An Empirical Study of Algebraic Algorithms in Perfect Matching

Austin V. Pham[†]

Project advisor: Kevin Cheung[‡]

Abstract. Algebraic algorithms in perfect matching can be conceptually simple and theoretically capable of outperforming combinatorial approaches in dense graphs, yet combinatorial algorithms remain the standard in practice. To investigate the practical viability of algebraic methods, we implement the algorithms of Lovász, Rabin and Vazirani, Harvey, and evaluate their correctness and run-times across a wide variety of graph instances. We benchmark these against Kolmogorov's Blossom V, the combinatorial perfect matching software widely considered to be the fastest of its kind. Our results show that our algebraic implementations are 5-250× slower than Blossom V in perfect matching detection, and 328-21000× slower for solving perfect matchings. In all cases, the dominant computational bottleneck was the $O(n^3)$ matrix multiplication routines performed by standard linear algebra computer libraries. Additionally, we find finite-field arithmetic to be essential for our purposes, since fixed-precision numerical arithmetic could not guarantee accurate computations in practice.

1. Introduction. Graph matching is a classic problem with a rich tradition in combinatorial optimization and computer science. Simply, given a network of nodes and connections (a.k.a. a graph of vertices and edges), it asks for the largest subset of connections that do not share a common node. Graph matching is central to many challenges in logistics, where an optimal assignment of limited resources is often critical. Algorithms for graph matching are relevant to applications in computer vision [13], job scheduling [20] [32], urban transportation [18], and intersatellite communications [31]. Moreover, graph matching often emerges as a major subroutine in other optimization problems, such as the Chinese Postman Problem [6] and the maximum cut problem in planar graphs [12]. The history of graph matching is extensive, and the tradition has seen many new developments in algorithmic solutions since the publication of Edmonds' famous *blossom* algorithm in 1965 [5]. Table 1 gives a brief history.

The graph matching problem has several variations—namely, bipartite, non-bipartite, unweighted maximum cardinality, maximum-weight, and minimum-weight—but our investigations will primarily concern perfect matching in graphs that are not assumed to be bipartite. Algorithms for this variant attempt to produce matchings that cover every vertex in a graph; however, it is important to note that routines of this kind can easily be extended for maximum cardinality matching via existing techniques [27] [24]. Since Edmonds' polynomial-time breakthrough for general graphs, mathematicians and computer scientists have traditionally improved upon his worst-case time complexity via one of two means: combinatorial techniques that make use of structures like augmenting paths or Edmonds' notion of blossom [5], or algebraic approaches that extend from the discoveries of Tutte [30]. Instead of examining graphs directly, algebraic algorithms interpret the discrete structure as a matrix and reduce the matching problem to computing the determinant, matrix rank, or inverse matrix. Despite the fact that algebraic algorithms are capable of offering competitive theoretical complexities in denser graphs, combinatorial solutions are overwhelmingly favored as the standard choice

[†]School of Mathematics and Statistics, Carleton University, Ottawa, Canada (austinpham@cmail.carleton.ca).
[‡]School of Mathematics and Statistics, Carleton University, Ottawa, Canada (kevin.cheung@carleton.ca).

Table 1
Summary of Cardinality Matching Algorithms.

Year	Authors	Time Complexity
1965	Edmonds [5]	$O(mn^2)$
1975	Even & Kariv [7]	$O(min(n^{2.5}, \sqrt{n}m \log n))$
1980	Micali & Vazirani [22]	$O(m\sqrt{n})$
1989	Rabin & Vazirani [27]	$O(n^{\omega+1})$ *
1991	Gabow & Tarjan [9]	$O(m\sqrt{n})$
2004	Goldberg & Karzanov [11]	$O(m\sqrt{n} \log (n^2/m)/ \log n)$
2004	Mucha & Sankowski [25]	$O(n^{\omega})$ *
2006	Harvey [14]	$O(n^{\omega})$ *

Note: n and m denote vertices and edges, respectively. Entries marked by $*$ are algebraic and involve randomization. Additionally, $\omega < 2.371339$ denotes the exponent belonging to the asymptotic complexity of matrix multiplication [2].

in practical applications. This antagonism is the primary subject of our research.

In this paper, we describe our implementations of algorithms proposed by Lovász, Rabin and Vazirani, and Harvey, which we use to investigate the practical viability of algebraic solutions. We compare these solutions to Kolmogorov's Blossom V software—the latest iteration of Edmonds' blossom algorithm—which is widely considered to be the experimentally fastest implementation of its kind. Our paper is organized as follows. In section 2, we provide an overview of the major algebraic developments used in perfect matching. In section 3, we describe our programmed implementations and their required linear algebra dependencies. In section 4, we present the computational results of our investigations. Finally, in section 5, we provide conclusions and discuss future research.

2. Background.

Let $G = (V, E)$ be an undirected simple graph and $n = |V|$, where V is a finite set of vertices, and $E \subseteq \{(i, j) : i, j \in V \wedge i \neq j\}$ is a set of edges. A *matching* of G is a subset $E' \in E$ where each vertex from V appears at most once in any edge of the matching, i.e., no edges in E' share a common vertex. A matching is considered to be *perfect* if every vertex in V appears exactly once in E', or, equally, if $|E'| = \frac{n}{2}$.

Algebraic algorithms for cardinality matching fundamentally concern themselves with G through G's associated *Tutte matrix*, an alternative representation that uniquely lends itself to non-combinatorial techniques which we will detail in subsection 2.1, subsection 2.2, and subsection 2.3. First discovered by Tutte in 1947 [30], a Tutte matrix T of G is an $n \times n$ symbolic matrix whose entries are defined by:

$$T_{i,j} = \begin{cases} x_{ij} & \text{if } (i, j) \in E \text{ and } i < j \\ -x_{ij} & \text{if } (i, j) \in E \text{ and } i > j \\ 0 & \text{Otherwise} \end{cases}$$

By definition, T is skew-symmetric. For each edge $(i, j) \in E$, T_{ij} is an indeterminate while its corresponding entry across the diagonal is the negative of the same variable. Given T of G, Tutte revealed that G has a perfect matching if and only if T is non-singular [30]. In other words, G has a perfect matching if and only if T's determinant is not identically zero. Unfortunately, this property by itself—as a means of verifying the existence of a perfect matching in a graph—is extremely limited by the symbolic nature of its determinant computation. Because we are constrained to indeterminate variables instead of constants for the entries of the matrix, evaluating the determinant may yield a polynomial with an exponential number of terms. However, Lovász showed that computing T's determinant could be performed in polynomial time by using randomization [19].

2.1. Detecting a Perfect Matching (Lovász' Algorithm). Originally proven by Zippel and Schwartz, Lemma 2.1 is fundamental to Lovász' solution.

Lemma 2.1 (Schwartz-Zippel [28][33]). *Let $p(x_1, ...x_n)$ be a non-zero polynomial of degree d over a field $\mathbb{F}$. If we randomly choose elements $s_1, ..., s_n$ from $S \subseteq \mathbb{F}$, then*

$$Pr[p(s_1, ..., s_n) = 0] \leq \frac{d}{|S|}$$

Since T's determinant is a non-zero polynomial when G has a perfect matching, we are able to give constants to T's indeterminate entries and determine whether T has full rank with high probability. In this case, perfect matching detection can instead take the form of Algorithm 2.1, first proposed by Lovász [19].

Algorithm 2.1 Lovász' Algorithm

Create Tutte matrix T
Substitute T's indeterminate entries with random integers drawn independently and uniformly from $\{1, ..., n^2\}$
if $det(T) \neq 0$ **then**
    return True (perfect matching exists)
else
    return False (No perfect matching exists, with high probability)
end if

This algorithm guarantees $O(n^\omega)$ time complexity—the bound for multiplying two $n \times n$ matrices, or computing the determinant or inverse of a square matrix [29]. Despite being non-deterministic, failures can only present themselves as false indications that a graph does not have a perfect matching; in this sense, only false negatives can occur (a graph with a perfect matching is said to not have a perfect matching), and the probability of having an incorrect verdict can be vastly reduced by performing multiple independent trials.

Further improving on Lovász' algorithm, Rabin and Vazirani proposed constructing the Tutte matrix over a finite field $\mathbb{F}_p$ instead of the rationals $\mathbb{Q}$ [27]. In this case, we find a prime number $p \geq n^2$, and then substitute T's indeterminate entries with random non-zero elements from $\mathbb{F}_p$. Although this approach has the same asymptotic time as Lovász', the advantage of this approach is that a finite field keeps the intermediate numbers produced in the determinant

computation bounded by p, thus guaranteeing more efficient arithmetic in practice. Had we used integers, our intermediate values may have grown expensively large.

2.2. Constructing a Perfect Matching. In addition to refining Lovász' Tutte matrix construction, Rabin and Vazirani also showed that constructing a perfect matching could be performed in $O(n^{\omega+1})$ time [27]. Compared to a naive self-reducibility algorithm (i.e., one that plucks every possible edge and recomputes the determinant until no removable edges remain) that requires $O(n^{\omega+2})$ time, Rabin and Vazirani showed that we can take advantage of the properties belonging to T's inverse matrix.

Lemma 2.2 (Rabin-Vazirani). *Let $G = (V, E)$ be an undirected simple graph that has a perfect matching. $G[V\backslash\{u, v\}]$ has a perfect matching if and only if $(T^{-1})_{u,v} \neq 0$.*

Since (u, v) must be in G's perfect matching in order for the subgraph $G[V\backslash\{u, v\}]$ to also have a perfect matching, it stands to reason that constructing a perfect matching can instead take the form of Algorithm 2.2.

Algorithm 2.2 Rabin-Vazirani's Algorithm

```
Construct Tutte matrix T
Substitute T's indeterminate entries with random non-zero elements from 𝔽_p
if det(T) = 0 then
    return "No perfect matching exists"
end if
m = ∅
while V(G) ≠ ∅ do
    Compute T^{-1}
    Find (i, j) ∈ E where T_ij ≠ 0
    Add edge (i, j) to m
    Remove vertices i and j from V(G)
end while
return m
```

Rabin and Vazirani's algorithm computes T^{-1} only $n/2$ times, while the naive approach could have computed the determinant as many as $O(n^2)$ times. This is certainly an improvement, but note how Rabin and Vazirani's algorithm is forced to recompute T^{-1} from scratch before each new edge of the matching is found; would the most optimal algorithm not benefit from the relations shared between each new iteration of T^{-1}? Mucha and Sankowski were the first to improve perfect matching to $O(n^{\omega})$ time complexity [25]—the current best-known bound for perfect matching in dense graphs—but Harvey proposed an $O(n^{\omega})$ solution that is strictly algebraic and conceptually simpler [14].

Algorithm 2.3 Harvey's Algorithm

```
function FindPerfectMatching(G = (V, E))
    Construct Tutte matrix T of G
    Substitute T's indeterminate entries with random non-zero elements from F_p
    if det(T) = 0 then
        return "No perfect matching exists"
    end if
    Compute N := T^{-1}
    DeleteEdgesWithin(V)
    return remaining edges in T
end function

function DeleteEdgesWithin(S)
    if |S| > 1 then
        Divide S equally into S_1 ∪ S_2
        for i ∈ 1, 2 do
            DeleteEdgesWithin(S_i)
            Update N
        end for
        DeleteEdgesCrossing(S_1 ∪ S_2)
    end if
end function

function DeleteEdgesCrossing(R, S)
    if |R| > 1 then
        Divide each set equally so R = R_1 ∪ R_2 and S = S_1 ∪ S_2
        for i ∈ {1, 2} and j ∈ {1, 2} do
            DeleteEdgesCrossing(R_i, S_i)
            Update N
        end for
    else
        Let r ∈ R and s ∈ S
        if T_rs ≠ 0 and N_rs ≠ −1/T_rs then
            Set T_rs and T_sr = 0
            Update N
        end if
    end if
end function
```

Harvey's technique for examining $E(G)$ can also be made to use Lemma 2.2 like Rabin and Vazirani's algorithm, but the virtue of Harvey's approach is the same in any case; for the sake of illustrating its advantage, consider Harvey's algorithm as instantiated in Algorithm 2.3, where T^{-1} is denoted by N. After verifying the existence of a perfect matching in G, we

delete every edge (r, s) where $N_{rs} \neq -1/T_{rs}$—this is an additional characteristic of the Tutte matrix—until the remaining edges in T represent G's perfect matching. Harvey's recursive divide-and-conquer approach for examining G's edges allows him to take advantage of lazy computations for N, since in the case that we find a removable edge, our required matrix update is only consequential to our recursive parent. Therefore, when we construct our perfect matching, we are entitled to perform only partial updates to N instead of the computations from scratch that characterize Rabin and Vazirani's algorithm.

2.3. Maximum Cardinality Matching. All perfect matching algorithms can be extended to maximum matching without any breakdown in asymptotic complexity. Several existing techniques lend themselves to this end, either via a modification of the original graph [27], or via a deduction of the largest subgraph G' that has a perfect matching [24].

Another notable property between G's maximum matching and G's symbolic Tutte matrix was generalized by Lovász:

Theorem 2.3 (Lovász [19]). *Let T be graph G's associated Tutte matrix, and M be G's maximum matching. Then, $rank(T) = 2 \cdot |M|$.*

Rabin and Vazirani proposed an efficient algorithm for computing the rank of T probabilistically [27]. Conversely, Geelen also demonstrated a method for evaluating T using integers—rather than indeterminates—that enables a deterministic rank computation [10].

3. Description of our Implementations. For the purposes of our investigation in algebraic matching algorithms, we chose Julia to be the language of our implementations and experiments. Relatively young compared to other developments in programming like Python or C++, Julia is a high-level language specially designed for high-performance scientific computing. Although younger languages often lack the package and library richness of more established languages, we found that Julia offered a thorough variety of functionalities more than sufficient for our investigations; in particular, we found value in Julia's LinearAlgebra library, and the Nemo computer algebra package [8].

In total, we created four distinct implementations over the course of our investigations [26]. These computer programs operate in a purely sequential manner and do not use parallelism.

Table 2
Overview of Implementations.

Name	Functionality	Type of Arithmetic	Padding Required?
1. LovaszDetection.jl	detection	numerical	no
2. RabinVaziraniDetection.jl	detection	finite-field (via Nemo)	no
3. HarveyMatching.jl	detection + construction	numerical	yes
4. HarveyNemoMatching.jl	detection + construction	finite-field (via Nemo)	yes

Our numerically implemented algorithms performed their matrix operations using the LAPACK and BLAS computer libraries through Julia's standard LinearAlgebra library, while our finite-field algorithms primarily used the FLINT library through Nemo. These two arrangements appeared to be unavoidable for our purposes, which meant that our implementations could not actually express $O(n^\omega)$ complexity, since virtually all popular linear algebra libraries use highly-optimized standard $O(n^3)$ matrix multiplication routines instead of the leading the-

oretical solutions (e.g., Coppersmith-Winograd [3], Le Gall [17], or, more recently, Alman *et al.* [2]). Despite having theoretical time complexities superior to $O(n^3)$, these fast matrix multiplication algorithms have enormous leading constant factors that make them impractical and unlikely to provide any advantage in our study. Therefore, our four programs could only be implemented with $O(n^3)$ time complexity.

Programs 1 and 2 are both implementations of Lovász' algorithm for perfect matching detection; however, the latter constructs the Tutte matrix T over a finite field, as per Rabin and Vazirani's contention against the use of integers[1]. These two programs served our first investigation regarding the practical implications of using finite fields versus integers for detection.

Secondarily, programs 3 and 4 were used to resolve our second question concerning the viability of Harvey's algorithm for matching compared to popular combinatorial alternatives. Both programs 3 and 4 are implementations of Harvey's algorithm for perfect matching construction, however, the former uses integers while the latter uses finite fields. Again, these two versions were created for the purpose of gauging their theoretical differences in practice. In the potential case that both versions proved to be equally correct and fast, HarveyMatching's program would be advantaged in the sense that it requires fewer dependencies to use.

Despite a few tweaks, our implementations are overall very faithful to their original conceptual instances. One minor change made belonged to HarveyNemoMatching and HarveyMatching's construction of the Tutte matrix. Because Harvey's recursive approach to examining every edge requires that the graph's number of vertices is a power of 2, Harvey suggests satisfying this constraint by padding the graph with a disconnected clique; however, we found it more efficient to add a disconnected cycle graph instead, since fewer edges are created and examined.

4. Computational Results. In this section, we first consider the performances of LovaszDetection and RabinVaziraniDetection in subsection 4.1, and subsequently our implementations of Harvey's algorithm compared to the code of Blossom V by Kolmogorov in subsection 4.2. Blossom V is considered to be the fastest practical implementation of Edmond's Blossom algorithm for minimum-cost perfect matching [16]. Designed in C++, it operates purely sequentially and does not take advantage of any parallelism. We will refer to it as B5. Despite having some of the fastest empirical times for most graph applications, it is marked by two notable limitations. First, B5 is incompatible with graphs that do not have a perfect matching. In the case that it is given a graph without a perfect matching, its program is characterized by undefined behavior and segfaults. Second, B5's theoretical time complexity is estimated by its author to still be $O(n^3m)$, despite having been shown to be capable of outperforming implementations with better time complexities—notably Melhorn and Schafer's $O(nm \log n)$ code [21].

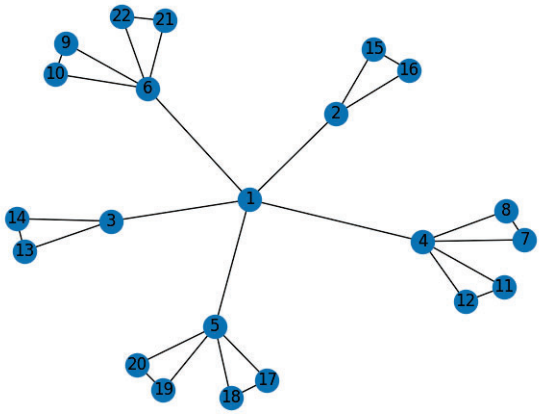
All tests were performed on an Apple M2 pro processor with 16 GB of memory, running macOS 15.5. Our experiments were conducted in version 1.11.5 of Julia and measured via the language's standard *time*() functionality. For every program tested, we began recording

[1] "[Lovász' algorithm], although polynomial time, is inefficient: since [T's determinant] is being computed over the rationals, the intermediate numbers in the computation may be $O(n \log^2 n)$ bits long. For the purpose of efficiency, we will substitute for the indeterminates in [T] randomly from a finite field $\mathbb{Z}_p$;" [27].

times once the adjacency matrix of a sample graph was given. Although B5 was originally implemented in C++, its functionality can be accessed in Julia via a wrapper, which we use here [23]. Because B5 is only compatible with graphs that have perfect matchings, our Julia code for B5 called the original C++ software on edges of the given graph as having 0-cost, but also supplied the software with additional non-existent 1-cost edges to guarantee that the B5's graph constraints were satisfied. Since B5's solutions are minimum-cost, we were then able to verify from the weights of the solution's edges whether the original tested graph had or did not have a perfect matching.

4.1. Outcomes in Detection. One may be tempted to use fixed-precision data types in hopes of faster times for arithmetic operations, but in practice, we found that their limited accuracy frequently led to incorrect results in detection. We observed that our LovaszDetection implementation easily gave incorrect solutions—especially on graphs without perfect matchings—whether we used Int or floating-point values for our Tutte matrix's substituted indiscriminate entries, or even if we changed our set of random values from $\{1, ...n^3\}$ to floating-point values between 0 and 1. In any of these circumstances, the accuracy of the program's determinant computation was unreliable, and the use of a numerical tolerance offered no advantage. One recorded instance of our tests involved a 22 node graph without a perfect matching, that yielded a computed determinant as large as 6^{16} instead of 0. We depict this graph instance in Figure 1.

Figure 1. *Depiction of our 22 node GenerateGraphs.jl's "NonPerfectB" instance.*



Julia's BigInt type may help the implementation's precision up to a point, but our tests found that its use resulted in performance speeds between 12 and 2611 times poorer than those of our finite-field implementation. RabinVaziraniDetection consistently produced the correct solutions for every graph we tested. We refer to the program as RV and report some of its times (in seconds) compared to our adapted use of B5 in Table 3, Table 4, Table 5, Table 6, and Table 7. Every graph instance recorded in each table has a perfect matching and each recorded time is the average of five trials. Table 4 and Table 5 use instances generated in our repository's GenerateGraphs.jl file [26], while Table 6 and Table 7 use graph families from

the first DIMACS implementation challenge, where k is the primary parameter for generating their respective instances [15]. To the right of RV's column in each table, in both seconds and as a percentage, we record the average portion of RV's running time that is exclusively spent computing the determinant. And, beside B5's column, we also note the ratio between RV and B5's total running time for each graph instance.

Table 3

Complete graphs.

n	RV	RV's Det() Computation	B5	RV/B5 Ratio
1000	0.21279459	0.1484589577 - 69.76%	0.01857919693	11.45337933
2000	0.9167212009	0.7066986561 - 77.08%	0.08541717529	10.73228186
3000	2.522559166	1.982972622 - 78.60%	0.2271552086	11.10500253
4000	4.152602005	3.417068322 - 82.28%	0.3815698147	10.88294159
5000	7.21569581	5.919749657 - 82.03%	0.9110708237	7.92001634
6000	11.27074122	9.460760276 - 83.94%	1.400329971	8.048632433
7000	18.4014576	14.38717508 - 78.18%	2.501266956	7.356854713
8000	22.521486	17.46560733 - 77.55%	3.726492137	6.043615596
9000	30.44923139	24.34894609 - 79.96%	5.185098529	5.872449909
10000	39.48073997	33.00698868 - 83.60%	7.312358856	5.399179766

Table 4

GenerateGraphs.jl Instance: Dense Graph (> 70% density).

n	RV	RV's Det() Computation	B5	RV/B5 Ratio
1000	0.1926362038	0.165282011 - 85.80%	0.01103520393	17.45651507
2000	0.9112217903	0.7824376424 - 85.86%	0.05518083572	16.51337422
3000	2.478900385	2.18177104 - 88.013%	0.1375358105	18.02367235
4000	4.167738628	3.628989299 - 87.07%	0.3411907673	12.21527376
5000	7.32070117	6.530325095 - 89.20%	0.6306585789	11.60802598
6000	11.46490483	10.48391406 - 91.44%	1.003965807	11.41961684
7000	18.48208799	15.91168467 - 86.09%	1.533049494	12.05576732
8000	22.51318779	18.95633602 - 84.20%	2.267210633	9.929905703
9000	30.4638792	25.85127568 - 84.85%	3.166404635	9.620968486
10000	39.56646881	33.45295731 - 84.54%	4.184051633	9.456496306

Table 5

GenerateGraphs.jl Instance: Sparse Graph (> 10% density).

n	RV	RV's Det() Computation	B5	RV/B5 Ratio
1000	0.1964646339	0.1765089035 - 89.84%	0.002152633667	91.26710083
2000	0.9107712269	0.8321564198 - 91.36%	0.009211778641	98.87029014
3000	2.480665398	2.265979211 - 91.34%	0.02276082039	108.9884
4000	4.119579363	3.873430967 - 94.02%	0.04330778122	95.12330687
5000	7.280148983	6.314190229 - 86.73%	0.07362442017	98.88225899
6000	11.48624959	10.0835313 - 87.78%	0.1229112148	93.45159924
7000	18.44555163	15.3165493 - 83.03%	0.2042821348	90.29449224
8000	22.51222978	18.10574142 - 80.42%	0.2744949758	82.01326713
9000	30.4144084	25.03954403 - 82.32%	0.3709200919	81.99719849
10000	39.5758698	33.02417231 - 83.44%	0.5049693882	78.37280976

Table 6
DIMACS Instance: t.f generator (K one-connected triangles).

k	n	RV	RV's Det() Computation	B5	RV/B5 Ratio
1000	3000	2.670252943	2.210223993 - 82.77%	0.01066536903	250.3666713
2000	6000	11.06074781	9.956852436 - 90.01%	0.04506459236	245.4420918
3000	9000	29.00880103	23.9282163 - 82.48%	0.1400350571	207.1538487
4000	12000	56.99316607	51.2194736 - 89.86%	0.2969954491	191.8991225
5000	15000	94.62952857	79.72319468 - 84.24%	0.5061861038	186.9461209

Table 7
DIMACS Instance: handcard.f generator ($\approx$ 44% density).

k	n	RV	RV's Det() Computation	B5	RV/B5 Ratio
100	600	0.07344355583	0.06496620178 - 88.45%	0.002544164658	28.86745385
200	1200	0.3011603355	0.2884907722 - 95.79%	0.0150267601	20.04160135
400	2400	1.412657404	1.309347312 - 92.68%	0.08415598869	16.78617798
800	4800	6.210098219	5.951874653 - 95.84%	0.5158162117	12.03936223

B5 performed significantly better than RabinVaziraniDetection across all tested graph types. Our algebraic implementation was least disadvantaged in denser graphs, where in the case of the complete graph K_{1000}, the ratio between RV and B5's execution times was only 11.45:1, compared to the sparser t.f 3000 node instance which yielded a ratio close to 250:1. These results reveal that the determinant computation represents the primary computational bottleneck in RV's execution, consistently representing the largest portion of its total runtime. However, as we will soon illustrate, the steady decline of the logarithm of the RV/B5 ratio seems to suggest that the performance gap between RV and B5 may actually continue to narrow. Figure 2, Figure 3, and Figure 4 plot the RV/B5 execution time ratios from Table 3, Table 4, and Table 5, with a fitted linear regression line demonstrating a negative correlation between graph size and performance gap.

Figure 2. *RV/B5 Ratio in Complete Graph.*

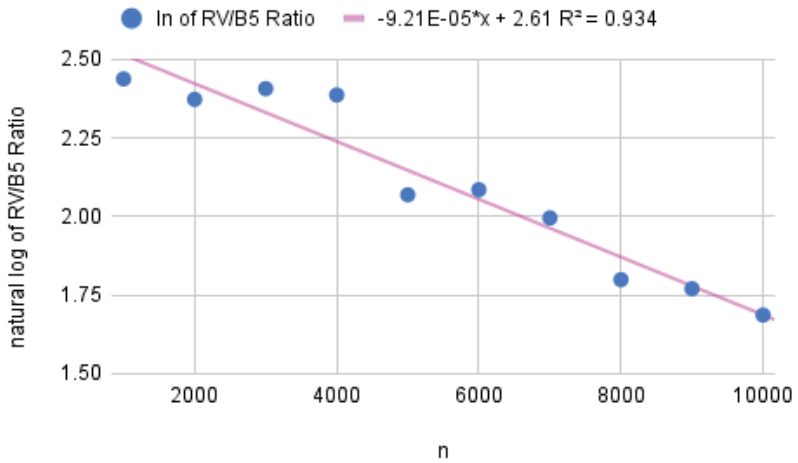


Figure 3. *RV/B5 Ratio in GenerateGraphs.jl Dense Instance.*

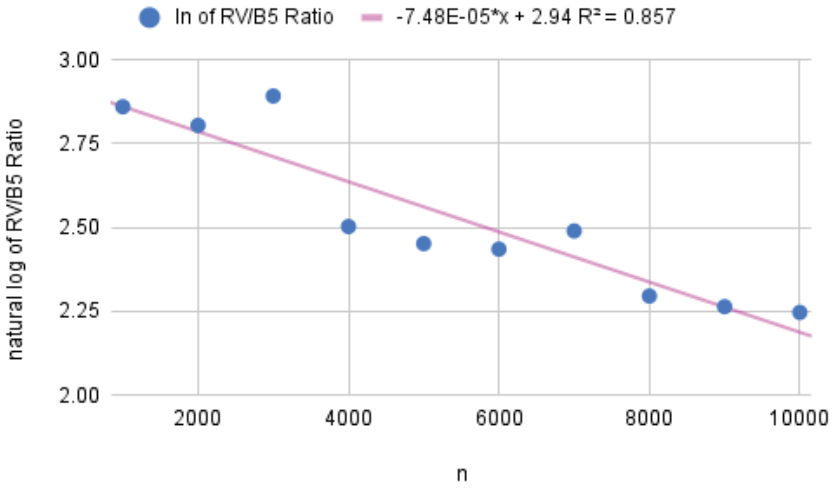
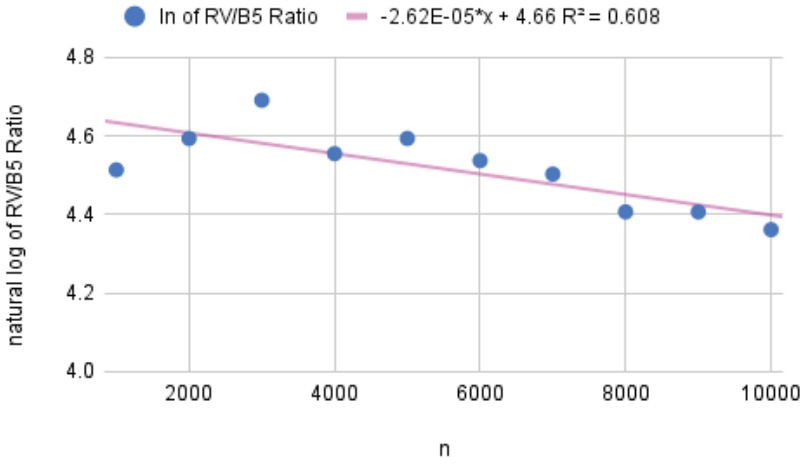


Figure 4. *RV/B5 Ratio in GenerateGraphs.jl Sparse Instance.*



From the rate of decrease observed in the logarithm of the RV/B5 ratios, it is thus conceivable that, if we are not limited by hardware constraints, RV could eventually outperform B5 in perfect matching detection.

4.2. Outcomes in Perfect Matching. Numerical arithmetic using fixed-precision data types again proved to be problematic in our practical implementations, where, in almost all cases, our HarveyMatching program either gave too few edges, too many edges, or edges nonexistent in $E(G)$. HarveyNemoMatching and B5 on the other hand, consistently gave the correct solutions to our variety of test graphs. We report some of their times in Table 8, Table 9, Table 10, Table 11, where we denote HarveyNemoMatching's runtime by HN. To the

right of HN's column, in seconds and as a percentage, we record the portion of HN's runtime that is spent constructing the perfect matching after it has verified that one exists. Similar to the previous section, our final column describes the ratio between our implementation HN and Kolmogorov's B5. All instances of these families have perfect matchings, and each recorded time is the average of five trials. According to the DIMACS online directory for matching generators, the hardcard.tf instances used in Table 8 were shown by Gabow to be hard for Edmond's blossom matching algorithm.

Table 8

DIMACS Instance: hardcard.f generator (≈ 44% density).

k	n	HN	HN's Construction Process	B5	HN/B5 Ratio
100	600	9.451244545	9.218592278 - 97.53%	0.01472458839	641.8681658
200	1200	45.50238714	44.41749153 - 97.61%	0.07987604141	569.6625213
400	2400	211.8013954	206.8307144 - 97.65%	0.6445405483	328.6083334

Table 9

Complete graphs.

n	HN	HN's Construction Process	B5	HN/B5 Ratio
500	4.847	4.669023312 - 96.32%	0.002376794815	2039.300982
1000	23.13475561	22.38597838 - 96.76%	0.01394119263	1659.453121
1500	81.06797	78.92195134 - 97.35%	0.0333302021	2432.26758
2000	126.8804	123.6493334 - 97.45%	0.05973377228	2124.098231
2500	311.7	304.1345627 - 97.57%	0.09420385361	3308.781839
3000	421.1216	411.7255823 - 97.76%	0.1486446857	2833.075383

Table 10

GenerateGraphs.jl Instance: Dense Graph (> 70% density).

n	HN	HN's Construction Process	B5	HN/B5 Ratio
500	4.58455	4.446337796 - 96.98%	0.002307987213	1986.384489
1000	22.13639426	21.52814631 - 97.25%	0.01032385826	2144.197809
1500	78.8999	77.07513766 - 97.68%	0.02598700523	3036.128992
2000	121.769	119.213546 - 97.90%	0.05186161995	2347.959823
2500	336.3496	329.7717594 - 98.04%	0.0817510128	4114.317223
3000	428.0066	420.1080945 - 98.15%	0.1250741005	3422.024211

Table 11

GenerateGraphs.jl Instance: Sparse Graph (> 10% density).

n	HN	HN's Construction Process	B5	HN/B5 Ratio
500	3.60959	3.570043669 - 98.90%	0.0005099773407	7077.941924
1000	17.90090494	17.64224399 - 98.55%	0.003607559204	4962.054378
1500	67.749	66.76478331 - 98.54%	0.004310750961	15716.28716
2000	99.6413	98.53544875 - 98.89%	0.008150196075	12225.63225
2500	284.044	279.6539101 - 98.45%	0.01364946365	20809.90193
3000	359.388	354.7722533 - 98.71%	0.02064976692	17403.97368

From our sample of execution times, one can observe that B5 is significantly more competitive than HarveyNemoMatching. HN's best performance relative to B5 occurred against the 2400 node hardcard.f instance, where the ratio between their respective runtimes was about 328:1. As shown in each table, HarveyNemoMatching's inferior performance appears to be overwhelmingly attributable to the cost of the algorithm's recursive construction process, which consistently represents over 95% of the entire execution time. This construction phase, as denoted by DeleteEdgesWithin() and DeleteEdgesCrossing() in Algorithm 2.3, is characterized by repeated matrix multiplication operations, which leads us to our conclusion that these $O(n^3)$ operations performed through the FLINT libary constitute HN's primary computational bottleneck.

5. Conclusions. In this paper, we described our findings regarding the implementation of algebraic perfect matching solutions, particularly Lovász' algorithm for perfect matching detection and Harvey's algorithm for perfect matching construction. In practice, numerical arithmetic using fixed-precision primitive data types proved to be insufficiently accurate for our purposes, thus supporting Rabin and Vazirani's proposal of the use of finite fields. Although Kolmogorov's popular Blossom V algorithm outperformed our implementations in both detection and construction, our selected algorithms are simple to implement and do not require the use of sophisticated combinatorial structures—instead, they use the graph's associated Tutte matrix to solve perfect matching via purely algebraic means.

Despite having theoretical $O(n^\omega)$ time complexities, Harvey and Lovász' superiority over modern combinatorial algorithms in sufficiently dense graphs could not be realized and observed in our study, especially as current computer libraries for mathematical routines do not necessarily employ matrix multiplication implementations better than $O(n^3)$ complexity. We posit that algebraic approaches in matching may not be advantageous over their combinatorial counterparts until implementations of leading theoretical matrix multiplication algorithms deemed effective in practice are available. Until then, in addition to combinatorial approaches, there may be more value in nontraditional solutions that use belief propagation [1] or approximation [4].

Acknowledgments. We thank Carleton University's Office of the Dean of Science for supporting our research through the Dean's Summer Research Internship award.

REFERENCES

[1] S.-S. Ahn, S. Park, M. Chertkov, and J. Shin, *Minimum weight perfect matching via blossom belief propagation*, in Advances in Neural Information Processing Systems, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, eds., vol. 28, Curran Associates, Inc., 2015, https://proceedings.neurips.cc/paper_files/paper/2015/file/dc5689792e08eb2e219dce49e64c885b-Paper.pdf.

[2] J. Alman, R. Duan, V. V. Williams, Y. Xu, Z. Xu, and R. Zhou, *More asymmetry yields faster matrix multiplication*, 2024, https://arxiv.org/abs/2404.16349, https://arxiv.org/abs/2404.16349.

[3] D. Coppersmith and S. Winograd, *Matrix multiplication via arithmetic progressions*, Journal of Symbolic Computation, 9 (1990), pp. 251–280, https://doi.org/https://doi.org/10.1016/S0747-7171(08)80013-2, https://www.sciencedirect.com/science/article/pii/S0747717108800132. Computational algebraic complexity editorial.

[4] R. Duan and S. Pettie, *Linear-time approximation for maximum weight matching*, J. ACM, 61 (2014), https://doi.org/10.1145/2529989, https://doi.org/10.1145/2529989.

[5] J. Edmonds, *Paths, trees, and flowers*, Canadian Journal of Mathematics, 17 (1965), p. 449–467, https://doi.org/10.4153/CJM-1965-045-4.

[6] J. Edmonds and E. Johnson, *Matching, euler tours and the chinese postman*, Mathematical Programming, 5 (1973), pp. 88–124, https://doi.org/10.1007/BF01580113.

[7] S. Even and O. Kariv, *An $O(n^{2.5})$ algorithm for maximum matching in general graphs*, in Proceedings of the 16th Annual Symposium on Foundations of Computer Science, SFCS '75, USA, 1975, IEEE Computer Society, p. 100–112, https://doi.org/10.1109/SFCS.1975.5, https://doi.org/10.1109/SFCS.1975.5.

[8] C. Fieker, W. Hart, T. Hofmann, and F. Johansson, *Nemo/hecke: Computer algebra and number theory packages for the julia programming language*, in Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC '17, New York, NY, USA, 2017, ACM, pp. 157–164, https://doi.org/10.1145/3087604.3087611, http://doi.acm.org/10.1145/3087604.3087611.

[9] H. N. Gabow and R. E. Tarjan, *Faster scaling algorithms for network problems*, SIAM Journal on Computing, 18 (1989), pp. 1013–1036.

[10] J. F. Geelen, *An algebraic matching algorithm*, Combinatorica, 20 (2000), pp. 61–70.

[11] A. V. Goldberg and A. V. Karzanov, *Maximum skew-symmetric flows and matchings*, Mathematical Programming, 100 (2004), pp. 537–568.

[12] F. Hadlock, *Finding a maximum cut of a planar graph in polynomial time*, SIAM Journal on Computing, 4 (1975), pp. 221–225, https://doi.org/10.1137/0204019, https://doi.org/10.1137/0204019, https://arxiv.org/abs/https://doi.org/10.1137/0204019.

[13] E. Hamuda, B. Mc Ginley, M. Glavin, and E. Jones, *Improved image processing-based crop detection using kalman filtering and the hungarian algorithm*, Computers and Electronics in Agriculture, 148 (2018), pp. 37–44, https://doi.org/https://doi.org/10.1016/j.compag.2018.02.027, https://www.sciencedirect.com/science/article/pii/S0168169917307718.

[14] N. J. A. Harvey, *Algebraic algorithms for matching and matroid problems*, SIAM Journal on Computing, 39 (2009), pp. 679–702, https://doi.org/10.1137/070684008.

[15] D. S. Johnson and C. C. McGeoch, *Network Flows and Matching: First DIMACS Implementation Challenge*, American Mathematical Society, USA, 1993.

[16] V. Kolmogorov, *Blossom v: a new implementation of a minimum cost perfect matching algorithm*, Mathematical Programming Computation, 1 (2009), pp. 43–67.

[17] F. Le Gall, *Powers of tensors and fast matrix multiplication*, in Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation, ISSAC '14, New York, NY, USA, 2014, Association for Computing Machinery, p. 296–303, https://doi.org/10.1145/2608628.2608664, https://doi.org/10.1145/2608628.2608664.

[18] D. Li, S. Tan, J. Zhang, and W. Jiang, *Hungarian method in a car-sharing system application research*, in Advances in Artificial Intelligence and Security, X. Sun, X. Zhang, Z. Xia, and E. Bertino, eds., Cham, 2021, Springer International Publishing, pp. 265–275.

[19] L. Lovász, *On determinants, matchings, and random algorithms.*, in FCT, vol. 79, 1979, pp. 565–574.

[20] M. MacLean, J. Lysikowski, R. Rege, D. Sendelbach, and A. Mihalic, *Optimizing medical student clerkship schedules using a novel application of the hungarian algorithm*, Academic Medicine, Publish Ahead of Print (2020), https://doi.org/10.1097/ACM.0000000000003676.

[21] K. Mehlhorn and G. Schaefer, *Implementation of $o(nm \log n)$ weighted matchings in general graphs: The power of data structures*, 01 2002, https://doi.org/10.1007/3-540-44691-5_3.

[22] S. Micali and V. V. Vazirani, *An $O(\sqrt{V}E)$ algorithm for finding maximum matching in general graphs*, in 21st Annual Symposium on Foundations of Computer Science (sfcs 1980), 1980, pp. 17–27, https://doi.org/10.1109/SFCS.1980.12.

[23] mlewe, *Blossomv.jl (v0.4.3)*, 2020, https://github.com/mlewe/BlossomV.jl.

[24] M. Mucha, *Finding maximum matchings via Gaussian elimination*, PhD thesis, Warsaw University, 2005.

[25] M. Mucha and P. Sankowski, *Maximum matchings via gaussian elimination*, in 45th Annual IEEE Symposium on Foundations of Computer Science, 2004, pp. 248–255, https://doi.org/10.1109/FOCS.2004.40.

[26] A. Pham, *An empirical study of algebraic algorithms in perfect matching*, https://github.com/apham0014/AN-EMPIRICAL-STUDY-OF-ALGEBRAIC-ALGORITHMS-IN-PERFECT-MATCHING.

[27] M. O. Rabin and V. V. Vazirani, *Maximum matchings in general graphs through randomization*, Journal of Algorithms, 10 (1989), pp. 557–567, https://doi.org/https://doi.org/10.1016/0196-6774(89)90005-9, https://www.sciencedirect.com/science/article/pii/0196677489900059.

[28] J. T. Schwartz, *Fast probabilistic algorithms for verification of polynomial identities*, Journal of the ACM (JACM), 27 (1980), pp. 701–717.

[29] V. Strassen, *Gaussian elimination is not optimal*, Numerische mathematik, 13 (1969), pp. 354–356.

[30] W. T. Tutte, *The factorization of linear graphs*, Journal of the London Mathematical Society, 1 (1947), pp. 107–111.

[31] B. Xu, K. Han, Q. Ren, W. Gong, F. Shao, Y. Wang, and J. Chang, *An optimized strategy for inter-satellite links assignments in gnss*, Advances in Space Research, 71 (2023), pp. 720–730, https://doi.org/https://doi.org/10.1016/j.asr.2022.08.082, https://www.sciencedirect.com/science/article/pii/S0273117722008183.

[32] S. Zhang, Y. Xue, H. Zhang, X. Zhou, K. Li, and R. Liu, *Improved hungarian algorithm–based task scheduling optimization strategy for remote sensing big data processing*, Geo-spatial Information Science, 27 (2024), pp. 1141–1154, https://doi.org/10.1080/10095020.2023.2178339, https://doi.org/10.1080/10095020.2023.2178339, https://arxiv.org/abs/https://doi.org/10.1080/10095020.2023.2178339.

[33] R. Zippel, *Probabilistic algorithms for sparse polynomials*, in International symposium on symbolic and algebraic manipulation, Springer, 1979, pp. 216–226.