Available online at www.sciencedirect.com
ScienceDirect
Computers & Operations Research 34 (2007) 2824–2839
computers & operations research
www.elsevier.com/locate/cor

Two fast algorithms for all-pairs shortest paths

C.W. Duin*

AKE, Faculteit Econometrie en Economie, Universiteit van Amsterdam, Roetersstraat 11, 1018 WB, Amsterdam

Available online 20 December 2005

Abstract

In a large, dense network, the computation of the 'distances', i.e., the shortest path lengths between all pairs of nodes, can take a long time with algorithms known from the literature.

We present two all-pairs shortest path algorithms, based on the equations of Bellman. These algorithms run fast, much faster than indicated by their time complexity bound of $O(n^2m)$, where n is the number of nodes and m the number of arcs.

As in Bellman's method 'candidate' distances are maintained and updated, obtaining the distances eventually. However, the order of updating the candidate distances is changed, in such a way that arcs are eliminated as soon as it is clear that they are not relevant for the update of candidate distances later in the algorithm. In dense graphs this can significantly reduce the computation time.

By scanning the arcs in order of increasing weight, arcs are eliminated earlier. By scanning the nodes in a 'pseudo-topological' order, the computation time can further decrease. In acyclic directed networks one of the resulting all-pairs algorithms runs in $O(m \log n + nm_0)$ time, where m_0 denotes the number of 'essential' arcs, i.e., arcs that are indispensable in some shortest path.

Keywords: Shortest paths; Bellman equations; Directed acyclic graph

1. Introduction

In a directed or undirected network with n nodes and m arcs (or edges), one can determine quickly shortest paths from a single source node to all other nodes as targets, with 'single-source' algorithms known from the literature [1]. However, the running time is larger by an order of magnitude n for the computation of shortest paths between all pairs of nodes.

Frequently, one desires a faster algorithm, especially in situations where all-pairs path computations are to be repeated. In communication networks, one repetitively updates the so-called routing table, using all-pairs computations (e.g., see [2]). Another example is the Steiner tree problem, with applications in VLSI design. Preprocessing programs can be quite effective, but they require multiple updates of all-pairs distances; see Duin and Volgenant [3]. When dealing with combinatorial optimisation problems (e.g., see [4]), a Lagrangian relaxation method to the shortest path problem may be appropriate. Lagrangian relaxation as applied in Bley et al. [5], in order to solve a network design problem, leads to a series of all-pairs shortest path computations in complete networks.

The two all-pairs algorithms of this paper are based on the equations of Bellman [6]; they run fast in sparse as well as dense networks. The order of the algorithmic operations is organised in such a way that the algorithms can eliminate

* Fax: +31 2052 54349.
E-mail address: c.w.duin@uva.nl.

0305-0548/$ - see front matter © 2005 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cor.2005.10.013

'redundant' arcs; algorithmic operations with those arcs may be skipped. Therefore, the algorithms can run faster than any other algorithm up to now, even though they do not have a very favorable time complexity of $O(n^2m)$.

Besides swiftness of computation, the algorithms have other advantages. In telecommunication networks, the paths represent connections and there can be associated with each arc on the path, a switching cost or a risk of failure. From a viewpoint of reliability a minimal number of arcs on the path may be most desirable, while the minimisation of the path's length is valuable as a secondary objective (choosing a shortest path from the collection of paths having the minimum number of arcs). One of the algorithms computes such paths in $O(nm)$ time. Applications modelled by a network often use a directed acyclic graph (DAG), especially when arcs represent transitions in time; a well-known example is a project network. The other algorithm runs on a DAG in $O(m \log n + nm_0)$ time, where m_0 $(\leqslant m)$ denotes the number of 'essential' arcs—those arcs (i, j) for which the arc itself is the only shortest path from i to j.

The paper's presentation is as follows. We review some known shortest path algorithms in Section 1.1. In Section 2, a preliminary algorithm is given as AllPairs1; it implements the Bellman equations for all pairs with $2n$ node lists. The accelerated algorithm AllPairs2 is faster by applying two rules for the elimination of redundant arcs. The two elimination rules are more effective when one processes the (noneliminated) arcs in order of increasing weight. In Section 3, we formulate AllPairs3, a simplified algorithm using only one singly linked list per node. By means of a 'pseudo-topological ordering' of the nodes we accelerate this algorithm. In Section 4.1, the time complexities are discussed; in Section 4.2, we evaluate the performance of the new algorithms numerically. Section 5 summarises the paper.

Throughout the paper, we consider as input a weighted directed graph $G = (V, A, c)$ with $n = |V|$ nodes and $m = |A|$ arcs; the real number $c(i, j)$ denotes the weight (cost or length) of the arc $(i, j) \in A$ with $c(i, j) = \infty$ for $(i, j) \notin A$. The 'distance' $d^*(i, j)$, i.e., the shortest path length with respect to c from a source node i to a target node j, can be computed efficiently under the condition that there are no cycles of total negative arc weight. If such 'nega-cycles' do not exist, then the original arc weights can be transformed to nonnegative weights such that the ordering of the paths by length is preserved (see [7]). Before one runs a shortest path algorithm requiring nonnegative weights, this transformation may be necessary.

All the algorithms discussed can be applied to undirected graphs as well, by means of the so-called 'digraph representation'; each undirected 'edge' (i, j) is then represented by two directed arcs (i, j) and (j, i) of equal weight $c(i, j) = c(j, i)$.

1.1. A short review

Let there be given paths of length $d(i, j) \leqslant c(i, j)$ for all pairs $\langle i, j\rangle \in V \times V$ (with $d(i, i) = 0$). Then $d(i, j) = d^*(i, j)$ for all pairs, if and only if the triangle inequalities hold (see [1])

$$d(i, j) \leqslant d(i, k) + d(k, j) \quad \text{for every } \langle i, k, j\rangle \in V \times V \times V. \tag{1}$$

The generic *label-correcting* algorithm follows a simple scheme. Maintaining tentative, (candidate) 'distance labels' d, one starts with the initialisation

$$\forall \langle i, j\rangle \in V \times V \text{ do } d(i, j) := \infty; \quad \forall i \in V \text{ do } d(i, i) := 0; \quad \forall (i, j) \in A \text{ do } d(i, j) := c(i, j). \tag{2}$$

For as long as (1) is violated, one selects in a 'label check' triplets $\langle i, k, j\rangle \in V \times V \times V$; this label check (for short: check) updates the label $d(i, j)$ as follows:

$$\text{if } d(i, j) > d(i, k) + d(k, j) \quad \text{then } d(i, j) := d(i, k) + d(k, j). \tag{3}$$

Algorithms differ in their selection rule: in what order do we select triplets? With clever rules one might arrive at stopping condition (1) with a bound on the number of checks. On the other hand, it may be possible to guarantee (1) in less time while executing checks for more triplets, provided that these triplets are selected more quickly. Classical algorithms have here a different emphasis. The algorithm of Floyd [19] is most popular, probably because of a simple design, rather than because of its running time. After initialisation (2) it generates and checks triplets in $O(1)$ time per triplet.

for $k := 1$ to n do for $i := 1$ to n do [if $d(i, k) < \infty$ then] for $j := 1$ to n do execute (3).