New Experimental Results for Bipartite Matching

João C. Setubal

Relatório Técnico DCC–07/92

Novembro de 1992

New Experimental Results for Bipartite Matching

João C. Setubal *

November 12, 1992

Abstract

We present experimental results for 3 bipartite matching algorithms on three classes of sparse graphs. Goldberg's maximum flow algorithm [Gol87, GT88b], specialized for unweighted bipartite graphs, is the most robust algorithm, being the fastest by a significant margin in two of the classes and competitive in the other one. The other two algorithms are Hopcroft and Karp's [HK73] and Alt, Blum, Mehlhorn, and Paul's [ABMP91], and both have better worst-case bounds than Goldberg's algorithm. The input classes used are variations of random bipartite graphs. We also show that speed-ups of up to 3.2 with respect to the sequential implementation can be obtained by parallelizing Goldberg's algorithm on a shared-memory multiprocessor using up to 12 processors.

1 Introduction

The *bipartite matching problem* is: given a bipartite graph $G = (U, V, E)$, with $n = |U| + |V|$ and $m = |E|$, we want to find a set of edges of maximum cardinality such that no edge in the set shares a vertex with any other vertex in the set. This set is the *maximum matching*. In

*Supported in part by Richard Anderson's NSF CER grant CCR-861966 and Brazilian Agency FAPESP grant 87/1385-7. This work was done while the author was on leave from State University of Campinas (UNICAMP, Brazil). E-mail address setubal@dcc.unicamp.br.

the case of sparse graphs the best sequential algorithm is by Hopcroft and Karp [HK73], which achieves a worst-case running time of $O(\sqrt{n}m)$. For dense graphs, the best algorithm is by Alt, Blum, Mehlhorn, and Paul [ABMP91], having a worst-case bound of $O(n^{1.5}\sqrt{m/\log n})$.

It is well known that bipartite matching is just a special case of the maximum flow problem. In fact, Hopcroft and Karp's algorithm can be seen as a version of Dinic's algorithm for maximum flow [Din70], specialized for unweighted bipartite graphs [ET75]. Experimental results for the maximum flow problem [DM89, AS92a] have shown that Goldberg's algorithm for maximum flow [Gol87, GT88b] is the fastest in practice for a large number of input classes, outperforming Dinic's algorithm (the second fastest) by a large margin. Given these results, it is natural to ask how do these two algorithms compare when solving bipartite matching problems in practice. The answer is not obvious since Goldberg's algorithm is very different from Dinic's algorithm. In addition, when Goldberg's algorithm is specialized for unweighted bipartite graphs it has a complexity of $O(nm)$. The algorithm by Alt, Blum, Mehlhorn, and Paul (henceforth called the ABMP algorithm) is the third algorithm considered, and it can be seen as a cross between Dinic's and Goldberg's algorithms. The experimental results presented in this paper indicate that for two of the input classes considered Goldberg's algorithm is indeed the faster algorithm, and the ABMP algorithm is the fastest on the other class, with Goldberg's a close second. In one of the input classes, Goldberg's algorithm is nearly 10 times faster than Dinic's and three times faster than the ABMP algorithm. The input classes considered are three variations of random bipartite graphs; they were designed to be representative of bipartite matching problems that might arise in practice. The largest instances solved have 120000 vertices.

On another set of experiments, Anderson and Setubal [AS92b] have shown how to obtain good speed-ups for Goldberg's maximum flow algorithm when implemented on a shared-memory multiprocessor with a small number of processors. So it is again natural to ask whether a parallel implementation of Goldberg's algorithm specialized for unweighted bipartite graphs can achieve some speed-up over the sequential imple-

mentation, on the same kind of platform. And again the answer is not obvious, and for the following reason. Any program to solve a bipartite matching problem should start with an initial matching. The simplest initial matching is the "greedy" one, in which a match is attempted between each vertex in U and the first vertex on its adjacency list. For the input classes described in this paper, this initial greedy matching matches over 80% of the vertices. From experience gained with the maximum flow problem [AS92b] it was feared that not much parallelism would be available to match the remaining 20% or so vertices. Yet in this paper we report success in obtaining speed-ups for all inputs. The numbers obtained look somewhat modest, ranging from 2.4 to 3.2 with 12 processors on the largest instances, but the point is that we demonstrate that *some* speed-up is possible. One application for such an implementation would be in the initial matching phase for a parallel algorithm for the assignment problem, such as the one described in [BMPT91]. We would also like to note that parallel bipartite matching has received considerable attention from the theory community [Gro92, GPST92, GPV88, GT88a, SM89], but the algorithms described do not seem suitable for implementation.

The remaining of the paper is divided in two parts: in the first (section 2), the sequential experiments are described, including implementations, test data and methodology, and results. In the second part (section 3), the same structure is used to describe the parallel experiments. Conclusions are given at the end.

2 Sequential Experiments

2.1 Implementations

In this section we briefly describe the particulars of our implementations of Dinic's, Goldberg's and ABMP algorithms. We assume the reader is familiar with all three algorithms.

All implementations work with the two partitions, U and V, as separate entities, each partition having its own data structure. All use the same initial greedy matching mentioned in the previous section. Dinic's

and Goldberg's algorithms use the concept of *source* and *sink* implicitly. (The source is considered to be on the U side and the sink on the V side.) Finally, all programs are relatively straightforward implementations of their respective algorithms and no special tricks were used, in order to keep the comparison as fair as possible. We note, however, that the ABMP implementation was the least mature of the three presented here.

The Dinic implementation has the following basic features:

- the implementation is able to work almost exclusively with the U partition, for both the layered graph construction and the augmenting path search. The vertices in the V partition serve only as "pointers" to vertices in the U partition, or to the implicit sink.
- the layered graph is also implicit, given by appropriate flag settings on the edges.
- two sub-implementations are presented: one (denoted by dinic_1) uses a strict layered graph; another (denoted by dinic_2) allows (implicit) layered-graph edges between exposed vertices in V and the sink even when the layer to which these vertices belong is equal to or greater than the sink's.

The Goldberg implementation has the following features:

- The implementation works with both U and V partitions, meaning that active vertices can belong to either of the two. However an active vertex is treated differently depending on whether it belongs to U or to V. For a vertex belonging to U to be active means that it is not currently matched and all edges incident to it belong to the residual graph. An active vertex belonging to V has been matched by more than one vertex, but some of the edges incident to it may not belong to the residual graph. These differences are reflected both in the push and relabel operations.
- The active vertices are processed in FIFO order, and discharge of an active vertex is done until its excess is zero.

- the initial greedy matching is complemented by the following action: if a vertex belonging to U was not able to find an unmatched vertex in V, it matches with its first neighbor in its adjacency list. As a consequence at the start all vertice in U are matched and the initial active vertices will be the "overmatched" vertices in V.

- a global relabeling heuristic, similar to that described in [AS92a], is used periodically, with frequency equal to $n/2$. This heuristic is a backwards breadth-first search performed on the residual graph, changing labels on vertices from approximate distances to the sink or to the source into exact distances.

The implementation of the ABMP algorithm has the following features:

- As originally proposed, the algorithm processes vertices up to a certain layer, then finds the last augmenting paths by using Dinic's algorithm. In this implementation we simply let the algorithm process all layers. A heuristic is used to determine when to stop processing vertices, and is based on the gap relabeling technique of Derigs and Meier [DM89].

- A global relabeling routine, very similar to the one used in the Goldberg implementation, is invoked periodically to relabel all vertices. The routine is called after every n relabels.

- A queue is used to manage the unmatched vertices in U. Every time the global relabeling routine is invoked, it flushes the queue and fills it with the currently unmatched vertices in U in the proper order.

2.2 Environment, input classes, and methodology of experiments

The sequential experiments were done on a DECstation 5000/125, running ULTRIX 4.1 with 32 MB of main memory. The programs were written in C, compiled with the regular `cc` compiler and using the optimizer

`-O` flag. Running times were measured with the system call `getrusage` by selecting field `ru_utime`.

As mentioned before, the graphs used to test the programs are variations of random bipartite graphs. In these graphs each vertex has an expected number of neighbors. The actual number is obtained by simulating a Poisson random variable which in turn approximates the binomial random variable in a real random graph [Fel68]. We used the value 5 for number of expected edges per vertex. In all classes $|U| = |V|$, and the resulting maximum matchings were always just below perfect, matching around 99% of the vertices. The classes, with the acronyms we use to designate them in this paper, are as follows:

- random (*random*). The neighbors for each vertex in U are chosen at random from all vertices in V.
- few groups (*few-g*). The vertices in U and V are divided into n_1 groups of n_2 vertices each. The neighbors for a vertex belonging to U in group i are chosen at random from vertices in groups $i-1$ and i in V, except for vertices in group 1. These have neighbors in group 1 in V only. The value for n_1 is fixed at 30, and only n_2 varies as we increase the total number of vertices ($n = 2n_1n_2$). Note that the first group in U and the last group in V have only 2.5 expected edges per vertex.
- many groups (*many-g*). Similar to the previous, but n_1 is fixed at 500.

The classes *few-g* and *many-g* were designed having in mind problems that can be reduced to bipartite matching, such as the maximum number of vertex-disjoint paths problem. In these problems the resulting graph in the reduction is bipartite, but if the graph is planar or nearly planar each vertex will only have as neighbors vertices in the surrounding area.

For each class, the programs were tested on instances having 30000, 60000, and 120000 vertices. Given the value selected for the expected number of neighbors per vertex, the number of edges was always approx-

	time (secs)				*Augm*		*Pushes*
n	dinic$_1$	dinic$_2$	goldberg	ABMP	dinic$_1$	dinic$_2$	goldberg
30000	5.9	3.9	2.5	2.0	10	6	37240
60000	15.8	9.7	5.9	4.4	12	6	84162
120000	35.9	22.5	12.5	8.8	13	7	162940
growth rates	1.30	1.26	1.16	1.07	0.19	0.11	1.06

Table 1: Results for class *random*. *Augm* is mean number of augmenting phases. *Pushes* is mean number of push operations.

imately $2.5n$. In each size, 20 instances were solved, using different seeds for the pseudo-random number generator (which was UNIX's `random()`).

Further characteristics of the experiments are as follows:

- At the end of each run the solution is checked for consistency and maximality.
- Running times reported exclude input, checking, and output time, and are means over the 20 instances solved in each size and class.
- Asymptotic performance (growth rate) was estimated by doing a power regression analysis using a standard least-squares method for the fit.

2.3 Results and analysis

In tables 1, 2, and 3 we report the running times, operation counts, and growth rates of these measurements for classes *random*, *few-g*, and *many-g*, respectively. The growth rates reported are approximate both because the number of data points is small and because of variance in the means on which they are based. In most cases the standard deviation of a mean was under 20% of the mean. For the Goldberg implementation on classes *random* and *many-g* the standard deviation of the mean number of pushes was as high as 29% of the mean; for the ABMP implementation

	time (secs)				*Augm*		*Pushes*
n	dinic$_1$	dinic$_2$	goldberg	ABMP	dinic$_1$	dinic$_2$	goldberg
30000	28.0	21.8	4.3	11.3	59	45	89202
60000	100.3	69.5	9.8	30.3	94	65	188318
120000	341.2	222.9	23.5	78.4	137	90	419030
growth rates	1.80	1.68	1.23	1.40	0.61	0.50	1.12

Table 2: Results for class *few-g*. *Augm* is mean number of augmenting phases. *Pushes* is mean number of push operations.

	time (secs)				*Augm*		*Pushes*
n	dinic$_1$	dinic$_2$	goldberg	ABMP	dinic$_1$	dinic$_2$	goldberg
30000	11.1	4.2	4.5	5.8	31	9	100962
60000	26.9	11.0	8.9	10.2	34	11	188791
120000	88.4	38.1	21.5	30.8	53	19	451838
growth rates	1.50	1.59	1.13	1.20	0.39	0.54	1.08

Table 3: Results for class *many-g*. *Augm* is mean number of augmenting phases. *Pushes* is mean number of push operations.

on class *many-g* the standard deviation for the running times were as high as 28% of the mean. Therefore the growth rates in those cases may be less accurate.

In any case it is clear from the tables that Goldberg's algorithm did very well. It was the fastest algorithm in two of the classes considered and the second fastest on the other class. In class *few-g* in particular it was nearly 10 times faster than implementation dinic$_2$ and three times faster than ABMP for the largest size tested. Note that the running time growth rates for Goldberg's algorithm are much better than the worst-case bounds.

Why did Dinic's algorithm, which has the best worst-case bound, fare so poorly? Profiles of Dinic's algorithm execution indicate that most of the time (between 75 and 85%) is spent in layered-graph construction, and the rest on augmenting path search. This imbalance is due to the following fact. In the initial augmenting phases many augmenting paths are found in each phase. In the final phases, however, only a few augmenting paths are discovered, but the whole cost of layered-graph construction has to be paid even if only one such path is found. Thus Dinic's algorithm fared relatively well when the number of phases was small, as in the *random* class, but did poorly when the number of phases was large, as in class *few-g*. The ABMP algorithm, which still looks for augmenting paths but does not have the concept of phases, fares better than Dinic's in all cases. Goldberg's algorithm, on the other hand, not only does not work in phases but also does away with the need for finding augmenting paths, and thus came out as the most robust algorithm, having similar running times in all classes considered.

Some other observations can be made by comparing the two Dinic implementations. Implementation dinic$_2$ was much better than dinic$_1$ even though dinic$_2$ can potentially have longer augmenting paths. Another observation is that some of the growth rates measured for these implementations do not agree with worst-case prediction. For sparse graphs, Dinic's algorithm has complexity $O(n^{1.5})$, and we see that implementation dinic$_1$ on class *few-g* and implementation dinic$_2$ on class *many-g* have running time growth rates larger than 1.5. This can also be

seen in the augmenting phases growth rate, which should be at most 0.5. On the other hand, the number of augmenting phases for a given size is much less than $\sqrt{n}$, the worst-case bound. Finally, it was surprising to see that Dinic's algorithm had better performance on class *many-g* than in class *few-g*, since we presumed that in class *many-g* the augmenting paths would be longer and lead to more augmenting phases.

3 Parallel Experiments

3.1 Implementation

The parallel implementation of Goldberg's algorithm follows the same structure as the sequential implementation and therefore it is relatively simple. The specific techniques for parallel implementation were the same as those described in [AS92b], with the following minor locking simplification: when relabeling an active vertex in U, it is not necessary to keep the vertex locked while scanning its edges, but only when its label is updated.

3.2 Environment, input classes, and methodology of experiments

The experiments were conducted on a Sequent Symmetry S81 with 20 Intel 16Mhz 80386 processors, and 32 megabytes of memory, running DYNIX 3.0. Each processor has a 64 Kbyte cache memory. The program was written in C using the Parallel Programming Library provided with Sequent systems, which allows the forking of processes, one per processor.

The input classes, sizes, and methodology used were similar to those for the sequential experiments, with the following differences:

- the number of expected edges per vertex was 4; as a result, $m \approx 2n$.
- only 10 instances per class and size were solved, but each instance was solved 4 times and running time for an instance was taken as the mean over these 4 runs. This is necessary because we observed significant variations from run to run.

$n = 30000$; speed-up = 2.4						
p	*seq*	1	2	4	8	12
time	8.7	15.7	9.1	5.9	4.2	3.9
disch	43238	46033	48139	51081	58249	70477
relab	17588	18812	20035	22179	29230	40934
$n = 60000$; speed-up = 2.6						
p	seq	1	2	4	8	12
time	18.5	33.5	18.7	11.3	8.4	7.2
disch	89254	97266	100888	105029	119027	143774
relab	36634	40822	42766	46083	60632	82911
$n = 120000$; speed-up = 3.2						
p	seq	1	2	4	8	12
time	77.1	134.7	65.4	36.4	27.0	24.7
disch	195755	208774	223232	231655	254705	280575
relab	82513	88646	98184	105154	130295	152426

Table 4: Results for class *random*. Time is in seconds; p is number of processors, and *seq* labels the column corresponding to the sequential program. Disch is mean number of discharges; relab is mean number of relabel operations.

- each instance was solved using 1, 2, 4, 8, and 12 processors.

3.3 Results and analysis

Tables 4, 5, and 6 present the results for the parallel implementation on classes *random*, *few-g*, and *many-g*, respectively. First note that each speed-up figure reported is the mean over the individual speed-ups computed for each instance. This individual speed-up in turn is computed as the sequential time for an instance divided by the mean time over 4 runs with 12 processors for that instance. As can be seen in the tables a speed-up was obtained for every class and size tested, and the speed-ups tend to get better as the size increases. On the other hand, at 12 pro-

$n = 30000$; speed-up = 1.9						
p	*seq*	1	2	4	8	12
time	13.8	26.9	15.5	10.0	7.5	7.4
disch	80356	89606	89919	92810	101417	115213
relab	37364	40707	41445	44323	53430	64887
$n = 60000$; speed-up = 2.6						
p	seq	1	2	4	8	12
time	33.5	61.4	34.8	20.8	14.8	13.0
disch	189324	203474	210476	215206	238042	277946
relab	89819	93791	99119	105415	127473	159276
$n = 120000$; speed-up = 3.1						
p	seq	1	2	4	8	12
time	87.6	152.5	85.0	53.1	33.7	28.5
disch	410474	426162	426565	450914	465816	528364
relab	196274	197184	199295	219446	240936	291049

Table 5: Results for class *few-g*. Time is in seconds; p is number of processors, and *seq* labels the column corresponding to the sequential program. Disch is mean number of discharges; relab is mean number of relabel operations.

$n = 30000$; speed-up = 2.1						
p	*seq*	1	2	4	8	12
time	14.5	25.9	14.7	9.4	7.1	6.9
disch	87164	90228	90029	93759	106636	126676
relab	39106	39639	40418	44045	54753	70406
$n = 60000$; speed-up = 2.0						
p	seq	1	2	4	8	12
time	35.5	65.5	38.2	24.0	19.0	18.2
disch	210858	229238	222816	221149	247307	283217
relab	97523	104207	102027	103839	125486	153867
$n = 120000$; speed-up = 2.4						
p	seq	1	2	4	8	12
time	85.3	151.8	88.1	58.8	39.6	39.9
disch	416300	456833	451365	463682	478123	563417
relab	193499	208765	208621	220081	241405	297968

Table 6: Results for class *many-g*. Time is in seconds; p is number of processors, and *seq* labels the column corresponding to the sequential program. Disch is mean number of discharges; relab is mean number of relabel operations.

cessors the decrease in running time over 8 processors is quite small, and in some cases there was actually an increase. The count disch reported on the tables refers to the action of getting an active vertex and pushing flow from it until its excess is zero.

Similarly to the maximum flow study [AS92b], hardware effects such as bus contention, and lack of parallelism (insufficient number of active vertices to keep all processors busy most of the time) are the main reasons that explain why it is difficult to obtain better speed-ups. However, one additional reason specific to this work is that the frequency of global relabelings at 12 processors is not high enough. This is what happens: by the time a global relabeling is completed, the next global relabeling is already due, because in the meantime the processors were able to complete more than the total number of discharges necessary to trigger the next global relabeling. As a consequence the number of discharges and relabel operations increases markedly when the number of processors increases from 8 to 12, as can be seen in the tables. It should be possible to overcome this problem, either by a careful tuning of the parameters involved or by having more than one global relabeling at the same time. However, we do not expect any large improvements in the speed-ups even with these changes.

4 Conclusions and further work

The basic conclusion that can be drawn from this study is that the practical efficiency of the algorithmic ideas in Goldberg's method for solving the maximum flow problem also extend to the more specialized bipartite matching problem. The results in this paper, coupled with results for the maximum flow problem [DM89, AS92a], and for the more general min-cost max-flow problem [GK92], show the remarkable power of the push-relabel method in practice. We believe that the results obtained in this work reinforce the need for a theoretical study of average case performance of Goldberg's algorithm, just as was observed in the maximum flow study.

In terms of other experiments, it would be interesting to test the

implementations presented in this paper on other classes of bipartite graphs.

5 Acknowledgments

I would like to thank Richard Anderson for a critical reading of the manuscript. In addition, the foundation of this work lies entirely in our joint work on the maximum flow problem.

References

[ABMP91] H. Alt, N. Blum, K. Mehlhorn, and M. Paul. Computing a maximum cardinality matching in a bipartite graph in time $O(n^{1.5}\sqrt{m/\log n})$. *Inform. Process. Lett.*, 37:237–240, 1991.

[AS92a] R. J. Anderson and J. C. Setubal. Goldberg's algorithm for maximum flow in perspective: a computational study. In David S. Johnson and Catherine C. McGeoch, editors, DIMACS *Implementation Challenge Workshop – Algorithms for network flows and matching*. DIMACS, 1992. To appear.

[AS92b] R. J. Anderson and J. C. Setubal. On the parallel implementation of Goldberg's maximum flow algorithm. In *Proc. 4th Symp. on Parallel Algorithms and Architectures*, pages 168–177. ACM Press, 1992.

[BMPT91] E. Balas, D. Miller, J. Pekny, and P. Toth. A parallel shortest augmenting path algorithm for the assignment problem. *JACM*, 38(4):985–1004, 1991.

[Din70] E. A. Dinic. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.

[DM89] U. Derigs and W. Meier. Implementing Goldberg's max-flow-algorithm — a computational investigation. *ZOR — Methods and Models of Operations Research*, 33:383–403, 1989.

[ET75] S. Even and R. E. Tarjan. Network flow and testing graph connectivity. *SIAM J. Comput.*, 4(4):507–518, 1975.

[Fel68] W. Feller. *An Introduction to Probability Theory and its Applications*, volume 1. John Wiley & Sons, New York, N. Y., third edition, 1968.

[GK92] A. V. Goldberg and M. Kharitonov. On implementing scaling push-relabel algorithms for the minimum-cost flow problem. Technical Report STAN-CS-92-1418, Department of Computer Science, Stanford University, March 1992.

[Gol87] A. V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers*. PhD thesis, Massachussetts Institute of Technology, Cambridge, Mass., January 1987.

[GPST92] A. V. Goldberg, S. A. Plotkin, D. B. Shmoys, and E. Tardos. Interior point methods in parallel computation. *SIAM J. Comput.*, February 1992.

[GPV88] A. V. Goldberg, S. A. Plotkin, and P. M. Vaidya. Sublinear-time parallel algorithms for matching and related problems. In *Proceedings of the IEEE Twenty-Ninth Annual Symposium on Foundations of Computer Science*, pages 174–185, 1988.

[Gro92] L. K. Grover. Fast parallel algorithms for bipartite matching. Technical report, School of Electrical Engineering, Cornell University, May 1992.

[GT88a] H. N. Gabow and R. E. Tarjan. Almost-optimum speed-ups for bipartite matching and related problems. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, pages 514–527, 1988.

[GT88b] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. Assoc. Comput. Mach.*, 35(4):921–940, 1988.

[HK73] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.

[SM89] B. Schieber and S. Moran. Parallel algorithms for maximum bipartite matchings and maximum 0-1 flows. *J. Parallel and Distributed Computing*, 6:20–38, 1989.

Relatórios Técnicos

01/92 **Applications of Finite Automata Representing Large Vocabularies,** *C. L. Lucchesi, T. Kowaltowski*

02/92 **Point Set Pattern Matching in d-Dimensions,** *P. J. de Rezende, D. T. Lee*

03/92 **On the Irrelevance of Edge Orientations on the Acyclic Directed Two Disjoint Paths Problem,** *C. L. Lucchesi, M. C. M. T. Giglio*

04/92 **A Note on Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams,** *W. Jacometti*

05/92 **An (l, u)-Transversal Theorem for Bipartite Graphs,** *C. L. Lucchesi, D. H. Younger*

06/92 **Implementing Integrity Control in Active Databases,** *C. B. Medeiros, M. J. Andrade*

Departamento de Ciência da Computação — IMECC
Caixa Postal 6065
Universidade Estadual de Campinas
13081-970 – Campinas – SP
BRASIL

`reltec@dcc.unicamp.br`