Sequential and Parallel Experimental Results with Bipartite Matching Algorithms

João C. Setubal

Relatório Técnico IC–96-09

Setembro de 1996

Sequential and Parallel Experimental Results with Bipartite Matching Algorithms*

João C. Setubal

Institute of Computing, CP 6176
University of Campinas
Campinas, SP 13083-970
Brazil
setubal@dcc.unicamp.br

Abstract

We present experimental results for four bipartite matching algorithms on 11 classes of graphs. The algorithms are depth-first search (DFS), breadth-first search (BFS), the push-relabel algorithm [GT88b], and the algorithm by Alt, Blum, Mehlhorn, and Paul (ABMP) [ABMP91]. DFS was thought to be a good choice for bipartite matching but our results show that, depending on the input graph, it can have very poor performance. BFS on the other hand has generally very good performance. The results also show that the ABMP and push-relabel implementations are similar in performance, but ABMP was faster in most cases. We did not find a clear-cut advantage of ABMP over BFS or vice-versa, but both the ABMP and push-relabel implementations have generally smaller growth rates than BFS, and should thus be preferred if very large problems are to be solved. For small problems BFS is the best choice. We also present experimental results from a parallel implementation of the push-relabel algorithm, showing that it can be up to three times faster than its sequential implementation, 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 $M \subseteq E$ of maximum cardinality such that no edge in the set shares a vertex with any other edge in the set. This set is a *maximum matching*. From a computational complexity point of view, in the case of sparse graphs, the best sequential algorithm for finding a maximum matching 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 $O(n^{1.5}\sqrt{m/\log n})$.

*This work was supported in part by grants from Brazilian Research Agencies FAPESP and CNPq.

Bipartite matching is an important problem from a practical point of view, since it has many applications [AMO93]. Therefore it is important to know what algorithms have good performance in practice. Other computational studies of bipartite matching algorithms have been done in the past, of which we are aware of two: Darby-Dowman [DD80] and Chang and McCormick [CM90]. These studies showed or mentioned that implementations of Hopcroft and Karp's algorithm were considerably slower than implementations of two simple augmenting-path algorithms: depth-first search (DFS) and breadth-first search (BFS). These two algorithms have worst-case running time of $O(nm)$. Darby-Dowman's experiments did not show a significant advantage of one type of search over the other, except that he noted that by comparing total execution time on all his experiments BFS "was found to be over 10% faster than" DFS, but no implementation "performed consistently better" on individual test cases. Chang and McCormick, on the other hand, did not even consider BFS, comparing their implementation of DFS to two others of DFS as well. The implication seems to be that DFS is *the* algorithm to solve bipartite matching problems in practice, in particular with a heuristic used by Chang and McCormick.

Two new algorithms for bipartite matching have arisen in the past decade: one is the already mentioned algorithm by Alt, Blum, Mehlhorn, and Paul [ABMP91] (which we call ABMP), and the other is the push-relabel algorithm, developed by Goldberg [Gol87] and generalized by Goldberg and Tarjan [GT88b]. This last algorithm was developed for the maximum flow problem, but it can be readily specialized for bipartite matching, yielding a running time of $O(nm)$. In a preliminary work [Set93] we showed that implementations of the ABMP and push-relabel algorithms were significantly faster than an implementation of Hopcroft-Karp's algorithm. The question then remained: are these new algorithms faster in practice than the simple-search algorithms? This paper tries to answer precisely this question.

In order to answer it we compared an implementation of DFS developed by Chang and McCormick [CM90] to three others that we developed, one for each of the other three algorithms. We used 11 different input classes, with different input sizes and different instances for the same size, resulting in a total of 390 problems. The largest of these problems had 131072 vertices and some 328000 edges. By comparison, in the main body of Darby-Dowman's thesis [DD80] results from runs on 11 problems are reported, the largest having no more than 1000 vertices and no more than 2500 edges; Chang and McCormick [CM90] solved 60 problems, the largest with 4000 vertices and 147952 edges.

A summary of the results is as follows. We found that there are several classes where DFS's performance is dismal compared to the others, being in one case two orders of magnitude slower than the fastest implementation. The ABMP and push-relabel implementations had similar performance, but ABMP was faster in most cases. BFS was in general faster than ABMP for small graphs (up to thousands of vertices), while ABMP was faster for large graphs in five of the classes.

The push-relabel algorithm has one advantage over the others: it can be parallelized with relative ease. We have done this and compared the parallel implementation to the sequential implementation, observing a speed-up of up to 3.2 with 12 shared-memory processors. With such a speed-up the push-relabel implementation can be the fastest of all implementations tested in several of the input classes studied.

The paper is structured as follows. We briefly describe the algorithms and their sequential implementations in section 2. We then describe how the sequential experiments were conducted in section 3. Results and analysis are presented in section 4. A general conclusion of the sequential experiments is given in section 5. The remaining sections describe the parallel experiments, and final comments are made at the end.

2 Sequential Implementations

2.1 General information

All implementations developed by the author deal with the two partitions, U and V, as separate entities, each partition having its own data structure. Thus we speak of a u vertex, $u \in U$, or of a v vertex, $v \in V$. The main data structure used is an adjacency list for each vertex.

We note that all implementations (including DFS) consider the vertices in the same order, and adjacency lists are exactly the same.

2.2 Initial Matching

All implementations, in a first phase, find an initial, maximal matching, so all algorithms start with precisely the same maximal matching. A greedy strategy is used: we scan the vertex list of the U partition (according to the input order) and try to find an unmatched neighbor for each u vertex.

2.3 Depth-First-Search

This algorithm finds a maximum matching by looking for augmenting paths in a depth-first manner. We used Chang and McCormick's code [CM90], written in FORTRAN-77, which has the following heuristic to speed up the search: whenever the search fails, all labeled vertices are discarded, since they can't be on any augmenting path. This heuristic improves the implementation's performance when maximum matchings are not perfect.

We replaced the original timing routines in the DFS code with calls to the same C timing interface used in all other implementations (described below). In addition, we wrote and tested a C DFS implementation and compared it to the FORTRAN-77 code, observing essentially no difference in performance. This means that the language (or compiler) was not a factor in the running times.

2.4 Breadth-First-Search

This implementation finds a maximum matching by looking for augmenting paths in a breadth-first manner. We wrote it in C using essentially the same ideas as in the DFS implementation, including the heuristic mentioned above.

2.5 Push-Relabel

We give here a brief description of the algorithm as applied to the bipartite matching problem. A full description can be found in [GT88b]. The algorithm works by applying the push and relabel operations to *active vertices*. A u vertex is active if it is unmatched, and its label is below n. A vertex's label is a lower bound on its distance to an exposed (unmatched) v vertex. The application of push to u consists of matching it to one of its neighbors that has a label with a value one less its own, regardless of whether this neighbor is already matched or not. A v vertex is active if it is "overmatched", that is, more than one u vertex is matched to it. We push from v by unmatching all its mates except one. The vertices to be unmatched also need to have labels one less than the v vertex. From time to time we can only push from an active vertex after a relabel operation: its label is changed to be one above the minimum of its neighbor's labels (mates' labels, in case of a v vertex). The algorithm terminates when there are no more active vertices.

The active vertices are processed in FIFO (queue) order. The initial greedy matching is complemented by the following action: if a u vertex was not able to find an unmatched v vertex, it mates with its first neighbor on its adjacency list. As a consequence, at the start of the algorithm proper all vertices in U are matched and the initial active vertices will be the "overmatched" vertices in V.

Even though the algorithm works as described, it has been found that a periodic global relabeling speeds up the implementation enormously. This observation was made in computational studies of the push-relabel algorithm for the maximum flow problem [DM89, AS93]. This global relabeling is a backwards breadth-first search performed on the residual graph (i.e. the directed graph implied by the current matchings), changing labels on vertices from approximate distances into exact distances. It is called every $m/2.5$ discharges, a discharge being the operation of taking an active vertex from the queue and trying to push from it. This frequency has a big impact on the implementation's performance, and the value used was determined empirically.

2.6 ABMP

The ABMP algorithm can be thought of as a cross between the simple-search and push-relabel algorithms, in the following sense: augmenting paths are sought, but distance labels on the vertices are used to determine the directions in which paths are extended. The distance labels play essentially the same role as they do in the push-relabel algorithm, and are therefore approximate distances to the exposed vertices in V. A relabeling takes place whenever a path cannot be extended. A global relabeling routine, very similar to the one used in the push-relabel implementation, is invoked periodically to relabel all vertices. The routine is called after every n relabels; this frequency value was also determined empirically.

As originally proposed [ABMP91], the algorithm processes vertices up to a certain distance, then finds the remaining augmenting paths using Hopcroft-Karp's algorithm. In our implementation we simply let the algorithm process all distances (up to n, since u vertices further than that cannot be matched). 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 (with labels below n) in increasing order of their labels.

2.7 Operation counts

We compared the implementations using two measures: CPU time and operation counts. Since the algorithms are different it is not clear how an operation count of one can be compared to another. In the case of the DFS, BFS, and ABMP implementations the operations we counted were of two types: *edge queries* and *edge switches*. An edge query is counted whenever we look at a neighbor of a vertex, be it from U or from V. An edge switch is counted whenever we change an edge from matched to unmatched or vice-versa. In the case of the push-relabel implementation we also counted edge queries, plus the number of pushes. A push is an operation very similar to an edge switch.

Note that by using operation counts as defined above we are underestimating the total number of operations performed by the ABMP and push-relabel implementations (since they are more complex). Thus a simple-search implementation having the same operation count as one of the others will in general be faster. With this caveat in mind we believe the operation count does give us a good measure to compare all algorithms. In addition, as will be seen, there is a very good correlation between operation counts and CPU time.

3 Setting for sequential experiments

3.1 Machine and compilers

The sequential experiments were done on a Sun SPARCstation 2, running SUNOS 4.1.3 with 32 MB of main memory.

The DFS implementation was Chang and McCormick's own FORTRAN-77 code, compiled with Sun's `f77` compiler using the -O option. All others were written in C by the author and compiled with Sun's `cc` compiler using the -O option.

3.2 Input graphs

The input graphs used to test the programs came from three different graph generators. By varying the generators' input parameters a total of 11 different classes of input graphs were obtained.

3.2.1 Generator 1

The first generator was written by the author and generates variations of random bipartite graphs. We use the following definition of a random bipartite graph: vertices from the U partition are considered in some arbitrary order. For each u vertex, x vertices from the V partition are chosen randomly and uniformly to be its neighbors. The variable x itself is a binomial random variable, such that the expected degree of each u vertex is d. The generator accepts as input the value of d and approximates the binomial random variable by simulating a Poisson random variable [Fel68].

Three input classes were obtained with this generator. All of them have $|U| = |V|$ and $d = 5$, and are thus described (with the tags we use to designate them in parenthesis):

- random (random). The neighbors for each vertex in U are chosen from all vertices in V.
- few groups (fewg). 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 $i-1$, i, and $i+1$ in V. There is wrap-around, in that for group 1 we use groups n_1, 1 and 2 in the choice of neighbors, and for group n_1 we use groups $n_1 - 1$, n_1 and 1. The value for n_1 is fixed at 32, and only n_2 varies as we increase the total number of vertices ($n = 2n_1n_2$).
- many groups (manyg). Similar to the previous, but n_1 is fixed at 256.

The classes fewg and manyg 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 original graph is planar or nearly planar each vertex will only have as neighbors vertices in the surrounding area.

The programs were tested on instances having 2^i vertices, where $i = \{14, 15, 16, 17\}$, for each class. Given that the value selected for the average degree was 5 the number of edges was always approximately $2.5n$. In each size, 20 instances were solved, using different seeds for the pseudo-random number generator (which was UNIX's `random()`). After building each instance the vertices on the U side are relabeled from 1 to $n/2$ by a random permutation.

For these classes, we observed in the experiments that the initial matching matched around 86% of the vertices and maximum matchings got to about 99% of the vertices.

3.2.2 Generator 2

This generator was written by J. Stolfi [personal communication]. It can generate six different kinds of bipartite graphs, identified by the tags band, fuzz, hexa, worm, rope, and zipf. The class descriptions, as provided by Stolfi, are as follows.

The graphs generated are all bipartite, with vertices $U[1 \ldots n_u]$ and $V[1 \ldots n_v]$. Being m the number of edges, $d_u = m/n_u$ and $d_v = m/n_v$ are the average degrees on each side. In all the graphs we generated we had $n_u = n_v = n_{1/2}$, and the average degree $d_u = d_v = d$ was 6.

These graphs are partially random, so each quadruple (class,n_u, n_v, m) actually defines a large number of non-isomorphic instances with similar structure. The random choices are a function of a user-specified integer seed. The quintuple (class,n_u, n_v, m,seed) does define a unique graph. For each of the classes the programs were tested on instances having 2^i vertices, where $i = \{12, 14, 16\}$. The number of edges was $3n$. In each size, 5 instances were solved, using different seeds for the pseudo-random number generator (which was a library routine of the MODULA-3 language).

After building each graph, as described below, the vertices on each side are relabeled from 1 to n_v by two random and independent permutations. The relabeling also depends on the user-given seed.

The class descriptions below consider only the "normal" case, where $n_u = n_v$ and the number of edges m is compatible with the class's structure — as it was in all of our tests. When m is too small or too large, some of the "required" edges may be missing, or extra "filler" edges may be present where none should.

band: Vertex $U[i]$ is always connected to vertex $V[i]$, and possibly also to $V[i+k]$ and/or $V[i-k]$, for small integers k. The probability of these extra edges decreases roughly linearly from 1.0 at $k = 0$ to 0.0 at $k \approx d$. The trivial pairing $U[i]$–$V[i]$ is thus a perfect matching. Other matchings may exist. For this class we observed in the experiments that the initial greedy matching paired about 93% of the vertices.

fuzz: About half of the vertices in U and V form a band-type graph, the "kernel". Each of the remaining vertices (the "fuzz") is connected to a distinct "kernel" vertex on the opposite side. The number of edges in the kernel is $m - n_{1/2}$, thus the average degree there is almost $2d$. The only perfect matching connects each fuzz vertex to the corresponding kernel vertex. A greedy maximal matching algorithm, using random labels, will tend to pair kernel vertices with kernel vertices, and stop well below the maximum. We observed in the experiments that the initial matching paired about 78% of the vertices. Augmenting paths are probably very short, at least for $d \approx 6$.

hexa: The vertices on each side are divided into $n_{1/2}/b$ blocks of size b. One random bipartite hexagon is added between each block i on one side and each of the blocks $i+k$ on the other side with $|k| \leq K$, for some K. The parameters b and K are chosen by the program in such a way that the average degree is correct (i.e., $3K/b = d$) but few pairs of hexagons will have more than one vertex in common. The adjacency matrix is divided into blocks of size $b \times b$. On each row, only the $2K+1$ blocks closest to the diagonal will be non-empty. Each non-empty block contains 6 nonzero entries, two on each of 3 distinct rows and 3 distinct columns.

For this class we oberved in the experiments that the initial matching paired about 83% of the vertices and maximum matchings paired about 95% of the vertices.

worm: The vertices one each side are grouped into t blocks of size $b = n_{1/2}/t$, numbered $U_0 \ldots U_{t-1}$ and $V_0 \ldots V_{t-1}$. Typically t is small, between 3 and 5. Block i on one side is connected to block $i+1$ on the other side, for $i = 0, 1, \ldots, t-2$; and block U_{t-1} is connected to block V_{t+1}. (Thus, the graph is a fat "worm" that is folded and twisted over itself, so that it zigzags between the two sides, first up and then down.) The connections between blocks alternate between perfect matchings ("m-type connections") and random bipartite graphs of average degree $d-1$ ("r-type connections"). The first and last connections are perfect matchings. The adjacency matrix is divided into t^2 blocks of size $b \times b$. All blocks are empty, except for the $2(t-1)$ blocks immediately adjacent to the diagonal, and the

last block on the diagonal itself. Each non-empty block contains either b ones on the block diagonal, or $(d-1)b$ ones randomly distributed.

The only perfect matching is the union of all m-type connections. A greedy match would tend to use r-type edges, since they are more numerous, and hence fall short of the maximum. For this class we oberved in the experiments that the initial matching paired between 80% and 84% of the vertices. Augmenting paths should have about $2t-1$ edges, and augmenting trees should have about $(d-1)^{t-1}$ vertices.

rope: This is a longer version of class worm. Block size is equal to d so there are n/d blocks. For this class we oberved in the experiments that the initial matching paired about 90% of the vertices.

zipf: This is a random bipartite graph where the edge between u_i and v_j has "ideal" probability roughly proportional to $1/(ij)$. Thus it is very dense near the "core" vertices u_0, v_0, and thins out slowly towards the "periphery". If m is large compared to $n_u n_v$, the "ideal" distribution above gives probabilities greater than 1 for some edges. Since parallel edges are not allowed, the algorithm implicitly reduces the actual probabilities of core edges to the range $[0 \ldots 1]$.

The maximum matching is probably quite low. For one thing, there must be many vertices of degree 0. Moreover, there must be many vertices of low degree that are attached to the same core vertices. In the experiments we observed that the initial matching paired between 62 and 69% of the vertices, while the maximum paired about 5% more vertices.

3.2.3 Generator 3

This generator was written by S. Frank Chang [personal communication]. It generates 0-1 matrices, and the nonzeros in each matrix are generated by blocks of rows. For each block of rows a range of columns is chosen by a random scheme based on input parameters. Another input parameter is the density of nonzeros. We obtained two input classes with this generator: with density 15% (called bcm15) and 26% (called bcm26). Both values generate graphs far denser than those from the other classes. We observed that with 15% density the initial matching paired from 61 to 68% of the vertices, and the maximum matching paired from 72 to 83% of the vertices. With 26% density the initial matching paired about 76% of the vertices and the maximum matching was perfect or nearly so.

We generated graphs with 2^i vertices, $i \in \{8, 9, 10\}$. In each size, 10 instances were generated using different seeds for the pseudo-random number generator (which was UNIX's `random()`).

3.3 Other information

Further characteristics of the experiments were as follows:

- At the end of each run the solution is checked for consistency and maximality in the C implementations.

- Running times for all implementations were measured with the system call `getrusage` by selecting field `ru_utime` (CPU time).
- Running times reported exclude input, checking, and output time, but do include the initial matching time. In addition, the figures reported are means over the number of instances solved in each size and class (the same applies to operation counts).
- Asymptotic performance (growth rate) was estimated by doing a power regression analysis of the data, for both time and operation counts. In the tables below we present this in the column indexed by k, which is the exponent of n given by the analysis. The value of k is somewhat uncertain due to the small number of data points in the regression analysis, and to the variance of the mean running time or mean operation count. More importantly, these growth rates do not necessarily reflect the complexity of the respective algorithms, since graph structure may change with increased size. Thus, as the graphs grow, they may become harder or easier for an implementation. Nevertheless the computed growth rates for each input class do give an useful (albeit rough) relative indication of how fast the solution time (or operation count) is increasing as the instances get larger.

4 Results and analysis of sequential experiments

We present the results below, separated by classes according to the generator they came from. Before that, we have some general remarks.

We observed a good correlation between running times and operation counts. Moreover, the growth rates for running times were in most cases within 10% of the growth rates for operation counts. These observations give us confidence both in the absolute running time values observed and in the performance of one implementation relative to the others.

4.1 Generator 1

In tables 1, 2, and 3 we report the running times, operation counts, and respective growth rates for classes random, fewg, and manyg, respectively.

These tables show that the ABMP, push-relabel and BFS implementations have similar performance, while DFS was relatively quite slow, being about 30 times slower than ABMP in classes random and fewg, for the largest size tested. BFS, although much faster than DFS, exhibited growth rates significantly larger than ABMP and push-relabel.

4.2 Generator 2

Tables 4 through 9 present the results for classes obtained from generator 2. In four of these classes (band, fuzz, rope, and zipf) the simple-search algorithms had similar performance and that was better than the performance of the other two algorithms. In class zipf, DFS was 5 times faster than ABMP for instances with $n = 2^{16}$. Note that maximum matchings in class zipf are far from perfect, thus in agreement with the expectation that DFS and BFS should have good performance in these cases. But note also that ABMP is

Table 1: Results for class random.

$n \rightarrow$	2^{14}	2^{15}	2^{16}	2^{17}	k
time (secs)					
ABMP	0.48	1.05	2.41	4.80	1.12
BFS	0.52	1.44	4.09	9.97	1.43
DFS	3.04	11.16	40.77	142.52	1.85
push-relabel	0.64	1.37	3.11	6.49	1.12
operation counts (thousands)					
ABMP	170	363	841	1678	1.11
BFS	255	677	1831	4289	1.37
DFS	1492	5143	18088	61860	1.79
push-relabel	311	672	1507	3100	1.11

Table 2: Results for class fewg.

$n \rightarrow$	2^{14}	2^{15}	2^{16}	2^{17}	k
time (secs)					
ABMP	0.50	1.13	2.41	4.90	1.10
BFS	0.50	1.52	3.95	10.13	1.44
DFS	2.70	10.54	37.91	141.10	1.90
push-relabel	0.61	1.33	3.09	6.75	1.16
operation counts (thousands)					
ABMP	190	417	868	1738	1.06
BFS	272	776	1889	4555	1.35
DFS	1384	5022	17011	61461	1.82
push-relabel	306	649	1496	3251	1.14

Table 3: Results for class manyg.

$n \rightarrow$	2^{14}	2^{15}	2^{16}	2^{17}	k
time (secs)					
ABMP	0.83	1.74	3.77	8.77	1.13
BFS	0.48	1.30	3.93	13.21	1.59
DFS	0.89	3.45	15.57	70.89	2.11
push-relabel	0.95	1.78	3.93	8.13	1.04
operation counts (thousands)					
ABMP	342	699	1481	3387	1.10
BFS	312	797	2252	7059	1.50
DFS	511	1812	7453	32019	1.99
push-relabel	519	928	2031	4089	1.01

Table 4: Results for class band.

$n \rightarrow$	2^{12}	2^{14}	2^{16}	k
time (secs)				
ABMP	0.11	0.42	1.90	1.03
BFS	0.03	0.15	0.59	1.07
DFS	0.03	0.18	0.87	1.21
push-relabel	0.13	0.65	2.45	1.06
operation counts (thousands)				
ABMP	46	161	709	0.99
BFS	21	84	287	0.94
DFS	14	100	439	1.24
push-relabel	75	319	1136	0.98

Table 5: Results for class fuzz. Running times for BFS and DFS with $n = 2^{12}$ were too small to obtain reliable values for k.

$n \rightarrow$	2^{12}	2^{14}	2^{16}	k
time (secs)				
ABMP	0.04	0.18	0.73	1.05
BFS	0.01	0.07	0.31	≈ 1
DFS	0.00	0.04	0.22	≈ 1
push-relabel	0.05	0.21	0.85	1.02
operation counts (thousands)				
ABMP	12	47	188	0.99
BFS	4	17	70	1.03
DFS	3	13	52	1.03
push-relabel	16	62	249	0.99

Table 6: Results for class hexa.

$n \rightarrow$	2^{12}	2^{14}	2^{16}	k
time (secs)				
ABMP	0.10	0.46	2.09	1.10
BFS	0.07	0.35	2.30	1.26
DFS	0.13	1.71	21.10	1.84
push-relabel	0.16	0.63	2.91	1.05
operation counts (thousands)				
ABMP	39	165	731	1.06
BFS	37	178	1033	1.20
DFS	76	870	9623	1.75
push-relabel	85	322	1385	1.01

Table 7: Results for class worm.

$n \rightarrow$	2^{12}	2^{14}	2^{16}	k
time (secs)				
ABMP	0.09	0.47	2.01	1.12
BFS	0.07	0.51	6.55	1.64
DFS	0.51	5.41	262.80	2.25
push-relabel	0.12	0.60	2.65	1.12
operation counts (thousands)				
ABMP	34	164	690	1.09
BFS	42	286	3159	1.56
DFS	295	2978	124738	2.18
push-relabel	56	264	1139	1.09

Table 8: Results for class rope.

$n \rightarrow$	2^{12}	2^{14}	2^{16}	k
time (secs)				
ABMP	0.23	0.98	4.09	1.04
BFS	0.14	0.60	2.86	1.09
DFS	0.12	0.63	3.08	1.17
push-relabel	0.26	1.22	4.87	1.06
operation counts (thousands)				
ABMP	96	383	1528	1.00
BFS	103	409	1835	1.04
DFS	86	398	1878	1.11
push-relabel	162	682	2596	1.00

Table 9: Results for class zipf.

$n \rightarrow$	2^{12}	2^{14}	2^{16}	k
time (secs)				
ABMP	0.08	0.30	1.29	1.00
BFS	0.02	0.07	0.28	0.95
DFS	0.02	0.06	0.25	0.91
push-relabel	0.17	0.68	2.50	0.97
operation counts (thousands)				
ABMP	27	103	464	1.03
BFS	9	28	98	0.86
DFS	8	25	88	0.86
push-relabel	116	463	1551	0.94

still solving those problems in little more than one second. In terms of growth rates all implementations are similar.

In classes hexa and worm we have again the same pattern detected in generator 1 classes: DFS has dismal performance, and BFS is much faster. However BFS still lags behind ABMP and push-relabel in terms of growth rate; thus ABMP comes out faster in the largest instances. Note that both hexa and worm have a block structure, as do fewg and manyg; in all these classes DFS had poor performance. But DFS was also slow in class random, which has no such structure.

The growth rates observed in these experiments also show that inputs can become harder or easier with increased size. So in class worm the running time growth rate of DFS (2.25) is greater than its worst-case complexity (2). In class zipf three of the implementations have sublinear growth rates, below the linear lower bound.

4.3 Generator 3

In tables 10 and 11 we present results from classes obtained with generator 3. In these classes running time depends primarily on m, since it grows faster than n when instances get larger. Therefore the growth rates shown were computed with respect to the average m shown in the tables.

BFS was fastest in both classes; DFS was fast in class bcm15 but it lagged behind both BFS and ABMP in class bcm26. As noted above, class bcm15 has maximum matchings far from perfect, whereas bcm26's maximum matchings are perfect or nearly perfect. So difference in performance here may indicate that DFS is more sensitive to the maximum matching cardinality than BFS, even though both programs use the same heuristic to discard labeled nodes (see section 2.3).

It is also noteworthy the poor performance of push-relabel. The main reason is that for these classes the implementations was very sensitive to the frequency of global relabeling.

Table 10: Results for class bcm15. Running times for BFS and DFS with $n = 2^8$ were too small to obtain reliable values for k.

$n \rightarrow$	2^8	2^9	2^{10}	
avg. $m \rightarrow$	9830	39322	157286	k
time (secs)				
ABMP	0.02	0.11	0.53	1.18
BFS	0.00	0.02	0.07	≈ 1
DFS	0.00	0.02	0.09	≈ 1
push-relabel	0.04	0.27	1.15	1.21
operation counts (thousands)				
ABMP	20	93	300	0.98
BFS	9	32	96	0.85
DFS	12	61	271	1.12
push-relabel	63	278	909	0.96

Table 11: Results for class bcm26.

$n \rightarrow$	2^8	2^9	2^{10}	
avg. $m \rightarrow$	17039	68157	272630	k
time (secs)				
ABMP	0.12	0.39	1.58	0.93
BFS	0.08	0.30	1.37	1.02
DFS	0.09	0.60	4.23	1.39
push-relabel	0.15	1.35	10.38	1.53
operation counts (thousands)				
ABMP	59	201	823	0.95
BFS	65	255	1201	1.05
DFS	109	796	5488	1.41
push-relabel	117	1127	8648	1.55

Table 12: Overall results of sequential experiments. Running time in seconds, all classes with $n = 2^{16}, 2.5n \leq m \leq 3n$. In boldface is the fastest time for each class.

class	ABMP	BFS	DFS	push-relabel
band	1.90	**0.59**	0.87	2.45
fewg	**2.41**	3.95	37.91	3.09
fuzz	0.73	0.31	**0.22**	0.85
hexa	**2.09**	2.30	21.10	2.91
manyg	**3.77**	3.93	15.57	3.93
random	**2.41**	4.09	40.77	3.11
rope	4.09	**2.86**	3.08	4.87
worm	**2.01**	6.55	262.80	2.65
zipf	1.29	0.28	**0.25**	2.50

The frequency adopted was good in some instances but bad in others, leading to quite different running times. This illustrates a weakness of push-relabel: when graphs are dense, it becomes difficult to obtain a global relabeling frequency that is good in all cases. The ABMP implementation also depends on global relabeling, but no such weaknesses showed up in these experiments.

5 Overall conclusion of sequential experiments

In Table 12 we summarize results for 9 of the input classes used (those classes where there were instances of approximately the same size). The one clear conclusion we can draw from this table is that DFS can have very poor performance compared to the others, and in particular compared to its counterpart BFS. Since the codes of these two implementations are similar in size and complexity, these results show that BFS should always be preferred over DFS. This result is consistent with the behaviour of DFS and BFS in maximum flow problems, as shown in [AS93]. We can also say that ABMP and push-relabel have similar performance. ABMP was fastest in most cases, but the running times are sufficiently close that we cannot be positive in choosing one over the other.

A comparison between ABMP and BFS is more difficult. While ABMP "won" in more classes than did DFS, we don't claim that the set of classes used is indeed representative of problems habitually solved in practice. We can nevertheless say that ABMP had in general smaller growth rates.

Taking together all these observations and those made in previous sections, the experimental results of this paper point towards the following recommendations for bipartite matching practitioners (with the following two assumptions: input graphs should be sparse and an initial greedy matching is used):

- In situations where small problems (up to thousands of vertices) have to be solved, either once or many times as sub-problems of other problems, BFS is the best choice.

- In situations where large problems (tens of thousands of vertices or larger) have to be solved, the ABMP or the push-relabel algorithms can be the best choice, depending on the input graph structure.
- When it is known beforehand that the maximum matching is well below perfect, BFS seems a better choice, even when the problem is large.

6 Parallel Implementation

In this section we describe the parallel implementation developed for the push-relabel algorithm. Parallel bipartite matching has attracted quite a bit of attention from the theoretical point of view [Gro92, GPST92, GPV88, GT88a, SM89], but the algorithms described do not seem suitable for implementation.

In the push-relabel algorithm, as seen in section 2.5, we have a collection of active vertices to which we apply the push and relabel operations. The algorithm is correct no matter the order in which these operations are applied, as long as they are applicable. This means that we can push from or relabel many active vertices simultaneously, making the algorithm suitable for a parallel, coarse-grained implementation.

Our implementation is geared towards shared-memory, few-processor machines. It is an adaptation of an implementation developed by Anderson and Setubal [AS95] for the maximum flow problem. Since the adaptation is relatively straightforward, here we limit ourselves to a description of its highlights.

There is a shared data structure (a *global queue*) that contains active vertices, and processors get the vertices from it and apply the push and/or relabel operations to them. Push operations may activate other vertices, which are then placed in the global queue. The performance of the implementation of course depends crucially on the number of active vertices available and how they are handed to the processors. Therefore, an adaptive scheme exists by which processors get a variable-sized set of active vertices from the global queue each time they access it. The size of this set decreases or increases with the decrease or increase in the total number of available active vertices throughout the execution.

Another critical aspect of the implementation is the way global relabeling is applied. It must be executed concurrently with push/relabel operations, and this in general makes the algorithm incorrect. A simple modification of global relabeling to allow concurrency was introduced in [AS95], and this same modification is used here.

7 Setting for the parallel experiments

The parallel experiments were conducted on a Sequent Symmetry S81 with 20 Intel 16Mhz 80386 processors, and 32 MB 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 parallel experiments were not as extensive as the sequential ones. They were conducted with the main goal of verifying whether some speed-up could be achieved or not.

The input classes used were obtained from a preliminary version of generator 1 (described in section 3.2.1). In these classes d was chosen as 4, and their description is as follows:

- random2. The neighbors for each vertex in U are chosen at random from all vertices in V.
- fewg2. 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 expected edges per vertex.
- manyg2. Similar to the previous, but n_1 is fixed at 500.

As can be seen, these classes are very similar to the corresponding ones used in the sequential experiments. The instance sizes used were $n \in \{30000, 60000, 120000\}$.

Ten instances per class and size were solved, and 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. Finally, each instance was solved using 1, 2, 4, 8, and 12 processors.

8 Results and analysis of parallel experiments

Tables 13, 14, and 15 present the results for the parallel implementation on classes random2, fewg2, and manyg2, respectively. Figure 1 is a plot of the running times for class fewg2; the other classes have similar plots.

Initially note that the parallel implementation with one processor differs from the sequential implementation in one important respect (besides the use of locks): global relabeling in the parallel implementation happens concurrently with push/relabel operations. This causes the observed increase in the number of operations from column *seq* to column $p = 1$ in the tables. Regarding the speed-up figures reported, note that each 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 speed-ups larger than one were 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 processors the decrease in running time over 8 processors is quite small, and in some cases there was actually an increase. We also note that an increase in the number of processors also in general causes more total work to be done. This can be seen in the increase in the total number of operations performed, given by lines *disch* and *relab* on the tables. (The count *disch* reported on the tables refers to the action of getting an active vertex and pushing from it until it becomes inactive.) In class manyg2 there was a slight decrease in the total number of operations when the number of processors went from one to two. As mentioned previously, the frequency of global relabeling has a big impact in the

Table 13: Results for class random2. Time is in seconds; p is number of processors, and *seq* labels the column corresponding to the sequential program. Line *disch* shows mean number of discharges; line *relab* shows mean number of relabel operations.

$n = 30000$; 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 14: Results for class fewg2. Time is in seconds; p is number of processors, and *seq* labels the column corresponding to the sequential program. Line *disch* shows mean number of discharges; line *relab* shows mean number of relabel operations.

$n = 30000$; 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 15: Results for class manyg2. Time is in seconds; p is number of processors, and *seq* labels the column corresponding to the sequential program. Line *disch* shows mean number of discharges; line *relab* shows 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

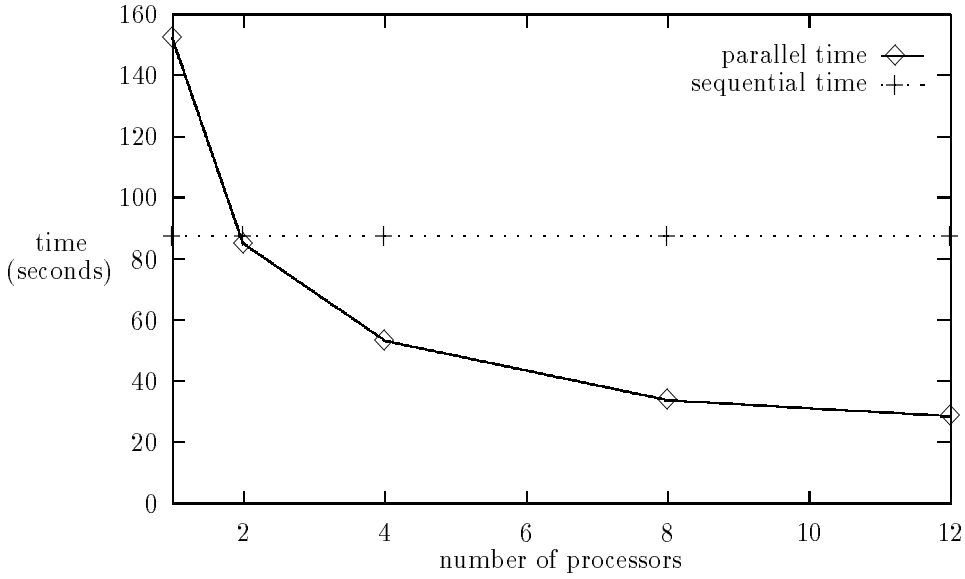

Figure 1: Plot of running times for class fewg2 on 120,000-vertex instances.

number of operations performed, and we believe that in this particular class two processors apply global relabeling with a frequency that is more efficient than the one applied by one processor.

Similarly to the maximum flow study [AS95], 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.

9 Conclusions, comments, and further work

Conclusions from the sequential experiments were already given in section 5. Here we comment on the relevance of the parallel results and outline some aspects where further work would be interesting.

As shown in section 8 the push-relabel algorithm can run up to three times faster than its sequential counterpart. If such a speed-up were attained for all classes and instances shown in table 12 the push-relabel parallel implementation would have been the fastest in six out of nine classes. This shows that for very large problems where the fastest possible running time is necessary a parallel implementation of the push-relabel algorithm should be considered.

We believe that the ABMP and push-relabel implementations developed for these experiments can be further improved. The push-relabel algorithm, in particular, has many variants, of which only one was used. One attractive variation is highest-label-first, where vertices are processed not in queue ordering but according to their distance labels. In the case of the maximum flow problem this variation is sometimes faster [AS93]. The flexibility of these algorithms also allow for the introduction of many heuristics. We used one of them, but there may be others. In particular it may be possible to better tune the global relabeling heuristic to obtain more efficient results.

Another aspect that should be studied in greater depth is the initial matching. We used a simple greedy strategy, but other methods can be used. We would like to know what the influence of this initial matching is on an implementation's performance. Simple experiments that we carried out showed that it is not necessarily the case that larger maximal matchings translate into faster overall running times.

In terms of other experiments, it would be interesting to test the implementations presented in this paper on other classes of bipartite graphs; and investigate whether it is

possible to obtain better parallel speed-ups, either with the push-relabel algorithm or with some other algorithm.

10 Acknowledgments

I gratefully acknowledge the contribution of Jorge Stolfi, who helped expand the range of input classes used; he also made many comments that improved this work. S. Frank Chang was very kind and prompt in sending me both his DFS implementation and his generator. Tom McCormick sent me his paper and made helpful comments on bipartite matching implementations. Ken Darby-Dowman kindly sent me the relevant chapter of his thesis. My student C. Castrequini helped with some preliminary experiments. Two anonymous referees made many important remarks, and in particular pointed out Darby-Dowman's thesis, without which I would never have considered the BFS implementation.

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.

[AMO93] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Englewood Cliffs, N. J., 1993.

[AS93] 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, *First* DIMACS *Implementation Challenge – Network Flows and Matching*, volume 12 of DIMACS *Series in Mathematics and Theoretical Computer Science*, pages 1–18. American Mathematical Society, 1993.

[AS95] R. J. Anderson and J. C. Setubal. Parallel implmentation of the push-relabel algorithm for maximum flow. *Journal of Parallel and Distributed Computing*, 1995. Accepted for publication.

[CM90] S. Frank Chang and S. Thomas McCormick. A faster implementation of a bipartite cardinality matching algorithm. Technical report, Faculty of Commerce and Business Administration, University of British Columbia, 1990. Working Paper 90-MSC-005.

[DD80] Kenneth Darby-Dowman. *The exploitation of sparsity in large scale linear programming problems — Data structures and restructuring algorithms*. PhD thesis, Brunel University, 1980.

[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.

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

[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. Using interior-point methods for fast parallel algorithms for bipartite matching and related problems. *SIAM J. Comput.*, 21(1):140–150, 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.

[Set93] J. C. Setubal. New experimental results for bipartite matching. In *Proceedings of* NETFLOW93, *tech. report TR-21/93, Dipartamento de Informatica, Universitá di Pisa*, pages 211–216, 1993.

[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 – 1995

95-01 **Paradigmas de algoritmos na solução de problemas de busca multidimensional,** *Pedro J. de Rezende, Renato Fileto*

95-02 **Adaptive enumeration of implicit surfaces with affine arithmetic,** *Luiz Henrique de Figueiredo, Jorge Stolfi*

95-03 **W3 no Ensino de Graduação?,** *Hans Liesenberg*

95-04 **A greedy method for edge-colouring odd maximum degree doubly chordal graphs,** *Celina M. H. de Figueiredo, João Meidanis, Célia Picinin de Mello*

95-05 **Protocols for Maintaining Consistency of Replicated Data,** *Ricardo Anido, N. C. Mendonça*

95-06 **Guaranteeing Full Fault Coverage for UIO-Based Methods,** *Ricardo Anido and Ana Cavalli*

95-07 **Xchart-Based Complex Dialogue Development,** *Fábio Nogueira de Lucena, Hans K.E. Liesenberg*

95-08 **A Direct Manipulation User Interface for Querying Geographic Databases,** *Juliano Lopes de Oliveira, Claudia Bauzer Medeiros*

95-09 **Bases for the Matching Lattice of Matching Covered Graphs,** *Cláudio L. Lucchesi, Marcelo H. Carvalho*

95-10 **A Highly Reconfigurable Neighborhood Image Processor based on Functional Programming,** *Neucimar J. Leite, Marcelo A. de Barros*

95-11 **Processador de Vizinhança para Filtragem Morfológica,** *Ilka Marinho Barros, Roberto de Alencar Lotufo, Neucimar Jerônimo Leite*

95-12 **Modelos Computacionais para Processamento Digital de Imagens em Arquiteturas Paralelas,** *Neucimar Jerônimo Leite*

95-13 **Modelos de Computação Paralela e Projeto de Algoritmos,** *Ronaldo Parente de Menezes e João Carlos Setubal*

95-14 **Vertex Splitting and Tension-Free Layout,** *P. Eades, C. F. X. de Mendonça N.*

95-15 **NP-Hardness Results for Tension-Free Layout,** *C. F. X. de Mendonça N., P. Eades, C. L. Lucchesi, J. Meidanis*

95-16 **Agentes Replicantes e Algoritmos de Eco,** *Marcos J. C. Euzébio*

95-17 **Anais da II Oficina Nacional em Problemas Combinatórios: Teoria, Algoritmos e Aplicações,** *Editores: Marcus Vinicius S. Poggi de Aragão, Cid Carvalho de Souza*

95-18 **Asynchronous Teams: A Multi-Algorithm Approach for Solving Combinatorial Multiobjective Optimization Problems,** *Rosiane de Freitas Rodrigues, Pedro Sérgio de Souza*

95-19 **wxWindows: Uma Introdução,** *Carlos Neves Júnior, Tallys Hoover Yunes, Fábio Nogueira de Lucena, Hans Kurt E. Liesenberg*

95-20 **John von Neumann: Suas Contribuições à Computação,** *Tomasz Kowaltowski*

95-21 **A Linear Time Algorithm for Binary Phylogeny using PQ-Trees,** *J. Meidanis and E. G. Munuera*

95-22 **Text Structure Aiming at Machine Translation,** *Horacio Saggion and Ariadne Carvalho*

95-23 **Cálculo de la Estructura de un Texto en un Sistema de Procesamiento de Lenguaje Natural,** *Horacio Saggion and Ariadne Carvalho*

95-24 **ATIFS: Um Ambiente de Testes baseado em Inje,c ao de Falhas por Software,** *Eliane Martins*

95-25 **Multiware Plataform: Some Issues About the Middleware Layer,** *Edmundo Roberto Mauro Madeira*

95-26 **WorkFlow Systems: a few definitions and a few suggestions,** *Paulo Barthelmess and Jacques Wainer*

95-27 **Workflow Modeling,** *Paulo Barthelmess and Jacques Wainer*

Relatórios Técnicos – 1996

96-01 **Construção de Interfaces Homem-Computador: Uma Proposta Revisada de Disciplina de Graduação,** *F'abio Nogueira Lucena and Hans K.E. Liesenberg*

96Abs **DCC-IMECC-UNICAMP Technical Reports 1992–1996 Abstracts,** *C. L. Lucchesi and P. J. de Rezende and J.Stolfi*

96-02 **Automatic visualization of two-dimensional cellular complexes,** *Rober Marcone Rosi and Jorge Stolfi*

Instituto de Computação
Universidade Estadual de Campinas
13081-970 – Campinas – SP
BRASIL

reltec@dcc.unicamp.br