Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

G.A.P. Kindervater, J.K. Lenstra

The parallel complexity of TSP heuristics

Department of Operations Research and System Theory   Report OS-R8609   September

90C27, 69F21, 69G11

The Parallel Complexity of TSP Heuristics

G.A.P. Kindervater, J.K. Lenstra
Centre for Mathematics and Computer Science, Amsterdam

The problems of finding an (approximate) traveling salesman tour by the nearest neighbor, nearest merger, nearest insertion, cheapest insertion and farthest insertion heuristics are log space complete for $\mathcal{P}$. We show this by giving log space transformations from the circuit value problem. Hence, it is unlikely that such tours can be obtained in polylogarithmic work space on a sequential computer or, by the parallel computation thesis, in polylogarithmic time on a computer with unbounded parallelism. We also show that the double minimum spanning tree and nearest addition heuristics can be implemented in polylogarithmic time on a polynomial number of processors.

1980 Mathematics Subject Classification: 90C27, 68Q15, 68Q25, 68R05.
Key Words & Phrases: parallel computing, computational complexity, polylog parallel algorithm, log space completeness for $\mathcal{P}$, traveling salesman problem, heuristics.
Note: This paper has been submitted for publication.

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 they do on a traditional sequential machine. To realize 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 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. These are the problems that have been shown to be *log space complete for* $\mathcal{P}$, i.e., they belong to $\mathcal{P}$ and each problem in $\mathcal{P}$ is reducible to them by a transformation using logarithmic work space. If any such problem would belong to POLYLOGSPACE, then it would follow that $\mathcal{P} \subseteq$ POLYLOGSPACE, which is not believed to be true. The first log space complete problem in $\mathcal{P}$ was identified by Cook [Cook 1974]. To prove that a problem P_1 in $\mathcal{P}$ is log space complete for $\mathcal{P}$, it is sufficient to show that every instance of a problem P_2 which is already known to be log space complete for $\mathcal{P}$ can be mapped on an instance of P_1 such that 'yes' instances of P_2 are mapped on 'yes' instances of P_1 and 'no' instances of P_2 on 'no' instances of P_1, where the transformation requires logarithmic work space [Garey & Johnson 1979]. 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 computations is the PRAM. This is a synchronized 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 the *parallel computation thesis*,

Report OS-R8609
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

which is generally believed to be true, 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 [Chandra, Kozen & Stockmeyer 1981; Goldschlager 1982]. 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 log space complete for $\mathcal{P}$ 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 log space complete for $\mathcal{P}$ by giving a log space transformation 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. The status of Christofides' heuristic remains unresolved.

General references on parallel combinatorial computing are [Cook 1981; Valiant 1983; Johnson 1983; Kindervater & Lenstra 1985, 1986]. Sequential TSP algorithms are dealt with in [Lawler, Lenstra, Rinnooy Kan & Shmoys 1985].

2. The Traveling Salesman Problem

Given a complete undirected graph and a length 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 length. This well known $\mathcal{NP}$-hard problem has been extensively studied. There are n vertices, numbered from 1 up to n, and the length of edge $\{i,j\}$ (the distance between vertex i and vertex j) will be denoted by d_{ij} $(i,j = 1,...,n)$. 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 length ∞. 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 length. If we add $10cn$ (with c the same as above) to the edge lengths in the TSP's 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 log space complete for $\mathcal{P}$, this is still true for the problems restricted to distance matrices that satisfy the triangle inequality.

In the rest of this section we will 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 T_1 and T_2 for which $\min\{d_{ik} \mid i \in T_1, k \in T_2\}$ is as small as possible. Let $\{i,j\}$ be an edge of T_1 and $\{k,l\}$ an edge of T_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 one tour left.

(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.

(*ii*) 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 only in the second step from each other. 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 spanning tree.

(*ii*) Double the edges of the minimum spanning tree and construct an Euler circuit.

(*iii*) Start at an given vertex and traverse the edges, skipping vertices visited before.

(8) *Christofides*

(*i*) Construct a minimum spanning tree and a minimum perfect matching on the vertices of odd degree in the tree.

(*ii*) Construct an Euler circuit through the edges of the tree and the matching.

(*iii*) Start at a given vertex and traverse the edges, skipping vertices visited before.

3. The Circuit Value Problem

A *logical circuit* is a sequence $\alpha = (\alpha_1,...,\alpha_m)$, 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 FALSE, given a truth assignment to the input gates. An example is given in Figure 1.

This problem has been shown to be log space complete for $\mathcal{P}$ [Ladner 1975], even if the input gates have fan out one (they appear once as input to another gate) and each NAND gate α_k ($k<m$) has fan out at most two [Goldschlager, Shaw & Staples 1982].

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 log space complete for $\mathcal{P}$. 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

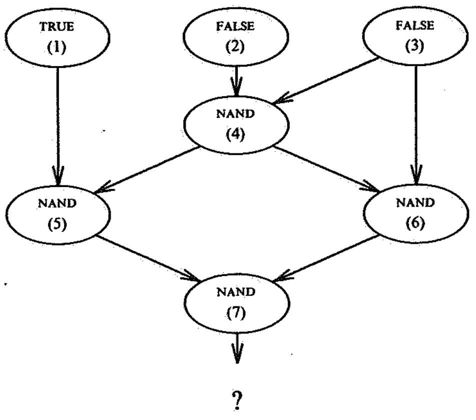


FIGURE 1. A logical circuit.

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 passed the subgraphs corresponding to its input gates.

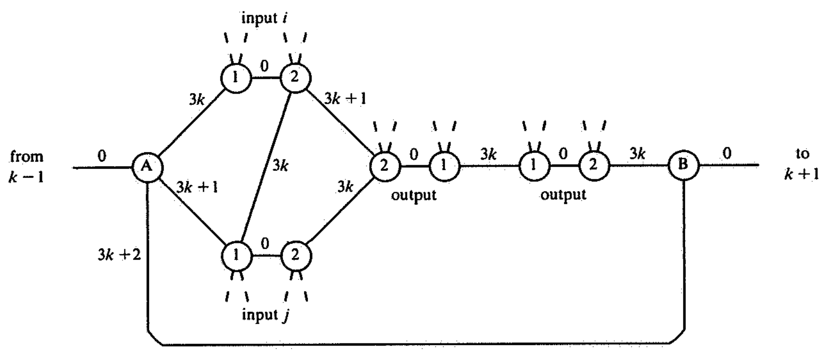


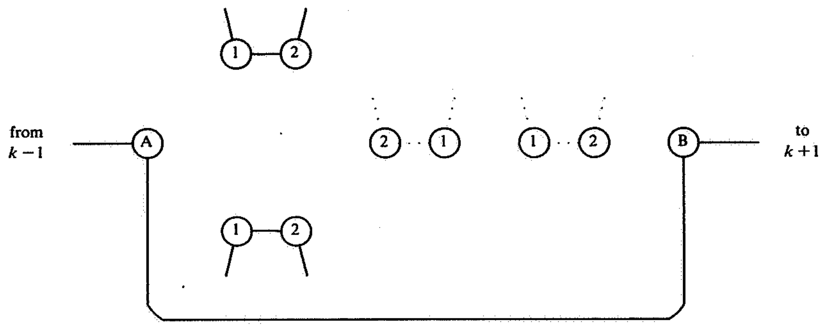
FIGURE 2. The representation of NAND gate k.

For NAND gate k ($k<m$) with fan out two ($\alpha_k = \alpha_i$ NAND α_j), we construct the subgraph as shown in Figure 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 subgraph for gate k. The edge length zero assures that corresponding vertices 1 and 2 are always neighbors in the obtained tour. If the fan out is one (zero), 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.

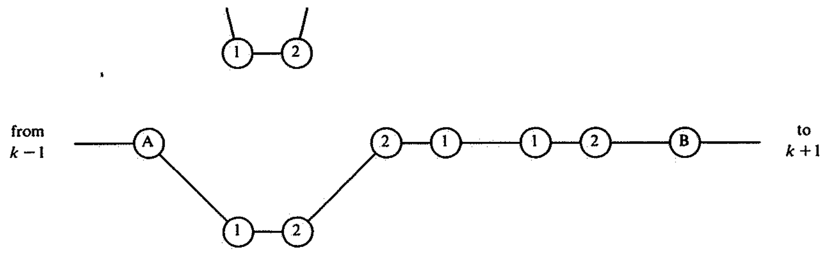
When the tour arrives at vertex A from subgraph $k-1$, there are three possibilities.

(*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 length zero 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 Figure 3a.

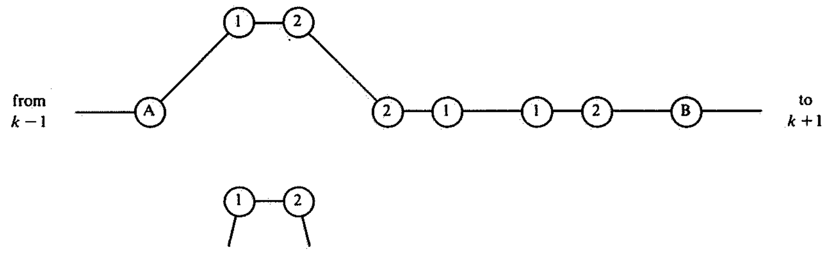
(*ii*) Either input i or input j is still unvisited. The tour will choose vertex 1 of this unvisited input as next vertex, since the edge length is less than the distance to vertex B. From here it goes to the corresponding vertex 2 (edge length is zero). As noted under (*i*), this vertex 2 has no unvisited neighbors in the subgraph where it appears as output. Therefore, the next vertex must belong to



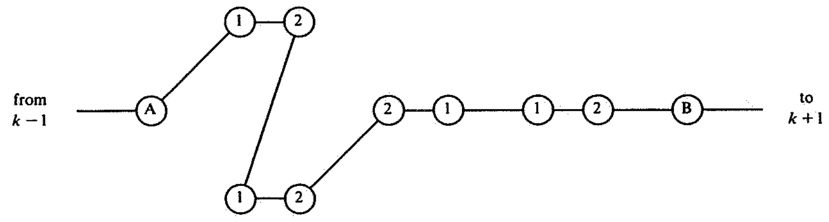
(a) TRUE NAND TRUE $\rightarrow$ FALSE



(b) TRUE NAND FALSE $\rightarrow$ TRUE



(c) FALSE NAND TRUE $\rightarrow$ TRUE



(d) FALSE NAND FALSE $\rightarrow$ TRUE

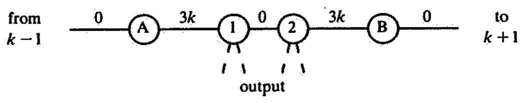
FIGURE 3. The possible situations.

subgraph k, i.e., the tour arrives at the outputs. Because edge lengths 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. Figures 3b and 3c.

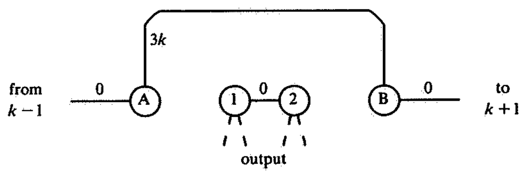
(*iii*) Both inputs are unvisited. The tour will pass through all vertices of subgraph k before going to subgraph $k+1$ (Figure 4d).

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.



(a) The representation of a TRUE input



(b) The representation of a FALSE input

FIGURE 4. The representation of input k.

For TRUE and FALSE inputs we construct the subgraphs as shown in Figure 4. The representation of NAND gate m (the last one) has a somewhat special structure. The output vertices are replaced by a vertex C. Both vertex B and C are connected to input 1 (see Figure 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.

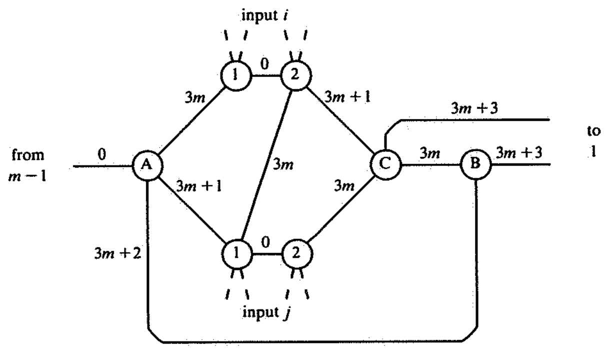


FIGURE 5. The representation of NAND gate m.

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 log space complete for $\mathcal{P}$. So, the construction of a nearest neighbor traveling salesman tour will probably 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 log space complete for $\mathcal{P}$ by giving a transformation from the circuit value problem.

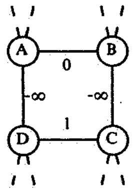


FIGURE 6. The representation of an arc.

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

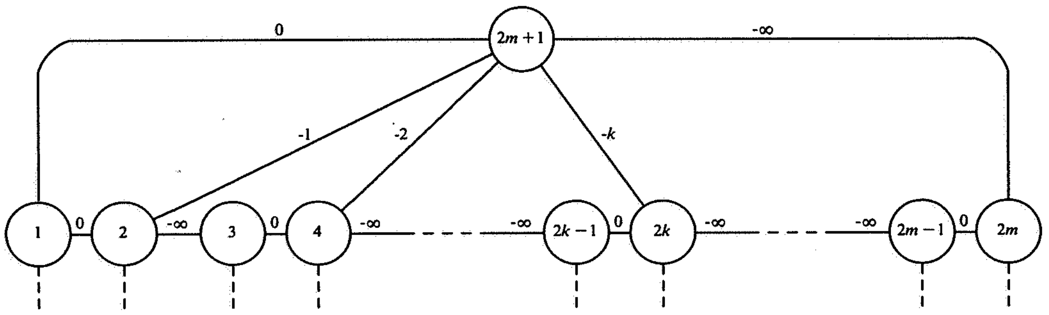


FIGURE 7. The extra graph.

We also construct the graph of Figure 7, where m is the number of gates of the circuit. In the beginning of the algorithm tours of the form $(2k)-(2k+1)-(2k)$ are made, for $k = 1,\ldots,m$. The edge lengths $-(m-1),\ldots,-1,0$ assure that the original tour $(2m)-(2m+1)-(2m)$ is merged with the other cycles of length 2 and finally with the self-loop $(1)-(1)$. The result is the tour $(2m+1)-(1)-(2)-\ldots-(2k-1)-(2k)-\ldots-(2m-1)-(2m)-(2m+1)$.

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

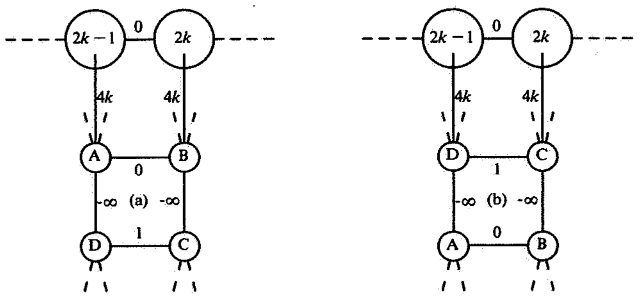


FIGURE 8. The representation of the input gates; (a) input gate k is TRUE; (b) input gate k is FALSE.

The cycles corresponding to arcs from input gates are connected to the extra graph as shown in Figure 8. If it is a TRUE (FALSE) input, the edges {A,B} and $\{2k-1,2k\}$ ({C,D} and $\{2k-1,2k\}$) are replaced by $\{2k-1,\mathrm{A}\}$ and $\{2k,\mathrm{B}\}$ ($\{2k-1,\mathrm{D}\}$ and $\{2k,\mathrm{C}\}$).

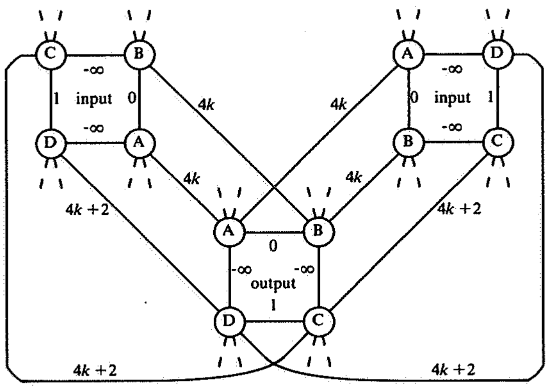
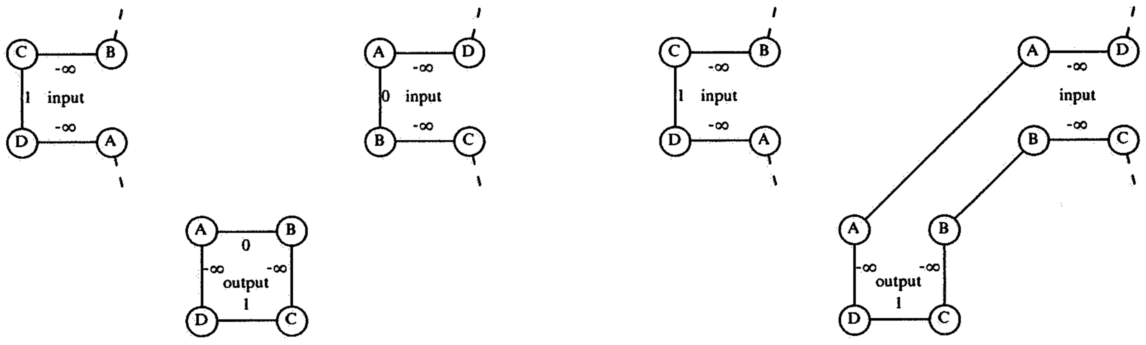


FIGURE 9. The representation of NAND gate k (fan out 1).

For NAND gate k with fan out one, the subgraphs are connected as shown in Figure 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 Figure 10).



(a) Situation before merging

(b) Situation after merging

FIGURE 10. TRUE NAND FALSE → TRUE.

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.

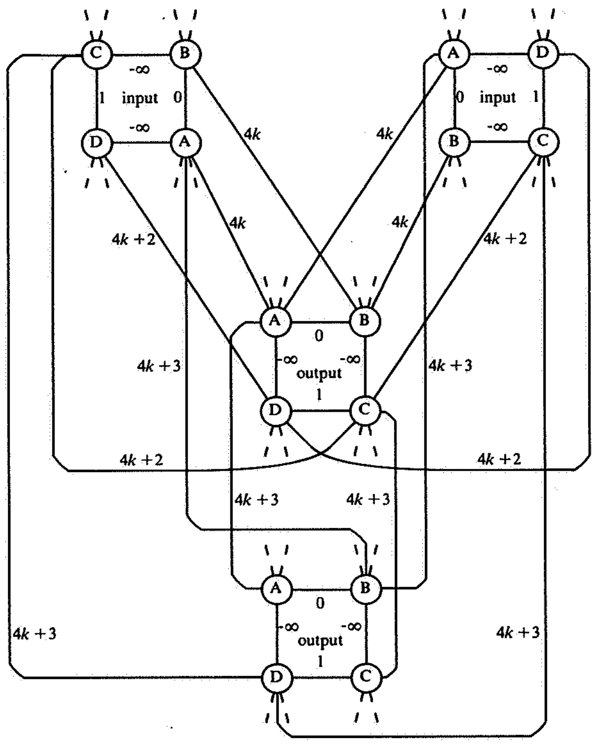


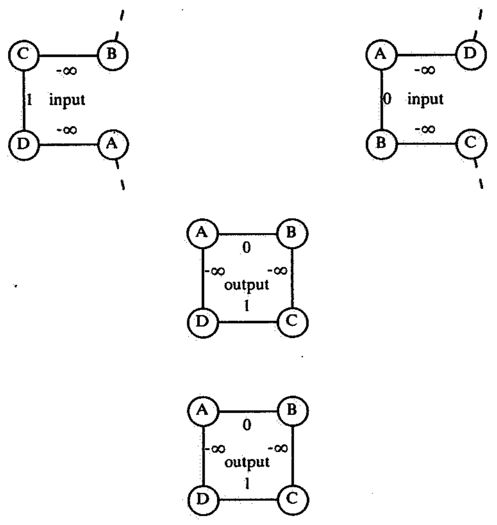
FIGURE 11. The representation of NAND gate k (fan out 2).

If NAND gate k has fan out two, we connect the subgraphs as shown in Figure 11. The case TRUE NAND FALSE $\rightarrow$ TRUE is illustrated in Figure 12.

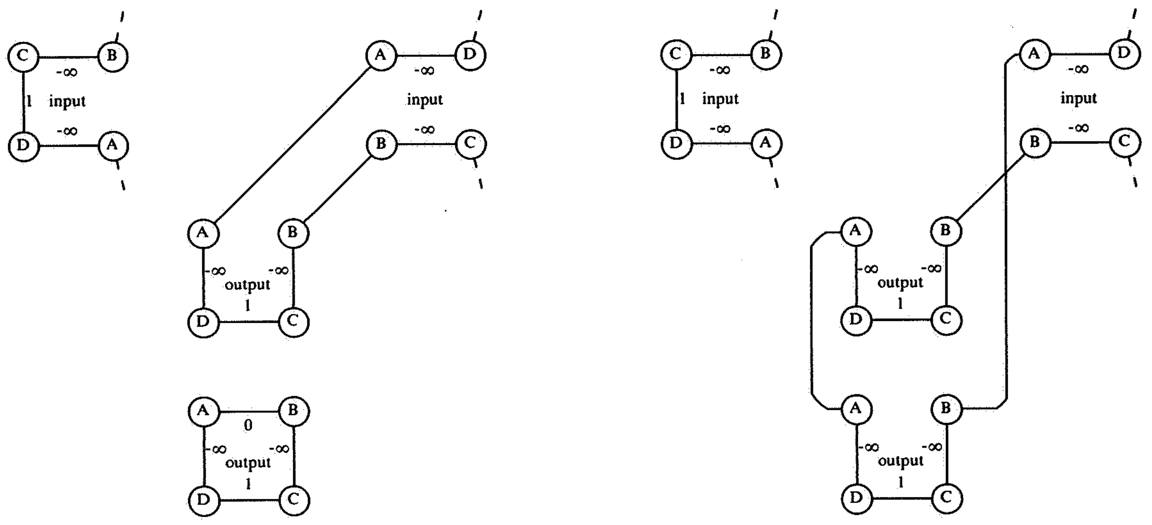
The other cases are left as exercises to the reader. Note that the output vertices of a subgraph are always inserted between two edges of length $-\infty$ of one of the inputs.

So far, we have assumed that there are no edges in the tour connecting both inputs of the same gate. Because of the way 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.

Considering all above, it follows that the circuit value of an instance of the circuit value problem is TRUE if and only if the nearest merger heuristic 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 log space complete for $\mathcal{P}$.



(a) Situation before merging.



(b) The two merging steps.

FIGURE 12. TRUE NAND FALSE → TRUE.

6. Nearest Insertion, Cheapest Insertion and Farthest Insertion

Given a distance matrix, a starting vertex and an edge $\{i,j\}$, the nearest insertion (cheapest insertion, farthest insertion) problem is the problem of deciding whether the nearest insertion (cheapest insertion, farthest insertion) heuristic produces a tour which contains edge $\{i,j\}$. The transformations from the circuit value problem showing that these problems are log space complete for P are similar to the one for the nearest merger problem. We will give only the crucial part of the transformation, leaving the details and the verification to the reader.

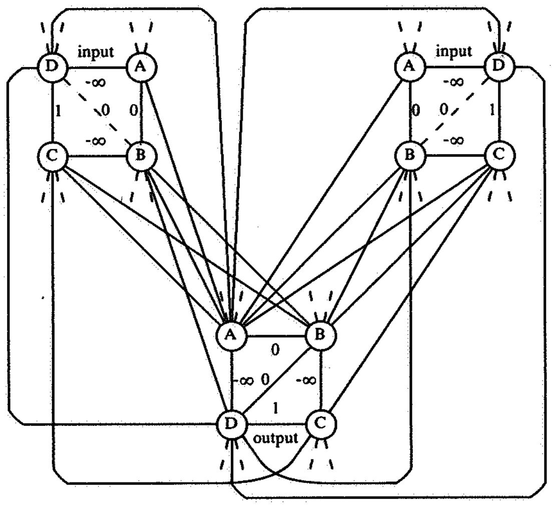


FIGURE 13. The representation of NAND gate k.
The edges from output vertex A to the vertices A and B of both inputs have length $2k$;
all other edges between input and output vertices have length $2k+1$.

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

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

Cheapest Insertion
The same transformation as described for the nearest insertion problem works for the cheapest insertion problem, because in each step both algorithms will choose the same vertex to be inserted in the graphs simulating the logical circuit. So, the cheapest insertion problem is log space complete for $\mathcal{P}$ 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 x_v be the number of the step in which vertex v 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 s_1, s_2, s_3, s_4 and s_5, such that the farthest insertion heuristic, when started in s_1, first builds the tour $s_1-s_2-s_3-s_4-s_5-s_1$. Edges originally incident to the starting vertex are made incident to s_3 and s_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 v to s_1 with value $-100n-x_v$, and to s_2 and s_5 with value $200n$. The edges to s_1 are the smallest in the graph and determine the order in which the vertices are added to the tour. The edges to s_2 and s_5 prohibit the exclusion of $\{s_1,s_2\}$ and $\{s_1,s_5\}$ 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 log space complete for $\mathcal{P}$.

7. Double Minimum Spanning Tree and Nearest Addition

In this section we will give implementations of the double minimum spanning tree and nearest addition heuristics that require polylogarithmic time on a PRAM with a polynomial number of processors.

We introduce the notation

```
**par** $[a \leqslant i \leqslant z]$ $s_i$
```

to denote that the statements s_i are to be executed in parallel for all values of the index i in the given range.

Double Minimum Spanning Tree

The double minimum spanning tree heuristic consists of three phases; see Section 2. Phase (*i*) (constructing a minimum spanning tree) can be performed in $O(\log^2 n)$ time with $O(n^2/\log^2 n)$ processors [Chin, Lam & Chen 1982]. Phase (*ii*) (doubling the minimum spanning tree and finding an Euler circuit) can be done within the same time and processor bounds using the techniques from [Awerbuch, Israeli & Shiloach 1984]. For phase (*iii*), we first have to find the first occurrence of each vertex and then eliminate all duplications. Let $v_1,...,v_i,...,v_{2n-1}$ denote the Eulerian tour obtained in the previous phase, where v_i is the *i*th vertex of the tour. We proceed as follows.

```
par $[1 \leqslant i,j \leqslant 2n-1]$ if $v_i = v_j$ then $c_{ij} \leftarrow 1$ else $c_{ij} \leftarrow 0$.
par $[1 \leqslant i \leqslant 2n-1]$ $d_i \leftarrow \min\{0, 1-\sum_{j=1}^{i-1} c_{ij}\}$.
par $[1 \leqslant i \leqslant 2n-1]$ $s_i \leftarrow \sum_{j=1}^{i} d_j$.
```

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

```
par $[1 \leqslant i \leqslant 2n-1]$ if $d_i = 1$ then $t_{s_i} \leftarrow v_i$.
```

Using the partial sums algorithm from [Dekel & Sahni 1983], 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.

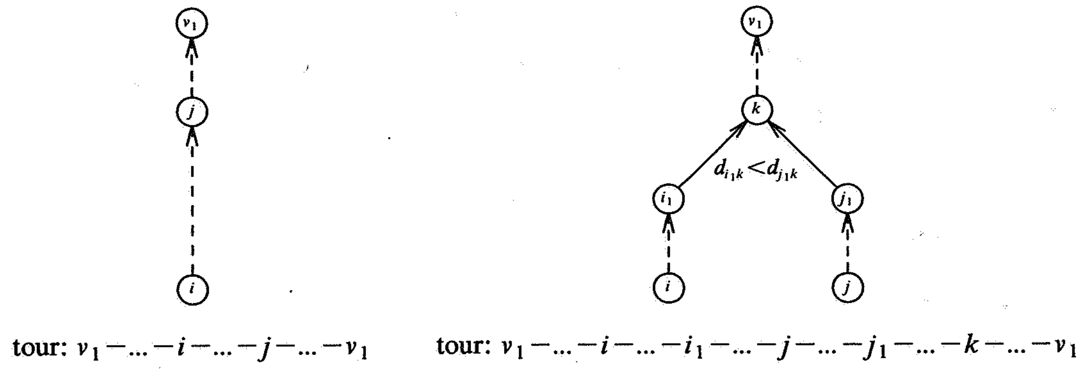
Nearest Addition

The order in which the the nearest addition heuristic adds vertices to the tour corresponds to way in which the algorithm from [Prim 1957] builds up a minimum spanning tree. Therefore, we first construct a minimum spanning tree and direct the edges towards v_1. By means of this spanning tree, we can determine for each two vertices *i* and *j* which one will be visited first. There are two possible situations (Figure 14). In the first situation one vertex is a descendant of the other. Since vertices are inserted directly before their fathers 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_1 (j_1) be the last vertex on the path from *i* (*j*) to *k*; $i_1 = i$ ($j_1 = j$) if the path consists of only one arc. If $d_{i_1k} < d_{j_1k}$, vertex i_1 will be inserted before vertex *k*, and after that vertex j_1 will be inserted in the tour directly before vertex *k* and thus behind vertex i_1.

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 to vertex v_1, getting arcs (i, T_i) for $i \in \{1,...,n\} \setminus v_1$. For convenience, we assume $T_{v_1} = v_1$. This requires $O(\log^2 n)$ time and $O(n^2/\log^2 n)$ processors [Chin, Lam & Chen 1982; Savage 1977].

(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 exist a path from vertex *i* to vertex *j*, $c_{ij} = 0$ otherwise). Let $V(i,l)$ denote the vertex at distance 2^l from vertex *i*, or v_1 if this vertex does not exist. The following statements do the job:



v_1
j
i
tour: $v_1 - ... - i - ... - j - ... - v_1$
v_1
k
$d_{i_1k} < d_{j_1k}$
i_1
j_1
i
j
tour: $v_1 - ... - i - ... - i_1 - ... - j - ... - j_1 - ... - k - ... - v_1$

FIGURE 14. The two possible situations.

```
par [$1 \le i \le n$] $V(i,0) \leftarrow T_i$.
for $l \leftarrow 1$ to $\lceil \log n \rceil$ do
    par [$1 \le i \le n$] $V(i,l) \leftarrow V(V(i,l-1),l-1)$.

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 $c_{iV(j,l)} = 0$ then $c_{iV(j,l)} = 1$.
```

(The 'if $c_{iV(j,l)} = 0$' 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 [$1 \le i \le n$, $(l-1)\lceil \frac{n}{x} \rceil + 1 \le j \le l \lceil \frac{n}{x} \rceil$] $c_{ij} \leftarrow 0$.
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 $c_{iV(j,l)} = 0$ then $c_{iV(j,l)} = 1$ & assign a free processor to $(i, V(j,l))$.
for $l \leftarrow x$ downto 0 do
    par [$1 \le i,j \le n$ & $(i,j)$ has a processor assigned to it] if $c_{iV(j,l)} = 0$ then
        $c_{iV(j,0)} = 1$ & assign the processor, assigned to $(i,j)$, to $(i, V(j,0))$.
```

By choosing $x = \lceil \log^2 n \rceil$, we can 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 D_i of vertices in the subtree rooted by i:

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

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

```
par [$1 \le i \le n$] $P_i \leftarrow \sum_j \{D_j \mid (T_i = T_j) \ \& \ (d_{iT_i} < d_{jT_j})\}$; $P_{v_1} \leftarrow 0$.
```

Finally, for each vertex i we compute the number of vertices visited later:

par $[1 \leqslant i \leqslant n]$ $AFTER_i \leftarrow \sum_j c_{ij}(1+P_j)$

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

8. Conclusion

We have shown that several traveling salesman heuristics are sequential in nature, i.e., their corresponding problems are log space complete for $\mathcal{P}$. We have also shown that the double minimum spanning tree and the nearest addition heuristics can be implemented so as to require polylogarithmic time on a polynomial number of machines. An open problem is presented by Christofides' heuristic, which constructs a tour from an Eulerian circuit consisting of a minimum spanning tree and a minimum perfect matching on the vertices of odd degree in the tree. A minimum spanning tree can be found in polylogarithmic parallel time, but whether this is also true for a minimum perfect matching is one of the outstanding open problems in parallel combinatorial optimization.

References

B. Awerbuch, A. Israeli, Y. Shiloach (1984). Finding Euler circuits in logarithmic parallel time. *Proc. 16th Annual ACM Symp. Theory of Computing*, 249-257.

A.K. Chandra, D.C. Kozen, L.J. Stockmeyer (1981). Alternation. *J. Assoc. Comput. Mach. 28*, 114-133.

F.Y. Chin, J. Lam, I-N. Chen (1982). Efficient parallel algorithms for some graph problems. *Comm. ACM 25*, 659-665.

S.A. Cook (1974). An observation on time-storage trade off. *J. Comput. System Sci. 9*, 308-316.

S.A. Cook (1981). Towards a complexity theory of synchronous parallel computation. *Enseign. Math. (2) 27*, 99-124.

E. Dekel, S. Sahni (1983). Binary trees and parallel scheduling algorithms. *IEEE Trans. Comput. C-32*, 307-315.

M.R. Garey, D.S. Johnson (1979). *Computers and Intractability: a Guide to the Theory of NP-Completeness*, Freeman, San Francisco.

L.M. Goldschlager (1982). A universal connection pattern for parallel computers. *J. Assoc. Comput. Mach. 29*, 1073-1086.

L.M. Goldschlager, R.A. Shaw, J. Staples (1982). The maximum flow problem is log space complete for P. *Theoret. Comput. Sci. 21*, 105-111.

D.S. Johnson (1983). The NP-completeness column: an ongoing guide; seventh edition. *J. Algorithms 4*, 189-203.

G.A.P. Kindervater, J.K. Lenstra (1985). Parallel algorithms. M. O'hEigeartaigh, J.K. Lenstra, A.H.G. Rinnooy Kan (eds.). *Combinatorial Optimization: Annotated Bibliographies*, Wiley, Chichester, 106-128.

G.A.P. Kindervater, J.K. Lenstra (1986). An introduction to parallelism in combinatorial optimization. *Discrete Appl. Math. 14*, 135-156.

R.E. Ladner (1975). The circuit value problem is log space complete for *P*. *SIGACT News 7.1*, 18-20.

E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, D.B. Shmoys (eds.) (1985). *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, Wiley, Chichester.

R.C. Prim (1957). Shortest connection networks and some generalizations. *Bell System Tech. J. 36*, 1389-1401.

C. Savage (1977). *Parallel Algorithms for Graph Theoretical Problems*, Ph.D. Thesis, University of Illinois, Urbana-Champaign.

L.G. Valiant (1983). Parallel computation. J.W. de Bakker, J. van Leeuwen (eds.). *Foundations of Computer Science IV, Distributed Systems: Part 1, Algorithms and Complexity*, Mathematical Centre Tract 158, Centre for Mathematics and Computer Science, Amsterdam, 35-48.