The University of Michigan
IVHS
Intelligent Vehicle-Highway Systems

Notes on Parallel Algorithms and Aggregation for Solving Shortest Path Problems

H. Edwin Romeijn

Department of Operations Research & Tinbergen Institute
Erasmus University
Rotterdam, The Netherlands

and

Robert L. Smith

Department of Industrial and Operations Engineering
University of Michigan
Ann Arbor, Michigan 48109

January 1991

IVHS Technical Report 91 - 03
For Release on May 1, 1992

UNIVERSITY OF MICHIGAN
TRANSPORTATION RESEARCH INSTITUTE • (313) 936-1066
2901 Baxter Road, Ann Arbor, MI 48109-2150, and
COLLEGE OF ENGINEERING • (313) 764-4333
4112 EECS, Ann Arbor, MI 48109-2122

Notes on Parallel Algorithms and Aggregation for Solving Shortest Path Problems

H. Edwin Romeijn

Department of Operations Research & Tinbergen Institute
Erasmus University
Rotterdam, The Netherlands

and

Robert L. Smith

Department of Industrial and Operations Engineering
University of Michigan
Ann Arbor, Michigan 48109

January 1991

IVHS Technical Report 91 - 03
For Release on May 1, 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. For more on this subject, see e.g. 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:

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

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 "k-loop" can be implemented using (a modification of) a matrix multiplication algorithm. For the latter problem, a variety of parallel algorithms exist, 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 ℓ = 1, ..., N DO
    SET c_ij = c_ij + a_iℓ b_ℓj
```

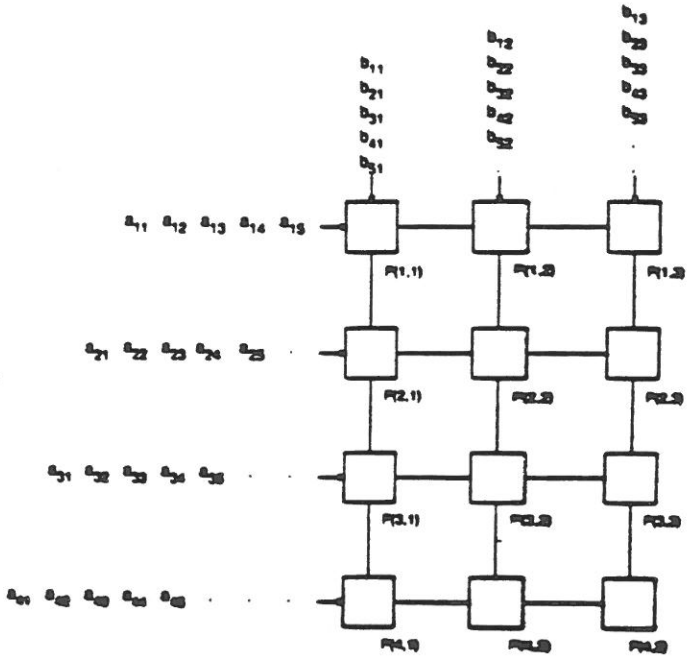

Figure 1: A parallel algorithm for multiplying two matrices[1]

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. 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 (see Akl, 1989).

[1] From Akl (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 N processors, and to let each processor compute all shortest paths from a single origin to all possible destinations. In this way we obtain an algorithm having time complexity $O(N \log N)$ for sparse graphs, and $O(N^2)$ for dense graphs. Note however 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 immensely. 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 neighbors

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 "macro-network" is a $\sqrt{M} \times \sqrt{M}$ mesh, and that every macronode itself is a $\sqrt{N/M} \times \sqrt{N/M}$ mesh.

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. In other words, we should minimize

$$\max\left(O\left(\left(\frac{N}{M}\right)^2 \log\left(\frac{N}{M}\right)\right), O(M^2 \log M)\right).$$

We can minimize this function if we only consider solutions of the form $M = O(N^\alpha)$, for $\alpha \in [0,1]$. Then the problem becomes:

$$\min_\alpha \max\left(O\left(N^{2-2\alpha} \log N\right), O\left(N^{2\alpha} \log N\right)\right)$$

the solution of which is given by: $\alpha = \frac{1}{2}$.

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) are dense, the results concerning the number of macronodes still remains the same, but the time complexity of the algorithm becomes $O(N^2)$ (see also section 2.3 above).

3.2 Level of aggregation

In the preceding section we considered only 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. Using the same type of analysis as above, we obtain that the number of macronodes of the lowest level (i.e. the number of macronodes of the smallest size) should be $O(N^{1/L}) = O(\sqrt[L]{N})$. The running time of the algorithm then becomes equal to $O(N^{2/L} \log N)$.

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. For our purposes 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 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 made a start with trying to use aggregation techniques, thereby surrendering the optimality of the solution obtained, but gaining in terms of computational effort and/or number of processors/computers needed to solve the problem. The main question that has to be addressed in the future is the influence of aggregation on the precision of the solution obtained. This will allow us to make a more rigorous statement about how to aggregate the problem, or even on whether to use aggregation at all.

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.

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.