Parallel Processing Letters
© World Scientific Publishing Company

PARALLEL REAL-TIME OPTIMIZATION: BEYOND SPEEDUP *

SELIM G. AKL and Stefan D. Bruda
Department of Computing and Information Science, Queen's University
Kingston, Ontario K7L 3N6, Canada
Email: {*akl,bruda*}*@cs.queensu.ca*

Received (received date)
Revised (revised date)
Communicated by (Name of Editor)

ABSTRACT

Traditionally, interest in parallel computation centered around the speedup provided by parallel algorithms over their sequential counterparts. In this paper, we ask a different type of question: Can parallel computers, due to their speed, do more than simply speed up the solution to a problem? We show that for real-time optimization problems, a parallel computer can obtain a solution that is better than that obtained by a sequential one. Specifically, a sequential and a parallel algorithm are exhibited for the problem of computing the best-possible approximation to the minimum-weight spanning tree of a connected, undirected and weighted graph whose vertices and edges are not all available at the outset, but instead arrive in real time. While the parallel algorithm succeeds in computing the exact minimum-weight spanning tree, the sequential algorithm can only manage to obtain an approximate solution. In the worst case, the ratio of the weight of the solution obtained sequentially to that of the solution computed in parallel can be arbitrarily large.

Keywords: parallel computation, speedup, real time, optimization, minimum-weight spanning tree.

1. Introduction

Ever since parallel computation appeared on the computer science scene as an alternative to conventional (that is, sequential) computing, questions were raised regarding the capabilities of parallel computers. The vast majority of these questions have to do with the *speedup*, if any, provided by parallel computers: Can parallel computers solve computational problems faster than sequential computers, and if so, how much faster? In this paper, we ask a different type of question: Can parallel computers, due to their speed, do more than simply speed up the solution

*This research was supported by the Natural Sciences and Engineering Research Council of Canada.

to a problem? In particular, can a parallel computer provide a solution to an optimization problem that is *better* than the best-possible solution that can be obtained sequentially?

We begin by reviewing some of the issues surrounding the notion of speedup, leading up to the central question of this paper. In what follows, the speedup provided by a parallel algorithm when solving a problem is defined as follows: Worst-case running time of the best sequential algorithm for the problem divided by the worst-case running time of the parallel algorithm. Throughout the paper we adopt the traditional definition of *time unit*, that is, the unit used to measure the running time of an algorithm: A time unit is the length of time required by a processor to: read a datum from memory, perform a constant-time operation (such as adding two numbers), and write a datum to memory.

1.1. Speedup

As the *raison-d'être* of parallel computers is the speeding up of computations performed sequentially (that is, using one processor), the first question to be asked was: *Is speedup possible at all?* More specifically, and perhaps more precisely, if a computation requires T_1 time units on a one-processor computer, can it be performed on a parallel computer with n processors, $n > 1$, in time $T_n = T_1/f(n)$, where $f(n)$ is $\omega(1)$ and $O(n)$, that is, $f(n)$ is asymptotically larger than any constant and asymptotically no larger than n?

It is now widely known that this question is answered in the affirmative, at least in theory. There is ample and well documented evidence of parallel algorithms whose running time satisfies the aforementioned condition. For instance, parallel algorithms using n processors and running in $O(\log n)$ time exist to

(i) Sort n numbers in non-decreasing order using comparisons [31],
(ii) Find the convex hull of n planar points [20],
(iii) Compute the discrete Fourier transform of n inputs [33],

to name just a few examples of problems for which the best sequential algorithms run in $O(n \log n)$ time.

1.2. Superlinear speedup

Having established that a speedup by a factor of $f(n)$ is indeed possible, it was natural to ask whether further speedup could be achieved through parallelism. The second question, therefore, was: *Is superlinear speedup possible?* In other words, can a computation requiring T_1 time units sequentially be performed on a parallel computer with n processors in time $T_n = T_1/g(n)$, where $g(n)$ is $\Omega(n)$, that is, $g(n)$ is asymptotically larger than n? Once again, several examples of computations satisfying this condition have been published. These examples, are less well known as they concern nonstandard, yet realistic, paradigms, including, for example, problems where

(i) All the data are not available at the outset of the computation, but instead arrive over time; the computation is considered complete when all the data arrived so far have been handled regardless of whether more data arrive later [6, 7, 24, 25],

(ii) The values of the data change as the algorithm proceeds; the computation is considered complete when all the corrections arrived so far have been handled regardless of whether more corrections arrive later [8, 24, 25],

(iii) A computation is involved that is not efficiently invertible: With a sufficient degree of parallelism, the inverse computation is not required; with an insufficient number of processors, the inverse computation becomes necessary [2].

1.3. Infinite speedup

Taking the line of reasoning expressed in section 1.2 to its logical limit, the ultimate question was: *Are there computations for which parallelism makes the difference between success and failure?* In other words, are there computational problems whose solution can be obtained only on a parallel computer with sufficiently many processors (while any computer with fewer processors is guaranteed to fail in computing the solution)? Recent work has demonstrated that the answer here is also positive. Problems were first described in [26] whose solution, for certain input sizes, requires infinite time when computed sequentially and finite time when computed in parallel. Several examples of such problems were presented in [26], and later in [1], including computations with deadlines, computations involving several streams of input, and computations where data arrive in real time but each new input depends on the previous output.

1.4. Beyond speedup

The purpose of this paper is to begin the exploration of other capabilities of parallel computers beyond speedup (as originally hinted to in [1]). To this end, we wish to ask whether parallel computers can, in some circumstances, do more than just speed up the computation. We show that for real-time optimization problems, a parallel computer can obtain a solution that is better than that obtained by a sequential one.

Specifically, the following computation is considered: Given a connected, undirected and weighted graph whose vertices and edges are not all available at the outset, but instead arrive in real time, it is required to find the best-possible approximation to the minimum-weight spanning tree of that graph. For this problem, a sequential and a parallel algorithm are exhibited. While the parallel algorithm succeeds in computing the *exact* minimum-weight spanning tree, the sequential algorithm can only manage to obtain an approximate solution. In the worst case, the ratio of the weight of the solution obtained sequentially to that of the solution computed in parallel can be arbitrarily large.

The remainder of this paper is organized as follows. In section 2 the real-time optimization paradigm is defined. Section 3 sets the minimum-weight spanning tree problem in the context of this paradigm, and presents the two algorithms together with their analyses. Some concluding remarks and suggestions for future investigations are offered in section 4.

2. Solving Real-Time Optimization Problems

In this section we describe the paradigm chosen for our analysis, namely, real-time optimization. We begin by defining optimization, then real-time computation.

2.1. Optimization

The family of optimization problems, as used in this paper, is defined as follows. Each problem in the family takes as input a finite set of M data. It is required to select a subset of this set, consisting of m elements (where m is a positive integer, which may or may not be given). The selected subset must satisfy certain conditions germane to the problem being solved. Among all subsets satisfying the conditions, we are to find the one maximizing (or minimizing) a given function ϕ of m arguments. This form of optimization is commonly referred to as *discrete* or *combinatorial* optimization [23, 27].

Examples of optimization problems include finding the shortest path between two vertices in an undirected, connected, and weighted graph, finding a maximum-sum subsequence of a sequence of numbers, and finding a minimum-weight complete matching for an even number of points in the plane (where the weight of the edge joining two points is the Euclidean distance separating them).

2.2. Real-time computation

Problems solved in real-time, as used in this paper, are characterized by the fact that all their data are not available at the outset of the computation. Instead, the data arrive as the algorithm proceeds. Initially, a small set of data is given and a partial solution to the problem is computed. Subsequently, more data arrive at regular intervals. Each new datum received must be incorporated into the solution. Real-time computation is sometimes known as *on-line* computation, by contrast with *off-line* computation in which all the required data are available at the outset [14, 16, 17, 18, 22, 30]. The adjectives *updating*, *incremental*, and *dynamic* are also often used to refer to algorithms that receive and process new data [4, 9, 10, 11].

Examples of real-time computations include sorting a sequence of numbers, computing the convex hull of a set of points in the plane, and finding the longest increasing subsequence of a sequence of numbers.

2.3. *Real-time optimization*

The computational paradigm used in this paper has the following characteristics:

(i) A set $\mathcal{S}$ of M data is given and represents the input to an optimization problem $\mathcal{P}$.
(ii) Initially, an optimal solution to $\mathcal{P}$ is known for the set $\mathcal{S}$.
(iii) Time is divided into *intervals*. Each interval is $\mathcal{T}$ time units long, where $\mathcal{T}$ is a function of M.
(iv) Additional data for $\mathcal{P}$ arrive in real time and must be used to compute a new solution to $\mathcal{P}$ without any knowledge of future data to be received. The solution should be optimal, or the best-possible one that can be computed in $\mathcal{T}$ time units. Specifically, at the beginning of each time interval:
   (a) If data were received at the beginning of the previous interval, the best solution computed so far (incorporating these data) is sent to an output device in order to be produced to the outside world;
   (b) If new data are received (in the present interval) they must be used to compute a new optimal (or best-possible) solution; such computation can last only the length of the present interval.
(v) There may or may not be a bound on the total number of data received throughout the computation.

3. Real-Time Minimum Spanning Trees

Our chosen problem for illustrating the ability of a parallel algorithm to do better than the best sequential algorithm when solving optimization problems in real time is the well-known *minimum-weight spanning tree problem*. We begin with a few definitions then introduce the problem to be solved. This is followed by a description of sequential and parallel solutions. An analysis of the two solutions then follows.

3.1. *The minimum-weight spanning tree problem*

A *tree* is a connected and undirected graph with no cycles. Given a connected and undirected graph G, a *spanning tree* of G is a subgraph of G which, in addition to being a tree, includes all the vertices of G. Suppose now that G is *weighted*, that is, a real number is associated with each of its edges. A *minimum-weight spanning tree* (MST) of G is a spanning tree of G, such that the sum of its edge weights is a minimum (among all spanning trees of G). Several sequential and parallel algorithms exist for solving the minimum-weight spanning tree problem efficiently; see, for example, [1, 5, 13, 20]. Note that even though the graph G may have more than one MST, we refer in what follows to 'the' MST of G for simplicity.

3.2. *Computing the MST in real time*

The real-time version of the minimum-weight spanning tree problem is defined as follows:

(i) A connected, undirected, and weighted graph G with n vertices and the $n(n-1)/2$ edges connecting them is given, where n is a positive integer larger than one. This graph represents the input to the minimum-weight spanning tree problem.

(ii) Initially, the MST of G is known; it consists of n vertices and the $n-1$ weighted edges connecting them. These $n-1$ edges defining the MST are the only data that matter for the rest of the computation (in other words, the remaining edges of G that are not in the MST, while still available, are irrelevant from this point on).

(iii) Time is divided into intervals of cn^ϵ time units, where c is a positive constant and $0 < \epsilon < 1$.

(iv) A new vertex and its associated edges are received at the beginning of every time interval. A new MST, or a best approximation possible to it, incorporating the new data must now be computed. The new tree is then produced as output at the beginning of the next time interval. Specifically,

   (a) At the beginning of the kth interval a new vertex is received along with $n+k-1$ weighted edges connecting it to the existing $n+k-1$ vertices, $k \geq 1$.

   (b) The best-possible approximation to the MST that can be obtained in cn^ϵ time units is then computed over the $n+k$ vertices and $(n+k-2)+(n+k-1)$ edges.

(v) At most n^2-n new vertices are received in all. If this many vertices are indeed received, the last tree will be produced as output $(n^2-n) \times cn^\epsilon$ time units after the beginning of computation.

3.3. *Sequential algorithm*

The problem to be solved is as follows. The MST (or an approximation thereof) is given for some graph G. The tree is defined over $n+k-1$ vertices, and consists therefore of $n+k-2$ edges. A new vertex is received as input along with its associated $n+k-1$ weighted edges connecting it to each of the existing vertices. It is required to compute a new MST (or an approximation thereof), that includes the new vertex and takes the new edges into consideration. This tree would have $n+k$ vertices and $n+k-1$ edges. An important requirement is that the computation must be completed in cn^ϵ time units.

There are two well-known algorithms for computing the MST sequentially [5, 13]. However, neither of these algorithms can be used here as they would require time on the order of $(n+k)^2$ and $(2n+2k-3)\log(2n+2k-3)$ time units, respectively. Two sequential algorithms are also known that are capable of updating an existing

MST [12, 34]. Unfortunately, these algorithms cannot be used either as they run in time on the order of $2n + 2k - 3$ time units.

The only viable approach for a sequential algorithm is to replace up to n^{ϵ} of the existing edges with an equal number of new edges of smaller weight, such that the entire computation is completed within one time interval. The resulting tree is not guaranteed to be the MST.

3.4. Parallel algorithm

The problem that a parallel computer needs to solve is exactly the same as in section 3.3. There exists a parallel algorithm for computing the MST (not an approximation to it) for a graph with $n+k$ vertices and $2n+2k-3$ edges in time on the order of $\log(n + k)$ time units using $2n + 2k - 3$ processors [3]. The algorithm runs on a Concurrent-Read Concurrent-Write Parallel Random Access Machine (CRCW PRAM) in which write conflicts are resolved using the MINIMUM rule: This means that when several processors attempt to write in the same memory location simultaneously, only the one writing the smallest value succeeds. (Another equivalent option is to use the PRIORITY rule, where processors are assigned certain priorities to resolve write conflicts: Here, a processor's priority is the weight and index of the edge associated with it.)

In the worst case, $n^2 - n$ new vertices are received. As a result, the last (and largest) problem to be solved by the parallel algorithm is to find the MST for a graph with n^2 vertices and $2n^2 - 3$ edges. This requires a running time on the order of $\log n$ time units, which is asymptotically smaller than n^{ϵ}. Consequently, the computation can be completed within one time interval.

3.5. Analysis

In the best case, the sequential algorithm manages to produce the (exact) MST for every new vertex received. Typically, however, the spanning tree produced after receiving a new vertex (and its associated edges) is only an approximation of the MST. In the worst case, this approximation can be arbitrarily bad.

For the sake of definiteness when comparing the sequential and parallel solutions, we make some concrete assumptions. Suppose that

(i) A spanning tree over $n+k-1$ vertices is known. Each of the $n+k-2$ existing edges has a weight of 2^n units (of weight).
(ii) A new vertex now arrives. Each of the $n + k - 1$ new edges has a weight of one unit (of weight).

Clearly, the MST for the new graph (now defined over $n + k$ vertices and $2n + 2k - 3$ edges) consists entirely of the newly received $n + k - 1$ edges.

The sequential algorithm can at best replace n^{ϵ} of the existing edges with new ones. This way it obtains an approximation to the MST whose weight is $2^n(n + k - 2) - 2^n n^{\epsilon} + n^{\epsilon}$ units. By contrast, the parallel algorithm can compute the MST exactly. Its solution has a weight of $n+k-1$ units. The ratio of the weight of the sequential solution to the weight of the parallel solution is therefore on the order of 2^n.

4. Conclusion

The overwhelming majority of theoretical and empirical analyses of parallel algorithms use the speedup provided by these algorithms as a measure of their goodness. Speedup is usually defined as the ratio of the time required by the best sequential algorithm solving the problem at hand to the time required by the parallel algorithm being evaluated. Here, *time* refers to *worst-case time* and is typically a function of the size of the problem. It is also customary to express the number of processors used by a parallel algorithm as a function of the size of the problem. For these reasons, speedup has been traditionally evaluated in terms of its relation to the number of processors. Thus, a speedup may be sublinear, linear, or superlinear in the number of processors.

In this paper we articulated the thesis that other measures of the goodness of parallel algorithms may be employed. In particular, one such measure proposed here is the quality of the solution obtained by a parallel algorithm to a real-time optimization problem. To illustrate this point we exhibited an example of such a problem, namely the computation of the minimum-weight spanning tree in an environment where the vertices and edges of the input graph arrive in real time. The ratio of the solution obtained by the best possible sequential algorithm to that obtained by the parallel algorithm was shown to grow arbitrarily large in the worst case.

It is important to note here that, while the underlying cause for the phenomenon observed is the fact that the parallel algorithm is *faster* than the sequential one, the net effect (that is, the phenomenon itself) is entirely distinct from speedup. Indeed, for the MST example studied, speedup is not at all spectacular. The algorithm of [34] can update the MST of a graph with m vertices in $O(m)$ time. On the other hand, the parallel algorithm of [3] solves the same problem in $O(\log m)$ time while requiring $m-1$ processors. The speedup here is $O(m/\log m)$ and hence *sublinear* in the number of processors! It can also be pointed out that the algorithm of section 3.4 would not have succeeded in obtaining an exact solution to the minimum-weight spanning tree problem had the time interval been smaller than $\log(n+k)$, or had the number of available processors been smaller than $2n+2k-3$.

It is of course possible to update an existing MST in parallel *in constant time*; however, this requires considerably many more processors than needed by the algorithm of section 3.4. The idea is as follows. Suppose that the current MST consists of $n+k-1$ vertices (and $n+k-2$ edges). Adding to this MST a new vertex along with the $n+k-1$ edges connecting it to the previous $n+k-1$ vertices creates a new graph containing a collection of cycles. Each of these cycles consists of at most $n+k$ edges, of which exactly two are new edges. There are, therefore, $\binom{n+k-1}{2}$ cycles in all. A new MST is obtained over the new graph by removing the edge of maximum weight from each of these cycles. (Since there are only $2n + 2k - 3$ edges, some edges are 'removed more than once', so to speak.) This can be done in $O(1)$ time and at most $(n + k) \times \binom{n+k-1}{2}$ processors on the CRCW PRAM using the MAXIMUM rule for resolving write conflicts. Variants of this algorithm are described in [10, 21, 28, 29, 32, 35, 36].

Additional examples of real-time optimization problems can be easily developed along the same lines outlined in this paper. These include the problems listed in section 2.1, namely, those calling for the computation of shortest paths, maximum-sum subsequences, and minimum-weight matchings. Furthermore, for the class of NP-hard problems, several real-time approximation algorithms have been proposed (for a survey, see [19]). However, all of these algorithms are sequential, and none of the problems states either the rate at which data are to be received or the rate at which results are to be produced. Developing parallel real-time approximation algorithms for NP-hard problems, that also take into account the input and output rates, appears to be a worthwhile prospect.

Another paradigm of real-time computation occurs when *corrections* to the existing data arrive on line and must be incorporated in the solution to the problem at hand [8, 25]. Within this framework, an interesting case arises in connection with the minimum-weight spanning tree problem when corrections to the weights of the edges currently in the MST are received in real time and must be taken into consideration. Sequential and parallel algorithms for this problem are described in [10, 15, 29]. However, while these algorithms update the MST as required, their analyses (much like those of the algorithms in [19]) do not allow for the corrections to arrive, or for the results to be produced, at a certain specified rate. Here too an open avenue for research suggests itself quite naturally.

Finally, other areas beside real-time optimization need to be explored for further measures to evaluate parallel algorithms. One such area, mentioned in [1], is numerical computation: Can the accuracy of a solution to a numerical problem be increased through the use of parallelism? Candidate numerical computations that present themselves naturally here are polynomial interpolation and power series manipulation, in which the data arrive in real time.

References

1. S. G. Akl, *Parallel Computation: Models and Methods* (Prentice-Hall, Upper Saddle River, New Jersey, 1997)
2. S. G. Akl and L. Fava Lindon, Paradigms admitting superunitary behaviour in parallel computation, *Parallel Algorithms and Applications* **11** (1997) 129–153.
3. B. Awerbuch and Y. Shiloach, New connectivity and MSF algorithms for shuffle-exchange network and PRAM, *IEEE Transactions on Computers* **36** (1987) 1258–1263.
4. L. Boxer and R. Miller, Dynamic computational geometry on meshes and hypercubes, *Journal of Supercomputing* **3** (1989) 161–191.

5. G. Brassard and P. Bratley, *Algorithmics: Theory and Practice* (Prentice Hall, Englewood Cliffs, New Jersey, 1998).

6. S. D. Bruda and S. G. Akl, On the data-accumulating paradigm, *Proceedings of the Fourth International Conference on Computer Science and Informatics*, Research Triangle Park, North Carolina, October 1998, 150–153.

7. S. D. Bruda and S. G. Akl, The characterization of data-accumulating algorithms, *Proceedings of the International Parallel Processing Symposium*, San Juan, Puerto Rico, April 1999.

8. S. D. Bruda and S. G. Akl, A case study in real-time parallel computation: Correcting algorithms, Technical Report No. 98-420, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, December 1998.

9. P. Chaudhuri, Finding and updating depth first spanning trees of acyclic digraphs in parallel, *The Computer Journal* **33** (1990) 247–251.

10. P. Chaudhuri, *Parallel Algorithms: Design and Analysis* (Prentice Hall, Sydney, Australia, 1992).

11. P. Chaudhuri, Parallel incremental algorithms for analyzing activity networks, *Parallel Algorithms and Applications* **13** (1998) 153–165.

12. F.Y. Chin and D. Houck, Algorithms for updating minimum spanning trees, *Journal of Computer and System Sciences* **16** (1978) 333–344.

13. T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms* (McGraw-Hill, New York, 1990).

14. S. Even and Y. Shiloach, An on-line edge deletion problem, *Journal of the ACM* **28** (1982) 1–4.

15. G. Frederickson, Data structures for on-line updating of minimum spanning trees, *Proceedings of the ACM Symposium on Theory of Computing*, Boston, Massachusetts, April 1983, 252–257.

16. D. Harel, *Algorithmics: The Spirit of Computing* (Addison Wesley, Reading, Massachusetts, 1987).

17. J.T. Havill and W.Mao, On-line algorithms for hybrid flow shop scheduling, *Proceedings of the Fourth International Conference on Computer Science and Informatics*, Research Triangle Park, North Carolina, October 1998, 134–137.

18. T. Ibaraki and N. Katoh, On-line computation of transitive closure graphs, *Information Processing Letters* **16** (1983) 95–97.

19. S. Irani and A.R. Karlin, Online computation, in *Approximation Algorithms for NP-Hard Problems*, ed. D.S. Hochbaum (International Thomson Publishing, Boston, Massachusetts, 1997) 521–564.

20. J. JáJá, *An Introduction to Parallel Algorithms* (Addison Wesley, Reading, Massachusetts, 1992).

21. H. Jung and K. Mehlhorn, Parallel algorithms for computing maximal independent sets in trees and for updating minimum spanning trees, *Information Processing Letters* **27** (1988) 227–236.

22. D.E. Knuth, *The Art of Computer Programming*, Vol. 1, *Fundamental Algorithms* (Addison-Wesley, Reading, Massachusetts, 1975).

23. E.L. Lawler, *Combinatorial Optimization: Networks and Matroids* (Holt, Rinehart & Winston, New York, 1976).

24. F. Luccio and L. Pagli, Computing with time-varying data: Sequential complexity and parallel speed-up, *Theory of Computing Systems* **31** (1998) 5–26.

25. F. Luccio, L. Pagli, and G. Pucci, Three non conventional paradigms of parallel computation, *Lecture Notes in Computer Science* **678** (1992) 166–175.

26. F. Luccio and L. Pagli, The p-shovelers problem (computing with time-varying data), *Proceedings of the Fourth Symposium on Parallel and Distributed Computing*, Arlington, Texas, December 1992, 188–193.

27. C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity* (Prentice Hall, Englewood Cliffs, New Jersey, 1982).

28. S. Pawagi, A parallel algorithm for multiple updates of minimum spanning trees, *Proceedings of the International Conference on Parallel Processing*, St. Charles, Illinois, August 1989, Vol. III, 9–15.

29. S. Pawagi and I.V. Ramakrishnan, An $O(\log n)$ algorithm for parallel update of minimum spanning trees, *Information Processing Letters* **22** (1986) 223–229.

30. G.J.E. Rawlins, *Compared to What? An Introduction to the Analysis of Algorithms* (W.H. Freeman, New York, 1992).

31. J.H. Reif, *Synthesis of Parallel Algorithms* (Morgan Kaufmann, San Mateo, California, 1993).

32. D.D. Sherlekar, S. Pawagi, and I.V. Ramakrishnan, $O(1)$ parallel time incremental graph algorithms, *Lecture Notes in Computer Science* **206** (1985) 477–493.

33. J.R. Smith, *The Design and Analysis of Parallel Algorithms* (Oxford University Press, New York, 1993).

34. P.M. Spira and A. Pan, On finding and updating spanning trees and shortest paths, *SIAM Journal on Computing* **4** (1975) 375–380.

35. Y.H. Tsin, On handling vertex deletion in updating minimum spanning trees, *Information Processing Letters* **27** (1988) 167–168.

36. P. Varman and K. Doshi, A parallel vertex insertion algorithm for minimum spanning trees, *Lecture Notes in Computer Science* **226** (1986) 424–433.