thus allowing local storage of their part of the cost matrix. In practice, setting "`OMP_PROC_BIND`" to "`close`" and "`OMP_PLACES`" to "`cores`" produced the best performance. When comparing these results to our GPU based implementation in Figure 7 and Figure 8, we can see that it is about one order of magnitude faster for our $1,024,000 \times 1,024,000$ test case. However, we can also see that the GPUs are still not fully utilized since the slope is still lower compared to the CPU version in Figure 6. Because of this, the GPU version adds very little overhead for re-evaluating the cost matrix when compared against the CPU version.

Up to this point, all cost functions were based on random values to some extend. However, for most problems, we cannot assume an underlying uniform random distribution. This becomes quite evident when looking at the image metric of Rubner et al. [2000] applied to a full cross-correlation

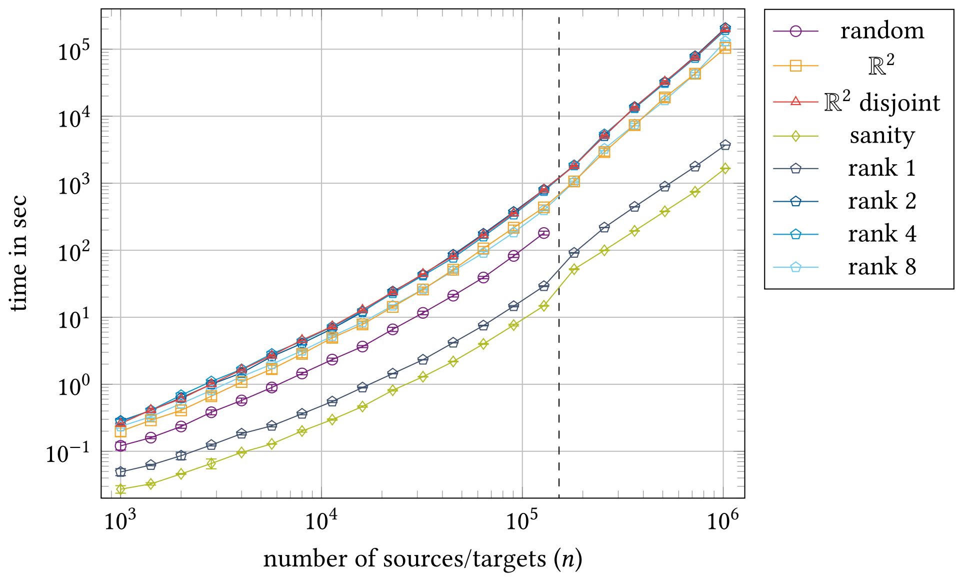

Fig. 6. Comparison of run times for different problem sizes using our cache-based (220GB) multi-threaded solver on our 32 core NUMA test machine. Note that the cost matrix of problems larger than 128000×128000 no longer fits into memory (dashed line).

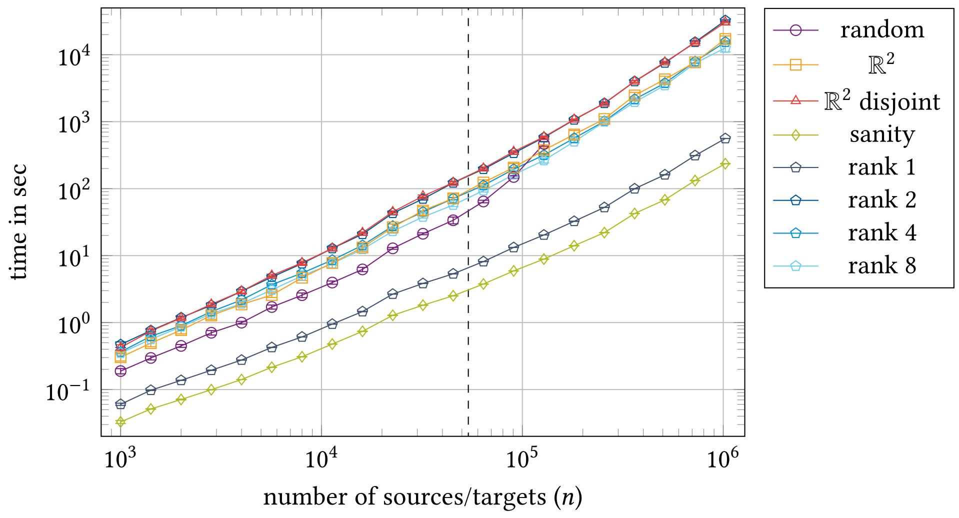

Fig. 7. Comparison of run times for different problem sizes using our GPU-based (8×3GB = 24GB) solver on our $8 \times$ GTX 980 test machine. The random cost matrix was calculated with the CPU (up to 122GB) and is transferred on demand, all other cost matrices are calculated on the GPU. Note that the cost matrix of problems larger than 45254×45254 no longer fits into GPU memory (dashed line).

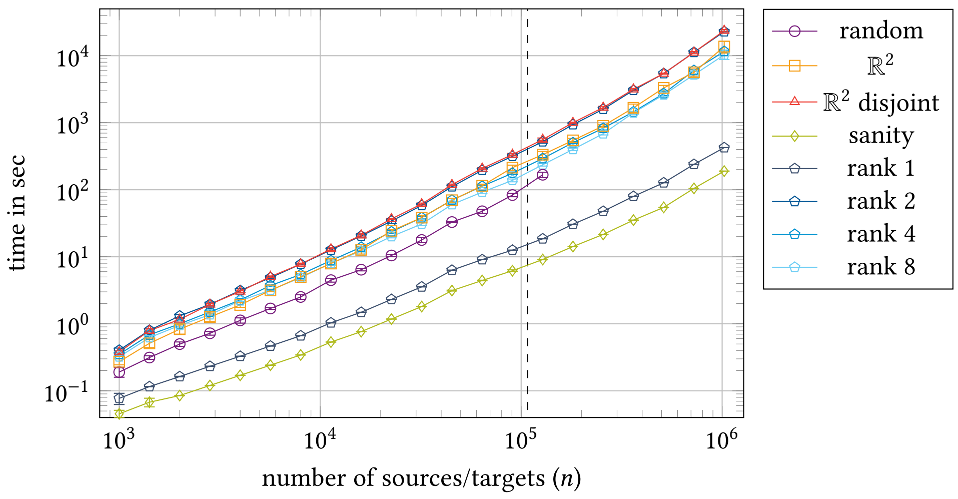

Fig. 8. Comparison of run times for different problem sizes using our GPU-based (8 × 14GB = 112GB) solver on our 8 × Tesla T4 test machine. The random cost matrix was calculated with the CPU (up to 122GB) and is transferred on demand, all other cost matrices are calculated on the GPU. Note that the cost matrix of problems larger than 90509 × 90509 no longer fits into GPU memory (dashed line).

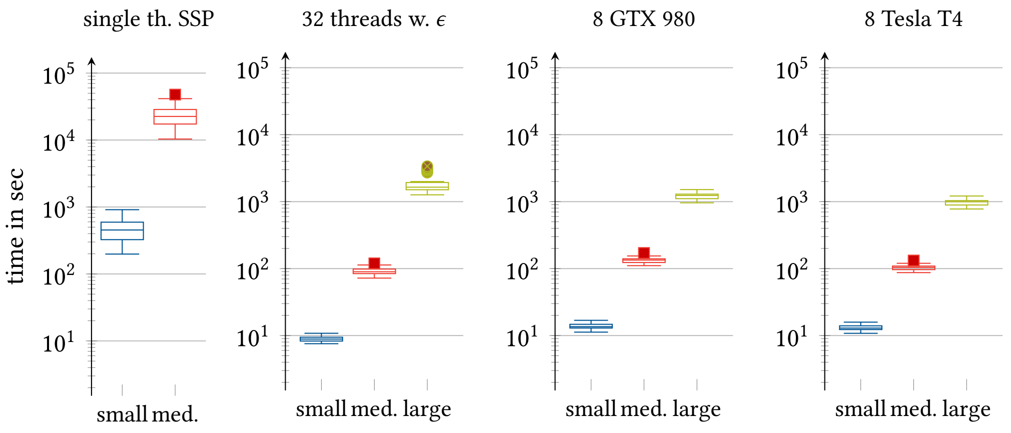

Fig. 9. From left to right, single threaded vanilla SSP, 32 threads with epsilon and 8 GPUs (GTX980 and Tesla T4) with epsilon. Image sizes approximately 11,000, 44,000 and 176,000 pixel, every image compared against every other in the same bin.

of 10 images (see Figure 10). For testing purposes, the images were scaled to three different sizes and each image was tested against all other images of the same size. The final cost of the LAP solution was used as a distance metric between the images and is visualized using multi-dimensional scaling in Figure 11. It turns out that the run time of the straightforward successive shortest path algorithm is very close to our random based worst-case test and we can already see a speedup of more than one order of magnitude for a problem size of 11000×11000 (see Figure 9 when using our ϵ-Pricing). When using our parallel implementation, this speedup is further increased to almost two orders of magnitude for this small problem. Also note that the spread in run time is reduced from a factor of 4

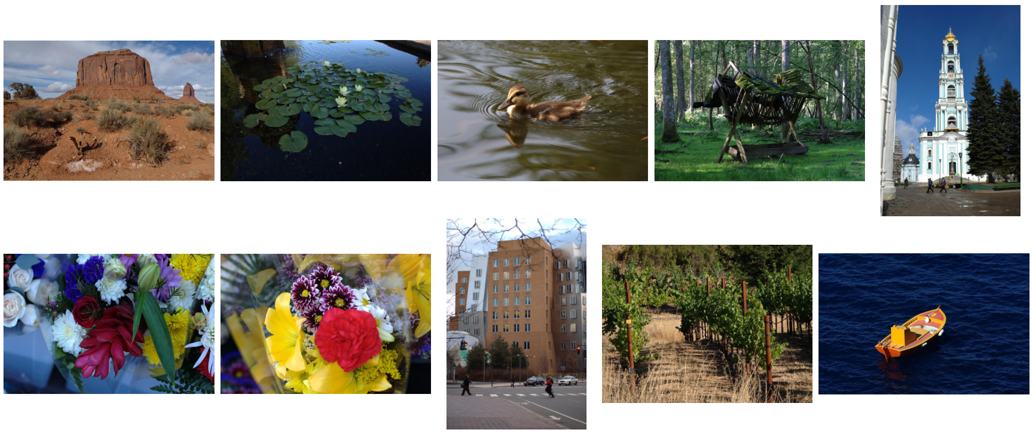

Fig. 10. Test images from MIT-Adobe FiveK Dataset [Bychkovsky et al. 2011].

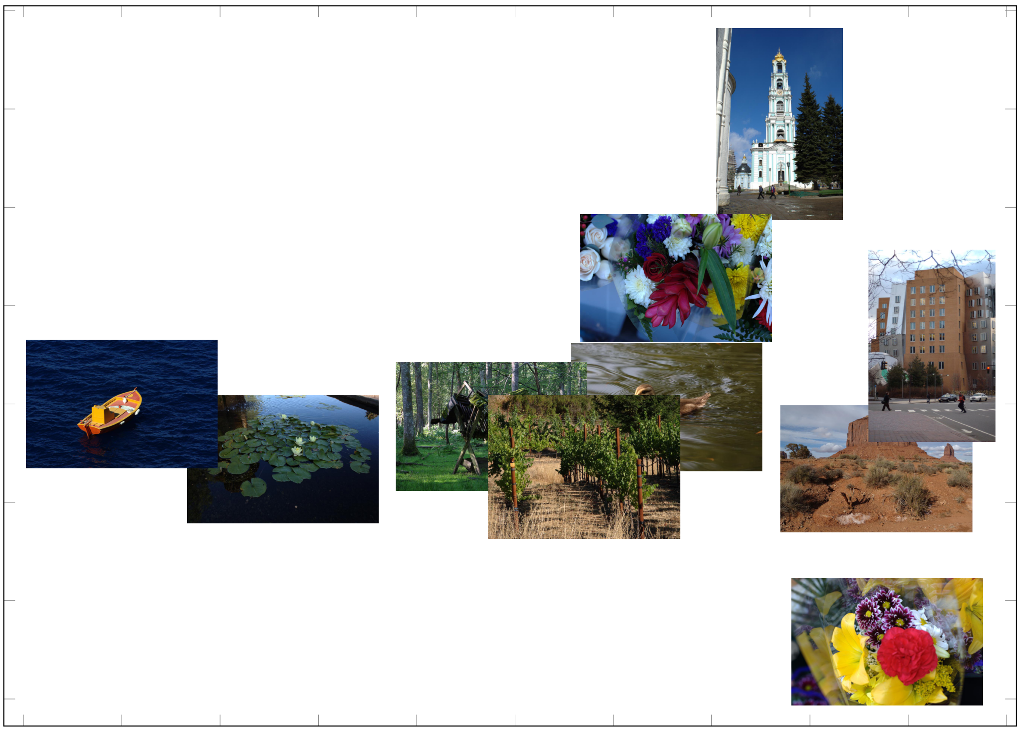

Fig. 11. Visualization of the linear assignment costs between image pairs using multidimensional scaling in 2D.

down to a factor of 2.2. For the largest problems with a size of about 176000 × 176000, we can see an estimated performance gain of more than three orders of magnitude when comparing the parallel implementation with ϵ-Pricing against the original SSP algorithm. Note that the original algorithm was not actually run for these problem sizes as the estimated run time would easily exceed 20 days for most of the 45 image comparisons whereas our parallel algorithm required about 20 minutes on average.

When comparing our algorithm with a parallel implementation of the forward auction algorithm of Bertsekas [1981], we need to add a final stage that scales ϵ to 0 if we use non-integer cost matrices. For better comparison, the implementation uses the same code base as our algorithm. As can be seen in Table 2, the run time performance becomes very unpredictable. In general, our approach scales well for all cost matrices while the auction algorithm, including our optimized version, only scales well for some cost matrices. Further investigation for our optimized auction algorithm is therefore required and is part of ongoing research. In additions, the current version of the auction algorithm implementation does not support matrices that do not fit into main memory.

When comparing the run time of our CPU based algorithm with the implementation of Date and Nagi [2016] (see Table 3), we can clearly see that the computational and/or bandwidth requirement for our approach is several orders of magnitude smaller. In fact, when comparing the corresponding GPU run times, we can see the GPUs are vastly underutilized when solving a 40000×40000 matrix. The run time is in this case limited by the number of kernel calls that need to be issued and by the synchronization between GPUs or between GPU and CPU. Overall, the run time performance of our CPU based implementation is better when compared to Date and Nagi [2016] even when their approach is using 16 GPUs. Additionally, they do not report any numbers for matrices larger than $40,000 \times 40,000$ or any other cost matrix than random integer values.

6 CONCLUSION

In this paper, we presented a new algorithm and its implementation for CPUs and GPUs in Uniform and Non-Uniform Memory Architectures, that is able to solve LAPs that are, to our knowledge, several orders of magnitude larger than previously solved problems. We evaluated our approach on different random based cost matricies, presenting both easy as well as hard to solve problems, and on application based cost matricies. We showed that we can efficiently solve LAPs with more than 1 trillion (1,024,000 × 1,024,000) arcs, even though the matrix no longer fits into the available memory. Instead we rely on caching and calculating or reading cost values on demand. In cases where values can not be calculated, they need to be read from disk limiting the run time performance to at the time it takes to read each line that is being scanned. We also presented a software system that can be used to readily solve an LAP based on arbitrary user defined cost functions that can simply be passed as lambda functions, CUDA device lambda functions or CUDA kernels to the solver.

ACKNOWLEDGMENTS

Part of this work was funded by Deutsche Forschungsgemeinschaft (DFG) under project number 407714161.

REFERENCES

Richard S Barr, Fred Glover, and Darwin Klingman. 1977. The alternating basis algorithm for assignment problems. *Mathematical Programming* 13, 1 (1977), 1–13.

Dimitri P. Bertsekas. 1981. A new algorithm for the assignment problem. *Mathematical Programming* 21, 1 (01 Dec 1981), 152–171. https://doi.org/10.1007/BF01584237

Dimitri P Bertsekas. 1988. The auction algorithm: A distributed relaxation method for the assignment problem. *Annals of operations research* 14, 1 (1988), 105–123.

Vladimir Bychkovsky, Sylvain Paris, Eric Chan, and Frédo Durand. 2011. Learning Photographic Global Tonal Adjustment with a Database of Input / Output Image Pairs. In *The Twenty-Fourth IEEE Conference on Computer Vision and Pattern Recognition.*

Ketan Date and Rakesh Nagi. 2016. GPU-accelerated Hungarian Algorithms for the Linear Assignment Problem. *Parallel Comput.* 57, C (Sept. 2016), 52–72. https://doi.org/10.1016/j.parco.2016.05.012

N	sanity			random			geometric			disjoint		
	auction	opt. auc.	our	auction	opt. auc.	our	auction	opt. auc.	our	auction	opt. auc.	our
1000	0.066	0.092	**0.027**	**0.115**	0.132	0.121	0.278	0.269	**0.176**	0.287	0.349	**0.266**
1414	0.098	0.131	**0.037**	0.164	0.185	**0.160**	0.444	0.498	**0.250**	**0.361**	0.443	0.405
2000	0.122	0.197	**0.045**	0.284	0.264	**0.234**	0.674	0.645	**0.424**	0.767	0.618	**0.600**
2828	0.205	0.307	**0.064**	0.402	**0.363**	0.382	1.247	0.945	**0.695**	1.353	1.205	**1.075**
4000	0.497	0.510	**0.098**	0.781	**0.524**	0.574	1.966	2.119	**1.064**	2.499	1.628	**1.586**
5656	1.203	0.854	**0.136**	1.446	**0.790**	0.896	5.121	2.698	**1.529**	5.209	**2.444**	2.628
8000	2.652	1.573	**0.202**	2.347	**1.271**	1.449	9.025	3.992	**2.531**	10.594	**4.351**	4.441
11313	6.802	2.924	**0.307**	4.760	**2.018**	2.335	16.758	5.675	**4.407**	18.697	**6.775**	7.206
16000	18.761	5.861	**0.568**	8.257	**2.882**	3.655	45.766	9.525	**6.867**	44.685	12.675	**12.242**
22627	61.766	11.196	**1.048**	16.923	**5.083**	6.595	89.952	19.684	**13.173**	88.436	24.047	**22.168**
32000	136.984	24.246	**1.863**	33.061	**9.430**	11.620	198.708	33.298	**21.975**	231.495	**38.680**	41.407
45254	433.944	50.363	**3.186**	78.577	**14.663**	21.005	432.013	62.389	**49.358**	505.249	80.658	**80.604**
64000	908.198	107.144	**5.799**	144.421	**27.191**	39.170	702.842	107.538	**85.847**	2,454.277	**126.528**	162.436
90509	—	191.851	**10.968**	—	**49.313**	82.266	—	**208.192**	214.031	—	**317.399**	339.728
128000	—	408.482	**20.309**	—	**84.491**	180.742	—	474.071	**412.446**	—	**628.011**	764.252

N	rank 1			rank 2			rank 4			rank 8		
	auction	opt. auc.	our	auction	opt. auc.	our	auction	opt. auc.	our	auction	opt. auc.	our
1000	2.625	1.657	**0.046**	0.316	0.421	**0.280**	**0.136**	0.207	0.214	**0.120**	0.154	0.214
1414	2.996	8.371	**0.063**	0.849	0.432	**0.402**	**0.198**	0.251	0.341	**0.165**	0.221	0.304
2000	3.650	37.986	**0.076**	1.066	0.820	**0.645**	0.372	**0.314**	0.491	**0.287**	0.327	0.449
2828	29.417	36.469	**0.119**	1.861	1.389	**1.047**	**0.469**	0.609	0.782	**0.399**	0.447	0.695
4000	79.878	43.608	**0.176**	3.159	3.368	**1.653**	1.075	**0.880**	1.213	0.594	**0.544**	1.105
5656	171.049	86.683	**0.230**	6.618	3.584	**2.639**	1.873	**1.191**	1.938	1.271	**0.977**	1.694
8000	4,615.522	672.360	**0.341**	21.244	6.933	**4.305**	3.794	**2.143**	2.996	2.196	**1.439**	2.715
11313	—	2,721.098	**0.532**	52.569	10.088	**7.427**	7.054	**3.295**	4.895	4.517	**2.264**	4.347
16000	—	18,513.863	**0.840**	63.288	19.440	**11.892**	13.752	**5.080**	8.032	7.852	**3.362**	7.277
22627	—	—	**1.367**	212.188	36.716	**21.372**	26.902	**8.622**	13.858	17.077	**5.702**	13.144
32000	—	—	**2.208**	433.887	68.118	**40.384**	51.492	**16.020**	24.853	29.346	**10.198**	20.656
45254	—	—	**3.910**	3,762.414	108.861	**78.483**	116.393	**37.477**	46.365	64.923	**15.851**	37.797
64000	—	—	**7.227**	4,709.437	368.165	**153.359**	242.934	**51.665**	86.176	124.508	**34.009**	69.341
90509	—	—	**14.253**	—	537.779	**318.422**	—	**126.035**	181.120	—	**52.645**	138.399
128000	—	—	**27.957**	—	962.755	**727.769**	—	**224.847**	398.337	—	**84.650**	303.174

Table 2. Comparison of the average run time performance in seconds of our CPU based algorithm with a plain and an optimized version of the parallel forward auction algorithm with ϵ-scaling using the same code base. Note that the run time performance of the auction algorithm is very unpredictable due to price haggling when ϵ reaches 0.

N	Date and Nagi			our				
	OMP-8	GPU	8xGPU	OMP-8	OMP-16	OMP-32	GTX 980	Tesla T4
10000	33.90	4.93	4.97	0.87	**0.75**	1.52	2.42	**2.34**
14142	(77.11)[1]	6.57	6.76	1.66	**1.42**	2.28	3.83	**3.64**
20000	137.98	8.26	8.48	3.36	**2.83**	3.99	5.89	**5.61**
28284	—	—	11.43	6.99	**5.78**	6.85	9.65	**8.67**
40000	—	—	14.91[2]	15.13	13.15	**11.50**	19.94[3]	**13.87**
56568	—	—	—	35.05	29.35	**21.57**	34.32[4]	**25.85**[3]
80000	—	—	—	81.39	59.64	**42.90**	61.37[5]	**48.29**[4]
113137	—	—	—	173.79	129.88	**88.84**	134.01[5]	**75.94**[5]
160000	—	—	—	574.61	339.40	**193.64**	245.54[5]	**173.35**[5]

Table 3. Comparison of the run time performance in seconds of our CPU and GPU based algorithm with the Hungarian algorithm implementation of Date and Nagi [2016] (fastest CPU and GPU version in bold). [1]: time for $N = 15000$. [2]: time for 16 GPU version. [3]: time for 2 GPUs. [4] time for 4 GPUs. [5]: time for 8 GPUs.

Mauro Dell'Amico and Paolo Toth. 2000. Algorithms and codes for dense assignment problems: the state of the art. *Discrete Applied Mathematics* 100, 1 (2000), 17 – 48. https://doi.org/10.1016/S0166-218X(99)00172-9

I. S. Duff. 1981. On Algorithms for Obtaining a Maximum Transversal. *ACM Trans. Math. Softw.* 7, 3 (Sept. 1981), 315–330. https://doi.org/10.1145/355958.355963

Iain S Duff and Jacko Koster. 1999. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM J. Matrix Anal. Appl.* 20, 4 (1999), 889–901.

Iain S Duff and Jacko Koster. 2001. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM J. Matrix Anal. Appl.* 22, 4 (2001), 973–996.

F. Dufossé, K. Kaya, and B. Uçar. 2014. Bipartite Matching Heuristics with Quality Guarantees on Shared Memory Parallel Computers. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium.* 540–549. https://doi.org/10.1109/IPDPS.2014.63

Jack Edmonds. 1965. Paths, trees, and flowers. *Canadian Journal of mathematics* 17, 3 (1965), 449–467.

Jack Edmonds and Richard M Karp. 1972. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM (JACM)* 19, 2 (1972), 248–264.

M. Engquist. 1980. *A Successive Shortest Path Algorithm for the Assignment Problem.* Defense Technical Information Center. https://books.google.de/books?id=IAQPOAAACAAJ

Michael L Fredman and Robert Endre Tarjan. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)* 34, 3 (1987), 596–615.

Andrew V Goldberg, Serge A Plotkin, David B Shmoys, and Éva Tardos. 1992. Using interior-point methods for fast parallel algorithms for bipartite matching and related problems. *SIAM J. Comput.* 21, 1 (1992), 140–150.

David S. Johnson and Catherine C. McGeoch (Eds.). 1993. *Network Flows and Matching: First DIMACS Implementation Challenge.* American Mathematical Society, Boston, MA, USA. http://dimacs.rutgers.edu/archive/Volumes/Vol12.html

R. Jonker and A. Volgenant. 1987. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing* 38, 4 (01 Dec 1987), 325–340. https://doi.org/10.1007/BF02278710

Ramakrishna Karedla, J. Spencer Love, and Bradley G. Wherry. 1994. Caching Strategies to Improve Disk System Performance. *Computer* 27, 3 (March 1994), 38–46. https://doi.org/10.1109/2.268884

Harold W Kuhn. 1955. The Hungarian method for the assignment problem. *Naval Research Logistics (NRL)* 2, 1-2 (1955), 83–97.

James Munkres. 1957. Algorithms for the Assignment and Transportation Problems. *J. Soc. Indust. Appl. Math.* 5, 1 (1957), 32–38. http://www.jstor.org/stable/2098689

J.B. Orlin and Y. Lee. 1993. *QuickMatch–a Very Fast Algorithm for the Assignment Problem.* Alfred P. Sloan School of Management, Massachusetts Institute of Technology. https://books.google.de/books?id=Ci-qGwAACAAJ

James B Orlin. 1997. A polynomial time primal network simplex algorithm for minimum cost flows. *Mathematical Programming* 78, 2 (1997), 109–129.

Tania Pouli and Erik Reinhard. 2011. Progressive color transfer for images of arbitrary dynamic range. *Computers & Graphics* 35, 1 (2011), 67 – 80. https://doi.org/10.1016/j.cag.2010.11.003 Extended Papers from Non-Photorealistic Animation and Rendering (NPAR) 2010.

Yossi Rubner, Carlo Tomasi, and Leonidas J. Guibas. 2000. The Earth Mover's Distance As a Metric for Image Retrieval. *Int. J. Comput. Vision* 40, 2 (Nov. 2000), 99–121. https://doi.org/10.1023/A:1026543900054

Madan Sathe, Olaf Schenk, and Helmar Burkhart. 2012. An auction-based weighted matching implementation on massively parallel architectures. *Parallel Comput.* 38, 12 (2012), 595–614. https://doi.org/10.1016/j.parco.2012.09.001

Bernhard Schmitzer. 2015. A sparse algorithm for dense optimal transport. In *International Conference on Scale Space and Variational Methods in Computer Vision*. Springer, 629–641.

Bernhard Schmitzer. 2016. A sparse multiscale algorithm for dense optimal transport. *Journal of Mathematical Imaging and Vision* 56, 2 (2016), 238–259.

Richard Sinkhorn and Paul Knopp. 1967. Concerning nonnegative matrices and doubly stochastic matrices. *Pacific J. Math.* 21, 2 (1967), 343–348.

N Tomizawa. 1971. On some techniques useful for solution of transportation network problems. *Networks* 1, 2 (1971), 173–194.

Cristina Nader Vasconcelos and Bodo Rosenhahn. 2009. Bipartite graph matching computation on GPU. In *International Workshop on Energy Minimization Methods in Computer Vision and Pattern Recognition*. Springer, 42–55.

John Von Neumann. 1953. A certain zero-sum two-person game equivalent to the optimal assignment problem. *Contributions to the Theory of Games* 2, 0 (1953), 5–12.

Li Zhang, Yuan Li, and R. Nevatia. 2008. Global data association for multi-object tracking using network flows. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*. 1–8. https://doi.org/10.1109/CVPR.2008.4587584

Received June 2018; revised July 2019; revised June 2020; revised November 2020; accepted December 2020