SHORTEST PATH ALGORITHMS: A COMPARISON

by

Bruce L. Golden

OR 044-75 October 1975

ABSTRACT

In this note we present some computational evidence to suggest that a version of Bellman's shortest path algorithm outperforms Treesort-Dijkstra's for a certain class of networks.

DISCUSSION

Many applications dealing with transportation and communication networks require the calculation of shortest paths. We discuss here the specific problem of finding the shortest paths from one node to all others. Our objective is to present some computational experience to support the claim that a version of Bellman's algorithm outperforms Treesort-Dijkstra's for a certain class of networks. This note complements a recent paper by Pape [8] and indicates that, for the class of networks under consideration, Pape's reluctance to use a variable length list of nodes to be scanned is unwarranted (the phrase "variable length list" refers to the fact that no realistic a priori bound on the length of the list is known).

A sequence of distinct arcs $(a_1, a_2, \ldots, a_p)$, where a_t and a_{t+1} are adjacent for $t=1, \ldots, p-1$ is called a path; a route is defined as a sequence of adjacent arcs which need not be distinct. We seek the shortest paths from the origin to all other nodes. Dreyfus discusses several such algorithms in his survey paper [3] primarily from the viewpoint of computational complexity.

Dijkstra's algorithm requires on the order of NN^2 additions and NN^2 comparisons in the worst case. This algorithm is a "label-setting" method which assigns permanent labels as it proceeds. Initially the set T consists of the origin alone. T is augmented one node at a time so that at each step, T is a set of permanently labeled nodes which corresponds to the shortest path tree for all nodes in T. Termination occurs when all nodes of the graph are in T. Labeling methods for computing shortest paths can be divided into two general classes, "label-setting" and "label-correcting" methods [5]. Label-setting methods are valid only for non-negative arc lengths.

Bellman's algorithm solves the problem in at most NN^3 additions and comparisons or detects the existence of a negative cycle. This algorithm is an example of the label-correcting approach in which no node labels are considered permanent until they all are, at termination.

If negative cycles exist then clearly there can be no shortest route on a network. The shortest path problem in that case has been shown to be combinatorially equivalent to the Traveling Salesman Problem [4].

The performance of shortest path algorithms is heavily dependent upon the following three factors:

(i) the sparseness of the network;

(ii) list processing and network representation in the computer code;

(iii) distance measures on the arcs.

The topological structure of the network clearly exerts a major influence on running time for any graph algorithm. Theoretical upper bounds have been calculated assuming a complete graph with every pair of nodes connected by an arc. If the graph is sparse, running times may be reduced significantly [5] (the same observation has been exploited with the minimal spanning tree problem [7].

Computer representation of networks is discussed in [5]. Distance measures will be mentioned later.

IMPLEMENTATION

For sparse graphs, Bellman's algorithm can be made quite efficient using a list structure which keeps track of which nodes can potentially label other nodes. This list is of nodes to be scanned.

The origin is the first element on the list. Those nodes which can be reached directly from the origin are labeled and placed on the list. We proceed downward from the top of the list and scan each member of the list possibly adding new members, if a new label is less than a current label. When we have scanned the entire list we have the shortest path tree. At the same time we trace the shortest path tree through the predecessor labels for each node. A flag associated with the active (unscanned) members of the list prevents us from placing one node on the active list more than once at any time.

In the Dijkstra algorithm a primary computational concern involves the determination of the minimum distance node at each step. We have implemented a modified Dijkstra algorithm where Floyd's Treesort algorithm is used for the sorting of these distances. This approach has been studied by Johnson [6] and by Kershenbaum and Van Slyke [7]. The node distances $d_1, d_2, \ldots, d_m$ where $m=2^k-1$ are arranged in a binary tree with k levels, called a heap. The essential property of a heap is that $d_i \leq d_{2i}$ and $d_i \leq d_{2i+1}$. Below is a heap for $k=3 (d_1=3, d_2=5, d_3=4$, and so on).

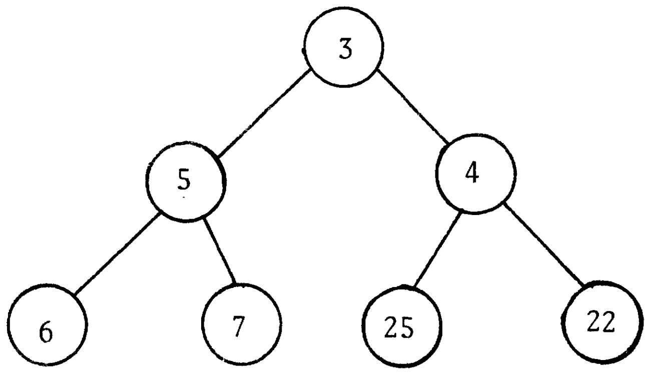

Figure 1

If the list is only of length $\ell \neq 2^k-1$ then we can fill positions $\ell+1, \ldots, 2^k-1$ for the smallest k such that $\ell < 2^k-1$ with distances of ∞. Clearly d_1 is the minimum node distance under consideration. If we remove d_1 from the heap, a new heap can be constructed with relative ease. The modified Dijkstra algorithm has node distances d_i composing the heap for all nodes i which are not yet in T. After a new node has been added to T, and has been scanned, we remove the top node of the heap and form a new heap.

COMPUTATIONAL EXPERIENCE

Bellman's and Dijkstra's algorithms have been coded and tested on M.I.T.'s IBM 370/168 system assuming non-negative arc distances. Two groups of networks are studied: Series A and Series B. Node coordinates are generated from a uniform probability distribution over a rectangular grid and then the euclidean distances are calculated between "randomly selected" pairs. These pairs are chosen in such a way that the out-degree of every node is equal to R for Series A and the out-degree of each node takes on the value 2,3,4,5, or 6 with equal probability for Series B. A computational consideration in Bellman's algorithm is how long the list of nodes to be scanned grows. With euclidean distances, one would expect for sparse networks that most nodes are not put on the list to

TABLE I. Computational Experience: Bellman vs. Dijkstra (Series A) (average running times given in seconds).

NN	R	DIJKSTRA TIME	BELLMAN TIME	AVERAGE LIST LENGTH	MAXIMUM LIST LENGTH
50	3	.021	.014	60.0	72
	4	.024	.013	55.6	66
	5	.027	.017	58.5	74
	6	.029	.021	59.7	70
100	3	.047	.026	117.6	131
	4	.055	.032	128.2	139
	5	.062	.037	124.6	141
	6	.065	.044	125.7	149
150	3	.084	.040	176.1	185
	4	.087	.053	197.6	235
	5	.102	.060	201.3	232
	6	.103	.066	195.1	220
250	3	.148	.072	323.3	389
	4	.164	.090	338.9	386
	5	.178	.103	327.1	363
	6	.190	.116	333.0	377
350	3	.230	.108	466.6	518
	4	.251	.128	473.8	523
	5	.264	.152	501.3	606
	6	.277	.171	509.2	595
500	3	.327	.155	703.0	819
	4	.368	.177	658.5	766
	5	.406	.214	699.7	802
	6	.422	.249	722.5	782
750	3	.538	.229	1012.6	1136
	4	.573	.286	1090.7	1262
	5	.637	.334	1068.2	1140
	6	.690	.402	1110.5	1241
1000	3	.733	.309	1389.5	1589
	4	.809	.383	1447.6	1661
	5	.848	.433	1462.0	1629
	6	.894	.518	1535.8	1676

be scanned more than once (precisely because of the triangle inequality).

For Series A, we generated ten networks of NN nodes where each node had fixed out-degree R for NN=50,100,150,250,350,750,1000 and R=3,4,5,6, and applied Bellman's and Dijkstra's algorithms to determine shortest paths. For Series B, we generated and tested ten networks of NN nodes where the out-degree for each node varied from 2 to 6. The mean running times for Series A and Series B are shown in Table I and Table II respectively. In addition, the average length of the list of nodes to be scanned and the maximum length are displayed.

For a given R (Series A) our results indicate that the relationship between NN and running time is nearly linear for both algorithms. Series B experiments indicate likewise. Bellman's algorithm clearly outperforms Dijkstra's algorithm; running times from Dijkstra's algorithm are about twice the running times from Bellman's. Interestingly, as the number of nodes increases, Bellman's algorithm becomes more and more attractive relative to Dijkstra's.

Our computational experience suggests that the variable length list does not become a great deal longer than NN. In fact, for $NN \leq 1000$, we can be confident that the list length will not exceed 2•NN for the class of networks discussed in this paper. Key properties in this class include network sparsity and euclidean distances.

ACKNOWLEDGMENTS

The author would like to thank Aaron Kershenbaum of Network Analysis Corporation and Professor Gabriel Handler of M.I.T. for the opportunity to perform this computational comparison. Helpful comments by Professor Thomas Magnanti of M.I.T. are appreciated. In addition, the author is grateful to NAC for the use of their modified Dijkstra code.

TABLE II. Computational Experience: Bellman vs. Dijkstra (Series B) (average running times given in seconds).

NN	DIJKSTRA TIME	BELLMAN TIME	AVERAGE LIST LENGTH	MAXIMUM LIST LENGTH
50	.028	.015	58.7	64
100	.055	.030	117.2	132
150	.091	.052	193.3	211
250	.165	.088	323.6	358
350	.238	.133	497.5	574
500	.365	.184	695.9	886
750	.574	.275	1050.9	1305
1000	.808	.382	1422.7	1501