Heuristic Initialization for Bipartite Matching Problems

JOHANNES LANGGUTH and FREDRIK MANNE
University of Bergen
and
PETER SANDERS
Karlsruhe University of Applied Sciences

It is a well-established result that improved pivoting in linear solvers can be achieved by computing a bipartite matching between matrix entries and positions on the main diagonal. With the availability of increasingly faster linear solvers, the speed of bipartite matching computations must keep up to avoid slowing down the main computation. Fast algorithms for bipartite matching, which are usually initialized with simple heuristics, have been known for a long time. However, the performance of these algorithms is largely dependent on the quality of the heuristic. We compare combinations of several known heuristics and exact algorithms to find fast combined methods, using real-world matrices as well as randomly generated instances. In addition, we present a new heuristic aimed at obtaining high-quality matchings and compare its impact on bipartite matching algorithms with that of other heuristics. The experiments suggest that its performance compares favorably to the best-known heuristics, and that it is especially suited for application in linear solvers.

Categories and Subject Descriptors: G.2.2 [**Graph Theory**]: Graph Algorithms; F.2 [**Analysis of Algorithms and Problem Complexity**]; F.2.1 [**Numerical Algorithms and Problems**]: Computations on Matrices

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Heuristics, matching

ACM Reference Format:

Langguth, J., Manne, F., and Sanders, P. 2010. Heuristic initialization for bipartite matching problems. ACM J. Exp. Algor. 15, Article 1.3 (February 2010), 22 pages.
DOI = 10.1145/1712655.1712656 http://doi.acm.org/10.1145/1712655.1712656

Author's address: J. Langguth (contact author) and F. Manne, University of Bergen, Department of Informatics, Thormhlensgate 55, N-5008 Bergen, Norway; email: Johanes.langguth@ii.uib.no; P. Sanders, Karlsruhe University of Applied Sciences, Institute for Theoretical Computer Science, Am Fasanengarten 5, 76131 Karlsruhe, Germany.

1. INTRODUCTION

The bipartite cardinality matching problem is defined as follows:

Given an undirected bipartite graph $G = (V_1, V_2, E), E \subseteq V_1 \times V_2$, find a maximum cardinality subset $M \subseteq E$ of pairwise nonadjacent edges. This set M is called a perfect matching if $|V_1| = |V_2| = |M|$. Clearly, not all bipartite graphs have a perfect matching.

The bipartite matching problem is a classical topic in combinatorial optimization that has been studied for almost a century. It has many applications [Ahuja et al. 1993] in various fields, and it is relevant for numerical computations due to the fact that it can be used to compute improved pivoting strategies in linear solvers [Schenk et al. 2003; Sangiovanni-Vincentelli 1976], making it an important problem in combinatorial scientific computing.

To that end, for a given $n \times n'$ matrix A, we define $G_A = (V_1, V_2, E)$, $|V_1| = n$, $|V_2| = n'$, $E = \{\{i, j\} \in V_1 \times V_2 : a_{i,j} \neq 0\}$, $|E| = m$ as the graph derived from A. Since we will be dealing with square matrices that do not contain all-zero rows or columns, G_A will have a perfect matching as long as A has full structural rank.

As any edge included in such a matching corresponds to an entry of A being permuted to the main diagonal, any perfect matching describes a pivot strategy that is optimal in the Brayton's sense, that is, Gaussian elimination performs the minimum number of arithmetic operations. This requires pivot elements to be nonzero, since otherwise a computation is performed to compute something which is always zero [Brayton et al. 1970; Sangiovanni-Vincentelli 1976].

Since the problem is easy for dense matrices, we restrict our attention to sparse instances only. Note that a solution can still be improved upon by increasing the weight of the matched edges. This can be achieved either by directly solving the considerably more difficult weighted perfect matching problem or by improving a perfect matching through repeated searches for weight augmenting cycles. In both cases, heuristic initializations can speed up the exact computations.

It is a well-known fact that most matching algorithms, with the exception of the Push-relabel algorithm [Cherkassky et al. 1998], can be accelerated by initializing them with an approximate matching. However, fast exact algorithms are available. One of the best theoretical bounds is $O(n\sqrt{mn/\log n})$ for the algorithm by Alt et al. [1991]. Furthermore, experimental studies have shown that the running times of many exact algorithms increase only slightly more than linearly with instance size for most test sets [Setubal 1996].

Thus, all initializations should have linear running time, which generally restricts them to local selection rules that are applied to all vertices. It is easy to see that the global Hall condition [Korte and Vygen 2006], which is necessary and sufficient for the existence of a perfect bipartite matching cannot be guaranteed to be fulfilled by such a local selection rule. Therefore, it is unlikely that a linear time exact algorithm for matching exists.

In this article, we analyze the effects of various initializations on the total running times of several exact algorithms. Our results show that these effects can be quite dramatic. Even the Push-relabel algorithm can be accelerated on many instances. In addition, we present a new initialization heuristic that is geared towards application in linear solvers and compare it with the known heuristics in regard to performance, solution quality, and usefulness as an initialization routine. We provide an overview of known matching heuristics in Section 2. Section 3 introduces our new approach, and we present our experimental results in Section 4, followed by our conclusions in Section 5. The appendix contains the results from our experiments.

2. MATCHING HEURISTICS

Matching heuristics have been studied for more than 30 years. Here, we give an overview of the most commonly known heuristics. They serve as a basis for comparison in our experiments. Although we focus only on bipartite graphs, these heuristics are suited for general graphs as well.

2.1 Simple Greedy

Probably the simplest and most widely used selection rule is Simple Greedy: For each vertex, find an unmatched neighbor and add the respective edge to the matching. If all neighbors are matched, leave the vertex unmatched. Even though its approximation ratio is only $\frac{1}{2}$, studies have shown that Simple Greedy usually produces solutions within at least $\frac{9}{10}$ of the optimum [Möhring and Müller-Hannemann 1995; Magun 1998]. Its worst-case running time is $O(m)$, and the expected running time is $O(n)$ if vertices are picked in random order.

2.2 Dynamic Mindegree

More sophisticated approaches exploit the fact that generally, the chance to match a vertex in such a way that one of its neighbors can no longer be matched is lower if vertices are preferably matched to vertices of low degree. This observation gives rise to the family of degree-based heuristics known as *Mindegree*. Their basic concept works as follows:

The algorithm determines the degree of each vertex and repeatedly picks one vertex of minimum degree and an unmatched neighbor to be matched to it. The matched edge and its incident vertices are removed from the graph and all of their neighbors have their degrees reduced by one, possibly changing the order in which further vertices are picked. The process is repeated until no further vertex can be matched. A natural approach for implementation would be to use a priority queue to keep track of the lowest degree vertices.

Depending on whether the neighbor is chosen randomly or also chosen by minimum degree, we speak of One-Sided Mindegree or Double-Sided Mindegree. Many variants of this scheme are possible.

2.3 Static Mindegree

This heuristic differs from Dynamic Mindegree in so far that after deleting a vertex, the degree of its neighbors is simply not updated. This saves a considerable amount of work, since each deletion of an edge $\{u, v\}$ now takes $O(1)$

instead of $O(deg(u) + deg(v))$ time, where $deg(v)$ is the degree i.e., the number of neighbors of the vertex v. However, the speedup that is achieved in this way comes at a high cost in terms of solution quality [Setubal 1996; Magun 1998]. Like Dynamic Mindegree, Static Mindegree can be implemented either one- or double-sided, but due to the fact that degrees are not updated, searching for a neighbor of minimum degree promises a smaller increase in solution quality in comparison to Double-Sided Dynamic Mindegree, at the same cost in running article. Thus, only One-Sided Static Mindegree will be treated in this article.

2.4 Limited Mindegree (Karp-Sipser)

It is also possible to restrict the Mindegree approach to vertices of low degree. In this case, if the lowest degree among all vertices in the graph is above a certain limit, the next vertex to be matched is selected randomly instead of picking a minimum degree vertex. If the lowest degree is below the limit, however, the next vertex is selected according to the Mindegree rule. This obviates the need of keeping track of the degree of many vertices while running the heuristic. However, this does not simplify the implementation significantly, and unlike Static Mindegree, only saves a constant amount of effort per vertex. The most notable application of this strategy is the Karp-Sipser heuristic [Karp and Sipser 1981], which only keeps track of vertices of degree 1. If there are no such vertices in the current graph, it behaves like Simple Greedy. Thus, it can be described as One-Sided Dynamic Mindegree limited to degree 1. As shown in the next sections, this is a very effective strategy. It is unclear whether limits other than 1 have any practical application. Preliminary experiments did not show improvements in running time in comparison to full (i.e., unrestricted Mindegree).

2.5 Initialization Considerations

All these techniques represent a trade-off between speed and expected solution quality. This has been studied before (see Magun [1998] and Möhring and Müller-Hannemann [1995]), but for our application of these heuristics as a preliminary step in exact algorithms, we face the additional difficulty that we are ultimately not interested in the performance or solution quality of the heuristics as such, but in the effect they have on the total running time of the algorithm for which we use them as initializations.

As this is not directly proportional to the quality of the initialization, but largely dependent on the exact algorithm used, we have to compare pairings of heuristics and exact algorithms for a proper analysis.

The heuristics discussed here all have in common that they return maximal matchings, which implies that they are at least $\frac{1}{2}$-approximation algorithms. However, for practical purposes the approximation guarantees yield little information. Instead, we consider the actual average quality, which is usually far better than $\frac{1}{2}$ or $\frac{2}{3}$. Therefore, we ignore the approximation ratios of the algorithms discussed here and refer to them as heuristics and not approximation algorithms.

3. THE ALTERNATING COMPONENT HEURISTIC

In this section, we present a new cardinality matching heuristic for sparse bipartite graphs. Our heuristic is based on the Double-Sided Dynamic Mindegree heuristic enhanced by a special treatment for vertices of degree 2, which is crucial for the improved solution quality. Unlike other heuristics, it is designed for bipartite graphs only. It can be used on general graphs, but for those it cannot be expected to produce better solutions than Double-Sided Dynamic Mindegree. Thus, in the following, we assume a bipartite graph $G = (V_1, V_2, E), E \subseteq V_1 \times V_2$.

3.1 Degree-Based Reductions

Similar to all other Dynamic Mindegree heuristics, vertices of higher degree are only considered after all vertices of degree 1 have been matched. Karp and Sipser [1981] show that it is always safe to do so and label this a degree 1 reduction. Denote by $\Gamma(u)$ the neighborhood of u, that is, the set of vertices adjacent to u. For a set of vertices, S let $\Gamma(S) = \bigcup_{v \in S} \Gamma(v)$.

Theorem 3.1 (Degree 1 Reduction [Karp and Sipser 1981]). *Given a graph $H = (V, E)$ and vertices $u, w \in V$ with $deg(u) = 1$ and $\Gamma(u) = \{w\}$, then there is always a maximum matching M on H with $\{v, w\} \in M$*

Obviously, if $deg(u) = 1$ and a perfect matching exists, unlike heuristics such as Simple Greedy, a Dynamic Mindegree heuristic cannot create a suboptimal matching by selecting the neighbor w of u to be matched with a different vertex. For $deg(u) = 2$, we can exploit the fact that one out of the two incident edges must be used, as shown in the next reduction by Karp and Sipser [1981].

Theorem 3.2 (Degree 2 Reduction [Karp and Sipser 1981]). *Given a graph $H = (V, E)$ and vertices $u, w_1, w_2 \in V$ with $deg(u) = 2$ and $\Gamma(u) = \{w_1, w_2\}$, then there is always a maximum matching M on H with $\{u, w_1\} \in M$ or $\{u, w_2\} \in M$.*

Using this reduction, it is easy to find a matching in H by first finding a matching in $H' = (V', E')$ where $V' = V \cup \{w'\} \setminus \{u, w_1, w_2\}$ and $\Gamma(w') = \{\Gamma(w_1) \cup \Gamma(w_2)\} \setminus \{u\}$. If w' is matched in H', then u is matched to either w_1 or w_2 in H and the number of unmatched vertices in H' and H is the same. On the other hand, if w' is not matched in H', then u can be matched to either w_1 or w_2 in H. Again, the number of unmatched vertices in H' and H is the same. Thus, a maximum matching in H' implies a maximum matching in H. For a complete proof, see Karp and Sipser [1981].

3.2 Alternating Components

The Mindegree heuristic always performs degree 1 reductions automatically. However, implementing degree 2 reductions in an explicit, straightforward way does not suggest itself for high performance, as it requires either a flexible and thereby slow data structure or additional running time to search through the edges of the vertices to be merged. Therefore, we propose Algorithm 1, which exploits these reductions implicitly. As a first step, we introduce the concept of alternating components.

Algorithm 1. Component-Based Heuristic

```
 1: Set $l(v) := v \ \forall\ v \in V_1, V_2$
 2: Keep vertices sorted by degree in a priority queue $PQ$
 3: Pick an unmarked vertex $v$ of minimum degree from $PQ$. If $PQ = \emptyset$ output $M$ and
    stop.
 4: **if** $deg(v) = 1$ **then**
 5:     add $\{v, w\}$ to $M$, remove $v$ and $w$ (and their incident edges) from $G$
 6:     **goto line 3**
 7: **end if**
 8: **if** $deg(v) = 2$ **then**
 9:     Find a maximal path $P$ containing $v$ with $deg(w) = 2 \ \forall\ w \in P$
10:     Mark all $w \in P$
11:     **if** $P$ is a cycle **then**
12:         select a vertex $w \in \Gamma(v)$ from $G$
13:         add $\{v, w\}$ to M, remove $v$ and $w$ (and all incident edges) from G
14:     **else**
15:         **if** $P$ is of even length **then**
16:             **if** $l(u) = l(u')$ for $u, u' \in \Gamma(P) \setminus P$ **then**
17:                 select a vertex $w \in \Gamma(v)$ from $G$
18:                 add $\{v, w\}$ to M, remove $v$ and $w$ (and all incident edges) from G
19:             **else**
20:                 $l(u) := l(u')$
21:             **end if**
22:         **end if**
23:     **end if**
24:     **goto line 3**
25: **end if**
26: **if** $deg(v) > 2$ **then**
27:     select a vertex $w \in \Gamma(v)$ from $G$
28:     add $\{v, w\}$ to M, remove $v$ and $w$ (and all incident edges) from G
29:     **goto line 3**
30: **end if**
```

Definition 3.3. Let P be the subgraph of G formed by the union of all paths in G that have two distinct endpoints of degree greater than 2, even length, and contain only interior vertices of degree 2. We call a connected component A of P an alternating component.

Now, our heuristic works as follows: Upon initialization, sort all vertices in G into a priority queue according to their degrees and maintain this queue until it is empty and the heuristic finishes. Vertices of degree 0 are immediately dropped from the queue.

Furthermore, all vertices of degree 2 are initialized as unmarked. Following the Mindegree heuristic, degree 1 vertices in the graph are matched and successively removed from G along with their neighbors. As soon as no degree 1 vertices exist in the current graph, check for unscanned degree 2 vertices. To scan such a vertex, we consider the degrees of both its neighbors. If a neighbor has degree 2 and is unscanned, we scan it in the same manner.

Otherwise, we have found an endpoint of degree greater than 2. If the search fails to find two vertices of degree greater than 2, we have discovered a simple cycle. Because G is bipartite, the cycle has even length, and we can match it immediately by arbitrarily choosing one of its edges. After matching this edge, or any other edge on the cycle, a degree 1 vertex appears on the cycle. Its incident edge, which must be part of the cycle, is matched immediately by the heuristic. In this manner, the entire cycle will be matched.

If the search finds two vertices of degree greater than 2, we label both endpoint vertices as belonging to the same component, provided the path between them is of even length. In order to keep track of which endpoints belong to the same alternating component, we use a union-find data structure. If both vertices already belong to the same component, we have also found a cycle. Again, it can be matched immediately as described earlier in the text. Note that we do not label vertices that are connected by a path of odd length.

If all degree 2 vertices in the current graph are scanned, the heuristic matches vertices of degree three and higher according to the Double-Sided Mindegree heuristic. Since matched vertices are removed from the graph, the degree of their neighbors decreases. This means that new vertices of degree 1 or 2 will appear, which are matched or scanned, as described earlier in the text. Thus, until we obtain a maximal matching existing components can grow, become cyclic, and, therefore, become matched.

Note that our Mindegree selection rule skips vertices of degree 2, but other vertices can be selected to be matched, even if they belong to an alternating component. A vertex in an alternating component can also be matched from a neighbor outside of the component.

To further enhance solution quality, for each alternating component A, we keep track of the vertex v in A that has the largest neighborhood outside of A. If a vertex in A is selected to be matched by the Double-Sided Mindegree heuristic, we instead match v to a vertex in its neighborhood outside of A. When we remove v from A, we obtain a perfect matching for the remaining vertices in A.

LEMMA 3.4. *Let $deg_G(u)$ be the degree of u in the current graph G. After removing a vertex u of $deg_G(u) > 2$ from a cycle-free alternating component A, there is a unique perfect matching M_A in A. Furthermore, M_A can be found by repeated degree 1 reductions.*

PROOF. By the definition of an alternating component, all neighbors of u in A are of degree 2. Thus, for each neighbor v of u in A, removing u creates one path that contains an odd number of degree 2 vertices. Let w, $w \neq u$ be the endpoint of this path and let y be the unique neighbor of w on this path. Clearly, $deg_G(w) > 2$ and $deg_G(y) = 2$.

Let A_w be the (possibly empty) cycle-free alternating component that is reachable from $\Gamma(w) \setminus \{y\}$. Note that $A_w \subset A$.

After removing u, the path is matched entirely by degree 1 reduction, starting with v and its remaining neighbor. Because the path contains an odd number of degree 2 vertices, w will be matched to y, thus removing it from A_w. If the path contains only one degree 2 vertex, then $y = v$.

Because $deg_G(w) > 2$ before the removal of w, all its neighbors in A_w are again of degree 2. Thus, removing w creates one new cycle-free alternating component connected to a degree 1 vertex for each other neighbor of w in A_w. Each of these has again a perfect matching, as described earlier in the text. Therefore, a unique maximum matching in $A \setminus \{u\}$ is obtained by repeated degree 1 reductions. □

Note that in case of a cyclic alternating component, a perfect matching is obtained as soon as a cycle is discovered:

LEMMA 3.5. *If an alternating component A contains a single cycle, A has two perfect matchings.*

PROOF. Because G is bipartite, the cycle itself is of even length and thus has two perfect matchings. Since it is the only cycle in A, after its removal A is a forest. Each connected component in this forest is a cycle-free alternating component connected to a degree 1 vertex v, and thus has a unique perfect matching, as shown in Lemma 3.4. □

Removing a vertex of degree 2 would simply split the component in two. This cannot happen during the heuristic, so we do not investigate this event further. However, we have not yet considered the case of an alternating component with more than one cycle.

LEMMA 3.6. *If an alternating component A contains multiple cycles, it has no perfect matching.*

PROOF. By Lemma 3.4, all trees connected to a cycle or a path between cycles have a unique perfect matching once the connecting vertex is removed. Therefore, we can ignore these trees and focus on cycles and paths connecting them. Let v be a vertex belonging to a cycle C in A, and let $deg_G(v) > 2$. Suppose for some perfect matching M that v is matched to a vertex not in C. Then, by parity there must be another vertex $v' \in C$ with $deg_G(v') > 2$ that is also not matched to a vertex in C. Without loss of generality, we can assume that each vertex on the path between v and v' in C is matched to a vertex in C. Then we have a path of even length between v and v', containing an odd number of additional vertices. Clearly, these cannot all be matched without using a vertex outside C, and thus, contradicting the perfectness of M.

This means that in any perfect matching M', each cycle is matched using only edges from C. Thus, since A contains multiple cycles and A is connected, there must be a path P connecting two cycles. However, P must be of even length and thus contain an odd number of vertices. Therefore, it cannot have a perfect matching. This means that if A is connected and has a perfect matching, it cannot contain more than one cycle. □

To obtain a perfect matching on a cyclic component, it is, therefore, sufficient to start matching as soon as the first cycle in a component is discovered.

3.3 Variant Approaches

Our suggested heuristic is the result of several series of experiments with similar approaches. Among these, we selected the variant that turned out to be the most promising. Tested variants that were discarded due to poor performance include the following:

The Component-Based heuristic can be modified to avoid matching vertices that belong to a component. In that case, components grow until a cycle appears. If this does not happen, the component must be matched when the heuristic finishes by selecting a vertex that remains unmatched, and match the rest of the acyclic component according to Lemma 3.4. However, experiments show that this is not advisable, since the number of components that remain acyclic in this scheme usually outweighs the errors our unmodified heuristic makes by matching acyclic components by far, resulting in lower solution quality.

In addition, we considered a variant using the limited Mindegree approach instead of full Mindegree for vertices with degree greater than 2, which means that these vertices are selected in arbitrary order, instead of being selected by degree precedence. This can be described as a Karp-Sipser heuristic with the addition of the build-up of alternating components, as explained in this section. However, this approach gave solutions of poor quality even in comparison to the unmodified Karp-Sipser heuristic which is also faster on the average.

In general, it is necessary to consider the effect that degree 2 reductions have on the Mindegree heuristic. The contraction removes 2 vertices and replaces them by a single vertex of their combined degree, thereby skewing the result of the Mindegree process. Any such heuristic that makes use of degree 2 reductions will have to deal with this effect in some way. We avoid this because we do not explicitly contract edges.

3.4 Greedy Enhancement

For graphs that have a perfect matching, we experimented with the following Greedy-Enhanced version of our Component-Based approach: Run the Simple Greedy heuristic and count the unmatched edges. If less than k vertices remain unmatched then keep the result. Otherwise, run the Component-Based heuristic, as described earlier in the text. To determine k for a given graph, we used the experimentally determined solution quality of the Component-Based heuristic, which matches about 99.9% of all vertices as a guideline since we want our heuristic to be invoked only if an improved solution can be expected. Further experiments suggested that setting k to $0.005|E|$ is a good rule of thumb. Note that our target application, that is, graphs derived from matrices of full structural rank, always have a perfect matching, making this approach particularly suitable.

4. EXPERIMENTS

In order to analyze our Component-Based heuristic, we compare it along with the other heuristics described in Section 2 in regard to performance and quality, that is, the ratio between unmatched vertices and total vertices. We then use these heuristics as an initialization for exact matching algorithms and measure the combined running time in order to find fast combined methods for solving the maximum cardinality matching problem, and estimate whether our heuristic yields competitive combinations.

4.1 Algorithms

The exact algorithms we use are Breadth-First-Search (BFS) for augmenting paths and the Hopcroft-Karp algorithm [Hopcroft and Karp 1973], both implemented by Florin Dobrian, and *ABMP*, a refinement of the Hopcroft-Karp algorithm by Alt et al. [1991] as well as the Push-relabel algorithm [Gabow and Tarjan 1988]. The latter two were implemented by João C. Setubal [1996].

All experiments were performed on an Intel Core2 Duo 2.4Ghz with 2GB memory running Fedora 8 (Linux Kernel 2.6.26.6-49.fc8). All codes were written in C++ and compiled using g++ version 4.1.2 20070925 (Red Hat 4.1.2-33) using the -O3 optimization option.

4.2 Test Instances

We used two test sets in our experiments. Test Set 1 contains 900 test instances, while Test Set 2 contains 72 test instances. The instances come from two different sources, one being a large collection of real-world matrices and the other a set of randomly generated graphs. Computational results and more details on the experimental setup can be found in the appendix.

4.2.1 *Generated Instances.* We used the HiLo and the Rbg random instance generators introduced in Cherkassky et al. [1998] and Setubal [1996] to generate two groups of random instances, each consisting of 13 different graphs. By design, these instances are difficult to solve with most exact matching algorithms. They are contained in both test sets.

The HiLo generator creates graphs with a unique perfect matching [Cherkassky et al. 1998]. It generates instances that are “difficult” for most exact algorithms, provided a random vertex permutation is used.

Let $G = (V_1 \cup V_2, E)$ be a graph produced by this generator. This graph is defined by three parameters, l, k, and d. Vertices of V_1 are partitioned into l groups, each containing k vertices. For $1 \leq i \leq k, 1 \leq j \leq l$, we refer to the i-th vertex in group j by x_i^j . Vertices of V_2 are partitioned similarly, and y_i^j is defined similarly to x_i^j . Each vertex x_i^j is connected to vertices y_p^j for $\max(1; i - d) \leq p \leq i$ and, if $j < l$, to vertices y_p^{j+1} for $\max(1; i - d) \leq p \leq i$. There are no other edges. Unlike Cherkassky et al. [1998], we set the number of groups l to values higher than 1, which results in significantly more difficult instances for all algorithms. We also permute vertices randomly. Table I shows the parameters used in the test instances.

The Rbg generators produce random bipartite graphs where the vertices of both partitions V_1 and V_2, are divided into k groups of equal size. For each vertex of the j -th group of V_1, the generator chooses y random neighbors from the $(i - 1)$-th through $(i + 1)$-th groups of V_2 (with wrap-around), where y is binomially distributed with mean d (thus, d equals mean vertex degree). The

Table I. Properties and Runtimes for the Test Instances from the HiLo Generator

Matrix	Vertices Per Side	Average Degree	Groups	ABMP Greedy	PR Greedy	BFS Greedy	HK Greedy
hilo640_1_10.mtx	640,000	10	1	9.01	52.65	3	52.73
Hilo640_2_10.mtx	640,000	10	2	21.74	88.2	5.48	79.02
hilo640_128_5.mtx	640,000	5	128	78.32	110.48	190.02	494.99
hilo640_128_10.mtx	640,000	10	128	262.99	319.69	555.6	685.62
Hilo640_256_5.mtx	640,000	5	256	95.69	110.24	280.96	599.68
Hilo640_512_5.mtx	640,000	5	512	83.65	101.11	346.62	670.41
Hilo640_1024_5.mtx	640,000	5	1,024	80.71	97.09	278.85	654.56
Hilo600_10000_5.mtx	600,000	5	10,000	46.33	148.24	46.47	482.22
Hilo600_1000_5.mtx	600,000	5	1,000	72.51	85.42	249.19	597.88
hilo320_128_10.mtx	320,000	10	128	97.74	116.6	211.61	322.97
hilo160_128_10.mtx	160,000	10	128	29.1	37.55	49.18	74.04
hilo80_128_10.mtx	80,000	10	128	11.37	12.59	12.88	18.76
hilo40_128_10.mtx	40,000	10	128	3.99	4.28	2.89	4.57
Average	483,076.92	7.69	—	68.7	98.78	171.75	364.42

For Karp-Sipser, dynamic mindegree, and component-based initialization, the combined runtimes are equal to the heuristic runtimes (see Table IV). All runtimes are given in seconds.

Table II. Properties and Runtimes for the Test Instances from the Rbg Generator

Matrix	Vertices per Side	Average Degree	Groups	ABMP Greedy	PR Greedy	BFS Greedy	HK Greedy
rbg2_512_128_10.max	512,000	10	128	1.11	9.57	8.89	3.58
rbg2_512_25600_10.max	512,000	10	25,600	0.85	5.23	0.26	1.99
rbg2_512_2560_10.max	512,000	10	2,560	1.18	7.1	0.58	3.21
rbg2_512_256_10.mtx	512,000	10	256	1.18	7.66	3.76	3.35
rbg2_512_256_40.mtx	512,000	40	256	5.19	1.12	5.37	5.28
rbg2_512_256_5.mtx	512,000	5	256	2	7.77	4.66	4.35
rbg2_512_32_10.mtx	512,000	10	32	0.93	7.97	35.58	2.66
rbg2_512_32_40.mtx	512,000	40	32	4.02	1.42	92.33	3.03
rbg2_512_32_5.mtx	512,000	5	32	0.87	2.04	14.9	4.64
rbg2_51_256_10.mtx	51,200	10	256	0.08	0.04	0.02	0.07
rbg256_10.mtx	256,000	10		0.95	7.94	0.92	1.73
rbg256_40.mtx	256,000	40		1.59	1.15	0.26	0.73
rbg256_5.mtx	256,000	5		0.86	4.42	2.82	3.39
Average	417,476.92	15.77		1.6	4.88	13.1	2.92

Runtimes are given in seconds.

indices i and j are not related because vertices in V_1 are randomly shuffled before neighbors in V_2 are assigned.

In Cherkassky et al. [1998] two families of problems are considered. These are called *fewg* (32 groups) and *manyg* (256 groups). Both have $d = 5$. In addition to these, we consider other numbers of groups and higher densities. Table II shows the parameters used in the test instances.

4.2.2 *Real-World Instances.* In addition to the generated graphs, we obtained real-world instances from the University of Florida Sparse Matrix Collection [Davis 2007]. We added a small group of 46 matrices of similar size to Test Set 2 in order to test combinations of heuristics and exact algorithms, and added an additional 828 matrices to Test Set 1 in order to test speed and quality of the heuristics. Thus, the test sets comprise a total of 72 and 900 instances, respectively, with Test Set 2 being a subset of Test Set 1. The instances were selected from the UF sparse matrix collection by removing nonquadratic matrices, matrices that are too large for the available memory, some very small matrices, and some extremely sparse matrices with very small maximum matchings.

4.3 Heuristic Results

We compared the Component-Based heuristic and its Greedy-Enhanced version with several other heuristics regarding their solution quality and performance. These heuristics are Simple Greedy, Static Mindegree, Single- and Double-Sided Karp-Sipser, as well as Single- and Double-Sided Dynamic Mindegree. We used Test Set 1 for these experiments.

Generally, the performance behaved as expected, with Simple Greedy being the fastest and our Component-Based heuristic which obviously requires the most work per vertex being the slowest. Running time for Simple Greedy is primarily determined by the number of vertices, and it generally grows almost linearly with the number of vertices for a given graph structure and average degree. The more elaborate heuristics are more strongly affected by the number of edges and by graph structure than the simpler ones, although this effect is somewhat unpredictable.

The quality generally also behaved as expected, with the slower heuristics delivering noticeably better results. This means that selecting a heuristic as an initialization for an exact algorithm is generally a trade-off between quality and speed, but the effect on the total running time is far from predictable. Of course, the quality is independent of the number of vertices, but denser graphs usually show less unmatched vertices.

4.3.1 *Results on Real-World Matrices.* For the real world instances, the Component-Based heuristic usually matched almost 99.9% of the vertices, while Simple Greedy matched about 95%. The results of the Simple Greedy heuristic are more susceptible to implementation details such as the order in which vertices are scanned. This could be circumvented by picking vertices randomly, but doing so is not advisable for performance reasons.

4.3.2 *Results on Generated Graphs.* In the HiLo instances which have a unique perfect matching, the Karp-Sipser, Component-Based, and Dynamic Mindegree heuristics give optimal solutions. It is easy to show that any heuristic which gives precedence to degree 1 reductions and performs them repeatedly will find a perfect matching here [Gilbert 2009]. The other heuristics yielded results comparable to their performance on real-world instances. For a setting of $l = 1$ and $d = 10$, the greedy matching paired about 96% of the vertices, provided that the vertices are initially permuted randomly. For both the Simple Greedy and the Static Mindegree heuristics values of $l > 1$ result in worse solutions, while a higher average degree results in better solutions. Details can be found in Tables III and IV. The running times are noticeably influenced by the number of edges but also by the number of groups. For a setting of $d = 5$,

Table III. Approximation by Various Heuristics for the Test Instances from the HiLo Generator

Matrix	Greedy Match	Static Mindegree	Karp-Sipser	Dynamic Mindeg.	Component Based
hilo640_1_10.mtx	49,850	55,964	0	0	0
Hilo640_2_10.mtx	166,840	45,602	0	0	0
hilo640_128_5.mtx	62,626	75,956	0	0	0
hilo640_128_10.mtx	21,018	25,912	0	0	0
Hilo640_256_5.mtx	61,330	75,510	0	0	0
Hilo640_512_5.mtx	61,586	74,774	0	0	0
Hilo640_1024_5.mtx	62,112	75,300	0	0	0
Hilo600_10000_5.mtx	84,130	83,446	0	0	0
Hilo600_1000_5.mtx	58,756	69,762	0	0	0
hilo320_128_10.mtx	21,018	25,912	0	0	0
hilo160_128_10.mtx	21,018	13,990	0	0	0
hilo80_128_10.mtx	5,838	6,752	0	0	0
hilo40_128_10.mtx	3,356	3,738	0	0	0
Average	52,267.54	48,662.92	0	0	0

The values denote the total number of unmatched vertices on both sides. Thus, the number of unmatched edges is half that figure.

Table IV. Running Times for Various Heuristics for the Test Instances from the HiLo Generator

Matrix	Greedy Match	Static Mindegree	Karp-Sipser	Dynamic Mindeg.	Component Based
hilo640_1_10.mtx	0.16	0.39	0.49	0.72	0.63
Hilo640_2_10.mtx	0.17	0.35	0.59	0.84	0.87
hilo640_128_5.mtx	0.16	0.41	0.55	0.76	0.79
hilo640_128_10.mtx	0.19	0.55	0.67	1.1	1.21
Hilo640_256_5.mtx	0.17	0.41	0.53	0.73	0.75
Hilo640_512_5.mtx	0.16	0.4	0.59	0.72	0.74
Hilo640_1024_5.mtx	0.16	0.4	0.56	0.82	0.92
Hilo600_10000_5.mtx	0.15	0.4	0.64	1.14	1.2
Hilo600_1000_5.mtx	0.15	0.38	0.5	0.77	0.83
hilo320_128_10.mtx	0.09	0.25	0.32	0.53	0.57
hilo160_128_10.mtx	0.04	0.12	0.14	0.24	0.26
hilo80_128_10.mtx	0.01	0.04	0.06	0.12	0.11
hilo40_128_10.mtx	0.01	0.02	0.03	0.06	0.05
Average	0.12	0.32	0.44	0.66	0.69

For Karp-Sipser, dynamic mindegree and component-based initialization, a perfect matching is obtained without starting an exact algorithm. Thus, times given here for these heuristics can be compared to combined running times. All running times are given in seconds.

a total of 640,000 vertices, and l at various powers of 2, the running time for most algorithms peaked at $l = 256$ (i.e., 256 groups).

In the Rbg instances, running time is more dependent on the number of edges than on the number of groups. The Component-Based and Dynamic Mindegree heuristics are strongly affected, while the running times for Simple Greedy and Static Mindegree are almost independent of graph density. Simple Greedy matches only about 86% of the vertices here. Other heuristics are significantly better. The quality of the Component-Based heuristic is close to the optimum (see Tables V and VI).

Table V. Number of Unmatched Vertices for Various Heuristics for the Test Instances from the Rbg Generator

Matrix	Greedy Match	Static Mindegree	Karp-Sipser	Dynamic Mindeg.	Component Based	Exact Solution
rbg2_512_128_10.mtx	17,726	6,415	172	24	24	24
rbg2_512_25600_10.mtx	15,430	5,902	5,916	722	652	24
rbg2_512_2560_10.mtx	17,546	6,281	1,480	30	28	28
rbg2_512_256_10.mtx	17,710	6,386	304	30	30	30
rbg2_512_256_40.mtx	4,465	2,330	264	0	0	0
rbg2_512_256_5.mtx	17,700	6,378	274	24	24	24
rbg2_512_32_10.mtx	17,553	6,512	118	34	34	34
rbg2_512_32_40.mtx	4,433	2,386	40	0	0	0
rbg2_512_32_5.mtx	35,456	12,309	3,886	3,838	3,842	3,836
rbg2_51_256_10.mtx	1,765	621	142	2	2	2
rbg256_10.mtx	17,765	6,476	46	26	30	26
rbg256_40.mtx	4,448	2,332	8	2	2	0
rbg256_5.mtx	35,168	12,247	3,724	3,716	3,714	3,714
Average	15,935.77	5,890.38	1,259.54	649.85	644.77	595.54

Note that a perfect matching exists only for some of the instances. The last column shows the minimum number of unmatched vertices for the instance.

Table VI. Running Times for Various Heuristics for the Test Instances from the Rbg Generator

Matrix	Greedy Match	Static Mindegree	Karp-Sipser	Dynamic Mindeg.	Component Based
rbg2_512_128_10.mtx	0.06	0.16	0.3	0.43	0.44
rbg2_512_25600_10.mtx	0.05	0.15	0.23	0.26	0.3
rbg2_512_2560_10.mtx	0.05	0.16	0.28	0.29	0.34
rbg2_512_256_10.mtx	0.06	0.16	0.29	0.38	0.4
rbg2_512_256_40.mtx	0.08	0.32	0.85	1.2	1.18
rbg2_512_256_5.mtx	0.06	0.19	0.29	0.37	0.39
rbg2_512_32_10.mtx	0.06	0.16	0.3	0.61	0.61
rbg2_512_32_40.mtx	0.08	0.32	0.86	2.46	2.29
rbg2_512_32_5.mtx	0.05	0.12	0.21	0.32	0.4
rbg2_51_256_10.mtx	0.01	0.01	0.02	0.02	0.02
rbg256_10.mtx	0.05	0.17	0.3	0.75	0.74
rbg256_40.mtx	0.08	0.33	0.88	3.3	3.04
rbg256_5.mtx	0.05	0.12	0.21	0.36	0.41
Average	0.06	0.18	0.39	0.83	0.81

All running times are given in seconds.

4.3.3 *Comparing the Heuristics.* Figure 1 shows that Simple Greedy and Static Mindegree trade quality for speed. Static Mindegree was approximately twice as fast as the Karp-Sipser heuristic, and Simple Greedy was again twice as fast as Static Mindegree. However, the number of errors was about 25 (Static Mindegree) and 40 (Simple Greedy) times higher than that of Karp-Sipser, which means that with these initializations the exact algorithms will have significantly more work to do.

Among the Dynamic Mindegree heuristics, the two-sided approach showed better quality than the one-sided approach while using comparable running time. Thus, we discarded the one-sided approach in further experiments.

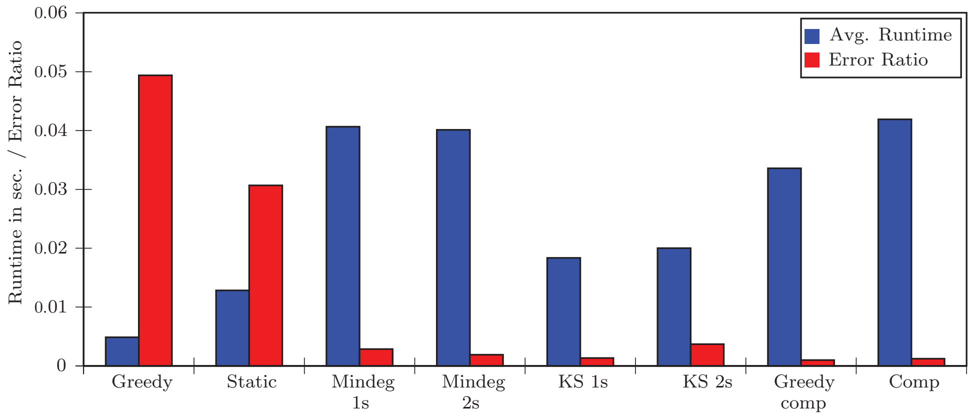

Fig. 1. Error ratios and average running times for heuristics on Test Set 1. Running time is denoted in seconds. Differences between one sided (1s) and two-sided (2s) heuristics are visible. Static refers to the Static Mindegree heuristic and KS to the Karp-Sipser heuristic. Comp is the Component-Based heuristic and Greedy comp is the Greedy-Enhanced version with a threshold of 0.005. Error ratio is the ratio between unmatched vertices and total vertices.

However, both their running times are only slightly better than the Component-Based heuristic which offers noticeably better solution quality, and thus faster exact computations (see Section 4.4).

In our experiments for the Karp-Sipser heuristic the one-sided approach was far superior to the two-sided approach. Consequently, only the one-sided approach was used in the following experiments. Furthermore, even though algorithmically the Karp-Sipser heuristic differs only in the limited size of the priority queue from the full Dynamic Mindegree heuristic and differences between the implementations go only as far as necessary, it ran approximately twice as fast and gave better average solution quality as well.

Möhring and Müller-Hannemann [1995], who performed a similar study for general matching on random graphs and obtained comparable results regarding the trade-off between quality and performance, also observed that the trade-off for the one-sided Karp-Sipser heuristic is comparatively good, making it preferable to One-Sided Dynamic Mindegree for performance reasons and to Static Mindegree for quality reasons. However, in their study the number of unmatched vertices for the One-Sided Dynamic Mindegree heuristic was approximately half the number of unmatched vertices for Karp-Sipser. In our experiments, the situation is similar for the Rbg instances (see Table V). On the other instance groups the Karp-Sipser heuristic and Dynamic Mindegree provided comparable solution quality.

A comparison between the Component-Based heuristic and Karp-Sipser is somewhat more difficult. Karp-Sipser is clearly faster, but the Component-Based heuristic consistently offered a slightly better average solution quality. Furthermore, the Component-Based heuristic achieves a maximum matching more often. If the maximum matching is perfect, maximality can be detected trivially by counting the number of unmatched vertices. In such a case, there is no need to start the exact algorithm, thus allowing significantly saving on running time. While our experiments include graphs without a perfect matching, the target application, that is, linear solvers, does not; therefore, this feature becomes a distinct advantage for the Component-Based heuristic.

Also, the correlation between the results of the Component-Based heuristic and Simple Greedy is weaker than that between Simple Greedy and the Karp-Sipser heuristic. This result inspired the Greedy Enhanced heuristic, which runs Simple Greedy and counts the number of unmatched vertices (see Section 3.4). If this number is greater than a given threshold, the Component-Based heuristic is used to improve the solution. Of course, the Simple Greedy result is retained if it is still better. This approach is very sensitive to the value of the threshold. Obviously, the threshold should be between the expected error ratio of the Simple Greedy and the Component-Based heuristic (i.e., between approximately 95% and 99.9%). Thresholds lower than 98% yielded lower quality than the unmodified Component-Based heuristics. Our experiments gave very good results for a threshold of 99.5%, reducing the error rate by about 20% and increasing performance by 40% compared to the Component-Based heuristic.

4.4 Initialized Exact Algorithms

In the previous section, we showed that all heuristics offer a trade-off between quality and performance. In order to find the heuristic that provides the largest performance improvement for the exact algorithms, we tested combinations between the heuristics and the ABMP, Push-relabel, Hopcroft-Karp, and BFS algorithms, and considered their combined running time on Test Set 2.

In general, we can state that both Simple Greedy and Static Mindegree show poor combined performance. This is to be expected for the generated graphs, as they are designed to be "difficult" for simple strategies, but even on the real-world instances the more sophisticated heuristics result in combined algorithms that are often faster by a factor of 5. Compared to this, the differences among the other heuristics are rather small.

Although there is no direct correlation between the number of matching errors made by the heuristic and the total running time for a given instance, the heuristics that gave the best trade-off between quality and performance, that is, Karp-Sipser and the Component-Based approach, also result in the best combined running times here.

Among the test instances, the HiLo graphs are an extreme case, since they are very hard to solve for most exact algorithms, yet they are easy for the Karp-Sipser, Component-Based, and Dynamic Mindegree heuristics. Therefore, using these heuristics instead of Simple Greedy provides extremely high speedups, which lie between a factor of 100 and 600. Running times using greedy initializations can be found in Table I. For Karp-Sipser, Component-Based, and Dynamic Mindegree initialization, the combined running time is equal to the heuristic running time, which is shown in Table V.

A similar effect can be observed in the Rbg instances, but since no heuristic consistently finds optimum solutions here, the effect is far weaker. Still, initialization using the Karp-Sipser, Component-Based, or Dynamic Mindegree heuristic improves overall performance noticeably, especially for the BFS

Table VII. Running Times for Combined Algorithms for the Test Instances from the Rbg Generator

Matrix	Comp ABMP	KS ABMP	Comp PR	KS PR.	Comp BFS	KS BFS	Comp HK	KS HK
rbg2_512_128_10.mtx	0.65	0.91	11.64	11.58	0.58	1.19	0.55	2.59
rbg2_512_25600_10.mtx	0.92	0.89	9.75	9.7	0.58	0.5	7.24	56.43
rbg2_512_2560_10.mtx	0.44	1.26	10.07	9.94	0.48	0.87	0.47	18.06
rbg2_512_256_10.mtx	0.56	1.05	8.97	8.82	0.53	1.09	0.51	4.18
rbg2_512_256_40.mtx	1.17	4.78	19.02	17.82	1.27	2.34	1.26	8.31
rbg2_512_256_5.mtx	2.4	2.22	9.07	8.91	0.52	1.35	0.51	3.86
rbg2_512_32_10.mtx	0.94	0.68	10.73	10.31	0.73	1.16	0.72	1.96
rbg2_512_32_40.mtx	2.24	3.4	20.89	19.16	2.46	1.95	2.43	2.17
rbg2_512_32_5.mtx	0.63	0.45	8.49	8.38	1.26	1.27	0.74	1.09
rbg2_51_256_10.mtx	0.03	0.06	0.54	0.56	0.04	0.06	0.04	0.11
rbg256_10.mtx	1.06	0.63	10.93	10.52	0.91	0.46	0.92	0.46
rbg256_40.mtx	4.44	2.22	23.08	20.02	3.19	1.02	3.21	1.03
rbg256_5.mtx	0.58	0.39	8.95	8.75	0.54	0.37	0.54	0.36
Average	1.24	1.46	11.7	11.11	1.01	1.05	1.47	7.74

We compare the component-based (Comp) heuristic with Karp-Sipser (KS). PR denotes Push-relabel and HK the Hopcroft-Karp algorithm. All running times are given in seconds.

algorithm (see Table VII). However, for denser graphs the performance gain declines. This stems from the fact that the performance of those heuristics is more dependent on the number of edges than Simple Greedy, which also has a strong tendency to provide solutions of higher quality in denser graphs. Together, this provides a very fast heuristic with good solution quality on dense graphs, which results in fast combined running times (see Table II). Still, even at $d = 40$, the Push-relabel algorithm using Simple Greedy initialization is only faster than the same algorithm using Component-Based initialization by a small margin.

4.4.1 *ABMP and Push-Relabel.* For the ABMP and the Push-relabel algorithm, all heuristics except Simple Greedy and Static Mindegree showed roughly the same behavior. They provided a speed-up by a factor of 3 for Push-relabel and up to 20 for ABMP compared to Simple Greedy initialization, as shown in Figure 2. The problems with using heuristic initialization for Push-relabel have been described before [Cherkassky et al. 1998]. In our test cases, it is certainly worthwhile to use this initialization, mostly due to high speed-ups on the HiLo instances. The Static Mindegree heuristic performed extremely poorly here. We recommend not to use it for initialization.

Note that because even the Simple Greedy heuristic solves many instances well, most significant speed-ups obtained by using more elaborate heuristics arise from large gains on a small number of matrices. The differences in median running time are far less pronounced. This is partially due to many running times being close to 0 and because of measurement inaccuracies, differences in running times that are smaller than 0.01 seconds are not registered.

For ABMP and Push-relabel, the Karp-Sipser heuristic "wins" against the Component-Based approach and against the Mindegree heuristics by a small

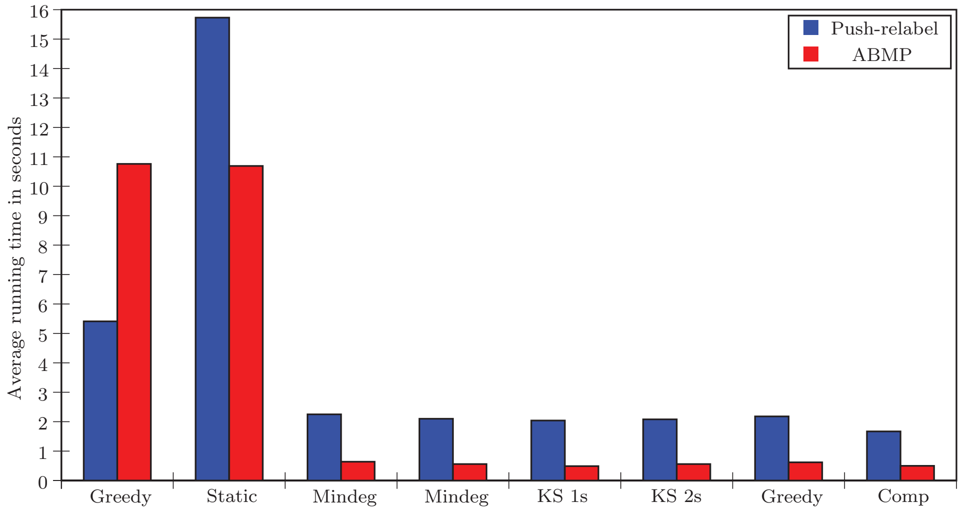

Fig. 2. Average running times in seconds for initialized exact algorithms on Test Set 2. Differences between one-sided (1s) and two-sided (2s) heuristics are visible. Static refers to the Static Mindegree heuristic and KS to the Karp-Sipser heuristic. Comp is the Component-Based heuristic and Greedy comp is the Greedy-Enhanced version with a threshold of 0.005. The Greedy-Enhanced version "wins" because it can more often avoid starting the costly exact algorithm entirely. ABMP profits significantly more from good initialization than Push-relabel.

margin, mostly due to better speed-up on ABMP. The Greedy-Enhanced Component-Based heuristic offers approximately the same speed-up on ABMP and the best running times for Push-relabel. This is due to finding a perfect matching more often, and thereby eschewing the need to start up any exact algorithm. The exact numbers can be found in Table VII.

4.4.2 *Hopcroft-Karp and BFS.* For the Hopcroft-Karp and augmenting path *BFS* algorithms, we focused on the results provided by the best heuristic initializations and compared them to Simple Greedy initialization.

The Hopcroft-Karp algorithm with Simple Greedy initialization showed very good performance on the real-world instances and weak performance on HiLo and Rbg. Thus, the algorithm benefits strongly from using better heuristic initialization in the latter two cases, but not so much in the former.

In comparison to Hopcroft-Karp, BFS performed much better on HiLo, but generally worse on the other test sets. In all experiments, it profits from Component-Based initialization, although obviously not on every instance. An exceptional case were experiments on the matrix Hamrle3, where BFS with Simple Greedy initialization was more than 20 times faster than either Hopcroft-Karp with Simple Greedy or both algorithms with Component-Based initialization. Furthermore, the Component-Based heuristic reduced the number of unmatched vertices by 90% compared to Simple Greedy here, but this led to short augmenting paths being used up by the Component-Based heuristic, thus requiring the exact algorithms to find relatively long augmenting paths, thereby taking significant extra time. Note that this matrix can be considered

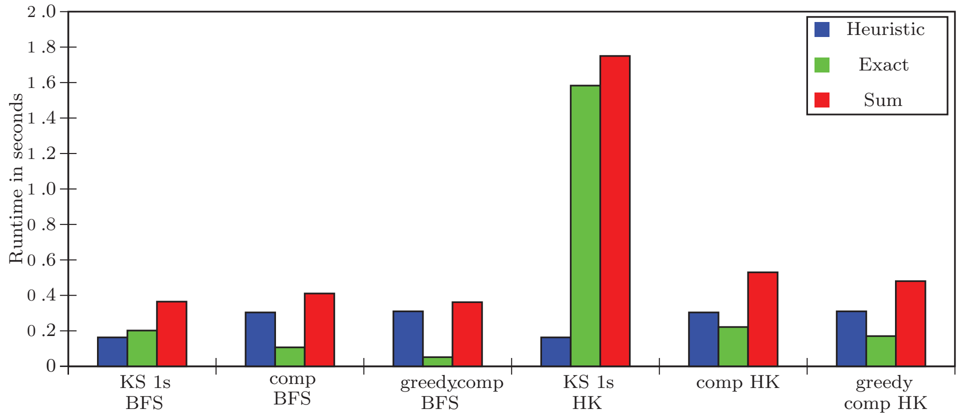

Fig. 3. Running times for differently initialized exact algorithms on Test Set 2. The Karp-Sipser (KS) heuristic spends less time on the initialization, resulting in longer running times for the exact algorithm. For BFS, the sum is almost the same as for the Greedy-Enhanced Component-Based heuristic. The Hopcroft-Karp algorithm takes somewhat longer due to the increased start-up cost, and performs poorly with Karp-Sipser initialization due to exceptionally weak results on two matrices.

a very special case. We did not observe comparable behavior on any other test instance.

For Test Set 2, Karp-Sipser performed somewhat better on the average than the unmodified Component-Based heuristic when used with BFS, as shown in Figure 3. The Greedy-Enhanced Component-Based heuristic showed approximately the same performance as Karp-Sipser. For the Hopcroft-Karp algorithm, Karp-Sipser performed significantly worse than the unmodified Component-Based heuristic and the Greedy-Enhanced Component-Based heuristic. Note that the composition of the total running time varies widely. The Karp-Sipser heuristic takes less time than the exact algorithms it initializes, while the Component-Based and the Greedy-Enhanced Component-Based heuristics take significantly more time than their exact algorithms. However, in the case of BFS, the resulting total running times are almost the same.

5. DISCUSSION

From what we have seen in the experiments, we can safely conclude that in most cases.

—Using more elaborate initializations than Simple Greedy usually improves performance.
—The one-sided Karp-Sipser heuristic is superior to the two-sided one.
—The two-sided Dynamic Mindegree heuristic is superior to the one-sided one.
—The Karp-Sipser heuristic yields faster combined algorithms than full Mindegree.
—The Component-Based heuristic offers better quality than the Karp-Sipser heuristic, at the cost of performance. Both are suitable as initializations.

—The Component-Based heuristic can be improved by running Simple Greedy and then determine whether a refinement is worthwhile. This Greedy-Enhanced Component-Based heuristic provides the best initialization.

—Several other variants of the Component-Based heuristic do not seem to work as well.

Performance improvements in the combined experiments appear because the more sophisticated heuristics give better solutions than simpler heuristics on “difficult” instances, but the improvement is marginal in the easy cases. However, even without the generated instances there were enough “difficult” instances in our test set to make the sophisticated heuristics superior, on average. Thus, unless further knowledge of the instances is available, it is to be expected that our findings will also apply to other real world instances as well. Therefore, we suggest to use the Karp-Sipser heuristic to initialize matching algorithms in “easy” sparse graphs. If the instances are known to be “difficult,” the Component-Based heuristic is even better, and if we know that a perfect matching exists, the Greedy-Enhanced Component-Based heuristic can be expected to offer the best results.

Considering the exact algorithms, it seems that Hopcroft-Karp and ABMP work better than Push-relabel in conjunction with sophisticated initialization heuristics, but BFS worked even better and gave the best final results. This is consistent with findings from Setubal [1996], since the remaining number of vertices to be matched after a good initialization is small. Preliminary testing also showed poor performance for the depth-first-search-based algorithm. This was also observed in Cherkassky et al. [1998] and Setubal [1996]. Whether BFS with a good initialization outperforms all other exact algorithms on sparse graphs remains a topic of further study.

It is important to remember that the implementations are not directly comparable. BFS and Hopcroft-Karp by Dobrian are more recent and significantly more elaborate than the implementations by Setubal. Thus, this study should not be used to compare ABMP or Push-relabel to Hopcroft-Karp. However, even with an optimal implementation, we do not expect ABMP to be clearly superior to Hopcroft-Karp as its reported advantages in the literature rely on a large number of unmatched vertices. A good heuristic might actually obviate the need for more sophisticated exact algorithms.

It remains to note that for linear solvers, the value of the matrix entries must be taken into account. This requires finding a perfect matching of maximum weight or approximatively maximum weight. While most of the heuristics described here do not allow easy adaptation to the weighted problem, they could again be used as an initialization step followed by weight-augmenting path search to obtain a perfect matching of approximate maximum weight. Alternatively, after using an initialization and an exact algorithm to obtain a perfect matching, as described in this article, one could use weight-augmenting cycles to obtain maximum weight. Preliminary tests suggest that weight-augmenting 4-cycles could be sufficient to provide solutions of high weight. Of course, these techniques can be used in any application of maximum weight matching. They are not restricted to linear solvers.

APPENDIX

A. COMPUTATIONAL RESULTS

All results presented here are pure computation times and disregard I/O, as this would completely dominate the linear running times. The results given in the first three tables are simple averages over the corresponding test sets. For the initialized exact algorithms, the combined running time for a given matrix and a particular combination of initialization and exact algorithm is obtained as follows: First, we take the time for running the heuristic and obtaining a (partial) solution. Although we then save the solution to a file, we do not include the time to do this. The solution is then read back in and used to initialize the exact algorithm. From this point, we time the exact algorithm until it has produced a solution. We then add the two times together to produce the combined running time. This allows us to deal efficiently with multiple foreign codes. Furthermore, timings from this method should be an upper bound for our initializations when comparing them to implementations that have built-in greedy initialization.

Table IA. Values for Figure 1

	Greedy	Static	Mindeg 1s	Mindeg 2s	Greedy Comp	Comp	KS 1s	KS 2s
Time	0.00486	0.01282	0.04063	0.04010	0.03357	0.04189	0.01835	0.02
Quality	0.04936	0.03066	0.00285	0.00189	0.00098	0.00122	0.00133	0.00369

Table IIA. Values for Figure 2

	Greedy	Static	Mindeg 1s	Mindeg 2s	Greedy Comp	Comp	KS 1s	KS 2s
ABMP	10.76	10.69	0.64	0.56	0.49	0.56	0.62	0.5
Push-relabel	5.41	15.73	2.25	2.1	2.04	2.08	2.18	1.67

Table IIIA. Values for Figure 3

	KS	Comp	Greedy	ksmatch1s	Comp	Greedy
	1s BFS	BFS	Comp BFS	HK	HK	Comp HK
Heuristic	0.1630	0.3038	0.3100	0.1630	0.3038	0.3100
Exact	0.2017	0.1069	0.0516	1.5830	0.2216	0.1703
Sum	0.3647	0.4106	0.3616	1.75	0.53	0.4803

REFERENCES

Alt, H., Blum, N., Mehlhorn, K., and Paul, M. 1991. Computing a maximum cardinality matching in a bipartite graph in time O(n1.5pm/logn). *Inf. Process. Lett. 37*, 4, 237–240.

Brayton, R. K., Gustavson, F. G., and Willoughby, R. A. 1970. Some results on sparse matrices. *Math. Comput. 24*, 112, 937–954.

Cherkassky, B. V., Goldberg, A. V., Martin, P., Setubal, J. C., and Stolfi, J. 1998. Augment or push: A computational study of bipartite matching and unit-capacity flow algorithms. *ACM J. Exp. Algorithmics 3*.

Davis, T. A. 2007. The University of Florida sparse matrix collection. *IEEE Trans. Circuits Syst.* http://www.cise.ufl.edu/~davis/techreports/matrices.pdf

GABOW, H. AND TARJAN, R. 1988. Almost-optimum speed-ups of algorithms for bipartite matching and related problems. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing (STOC'88)*. ACM, New York, 514–527.

GILBERT, J. R. 2009. Private communication.

HOPCROFT, J. E. AND KARP, R. M. 1973. An *n*5/2 algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput. 2,* 4, 225–231.

KARP, R. M. AND SIPSER, M. 1981. Maximum matching's in sparse random graphs. In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science (FOCS'81).* IEEE, Los Alamitos, CA, 364–375.

KORTE, B. H. AND VYGEN, J. 2006. *Combinatorial Optimization: Theory and Algorithms.* Springer, Berlin.

MAGUN, J. 1998. Greedy matching algorithms: An experimental study. *ACM J. Exp. Algorithms 3*, 6.

MÖHRING, R. AND MÜLLER-HANNEMANN, M. 1995. Cardinality matching: Heuristic search for augmenting paths. Tech. rep., Technische Universität Berlin.

SANGIOVANNI-VINCENTELLI, A. 1976. A note on bipartite graphs and pivot selection in sparse matrices. *IEEE Trans. Circuits Syst. 23*, 12, 817–821.

SCHENK, O., HAGEMANN, M., AND ROLLIN, S. 2003. Recent advances in sparse linear solver technology for semiconductor device simulation matrices. In *Proceedings of the IEEE International Conference on Simulation of Semiconductor Processes and Devices*. IEEE, Los Alamitos, CA, 103–108.

SETUBAL, J. C. 1996. Sequential and parallel experimental results with bipartite matching algorithms. Tech. Rep.

Received July 2009; revised November 2009; accepted December 2009