Representing all Minimum Spanning Trees with Applications to Counting and Generation

David Eppstein

Department of Information and Computer Science
University of California, Irvine, CA 92717
http://www.ics.uci.edu/∼eppstein/

Tech. Report 95-50

December 15, 1995

Abstract

We show that for any edge-weighted graph G there is an *equivalent graph* EG such that the minimum spanning trees of G correspond one-for-one with the spanning trees of EG. The equivalent graph can be constructed in time $O(m + n \log n)$ given a single minimum spanning tree of G. As a consequence we can count the minimum spanning trees of G in time $O(m + n^{2.376})$, generate a random minimum spanning tree in time $O(mn)$, and list all minimum spanning trees in time $O(m+n \log n+k)$ where k denotes the number of minimum spanning trees generated. We also discuss similar equivalent graph constructions for shortest paths, minimum cost flows, and bipartite matching.

1 Introduction

When studying minimum spanning trees or many other problems, one often makes an assumption of *general position*: for minimum spanning trees, one can infinitesimally perturb the edge weights so that all are distinct, in this way picking out a unique solution. This is sufficiently useful that many authors have studied general methods of achieving general position (e.g. see [8, 20]).

However it is unreasonable to expect general position instances to arise in practice (where weights may often be small integers). Further many interesting problems arise specifically in situations when the input is not in general position. One may ask how many different minimum spanning trees there are in a graph, one may wish to generate them all, or one may wish to sample one tree randomly from the space of minimum spanning trees.

In this paper, we answer these questions in a very general way, by translating them to equivalent unweighted problems. Specifically, we construct from any given edge-weighted graph G an *equivalent graph* EG without weights, such that the minimum spanning trees of G correspond one-for-one with the spanning trees of EG. Our construction involves a simple but novel local transformation, which we call the *sliding transformation*. By this construction, many problems of counting and generating minimum spanning trees become as easy as their unweighted versions.

Note that one obvious approach for our equivalent graph construction does not work. One might try removing those edges in G not part of any minimum spanning tree, but a simple example shows that this is not sufficient: if G is an isosceles triangle with the two sides longer than the base, all edges are part of some minimum spanning tree but the tree consisting of the two sides is not minimum. In this example one can instead contract edges that are part of all minimum spanning trees, but this does not work in general—it is not always possible for the equivalent graph to be formed as a *minor* of G. However we note that the first idea, of removing all edges not part of an optimal solution, does give a simple construction for equivalent graphs of problems other than minimum spanning trees: it is easy to see that we can use this idea to reduce shortest paths to unweighted paths in directed graphs, and we show later that the same idea also produces equivalent unweighted graphs for minimum cost bipartite matching.

1.1 New Results

We show the following results:

- For any edge-weighted graph G, we can in time $O(m + n \log n)$ find an *equivalent graph* EG such that the minimum spanning trees of G are in one-to-one correspondence with the spanning trees of EG.
- We can count the number of minimum spanning trees of G in polynomial time, by combining our equivalent graph construction with Kirchhoff's famous *matrix-tree theorem*.
- We can find a tree chosen uniformly at random from the minimum spanning trees of G in polynomial time, by using any of several unweighted spanning tree smapling algorithms.

- We can generate a list of all minimum spanning trees of G (in an implicit format in which each tree is described by its differences from some previous tree) in time $O(m+n\log n+k)$, where k denotes the number of trees generated, by applying the unweighted generation algorithm of Kapoor and Ramesh [17].

Similar results follow for planar point sets by using our construction on the Delaunay triangulation; we omit the details.

We give more detailed time bounds for counting and random sampling later; the bounds vary depending on the model of computation. (For example the main step in counting MSTs, computing a certain determinant, can be done in $O(n^{2.376})$ arithmetic operations [6] but a more realistic bound of $O(n^4)$ results from using Gaussian elimination and only allowing arithmetic on $O(\log n)$-bit values). In any case our time bounds are essentially the same as those for the corresponding unweighted problems.

We also briefly discuss similar (but more trivial) equivalent graph constructions for shortest paths, minimum cost flows, and minimum cost bipartite matchings. In each case the weighted problem is transformed into an unweighted problem of the same type.

1.2 Related Work

The problem of listing all minimum spanning trees is a special case of finding the k minimum-weight spanning trees for some input parameter k, which has been well studied. The best bounds known for this problem are $O(M + k\min(n,k)^{1/2})$ [9, 10], where M denotes the time to find a single minimum spanning tree. However this is slower than the bound we give for the problem of listing all minimum spanning trees, and does not extend to the problems of counting or randomly generating minimum spanning trees.

Gavril [15] considered the same problems studied here of counting and generating (but not sampling) minimum spanning trees, and described an application of these problems to tree representations of acyclic hypergraphs. He solved the counting problem by constructing a tree-like recursive structure, the root of which is the subgraph G' formed by removing all non-maximum-weight edges from G, and each subtree of which is constructed recursively from the components of $G - G'$. The minimum spanning trees of G can then be counted by multiplying together the numbers of spanning trees at each node of this structure, and all minimum spanning trees can be generated by applying a backtracking procedure to this structure. In retrospect, this can be reinterpreted as providing an equivalent graph construction with similar properties to ours, but it is more complicated and the time bounds of Gavril's paper are not as good as ours (nor even as good as a solution based on an algorithm for the k minimum spanning trees).

Finally, we note that there is a weighted version of the matrix-tree theorem (e.g. see [22], theorem 3). This states that, if each edge of a graph is given a weight $w(e)$, one can compute the sum over spanning trees

$$\sum_T \prod_{e\in T} w(e)$$

as a certain matrix determinant. On the face of it, this seems to have little to do with minimum spanning trees. However, by replacing all weights by polynomials of the form x^c where c is a small integer, computing a determinant of a polynomial-valued matrix, and extracting the appropriate coefficient of the resulting polynomial, one can use this weighted matrix-tree theorem to count all minimum spanning trees. The results one obtains in this way are less efficient than ours, and do not directly apply to problems of sampling and generation of minimum spanning trees.

1.3 Notation

We use m to denote the number of edges in a given graph, and n to denote the number of vertices. We let $w(e)$ denote the weight or cost of an edge in a graph. By extension $w(S)$ denotes the sum of weights in a subgraph S.

Throughout, we allow our graphs to have self-loops and multiple adjacencies; these can be eliminated if desired but it simplifies the theory to allow them. However note that with this convention, it is not safe to assume that $m = O(n^2)$.

2 The Equivalent Graph

2.1 The Sliding Transformation

A common method of extracting information from graphs is via *minors*: graphs obtained by the two local modifications of edge contraction and edge deletion. As discussed above, graph minors are insufficient for our purposes. Nevertheless our equivalent graph will also be constructed using a form of local modification, which we call the *sliding transformation*.

Let edges $e = (u, v)$ and $f = (v, w)$ share a common vertex v, and suppose that $w(e) < w(f)$. We define the result of *sliding* edge f along edge e as the graph G' formed by deleting f from G and inserting in its place an edge $f' = (u, w)$ with the same weight. This is only valid if $w(e) < w(f)$; if instead $w(e) \geq w(f)$ we do not perform this sliding operation.

Think of f and f' as identified with each other, so that the subgraphs of G correspond one-for-one with the subgraphs of G'. This correspondence does not preserve certain structure—for instance, f' may be a self-loop, so a tree in G may correspond to a non-tree subgraph of G'. However the following result shows that sliding preserves the space of minimum spanning trees.

Lemma 1. *Let G' be formed from G by any sequence of sliding operations. Then each set of edges giving a minimum spanning tree of G corresponds to a unique minimum spanning tree in G' and vice versa.*

2.2 The Equivalent Graph

We are now ready to define our equivalent graph EG. Let T_0 be some particular minimum spanning tree in G, and choose some vertex to be the root of T_0. Then we form EG by repeatedly performing sliding operations that slide an edge $f = (v, w)$ along an edge $e = (u, v)$ as long as e is in T_0 and u is closer to the root of T_0 than is v.

Lemma 2. *If we are given a graph G and a rooted minimum spanning tree T_0, then the graph EG described above is well-defined and does not depend on the order in which sliding transformations are performed. (However it may depend on the choise of T_0.)*

Note that T_0 remains a tree in EG, having undergone certain "path compression" operations. We now show that all spanning trees in EG are minimum; therefore the minimum spanning trees of G correspond one-for-one with the spanning trees of EG, as desired.

We need some technical results on minimum spanning trees. Given a tree T and weight w, we define $n(w, T)$ to be the number of edges in T having weight w.

Lemma 3. *Any tree T in EG is minimum iff for each w, $n(w, T) = n(w, T_0)$.*

Lemma 4. *For any w and any tree T in EG, $n(w, T) = n(w, T_0)$.*

Proof: Let w^* be the maximum edge weight in G. We first show that $n(w^*, T) = n(w^*, T_0)$. Note that in T_0, the top endpoints of any edges of weight w^* (if such edges exist) will be moved by the sliding transformation until either they run into other edges of the same weight, or they reach the root of the tree. Therefore the edges of weight w^* in T_0 form a subtree S containing the root of the tree. Suppose this subtree has k vertices, then there are $k - 1$ such edges in T_0. Any other edge of weight w^* in EG must connect two vertices in this subtree, so the edges of weight w^* in T span a forest in S, and there can be at most $k - 1$ such edges. On the other hand there must be at least $k - 1$ such edges else T_0 would not be a minimum spanning tree.

To extend this argument to weights other than w^*, note that the removal of S partitions T_0 into one or more components. The same argument as above applies to the maximum edge weight within each component, and hence to all edges in G. $\square$

Corollary 1. *Each spanning tree of EG is minimum, and the minimum spanning trees of G correspond one-for-one with the spanning trees of EG.*

2.3 Constructing The Equivalent Graph

Recall that the equivalent graph EG is formed by finding a rooted minimum spanning tree T_0 of G, and then sliding all edges in G as far as possible along paths towards the root of T_0. Implemented directly, this could take time $O(mn)$. We now show how to implement this efficiently, in time $O(m + n \log n)$.

We can find T_0 by any of various classical minimum spanning tree algorithms; for instance Prim's algorithm can be implemented to run in time $O(m + n \log n)$, which is sufficient for our purposes. Our problem then becomes one of finding, for each endpoint u of each edge $e = (u, v)$ in G, the maximal path from u in T_0 in which all edge weights are smaller than that of e.

The following simple technique suffices to achieve $O(m \log n)$ time. For any edge e in T_0, define the *heavy ancestor* of e to be the first edge on the path from e to the root having weight greater than that of e. If one lists e, its heavy ancestor e', the heavy ancestor of e' and so on, this gives a sequence of edges with increasing weights which we call the *heavy sequence* of e.

We then simply traverse T_0 in postorder. We maintain at each step the heavy sequence of the most recently traversed edge in an array A, together with an index ℓ denoting the length of the heavy sequence. When we traverse an edge $e = (u, v)$, we find its heavy ancestor e' by binary search in A. At this point we update ℓ and change the entry $A[\ell]$ to point to e, leaving the rest of A unchanged. However we store the old values of ℓ and $A[\ell]$ so they can be restored later. Next, we consider all edges having endpoints at v, and find the path on which each such edge slides by binary search in A. Then, after traversing the children of v, we reverse the traversal through e and undo the changes we made to ℓ and A.

The time for this method is $O(\log n)$ per binary search, or $O(m \log n)$ total. (More precisely it is $O(m \log d)$ where d is the depth of T_0.) To improve this to the $O(m + n \log n)$ bound claimed, we apply a *sensitivity analysis* algorithm [7, 18, 21]. Such algorithms find for each edge $e = (u, v)$ in $G - T_0$ the heaviest edge e' on the corresponding path from u to v in T_0. We use this information in two ways. First, e is in some MST iff $w(e) = w(e')$, so we can delete from EG all edges not part of some MST (this entails a modification to our definition of EG but does not harm correctness since with the original definition such edges would only become self-loops).

For those edges for which $w(e) = w(e')$ we apply the sensitivity analysis again, on a graph formed by perturbing the weights of G. With an appropriate perturbation, this can be used to find for each $e = (u, v)$ the edges e' and e'' with weights equal to $w(e)$, closest to u and v among all equal-weight edges on the path from u to v in T_0. We next use the algorithm described earlier to find the equivalent graph ET_0 of T_0 in time $O(n \log n)$. Finally, we add to this graph the edges of $G - T_0$, in the positions resulting after the sliding transformation, in constant time each by using this sensitivity analysis information.

Theorem 1. *For any edge-weighted graph G, we can in time $O(m + n \log n)$ find an equivalent graph EG such that the minimum spanning trees of G are in one-to-one correspondence with the spanning trees of EG.*

2.4 Lower Bounds

The $O(m + n \log n)$ time bound above falls short of known algorithms for minimum spanning trees, which are linear or close to linear depending on one's model of computation [12, 13, 19]. However we now describe a matching lower bound for constructing EG as defined above (it may remain possible for a faster algorithm to construct a different equivalent graph).

Theorem 2. *In the algebraic decision tree model of computation, it requires time $\Omega(n \log n)$ to compute the equivalent graph EG defined above.*

Proof: We start with the problem of, given two sets A and B of n real numbers, in which A is sorted but B is not, testing whether any member of A occurs in B. This can be shown to have an $\Omega(n \log n)$ lower bound by known techniques [2].

We translate this to our equivalent graph construction problem as follows. Let G be a path of n edges together with n edges connecting one endpoint of the path to n additional vertices. We let T_0 be formed by rooting the unique spanning tree of G at the degree-one endpoint of the path. (With a more complicated construction we could instead choose the root of T_0 arbitrarily.) We give the edges in the path the weights in A, in increasing order as one progresses towards the root. We give the remaining edges in G the weights in B. Then the sliding transformation shifts each edge corresponding to a weight in B up the path until it reaches a weight equal or greater to its own. One can then test whether any member of A occurs in B by testing, for each edge in B, whether it has the same weight as its parent in EG. $\square$

3 Applications

We now solve several problems of counting or generating minimum spanning trees, by applying a solution of the unweighted problem to the equivalent graph EG constructed above.

3.1 Counting

We first consider the problem of determining the number of minimum spanning trees in a given graph. To our knowledge this problem has not been considered before, although its unweighted version (counting spanning trees) is quite well known.

The *matrix-tree* theorem, as formulated e.g. in [3], is that the number of spanning trees in G equals $\det(KK')$, where we form K by deleting a row from the incidence matrix of G. KK' can also be constructed directly from the adjacency matrix of G. The time to count spanning trees is dominated by the determinant evaluation, which can be done in $O(n^{2.376})$ arithmetic operations [6]. In a more limited computational model in which constant time operations are allowed only on words of $O(\log n)$ bits, and using a more practical matrix multiplication algorithm, the time might be as much as $O(n^4)$. For simplicity of exposition we adopt the former model.

Theorem 3. *We can count the number of minimum spanning trees of G in $O(m + n^{2.376})$ arithmetic operations.*

Proof: We construct EG and count its spanning trees using the matrix-tree theorem. $\square$

3.2 Generation

We next consider the problem of listing all minimum spanning trees of a graph. This problem was posed in 1994 by J. L. Ganley [14] and was the starting point for our work here. It was also solved earlier by Gavril [15]. There have been many papers on the related problem of listing the k best spanning trees for some given parameter k. Algorithms for the k best spanning trees can also be adapted to list all spanning trees with total weight better than some given bound; the best methods for this problem of bounded spanning tree generation take time $O(M + \min\{k^{3/2}, kn^{1/2}\})$, where M is the time to compute a single minimum spanning tree [9, 10]. This bound therefore can be applied to minimum spanning tree generation. Note that the time per tree is $o(n)$, so there is not time to output the trees explicitly; instead an implicit representation is generated in which the edge set of each tree is described by its symmetric difference with some previous tree.

Kapoor and Ramesh [17] use a similar implicit representation in two algorithms for listing all spanning trees of a graph, in the much better time bound $O(m+n+k)$ where again k denotes the number of trees output. (See their paper for references to earlier work on this problem.) They also give a third algorithm that lists all spanning trees in sorted order by weight, in time $O(mn+k\log n)$. One technique for listing minimum spanning trees would be to run this third algorithm until it produces a non-minimum tree, howevever this would not give a good time bound as the analysis of this algorithm amortizes present work against future trees.

We next combine our equivalent graph technique with either of the first two algorithms of Kapoor and Ramesh to solve the minimum spanning tree generation problem efficiently.

Theorem 4. *We can generate a list of all minimum spanning trees of G (in an implicit format in which each tree is described by its differences from some previous tree) in time $O(m+n\log n+k)$, where k denotes the number of trees generated.*

Proof: We construct EG in time $O(m+n\log n)$, and then use the algorithms of Kapoor and Ramesh to list all spanning trees of EG in time $O(m+n+k)$. $\square$

3.3 Sampling

Several authors have considered the problem of choosing a spanning tree uniformly at random. We now adapt these results to random generation of minimum spanning trees.

There are three basic approaches for sampling random spanning trees. The first is to select one edge at a time to be in or out of the tree, using the the matrix-tree theorem to find the appropriate probabilities; this gives an exactly uniform choice in $O(mn^{2.376})$ arithmetic operations [16]. (Broder [5] cites two papers with more sophisticated versions of this approach that reduce the time to that for a single determinant.)

The second approach to spanning tree generation can be found in papers by Aldous and Broder [1, 5]. Their method takes a random walk on the vertices of given graph, and selects for each vertex (except the start point) the edge by which that vertex was first visited. These $n-1$ edges form a spanning tree which can be shown to be selected exactly uniformly at random. The expected time of this algorithm is the *covering time* of the graph, $O(mn)$ in the worst case.

The third approach has been much used recently for similar random generation problems: take a different random walk, in a (large) graph representing the space of solutions to a given problem. After sufficiently many steps, the distribution of possible solutions corresponding to the endpoint of the walk will be very close to uniform. Broder [5] suggested this approach, and showed that a certain random walk converges to the uniform distribution within a polynomial number of steps. Further, each step of the walk takes $O(\log n)$ time, so the time to generate a tree with probabilities within $(1+\epsilon)$ of uniform is $O((n\log n+\log\frac{1}{\epsilon})mn^2\log n)$ [11]. Feder and Mihail [11] describe a different random walk, which converges more quickly but is harder to implement. With the aid of fast dynamic graph algorithms [9, 10] their method can select a tree in time $O((n \log n + \log \frac{1}{\epsilon})n^{3.5})$, after some data structure initialization taking time $O(m)$. This bound has an advantage for graphs with very many edges, since the dependence on m is linear and occurs only once even if several samples are drawn.

Our equivalent graph construction lets us use any of these methods to select a random minimum spanning tree. The random walk method could instead be applied directly (giving an algorithm based on a random walk in the space of minimum spanning trees of a graph) but this seemingly requires different proofs of convergence and more complicated algorithms for performing the steps of the walk; our equivalent graph method instead allows the previous proofs and algorithms to be applied as is. We state without proof the result of translating two of the algorithms cited above:

Theorem 5. *In expected time $O(mn)$ we can find a spanning tree chosen uniformly at random among all minimum spanning trees of a given graph. Alternately, after an $O(m + n \log n)$ time preprocessing stage, we can sample the minimum spanning trees of G with probabilities within $(1+\epsilon)$ of uniform, in time $O((n \log n + \log \frac{1}{\epsilon})n^{3.5})$ per sample.*

4 Paths, Flows, and Matchings

We next briefly discuss similar construction of equivalent graphs for the problems of finding shortest paths, minimum cost flows, and minimum weight bipartite matchings. In each case we represent the space of solutions as a similar unweighted problem on a similar graph. The constructions are, surprisingly, easier than those for the minimum spanning tree problem.

Theorem 6. *Let weighted directed graph G and vertices s, t be given. Construct a graph G' by removing from G all edges not on a shortest path from s to t. Then the shortest paths from s to t in G are in one-to-one correspondence with the paths from s to t in G'.*

The proof involves applying a potential function so that the edges in G' have weight zero, and other edge weights in G become positive; we omit the details.

For the minimum cost flow problem, we form an equivalent graph by modifying the edge capacities. Given flow graph G and terminals s, t, Let $f_{\min}(e)$ denote the minimum amount of flow on edge e among all minimum cost maximum flows from s to t. Similarly let $f_{\max}(e)$ denote the maximum flow amount on e in the same space of minimum cost flows.

Theorem 7. *Let flow graph G' be formed by replacing the minimum and maximum flow capacities of each edge in G by $f_{\min}(e)$ and $f_{\max}(e)$ respectively. Then the minimum cost flows from s to t in G are in one-to-one correspondence with the maximum flows from s to t in G'.*

Proof: First note that any minimum cost maximum flow in G is also a maximum flow in G'. Suppose some flow $F = F(e)$ in G' does not have minimum cost; then the cost can be reduced by pushing flow around some cycle C.

Now consider a collection of minimum cost flows in G' formed as follows. For each edge e, let $F^e_{\min}$ be any minimum cost flow having flow amount $f_{\min}(e)$ on e, and define $F^e_{\max}$ similarly. Let

F' be formed as the average of these $2m$ flows. Then the only saturated edges in F' are those for which $f_{\min}(e) = f_{\max}(e)$, so in F' one can push flow around the same cycle C, contradicting the fact that F' has minimum cost. □

We now apply this result in the construction of an equivalent graph for bipartite matching, based on a standard transformation from this problem to minimum cost flow.

Theorem 8. *Let $G = (U, V, E)$ be a weighted undirected bipartite graph with at least one perfect matching, and let G' be formed by removing from G those edges not part of any minimum cost perfect matching. Then the minimum cost matchings in G are in one-to-one correspondence with the perfect matchings in G'.*

Proof: The minimum cost perfect matchings correspond to the minimum cost integer maximum flows in a flow graph formed by directing all edges from U to V, adding edges from a terminal s to U and from V to a terminal t, and giving each edge unit capacity. By applying Theorem 7 to this flow graph, we get an equivalent graph which has been modified only by restricting the capacities of certain edges. The restrictions must be to integer values, which must be either zero or one. Thus some edges are forced to have unit flow (and hence if in G, to be in the matching, forcing the neighboring edges to not be in the matching) while some other edges are forced to have zero flow (and hence effectively are removed from G). The effect in G of this transformation can thus be described simply as removing certain edges from G, and the edges removed can only be the ones described in the statement of the theorem. □

A simple example shows that the approach of Theorem 8 does not work for non-bipartite matching: let $G = K_6$ with unit edge weights on two disjoint triangles, and edge weight two on the nine remaining edges. Then each weight-two edge is in a unique minimum cost matching, so no edge can be removed from G, however there are many non-minimum matchings in G. In this example one can find an equivalent graph by contracting the two triangles in G; perhaps a similar blossom-contraction method will work for general non-bipartite matching.

The equivalent graph for shortest paths can be found by using two iterations of Dijkstra's algorithm to compute the distances of each vertex from s and t. It remains an interesting problem whether similarly efficient methods can be used to speed up the computation of equivalent graphs for flows and matchings over the obvious method of testing each edge separately.

Acknowledgements

Work supported in part by NSF grant CCR-9258355 and by matching funds from Xerox Corp. I would like to thank Mike Dillencourt for the suggestion to apply the equivalent graph idea to matchings, and Nati Linial for a pointer to the weighted matrix-tree theorem.

References

[1] D. Aldous. The random walk construction for spanning trees and uniform labeled trees. *SIAM J. Disc. Math.* 3 (1990) 450–465.

[2] M. Ben-Or. Lower bounds for algebraic computation trees. *15th ACM Symp. Theory of Computing* (1983) 80–86.

[3] J. A. Bondy and U. S. R. Murty. Graph Theory with Applications. North Holland, 1976, pp. 218–219.

[4] A. Z. Broder. How hard is it to marry at random? (On the approximation of the permanent), *18th ACM Symp. Theory of Computing* (1986) 50–58.

[5] A. Z. Broder. Generating random spanning trees. *30th IEEE Symp. Foundations of Computer Science* (1989) 442–447.

[6] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *J. Symbolic Computation* 9 (1990) 251–80.

[7] B. Dixon, M. Rauch, and R. E. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM J. Comput.* 21 (1992) 1184–1192.

[8] H. Edelsbrunner and E. P. Mücke. Simulation of Simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graphics* 9 (1990) 66–104.

[9] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification—A technique for speeding up dynamic graph algorithms. *33rd IEEE Symp. Foundations of Computer Science* (1992) 60–69.

[10] D. Eppstein, Z. Galil, and G. F. Italiano. Improved sparsification. Tech. Rep. 93-20, Dept. of Information and Computer Science, Univ. of California, Irvine, 1993.

[11] T. Feder and M. Mihail. Balanced matroids. *24th ACM Symp. Theory of Computing* (1992) 26–38.

[12] M.L. Fredman and D.E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *31st IEEE Symp. Foundations of Computer Science* (1990) 719–725.

[13] H.N. Gabow, Z. Galil, T. Spencer, and R. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica* 6 (1986) 109–122.

[14] J. L. Ganley. Enumerating minimum spanning trees. Posting to bulletin board comp.theory, 31 Oct 1994.

[15] F. Gavril. Generating the maximum spanning trees of a weighted graph. *J. Algorithms* 8 (1987) 592–597.

[16] A. Guénoche. Random spanning tree. *J. Algorithms* 4 (1983) 214–220.

[17] S. Kapoor and H. Ramesh. Algorithms for enumerating all spanning trees of undirected and weighted graphs. *SIAM J. Comput.* 24 (1995) 247–265.

[18] V. King. A simpler minimum spanning tree verification algorithm. *4th Worksh. Algorithms and Data Structures* (1995) to appear.

[19] P. Klein and R. E. Tarjan. A randomized linear-time algorithm for finding minimum spanning trees. *26th ACM Symp. Theory of Computing* (1994) 9–15.

[20] K. Mulmuley, U. V. Vazirani, and V. V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica* 7 (1987) 105–114.

[21] R. E. Tarjan. Applications of path compression on balanced trees. *J. ACM* 26 (1979) 690–715.

[22] H. Trent. A note on the enumeration and listing of all possible trees in a connected linear graph. *Proc. Nat. Acad. Sci. U.S.A.* 40 (1954) 1004–1007.