Computing maximum-cardinality matchings in sparse general graphs[*]

John Kececioglu[†] Justin Pecqueur[‡]

May 11, 1998

Abstract We give an experimental study of a new $O(mn\,\alpha(m,n))$ time implementation of Edmonds' algorithm for a maximum-cardinality matching in a sparse general graph of n vertices and m edges. The implementation incorporates several optimizations stemming from choosing a depth-first order in which to examine edges during the search for augmenting paths, and we study the iteraction between several heuristics with the potential to speed up the code in practice. From experiments on several classes of graphs, we conclude that the simplest optimization, a stopping-test for the depth-first search for an alternating path, results in the greatest performance gain. The resulting code appears to be the fastest among those publicly available on random, random d-regular, random k-cycle, and k-nearest neighbor graphs with up to 100,000 vertices and 500,000 edges, achieving on the largest graphs a speedup of a factor of 4 to 350 over the LEDA and two of the DIMACS challenge codes.

Keywords Unweighted matchings, nonbipartite graphs, algorithm implementation, experimental analysis of algorithms

1 Introduction

One of the classic problems of combinatorial optimization is maximum-cardinality matching in general graphs. A *matching* of an undirected graph $G = (V, E)$ is a subset of the edges $M \subseteq E$ such that no two edges in M touch a common vertex. A *maximum-cardinality* matching is a matching with the maximum number of edges. Unless otherwise stated, by a maximum matching in this paper we mean a maximum-cardinality matching, and by a graph we mean in general a nonbipartite graph.

Among the fundamental polynomial-time results in combinatorial optimization is Edmonds' algorithm for maximum matching [6]. This paper gives an experimental study of a new implementation of Edmonds' algorithm for large sparse graphs. Our own motivation comes from the problem of large-scale DNA sequence assembly [13] in computational biology, which can be approximated in the presence of error using maximum-weight matchings

[*]Extended abstract submitted to the 2nd *Workshop on Algorithm Engineering* (WAE '98).

[†]Corresponding author. Department of Computer Science, The University of Georgia, Athens, GA 30602-7404, USA. Electronic mail: `kece@cs.uga.edu`. Fax `706-542-2966`. Research supported by a National Science Foundation CAREER Award, Grant DBI-9722339.

[‡]Department of Computer Science, The University of Georgia, Athens, GA 30602-7404, USA. Electronic mail: `andre@cs.uga.edu`.

in sparse nonbipartite graphs [12]. While our ultimate goal for DNA sequence assembly is to implement the more general *weighted* sparse-graph matching algorithm of Galil, Micali and Gabow [10], in attempting to understand this complex algorithm at the level of detail necessary to produce an implementation, we were lead to first studying the simpler problem of cardinality matching, and the present work of implementing an efficient sparse cardinality-matching algorithm.

There have been several implementation studies of cardinality-matching algorithms, notably from the first DIMACS algorithm implementation challenge. Crocker [5] and Mattingly and Ritchey [16] both give software implementations of the $O(n^{1/2}m)$ algorithm of Micali and Vazirani [19] and perform experimental studies of their implementations. Rothberg [22] gives an implementation of Gabow's $O(n^3)$ version [8] of Edmonds' algorithm. LEDA [17], a library of combinatorial and geometric data structures and algorithms designed by Mehlhorn and Näher, also provides an $O(mn\,\alpha(m,n))$ implementation [18] of Edmonds' algorithm. Most recently, Magun [15] investigates several greedy matching heuristics empirically, and Shapira [23] derives worst-case bounds on the quality of an $O(m+n)$ greedy heuristic.

The plan of the paper is as follows. In the next section we sketch Edmonds' algorithm. Section 3 follows with a discussion of our particular implementation, highlighting several optimizations and heuristics that appear to be unique to our implementation. Section 4 presents results from experiments with ours and several other publicly available implementations on several classes of both random and structured graphs, which suggest that the new code is among the fastest available for large sparse graphs. Finally Section 5 closes with some directions we wish to pursue further.

2 Edmonds' algorithm

We now give a very brief sketch of Edmonds' algorithm. The algorithm is rather involved, and to meet space guidelines for submissions, we will defer to readers familiar with its basic ideas. Good expositions may be found in Tarjan [24], Gibbons [11], Ahuja, Magnanti and Orlin [1], and Lawler [14].

The following terms are used throughout. A vertex v is *unmatched* with respect to matching M if no edge of M touches v. An *alternating path* in G with respect to a matching M is a path whose edges alternate between being in and out of M. An *augmenting path* is an alternating path that begins and ends at an unmatched vertex. A *blossom* is an alternating path that forms a cycle of odd length; note that on any such cycle there must be a vertex incident to two unmatched edges on the cycle, which is called the *base* of the blossom.

The essential step of the algorithm is to find an augmenting path with respect to a current matching M, which may initially be taken to be the empty matching. If M has an augmenting path, its cardinality may be increased by one, and if M has no augmenting path, it is optimal [4]. A *phase* of the algorithm consists of searching for an augmenting path, and terminates on finding such a path or determining that there is none.

An augmenting path may be found by exploring along search trees rooted at unmatched vertices. As edges at successive levels of the trees alternate between being out of M and in M, they are called *alternating trees*. Vertices at even depths from the root are called *even*, and vertices at odd depths are called *odd*. During the search a blossom may be discovered, at which point all vertices on the cycle are shrunk into a single *supervertex* that is identified with the base of the blossom. The key observation of Edmonds is that the shrunken graph has an augmenting path if and only if the original graph has one [6].

With suitable data structures, a phase can be implemented to run in $O(m\,\alpha(m,n))$ time, which Tarjan [24] credits to Gabow. As there are at most $n/2$ phases, this gives an $O(mn\,\alpha(m,n))$ time algorithm. This can further be reduced to $O(mn)$ time using the disjoint-set result of Gabow and Tarjan [9].

3 Implementation

The data structures we use to implement Edmonds' algorithm are as follows. Edges in the undirected graph G are assigned an arbitrary orientation, so that each undirected edge is represented by one directed edge that may be traversed in either direction. We access the neighborhood of a vertex by maintaining with each vertex a list of in-edges and out-edges. When detecting a blossom, the graph is not actually shrunk, but the partition of vertices in the original graph into blossoms is maintained via the disjoint-set data structure. We use the disjoint-set path-halving variant of Tarjan and van Leeuwen [24]. To a limited extent we also perform our own memory management by maintaining for each dynamically allocated datatype a *pool* of free objects; when a new object is requested and the corresponding pool is empty, we ask for a whole block of objects of that type with one call to `malloc` and add all the new objects to the pool; this reduces the number of calls to `malloc` and `free`.

Perhaps the most significant choice in the implementation is the order in which to examine unexplored edges of the graph. We chose to examine them in the order of a *depth-first search*, so that we grow alternating trees one at a time, until we discover an augmenting path or that there is none starting from the current root. This choice permits several optimizations, which we detail next. We then follow with four additional heuristics that we considered in the variants we implemented, which are examined experimentally in Section 4.

3.1 Optimizations

Shrinking in one pass A key advantage of examining edges in depth-first search order is that on encountering an edge $e = (v, w)$ between two even-labeled vertices, it is not hard to show that this always forms a blossom. Hence the algorithm avoids performing a walk to determine whether e forms an augmenting path or a cycle. Furthermore, one endpoint of e must be an ancestor of the other in the alternating tree. Thus the algorithm further avoids an interleaved walk to determine the nearest common ancestor of v and w.

To easily identify the ancestor, we assign an *age* to each vertex from a global time counter that is incremented as vertices are reached when growing an alternating tree. The nearest common ancestor of v and v is then the younger of the two, and we can shrink the detected blossom in one pass while walking to the ancestor.

Not expanding unsuccessful trees On returning from a search of an alternating tree that does not lead to an augmenting path, we leave all blossoms in the tree shrunken. It is unnecessary to expand them: the depth-first search is exhaustive, so no future augmentations will ever pass through vertices of the tree.

Identifying search tree roots quickly After completing a successful or unsuccessful search from an alternating tree, the next vertex from which to start a search must be identified.

Rather than scanning the vertices of the graph to identify an unreached unmatched vertex for the root of the next search tree, we maintain one list across all phases of the algorithm of unreached unmatched vertices. When an unmatched vertex is first encountered by a search, we unlink it from this list. Such a vertex can never be added back to the list, since if it is reached by an ultimately unsuccessful search, it will always remain reached, while if it is reached by an ultimately successful search, it will always remain matched. To quickly identify the next search tree root, we simply pop the next vertex from this list.

3.2 Heuristics

We now describe four potential heuristics for speeding up the implementation, three of which have been suggested in the literature, and one which is to our knowledge new.

Initialization with a greedy matching While Edmonds' algorithm is usually described as starting from the empty matching, it can be started with any valid matching. Conventional wisdom says that starting from a near-optimal matching should speed up the algorithm since this reduces the number of augmentations. (In reality though the situation is far from clear, since after performing as many augmentations as there are edges in the initial matching, the algorithm proceeds on a contracted graph, which could conceivably be preferable to working in the original expanded graph.)

We consider starting from an initial matching obtained by the following standard greedy heuristic. Form a maximal matching by repeatedly selecting a vertex v that has minimum degree in the subgraph $\widetilde{G}$ induced by the currently unmatched vertices, and select an edge (v, w) to an unmatched vertex w that has minimum degree in $\widetilde{G}$ over all vertices incident to v. The entire procedure can be implemented to run in $O(m+n)$ time using a discrete bucketed heap of unmatched vertices prioritized by degree in $\widetilde{G}$.

Early termination of depth-first searches In the depth-first search, unexplored edges are pushed onto a search stack, and later popped off as they are explored. Edges are pushed on the stack when first encountering a vertex that gets labeled even, or when shrinking a blossom into an even supervertex. In both situations, when pushing an edge e we can test whether its other end touches an unreached unmatched vertex. If so, edge e completes an augmenting path; further pushing edges onto the stack will simply bury e, and if we instead leave e exposed on the top of the stack, the next iteration of the search will pop e off and immediately discover the augmenting path. This simple *stopping test* has the potential to very quickly halt an ultimately successful search.

Delayed shrinking of blossoms We also consider a suggestion of Applegate and Cook [2] originally made in the context of weighted matchings. Since the algorithm performs work when shrinking and later expanding blossoms, it may be worth postponing the formation of blossoms as long as possible. Before pushing unexplored edges onto the search stack, we can test whether the edge forms a blossom or not. Edges that do not form blossoms are given precedence and put on the stack above edges that create blossoms. We implement this by maintaining two lists, of cycle-forming and non-cycle-forming edges, when scanning the neighborhood of a vertex; after completing the scan these two lists are concatenated onto the search stack in the appropriate order.

Lazy expansion of blossoms As observed by Tarjan [24], on finding an augmenting path P in a successful search of an alternating tree T, the only blossoms we need to immediately expand are those on the augmenting path; blossoms in T that are not on path P can remain shrunken. When a later search encounters a vertex in this prior tree T, it can at that moment be expanded lazily and treated as having been labeled unreached.

We implement this idea as follows. On an unsuccessful search, we delete all vertices in the alternating tree from the graph. At the start of a new search, we record the current time, call it the current *epoch*. Then when a vertex is encountered during a search, we first examine its age. If its age is from a prior epoch, we first lazily expand the blossom, consider its members as having been labeled unreached, and proceed as before. This requires that with each blossom we keep a list of its members, which can be maintained in constant time during blossom contraction and expansion.

4 Experimental results

We now study the performance of several implementations resulting from different combinations of the heuristics described above, together with several publicly available codes from other authors, by conducting computational experiments on both random and structured graphs.

In our experiments we tested the following implementations. Their time complexities are stated in terms of n, the number of vertices in the graph, and m, the number of edges.

(1) An $O(mn\,\alpha(m,n))$ time implementation, written in C by the first author, of the general approach described by Tarjan [24] with the depth-first search optimizations described in Section 3. To decide which of the *heuristics* of Section 3 to incorporate, sixteen variants of this basic implementation were written, comprising all possible combinations of the four heuristics. After performing several computational tests with these sixteen variants as described below, one overall winner with the best combination of heuristics was selected. In the experiments this implementation is called `Kececioglu`.

The implementation is part of a freely-available object-oriented library of fundamental string and graph algorithms being developed by the first author, called DALI, for "a **d**iscrete **al**gorithms **li**brary." DALI currently contains general implementations of several commonly-used data structures, including lists, multidimensional arrays, search trees, hash tables, mergeable heaps, disjoint sets, and undirected and directed graphs, as well as efficient algorithms for shortest paths, minimum spanning trees, maximum flow, minimum cut, maximum weight branchings, and nearest common ancestors. DALI's design emphasizes code reusability, portability, and efficiency. In contrast to many algorithm collections, the implementation presents a uniform, consistent interface; in contrast to many object-oriented libraries, the implementation is lightweight, with low operation overhead and small object-code size. The maximum-cardinality matching code with comments comprises about 1200 lines of C, and is built on top of a general list library of roughly 650 lines, a disjoint set library of roughly 350 lines, and a directed graph library of roughly 1150 lines.

(2) **Stefan Näher's** $O(mn\,\alpha(m,n))$ time implementation of Tarjan's [24] approach, written in C++ as part of the LEDA library [18] of efficient data structures and algorithms

developed by Kurt Mehlhorn and Stefan Näher [17]. The implementation comes with two different heuristics for constructing the greedy matching used to initialize the algorithm. In the experiments the two resulting codes are called `LEDA 1` and `LEDA 2`.

(3) **Steven Crocker's** [5] $O(m\sqrt{n})$ time implementation, written in Pascal, of Micali and Vazirani's algorithm [19]. In the experiments this implementation is called `Crocker`.

(4) **Ed Rothberg's** [22] $O(n^3)$ time implementation, written in C, of Gabow's version [8] of Edmonds' algorithm. In the experiments this implementation is called `Rothberg`.

These codes were compared by the second author across five different classes of graphs.

(1) **Random graphs** with n vertices and m edges. To test the codes on large-scale inputs it is important to generate a random graph in $O(m+n)$ space, in contrast to the $O(n^2)$ space needed by the straightforward random graph generator. To do this we generated a random subset of size m from $\{1, \ldots, \binom{n}{2}\}$ using Floyd's algorithm [3], translating the chosen integers into unordered pairs of vertices by a bijection. Since for large n, $\binom{n}{2}$ can easily exceed the machine representation, we also implemented an arbitrary integer arithmetic package. The resulting generator allowed us to generate large sparse random graphs in $O(n + m \log n)$ time in $O(m+n)$ space.

(2) **Random k-cycle graphs** with n vertices and m edges. For a given n, m, and k, we formed a graph on n vertices by repeated choosing a random subset of k vertices, connecting them with a random cycle, eliminating parallel edges, and taking the union of such cycles until the graph contained at least m edges. In our experiments, we chose $k = 3$, as this produced the hardest instances for the codes.

(3) **Random near-regular graphs** with degree d on n vertices. For a given n and d, form a graph on n vertices as follows. Maintain an array of length n representing a list of vertices and their unused degree capacity. All vertices in the array are initialized to degree capacity d. Then repeatedly pick the leftmost vertex v from the array, delete it, and generate a random subset of size $\min\{d', n'\}$ over the remaining vertices, where d' is the unused degree capacity of v and n' is the number of remaining vertices. An edge is added from v to each vertex in the subset, the degree capacities of the vertices are decremented, and any vertex with capacity zero is deleted from the list.

In contrast to the standard regular-graph generator, the graphs generated by this procedure do not contain self-loops or parallel edges. On the other hand, they are not guaranteed to be d-regular, though there are at most d vertices in the generated graph with degree less than d. The generator can be implemented to run in $O(nd \log d)$ time using only $O(n)$ working storage.

(4) **K-nearest neighbor graphs** on n points in the plane. For a given k, we formed a graph by choosing a point set from Gerhard Reinelt's TSPLIB [21] library of Euclidean traveling salesman test problems, and connecting each point to its k nearest neighbors.

(5) **Gabow's graph** [8] on $2n$ vertices, constructed to elicit worst-case behavior from Edmonds' algorithm. This very dense graph consists of the complete graph K_n together with a perfect matching connecting K_n to another disjoint set of n vertices.

All experiments were run on a dedicated Sun Ultra 1 workstation with 256 Mb of RAM, and a 200 MHz processor. In the experiments on random graphs, each data point represents an average of ten graphs, where care was taken to perform comparisons across codes on the same ten random graphs. Times reported are user times in seconds obtained by the UNIX `time` command under the Bourne shell. The time measured is the amount of time taken to read in the graph, compute a matching, and output its cardinality. All C codes were compiled under `gcc` with the `-O3` optimization.

In the Appendix we plot running times for experiments on the five classes of graphs. In the first set of experiments, shown in Figure 1, we tested the sixteen variants of our implementation on random graphs with 5,000 to 50,000 vertices, and n to $10n$ edges. Figure 1 shows results with $n = 50,000$. The string of letters that labels a curve describes which of the four heuristics the variant contains. The top four plots divide the sixteen variants into four large groups of four variants each, according to whether they start from a greedy initial matching (`G`) or use the stopping test for a depth-first search (`S`). From each of the four groups we selected a variant with the best running time and these four winners are displayed in the bottommost plot, except that we do not include the winner from the initial group not containing the `G` and `S` heuristics, as this group's times are much slower than all other groups. Instead we include two representatives, `G` and `GD`, from the second group.

The most striking feature of Figure 1 is that incorporating just the `G` or `S` heuristic alone into the basic implementation gives a speedup of a factor of 30 to 35 on the largest graphs. The improvement with the `G` variant supports the conventional wisdom that to speed up an exact matching procedure one should start from an initial approximate matching.

The next most striking feature of Figure 1 is that *it takes more time on graphs with high degree to start with a greedy initial matching than to start with an empty initial matching and use the stopping test alone* (which can be seen by comparing the `G` and `S` curves in the bottom plot). It came as a surprise to the authors that the greedy matching heuristic actually slows down the `S` variant (which can be seen by comparing the `S` and `GS` curves in the bottom plot), especially since the greedy heuristic runs in $O(m + n)$ time. To understand this better we compared the `S` variant, which finds an exact matching, to the greedy heuristic alone, which in general finds an approximate matching, and found that the exact `S` code was in fact faster: the greedy heuristic must examine every edge in the graph to compute the degree of each vertex, while with the stopping test the `S` code could avoid looking at every edge on graphs with high degree. Nevertheless, of the sixteen variants, we chose the `GS` variant as the overall winner for further comparison (even though it sometimes loses to the `S` variant) as the `GS` code was the most stable across a wide range of vertex degrees.

(It is also interesting that the lazy expansion (`L`) and delayed shrinking (`D`) heuristics tend to give relatively little benefit, and in some combinations slow down the code.)

Figures 2 and 3 compare this `GS` code to the LEDA, Rothberg, and Crocker codes on random graphs with n vertices and m edges. In the first set of plots n is fixed and m is varying, and in the second set this is reversed. The graphs on the left display all codes together, while the graphs on the right display just the two fastest codes for easier visual comparison. Across these random graphs the consistently fastest code is `Kececioglu` (which also appears to have the smoothest growth in running time) followed by `Crocker`. The difference in speed generally increases the larger the graph (except for `LEDA 2` which improves on larger graphs), with speedups on the largest graphs of as much as 50 versus `LEDA 2`, 350 versus `Rothberg`, and 4 versus `Crocker`.

Table 1 Mean and variance of running time on random graphs on 100,000 vertices. Times are in seconds, and averaged over 10 trials.

	LEDA 1		LEDA 2		Rothberg		Crocker		Kececioglu	
edges	mean	var	mean	var	mean	var	mean	var	mean	var
50,000	436.21	10.92	149.90	7.88	909.07	1.50	22.27	2.35	3.27	0.10
100,000	1334.85	16.11	489.09	12.27	1142.09	313.78	30.29	1.16	19.06	2.39
250,000	1518.15	16.70	183.02	5.87	1583.15	1783.58	42.98	2.08	13.05	2.88
500,000	924.66	10.88	119.83	3.32	1513.85	2142.98	73.59	4.15	18.30	6.14

Table 2 Mean and variance of running times on random d-regular graphs on 100,000 vertices. Times are in seconds, and averaged over 10 trials.

	LEDA 1		LEDA 2		Rothberg		Crocker		Kececioglu	
degree d	mean	var	mean	var	mean	var	mean	var	mean	var
2	1393.91	12.71	795.33	14.83	86.11	1.06	147.36	240.65	4.39	1.09
3	1130.84	11.14	455.39	9.22	436.04	388.71	25.51	1.47	6.40	1.44
5	837.71	5.56	233.75	7.99	443.36	540.58	36.98	0.57	9.46	0.96
10	542.10	7.22	125.35	3.83	570.69	382.52	69.71	1.71	14.01	0.92

Figure 4 plots times for random near-regular graphs, and Figures 5 and 6 plot times for random 3-cycle graphs. The trends observed above continue for both these classes of graphs. The random d-regular graphs appear to be somewhat easier instances than the corresponding random graphs with the same n and m. On the random 3-cycle graphs, it is interesting that the times for the Rothberg and Crocker codes increase as the size of the graphs increase, while the times for the LEDA and Kececioglu codes, which both implement the basic approach outlined by Tarjan [24], after a critical size tend to remain constant or decrease.

Figure 7 compares the codes on structured, nonrandom graphs. The top four plots show k-nearest neighbor graphs with k from 1 to 10 for Euclidean traveling salesman instances from TSPLIB [21] on roughly 4,000, 6,000, 12,000, and 34,000 points. These graphs appear to be somewhat harder than the corresponding random graphs with the same m and n, but the ranking of the codes is essentially unchanged. The bottom plots are for the dense graph designed by Gabow [8] to force Edmonds' algorithm to take $\Omega(n^3)$ time. In this plot the curve for the `LEDA` codes terminates early due to exhausting available memory. Again the ranking of the codes is essentially the same.

Finally Tables 1 and 2 give variances in running times on random and near-regular graphs. As suggested by the prior plots, the variance in running time for the Rothberg code is quite high, while the variance in the Crocker code appears to be generally the lowest, with the Kececioglu code being roughly comparable. The high Rothberg variance is intuitively plausible, as its running time is closely tied to the amount of work performed in scanning sets during shrinks, which is not well-characterized by the number of vertices and edges in the graph.

5 Conclusions

We have described computational experience with a new $O(mn\,\alpha(m,n))$ time implementation of Edmonds' algorithm for maximum-cardinality matching in sparse general graphs, together with a careful study of several heuristics suggested for speeding up the code in practice. The new code appears to be the fastest currently available for large sparse graphs, and across several classes of both random and structured inputs was roughly 4 to 350 times faster on the largest graphs than other published codes. Interestingly the greatest performance gains in the code were obtained not by the more sophisticated heuristics but by incorporating a very simple stopping test into the depth-first search. On very large graphs, the resulting exact-matching code started from an empty matching is even faster than a linear-time approximate-matching heuristic, and hence in contrast to conventional wisdom is actually slowed down when started from the corresponding near-optimal matching.

For the final conference version we wish to perform several additional experiments. To determine whether the running time of the new code is really proportional to the product mn, we need to study its performance across families of graphs with say n varying and $m = cn$, rather than studying slices with n fixed or m fixed. We also wish to identify why the code performs well by instrumenting the software to gather statistics such as the number of times an edge is examined, sizes of alternating trees, lengths of augmenting paths, and the percentage of vertices in dead alternating trees that will not be augmented in subsequent phases.

The new implementation is publicly available and may be accessed at

`http://www.cs.uga.edu/~kece/Research/software.html`

References

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

[2] Applegate, D. and W. Cook. "Solving large-scale matching problems." In *Network Flows and Matching: First DIMACS Implementation Challenge*, D.S. Johnson and C.C. McGeoch, editors, DIMACS Series in Discrete Mathematics and Theoretical Computer Science 12, 557–576, 1993.

[3] Bentley, J. and R. Floyd. "Programming pearls: A sample of brilliance." *Communications of the ACM*, 754–757, September 1987.

[4] Berge, C. "Two theorems in graph theory." *Proceedings of the National Academy of Sciences USA* 43, 842–844, 1957.

[5] Crocker, S.T. "An experimental comparison of two maximum cardinality matching programs." In *Network Flows and Matching: First DIMACS Implementation Challenge*, D.S. Johnson and C.C. McGeoch, editors, DIMACS Series in Discrete Mathematics and Theoretical Computer Science 12, 519–537, 1993.

[6] Edmonds, J. "Paths, trees, and flowers." *Canadian Journal of Mathematics* 17, 449–467, 1965.

[7] Edmonds, J. "Maximum matching and a polyhedron with 0,1-vertices." *Journal of Research of the National Bureau of Standards* 69B, 125–130, 1965.

[8] Gabow, H. "An efficient implementation of Edmonds' algorithm for maximum matchings on graphs." *Journal of the ACM* 23, 221–234, 1976.

[9] Gabow, H.N. and R.E. Tarjan. "A linear-time algorithm for a special case of disjoint set union." *Journal of Computer and System Sciences* 30:2, 209–221, 1985.

[10] Galil, Z., S. Micali and H.N. Gabow. "An $O(EV \log V)$ algorithm for finding a maximal weighted matching in general graphs." *SIAM Journal on Computing* 15, 120–130, 1986.

[11] Gibbons, A. *Algorithmic Graph Theory.* Cambridge University Press, Cambridge, 1985.

[12] Kececioglu, J. *Exact and Approximation Algorithms for DNA Sequence Reconstruction.* PhD dissertation, Technical Report 91-26, Department of Computer Science, The University of Arizona, December 1991.

[13] Kececioglu, J.D. and E.W. Myers. "Combinatorial algorithms for DNA sequence assembly." *Algorithmica* 13:1/2, 7–51, 1995.

[14] Lawler, E.L. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, New York, 1976.

[15] Magun, J. "Greedy matching algorithms: An experimental study." Proceedings of the 1st Workshop on *Algorithm Engineering*, 22–31, 1997.
`http://www.dsi.unive.it/~wae97/proceedings`

[16] Mattingly, R.B. and N.P. Ritchey. "Implementing an $O(\sqrt{N}M)$ cardinality matching algorithm." In *Network Flows and Matching: First DIMACS Implementation Challenge*, D.S. Johnson and C.C. McGeoch, editors, DIMACS Series in Discrete Mathematics and Theoretical Computer Science 12, 539–556, 1993.

[17] Mehlhorn, K. and S. Näher. "LEDA: A platform for combinatorial and geometric computing." *Communications of the ACM* 38:1, 96–102, 1995.

[18] Mehlhorn, K., S. Näher and S. Uhrig. LEDA Release 3.4 module `mc_matching.cc`. Computer software, 1996. `http://www.mpi-sb.mpg.de/LEDA/leda.html`

[19] Micali, S. and V.V. Vazirani. "An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding a maximum matching in general graphs." Procedings of the 21st Annual IEEE *Symposium on the Foundations of Computer Science*, 17–27, 1980.

[20] Möhring, R.H. and M. Müller-Hannemann. "Cardinality matching: Heuristic search for augmenting paths." Technical Report 439, Fachbereich Mathematik, Technische Universität Berlin, 1995. `ftp://ftp.math.tu-berlin.de/pub/Preprints/combi/Report-439-1995.ps.Z`

[21] Reinelt, G. "TSPLIB: A traveling salesman problem library." *ORSA Journal on Computing* 3, 376–384, 1991.

[22] Rothberg, E. Implementation of Gabow's $O(n^3)$ version of Edmonds' algorithm for unweighted nonbipartite matching. Computer software, 1985.
`ftp://dimacs.rutgers.edu/pub/netflow/matching/cardinality/solver-1`

[23] Shapira, A. "An exact performance bound for an $O(m+n)$ time greedy matching procedure." *Electronic Journal of Combinatorics* 4:1, R25, 1997. `http://www.combinatorics.org`

[24] Tarjan, R.E. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[25] Vazirani, V.V. "A theory of alternating paths and blossoms for proving correctness of the $O(\sqrt{V}E)$ general graph maximum matching algorithm." *Combinatorica* 14:1, 71–109, 1994.

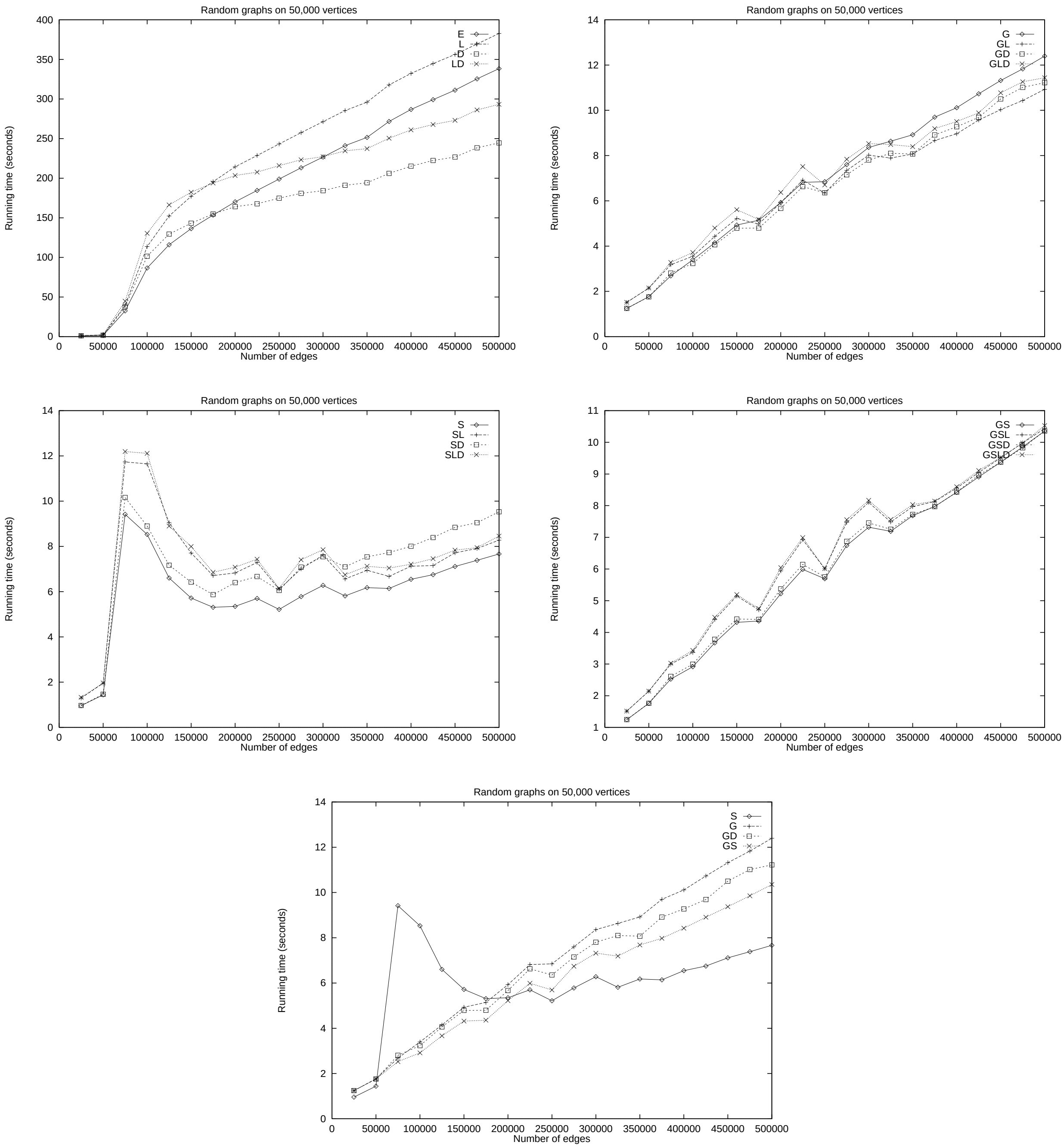

Figure 1 Running times for all combinations of heuristics on random graphs on 50,000 vertices. `E` is no heuristics, `L` is lazy expansion, `D` is delayed shrinking, `G` is greedy initial matching, and `S` is stopping rule.

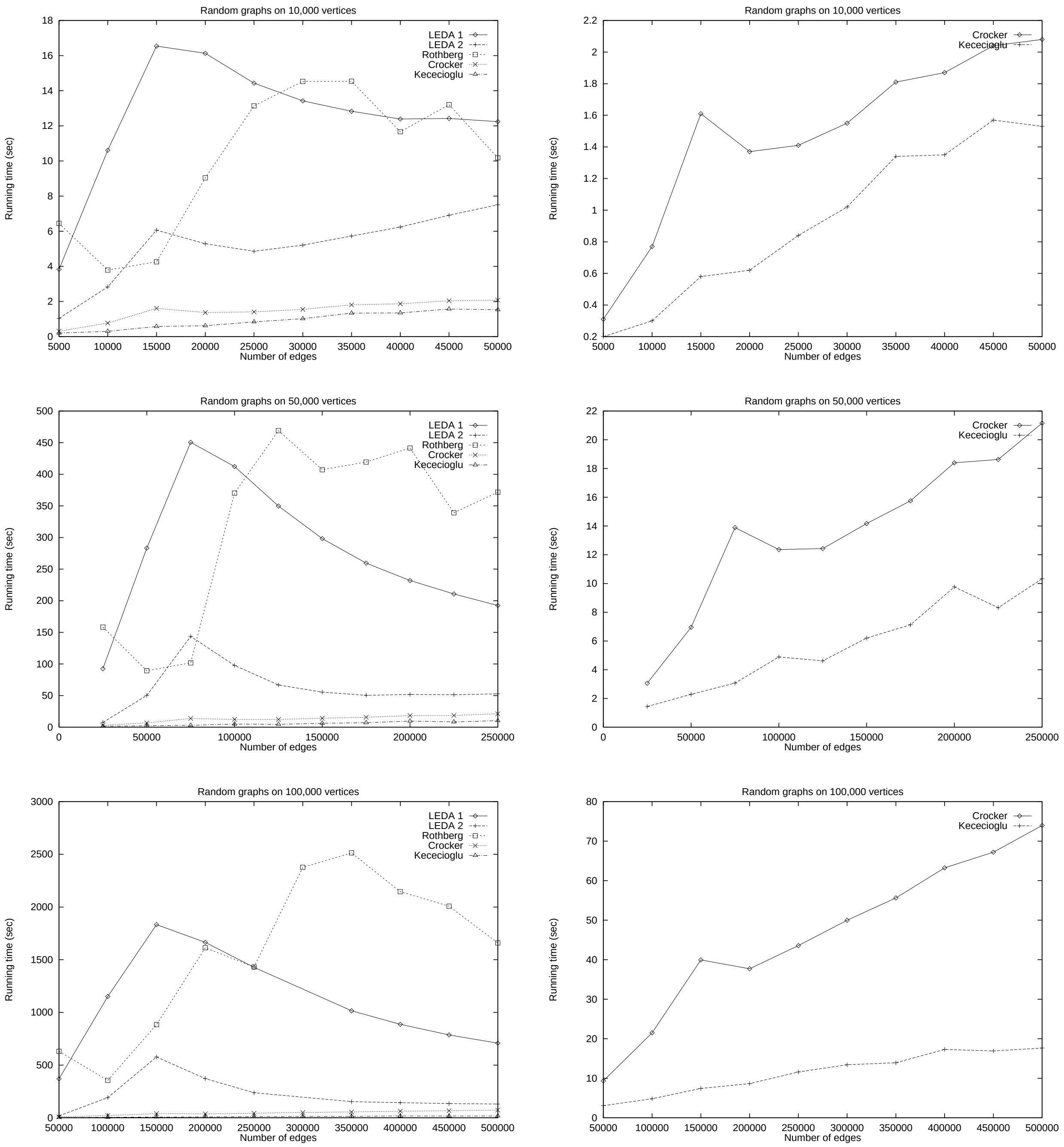

Figure 2 Running times for all codes on random graphs with number of vertices fixed and number of edges varying.

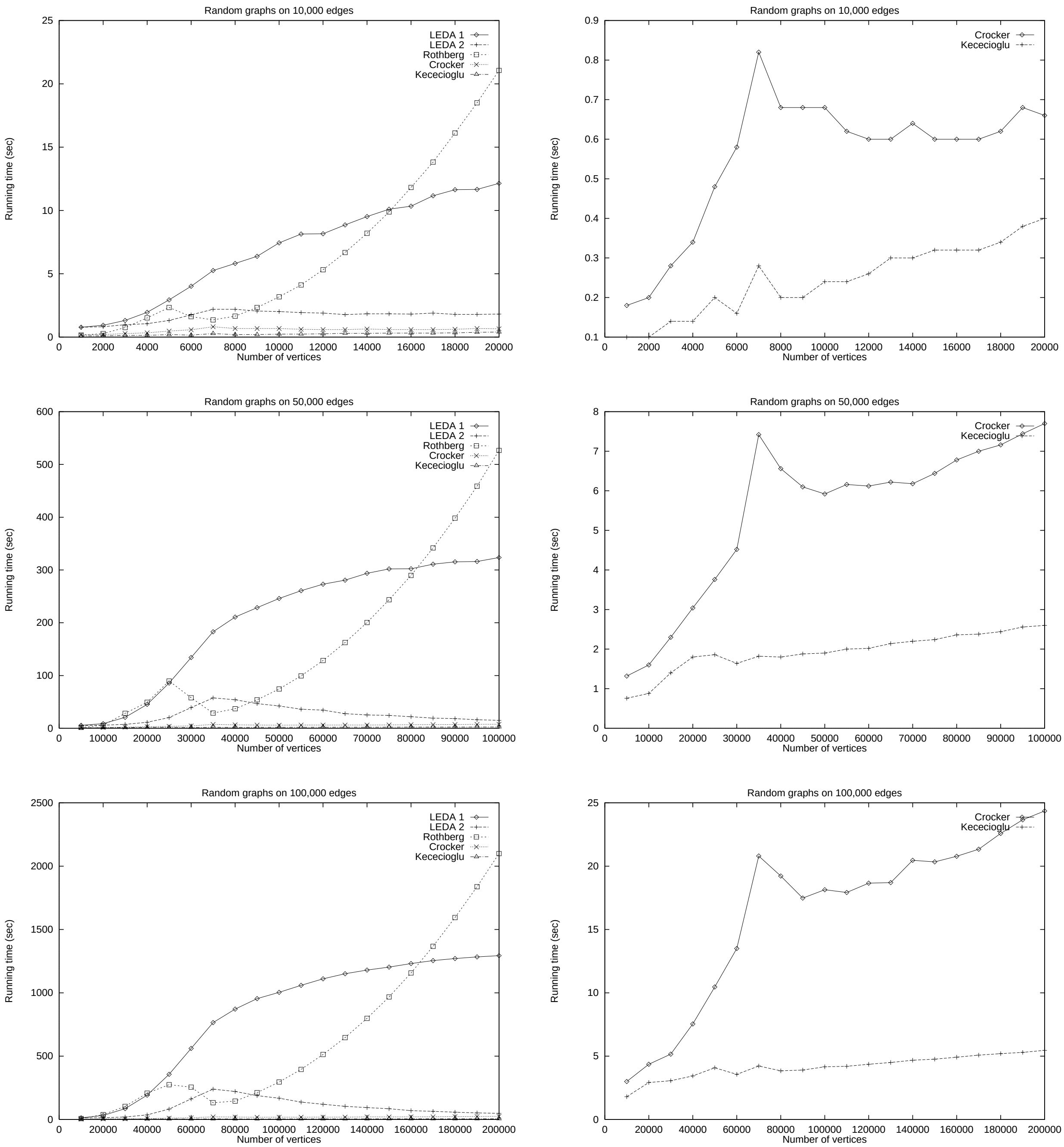

Figure 3 Running times for all codes on random graphs with number of edges fixed and number of vertices varying.

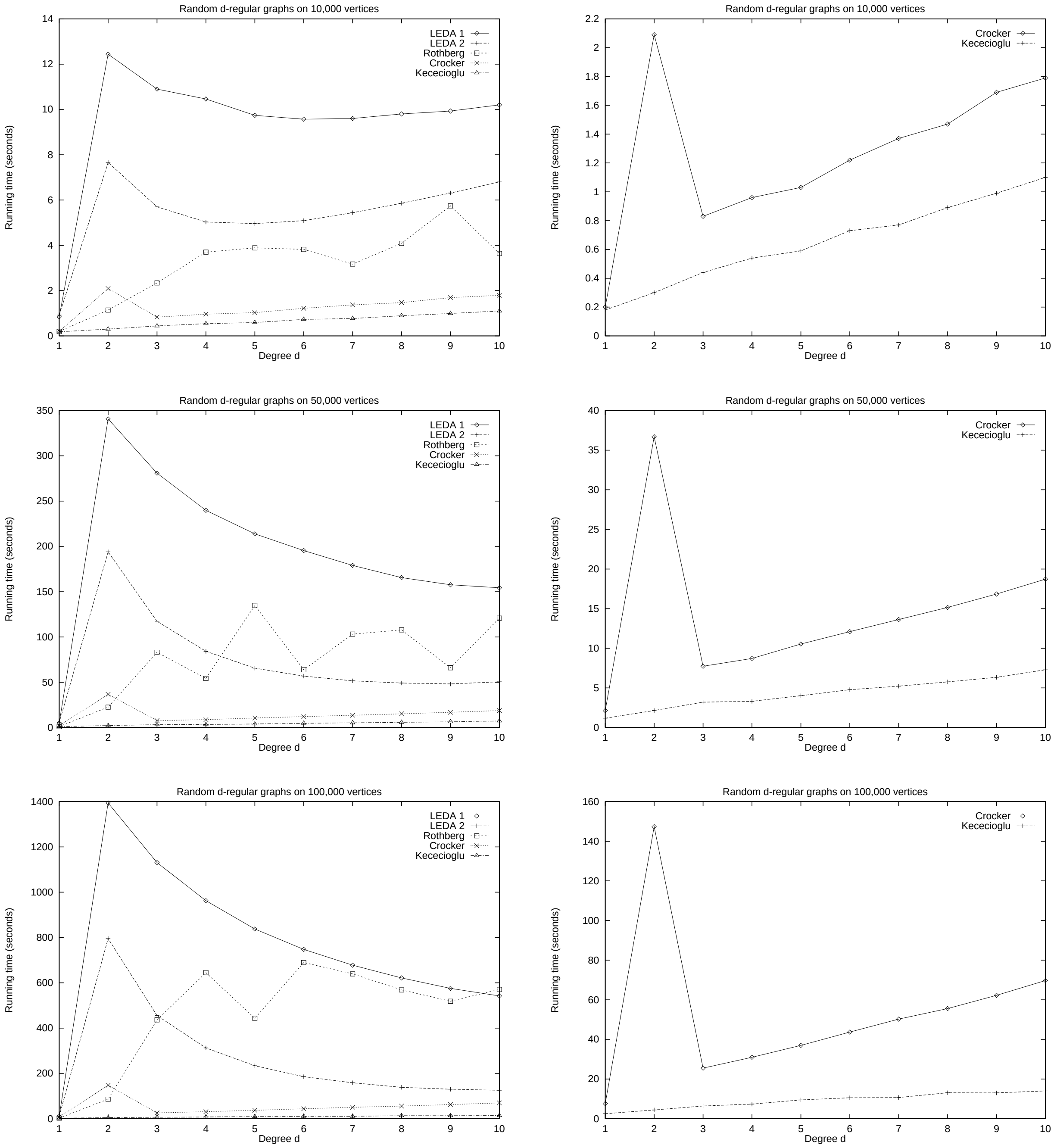

Figure 4 Running times for all codes on random d-regular graphs.

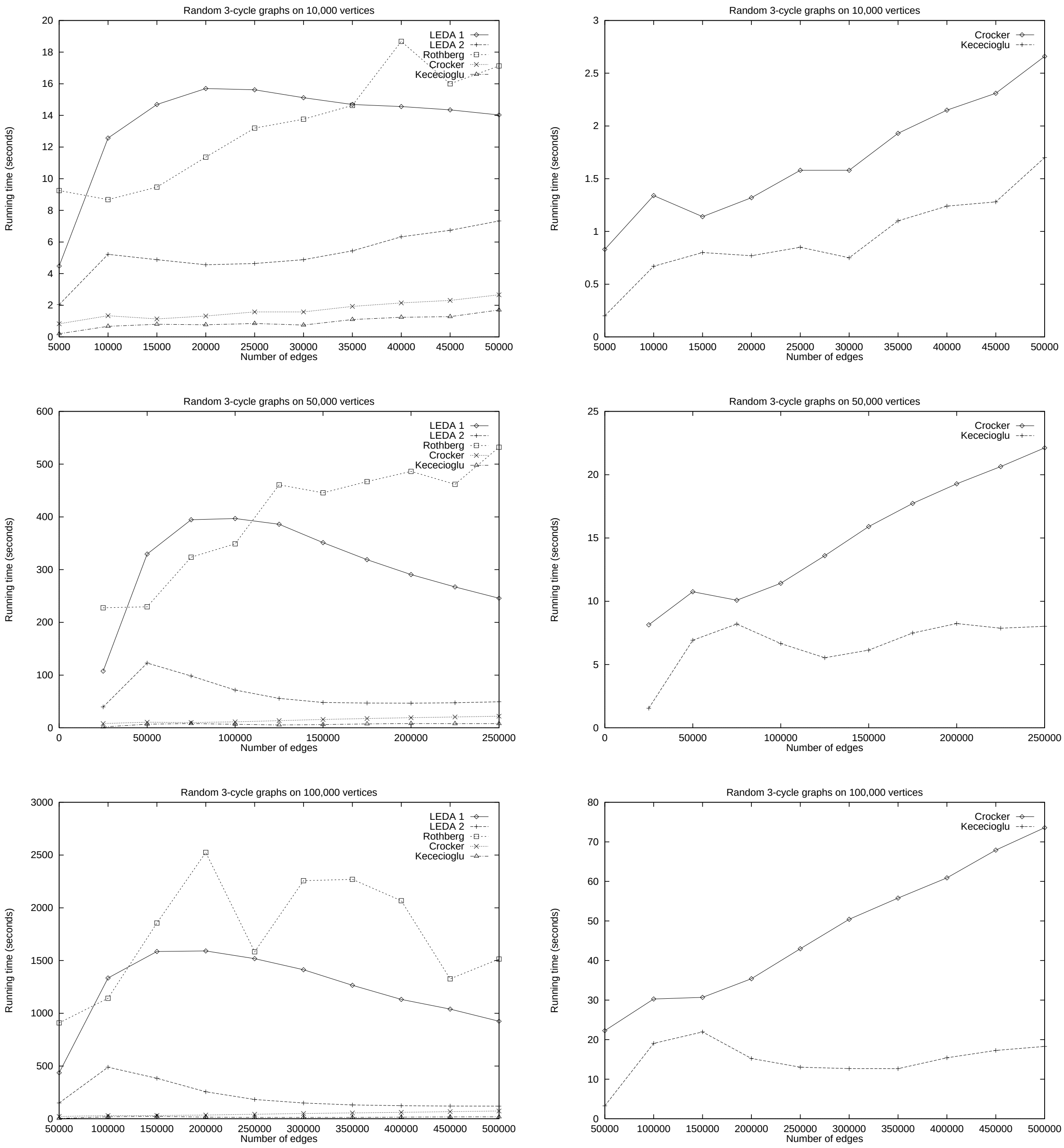

Figure 5 Running times for all codes on random 3-cycle graphs with number of vertices fixed and number of edges varying.

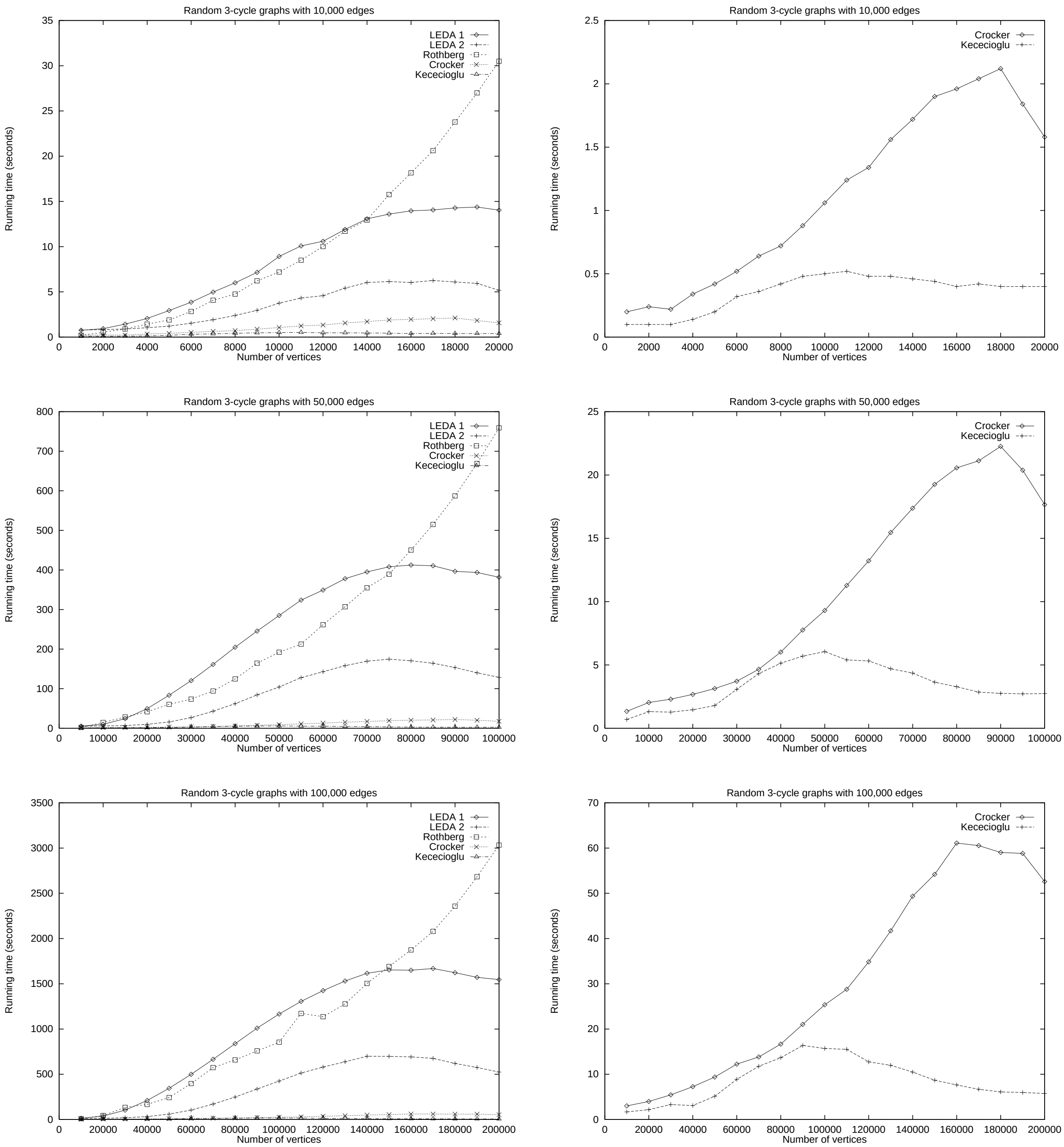

Figure 6 Running times for all codes on random 3-cycle graphs with number of edges fixed and number of vertices varying.

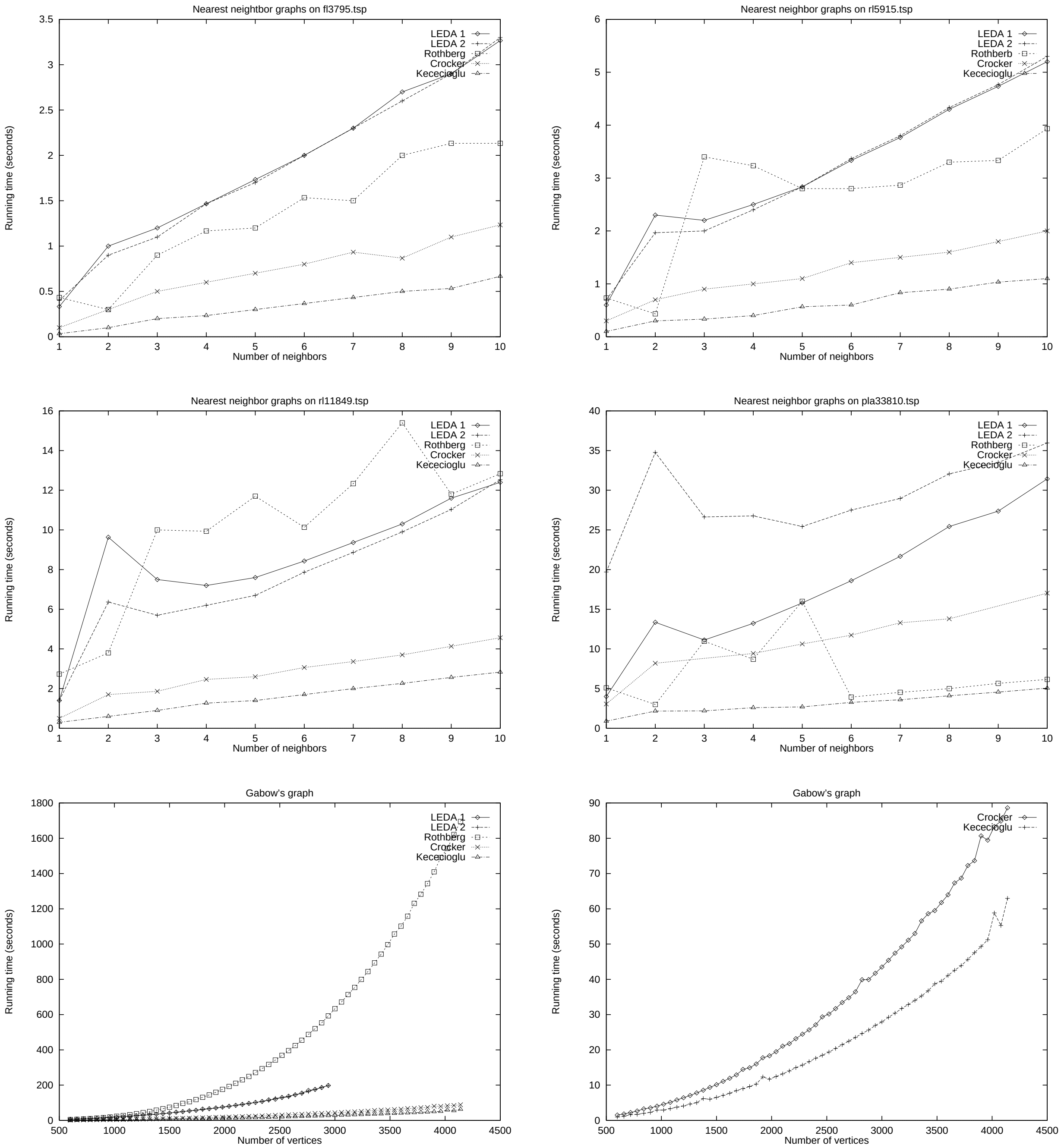

Figure 7 Running times for all codes on structured non-random graphs. Nearest-neighbor graphs are generated from Euclidean traveling salesman test sets. Gabow's graph is a worst-case instance for Edmonds' algorithm.