Parallel Algorithms and Aggregation for Solving Shortest-Path Problems

H. Edwin Romeijn
Erasmus University, Rotterdam

Robert L. Smith
University of Michigan, Ann Arbor

The University of Michigan
IVHS
Intelligent Vehicle-Highway Systems
College of Engineering • (313) 764-4332
Professor Kan Chen, 4112 EECS, Ann Arbor, MI 48109-2122, and
University of Michigan Transportation Research Institute • (313) 936-1066
Robert D. Ervin, 2901 Baxter Road, Ann Arbor, MI 48109-2150

IVHS Technical Report: #92-06

Parallel Algorithms and Aggregation for Solving Shortest-Path Problems

H. Edwin Romeijn
Department of Operations Research & Tinbergen Institute
Erasmus University Rotterdam
Rotterdam, The Netherlands

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

January 1991
Revised July 1991
Revised February 1992

1 Introduction

In this paper we will review existing algorithms for solving shortest-path problems, and investigate the possibility of using parallel computing and/or aggregation techniques to speed up the algorithms. This problem arises as part of the IVHS (Intelligent Vehicle/Highway Systems) research project in Traffic Modeling. For more on this subject, see Kaufman and Smith (1990).

2 Literature review

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} 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 an acyclic graph, 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:

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

In the remainder we will assume that there are no cycles of negative length in the graph.

(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 $f_{ij}(k) = \min\left(f_{ij}(k-1), f_{ik}(k-1) + f_{kj}(k-1)\right)$
```

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 $2^{\lceil \log(N-1) \rceil} \geq N-1$, we have that $f_{ij} = f_{ij}^{2^{\lceil \log(N-1) \rceil}}$. So, the f-values can be computed recursively using the following algorithm:

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

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 matrix multiplication algorithm. For the latter problem, a variety of parallel algorithms exists, one of which we will discuss in some more detail in the next section.

2.2.2 Matrix multiplication

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

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

In figure 1 we illustrate how this algorithm can be programmed in a parallel fashion by using a mesh-connected parallel computer with N^2 processors.

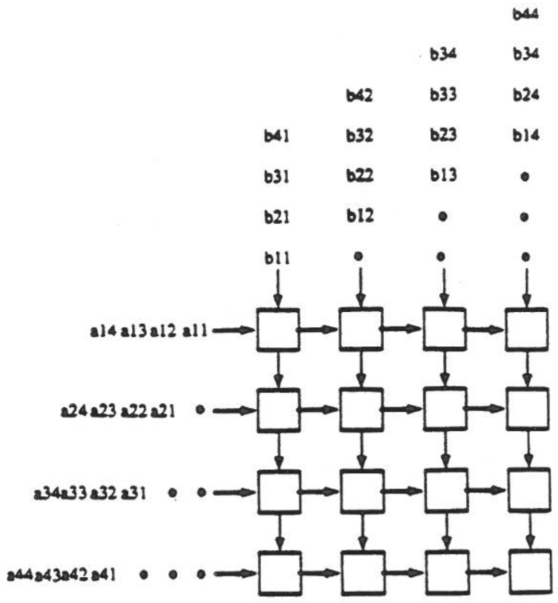

Figure 1: A parallel algorithm for multiplying two matrices

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 of minimum. So, we obtain a parallel algorithm for shortest-path calculation having complexity $O(N \log N)$, while 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 one of the algorithms of this type yields parallel algorithms 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 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, 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 an algorithm 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, it turns out to be possible to reduce the 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 whether we can reduce the number of processors necessary to solve the problem, while keeping the time complexity of the algorithm equal to $O(N \log N)$. Obviously, we will now have to sacrifice something other than time to be able to achieve this goal. In particular, we will give up the goal of obtaining a truly optimal solution.

3.1 A simple model

We will start by considering the following (simplified) model. Let $G = (V, A)$ be a graph, and let every node have exactly 4 neigbors (see figure 2).

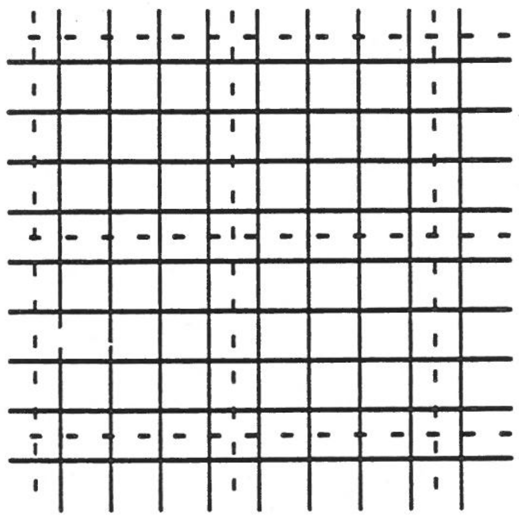

Figure 2: "Manhattan"

Assume the graph (having N nodes) is a $\sqrt{N} \times \sqrt{N}$ mesh. We will aggregate nodes so that we have M "macronodes." Assume that we aggregate in such a way that the "macronetwork" is a $\sqrt{M} \times \sqrt{M}$ mesh, and that every macronode itself is a $\sqrt{N/M} \times \sqrt{N/M}$ mesh. 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, so that the $M + 1$ subproblems can be solved in parallel. Then:

- 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
- 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. We obtain this objective if we choose M in such a way that all subproblems are of the same size; i.e., the number of nodes inside each macronode is equal to the number of macronodes: $N/M = M$, or $M = \sqrt{N}$. So we can conclude that using $M+1 = \sqrt{N}+1$ (or $M = O(\sqrt{N})$) processors, we can obtain an approximate solution to the shortest-path problem in $O(N \log N)$ time. Although the model presented here is very simple, the results still hold if we make the assumption that we only consider partitions of the network into macronodes having the property that

1. each macronode has the same (sparsity) structure as the original network (with respect to average number of neighbors etc.)
2. the macronetwork obtained by aggregating the nodes inside each macronode to form one node also has the same (sparsity) structure as the original network.

If the original network (and the macronodes) is dense, the results concerning the number of macronodes still 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 shortest-path problem that can be obtained in $O(N \log N)$ time consists of a table of shortest-path lengths for each of the macronodes, and 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 macronode. Since the number of nodes on a shortest-path will be $O(N^{1/4})$ on average, this can be performed in $O(N^{5/4})$ time sequentially. Obviously this procedure can easily be parallellized to use $\sqrt{N}$ processors, yielding a time complexity of $O(N^{3/4})$, thereby keeping the complexity of the total algorithm unchanged. 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 using $\sqrt{N}$ processors reduces this to $O(N\sqrt{N})$ time. The complexity of Dijkstra, when implemented using $\sqrt{N}$ processors, is $O(N\sqrt{N} \log N)$. 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 much larger: for instance, in cases where not all shortest-paths are required, or when one needs fast on-line shortest-path information. Note that in practice, when the macronodes correspond to metropolitan areas, many of the shortest-path requests will be for paths *inside* a macronode. This information is readily available, and moreover, for many origin/destination pairs this information will be *exact* instead of *approximate*. 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 Level of aggregation

In the preceding section we only considered the case where aggregation takes place only one level down. In this case we will say that the aggregation level is $L = 2$. It is possible to generalize the results to the case where we consider arbitrary 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 solving exactly a shortest-path problem.

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 $\frac{N-1}{N^{1/L}-1} = O(N^{1-1/L})$ processors, we can obtain an approximate solution to the shortest-path problem in $O(N^{2/L} \log N)$ 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 lenghts 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 processors then gives (after substitution of $M = O(N^{1/L})$): $O(N^{3/2L}) < O(N^{2/L})$. The second phase (when implemented directly, but see the notes in the previous section) 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.

3.3 The Sequential Aggregation Disaggregation Algorithm

In Bean et al. (1987) an algorithm, called SADA (Sequential Aggregation Disaggregation Algorithm) was introduced for computing approximately the shortest-path lengths in an *acyclic* graph using node aggregation. An error analysis was given, including an explicit error bound. We are interested in a generalization of these results to the case of a *cyclic* graph. This will also be a subject of future research.

4 Aggregation in practice

For an arbitrary network it is not clear how one should aggregate nodes into macronodes. 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 the shortest-path between nodes i and j, and let $\hat{f}_{ij}$ denote the length of the path computed by the decomposition algorithm. Furthermore, decompose each of those lengths as follows:*

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

Here a superscript C denotes 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*

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

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 indicates that the network should be aggregated in such a way that edges *within* macronodes are *short*, and edges *connecting* macronodes are *long*. Using this, we can formulate the aggregation problem 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:

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

$$\text{subject to } \sum_{m=1}^{M} x_{ij} = 1 \qquad i = 1, \ldots, N$$

$$x \in C.$$

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 aggregations that correspond to aggregations of the network in *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 efficiently for large-scale networks. In the remainder of this section we will construct a heuristic for the problem where, for the time being, we only take into account the *second* term in the objective function. In other words, we will concentrate on finding a *nearly* feasible solution to the aggregation problem, without paying too much attention to the quality of the solution with respect to the part of the objective function that influences the error in shortest-path lengths obtained using the decomposition algorithm.

The heuristic that we will introduce shortly uses the transformation of the (directed) graph $G = (V, A)$ to an undirected graph, $G' = (V, E)$, where

$$E = \{(i, j) : i \leq j \text{ and } \{(i, j), (j, i)\} \cap A \neq \phi\}.$$

For an undirected graph we define the notion of a spanning tree:

Definition 4.2 *A* spanning tree *in an undirected graph $G = (V, E)$ is a subset S of E such that (V, S) is connected and acyclic.*

The proposed heuristic will consist of two phases:

Phase 1: Find a *spanning tree* in the network.

Phase 2: Decompose the spanning tree into subtrees, each representing a macronode in the network.

Definition 4.3 *If we have a cost function associated with the edges in the network, the* minimal-cost spanning tree *will be a spanning tree such that the sum of the cost of all edges in the tree is minimal.*

In a sparse network, we can find a minimum-cost spanning tree in $O(N \log N)$ time using Kruskal's algorithm (see e.g. Murty, 1992).

Denote the set of nodes by V, and let the set of edges in the spanning tree found in phase 1 be denoted by S. The following heuristic will construct a decomposition of V into macronodes, each given by a set $V_m \subset V$.

Aggregation algorithm

Step 0. $m = 0$.

Step 1. Set

$$m \leftarrow m + 1.$$

Find the longest path (with respect to number of nodes) in the network (V, S). Let this path be $(i_1, \ldots, i_k)$. Candidate macronodes will be of the form:

$$\cup_{p=1}^{\ell} (\{i \in V : (i, i_p) \in S\} \cup \{i_p\})$$

or

$$\cup_{p=\ell}^{k} (\{i \in V : (i, i_p) \in S\} \cup \{i_p\}).$$

Choose the macronode whose cardinality is closest to $\sqrt{N}$. Let this macronode be given by V_m.

Step 2. Set

$$\begin{aligned} V &\leftarrow V - V_m, \text{ and} \\ S &\leftarrow \{(i,j) \in S : i, j \notin V_m\}. \end{aligned}$$

If $V = \phi$, stop. Otherwise, return to step 1.

As mentioned before, this algorithm does not take into account the first part of the objective function of the algorithm described above. However, it is possible to modify the algorithm as follows to try to obtain a good solution with respect to this objective. In the first phase we can try to choose the edge-costs in a way that is favorable towards the objective, and/or we can add constraints to the spanning tree problem to enforce a good aggregation in the second phase. In the second phase of the heuristic we could use a different criterion than just cardinality for distinguishing between different choices of macronodes.

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 networks inside the macronodes. The distance matrices have been randomly generated. Initially, we have generated the matrices as follows: if $(i, j) \in A$, then t_{ij} is uniformly distributed on $[0, 1]$. As a measure of the error of the approximation algorithm we used the following expression:

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

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. 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 error in shortest-path length. In table 2, computation times for Dijkstra's algorithm (sequential implementation and 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

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

N	Dijkstra sequential	Dijkstra $\sqrt{N}$ processors	Aggregation $\sqrt{N}+1$ 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)

the computational advantage of the aggregation algorithm over Dijkstra's algorithm.

5.2 Aggregation in practice

In this section we will show the results of running the aggregation and decomposition algorithms presented in sections 3 and 4 on a more realistic network, and on a "real-world" network. These networks do not have the Manhattan type structure, but they *are* sparse.

The first network has 19 nodes. Choosing an aggregation level of $L = 2$, and following the analysis in section 3.1, we should aim for 4 macronodes. In figure 3 we show the network and the macronodes formed by the aggregation algorithm.

The aggregation algorithm created 5 macronodes, of size $5 - 4 - 4 - 4 - 2$. The target size of the macronodes was $\sqrt{19} \approx 4.36$.

Running the decomposition algorithm for computing shortest-paths yields the following results: the computation time is 0.017 seconds (when implemented *sequentially*), which compares favorably with the computation time for Dijkstra's algorithm: 0.17 seconds (when implemented sequentially). The average error in shortest-path length is equal to 0.14.

The second network is the network for Troy, Michigan. It has 170 nodes,

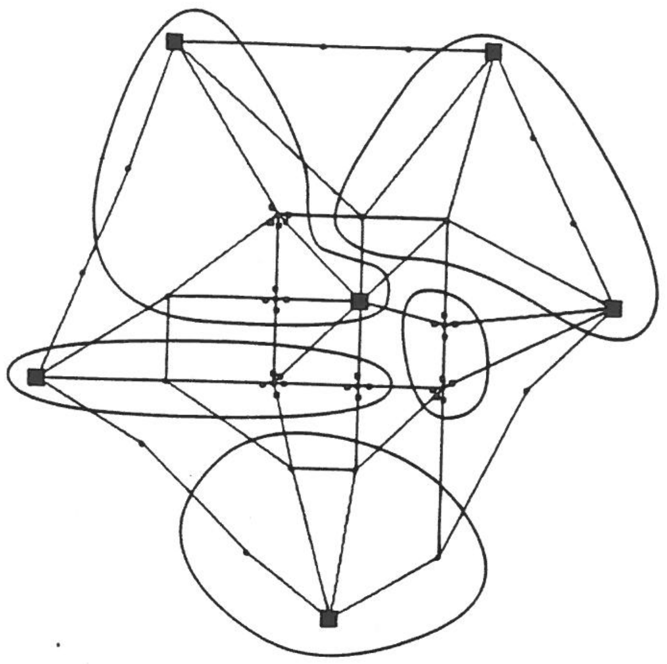

Figure 3: "St. Louis"

which means that the optimal number of macronodes is 13. The aggregation algorithm created 14 macronodes. The first 13 macronodes have sizes between 10 and 14. The last macronode has cardinality 4. The target size of the macronodes was $\sqrt{170} \approx 13.04$.

The sequential computation time for the decomposition algorithm is 0.57 seconds, while computing the exact shortest-paths using Dijkstra's algorithm takes 3.64 seconds. The average error is equal to 0.26.

Concluding, we can say that the aggregation and decomposition algorithms yield satisfactory results. The aggregation algorithm constructs a good decomposition of the network in macronodes, where the size of the macronodes is roughly equal. The decomposition algorithm combines significant computational savings with an acceptable error in the solution.

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 propose a heuristic for constructing the subnetworks. 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. We also tested the preformance of the algorithms on a small, but more realistic, network, and on a "real-world" network. These results indicate that acceptable error levels can be attained for a suitable choice of macronodes. Moreover, they show that the proposed heuristic for constructing macronodes yields such a choice.

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. 1990. Fastest paths in dynamic networks with applications to Intelligent Vehicle/Highway Systems. Working paper, Department of Industrial and Operations Engineering, The University of Michigan, Ann Arbor, MI.

Murty, K.G. 1992. *Network Programming.* To appear.

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