COMPUTING SHORTEST PATHS IN NETWORKS DERIVED FROM RECURRENCE RELATIONS

by

E. L. Lawler

Memorandum No. UCB/ERL/IGCT M91/7

28 January 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

COMPUTING SHORTEST PATHS IN NETWORKS DERIVED FROM RECURRENCE RELATIONS

by

E. L. Lawler

Memorandum No. UCB/ERL/IGCT M91/7

28 January 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Computing Shortest Paths in Networks Derived from Recurrence Relations

E. L. Lawler

Computer Science Division
University of California
Berkeley, CA 94720

Dedicated to the memory of Paolo M. Camerini

ABSTRACT

Dynamic programming formulations of optimization problems often call for the computation of shortest paths in networks derived from recurrence relations. These derived networks tend to be very large, but they are also very regular and lend themselves to the computation of nontrivial lower bounds on path lengths. In this tutorial paper we describe unidirectional and bidirectional search procedures that make use of bounding information in computing shortest paths. When applied to many optimization problems, these shortest path algorithms capture the advantages of both dynamic programming and branch-and-bound.

1. Introduction

Sometimes networks are meant to model the real world. Examples are highway networks, communication networks, and interconnection networks for VLSI design. These "physical" networks are often quite irregular and complex. Other networks arise from purely mathematical structures, particularly recurrence relations. Such "derived" networks tend to be very large, but they are also very regular.

For example, many sequencing problems lend themselves to dynamic programming formulations that call for the computation of a value $u(N)$ defined by recurrence relations over subsets $X \subseteq N = \{1, 2, ..., n\}$. A generic form of these recurrence relations is

$$u(X) = \min \{u(X-j) + c(X-j, X) \mid j \in X\}, \quad X \neq \varnothing, \tag{1.1}$$
$$u(\varnothing) = 0.$$

(Here and in the sequel we let $X+j$ denote $X \cup \{j\}$ and $X-j$ denote $X-\{j\}$.) By appropriately specifying values for $c(X-j, X)$, equations (1.1) can be made to apply to such diverse problems as single-machine sequencing, linear arrangement, the weighted acyclic subgraph problem, and assembly line balancing, as we show in the next section.

It is easy to see that the problem of solving equations (1.1) for $u(N)$ is equivalent to the problem of finding the length of a shortest path from node $\varnothing$ to node N in a network in which each of the 2^n nodes is identified with a subset $X \subseteq N$ and in which the length of arc $(X-j, X)$ is $c(X-j, X)$. The network is layered and has the appearance of a hypercube whose edges are directed from one layer to the next. The equations (1.1) actually suggest an algorithm for finding a shortest path by computing the values $u(X)$ one layer at a time.

The most immediate, and most obvious, difficulty in trying to solve a shortest path problem such as that derived from equations (1.1) is that the network may be exceedingly large. Because the number of nodes is exponential in the size of the set N, it may be impossible to store an explicit representation of the entire network in computer memory. In fact, it may be excessively time consuming simply to generate a listing of all the nodes.

Nevertheless, derived networks have properties that lend themselves to computational exploitation. The regularity and symmetry of these networks means that local structure is readily ascertained. That is, given the designation of a node X, one can readily determine the arcs incident to X and also the lengths of those arcs. This means that it is unecessary to maintain an explicit representation of the network; those parts of the network that are needed in the search for a shortest path can be generated upon demand.

A second property of derived networks that can be exploited is that of lower bounding of path lengths. It is usually possible to compute a nontrivial lower bound on the length of a shortest path from $\varnothing$ to X, and from X to N. In much the same way that bounds are used to truncate branch-and-bound searches, these bounds can be used to eliminate parts of the network in the search for a shortest path from $\varnothing$ to N. The resulting "hybrid" computation captures some of the best features of both branch-and-bound and dynamic programming. Cf. [7].

We shall describe four types of procedures for computing shortest paths. First, we consider simple unidirectional search, which amounts to nothing more than the well known Dijkstra shortest path algorithm [3]. We then generalize unidirectional search to unidirectional search with bounds, bidirectional search, and bidirectional search with bounds. These procedures are slightly modified versions of those given in [6].

2. Some Specific Examples

For the sake of concreteness, we shall review some specific applications of the generic equations (1.1). In each case, we indicate how $c(X-j,X)$, the length of arc $(X-j,X)$, is defined. We also indicate a simple, but nontrivial, lower bound $g(X)$ on the length of a shortest path from node $\varnothing$ to node X, and a similar lower bound $h(X)$ on the length of a shortest path from X to N. These lower bounds are meant to be illustrative only; in many cases, stronger bounds can be obtained.

Single–Machine Sequencing:

Let N denote a set of n jobs, $j = 1,2,...,n$, each with a processing time $p_j \geq 0$. The jobs are to be sequenced for processing by a single machine, with the first job starting at

time zero and with no preemptions or machine idleness until the last job is completed. If job j is completed at time t, a cost $f_j(t)$ is incurred, where f_j, $j = 1, 2, ..., n$, is a nondecreasing, but otherwise arbitrarily specified, penalty function. The objective is to minimize the sum of these costs. For this problem,

$$c(X-j, X) = f_j(p(X)),$$

$$g(X) = \max\{\sum_{i \in X} f_i(p_i), \quad \min\{f_i(p(X)) \mid i \in X\}\},$$

$$h(X) = \max\{\sum_{i \in N-X} f_i(p(X)+p_i), \quad \min\{f_i(p(N)) \mid i \in N-X\}\},$$

where

$$p(X) = \sum_{i \in X} p_i.$$

Linear Arrangement:

Let $G = (N, E)$ be a given graph, with N its set of vertices and E its set of edges. The vertices are to be placed at unit intervals on a line so as to minimize the sum of the lengths of its edges. Let $m(X)$ denote the number of edges extending across the cut $(X, N-X)$. For this problem,

$$c(X-j, X) = m(X-j),$$

$$g(X) = \frac{1}{2}(\sum_{i \in X} d(i) + m(X)),$$

$$h(X) = \frac{1}{2}(\sum_{i \in N-X} d(i) + m(X)),$$

where $d(i)$ is the degree of vertex i.

Weighted Acyclic Subgraph Problem:

This problem is equivalent to the following: Given an $n \times n$ matrix $W = (w_{ij})$, find a permutation matrix P such that the sum of the entries below the diagonal in PWP^{-1} is minimized. Here

$$c(X-j,X) = \sum_{i \in N-X} w_{ij},$$

$$g(X) = \sum_{j \in X} \sum_{i \in N-X} w_{ij},$$

$$h(X) = \min\{ \sum_{i \in (N-X)-j} w_{ij} \mid j \in N-X \}.$$

Assembly Line Balancing:

Let N denote a set of tasks, each with a processing time $0 < p_j \leq 1$. These tasks are to be assigned to workstations on an assembly line, where the sum of the processing times of the tasks assigned to a given workstation may not exceed unity (the normalized "cycle time" of the line). Moreover, the assignment of tasks to workstations must respect certain given precedence constraints "-->", where i --> j means that j may not be assigned to a workstation that precedes i. As shown by Held, et al [5], the minimum number of workstations is given by $u(N)$, when arc lengths are defined as follows. If j has one or more predecessors in $X-j$ then $c(X-j,X) = +\infty$, else

$$c(X-j,X) = \begin{cases} p_j, & \text{if } u(X-j)+p_j \leq \lceil u(X-j) \rceil, \\ \lceil u(X-j) \rceil - u(X-j) + p_j, & \text{otherwise.} \end{cases}$$

$$g(X) = \sum_{j \in X} p_j,$$

$$h(X) = \sum_{j \in N-X} p_j.$$

Observe that the value of $c(X-j,X)$ depends on $u(X-j)$. Hence one cannot determine the length of arc $(X-j,X)$ until the length of a shortest path from $\varnothing$ to $X-j$ has been found. This presents no difficulty for the shortest path computations we shall describe.

3. A Generic Search Procedure

For notational convenience, we shall hereafter denote the nodes of the network by integers, with s and t denoting the end points of the shortest path that is to be found. The length of arc (i,j) will be denoted $c(i,j)$.

Each search procedure involves the application of numerical labels to nodes, where the value of a label $u(j)$ (or $v(j)$) on node j represents the length of the shortest path that has been discovered from s to j (or j to t). Labels are either "tentative" or "permanent". Following initialization, each search procedure contains a **while** loop of the form:

while $LB < UB$

choose a tentatively-labeled node for scanning and make its label permanent;

"scan" the chosen node by updating existing tentative labels, and applying new tentative labels, on the nodes that are adjacent to it;

revise LB and UB;

endwhile

In the statement of this generic **while** loop, UB is an upper bound on the length of a shortest path from s to t; invariably this is the length of the shortest path that has actually been discovered. LB is a lower bound on the length of any path that has length less than UB. When the computation terminates, UB equals the length of a shortest path.

For each of the four search procedures we shall describe how a tentatively-labeled node is chosen for scanning, how scanning is performed, and how the bounds LB and UB are revised. We shall also establish the validity of each procedure by showing that certain assertions are valid at each iteration of the **while** loop.

3. Unidirectional Search

As we remarked, the unidirectional search procedure we shall present is a straightforward adaptation of Dijkstra's well known shortest path algorithm [3]. We let S denote the set of permanently-labeled nodes and $\bar{S}$ the set of tentatively-labeled nodes. Since the

reader is presumably familiar with this algorithm, we simply summarize its particulars for comparison with the other search procedures, numbering certain lines for later reference.

***Assumption*:**

$$c(i,j) \geq 0, \qquad \text{for all arcs } (i,j).$$

***Initial Conditions*:**

$$\begin{aligned} &u(s) := 0; \\ &\tilde{S} := \{s\}; \\ &S := \varnothing; \\ &LB := 0; \\ &UB := +\infty; \end{aligned} \tag{3.1}$$

***Choice Rule* for *Scanning*:**

$$\begin{aligned} &k := argmin\ \{u(j) \mid j \in \tilde{S}\} \\ &S := S + k \\ &\tilde{S} := \tilde{S} - k \end{aligned} \tag{3.2}$$

(Here $argmin\ \{u(j) \mid j \in \tilde{S}\}$ denotes any node $k \in \tilde{S}$ such that $u(k) = \min\{u(j) \mid j \in \tilde{S}\}$.)

***Scanning*:**

for each arc (k,j) **do**
  if $j \in \tilde{S}$ **then**
    $u(j) := \min\{u(j), u(k) + c(k,j)\}$; **fi**
  if $j \notin S \cup \tilde{S}$ **then**
    $u(j) := u(k) + c(k,j)$;
    $\tilde{S} := \tilde{S} \cup \{k\}$; **fi od**

Lower Bound LB:

$$LB := \min\{u(j) \mid j \in \bar{S}\}; \tag{3.3}$$

Upper Bound UB:

if $t \in \bar{S}$ **then** $UB := u(t)$ **else** $UB := +\infty$;

Loop Invariants:

(3.4) If $j \in S$ then $u(j)$ is the length of a shortest path from s to j.

(3.5) If $j \in \bar{S}$ then $u(j)$ is the length of a shortest path from s to j, subject to the constraint that all nodes in the path (except j) are in S.

(3.6) If UB is finite, there is a path from s to t with length UB.

(3.7) There is no path from s to t with length less than LB.

To prove the validity of the loop invariants we argue as follows. Suppose (3.4) and (3.5) are true at the beginning of an iteration. Let k be the node chosen for scanning. We assert that $u(k)$ is the length of a shortest path from s to k. Proof of this assertion is by contradiction. Assume there exists a shorter path P, and $k' \in \bar{S}$ is the first node in P not in S. It follows from (3.5) and the choice of k that the length of the prefix of P from s to k' is $u(k') \geq u(k)$. Because all arc lengths are nonnegative, the suffix of the path from k' to k is nonnegative. It follows that the length of P is at least $u(k') \geq u(k)$, contrary to assumption. Hence $u(k)$ must be the length of a shortest path to k and placing k in S leaves (3.4) valid. The scanning of k makes (3.5) valid at the end of the iteration. The validity of (3.6) is obvious. The validity of (3.7) follows from the fact that any path from s to t must contain some node $k' \in \bar{S}$ and the length of such a path must be at least $u(k') \geq LB$.

4. Unidirectional Search with Bounds

Hart, et al [4] appear to have been among the first to notice that a lower bound $h(j)$ on the length of a shortest path from j to t can be used to improve the efficiency of unidirectional search. In the artificial intelligence community, $h(j)$ is often referred to as a "heuristic function" and the mode of search as "heuristically guided search" or the "A^*" algorithm.

We shall assume that lower bounds on path lengths satisfy certain *consistency conditions:*

$$h(i) \leq c(i,j) + h(j), \quad \text{for all arcs } (i,j). \tag{4.1}$$

Informally, what these conditions say is that the length of a shortest path from i to t should be no greater than the length of a shortest path from i to t, subject to the constraint that the first arc in the path is from i to j. It is ordinarily the case that bounds derived for combinatorial optimization problems satisfy conditions (4.1). We leave it as an exercise for the reader to verify that this is indeed true for each of the bounds given in Section 2.

Bounds satisfying conditions (4.1) enable us to define a new set of nonnegative arc lengths $c_h(i,j)$:

$$c_h(i,j) = c(i,j) - h(i) + h(j) \geq 0.$$

Notice that these arc lengths are nonnegative, whether or not this is so for the original arc lengths $c(i,j)$. Notice also that if P is any path from s to t, then the length of P with respect to arc lengths $c_h(i,j)$ differs by a constant from the length of the same path with respect to arc lengths $c(i,j)$:

$$\sum_{(i,j)\in P} c_h(i,j) = \sum_{(i,j)\in P} c(i,j) - h(s) + h(t).$$

Accordingly, the computation of a shortest path with respect to lengths $c_h(i,j)$ yields a shortest path with respect to lengths $c(i,j)$.

It is a straightforward matter to modify the **while** loop of the unidirectional search procedure to use arc lengths $c_h(i,j)$. All that is required is to modify statements (3.2) and (3.3) to read

$$k := argmin\ \{u(j) - h(s) + h(j) \mid j \in \tilde{S}\};$$
$$LB := \min\{u(j) - h(s) + h(j) \mid j \in \tilde{S}\};$$

However, (assuming $h(t) = 0$) adding the constant $h(s)$ to all path lengths will cause all path lengths to be expressed in terms of the original arc lengths $c(i,j)$. Accordingly, we choose to modify statements (3.1)-(3.3) to read as follows:

$$LB := h(s); \tag{3.1'}$$
$$k := argmin\ \{u(j) + h(j) \mid j \in \tilde{S}\}; \tag{3.2'}$$
$$LB := \min\{u(j) + h(j) \mid j \in \tilde{S}\}; \tag{3.3'}$$

We have now described all that is necessary to modify the unidirectional search procedure so that it makes use of consistent bounds on path lengths.

Does the use of bounds improve the efficiency of the search procedure? Let $u(j)$ denote the true length of a shortest path to j. Without bounds, node j is scanned if and only if $u(j) < u(t)$. With bounds, node j is scanned if and only if $u(j) + h(j) < u(t)$. It follows that the number of nodes scanned cannot be increased by the use of bounds, provided they are nonnegative. And even rather weak bounds may significantly reduce the number of nodes that are scanned.

Of course, stronger bounds are better than weak bounds. (In the extreme case, when $h(j)$ denotes the true length of a shortest path from j to t, the only nodes scanned are those which actually lie on a shortest path from s to t.) Fortunately, it is not necessary to make a choice between two or more alternative bounds: If $h^{(1)}, h^{(2)}, ..., h^{(m)}$ are all bounds satisfying the consistency conditions (4.1), then

$$h(j) = \max\{h^{(1)}(j), ..., h^{(m)}(j)\}$$

also satisfies the consistency conditions.

5. Bidirectional Search

We have described unidirectional search in terms of a labeling of nodes proceeding "forward" from node s. By symmetry, we could equally well have labeled "backward" from node t. It seems only natural to consider a bidirectional type of search in which labeling proceeds simultaneously in both directions. Since the middle of a derived network is often its "thickest" part, there is some intuitive appeal to the idea of approaching the difficult middle from both sides.

Various authors, e.g., Pohl [8], have proposed bidirectional variants of Dijkstra's algorithm. These variants turn out to be a bit tricky, because it is easy to mistate termination conditions. (And at least one eminent researcher once did so.) The procedure we present here differs in some minor but significant respects from these previous proposals.

First let us set some notation. As before, we assume that all arc lengths $c(i,j)$ are nonnegative. And as before, $u(j)$ denotes a label applied to node j by forward labeling from node s, S denotes set of nodes with permanent u-labels, and $\tilde{S}$ denotes the set of nodes with tentative u-labels. Symmetrically, $v(j)$ will denote a label applied by backward labeling from node t, T will denote the set of nodes with permanent v-labels, and $\tilde{T}$ will denote the set of nodes with tentative v-labels.

We begin by applying the tentative labels $u(s) = 0$ and $v(t) = 0$. At each iteration of the **while** loop we choose either to scan forward from a node k, where

$$k := \textit{argmin}\ \{u(j) \mid j \in \tilde{S}\},$$

or backward from a node l, where

$$l := \textit{argmin}\ \{v(j) \mid j \in \tilde{T}\}.$$

The scanning of k is exactly the same as in unidirectional search, and the scanning of l is its symmetric equivalent.

Eventually some node will receive both a u-label and a v-label. When this happens, we have discovered a path from s to t. Accordingly, we let

$$UB := \min\{u(j) + v(j) \mid j \in (S \cup \tilde{S}) \cap (T \cup \tilde{T})\}.$$

Now all that remains to be explained is how LB is determined. We wish LB to be a lower bound on the length of any path that is shorter than one that has been discovered. Assume sets S and T are disjoint. Suppose P is a path from s to t with length less than UB. We assert that the sequence of nodes through which P passes decomposes into a prefix containing only nodes in S, a middle part from a node $i \in \tilde{S}-T$ to a node $j \in \tilde{T}-S$, and a suffix containing only nodes in T. A lower bound on the length of P from s to i is

$$\min\{u(i) \mid i \in \tilde{S}-T\},$$

a lower bound on the length of P from j to t is

$$\min\{v(j) \mid j \in \tilde{T}-S\},$$

and the length of P from i to j is nonnegative. Hence a lower bound on the length of any path shorter than UB is

$$LB := \min\{u(i) \mid i \in \tilde{S}-T\} + \min\{v(j) \mid j \in \tilde{T}-S\}.$$

We summarize the bidirectional search procedure as follows:

Assumption:

$c(i,j) \geq 0$, for all arcs (i,j).

Initial Conditions:

$u(s) := 0;$
$\tilde{S} := \{s\};$
$S := \varnothing;$

$v(t) := 0;$
$\tilde{T} := \{t\};$
$T := \varnothing;$

$LB := 0;$
$UB := +\infty;$

Choice Rule for Scanning:

If scanning forward **then**

$k := argmin\ \{u(j) \mid j \in \tilde{S}\}$;
$\tilde{S} := \tilde{S} - k$;
$S := S + k$;
if scanning backward **then**

$l := argmin\ \{u(j) \mid j \in \tilde{T}\}$;
$\tilde{T} := \tilde{T} - k$;
$T := T + k$;

Scanning:

if scanning forward **then**
  for each arc (k, j) **do**
    if $j \in \tilde{S}$ **then**
      $u(j) := \min\{u(j), u(k) + c(k, j)\}$; **fi**
    if $j \notin S \cup \tilde{S}$ **then**
      $u(j) := u(k) + c(k, j)$;
      $\tilde{S} := \tilde{S} \cup \{k\}$; **fi od**
fi

if scanning backward **then**
  for each arc (j, l) **do**
    if $j \in \tilde{T}$ **then**
      $v(j) := \min\{u(j), u(l) + c(j, l)\}$;
    if $l \notin T \cup \tilde{T}$ **then**
      $v(j) := u(l) + c(j, l)$;
      $\tilde{T} := \tilde{T} \cup \{k\}$; **fi od**
fi

Lower Bound LB:

$$LB := \min\{u(i) \mid i \in \tilde{S}-T\} + \min\{v(j) \mid j \in \tilde{T}-S\}$$

***Upper Bound UB*:**

$$UB := \min\{u(j)+v(j) \mid j \in (S \cup \tilde{S}) \cap (T \cup \tilde{T})\};$$

***Loop Invariants*:**

(5.1) If $j \in S$ then $u(j)$ is the length of a shortest path from s to j. If $j \in T$ then $v(j)$ is the length of a shortest path from j to t.

(5.2) If $j \in \tilde{S}$ then $u(j)$ is the length of a shortest path from s to j, subject to the constraint that all nodes in the path (except j) are in S. If $j \in \tilde{T}$ then $v(j)$ is the length of a shortest path from j to t, subject to the constraint that all nodes in the path (except j) are in T.

(5.3) If UB is finite, there is a path from s to t with length UB.

(5.4) If there is a path from s to t with length shorter than UB, then such a path has length at least LB.

The validity of the loop invariants (5.1) and (5.2) follows from exactly the same argument we made in the case of unidirectional search. The validity of (5.3) is obvious, and (5.4) follows from the reasoning given in the text.

Interestingly, the algorithm never scans the same node in both a forward and a backward direction. That is, the algorithm terminates before any node receives both a permanent u-label and a permanent v-label. For example, suppose $l \in S \cap \tilde{T}$ and $v(l) = \min\{v(j) \mid j \in \tilde{T}\}$. Then $u(l)+v(l) \geq UB$. But also $l \in S$ implies $u(l) \leq \min\{u(j) \mid j \in \tilde{S}\}$, hence $u(l)+v(l) \leq LB$, implying that the procedure has already terminated. It follows that the algorithm is unaffected if we modify the choice rule by having

$$k := argmin\ \{u(j) \mid j \in \tilde{S}-T\},$$
$$l := argmin\ \{v(j) \mid j \in \tilde{T}-S\}.$$

Intuition suggests that it is indeed wasted effort to scan any node in both the forward and the backward direction. If, as in the preceding example, $l \in S \cap \tilde{T}$ and

$v(l) = \min\{v(j) \mid j \in \tilde{T}\}$, then a shortest path from s to l has been found, and also a shortest path from l to t. If l does lie on a shortest path from s to t, we have constructed such a path and there is nothing more to be learned by scanning backward from l.

Finally, note that the bidirectional procedure is nondeterministic, in that at each iteration one is free to choose a node k for forward scanning or a node l for backward scanning. A reasonable strategy is to scan alternately forward and backward. This strategy ensures that, in the worst case, no more than twice as many nodes are scanned as would have been scanned with a unidirectional search. Very possibly, of course, the number of scanned nodes will be a good deal less than with unidirectional search.

6. Bidirectional Search with Bounds

First let us suppose only one type of bound on path lengths is available, namely, a bound $h(j)$ on the length of a shortest path from j to t. In order to find out how the bidirectional search procedure should be revised to take advantage of this bound, we adopt the same approach as in Section 4. That is, we begin by considering how we would carry out bidirectional search without bounds, but with respect to arc lengths

$$c_h(i,j) = c(i,j) + h(j) - h(i).$$

The change in arc lengths implies that we should modify the choice rule to make

$$k := \textit{argmin}\{u(i) + h(i) - h(s) \mid i \in \tilde{S}\},$$
$$l := \textit{argmin}\{v(j) - h(j) + h(t) \mid j \in \tilde{T}\}.$$

But as we observed in in Section 4, (assuming $h(t) = 0$) adding the constant $h(s)$ to all path lengths causes all path lengths to be expressed in terms of the original arc lengths $c(i,j)$. Hence we modify the choice rule to read

$$k := \textit{argmin}\{u(i) + h(i) \mid i \in \tilde{S}\}; \tag{6.1}$$
$$l := \textit{argmin}\{v(j) - h(j) \mid j \in \tilde{T}\}; \tag{6.2}$$

Since we are again measuring path lengths in terms arc lengths $c(i,j)$, the computation of UB remains as before. In order to compute LB, we reason as follows. As we observed in the previous section, any path P from s to t with length less than UB has a

prefix from s to some node $i \in \bar{S} - T$, with length $u(i)$, a suffix from some node $j \in \bar{T} - S$ to t, with length $v(j)$, and a middle part from i to j. It follows, by repeated application of (4.1), that $h(i) - h(j)$ is a lower bound on the length of P from i to j. Let LB_h denote the resulting lower bound:

$$LB_h := \min\{u(i) + h(i) \mid i \in \bar{S} - T\} + \min\{v(j) - h(j) \mid j \in \bar{T} - S\}; \qquad (6.3)$$

If we do use only a single type of bound $h(j)$ on path lengths, then no other modifications to the bidirectional search procedure are necessary, other than those we have indicated to the choice rule and to the computation of LB. However, as we indicated in Section 3, it is often feasible to compute two types of bounds on path length: in addition to $h(j)$, a symmetric lower bound $g(j)$ on the length of a shortest path from s to j. We may assume that $g(j)$ satisfies consistency conditions symmetric to (4.1), i.e.,

$$g(j) \le c(i,j) + g(i), \quad \text{for all arcs } (i,j). \qquad (6.4)$$

How should we make use of these alternate lower bounds?

The lower bounds $g(j)$ imply the following modifications in the choice rule and in the computation of LB:

$$k := \textit{argmin}\{u(i) - g(i) \mid i \in \bar{S}\}; \qquad (6.1')$$

$$l := \textit{argmin}\{v(j) + g(j) \mid j \in \bar{T}\}; \qquad (6.2')$$

$$LB_g := \min\{u(i) - g(i) \mid i \in \bar{S} - T\} + \min\{v(j) + g(j) \mid j \in \bar{T} - S\}; \qquad (6.3')$$

It is certainly legitimate to use as a lower bound the larger of LB_g and LB_h, i.e.,

$$LB := \max\{LB_g, LB_h\};$$

And, intuitively, it seems desirable to use a choice rule that will tend to increase LB at the next iteration of the **while** loop. That is, if $LB = LB_h$ use (6.1) or (6.2), and if $LB = LB_g$, use (6.1') or (6.2').

One consequence of this proposal is that we can no longer guarantee that a given node will never be scanned in both a forward direction and a backward direction. Suppose $LB = LB_g$, $l \in S \cap \bar{T}$ and $v(l) = \min\{v(j) + g(j) \mid j \in \bar{T}\}$. Then $u(l)$ is the length

of a shortest path from s to l, $v(l)$ is the length of a shortest path from l to t, and $u(l)+v(l) \geq UB$. However, $l \in S$ does not imply $u(l)-g(l) \leq \min\{u(i)-g(i) \mid i \in \tilde{S}\}$, because it may be the case that l was chosen for scanning in the forward direction when $LB = LB_h$. Hence it does not follow that $UB \leq u(l)+v(l) \leq LB$, as in search without bounds.

Nevertheless, it is wasted effort to scan any node in more than one direction. The way to prevent this from happening is to modify (6.1) and (6.1') so that minimization is carried out only over $i \in \tilde{S}-T$ and (6.2) and (6.2') so that minimization is only over $j \in \tilde{T}-S$. This modification has the effect of causing the choice rule to ignore any node that cannot lie on an (s,t) path shorter than UB. But this also means that the loop invariants must be modified. For example, we now have: If $i \in S$ *and* i *lies on a path from* s *to* t *shorter than* UB then $u(i)$ is the length of a shortest path from s to i.

We now summarize the changes that are required in the bidirectional search procedure, in order to accommodate bounds $g(j)$ and $h(j)$:

Assumption:

$h(j)$ and $g(j)$ each satisfy the consistency conditions,

$$h(i) \leq c(i,j)+h(j),$$

$$g(i) \leq c(i,j)+g(j), \quad \text{for all arcs } (i,j).$$

It is *not* necessary that arc lengths be nonnegative.

Initial Conditions:

The same as bidirectional search without bounds, except:

$LB_g := g(t)$;

$LB_h := h(s)$;

$LB := \max\{LB_g, LB_h\}$;

Choice Rule for Scanning:

if scanning forward and $LB = LB_g$ then

$k := argmin\ \{u(i) - g(i) \mid i \in \tilde{S} - T\}$; **fi**

if scanning forward and $LB = LB_h$ **then**

$k := argmin\ \{u(i) + h(i) \mid i \in \tilde{S} - T\}$; **fi**

if scanning backward and $LB = LB_g$ **then**

$l := argmin\ \{v(j) + g(j) \mid j \in \tilde{T} - S\}$; **fi**

if scanning backward and $LB = LB_h$ **then**

$l := argmin\ \{v(j) - h(j) \mid j \in \tilde{T} - S\}$; **fi**

Lower Bound LB:

$$LB_g := \min\{u(i) - g(i) \mid i \in \tilde{S} - T\} + \min\{v(j) + g(j) \mid j \in \tilde{T} - S\};$$
$$LB_h := \min\{u(i) + h(i) \mid i \in \tilde{S} - T\} + \min\{v(j) - h(j) \mid j \in \tilde{T} - S\};$$
$$LB := \max\{LB_g, LB_h\};$$

Loop Invariants:

(6.8) If $j \in S$ and j is contained in a path from s to t shorter than UB then $u(j)$ is the length of a shortest path from s to j. If $j \in T$ and j is contained in a path from s to t shorter than UB then $v(j)$ is the length of a shortest path from j to t.

(6.9) If $j \in \tilde{S}$ and j is contained in a path from s to t shorter than UB then $u(j)$ is the length of a shortest path from s to j, subject to the constraint that all nodes in the path (except j) are in S. If $j \in \tilde{T}$ and j is contained in a path from s to t shorter than UB then $v(j)$ is the length of a shortest path from j to t, subject to the constraint that all nodes in the path (except j) are in T.

(6.10) If UB is finite, there is a path from s to t with length UB.

(6.11) If there is a path from s to t with length shorter than UB, then such a path has length at least LB.

Acknowledgement

The author gratefully acknowledges the contributions of Michael Luby and Bruce Parker who coauthored [6], on which this tutorial is based. Research reported in this paper was supported by NSF grants CCR-8704184 and IRI-9045635.

References

[1] D. de Champeaux and L. Sint, "An Improved Bidirectional Heuristic Search Algorithm, *J. ACM, 24,* (1977), 177-191.

[2] D. de Champeaux, "Bidrectional Search Again," *J. ACM, 30,* (1983), 22-32.

[3] E. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik, 1,* (1959), 269-271.

[4] P. Hart, N. Nilsson and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Trans. Sys. Sci. Cyber., 4,* (1968), 100-107.

[5] M. Held, R. M. Karp and R. Shareshian, "Assembly-Line Balancing -- Dynamic Programming with Precedence Constraints," *Operations Research, 11,* (1963), 442-452.

[6] E. L. Lawler, M. G. Luby and B. Parker, "Finding Shortest Paths in Very Large Networks," *Proc. WG'83,* Workshop on Graph Theoretic Concepts in Computer Science, M. Nagle and J. Perl, eds., Trauner Verlag, (1983), 184-199.

[7] T. L. Morin and R. E. Marsten, "Branch-and-Bound Strategies for Dynamic Programming," *Operations Research, 24,* (1976), 611-627.

[8] I. Pohl, "Bi-Directional Search," in *Machine Intelligence 6,* B. Meltzer and D. Michie, eds., 1971, Edinburgh University Press, 127-140.