ALGORITHMS FOR SHORTEST PATHS†

Donald B. Johnson

TR 73-169

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University for the Degree of

Doctor of Philosophy

May 1973

---

†This research was supported by the National Science Foundation under Grant GJ-28176.

ACKNOWLEDGEMENTS

I am grateful to David Gries for his critical reading of the manuscript and to John Hopcroft for helpful conversations during the course of my research.

ALGORITHMS FOR SHORTEST PATHS†

Donald B. Johnson
Cornell University
Ithaca, New York

ABSTRACT

New algorithms are presented for the general all pairs and single source shortest path problems and for the single source problem restricted to nonnegative arc weights. The new algorithms are faster on sparse networks than algorithms previously known. In addition, new results are presented on the behaviour of well-known algorithms, two of which have exponential running times under surprisingly innocuous conditions.

† This research was done with the support of a Graduate Fellowship from the National Science Foundation and with partial support from NSF Grant GJ-28176.

CONTENTS

CONTENTS (Continued)

CONTENTS (Continued)

PART I: BASIC IDEAS AND OVERVIEW

1. Introduction

Finding shortest paths is a fundamental optimization problem. The best routings for vehicles or messages, for instance, are shortest paths with respect to costs associated with the arcs of the transportation or communication networks involved. Many other important network problems involve shortest path computations in their solution.

Shortest path problems are often quite large. Therefore algorithms (such as we present) with better asymptotic time bounds than were known before are of practical as well as theoretical interest. Our bounds are presented in terms of the number of arcs as well as the number of nodes, and show clearly that sparse networks always can be processed faster than complete networks in the worst case.

This work also contributes to the theory of algorithms. Not only do we give some algorithms which have running times which are optimum to within a constant factor, we show that certain known algorithms formerly believed to have polynomial running time have exponential running time under surprisingly innocuous conditions. These results, we feel, give interesting examples of "what makes good algorithms good."

1.1 Summary of Main Results

Our work concerns shortest path algorithms which add arc weights and compare the sums. We present results for single source and all pairs problems. For single source problems, we use adaptive algorithms derivable from Ford's algorithm [For56]. (In an adaptive algorithm the order in which arcs are considered depends on the outcomes of path weight comparisons. In non-adaptive algorithms the order is predetermined.) The measure under which we compare algorithms is running time on a random access computer for the worst case amonq problems of a given size.

1. A class of adaptive algorithms under arc set partitions is introduced (Section 11). This generalization is shown to include all well known algorithms for the single source problem, most of them as cases of the trivial partition of one subset. Yen's algorithm [Yen70] is reformulated as an adaptive algorithm under an arbitrary element of a set of two-subset partitions.

A unique partition is shown which gives an algorithm whose worst case region in problem space is a proper subset of the worst case region for any known algorithm on problems with unrestricted networks. In addition to having a bound of improved order in a region of problem space, the new algorithm has a constant 7/8 of Yen's, hitherto the best known. The new algorithm adapts to networks with "few" negative arcs, reducing to an improved form of Dijkstra's algorithm [Dij59] when there are none.

2. A new algorithm for the all pairs problem is presented which reduces the running time in the worst case from $O(n^3)$ to $O(n^{2+1/k}+n\alpha)$, where α is the number of arcs and $k\geq 1$, independent of n (Section 14.2). This result holds for any network.

3. It is proved that the decomposition algorithm of Hu and Torres [H&T69] can be $O(n^3)$ on networks with as few as $2n$ arcs (Section 14.1). Thus their method is not competitive in general with the above result.

4. It is shown that the Warshall-Floyd algorithm [Flo62a] for the all pairs problem is optimum among all non-adaptive algorithms using adds and compares (Section 13.3). This result is stronger than the previous result of Iri and Nakamori.

5. It is proved that a generalization of Dijkstra's algorithm thought to be $O(n^3)$ on networks with arcs of negative weight is actually $O(n2^n)$ (Section 10.1). A similar result is shown for a straightforward implementation of Ford's algorithm, even for networks with nonnegative arcs (Section 4.1).

6. If the set of active nodes in Dijkstra's algorithm is maintained as a tree, the bound of $O(n^2)$ which holds regardless of α becomes a bound which ranges from $O(n \log n)$ to $O(n^2)$ as α grows from $n-1$ to $n(n-1)$ (Section 9). Using a fixed depth rather than a fixed degree tree gives a bound linear in α for $n^{1+1/k} \leq \alpha \leq n(n-1)$, $k \geq 1$. This bound is optimum within a constant factor and is superior to bounds previously reported. In general, the optimum "shape" for a tree used to maintain a partial ordering depends on the relation between the number of interrogations and the number of additions or updates.

7. An algorithm which runs in the bound $O(n+\alpha)$ is shown to solve the single source problem on nonnegative arc networks which are restricted cyclic, those in which the number of cycles in any strongly connected region is bounded by a constant independent of n and α (Section 9.2). The same bound, $O(n+\alpha)$, is also shown to hold for a decision procedure which verifies a claimed solution to any single source problem. Thus verification is faster than solving a problem in the first place.

8. In the course of presenting the above results, new and known results are related by a unified theory.

1.2 Organization

These pages contain fifteen sections, grouped into four parts. In Part I, the introduction is followed by necessary definitions in Section 2 and a survey of important results relevant to this work in Section 3.

Part II contains a step by step development of our single source results from reasonably basic concepts. Section 4 develops well known algorithms and proves results about them which we need later. It is here that the reader may see that Ford's algorithm is indeed basic to the theory of single source algorithms as we know it. Sections 5 through 8 give both new and known individual results based on the "wave" algorithm introduced in Section 4.

In Section 9 we give the analysis of trees which results in our algorithm which has an optimum time bound for a large class of networks with nonnegative arcs. Section 10 shows that, contrary to folklore widely believed, relaxing the nonnegative arc weight restriction is disastrous indeed.

Section 11 introduces our class of arc set partition algorithms and gives the best algorithm presently known for the unrestricted single source problem. Single source results are summarized briefly in Section 12.

Part III develops results for the all pairs problem. Section 13 covers non-adaptive algorithms, including an optimality result. Section 14 builds directly on the results of Part II, giving a new algorithm for the all pairs problem which runs faster on sparse than on complete networks.

Part IV contains references. Section 1.1 may be referred to for conclusions.

2. Preliminaries

2.1 Networks, Paths, and Problems

A digraph $D = (V,X)$ consists of a nonempty and finite set of nodes V of cardinality n, and a set of arcs $X = \{(u,v) \mid u,v \in V \text{ and } u \neq v\} \subset V\times V$ of cardinality α. We will find occasion to number the elements of the node set $V = \{v_1,v_2,\ldots,v_n\}$, but no ordering is implied by the definitions. Sometimes we will let a node index stand for a node, for example, $V = \{1,2,\ldots,n\}$, or $(i,j) \in X$. In every case the meaning should be clear. Notice that our definitions exclude multigraphs.

Every digraph D has an equivalent (undirected) graph $G = (V,E)$ where $E = \{(u,v) \mid (u,v) \in X \text{ or } (v,u) \in X\}$, a set of unordered pairs of nodes called edges.

A network $N = (D,w)$ is a digraph together with a real-valued function w defined for $(u,v) \in X$. The real number $w(u,v)$ is the weight of arc (u,v).

A path (from v to u) in N or D is a sequence

$$q_{vu} = (v=v_{i_1},x_{j_2},v_{i_2},x_{j_3},v_{i_3},\ldots,x_{j_k},v_{i_k}=u)$$

of nodes alternated with arcs such that $x_{j_{\ell+1}} = (v_{i_\ell},v_{i_{\ell+1}})$ for $1\leq\ell<k$ and $k>0$. As may be seen, the trivial path is allowed. In order to harmonize with the literature on shortest paths we do not require the nodes and arcs of a path to be distinct. A path in which all nodes (except possibly the first and last) are distinct is an elementary path. We will generally denote an elementary path from v to u as p_{vu}. Clearly, for elementary paths, $k \leq n+1$, and it follows that there are finitely many elementary paths in any network. On the other hand, the path set for a network, denoted $\Pi(V \times V)$, may have infinite membership.

We will hereafter let the arcs in a path be implied, representing q_{vu} as the node sequence $(v=v_{i_1}, v_{i_2}, \ldots, v_{i_k}=u)$.

We extend the weight function to paths in the following manner:

$$w(q_{vu}) = \sum_{x \in q_{vu}} w(x) .$$

When $k = 1$, $w(q_{vu}) = 0$, the empty sum.

A directed cycle is a nontrivial path $(u, \ldots, u)$ in which all intermediate nodes are distinct. Any cyclic permutation of a cycle is the same cycle. The trivial path (u) is not a cycle by this definition.

Define $u \, R \, v$ if there exists paths $(u, \ldots, v)$ and $(v, \ldots, u)$. R is an equivalence relation. Any digraph in which $u \, R \, v$ for all $u, v \in V$ is strongly connected. Furthermore, the subgraphs $D_i = (V_i, \{(u,v) \mid (u,v) \in X \text{ and } u,v \in V_i\})$, where V_i is an equivalence class under R, are the strongly connected components (or strong components for short) of D. It will be convenient to let V_i stand for D_i.

A digraph D is acyclic if no path in D is a directed cycle. It is obvious from the definition that an acyclic graph has no strong components. Let $\hat{D}$, the condensation of D, be a digraph with nodes $\hat{v}_i$ corresponding to the strong components V_i of D. There is an arc $(\hat{v}_i,\hat{v}_j)$ in $\hat{D}$ whenever there is an arc in D from some node in V_i to some node in V_j. Since the strong components are maximal strongly connected subgraphs, it follows that any condensation $\hat{D}$ is acyclic.

(2.1) LEMMA. For nodes v and u, an elementary path p_{vu} exists in D if and only if there exists a path q_{vu}.

Proof: We have $\exists p_{vu} \Rightarrow \exists q_{vu}$ by definition. For the reverse implication, let a path q_{vu} be given. If q_{vu} is not elementary, find some repeated node and delete all nodes between the two instances of the repeated node and one of the instances of the repeated node, leaving a new path q_{vu}. Continue the process until some q_{vu} is elementary. $\square$

We call the elementary path p_{vu} produced from q_{vu} by the above process a reduction of q_{vu}. A reduction is not necessarily unique.

A shortest path from v to u is a path q_{vu} such that $w(q_{vu})$ is a minimum over all paths from v to u. Note that the number of arcs is immaterial. A better term would be "minimum weight path." However, the term "shortest path" is entrenched in the literature from the days when, in the only problems considered, "weight" and "length" were interchangeable. We denote the number of arcs in a path q_{vu} by $|q_{vu}|$, the <u>arc length</u>. A path with fewest arcs is <u>arc-shortest</u>.

(2.2) <u>LEMMA</u>. <u>A shortest path from v to u is defined if and only if there exists a path q_{vu} and all such paths include no directed cycle of weight less than zero.</u>

<u>Proof</u>: Let a shortest path p_{vu} be given. Thus there is a path $q_{vu} = p_{vu}$. If there were a path $\hat{q}_{vu}$ containing a cycle of negative weight, another path $\tilde{q}_{vu}$ could be found in which this cycle is repeated a number of times sufficient for $w(\tilde{q}_{vu}) < w(p_{vu})$ contrary to assumption.

Let there be a path q_{vu} and suppose no path from v to u includes a cycle of negative weight. Obviously, if p_{vu} is a reduction of q_{vu}, then $w(p_{vu}) \leq w(q_{vu})$. Thus the weight of a member of a set of elementary paths bounds from below all path weights. Since there can be but a finite number of elementary paths, among them is a path $\hat{p}_{vu}$ such that $w(\hat{p}_{vu}) \leq w(q_{vu})$ for all paths q_{vu}. By definition, $\hat{p}_{vu}$ is a shortest path. □

(2.3) <u>COROLLARY</u>. <u>If there exists a shortest path q_{vu} then there exists an elementary shortest path p_{vu}.</u> □

Let a node v be reachable from u if and only if there is a path q_{uv} .

(2.4) COROLLARY. A shortest path from v to u is defined in an acyclic network for every u reachable from v . □

(2.5) LEMMA. For any shortest path $(i_1,i_2,\ldots,i_k)$, each subpath $(i_j,i_{j+1},\ldots,i_{j+\ell})$, $1 \leq j < j+\ell \leq k$ is a shortest path from i_j to $i_{j+\ell}$. Also if $(i_1, i_2,\ldots,i_k)$ is arc-shortest as well as a shortest path, then all its subpaths are arc-shortest.

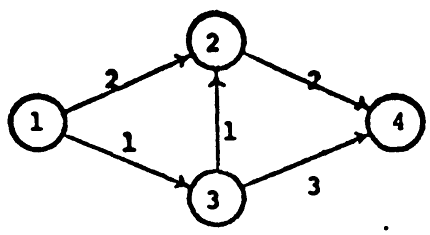
Proof: Assume, to the contrary, that there exists such a subpath which is not shortest (arc-shortest) from i_j to $i_{j+\ell}$. But this implies a shorter (arc-shorter) path from i_1 to i_k . □

A shortest path problem $\phi(N,\psi)$ is a network N and a set ψ of ordered pairs of nodes between which shortest paths are to be sought. Among all possible problems for a given network N we distinguish two, the single source and the all pairs problems. For the single source problem we seek shortest paths from a distinguished source node s to all other nodes, that is, $\psi_s = \{(s,v) \mid v \in (V-\{s\})\}$. We may abbreviate $\psi_s = V - \{s\}$, since s is understood. In the all pairs problem, all node pairs are considered except nodes paired with themselves, giving $\psi_a = \{(u,v) \mid u,v \in V , u \neq v\}$. Thus a single source problem can be stated as $\phi(N,s)$ and an all pairs problem as $\phi(N)$, where ψ is understood. It is a simple matter to extend our definitions to admit problems where shortest cycles are sought as well (recall cycles must be non-trivial). Our problem definitions exclude cycles, however, to simplify the exposition.

The complete solution of a problem $\phi(N,\psi)$ is an assignment $\sigma: \psi \to 2^{\Pi(V\times V)}$ of a subset of the path set $\Pi(V\times V)$ to each element of ψ. If a shortest path is defined for $(v,u) \in \psi$, the set of all shortest paths from v to u is assigned to (v,u). If there is no path at all from v to u, the empty set $\emptyset$ is assigned. If, however, there is a path q_{vu} containing a cycle of negative weight, the assignment σ is undefined for $(v,u) \in \psi$.

A solution of $\phi(N,\psi)$ is a set composed of one element from each set of the complete solution $\sigma(\psi)$. We are setting up a distinction between "a" solution and the complete solution, which is an elementwise union of all solutions. A solution represents a complete solution, but in general some equivalent paths are missing.

Lemma 2.5 suggests that there exist solutions for which the union of solution paths is a tree or a set of trees rooted in the initial nodes of the pairs of ψ. That is, if p_{vu} is presented for (v,u), a subpath of p_{vu} is the path presented for every other appropriate pair in ψ. To illustrate, consider problem $\phi(N,1)$ on the following network:



There are six solutions, two of which are

$$\{((1,2),(1,3,2)),\ ((1,3),(1,3)),\ ((1,4),(1,2,4))\}$$

and

$$\{((1,2),(1,2)),\ ((1,3),(1,3)),\ ((1,4),(1,2,4))\} .$$

The second solution is one of four tree solutions. The algorithms we give always produce a tree solution if a solution is defined at all points.

In the cases of single source and all pairs problems, a tree solution may be represented by a predecessor function $\pi: \psi \to V$. By $\pi(v,u)$ is meant the predecessor of u on a shortest path p_{vu} , that is

$$(2.6) \quad \pi(v,u) = \begin{cases} \hat{u} & \text{if } \exists (v,\ldots,\hat{u},u) \in \sigma(v,u) \\ \Lambda & \text{if } \sigma(v,u) = \emptyset \\ \text{undefined} & \text{if } \sigma(v,u) \text{ is undefined.} \end{cases}$$

Lemma 2.5 assures us that shortest paths are traced in their entirety by $\pi(\psi)$. Strictly speaking, π represents a tree solution, but we will call π a tree solution when convenient. Among all solutions in a complete solution, a tree solution $\pi(\psi)$ is minimal in the sense that there is no solution with fewer arcs.

Complete solutions may be represented by a minimum weight function $\mu : \psi \to \mathbb{R} \cup \{\infty\}$ such that

$$(2.7) \quad \mu(v,u) = \begin{cases} \infty & \text{if } u \text{ not reachable from } v \\ w(p_{vu}) & p_{vu} \text{ a shortest path} \\ -\infty & \text{if there exists } q_{vu} \text{ with a negative cycle,} \end{cases}$$

where $-\infty$ means undefined and satisfies the relation $-\infty < z$ for all $z \in (\mathbb{R} \cup \{\infty\})$.

When discussing single source problems, we will consistently delete references to the source node s , writing p_v for p_{sv} , $\pi(v)$ for $\pi(s,v)$, and $\mu(v)$ for $\mu(s,v)$.

Our algorithms will report π or μ (or both) whenever $\mu(\psi)$ is defined for all pairs in ψ . Otherwise the result of the algorithm will not define π or μ . Thus, for the single source problem, a solution is reported only when no cycle of negative weight is reachable from s . For the all pairs problem, a solution is reported only when no cycle of negative weight exists anywhere.

It is common for authors in this field to give algorithms which report μ only, since some solution of the complete solution always can be generated from μ . One drawback of this approach is that the simple reconstruction methods usually proposed do not guarantee finding a solution tree of minimum depth, yet this is the structure over which some good algorithms compute the sums of μ . In the first algorithms we give, we retain information and give π for the tree actually found by the algorithm. We also show an algorithm, optimum to within a constant, for reconstructing a minimum depth solution from μ . Once this reconstruction algorithm is given, we simplify later algorithms by omitting the statements which trace paths. It should be clear how to insert them.

Some authors (see Knuth [Knu68] for instance) reserve the term "algorithm" for computational procedures which terminate under all inputs. While the distinction is important to us, we choose a looser interpretation of the word "algorithm" as the lesser evil. Otherwise we would have to apply the name "Ford's computational procedure" to the procedure canonized in the literature "Ford's algorithm." Where needed we will prove that some algorithms terminate and identify the class of problems under which others do not. This class will invariably be problems with partial solutions.

2.2 Worst Case Analysis

It is common to report the running times of algorithms as upper bounds which depend on the size of the problem. This is "worst case" analysis. With it there is the assurance that no problem of a certain size will take longer to run than the bound given. Worst case analysis is particularly appropriate for shortest path computations. There are two reasons. Analysis of average running time is difficult, and the concept itself is elusive because it is not clear what a "random" distribution of networks is when arcs of negative weight are allowed but there can be no cycles of negative weight.

The size of a problem $\phi(N,\psi)$ may be defined by the size parameters of the network, $n = |V|$ and $\alpha = |X|$, and a measure of the size of the answer $|\psi|$. For both the single source and all pairs problems, $|\psi|$ is a fixed function of n. Thus we seek time bounds $T(n,\alpha)$ depending on n and α, such that $T(n,\alpha)$ is the time taken by a certain algorithm to solve some problem of size (n,α) and no problem of this size takes longer. These bounds may be translated into $T(n)$ by letting $\alpha = n(n-1)$, the maximum number of arcs possible.

The actual time bound $T(n,\alpha)$ depends on many factors involved in the implementation of an algorithm. There are, however, basic operations which must be implemented in some way. We assume that among reasonable implementations the times consumed by basic operations (adds, compares, indexing, assignment, etc.) will all be related by constant bounds. Under this assumption it is easy to analyze the order of time bounds $T(n,\alpha)$. We say

$$f(x_1,x_2,\ldots,x_n) = O(g(x_1,x_2,\ldots,x_n))$$

when there exists a constant k such that for all $x_1,x_2,\ldots,x_n$

$$|f(x_1,x_2,\ldots,x_n)| \le k|g(x_1,x_2,\ldots,x_n)| .$$

By the same token, it will at times be useful to compare the constant of algorithms with time bounds of the same order by counting only the <u>active operations</u>, adds and compares on path weights. If we let $\tau(n,\alpha)$ be the count of active operations, we will say $\tau(n,\alpha)$ is approximated by $f(n,\alpha)$ (written as $\tau(n,\alpha) \doteq f(n,\alpha)$) if $f(n,\alpha)$ is composed of the highest order terms in $\tau(n,\alpha)$. When the highest order terms in $\tau(n,\alpha)$ are bounded by $f(n,\alpha)$ we write $\tau(n,\alpha) \lessdot f(n,\alpha)$. Clearly, when active operations dominate,

$$T(n,\alpha) \le O(g(n,\alpha)) \iff \tau(n,\alpha) \le O(g(n,\alpha))$$

and

$$\tau(n,\alpha) \lessdot g(n,\alpha) \Rightarrow T(n,\alpha) \le O(g(n,\alpha)) .$$

It is important to notice that a problem $\phi(N,s)$ or $\phi(N)$ is unchanged if someone names or renames the members of V or X. If an algorithm names elements in order to manipulate them, or if input is sequential and thus implies a naming, a worst case is one with the worst possible naming or sequencing. This observation has practical consequences since certain desirable orderings of V are costly to produce. With respect to X, it turns out that any ordering which we will want can be constructed in $O(n+\alpha)$ operations, which is always equaled or dominated by other costs.

There is another aspect to the "size" of a problem. Among the tree solutions $\pi_1, \pi_2, \ldots, \pi_k$ of a given single source problem $\phi(n,s)$, there is, as we have said, a "best" solution tree, one of minimum depth. Thus a problem may have a deep best solution tree or a shallow one. It turns out that the time bounds for some of the algorithms we give are sensitive to the depth of a best solution tree. If we let t be the number of nodes in an arc-longest path from the root of a best solution tree, then we can speak of the time bound $T(t,\alpha)$. Since $t \leq n$, these bounds may be translated into $T(n,\alpha)$ by letting $t = n$. Notice that the depth of the best solution tree is name-independent, as are n and α .

All logarithms are base 2 unless otherwise noted.

2.3 Iteration Theorem

Proving correctness of programs generally involves induction on the execution of loops in the program. A suitable property which can be proved to hold before a loop is entered and to be invariant over any one execution of the statements in the loop must, by induction, hold after any number of iterations which occur. Following the notation used by Hoare [Hor69] we state the result as follows:

(2.8) $P\{\underline{\text{if}}\ B\ \underline{\text{then}}\ S\}P \Rightarrow P\{\underline{\text{while}}\ B\ \underline{\text{do}}\ S\}\ P \wedge \neg B.$

This theorem may be read, "if P is invariant over the conditional statement <u>if</u> B <u>then</u> S , then P and not B holds following <u>while</u> B <u>do</u> S provided P holds at the outset and the <u>while</u>-statement terminates."

An alternative statement which avoids the question of termination of the loop is:

(2.9) P{<u>if</u> B <u>then</u> S}P => P{<u>while</u> B <u>do</u> (S[P])}

which simply states that, given invariance of P over S when B is true, P will hold after any sequence of iterations of S which occur.

Clearly the theorem is true in either form, as may be shown by an elementary inductive argument. We will use the theorem in our treatment to avoid many repetitions of such arguments.

3. Shortest Path Algorithms: Some History and Interrelations

The literature on shortest paths is largely oriented toward algorithms, not toward problems. While this is quite a natural phenomenon in the evolution of algorithms, it tends toward defining problems in the light of the characteristics of known algorithms, thus obscuring the relationships between problems. In particular, a clear relationship between the single source and all pairs problems has not been evident mainly, it appears, because of the dissimilarity between important algorithms for the two problems, Ford's algorithm [For56, Ber58] and related algorithms on the one hand, and the Warshall-Floyd algorithm [Flo62a] on the other. Unifying these streams of development has been hindered by the lack of a good all pairs algorithm directly related to some good single source algorithm. Our work remedies this lack by giving an algorithm, employing a fast single source algorithm as a procedure, which is essentially as fast as algorithms previously known on every all pairs problem and faster on most.

Among single source problems there have been two lines of development as well, originating in Ford's algorithm and Dijkstra's algorithm [Dij59]. We show how the latter may be derived from Ford's algorithm. This approach not only unifies the main single source algorithms in one theory but leads to a new class of single source algorithms under arc set partitions. In this class we find an algorithm superior to any known before.

Having made these observations, we now outline briefly some main developments in each area. The reader is also referred to a paper by Dreyfus [Dry69], the most recent good survey we know of. He cites a number of early references we omit.

3.1 All Pairs and Matrix Algorithms

The reachability implied by a path in a network is a transitive relation. This fact naturally suggests a relation between finding the transitive closure of a Boolean connection matrix and finding shortest paths between all pairs of nodes. However, associating costs with connections requires that from time to time a best path be chosen among alternatives. In computing the transitive closure, if a choice must be made, it can be arbitrary.

Transitive closure can be found by successive "and-or" multiplication as suggested by the definition, A^* such that $A \cdot A^* = A^*$. (It is easily shown that n multiplications suffice.) Analogously, shortest paths between all pairs can be found by successive "plus-min" multiplication. If one defines each node as connected to itself, then successive squarings suffice, and either process is $O(n^3 \log n)$, where each multiplication takes $O(n^3)$ and there must be $\lceil \log(n-1) \rceil$ of them.

In 1962, Warshall [War62] showed how to find the transitive closure of a Boolean matrix in $O(n^3)$ operations. Later in the year, Floyd [Flo62a] extended the method to shortest paths. For shortest paths, Floyd's result is optimal among algorithms with a predetermined sequence of plus and min operations as we show in Section 3.3. Optimality had previously been shown, by Iri and Nakamori [I&N72], using other techniques, for a restricted class of such algorithms which includes the Warshall-Floyd algorithm and a later algorithm due to Dantzig [Dan67].

Warshall's result is not optimal for transitive closure, however, even among algorithms with a predetermined sequence of operations. Munro (see reference in Fischer and Meyer [F&M72]) has shown that transitive closure can be found in "and-or" multiplication time, which is $O(n^{\log_2 7})$ when done by an adaptation of Strassen's algorithm [Str69]. Even taking into account the disparity between bitwise and integer operations, this result is clearly superior to $O(n^3)$. Strassen's algorithm does not apply to shortest path algorithms employing plus and min because it requires a unique inverse under the operation analogous to min in "plus-min" multiplication.

In 1967, a method of cascaded multiplication was proposed by Farbey, Land, and Murchland [FLM67]. In this method, a series of inner products is computed, employing new values as soon as they are available. Their algorithm takes exactly twice the time the Warshall-Floyd algorithm does, which degradation we can attribute to requiring subsequences of plus and min operations to be restricted to computing inner products. Hu [Hu67] also discusses this method, refining the presentation and the proofs.

One problem associated with all matrix algorithms for shortest paths is that they do unnecessary work on incomplete networks, those with fewer than $n(n-1)$ arcs. The standard treatment is to include "dummy" arcs which, consequently, participate in the computation. Hu and Torres [H&T69] (see also Hu [Hu68]) circumvent this necessity by observing that networks in which the node indices can be permuted so as to eliminate connections far from the diagonal of the connection matrix can be treated as a sequence of smaller problems on complete networks. For some networks, their method is $O(n^2)$. However, we show in Section 14.1 that there are networks with as few as $2n-2$ arcs on which Hu and Torres' algorithm is $O(n^3)$. Thus it must be considered $O(n^3)$ in the worst case. We conjecture that, in general, to find a good decomposition costs of the order of computing shortest paths if the decomposition is not known in advance.

Departing from matrix methods, we give in Section 14.2 a new all pairs algorithm which is $O(n\alpha)$ on any network over the range $n^{1+1/k} \leq \alpha \leq n(n-1)$ for some k independent of n. With appropriate refinements, the multiplicative constant of the bound is not much greater than the constant of the Floyd-Warshall algorithm. The new algorithm is not related to any of the matrix algorithms mentioned. Rather it employs good single source algorithms as procedures, in effect doing n+1 single source problems to find a solution on unrestricted networks.

3.2 Single Source Algorithms

A solution to any single source problem is embedded in a solution to the all pairs problem on the same network, provided shortest paths are defined for every pair. Under this condition, then, to find any desired single source solution it is sufficient to solve an all pairs problem. However, more efficient algorithms are known for the single source than for the all pairs problem. These algorithms exploit the fact that for any single source problem with a solution there is always a tree solution rooted in the source node.

The Warshall-Floyd algorithm concatenates a pair of paths whose weights are known in order to find (possibly) a path with a lesser weight than has been computed already for some pair. The tree property of single source solutions allows a specialization of this operation: concatenation of single arcs to paths of known weight originating in the source node. Actually, this idea predates Warshall's. It was employed in 1956 by Ford [For56, Ber58] in what is generally regarded as the first good single source algorithm. All other good single source algorithms we know can be derived from Ford's algorithm as special cases employing particular strategies to order the choice of arcs to concatenate to paths. In Section 4, the important lines of derivation are shown.

In 1957, Moore presented a conference paper employing the same idea in an algorithm with a breadth first strategy, one which chooses together for concatenation all arcs originating in a single node. His work was published in 1959 [Mor59]. Both Ford's and Moore's algorithms are correct on networks with negative as well as nonnegative arcs, as long as no cycle reachable from the source node has negative weight. However, in the references cited, the analysis is confined to networks with nonnegative arcs. In fact, Berge's proof ([Ber58], pp. 68-69) relies on this property. The broader applicability of Ford's algorithm is recognized in a later reference [F&F62].

Ford leaves to the implementor of his algorithm the choice of a strategy by which to order the choice of arcs for concatenation. This is a significant omission. In Section 4.1 we show that at least one apparently good strategy gives an exponential running time in the worst case, even when negative arcs are forbidden. However, under a strategy such as was employed by Bellman in an independent presentation of a related algorithm [Bel58], Ford's algorithm is $O(n^3)$ in the worst case, as is Moore's algorithm. Bellman's algorithm is tied to a matrix representation for the given network, so it always operates on a complete network, absent arcs being represented by arcs of infinite weight. Thus Bellman's algorithm is always $O(n^3)$. A list representation for the network allows a strategy under which Ford's algorithm runs in $O(t\alpha) \leq O(n\alpha)$ operations, as is described in Section 4.2.

Ford's algorithm does not detect negative cycles. In fact, both Ford's and Moore's algorithms may fail to terminate if there is a negative cycle. Bellman's algorithm always terminates, but it is not possible in all cases to tell if the solution is correct unless it is known independently that there are no negative cycles.

It is a fairly simple matter to modify Ford's algorithm so that it counts operations and thereby detects a non-terminating computation which signals a negative cycle. The algorithm described in Section 4.2 is so modified. More complicated algorithms which detect negative cycles have been given by Dantzig, Blattner, and Rao [DBR67a] and by Yen [Yen70]. Of these, Yen's is superior in the multiplicative constant of the worst case bound, $O(n^3)$. However, his algorithm has a larger constant when there are negative cycles than when there are not. In Section 11 we give Yen's algorithm in the context of our class of arc set partition algorithms, of which a similar algorithm is a member. We also give an algorithm of ours, an algorithm under the arc set partition $X_p \cup X_m$, the arcs of nonnegative and negative weight, respectively. This algorithm detects negative cycles in exactly the same bound in which it computes shortest paths, and the multiplicative constant of the bound is slightly better than Yen's. More important, our algorithm is faster in order than other algorithms on a large class of networks, those with "few" negative arcs. It reduces to a variant of Dijkstra's algorithm (see below) when all arcs are nonnegative.

The above discussion applies to networks which may have negative as well as nonnegative arcs. For networks with only nonnegative arcs (distances in a more conventional sense) there are faster algorithms than the algorithms of other workers mentioned above. In 1959, Dijkstra [Dij59] proposed a breadth first algorithm which runs in $O(n^2)$ operations in the worst case. His algorithm is a special case of the general breadth first or "wave" algorithm on which we build several of our results. In particular, in Section 9.1.2 we give an algorithm which runs in $O(\alpha)$ time over the range $n^{1+1/k} \leq \alpha \leq n(n-1)$ provided there are no negative arcs. This work is closely related to and extends the work of Murchland [Mur69] and E. Johnson [Joh72]. Dantzig [Dan60] also gives an algorithm which is better than Dijkstra's when α is quite small, but is worse otherwise. The bound, however, is everywhere inferior to ours.

What distinguishes Dijkstra's algorithm is that it employs a breadth first strategy in which the node from which the search proceeds at each step is one for which the shortest path weight has been found. Among all the nodes which are "active" in the breadth first search, the weight of the one with least weight is a shortest path weight, provided there are no arcs of negative weight. The rule of choice Dijkstra employs thus allows a permanently marked node to be identified at each stage of the algorithm. This property does not carry over to unrestricted networks, however. In Section 10 we show that using the least weight (Dijkstra's) rule gives an exponential running time in the worst case with negative arcs.

Good selection rules in the "wave" algorithm are the key element in much of our work. In particular, it is being able to identify arc set partitions for which a good rule exists for each element of the partition that enables us to make certain arc set partition algorithms fast.

The literature is virtually silent on the important question of the utility of general algorithms. While it is easy to think of problems which are described by networks with nonnegative arcs, examples of problems for which general algorithms are needed do not come to mind so easily. We know of one important class of problems for which a general, negative cycle-detecting algorithm is needed, that of finding optimum cycles in doubly-weighted networks. In such a network, each arc has two weights, a transit cost and a transit time, for example. An optimum cycle is one in which the ratio of the sum of one class of weights to the sum of the other class over the cycle is optimized. For instance, one could maximize profit per unit time. The best method known for solving this problem [Law72] generates a sequence of problems for which a negative cycle-detecting algorithm is needed (see also [Law67] and [DBR67b]).

3.3 Single Pair Algorithms

Obviously a single pair solution is embedded in a solution to the appropriate single source and all pairs problems. Pohl has shown [Poh69] that expected running time can be decreased on single pair problems below that for single source algorithms in which the algorithm terminates as soon as the solution for the desired pair is identified. However, his algorithm is inferior in the worst case and applies only to nonnegatively weighted networks.

PART II: SINGLE SOURCE ALGORITHMS

4. The Basic Algorithm

The best algorithms known for the single source problem concatenate arcs to subpaths in order to find new paths. As soon as a new path is found, its weight is compared to the weight of an old path (if there is one) and the path of greater weight is discarded.

We now develop this idea into a basic algorithm to which known algorithms can be related and on which many of our results will be built. We will analyze this algorithm rather carefully because it will be possible to use many of its properties to understand the more sophisticated algorithms which can be derived from it.

Let d and f be n-element array variables defined in some algorithm. The elements of d will record the weights of paths some part of which is traced by f. That is, $f(v)$ will contain the predecessor of v on some path, of weight $d(v)$, from s to v (provided $d(v) < \infty$, of course). If at termination $d(v) = \mu(v)$ for all $v \neq s$, we will have a correct solution. That is, the pointer chain in f will trace a shortest path in the reverse direction from v to s.

Let $B(X)$ be the property

(4.1) $B(X) \equiv \exists (u,v) \in X$ such that $d(v) > d(u) + w(u,v)$.

When $B(X)$ is true there is an arc in X which can be used to reduce some point in d .

(4.2) LEMMA. For all $d: v \to (\mathbb{R} \cup \{\infty\})$ such that $d(v) = w(q_v)$, where q_v is some finite path from s to v ,

$$\neg B(X) \iff \forall v (d(v) = \mu(v)) .$$

Proof: Let $\neg B(X)$ be given. Assume there exists v such that $d(v) \neq \mu(v)$. If $d(v) < \mu(v)$ there is a path with weight less than $\mu(v)$, which is impossible. Thus $d(v) > \mu(v)$. This implies $\mu(v) < \infty$.

If $\mu(v)$ is defined there must be a path

$$p_v = (s=v_{i_1}, v_{i_2}, \ldots, v_{i_k}=v)$$

such that $w(p_v) = \mu(v)$. However, $d(v) > \mu(v)$. So let j be the minimum index such that $d(v_{i_j}) > \mu(v_{i_j})$. Clearly $j > 1$. Thus $d(v_{i_j}) > d(v_{i_{j-1}}) + w(v_{i_{j-1}}, v_{i_j})$, which is a contradiction of $\neg B(X)$.

If $\mu(v)$ is not defined $(= -\infty)$, there must be some cycle of negative weight

$$q_{vv} = (v=v_{i_1}, v_{i_2}, \ldots, v_{i_k}, v_{i_1}=v).$$

But $\neg B(X)$ implies $d(v_{i_{\ell+1}}) = d(v_{i_\ell}) + w(v_{i_\ell}, v_{i_{\ell+1}})$, for $1 \leq \ell < k$ and $d(v_{i_1}) = d(v_{i_k}) + w(v_{i_k}, v_{i_1})$.

Summing over the cycle gives

$$d(v_{i_1}) = d(v_{i_1}) + w(q_{vv}) \quad .$$

in contradiction to $w(q_{vv}) < 0$.

For the reverse implication, let $\forall v(d(v) = \mu(v))$ be given. If $B(X)$ holds there is an arc (u,v) such that $d(v) > d(u) + w(u,v)$ implying a path p_v from s with $w(p_v) < \mu(v)$, which cannot be. □

Lemma 4.2 suggests the following algorithm:

(4.3) ALGORITHM B: Basic Single Source Algorithm

(1) [Initialize] d(v):= ∞ for v ≠ s; d(s):= 0;
f(v):= Λ for all v;

(2) [Search and Replace]

while B(X) **do**

(4.4) $S_1 \equiv$

```
for some arc (u,v)
    satisfying B(X) do
  begin
    d(v):= d(u)+w(u,v);
    f(v):= u
  end
```

Every good algorithm known for the single source problem has the fundamental properties of Algorithm B. All weights recorded in d are weights of paths from s . When $\mu(v)$ is defined for all v , these paths are elementary. The algorithm enumerates elementary paths in some sequence of sufficient length to guarantee that shortest paths have been found for every node. When $\mu(v)$ is undefined for some v, a search for a reducing arc will always succeed, in effect enumerating paths from the infinite set $\Pi(V\times V)$. Thus, knowledge of a time bound sufficient to enumerate a sequence of elementary paths, which contains a shortest path for each node, can be used to detect problems for which a full solution cannot be found.

Let P be the property

$$(4.5)\qquad P \equiv \forall v\left(d(v) = \begin{cases} \infty & \text{or} \\ w(q_v) & \text{where } q_v=(s,\dots,f(v),v) \text{ is a finite path} \end{cases}\right).$$

Property P says that if $d(v) < \infty$, $d(v)$ is the weight of some finite path from s whose last arc is $(f(v),v)$. This property is preserved by S_1 (4.4), as the following lemma shows.

(4.6) <u>LEMMA</u>. $P\{$<u>while</u> $B(X)$ <u>do</u> $(S_1[P])\}(P\wedge\neg B)$.

<u>Proof</u>: We first prove

$$P\{\underline{\text{if}}\ B(X)\ \underline{\text{then}}\ S_1\}P$$

from which the lemma follows immediately by application of the iteration theorem (2.8 and 2.9).

If $B(X)$ does not hold, the result is trivial since S_1 is not executed and we naturally forbid the evaluation of $B(X)$ to alter the assignments d and f.

When $B(X)$ holds, let (u,v) satisfy $B(X)$.. We know that $d(u) \neq \infty$ because for all $y \in \mathbb{R}$, $\infty + y = \infty$. Therefore $d(u) = w(s,\ldots,u)$, a path, because P holds before S_1. Hence, $(s,\ldots,u,v,)$ is also a path whose weight $w(s,\ldots,u,v) = w(s,\ldots,u) + w(u,v)$. We complete the proof by observing that $f(v)$ becomes u by execution of S_1. □

We now prove that a shortest path is properly recorded in f for every node v for which $d(v)$ is a shortest path weight. It should be noted that, in many cases, f does not record correctly paths for u with weight $d(u) > \mu(u)$.

Let Q be the property

$$(4.7) \quad Q \equiv \forall v \left(d(v) = \mu(v) \Rightarrow d(v) = \begin{cases} \infty & \text{or} \\ w(s{=}v_{i_1}, v_{i_2}, \ldots, v_{i_k}{=}v) \\ \quad \text{where } v_{i_j} = f(v_{i_{j+1}}), 1 \le j < k \end{cases} \right).$$

(4.8) <u>LEMMA</u>. <u>If</u> $\mu(v)$ <u>is defined for all</u> v,

$$Q\{\underline{\text{while}}\ B(X)\ \underline{\text{do}}\ (S_1[Q])\}(Q \wedge \neg B).$$

<u>Proof</u>: As before, the result is proved by proving $Q\{\underline{\text{if}}\ B(X)\ \underline{\text{do}}\ S_1\}Q$, which is trivially true if $\neg B(X)$. So, let (u,v) be the arc satisfying $B(X)$ which S_1 employs to reduce $d(v)$ and assign u to $f(v)$. As before, we find $d(u) < \infty$. There are two cases.

If execution of S_1 leaves $d(v) > \mu(v)$, Q cannot depend on $f(v)$. If it did, this fact would imply the existence of some path $p_{\hat{v}} = p_{sv}p_{v\hat{v}}$ such that $w(p_{\hat{v}}) = \mu(\hat{v})$ but $w(p_{sv}) > \mu(v)$, a contradiction of Lemma 2.5.

In the second case, S_1 leaves $d(v) = \mu(v)$, and we know by the same argument that Q did not depend on $f(v)$ before S_1 . Therefore $f(v)$ can be allowed to change value. In addition, it must be that $d(u) = \mu(u)$ before S_1 . By Q , then, there is a shortest path to u correctly recorded in f . Execution of S_1 extends this path correctly to v . □

(4.9) <u>LEMMA</u>. <u>Algorithm B terminates if and only if</u> $\mu(v)$ <u>is defined for all</u> v .

<u>Proof</u>: By Lemma 4.6, the condition in Lemma 4.2 on the assignment d is satisfied. Termination implies $\neg B(X)$ which in turn implies $\forall v(d(v) = \mu(v))$. This condition can only occur in finite time if $\mu(v)$ is defined for all v .

Given $\mu(v)$ defined for all v , Lemma 4.6 can be strengthened to assert that $d(v)$ is always the weight of some elementary path or ∞ . There are finitely many such paths in any finite network, and each iteration changes some $d(v)$. Termination must therefore occur. □

It is clear from Lemma 4.2 that termination implies that d equals μ for a complete solution. By Lemma 4.8 we conclude that f equals a solution π . Furthermore, Lemmas 4.2 and 4.6 imply that $(B(X) \wedge P)$ is invariant over all iterations of Algorithm B when termination does not occur. This gives us two theorems.

(4.10) THEOREM. *If* $\mu(v)$ *is defined for all* v , *Algorithm B terminates with* $d(v) = \mu(v)$, $f(v) = \pi(v)$, *for all* v *and some solution represented by* π . □

(4.11) THEOREM. *If, for some* v, $\mu(v)$ *is not defined, Algorithm B does not terminate, and at the end of any iteration* $(B(X) \wedge P)$ *holds, that is,*

$$P\{\underline{\text{Repeat}}\ (S_1[B(X) \wedge P])\ \underline{\text{until}}\ \neg B(X)\}\ \underline{\text{holds}}.\ \square$$

Algorithm B is fundamental, but not very useful. Not only can it fail to terminate, it says nothing about how to evaluate $B(X)$. We now refine Algorithm B to produce some useful and well-known algorithms. First we take the obvious approach and use an exhaustive search of X to evaluate $B(X)$. Refinements under exhaustive searching give Ford's algorithm [For56] (see also [Ber58] pp. 68-69), Bellman's strategy in Ford's algorithm [Bel58], and a better algorithm than either as an intermediate result.

Next we consider sufficient searches which are not exhaustive. Refinements in this direction from Algorithm B lead to arc set partition algorithms (Section 11) and to the wave algorithm and its special cases, Dijkstra's algorithm [Dij59] (Section 9) and Moore's algorithm [Mor59] (Section 10.2). Each refinement is a substitution of one process (possibly only a test) for another. To preserve Theorem 4.10 for the refined algorithms we will have to show that the path properties of the assignment d are preserved and, in the case of refinements of control predicates, that termination implies $\neg B(X)$. Since f is changed exactly where d is, the correct recording of a solution tree π will follow.

For those refinements under which we wish to preserve Theorem 4.11, it will be sufficient to show that the path property is preserved. Refinements will be made in steps for which simple arguments will suffice to show the preservation of necessary properties.

4.1 Ford's Algorithm

There is no requirement for Algorithm B to use the same process to evaluate $B(X)$ as it does to select a reducing arc for the execution of S_1 when $B(X)$ is true. It is natural, however, to combine these processes by evaluating $B(X)$ by searching X , testing each arc (u,v) in turn for the property $d(v) \geq d(u) + w(u,v)$. If no such arc is found, this implies $\neg B(X)$ and S_1 is not executed. Otherwise, any arc for which the property holds may be "remembered" for use in S_1 . With this refinement we obtain possibly the earliest shortest path algorithm to be published.

(4.12) **ALGORITHM F: Ford's Algorithm [For56, Ber58]**

```
(1) [Initialize]  d(v):= ∞ for v≠s; d(s):= 0;
                  f(v):= Λ for all v;

(2) [Search and Replace]
      repeat
        begin
          search for an arc (u,v) satisfying B(X);
          if the search succeeds then
            begin d(v):= d(u)+w(u,v); f(v):= u end
        end
      until the search fails
```

If the search of X is finite and exhaustive, Theorems 4.10 and 4.11 will hold for Algorithm F because the search succeeds if and only if $B(X)$, and because conditions of the assignment d are unchanged.

It is sometimes claimed (see for instance [Dry69], [Yen70]) that Ford's algorithm has a bound $T(n) = 0(n^3)$, presumably because any "sensible" way to implement a search of X is thought to give an algorithm with this bound. The algorithm, however, can be exponential under very simple search strategies. Consider the following "procedural" implementation of Ford's algorithm in which $X = (X_1, X_2, \ldots, X_\alpha)$ is an ordering of X and $\mathrm{first}(X_i) = u$, $\mathrm{second}(X_i) = v$ when $X_i = (u,v)$.

(4.13)
```
procedure Ford;
begin
  procedure search(u,v,found);
    begin
      i:= 0;
      repeat
        begin
          i:= i+1; u:= first(X_i); v:= second(X_i)
        end
      until (d(v)>d(u)+w(u,v)) or (i≥α);
      found:= (d(v)>d(u)+w(u,v))
    end search
  for all v≠s do
    begin d(v):= ∞; f(v):= Λ end;
  d(s):= 0; f(s):= Λ;
  repeat
    begin
      search(u,v,found);
      if found then
        begin d(v):= d(u)+w(u,v); f(v):= u end
    end
  until ¬found
end Ford;
```

If we assume that the given network and the arrays d and f are somehow globally supplied it is clear that this procedure is a straightforward implementation of Ford's algorithm employing the simplest possible exhaustive search of X. We use it to prove the following theorem.

(4.14) THEOREM. *Ford's algorithm can realize a running time* $T(n) \geq O(2^n)$.

<u>Proof</u>: Let N_n be a network with n nodes such that $X = \{(i,j) | 1 \le j < i \le n\}$ and

$$w(2,1) = 1$$
$$w(i,j) = 2^{i-j} w(i-1,1) \;,\quad 1 \le j < i < n,\ i > 2$$
$$w(n,n-1) \ge 2w(n-1,1) \;,\quad n > 2$$
$$w(n,j-1) \ge 2w(n,j) \;,\quad n > 2,\ 1 < j < n \;.$$

For example, the weights for N_5 are

w: i \ j	1	2	3	4	5
1					
2	1				
3	4	2			
4	32	16	8		
5	$\ge 2w(5,2)$	$\ge 2w(5,3)$	$\ge 2w(5,4)$	≥ 64	

We claim that the "procedural" Ford's algorithm (4.13) invokes the procedure search 2^{n-1} times on N_n when X is ordered by columns as follows:

$$X = ((2,1),(3,1),\ldots,(n,1),$$
$$(3,2),(4,2),\ldots,(n,2),$$
$$\ldots,(n,n-1)),$$

and the problem is posed with $s = n$.

Each call but the last to search results in an assignment of a value in d . Let a sequence S_n be formed from the values so assigned. For N_5 , the sequence begins

$$S_5 = (w(5,1), w(5,2), w(5,2)+1, w(5,3), w(5,3)+4, \\ w(5,3)+2, w(5,3)+3, \ldots)$$

For any N_n we claim there are exactly $2^{n-1}-1$ elements in S_n. This hypothesis can be verified directly for $n = 2$; we get $S_2 = (1)$.

Assume the hypothesis for all $N_1, N_2, \ldots N_n$, and consider the result of running the algorithm on any N_{n+1}. Since arc $(n+1,n)$ is the only path from s to n, it is clear that $w(n+1,n)$ will appear somewhere in the sequence S_{n+1}. Consider the sequence of searches before this point. It must be identical to that on some $\hat{N}_n$ in which row n is row $n+1$ from N_{n+1}. Thus $2^{n-1}-1$ entries in S_{n+1} precede $w(n+1,n)$.

By the nonincreasing property of d we know that the remaining elements in S_{n+1} will not involve any weights on row $n+1$ except $w(n+1,n)$. Since $w(n+1,n)$ plus any $n-2$ weights from rows above row $n+1$ is less than any $d(i)$, $i < n$, it is clear that the remainder of S_{n+1} will be a sequence identical to that formed by some $\tilde{N}_n$ but with $d(s) = w(n+1,n)$ at the start.

We conclude that S_{n+1} has $2^{n-1} - 1 + 1 + 2^{n-1} - 1 = 2^n - 1$ elements, and by induction, that there are 2^{n-1} calls on search for any N_n. □

It appears that an exhaustive search which always starts at the beginning of some fixed ordering of X is an extremely unfortunate choice. What is evidently required is a search which retains some information from previous searches. As the next section shows, remembering the point at which the last search left off is sufficient to yield an $O(n^3)$ algorithm.

4.2 Refinements with Exhaustive Search

To get some algorithms with good bounds on running times, we can refine Algorithm F (4.12). For the present we confine ourselves to search strategies which are potentially exhaustive in each invocation. Later we will develop algorithms which retain information from path weight comparisons. This retention allows the scope of the search to be reduced to a subset of X .

It is clear that if we evaluate B(X) with a search as below, found <=> B(X) will hold on termination of the search.

```
found:= false;
repeat
  begin
    choose (u,v) ∈ X;
    if d(v)>d(u)+w(u,v) then found:= true
  end
until found or all arcs in X have been chosen     .
```

We could substitute this search procedure directly into Algorithm F since testing on found can determine if the search succeeded. Notice, however, that the assignment statement d(v):= d(u) + w(u,v) is executed only if and immediately after found has been set true. Thus we can move the assignment statement into the search loop, giving

```
        (2)  [Search and Replace]
               repeat
                 begin
                   found:= false;
                   while ¬found and
                         some arc in X is unchosen do
                     begin
                       choose(u,v) ∈ X;
                     ⎧ if d(v)>d(u)+w(u,v) then
                     ⎪   begin
                     ⎪     found:= true;
(4.15)       S2 ≡    ⎨     d(v):= d(u)+w(u,v);
                     ⎪     f(v):= u
                     ⎩   end
                     end
                 end
               until ¬found
```

Theorems 4.10 and 4.11 will hold when the above refinement is substituted into Algorithm F (4.12), if choosing $(u,v) \in X$ is a finite process which, when repeated, eventually chooses every arc in X .

Now consider a sufficient bound y for some rule of choice so that every arc will be chosen within y choices. With y so defined, Theorems 4.10 and 4.11 will hold for Algorithm F with the following refinement:

(2) [Search and Replace]

```
repeat
  begin
    found:= false; count:= 1;
    while count≤y do
      begin
        choose (u,v) ∈ X;
        S2 (4.15);
        count:= count+1
      end
  end
until ¬found
```

To find a sufficient value for y, let the rule for choosing $(u,v) \in X$ be

choose X_{count}

in some ordering $X = (X_1, X_2, \ldots, X_\alpha)$. The first α choices will be exhaustive, so $y = \alpha$ is sufficient under this rule of choice. Let us use these ideas to rewrite the inner loop once more. In addition we introduce a variable pathlength which counts the number of entries to the inner loop.

(4.16) $S_3 \equiv$

```
pathlength:= pathlength+1;
found:= false; count:= 1;
while count≤α do
  begin
    for (u,v)=X_count do S2 (4.15);
    count:= count+1
  end .
```

It is clear that Theorems 4.10 and 4.11 hold for Algorithm F (4.12) in which Step (2) is replaced by

```
(2)  [Search and Replace]
       repeat
         begin
           S3 (4.16)
         end
       until ¬found
```

and the variable pathlength is ignored.

To bound the outer loop, define the property R as

(4.17) $R \equiv d(v) = \mu(v)$ for all v for which there exists a shortest path p_v such that $|p_v| \leq$ pathlength.

(4.18) LEMMA. If $\mu(v)$ is defined for all v then $R\{S_3\}R$.

Proof: We need consider only nodes v such that the arc-shortest shortest path p_v has exactly pathlength + 1 arcs. By assumption, for some such path $p_v = (s=v_{i_1}, v_{i_2}, \ldots, v_{i_\ell}, v)$, it is true that $d(v_{i_\ell}) = \mu(v_{i_\ell})$, so S_3 (4.16) will set $d(v) = \mu(v)$ and $f(v) = v_{i_\ell}$, since it tests every arc. □

The lemma and the preceeding discussion suggest

(4.19) ALGORITHM BF: Best Exhaustive Search in Ford's Algorithm .

(1) [Initialize] $d(v) := \infty$ for $v \neq s$; $d(s) := 0$;
$f(v) := \Lambda$ for all v;
pathlength := 0;

(2) [Search and Replace]
<u>repeat</u>
<u>begin</u> S_3 (4.16) <u>end</u>
<u>until</u> $\neg$found <u>or</u> pathlength$\geq$n .

This algorithm runs in time proportional to the depth t of a shortest path tree of least depth.

(4.20) <u>THEOREM.</u> <u>Algorithm BF terminates</u>
(i) <u>in</u> $O(t\alpha)$ <u>steps with</u> $\neg$found <u>and</u> $d(v) = \mu(v)$, $f(v) = \pi(v)$ <u>for all</u> v <u>and some least depth tree</u> π <u>if</u> μ <u>is everywhere defined</u>,
(ii) <u>in</u> $O(n\alpha)$ <u>steps with</u> found <u>if</u> μ <u>is undefined for some</u> v.

<u>Proof</u>: Part (ii) is immediate since we have Theorem 4.11 for the algorithm without the bound pathlength $\leq$ n.

For part (i) we first prove

$$R\{\underline{\text{repeat}}\ S_3\ \underline{\text{until}}\ \neg\text{found}\ \underline{\text{or}}\ \text{pathlength} \geq t-1\}(R \wedge \neg B(X)),$$

which follows directly from Lemma 4.18, the iteration theorem, and the fact that for all v there is some shortest path p_v such that $|p_v| \leq t-1$. If also $\neg$found holds we have our theorem, since Theorem 4.10 holds without the bound pathlength $\leq n$ . .

On the other hand, found $\Rightarrow$ (pathlength $\geq$ t-1) , and (pathlength $\geq$ t-1) $\wedge$ R $\Rightarrow$ ($d(v) = \mu(v)$ for all v) , since every v must have a shortest path with no more than t-1 arcs. From Lemma 4.2 we know that $\forall v(d(v) = \mu(v)) \Rightarrow \neg B(X)$, and the rest of the result follows because $\neg B(X)$ occurs when pathlength < t so found is set <u>false</u> and remains <u>false</u> in the final iteration. □

This is the best result known under an exhaustive search strategy. A relaxation of the bounds in Algorithm BF leads to Bellman's algorithm [Bel58], a derivative of Ford's algorithm with explicit iteration indices. Merely delete the variable found so that (pathlength $\geq$ n-1) $\Rightarrow \neg B(X)$. Then observe that $\alpha < n^2$, so it is permissible to insert dummy arcs of infinite weight into X giving a complete network. Correctness is not affected since ∞ is the identity element under minimization. This gives the following algorithm for which $T(n,\alpha) = O(n^3)$.

(4.21) Bellman's Algorithm [Bel58]

```
d(v):= ∞ for v ≠ s; f(v):= Λ, for all v;
d(s):= 0;
for k:= 1 until n-1 do
  for i:= 1 until n do
    for j:= 1 until n do
      begin
        d(v_j):= min(d(v_j),d(v_i)+w(v_i,v_j));
        if d(v_j) changes then f(v_j):= v_i
      end
```

This algorithm always consumes $O(n^3)$ operations, and $\mu(v)$ undefined can only be detected if a negative cycle on a path to v includes s. Detection of this incomplete sort may be accomplished by testing $d(s)$ against zero after termination.

4.3 Restricted Search Strategies, Adaptive Algorithms

Consider some set A of arcs such that $B(A) <=> B(X)$. Clearly the lemmas and theorems which hold for Algorithm B (4.3) will hold under the refinement:

(4.22) (2) [Search and Replace] <u>while</u> B(A) <u>do</u> S_1.

We may let A vary in membership during execution as long as A always contains at least one arc, if one exists, satisfying B(X).

Existence of a set A with the desired property allows the body of Algorithm F to be restated, replacing X with A. Theorems 4.10 and 4.11 are preserved.

(4.23) (2) [Search and Replace]

```
repeat
  begin
    search for an arc (u,v) satisfying B(A);
    if the search succeeds then
      begin
        d(v):= d(u) + w(u,v);
        f(v):= u
      end
  end
until the search fails
```

Clearly any arc in A, once it is tested, no longer satisfies $B(A)$, while any arc originating in some node whose assignment d is reduced may possibly satisfy $B(A)$. We may thus define

(4.24) $A \equiv \{(u,v) \mid (u,v) \in X$ and (u,v) has not been tested for satisfaction of $B(A)$ since $d(u)$ was last set to a value (not ∞) either in initialization or the body of the algorithn$\}$.

Under this definition, $(A = \emptyset) \Rightarrow \neg B(X) \Leftrightarrow$ (the search will fail). Thus we may further refine Algorithm F (4.12), preserving Theorems 4.10 and 4.11.

```
(4.25)  (1)  [Initialize]  d(v):= ∞ for v ≠ s; d(s):= 0;
                           f(v):= Λ for all v;
                           A:= {(s,v) | (s,v) ∈ X};

        (2)  [Search and Replace]
               while A ≠ ∅ do
                 begin
                   search for an arc (u,v) satisfying B(A);
                   if the search succeeds then
                     begin
                       d(v):= d(u) + w(u,v);
                       f(v):= u;
                       A:= (A - {(u,v)}) ∪ {(v,v̂) | (v,v̂) ∈ X}
                     end
                 end
```

As before, we can define the search process as a sequence of choices. Also, the operations performed with the first (u,v) to satisfy $B(A)$ can be moved into the search loop.

We may invoke the same arguments used when refining under an exhaustive search strategy in Section 4.2 to verify that Theorems 4.10 and 4.11 hold for

(4.26) ALGORITHM S: Basic Selective Search Algorithm

```
(1) [Initialize]  d(v):= ∞ for v ≠ s; d(s):= 0;
                  f(v) = Λ for all v;
                  A:= {(s,v) | (s,v) ∈ X};
(2) [Search and Replace]
      while A ≠ ∅ do
        begin
          choose (u,v) ∈ A; A:= A-{(u,v)};
          if d(v)>d(u)+w(u,v) then
            begin
              d(v):= d(u)+w(u,v);
              f(v):= u;
              A:= A ∪ {(v,v̂) | (v,v̂) ∈ X}

      end
```

Algorithm S is an adaptive algorithm under any rule of choice since the composition of A depends on the given problem $\phi(N,s)$.

From this point we proceed with a line of development which leads to Dijkstra's algorithm [Dij59] and to a variety of related results, some known, some new. Later, in Section 11, we return to Algorithm S to develop our theory of adaptive algorithms under arc set partitions.

We can now consider simple ways of representing A . If we agree to choose elements in A so that all arcs originating from a given node are chosen in sequence, we can represent A as a set U of originating nodes. This is merely imposing a local ordering on A , so Theorems 4.10 and 4.11 are preserved so long as the rule of choice can exhaust A . Thus we may define

(4.27) U = {u|d(u) has been assigned a new value (not ∞) since the arcs originating in u were last chosen to test for satisfaction of B(A)}.

Notice that $(U = \emptyset) \Rightarrow \neg B(X)$.

With this last refinement we arrive at

(4.28) ALGORITHM W: Basic Wave Algorithm

```
(1) [Initialize]  d(v):= ∞ for v ≠ s; d(s):= 0;
                  f(v):= Λ for all v;
                  U:= {s};
(2) [Search and Replace]
                while U ≠ ∅ do
                  begin
                (2a) choose u ∈ U; U:= U-{u};
                (2b) for each (u,v) ∈ X do
                (2b1)  if d(v) > d(u)+w(u,v) then
                          begin
                (2b1a)      d(v):= d(u)+w(u,v);
                (2b1b)      f(v):= u;
                (2b1c)      U:= U ∪ {v}
                          end
                  end
```

(4.29) $S_4 \equiv$ (steps (2a)–(2b1c) above, from begin to end)

By the arguments given it is clear that Theorems 4.10 and 4.11 hold for Algorithm W.

The descriptive term "wave" arises from the "breadth first" nature of Algorithm W. When a node u in U is a site of activity, all arcs from u are looked at. In fact, the algorithm would be correct if all arcs from u were tested in parallel.

There is a close analogy between Algorithms F (4.12) and W. Both fail to terminate if some $\mu(v)$ is undefined, and both terminate correctly otherwise. Furthermore, both can have exponential running times under simple rules of choice, as we have shown in Section 4.1 for Algorithm F and will show for Algorithm W in Section 9.1.

What is new about Algorithm W are the properties of set U. Together with d and f, it retains "all we need to know" about earlier execution. This notion is formalized below in two technical lemmas, 4.31 and 4.33. The first of the lemmas states, roughly, that exploration of paths originating in the nodes of U is sufficient to complete a solution of the problem. Lemma 4.33 states that all further computation is confined exclusively to such paths.

Let Y be a set variable (the name in an algorithm of a sequence of sets generated during execution), and define the property $P(Y)$ as follows:

(4.30) $P(Y) \equiv$ for all v such that $d(v) \neq \mu(v)$

there exists a shortest path

$p_v = (s = v_{i_1}, v_{i_2}, \ldots, v_{i_j}, \ldots, v_{i_\ell} = v)$

such that

(i) $d(v_{i_j}) = \mu(v_{i_j}) < \infty$

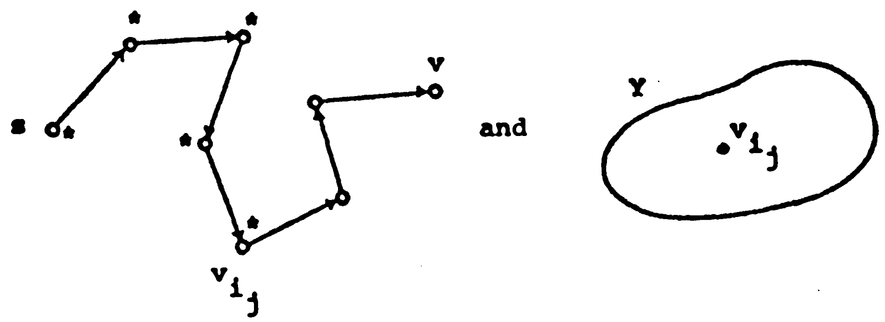
(ii) $f(v_{i_k}) = v_{i_{k-1}}, \; 1 < k \leq j$

(iii) $d(v_{i_k}) > \mu(v_{i_k}), \; j < k \leq \ell$

(iv) $v_{i_j} \in Y$

The property $P(Y)$ may be visualized in the diagram below. Starred nodes are those for which d records a shortest path weight.

$d(v) \neq \mu(v) \Rightarrow$



(4.31) <u>LEMMA</u>. <u>If</u> $\mu(v)$ <u>is</u> <u>defined</u> <u>for</u> <u>all</u> v <u>then</u>

$P(U)$ {<u>while</u> $U \neq \emptyset$ <u>do</u> $(S_4[P(U)])$}

<u>Proof</u>: It is sufficient to prove

$P(U)$ {<u>if</u> $U \neq \emptyset$ <u>then</u> S_4} $P(U)$.

Assume to the contrary that there is a node v for which the property is not preserved. This implies that on the given path p_v some node v_{i_k} beyond v_{i_j} has $d(v_{i_k})$ set to $\mu(v_{i_k})$. Without loss of generality let v_{i_k} be the last such node on p_v. Lemma 4.8 implies a shortest path $(s = v_{m_1}, v_{m_2}, \ldots, v_{m_p} = v_{i_k})$ such that $f(v_{m_r}) = v_{m_{r-1}}$, $1 < r \leq p$. But we must conclude $(s = v_{m_1}, v_{m_2}, \ldots, v_{i_k}, v_{i_{k+1}}, \ldots, v_{i_\ell} = v)$ satisfies $P(U)$, a contradiction. Therefore, $P(U)$ must hold after S_4 (4.29). □

In order to discuss the construct named by a certain variable at different points during execution of an algorithm we employ superscripts. Thus, Y^i means the set named by Y at time i. We let the unsuperscripted variable Y mean the set Y at the present moment of discourse. In $P(Y)$ above, for instance, we mean the set Y at the time we evaluate P.

We now define a property R on Y and Y°, sets at the moment of evaluation and at some previous time, respectively.

(4.32) $R(Y,Y^\circ) \equiv Y \subseteq \{v \mid$ for some $u \in Y^\circ$ there exists a path q_{uv} such that $d^\circ(u) + w(q_{uv}) = d(v)\}$, where $d^\circ(u)$ is the value of $d(u)$ at the time $Y = Y^\circ$.

(4.33) <u>LEMMA</u>. <u>Under any input</u>

$$(U^\circ{=}U)\{\underline{\text{while}}\ U \neq \emptyset\ \underline{\text{do}}\ (S_4[R(U,U^\circ)]\}.$$

<u>Proof</u>: Since $(U^\circ{=}U) \Rightarrow R(U,U^\circ)$ it suffices to prove

$$R(U,U^\circ)\{\underline{\text{if}}\ U \neq \emptyset\ \underline{\text{then}}\ S_4\}R(U,U^\circ).$$

Consider a node v added to U over S_4 (4.29). Let $d(v) = z$ before v is appended to U. Let u be chosen in Step (2a) of S_4. By $R(U,U^*)$ there exists $\hat{u} \in U^*$ such that $d^*(\hat{u}) + w(q_{\hat{u}u}) = d(u)$. The execution of S_4 sets $d(v) = d(u) + w(u,v)$. Since $w(q_{\hat{u}u}) + w(u,v) = w(q_{\hat{u}v})$ we get by substitution $d^*(\hat{u}) + w(q_{\hat{u}v}) = d(v)$, which satisfies $R(U,U^*)$ for v. $\square$

Theorem 4.10 tells us that Algorithm W (4.28) gives correct answers whenever $\mu(v)$ is defined for all v. In particular, it is correct for any exhaustive (with respect to U) and finite selection rule in Step (2a). The particular rule chosen will affect the running time. If we always select a node u such that u will never return to U, there will be at most n iterations of Step (2). Lemma 4.31 assures us that such a choice can always be made. We will consider in following sections <u>how</u> to make such a choice.

Let the cost of executing Step (2a) (choosing and deleting u from U) be a monotonic function $c(|U|)$. Let the cost of each iteration of Step (2b) be a monotonic function $b(|U|)$. Clearly $c(|U|) \leq c(n)$ and $b(|U|) \leq b(n)$. Notice that the cost in Step (2b) includes choosing "the next arc" originating in u and, possibly, adjoining some v to U, as well as other operations clearly of constant cost. The cost of choosing the next arc originating in u can be made in constant time if X is represented as a set of lists, one for each node, of the arc originating in that node. We assume such a representation for X is provided, for if it is not, we can generate it in $O(n+\alpha)$ operations, a bound which will never dominate the other costs in the algorithms we give.

(4.34) LEMMA. *If Step (2a) always chooses* $u \in U$ *such that* u *does not return to* U, *Algorithm W (4.28) runs in time*

$$T(n,\alpha) \le O\left(\sum_{i=1}^{n} c(|U_i|) + \sum_{i=1}^{\alpha} b(|U_i|)\right) \le O(nc(n) + \alpha b(n)).$$

Proof: Step (2) can have at most n iterations, since there can be no more than n deletions from U. Thus the cost of (2a) is bounded by $O(\sum_{i=1}^{n} c(|U_i|))$. Step (2b1) is executed at most once for each arc. This implies an overall cost for (2b) bounded by $O(\sum_{i=1}^{\alpha} b(|U_i|)$. Initialization and loop control are $O(n)$. □

(4.35) COROLLARY. *If Step (2a) always chooses* $u \in U$ *such that* $d(u) = \mu(u)$, *Algorithm W (4.28) runs in time*

$$T(n,\alpha) \le O\left(\sum_{i=1}^{n} c(|U_i|) + \sum_{i=1}^{\alpha} b(|U_i|)\right) \le O(nc(n) + \alpha b(n)). \quad \square$$

Algorithm W, even under rules of choice which guarantee no node returns to U, does not give a solution tree of minimum depth. Simple counter examples can be constructed. However, an additional restriction on the rule of choice gives the desired result.

(4.36) <u>LEMMA. Among all nodes in</u> U <u>which will not return, there is a node</u> u <u>for which the path recorded in</u> f <u>is arc-shortest among all</u> $v \in U$. <u>If such a node</u> u <u>is always chosen,</u> f <u>will equal a minimum depth solution</u> π <u>on termination of Algorithm W (4.28).</u>

<u>Proof</u>: Let Q be the property

(4.37) $Q \equiv$ for all v such that $d(v) = \mu(v)$
f records an arc-shortest path p_v, and
for any such p_v and for all $\hat{v}$ such
that $d(\hat{v}) > \mu(\hat{v})$ there is no shortest
path $p_{\hat{v}}$ such that $|p_{\hat{v}}| \leq |p_v|$.

It is clear that if Q holds, the paths which satisfy P(U) (4.30) can be arc-shortest. Let them be so, and also such that f records the prefix (in property P(U)) $(s = v_{i_1}, v_{i_2}, \ldots, v_{i_j})$. We must prove

(4.38) $P(U) \wedge Q\{$<u>while</u> $U \neq \emptyset$ <u>do</u> $(S_4[P(U) \wedge Q])\}$.

If S_5 (4.29) reduces d , at point v_{i_k}, to $\mu(v_{i_k})$ then the path recorded in f for v_{i_k} must be arc-shortest if Q holds before S_4 . This gives Q following S_4 , and it is again clear that the paths which satisfy P(U) can be arc-shortest. Application of the iteration theorem proves (4.38). But $(U \neq \emptyset) \wedge Q$ implies f records arc-shortest paths for all v . □

5. A Decision Procedure

We have good time bounds for Algorithm W (4.28) provided we can always identify efficiently some $u \in U$ which will not return to U. This result leads directly to a decision procedure for shortest paths. Given a problem $\phi(N,s)$ and an assignment $m: V \to \mathbb{R}$, the assertion to be verified is $\forall v(m(v) = \mu(v))$, that is, we must verify that m represents a complete solution to $\phi(N,s)$. There is decision procedure for this problem which runs in $O(n+\alpha)$ operations, as we prove below.

This decision procedure is optimum to within a constant, since any procedure must expend $O(n+\alpha)$ operations just to examine the input. In contrast, solutions can be <u>computed</u> in this time bound only for restricted classes of problems such as acyclic networks (Section 7), networks with nonnegative arc weights and either restricted cycle structure or more than $n^{1+1/k}$ arcs, $k>1$ (Section 9.2 and 9.1.2), and problems with solution trees of fixed depths independent of n (Section (4.2).

The $O(n+\alpha)$ bound is achieved by partitioning U dynamically into two subsets, U_m and U_r in an algorithm similar to Algorithm W. Whenever a node v would be put into U, $d(v)$ is compared with $m(v)$. If $d(v) = m(v)$, v is put into U_m and removed (if necessary) from U_r. Otherwise v is put in U_r. The rule of choice is simply "choose any $u \in U_m$." Clearly no such u can return to $U = U_m \cup U_r$ since subsequent changes to $d(u)$ guarantee $d(u) \neq m(u)$. Return to U_r is prevented by detecting the error $d(v) < m(v)$.

(5.1) ALGORITHM D: A Decision Procedure for the Single Source Problem

```
(1) [Initialize]  (1a)  d(v):= ∞ for v ≠ s; d(s):= 0;
                        U_m:= {s};
                  (*1b) if m(s) ≠ 0 then reject;
(2) [Search and Replace]
            while U_m ≠ Ø do
              begin
      (2a)      choose u ∈ U_m; U_m:= U_m-{u};
      (2b)      for each (u,v) ∈ X do
      (2b1)       if d(v)>d(u)+w(u,v) then
                    begin
      (2b1a)          d(v):= d(u)+w(u,v);
      (*2b1b)         if d(v)<m(v) then reject;
      (2b1c)          if d(v)=m(v) then
                        begin
                          U_m:= U_m ∪ {v};
                          U_r:= U_r ∩ (V-{v})
                        end
                      else U_r:= U_r ∪ {v}
                    end
              end;
(*3) [Check] for each v ∈ V do
               if d(v)≠m(v) then reject;
             if U_r=Ø then accept
                      else reject
```

(5.2) THEOREM. *Algorithm D runs in* $T(n,\alpha) = O(n+\alpha)$ *operations; it accepts a decision problem* $(\phi(N,S),m)$ *if and only if* $\forall v(m(v) = \mu(v))$.

Proof: Consider Algorithm $\hat{D}$ obtained from D by deleting the starred statements. Let $U = U_r \cup U_m$. Algorithm $\hat{D}$ is identical to Algorithm W (4.28) except for the control expression $U_m \neq \emptyset$ of Step (2). Since $U_m \subseteq U$ we conclude that $\hat{D}$ terminates if W does, and that at all times $d(v) \geq \mu(v)$, for all v .

The addition of the starred statements does not affect this conclusion since each starred statement is finite and does not change d . Moreover, Algorithm D always terminates since any node can be admitted to U_m only once, and each iteration deletes a node from U_m . Because the starred statements cost no more than a constant times n overall, and since $U_m \subseteq U$, we can extend Lemma 4.34 to Algorithm D. Evidently c and b of the lemma are constant functions. Thus it is proved that D terminates with $d(v) \geq \mu(v)$, for all v , in exactly $O(n+\alpha)$ operations, (since examining every arc and node requires at least $O(n+\alpha)$ operations).

Assume $m(v) = \mu(v)$ for all v . Any node v , such that $d(v) = m(v) < \infty$, will belong to U_m some time and cannot belong to U_r after being in U_m . We can extend Lemma 4.31 to D and thus conclude $B(X) \Rightarrow (U_m \neq \emptyset)$. Also, (*2b1b) will never cause rejection since $d(v) \geq m(v)$ for all

v . Thus (*3) will be reached with $(U_m = \emptyset) \Rightarrow$ (by Lemma 4.31) $\neg B(X) \Rightarrow$ (by Lemma 4.2) $\forall v(d(v) = \mu(v)) \Rightarrow$ (by above argument) $(U_r = \emptyset) \Rightarrow$ (D accepts).

On the other hand, assume $m(v) \neq \mu(v)$ for some v . There are two cases. Either $m(v) < \mu(v)$ or $m(v) > \mu(v)$. In the second case it may be that $\mu(v)$ is undefined $(= -\infty)$.

If $m(v) < \mu(v)$, rejection will occur in (*3) since $d(v) \geq \mu(v)$ for all v .

If $m(v) > \mu(v)$ we need consider only the assumption that, when (*3) is reached, $d(u) = m(u)$ for all u . This implies that $m(u) \geq \mu(u)$ for all u . Consider a true shortest path for v , $p_v = (s=v_{i_1}, v_{i_2}, \ldots, v_{i_k}=v)$. Let v_{i_j} be the first node on p_v for which $m(v_{i_j}) > \mu(v_{i_j})$. At some time $v_{i_{j-1}}$ must have been in U_m and have been chosen in (2a). But this would have caused $d(v_{i_j})$ to be set to $\mu(v_{i_j})$ in Step (2b), contrary to assumption. □

6. A Constructor for Minimal Depth Solutions

Consider Algorithm D (5.1) of the previous section. If the asserted $m: V \to \mathbb{R}$ is known to be correct, the starred statements can be removed. Then, if the statements of Algorithm W (4.28) which record f are reinserted, and if the rule of choice in (2a) becomes

> "choose $u \in U_m$ such that f records an arc-shortest path among all $v \in U_m$,"

we have by Theorem 5.2 and Lemma 4.36 a procedure which finds a solution tree of minimum depth for a complete solution represented by m. The body of this procedure is

(6.1) (2) [Search and Replace]

(6.2) $S_5 \equiv$

```
while Um ≠ ∅ do
  begin
    choose u ∈ Um such that psu recorded
    in f is arc-minimal; Um:= Um-{u};
    for each (u,v) ∈ X do
      if d(v)>d(u)+w(u,v) then
        begin
          d(v):= d(u)+w(u,v); f(v):= u;
          if d(v)=m(v) then
            begin
              Um:= Um ∪ {v};
              Ur:= Ur ∩ (V-{v})
            end
            else Ur:= Ur ∪ {v}
        end
  end
```

If U_m is organized as a queue, the rule of choice and the insertion procedure can both be implemented in constant time. This assertion is proved with the aid of

(6.3) $R \equiv$ if queue U_m is nonempty, it is composed of a sequence of nodes with paths from s recorded in f of equal length k followed, possibly, by a sequence with paths recorded in f of arc length $k+1$.

Clearly, $R\{\underline{\text{if}}\ U_m \neq \emptyset\ \underline{\text{then}}\ S_5\}R$ holds, giving

(6.4) THEOREM. *There exists a procedure to find a solution tree of minimal depth, from any* m *representing a complete solution, which runs in* $T(n,\alpha) = O(n+\alpha)$ *operations*. □

This result is optimum to within a constant factor because the input is $O(n+\alpha)$ in size.

Now that we have this result, it is clear that solution trees of minimum depth can always be had if any solution can be. In the remaining sections we will not record path predecessors in f . In every case it will be clear where the statements to do so could be inserted.

7. Acyclic Networks

For any pair of nodes (u,v) in an acyclic digraph, if there is a path from u to v there can be no path from v to u (otherwise a directed cycle would exist). Let $u \succeq v$ denote "there is no path from v to u." Clearly $\succeq$ induces a partial ordering on V.

Let $v_{i_1} \succeq v_{i_2} \succeq \cdots \succeq v_{i_n}$ be a partial ordering of V. From Lemma 4.33 it is clear that we can implement as follows a rule of choice which will guarantee that the node chosen will not return to U:

$$\text{"choose } v_{i_j} \in U \text{ such that } j = \min(k \mid v_{i_k} \in U)."$$

Shortest paths are always defined in an acyclic network connected from s (Corollary 2.4). Thus, Algorithm W (4.28) always terminates on such networks. Our rule of choice will always succeed when $U \neq \emptyset$, so we conclude that Algorithm W under this rule is correct on acyclic networks.

To implement the rule of choice we need only construct a procedure which advances through an ordering of V on $\succeq$ until it finds a node in U, and then on the next search resumes where it left off. Thus we have

(7.1) ALGORITHM A: Shortest Path in Acyclic Networks

(1) [Initialize]

(1a) Generate an indexing $(i_1, i_2, \ldots, i_n)$ such that $j < k$ if $(v_{i_j}, v_{i_k}) \in X$; <u>if</u> such an indexing does not exist <u>then</u> report "failure";

(1b) $d(v) := \infty$ for $v \neq s$; $d(s) := 0$; $f(v) := \Lambda$ for all v; $U := \{s\}$; $j := 0$;

(2) [Search and Replace]

<u>while</u> $U \neq \emptyset$ <u>do</u>

S_4 (4.29) (in which the rule of choice in (2a) is implemented in the following way:

<u>repeat</u> $j := j+1$
<u>until</u> $(v_{i_j} \in U)$;
choose $u = v_{i_j}$;
$U := U - \{u\}$)

We have already argued correctness. Since the indexing required in Step (1a) can be found in $O(n)$ operations (see Knuth [Knu68] pp. 258-265), and the added work for searching V for members of U is also at most $O(n)$, Lemma 4.34 gives us

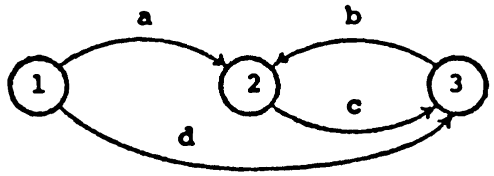
(7.2) <u>THEOREM.</u> <u>Algorithm A runs in</u> $T(n,\alpha) = O(n+\alpha)$ <u>operations and solves any single source problem on an acyclic network or reports</u> "failure" <u>if there is a cycle</u>. □

We use this well known result in algorithms we develop in Section 11, where we employ Algorithm A on subnetworks which are acyclic.

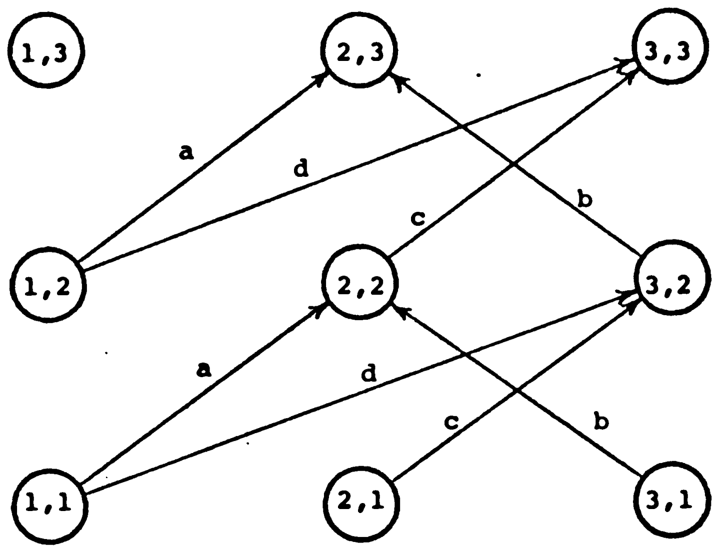
Given an arbitrary network N, an acyclic network $\hat{N}$ may be defined on which Algorithm A may solve a single source problem with a solution related to that on N. Define $\hat{N}$, a network with n^2 nodes, as $\hat{N} = (\hat{V},\hat{X})$ where

$$\hat{V} = \{v_{i,j} \mid v_i \in V,\ 1 \le j \le n\}$$
$$\hat{X} = \{(v_{i,j},v_{k,j+1}) \mid (v_i,v_k) \in X,\ 1 \le j < n\}$$
$$w(v_{i,j},v_{k,j+1}) = w(v_i,v_k),\ (v_i,v_k) \in X,\ 1 \le j < n.$$

$\hat{N}$ is acyclic, and any path $p_v = (s=v_{i_1},v_{i_2},\ldots,v_{i_k}=v)$ in N has an image $\hat{p}_v = (v_{i_1,1},v_{i_2,2},\ldots,v_{i_k,k})$ in $\hat{N}$ and $w(p_v) = w(\hat{p}_v)$. For example, if N is the network



then $\hat{N}$ is



A solution for N is obtained from a solution for $\hat{N}$ as follows:

$$d(v_{i_j}) = \min_k (\hat{d}(v_{i_j,k}).$$

Alternatively, additional arcs of zero weight can be supplied.

$$(v_{i,j}, v_{i,j+1}),\ 1 \leq i \leq n,\ 1 \leq j < n$$

to connect replicas in $\hat{N}$ of nodes in N. Now a solution for N is simply

$$d(v_{i_j}) = \hat{d}(v_{i_j,n}).$$

The construction of $\hat{N}$ costs $O(n^2+n\alpha)$, while running Algorithm A on $\hat{N}$ costs $O(n^2+n\alpha)$, so the bound is comparable when $n=t$ for that on Algorithm BF (4.19). We do not, however, have the advantage of early termination when $t < n$, that is, when there is a shallow solution tree. In comparably clever implementations, the constant for Algorithm BF should be better too.

8. Uniform Positive Arc Weight Networks (arc-shortest paths)

Let a network be defined in which every arc has the same weight. If the network is acyclic the algorithm of the previous section can be used to find shortest paths. If there are cycles reachable from s, a solution to the shortest path problem will be everywhere defined if and only if the uniform arc weight is non-negative.

If this weight is zero, the minimum depth constructor of Section 6 may be used. For any positive weight, the shortest path problem is equivalent to the arc-shortest path problem.

Let set U in Algorithm W (4.28) be maintained as a queue so that it takes constant time to adjoin nodes to U and to choose the oldest node in U. We need only prove that a first-in-first-out rule of choice guarantees no node will return to U and we will have an $O(n+\alpha)$ algorithm for the uniform positive arc weight problem. Any solution will be of minimal depth.

Let the uniform arc weight be $k > 0$ and let R be the property

(8.1) $R \equiv U$ contains nodes of some arc distance j from s followed possibly by nodes of arc distance $j+1$.

It is evident that $R\{\text{if } U \neq \emptyset \text{ then } S_4 \text{ (4.29)}\}R$ holds. Since R is true when $U = \{s\}$, the desired result follows.

Berge [Ber58] gives an equivalent result.

9. Nonnegative Arc Weight Networks

Lemma 4.34 promises a good bound for Algorithm W (4.28) whenever there is an efficient rule of choice which guarantees no node will return to U . In preceding sections we found that practical rules exist when the solution is "known," and when problems are restricted to acyclic networks or to networks with a fixed positive arc weight. Later, in Section 14, we treat a result which allows the bound of Lemma 4.34 to be realized when a solution to a dual problem is known.

Good bounds are also obtained when networks are allowed to have no arcs of negative weight. This case is particularly important because it includes, for instance, all networks in which weights are in fact distances in a metric space, or are costs such as time.

9.1 Dijkstra's Selection Rule

In 1959, Dijkstra [Dij59] proposed an algorithm with a rule we call Dijkstra's selection rule which always picks a node u for which $d(u) = \mu(u)$ provided there are no arcs of negative weight. The rule is

(9.1) "choose $u \in U$ such that $d(u) = \min_{v \in U} (d(v))$."

(9.2) LEMMA. *Dijkstra's selection rule in Algorithm W (4.28) guarantees that, once chosen, no node will return to U provided the problem $\phi(N,s)$ has no* arcs of negative weight reachable from s.

Proof: Assume to the contrary that some node $\hat{v}$ returns to U. Let U^* be U when $\hat{v}$ is first selected. By the rule of choice, $d^*(v) \geq d^*(\hat{v})$ for all $v \in U$. When $\hat{v}$ first returns to U, its weight $d(\hat{v})$ must satisfy $d(\hat{v}) < d^*(\hat{v})$. But by Lemma 4.33, there exists u such that

$$d^*(u) + w(q_{u\hat{v}}) = d(\hat{v}) < d^*(\hat{v}) \leq d^*(u)$$

implying the existence of a path $q_{u\hat{v}}$ such that $w(q_{u\hat{v}}) < 0$, which would require at least one arc of negative weight. □

9.1.1 Unordered Set Realization

Algorithms employing Dijkstra's rule of choice commonly have been analyzed assuming a search time linear in $|U|$ for finding a minimum $d(v)$. Under this assumption we have

(9.3) THEOREM. If U is maintained unordered with respect to d, Algorithm W (4.28) with Dijkstra's selection rule (9.1) runs in $T(n,\alpha) = O(n^2)$ operations or $\tau(n,\alpha) \leq 1/2\, n^2 + 2\alpha$ active operations on any problem $\phi(N,s)$ for which N has no arcs of negative weight reachable from s and $\alpha \geq n-1$.

Proof: It takes $O(|U|)$ operations to find a minimizing $d(v)$, since every member of U must be tested. If $\alpha \geq n-1$ it is possible that $|U| = n-1$ following the first iteration. Thus

$|U|$ will decrease by one at each following iteration giving for the first term in the bound of Lemma 4.34

$$\sum_{i=1}^{n} c(|U_i|) \leq k_1 \sum_{i=1}^{n} |U_i| = k_1(1+\sum_{i=1}^{n-1} i) \approx \frac{k_1}{2} n^2.$$

The second term simply involves α constant operations,

$$\sum_{i=1}^{\alpha} b(|U_i|) \leq \sum_{i=1}^{\alpha} k_2 = k_2\alpha.$$

Thus we conclude that $T(n,\alpha) = O(n^2)$. Furthermore, for counting active operations, $k_1 = 1$ and $k_2 = 2$, giving $\tau(n,\alpha) \leq 1/2n^2 + 2\alpha$. $\square$

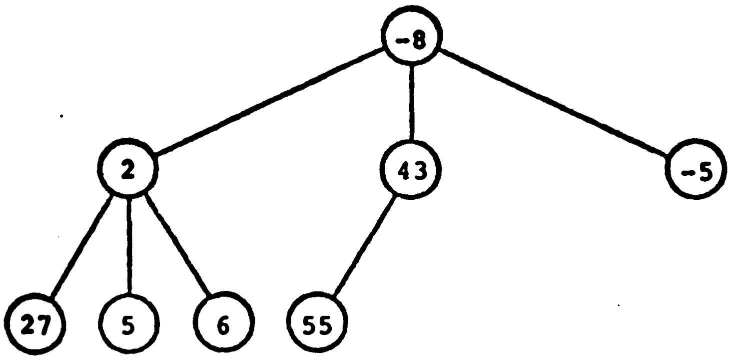
A recent result by Yen [Yen72] gives a bound $\tau(n,\alpha) = 3/2n^2$. The method relies on a matrix representation for w . His idea is to remove from consideration every arc $(v,u) \in X$ once u has been removed from U . The method employs an indexing array to eliminate columns from the matrix w , realizing Dijkstra's node deletion [Dij59].

9.1.2 Partially Ordered Set Realization

The bound $O(n^2)$ of Theorem 9.3 can be improved when $\alpha < n(n-1)$ by saving information gained from searches of U . It is not helpful to maintain U fully sorted on d , however, for this would make $c(|U|)$ of the order of $|U| \log|u|$. Instead, we use a partial ordering of U similar to the tree structure introduced by Floyd in the algorithm Treesort

[Flo62b]. Maintaining U as a tree appears to be the basis of the only known asymptotic improvements over the long standing result of $O(n^2)$ for any $\phi(N,s)$ with arc weights nonnegative. Using a tree (or "heap") was evidently first reported for this application by Murchland [Mur69]. In the reference cited, however, Murchland fails to note that his treatment yields a worst case bound on complete networks of $T(n,n(n-1)) \le O(n^2 \log n)$, not as good as the original algorithm. The present author made an independent discovery of the applicability of a tree structure for U which he developed in two directions, one which gives the best constant (minimizes $\tau(n,n(n-1))$ and another which gives a bound linear in α except when α is quite small. Recently, E. Johnson [Joh72] reported the flaw in Murchland's paper and showed an elaborate method which is of no better order than ours.

Following Williams [Wil64] we call a full β-ary tree, such that every path from the root is ordered on some transitive binary relation on the "value" of the positions, a <u>heap</u>. We will assume that a least element is at the root, and that the lowest level of the tree is filled from the left. For example, one possible heap for the values $\{5, 27, -5, 43, 2, -8, 6, 55\}$ is



In this example, $\beta = 3$. Notice that we call the elements of the tree positions. In our application, each non-empty position will possess some node v in a network, and the value of the position will be the value $d(v)$.

At present, we are concerned with restoring the heap when a single position in the heap has its value changed.

If the value decreases (this case includes the case where a new node is added at the leftmost empty position on the lowest level), the heap is restored if the path from the root to the position of decreased value is reordered. This may be done by comparing the value of the changed position with the position above (its predecessor in the tree). If the changed position has lesser value, the values are interchanged, and the process is repeated on the position with the original changed value until no interchange is called for or the root is reached. The cost of this process is proportional to distance the changed value moves in the tree, and is independent of the degree of branching, β , of the tree.

This cost is bounded by the order of the depth of the tree, $O(\log_\beta n)$ where there are n elements in the tree.

The algorithm for restoring the heap following a reduction in some value $d(v)$ is

```
(9.4)     q:= position(v);
          repeat
            if q not the root then
              if d(v) < d(node(predecessor(q))) then
                begin
                  node(q): = node(predecessor(q));
                  q: = predecessor(q)
                end
          until no node is moved;
          node(q):= v
```

If a value increases, the ordering of the entire subtree headed by the position of changed value is affected. The reader may verify, however, that it is sufficient to reorder the path from the changed position toward the leaves which is of least value at each level. We have an analogous situation to decreased values except that now the process is proportional to the length of the path times the degree of branching β since one of β choices must be made at each position of the path but the last. Thus the cost for reordering following an increase in some value is bounded by $O(\beta \log_\beta n)$.

The algorithm for restoring the heap following an increase in some value d(v) is

(9.5)
```
q:= position(v);
repeat
  if q not a leaf then
    begin
      p:= position of node u of minimum d(u)
          among descendants of position(q);
      if d(v) > d(node(p)) then
        begin
          node(q):= u;
          q:= p
        end
    end
until no node is moved;
node(q):= v
```

In the course of Algorithm W (4.28) we will change values associated with nodes (creating new positions when necessary on the bottom of the heap) and also identify and remove the least element of the heap. Since the least element is always at the root of the heap, identifying it is a constant cost. We remove it by replacing it with the value from the rightmost position on the lowest level of the tree. This preserves the fullness of the heap. Restoring order is then $O(\beta \log_\beta \eta)$, since the removal in a heap of size $\eta+1$ is equivalent to an increase of the root value in a heap of size η.

For binary trees, $\beta = 2$, and we can ignore β in the O-notation. Later we will consider degrees of branching which depend on η .

(9.6) LEMMA. *For any problem $\phi(N,s)$ with no arcs of negative weight reachable from s, if U is maintained as a heap with β independent of n, there are rules under which Algorithm W (4.28) runs in* $T(n,\alpha) \leq O(n \log n + \alpha \log n)$ *operations* (this essentially is Murchland's result [Mur69]).

Proof: We use Dijkstra's selection rule (9.1), so Lemma 4.34 applies. Under the heap discipline, Dijkstra's rule is to select and remove the root of the heap. Thus $c(|U|) \leq k_1 \log|U|$ for heaps with fixed constant degrees of branching. A change in d is always a reduction, so $b(|U|) \leq k_2 \log|U|$ regardless of branching degree. The desired bound follows from Lemma 4.34. □

Williams [Wil64] gives an implementation of heaps, on which these operations may be performed, which requires no extra storage for pointers.

Immediate Reordering

Of course, the naive method of Lemma 9.6 gives a bad bound indeed when $\alpha = n(n-1)$. We present, at first, two ways in which this flaw can be avoided. Both reorder U immediately after each change in d .

(9.7) THEOREM. For problems $\phi(N,s)$ with no negative arcs reachable from s there exists an algorithm which runs in $T(n,\alpha) \leq O(\min(n\log n + \alpha\log n, n^2))$ operations.

Proof: The algorithm first counts the arcs in $O(\alpha)$ time. If $\alpha < n^2/\log n$, it runs the algorithm of Lemma 9.6; otherwise it applies the algorithm of Theorem 9.3. □

(9.8) COROLLARY. For any all pairs problem $\phi(N)$ with no negative arcs there exists an algorithm which runs in $T(n,\alpha) \leq O(\min(n^2 \log n + n\alpha \log n, n^3))$ operations.

Proof: Run the algorithm of Theorem 9.3 once for each $v \in V$. □

When $\alpha = n(n-1)$, the method of Theorem 9.7 is superior to any others known because there is no overhead associated with a tree structure for U. In particular, if we count active operations we find

(9.9) COROLLARY. There exists an $O(\min(n \log n + \alpha \log n, n^2))$ algorithm for $\phi(N,s)$ without negative arcs for which $\tau(n,n(n-1)) \leq \frac{3}{2} n^2$.

Proof: Use the method of Theorem 9.7 but apply Yen's method (cited at the end of Section 9.1.1) when $\alpha \geq n^2/\log n$. □

Of course, Corollary 9.9 ignores the cost of setting up an array for w if it does not already exist, the counting of arcs, and other overhead, since τ counts only active operations.

While the above approach gives an algorithm with asymptotic improvement when $\alpha < n^2/\log n$ and the best known constant when $\alpha = n(n-1)$ among algorithms sensitive to α, testing to decide which of two algorithms to run is inelegant. And, furthermore, we would like asymptotic improvement in the region $n^2/\log n \leq \alpha < n(n-1)$.

We extend our results on algorithms which restore order to U after every path weight reduction by considering trees with degree of branching $\beta = f(n)$, for some function f.

The results which followed from Lemma 9.6 were the special case of $f(n)$ a constant. When ordering a path from the root by moving a node downward toward the leaves, a β-way comparison is needed at each level to determine which branch the path least on d takes. When β was a constant independent of n we ignored the factor β, but now we must consider it. On the other hand, when nodes are appended to U in Step (2blc) of Algorithm W (4.28) they move upward toward the root. Finding the path here is a constant cost at each level. This gives us

(9.10) THEOREM. For problems $\phi(N,s)$ with no negative arcs reachable from s, Algorithm W (under Dijkstra's selection rule (9.1)) runs in $T(n,\alpha) \leq O(n\beta\log_\beta n + \alpha\log_\beta n)$ operations when U is maintained as a tree *with discrete degree of branching* $\beta > 1$.

Proof: The result follows by generalization of the arguments for Lemma 9.6. □

(9.11) COROLLARY. *When* $\beta = \lceil n^{1/k} \rceil$ *for fixed* $k \geq 1$ *independent of* n, *the bound of Theorem 9.10 is* $T(n,\alpha) \leq O(n^{1+1/k} + \alpha)$. □ .

Thus for the single source problem we have a result for which there is a bound linear in α in the range $n^{1+1/k} \leq \alpha \leq n(n-1)$ defined by some fixed k . When $k = 2$, for instance, U will be maintained as a tree with three levels and degree of branching $\lceil \sqrt{n} \rceil$.

(9.12) COROLLARY. *For any all pairs problem* $\phi(N)$ *with no negative arcs there exists an algorithm which runs in* $T(n,\alpha) \leq O(n^{2+1/k} + n\alpha)$ *operations, for fixed* $k \geq 1$.

Deferred Reordering

We obtained the above results by restoring the ordering to U each time $d(v)$ is reduced for any v . This strategy is sufficient, and it gives a simple and useful result. However, it is only necessary to have U ordered when the selection is made in Step (2a) of Algorithm W (4.28). E. Johnson [Joh72] gives an algorithm in which reordering is so defined. We now present his ideas and extend the analysis to obtain a bound for the general case. With this new bound we will show that deferred reordering has no advantage over immediate reordering.

A full binary tree may be ordered over paths from the root on some transitive binary relation in time proportional to the number of elements in the tree. A way to do so first appeared in Floyd's algorithm Treesort 3 [Flo64]. The result may be generalized to trees of any fixed degree of branching β .

Let the levels of the tree be numbered from the root, beginning with one. If there are η positions in the tree there will be $L = \lceil \log_\beta((\beta-1)\eta + 1) \rceil$ levels in the tree, the last of which may not be full. We assume the last level is filled (say) from the left.

The following algorithm will order a tree on some given relation over paths from the root.

(9.13) ALGORITHM T: Tree Ordering

```
for ℓ = L-1 step -1 until 1 do
  for each position on level ℓ do
    order the subtree of which it is the root
```

Correctness is obvious since the entire tree is ordered when $\ell = 1$. We can refine Algorithm T so that it runs in linear time.

(9.14) <u>LEMMA</u>. <u>Algorithm T can be implemented to run in no more than</u> $(\frac{\beta}{\beta-1})n$ <u>comparisons on a full tree</u>.

<u>Proof</u>: Consider a full tree such that $n = \frac{\beta^L-1}{\beta-1}$. When the algorithm reaches level ℓ we know that all subtrees rooted in level $\ell+1$ are ordered as required. Thus to order any subtree rooted in position p on level ℓ , only one path from p toward the leaves on the tree (downward) must be reordered. This will require no more than $\beta(L-\ell)$ comparisons, one for each branch headed by positions along the path. Since there are $\beta^{\ell-1}$ positions on each level, the cost for one level is no more than $\beta^{\ell}(L-\ell)$ comparisons (tree branches).

Summing over all levels we get

$$\sum_{\ell=1}^{L-1} \beta^{\ell}(L-\ell) = L\left(\frac{\beta^L-1}{\beta-1} - 1\right) - \frac{(L-1)\beta^{L+1}-L\beta^L+\beta}{(\beta-1)^2}$$

$$= \frac{\beta^{L+1}-\beta}{(\beta-1)^2} - \frac{L\beta}{\beta-1} = \frac{\beta}{\beta-1}(n-L)$$

$$= \left(\frac{\beta}{\beta-1}\right)n - O(\log_\beta n) \;.\; \square$$

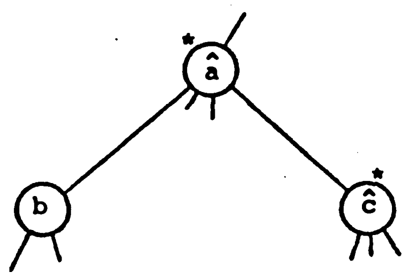
We will call the implementation presented in the above proof Algorithm T1.

With this result we can incorporate into Step (2a) of Algorithm W (4.28) a reordering of U before selecting u . This gives a cost for (2a) of $O(|U| + \log|U|) = O(|U|)$, so, as we had with no tree at all, the algorithm is $O(n^2)$ in the worst case.

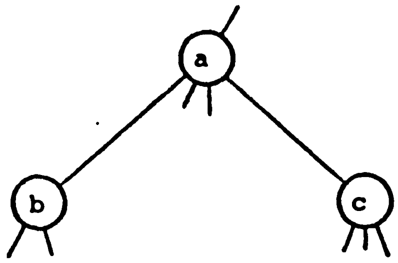
This result is little interest to us as it stands. However, it can be improved if we observe that the trees which must be reordered are ordered heaps in which the values at some positions have been reduced. Let such a tree be called a <u>restricted tree</u>.

When ordering a path downward toward the leaves from some position p in tree U , we begin by comparing the values associated with $\beta+1$ positions, p and its descendants. Order is restored by exchanging the value of p with the least value. If p has the least value, no exchange need occur, and we can ignore the remainder of the path toward the leaves.

From this it follows that if Algorithm T1 preserves the restricted property of a tree at each step, it need never expend comparisons of the value at any position with other than its descendants with reduced values. The following diagram may be used to verify this assertion. Let a restricted tree (of which we show a part)

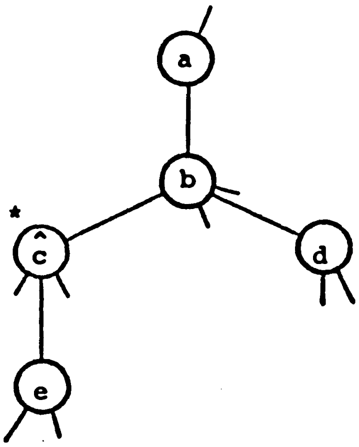


arise from an original heap



where the letters shown in the positions are values and a star indicates a reduced value. Note that the "reduced" property belongs to a value and moves with it if the value moves in the tree. It must be that $\hat{a} < b$ since $a \leq b$ by the heap property and $\hat{a} < a$ because of the reduction in value.

It is also easy to show that Algorithm T1 (9.14) preserves the restricted property of a given tree at each step. Consider the following region of a tree in which values $\hat{c}$ and b will be interchanged:



Because the tree is restricted,

(9.15) $a \leq b$

(9.16) $b \leq d$

(9.17) $b \leq e$

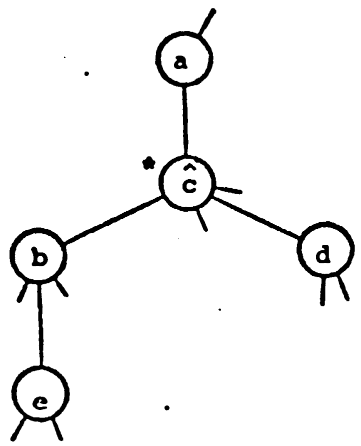
(9.18) $\hat{c} < e$ (since $c \leq e$ and $\hat{c} < c$)

(9.19) $a \leq d$

But $\hat{c}$ and b are to be interchanged, so

(9.20) $\hat{c} < b$.

Following the interchange, the tree is relabelled



This tree is also restricted since

$a \leq b$ (9.15)

$a \leq d$ (9.19)

$\hat{c} < b$ (9.20)

$b \leq e$ (9.17)

$\hat{c} < d$ (9.20) and (9.16).

The above arguments show that the following refinement of Algorithm T (9.13) correctly restores heap order to restricted trees.

(9.21) ALGORITHM T2: Restricted Tree Reordering

```
for ℓ=L-1 step -1 until 1 do
  for each position p on level ℓ
      which has a descendant with a node of
      reduced value do
        test(p)
```

where test (p) is the procedure

```
q:= position of least-valued descendant (among
    reduced values) of p;
if value at p > value at q then
  begin
    exchange nodes at p and q;
    if q not a leaf then test(q)
  end
```

Implementation of Algorithm T2 requires two kinds of lists. For each level there must be a list of positions with descendants of reduced value. For each such position there must be a list of its descendants of reduced value. Each interchange of values between the positions involves at most a fixed number of list insertions and deletions for maintaining the lists. If deletions from the level lists are postponed, all lists may be singly-linked instead of doubly-linked as proposed by E. Johnson [Joh72].

We now consider the running time of Algorithm T2 (9.21). Let a branch in a restricted tree be <u>covered</u> by a reduced value if it belongs to a path from the root to the position holding the reduced value. Let $\zeta(n,\delta)$ be defined as

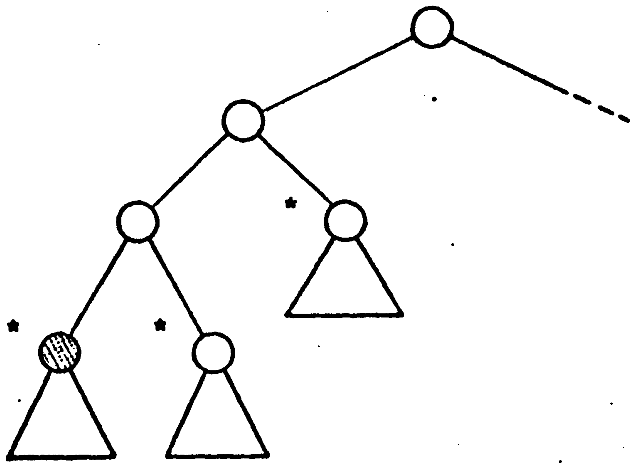
(9.22) $\zeta(n,\delta) \equiv$ the maximum number of branches which can be covered in a tree of size n by δ reduced values.

(9.23) <u>LEMMA. The running time of Algorithm T2 on a restricted tree of size n in which $\delta \geq 1$ values have been reduced is $O(\zeta(n,\delta))$ in the worst case.</u>

<u>Proof</u>: Consider a case in which the δ reduced values occupy some (or all) of the leaves of the tree and, after heap order is restored, these values fill a full subtree at the root. In other words, the changed values move as far as possible in tree. The sum of the number of branches over which the values move plus the final comparison for each value (to determine that it moves no further) is equal to $\zeta(n,\delta)+1$. Thus Algorithm T2 (9.21) must expend at least $O(\zeta(n,\delta))$ operations in the worst case.

To show that no more than $O(\zeta(n,\delta))$ operations are needed, we recall that Algorithm T2 makes no comparisons in a subtree in which no reduced value appears below its root. Since reduced values move only toward the root from their original positions, it is sufficient to consider as free from comparisons the subtree in which no reduced values appear originally. Let such subtrees be called exempt subtrees, and the branches which are never considered by Algorithm T2 (as a consequence of being associated with exempt subtrees) exempt branches.

In the following diagram of part of a restricted tree, subtrees are shown as triangles. Positions with reduced values are darkened; positions which head exempt subtrees are starred.



It should be clear from a careful reading of Algorithm T2 (9.21) that the branch leading to an exempt subtree whose root has an unreduced value is an exempt branch. Thus the exempt branches are exactly those branches which are not covered by reduced values.

From the proof of Lemma 9.14 it is clear that covering all exempt branches in a full restricted tree would add a number of comparisons to Algorithm T2 (9.21) at least equal to

$$(\tfrac{\beta}{\beta-1})\nu - O(\log_\beta \nu) \quad ,$$

where ν is the number of exempt branches. But $\nu = (\eta-1) - \zeta(\eta,\delta)$, so we conclude that the number of comparisons for Algorithm T2 on full trees is bounded by

$$(\tfrac{\beta}{\beta-1})\eta - O(\log_\beta \eta) - (\tfrac{\beta}{\beta-1})(\eta-1) + (\tfrac{\beta}{\beta-1})\zeta(\eta,\delta) + O(\log_\beta \eta)$$
$$= O(\zeta(\eta,\delta)) \quad .$$

To translate this result to other than full trees, we observe that a worst case tree can be made a worst case full tree by moving at most δ reduced positions down one level. Thus, the number of comparisons for any tree is bounded by

$$O(\zeta(\eta,\delta) + \delta) \quad .$$

To get a bound on the total work for reordering a tree we must add in the costs for searching the level lists and moving values to effect the reordering. Searching the lists is $O(\delta + \log\eta)$ since there is one list for each level and all the lists can have no more entries than δ . Moving values is at most proportional to the number of comparisons, so we conclude that the running time for Algorithm T2 on any restricted tree is bounded by

$$O(\zeta(\eta,\delta) + \delta + \log\eta)$$

which equals $O(\zeta(\eta,\delta))$ since $O(\log\eta) = O(\zeta(\eta,1))$ and $O(\delta) \leq O(\zeta(\eta,\delta))$. □

We now turn to bounding $\zeta(\eta,\delta)$ (9.22).

(9.24) LEMMA.

(i) $\zeta(\eta,0) = 0$

(ii) $\zeta(\eta,1) = L-1$

(iii) $\zeta(\eta,\delta) < \min(\eta,\delta(\log_\beta(\frac{\eta(\beta-1)+1}{\delta-1}) + \frac{\beta}{\beta-1}) - \frac{\beta+1}{\text{[illegible]}})$,

$\eta > \beta+1$, $\delta>1$.

Proof: Parts (i) and (ii) are obvious. To prove (iii) we will compute the sum of branches which results from marking leaves one by one up to δ in such a way that the longest uncovered path of branches is covered by each marking. (Obviously the number of branches covered cannot exceed $\eta-1$.) Branches are added in once when first covered. This amounts to filling a subtree of δ nodes from the root downward, each position on level ℓ contributing $L-\ell$ to the value of $\zeta(\eta,\delta)$, giving

$$\zeta(\eta,\delta) = (L-1) + (\beta-1)(L-1) + \beta(\beta-1)(L-2) + \beta^2(\beta-1)(L-3) + \ldots + \beta^{\lambda-1}(\beta-1)(L-\lambda) + (\delta-\beta^{\lambda})(L-\lambda-1)$$

where $\lambda = \lfloor \log_{\beta}(\delta-1) \rfloor$ and η is restricted to full trees, that is, $\eta = \frac{\beta^j - 1}{\beta - 1}$, $j = 2,3,\ldots$. Thus

$$\zeta(\eta,\delta) = (L-1) + \sum_{\ell=0}^{\lambda-1} (L-\ell-1)(\beta-1)\beta^{\ell} + (\delta-\beta^{\lambda})(L-\lambda-1)$$

$$= \delta(L-\lambda-1) + \frac{\beta}{\beta-1}(\beta^{\lambda}-1).$$

If we let $\log_{\beta}(\delta-1) = \lambda + \Delta,\ 0 \leq \Delta < 1$,

$$\zeta(\eta,\delta) = \delta(\log_{\beta}(\eta(\beta-1)+1) - \log_{\beta}(\delta-1)+\Delta-1) + \frac{\beta}{\beta-1}(\beta^{\log_{\beta}(\delta-1)-\Delta}-1)$$

$$= \delta\log_{\beta}\left(\frac{\eta(\beta-1)+1}{\beta(\delta-1)}\right) + \delta\Delta + \frac{\beta(\delta-1)}{\beta^{\Delta}(\beta-1)} - \frac{\beta}{\beta-1}.$$

In the interval $0 \leq \Delta < 1$,

$$\delta\Delta + \frac{\beta}{\beta-1}\left(\frac{\delta-1}{\beta^{\Delta}}\right) < \frac{\beta\delta-1}{\beta-1} .$$

Thus

$$\zeta(\eta,\delta) < \delta\left(\log_{\beta}\left(\frac{\eta(\beta-1)+1}{\beta(\delta-1)}\right) + \frac{\beta}{\beta-1}\right) - \frac{\beta+1}{\beta-1} ,$$

$$\text{for} \quad \eta = \frac{\beta^j-1}{\beta-1} ,\ j = 2,3,\ldots .$$

So for any $\eta \geq \beta+1$,

$$\zeta(n,\delta) < \delta(\log_\beta(\frac{\beta^{\lceil \log_\beta(n(\beta-1)+1)\rceil}}{\beta(\delta-1)} + \frac{\beta}{\beta-1}) - \frac{\beta+1}{\beta-1} <$$

$$\delta(\log_\beta(\frac{n(\beta-1)+1}{\delta-1}) + \frac{\beta}{\beta-1}) - \frac{\beta+1}{\beta-1} . \quad \Box$$

Let us apply this result to set U in Algorithm W (4.28), modified to include reordering of U in Step (2a). Lemma 9.24 gives us a bound on one such reordering. It follows that n reorderings are bounded by

$$(9.25) \quad \Sigma(n,\alpha) = \sum_{i=0}^{j_1-1} \zeta(|U_i|,0) + \sum_{i=j_1}^{j_2-1} \zeta(|U_i|,1) +$$

$$\sum_{i=j_2}^{n-1} (\delta_i(\log_\beta \frac{|U_i|(\beta-1)+1}{\delta_i-1} + \frac{\beta}{\beta-1}) \geq$$

$$\sum_{i=0}^{n-1} \zeta(|U_i|,\delta_i),$$

for the appropriate indexing of the sequence of sets $(U_1, U_2,\ldots,U_n)$, and where $j_1 \leq j_2 \leq n$, and δ_i is the number of points at which d is reduced in the iteration which produces U_i, the tree which has to be reordered. Obviously $\delta_i \leq |U_i|$ and $\sum_{i=0}^{n-1} \delta_i \leq \alpha$. Our task is to maximize $\Sigma(n,\alpha)$ (9.25) under these restrictions.

The maximum will occur when δ_i is such that the next path which would be covered in U_i is no shorter than any uncovered path in U_j for $0 \leq i, j \leq n-1$. Thus very small trees will have no changes at all.

We now assume that any tree which is incomplete on its last level (that is, $|U_i| \neq \frac{\beta^j-1}{\beta-1}$, $j = 2,3,\ldots$) will have single branches added to bring all leaves on level L-1 down to level L. This makes all initial positions for δ occupy level L. Under this assumption let ℓ_i be the number of branches in the longest uncovered path in U_i. That is,

$$L - \ell_i = \lfloor \log_\beta(\delta_i(\beta-1)+1) \rfloor = L - \ell_i$$

$$L - \ell_i \leq \log_\beta(\delta_i(\beta-1)+1) < L - \ell_i + 1$$

$$\lceil \log_\beta(|U_i|(\beta-1)+1) \rceil - \ell_i \leq \log_\beta(\delta_i(\beta-1)+1) <$$
$$\lceil \log_\beta(|U_i|(\beta-1)+1) \rceil - \ell_i + 1$$

$$\log_\beta\left(\frac{|U_i|(\beta-1)+1}{\beta^{\ell_i}}\right) \leq \log_\beta(\delta_i(\beta-1)+1) <$$
$$\log_\beta\left(\frac{|U_i|(\beta-1)+1}{\beta^{\ell_i-2}}\right).$$

For $\delta_i \geq 2$ and $|U_i| = i$,

$$\frac{i(\beta-1)+1}{\beta^{\ell_i+1}} < \delta_i < \frac{i(\beta-1)+1}{\beta^{\ell_i-2}(\beta-1)}$$

$$\frac{i(\beta-1)+1}{\delta_i-1} < \beta^{\ell_i+2} .$$

We may now substitute in (9.25), noting that ℓ_i is fixed for all i.

$$\Sigma(n,\alpha) < \sum_{i=0}^{j_1-1} 0 + \sum_{i=j_1}^{j_2-1} (L_i-1) + \sum_{i=j_2}^{n-1} \delta_i (\log_\beta(\beta^{\ell_i+2}) + \frac{\beta}{\beta-1}) .$$

Of course, any element of the last sum is greater than L_i-1, so we have

$$(9.26) \quad \Sigma(n,\alpha) < \sum_{i=0}^{n-1} \delta_i (\log_\beta(\beta^{\ell_i+2}) + \frac{\beta}{\beta-1}) .$$

We also know that if the bound α on changes is realized,

$$\alpha = \sum_{i=0}^{n-1} \delta_i < \sum_{i=0}^{n-1} \frac{i(\beta-1) + 1}{\beta^{\ell_i-2}(\beta-1)}$$

from which we find the relations (recall ℓ_i does not depend on i)

$$(\beta-1)\,\beta^{\ell_i-2}\,\alpha < \frac{(\beta-1)n(n-1)}{2} + n$$

and

$$\beta^{\ell_i+2} < \frac{\beta^4}{(\beta-1)\alpha}\left(\frac{(\beta-1)n(n-1)}{2} + n\right).$$

Since every variable within the sum in (9.26) is fixed except δ_i, we get

$$(9.27) \quad \Sigma(n,\alpha) < \alpha(\log_\beta(\beta^4(\frac{n(n-1)}{2\alpha} + \frac{n}{(\beta-1)\alpha})) + \frac{\beta}{\beta-1}) .$$

A similar proof can be shown which bounds $\Sigma(n,\alpha)$ from below by an expression of the same order.

Formula (9.27) bounds the sum of the number of branches covered for all sets U_i to be reordered in Step (2a) of Algorithm W (4.28). Since reordering using Algorithm T2 (9.21) is of this order (Lemma 9.23), and all other steps in Algorithm W are bounded by $O(\alpha)$, we have

(9.28) THEOREM. *Under the deferred reordering procedure of Algorithm T2 (9.21), Algorithm W (4.28) with Dijkstra's selection rule runs in* $T(n,\alpha) \leq O(\beta n \log_\beta n + \alpha(1 + \log_\beta(\frac{n^2}{\alpha})))$ *operations on problems with no arcs of negative weight.* □

How does this bound under the best procedure known for deferred reordering compare with the bound for immediate reordering, $O(\beta n \log_\beta n + \alpha \log_\beta n)$ (Theorem 9.10)? At first glance, deferred reordering appears to give a better bound at other than the extreme points, $\alpha = n-1$ and $\alpha = n(n-1)$. However, the bounds are of equal order when β is chosen optimally, as we now show.

Consider the range $n-1 \leq \alpha \leq n^{1+1/k}$ for some k independent of n. In this range, $\log_\beta n \geq \log_\beta(\frac{n^2}{\alpha}) \geq \frac{k-1}{k} \log_\beta n$, and the bounds for immediate and deferred reordering are obviously of equal order for any β.

Corollary 9.11 showed that there is a choice for β ($\beta = \lceil n^{1/k} \rceil$) which makes Algorithm W (4.28) with immediate reordering linear in α (and therefore of optimum order) in the range $n^{1+1/k} \leq \alpha \leq n(n-1)$. Similarly, if $\beta = \lceil (\frac{n^2}{\alpha})^{1/k} \rceil$ we get a bound under deferred reordering of $T(n,\alpha) \leq O(n(\frac{n^2}{\alpha})^{1/k} + \alpha)$. This bound is linear in α for all α greater than

$$\alpha = n(\frac{n^2}{\alpha})^{1/k}$$

$$\alpha = n^{(k+2)/(k+1)} < n^{1+1/k} .$$

We conclude that deferred reordering is of no advantage, particularly in view of the larger constant which arises from the level and reduced value list manipulations in Algorithm T2 (9.21).

If the analysis of deferred reordering is confined to regular networks (those with a uniform number of inward and outward arcs at each node) the analysis and bound are quite simple, though still no improvement over an appropriate strategy for immediate reordering.

(9.29) COROLLARY. (E. Johnson [Joh72]). *When* $\beta = 2$ *and a network of uniform degree* $\Delta = |\{v | (u,v) \in X$ for fixed $u \in V\}|$, *the bound is* $T(n,\alpha) \leq O(n\Delta(1 + \log(\frac{n}{\Delta})))$ *operations*. □

9.1.3 Dijkstra's Algorithm

Dijkstra's algorithm [Dij59] is a variant of Algorithm W (4.28) in which any node which has been removed from U is forbidden to return. Consequently, the algorithm always terminates. It reports a correct solution whenever Algorithm W does not attempt to return any node to U . Thus it finds solutions, in $O(n^2)$ time, to a set of single source problems which properly includes all problems with no arcs of negative weight.

9.2 Restricted Cyclic Networks with Nonnegative Arcs

We call a network whose strongly connected components are either single nodes or single cycles a simple cyclic network. An $O(n+\alpha)$ algorithm exists which solves the single source problem on such networks when there are no arcs of negative weight. The algorithm first identifies the cyclic strong components, each one being recorded as a circularly-linked list. Then a modification of Algorithm A (7.1) is run on the network.

An $O(n+\alpha)$ algorithm due to Tarjan [Tar72] identifies the strong components of N . Once they are found, it is verified whether each component is in fact a single node or cycle. A strong component of more than one node is a single cycle if each node originates and terminates exactly one arc of the component. This property can be verified for all strong components in $O(n+\alpha)$ operations while at the same time generating a circularly-linked list for each cycle.

The modification of Algorithm A is run on an acyclic network equivalent, with respect to shortest paths, to the given network. As the computation proceeds, it becomes possible to expand the cycles into acyclic components.

Our strategy is to order the node set of the condensation $\hat{N}$ of N , knowing that as soon as a permanently marked node v is identified in any cycle (strong component) V_i , the arc of the cycle which enters v is known not to be a reducing arc. So it may be deleted, making V_i acyclic. The individual nodes of V_i can thus be inserted into the ordering for Algorithm A in cycle order, beginning with v . This process will succeed if we can always transform a cycle V_i in time to let Algorithm A select a next node in Step (2a). The following algorithm meets this requirement. The primitives "head" and "tail" operate in the conventional way to produce the first element of a list and the list of all elements but the first, respectively. The operator "$\circ$" concatenates lists. A symbol with a hat "^" refers to the condensation $\hat{N}$ of the given network N . Thus the list L first contains elements of $\hat{V}$, nodes and sets of nodes from V . The algorithm proceeds identically to Algorithm A as long as each element of $\hat{V}$ encountered in L is an individual node of V . When a set V_i (representing a cycle) is encountered, it is replaced by <u>its</u> elements, which belong to V .

(9.30) ALGORITHM SC: Simple Cyclic Network Algorithm

(1) [Initialize] (1a) Generate the condensation
$\hat{N} = (\hat{V},\hat{X})$ of $N = (V,X)$;
__if__ $\hat{N}$ is not simple cyclic __then__
report "failure"
__else__ generate a circular list
$V_i = (v_{i_1}, v_{i_2}, \ldots, v_{i_{n_i}})$ for
each cycle V_i;

(1b) Create a list $L = (\hat{v}_{j_1}, \hat{v}_{j_2}, \ldots, \hat{v}_{j_\ell})$ such that $\hat{v}_{j_i}$ precedes $\hat{v}_{j_k}$ if $(\hat{v}_{j_i}, \hat{v}_{j_k}) \in \hat{X}$;

(1c) $d(v) := \infty$ for $v \in (V-\{s\})$; $d(s) := 0$;
$U := \{\hat{v}_{j_k} \mid s = \hat{v}_{j_k}$ or $s \in \hat{v}_{j_k}\}$;

(2) [Search and Replace]

__while__ $U \neq \emptyset$ __do__
__begin__

(2a)
__while__ head(L) $\not\in$ U __do__ L:= tail(L);
$U := U - \{\text{head}(L)\}$;
$L := \bar{L} \oplus \text{tail}(L)$
where $\bar{L} = (v_{j_1}, v_{j_2}, \ldots, v_{j_p}) =$
head(L) in cycle order and where
$d(v_{j_1}) = \min_{v \in \text{head}(L)} (d(v))$;
$U := U \cup \{v_{j_2}, \ldots, v_{j_p}\}$;
$u := v_{j_1}$;
L:= tail(L);

```
(2b)  for each (u,v) ∈ X do
        if d(v)>d(u)+w(u,v) then
          begin
            d(v):= d(u)+w(u,v);
            U:= U ∪ {v}
          end
      end
```

(9.31) THEOREM. <u>Algorithm SC gives a solution to the single source problem for every simple cyclic network with arcs of nonnegative weight. On networks that are not simple cyclic it reports failure.</u>

<u>Proof</u>: Following our preliminary discussion, all we need to prove is that, whenever Step (2a) chooses some nontrivial cycle V_i, $d(v) = \min_{u \in V_i} (d(u))$ implies $d(v) = \mu(v)$ for some $v \in V_i$. As a contrary assumption let V_i be the first nontrivial cycle selected in Step (2a) for which there exists $v \in V_i$ such that $d(v) > \mu(v)$ and $d(v) = \min_{u \in V_i} (d(u))$. By the properties of Algorithm A (7.1) we know that $d(u) = \mu(u)$ for all u already deleted from L. So $d(v) = w(s=u_{i_1}, u_{i_2}, \ldots, u_{i_\ell}=v)$ where all nodes but v have been deleted from L. By assumption there is a path $p_v = (s=u_{j_1}, u_{j_2}, \ldots, u_{j_m}, v_{k_1}, v_{k_2}, \ldots, v_{k_p}=v)$ such that $w(p_v) = \mu(v)$ and the v_k's belong to V_i. Thus our assumption requires $w(u_{j_1}, u_{j_2}, \ldots, u_{j_m}, v_{k_1}) \leq d(v)$, implying $w(v_{k_1}, v_{k_2}, \ldots, v_{k_p}) < 0$, a contradiction of a condition of the theorem. □

(9.32) THEOREM. *Algorithm SC runs in* $O(n+\alpha)$ *operations.*

Proof: Using the algorithm of Tarjan [Tar72], initialization is $O(n+\alpha)$. From our analysis of Algorithm A (7.1), all but (2a) of Step (2) must be $O(n+\alpha)$ since (2a) can select each node in V at most once. Step (2a) chooses every node in V exactly once and finds minima (once each) in disjoint subsets of V, all of which is $O(n)$. □

(9.33) COROLLARY. *There is a generalization of Algorithm SC which runs in* $O(n+\alpha)$ *operations on any network with arcs of nonnegative weight for which the strong components consist of at most* k *distinct cycles,* k *independent of* n.

Proof: In Step (1a) merely find the condensation $\hat{N}$, maintaining each V_i unordered.

Step (2a) is modified so that a strong component V_i is replaced in L by the sublist ℓ_i, where head$(\ell_i) = v$ which minimizes d in V_i and tail(ℓ_i) is created from the acyclic condensation of $(V_i-\{v\})$. This form of replacement is repeated for $\min(k,|V_i|)$ times. If nodes remain in V_i following this process, it is checked whether that which remains of V_i has more than one cycle. Depending on the outcome, the algorithm reports "failure" or the properly-ordered sublist is inserted into L as before. Clearly k bounds the number of incomplete replacements of ℓ_i for

V_i or a subgraph of V_i and thus the number of extra minimizations over disjoint subsets of V. We conclude the bound of Theorem 9.32 is at most increased by a constant multiple. □

10. Rules of Choice for Networks with Unrestricted Arc Weights

Algorithm W (4.28) is correct on any problem without cycles of negative weight reachable from s, regardless of the rule of choice employed in Step (2a). There are wide variations in running time, however, between rules.

The most disastrous running time situation we know of arises when Dijkstra's rule (9.1) is used on unrestricted networks. Doing so is reported valid, for instance, by both Murchland [Mur69] and Edmonds and Karp [E&K72]. But nowhere do we find in the literature an inkling that Algorithm W under Dijkstra's rule can take exponential time in worst cases (as we prove below). Obviously, on unrestricted networks, Dijkstra's rule no longer satisfies the criterion of Lemma 4.34 that each node chosen not return to U.

Another rule which does not meet the criterion of Lemma 4.34 is a first-in-first-out rule. However, this rule is attractive because it can be implemented to run in constant time. With this rule, Algorithm W becomes essentially the algorithm described by Moore [Mor59]. It runs in time $O(t\alpha)$.

10.1 An Exponential Result with Dijkstra's Rule

(10.1) THEOREM. Algorithm W (4.28) can take $O(n2^n)$ operations to terminate under Dijkstra's selection rule (9.1).

<u>Proof</u>: Let $N_n(y)$, a complete network with $n \geq 3$ nodes, have arc weights

$$\text{(10.2)}\quad \begin{array}{ll} \text{(i)} & w(i,j) = j - i - 2^{i-2}, \quad 1 \leq j < i < n \\ \text{(ii)} & w(n,j) = j - n - 2^{n-2} - y, \quad 1 \leq j < n \\ \text{(iii)} & w(i,j) = r_n + y, \quad 1 \leq i < j \leq n \end{array}$$

where $r_n = 2^n$, $y \geq 0$,

and let a single source problem be posed with $s = n$.

For example, the weights of $N_7(y)$ are

$i \setminus j$	1	2	3	4	5	6	7
1		128+y	128+y	128+y	128+y	128+y	128+y
2	-2		128+y	128+y	128+y	128+y	128+y
3	-4	-3		128+y	128+y	128+y	128+y
4	-7	-6	-5		128+y	128+y	128+y
5	-12	-11	-10	-9		128+y	128+y
6	-21	-20	-19	-18	-17		128+y
7	-38-y	-37-y	-36-y	-35-y	-34-y	-33-y	

Any cycle of $N_n(y)$ can have at most $n-1$ arcs of negative weight since there must be at least one arc of weight r_n+y to complete any cycle. When $r_n = 2^n$ there are no cycles of negative weight since

$$\left|\min_{\substack{i_1,i_2,\\ \ldots i_n}} (w(i_1,i_2) + w(i_2,i_3) + \ldots + w(i_{n-1},i_n))\right| <$$

$$\sum_{i=2}^{n} \left|\min_{j<i} (w(i,j))\right| = \sum_{i=2}^{n} |w(i,1)| =$$

$$y + \sum_{i=2}^{n} (i + 2^{i-2} - 1) = y + \frac{1}{2} n(n-1) + 2^{n-1}-1 <$$
$$y + 2^n, \text{ for } n > 0 .$$

We claim the following property for Algorithm W under Dijkstra's rule when run on $N_n(y)$:

(10.3) $P_n \equiv$ (i) There are exactly 2^{n-1} admissions of nodes to U, and
(ii) Upon termination
$d(n) = c$
$d(i) = c - y - 2^{n-1} - n + i + 2^{i-1}$,

where c is the value to which Algorithm W initializes $d(s)$ (this is a generalization introduced for the purpose of the proof).

P_3 may be verified directly: there are $4 = 2^{3-1}$ admissions and $d(1) = c - y - 5$, $d(2) = c - y - 3$, and $d(3) = c$.

Assume P_n for all N_n , and then consider $\hat{N}_{n+1}(y)$ identical to $N_{n+1}(y)$ except arc $(n+1,n)$ is missing. Since there is no connection from $s = n+1$ to node n , the entire nth row of the weight matrix is ignored. So $\hat{N}_{n+1}(y)$ is identical under the algorithm to $N_n(\bar{y})$ for which $\bar{y} = y + 2^{n-2}+1$. Since P_n holds for $N_n(\bar{y})$, it must hold on $\hat{N}_{n+1}(y)$ (with the obvious renaming of $n+1$ by n).

When the algorithm is run on $N_{n+1}(y)$, $s = n+1$ is admitted to U in the initialization step, followed in the first iteration by nodes $1,2,\ldots,n$. Since admissions occur only when d is reduced, it is clear that node n will remain unchosen until $U = \{n\}$. Thus there is a first phase of execution on $N_{n+1}(y)$ identical to that on $\hat{N}_{n+1}(y)$ but with one extra admission for node n. That is, there are $2^{n-1} + 1$ admissions leaving $U = \{n\}$ and

$$\begin{aligned} (10.4)\quad d(i) &= c - \bar{y} - 2^{n-1} - n + i + 2^{i-1} \\ &= c - y - 3\cdot 2^{n-2} - 1 - n + i + 2^{i-1}, \quad 1 \le i < n \end{aligned}$$

$$\begin{aligned} (10.5)\quad d(n) &= c + w(n+1,n) \\ &= c - 1 - 2^{n-1} - y \end{aligned}$$

$$(10.6)\quad d(n+1) = c.$$

Because $U = \{n\}$, n will be chosen next in Step (2a). We claim that Step (2b) will admit nodes $1,2,\ldots,n-1$ to U because

$$(10.7)\quad d(i) > d(n) + w(n,i), \quad 1 \le i < n.$$

To verify this relation (10.7) we expand both sides, using the definition of $N_{n+1}(y)$ (10.2) and the expressions for $d(i)$ and $d(n)$ ((10.4) and (10.5)) which hold when $U = \{n\}$.

$$d(i) > c - 1 - 2^{n-1} - y + i - n - 2^{n-2}$$
$$d(i) > c - y - 3\cdot 2^{n-2} - 1 - n + i$$
$$2^{i-1} > 0,$$

which is true for all $i \ge 1$ as required.

But admission of nodes $1,2,\ldots,n-1$ is the first step of execution on network $N_n(0)$ with $c_0 = d(n) = c - 1 - 2^{n-1} - y$. So P_n holds again, giving $2^{n-1} - 1$ more admissions in addition to the first $2^{n-1} + 1$ of the first phase on $N_{n+1}(y)$. This gives $2^{(n+1)-1}$ admissions, verifying (i) of P_{n+1} (10.3).

We may use the assumption P_n to conclude that the final values of d are

$$
\begin{aligned}
d(n+1) &= c \\
d(n) &= c - 1 - 2^{n-1} - y \\
&= c - y - 2^{(n+1)-1} - (n+1) + n + 2^{n-1} \\
d(i) &= c_0 - 0 - 2^{n-1} - n + i + 2^{i-1} \\
&= c - 1 - 2^{n-1} - y - 2^{n-1} - n + i + 2^{i-1} \\
&= c - y - 2^{(n+1)-1} - (n+1) + i + 2^{i-1}
\end{aligned}
$$

which verify (ii) of P_{n+1}.

By induction, there are 2^{n-1} admissions for any $n \geq 3$, each of which causes $n-1$ arcs to be explored. Thus execution takes $O((n-1)2^{n-1}) = O(n2^n)$. □

It can be shown that there is no problem which causes Algorithm W (4.28) to run longer than the bound of the theorem. A dependency graph of new values of d may be constructed. There will be a node in the graph for each change of a point in d. Each node will terminate exactly one arc from the value on which the node depends. This graph will be a tree with at most 2^{n-1} nodes because no reduction at node i can depend on a previous $d(i)$ or there would be a cycle of negative weight.

10.2 An $O(n^3)$ Variant of Dijkstra's Rule

Theorem 10.1 shows that Dijkstra's rule (9.1), which gives a good bound under the restriction of arcs to nonnegative weight, can be a very bad rule indeed when the restriction is lifted. We should not, however, conclude that Algorithm W (4.28) is inherently bad for unrestricted problems. In Section 11, in fact, we get very good results from a related algorithm.

We can also get an $O(n^3)$ bound on complete networks if we, in effect, run a sequence of n executions of Algorithm W with Dijkstra's rule. The necessary effect may be accomplished in Algorithm W, using as a rule of choice

$$\text{"choose } u \in U \text{ such that } d(u) = \min_{v \in \hat{U}}(d(v))\text{"}$$

where $\hat{U}$ is a subset of U comprised of nodes which have been returned to U the fewest number of times. The idea is that when the fewest number of returns (for members of U) is i, all nodes with shortest paths of arc length $i+1$ or less have had shortest paths found. The proof is left to the reader.

10.3 Moore's Algorithm

Let Algorithm W (4.28) choose, in Step (2a), the oldest node in U . This gives an algorithm very close to the one proposed by Moore [Mor59]. The difference is that in Moore's algorithm, "stages" are clearly distinguishable because nodes to be admitted to U are saved and put in all at once when U becomes empty. It should be clear, from the development in Section 4, that Moore's algorithm (below) is a case of Ford's algorithm under a selective search strategy.

(10.8) ALGORITHM M: Moore's Algorithm [Mor59]

```
(1) [Intialize]  d(v):= ∞ for v ≠ s; d(s):= 0;
                 U:= {s};
(2) [Search and Replace]
          while U ≠ Ø do
            begin
              T:= Ø;
              while U ≠ Ø do
                begin
                  choose "oldest" u ∈ U; U:= U-{u};
                  for each (u,v) ∈ X do
                    if d(v) > d(u) + w(u,v) then
                      begin
                        d(v):= d(u) + w(u,v);
                        T:= T ∪ {v}
                      end
                end
              U:= T
            end
```

We may adapt Lemma 4.31 and conclude that, each time U becomes empty, a new arc has been added to any shortest path needed for a solution. Also, when T is assigned to U, at stage i, there must be no more than n-i members in T.

When the details of this argument are filled in it is clear that Algorithm M runs in time no greater than O(tα) and, since in no stage are all arcs considered, the constant must be better than for Algorithm BF (4.19). Of course, as with Algorithm W, termination will not occur if there is a cycle of negative weight reachable from s.

11. Arc Set Partition Algorithms

In this section we develop a class of arc set partition algorithms which includes Algorithm W (4.28) as a special case with a trivial partition of one subset. We then show how a unique partition on arc weights gives a new single source algorithm with improved order in a significant region of problem space. Yen's algorithm [Yen70], the best previously known for complete networks, is a variant (with reduced power) of an arc set partition algorithm under an arbitrary member of a set of two-subset partitions. Reduction in power comes from loss of information from the arc set partition algorithm, and from using a matrix representation for X .

11.1 The k-Partition Algorithm

Let there be a partition of the arc set $X = X_1 \cup X_2 \cup \ldots \cup X_k$, $X_i \cap X_j = \emptyset$ for $i \neq j$. In Algorithm S (4.26) let the rule of choice be to choose from $A \cap X_1$ for as long as $A \cap X_1 \neq \emptyset$, then choose from $A \cap X_2$ similarly, and continue in cyclic order over the partition of X until $A = \emptyset$. If we let

(11.1) $S_6(\ell)$ ≡ <u>begin</u>

```
        choose (u,v) ∈ (A ∩ X_ℓ); A:= A-{(u,v)};
        if d(v)>d(u)+w(u,v) then
          begin
            d(v):= d(u)+w(u,v);
            A:= A ∪ {(v,v̂)|(v,v̂) ∈ X}
          end
      end
```

we have the following algorithm for which Theorems 4.10 and 4.11 hold because the algorithm is simply Algorithm S upon which a rule of choice exhaustive of A has been imposed. (Note that termination of an iteration <u>while</u> $(A \cap X) \neq \emptyset$ <u>do</u> S_4 implies termination of <u>while</u> $(A \cap X_i) \neq \emptyset$ <u>do</u> $S_6(i)$ for any $X_i \subset X$.)

(11.2) ALGORITHM ASP(k): Basic k-Partition Algorithm

(1) [Initialize] $d(v) := \infty$ for $v \neq s$; $d(s) := 0$;
$A := \{(s,v) \mid (s,v) \in X\}$;

(2) [Search and Replace]

```
while A ≠ ∅ do
begin
  while A ∩ X_1 ≠ ∅ do S_6(1);
  while A ∩ X_2 ≠ ∅ do S_6(2);
    ⋮
  while A ∩ X_k ≠ ∅ do S_6(k)
end
```

In a direct parallel with the development of Algorithm W (4.28), let

(11.3) $S_7(\ell) \equiv$ begin

```
choose u ∈ U_ℓ; U_ℓ:= U_ℓ-{u};
for each (u,v) ∈ X_ℓ do
  if d(v)>d(u)+w(u,v) then
    begin
      d(v):= d(u)+w(u,v);
      U_1:= U_1 ∪ {v};
      U_2:= U_2 ∪ {v};
          ⋮
      U_k:= U_k ∪ {v}
    end
end
```

The resulting wave algorithm on arc set partitions is

(11.4) ALGORITHM ASPW(k): k-Partition Wave Algorithm

(1) [Initialize] $d(v):=\infty$ for $v \neq s$; $d(s):=0$;
$U_1:=U_2:=\ldots:=U_k:=\{s\}$;

(2) [Search and Replace]

```
while U_1 ∪ U_2 ∪ ... ∪ U_k ≠ ∅ do
  begin
    while U_1 ≠ ∅ do S_7(1);
    while U_2 ≠ ∅ do S_7(2);
        ⋮
    while U_k ≠ ∅ do S_7(k)
  end
```

To show Algorithm ASPW(k) is a refinement of Algorithm ASP(k) which preserves Theorems 4.10 and 4.11 we define

(11.5) $T \equiv$ (i) $A \cap X_\ell \subseteq \{(u,v) \mid u \in U_\ell$ and $(u,v) \in \dot{X}_\ell\}$ for $1 \leq \ell \leq k$
(ii) $A \subseteq \{(u,v) \mid u \in (U_1 \cup U_2 \cup \ldots \cup U_k)$ and $(u,v) \in X\}$.

It is obvious from the initialization steps of ASP(k) and ASPW(k) that T holds following initialization. It remains to prove that $T\left\{\begin{matrix} S_6\ (j) \\ S_7\ (j) \end{matrix}\right\}T$ for $1 \leq j \leq k$ provided $S_7(j)$ terminates. (Our notation means invariance of T over parallel executions of $S_6(j)$ and $S_7(j)$.) Consider (i) of T. For $\ell = j$, the result is immediate, since $A \cap X_j = \emptyset$ on termination of $S_6(j)$. For $\ell \neq j$, U_ℓ loses no members. Thus it is only new members of A which concern us. Let some arc $(v,\hat{v}) \in X_\ell$ be joined to A during execution of $S_6(j)$. This implies some (u,v) such that $(u,v) \in X_j$ and $u \in U_j$. If so, v is joined to U_ℓ in $S_7(j)$, which preserves (i) of T. Part (ii) of T follows directly as the union over all ℓ of (i).

We can extend Lemmas 4.31 and 4.33 to Algorithm ASPW(k). Let $P(Y)$ be a property defined on set Y as follows:

(11.6) $P(Y) \equiv$ for all v such that $d(v) \neq \mu(v)$, for all shortest paths p_v there exists j such that

$$p_v = (s=v_{i_1}, v_{i_2}, \ldots, v_{i_j}, \ldots, v_{i_m} = v) \text{ and}$$

(i) $d(v_{i_j}) = \mu(v_{i_j}) < \infty$

(ii) $d(v_{i_k}) > \mu(v_{i_k}),\ j < k \leq m$

(iii) $v_{i_j} \in Y$

(iv) if $v_{i_j} \notin U_\ell$ then $(v_{i_j}, v_{i_{j+1}}) \in (X - X_\ell)$,

and let R be a property defined on set Y and a predecessor set Y°:

(11.7) $R(Y,Y^\circ) \equiv Y \subseteq \{v \mid$ for some $u \in Y^\circ$ there exists a path q_{uv} such that $d^\circ(u) + w(q_{uv}) = d(v)$, where $d^\circ(u)$ is the value of $d(u)$ at the time $Y=Y^\circ\}$.

Also, we define $U \equiv U_1 \cup U_2 \cup \ldots \cup U_k$.

(11.8) LEMMA. *If* $\mu(v)$ *is defined for all* v *then*

$$P(U)(\text{while } U_\ell \neq \emptyset \text{ do } (S_7(\ell)[P(U)]))$$

for $1 \leq \ell \leq k$.

Proof: Parts (i), (ii), and (iii) of $P(U)$ (11.6) follow directly from the arguments used for Lemma 4.31. If $(v_{i_j}, v_{i_{j+1}}) \notin X_\ell$, preservation of (iv) follows immediately because $(v_{i_j}, v_{i_{j+1}}) \notin X_\ell \Rightarrow (v_{i_j}, v_{i_{j+1}}) \in (X - X_\ell)$. If $(v_{i_j}, v_{i_{j+1}}) \in X_\ell$, preservation of (iv) is trivial except when v_{i_j} is deleted from U_ℓ . But if v_{i_j} is deleted from U_ℓ , $v_{i_{j+1}}$ and possibly $v_{i_{j+r}}$, $j+1 < j+r \leq m$, will be found.

The last such node on p_v will become the new v_{i_j} which satisfies all of $P(U)$. □

(11.9) LEMMA. Under any input

$$(U^* = U)\{\underline{\text{while}}\ U = \emptyset\ \underline{\text{do}}\ (S_7(\ell)[R(U,U^*)]\}$$

for $1 \leq \ell \leq k$.

Proof: The proof is identical to the proof for Lemma 4.33. □

11.2 2-Partition Algorithms

Algorithm ASPW(2) (11.4), the algorithm under two-subset partitions, has a special property. If X_1 contains all arcs leading from s (that is, $\{(s,v) \mid (s,v) \in X\} \subseteq X_1$) , termination occurs after the first execution of an inner while-clause in which no point in d changes. This follows from the conditions $U_i = \emptyset$, prior to the inner while-clause on $X_{(i \bmod 2)+1}$, and $U_{(i \bmod 2)+1} = \emptyset$ after. Thus, no change in d implies $(U_1 \cup U_2) = \emptyset$. In Algorithm ASPW(2) there are two inner while-clauses. We call execution of one of them a half iteration.

With the above restriction on the partition $X = X_1 \cup X_2$ we can bound the number of half iterations on any problem for which $\mu(v)$ is defined for all v . To do so we observe an obvious property of paths. Any path is a concatenation of uniform subpaths composed entirely of arcs from X_1 alternated with uniform subpaths entirely from X_2 . (It is meant, of course, that concatenation eliminates one of the duplicate nodes where subpaths join.) For shortest paths originating in s , there will always be an initial uniform subpath from X_1 , and no shortest path of m arcs can have more than m uniform subpaths.

(11.10) LEMMA. *After half iteration i on Algorithm ASPW(2) (11.4) with all arcs from s in X_1, for all nodes v with a shortest path with i or fewer uniform subpaths, $d(v) = \mu(v)$.*

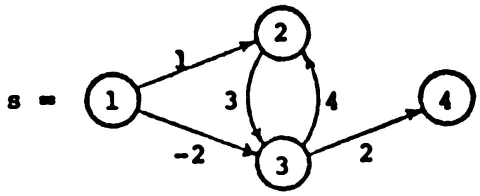
Proof: Lemma 11.8 implies that all shortest paths to nodes for which $d(v) > \mu(v)$ are extended by one uniform subpath at each half iteration. The lemma follows from an application of the iteration theorem. □

(11.11) COROLLARY. *Algorithm ASPW(2) (11.4) executes no more than t half iterations when X_1 contains all arcs from s, where t is the number of levels in a best solution tree.* □

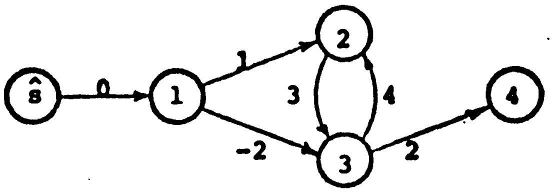
We are now ready to present our general single source algorithm based on ASPW(2). As in Section 4, we get a good bound if we adopt rules of choice which guarantee that, once chosen, a node will not return to U_j during the remainder of the half iteration in which it is chosen. Also, we introduce a transformation on the input network which allows any partition of X to meet the requirement that X_1 contain all arcs from s. For any problem $\phi(((V,X),w),s)$ define $\phi(((\hat{V},\hat{X}),\hat{w}),\hat{s})$ where for some $\hat{s} \notin V$

$$\hat{V} = V \cup \{\hat{s}\}$$
$$\hat{X} = X \cup \{(\hat{s},s)\}$$
$\hat{w}$ is w extended to $(\hat{s},s)$ so that $\hat{w}(\hat{s},s) = 0$.

For example, the following single source problem



is transformed into



The solutions to $\phi(N,s)$ and $\phi(\hat{N},\hat{s})$ are closely related: $\mu(\hat{s},u) = \mu(s,u)$ for all $u \in (V - \{s\})$. So it is sufficient to compute an assignment $\hat{d}$ for $(\hat{N},\hat{s})$ and report it as d for $\phi(N,s)$.

(11.12) ALGORITHM GP: General Single Source Algorithm on 2-Partitions

```
(1) [Initialize]
        (1a)  Initialize X₁ and X₂ such that
              X = X₁ ∪ X₂ and X₁ ∩ X₂ = ∅;
              X₁:= X₁ ∪ {(ŝ,s) | ŝ ∉ V, w(ŝ,s)=0};
              m:= number of nodes reachable from ŝ;
        (1b)  d(v):= ∞ for v ∈ V; d(ŝ):= 0;
              U₁:= {ŝ};  U₂:= ∅; count:= 0;
(2) [Search and Replace]
      while 0 < |U₁| ≤ m+1-count do
        begin
                   ⎧ innercount:= m+1-count;
                   ⎪ while 0 < |U₁| ≤ innercount do
                   ⎪   begin
                   ⎪     S₇(1) where the rule of choice guarantees
(11.13) S₈(1) ≡    ⎨          that u chosen will not return to
                   ⎪          U₁ if μ(v) is defined for all v;
                   ⎪     innercount:= innercount-1
                   ⎩   end
           count:= count+1;
           if |U₂| > 0 then
             begin
                   ⎧ innercount:= m+1-count;
                   ⎪ while 0 < |U₂| ≤ innercount do
                   ⎪   begin
        S₈(2) ≡    ⎨     S₇(2) where the rule of choice
                   ⎪          guarantees that u chosen will
                   ⎪          not return to U₂ if μ(v) is
                   ⎪          defined for all v;
                   ⎪     innercount:= innercount-1
                   ⎩   end
               count:= count+1
             end
        end
(3) [Check]  if |U₁| = 0 then report "success"
                         else report "failure".
```

Assuming the rules of choice can be realized, all we need to prove correctness is that the upper bounds on $|U_1|$ and $|U_2|$ are never violated on input for which $\mu(v)$ is defined for all v .

The variable count contains the number of half iterations executed. Because we have added the arc $(\hat{s},s)$, there may be m uniform subpaths in some shortest path, but no more. We conclude that the relation $0 < m+1 - \text{count}$ will never force premature termination when $\mu(v)$ is defined for all v . Also it is obvious that $U_1 = U$ at the beginning of $S_8(1)$; therefore the substitution of U_1 for U in the main control relation is valid.

Now let a set C_i and a property $Q(Y)$ on set Y be

(11.14) $C_i \equiv \{v | d(v) > \mu(v) \text{ when count became } i\}$

(11.15) $Q(Y) \equiv (|C_{count}| < m\text{-count}) \wedge (Y \subseteq C_{count-1})$,

where C_{-1} is defined as $V \cup \{\hat{s}\}$. It is obvious that $Q(U)$ holds following initialization (of course we mean $U = U_1 \cup U_2$). To show that the bound $|U_1| \leq m+1\text{-count}$ is not violated it is sufficient to prove

$$Q(U)(S_8(i);\ \text{count:= count+1})Q(U), \text{ for } i = 1,2.$$

But this invariance follows directly from Lemmas 11.9 and 11.10 since we know that there are at least $i+1$ nodes for which $d(v) = \mu(v)$ when all nodes u with shortest paths with i or fewer uniform subpaths are such that $d(u) = \mu(u)$. Thus

$|U_1| \leq$ m+1-count always holds when $\mu(v)$ is everywhere defined.

Furthermore, we can use property $Q(U)$ to verify the correctness of the bound innercount on $|U_1|$ and $|U_2|$. Since $U_1 \subseteq U \subseteq C_{count-1}$, and $S_8(1)$ (11.13) deletes from U_1 at each iteration a node which cannot return, the relation $|U_1| \leq$ innercount is always satisfied; and similarly for $|U_2| \leq$ innercount in $S_8(2)$.

Because of Theorem 4.11, we know that any finite bound will be exceeded when $\mu(v)$ is undefined for some v . The above arguments have proved

(11.16) THEOREM. Algorithm GP (11.12) always terminates and

(i) When $\mu(v)$ is defined for all v, Algorithm GP reports "success" and $d(v) = \mu(v)$ for all v upon termination.

(ii) When there exists v for which $\mu(v)$ is undefined, Algorithm GP reports "failure". □

We now develop a general expression for a bound on running time. To allow our work to be compared with the work of others in the field, we develop a bound on active operations.

Let $c_1(|U_1|)$ and $c_2(|U_2|)$ be monotonic functions which bound the active operations involved in choosing u in $S_7(1)$ and $S_7(2)$ (11.3), respectively. To allow flexibility in specifying rules of choice, we separate the cost of testing (and possibly changing) $d(v)$ into two components. There is a constant cost of 2, one add and one compare, for the test and change, since the second addition we show can be eliminated by storing the result of the addition $d(u) + w(u,v)$ for use in both the test and replacement. The other component of the cost is the monotonic function $\bar{b}_1(|U_1|)$ or $\bar{b}_2(|U_2|)$, bounds on the active operations, if any, involved in immediate re-ordering of the candidate set in $S_7(1)$ and $S_7(2)$ respectively.

(11.17) LEMMA. Algorithm GP terminates in

$$\tau(n,\alpha) \le \frac{1}{2}\left(\sum_{k=1}^{n+1} \sum_{i=1}^{k} (c_1(i) + c_2(i))\right) + \alpha\left(n - \frac{\alpha}{2n}\right)$$

$$+ \bar{b}_1(n+1)\,\frac{|X_1|}{2}\left(n - \frac{|X_1|}{2n}\right)$$

$$+ \bar{b}_2(n+1)\,\frac{|X_2|}{2}\left(n - \frac{|X_2|}{2n}\right)$$

active operations.

Proof: We make use of the bounds on set sizes in Algorithm GP to sum bounds on active operations.

$$\text{(11.18)}\quad \tau(m,\alpha) \le \sum_{\substack{k=m+1,\\ m-1,\ldots,\\ \begin{cases}1 \,(m \text{ even})\\ 2\,(m \text{ odd})\end{cases}}} \sum_{i=1}^{k} \left(c_1(i) + (2+\bar{b}_1(i))\cdot |\{(u_{j_i},v) \mid (u_{j_i},v) \in X_1\}|\right)$$

$$+ \sum_{\substack{k=m,\\ m-2,\ldots,\\ \begin{cases}1 (m\ \text{odd})\\ 2 (m\ \text{even})\end{cases}}} \quad \sum_{i=1}^{k} \left(c_2(i) + (2+\bar{b}_2(i))\cdot |\{(u_{r_i},v \mid (u_{r_i},v) \notin X_2\}|\right)$$

The first sum in (11.18) is for all executions of $S_7(1)$, the second for $S_7(2)$ (11.3). The inner sums are of the same form as for Algorithm W (4.28) as found in Lemma 4.34, giving the active operations expended in one execution of S_7.

Except for $\hat{s} \in U_1$, any node $v \in U_1$ either belonged to U_2 in the previous half iteration or will belong to U_2 in the next. Thus in (11.18) we can telescope the constant active operations involved in testing and replacing $d(v)$. Also using the monotonicity of c_1, c_2, b_1, b_2, we get

$$(11.19)\quad \tau(m,\alpha) \le \frac{1}{2} \sum_{k=1}^{m+1} \sum_{i=1}^{k} \Big(c_1(i) + c_2(i) + 2|\{(u_{t_i},v) \mid (u_{t_i},v) \in X\}| + \bar{b}_1(i)|\{(u_{j_i},v) \mid (u_{j_i},v) \in X_1\}| + \bar{b}_2(i)|\{(u_{r_i},v) \mid (u_{r_i},v) \in X_2\}|\Big).$$

The sum of the memberships of a sequence of subsets of an arc set Y may be bounded as follows:

$$(11.20)\quad \sum_{k=1}^{m+1} \sum_{i=1}^{k} |\{(u_{j_i},v) \mid (u_{j_i},v) \in Y\}| \le \sum_{i=1}^{|Y|/(n-1)} (m-1)i + \sum_{i=1+|Y|/(n-1)}^{m+1} |Y| \le |Y| \left(m - \frac{|Y|}{2m}\right), \quad \text{for } 0 \le |Y| \le m+1 \cdot$$

Substituting (11.20) in (11.19), and using the fact $n \geq m$, we get

$$(11.21)\quad \tau(n,\alpha) \leq \frac{1}{2} \sum_{k=1}^{n+1} \sum_{i=1}^{k} (c_1(i) + c_2(i)) + \alpha(n - \frac{\alpha}{2n}) + \bar{b}_1(n+1) \frac{|X_1|}{2} (n - \frac{|X_1|}{2n}) + \bar{b}_2(n+1) \frac{|X_2|}{2} (n - \frac{|X_2|}{2n}). \quad \square$$

A Unique Partition

It remains to specify a partition and rules of choice in Algorithm GP (11.12). If we let

$$(11.22)\quad X_1 = X_p = \{(u,v) \mid (u,v) \in X,\ u \text{ reachable from } s,\ w(u,v) \geq 0\}$$

$$X_2 = X_m = \{(u,v) \mid (u,v) \in X,\ u \text{ reachable from } s,\ w(u,v) < 0\},$$

a unique partition of X is defined which gives a good rule of choice in each half iteration. Dijkstra's rule (9.1) may be used on the subnetwork (V,X_p). Since (V,X_m) must be acyclic if there is no cycle of negative weight reachable from s, the acyclic rule based on an a priori ordering of V is correct on (V,X_m). To implement Dijkstra's rule efficiently we can maintain U_1 as a tree in one of the ways used in Section 9.1.2. To implement the rule of choice on the acyclic subgraph (V,X_m) , we will need an ordering of the nodes on X_m such as appears in Algorithm A (7.1). Ordering V is best done in initialization. So, under $X = X_p \cup X_m$, initialization in Algorithm GP (11.12) becomes

(11.23) (1) [Initialize]

(1a) $\bar{X} := X \cup \{(\hat{s},s) \mid \hat{s} \notin V,\ w(\hat{s},s) = 0\}$;
$X_1 := \{(u,v) \mid (u,v) \in \bar{X},\ w(u,v) \geq 0,$ and u reachable from $\hat{s}\}$;
$X_2 := \{(u,v) \mid (u,v) \in \bar{X},\ w(u,v) < 0,$ and u reachable from $\hat{s}\}$;
m:= number of nodes reachable from $\hat{s}$;

(1b) $d(v) := \infty$ for $v \in V$; $d(\hat{s}) := 0$; $U_1 := \{\hat{s}\}$;
$U_2 := \emptyset$; count:= 0;

(1c) Generate an indexing $(i_1, i_2, \ldots, i_m)$ of nodes reachable from $\hat{s}$ such that $j < k$ if $(V[i_j], V[i_k]) \in X_2$;
<u>if</u> such an indexing does not exist
<u>then</u> report "failure".

It should be clear that the initialization is $O(n+\alpha)$. Necessary details of $S_8(1)$ and $S_8(2)$ may be adopted from the presentations of Algorithms W (4.28) and A (7.1).

(11.24) <u>THEOREM</u>. <u>Under</u> $X = X_p \cup X_m$ <u>Algorithm GP (11.12) has the following bounds</u>

(a) $\tau(n,n(n-1)) \leq \frac{7}{12} n^3$ <u>with an unordered set realization for</u> U_1,

(b) $T(n,\alpha) \leq O(n^{2+1/k} + \alpha n)$ with a constant-depth tree of $k+1$ levels for U_1,

(c) $T(n,\alpha) \leq O(\beta n^2 \log_\beta n + n\alpha(1+\log_\beta(\frac{n^2}{\alpha})))$ with a tree with degree of branching β for U_1.

Proof: Since initialization is $O(n+\alpha)$ we need only consider Step (2) of the algorithm.

For part (a), we use an unordered realization for U_1, so to implement Dijkstra's rule (9.1) costs $c_1(i) = i$. The acyclic rule gives $c_2(i) = 0$, because no active operations are required. By the same argument, $\bar{b}_1(i) = \bar{b}_2(i) = 0$. Since $\alpha = n(n-1)$ we get from the bound of Lemma 11.17

$$\tau(n,n(n-1)) \leq \frac{1}{2} \left(\sum_{k=1}^{n+1} \sum_{i=1}^{k} i \right) + n^2(n - \frac{n^2}{2n})$$

$$\leq \frac{7}{12} n^3.$$

For part (b) we use a tree with $k+1$ levels for U_1. Thus $c_1(i) \leq kn^{1/k}$ to implement Dijkstra's rule. Again $c_2(i) = \bar{b}_2(i) = 0$. We assume immediate reordering of U_1, so $\bar{b}_1(n) = k$. Since no operations dominate active ones, substituting in the expression of Lemma 11.17 gives

$$\tau(n,\alpha) \leq \frac{1}{2} \sum_{j=1}^{n+1} \sum_{i=1}^{j} kn^{1/k} + \alpha(n - \frac{\alpha}{2n}) + k\frac{\alpha}{2}(n - \frac{\alpha}{2n})$$

$$\leq \frac{1}{4} kn^{2+1/k} + (1+\frac{k}{2})\alpha(n - \frac{\alpha}{2n})$$

which implies

$$T(n,\alpha) \leq O(n^{2+1/k} + \alpha n).$$

For part (c), we use the other bounds developed in Section 9.1.2. Cost of reordering is incorporated into the cost of selection, so we get, from Theorem 9.28,

$$\sum_{i=1}^{j} c_1(i) \leq O(\beta j \log_\beta j + \alpha(1 + \log_\beta(\frac{j^2}{\alpha}))),$$

implying

$$T(n,\alpha) \leq O(\beta n^2 \log_\beta n + n\alpha(1 + \log_\beta(\frac{n^2}{\alpha}))). \quad \square$$

For any $\alpha \geq n-1$ a worst case problem for Algorithm GP (11.12) under $X = X_p \cup X_m$ has linear (depth n) best solution trees and in each linear tree negative and nonnegative arcs alternate. For Algorithm BF (4.19), by comparison, any problem with linear best solution trees can be a worst case for some ordering of X. When there are no arcs of negative weight, Algorithm GP is $O(n)$ better than Algorithm BF, and in general, this improvement holds true for problems with a "few" negative arcs.

A Non-Unique Partition and Yen's Algorithm

If an ordering $V = (1,2,\ldots,n)$ is imposed on the nodes then $X = \vec{X} \cup \overleftarrow{X}$ is a partition of X, where $\vec{X}$ is the set of arcs in which indices increase from origin to destination node, and $\overleftarrow{X}$ is the set of arcs in which indices decrease. It is our intention to use the partition $X = \vec{X} \cup \overleftarrow{X}$ in

Algorithm GP. In order to realize good rules of choice we further partition $\vec{X}$ and $\overleftarrow{X}$ giving

$$\vec{X} = \{\vec{X}_1 \cup \vec{X}_2 \cup \ldots \cup \vec{X}_n\}$$

where $\vec{X}_i = \{(i,j) \mid (i,j) \in X \text{ and } i < j\}$,

and

$$\overleftarrow{X} = \{\overleftarrow{X}_n \cup \overleftarrow{X}_{n-1} \cup \ldots \cup \overleftarrow{X}_1\}$$

where $\overleftarrow{X}_i = \{(i,j) \mid (i,j) \in X \text{ and } i > j\}$.

Let the rule of choice in $S_8(1)$ (11.13) be

"choose $u \in U_1$ such that $u = \min\limits_{v \in U_1} (v)$"

and in $S_8(2)$

"choose $u \in U_2$ such that $u = \max\limits_{v \in U_2} (v)$".

In other words, when exploring under $\vec{X}$, choose the originating node with minimum index, and conversely under $\overleftarrow{X}$. It is obvious that these rules satisfy the requirement of no return to U_1 and U_2.

(11.25) <u>THEOREM</u>. <u>Under</u> $X = \vec{X} \cup \overleftarrow{X}$ <u>Algorithm</u> <u>GP</u> <u>(11.12)</u> <u>runs</u> in $\tau(n,\alpha) \leq \alpha(n - \frac{\alpha}{2n})$ <u>active</u> <u>operations</u> <u>and</u> <u>time</u> $T(n,\alpha) \leq O(n^2 + \alpha n)$ <u>overall</u>.

<u>Proof</u>: The proof is immediate from Lemma 11.17, since there are no active operations in choosing u from or inserting v

into U_1 or U_2. Loop control is, of course, $O(n^2)$. □

(11.26) *COROLLARY. Under* $X = \overrightarrow{X} \cup \overleftarrow{X}$ *Algorithm GP runs in* $\tau(n,n(n-1)) \le \frac{1}{2} n^3$ *active operations.* □

(11.27) *COROLLARY.* (Yen [Yen70]). *There exists an algorithm for the unrestricted single source problem which runs in* $\tau(n,\alpha) \le \frac{2}{3} n^3$ *active operations.* □

This latter result is the bound for Yen's Algorithm [Yen70], which may be viewed as an implementation of Algorithm GP under $X = \overrightarrow{X} \cup \overleftarrow{X}$ in which all arcs absent from X are incorporated into the problem description as dummy arcs with infinite weight. The constant 2/3 (versus 1/2 for Algorithm GP) arises because the identity of the nodes in U_1 and U_2 is not maintained, but only the cardinalities, $|U_1|$ and $|U_2|$.

The worst case region for Algorithm GP under $X = \overrightarrow{X} \cup \overleftarrow{X}$ is identical to the region for Algorithm BF, the entire set of linear (depth n) solution trees. It is true that an indexing for V exists under which some linear solution tree is one uniform subpath. If such an indexing is known, the shortest path problem on unrestricted networks is reduced to at most two problems on acyclic networks. Given known algorithms, then, finding a "good" indexing is equivalent within a constant factor to the unrestricted single source problem. With no sufficiently efficient way known to generate an indexing, worst case analysis requires that a "worst" indexing be assumed.

12. Single Source Results Summarized

The basic idea in all the single source algorithms we have discussed is to continue searching for arcs which will reduce some known path weight until no such arc can be found or some time bound has been exceeded. This idea can be developed in two major directions (Section 4), one using an exhaustive search and the other using a "breadth-first" restricted search. There are naive exhaustive search algorithms with exponential running times, but the best exhaustive search algorithm, Algorithm BF (4.19), runs in $O(n\alpha)$ time.

The fundamental restricted search, or adaptive, algorithm is Algorithm W (4.28). It, too, is exponential on unrestricted networks (Section 10), but consideration of good rules of choice in the set U' of "active" nodes leads to optimum algorithms (running time $O(n+\alpha)$) for a decision procedure for shortest path solutions (Section 5), a constructor for minimal depth solution trees (Section 6), and algorithms for finding shortest paths in acyclic or uniform positive arc weight networks (Sections 7 and 8).

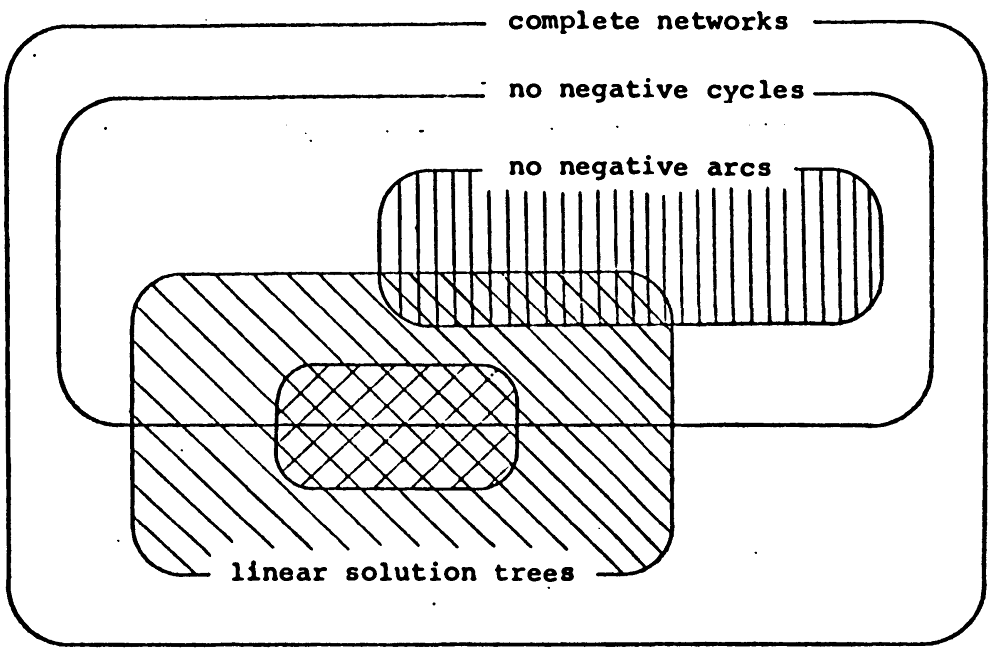
The most important class of networks on which Algorithm W has good running times is the class of those with non-negative arc weights (Section 9). We use a fixed-depth heap to give a very simple algorithm which runs in time $O(n^{1+1/k}+\alpha)$. We extend the analysis of a much more complex scheme [Joh72] and show that it is asymptotically no better, and worse in the multiplicative constant.

Perhaps the most interesting result based on Algorithm W is its generalization to the class of arc set partition algorithms (Section 11). In this class we find a new algorithm (superior to any previously known), Algorithm GP (11.12) under certain partitions, in particular $X_p \cup X_m$.

Algorithm GP under $X_p \cup X_m$ is important for two main reasons. First, it runs in the best asymptotic bounds known on any single source problem, except for problems on acyclic networks with very few arcs. We can, of course, extend our claim to these networks as well by incorporating a cycle test, at a cost of $O(n+\alpha)$, into initialization. The adaptive properties of the algorithm are particularly interesting on networks with "few" negative arcs, a class for which no really satisfactory algorithm has before existed.

The second important property of Algorithm GP under $X_p \cup X_m$ is the small class of problems on which worst case running time is realized. As discussed in Section 11, for any α the worst case region in problem space for Algorithm GP under $X_p \cup X_m$ is properly contained in the worst case region for algorithms previously known. We summarize this result (for the case of complete networks) in the Venn diagram below.

Finally, there is the problem of detecting negative cycles in networks with arcs of negative weight. While it is possible, for instance, to decide in time $O(r+\alpha)$ whether a network is acyclic or has no negative arcs, detecting the existence of negative cycles is not doable (as far as we know) in trivial $O(n+\alpha)$ time. Consequently we allow *a priori* classification in the case of acyclic or nonnegative arc networks, but forbid it on the question of negative cycles. Our treatment of the question of negative cycles is embedded in our treatment of algorithms for unrestricted networks. In each case, our bounds are bounds for negative cycle detection as well as finding shortest paths. We find the two problems can be solved in times of equal order.



WORST CASE REGIONS

Algorithms BF and GP under $\vec{X} \cup \overleftarrow{X}$

Algorithm GP under $X_p \cup X_m$

Algorithm W (restricted to nonnegative arcs)

PART III: ALL PAIRS ALGORITHMS

13. Non-Adaptive Algorithms

13.1 Triple Algorithms

Among algorithms which compute shortest path weights by adding path weights and comparing sums, non-adaptive algorithms are those with a predetermined sequence of adds and compares. The best such algorithms known are composed of a predetermined sequence of triple operations in which a path weight is replaced by the minimum of its former value and the sum of two other path weights. Iri and Nakamori [I&N72] exhibit a set of algorithms which perform exactly $n(n-1)(n-2)$ triple operations and prove this class optimal among all non-adaptive algorithms using triple operations.

Some of these algorithms were known prior to the work of Iri and Nakamori. The earliest to appear in the literature is an extension by Floyd [Flo62a] of an algorithm by Warshall [War62] for the transitive closure of Boolean matrices. Any transitive closure problem is equivalent to some shortest path problem with all arc weights zero. In a sense, however, this equivalence is deceptive, for as we discuss shortly there are better methods for transitive closure which do not generalize to less restricted path problems.

The original Warshall-Floyd algorithm is, in fact, an adaptive algorithm since the sequence of triple operations depends on the data. But this dependence is sufficiently weak for there to be no effect on asymptotic performance in worst cases with as few as $2n-2$ arcs. Furthermore, there is an important extension, in which exactly n^3 triple operations are performed, which detects cycles of negative weight. The original algorithm assumes $\mu(v,u)$ to be defined for all pairs (v,u) . Because of these details we will present the Warshall-Floyd algorithm in a generalized form.

(13.1) ALGORITHM WF(S): Generalized Warshall-Floyd Algorithm for the All Pairs Problem

(1) [Initialize] $d(i,j) := w(i,j),\ (i,j) \in X;$
$d(i,j) := \infty,\ (i,j) \notin X,\ i \neq j;$
$d(i,i) := 0,\ i \in V;$

(2) [Search and Replace]

```
k:= 0;
while k < n do
  begin
    k:= k+1;
    S
  end
```

(3) [Check] **if** $d(i,i) < 0$ for any $i \in V$ **then** report "failure" **else** report "success".

To particularize Algorithm WF(S) consider the following statement:

(13.2) $S_9 \equiv$ for $i \in V$ do
for $j \in V$ do
$d(i,j) := \min(d(i,j), d(i,k)+d(k,j))$.

(13.3) THEOREM. Algorithm WF(S_9) terminates in $O(n^3)$ operations and

(i) reports "success" and $d(i,j) = \mu(i,j)$ for all pairs (i,j) if there is no cycle of negative weight,

(ii) reports "failure" otherwise.

Proof: The time bound is obvious from inspection of the program.

For correctness, let

$$T \equiv d(i,j) = \min_{v_\ell \in V_k} (w(i, v_{\ell_1}, v_{\ell_2}, \ldots, v_{\ell_m}, j)),\text{ where}$$

$V_k = \{1,2,\ldots,k\}$. Clearly T holds before Step (2) of (13.1). To prove $T\{$if $k < n$ then $(k := k+1;\ S_9)\}T$, let $\hat{k} = k+1$ for the k under which T is satisfied initially. It is clear that S_9 examines every triple $\langle i,\hat{k},j\rangle$, replacing $d(i,j)$ if and only if there is a shorter path via $\{1,2,\ldots,\hat{k}\}$ than via $\{1,2,\ldots,\hat{k}-1\}$. But this satisfies T after S_9. By the iteration theorem we have

$(k=0) \wedge T\{\underline{\text{while}}\ k < n\ \underline{\text{do}}\ (k := k+1;\ S_9)\}\ T \wedge (k=n)$.

If there is no cycle of negative weight, $(T \wedge (k=n))$ implies $d(i,j) = \mu(i,j)$ for all $i,j \in V$. Otherwise, $(T \wedge (k=n))$ implies $d(i,i) < 0$ for some $i \in V$. □

This is the basic result. As stated above, the original algorithm [Flo62a] differs slightly. It examines only the triples $\langle i,k,j \rangle$ for which i,j,k are distinct, and doesn't look at triples for which one of the summands is ∞. We incorporate these modifications as follows:

(13.4) $S_{10} \equiv$ <u>for</u> $i \in (V - \{k\})$ <u>do</u>
<u>if</u> $d(i,k) < \infty$ <u>then</u>
<u>for</u> $j \in (V-\{i,k\})$ <u>do</u>
$d(i,j) := \min(d(i,j), d(i,k)+d(k,j))$.

Appropriate modifications of the proof of Theorem 13.3 give

(13.5) <u>COROLLARY</u>. <u>Algorithm</u> <u>WF</u>(S_{10}) <u>executes no more than</u> $n(n-1)(n-2)$ <u>triple operations; if there is no cycle of negative weight</u>, $d(i,j) = \mu(i,j)$ <u>for all</u> (i,j). □

The adaptation has little power in general, however, as the following theorem demonstrates.

(13.6) <u>THEOREM</u>. <u>For</u> $\alpha \geq 2n-2$, <u>Algorithm</u> <u>WF</u>(S_{10}) <u>executes</u> $T(n,\alpha) = O(n^3)$ <u>operations in the worst case</u>.

<u>Proof</u>: Let N be a network with $2n-2$ arcs as follows:

$$X = \{(1,2),(1,3),\ldots,(n,n),(2,1),(3,1),\ldots,(n,1)\}.$$

When $k=1$ to start with, there are $(n-1)(n-2)$ triple operations on $\langle i,1,j\rangle$, filling out the entire matrix d with non-infinite entries. A similar result can be had regardless of the first value k assumes. □

Because of Theorem 13.6, we may consider Algorithms $WF(S_9)$ and $WF(S_{10})$ (13.1, 13.2, 13.4) equivalent under worst cases. In effect, the Warshall-Floyd algorithm is non-adaptive.

13.2 Multiplication Algorithms

Let us now restrict non-adaptive algorithms, using triple operations, to sequences of triples in which the triple $\langle i,k,j\rangle$ occurs in a subsequence with elements, in some order, from $\{\langle i,1,j\rangle,\langle i,2,k\rangle,\ldots,\langle i,n,j\rangle\}$. Since min is associative and commutative, such a subsequence is equivalent to the "plus-min" inner product

$$\begin{aligned} d(i,j) :&= (d(i,1) + d(1,j)) \perp (d(i,2) + d(2,j)) \perp \ldots \\ &\perp (d(i,n) + d(n,j)) = \min_k (d(i,k) + d(k,j)), \end{aligned}$$

where $\perp$ is infix notation for minimization.

Farbey, Land, and Murchland [FLM67] show an algorithm restricted to sequences of "plus-min" inner products with $\tau(n,n(n-1)) \approx 4n^3$, exactly twice the bound for optimal algorithms on unrestricted triple sequences. (Hu [Hu67] gives an extensive discussion also.) The bound is achieved by a sequence which is the concatenation of a forward and reverse row-wise sequence of inner products:

$$((1,1),(1,2),\ldots,(1,n),(2,1),\ldots,(n,n),$$
$$(n,n-1),\ldots,(n,1),(n-1,n),\ldots,(1,1)),$$

where (i,j) stands for $d(i,j) := \min_k(d(i,k)+d(k,j))$. It is not surprising that the best known bound under sequences of inner product operations is not as good as on unrestricted sequences of triples.

The Farbey-Land-Murchland algorithm was called by them the "cascade algorithm" because of their view of it as an improvement over "plus-min" matrix multiplication in which "new values are used as soon as they are computed." If, indeed, matrix multiplication is employed to solve the all pairs problem, $\lceil \log(n-1) \rceil$ multiplications are necessary to find all shortest paths. The first squaring of the matrix $D = [d(i,j)]$ gives a matrix D^2 in which shortest paths have been found for all pairs for which shortest paths with two or fewer arcs exist. Subsequent squarings are such that the ith square, D^{2^i}, has found paths with 2^i or fewer arcs. Thus repeated squaring is an $O(n^3 \log n)$ method.

Matrix multiplication on integer matrices can be done in $O(n^\gamma)$, where $\gamma = \log_2 7$, approximately 2.81, using Strassen's algorithm [Str69]. This result gives an $O(n^\gamma)$ algorithm for transitive closure in which a matrix product on the integers is normalized to (0,1) . The reader is referred to a paper by Fischer and Meyer [F&M72] for a discussion of this and related results. Unfortunately, we cannot generalize these methods to shortest path problems. Strassen's algorithm relies on the ring properties of the integers. In "plus-min" multiplication, min is the analog to + in ordinary matrix multiplication, and there is no inverse under min. In fact, Kerr [Ker70] has shown that n^3 additions are necessary for "plus-min" multiplication. Thus we conclude that fast matrix multiplication will not give any direct aid in reducing the bound of non-adaptive all pairs algorithms.

13.3 Optimality Without Restriction to Triple Operations

Kerr's proof (pp. 61-64, [Ker70]) argues on the form of expressions for the elements of the product matrix of an arbitrary "plus-min" multiplication. We can use his technique to strengthen the optimality result for non-adaptive "plus-min" algorithms cited in Section 13.1 so that it holds for any algorithm, not just those restricted to triple operations.

Consider the problem of squaring a matrix A from the class $\mathcal{A}$ of arbitrary, square, integer matrices with zero diagonal elements and such that $A \cdot A = A^*$, where $A \cdot A^* = A^*$ and, of course, "plus-min" multiplication is meant.

To get the needed result on $\mathcal{A}$, we must have the following lemma:

(13.7) <u>LEMMA</u>. <u>The following matrices with diagonal elements zero, and for</u> $1 \le i \ne j \le n$, <u>belong to</u> $\mathcal{Q}$:

$$A_1(i,j) \equiv \min(a_{i1}+a_{1j},\dots,a_{in}+a_{nj}) = 1,$$
$$a_{k\ell} = 0 \text{ for } k \ne i \text{ and } \ell \ne j,$$

$$A_2(i,j) \equiv a_{ik} = a_{kj} = 0 \text{ for some } k,$$
all other off-diagonal elements
equal 1,

$$A_3(i,j) \equiv a_{ik} = a_{kj} = 1 \text{ for some } k,$$
$a_{\ell m} = 0$ for some $a_{\ell m}$ not a_{ik}, a_{kj},
or a_{ij},
all other off-diagonal elements equal 2.

<u>Proof</u>: In each case what must be verified is that $A^2 = A^*$.
$A_1(i,j)$: Clearly $a^2_{k\ell} = 0$ for $k \ne i$ and $\ell \ne j$, and $a^2_{ij} = 1$. For the other elements in the ith row (jth column), they are all 1 in A_1^2 if the corresponding row (column) is all 1's. Otherwise all other row (column) elements in A_1^2 are 0 . In either case the row (column) is invariant under further premultiplication.
$A_2(i,j)$: $a_{ik} = a_{kj} = 0 \Rightarrow a^2_{ik} = a^2_{kj} = a^2_{ij} = 0$. All other off-diagonal elements in A_2^2 are 1 , implying $A_2^2 = A_2^3$.
$A_3(i,j)$: $A_3^2 = A_3$, so $A_3^2 = A_3^3$. □

An optimal non-adaptive algorithm for $A \cdot A$ must compute a set of expressions $\{E_{ij} \mid 1 \le i \ne j \le n\}$ such that

$$E_{ij} = M_{ij} \equiv (a_{i1}+a_{1j}) \perp \cdots \perp (a_{ij}+0) \perp \cdots \perp (0+a_{ij}) \perp \cdots \perp (a_{in}+a_{nj}) \ .$$

Eliminating additions of zero, M_{ij} is easily reduced to an expression with $n-2$ additions and $n-2$ dyadic minimizations.

Whatever the form of E_{ij}, because of associativity and commutativity of plus and min together with distributivity of plus over min, there exists $\bar{E}_{ij} = E_{ij}$ of the form

$$\bar{E}_{ij} \equiv S_1 \perp S_2 \perp \cdots \perp S_t$$

where $S_1,\ldots,S_t$ are sums of off-diagonal elements of A.

We claim that $\bar{E}_{ij}$ has the form

$$\begin{aligned}\bar{E}_{ij} \equiv\ & (a_{i1}+a_{1j}) \perp \cdots \perp a_{ij} \perp \cdots \perp (a_{in}+a_{nj}) \\ & \perp (a_{i1}+a_{1j}+[\ ]) \perp \cdots \perp (a_{ij}+[\ ]) \perp \cdots \\ & \perp (a_{in}+a_{nj}+[\ ])\end{aligned}$$

where [ ] stands for any expression whatever. To prove our claim, we first assume to the contrary that there is some term $(\alpha+ \ldots +\gamma)$ in $\bar{E}_{ij}$ which contains neither a_{ij} nor some pair a_{ik} and a_{kj}. Setting $\alpha = \ldots = \gamma = 0$, all a_{ik} and a_{kj}, $1 \leq k \leq n$, which are not $\alpha, \ldots,$ or γ to 1, and all other a's to 0 gives $E_{ij} = 0$ but $M_{ij} = 1$. Since by Lemma 13.7 any such matrix A is such that $A \cdot A = A^*$, there is a contradiction.

On the other hand, assume in contradiction to the form of $\bar{E}_{ij}$ that some term $(a_{ik}+a_{kj})$ is absent. Setting

$a_{ik} = a_{kj} = 0$ and all other off-diagonal elements to 1 gives $\bar{E}_{ij} = 1$ but $M_{ij} = 0$. This matrix is of form $A_2(i,j)$, so by Lemma 13.7 it is in the class $\mathcal{A}$. This proves that $\bar{E}_{ij}$ is of the form claimed.

If $(a_{ik}+a_{kj})$ in $\bar{E}_{ij}$ did not appear in E_{ij} but was obtained using distributivity, it must have come either from $a_{ik} + (a_{kj} \perp [\ ])$ or $(a_{ik} \perp [\ ]) + a_{kj}$. We conclude that, even if these expressions were intermediate results, there must by repeated reduction be in E_{ij} an expression of the form $(a_{ik} \perp [\ ]) + (a_{kj} \perp [\ ])$, so E_{ij} must be of the form

$$(13.8)\quad E_{ij} \equiv ((a_{i1} \perp [\ ]) + (a_{1j} \perp [\ ])) \perp \cdots \perp (a_{ij} \perp [\ ]) \perp \cdots \perp ((a_{in} \perp [\ ]) + (a_{nj} \perp [\ ])) \perp [\ ] .$$

Arguing on the expressions, we have shown that there must be $n-2$ additions for each element of A^*. But, perhaps, there are common subexpressions, implying fewer than $n(n-1)(n-2)$ additions over all. If this were true, there would have to be two identical additions implying an addition of the form $(a_{ik} \perp a_{\ell m} \perp [\ ]) + (a_{kj} \perp [\ ])$, where $a_{\ell m}$ is neither a_{ik}, a_{kj}, nor a_{ij}. However, let $a_{ik} = a_{kj} = 1$, $a_{\ell m} = 0$, and all other off-diagonal elements of A equal 2. Such a matrix is of the form of $A_3(i,j)$, so it is a member of $\mathcal{A}$. But these values give $E_{ij} = 1$ when $M_{ij} = 2$, proving that "plus-min" multiplication in $\mathcal{A}$ requires $n(n-1)(n-2)$ additions, and in particular, that each E_{ij} must actually be computed according to the form in (13.8). We know how to compute E_{ij} with all expressions [ ] empty (simply the expression M_{ij}), so we conclude that the inner min operations are not necessary. The n-2 outer ones are, however, because all sums are unique. This gives us

(13.9) LEMMA. *An optimal non-adaptive algorithm for squaring an arbitrary matrix in $\mathcal{A}$ must use $n(n-1)(n-2)$ additions and an equal number of dyadic minimizations if no other operations are allowed.* □

With this lemma on the restricted class of matrices $\mathcal{A}$, our optimality result is immediate.

(13.10) THEOREM. *An optimal non-adaptive algorithm for shortest paths between all pairs must have $n(n-1)(n-2)$ additions and $n(n-1)(n-2)$ dyadic minimizations if no other operations are allowed.*

Proof: If shortest paths could be found in fewer operations in a non-adaptive algorithm, squaring in $\mathcal{A}$ could be done in fewer, since A^2 is the shortest path matrix for $A \in \mathcal{A}$. But Lemma 13.9 has shown faster squaring is impossible. □

The theorem allows us to conclude that Algorithm WF(S_{10}) (13.1, 13.4) is optimum with respect to additions and minimizations on path weights among non-adaptive algorithms. Thus triple operations cannot be improved on in the context of non-adaptive algorithms using the component operations.

14. Adaptive Algorithms for the All Pairs Problem

14.1 Decomposition Algorithm

In order to define a (node) cutset in a digraph we refer to the equivalent (undirected) graph, the graph G on the same nodes but with an undirected edge wherever in the digraph there is an arc in either direction. Two nodes u and v are adjacent in G if G has an undirected edge (u,v). Two subsets $A_1 \subset V$ and $A_2 \subset V$ are non-adjacent in G if for no $u \in A_1$ and $v \in A_2$ are u and v adjacent. A cutset in a network N with equivalent graph G is a set $Y \subset V$ such that there is a non-trivial partition $V = A_1 \cup Y \cup A_2$ such that A_1 and A_2 are non-adjacent in G. A network N is linearly decomposable (see Hu [Hu68]) if there exist k disjoint and non-adjacent cutsets $Y_i \subset V$, $1 \leq i \leq k$. This definition implies a partition

$$V = A_1 \cup Y_1 \cup A_2 \cup Y_2 \cup \ldots \cup Y_k \cup A_{k+1}$$

such that, for $1 \leq i \leq k$, Y_i is a cutset of the subgraph on $A_i \cup Y_i \cup A_{i+1}$.

Interest in linearly decomposable networks arises because of an algorithm of Hu and Torres [H&T69] which has an improved bound for the all pairs problem on such networks, provided k is large enough. Their bound for their algorithm ranges from $O(n^2)$ to $O(n^3)$ as k ranges from $O(n)$ to $O(1)$. The algorithm solves, in an order depending on the decomposition, individual all pairs problems for the subnetworks $Y_i \cup A_{i+1} \cup Y_{i+1}$. New values are employed as soon as they are obtained, as in the cascade method.

There are two important questions to be answered if this decomposition algorithm is to be useful. First, are all highly incomplete networks linearly decomposable with some k which grows with n? And, in any case, can a large set of cutsets be identified efficiently?

(14.1) LEMMA. <u>For all $\alpha \geq 2n-2$ there exist networks with no more than one node cutset.</u>

<u>Proof</u>: Let $N = (V,X)$ such that there are two distinguished nodes s and t of which one or the other is incident to every arc. That is,

$$X = \{(s,v) \text{ or } (v,s) \mid v \in (V - \{s,t\})\} \cup \{(t,v) \text{ or } (v,t) \mid v \in (V - \{s,t\})\} .$$

Arcs can be defined so that N is strongly connected, but this is unimportant for the proof at hand.

In G equivalent to N no two nodes are more than two edges apart. Therefore there can be at most one node cutset in N. Adding arcs cannot increase the number of cutsets. □

We conclude that there are networks with as few as $\alpha = 2n-2$ arcs on which Hu and Torres' algorithm takes $O(n^3)$ operations.

The second question, as far as we know, defines an open problem which will have to be solved favorably (say with an $O(n^2)$ method) if Hu and Torres' algorithm is to be used successfully on problems other than those for which a linear decomposition is given a priori.

14.2 A New All Pairs Algorithm

We now present an all pairs algorithm which is of order less than $O(n^3)$ on all networks for which $\alpha < O(n^2)$. A particularly interesting property of this algorithm is that, for the first time, a good algorithm for unrestricted all pairs problems is directly related to the best algorithms known for the single source problem.

Following Nemhauser [Nem72] we show yet another way to implement a selection rule in Algorithm W (4.28) so that nodes chosen will not return to U. The rule relies on a function, with certain properties, on the nodes of N. The following lemma shows how the function is used, and one way in which it can be computed. No general method is known for computing a suitable function in less than the time needed to solve an unrestricted single source problem. Thus the methods we show in this section have no known utility for the single source problem.

(14.2) LEMMA. (Nemhauser [Nem72]) *There exists a function h for a given strongly connected network N such that the rule* choose $u \in U$ such that

$$d(u) + h(u) = \min_{v \in U} (d(v) + h(v))$$

<u>in Algorithm W (4.28) always chooses</u> u <u>such that</u> $d(u) = \mu(u)$ <u>whenever</u> $\mu(v)$ <u>is defined for all</u> v.

<u>Proof</u>: Define h on strongly connected networks to satisfy

$$h(u) \le h(v) + w(u,v).$$

Extend h to paths $q_{uv} = (u{=}v_{i_1}, v_{i_2}, \ldots, v_{i_k}{=}v)$ by summing the inequalities

$$\begin{array}{l} h(v_{i_1}) \le h(v_{i_2}) + w(v_{i_1}, v_{i_2}) \\ h(v_{i_2}) \le h(v_{i_3}) + w(v_{i_2}, v_{i_3}) \\ \qquad\vdots \\ h(v_{i_{k-1}}) \le h(v_{i_k}) + w(v_{i_{k-1}}, v_{i_k}) \\ \hline h(u) \le h(v) + w(q_{uv}) \end{array}$$

The function h exists only if $\mu(v)$ is defined for all v, for if there is a cycle of negative weight q_{vv}, then

$$h(v) \le h(v) + w(q_{vv}) < h(v),$$

which is impossible. When $\mu(v)$ is defined for all v, then let $h = g$, where $g(v)$ is the shortest path <u>to</u> some fixed node $\hat{u}$ from each $v \in V$. The function g has the desired property, $g(u) \le g(v) + w(u,v)$, for if it did not there would be u and v such that $g(u) > g(v) + w(u,v)$, implying a path from u to $\hat{u}$ shorter than $g(u)$.

Consider the rule of choice. By Lemma 4.6, $d(u) \geq \mu(u)$. Assume in contradiction that for u , chosen by the rule of choice, $d(u) > \mu(u)$. By Lemma 4.31 there must be a path

$$p_{vu} = (v=v_{i_1}, v_{i_2}, \ldots, v_{i_\ell}=u)$$

such that $v \in U$, $d(v) = \mu(v)$, and $d(v) + w(p_{vu}) = \mu(u) < d(u)$. But we know that

$$h(v) \leq h(u) + w(p_{vu}).$$

Adding $d(v)$ to each side of this inequality gives

$$\begin{aligned} d(v) + h(v) &\leq h(u) + d(v) + w(p_{vu}) \\ &< h(u) + d(u) \end{aligned}$$

in contradiction of the selection rule. □

The above result is equivalent to a result of Edmonds and Karp [E&K72] in which networks with negative arcs are transformed into networks equivalent with respect to shortest paths, but without arcs of negative weight.

Lemma 14.2 motivates the following algorithm:

(14.3) ALGORITHM AP: Shortest Paths for All Pairs

(1) [Construct Equivalent Strong Network and its Dual]

(1a) Find strong regions of N: $V_1, V_2, \ldots, V_j$;

(1b) Construct $\hat{N} = ((\hat{V}, \hat{X}), \hat{w})$ where

$\hat{V} = V$

$\hat{X} = X \cup ((v_{i_k}, v_{i_{\bar{k}}}) \mid v_{i_k} \in V_k,$
$v_{i_{\bar{k}}} \in V_{\bar{k}},$
$\bar{k} = k+1$ for $1 \le k < j$,
$\bar{k}=1$ for $k=j)$

$\hat{w}(\hat{X} \cap X) = w(X)$

$\hat{w}(\hat{X} - X) = \{n(\max_{x \in X} |w(x)|)\}$;

(1c) Construct $\bar{N} = ((\bar{V}, \bar{X}), \bar{w})$ where

$\bar{V} = V$

$\bar{X} = \{(u,v) \mid (v,u) \in \hat{X}\}$

$\bar{w}(u,v) = \hat{w}(v,u)$, for all $(v,u) \in \hat{X}$;

(2) [Find h]

(2a) Choose $s \in \bar{V}$;

(2b) Solve $\phi(\bar{N}, s)$;
<u>if</u> $\mu(v)$ undefined for any $v \in \bar{V}$ <u>then</u>
report "failure";

(2c) <u>for</u> all $\hat{v} \in \hat{V}$ <u>do</u> $h(\hat{v}) := d(\bar{v})$;

(3) [Solve All Pairs Problem on $\hat{N}$]

<u>for</u> each $v \in \hat{V}$ <u>do</u>
<u>begin</u>
Solve $\phi(\hat{N}, v)$ using Algorithm W (4.28) with
rule of choice "choose $u \in U$ such that
$d(u)+h(u) = \min_{v \in U}(d(v)+h(v))$;
[Report solution as $d(v,u)$ for each $u \in V$]
<u>for</u> each $u \in \hat{V}$ <u>do</u>
$d(v,u) := d(u)$
<u>end</u>

(4) [Transform Solution to N]
    <u>for</u> each $v \in V$ <u>do</u>
      <u>for</u> each $u \in V$ <u>do</u>
        $d(v,u) := $ <u>if</u> $d(v,u) \geq n(\max_{x \in X}|w(x)|)$.
                <u>then</u> ∞ <u>else</u> $d(v,u)$

(14.4) THEOREM. *If there is no cycle of negative weight, Algorithm AP finds shortest paths for all pairs; otherwise it reports "failure".*

Proof: First we prove that Algorithm AP solves $\phi(\hat{N})$. It is evident from the construction in Steps (1a) and (1b) that $\hat{N}$ is strongly connected. From Step (1c) it follows that $\bar{N}$ is such that there exists $\bar{q}_{uv} = (u=v_{i_1}, v_{i_2}, \ldots, v_{i_k}=v)$ in $\bar{N}$ if and only if there exists $\hat{q}_{vu} = (v=v_{i_k}, v_{i_{k-1}}, \ldots, v_{i_1}=u)$ in $\hat{N}$ and $\bar{w}(\bar{q}_{uv}) = \hat{w}(\hat{q}_{vu})$ for all $\bar{q}_{uv}$ in $\bar{N}$. Thus Step (2) produces a labelling function h satisfying Lemma 14.2 if and only if $\mu(s,v)$ is defined for all $v \in V$. Since $\hat{N}$ is strongly connected, $\hat{\mu}(s,v)$ is defined for all $v \in V$ if and only if $\hat{\mu}(v,u)$ is defined for all v and u . We conclude that if $\hat{\mu}(v,u)$ is not defined for some (v,u) , "failure" is always reported.

It follows from Lemma 14.2 that Step (3) solves $\phi(\hat{N})$, reporting μ in the matrix d , if and only if there is no cycle of negative weight in $\hat{N}$. By construction, $\hat{N}$ can contain a cycle of negative weight if and only if N does, since $n(\max_{x \in X}(|w(x)|))$ is greater than the absolute value of the sum of the weights of any $n-1$ arcs in N. Thus Step (3) solves $\phi(\hat{N})$ if and only if there are no cycles of negative weight in N.

For any pair of nodes (u,v), $(\mu(u,v) < \infty) \Rightarrow (\mu(u,v) < n(\max_{x \in X}(|w(x)|)))$. Thus the transformation of solutions in Step (4) must be correct since all arcs in N appear with the same weights in $\hat{N}$. □

(14.5) <u>THEOREM</u>. <u>Algorithm</u> <u>AP</u> <u>terminates</u> <u>in</u> $T(n,\alpha) \leq O(\min(n^{2+1/k}+n(\alpha+n),\ n(\alpha+n)(1+\log\frac{n^2}{(\alpha+n)})))$ <u>operations</u>.

<u>Proof</u>: There is an $O(n+\alpha)$ algorithm due to Tarjan [Tar72] for finding strongly connected regions. If $\hat{N}$ shares the data structure for N, Step (1b) can be $O(n)$, since $j \leq n$. Step (1c) can require $O(n+\alpha)$ operations since the list structure of $\hat{X}$ must be inverted for the necessary representation of $\bar{X}$. Thus Step (1) is $O(n+\alpha)$.

The order of Step (2) will be the order of whatever algorithm is chosen to solve $\phi(\bar{N},s)$. If we use Algorithm GP (11.12) under $X_p \cup X_m$, the bound will be

$$O(\min(n^{2+1/k}+n(\alpha+n),\ n^2\log n + n(\alpha+n)(1+\log\frac{n^2}{\alpha+n}))),$$

using appropriate bounds from Theorem 11.24 and the fact that $\bar{\alpha} \leq \alpha + n$. In showing a minimum of the two bounds in Theorem 11.24, we are assuming that a simple procedure exists for choosing which discipline to use, depending on α.

The bound for Step (3) is identical, being obtained by multiplying n times the bound for Algorithm W (see Corollary 9.11 and Theorem 9.28). Finally, Step (4) is $O(n^2)$. We conclude that the bounds for Steps (2) and (3) dominate. The bound is simplified by noting that

$$O(n^2 \log n) \leq O(n(\alpha+n)(1+\log(\frac{n^2}{\alpha+n}))). \quad \Box$$

We have shown a bound for Algorithm AP (14.3) which in the worst case is everywhere better, on incomplete networks for which α grows more slowly than n^2, than Hu and Torres' algorithm. We know of no other all pairs algorithm which has better than an $O(n^3)$ bound on unrestricted networks, regardless of α.

PART IV

15. References

[Bel58] Bellman, R. E., "On a Routing Problem," Quart. Appl. Math. 16, 1, 87-90 (1958).

[Ber58] Berge, C., Théorie des Graphs et Ses Applications, Dunod, Paris, 1958.

[Dan60] Dantzig, G. B., "On the Shortest Route Through a Network," Management Science 6, 2, 187-190 (1960).

[Dan67] Dantzig, G. B., "All Shortest Routes in a Graph," Theory of Graphs International Symposium, Rome, July, 1966, Gordon and Breach, New York, 1967, 91-92.

[DBR67a] Dantzig, G. B., Blattner, W. O., and Rao, M. R., "All Shortest Routes from a Fixed Origin in a Graph," Theory of Graphs International Symposium, Rome, July, 1966, Gordon and Breach, New York, 1967, 85-90.

[DBR67b] Dantzig, G. B., Blattner, W. O., and Rao, M. R., "Finding a Cycle in a Graph with Minimum Cost to Time Ratio with Application to a Ship Routing Problem," Theory of Graphs International Symposium, Rome, July, 1966, Gordon and Breach, New York, 1967, 77-83.

[Dij59] Dijkstra, E. W., "A Note on Two Problems in Connexion with Graphs," Numer. Math. 1, 269-271 (1959).

[Dry69] Dreyfus, S. E., "An Appraisal of Some Shortest-Path Algorithms," Operations Res. 17, 3, 395-412 (May-June 1969).

[E&K72] Edmonds, J., and Karp, R. M., "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems," J. ACM 19, 2, 248-264 (April 1972).

[FLM67] Farbey, B. A., Land, A. H., and Murchland, J. D., "The Cascade Algorithm for Finding All Shortest Distances in a Directed Graph," Management Science 14, 1, 19-28 (Sept. 1967).

[F&M72] Fischer, M. J., and Meyer, A. R., "Boolean Matrix Multiplication and Transitive Closure," Proc. Twelfth Annual Symposium on Switching and Automata Theory, October 13-15, 1971, IEEE Computer Society, 1972, 129-131.

[F&R71] Florian, M., and Robert, P., "A Direct Search Method to Locate Negative Cycles in a Graph," Management Science 17, 5, 307-310 (Jan. 1971).

[Flo62a] Floyd, R. W., "Algorithm 97 Shortest Path," Comm. ACM 5, 345 (June 1962).

[Flo62b] Floyd, R. W., "Algorithm 113 Treesort," Comm. ACM 5, 434 (Aug. 1962).

[Flo64] Floyd, R. W., "Algorithm 241 Treesort 3," Comm. ACM 7, 12, 701 (Dec. 1964).

[For56] Ford, L. R., Jr., Network Flow Theory, The Rand Corporation, P-923, August, 1956.

[F&F62] Ford, L. R., Jr., and Fulkerson, D. R., Flows in Networks, Princeton Univ. Press, 1962.

[Fox69] Fox, Bennett, "Finding Minimal Cost-Time Ratio Circuits," Operations Res. 17, 3, 546-550 (May-June 1969).

[Hor69] Hoare, C. A. R., "An Axiomatic Basis for Computer Programming," Comm. ACM 12, 576-583 (1969).

[Hu67] Hu, T. C., "Revised Matrix Algorithms for Shortest Paths," SIAM J. Appl. Math 15, 1, 207-218 (Jan. 1967).

[Hu68] Hu, T. C., "A Decomposition Algorithm for Shortest Paths in a Network," Operations Res. 16, 1, 91-102 (Jan.-Feb. 1968).

[H&T69] Hu, T. C., and Torres, W. T., "Shortcut in the Decomposition Algorithm for Shortest Paths in a Network," IBM J. Res. Devel. 13, 4, 387-390 (July 1969).

[I&N72] Iri, M., and Nakamori, M., Path-Sets, Operator Semigroups and Shortest-Path Algorithms on a Network, RAAG Research Notes, Third Series, No. 185, Faculty of Engineering, Univ. of Tokyo, Hongo, Bunkyo-ku, Tokyo, Japan (Sept. 1972).

[Joh72] Johnson, E. L., "On Shortest Paths and Sorting," Proceedings ACM 25th Annual Conference, August 1972, Boston, vol. I, 510-517 (Aug. 1972).

[Ker70] Kerr, L. R., The Effect of Algebraic Structure on the Computational Complexity of Matrix Multiplication, (Ph.D. Thesis) Cornell Univ., June 1970).

[Knu68] Knuth, D. E., The Art of Computer Programming, Vol. 1, Addison-Wesley, Reading, Mass., 1968.

[Law67] Lawler, E. L., "Optimal Cycles in Doubly Weighted Directed Linear Graphs," Theory of Graphs International Symposium, Rome, July, 1966, Gordon and Breach, New York, 1967, 20[illegible]3.

[Law72] Lawler, E. L., Optimal Cycles in Graphs and the Minimal Cost-to-Time Ratio Problem, Memorandum No. ERL-M343, Electronics Research Laboratory, College of Engineering, Univ. of California, Berkeley (May 10, 1972).

[Mor59] Moore, E. F., "The Shortest Path through a Maze," Proceedings International Symposium on the Theory of Switching, Part II, April 2-5, 1957, The Annals of the Computation Laboratory of Harvard Univ. 30, Harvard Univ. Press, 1959, 285-292.

[Mur69] Murchland, J. D., "The 'Once-Through' Method of Finding All Shortest Distances in a Graph from a Single Origin," Transport Network Theory Unit, London Graduate School of Business Studies, Report LBS-TNT-56.1 (Feb. 1969).

[Nem72] Nemhauser, G. L., "A Generalized Permanent Label Setting Algorithm for the Shortest Path between Specified Nodes," J. Math. Anal. Appl. 38, 2, 328-334 (May 1972).

[Poh69] Pohl, I., Bi-Directional and Heuristic Search in Path Problems, SLAC Report 104, Stanford Linear Accelerator Center, Stanford Univ., Stanford, California (May 1969).

[Str69] Strassen, V., "Gaussian Elimination is not Optimal," Numer. Math. 13, 354-356 (1969).

[Tar72] Tarjan, R. E., "Depth First Search and Linear Graph Algorithms," SIAM J. Comput. 1, 2, 146-160 (June 1972).

[War62] Warshall, S., "A Theorem on Boolean Matrices," J. ACM 9, 11-12 (Jan. 1962).

[Wil64] Williams, J. W. J., "Algorithm 232: Heapsort," Comm. ACM 7, 347-348 (June 1964).

[Yen70] Yen, J. Y., "A Shortest Path Algorithm," (Ph.D. Thesis) Univ. of California, Berkeley, 1970.

[Yen72] Yen, J. Y., "Finding the Lengths of All Shortest Paths in N-Node Nonnegative-Distance Complete Networks Using $\frac{1}{2}$ N^3 Additions and N^3 Compares," J. ACM 19, 3, 423-425 (July 1972).