PARALLEL ALGORITHMS FOR SOLVING AGGREGATED SHORTEST PATH PROBLEMS

H. Edwin Romeijn
Rotterdam School of Management
Erasmus University Rotterdam
Rotterdam, The Netherlands

Robert L. Smith
Department of Industrial and Operations Engineering
The University of Michigan
Ann Arbor, MI 48109-2117

Technical Report 93-25

September 1993

Parallel Algorithms for Solving Aggregated Shortest Path Problems*

H. Edwin Romeijn
Rotterdam School of Management
Erasmus University Rotterdam
Rotterdam, The Netherlands

Robert L. Smith
Department of Industrial and Operations Engineering
The University of Michigan
Ann Arbor, Michigan

September 14, 1993

Abstract

We consider the problem of finding all pairs of shortest paths within a large scale directed network. We form nested partitions of the original micro-nodes to create a hierarchy of macro-nodes. Combining all pairs of shortest paths within macro-nodes at each level yields a hierarchical decomposition algorithm that reduces computation time by as much as $O(\sqrt{N})$. Motivated by an IVHS application, bounds on resulting errors are explored accompanied by an empirical test of their magnitude.

Key Words and Phrases

Shortest path, parallel algorithm, aggregation, IVHS, traffic assignment.

1 Introduction

It is a well-known principle that every additive deterministic dynamic programming formulation can be equivalently viewed as the problem of finding the shortest path in a directed

*This work was supported in part by the IVHS Program at the University of Michigan.

network where the states, decisions, and decision costs of the former correspond to the nodes, arcs, and arc lengths of the latter (Dreyfus and Law [1977]). It is perhaps for this reason that the task of efficiently computing shortest paths figures so prominently in the mathematical programming literature. Our interest in this paper is directed toward solving very large scale shortest path problems, motivated by the problem of finding minimum travel time paths for an IVHS or Intelligent Vehicle Highway System (Kaufman and Smith [1993]).

There are fundamentally two types of shortest path problem that arise in an IVHS context. The first is a subproblem of the dynamic traffic assignment problem. The problem is to anticipate vehicular volumes along links in a traffic network by routing future trips, each characterized by an origin, departure time, and destination. The assumption made is that vehicles will take the path achieving minimum trip time based upon dynamic link travel times. These dynamic link travel times are updated recursively in an attempt to asymptotically obtain a stable link time forecast. It is easy to demonstrate that these trips then are in dynamic equilibrium (Kaufman, Smith, and Wunderlich [1991]). The second application arises in real time wherein a vehicle requests a minimum travel time path from its origin to desired destination based upon forecasted dynamic link travel times. In both cases, shortest path computations constitute the greater part of the computational effort where a realistic problem may have hundreds of thousands of nodes, all of which my be potential origins and destinations.

We begin in section 2 with a review of conventional parallel shortest-path algorithms and network aggregation models to attempt to accelerate the computational effort. In section 3 we combine these two approaches into a unified decomposition algorithm. We show how the parallel nature of the algorithm induces a natural choice for how the network should be aggregated. In section 4, the results of computational experiments are reported suggesting several orders of magnitude improvement in computational times over conventional approaches.

2 Background

In this section some of the sequential and parallel shortest-path algorithms from the literature will be discussed. We will start by introducing some notation.

Let $G = (V, A)$ be a graph, where $V = \{1, \ldots, N\}$ is the set of nodes, and $A \subset V \times V$ is the set of arcs. Here $(i, j) \in A$ if there exists an arc from node $i \in V$ to node $j \in V$. Furthermore, let $t_{ij} \geq 0$ denote the distance (or travel time, or some other measure of cost) from i to j. If $(i, j) \notin A$ then $t_{ij} = +\infty$. Note that the travel time from node i to node j

is assumed to be stationary, i.e., independent of the actual arrival time at node i. Let f_{ij} denote the length of the shortest-path from i to j in the graph.

2.1 Sequential shortest-path algorithms

2.1.1 Acyclic graphs

If G is a graph without (directed) cycles, then without loss of generality we can assume that the elements in V are ordered in such a way that $(i,j) \in A$ implies $i < j$. The shortest-path lengths then satisfy the following recursion:

$$f_{ij} = \min_{i \leq k < j} \{f_{ik} + t_{kj}\}$$

for $i = 1, \ldots, N-1$ and $j = i+1, \ldots, N$. We can solve for the f-values using dynamic programming, using either the *recursive-fixing method* or the *reaching method:*

(i) *Recursive-fixing method:*

```
DO for $i = 1, \ldots, N-1$
    SET $f_{ii} = 0$ and $f_{ij} = \infty$ for $j = i+1, \ldots, N$
    DO for $j = i+1, \ldots, N$
        DO for $k = i, \ldots, j-1$
            $f_{ij} = \min(f_{ij}, f_{ik} + t_{kj})$
```

(ii) *Reaching method:*

```
DO for $i = 1, \ldots, N-1$
    SET $f_{ii} = 0$ and $f_{ij} = \infty$ for $j = i+1, \ldots, N$
    DO for $k = i+1, \ldots, N-1$
        DO for $j = k+1, \ldots, N$
            $f_{ij} = \min(f_{ij}, f_{ik} + t_{kj})$
```

For both methods the time necessary to compute all shortest-paths in the graph is $O(N^3)$ (see Denardo [1982]).

2.1.2 Cyclic graphs

For cyclic graphs the lengths of the shortest-paths satisfy the following functional equation:

$$f_{ij} = \min_{k \neq j} \{f_{ik} + t_{kj}\}$$

for $i, j = 1, \ldots, N$.

(i) *Dijkstra's method:*
This method is basically an adaptation of the reaching method for acyclic graphs discussed above. For fixed i, this method computes the values of f_{ij} in nondecreasing value sequence:

```
DO for $i = 1, \ldots, N$
    SET $f_{ii} = 0$, SET $f_{ij} = t_{ij}$ for $i \neq j = 1, \ldots, N$, and SET $T = V - \{i\}$
    REPEAT
        SET $k = \arg\min_{j \in T} f_{ij}$
        SET $T = T - \{k\}$. IF $T = \emptyset$ STOP, otherwise
        DO for $j \in T$
            $f_{ij} = \min(f_{ij}, f_{ik} + t_{kj})$
```

The complexity of this algorithm is $O(N^3)$ for dense graphs, and $O(nN^2 \log N)$ for sparse graphs, where n is the average number of arcs emanating from a node (see Dreyfus and Law [1977]).

(ii) *Floyd's algorithm:*
Let $f_{ij}(k)$ denote the length of the shortest-path from i to j, where the shortest-path only uses nodes from the set $\{1, \ldots, k\}$. Then obviously $f_{ij} = f_{ij}(N)$. We can now solve recursively for the values of k:

```
SET $f_{ij}(0) = t_{ij}$ for all $(i, j)$
DO for $k = 1, \ldots, N$
    FOR ALL $(i, j)$ SET $f_{ij}(k) = \min(f_{ij}(k-1), f_{ik}(k-1) + f_{kj}(k-1))$
```

Again, the complexity of this algorithm is $O(N^3)$.

In the following section we will see that an adaptation of the last method (Floyd's method) is especially suited for use in a parallel-computing environment.

2.2 Parallel algorithms

2.2.1 A modification of Floyd's algorithm

In the original version of Floyd's algorithm, $f_{ij}(k)$ denotes the length of the shortest-path from i to j using only intermediate nodes from the set $\{1, \ldots, k\}$. As an alternative, let us denote the length of the shortest-path from i to j using at most $k-1$ intermediate nodes (i.e., using at most k arcs) by f_{ij}^k. Then, using the inequality $2^{\lceil \log(N-1) \rceil} \geq N-1$, we have that $f_{ij} = f_{ij}^{2^{\lceil \log(N-1) \rceil}}$ since there is always a shortest path without directed cycles. So, the f-values can be computed recursively using the following algorithm:

```
SET $f_{ij}^1 = t_{ij}$ for all $(i,j)$.
DO for $k = 1, \ldots, \lceil \log(N-1) \rceil$
  FOR ALL $(i,j)$
    SET $f_{ij}^{2^k} = \min_{\ell \in V} \left( f_{i\ell}^{2^k-1} + f_{\ell j}^{2^k-1} \right)$
```

When implemented sequentially, this algorithm has complexity $O(N^3 \log N)$. However, the part of the algorithm inside the outer loop can be implemented using a modification of a parallel matrix multiplication algorithm. For the latter problem, a variety of algorithms exists, one of which we will discuss in some more detail in the next section.

2.2.2 Parallel matrix multiplication

Consider the problem of multiplying two $N \times N$ matrices, say A and B. Let $C = AB$. An algorithm for computing C is:

```
FOR ALL $(i,j)$
  SET $c_{ij} = 0$
  FOR $\ell = 1, \ldots, N$
    SET $c_{ij} = c_{ij} + a_{i\ell} b_{\ell j}$
```

In figure 1 we illustrate how this algorithm can be implemented in a synchronous parallel fashion by using a mesh-connected parallel computer with N^2 processors. The time complexity of this parallel algorithm can be shown to be $O(N)$. Comparing the modified Floyd's algorithm and the matrix multiplication algorithm it is clear that we obtain the innermost loop of the former algorithm from the latter algorithm by replacing multiplication by addition and addition by taking a minimum. We thus obtain a parallel algorithm for shortest-path

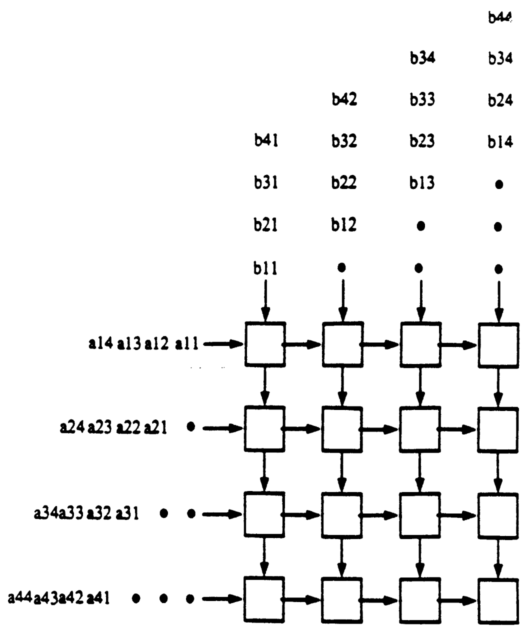

Figure 1: A parallel algorithm for multiplying two matrices

calculation having complexity $O(N \log N)$, by using N^2 processors.

The complexity of parallel-matrix computation can be reduced to $O(\log N)$ if we use N^3 processors, which are connected as the vertices of a hypercube. Using an algorithm of this type yields a parallel algorithm for shortest-path computation with complexity $O(\log^2 N)$. For more detail on parallel algorithms for matrix computation, and their use in parallel shortest-path computation we refer the reader to Quinn [1987], Akl [1989], and Bertsekas and Tsitsiklis [1989].

2.2.3 Another "parallel" algorithm

An obvious way of parallelizing (for example) Dijkstra's algorithm for the all-pairs shortest-path problem is to use $P \leq N$ processors working in parallel, and to let each processor compute all shortest-paths from at most $\lceil N/P \rceil$ origins to all possible destinations. In this way we obtain a parallel implementation having time complexity $O(\lceil N/P \rceil N \log N)$ for sparse graphs, and $O(\lceil N/P \rceil N^2)$ for dense graphs. If we choose the number of processors to be equal to the number of nodes, the complexities become $O(N \log N)$ and $O(N^2)$ respectively.

Note that the notion of "processor" used in this context differs from the one used in the preceding section, since the tasks that have to be performed by the two processors differ greatly. On the other hand, the latter algorithm is only pseudo-parallel in the sense that there is no communication necessary between the processors. This, of course, is an enormous practical advantage, since no real parallel computer architecture is necessary.

3 Aggregation

In the previous section we have seen that we can solve the all-pairs shortest-path problem in $O(N^2 \log N)$ time (for sparse networks) when using a sequential algorithm. Moreover, we saw that it is possible to reduce this time by a factor N to $O(N \log N)$ by using N^2 small or N large processors in a parallel fashion. In this section we will investigate how by aggregating nodes we can reduce the number of processors necessary to solve the problem, while keeping the time complexity of the algorithm equal to $O(N \log N)$. We will consider the errors involved in this procedure in section 4. See Bean, Birge, and Smith [1987] for a serial aggregation procedure for shortest paths in acyclic networks.

3.1 A simple model

We will start by considering the following model. Let $G = (V, A)$ be a graph, and suppose every nonboundary node has exactly 4 neighbors. We will refer to this as a Manhattan network.

More precisely, suppose G is topologically equivalent to a $\sqrt{N} \times \sqrt{N}$ mesh (see figure 2). We will aggregate nodes by forming a partition of the nodes of the network into M classes called macronodes. Moreover we aggregate in such a way that the "macronetwork" of M macronodes is a $\sqrt{M} \times \sqrt{M}$ mesh, and that every macronode itself is a $\sqrt{N/M} \times \sqrt{N/M}$ mesh.A macronode is present between two macro-nodes if and only if there is an arc connecting two nodes in threir respective aggregate classes. Define the arc lengths in the macronetwork to be the shortest of the lengths of all (micro) arcs connecting two macronodes.

We can approximately solve the shortest-path problem for G by finding all shortest-paths in the macronetwork, and also all shortest-paths in each macronode, and then combine these to get paths connecting all pairs of nodes. Of course, these paths are not necessarily shortest-paths in G, even if all subproblems are solved optimally.

Suppose we have $M + 1$ processors. We can solve for all shortest paths inside each of the M

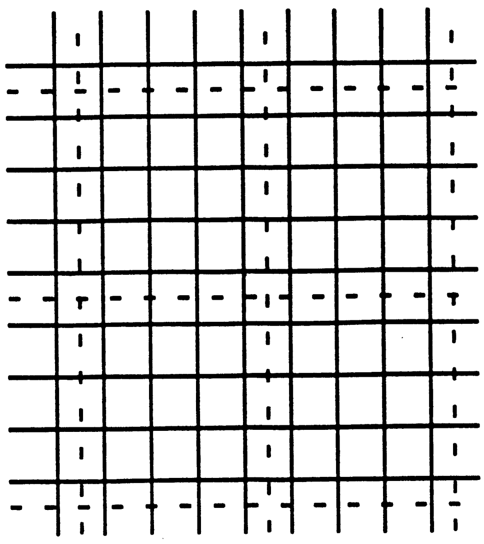

Figure 2: "Manhattan Network"

macro-nodes with M of the processors in parallel while the $(M+1)$-st processor solves for all shortest paths in the macronetwork. We have

1. the time necessary to compute all shortest-paths inside one of the macronodes is $O((N/M)^2 \log(N/M))$ if we use Dijkstra's algorithm
2. the time necessary to compute all shortest-paths in the macronetwork is $O(M^2 \log M)$.

We now want to minimize the time necessary for *all* $M+1$ processors to complete their task. That is, we want to solve the problem:

$$\min_{1 \le M \le N} (\max(O((N/M)^2 \log(N/M)), O(M^2 \log M))).$$

The solution M^* can be easily shown to be attained by requiring the number of nodes inside each macronode to be equal to the number of macronodes: $N/M^* = M^*$, or

$$M^* = \sqrt{N}.$$

Note that this remains the solution for a more general problem where the size of each macronode can be variable. So we conclude that using $M^*+1 = \sqrt{N}+1$ (or $M^* = O(\sqrt{N})$) processors, we obtain an approximate solution to the shortest-path problem in $O(N \log \sqrt{N})$

or $O(N \log N)$ time. Although the model presented here is very simple, the general results still hold if we only make the assumption that we only consider partitions of the network into macronodes having the property that

1. each macronode has the same structure as the original network
2. the macronetwork obtained by aggregating the nodes inside each macronode to form one node also has the same structure as the original network.

If the original network (and the macronodes) is dense, the results concerning the number of macronodes again remain the same, but the time complexity of the algorithm becomes $O(N^2)$ (see also section 2.3 above).

Note that a problem can occur if there are "one-way-streets" in the network; i.e., if there exists a pair $i, j \in V$ such that $(i,j) \in A$ and $(j,i) \notin A$. If link (i,j) ends up *inside* a macronode, it is possible that there exists a path from j to i of finite length in the network, while the decomposition algorithm returns with a path length of $+\infty$. The reason for this is that, for a given pair of nodes within a macronode, it is possible that there does not exist a path between those nodes that is completely contained in the macronode.

The approximate solution to the all-pairs shortest-path problem that can be obtained in $O(N \log N)$ time consists of tables of shortest-path lengths for each of the macronodes, together with a table for the macronetwork. To obtain approximate shortest-path lengths for the original micronetwork these results need to be combined. As a first step towards this we modify the entries in the table corresponding to the macronetwork as follows: for every intermediate macronode on a shortest-path in the macronetwork, add the shortest-path length from the *entry-micronode* to the *exit-micronode*. This shortest-path length can be found in the shortest-path table of the corresponding macronode. Since the number of nodes on a shortest macro-path will be $O(N^{1/4})$ on average, this can be performed in $O((\sqrt{N})^2 N^{1/4})$ or $O(N^{5/4})$ time sequentially. Obviously this procedure can easily be parallelized by using another $\sqrt{N}$ processors, yielding a time complexity of $O(\sqrt{N} N^{1/4})$ or $O(N^{3/4})$, whose inclusion still keeps the complexity of the total algorithm unchanged at $O(N \log N)$. For every pair (i,j), $i,j \in V$ in the original micro-network the time to find an approximate shortest-path length is now reduced to 2 additions and 3 table-lookups: the length of the approximate shortest-path from i to j equals the length of the (approximate) shortest-path from i to the exit-node of the macronode containing i; plus the (approximate) shortest-path length from the exit-node of the macronode containing i to the entry-node of the macronode containing j; plus the (approximate) length of the shortest-path from the entry-node of the macronode containing j to j. Obviously, performing this procedure for *every* pair (i,j) takes $O(N^2)$ time. Parallel implementation however using another $\sqrt{N}$ processors reduces this to

$O(N^2/\sqrt{N})$ or $O(N\sqrt{N})$ time. Alternatively, the complexity of Dijkstra, when implemented using $\sqrt{N}$ processors, is $(N/\sqrt{N})O(N \log N)$ or $O(N\sqrt{N}\log N)$. In other words, *computing approximate shortest-path lengths using the aggregation algorithm introduced in this section yields a savings of* $O(\log N)$ *in time complexity.*

In practice, the savings could be much larger. As an example, consider the case where not all shortest-paths are required at one time, for example when queries are made for on-line shortest path information. An instance of the latter is the second IVHS application discussed in the introduction where a single vehicle makes a real time request for a minimum travel time problem. As another example, when the macronodes correspond to metropolitan areas, many of the shortest-path requests will be for paths *inside* a macronode. This information is immediately available, and moreover, for many origin/destination pairs this information will be *exact* instead of *approximate.* The magnitude of the error would depend upon the relative distribution of trips within as opposed to between metropolitan areas. In this case the last (and most time consuming) part of the aggregation algorithm can be avoided, since any shortest-path length can be computed in constant time upon request, yielding an effective time complexity of $O(N \log N)$ for the algorithm. In contrast, an equivalent savings in time cannot be obtained when using Dijkstra's algorithm. *In this case the savings in complexity of the aggregation algorithm over Dijkstra's algorithm increases to* $O(\sqrt{N})$.

3.2 Multi-Level Aggregation

In the preceding section we considered the case where aggregation takes place only one level down. In this case we will say that the aggregation is over two levels. We now generalize the results to the case of an arbitrary number of aggregation levels L. The idea is again to aggregate the $\sqrt{N} \times \sqrt{N}$ mesh into a macronetwork that is a $\sqrt{M} \times \sqrt{M}$ mesh. Each of the M macronodes of level 1 itself is a $\sqrt{N/M} \times \sqrt{N/M}$ mesh. Then, aggregate each level 1 macronode into a $\sqrt{M} \times \sqrt{M}$ mesh. Each macronode of level 2 is then a $\sqrt{N/M^2} \times \sqrt{N/M^2}$ mesh. Continue this until we have macronodes of level $L-1$ which are $\sqrt{N/M^{L-1}} \times \sqrt{N/M^{L-1}}$ meshes. So now we have 1 macronetwork of level 1 having M nodes, M macronetworks of level 2 having M nodes, $\ldots$, M^{L-2} macronetworks of level $L-1$ having M nodes, and M^{L-1} level L micronetworks having N/M^{L-1} nodes. Assume we have $1 + M + \cdots + M^{L-1} = \frac{M^L-1}{M-1} \equiv P$ processors, each exactly solving a shortest-path problem.

Inductively using the same reasoning as in the case of $L = 2$ above we obtain that it is optimal to choose M according to $N/M^{*L-1} = M^*$, or

$$M^* = N^{1/L}.$$

So, using $(N-1)/(N^{1/L}-1) = O(N^{1-1/L})$ processors and treating L as a constant, *we can obtain an approximate solution to the shortest-path problem in $O(M^{*2}\log M^*)$ or $O(N^{2/L}\log N)$ time.*

As in the case of $L=2$ we need to combine the results of the exact solutions to the subproblems to get shortest-path lengths in the original network. The (sequential) complexity of the first phase is given by $M^2\sqrt{M}$ (the number of operations per network of M nodes), multiplied by $\frac{M^{L-1}-1}{M-1}$ (the number of subnetworks of size M), yielding a complexity of $O(M^{2+L}\sqrt{M})$. A parallel implementation using $P = O((M^2-1)/(M-1))$ processors then gives (after substitution of $M = O(N^{1/L})$): $O(N^{3/2L}) < O(N^{2/L})$. The second phase takes $O(N^{1+1/L})$ time when implemented in a parallel fashion.

It would be interesting to address the question: what is the "optimal" choice for L? To answer this question we have to define what we mean by "optimal." However, one of the elements that would certainly have to be included here is the effect of aggregating a certain number of levels down on the precision of the solution obtained by the algorithm. Obviously, there will be a negative influence of increasing the level of aggregation on the precision of the solution, but at this point this is all we really can say about this effect. So for now the question of optimal aggregation level choice remains an open issue for future research.

4 Aggregation in practice

For a general network it is not clear how one should aggregate nodes into macronodes. However, returning to the two level aggregation case of section 2, the following theorem gives an upper bound on the absolute error made in approximating the shortest-path length for a given origin/destination pair.

Theorem 4.1 *Let f_{ij} denote the length of a shortest-path between nodes i and j, and let $\hat{f}_{ij}$ denote the length of an approximate shortest path computed by the decomposition algorithm. Furthermore, decompose each of those lengths as follows:*

$$\begin{aligned} f_{ij} &= f_{ij}^C + f_{ij}^W \\ \hat{f}_{ij} &= \hat{f}_{ij}^C + \hat{f}_{ij}^W \end{aligned}$$

where a superscript C provides the length of all edges on a path that are connecting *macronodes, and W denotes the length of all edges on a path that are entirely* within *macronodes. Then*

$$\hat{f}_{ij} - f_{ij} \leq \hat{f}_{ij}^W.$$

Proof: By construction, $\hat{f}_{ij}^C \leq f_{ij}^C$, and by definition $f_{ij}^W \geq 0$. So we have

$$\begin{aligned} \hat{f}_{ij} - f_{ij} &= \hat{f}_{ij}^C - f_{ij}^C + \hat{f}_{ij}^W - f_{ij}^W \\ &\leq \hat{f}_{ij}^W . \quad \blacksquare \end{aligned}$$

This theorem suggests that the network should be aggregated in such a way that edges *within* macronodes are relatively *short*, and edges *connecting* macronodes are relatively *long*.

A possible approach to aggregating an arbitrary network based on the previous observation is the following. The aggregation problem can be formulated as an optimization problem. Let M be the desired number of macronodes, to be specified in advance. Let x_{im} be a $(0,1)$-variable which is equal to one if node i is an element of macronode m, and zero otherwise. Now consider the following optimization problem:

$$\max \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \sum_{m=1}^{M} t_{ij}(x_{im} - x_{jm})^2 - \lambda \sum_{m=1}^{M} \left(\sum_{i=1}^{N} x_{ij} - N/M \right)^2$$

subject to

$$\begin{aligned} \sum_{m=1}^{M} x_{ij} &= 1 \qquad i = 1, \ldots, N \\ x &\in C. \end{aligned}$$

The first term in the objective function is equal to the sum of the lengths of all edges connecting macronodes. Note that maximizing this sum is equivalent to minimizing the sum of the lengths of all edges within macronodes. The second term in the objective function will insure, for a suitably chosen value for the penalty parameter λ, that the cardinality of each of the macronodes is approximately equal to N/M. Alternatively we could choose to include constraints that fix the size of the various macronodes. However, in general it may not be possible to find an aggregation that satisfies the cardinality constraints exactly. Moreover, even if such aggregations exist, they may be undesirable with respect to the first part of the objective function as stated above. Therefore it is preferable to incorporate this constraint in the objective function. The first set of constraints insure that every node is in exactly one macronode. Finally, the set C denotes the set of 0-1 solutions $x = (x_{ij})$ that correspond to aggregations of the network into *connected* components.

Although it is instructive to study the aggregation problem in this form, there can be little hope that this problem can be solved optimally for large-scale networks. One would therefore need to resort to developing heuristics for approximating the solution to this problem.

5 Experimental results

5.1 The decomposition algorithm

In this section we will report some experimental results on the comparison of Dijkstra's algorithm with the decomposition algorithm introduced in section 3, for the case where the number of levels of aggregation is $L = 2$. We have considered networks of the Manhattan type, and we have aggregated the nodes in the obvious way, creating a situation where the macronetwork looks exactly the same as each of the subnetworks inside the macronodes. The distance matrices were randomly generated. In the first experiment, we generated the matrices as follows: if $(i, j) \in A$, then t_{ij} is uniformly and independently distributed on $[0, 1]$. As a measure of the relative error of the approximation algorithm we used the following:

$$\epsilon = \frac{\sum_{i=1}^{N} \sum_{j=1}^{N} (\hat{f}_{ij} - f_{ij})}{\sum_{i=1}^{N} \sum_{j=1}^{N} f_{ij}}$$

where f_{ij} is the exact length of the shortest-path from i to j as found by Dijkstra's algorithm, and $\hat{f}_{ij}$ is the approximate length of the shortest-path from i to j as found by the aggregation algorithm. The error ϵ can be interpreted as the average percent error of an origin-destination pair selected at random. Note that a better aggregate error measure would incorporate information about frequencies of the various trips, to reflect the fact that an error in a very infrequent trip is less important than an error in a frequently occuring trip. However, in the absence of this information we will make the assumption that all origin/destination pairs (i, j) occur with the same frequencies.

In the next experiments we changed the distribution of the distances to simulate metropolitan areas: if we assume that each macronode represents a metropolitan area, the arc lengths within a macronode will generally be smaller than the arc lengths connecting macronodes. To model this, we generate arc lengths within macronodes from the uniform distribution on $[0, 1]$, and arc lengths connecting macronodes from the uniform distribution on $[0, r]$, for varying values of $r > 1$. This model will also illustrate theorem 4.1 from the previous section: the longer the arcs connecting macronodes are (compared to arcs inside macronodes), the smaller the error in shortest-path lengths obtained using the aggregation algorithm should be. The results from the experiments are reported in tables 1 and 2. All entries are averages over 10 runs. The entries in table 1 represent the average relative error ϵ in shortest-path length. In table 2, computation times for Dijkstra's algorithm (sequential implementation as well as implementation using $\sqrt{N}$ processors) and for both phases of the aggregation algorithm (using $\sqrt{N}+1$ processors) are given.

N	$r=1$	$r=N^{\frac{1}{4}}$	$r=\sqrt{N}$	$r=N^{\frac{3}{4}}$
16	0.19	0.09	0.05	0.02
81	0.41	0.18	0.04	0.01
256	0.44	0.20	0.04	0.01
625	0.53	0.21	0.05	0.01

Table 1: Average error in shortest-path lengths

N	Dijkstra sequential	Dijkstra in parallel with $\sqrt{N}$ processors	Aggregation in parallel with $\sqrt{N}+1$ processors
16	0.033	0.008	0.003
81	1.340	0.149	0.020
256	15.260	0.954	0.093
625	100.480	4.019	0.310

Table 2: Computation times (seconds; Macintosh IIfx)

Table 1 confirms the result of theorem 4.1: increasing the value of r decreases the relative error of the shortest-path lengths. Of course it is as yet unclear what a reasonable value of r would be representing real-world networks. The results also show that the difference between edge lengths connecting macronodes and inside macronodes should be larger as the size of the network increases in order to obtain a certain error level. Table 2 shows the computational advantage of the aggregation algorithm over Dijkstra's algorithm.

6 Summary and suggestions for future research

In this paper we have summarized existing methods for solving shortest-path problems. In particular, we have addressed both sequential and parallel algorithms. Next, we have developed a new decomposition algorithm, thereby surrendering the optimality of the solution obtained, but gaining in terms of computational effort and number of processors/computers needed to solve the problem. The idea of the algorithm is to decompose the network into smaller subnetworks, and a macronetwork in which each of the subnetworks is a node. Then all subproblems are solved exactly (in parallel), and the results are combined to obtain approximate shortest-paths for the original network. We have empirically investigated the influence of the decomposition algorithm on the precision of the solution obtained through a simulation study over a class of networks. These results provide hope that acceptable error levels can be attained for a suitable choice of macronodes.

References

Akl, S.G. 1989. *The Design and Analysis of Parallel Algorithms.* Prentice-Hall, Englewood Cliffs, NJ.

Bean, J.C., J.R. Birge, and R.L. Smith. 1987. Aggregation in dynamic programming. *Operations Research* **35**, 215-220.

Bertsekas, D.P., and J.N. Tsitsiklis. 1989. *Parallel and Distributed Computation.* Prentice-Hall, Englewood Cliffs, NJ.

Denardo, E.V. 1982. *Dynamic Programming: Models and Applications.* Prentice-Hall, Englewood Cliffs, NJ.

Dreyfus, S.E., and A.M. Law. 1977. *The Art and Theory of Dynamic Programming.* Academic Press, New York, NY.

Kaufman, D.E., and R.L. Smith. 1993. Fastest paths in time-dependent networks for IVHS applications. *IVHS Journal* **1**(1), 1-11.

Kaufman, D.E., R.L. Smith, and K.E. Wunderlich. 1991. An iterative routing/assignment method for anticipatory real-time route guidance. *IEEE VNIS Conference Proceedings*, Dearborn, MI, October 20-23, 693-700.

Murty, K.G. 1992. *Network Programming.* Prentice-Hall, Englewood Cliffs, NJ.

Quinn, M.J. 1987. *Designing Efficient Algorithms for Parallel Computers.* McGraw-Hill.