Engineering fast almost optimal algorithms for bipartite graph matching: Extended version

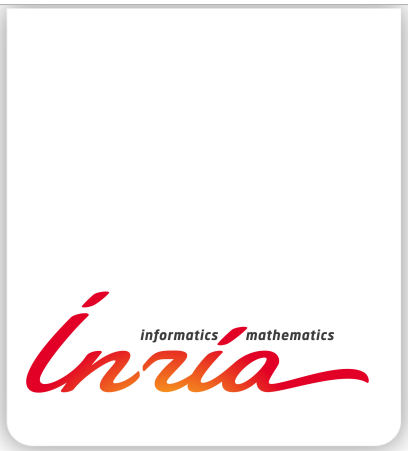
Ioannis Panagiotas, Bora Uçar

Engineering fast almost optimal algorithms for bipartite graph matching: Extended version

Ioannis Panagiotas, Bora Uçar

RESEARCH REPORT
N° 9321
January 2020
Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--9321--FR+ENG



Engineering fast almost optimal algorithms for bipartite graph matching: Extended version

Ioannis Panagiotas*, Bora Uçar†

Project-Team ROMA

Research Report n° 9321 — January 2020 — 20 pages

Abstract: We consider the maximum cardinality matching problem in bipartite graphs. There are a number of exact, deterministic algorithms for this purpose, whose complexities are high in practice. There are randomized approaches for special classes of bipartite graphs. Random 2-out bipartite graphs, where each vertex chooses two neighbors at random from the other side, form one class for which there is an $O(m + n \log n)$-time Monte Carlo algorithm. Regular bipartite graphs, where all vertices have the same degree, form another class for which there is an expected $O(m + n \log n)$-time Las Vegas algorithm. We investigate these two randomized algorithms and turn them into practical heuristics. We compare the performance of the resulting heuristics and show that they obtain near optimal results in practice and are comparable with the standard approaches.

Key-words: bipartite graphs, matching, randomized algorithm

* ENS Lyon, 46, allée d'Italie, ENS Lyon, Lyon F-69364, France.
† CNRS and LIP (UMR5668 CNRS-ENS Lyon-INRIA-UCBL), 46, allée d'Italie, ENS Lyon, Lyon F-69364, France.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Algorithmes rapides quasi-optimaux pour trouver des couplages dans de graphes bipartis: Version étendue

Résumé : Nous considérons le problème de couplage maximale dans les graphes bipartis. Il existe un certain nombre d'algorithmes exacts et déterministes à cet effet, dont les complexités sont trop élevées dans la pratique. Il existe des approches randomisées pour des classes spéciales de graphes bipartites. Graphes bipartites aléatoires à 2 sorties, où chaque sommet choisit deux voisins au hasard de l'autre côté, forment une classe pour laquelle il existe un algorithme de type Monte-Carlo. Graphes bipartis réguliers, où tous les sommets ont le même degré, forment une autre classe pour laquelle il y a un algorithm de type Las Vegas. Ces deux algorithmes ont une complexité de $O(m + n \log n)$. Nous étudions ces deux algorithmes randomisés et les transformons en heuristiques pratiques applicable à tous les graphes bipartis. Nous comparons les performances des heuristiques obtenues et montrons qu'elles peuvent obtenir des résultats quasi-optimaux en pratique et sont comparables aux approches standard.

Mots-clés : graphe biparti, couplage, algorithme randomisé

1 Introduction

A matching in a graph is a set of edges, such that no two of them share a common vertex. We consider the *maximum cardinality problem* in bipartite graphs which asks for a matching with the maximum cardinality. There are a number of exact algorithms for this problem. The best known algorithms [12] run in $O(m\sqrt{n})$ time for a graph with n vertices and m edges. Such complexity can be prohibiting for large instances. For this reason, there is significant interest in algorithms which can find large matchings in linear or near linear time [20]. In practice, these large matchings can be used to initialize the exact algorithms for improved run time [18].

We investigate two randomized algorithms due to Karp et al. [14] and Goel et al. [10] which both run in $O(m + n \log n)$ time. The former algorithm finds, almost surely, maximum cardinality matchings on random graphs formed by allowing each vertex to select two vertices from the other side uniformly at random. The latter algorithm finds maximum cardinality matchings in regular bipartite graphs, where all vertices have equal degree. In both of these classes of graphs, the bipartite graphs have equal number of vertices in each part, and the maximum cardinality matchings cover all vertices (such matchings are called perfect). We adapt these two algorithms to design efficient heuristics for the maximum cardinality matching problem in bipartite graphs. We discuss our implementations and investigate the performance of the resulting heuristics both in terms of run time and the matching cardinality. Both heuristics run in near linear time and obtain matchings whose cardinality is more than 0.99 of the maximum.

The rest of the report is organized as follows. In Section 2, we give the necessary background. In Sections 3 and 4 we review the existing randomized algorithms and then discuss how we adapt them. Section 5 contains the experimental results, and Section 6 concludes the paper.

2 Background and notation

Let $G = (R \cup C, E)$ be a bipartite graph, where R and C represent the two disjoint set of vertices, and E is the set of edges. The bipartite graph G can be represented with a sparse matrix $\mathbf{A}_G$. The vertex $r_i \in R$ corresponds to the ith row, and the vertex $c_j \in C$ corresponds to the jth column, so that $\mathbf{A}_G(i, j) = 1$ if and only if $(r_i, c_j) \in E$. Hence, we will refer to vertices of R as *rows* and to those of C as *columns* from this point on, and use $\mathbf{A}$ to refer to $\mathbf{A}_G$, as G will be clear.

Let $\mathcal{M}$ be a matching. For $(u, v) \in \mathcal{M}$, the vertices u and v are matched, and they are each other's mate. A vertex is called free if it is not matched by $\mathcal{M}$. If there are no free vertices in R or in C, then $\mathcal{M}$ is called perfect. An augmenting path with respect to $\mathcal{M}$ is a path which starts with a free vertex and ends at another free vertex, where every second edge is in $\mathcal{M}$. A matching is maximum if and only if there are no augmenting paths [2].

A square matrix is called doubly stochastic if the sum of entries in each row and column is equal to one. An $n \times n$ matrix $\mathbf{A}$ has *support* if there is a perfect matching in the associated bipartite graph G. $\mathbf{A}$ is said to have *total support* if each edge in G is used in a perfect matching. A square matrix is fully indecomposable, if it has total support and cannot be permuted into a block diagonal matrix. Any nonnegative matrix $\mathbf{A}$ with total support can be scaled with two positive diagonal matrices $\mathbf{D_R}$ and $\mathbf{D_C}$ such that $\mathbf{A_S} = \mathbf{D_R A D_C}$ is doubly stochastic, and if $\mathbf{A}$ is fully indecomposable, then the matrices $\mathbf{D_R}$ and $\mathbf{D_C}$ are unique.

The Sinkhorn–Knopp algorithm [21] is a well-known method to to scale a given matrix. This is an iterative algorithm, where at each iteration each row is normalized to have unit length,

and then each column is normalized to have unit length. If a given matrix $\mathbf{A}$ has total support, then Sinkhorn–Knopp algorithm finds the unique scaling matrices. If $\mathbf{A}$ has support but not total support, then entries that cannot be put into a perfect matching tend to zero. The method converges with an asymptotical convergence rate depending on the second singular value of the final doubly stochastic matrix. There are other iterative, faster converging methods [1, 3, 17], whose iterations are more sophisticated than that of Sinkhorn–Knopp's.

A k-out subgraph G_k of a host graph G is defined by allowing each vertex in G to randomly select uniformly k of its neighbors and the union of all selections forms the edge set of G_k. Walkup [22] shows that in the pure random k-out setting, where the host graph is the complete bipartite graph, the resulting G_k has a perfect matching with high probability for $k \geq 2$. We do not know any general result about properties of G_2 sampled from any arbitrary host graph. Frieze and Johansson [9] investigate some other properties of G_ks on host graphs where the minimum degree of a vertex is at least $n/2$. An alternative is to use the doubly stochastic matrix $\mathbf{A_S}$ that results by scaling the matrix representation $\mathbf{A}$ of G for sampling. More specifically, the ith row selects the jth column with probability $\mathbf{A_S}(i,j)$ and vice versa. Dufossé et al. [7] show that in a G_1 generated with the probabilities corresponding to the entries of $\mathbf{A_S}$, the maximum cardinality of a matching (in expectation) is of size $0.866 \cdot n$ when $\mathbf{A}$ has total support. We give some experiments in which G_2s generated using the same probabilities have perfect matchings in large majority of the cases.

Two popular classes of randomized algorithms are *Las Vegas* and *Monte Carlo* algorithms. *Las Vegas* algorithms always return a correct answer, but their run time can depend on random choices, whereas Monte Carlo algorithms can fail with small probability, but their complexity is independent of the random choices made (see for example [19, p. 70]).

There are a number of heuristics for the cardinality matching problem [18, 20]. Among those, that by Karp and Sipser [13] is very well known. We summarize this heuristic here. It is based on performing reductions on a graph without any degree-0 vertices (they are discarded immediately when they arise); when a degree-1 or degree-2 vertex appears, the heuristic Karp–Sipser (KS) reduces the problem to a smaller one via the following rules:

- **Rule-1**: At any time, if a degree-1 vertex u with neighbor v appears, the edge (u, v) is added to the matching and both vertices are removed from the graph. This is an optimal decision in the sense that (u, v) is contained in at least one maximum matching in the current graph.
- **Rule-2**: At any time, if a degree-2 vertex u with neighbors v and w appears, u and its edges are removed from the graph, and v and w are merged to create a new vertex vw. This new vertex's neighbor set is the union of those of v and w (excluding u). A maximum cardinality matching for the reduced graph can be extended to a maximum cardinality matching in the current graph by matching u with either v or w depending on vw's mate. In practice, any outstanding Rule-1 reductions are performed before doing a degree-2 reduction.
- **Else**: When neither of the above rules is applicable, a randomly chosen edge from the graph is added to the matching. We note that such a decision might not be optimal.

A restricted variant of KS applies Rule-1 only, which can be implemented in $O(m + n)$ time. A recent study [15] proposes an implementation of the original variant with $O(m \log n)$ expected time complexity.

3 2OUTMC: Monte Carlo on 2-out graphs

Here, we first review the Monte Carlo algorithm by Karp et al. [14], which with high probability finds a perfect matching in a random 2-out bipartite graph, sampled from the complete bipartite graph. Then, we discuss how the algorithm can be turned into a very effective heuristic.

The random 2-out bipartite graph B_{2o} is constructed by selecting uniformly at random two row vertices for each column, and two column vertices for each row. These selections form the edges of B_{2o}. Given the edge of B_{2o}, Karp et al. define two multigraphs. The *Column-Graph* (CG) is the multigraph whose vertices are the rows, and whose edges are the choices of the columns. That is, there is an edge in CG for a column vertex in B_{2o}. Parallel edges occur if two columns select the same rows. The *Row-Graph* (RG) is defined similarly. The main idea to show that B_{2o} has a perfect matching is the following. In a component of CG that contains a cycle, it is possible to match all rows (vertices in CG) with one of the columns that have selected them (edges in CG). On the other hand in a tree component of CG, in any matching (pairing of edges with vertices) there will always be a free row vertex. As a consequence, when one or more trees appear in CG, the choices of the columns alone do not suffice to find a perfect matching, and those of the rows must be used. The algorithm thus keeps track of the tree components of CG and tries to identify one row vertex per tree component whose selections (edges in B_{2o}) should be taken into account. The columns selected by such a row could be used for a set of rows belonging in tree components. Thus one should go back and forth identifying trees in CG and analyzing components in RG. Karp et al.'s algorithm, which is described in Algorithm 1, formalizes this approach.

The algorithm operates on H_1, a copy of CG, and H_2, a copy of RG initially devoid of edges. In each step, it picks a tree from H_1 and marks one of its vertices x. This signifies that x can only be matched with one of its choices. Then, the edge of x is inserted in H_2. The algorithm then finds the component Q_x in H_2 containing the edge x, and selects an unchecked column y from Q_x. Column y is checked, which means that it can only be matched with a marked vertex. As y's choices are rendered useless now, the corresponding edge is removed from H_1 upon which new trees can be created.

For each tree vertex x identified in H_1, one should be able to find a vertex in the associated component Q_x, so that x can be matched in that component. Otherwise, Q_x has more edges than vertices, and any matching will leave some edges unpaired (or some marked rows free). The algorithm returns failure upon detecting this off case (Line 10).

The algorithm terminates successfully if all trees have a marked vertex (or if CG did not have any trees in the first place). If this happens, every component in H_1 will have as many edges as unmarked vertices. Likewise, each component in H_2 will have as many edges as checked vertices. It is therefore possible to orient the edges in either H_1 or H_2 such that each vertex (excluding marked rows or unchecked columns) is matched with a unique adjacent edge. This gives a perfect matching in B_{2o} which can be found by the Karp–Sipser heuristic in linear time. An example of the algorithm is shown in Figure 1a.

Algorithm 1 finds a perfect matching with probability $1-O(n^{-\alpha})$, where α is a positive constant. This result follows from the tie breaking rule at Line 12, which prefers choosing edges of CG that lie in cycles (called CORE, see Line 3). In case the tie-breaking in that line is done arbitrarily, the probability of success drops to $2/3 + O(n^{-\frac{1}{3}})$.

The authors then describe how to efficiently implement the algorithm such that it runs in $O(n \log n)$ worst case time. They identify two main tasks:

Algorithm 1 2OUTMC: Monte Carlo on 2-out graphs

```
1: H1 ← CG, H2 ← empty graph with columns as vertices
2: All vertices in H1 are unmarked, all vertices in H2 are unchecked
3: CORE ← edges in cycles of CG
4: while there exists a tree T in H1 with no marked vertex do
5:    Let x be a random vertex of T      ▶ x is a column vertex
6:    marked[x] ← true      ▶ x must be matched with one of its choices
7:    Add the edge of x in H2
8:    Let Q_x be the component in H2 containing the edge of x
9:    if Q_x has no unchecked vertices then
10:      Return Fail      ▶ Q_x has more edges than vertices (no 1-1 pairing possible)
11:   else
12:      Select an unchecked vertex y of Q_x. In case of ties, prefer one from CORE
13:      checked[y] ← true      ▶ y will be matched with a row that selected it
14:      delete y in H1      ▶ The algorithm forgets y's choices
15:   end if
16: end while
```

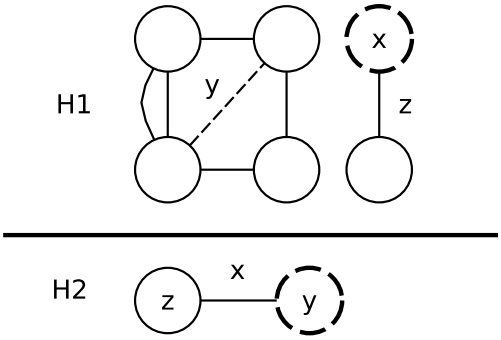
- **Task A**: Keep track of the tree components during edge deletions in H_1.
- **Task B**: Keep track of the connected components during edge insertions in H_2, and the single unchecked vertex in each component.

Task B can be efficiently done in amortized near linear time (over the course of the algorithm) by using a union-find data-structure. The only change is that the root of a component of H_2 must keep the identity of the single unchecked vertex in the component.

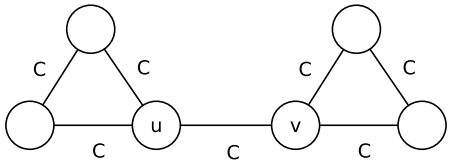
For Task A, Karp et al. propose the following. In the beginning, the edges of CG are labeled as $\mathcal{F}$, if their deletion creates a tree; $\mathcal{T}$, if they belong to a tree component; and $\mathcal{C}$ otherwise. Let c-degree of a vertex v be the number of $\mathcal{C}$ edges incident on v. During deleting the edge (u, v) from H_1, one of the following is performed depending on the label of (u, v).

- **Case 1**: (u, v) is $\mathcal{C}$: The c-degrees of u and v are decreased by one. Then, while there is a vertex with a single $\mathcal{C}$ edge; its $\mathcal{C}$ edge is relabeled as $\mathcal{F}$.
- **Case 2**: (u, v) is $\mathcal{F}$: Using a dove-tailed depth-first search, where depth-first searches from u and v are interleaved, the tree component created can be found in time proportional to its size. One then changes the labels of all edges in this tree from $\mathcal{F}$ to $\mathcal{T}$.
- **Case 3**: (u, v) is $\mathcal{T}$: Deleting (u, v) creates two trees. As in the previous case, a dove-tailed DFS is used to find these two trees in time proportional to the size of the smaller one. The new trees are to be examined by the algorithm.

The above procedure has a slight oversight and can fail to identify some of the new tree components. We demonstrate this by an example. In Figure 1b, if the edge between vertices u and v gets deleted, then the connected component is split into two triangles. The c-degree of both u and v decreases to two, and as both are greater to one, the deletion procedure stops without any action. However, both triangles are unicylic. If an edge is deleted from either triangle, then Case-1 does not recognize that the remaining edges should be relabeled as $\mathcal{T}$ not $\mathcal{F}$. We propose a fix in Lemma 1.



(a) Tree vertex x is marked and inserted into H_2. Then y is checked for being in CORE and is deleted from H_1.



(b) If another edge is deleted after (u, v), Algorithm 1 fails to realize the creation of a tree.

Figure 1 – Examples for Algorithm 1.

Lemma 1. *Let u be an endpoint of a deleted edge (u, v) with label $\mathcal{C}$. Apply the procedure of Case-1 until we arrive at a vertex p with* c-degree$[p] \neq 1$. *If* c-degree$[p] = 0$*, then u's component has become a tree.*

Proof. We claim that p and v are the same vertex. Each vertex on the path from u to p had its c-degree affected twice (from 2 to 0), except p. Hence for p to become 0, its c-degree must have been equal to 1. If $p \neq v$, then p should had its $\mathcal{C}$ edge relabeled during another deletion process. Therefore, prior to the deletion of (u, v), there was a cycle on H_1 with all vertices having c-degree equal to 2, and both their $\mathcal{C}$ edges participated in the cycle. Any outgoing edges from vertices of the cycle therefore were labeled $\mathcal{F}$, and their deletion led to a tree being formed. The component was therefore unicyclic before. □

Case 1-continuation is therefore as follows:

- Once there are no vertices with c-degree equal to 1, take the last vertex v whose c-degree was reduced. If c-degree$[v] = 0$, then relabel all edges in vs component from $\mathcal{F}$ to $\mathcal{T}$.

This addition has overall $O(n)$ cost, because each edge can change label at most twice.

3.1 Adaptations 2OUTMC

Algorithm 1 works well when the random 2-out graph is sampled from $K_{n,n}$. However, in the case of an arbitrary host graph, the underlying theory is not shown to hold, and the algorithm can potentially make erroneous decisions. We propose one mandatory addition, two optional simple heuristics to improve its performance, and highlight the importance of scaling.

3.1.1 A mandatory addition

If Algorithm 1 reaches Line 10 during execution, it quits immediately before examining all trees in H_1. As our goal is to maximize the cardinality of the returned matching, rather than returning, we instead continue executing the algorithm. More specifically, we sample a different vertex in the

tree, in case we can avoid ending up in Line 10 again as follows. In each tree T, we keep a list L_T of unmarked vertices. At Line 5 we randomly sample x from L_T and discard it from the list. Contrary to Algorithm 1, we neither mark x nor insert it in H_2 yet. Instead, we examine first whether the component in H_2 of either of the two choices of x has an unchecked column y. If y exists, we mark x, insert it to H_2 and continue by deleting y from H_1. Otherwise, we perform the same set of actions with another randomly sampled vertex from L_T. If L_T becomes empty, and no vertex was marked, we abandon T and proceed to another tree. Each such tree in the final state of H_1 decreases the cardinality of the returned matching by one, as a row is left free. If T is split into two trees, the lists of unmarked vertices for the new trees contain only those vertices still inside L_T at the moment of split. This is necessary to avoid sampling vertices more than once.

3.1.2 Heuristic 1: Delayed tree vertex selection during Line 5

The ideal case at Line 5 of Algorithm 1 is to select an x such that x's insertion as an edge to H_2 does not lead to a new tree in H_1 after the deletion of the edge corresponding to the unchecked vertex of the connected component Q_x. This is only possible if Q_x contains an unchecked column labeled as $\mathcal{C}$ in H_1. Otherwise, a new tree will be created in H_1, and the algorithm will have to process it in a future step. In the first heuristic, we greedily select an x such that, if possible, the creation of a tree in H_1 is avoided.

We replace L_T is with two lists L_T^1 and L_T^2. The lists L_T^1 contains those unmarked vertices of T whose insertion in H_2 leads to a new tree; L_T^2 contains all other L_T vertices that have not been tried yet. At first, we sample x from L_T^2 and see whether the components of x's choices in H_2 have an unchecked vertex of type $\mathcal{C}$ in H_1. If they have, x is marked and inserted to H_2. Otherwise, x is inserted in L_T^1, and we consider another random vertex of L_T^2. If L_T^2 becomes empty, we start sampling from L_T^1. As in the case of the L_T lists, if T is split into two parts, we do not reinsert evicted vertices to the lists to avoid sampling them more times than necessary.

With the union-find data structure, we require constant amortized time per sample and each vertex can be sampled at most twice, so the overall complexity is almost linear in n.

3.1.3 Heuristic 2: Online creation of the RG multigraph

In the second heuristic, the decisions of the rows are not given as input, but are instead defined during the course of the algorithm. Similar to the previous idea, this heuristic aims to reduce the possibility that a tree in H_1 gets created following an edge insertion into H_2.

More specifically, consider x chosen at Line 5. By our assumption, x has not picked its two choices yet, and we choose them at that exact time, in the way that benefits the algorithm the most. This is done as follows. Initially, we iterate over all of x's neighbors in the host graph G. Let c be one of x's neighbors and c^* be the sole unchecked vertex in c's connected component in H_2 or -1 if no unchecked vertices exist. If c^* is equal to -1, c's value is 0. If c^* has label $\mathcal{F}$ or $\mathcal{T}$ in H_1, c's value is 1. Otherwise, c's value is 2. Based on these assigned values, we partition the neighbors of x in G into three disjoint sets C_0, C_1 and C_2 such that C_i contains all neighbors of x with value equal to i. Selecting columns from C_2 is preferred, as columns in C_2 can avoid creating a tree in H_1. Row x will attempt to sample first from C_2, and if needed from C_1 or C_0, with a preference for C_1 over C_0. The sets C_0, C_1 and C_2 can be kept implicitly, and the selection process for each row x requires $O(d_x \cdot \log n)$ time, where d_x is the degree of x in G. Therefore the overall cost of this heuristic is almost linear in m using union-find.

3.1.4 The final 2OUTMC algorithm

Even with the methods discussed in this Subsection 3.1, if the vertex selections are done with uniform probability, the algorithm might yield poor results. Uniform selection favors vertices of high degree and can lead to unbalanced selections, where the majority of columns make their choices from a small set of rows and vice versa. Such an example is a graph where the ith row and ith column are connected for $i = 1, \ldots, n$, and additionally the first ℓ rows and columns are connected with every vertex on the opposite side. Then, in expectation $O(\frac{\ell-1}{\ell+1} \cdot n)$ rows (resp. columns) make both choices from the first ℓ columns (resp. rows), which can lead to a small maximum cardinality matching in the 2-out subgraph. We propose therefore that random decisions are not uniform.

The overall algorithm 2OUTMC is as follows. It takes the matrix representation of the given bipartite graph and scales it with a few steps of the Sinkhorn–Knopp algorithm to obtain $\mathbf{A_S}$. It then chooses two random neighbors for each column and row (if the second heuristic is not considered) using their respective probability distributions in the corresponding row and column of $\mathbf{A_S}$, which are given as input to Algorithm 1 modified accordingly with our suggestions in this subsection. In the end, Karp–Sipser is called to retrieve the matching dictated by the algorithm.

4 TRUNCRW: Truncated random walk with nonuniform sampling

Goel et al. [10] propose a randomized algorithm (of the Las Vegas type) that finds a perfect matching in a d-regular bipartite graph with n vertices in each side in $O(n \log n)$ time in expectation. This algorithm starts a random walk from a randomly chosen free column-vertex. At a column vertex c, the algorithm selects uniformly at random one of the row-vertices that are not matched to c, and goes to the chosen row vertex r. If r is free, then an augmenting path is obtained by removing possible loops from the walk. If r is matched, then the random walk goes to the mate of r. Goel et al. show that the total length of the random walks is $O(n \log n)$ in expectation, and thus the algorithm obtains a perfect matching in the stated time [10, Theorem 4]. They also show that one can obtain a Monte Carlo-type algorithm by truncating the random walks. The expected length of an augmenting path with respect to a given matching of cardinality j is $2(4 + 2n/(n - j))$, and the random walks could be truncated at this length to obtain near optimal matchings in $O(n \log n)$ time.

A random walk is easy to implement for d-regular bipartite graphs. At a column vertex c, one can create a random number between 1 and d in $O(1)$ time and choose the neighbor at that position, and repeat the experiment if the mate of c is chosen. This will take $O(1)$ time in expectation for each step of the walk, and the run time bound of $O(n \log n)$ is maintained.

Goel et al. discuss that the random-walk based algorithm will work for finding perfect matchings in the bipartite graph representation of a doubly stochastic matrix. When a given matrix has constant row and column sums and each entry is integral (in which case the matrix is doubly stochastic), by using an existing data structure [11] one can attain the same $O(n \log n)$ run time bound. For general doubly stochastic matrices without any bound on the entries, Goel et al. discuss an augmented binary search tree with which each selection step of the random walk can be implemented in $O(\log n)$ time, and obtain a run time of $O(m + n \log^2 n)$ in expectation, with a total of $O(m)$ preprocessing time.

$$\begin{pmatrix} \sqrt{2} & & & \\ & \frac{1}{\sqrt{2}} & & \\ & & \frac{1}{\sqrt{8}} & \\ & & & \frac{1}{\sqrt{8}} \end{pmatrix} \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{8}} & & & \\ & \frac{1}{\sqrt{8}} & & \\ & & \frac{1}{\sqrt{2}} & \\ & & & \sqrt{2} \end{pmatrix} = \begin{pmatrix} 1/2 & 1/2 & 0 & 0 \\ 1/4 & 1/4 & 1/2 & 0 \\ 1/8 & 1/8 & 1/4 & 1/2 \\ 1/8 & 1/8 & 1/4 & 1/2 \end{pmatrix}$$

Figure 2 – The matrix $\mathbf{A}$ associated with a 4×4 Hessenberg matrix, the scaling matrices $\mathbf{D_R}$ and $\mathbf{D_C}$, and the resulting doubly stochastic matrix $\mathbf{A_S} = \mathbf{D_R A D_C}$. In general, the $(n, 1)$-entry in the scaled matrix is of value $1/2^{n-1}$.

4.1 Implementation

We propose an efficient implementation of the truncated random walk-based matching heuristic.

Let c be a free column vertex with respect to a given matching of cardinality j. Assuming there is a perfect matching, one can find an augmenting path to match c, and a random walk can find it. The $O(\frac{n}{n-j})$ bound on the expected length of such a path will not hold if the bipartite graph is not regular. One may perform more than m steps, which is the worst case time complexity of deterministically finding an augmenting path starting from a free vertex.

We propose adaptations to make random selections more useful. Given a bipartite graph, we first scale its matrix representation $\mathbf{A}$ to obtain a doubly stochastic matrix $\mathbf{A_S}$ for random selections. The expected length of a random walk to find an augmenting path holds when $\mathbf{A_S}$ has bounded nonzero entries. In general, ones does not have any bound on the entries of $\mathbf{A_S}$. Consider the matrix $\mathbf{A}$ associated with an upper Hessenberg matrix of size n. $\mathbf{A}$ has a full lower triangular part, and additional $n-1$ entries $\mathbf{A}(i-1, i) = 1$ for $i = 2, \ldots, n$, and fully indecomposable. The 4×4 example along with its unique scaling matrices are shown in Fig. 2. In the resulting scaled matrix $\mathbf{A_S}(n, 1) = 1/2^{n-1}$ whose inverse is not bounded polynomially in n. Therefore, one needs an algorithm with $O(\log n)$ cost to select the next row vertex randomly from a given column vertex.

The main components of the proposed approach are as follows. For each column vertex c, with d_c neighbors, we have:

- $\mathsf{adj}_c[1, \ldots, d_c]$: an array keeping the neighbors of c.
- $\mathsf{wghts}_c[1, \ldots, d_c]$: the weight of the edges incident on c. This array is parallel to the first one so that the weight of the edge $(c, \mathsf{adj}_c[i])$ is $\mathsf{wghts}_c[i]$.
- $\mathsf{medge}[c]$: the position of the mate of c in the array adj_c, or -1 if c is not matched.

At the beginning, we compute the prefix sum of $\mathsf{wghts}_c[1, \ldots, d_c]$. After this operation, the total weight of the edges incident on c is $\mathsf{wghts}_c[d_c]$, and the weight of the edge $(c, \mathsf{mate}[c])$ is $\mathsf{wghts}_c[\mathsf{medge}[c]] - \mathsf{wghts}_c[\mathsf{medge}[c] - 1]$, assuming that $\mathsf{wghts}_c[0]$ signifies zero.

Given the prefix sums in $\mathsf{wghts}_c[1, \ldots, d_c]$, the position of the mate of c at $\mathsf{medge}[c]$, we can choose a random neighbor (which is not equal to $\mathsf{mate}[c]$) as shown in Algorithm 2. We use a binary search function, binSearch which takes an array, the array's start and end positions, a target value, and returns the smallest index of an array element which is larger than the given value with binary search (we skip the details of this search function). At Line 5, since c does not have a mate, we search in the whole list. At Line 8, since the prefix sum just before $\mathsf{medge}[c]$ is larger than the target value, we search in the first part of wghts_c until the current mate located at $\mathsf{medge}[c]$. At

Line 10, we search on the right of $\mathsf{medge}[c]$, by a modified target value to not to update the prefix sums when the mate changes.

Algorithm 2 Sampling a random neighbor of the column vertex c with d_c neighbors.

Require:
$\mathsf{adj}_c[1,\ldots,d_c]$: the neighbors of c
$\mathsf{wghts}_c[1,\ldots,d_c]$: the prefix sum of weights
medge_c: the position of c's mate at $\mathsf{adj}_c[1,\ldots,d_c]$; or -1 if c is not matched.

1: mwght $\leftarrow \mathsf{wghts}_c[\mathsf{medge}_c] - \mathsf{wghts}_c[\mathsf{medge}_c - 1]$ if $\mathsf{medge}_c \neq -1$, otherwise 0
2: totalW $\leftarrow \mathsf{wghts}_c[d_c] -$ mwght ▶ The total weight of the edges that can be sampled
3: create a random value rv between 0 and totalW
4: **if** $\mathsf{medge}_c = -1$ **then**
5: **return** binarySearch($\mathsf{wghts}_c[1,\ldots,d_c], rv$)
6: **else**
7: **if** $\mathsf{wghts}_c[\mathsf{medge}_c]-$mwght $\geq rv$ **then**
8: **return** binSearch($\mathsf{wghts}_c[1,\ldots,\mathsf{medge}_c - 1], rv$)
9: **else**
10: **return** binSearch($\mathsf{wghts}_c[\mathsf{medge}_c + 1,\ldots,d_c], rv +$ mwght) $+ \mathsf{medge}_c$
11: **end if**
12: **end if**

The sampling algorithm returns the index of the neighbor in adj_c different from the current mate in time $O(\log d_c)$, independent of the values of the edges. It thus respects the required run time bound. If we were to apply the rejection sampling (as discussed before for the regular bipartite graphs), the run time would depend on the value of the matching edge that we want to avoid. This could of course lead to an expected run time of more than $O(n)$.

There are two key components of Algorithm 2. The first one is the prefix sum, which is computed once before the random walks start and does not change. The second one is $\mathsf{medge}[c]$, the position of $\mathsf{mate}[c]$ in adj_c. The value $\mathsf{medge}[c]$ changes and needs to be updated when we perform an augmentation. We handle this update as follows. We keep the random walk in a stack by storing only the column vertices, as the row vertices direct the walk to their mate, or terminate the walk if not matched. We discard the cycles from the random walk as soon as they arise—this way we only store a path on the stack, and its length can be at most n. Storing a path also enables keeping the $\mathsf{medge}[\cdot]$ up-to-date. Every time we sample an outgoing edge from a column vertex c, we assign the location of the sampled row vertex in adj_c to a variable $\mathsf{nmedge}[c]$. When we find a free row, the stack contains the column vertices of the corresponding augmenting path, whose new mates' locations are in $\mathsf{nmedge}[\cdot]$ and thus can be used to update $\mathsf{medge}[\cdot]$.

We also incorporate a known heuristic called look-ahead [5, 6] for speeding up the augmenting path search in practice. In this heuristic, before sampling an arbitrary row vertex from a column c, we check if there is a free row vertex in the adjacency list of c. If so, such a row is returned, and the random walk terminates. The implementation of this heuristic has a total overhead of $O(m)$ for the whole course of the algorithm [5, 6].

The described procedure will work gracefully in expected $O(m + n \log n)$ time for regular bipartite graphs and for doubly stochastic matrices where the nonzero values do not differ by large. On the other hand, when there are large differences in edge weights, a random walk can get stuck in a

cycle. That is why truncating the long walks is necessary to make the algorithm work for any given doubly stochastic matrix. Furthermore, such a truncation is necessary if we are given a bipartite graph, and we scale its matrix representation. For the overall approach to be practical, we should not apply the scaling algorithms until convergence. As in the previous approaches [7, 8], we allocate a linear time of $O(m+n)$ for scaling. Applying Sinkhorn–Knopp algorithm for a few iterations will thus be allowable. The known convergence bounds for the Sinkhorn–Knopp algorithm [16, Thm. 4.5] apply asymptotically, therefore we do not have any bounds on the error after a few iterations; it can be large. That is why truncation makes the random walk based augmenting path search practical.

The overall algorithm TruncRW is thus as follows. It takes the matrix representation of the given bipartite graph and scales it with a few steps of the Sinkhorn–Knopp algorithm. Then for $j = 0$ to $n-1$, it uniformly at random picks a free column vertex, and starts a random walk starting from that column, for at most $2(4+2n/(n-j))$ steps, after which the walk is truncated. In principle, one can try a few more random walks from the the same column vertex upon a truncation.

4.2 Further comments

We can easily apply TruncRW to bipartite graphs with different number of vertices in each side. This is based on the fact that we can scale a rectangular $n_1 \times n_2$ matrix (say $n_1 \geq n_2$) so that all columns have sum of 1, and all rows have equal sum of n_2/n_1, if there is matching covering all columns, and all entries can be put in such a matching. Then, all components of TruncRW work without any change.

If there is no total support, then Sinkhorn–Knopp works in such a way that the entries that cannot put into a perfect matching tend to zero. This is helpful in TruncRW's context, as the corresponding edges will not likely be selected in a random walk. If there is no perfect matching, then little is known about scaling. It is our experience that the Sinkhorn–Knopp iterations tend to zero out entries that cannot be put into a maximum cardinality matching. Therefore, in this case again, scaling, random selection, and truncation should help. We present some experiments to support this observation, and leave the question of showing this theoretically as an open problem.

5 Experiments

We implemented 2outMC and TruncRW in C/C++, and the codes are accessible from `http://perso.ens-lyon.fr/bora.ucar/codes.html`. The codes are compiled with "-O3" flag, and were run on a machine with 2 x Intel Xeon CPU E5-2695 CPUs and 756 GB RAM.

2outMC and TruncRW are evaluated both on real-life and synthetic bipartite graphs with, unless otherwise stated, equal number of vertices in each side. The real-life graphs correspond to a set of 39 large sparse square matrices from the SuiteSparse Collection [4]. These matrices are automatically selected from all square matrices available at the collection, with $1,000,000 \leq n \leq 28,000,000$, and with at least two nonzeros per row or column. The associated bipartite graphs have perfect matchings. We compared the two algorithms against the widely used version of Karp–Sipser, which we will refer to as Karp–Sipser. The practical version of Sinkhorn-Knopp is shown with SK-t, where t is the number of allowed iterations.

We also investigated if random 2-out bipartite graphs of a general host graph have perfect matchings if rows and columns select neighbors with the probabilities in the scaled matrix repre-

$\frac{m}{n}$	[0,10)		[10,20)		[20,30)		[30,40)		[40,50)	
#Instances	27		5		5		1		1	
	#PM	deficiency	#PM	deficiency	#PM	deficiency	#PM	deficiency	#PM	deficiency
Model M_1	0	223	0	8	1	20	0	2	1	0
Model M_2	27	0	3	3	1	10	0	1	0	1

Table 1 – We divide the graphs into five groups. The ith group consists of graphs whose $\frac{m}{n}$ ratio is between $10(i-1)$ and $10i$. For each goup, we give the number of instances in which a 2-out graph built using the models M_1 and M_2 has a perfect matching. We also give the largest difference from the maximum cardinality of a matching. The results are with SK-5.

sentation.

5.1 Investigation of the maximum cardinality matching in 2-out subgraphs

In the first set of experiments we investigate the claim that G_2 will likely have a perfect matching for G, if created with the probabilities in the scaled matrix.

We consider two different models to create G_2. In the model M_1, row choices are independent of the column choices. Under this model, a row and a column can select each other resulting in parallel edges—one of them is then discarded. The model M_2 tries to avoid parallel edges. In this model, all columns perform their selections. Then, each row r attempts to randomly choose two columns, only from those that did not select r. These selections again are based on the scaled matrix. In this model, parallel edges can arise (and be discarded) exist only when a vertex v is connected in the 2-out graph with all of its neighbors in G, because it is impossible for v to select otherwise. We experimented three times with each real-life graph. M_i's result is the maximum of those three experiments. In each test, we first created the choices of all columns. Then we allowed the two models to generate the choices of the rows accordingly. The results are shown in Table 1 and are with SK-5. As seen in this table, the random G_2 graphs generated with the model M_1 have near perfect matchings, but they do not contain perfect matchings in most cases. In contrast, the random G_2 graphs generated by M_2 in many cases contain a perfect matching. In only a few graphs this does not hold true, and in these cases the deficiency is no more than 10.

5.2 Tuning 2OUTMC and TRUNCRW

We first evaluate different algorithmic choices made for 2OUTMC and TRUNCRW. The experiments in this section are on real-life instances, and with five iterations of the scaling algorithm.

5.2.1 Investigating 2OUTMC

We perform a similar analysis to see the effects of the proposed two heuristics of Subsection 3.1 on 2OUTMC. Without the two heuristics 2OUTMC on average finds matchings with quality 0.9983 on the instances shown in Table 2. If the two heuristics are enabled, then 2OUTMC obtains matchings with average matching quality 0.9997. Looking over the run time analysis of the two approaches shown in Table 2, we observe that the two heuristics increase the run time, and significantly on one instance. We find the increase acceptable, and keep the heuristics on for 2OUTMC below, except when noted.

instance	n	m	1 attempt time	5 attempts time	10 attempts time	Doubled length walk time
hugebubbles-00020	21	63	18.10	40.91	92.39	27.04
channel-500	4	85	5.59	10.41	13.04	9.70
cage15	5	99	7.22	10.98	11.81	8.89
delaunay_24	16	100	12.30	24.95	24.37	16.79
nlpkkt240	28	760	43.66	213.77	390.04	118.37

Table 2 – Execution times in seconds of different alternatives for TruncRW in five large graphs. The results are with SK-5. The values of n and m are in the order of million. The matrix name channel-500x100x100-b050 is shortened to channel-500.

5.2.2 Tuning TruncRW

The experiments here are on real-life instances and with SK-5.

Recall that TruncRW tries to find a mate for a column-vertex a certain number of times before giving up and moving to the next vertex. When we allowed TruncRW just a single attempt, it was unable to find a perfect matching in any of the cases, and its average quality was 0.9984. When we allowed five attempts, TruncRW found a perfect matching for 13 graphs, and its average quality was 0.9999. With 10 attempts, it managed to find a perfect matching in 5 additional graphs. This verifies that allowing more attempts indeed improves on the performance of the algorithm. The drawback, however, is the increased run time that comes with the increasing number of attempts. This is shown in Table 2, where we compare different number of attempts, on five of the largest instances in our collection. While in a few graphs this additional time is perhaps acceptable, in nlppkt240 the difference is huge with 5 or 10 attempts. Since additional attempts bring only little improvement in quality, TruncRW starts a random walk from a vertex only once.

We also test the effects of the look-ahead mechanism. Let us define the walk efficiency of TruncRW as the ratio of the cardinality of the matching found to the total length of the random walks. The higher this ratio, the more useful the random walks are. We evaluate the walk efficiency on a set of seven instances (real-life instances having at most 10000000 edges). We test both with and without scaling and report the results of the 14 tests. In 13 cases, the look-ahead mechanism improved the walk efficiency. The geometric mean (of 14 cases) of the ratios of walk efficiencies with look-ahead to that of without was 1.37. In the case where the look-ahead did not help (ratio was 0.71 in Hamrle3), the maximum deviation of a row or column sum from one after SK-5 was 0.28, which is high. We conclude that the look-ahead mechanism is very helpful.

Finally we test the effects that the length of the augmenting walk has on TruncRW. We doubled the allowed the length of a random walk to $4(4 + 2n/(n - j))$. On average, the quality rose from 0.9984 to 0.9998. This modification was not able to find a perfect matching in any of the 39 instances. Results with its run time on the five large graphs can be seen on Table 2. As seen in the table, the increase in the run time can be too large to consider allowing longer walks. We therefore keep $2(4 + 2n/(n - j))$ as the truncation length.

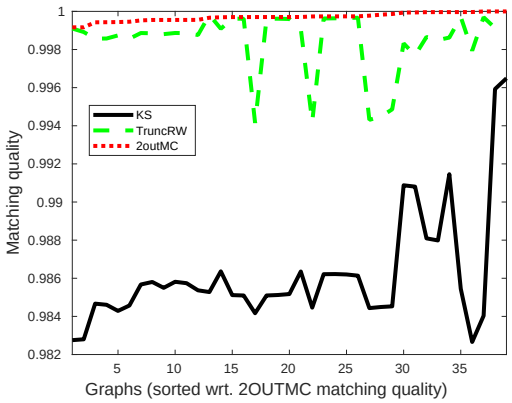


Figure 3 – The average matching found by 2OUTMC with heuristics and TRUNCRW in comparison with KARP–SIPSER.

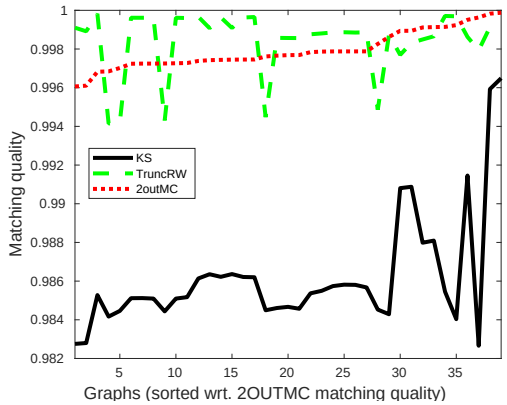


Figure 4 – The average matching found by 2OUTMC (without heuristics) and TRUNCRW in comparison with KARP–SIPSER.

5.3 Quality comparisons

5.3.1 Results with real-life graphs

We now compare 2OUTMC and TRUNCRW against each other on the real-life graphs. The results can be seen in Figure 3, where we plot the the ratio of the cardinality of the matchings found by different algorithms to the maximum cardinality of the matching. The results are with five iterations of SK. As can be observed, both 2OUTMC and TRUNCRW obtain near perfect results. The average quality of 2OUTMC is 0.9997 and that of TRUNCRW 0.9984. Both algorithms never drop below 0.99 in any of the 39 cases.

The figure also shows the quality of KARP–SIPSER. It obtains a matching of 0.9862 times the maximum on average, and always has smaller cardinality than TRUNCRW and 2OUTMC. While all algorithms obtain matchings of high quality, the absolute different is remarkable in some cases. For example, the largest difference observed between the cardinalities of 2OUTMC and KARP–SIPSER was 434,865 additional matched vertices in favor of 2OUTMC.

	Karp–Sipser	2outMC			TruncRW (1 attempt)			TruncRW (5 attempts)		
n	Quality	Uniform	SK-5	SK-20	Uniform	SK-5	SK-20	Uniform	SK-5	SK-20
2500	0.80	0.80	0.92	0.95	0.80	0.90	0.94	0.85	0.93	0.96
5000	0.74	0.80	0.92	0.95	0.81	0.90	0.94	0.85	0.92	0.95
10000	0.77	0.80	0.92	0.95	0.81	0.90	0.94	0.85	0.92	0.96

Table 3 – Average quality of the matchings found by the algorithms on graphs from the synthetic family $\mathcal{I}$ for $n \in \{2500, 5000, 10000\}$.

	Karp–Sipser	2outMC			TruncRW		
h	Quality	Uniform	SK-5	SK-20	Uniform	SK-5	SK-20
2	0.97	0.63	1.00	1.00	0.73	0.99	0.99
8	0.82	0.63	0.99	0.99	0.73	0.99	0.99
32	0.69	0.63	0.99	0.99	0.73	0.99	0.99
128	0.63	0.63	0.99	0.99	0.73	0.99	0.99
512	0.64	0.63	0.98	0.99	0.73	0.99	0.99

Table 4 – Average quality of the matchings found by the algorithms on graphs from the synthetic family $\mathcal{J}$ for $n = 5000$ and various values of h.

5.3.2 Results with synthetic graphs

We now give some results with two synthetic families $\mathcal{I}$ and $\mathcal{J}$ of graphs, whose matrix representation does not have total support. These families also highlight the importance of scaling.

A bipartite graph G with n vertices per side belonging to $\mathcal{I}$ is defined as follows. For $i \leq j$, we connect row-vertex i with column-vertex j, we also connect row vertex r_2 with column vertex c_1, and also row vertex r_n with with column vertex c_{n-1}. The graphs of this family are hard for the common Karp–Sipser heuristic, because only a few of the edges participate in a perfect matching. Hence it can make a lot of erroneous decisions due to randomness. Likewise, due to the vast number of entries without support in the matrix representation SK will take many iterations to properly scale the matrix.

In Table 3, we give the results of the algorithms for a few graphs from this family. As can be seen, despite the lack of total support, both 2outMC and TruncRW obtain better results than Karp–Sipser, being over 0.9 in all cases, with the difference in quality being about 20%. With more iterations of the scaling algorithm, the quality becomes better. If we sample with uniform probability rather than use SK, there is a 10% drop in quality.

The second family $\mathcal{J}$ consists of another family of graphs for which Karp–Sipser exhibits poor behavior. To create a member of $\mathcal{J}$, we separate vertex set R into $R_1 = \{r_1, \ldots, r_{n/2}\}$ and $R_2 = \{r_{n/2+1}, \ldots, r_n\}$ and likewise for C. At first, we connect all vertices of R_1 with those of C_1. Edges $(r_i, c_{n/2+i})$ and $(r_{n/2+i}, c_i)$ for $i = 1, \ldots, n/2$ are added to introduce a perfect matching. A parameter h is used to connect h vertices from R_1, and h vertices from C_1 to every vertex on the opposite side. As h gets larger, Karp–Sipser has more and more difficulty to apply its deterministic reduction, and its quality drops significantly. As can be seen in Table 4, its quality drops over 30% between $h = 2$ and $h = 512$. On the contrary, 2outMC and TruncRW both obtain a near perfect matching. With the exception of $h = 512$ they never drop below 0.99 with five iterations of SK. The matrices associated with the graphs from $\mathcal{J}$ do not have total support. Nonetheless, just a few iterations sufficed to obtain good results. If vertices select with uniform

	10000×10000		12000×10000	
d	sprank	TruncRW	sprank	TruncRW
2	7787	0.9888	8724	0.9919
3	9266	0.9697	9667	0.9958
4	9761	0.9828	9899	0.9995
5	9918	0.9922	9973	1.0000

Table 5 – The quality of TruncRW on bipartite graphs without perfect matchings.

probability, the quality is significantly reduced, especially for 2outMC.

5.3.3 TruncRW's quality on bipartite graphs without perfect matchings

We analyze TruncRW on a class of bipartite graphs without perfect matchings. These bipartite graphs correspond to square (10000×10000) and rectangular matrices (12000×10000) with a uniform nonzero distribution. These matrices are generated with `sprand` command of Maltab and have about $d \times 10000$ nonzeros for $d = 2, 3, 4, 5$. The matrix representation of the bipartite graphs are scaled with 10 iterations of SK. For each d, we create five random matrices and run TruncRW on the corresponding five instances with the default configuration. We report the worst quality of the five instances in Table 5. As seen in this table, TruncRW works just fine for this case. We did not report in the table but with increased SK iterations, the results improve, which is in accordance with earlier work [7].

5.4 Run time comparisons

Here, we compare the two algorithms in terms of run time against Karp–Sipser on the five large graphs of Table 2. We give results in Table 6 with SK-1 and SK-5, as well as with no scaling (uniform selection).

Karp–Sipser is a linear time algorithm and is usually faster than both TruncRW and 2outMC. However, as we can observe, TruncRW is only slightly slower than Karp–Sipser and in the largest instance its performance is slightly better. 2outMC exhibits the worst performance, but it is still close to Karp–Sipser except for the first and fourth instances. The run time of the two algorithms also seems unaffected by whether or not one uses a scaling algorithm. The quality of the matchings found is 0.99, even with uniform selections. The larger run time of 2outMC can partly be explained by the fact that it is more sophisticated than TruncRW. We close the discussion on performance by noting that implementations of TruncRW and 2outMC can potentially be improved.

6 Conclusions

We have examined two randomized algorithms for the maximum cardinality matching problem in bipartite graphs. These algorithms originally were designed for random 2-out graphs, and d-regular bipartite graphs. We have shown how to convert them into efficient and effective heuristics. Our experimental results show that these approaches obtain near perfect matchings in real-life and synthetic instances while having a near linear time run time.

			SK-t		2OUTMC				TRUNCRW			
	KARP–SIPSER		$t=1$	$t=5$	uniform		SK-1	SK-5	uniform		SK-1	SK-5
instance	quality	time	time	time	quality	time	time	time	quality	time	time	time
hugebubbles-00020	0.99	10.10	1.12	4.79	0.99	36.03	35.33	35.37	0.99	18.35	18.14	18.10
channel-500	0.99	5.07	0.27	1.18	0.99	6.52	6.41	6.41	0.99	5.55	5.45	5.59
cage15	0.99	6.15	0.32	1.39	0.99	8.60	9.14	8.63	0.99	7.20	7.19	7.22
delaunay_24	0.99	10.09	0.79	3.20	0.99	20.92	21.27	21.43	0.99	12.33	12.21	12.30
nlpkkt240	0.98	46.73	1.92	8.50	0.99	43.01	42.00	43.44	0.99	44.60	44.44	43.66

Table 6 – Run times in seconds of the algorithms on five large graphs with no scaling, or with SK-1 and SK-5. SK-t's time should be added to that of 2OUTMC and TRUNCRW.

Our adaptations of the two algorithms are based on the premise that 2-out graphs sampled from a host graph have perfect matchings, assuming that the matrix representation of the host graph is fully indecomposable. We showed evidence that this may be true and even if not, the sampled graphs have close to perfect matchings. A proof or the disproof of such 2-out graphs having perfect matchings is certainly welcome. Furthermore, this work is the first attempt to implement 2OUTMC. Therefore, there is room for improved performance.

References

[1] Z. Allen-Zhu, Y. Li, R. Mendes de Oliveira, and A. Wigderson. Much faster algorithms for matrix scaling. In *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS*, pages 890–901, Berkeley, CA, USA, October 2017.

[2] C. Berge. Two theorems in graph theory. *Proceedings of the National Academy of Sciences of the USA*, 43:842–844, 1957.

[3] M. B. Cohen, A. Madry, D. Tsipras, and A. Vladu. Matrix scaling and balancing via box constrained newton's method and interior point methods. In *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS*, pages 902–913, Berkeley, CA, USA, October 2017.

[4] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, 2011. ISSN 0098-3500.

[5] I. S. Duff. On algorithms for obtaining a maximum transversal. *ACM Transactions on Mathematical Software*, 7(3):315–330, 1981.

[6] I. S. Duff, K. Kaya, and B. Uçar. Design, implementation, and analysis of maximum transversal algorithms. *ACM Transactions on Mathematical Software*, 38:13:1–13:31, 2011.

[7] F. Dufossé, K. Kaya, and B. Uçar. Two approximation algorithms for bipartite matching on multicore architectures. *Journal of Parallel and Distributed Computing*, 85:62–78, 2015.

[8] F. Dufossé, K. Kaya, I. Panagiotas, and B. Uçar. Approximation algorithms for maximum matchings in undirected graphs. In *2018 Proceedings of the Seventh SIAM Workshop on Combinatorial Scientific Computing*, pages 56–65, 2018.

[9] A. Frieze and T. Johansson. On random k-out subgraphs of large graphs. *Random Structures & Algorithms*, 50(2):143–157, 2017.

[10] A. Goel, M. Kapralov, and S. Khanna. Perfect matchings in $O(n \log n)$ time in regular bipartite graphs. *SIAM Journal on Computing*, 42(3):1392–1404, 2013.

[11] T. Hagerup, K. Mehlhorn, and J. I. Munro. Maintaining discrete probability distributions optimally. In A. Lingas, R. Karlsson, and S. Carlsson, editors, *20th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 253–264, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[12] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[13] R. M. Karp and M. Sipser. Maximum matching in sparse random graphs. In *22nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 364–375, Los Alamitos, CA, USA, 1981. IEEE Computer Society.

[14] R. M. Karp, A. H. G. Rinnooy Kan, and R. V. Vohra. Average case analysis of a heuristic for the assignment problem. *Mathematics of Operations Research*, 19(3):513–522, 1994.

[15] K. Kaya, J. Langguth, I. Panagiotas, and B. Uçar. Karp–Sipser based kernels for bipartite graph matching. In *SIAM Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 134–145, Salt Lake City, Utah, US, January 2020.

[16] P. A. Knight. The Sinkhorn–Knopp algorithm: Convergence and applications. *SIAM Journal on Matrix Analysis and Applications*, 30(1):261–275, 2008.

[17] P. A. Knight and D. Ruiz. A fast algorithm for matrix balancing. *IMA Journal of Numerical Analysis*, 33(3):1029–1047, 2013.

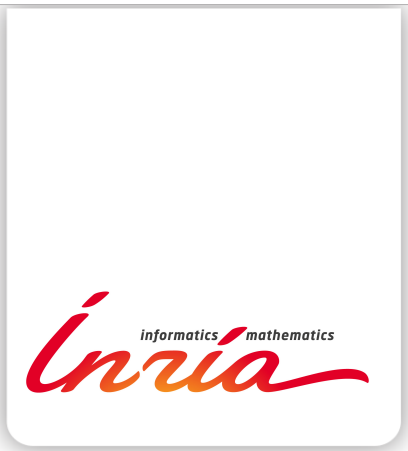
[18] J. Langguth, F. Manne, and P. Sanders. Heuristic initialization for bipartite matching problems. *Journal of Experimental Algorithmics (JEA)*, 15:1–22, 2010.

[19] M. Mitzenmacher and E. Upfal. *Probability and computing: Randomized algorithms and probabilistic analysis*. Cambridge University Press, 1st edition, 2005.

[20] A. Pothen, S. M. Ferdous, and F. Manne. Approximation algorithms in combinatorial scientific computing. *Acta Numerica*, 28:541–633, 2019.

[21] R. Sinkhorn and P. Knopp. Concerning nonnegative matrices and doubly stochastic matrices. *Pacific Journal of Mathematics*, 21(2):343–348, 1967.

[22] D. Walkup. Matchings in random regular bipartite digraphs. *Discrete Mathematics*, 31(1): 59–64, 1980.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée

655 avenue de l'Europe Montbonnot

38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399