JOURNAL OF ALGORITHMS 10, 249–270 (1989)

The Parallel Complexity of TSP Heuristics

GERARD A. P. KINDERVATER AND JAN KAREL LENSTRA

Centre for Mathematics and Computer Science, Amsterdam and Erasmus University, Rotterdam, The Netherlands

AND

DAVID B. SHMOYS

Massachusetts Institute of Technology, Cambridge, Massachusetts 02139

Received February 1, 1988; revised February 22, 1988

We consider eight heuristics for constructing approximate solutions to the traveling salesman problem and analyze their complexity in a model of parallel computation. The problems of finding a tour by the nearest neighbor, nearest merger, nearest insertion, cheapest insertion, and farthest insertion heuristics are shown to be $\mathcal{P}$-complete. Hence, it is unlikely that such tours can be obtained in polylogarithmic work space on a sequential computer or in polylogarithmic time on a computer with unbounded parallelism. The double minimum spanning tree and nearest addition heuristics can be implemented to run in polylogarithmic time on a polynomial number of processors. For the Christofides heuristic, we give a randomized polylogarithmic approximation scheme requiring a polynomial number of processors.

1. INTRODUCTION

With the advent of parallel architectures the design of algorithms has to be reconsidered. For a computer with p parallel processors, the best one can hope to achieve is to develop algorithms that run p times as fast as on a traditional sequential machine. To attain this, the intrinsic parallelism in the problem at hand has to be optimally exploited and the overhead due to the communication between processors, usually through a fixed interconnection network, has to be kept within limits. In theoretical models, an

0196-6774/89 $3.00

unbounded number of processors is available and the processors communicate with each other through a shared memory in unit time. For such a model, new complexity classes can be identified.

Within the class $\mathcal{P}$ of decision problems which are solvable in polynomial time by a *random access machine* (RAM), there exist problems that can be solved in *polylogarithmic* work space $(\log n)^{O(1)}$, i.e., work space that is polynomial in the logarithm of the problem size n; they belong to the class POLYLOGSPACE. On the other hand, $\mathcal{P}$ also contains problems that are unlikely to admit solution in polylogarithmic work space. One way to provide evidence that a problem is computationally hard in this sense is to show that it is *$\mathcal{P}$-complete with respect to log-space transformations*, i.e., it belongs to $\mathcal{P}$ and each problem in $\mathcal{P}$ is reducible to it by a transformation using logarithmic work space. If such a problem would belong to POLYLOGSPACE, then it would follow that $\mathcal{P} \subseteq$ POLYLOGSPACE, which is believed to be false. The first $\mathcal{P}$-complete problem was identified by Cook [3]. To prove that a problem P_1 in $\mathcal{P}$ is $\mathcal{P}$-complete, it is sufficient to show that every instance of a problem P_2 which is already known to be $\mathcal{P}$-complete can be mapped to an instance of P_1 such that "yes" instances of P_2 are mapped to "yes" instances of P_1 and "no" instances of P_2 to "no" instances of P_1, where the transformation requires logarithmic work space [8]. It is said that P_2 is *log-space transformable* to P_1.

These concepts have found application in the field of parallel computing. The most common model for parallel computation is the PRAM. This is a synchronous machine with an unbounded number of processors and a shared memory. It allows simultaneous reads from the same memory location but disallows simultaneous writes into the same memory location. The computation starts with one processor activated; at any step, an active processor can do a standard operation or activate another processor; and the computation stops when the initial processor halts. By a theorem due to Fortune and Wylie [7], the class of problems solvable in $F(n)^{O(1)}$ time by a PRAM is equal to the class of problems solvable in $F(n)^{O(1)}$ work space by a RAM, if $F(n) \geq \log n$. It follows that problems which can be solved in polylogarithmic work space by a RAM are solvable in polylogarithmic time on a PRAM, and that the solution in parallel polylogarithmic time of problems which have been shown to be $\mathcal{P}$-complete is very unlikely. In other terms, a dramatic speedup from polynomial to polylogarithmic time by parallelism can be expected only for those problems in $\mathcal{P}$ that belong to the class POLYLOGSPACE.

In this paper we analyze a number of heuristics for the *traveling salesman problem* (TSP) with respect to the theoretical PRAM model. Each of these heuristics can be turned into a decision problem by posing a question about the result of the algorithm, such as, for example, "does the tour obtained by starting the nearest neighbor heuristic in vertex v_1 visit vertex v_2 as the last

one before returning to vertex v_1?" or "does the tour obtained by the nearest merger algorithm contain edge $\{i, j\}$?"

We show that the nearest neighbor, nearest merger, nearest insertion, cheapest insertion, and farthest insertion problems thus obtained are $\mathcal{P}$-complete by giving log-space transformations from the circuit value problem. Hence, it is unlikely that the corresponding heuristics can be implemented to run in polylogarithmic time, even if unbounded parallelism is allowed. The double minimum spanning tree and nearest addition problems are shown to belong to the class $\mathcal{NC}$, i.e., the corresponding heuristics are implemented in polylogarithmic time on a polynomial number of processors. If the Christofides heuristic can be implemented in this way remains an open question. However, we give a family of randomized algorithms that run in polylogarithmic time on a polynomial number of processors and whose performance asymptotically approaches the performance of the Christofides heuristic.

The results presented in this paper are primarily of theoretical interest. In a practical setting, there is a fixed or perhaps linear number of processors. In such an environment, the distinction between the heuristics disappears: all the heuristics considered here have parallel implementations with almost optimal speedup.

General references on parallel combinatorial computing are by Cook [4], Valiant [19], Johnson [10], and Kindervater and Lenstra [12, 13]. Sequential TSP algorithms are dealt with by Lawler, Lenstra, Rinnooy Kan and Shmoys [15].

2. The Traveling Salesman Problem

Given a complete undirected graph and a weight for each edge, the traveling salesman problem is the problem of finding a Hamiltonian cycle (i.e., a cycle passing through each vertex exactly once) of minimum total weight. This well known $\mathcal{NP}$-hard problem has been extensively studied. There are n vertices, numbered from 1 up to n, and the weight of edge $\{i, j\}$ (the distance between vertex i and vertex j) will be denoted by d_{ij} $(i, j = 1, \ldots, n)$. The transformations that we will present in this paper are partly defined by means of figures. Edges not shown in the figures are assumed to have a weight ∞. To assure that the transformations require only logarithmic work space, we substitute $(-)cn$ for $(-)\infty$, where $c = 100$ can be seen to be sufficient.

We will discuss the implementation of a number of heuristics on a PRAM. The tours produced by the heuristics considered below are the same when a constant is added to each edge weight. If we add $10cn$ (with c the same as above) to the edge weight in the TSPs constructed by the

transformations, the resulting problems will satisfy the triangle inequality: $d_{ij} \leq d_{ik} + d_{kj}$ for all i, j, k. So, if we show that the nearest neighbor, nearest merger, nearest insertion, cheapest insertion, and farthest insertion problems are $\mathcal{P}$-complete, this is still true for the problems restricted to distance matrices that satisfy the triangle inequality.

We will now describe the heuristics in detail.

(1) *Nearest neighbor*

(i) Start at a given vertex.

(ii) Among all vertices not yet visited, choose as the next vertex the one that is closest to the current vertex. Repeat this step until all vertices have been visited.

(iii) Return to the starting vertex.

(2) *Nearest merger*

(i) Start with n partial tours, each consisting of a single city and a self-loop.

(ii) Merge the tours C_1 and C_2 for which $\min\{d_{ik} | i \in C_1, k \in C_2\}$ is as small as possible. Let $\{i, j\}$ be an edge of C_1 and $\{k, l\}$ an edge of C_2 for which $d_{ik} + d_{jl} - d_{ij} - d_{kl}$ is minimal. The merged tour is then constructed by replacing edges $\{i, j\}$ and $\{k, l\}$ by $\{i, k\}$ and $\{j, l\}$. Repeat this step until there is a complete tour.

(3) *Nearest addition*

(i) Start with a tour consisting of a given vertex and a self-loop.

(ii) Find vertices j and k with k belonging to the tour and j not for which d_{jk} is minimal, and insert j directly before k. Repeat this step until all vertices are inserted.

(4) *Nearest insertion*

(i) Start with a tour consisting of a given vertex and a self-loop.

(ii) Find a vertex not on the tour which is closest to a vertex already contained in the tour.

(iii) Insert this vertex between two neighboring vertices on the tour in the cheapest possible way. If the tour is still incomplete go to step (ii).

(5) *Cheapest insertion*

(i) Start with a tour consisting of a given vertex and a self-loop.

(i) Find a vertex not on the tour which can be inserted between two neighboring vertices on the tour in the cheapest possible way.

(iii) Insert this vertex between two neighboring vertices on the tour in the cheapest possible way. If the tour is still incomplete go to step (ii).

(6) *Farthest insertion*

(i) Start with a tour consisting of a given vertex and a self-loop.

(ii) Find a vertex not on the tour for which the minimum distance to a vertex on the tour is maximal.

(iii) Insert this vertex between two neighboring vertices on the tour in the cheapest possible way. If the tour is still incomplete go to step (ii).

The nearest, cheapest, and farthest insertion heuristics differ from each other only in the second step. They choose the next vertex to be inserted in the tour on different grounds but the actual insertion is done in the same way.

(7) *Double minimum spanning tree*

(i) Construct a minimum-weight spanning tree and double its edges.
(ii) Construct an Eulerian cycle in the graph obtained in step (i) (i.e., a cycle passing through each of its edges exactly once).
(iii) Start at a given vertex and traverse the Eulerian cycle, skipping vertices visited before.

(8) *Christofides*

(i) Construct a minimum-weight spanning tree and a minimum-weight perfect matching on the vertices of odd degree in the tree.
(ii) Construct an Eulerian cycle in the graph obtained in step (i).
(iii) Start at a given vertex and traverse the Eulerian cycle, skipping vertices visited before.

These heuristics can all be implemented to run in low-order polynomial time on a sequential computer. We should recall the results concerning their worst-case performance on TSP instances that satisfy the triangle inequality; see Chapter 5 of Lawler, Lenstra, Rinnooy Kan and Shmoys [15] for details. The nearest neighbor tour may be arbitrarily bad in comparison with the optimum. The nearest merger, nearest addition, nearest insertion, cheapest insertion, and double minimum spanning tree heuristics produce tours that are guaranteed to be less than twice as long as the optimum; the performance of the farthest insertion heuristic is unknown. The Christofides heuristic always does better than one-and-a-half times the optimum. The crucial facts in proving these bounds are that the minimum spanning tree is strictly shorter than the shortest tour, that the minimum perfect matching on any subset of vertices is no longer than half the shortest tour, and that no tour is longer than the Eulerian cycle from which it is obtained.

3. The Circuit Value Problem

A *logical circuit* is a sequence $\alpha = (\alpha_1, \ldots, \alpha_m)$, where each α_k is an input gate (having a value TRUE or FALSE) or a NAND gate ($\alpha_k = \alpha_i$ NAND $\alpha_j = \neg(\alpha_i \wedge \alpha_j)$, for some $i, j < k \leq m$). The *circuit value problem* is the problem of determining whether the last gate α_m receives the value TRUE or

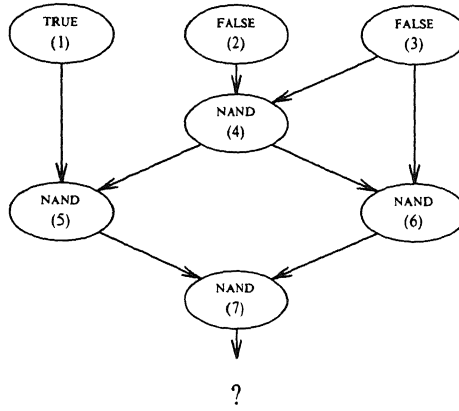


FIG. 1. A logical circuit.

FALSE, given a truth assignment to the input gates. An example is given in Fig. 1.

This problem has been shown to be $\mathscr{P}$-complete [14], even if the input gates have fan-out 1 (they appear once as input to another gate) and each NAND gate α_k $(k < m)$ has fan-out at most 2 [9].

4. NEAREST NEIGHBOR

For the nearest neighbor heuristic we define the nearest neighbor problem in the following way: given a distance matrix (d_{ij}) and two vertices v_1 and v_2, does the nearest neighbor tour starting at vertex v_1 visit vertex v_2 as the last one before returning to vertex v_1? We will show that this decision problem is $\mathscr{P}$-complete. For each instance of the circuit value problem, we construct a graph in such a way that the circuit value of the considered instance is TRUE if and only if the nearest neighbor problem returns a "yes" answer.

Each gate in the circuit is represented by a subgraph. The nearest neighbor tour will visit the subgraphs in the order in which the corresponding gates are numbered in the circuit. This ensures that if the tour visits a subgraph corresponding to a non-input gate, it has traversed the subgraphs corresponding to its input gates.

For NAND gate k $(k < m)$ with fan-out 2 $(\alpha_k = \alpha_i \text{ NAND } \alpha_j)$, we construct the subgraph as shown in Fig. 2. The vertex pairs ①—② are used to connect the different subgraphs. If gate i is input to gate k, a ①—② pair appears as output in the subgraph for gate i and also as input in the

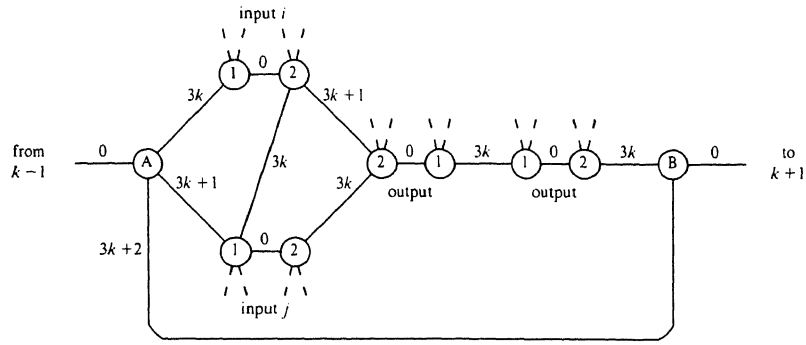


FIG. 2. The representation of NAND gate k.

subgraph for gate k. The edge weight 0 assures that corresponding vertices 1 and 2 are always neighbors in the obtained tour. If the fan-out is 1 (0), we construct the same subgraph with one arbitrary (1)—(2) pair of output vertices (without output vertices). The subgraph is constructed in such a way that if the nearest neighbor tour enters the subgraph at vertex A from subgraph $k - 1$, it leaves this subgraph through vertex B to subgraph $k + 1$. We associate a TRUE (FALSE) value with this subgraph if the nearest neighbor tour on its way from A to B passes (does not pass) through the output vertices.

When the tour arrives at vertex A from subgraph $k - 1$, there are three possibilities:

(i) Inputs i and j have both been visited already. In this case the tour must go directly to vertex B and then it will choose the edge of weight 0 to subgraph $k + 1$. This will be the only case where the output vertices are not immediately visited. Note that as a result either output vertex 2 has its corresponding vertex 1 left as its only unvisited neighbor within the subgraph. See Fig. 3a.

(ii) Either input i or input j is still unvisited. The tour will choose vertex 1 of this unvisited input as next vertex, since the edge weight is less than the distance to vertex B. From here it goes to the corresponding vertex 2 (edge weight is 0). As noted under (i), this vertex 2 has no unvisited neighbors in the subgraph where it appears as output. Therefore, the next vertex must belong to subgraph k, i.e., the tour arrives at the outputs. Because edge weights in a subgraph are proportional to the number of that subgraph and outputs belong to subgraphs with a higher number, the nearest neighbor algorithm will visit all output vertices and after that vertex B before leaving subgraph k to subgraph $k + 1$, cf. Fig. 3b and c.

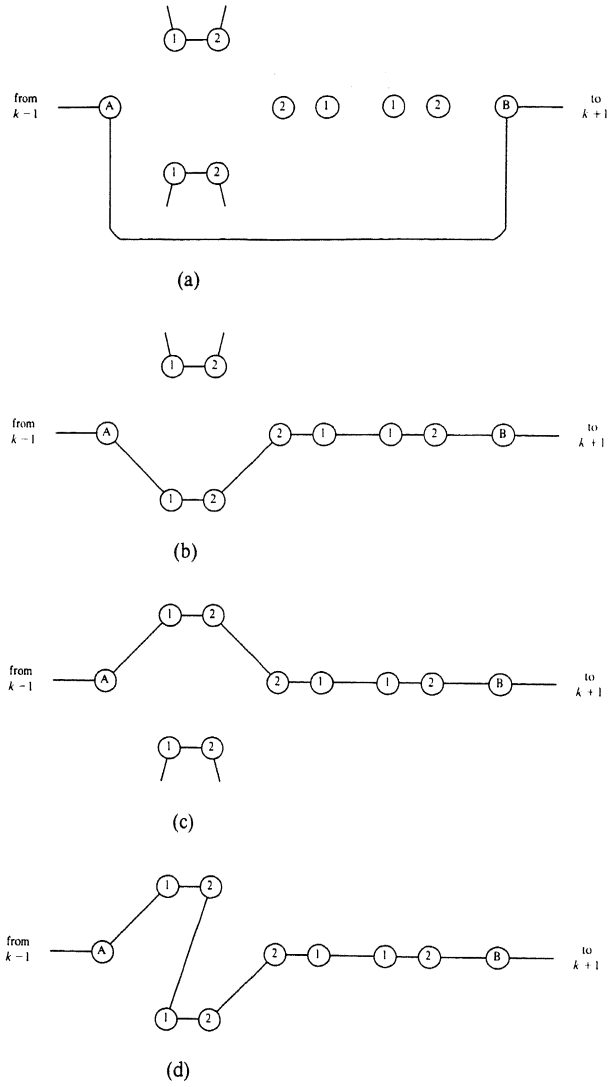


FIG. 3. The possible situations: (a) TRUE NAND TRUE → FALSE; (b) TRUE NAND FALSE → TRUE; (c) FALSE NAND TRUE → TRUE; (d) FALSE NAND FALSE → TRUE.

(iii) Both inputs are unvisited. The tour will pass through all vertices of subgraph k before going to subgraph $k + 1$ (Fig. 3d).

Note that in all cases all unvisited input vertices are included in the tour. To summarize the results, the nearest neighbor tour from A to B passes through the output vertices if and only if at least one of the input vertices is not yet visited. In the circuit value problem, this corresponds to the fact that a NAND gate produces the value TRUE if and only if at least one of the inputs is FALSE.

For TRUE and FALSE inputs we construct the subgraphs as shown in Fig. 4. The subgraph corresponding to input 1 is a special case. Instead of the edge of length 0, it has two edges of length $3m + 3$ which connect it to the subgraph corresponding to NAND gate m. The representation of this last gate has a somewhat special structure. The output vertices are replaced by a vertex C. Both vertices B and C are connected to input 1 (see Fig. 5). If the tour arrives at vertex A of this gate and we are in situation (i), the tour will go directly to vertex B and from there to vertex C before it leaves subgraph m. Otherwise vertex B will be the last vertex to be visited of this last subgraph.

It should now be clear that a nearest neighbor tour starting at the A-vertex of input 1 visits the B-vertex of the last gate as the last vertex if and only if the circuit computes the value TRUE. Since the transformation can be performed using work space which is logarithmic in the size of the circuit, the nearest neighbor problem is $\mathscr{P}$-complete. So, the construction of

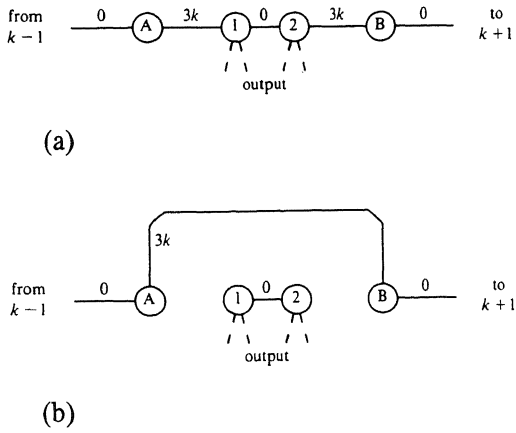


FIG. 4. The representation of input k: (a) the representation of a TRUE input; (b) the representation of a FALSE input.

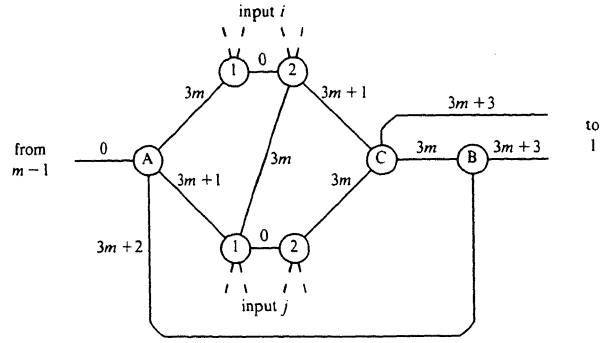


FIG. 5. The representation of NAND gate m.

a nearest neighbor traveling salesman tour will probabily require superpolylogarithmic work space or superpolylogarithmic parallel time.

5. NEAREST MERGER

Given a distance matrix and an edge $\{i, j\}$, the nearest merger problem is the problem of deciding whether the tour produced by the nearest merger heuristic contains $\{i, j\}$. We will show that this problem is $\mathcal{P}$-complete by giving a transformation from the circuit value problem.

Consider an instance of the circuit value problem. For each arc, we construct a graph as shown in Fig. 6. Gates with fan-out 0 (for example, the last gate) are assumed to have an arc from itself to a dummy vertex. The dashed edges have a weight greater than 0 and will be described later. The nearest merger heurstic first builds the tours A–D–A and B–C–B and then merges them to form the tour A–B–C–D–A.

We also construct the graph of Fig. 7, where m is the number of gates of the circuit. The algorithm starts by constructing partial tours of the form $(2k) - (2k + 1) - (2k)$, for $k = 1, \ldots, m$. The edge weights

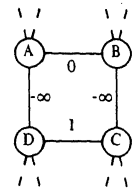


FIG. 6. The representation of an arc.

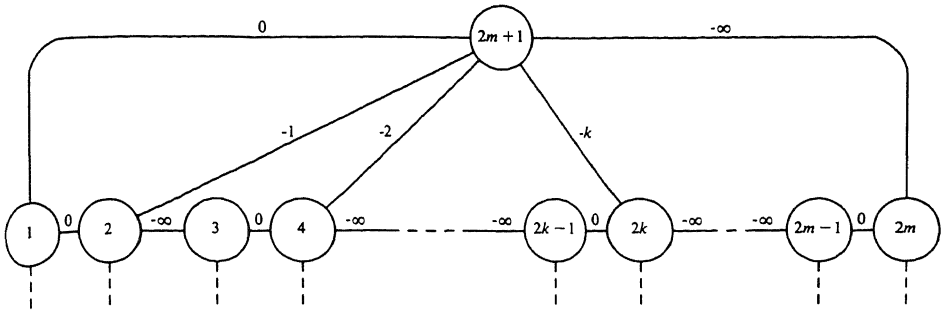
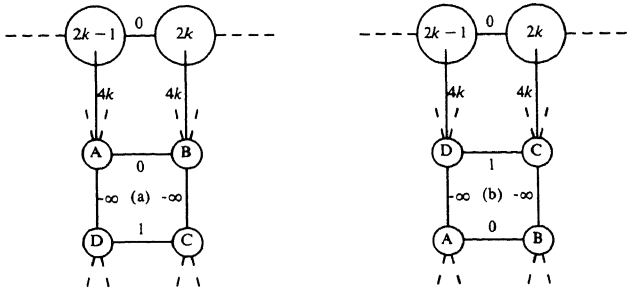


FIG. 7. The extra graph.

$-(m-1), \ldots, -1, 0$ assure that the original tour $(2m) - (2m+1) - (2m)$ is merged with the other cycles of weight 2 and finally with the self-loop $(1) - (1)$. the result is the tour $(2m+1) - (1) - (2) - \cdots - (2k-1) - (2k) - \cdots - (2m-1) - (2m) - (2m+1)$.

We will now describe how the graphs are connected. The edge weights are chosen such that the nearest merger heuristic will merge the cycles of Fig. 6 with the (extended) tour of the extra graph of Fig. 7 in the order of the numbers of the gates where the corresponding arcs begin. If a cycle of Fig. 6 is added, an edge of weight 0 or 1 will remain in the tour. At this point, we associate a value TRUE (FALSE) with the arc if the edge of weight 1 (0) still belongs to the tour.

Each cycle corresponding to an arc from an input gate is connected to the extra graph as shown in Fig. 8. If the input is TRUE (FALSE), the edges



(a) (b)

FIG. 8. The representation of the input gates; (a) input gate k is TRUE; (b) input gate k is FALSE.

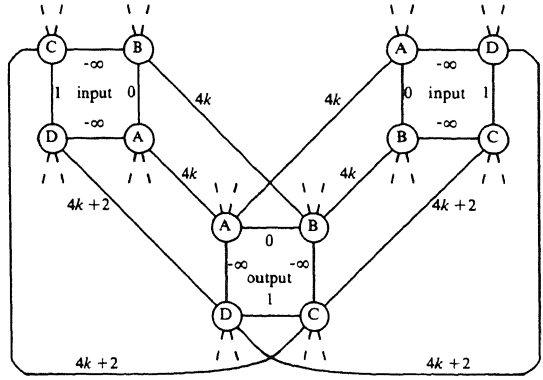


FIG. 9. The representation of NAND gate k (fan-out 1).

$\{A, B\}$ and $\{2k-1, 2k\}$ ($\{C, D\}$ and $\{2k-1, 2k\}$) are replaced by $\{2k-1, A\}$ and $\{2k, B\}$ ($\{2k-1, D\}$ and $\{2k, C\}$).

For NAND gate k with fan-out 1, the subgraphs are connected as shown in Fig. 9. Let us assume that there are no edges in the tour connecting the two inputs. We consider the case where one input (left) has the associated value TRUE and the other one the value FALSE in detail (see Fig. 10). There are two candidates for the merge operation: replace the edge $\{C, D\}$ of the left input and the edge $\{C, D\}$ of the output by the edges between the C and D vertices, or replace the edge $\{A, B\}$ of the right input and the edge $\{A, B\}$ of the output by the edges between the A and B vertices. The last replacement will be chosen, since it is cheaper.

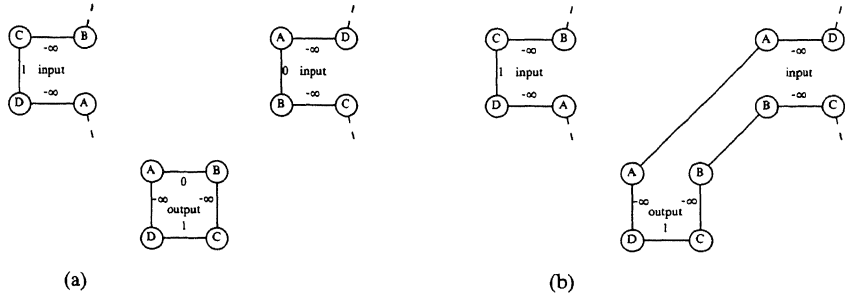


FIG. 10. TRUE AND FALSE → TRUE: (a) situation before merging; (b) situation after merging.

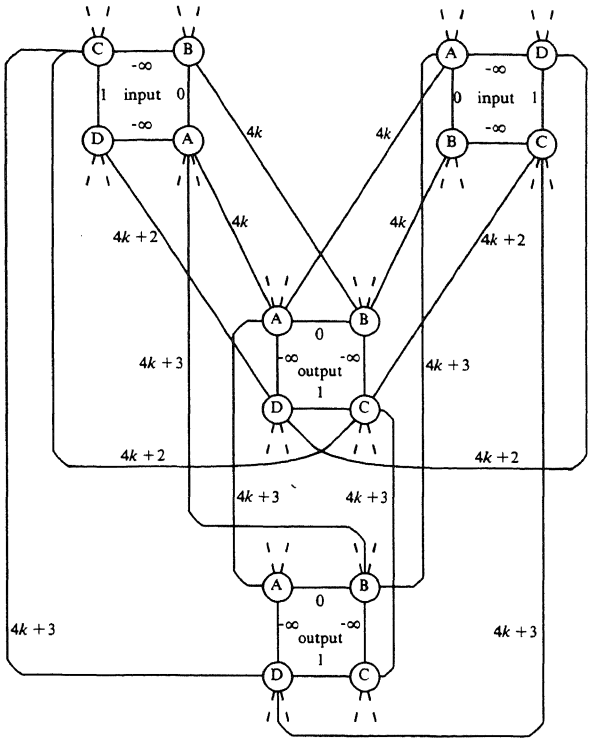


FIG. 11. The representation of NAND gate k (fan-out2).

If NAND gate k has fan-out 2, we connect the subgraphs as shown in Fig. 11. The case TRUE NAND FALSE → TRUE is illustrated in Fig. 12.

The other cases are left as exercises to the reader. Note that the output vertices of a subgraph are always inserted between two edges of weight $-\infty$ of one of the inputs.

So far, we have assumed that there are no edges in the tour connecting both inputs of the same gate. Because of the way that output vertices are inserted in the tour, connecting edges can only occur when the inputs are outputs from the same gate. These edges stretch between vertices with the same label (A or C). It is, however, impossible to remove them from the tour at low cost. Therefore, the same replacements will be made as in the case where there are no interconnecting edges between the inputs.

The above arguments imply that the circuit value of an instance of the circuit value problem is TRUE if and only if the nearest merger heuristic

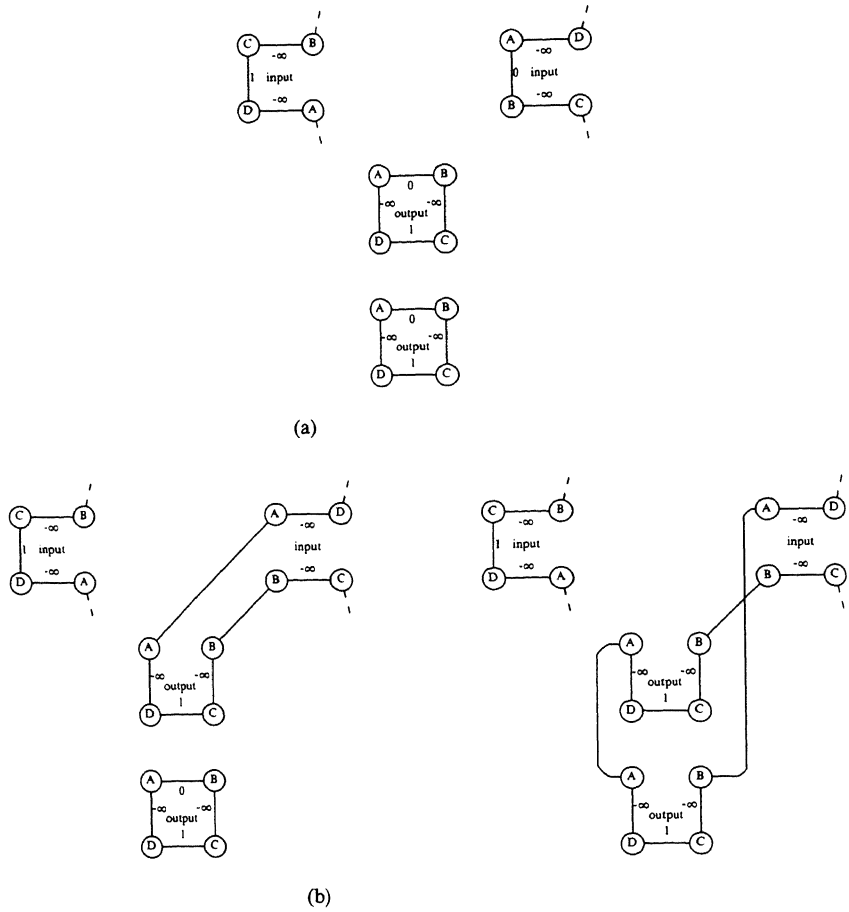


FIG. 12. TRUE NAND FALSE → TRUE: (a) situation before merging; (b) the two merging steps.

produces a tour which contains edge $\{C, D\}$ of the subgraph corresponding to the arc starting from the last gate of the circuit. The transformation can be done in logarithmic work space. Hence, the nearest merger problem is $\mathcal{P}$-complete.

6. NEAREST INSERTION, CHEAPEST INSERTION, AND FARTHEST INSERTION

Given a distance matrix, a starting vertex, and an edge $\{i, j\}$, the nearest insertion (cheapest inserton, farthest insertion) problem is the problem of

eciding whether the nearest insertion (cheapest insertion, farthest inser-on) heuristic produces a tour which contains edge $\{i, j\}$. The transforma-ons from the circuit value problem showing that these problems are $\mathcal{P}$-complete are similar to the one for the nearest merger problem. We will ive only the crucial part of the transformation, leaving the details and the erification to the reader.

'earest Insertion

For NAND gate k with fan-out 0 or 1 to be simulated, we construct the raph of Fig. 13. The output vertices are inserted in the tour in the order A,), B, and C between vertices A and B or B and C of one of the inputs. The epresentation if the fan-out is 2 is straightforward and not given here edges between both outputs get a weight -1). Representing the inputs to ne circuits and starting up the algorithm is similar to the nearest merger ase and also straightforward. The result of the nearest insertion algorithm a tour which contains the edge of weight 1 of the output arc of gate m if nd only if the circuit produces the value true.

The transformation requires only logarithmic work space and hence the earest insertion problem is $\mathcal{P}$-complete.

'heapest Insertion

The same transformation as described for the nearest insertion problem orks for the cheapest insertion problem, because in each step both

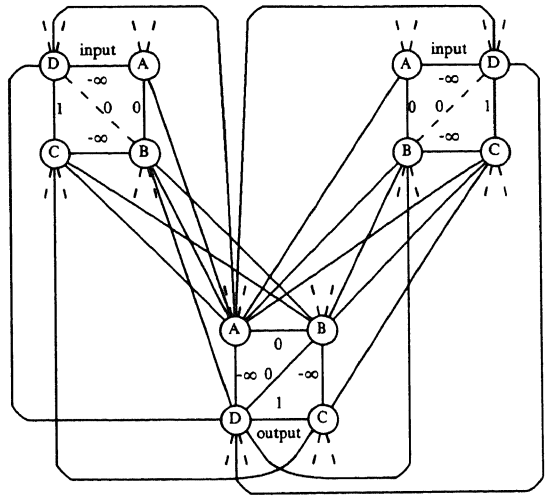


FIG. 13. The representation of NAND gate k. The edges from output vertex A to the ertices A and B of both inputs have weight $2k$; all other edges between input and output ertices have weight $2k + 1$.

algorithms will choose the same vertex to be inserted in the graphs simulating the logical circuit. So, the cheapest insertion problem is $\mathscr{P}$-complete as well.

Farthest Insertion

We can use almost the same transformation as in the previous two cases. We construct the same graph as before. Let s_i be the number of the step in which vertex i would be included in the tour by the nearest insertion heuristic; this number is known before the heuristic is actually executed. We replace the starting vertex by starting vertices v_1, v_2, v_3, v_4, and v_5, such that the farthest insertion heuristic, when started in v_1, first builds the tour $v_1 - v_2 - v_3 - v_4 - v_5 - v_1$. Edges originally incident to the starting vertex are made incident to v_3 and v_4. In this transformation we have to fill in a value for $(-)\infty$ explicitly; we take $(-)100n$. We add edges from all non-starting vertices i to v_1 of weight $-100n - s_i$, and to v_2 and v_5 of weight $200n$. The edges to v_1 have the smallest weights and determine the order in which the vertices are added to the tour. The edges to v_2 and v_5 prohibit the exclusion of $\{v_1, v_2\}$ and $\{v_1, v_5\}$ from the tour. Now, the farthest insertion heuristic will add the vertices in the same order and in the same way to the tour as the nearest insertion heuristic. Herewith, the farthest insertion problem is $\mathscr{P}$-complete.

7. Double Minimum Spanning Tree

In Sections 7, 8, and 9 we will consider implementations of the double minimum spanning tree, Christofides, and nearest addition heuristics that require polylogarithmic time on a PRAM with a polynomial number of processors. We will use the notation

$$\textbf{par}[B(i)]\,S(i)$$

to denote that the statement $S(i)$ is to be executed in parallel for all values of i satisfying the condition $B(i)$.

The double minimum spanning tree heuristic consists of three phases; see Section 2. Phase (i) (constructing a minimum spanning tree and doubling its edges) can be performed in $O(\log^2 n)$ time with $O(n^2/\log^2 n)$ processors [2]. Phase (ii) (finding an Eulerian cycle) can be done within the same time and processor bounds using the techniques from Awerbuch, Israeli, and Shiloach [1]. For phase (iii), we first have to find the first occurrence of each vertex and then eliminate all duplications. Let $v_1, \ldots, v_i, \ldots, v_{2n-1}$ denote the Eulerian cycle obtained in the previous phase, where v_i is the ith

vertex of the tour. We proceed as follows.

$$\textbf{par}[1 \le i, j \le 2n - 1] \textbf{ if } v_i = v_j \textbf{ then } c_{ij} \leftarrow 1 \textbf{ else } c_{ij} \leftarrow 0.$$

$$\textbf{par}[1 \le i \le 2n - 1]\delta_i \leftarrow \max\{0, 1 - \sum_{j=1}^{i-1} c_{ij}\}.$$

$$\textbf{par}[1 \le i \le 2n - 1]\, p_i \leftarrow \sum_{j=1}^{i} \delta_j.$$

Note that $\delta_i = 1$ if v_i occurs for the first time in the tour, $\delta_i = 0$ otherwise, and that p_i denotes the number of different vertices in $v_1, \ldots, v_i$. We obtain the tour $t_1 - t_2 - \cdots - t_n - t_1$ by:

$$\textbf{par}[1 \le i \le 2n - 1] \textbf{ if } \delta_i = 1 \textbf{ then } t_{p_i} \leftarrow v_i.$$

Using the partial sums algorithm from Dekel and Sahni [6], we can implement phase (iii) within the same resource bounds as the previous phases. So, we end up with an algorithm that runs in $O(\log^2 n)$ time on $O(n^2/\log^2 n)$ processors, which is best possible with respect to the $O(n^2)$ sequential implementation.

8. Christofides

The Christofides heuristic also consists of three phases. Since the second and third phases are identical to the corresponding phases of the double minimum spanning tree heuristic given above, we need only focus on implementing the first phase.

Unfortunately, it is an open question if the minimum perfect matching problem belongs to $\mathcal{NC}$. Karp, Upfal, and Wigderson [11] developed a *randomized* algorithm for the problem, which produces the correct answer with probability greater than 0.5. However, it runs in polylogarithmic time on a polynomial number of processors only if the edge weights are specified in *unary*. A more efficient algorithm of the same type is due to Mulmuley, Vazirani, and Vazirani [16]. It runs in time $O(\log^2 n)$ on $n^3 d_{\max} P(n)$ processors, where $d_{\max}$ is the maximum edge weight and $P(n)$ is the time needed to multiply two $n \times n$ matrices on a RAM. ($P(n) < n^{2.376}$; see Coppersmith and Winograd [5].)

We give an *approximation scheme* for the Christofides heuristic, i.e., a family of algorithms that asymptotically approach its performance. More precisely, for each $\varepsilon > 0$ we give a randomized algorithm that runs in polylogarithmic time on a polynomial number of processors and, if the

distances satisfy the triangle inequality, delivers a tour of length less than $(3/2 + \varepsilon)$ times the shortest tour length; the running time is independent of ε and the number of processors is polynomial in $(1/\varepsilon)$. The approach is based on the idea that an approximate minimum perfect matching will suffice to obtain an approximate Christofides tour and that an approximate minimum perfect matching can be obtained by solving a matching problem with weights bounded by a polynomial in n. It will be useful to let $d(G) = \sum_{\{i,j\} \in E} d_{ij}$ for any graph $G = (V, E)$ and weight function d.

For the first phase of the heuristic we construct two Eulerian graphs and select the one of smallest total weight. The first of these graphs is a double minimum spanning tree. For the second we proceed as follows:

(i) Find a minimum spanning tree T and identify the set V of vertices of odd degree in T.

(ii) Set $\mu = 2\varepsilon d(T)/|V|$ and $E = \{\{i, j\} \subseteq V | d_{ij} \leq 2d(T)/3\}$. For all $\{i, j\} \in E$, set $\tilde{d}_{ij} = \lfloor d_{ij}/\mu \rfloor$.

(iii) Find a minimum perfect matching $\tilde{M}$ on $G(V, E)$ with edge weights $\tilde{d}$ and add these edges to the minimum spanning tree.

We first show that this procedure has the claimed performance guarantee. To do this we show that one of the Eulerian graphs produced has total weight less than $(3/2 + \varepsilon)d(C)$, where C is a shortest tour. Let M denote a minimum perfect matching on V with edge weights d_{ij}. If $d(T) \leq 3d(M)/2$, then the double minimum spanning tree has weight at most $2d(T) \leq 3d(M) \leq (3/2)d(C)$.

Now assume that $d(T) > 3d(M)/2$. Note that for each $\{i, j\} \notin E$, $d_{ij} > 2d(T)/3 > d(M)$, so that $M \subseteq E$. Since $\mu\tilde{d}_{ij} \leq d_{ij} \leq \mu\tilde{d}_{ij} + \mu$ for $\{i, j\} \in E$, we have

$$\begin{aligned} d(\tilde{M}) &\leq \sum_{\{i,j\} \in \tilde{M}} \left(\mu\tilde{d}_{ij} + \mu\right) \\ &= \mu\tilde{d}(\tilde{M}) + \mu|V|/2 \leq \mu\tilde{d}(M) + \varepsilon d(T) \\ &\leq d(M) + \varepsilon d(T) \leq (1/2 + \varepsilon)d(C), \end{aligned}$$

and hence $d(T) + d(\tilde{M}) < (3/2 + \varepsilon)d(C)$.

As to the resource bounds, $O(\log^2 n)$ time and $(n^2 \log^2(1/\varepsilon))/\log^2 n$ processors suffice for all of the computations except for finding the minimum perfect matching. This subroutine requires $O(\log^2 n)$ time and $n^3 d_{\max} P(n)$ processors [16]. By observing that

$$\max_{\{i,j\} \in V} \tilde{d}_{ij} \leq \left\lfloor \frac{2d(T)/3}{2\varepsilon d(T)/|V|} \right\rfloor = \left\lfloor \frac{|V|}{3\varepsilon} \right\rfloor = O(n/\varepsilon),$$

we conclude that the number of processors required is $O(n^4 P(n)/\varepsilon)$.

9. Nearest Addition

Let v_1 be the given starting vertex. The order in which the nearest addition heuristic adds vertices to the tour corresponds to the way in which the algorithm from Prim [17] builds up a minimum spanning tree, starting from v_1. Therefore, we first construct a minimum spanning tree and direct its edges towards v_1. By means of this tree, we can determine for each two vertices i and j which one will be visited first. There are two possible situations (Fig. 14). In the first situation, one vertex is a descendant of the other. Since each vertex is inserted immediately before its parent, the descendant will appear earlier in the tour than the ancestor. In the second situation, no vertex is a descendant of the other. Let k be the first common ancestor of i and j and let i' (j') be the last vertex on the path from i (j) to k; $i' = i$ ($j' = j$) if the path consists of only one arc. If $d_{i'k} < d_{j'k}$, then vertex i' will be inserted before vertex k, and after that vertex j' will be inserted in the tour immediately before vertex k and thus after vertex i'.

A detailed description of the algorithm is given below. It has a running time of $O(\log^2 n)$ on $O(n^2/\log^2 n)$ processors. Without loss of generality we assume that all distances are distinct.

(i) First, we construct a minimum spanning tree and direct it towards vertex v_1, generating arcs $(i, t(i))$ for $i \in \{1, \ldots, n\} \setminus v_1$. For convenience, we assume $t(v_1) = v_1$. This requires $O(\log^2 n)$ time and $O(n^2/\log^2 n)$ processors [2, 18].

(ii) The next step is to construct an $n \times n$ 0–1 matrix (c_{ij}), representing the transitive closure of the tree ($c_{ij} = 1$ if there exists a path from vertex i to vertex j, $c_{ij} = 0$ otherwise). Let $u(i, l)$ denote the vertex at distance 2^l from vertex i, or v_1 if this vertex does not exist. The following

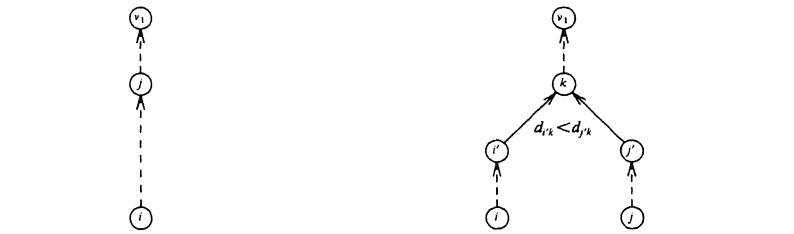


FIG. 14. The two possible situations.

statements do the job:

par$[1 \le i \le n]u(i,0) \leftarrow t(i)$.
for $l \leftarrow 1$ **to** $\lceil \log n \rceil$ **do**
    par$[1 \le i \le n]u(i,l) \leftarrow u(u(i,l-1),l-1)$.
par$[1 \le i, j \le n]c_{ij} \leftarrow 0$.
par$[1 \le i \le n]c_{ii} \leftarrow 1$.
for $l \leftarrow \lceil \log n \rceil$ **downto** 0 **do**
    par$[1 \le i, j \le n]$ **if** $c_{ij} = 1$ **then if** $c_{iu(j,l)} = 0$ **then** $c_{iu(j,l)} = 1$.

(The "**if** $c_{iu(j,l)} = 0$" condition is added to avoid simultaneous writes into c_{iv_1}.) These operations can be performed in logarithmic time with $O(n^2)$ processors. To reduce the number of processors, we have to observe that in each iteration of the last **for** l loop we only have to look at those pairs (i, j) for which $c_{ij} = 1$. The number of these pairs doubles in each iteration. Therefore, we perform the last iterations of the **for** loop in a different way. We replace the computation of the c-matrix by the following, where the parameter x will be chosen later:

for $l \leftarrow 1$ **to** x **do**
    par$[1 \le i \le n, (l-1)\lfloor \frac{n}{x} \rfloor + 1 \le j \le l\lfloor \frac{n}{x} \rfloor]c_{ij} \leftarrow 0$.
par$[1 \le i \le n]c_{ii} \leftarrow 1$ & assign a processor to (i, i).
for $l \leftarrow \lceil \log n \rceil$ **downto** $\lfloor \log x \rfloor$ **do**
    par$[1 \le i, j \le n$ & (i, j) has a processor assigned to it]
        if $c_{iu(j,l)} = 0$ **then** $c_{iu(j,l)} = 1$ & assign a free processor to $(i, u(j, l))$.
for $l \leftarrow x$ **down to** 0 **do**
    par$[1 \le i, j \le n$ & (i, j) has a processor assigned to it] **if** $c_{iu(j,l)} = 0$ **then**
        $c_{iu(j,0)} = 1$ & assign the processor, assigned to (i, j) to $(i, u(j, 0))$.

By choosing $x = \lceil \log^2 n \rceil$, we achieve a running time of $O(\log^2 n)$ with only $O(n^2/\log^2 n)$ processors.

(iii) Now, we compute the total number s_i of vertices in the subtree rooted by i:

$$\mathbf{par}[1 \le i \le n]s_i \leftarrow \sum_j c_{ji}.$$

Let r_i denote the number of descendants of the parent of vertex i which will be visited after vertex i in the tour:

$$\mathbf{par}[1 \le i \le n]r_i \leftarrow \sum_j \{s_j | (t(i) = t(j)) \,\&\, (d_{it(i)} < d_{jt(j)})\}; \; r_{v_1} \leftarrow 0.$$

Finally, we compute for each vertex i the total number q_i of vertices visited after i:

$$\mathbf{par}[1 \le i \le n]q_i \leftarrow \sum_j c_{ij}(1 + r_j)$$

(if $c_{ij} = 1$, then 1 for j and r_j for the descendants of the parent of j), and a nearest addition traveling salesman tour has been determined. These last steps require the same time and processor bounds as the previous ones.

10. Conclusion

We have shown that five simple traveling salesman heuristics are sequential in nature, to the extent that they are likely to require superpolylogarithmic time irrespective of the amount of parallelism allowed. We have also shown that the double minimum spanning tree and the nearest addition heuristics can be implemented to run in polylogarithmic time on a polynomial number of processors. Such a result cannot be obtained for the Christofides heuristic as long as the parallel complexity of the matching problem is unresolved. However, we have given a family of randomized polylogarithmic algorithms that asymptotically approach the performance of the Christofides heuristic.

Each of the eight heuristics considered above constructs a single traveling tour. Other heuristics start from a given tour and perform a series of improvements, using neighborhood search, until a local optimum is obtained. An obvious research question is what an analysis as pursued in this paper can tell us about the complexity of such iterative improvement methods.

References

1. B. Awerbuch, A. Israeli, and Y. Shiloach, Finding Euler circuits in logarithmic parallel time, *in* "Proceedings, 16th Annual ACM Symp. Theory of Computing, 1984," pp. 249–257.
2. F. Y. Chin, J. Lam, and I-N. Chen, Efficient parallel algorithms for some graph problems, *Comm. ACM* **25** (1982), 659–665.
3. S. A. Cook, An observation on time-storage trade-off, *J. Comput. System Sci.* **9** (1974), 308–316.
4. S. A. Cook, Towards a complexity theory of synchronous parallel computation, *Enseign. Math. (2)* **27** (1981), 99–124.
5. D. Coppersmith and S. Winograd, Matrix multiplication via arithmetic progressions, *in* "Proceedings, 19th Annual ACM Symp. Theory of Computing, 1987," pp. 1–6.
6. E. Dekel and S. Sahni, Binary trees and parallel scheduling algorithms, *IEEE Trans. Comput.* **C-32** (1983), 307–315.
7. S. Fortune and J. Wyllie, Parallelism in random access machines, *in* "Proceedings, 10th Annual ACM Symp. Theory Computing, 1978," p. 114–118.
8. M. R. Garey and D. S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness," Freeman, San Francisco, 1979.
9. L. M. Goldschlager, R. A. Shaw, and J. Staples, The maximum flow problem is log space complete for P, *Theoret. Comput. Sci.* **21** (1982), 105–111.

10. D. S. JOHNSON, The NP-completeness column: An ongoing guide; seventh edition, *J. Algorithms* **4** (1983), 189–203.
11. R. M. KARP, E. UPFAL, and A. Widgderson (1986). Constructing a perfect matching is in random NC, *Combinatorica* **6** (1986), 35–48.
12. G. A. P. KINDERVATER AND J. K. LENSTRA, Parallel algorithms, *in* "Combinatorial Optimization: Annotated Bibliographies" (M. O'hEigeartaigh, J. K. Lenstra, and A. H. G. Rinnooy Kan, Eds.), pp. 106–128, Wiley, Chichester, 1985.
13. G. A. P. KINDERVATER AND J. K. LENSTRA, An introduction to parallelism in combinatorial optimization, *Discrete Appl. Math.* **14** (1986), 135–156.
14. R. E. LADNER, The circuit value problem is log space complete for *P*, *SIGACT News* **7** No. 1 (1975), 18–20.
15. E. L. LAWLER, J. K. LENSTRA, A. H. G. RINNOOY KAN, AND D. B. SHMOYS (Eds.), "The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization," Wiley, Chichester, 1985.
16. K. MULMULEY, U. V. VAZIRANI, AND V. V. VAZIRANI, Matching is as easy as matrix inversion, *in* "Proceedings, 19th Annual ACM Symp. Theory of Computing, 1987" pp. 345–354.
17. R. C. PRIM, Shortest connection networks and some generalizations, *Bell System Tech. J.* **36** (1957), 1389–1401.
18. C. SAVÁGE, "Parallel Algorithms for Graph Theoretical Problems," Ph.D. Thesis, University of Illinois, Urbana-Champaign, 1977.
19. L. G. VALIANT, Parallel computation, *in* "Foundations of Computer Science IV, Distributed Systems: Part 1, Algorithms and Complexity" (J. W. de Bakker and J. van Leeuwen Eds.), Mathematical Centre Tract 158, pp. 35–48, Centre for Mathematics and Computer Science, Amsterdam, 1983.