Performance Evaluation of Maximum Matching Algorithm Implementations

Jeremy Perez Janak Subedi

Computer Science Department, Union College

Question

How do variations in graph structure impact the efficiency of implementations of Maximum Matching algorithms?

Introduction

The maximum matching problem is a theoretical computational problem focused on finding the largest set of edges in a graph where no two edges share a vertex.

The complexity of the max matching problem varies significantly based on the type of graph. The classes of approaches likewise are significantly different.

- For **bipartite graphs** ($d = 2$), the problem is solvable in polynomial time.

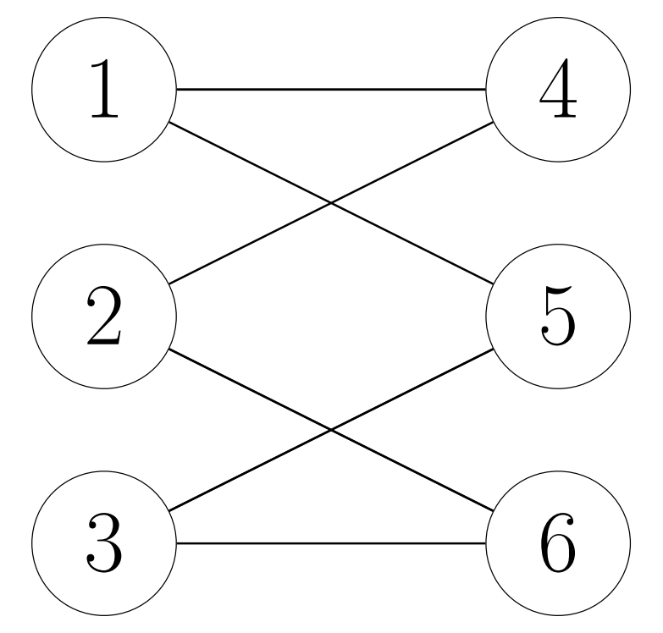

Figure 1:A bipartite graph with maximum matching of 3

- For **d-partite graphs** ($d > 2$), the problem becomes NP-hard, requiring more complex and often approximate methods.

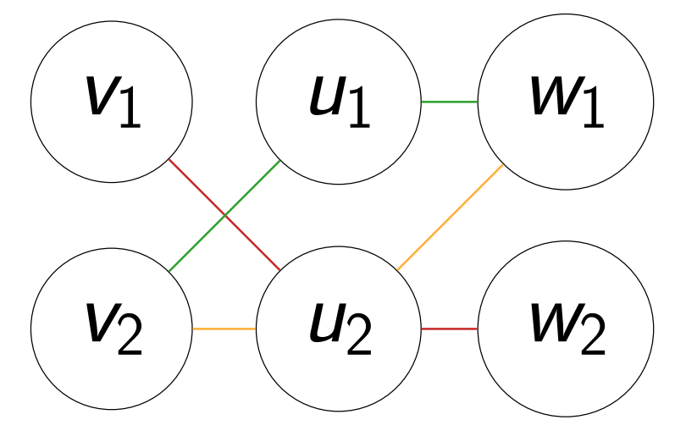

Figure 2:A tripartite graph with maximum matching of 2

Classes of algorithms which we've implemented include

- Greedy Algorithms
- Tree Traversal Algorithms
- Heuristic Algorithms
- Approximation Algorithms
- Network Flow Algorithms
- Integer Programming Algorithms

Methods

We focus on developing **courses**, which are specialized test sets, meant to highlight an algorithm's dependence on a feature of the graph.

We then performed a series of performance evaluations by **racing** every algorithm across a series of these courses and assessing each algorithm's accuracy, and run time on these courses. We hope to capture performance disparities between each algorithm.

Races and Courses

Each course was developed to emphasize different graph characteristics based on the following properties:

- d: the number of partitions in the graph,
- n: the number of vertices,
- m: the number of edges, and
- E: the set of edges.

By treating all possible graph configurations as a set A, any subset $C \subseteq A$ can represent a course. Since $A = C \cup C^C$, we made sure to include the complement sets as additional courses. Some examples of courses we have studied are:

1. Perfect Matching ($n = m$)
2. Non-Perfect Matching ($n! = m$)
3. Large Vertex Count ($n > 1000$)
4. Dense Graphs ($m > \frac{n^d}{2}$)
5. Sparse Graphs ($m << \frac{n^d}{2}$)

Performance Metrics

For a given solver, let C denote the set of test cases in a course. Let $S \subseteq C$ denote the set of solved test cases. For any test case $e \in C$, let t_e denote the time taken to complete a test case in seconds.

- **Accuracy** measures the number of cases correctly solved over the total number of cases with timeouts.
- The **Total Time** (TT) metric is the total time taken to solve an entire course.

$$TT = \sum_{e \in S} t_e$$

Results for Bipartite Solvers

Solver	Courses With 1s Timeout			
	N > 1000		Non-Perfect	
	TT (s)	ACC (%)	TT (s)	ACC(%)
● Vinh-Hopcroft-K	1	100	7	100
● Kurtik-Hopcroft-K	2	100	7	100
● Vinh-Edmond-K	2	100	8	100
● EthanO-Hopcroft-K	11	100	54	100
● Jacob-PULP-IP	12	100	35	100
● Kurtik-Ford-Fulkerson	21	100	28	100
● Huyen-Edmonds-K	80	19.1	90	84.1
● Jeremy-GPT-Approx	89	0	261	70.86

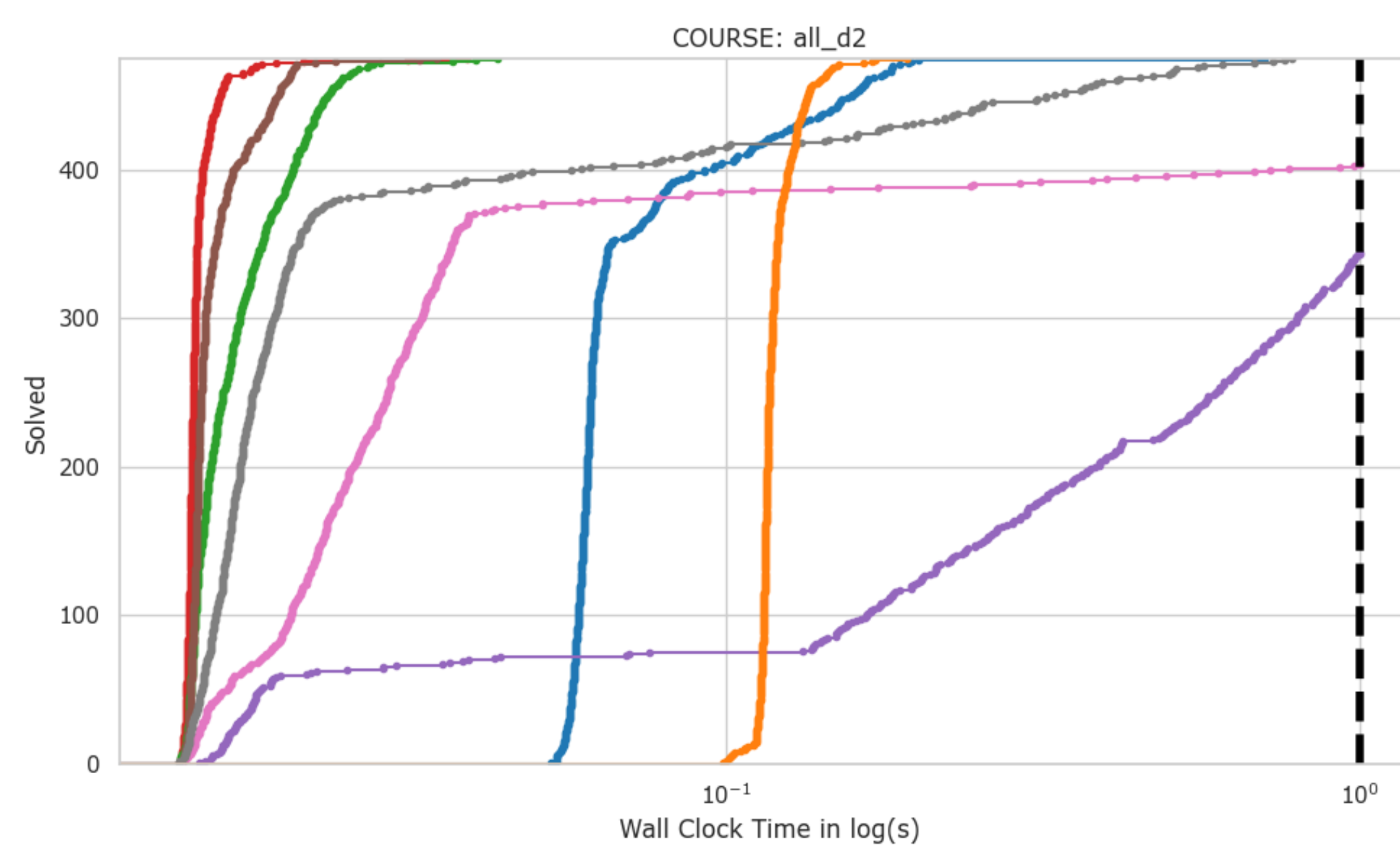

Figure 3:Performance of d2 Solvers against all test with a 1 second timeout

With a 1 second timeout, some algorithms performed fairly bad, for instance, the GPT-Approximation algorithm was not able to complete a single large graph instance for graph's with more than 1000 vertices. This clearly showcases that the algorithm is really bad at handling large graphs where as, an algorithm like Hopcroft Karp performed fairly well. This framework succeeds at effectively comparing algorithms runtimes as intended.

By varying the runtime between implementations of the same algorithm, we can visualize just how efficient they are with respect to each other.

Our preliminary results indicate that Hopcroft-Karp remains to be the fastest algorithm for Bipartite problems.

This seems to work as an informal runtime analysis.

Results for Hypergraph Solvers

Solver	Courses With 5m Timeout			
	Random		Non-Perfect	
	TT (s)	ACC (%)	TT (s)	ACC(%)
● Janak-BruteForce	148	79.6	40	18.68
● Jeremy-A*	655	11.66	10	86.81
● Jacob-PULP-IP	938	99.17	646	96.70
● Jeremy-CPLEX-IP	60	87.10	60	13.19
● Jeremy-Greedy	2	3.03	2	9.89

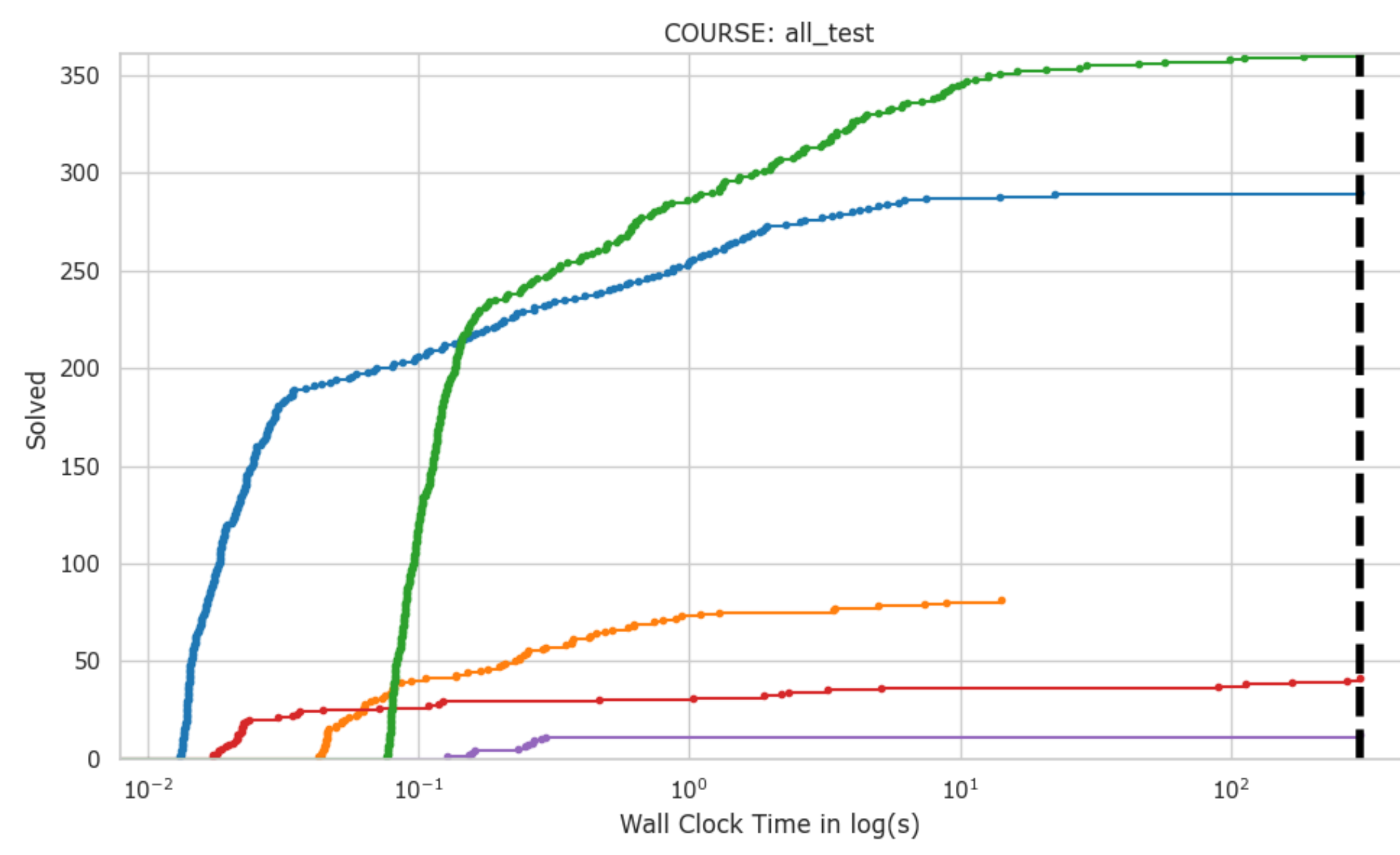

Figure 4:Performance of d-partite Solvers against all tests with a 5-minute timeout.

Conclusions

After seeing great disparities in performance in d-partite algorithms, understanding how we've partitioned the solution space and how algorithms traverse into it, we find this to be an effective model for partitioning all possible graph configurations.

References

[1] Hopcroft, J. E., & Karp, R. M. (1973). An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs. *SIAM Journal on Computing*, 2(4), 225-231. doi:10.1137/0202019.

[2] Edmonds, J., & Karp, R. M. (1972). Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *J. ACM*, 19(2), 248-264. doi:10.1145/321694.321699.

[3] Ford, L. R., & Fulkerson, D. R. (1956). Maximal Flow Through a Network. *Canadian Journal of Mathematics*, 8, 399-404. doi:10.4153/CJM-1956-045-5.

[4] Mitchell, S. D., & Roy, J. (2023). PuLP: A Python Library for Linear Optimization. Available at: `https://coin-or.github.io/pulp/`. Accessed: 2023-11-05.

[5] IBM ILOG CPLEX. (2023). CPLEX Optimizer. Available at: `https://www.ibm.com/products/ilog-cplex-optimization-studio`. Accessed: 2023-11-05.