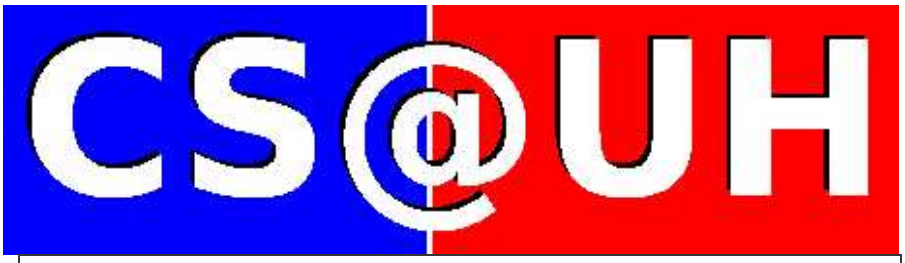


Fast Filtering Heuristics for Perfect Bipartite Matchings[1]

Rakesh M. Verma[1] and Jack Wiedrick
Computer Science Department
University of Houston
Houston, TX, 77204, USA
http://www.cs.uh.edu

UH-CS-08-19
December 20, 2008

Keywords: Graphs, Bipartite Matching, Heuristics, Performance

Abstract

The bipartite matching problem has wide applicability. For example, in computer science, researchers have applied bipartite matching in objection recognition, image processing, scheduling, genomics, term rewriting and formal verification, and computer security. We present a heuristic model of perfect matchings in bipartite graphs. The goal of the model is to explore the use of low-cost heuristics in applications where only perfect matchings are desired. The heuristics test for a number of conditions necessary for perfect matchings to exist in a graph, and as such, they can be used as filters to quickly identify and exclude unsuitable graphs from further processing. Whereas the best maximum cardinality bipartite matching algorithms run in about $O(n^{5/2})$ time (n being the number of vertices in the bipartite graph), we show that it's possible to make a reliable *prima facie* guess about the potential for a perfect matching in a given graph in roughly $O(n^2 \log n)$ time without bookkeeping costs. Although we represent bipartite graphs by matrices, the heuristics can be adapted to other representations as well. Proofs of validity and runtime analysis of the heuristics are presented, along with a discussion of their performance characteristics in practice. A software framework is described that implements each of the components of the heuristic model and includes various test facilities for exercising them. Empirical results from the framework demonstrate the effectiveness of the heuristics for graphs with up to 1000 vertices. We find that the heuristic model shows exceptional performance in identifying a very large percentage of graphs of this size that do not contain a perfect matching (our tests indicate better than 99.9% on average), while never excluding any graph that does.

[1]Research partly supported by NSF grants CCF-0306475 and DUE-0313880.

Fast Filtering Heuristics for Perfect Bipartite Matchings

Rakesh M. Verma[2] and Jack Wiedrick

Computer Science Department, University of Houston, Houston, Texas, 77024

713-743-3348
713-743-3335 (fax)
rverma@uh.edu

Abstract

The bipartite matching problem has wide applicability. For example, in the computer science field, researchers have applied bipartite matching in objection recognition, image processing, scheduling, genomics, term rewriting and formal verification, and computer security. This list of applications is illustrative and is by no means exhaustive. We present a heuristic model of perfect matchings in bipartite graphs. The goal of the model is to explore the use of low-cost heuristics in applications where only perfect matchings are desired (as opposed to merely maximum cardinality matchings). The heuristics test for a number of conditions necessary for perfect matchings to exist in a graph, and as such, they can be used as filters to quickly identify and exclude unsuitable graphs from further processing. Whereas the best maximum cardinality bipartite matching algorithms run in about $O(n^{5/2})$ time (n being the number of vertices in the bipartite graph), we show that it's possible to make a reliable *prima facie* guess about the potential for a perfect matching in a given graph in roughly $O(n^2 \log n)$ time without bookkeeping costs. Although we represent bipartite graphs as square Boolean matrices and develop the heuristic model within the context of that abstraction, the heuristics can be adapted to other representations as well. Proofs of validity and runtime analysis of the heuristics are presented, along with a discussion of their performance characteristics in practice. A software framework is described that implements each of the components of the heuristic model and includes various test facilities for exercising them. Empirical results from the framework demonstrate the effectiveness of the heuristics for graphs with up to 1000 vertices. We find that the heuristic model shows exceptional performance in identifying a very large percentage of graphs of this size that do not contain a perfect matching (our tests indicate better than 99.9% on average), while never excluding any graph that does.

[2] Research supported in part by NSF grants CCF 0306475 and DUE 0313880.

1. The Problem

There are many situations where two groups of objects must be matched to each other in some optimal way. A classic example is boys and girls in a dance class: each student should have a partner to dance with and ideally the partner should be of the opposite sex. If we insist that the partners *must* be of the opposite sex then we have a bipartite matching problem. If we further insist that the partners be *compatible* in some fashion, we're forced to spend time querying each dancer for potential mismatches as we try to pair them up. A further complication arises if we require *every* dancer to have a partner, because we're no longer allowed to assume that a compatible match is viable for the needs of the class. It then becomes a perfect matching problem, and in order to solve it we usually have to at least *try* to pair up the girls and boys (i.e., execute some maximum cardinality bipartite matching procedure through to completion) before we can discover whether or not it's possible to find a compatible partner for every student in a given class. That fact makes this particular problem highly amenable to problem space filtering—if we could somehow "look" at a class and quickly discern whether a perfect matching between boys and girls was even likely, we could avoid all the effort of the matching procedure in hopeless cases.

This problem has broad application. We mention a few of the computer science applications to illustrate its applicability. In the computer science field, researchers have applied bipartite matching in objection recognition [12], image processing [], scheduling [], genomics, term rewriting and formal verification [13], and computer security. A conceptually simple example from the computing domain is the question of how to assign parallel threads of execution to different processors—based on each processor's availability or affinity for a certain type of work—so that the overall task is guaranteed to be completed by a given deadline. Another is in rule-based programming, where an input term must match the left-hand side of a rule completely, perhaps subject to associative or commutative transformations, before it can be reduced to the right-hand side.

To the best of our current knowledge on the problem, we are always forced to examine every compatible pairing at least once times a certain factor dependent on the size of the graph (roughly the square root of the vertex count; see section 3). But we would prefer to query each vertex *just once*, and based on the existence or absence of edges—and perhaps using cheap analytical tools like sorting and comparison—make a reasonably accurate guess about whether they can all be paired up or not. Such a procedure would constitute a heuristic model of the problem that could be used to filter input to a proper algorithm, hopefully reducing the computational cost of looking for solutions in the average case. In the remainder of this paper, we explore the components of one such heuristic model for perfect bipartite matchings.

2. Definitions

2.1. Graphs

A *graph* $G = (V, E)$ is a set of *vertices* in V connected by *edges* in E. A *subgraph* $G' = (V', E')$ as used in this paper is a subset of vertices $V' \subseteq V$ and all edges $E' \subseteq E$ incident on those vertices. A *bipartite graph* $G = (V_1 \cup V_2, E \subseteq \{V_1 \times V_2 \mid V_1 \cap V_2 = \varnothing\})$ is one whose vertices can be partitioned into two independent sets V_1 and V_2 such that no edges in E connect vertices in the same set. A *matching* M on G is a set of pairwise nonadjacent elements of E. Equivalently, M can be expressed as a function $F: V_1 \to V_2 = \{(u, F(u)) \mid u \in V_1, F(u) \in V_2\}$. We then call M a *perfect matching* if the corresponding function F_M is a bijection. If M is a perfect matching, then every nonempty subset of M is a *partial perfect matching* on G.

2.2. Matrices

Given an $m \times n$ *Boolean matrix* $A: \{1, 2, \Lambda, m\} \times \{1, 2, \Lambda, n\} \to \{0, 1\}$, m is the number of *row vectors* in A and n is the number of *column vectors*. (This defines row-major ordering.) The $p \times q$ *submatrix* A' of A, where $p \le m$ and $q \le n$, is defined by selecting the intersection of any p rows and q columns from A. (Note that the rows and columns need not be adjacent.) A is considered a *representative* of another $m \times n$ Boolean matrix B if B completely covers A, i.e., if $A \wedge B = A$, where the "$\wedge$" operation is the pointwise Boolean AND of the matrices. If A can be transformed into the identity matrix through some row-ordering operation, then A is a *matching representative* of B. (Note that both A and B must be square in this case.)

2.3. Vectors

For a *vector* $X = (x_1, x_2, \Lambda, x_m)$, let $|X|$ be the *size* of X and $X(i)$ be the *value* of the i-th element of X. The *count* of X, indicated here as $c(X)$, is defined as the number of nonzero entries in X. Given a function $f: \omega \to \omega$, we say X is f *-sparse* if $c(X) \le f(|X|)$. X is *less than* another vector Y if $X(i) < Y(i)$ for some i and $X(j) = Y(j)$ for all $j < i$. (This defines lexicographic ordering.) Given an $m \times n$ Boolean matrix A and row vectors (r_1, r_2, Λ, r_m) in A, we say that A is *row-sorted* if $r_i \le r_j$ whenever $i < j$.

3. Background

Bipartite matching is related to the maximum flow problem in two-stage networks and can be easily reduced to it. Ford and Fulkerson [4] showed that networks can be defined so that a maximum flow corresponds to a maximum cardinality matching in the corresponding bipartite graph. See Kennedy [7] for a good overview of this approach.

As a commonly encountered graph problem with a tractable solution, bipartite matching has many applications [2], but despite extensive research on the problem going back many decades (Hall's Theorem [5], below, was an early formulation), no one has discovered an algorithm with running time much better than $O(E\sqrt{V})$, which approaches $O(V^{5/2})$ as E approaches its worst-case bound of V^2. An algorithm with this bound was first discovered by Hopcroft and Karp [6], but was later refined logarithmically to $O(V^{3/2}\sqrt{E/\log V})$ in Alt, Blum, Mehlhorn, and Paul [1], and to $O(E\sqrt{V}\,\frac{\log(V^2/E)}{\log V})$ in Feder and Motwani [3].

Hall's Theorem*: Given a bipartite graph $G=(V_1\cup V_2,E)$, G has a perfect matching iff for every subset $S\subseteq V_1$, $|Adj(S)|\geq|S|$, where $Adj(S)$ denotes the set of vertices in V_2 that are adjacent to some vertex of S.*

While most work in the literature concentrates on developing faster ways to compute a maximum cardinality bipartite matching, we believe that inadequate attention has been devoted to developing ways to *avoid* such computation in situations where nothing less than a perfect matching will be useful.

4. A Heuristic Model

4.1. Encoding

Bipartite graphs are mathematically representable in many ways, but we have found the adjacency matrix representation of the graph's edge-space most convenient for our purposes because of the ease with which the component row and column vectors of a matrix can be sorted and compared. While inefficient for very large and/or very sparse graphs, the matrix representation is quite workable for small-scale graphs of around 1000 vertices or less, and have the added advantage of a straightforward and efficient software implementation, being typically stored in contiguous memory for ease of access via simple pointer arithmetic. Note, however, that although we frame our heuristic model in these terms, the matrix operations invoked in the operation of the heuristics are merely a front-end abstraction and can be trivially translated to any underlying graph representation desired.

Thus, we encode a bipartite graph $G=(V_1\cup V_2,E\subseteq\{V_1\times V_2\mid V_1\cap V_2=\varnothing\})$ as a Boolean matrix $G_{[]}:\{1,2,\Lambda\ ,|V_1|\}\times\{1,2,\Lambda\ ,|V_2|\}\rightarrow\{0,1\}$ where $G_{[]}(i,j)=\begin{cases}1 & (i,j)\in E\\ 0 & (i,j)\notin E\end{cases}$. We will only consider square matrices ($|V_1|=|V_2|$), since a perfect matching is impossible when that condition is not met. Throughout this paper we denote the size of $G_{[]}$ as g, where

$g = |V_1| \cdot |V_2|$, the number of potential undirected edges in G. Any matching $M \subseteq E$ on G can be similarly represented as a Boolean matrix $M_{[G]}$, defined analogously to $G_{[]}$.

4.2. The Goal

Given $G_{[]}$ as defined above, the problem we stated in section 1—that of making a *prima facie* determination as to whether the bipartite graph G could contain a perfect matching—is now reduced to the following question:

Does there exist a matrix that is a matching representative for $G_{[]}$?

If such a matching representative exists (see section 2.2), its existence would indicate that a perfect matching on G also exists. (For simple proof of this fact, consider that the identity matrix encodes a perfect matching between its rows and columns.) Note that we are *not* interested in identifying a specific matching representative per se, only in determining that one may exist. Furthermore, we wish to do so using only low-cost heuristic methods that run in time roughly linear in g and that never reject a matrix that does in fact have a matching representative. In other words, the heuristic model should encode only *easily discoverable* and *necessary* conditions for the existence of a matching representative for $G_{[]}$.

4.3. The Heuristics

1) *No empty columns.* For every column c_j of $G_{[]}$, $c(c_j) > 0$.
2) *Row-sorted and big enough.* When $G_{[]}$ is row-sorted (see section 2.3), each row of $G_{[]}$ should not be less than the corresponding row of a g-sized row-sorted identity matrix.
3) *Eliminate "forced moves".* If any k adjacent rows $r_i, r_{i+1}, \Lambda, r_{i+k-1}$ in $G_{[]}$ are "identical" (i.e., $r_i(j) = r_{i+1}(j) = \Lambda\ r_{i+k-1}(j)$ for all j in the domain), where $k \geq 1$ (rows are considered self-identical for this heuristic), and $c(r_i) = k$, then the set of rows in question encodes a partial perfect matching on G. $G_{[]}$ should be reduced to the submatrix $G'_{[]}$ formed by eliminating from $G_{[]}$ the k identical rows r_i and all columns c_j where $r_i(j) = 1$. After reduction, $G'_{[]}$ should be tested again. (Note that we also check for forced moves in columns, but only when $k = 1$, as checking when $k > 1$ would require a column-sort of the matrix; see sections 4.7 and 5.1.)
4) *No small neighborhoods.* Define a function f on the rows r_i of $G_{[]}$ so that $f(r_i) \leq |r_i|$ (e.g., $f(r_i) = \frac{|r_i|}{2}$). If any row in $G_{[]}$ is f-sparse (see section 2.3), then add it to an initially empty set R_S. For all rows $r_i \in R_S$ where $r_i(j) = 1$ for some j in the domain

(i.e., where row i has an edge to column j), add j to an initially empty set C if j is not already a member of C; when construction of C is complete, $|C| \geq |R_S|$.

5) *No deficient row or column sequences.* If any k adjacent rows in $G_{[]}$ are "identical"—as defined in heuristic (3) except that we disallow the $k = 1$ case (see section 4.7)—then $c(r_i) \geq k$; a similar property should hold for the columns of $G_{[]}$.

4.4. Examples of the Heuristics at Work

Consider the following 7 x 7 matrices:

$$M_1 = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \quad M_2 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \quad M_3 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{matrix}$$

$$\begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{matrix} \qquad \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{matrix} \qquad \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{matrix}$$

M_1 is a randomly generated matrix, M_2 is a row-sorted transformation of M_1, and M_3 is the row-sorted identity matrix of size $|M_1|$. Let's examine what happens when we run each heuristic to completion on M_2. First, by examining column 2, the reader should verify that there is no perfect matching of rows and columns in M_1.

Heuristic (1): It is easy to see that column 2 is empty in M_1 and M_2. Therefore, both M_1 and M_2 fail this heuristic. Note that, unlike the other heuristics in the set, row-sorting is unnecessary; a simple examination of columns will suffice. As we will see later, this becomes a big advantage as the matching problems grow in size.

Heuristic (2): Most rows of M_2 are at least as large as the corresponding rows of M_3, but since row 5 is not, M_2 will fail this heuristic. Remember that a row comparison between M_1 and M_3 would be meaningless; we must use the row-sorted M_2 instead.

Heuristic (3): Row 1 of M_2 contains a forced move since it has only one "1" and cannot be matched to any column other than 7. Therefore, we can remove row 1 and column 7 from consideration, yielding the submatrix:

$$M_2' = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Now rows 1 & 2 and columns 5 & 6 of M_2' form a combined forced move; the two rows and two columns are then removed to form:

$$M_2'' = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

All the rows of M_2'' now contain forced moves of one sort or another, but in order for the heuristic to run correctly we have to eliminate them one at a time:

$$M_2''' = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

Removing the first forced move also eliminated the second one! It's now clear that M_2 cannot contain a perfect matching, but the heuristic isn't smart enough to discover that yet. It sees that rows 2 & 3 and columns 1 & 3 contain forced moves, so they are removed:

$$M_2'''' = [0]$$

Since removing every forced move left us with no "1"s remaining, M_2 fails this heuristic. But note all the extra work involved in recursively constructing and testing submatrices. Moreover, sorting by rows is crucial if we want to detect the duplicate rows in linear time. We will discuss the impact of these facts later on (see section 5.1).

Heuristic (4): Initially, we can define a sparse row as one containing a single "1" or less. By this definition, rows 1, 4, & 5 of M_2 are sparse. Row 1 covers column 1 and row 4 covers column 4, but row 5 doesn't cover any additional columns, so two columns must somehow be split between three rows, an impossibility. Thus, M_2 fails this heuristic for this definition of sparsity. But if we had defined a sparse row as consisting of two "1"s or less, then every row except 3 is sparse. Those six rows cover six columns (1, 3, 4, 5, 6, & 7), so M_2 would pass the heuristic for that definition of sparsity. This shows that the effectiveness of heuristic (4) is sensitive to the overall density of the matrices being

considered. A value of around 50% (three “1”s or less for this example) seems to work well on random matrices.

Heuristic (5): There are two sets of duplicate rows in M_2: rows 4 & 5, and rows 6 & 7. In the first case, each row contains a single “1”, whereas in the second case each row contains two “1”s. Rows 6 & 7 can still both participate in a perfect matching if row 6 is matched to column 1 and row 7 to column 3 or vice versa, but rows 4 & 5 cannot. If either row claims its “1”, then the other cannot have it, so M_2 fails this heuristic. Like with heuristic (3), row-sorting is not theoretically required, but crucial for reducing time complexity.

4.5. Proof that the Heuristic Set is Valid

Now we prove five theorems (each corresponding to a heuristic above) that establish the validity of the heuristic set. Taken together, these theorems show that none of the heuristics will reject any matrix that has a perfect matching of rows and columns.

Theorem 1*: If a square matrix M has a perfect matching of rows and columns, then for every column c_j of M, $c(c_j)$ is required to be greater than zero.*

Proof: Trivial. ◊

Theorem 2*: If a row-sorted square matrix M has a perfect matching of rows and columns, then each row of M must be at least as large as the corresponding row of the row-sorted identity matrix I' of size $|M|$.*

Proof (by contradiction): Assume that M has a perfect matching of rows and columns and that there is a row r_i in M that is not as large as the corresponding row r_i' in I'. Two cases exist: Either $c(r_i) = 0$, or $c(r_i) > 0$. The first case is trivial: If one a row of M is empty, then there can be no matching column for that row and one of the columns of M must therefore go unmatched (remember that M is square), so the assertion is contradicted. In the second case, first form a partition of the columns of r_i' using the single “1” in the row as a pivot point. Call one partition L and include in L the column with the “1” and every column to the left of it. All columns to the right of it go into the second partition, R. Note that $c(r_i) \le |R|$ because $r_i < r_i'$. There are $i-1$ rows above r_i, and since M is row-sorted all of them are less than or equal to r_i. Thus, each of the rows above r_i can have at most $|R|$ nonzero entries. Moreover, because of the way the partition R was formed, $|R| = i-1$. But if M has a perfect matching, then the $i-1$ rows above r_i must be matched to $i-1 = |R|$ columns. Since r_i can only be matched to one of the same $|R|$ columns, one of the columns is matched to more than one row, which contradicts the properties of a perfect matching. ◊

***Corollary 1**: A row-sorted square matrix M can be compared to the row-sorted identity matrix I' of the same size to determine the possibility of a matching in M.*

The following lemma is useful in proving the next theorem.

***Lemma 1**: If a bipartite graph $G = (V, E)$ contains a perfect matching $M \subseteq E$, then any nonempty subset $M' \subseteq M$ is a perfect matching on the subgraph G' induced by M'.*

Proof: M' is a matching because M is a matching. Since M' is nonempty, it contains at least one edge, and this edge is a perfect matching between the two endpoints of the edge. If M' contains more than one edge, then the edges are nonadjacent because M' is a matching. Thus, every edge in M' connects two distinct vertices in G'. Since the edges in M' cover all vertices in G' (by construction), M' is a perfect matching. ◊

***Theorem 3**: If a square matrix M has a perfect matching of rows and columns, and a submatrix $M' \subseteq M$ contains a single unique perfect matching, then the portions of M not covered by M' must also contain a perfect matching disjoint from the one in M'.*

Proof: We prove a more general result. Assume that a set $S = \{R \times C \mid R \cap C = \varnothing\}$ contains a perfect matching between R and C. Partition R into two disjoint sets R_1 and R_2 such that R_1 can only be matched to a set $C_1 \subseteq C$ and to no other subset of C. By Lemma 1, such a matching will always exist. Now define the set $C_2 = C - C_1$. Since the members of R_1 can only be matched to the members of C_1, the matching between R_1 and C_1 must be present in any perfect matching on S. Moreover, if R_2 cannot be matched perfectly to C_2, then S cannot contain a perfect matching. Therefore, R_2 and C_2 must also have a perfect matching, which by construction is disjoint from the one between R_1 and C_1. ◊

***Corollary 2**: If S contains a perfect matching, that matching can be found by first finding a matching on $R_1 \times C_1$ and then finding a matching on $R_2 \times C_2$.*

***Corollary 3**: If C_2 is always empty regardless of how R_1 is constructed, then S contains $|R|! = |C|!$ different perfect matchings.*

Proof: R is partitioned such that R_1 can only be matched to C_1 and not to any other subset of C. Thus, if all elements in R must be used in order to achieve that result, it follows that every element of C can be matched in $|R|$ ways, and therefore that $|R|!$ perfect matchings exist; $|R| = |C|$ because S contains a perfect matching, so $|R|! = |C|!$. ◊

***Theorem 4**: If a square matrix M has a perfect matching of rows and columns, then the nonzero entries in any set of k rows in M must cover at least k distinct columns.*

Proof: This is a restatement of Hall's Theorem (see section 3). ◊

Theorem 5*: If a square matrix M has a perfect matching of rows and columns, then the number of duplicated identical rows in M cannot be greater than the number of nonzero entries in each identical row. A similar theorem holds for the columns of M .*

Proof (by contradiction): Assume that M contains a perfect matching and is comprised of $k+n$ identical rows with k nonzero entries in each row, where $n>0$. After matching k of the rows to the corresponding k columns that contain nonzero entries, n rows are left unmatched. Each of the rows in M has only k nonzero entries, and since each of those entries has been matched to one of k columns, the remaining entries in the unmatched rows are zero entries, which cannot be matched to any column, so n rows will be left out of any matching, which contradicts the assertion. The proof for columns is similar. ◊

Corollary 4*: The rows in question need not be identical. If only k columns must be matched to $k+n$ rows, there cannot be a perfect matching unless $n=0$.*

4.6. Independence of the Heuristics

Establishing a clear picture of the implicational relationships between the heuristics in the set is an ongoing research goal, made difficult by the many combinatorial possibilities. (In the 7 x 7 case, for example, there are 2^{49} = 562,949,953,421,312 possible Boolean matrices, and even if 99.9% of those are unsorted and thus removed from consideration, well over 500 billion possibilities remain.) Here, we present our findings so far.

When we say a heuristic A is *independent* of heuristic B, we mean that A will correctly filter a *class* of matrices of arbitrary size where B will not. This definition differs from strong independence, or true orthogonality. Note that in the above sense, none of the heuristics in our set appears to be independent from the others for very small cases (matrices 4 x 4 or smaller), so we will restrict our attention to 5 x 5 and larger matrices. As a last caveat, heuristic (4) shows different characteristics depending on the cutoff density value for sparsity (see section 4.4). In the constructions we assume a constant value of 50% and leave it as an exercise to extend the results to other values.

Heuristic (1) is the most independent of the set since it considers only column states, whereas the others focus primarily on row states. We define a distinguishing class of matrices for this heuristic using the following construction:

1) Begin with a zero-filled square $n\times n$ matrix, where $n>4$.
2) Fill the bottom row with "1"s except in the rightmost column, which will retain all "0"s.
3) The remaining rows must each contain exactly $n-2$ "1"s, arranged so that no two rows are identical. Here is a 5 x 5 example:

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Heuristic (3) can also be distinguished with a relatively simple construction. The idea is to reduce matching possibilities in such a way that the other heuristics are "fooled":

1) Begin with a zero-filled square $n \times n$ matrix, where $n > 4$.
2) Fill the bottom row with "1"s.
3) Place a single "1" in the rightmost column of the top row.
4) The leftmost column will retain all "0"s except in the bottom row.
5) Place "1"s in rows 2 through $n-1$ so that each of the rows in question contains at least two but no more than $n/2$ "1"s, configured so that there are no more than k identical rows for each pattern containing k "1"s; furthermore, for each row r_i, some entry $j \leq n-i+1$ in that row must contain a "1". Here is a 5 x 5 example:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

We have been unable to construct simple algorithms defining distinguishing classes for the other heuristics, and the only examples we have been able to find are for matrices 7 x 7 or larger for heuristic (4) and 6 x 6 or larger for heuristic (5).

Below is an example of a 7 x 7 matrix distinguishing heuristic (4). Smaller examples cannot be found because of the requirement that several rows be sparse (i.e., 50% "1"s or less) without creating any empty columns, duplicate rows or allowing the rows to become too small. Furthermore, we must have k sparse rows that cover only $k-1$ or fewer columns. In the following example, we have managed to create four sparse rows at the top of the matrix that meet those requirements:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Below is an example of a 6 x 6 matrix distinguishing heuristic (5). The trick is to create three identical columns containing only two “1”s each, arranging the rest of the matrix so as to fool the remaining heuristics:

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 \end{bmatrix}$$

We have yet to discover a matrix that heuristic (2) will filter when the others will not. It is possible that heuristic (2) is not independent in the sense mentioned above. We can show that heuristic (2) is *pairwise* independent from each of the others, but in the cases we have tested so far, the combination of heuristics (1), (3), (4), and (5) seems to be sufficient to compensate when (2) is missing. However, we retain (2) because the system performs faster when it’s included than when it’s not.

4.7. Additional Heuristics

In the course of testing and developing the heuristic set, we considered another heuristic that later proved redundant in the system:

6) *No empty rows.* For every row vector r_i of $G_{[]}$, $c(r_i) > 0$.

It turns out that heuristic (6) is logically implied by the more powerful heuristic (2)—since the row-sorted identity matrix contains a “1” in every row, and in order to pass heuristic (2), the rows of a matrix must be at least as large as those of the row-sorted identity matrix, all rows must therefore contain at least one nonzero entry. But the converse case is also interesting. In combination with the other heuristics, heuristic (6) can replace (2) with no apparent loss of effectiveness. As expected, using the simpler heuristic (6) leads to modest speedups (it doesn’t require the relatively expensive row-sorting operation), but unlike (2), heuristic (6) is not pairwise independent of (4)—easy to see by noting that empty rows are always sparse yet contribute nothing to column coverage. Eliminating the redundancy means reworking heuristic (4) to have it view empty rows as non-sparse. Not only is this against the spirit of (4), it complicates and slows its operation by requiring a special-case test for every row examined. Furthermore, the minor speed gains that can be achieved by replacing heuristic (2) with (6) may disappear for large matrices—our tests seemed to indicate declining returns from (6) as matrix size was increased. For all these reasons, heuristic (6) was eliminated from the set, but the implementation was retained in the test framework (see section 6.2) so users would have the option of using it in their own custom tests.

Astute readers might also wonder about the different definitions of “identical” in heuristics (3) and (5). If columns were allowed to be self-identical in heuristic (5), then heuristic (1) would become subsumed by heuristic (5)—a single empty “duplicate” column would indicate the absence of a perfect matching. The problem is that heuristic (5) requires an *extra* sorting step in order to locate duplicate columns, and sorting is expensive compared to heuristic (1), which is a simple linear search with typical early termination (see section 5.1). A similar argument holds for rows, as empty rows are more rapidly detected by heuristic (2) instead. Thus, the more general version of heuristic (5) was rejected because allowing heuristics (1) and (2) to run before (5) has the effect of eliminating many matrices from consideration by the costlier (5).

4.8. Synthesis

The above heuristics reflect conditions necessary for any matrix to have a perfect matching of rows and columns, so a matrix that fails one of them is guaranteed not to have a perfect matching. But the converse is not true: A matrix may meet all the above conditions and still not contain a perfect matching. This can be demonstrated empirically through application of the heuristics on actual matrices. For example, the following 7 x 7 matrix passes all of the heuristics for any definition of “sparse row”, but encodes no perfect matching:

$$\begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \quad \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{matrix}$$

(Rows 1, 4, 5, & 6 cannot all be matched.)

Nevertheless, the heuristic model proves to be an exceptionally reliable and relatively low-cost approximation of perfect matchings, making it ideal for use as an input filter to true matching algorithms in applications where only perfect matchings are desired.

5. Performance

5.1. Runtime Characteristics

The time requirements for most of the heuristics can easily be shown to be close to linear in g in the worst case (recall that g is the number of potential edges in a graph; see section 4.1). Here we present detailed runtime analyses for each of the heuristics in turn:

1) *No empty columns.* Checking each column of a matrix for the presence of nonzero entries requires only $O(\sqrt{g} \cdot \sqrt{g}) = O(g)$ accesses in the worst case.
2) *Row-sorted and big enough.* Using an optimal algorithm, sorting the rows of a matrix requires $O(\sqrt{g} \cdot \log \sqrt{g})$ row comparisons, where each row entails at most $O(\sqrt{g})$ element comparisons. Thus, the worst case for the sorting stage of this heuristic is $O(\sqrt{g} \cdot \sqrt{g} \cdot \log \sqrt{g}) = O(g \log \sqrt{g})$, which is only slightly worse than linear in g. For the second stage, we note that while comparing row values between two same-size matrices requires $O(\sqrt{g} \cdot \sqrt{g}) = O(g)$ comparisons in general, when the rows of the matrices are sorted and the algorithm breaks off at the first mismatch, the average time of failure is closer to $O(\sqrt{g})$—the cost of comparing two single rows—because the smallest rows will be examined first. So sorting makes this stage faster, although the worst-case performance for both stages combined is still $O(g + g \log \sqrt{g})$.
3) *Eliminate "forced moves".* Finding duplicate rows cannot be done in linear time if the rows of the matrix are unsorted. When they are sorted, it requires comparing a maximum of $\sqrt{g}$ rows in a single pass, with each comparison taking at most $O(\sqrt{g})$ accesses; thus, $O(g)$. However, eliminating forced moves also requires recursively constructing a series of possibly noncontiguous submatrices for further testing, which will add considerably to the time and space complexity. In the typical case of simply constructing a new submatrix using copy operations each time, the construction will take $O(g)$ time and may be performed as many as $\sqrt{g}$ times; even worse, duplicate rows must be rediscovered each time, which requires more sorting operations. Thus, the total time complexity for applying this test recursively to problem completion is $O\left(\left(g + g \log \sqrt{g}\right) \cdot g \cdot \sqrt{g}\right) = O\left(g^2 \sqrt{g}\left(1 + \log \sqrt{g}\right)\right)$, which is much worse than linear. But as expensive as this test is on its own (performing a full matching from the start is actually cheaper), we have seen that when combined with the other tests and run relatively late in the sequence, it will only run on a fraction of the candidate matrices because most matrices without a matching will fail at an earlier step (see section 5.4). Additionally, reducing the matrices both makes them smaller and changes their global structure sufficiently that in many cases other, cheaper heuristics can filter out the new submatrices before this one has a chance to run again. In practice, this heuristic seems to have only a limited impact on the runtime of even large problems, but the average-case performance is difficult to quantify precisely.
4) *No small neighborhoods.* The time required will depend on how many rows are chosen for the "sparse" subset. In the worst case, we choose all rows and examine each element of each row, yielding $O(\sqrt{g} \cdot \sqrt{g}) = O(g)$. This heuristic performs slightly better when the matrix is row-sorted, because all the sparsest rows will group together at the top of the matrix. (The sort comes free when heuristic (4) is run at any point after heuristic (2), because the latter always performs a row-sort.)
5) *No deficient row or column sequences.* The complexity of finding deficient row sequences is the same as heuristic (3) for the first stage ($O(g)$ to find duplicate rows), but for the second stage, the matrix must be *re-sorted* by columns, yielding a

complexity of $O(g + g \log \sqrt{g})$. However, the cost of this heuristic is generally linear in practice because small duplicate rows are searched for first. If any are found, then the procedure terminates early without needing to re-sort and search for small duplicate columns.

5.2. Random Graph Model

Exhaustive enumeration of matrices is feasible for small cases, but we were forced to rely on a random graph generator for performance testing of the heuristics against matrices of larger dimensions. Our approach combines a pseudorandom number generator with a simple, user-modifiable density function (see section 6.2). We populate each graph with enough randomly generated edges to approximate the specified density, expressed as a percentage of the edge-space. The density function can also be randomized, yielding sets of graphs with no fixed density bound. Furthermore, the appearance of any graph in a given enumeration of graphs over a range of "values" (edge configurations) can itself be randomized according to a weighted probability function that simulates the right half of a standard distribution curve—enumerating most of the graphs near the beginning of the range (where they are sparser) and relatively less toward the end (where they are denser).

We chose Knuth's recommended "portable" pseudorandom number generator [8], which is based on the subtractive recursive polynomial $X_n = (X_{55} - X_{24}) \bmod m$, where m is some sufficiently large value (we follow Knuth in using 10^9). This generator has a fairly large period (on the order of 2^{55}) compared to the linear congruential generators typically used to implement the C Standard Library $rand()$ function, and is also very fast in practice, using only a single subtract and multiply for most iterations. Our implementation performs two cranks for each generation cycle to produce a pair of values for indexing a matrix.

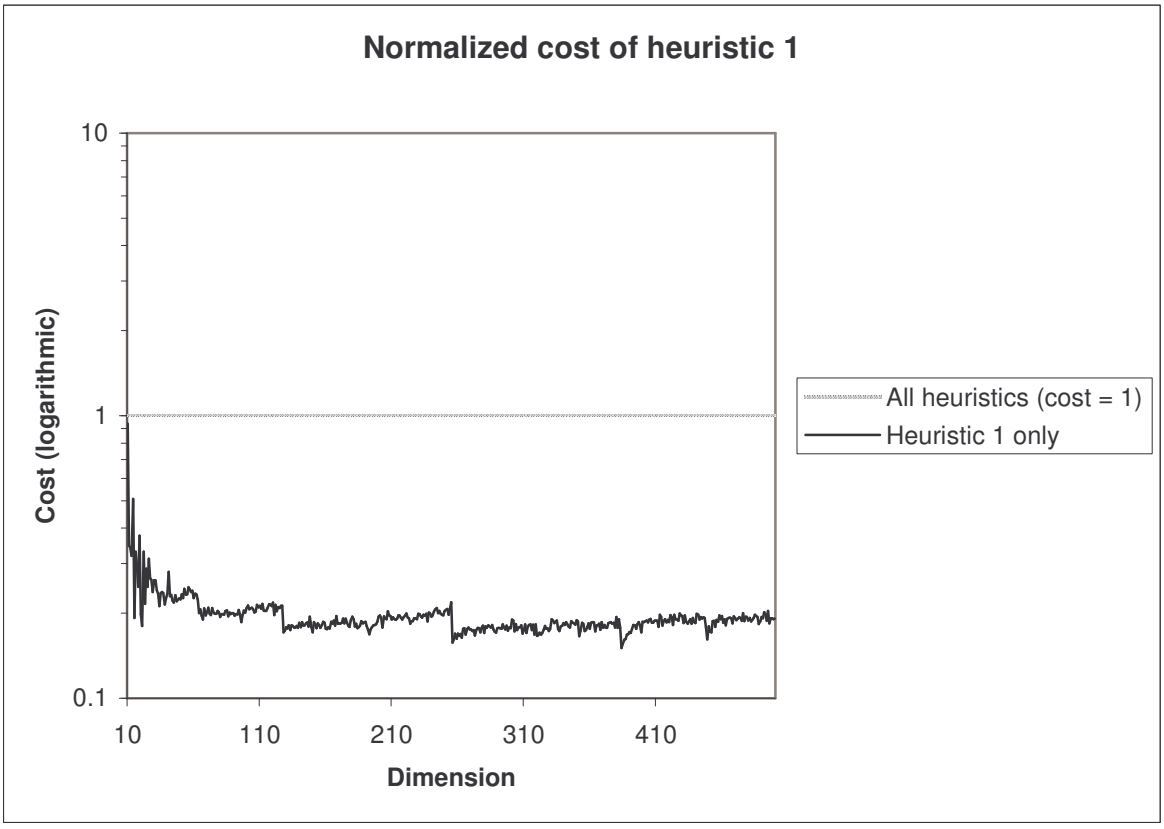
For the density function we use the equation $|E| \approx n^2 d$, where n represents one dimension of the edge-space and can be any value in the range $[1..\sqrt{g}]$, and d is a density value between 1% and 100%. The equation is approximate owing to the possibility that some edges will be generated more than once for the same graph.

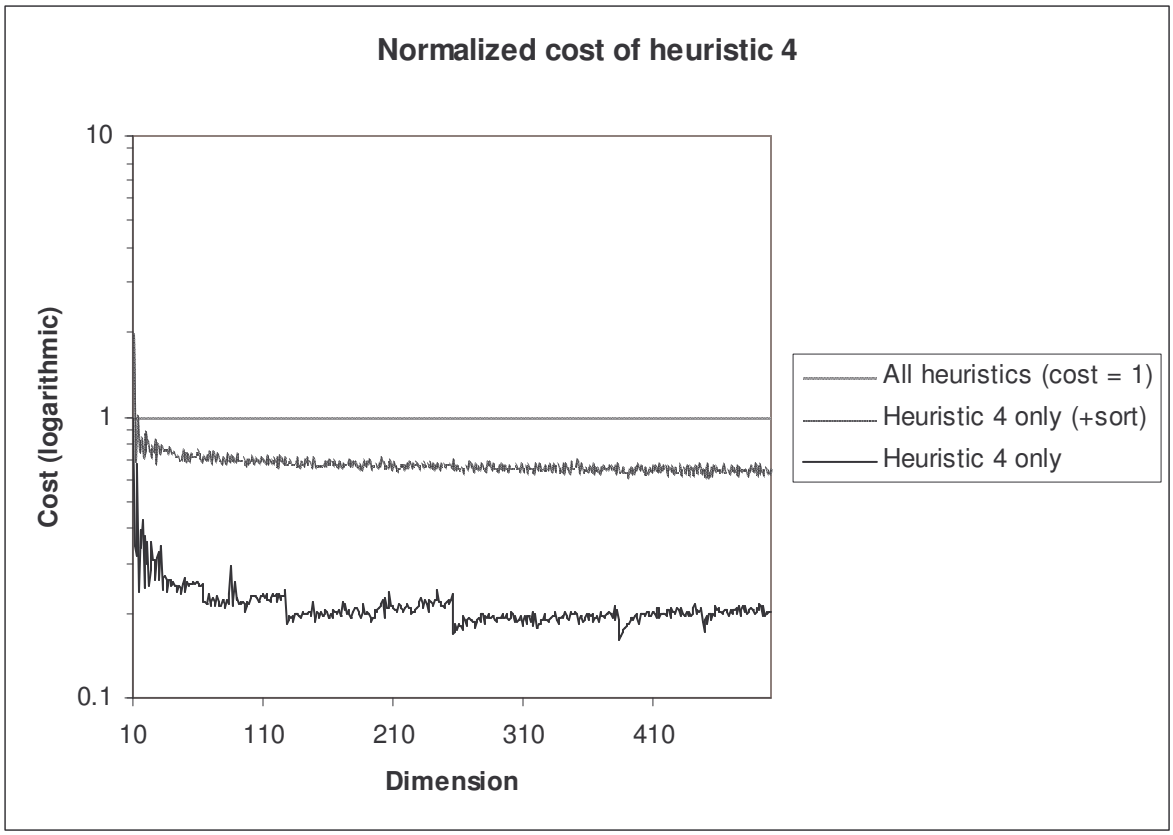
5.3. Observed Runtime Costs

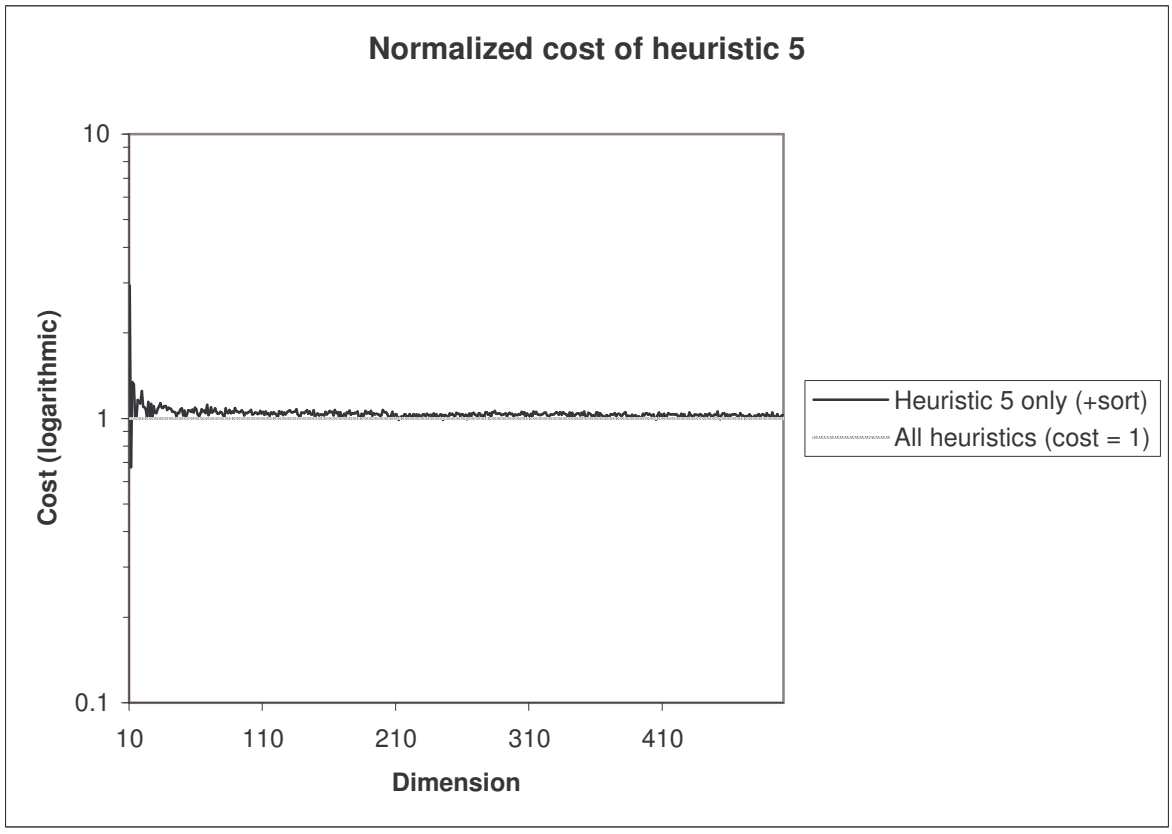
Based on the analysis in section 5.1, we can surmise that running all the heuristics in sequence (assuming a linear average-case runtime for heuristic (3) for ease of discussion) will require a worst-case time of roughly $O(5g + 2g \log \sqrt{g}) = O(g \log \sqrt{g})$, which suggests that the sorting time dominates. Since sorting is cheaper than matching (which is around $O(g^{5/4})$ with much higher bookkeeping costs; see sections 3 and 6.4), it

always makes sense to run these heuristics beforehand to determine whether a matrix might contain a perfect matching before attempting to compute one.

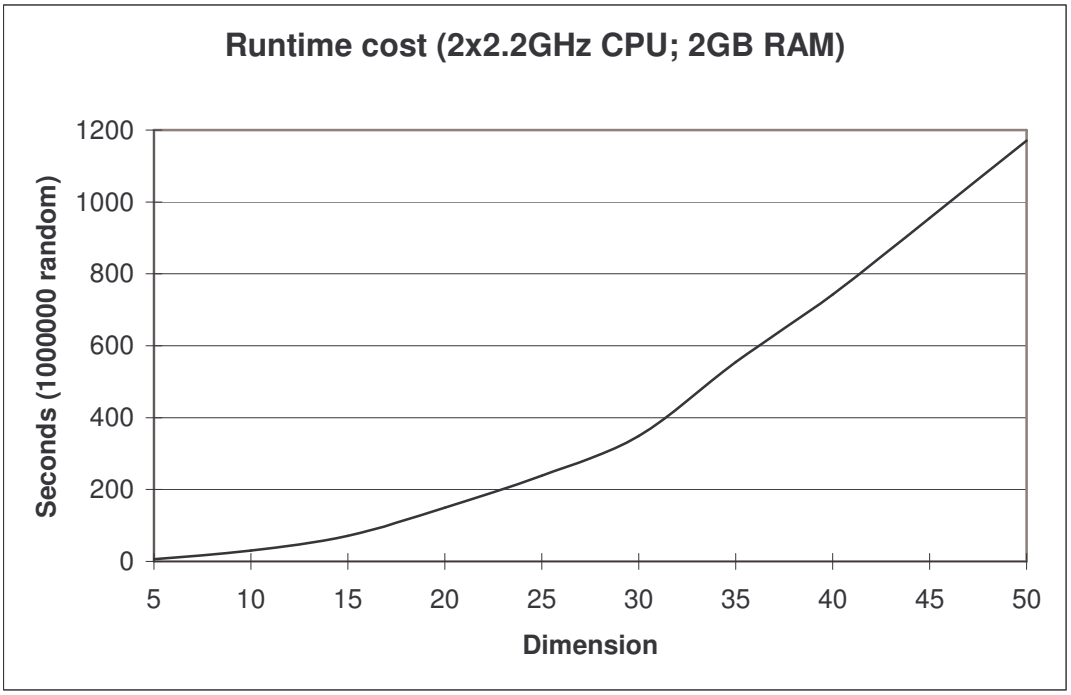
Below are a series of figures showing the normalized cost of each heuristic compared to the performance of all the heuristics running in sequence on 1000 random matrices for each dimension in the range 10 x 10 through 500 x 500. They largely bear out the analysis of section 5.1, showing that heuristics (1) and (4) (without a sort) are very cheap, that the sorting costs of heuristics (2) and (5) approximate the aggregate cost, and that heuristic (3) is wildly expensive when run alone. Further research is needed to determine the precise runtime impact of heuristic (3) in practice (i.e., when not run alone).




Normalized cost of heuristic 4
Cost (logarithmic)
10
1
0.1
10
110
210
310
410
Dimension
All heuristics (cost = 1)
Heuristic 4 only (+sort)
Heuristic 4 only


Normalized cost of heuristic 5
Cost (logarithmic)
10
1
0.1
10
110
210
310
410
Dimension
Heuristic 5 only (+sort)
All heuristics (cost = 1)

A further series of runs on one million random matrices of dimensions in the range 5 x 5 through 50 x 50 was timed on a single machine (dual 2.2 GHz processors, 128 KB L1 + 512 KB L2 cache memory per core, 2 GB 2 x 200 MHz DDR RAM on a 5 x 200 MHz front-side bus), using identical operating conditions for each run and restarting each time to minimize cache effects. The resulting relation between dimension and absolute runtime, shown below, confirms that the runtime cost is roughly quadratic with respect to dimension, i.e., linear in the size of the matrix.



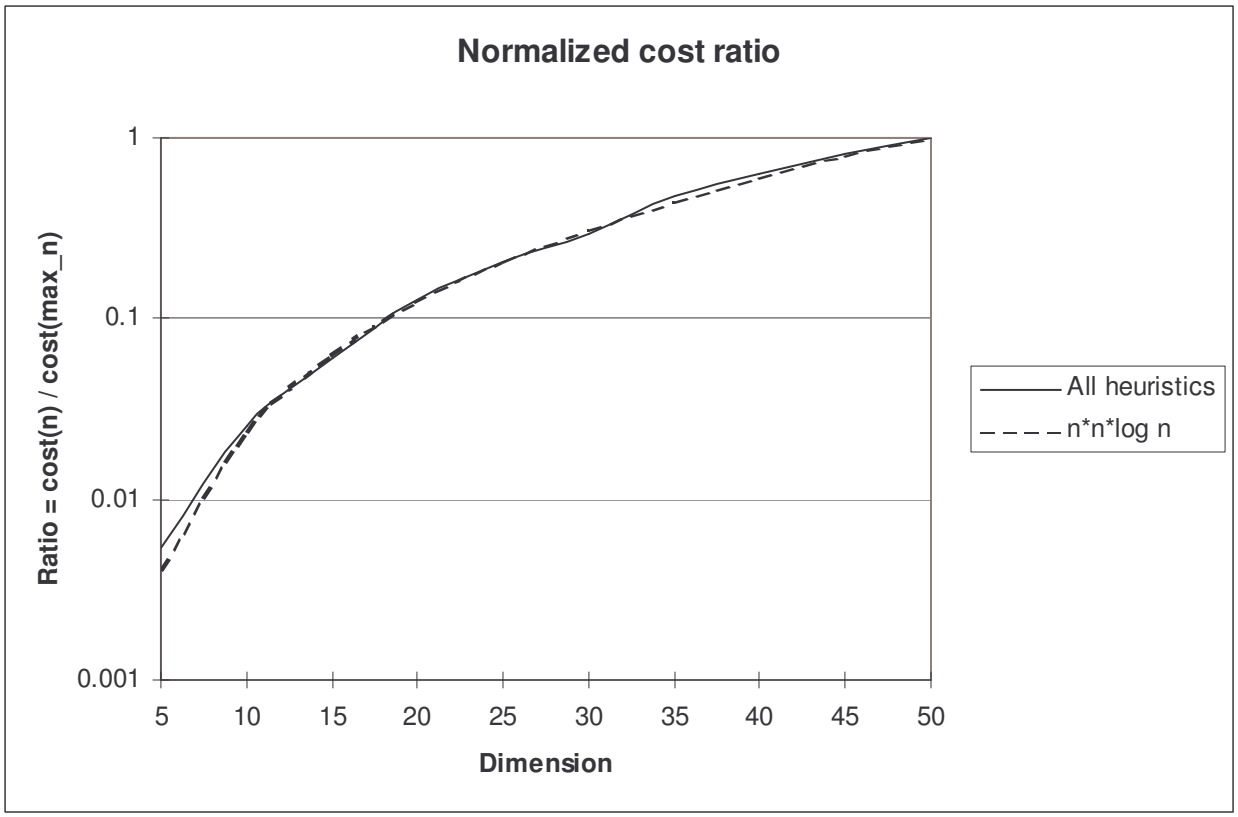
To further compare the observed runtime performance with our asymptotic analysis in section 5.1, we plotted the values from the previous figure as a series of ratios between the observed runtime cost at each dimension and the observed runtime cost at the highest dimension tested (50 x 50); this expresses the growth of the function as a normalized curve. We then plotted the normalized curve of $\underset{n=5\to 50}{f(n) = n^2 \log n}$ in the same fashion and compared it to our observed growth. As the figure below shows, the curves match almost exactly, providing good empirical confirmation of the predicted cost of running the full suite of heuristics in sequence.

$$observed(5x5)/observed(50x50) = 6/1200 = 0.005 \approx 5^2 \log_2 5 / 50^2 \log_2 50 = 0.0043$$

$$observed(25x25)/observed(50x50) = 240/1200 = 0.2 \approx 25^2 \log_2 25 / 50^2 \log_2 50 = 0.21$$

$$observed(45x45)/observed(50x50) = 950/1200 = 0.791 \approx 45^2 \log_2 45 / 50^2 \log_2 50 = 0.792$$

$$etc.$$



5.4. Empirical Evaluation

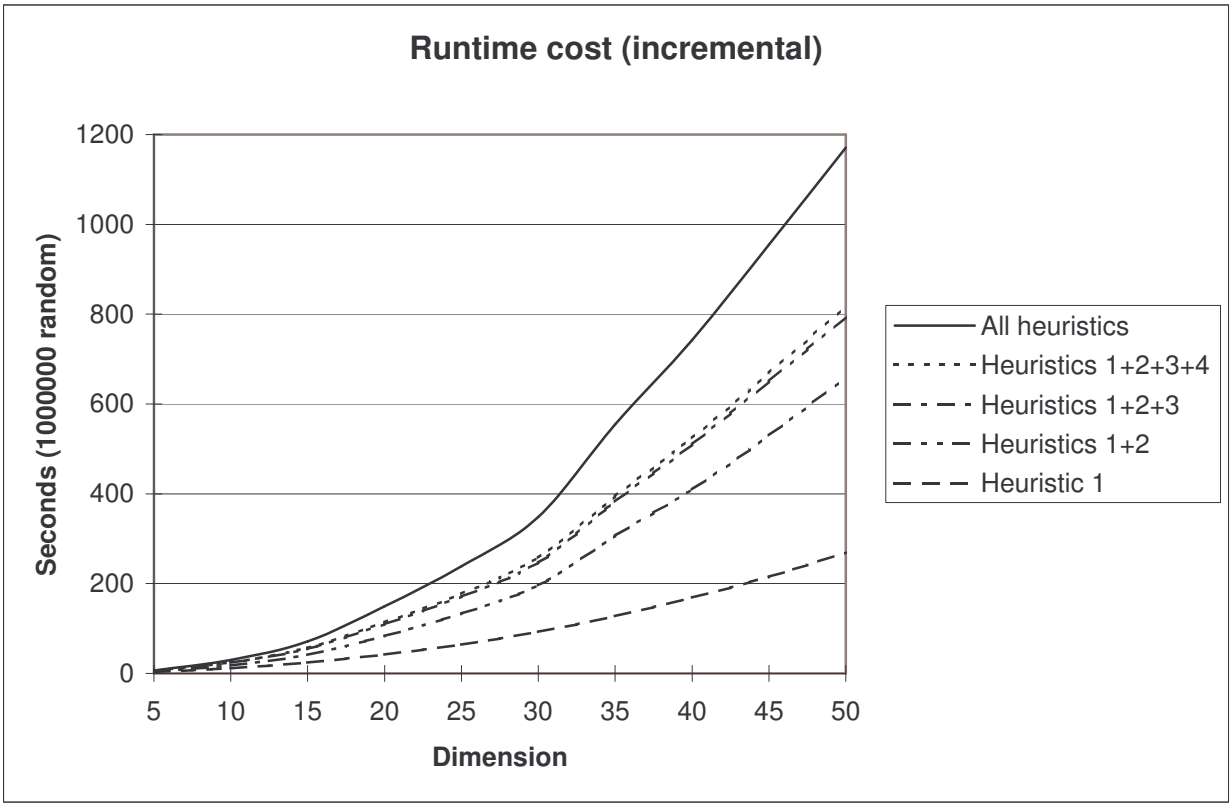
The order of the heuristics has been carefully chosen to provide the fastest possible runtimes for the system while still retaining maximum filtering power, as verified through empirical testing (see below). Placing heuristic (1) before (2) leads to the best speedups because (1) can filter out large numbers of candidates without needing to sort them.

Relative running times on 10^6 random 10 x 10 matrices (in order of fastest to slowest)

Test Order	Matrix Count	Time	Matrices Processed Per Second
102345	1000000	32.953	30346
102435	1000000	33.343	29991
102534	1000000	38.546	25943
021345	1000000	41.093	24335
041235	1000000	42.312	23634
051234	1000000	49.000	20408
031245	1000000	67.203	14880

Heuristics (1) and (2) should run first as they are the simplest and fastest. Altering the relative ordering of (3), (4), and (5) seems to have only minimal impact, but the extra column-sort required by (5) (see section 5.1) means that it runs slower than the other heuristics in typical cases, and should generally come last. Heuristic (3)'s bad worst-case time only kicks in when it runs on its own; it's faster than both (4) and (5) when (1) and (2) run first. When the cost of the heuristics is broken down incrementally (see chart

below), the largest jumps in cost are for (2) and (5)—in both cases because of a sorting step.



Exhaustive test runs for dimensions 1 x 1 through 6 x 6 prove that these heuristics are a *perfect* approximation of the matching algorithm in the case where the matrix being tested is 6 x 6 or smaller. (We know from examples like the one in section 4.8 that the approximation is not perfect in the case of matrices 7 x 7 or greater.)

Exhaustive test results for 1 x 1 – 6 x 6 matrices

Dimension	Matrix Count	Filtered With No Match	Time (sec)	Total Matrices Trapped By					
				Test 0 (unsorted)	Test 1	Test 2	Test 3	Test 4	Test 5
1 x 1	2	100%	< 1			1			
2 x 2	16	100%	< 1	6	5	1			
3 x 3	512	100%	< 1	392	49	16	4		
4 x 4	65536	100%	0.062	61660	1129	520	229		
5 x 5	33554432	100%	20.047	33177440	70131	41630	24049		
6 x 6	68719476736	100%	34194.600	68599599264	13167845	9314089	5791150	2416	320

We ignore unsorted matrices for the exhaustive test runs because sorting and then testing them would only create redundant work for the program, as well as skew the counts. (The single bad 1 x 1 matrix was filtered under “Test 2” rather than “Test 1” because it is treated as a special case in the code.)

After executing a series of runs of dimensions from 7 x 7 up through 20 x 20 using a reasonably large sample size (100,000 random matrices from each dimension), we

found that well over 99.9% of matrices without perfect matchings were consistently being filtered by the heuristics (mainly the first three).

Performance results for 7 x 7 – 20 x 20 matrices (10^5 random each)

Dimension	Matrix Count	Filtered With No Match	Time (sec)	Total Matrices Trapped By				
				Test 1	Test 2	Test 3	Test 4	Test 5
7 x 7	100000	100%	1.812	60235	9126	3093		
8 x 8	100000	99.997%	2.407	58429	8636	3097		
9 x 9	100000	99.991%	3.093	56886	8365	2970		
10 x 10	100000	99.985%	3.844	55233	7788	2896		1
11 x 11	100000	99.983%	4.703	53560	7513	2702		2
12 x 12	100000	99.984%	5.469	52347	7165	2548		
13 x 13	100000	99.990%	6.422	50831	6857	2400		
14 x 14	100000	99.971%	7.484	49269	6683	2318		
15 x 15	100000	99.977%	8.516	48823	6397	2076		
16 x 16	100000	99.978%	10.484	47117	6220	2032		
17 x 17	100000	99.960%	11.938	46652	5989	1898		1
18 x 18	100000	99.977%	13.640	45207	5791	1853		
19 x 19	100000	99.983%	15.297	44595	5642	1716		
20 x 20	100000	99.980%	17.016	43299	5455	1577		

What these results show is that the heuristics perform very well on small matching problems. Running times are also impressive, with near-linear slowdown as the problem size increases: a 20 x 20 matrix is four times larger than a 10 x 10 matrix, and takes just over four times longer to process.

Heuristic (4) was an interesting case. Although it was disappointingly weak at filtering the output of the earlier heuristics, on its own (with a “sparse” setting of 50%) it was found to be quite powerful, with better than 83% effectiveness in the 5 x 5 case (exhaustive) and 88% on a random selection of one million 20 x 20 matrices—actually improving as the problem size grows. In the random 20 x 20 test case just mentioned it proved over three times faster (without a sort) than running the complete sequence, and with only an 11% drop in effectiveness. On 1000 random 500 x 500 matrices, it was five times faster with only a 1% drop in effectiveness.

Performance results for heuristic (4) in combination and alone

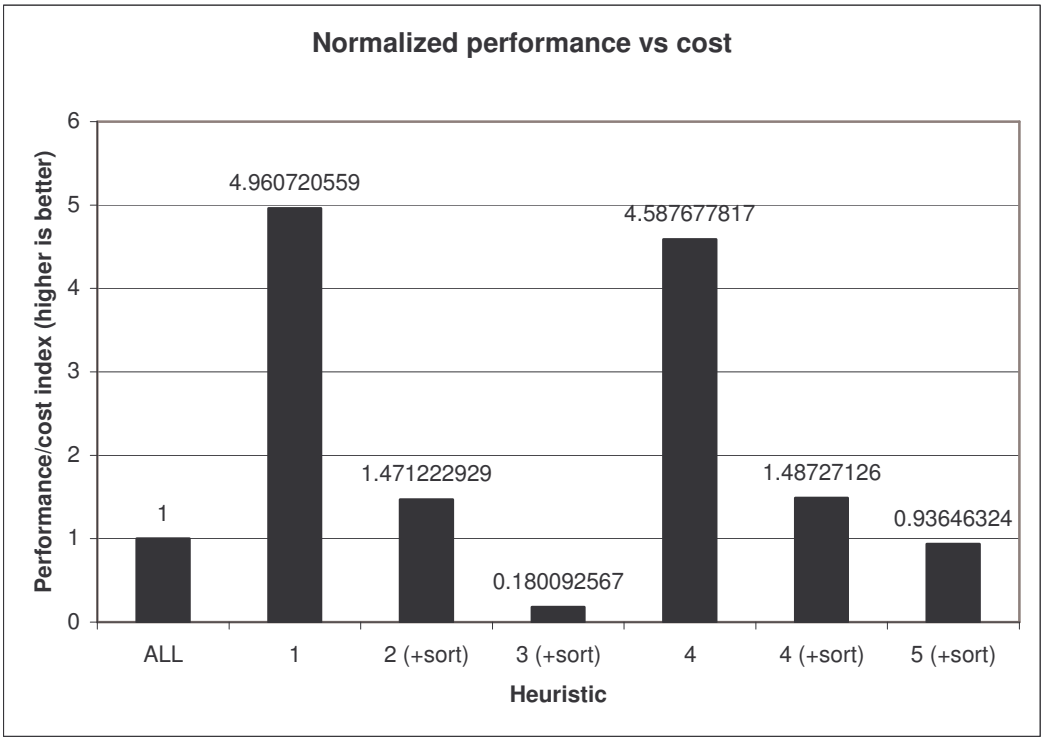
Dimension	Tests	Matrix Count	Filtered With No Match	Time (sec)	Total Matrices Trapped By				
					Test 1	Test 2	Test 3	Test 4	Test 5
5 x 5	4	33554432	83.492%	249.296				5778026	
5 x 5	102345	33554432	100%	474.812	4925281	3956520	1481560		
7 x 7	102345	1000000	99.999%	18.093	600817	92622	30492	10	4
20 x 20	4	1000000	88.191%	49.750				440369	
20 x 20	102345	1000000	99.978%	158.984	435386	54967	15990		
500 x 500	4	1000	99.143%	40.906				66	
500 x 500	102345	1000	100%	203.484	66	8			

When run alone, each of the heuristics seems to improve in effectiveness on random graphs as the problem grows, but running (5) alone takes about the same amount of time as running all of the heuristics together, and (3) alone is orders of magnitude slower. Unsurprisingly, (1) achieves the fastest solo times, although its effectiveness lags just marginally behind (4). But when (1) is combined with the rejected heuristic (6) (see section 4.7), the combination outstrips (4) in both speed and effectiveness.

Performance results on 10^6 random 20 x 20 matrices for heuristic (1) alone and in combination with the rejected "no empty rows" heuristic (6)

Dimension	Tests	Matrix Count	Filtered With No Match	Time (sec)	Total Matrices Trapped By	
					Test 1	Test 6
20 x 20	1	1000000	87.413%	43.312	435386	
20 x 20	16	1000000	96.754%	41.390	435386	54511

In short, these tests demonstrate that the solo performance/cost ratio is very good for the simpler heuristics that don't require sorts, and average to poor for the more complex ones that do.



For tests of matrices of larger dimensions (up to 500 x 500 have been tested), the heuristics as a group continue to catch and filter more than 99.9% of random matrices that do not encode a perfect matching, so we conclude that they are a reliable and reasonably low-cost approximation of perfect matchings in small to medium-sized bipartite graphs, and therefore a fast and effective filtering strategy for applications where only perfect matchings are desired.

6. Test Framework Software

6.1. Overview

A software framework ("adjmat", short for "*adj*acency *mat*rix") for testing the heuristic model was written in C++. The framework contains powerful and efficient implementations of Boolean vector and matrix data structures, allowing adjacency matrices of arbitrary dimension to be generated and passed through the heuristics in any order. Matrices can be exhaustively or manually enumerated for dimensions up to 8 x 8 (the largest Boolean matrix that will fit into a 64-bit integer), and randomly enumerated for any dimension. Reporting capabilities include options to output all matrices considered, or only unfiltered ones, along with matchings when they exist. Various flag settings and statistics on the effectiveness of the heuristics can also be output. Complete source code is available from the authors upon request.

6.2. Command Line Interface

adjmat {-? | dimensions [options]}

-? Usage help
If this flag is present as the first argument, then a concise help screen is displayed. If it appears in any other position, then it is ignored.

dimensions
This is a required parameter indicating the dimensions of matrices to test. Unless the –x flag is also set, a single value means that matrices from dimension 1 x 1 up to the specified dimension will be considered. (E.g., "adjmat 3" prints all matrices from 1 x 1 to 3 x 3.) A range of values can be specified using comma or hyphen notation. (E.g., "adjmat 2,4" prints all 2 x 2 and 4 x 4 matrices, while "adjmat 2-4" prints 2 x 2, 3 x 3, and 4 x 4.) The –x flag is automatically set when a custom range is specified. The value of <dimensions> must be more than "1".

-v Verbose mode
This will output the status of all flags, format parameters, and time statistics.

-x Process specified dimensions exclusively
This overrides the default behavior of enumerating matrices of dimension up to the given dimension. When set, only the provided dimensions are considered.

-i Ignore unsorted matrices
If this flag is set, then all enumerated matrices are first checked to see if they are row-sorted, and if not, then they are removed from further consideration. When the –f flag is set, use –i to override the default behavior of sorting all matrices.

-f Apply filters to the matrices

Unless this flag is set, the specified matrices will merely be enumerated. Set this to use the heuristic test framework.

-p Prove filter validity (no effect if –f is not also set)
If this flag is set, a matching will be attempted on every matrix caught by one or more of the heuristics. Use this to verify that the heuristics are working correctly.

-m Output matchings if they exist
If a matching exists for any matrix that would normally be output according to the current flags, then setting –m will enable output of the matching as a matrix.

-u Output only unfiltered matrices (no effect if –f is not also set; overrides –a)
Set this flag to print only the matrices *not* filtered by the heuristics.

-a Output all matrices and comment out filtered ones (if –f is also set)
This flag overrides the default behavior of not outputting matrices that do not contain a matching.

-L Output the matrices as linearized binary strings
Use this flag to compress the output and save space. Matrices are printed as one long string instead of as a set of rows.

-d Don't output matrices (overrides –m, –a, –u, –L)
If this flag is set, then no matrices are output, regardless of other output settings.

-n Output matrix identifiers
Set this flag to output a unique numeric identifier for each matrix considered within a given dimension. (Useful for exhaustive or manual enumeration.)

-s Output statistics
Set this flag to output test statistics for each dimension of matrices considered.

-c{d} Set cutoff density for sparse rows at <d>%
This flag allows control of the definition of a “sparse row” as used in test 4. Set to any value in the range [0,100]. The default value is 50.

-r[n[,d[,s]]] Enumerate <n> random matrices of <d>% density using seed <s>
This flag allows random enumeration (essential for matrices larger than 8 x 8). The number and density of matrices are adjustable, as well as the initial seed value for the pseudorandom number generator. The default number is 100, the default density is randomized (a value of “0” sets this), and the default seed is 0. Note that because of the nature of the pseudorandom number generator, even a density value of “100” does not guarantee a full matrix, but a value of “100” should give more coverage than “90”, etc.

-e{1,2,...|1-2,...} Enumerate only the listed matrices (up to 8 x 8)

Matrices can be manually enumerated using this flag, but only matrices of dimension up to 8 x 8 (because of implementation limitations). Each number in range is converted to binary and interpreted as a long string consisting of the rows of a matrix. A matrix of the same numeric value may be enumerated several times, depending on the dimensions parameter.

-t{0|1|2|3|4|5|6}+ Specify which tests to perform and in what order

This flag allows the user to selectively run heuristics and fine-tune their running order. At least one test must be specified or the program will default to doing the complete suite. The tests are: 0 = sort the matrix; 1 = test each column for the presence of a “1”; 2 = compare against the row-sorted identity matrix; 3 = test for forced moves; 4 = test for small neighborhoods (using the cutoff density as described above); 5 = test for small duplicate rows; 6 = test each row for the presence of a “1” (not included by default; see section 4.7). Note that if this flag is set, the matrix will not be sorted by default unless test 0 is specified; this is to allow fine-tuning of the timing of sorting. The default test suite is “102345”.

6.3. Explanation of Output

The following is a typical example of output from adjmat:

```
#Dim:
{
7
}
#Flags:
#         Ver
#         Xcl
#         Ign
#         Fil
#         Prv
#         Dnt
#         Sta
#         Ctf       {50}
#         Rnd       {1000000,50,33333}
#         Tst       {1,6,0,2,3,4,5}

#========7x7

#Testing 1000000 7x7 matrices took 2.718 seconds.
#Rate of testing = 367918 matrices/second.
#**STAT**7x7
#Total matrices:     1000000
#Total passed:       146
#Total bad failures: 0
#Percent passed:     0.0146%
#Total filtered:     999854
#         Not sorted:                            999758
#         Rows without a 1:                      31
#         Columns without a 1:                   40
#         Not big enough:                        3
#         Not enough pairwise unique 1s:         22
#         Sparse rows don't cover enough columns: 0
#             (Sparse = 50% of row capacity)
#         Duplicate rows/cols with not enough 1s: 0
#Percent filtered:   99.9854%
#Total unfiltered:   0
#Percent unfiltered: 0%
#Unfiltered:Passed = 0:146 = 0%
#Total matchings:    146
#Greedy matchings:   88
#Greedy:Total =      88:146 = 60.274%
#********
```

The header information beginning with “#Dim” is present whenever the –v (verbose) flag is set; it shows the values of the command-line parameters. (Verbose mode also generates time statistics for each dimension showing how long the run of tests took in seconds and providing a testing rate in matrices per second.) In this case, adjmat was

run on one million randomly generated (Rnd) 7 x 7 matrices (Dim = {7}) exclusively (Xcl) in verbose mode (Ver), ignoring unsorted matrices (Ign). The program was instructed to check each filtered matrix for a matching (Prv), matrix output was disabled (Dnt), and statistics output was enabled (Sta). The cutoff value for heuristic (4) was set at 50% (Ctf = {50}), the random matrix generator was instructed to produced matrices of 50% density, starting with a seed value of 33333 (Rnd = {1000000,50,33333}), and the tests were manually specified to run in the order “no empty columns” > “no empty rows” > “sort” > “big enough” > “eliminate forced moves” > “no small neighborhoods” > “no small duplicates” (Tst = {1,6,0,2,3,4,5}).

The statistics listing at the bottom gives a great deal of information about how the heuristics performed on the test set. Line by line, here is the breakdown:

- “Total matrices”: Number of matrices tested; should match command-line value.
- “Total passed”: Number of matrices not rejected by any heuristic.
- “Total bad failures”: Number of matrices containing a matching that were rejected by the heuristics. (This should always be 0, or the program has a bug!)
- “Percent passed”: Ratio of “Total passed” to “Total matrices” as a percentage.
- “Total filtered”: Number of matrices caught by the filters; an individual breakdown of values for each of the heuristics is provided immediately below this line; the sum of the individual values should equal the total amount.
- “Percent filtered”: Ratio of “Total filtered” to “Total matrices” as a percentage.
- “Total unfiltered”: Number of matrices not containing a matching that were not caught by the heuristics; the lower, the better.
- “Percent unfiltered”: Ratio of “Total unfiltered” to “Total matrices” as a percentage; provided for completeness, but not a measure of performance.
- “Unfiltered:Passed”: Ratio of “Total unfiltered” to “Total passed” as both a ratio and a percentage; this gives an idea of how many filtering opportunities were missed and is the best measure of heuristic performance.
- “Total matchings”: Should be equal to the difference between “Total passed” and “Total unfiltered” (if not, there is a bug); this counter is only incremented after the program actually performs a matching, so it’s useful as a check that the full matching is behaving correctly.
- “Greedy matchings”: Number of matchings that were discovered by a greedy matching procedure without needing to run the full matching algorithm (see section 6.4); the higher, the better.
- “Greedy:Total”: Ratio of “Greedy matchings” to “Total matchings” as both a ratio and a percentage.

The next example shows the output after adjmat is run with the same parameters as above, except that unsorted matrices are no longer ignored (i.e., they are sorted by rows prior to the running of the test suite), matrix identifiers have been added (indicating where in the enumeration the matrix appeared), and verbose mode is disabled.

```
#---57514
#0000101
#0010010
#0100001
#0100100
#0100101
#1011011
#1111110
#(Not filtered)

#--233955
#0011010
#0101000
#1001000
#1010101
#1100000
#1100111
#1101000
#(Not filtered)

#--385184
#0000101
#0011010
#0111110
#1000001
#1000100
#1000101
#1100110
#(Not filtered)

#--892493
#0001010
#0010100
#0100100
#0110000
#0110100
#1000111
#1101001
#(Not filtered)

#**STAT**7x7
#Total matrices:     1000000
#Total passed:       613090
#Total bad failures: 0
#Percent passed:     61.309%
#Total filtered:     386910
#         Not sorted:                          0
#         Rows without a 1:                    139162
#         Columns without a 1:                 166533
#         Not big enough:                      7859
#         Not enough pairwise unique 1s:       73331
#         Sparse rows don't cover enough columns: 15
#            (Sparse = 50% of row capacity)
#         Duplicate rows/cols with not enough 1s: 10
#Percent filtered:   38.691%
#Total unfiltered:   4
#Percent unfiltered: 0.0004%
#Unfiltered:Passed = 4:613090 = 0.000652433%
#Total matchings:    613086
#Greedy matchings:   356703
#Greedy:Total =      356703:613086 = 58.1816%
#********
```

Statistics show that the performance of the heuristics on this group of matrices was better than 99.999%. Finally, an example to demonstrate the effect of –e and –m flags:

```
#Dim:
{
7
}
#Flags:
#         Ver
#         Xcl
#         Mch
#         All
#         Enm
{
481927363215310
}

#========7x7

1101101
1001001
1110111
0101010
0101010
1111111
1001110
#%MATCHING%
# 1000000
# 0001000
# 0000001
# 0000010
# 0100000
# 0010000
```

```
# 0000100
#%%%%%%%%%

#Testing 1 7x7 matrices took 0 seconds.
#Rate of testing = 0 matrices/second.
```

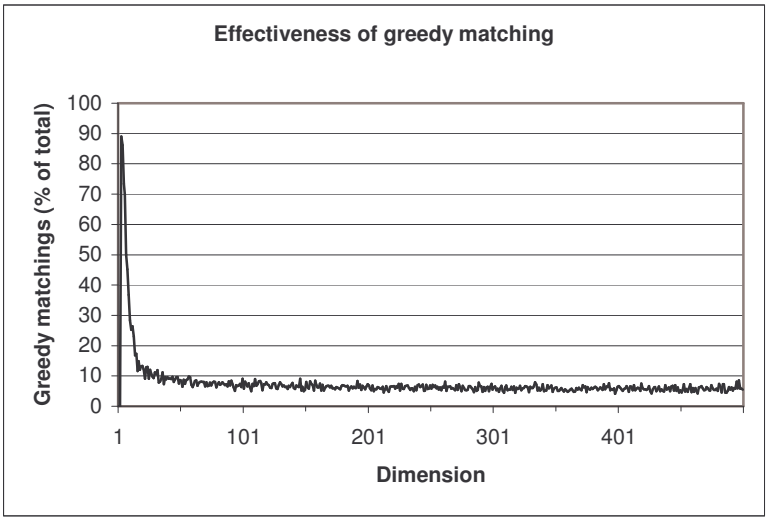
Here, we instructed the program to output a matrix of our specification (Enm; note that 481927363215310_{10} = $1101101100100111101110101010010101011111111001110_2$) and find a matching for it if possible (Mch), printing it out even if no matching existed (All). The –x (Xcl) flag was set automatically by the presence of the –e flag. The time statistics are useless in this case because the process terminated so rapidly.

6.4. Hopcroft-Karp Implementation

A matching procedure is used to test the performance of our heuristics. If a matrix passes all of the heuristics, then we attempt to compute a matching, and if none is found, then we know that the matrix has slipped through the cracks. We also use the matching algorithm to verify that a perfect matching does *not* exist in a matrix that was rejected by a heuristic (this is enabled by a command-line option; see section 6.2).

The matchings are done using the Hopcroft-Karp algorithm [6], adapted for a Boolean matrix representation of bipartite graphs. This algorithm was chosen because it has one of the lowest worst-case bounds for bipartite matching algorithms (see section 3), and because it's easy to understand and implement, requiring only basic data structures.

Two optimizations to the algorithm were undertaken. First, we implemented the breadth-first search for augmenting paths using a one-step lookahead—conceptually analogous to loop unrolling. Second, before the main procedure we added a "greedy matching" pass—attempting to pair up as many unmatched vertices as possible in order of discovery. The greedy matching makes a single pass through the rows of the matrix, matching each row with the first available column. If no columns are available, then the row is skipped. Despite the crudeness of the procedure, it finds a perfect matching about 80% of the time in small cases, but rapidly deteriorates to a near-constant level of about 5% for larger cases.



7. Concluding Words

Our goal was to find a reliable, low-cost, *prima facie* means of determining whether a given bipartite graph has a perfect matching. It was hoped that the capacity to make such a determination would enable faster disposition of perfect matching problems by eliminating from processing any classes of graphs that cannot have perfect matchings—there is no value in running an expensive matching procedure when you don't have to. The heuristic filter model we developed toward this end has proven to be both exceptionally reliable and reasonably low in cost, allowing us to successfully identify and filter out more than 99.9% of unusable candidate graphs in time proportional to the cost of sorting the vertices of the graph by edge counts (an essentially *prima facie* measure). Moreover, the model extends the legacy of Hall's Theorem in offering promising lines for further research into the conditions necessary for perfect matchings to exist in bipartite graphs. We believe that the framework we have developed is immediately usable in such research and extendable to many classes of problems where rapid computation of perfect matchings is desired.

References

[1] Alt H, Blum N, Mehlhorn K, Paul M. Computing a maximum cardinality matching in a bipartite graph in time $O\left(n^{1.5}\sqrt{m/\log n}\right)$. *Information Processing Letters* 1991; **37**: 237-240.

[2] Anderson T, Owicki S, Saxe J, Thacker C. High-speed switch scheduling for local-area networks. *ACM Transactions on Computer Systems* (TOCS), 1993

[3] Cormen T, Leiserson C, Rivest R. *Introduction to Algorithms*. MIT Press: Cambridge, 1990; 600-604.

[4] Dekel E, Sahni S. A Parallel Matching Algorithm for Convex Bipartite Graphs and Applications to Scheduling. - 1981

[5] Feder T, Motwani R. Clique Partitions, Graph Compression and Speeding-up Algorithms. *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*. ACM Press, New York, 1991; 123-133.

[6] Ford L, Fulkerson D. *Flows in Networks.* Princeton University Press: Princeton, 1962.

[7] Hall P. On Representatives in Subsets. *Journal of the London Mathematical Society* 1935; **10**: 26-30.

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

[9] Kennedy R. *Solving Unweighted and Weighted Bipartite Matching Problems in Theory and Practice*. Ph.D. Thesis: Stanford University, Palo Alto, 1995.

[10] Kim WY, Kak AC. 3-D object recognition using bipartite matching embedded in discrete relaxation. *IEEE Transactions on Pattern Analysis and Machine*, 1991

[11] Knuth D. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms, 2nd Ed*. Addison-Wesley Professional: Reading, 1981; 171-173.

[12] Shokoufandeh A, Dickinson S. Applications of Bipartite Matching to Problems in Object Recognition*. Proceedings, ICCV Workshop on Graph Algorithms and*

Computer, 1999
[13] Taylor W. Protein Structure Comparison Using Bipartite Graph Matching and Its Application to Protein Structure - Molecular & Cellular Proteomics, 2002 - ASBMB
[14] Verma R, Ramakrishnan I. Tight Complexity Bounds for Term Matching Problems. *Information and Computation Journal*, 1992; **101** (1): 33-69.

[EOF]