Karp-Sipser based kernels for bipartite graph matching

Kamer Kaya, Johannes Langguth, Ioannis Panagiotas, Bora Uçar

Karp-Sipser based kernels for bipartite graph matching

Kamer Kaya* Johannes Langguth† Ioannis Panagiotas‡ Bora Uçar§

Abstract

We consider Karp–Sipser, a well known matching heuristic in the context of data reduction for the maximum cardinality matching problem. We describe an efficient implementation as well as modifications to reduce its time complexity in worst case instances, both in theory and in practical cases. We compare experimentally against its widely used simpler variant and show cases for which the full algorithm yields better performance.

1 Introduction

Data reduction, or preprocessing, is applied in different problems to shrink the size of the instance before actually solving the problem. The aim is to significantly reduce the size with fast methods so that the exact algorithm whose run time complexity might be high can benefit from the reduction. In this paper, we focus on data reduction algorithms in the context of finding maximum cardinality matchings in bipartite graphs.

Karp and Sipser [9] describe two reduction rules for the maximum cardinality matching problem in unweighted, simple bipartite graphs. The first rule matches a vertex with a single edge to its unique neighbor. The second rule takes a vertex v with two neighbors u and w, removes v and merges u and w into a single vertex. The first rule is easy to implement efficiently, while a straightforward implementation of the second reduction rule can take quadratic time, which is potentially higher than the $O(m\sqrt{n})$ time the exact matching algorithm [8] takes for sparse bipartite graphs with m edges and $2n$ vertices equally distributed to each part. When used as a heuristic for the matching problem, a random edge is added to the matching when the reduction rules do not apply. When an edge u, v is added to the matching, other edges incident on u and v are discarded. This reduces the degree of the vertices adjacent to u and v and hence creates opportunities to apply the reduction rules.

*Sabancı University, Computer Science and Engineering, Turkey, kaya@sabanciuniv.edu

†Simula Research Laboratory, Norway, langguth@simula.no

‡ENS Lyon, ioannis.panagiotas@ens-lyon.fr

§LIP, UMR5668 (CNRS - ENS Lyon - UCBL - Université de Lyon - INRIA), Lyon, France, bora.ucar@ens-lyon.fr

Our focus in this paper is to design and analyze efficient algorithms for implementing both reduction rules for the matching problem. To see the impact of data reduction we conducted several experiments using only the first rule and also, both rules. We also investigate these alternatives when Karp–Sipser is used as a cheap matching heuristic. Note that in the former model, the matching decisions are concordant with at least one maximum cardinality matching. However in the latter case, there may not be a maximum cardinality matching obeying all these decisions.

The paper is organized as follows: Section 2 introduces the notation and the background on the Karp–Sipser heuristic. The original heuristic with two rules as proposed is discussed in Section 3 with various approaches for implementation. Section 4 presents the experimental results and Section 5 summarizes the literature focusing on the original Karp–Sipser heuristic. Section 6 concludes the paper.

2 Notation and background

Let $G = (V_A \cup V_B, E)$ be a simple bipartite graph with $|V_A| = |V_B| = n$ and $|E| = m$. Two vertices are neighbors in G, if they share an edge. A vertex with k neighbors is called a *degree-k* vertex. The degree of a vertex v is shown as d_v. A matching in a graph G is a set of disjoint edges, such that no two of them share a common vertex. We use $\mathcal{M}(G)$ to refer to the *maximum cardinality of a matching* in G. If $\mathcal{M}(G) = n$, a maximum cardinality matching is called *perfect*.

For a graph G, there exist polynomial time algorithms to find a matching with cardinality $\mathcal{M}(G)$. In practice, one very popular strategy to find $\mathcal{M}(G)$ is to use a two-step process. In the first step, a *cheap* initialization heuristic is used in order to quickly find a matching of large cardinality which is usually *maximal* i.e. cannot be extended via simple edge additions. This matching is given as an input to the second step and improved via a (relatively) more expensive *exact* algorithm. It has been empirically shown that this strategy is much faster than running an exact algorithm from the beginning—see for example Langguth et al. [12].

While there exist different initialization heuristics for the first step, the Karp–Sipser (KS) heuristic empirically performs better than many, if not all, on aver-

age [6, 12]. In this paper, we will focus on this heuristic and its implementation. Karp–Sipser is based on performing reductions on a graph with no degree-0 vertices (they are discarded throughout); when a degree-1 or degree-2 vertex appears, 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 decision is optimal in the sense that there exists at least one maximum cardinality matching in the current graph containing $\{u, v\}$.

- **Rule-2**: At any time, if there are no degree-1 vertices, and a degree-2 vertex u with neighbors v and w appears, u and its edges are removed from the current graph, and v and w are merged to create a new vertex vw whose set of neighbors is the union of those of v and w (excluding u). Karp and Sipser showed that a maximum cardinality matching for the reduced graph can be extended to obtain maximum cardinality matching for the original graph by matching u with either v or w depending on vw's match.

- When none of these rules can be applied, a random edge from the current graph is added to the matching. This decision may not be optimal. Therefore, this step is not performed in the data reduction setting.

Both Rule-1 and Rule-2 have the property that they preserve $\mathcal{M}(G)$. We will use the notation $G^{(0)} = G$ to denote the initial graph, and $G^{(t)}$ to denote the graph after t random or rule-based decisions. Let $G^{(k)}$ be the first graph where neither Rule-1 nor Rule-2 is applicable. We call $G^{(k)}$ a *kernel* of G for the maximum cardinality matching problem, i.e., a reduced, smaller graph where one can obtain a maximum cardinality matching for G given a maximum cardinality matching for $G^{(k)}$ by following the reductions in reverse order. Thus, in addition to the well known initialization strategy, another use of the Karp–Sipser algorithm is to obtain $G^{(k)}$ initially, and then apply an exact algorithm on the smaller subgraph $G^{(k)}$, rather than G. However, obtaining $G^{(k)}$ can be computationally expensive.

Rule-1 is simple to implement. However, Rule-2 is more complicated and requires more effort to be used in a matching heuristic. That is why in practice, the KS heuristic has usually been associated with its simpler variant KS_{R1}, which only applies Rule-1, and when necessary, random choices. Although it does not exploit the second rule, KS_{R1} has been shown to obtain large cardinality maximal matchings on real life graphs, see e.g. [6, 10].

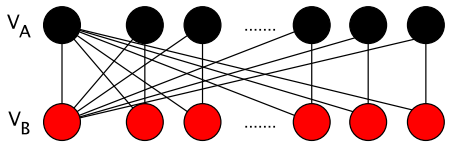


Figure 1: A toy bipartite graph $G = (V_A \cup V_B, E)$ with vertex sets V_A and V_B. Except for the leftmost vertices from each vertex set, each vertex has two neighbors and will be removed by Rule-2 of KS. For every $G^{(t)}$, the merged vertices will have degrees $(n+1-t)$ and 2. Hence, the total time for the naive KS implementation on this instance is $\Theta(n^2)$.

3 Karp-Sipser: The Heuristic with two rules

KS_{R1} can be implemented in $O(n+m)$ time by keeping an up-to-date list of all degree-1 vertices. When the edge $\{u, v\}$ is added to the matching, if u is a degree-1 vertex, one needs to visit only v's neighbors to reduce their degrees. Otherwise, u's and v's neighbors must be visited and their degrees are updated. Hence, with only Rule-1, each adjacency list is accessed at most once and KS_{R1} runs in linear time. This implies that the complexity of KS depends on the the cost of applying Rule-2.

For any graph $G^{(t)}$ that appears during the execution of the heuristic, let u be a degree-2 vertex with neighbors v and w. According to Rule-2, v and w must be merged to $vw \in G^{(t+1)}$. With a naive approach, this operation takes $\Theta(d_v + d_w)$ time. Since there can be at most n merge operations, the time complexity of this strategy is $O(n^2)$. This bound is tight as one can create highly sparse graphs for which KS requires quadratic time as shown in Figure 1.

3.1 An expected $O(m \log n)$ time algorithm.

Let u be a degree-2 vertex with neighbors v and w. For KS, a merge operation due to u performs three steps:

1. Merging v's adjacency list with that of w,
2. Performing degree reductions for the vertices that are both v's and w's neighbors,
3. Updating the adjacency lists of the vertices which are neighbors of v and w.

Assume that it is possible to perform this merge operation in $O(\min(d_v, d_w))$ time. We will label the KS variant with this merge complexity as KS-min and show in the following theorem that its worst-case time complexity is $O(m \log n)$ which is better than $O(n^2)$ for sparse graphs where $m \ll n^2$.

THEOREM 3.1. KS-min *runs in* $O(m \log n)$ *time.*

Proof. Consider a hypothetical algorithm KS*-min which operates on a multi-graph where multiple/parallel edges are allowed between two vertices while merging the vertices via Rule-2. Hence the adjacency lists contain parallel edges. This means that KS*-min keeps all the edges in the graph at any time step and simply combines the adjacency lists of the two vertices during a merge operation. Similar to KS-min, we assume that the cost of each merge is equal to the size of the smaller adjacency list. Hence, for a single merge, the amortized cost per edge in the smaller list becomes $O(1)$.

Let v and w be the vertices in $G^{(t)}$ and d_v, d_w with $d_v \leq d_w$ be the number of edges (parallel edges are allowed) incident on v and w. Let vw be the merged vertex in $G^{(t+1)}$. Note that d_{vw} is at most m, the total number of edges. We also have $2d_v \leq d_{vw}$, since we allow parallel edges. Therefore, a single edge can be in the smaller adjacency list in at most $\log m$ merge operations. Having $O(1)$ amortized complexity for each such edge, the total time complexity of KS*-min becomes $O(m \log m)$. Since m is $O(n^2)$, the complexity is $O(m \log n)$.

To analyze the complexity of KS-min, assume the same merge operations are followed by KS*-min and KS-min. Since the sizes of the merged adjacency lists in KS-min are always smaller than or equal to the corresponding lists in KS*-min, the time complexity of KS-min is also $O(m \log n)$. □

It remains to discuss how to implement a merge operation on v and w in KS-min. Assume $d_v \leq d_w$. First, the larger list, i.e., w's list is kept intact and w becomes the merged vertex vw in the reduced graph. Then each neighbor x of v is processed one after another. To keep the graph simple and to correctly update the degree reductions for the common neighbors of v and w, x is first searched in the adjacency list of w. If x is already in w's list its degree is decremented and v is removed from x's adjacency list. Otherwise, x is inserted to w's list. Furthermore, v is also replaced by w in x's list.

To perform these operations in expected constant time per edge, we can use a hash table to store all the edges in the graph. This hash table is used to query the existence of x in w's list by looking whether $\{x, w\}$ exists or not in the hash table. With this data structure, we have $O(m \log n)$ expected time complexity for KS-min and linear space. Instead of a hash table, one can use a data structure to store the edges with $O(\log m)$ insertion, update and lookup time such as a binary search tree. Such a data structure yields an $O(m \log^2 n)$ worst-case time complexity for KS-min in linear space.

3.2 An implementation with list caching. In order to merge the adjacency lists of two vertices v and w, one can use a dense, 0-1 array L of size n. This array represents v's adjacency list with $L[x] = 1$ iff $\{v, x\}$ exists in the current graph. Once such a list is created for one of the merging vertices, the edges of the other vertex can be looked up in L. Note that this array needs to be created every time there is a merge involving v and each time, one needs to re-iterate over v's adjacency list.

A natural optimization to this approach would be to allocate such arrays for some vertices and persistently keep them in memory, i.e., *cache* them. With caching, we do not have to re-iterate over these vertices' adjacency lists each time they participate in a merge. Indeed, if we cache two arrays for the two leftmost vertices in Figure 1, the complexity drops from $\Theta(n^2)$ to $\Theta(n)$. We refer to this variant as KS_{cache}.

There are many different strategies to decide on which vertices to keep in the cache. For example, one can cache the lists for the k most recently merged vertices, or k highest degree vertices. In Theorem 3.2, we show a negative result holding for any arbitrary caching strategy.

THEOREM 3.2. *KS_{cache} using k arrays and applying Rule 1, Rule 2 and random decisions has an instance requiring $\Omega\left(n^2/k\right)$ time for all possible caching policies.*

Proof. Assume that we have enough memory to cache the adjacency lists of k vertices in dense form. Let G be the $n \times n$ bipartite graph shown in Fig. 2 in which the vertices in V_A and V_B are shown in black and red, respectively. The graph contains $2k$ identical subgraphs, each having $\frac{\left(\frac{n}{2k}-1\right)}{2}$ two-by-two complete bipartite structures and an additional V_A (V_B) vertex that is connected to every other V_B (V_A) vertex in its subgraph. These extra vertices are labeled as a_i and b_i for the ith subgraph (in fact, they correspond to the left most vertices in Fig. 1).

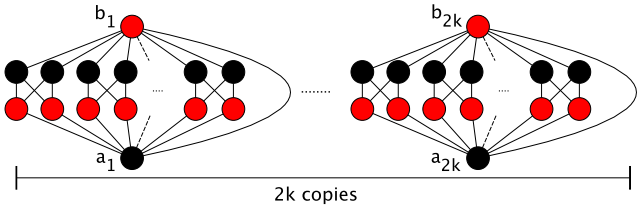


Figure 2: A toy bipartite graph $G = (V_A \cup V_B, E)$ where $|V_A| = |V_B| = n$ and the vertices in V_A and V_B are colored with black and red, respectively.

As the figure shows, the minimum degree in the initial graph is three and a random decision is required.

When a random decision hits into a two-by-two part in the ith subgraph, the corresponding matching and the removal of the matched vertices yield two possible merges involving either a_i or b_i. These potential merges eliminate each other, i.e., only one of them is possible. Furthermore, after the merge operation, the minimum degree in the graph will go back to three. Hence, assuming all the random edges are from the two-by-two bipartite subgraphs, each random choice is followed by a merge which is then followed by another random decision. Since the decisions are random, we cannot build a caching strategy based on them.

Hence, regardless of caching policy, it is always possible that the dense form of both a_i and b_i is not in the cache until at least k components are reduced completely. Doing so for only one component without touching to others takes

$$\sum_{j=0}^{\left(\frac{n}{2k}-1\right)/2} \left(\frac{n}{2k} - 2j\right) = \Theta\left(\frac{n^2}{k^2}\right)$$

time since the degrees of a_i (as well as b_i) vertices are reduced by two after each step. Since at least k components must be consumed, the complexity until this point is $\Omega\left(n^2/k\right)$ which implies a lower bound on the execution time of KS_{cache} for the instance in Fig 2 assuming that the random choices always hit two-by-two subgraphs. □

3.3 An alternating component approach. Here, we consider and extend an idea that was first introduced in [12]. There, the authors used the term *alternating component* to refer to a connected subgraph consisting of a set of *boundary vertices* connected by paths. The boundary vertices are vertices of arbitrary degree greater than one. The paths are required to contain an even number of edges, have a boundary vertex as beginning and endpoint, and all other vertices in each path are required to be of degree 2. An example can be seen in Fig. 3. In addition, we only consider components whose paths are maximal, i.e., they cannot be extended without violating the above definition.

If an alternating component contains a cycle, then there is no need to explicitly merge vertices in the component. Since every edge has at least one incident vertex of degree 2, at least half of the vertices in such a cycle have degree 2. Thus, for any edge of the cycle, there is a maximum cardinality matching containing that edge. Thus, we can pick an arbitrary edge and match it. The remaining component will be matched via the application of Rule-1.

If a component is acyclic, it can be immediately merged into a single vertex via Rule-2. To see this, consider two vertices that are both connected to a degree-2 vertex in a simple alternating component. Thus, each application of Rule-2 will reduce the length of a path between boundary vertices in an alternating component by two, and because the length is even, they will be merged when it reaches zero. The same applies to any such path in an alternating component. The advantage of doing so is that we do not have to perform merges immediately. Instead, we maintain components whose merges can then be performed together more efficiently.

In [12], only the first operation was performed. Acyclic components were maintained, but not explicitly merged. This resulted in a useful heuristic, but it is not equivalent to an implementation of Rule-2. Here, we show how to use alternating components to implement the full KS algorithm. We will call this version KS_{comp}.

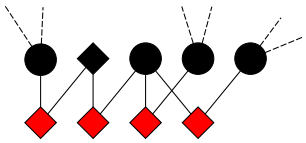


Figure 3: An example of a component: diamond shaped vertices correspond to the degree-2 vertices and circle shaped vertices correspond to boundary vertices. The dashed lines correspond to additional edges, unrelated to the displayed component.

Similar to the basic implementation, the algorithm keeps stacks of degree-1 and degree-2 vertices. If a degree-1 vertex exists at any time in the remaining graph, we add its edge to the matching. For a degree-2 vertex u, we create $P(u)$, a maximal even-length path containing u and only degree-2 vertices except for the first and last vertex in $P(u)$. Let l, r be these boundary vertices in $P(u)$. We use a union-find data structure to keep track of membership in components. At the start of the algorithm, every vertex is its own component. Now, if l and r already belong to the same component C, then adding the additional path must close a cycle in C. In that case we arbitrarily add one of the edges of u to the matching and the component is then cleared out by Rule-1 reductions. If no circle exists, then we simply update the labels of l, r to signal that their components have been joined together. No merge is performed at this point.

When there are no degree-2 vertices left, we merge the remaining acyclic components one by one. We refer to this as a *component shrink*. Let C be component with boundary vertices $b_1, \ldots, b_k$. To perform the shrink we only require $O(\sum_{b_i \in C} d_{b_i})$ since we only need to access

the adjacency list of each b_i once.

If during these merges, a new degree-2 vertex becomes available (this can happen if b_i and b_j are both adjacent to some vertex w with $d(w) = 3$ before the shrink) the merging stops and the degree-2 operation is applied, which can connect two components. If all components have been shrunk, and no degree-1 or degree-2 vertices exists, KS_{comp} selects a random edge to be matched.

Using alternating components can prove useful in the sense that they provide a structured way to perform reductions such that they avoid re-reading adjacency lists of vertices (which was also the motivation of KS_{cache}). For example, it takes only linear time for the worst case instance of Figure 1. However, it can degrade to the worst case performance of the basic implementation if there are no nontrivial alternating components in the graph:

THEOREM 3.3. *The KS_{comp} variant requires $O(n^2)$ in the worst case.*

Proof. The proof is similar to that of Thm. 3.2 hence we skip the details. In short, if the algorithm is given an input instance as in Fig. 2, there will be no non-trivial components. Assuming all random choices hit into the two-by-two complete bipartite subgraphs, there will always be a random choice following a single merge. Hence, on this instance, the component approach behaves just like the basic implementation. □

We note that the approach of using alternating components bears some similarity with the approach used in [3], which performs all *currently existing* Rule-2 operations in linear time. However, the authors consider solely Rule-2 and do not allow neither Rule-1 nor random decisions. They propose a complicated static linear-time algorithm, operating on the DFS-tree of a given graph and avoiding to do merges explicitly by doing implied reductions. Let us refer to a *phase*, as a series of consecutive Rule-1 or random operations done by KS. Hence, phases are separated from each other by one or more Rule-2 applications. Their $O(n + m)$ time algorithm would need to be rerun between the different phases and thus can similarly potentially yield a complexity of $O(n^2)$.

3.4 Fast recovery of the matching. As described above, a merge operation creates a new vertex that may be involved in upcoming merges. Hence, the matched vertices returned by KS do not necessarily appear in $G^{(0)}$. The standard way to recover the actual matching is by keeping a stack S containing all the information related to each merge [13]. After the heuristic, the stored merges are popped one after another from S.

Assume that during the heuristic a degree-2 vertex u appears with neighbors v and w; hence, a new vertex vw is created. While recovering the actual matching, if vw is unmatched we can match either v or w with u and continue. Otherwise, we need to check whether vw is matched with one of v's or w's neighbors stored in S. Using similar reasoning as in Theorem 3.1, this approach takes $O(m \log n)$ time and space in the worst case, assuming the smaller neighborhood (among v's and w's) is stored in the stack. Here, we propose an algorithm with $O(n)$ time complexity.

To recover the matching efficiently, we use a graph $F \subseteq G$. Initially, F has no edges but contains all vertices of G. During the heuristic, assume a merge is being performed due to a vertex having two edges e and e'. Let $\{u, v\}$ and $\{u', w\}$ be the original endpoints of e and e' in G, respectively. We add the edges $e_1 = \{u, v\}$ and $e_2 = \{u', w\}$ to F and set $twin(e_1) = e_2$ and $twin(e_2) = e_1$. The association of e_1 and e_2 as twins exploits the implicit Rule-2 property that either e_1 or e_2 can be in a maximum cardinality matching, but not both.

PROPOSITION 3.1. *F is a forest.*

Proof. Assume that F contains a cycle. Let $C \subseteq F \subseteq G$ be a simple cycle with length ℓ. Let C be first initiated with a merge in $G^{(t)}$, and let u be the degree-2 vertex for that merge with neighbors v and w. Both $\{u, v\}$ and its twin $\{u, w\}$ must be in C, since u will not exist in $G^{(t+1)}$. Let C' be the reduced cycle obtained by removing u from C and merging v and w. The above steps can be repeated by using the edge that initiates C'. At each step, the cycle will be further reduced and after $l-1$ steps, only two vertices and a single edge will remain. As parallel edges are not allowed in the graph $G^{(\cdot)}$, such a merge cannot exist to complete the cycle. □

Let M' be the set of matched edges found by KS and let M be the the matching constructed with the original versions of M''s edges from $G^{(0)}$. Let $e = \{u, w\}$ be an edge in M. Then for all edges $e' \neq e$ of u and $e'' \neq e$ of w in F, we add $e^* = twin(e')$ and $e^{**} = twin(e'')$ to M. We then remove u and w from F (i.e., $F = F \setminus \{u, w\}$) and consider the next matched edge in M. If no more nodes in the current tree are matched, we select a vertex of degree-1 in F, add its unique edge to M and repeat.

LEMMA 3.1. *The algorithm described above to recover the matching is correct.*

Proof. Let e and e' be two twin edges which have not been included in the matching M. If one of the endpoints of e is matched in M, then it is no longer

possible to include e in M. By Rule-2, one of the twins must necessarily participate in a maximum cardinality matching. Hence, we can extend M by using e's twin edge. This means that as long as there exist matched vertices in F, the algorithm correctly extends the matching.

If there are no matched vertices in F, for any twins e and e', we know that either can be in a maximum cardinality matching. Hence, we can arbitrarily add either to M. For simplicity, we chose the edge of a degree-1 node, which always exists since F is always a forest. □

4 Experiments

We implemented the algorithms discussed in the last section. Except where noted, the Karp–Sipser algorithm variant being used is the default one. When KS is used as a heuristic, we first generate a random permutation of the edges with uniform probability, which we store in a list. Then, in the event of a random decision, the first available edge (that is without any matched endpoints) is returned from this list.

All the codes are written in `C++` and compiled with `gcc 8.3.0` with `-O3` optimization flag. They are tested on two different machines; The first machine had 2 x AMD EPYC 7551 CPUs and 256 GB RAM (Arch 1). The other had 4 x Intel Xeon E7-8890 CPUs and 1.5 TB RAM (Arch 2).

4.1 Experiments with real graphs: For the real-life dataset, we used bipartite graphs with $n = |V_A| = |V_B|$ and $10^6 \leq n \leq 10^7$ from the SuiteSparse collection [5]. There were 93 such graphs with $3 \times 10^6 \leq m \leq 3.1 \times 10^8$. For each of these matrices, we performed a total of sixteen runs. More specifically, we randomly permuted each matrix four times, and then measured the timings for both KS_{R1} and the default KS algorithm four times in the permuted matrix. Permuting the matrix between runs can potentially affect the run-time for both KS heuristics as well as the exact algorithm used afterwards. To analyze the impact of adding Rule-2 for data reduction, two exact matching algorithms push-relabel (PR) [7] and Pothen-Fan+ (PF) [6, 16] are used. In previous studies [10], these two algorithms are shown to be the best performing algorithms from a set of alternatives, including the asymptotically fastest one [8].

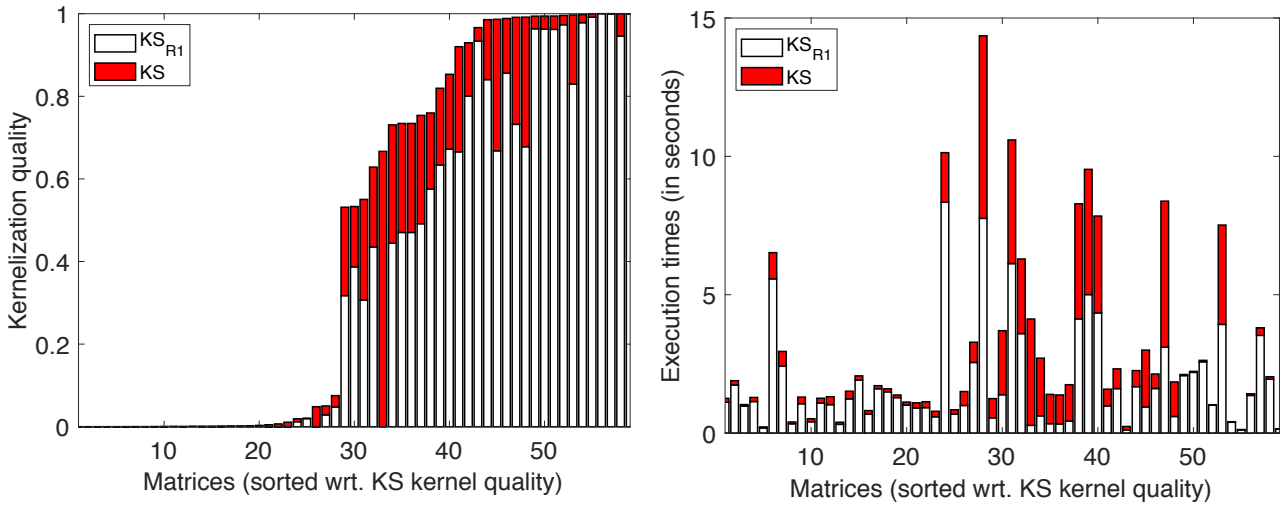
4.1.1 Using KS for kernelization: We first consider the use of KS as a kernelization tool and run experiments on Arch 1: we first run KS and KS_{R1}, with no random decisions, to find their kernels $G^{(k)}$ and $G^{(k')}$ where $k \geq k'$. After that, an exact algorithm, PR or PF, is executed to find a maximum cardinality matching. As KS requires a minimum degree ≤ 2, for this experiment, we only use the graphs having min degree one or two. There are 59 such graphs. The results are summarized in Figure 4.

To measure the impact of the second rule, we measured the kernel quality for each heuristic. The quality is computed as the number of times a rule is applied during KS and KS_{R1} normalized with respect to size of the maximum cardinality matching. Figure 4(a) shows that in terms of quality, for around 30 matrices, the second rule has an impact and for one matrix, it increased the quality from almost 0 to 0.65. Hence, KS obtains a kernel of smaller size compared to KS_{R1}. Furthermore, as Fig. 4(b) shows, KS is slightly slower compared to KS_{R1} differing in most graphs by less than a second.

To evaluate the impact of additional size reduction obtained via Rule 2 on the exact matching process, we measured the execution times of PR and PF given the kernels $G^{(k)}$ and $G^{(k')}$ obtained by KS and KS_{R1}. Figure 4(c) shows the speedup obtained, i.e., the execution time of PR/PF on $G^{(k')}$ divided by the execution time of the same algorithm on $G^{(k)}$. In the figure, if both PR and PF speedups are positive (negative) for the same matrix the larger (smaller) one is shown on top of the other. The first observation is that the speedups are mostly in the positive side and a good kernel can significantly improve the execution time of the matching phase. The second observation is that the impact of kernelization heavily depends on the algorithm used since PR and PF can behave in a different way. For instance, in one matrix, although $G^{(k)}$ is smaller than $G^{(k')}$, PF runs slower on $G^{(k)}$. Yet, even with such fluctuations, for both algorithms, smaller kernels usually yield better performance.

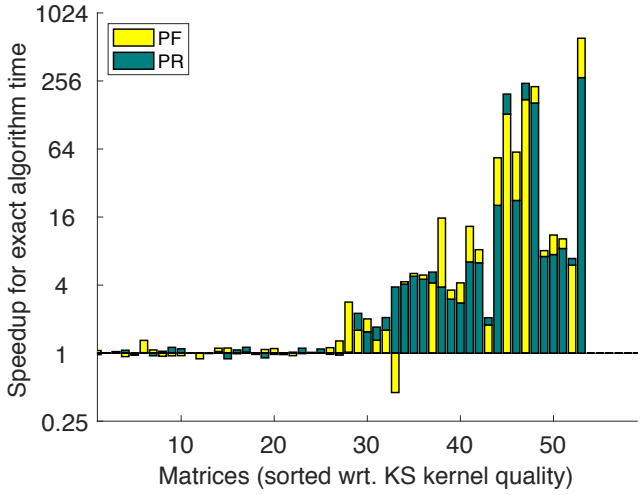
4.1.2 Using KS as a heuristic: To check if KS with Rule-2 is also useful as an initial, cheap matching heuristic, we let KS and KS_{R1} run in their entirety and input the returned matchings to the exact algorithms. The experiments are performed on Arch 2. Figure 5 summarizes the results of these experiments. Both KS_{R1} and KS obtain excellent results with matching quality almost always larger than 0.97 of $\mathcal{M}(G)$. KS is able to match 1% more vertices than KS_{R1}, which corresponds to more than ten thousand vertices due to the size of our instances.

As in the previous experiments, the exact algorithms tend to be faster when initialized with KS. However, since the impact, i.e., the difference in the quality, is higher for kernelization compared to matching, the speedups are not as large as the previous set of experiments. If we look at the total run time to find a maxi-



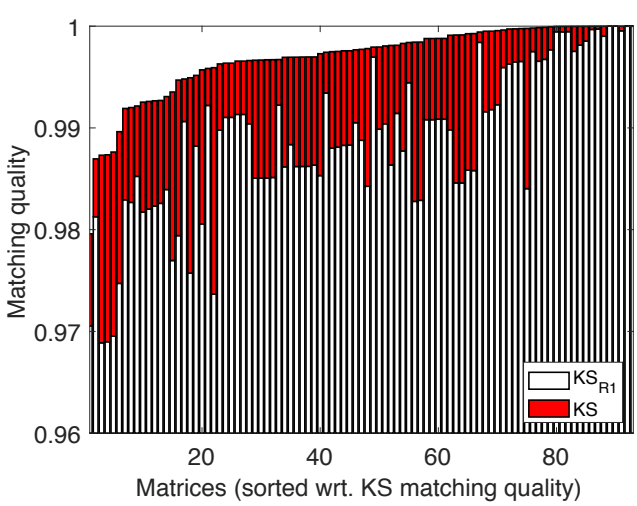
(a) Kernel quality: $\#rules/\mathcal{M}(G)$

(b) Execution times of KS_{R1} and KS in seconds

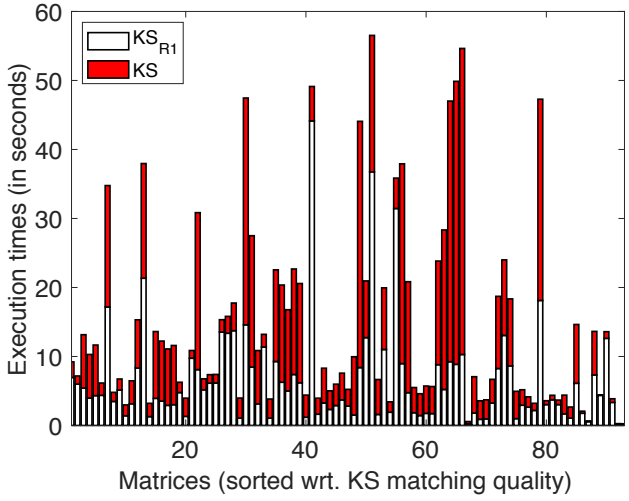


(c) Speedups for PF/PR when the KS kernel is used instead of KS_{R1} kernel

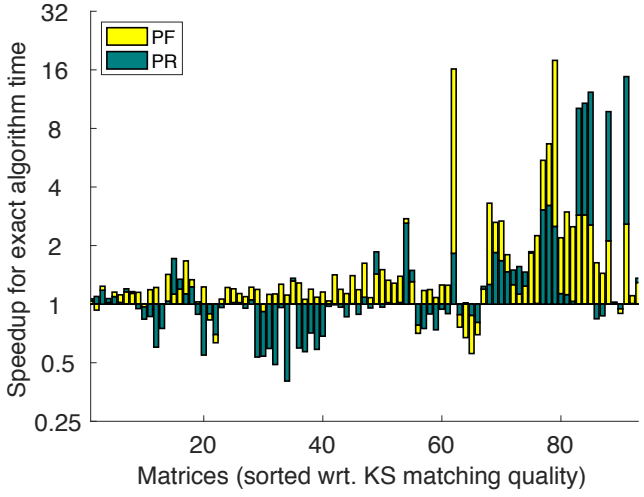
Figure 4: (a) The kernel quality of KS and KS_{R1} for the 59/93 graphs having at least a single vertex with degree strictly less than three. The quality is measured as the number of rules applied during a heuristic normalized with respect to size of the maximum cardinality matching. (b) The execution times (in sec.) of these two heuristics are also shown. (c) The speedups for the exact matching algorithms PR and PF when the KS kernel is used instead of KS_{R1}. A value higher than one is in favor of Rule 2 implementation. In all figures, the x-axis, i.e., matrices, is organized with respect to non-decreasing kernel quality for KS.



(a) Matching quality: $\#decisions/\mathcal{M}(G)$



(b) Execution times of KS_{R1} and KS in seconds



(c) Speedups for PF/PR when KS is used as the initial heuristic instead of KS_{R1}

Figure 5: (a) The matching quality of KS and KS_{R1} for all 93 graphs. The quality is measured as the number of decisions, including random ones, applied during a heuristic normalized with respect to size of the maximum cardinality matching. (b) The execution times (in sec.) of these two heuristics are also shown. (c) The speedups for the exact matching algorithms PR and PF when KS is used instead of KS_{R1}. A value higher than one is in favor of Rule 2 implementation. In all figures, the x-axis, i.e., matrices, is organized with respect to non-decreasing matching quality for KS.

mum cardinality matching, we see that on average using KS rather than KS_{R1} can yield a 20% slow-down using PF and 40% with PR. This is understandable since KS itself does significantly more work than its simpler variant. On the other hand there exist instances where using KS rather than KS_{R1} can be crucial. For example, consider the graph `com-LiveJournal` for which PF, initialized with KS_{R1}'s matching, results in a run time of about 1060 seconds on average, whereas PF requires 65 seconds on average, if initialized with KS. That is why we are also interested in parallel implementations of Rule-1 and Rule-2 which will potentially bring the performance of KS closer to that of KS_{R1} and hence reap more benefits from the speedups seen in Figure 5(c).

4.2 Experiments with synthetic datasets: Here we focus on special, synthetic instances, for which Rule-2 increases the probability of finding a maximum cardinality matching. Similar examples have also been described in [1].

4.2.1 Optimal instances for KS: The first dataset is a family of synthetic graphs where Rule-2 is sufficient to find a maximum cardinality matching. Let $G = (V_A \cup V_B, E)$ be a bipartite graph with $|V_A| = |V_B| = n$. For any $1 \leq i \leq j \leq n$, there is the edge $\{a_i, b_j\}$ in G where $a_i \in V_A$ and $b_j \in V_B$. The edges $\{a_2, b_1\}$ and $\{a_n, b_{n-1}\}$ also exist.

In the graph, the vertices $a_{n-1}, a_n \in V_A$ and $b_1, b_2 \in V_B$ have degree equal to two. Assume without loss of generality that we apply Rule-2 on b_1, to merge a_1 and a_2. Then in the reduced graph, a_2a_1 will be a degree-1 vertex hence Rule-1 can now be applied. We continue with Rule-1 until four vertices remain; applying Rule-2 followed by Rule-1 yields a perfect matching. Hence KS with Rule-2 always find a perfect matching for these synthetic graphs. Furthermore, since Rule-2 is applied only twice, KS requires linear time. If however, only Rule-1 reductions are allowed, no reduction is possible at first and KS_{R1} will immediately resort to random decisions. In Table 1, we provide the average quality returned by KS_{R1} for different values of n as well as run times. As can be seen, there is 25% quality difference in favor of KS. Interestingly, KS_{R1} requires slightly more time than KS. This can be explained by the fact that due to the random decisions, during most steps KS_{R1} needs to iterate over two adjacency lists, while KS needs to iterate over only one, due to it applying Rule 1. Furthermore, the cost of both exact algorithms increases with n.

4.2.2 2-out graphs: The random 2-out graphs are created by initially considering an empty bipartite graph. Then each vertex of the graph randomly picks two other vertices with uniform probability from the the other side. A 2-out graph therefore has approximately $4n$ edges (in case of parallel edges, only one is kept). Walkup [17] showed that it is almost certain that such graphs contain a perfect matching.

By construction, each vertex in a 2-out graph has degree at least two and KS_{R1} will resort to random decisions immediately. This is not the case however for KS. In expectation, a random 2-out graph will initially have about $\frac{2n}{e^2}$ degree-2 vertices. Hence Rule-2 can potentially be applied a significant amount of times before the graph runs out of degree-2 vertices. Consequently, because it has a smaller kernel and can apply its rules easier, KS will have to make random decisions less frequently than KS_{R1} and therefore has less of a chance to make an erroneous decision.

In Table 2, we compare KS_{R1} and KS on varying sizes of 2-out graphs and show the percentage of tests where the two algorithms found a perfect matching. For each $n \in \{10000, 25000, 50000\}$, we generate 20 different random 2-out graphs. Each distinct random 2-out graph is also used four times to allow alternative random decisions on different runs. As the table shows, KS_{R1} is never able to find a perfect matching on such graphs. On the other hand, KS is able to find one in a significant percentage of the trials. Nonetheless, both algorithms output matchings of high quality. i.e., greater than 0.99. In addition, both KS and KS_{R1} require less than a second in all of the instances.

4.2.3 Experiments with worst-case inputs: In the results with the real graphs for Section 4.1, it was

	KS_{R1}				KS	
n	quality	time	PF	PR	quality	time
7,500	0.75	1.01	1.29	1.30	1	0.73
10,000	0.75	1.95	2.86	2.57	1	1.46
15,000	0.74	5.61	7.48	6.37	1	4.48
20,000	0.74	10.34	17.08	12.37	1	8.82

Table 1: The average performance on the family of graphs referenced in Section 4.2.1 for $n \in \{7500, 10000, 15000, 20000\}$. For these graphs, KS finds a perfect matching. The PF and PR timings do not include the time needed for KS_{R1}.

n	KS_{R1}	KS
10,000	0	0.68
25,000	0	0.64
50,000	0	0.56

Table 2: The ratio of tests for which KS_{R1} and KS find a perfect matching in random 2-out graphs.

n	KS_{R1}	KS	KS_{cache}
40,000	0.00	3.44	0.04
80,000	0.01	15.32	0.11
160,000	0.03	64.53	0.22
320,000	0.10	265.89	0.48

Table 3: The run time in seconds for KS_{R1}, KS, and KS_{cache} in the worst-case instance shown in Figure 1 for $n \in \{40000, 80000, 160000, 320000\}$.

observed that KS behaves in practice similar to a linear time algorithm. For this reason, the various modifications discussed in Section 3 in most cases did not have a noticeable positive impact on the performance.

However, we do note that our modifications can indeed be useful and we demonstrate in Table 3 the performance for various values of n for the instance in Figure 1. For simplicity, we provide only results with KS_{cache} but an implementation using a hash table, and KS_{comp} both had equivalent performance. As can be seen, while KS_{R1} and KS_{cache} require in all instances less than a second, KS struggles as n increases. For example, it requires over four minutes in the last case. This shows that in certain situations the sub-quadratic algorithms of Subsection 3.1 might be crucial due to their reduced worst-case behavior. Similarly the other algorithms in Section 3 might prove useful as they can avoid some pitfalls that harm the performance of KS.

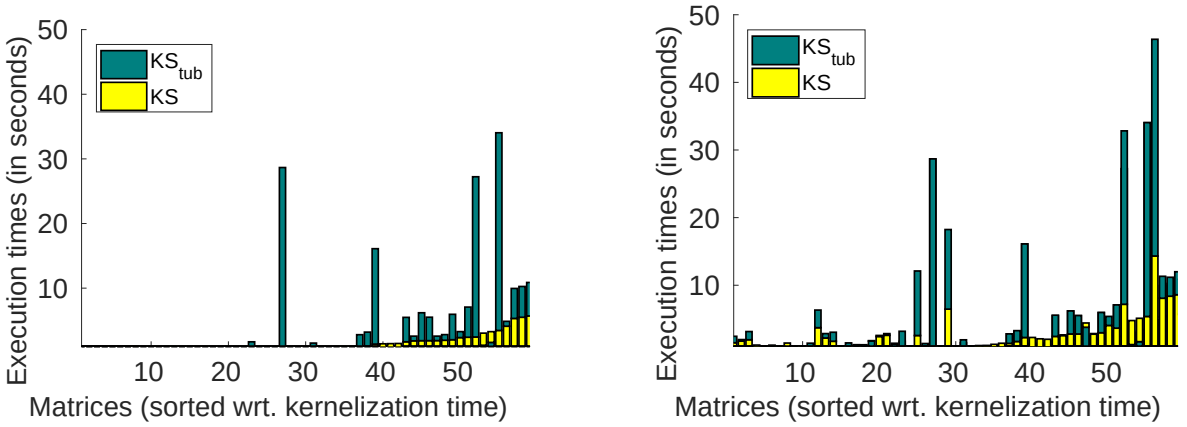
4.3 Experiments with another implementation: Korenwein et al. [11] provide an implementation in C++ which finds the kernel $G^{(k)}$ of a given undirected graph G. We use KS_{TUB} to denote this code. The code KS_{TUB} works along the same lines as our kernelization code. Initially, Rule-1 or Rule-2 are applied for as long as it is possible, and the kernel $G^{(k)}$ is created. Then, the vertices in $G^{(k)}$ are renumbered to be from 1 until $|V_{G^{(k)}}|$. This smaller representation of the kernel is then given as input for the exact matching algorithm. The code KS_{TUB} was compiled according to the instructions.

We now list some of the differences between our implementation and KS_{TUB} before giving experimental results. One important difference between our implementation and KS_{TUB} is that we prioritize Rule-1 over Rule-2. In KS_{TUB}, vertices of degree one and two are stored together in a container, and the appropriate reduction rule is executed when a vertex is accessed from this container, depending on its degree. As a consequence, KS_{TUB} might apply Rule-2 more frequently, which can harm the performance. Indeed, consider an extension of the graph shown in Figure 1, in which the two leftmost vertices are connected with an additional degree-1 vertex each. Then, by applying Rule-1 on one of the new vertices, it becomes possible to find the kernel in linear time. If, however, the application of Rule-1 is postponed in favor of Rule-2, then the worst case run time becomes quadratic. The code KS_{TUB} also uses linked lists which are not as cache-friendly as our vector-based implementation. Furthermore, our implementation also returns a matching for G, whereas KS_{TUB} returns only the kernel and does not return any information about which edges or vertices of G have been involved in the Rule-2 reductions. Hence, it is not possible to extend the matching found in the kernel $G^{(k)}$ to a maximum cardinality matching for G using KS_{TUB} as is.

We now give the run time comparison of the two codes. The experiments were conducted on Arch 1. For the sake of controlled experimentation, we made the two codes have the same permutation of the input graphs. KS_{TUB} normally prints output; we removed this part for fairness. In Figure 6, we compare the two codes on the bipartite graphs used in Section 4.1.1. For each implementation, we report the minimum time observed over four runs. In both implementations, we measured the time to apply Rule-1 and Rule-2 and the time needed to renumber the vertices in $G^{(k)}$. In addition, for our implementation, we included additional costs such as the initial cost required to create the adjacency list representation of G and as well as the time needed by the algorithm of Section 3.4 to find a maximum cardinality matching of the original graph (by using and expanding the one on the kernel to comply with the Rule-2 applications). As can be seen, our implementation is almost in all cases faster than KS_{TUB} and in some situations significantly so. In regards to the time required solely for the purpose of kernelization, shown in Figure 6(a), we see that our implementation is about 3.19 times faster on average than KS_{TUB}. The maximum time our code requires is 5.8 seconds, whereas KS_{TUB} exceeds 10 seconds on six instances. The difference in speed becomes more noticeable in instances such as `kron_g500-logn21`, where our implementation takes 3 seconds, while KS_{TUB} takes 34 seconds.

The difference needed to renumber the vertices in the kernel can similarly be as striking as can be observed in Figure 6(b). For example, on the bipartite graph `Com-Orkut`, our code to renumber vertices takes 5.6 seconds, whereas KS_{TUB} requires 41 seconds. We estimate that this significant difference derives from the fact that KS_{TUB} creates an entirely new class object, for which it then has to initialize a number of linked lists (one per vertex). In contrast, our code needs to initialize only four arrays as $G^{(k)}$ is recreated in the standard CSR/CSC formats of sparse matrices to be given as input to the exact algorithms.

Finally, looking on the overall time, again in Fig-



(a) Comparison only for kernelization between KS and KS_{TUB}.

(b) Overall time comparison between KS and KS_{TUB}.

Figure 6: Comparison of our implementation with KS_{TUB} [11] on the graphs of Section 4.1.1. Figure 6(a) gives only the time needed to find the kernel by applying Rule-1 and Rule-2. Figure 6(b) gives the overall time by adding the time needed to renumber the vertices of the kernel $G^{(k)}$. In addition, for our implementation Figure 6(b) also includes the time required to recover the maximum matching using the algorithm of Section 3.4, and the time to create the adjacency list representation for G.

ure 6(b), our implementation is about 2.17 times faster than KS_{TUB} even with recovery of a maximum cardinality matching for the initial graph. Our implementation never exceeds 15 seconds in overall time, whereas there exist six graphs where KS_{TUB} exceeds this time bound.

5 Related work

Early work by Möhring and Müller-Hannemann [15] popularized the use of Karp-Sipser as the name for KS_{R1}, and later work [6, 10, 12] followed suit. To the best of our knowledge, there are only a few studies focusing on implementing KS as in its original, proposed form. Magun [13] studied the cardinalities of matchings produced by KS, KS_{R1}, and other extensions in random graphs. The paper claims linear running time for all heuristics, but from the pseudocode and description in the paper, it is not clear whether the reductions are implemented in a way that is equivalent to ours. Analysis of the source code however does suggests that the implementation is equivalent and hence can similarly require quadratic time. The paper also does not study practical running time. Mertzios et al. [14] apply the first reduction rule and a restricted version of the second rule described before in linear time to obtain a linear-size kernel with respect to a graph parameter (known as feedback edge number) for computing maximum cardinality matchings in undirected graphs. A cubic kernel on bipartite graphs with the same reduction rules is also shown. Korenwein et al. [11] discuss extensions of Karp–Sipser reduction rules for the maximum weighted matching problem. Their results indicate that kernelization is less successful in the weighted case in comparison to the unweighted case. Bartha and Krezs [3] discuss a linear time algorithm when only the second rule of Karp–Sipser is considered. However, using this algorithm does not result in a linear time matching heuristic. Furthermore, several theoretical papers [2, 4] analyze the expected matching quality of the Karp-Sipser algorithm and derived variants for sparse random graphs.

6 Conclusion

We have investigated two data reduction rules for the maximum cardinality matching problem proposed by Karp and Sipser [9]. While the first rule has a simple, worst case linear time implementation, the second rule can take quadratic time. We have focused on the second rule which merges the two neighbors of a degree-2 vertex. We have considered and analyzed three different algorithms with different levels of sophistication, and proposed an efficient algorithm to recover the matching in the original graph. For two of the these algorithms, we showed that their worst-case performance can still be quadratic in terms of n, whereas the third approach has sub-quadratic complexity in sparse graphs. On a set of experiments with real life, random, and constructed problem instances we showed that the second rule

indeed increases the cardinality of the matching found by Karp–Sipser and can lead to a drop in the run time of the exact algorithm afterwards.

One open question is whether a linear time implementation for the second rule is actually possible. As discussed in the text, it seems that the approach of Bartha and Krezs [3] does not achieve a linear time complexity when degree-2 reductions are interleaved with degree-1 reductions or random choices, made in the matching context. With these, we conjecture that the answer to our question is negative. We are also interested in a parallel version of the Karp–Sipser algorithm.

References

[1] M. Anastos and A. Frieze. Finding perfect matchings in random cubic graphs in linear time. *arXiv preprint arXiv:1808.00825*, 2018.

[2] J. Aronson, A. M. Frieze, and B. G. Pittel. Maximum matchings in sparse random graphs: Karp-sipser revisited. *Random Struct. Algorithms*, 12(2):111–177, 1998.

[3] M. Bartha and M. Kresz. A depth-first algorithm to reduce graphs in linear time. In *11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 273–281, Sep. 2009.

[4] P. Chebolu, A. M. Frieze, and P. Melsted. Finding a maximum matching in a sparse random graph in $O(n)$ expected time. *J. ACM*, 57(4):24:1–24:27, 2010.

[5] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1, 2011.

[6] I. S. Duff, K. Kaya, and B. Uçar. Design, implementation, and analysis of maximum transversal algorithms. *ACM Transactions on Mathematical Software*, 38(2):13, 2011.

[7] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, October 1988.

[8] 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.

[9] 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.

[10] K. Kaya, J. Langguth, F. Manne, and B. Uçar. Push-relabel based algorithms for the maximum transversal problem. *Computers & Operations Research*, 40(5):1266–1275, 2013.

[11] V. Korenwein, A. Nichterlein, P. Zschoche, and R. Niedermeier. Data reduction for maximum matching on real-world graphs: theory and experiments. *arXiv preprint arXiv:1806.09683*, 2018.

[12] J. Langguth, F. Manne, and P. Sanders. Heuristic initialization for bipartite matching problems. *J. Exp. Algorithmics*, 15:1.3:1.1–1.3:1.22, 2010.

[13] J. Magun. Greeding matching algorithms, an experimental study. *J. Exp. Algorithmics*, 3, September 1998.

[14] G. B. Mertzios, A. Nichterlein, and R. Niedermeier. The power of linear-time data reduction for maximum matching. In *42nd International Symposium on Mathematical Foundations of Computer Science (MFCS 2017)*, volume 83 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 46:1–46:14, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[15] R. H. Möhring and M. Müller-Hannemann. Cardinality matching: Heuristic search for augmenting paths. Technical report, Technische Universität Berlin, 1995.

[16] A. Pothen and C.-J. Fan. Computing the block triangular form of a sparse matrix. *ACM Transactions on Mathematical Software*, 16(4):303–324, 1990.

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