A linear time algorithm for single source shortest path problem

Pradipta Prometheus Mitra
Ragib Hasan
M. Kaykobad
Department of Computer Science & Engineering,
Bangladesh University of Engineering & Technology,
Dhaka-1000, Bangladesh
email: shmitra@yahoo.com, ragibhasan@yahoo.com, audwit@bdonline.com

Abstract

The shortest path problem is one of the most studied problems in computer science. In this paper, we suggest an algorithm for the solution of the single source shortest path problem in O(mk+n) time, where k, the ratio of maximum and minimum weights of the edges of the graph, is bounded by a constant, m is the number of edges and n is the number of vertices. This algorithm will have better performance than any other algorithm if the graph is highly sparse, and value of k is small.

1. Introduction

The single source shortest path is one of the oldest classical problems in algorithm theory. Given a positively weighted directed graph G with a source vertex s, this problem asks for finding the shortest path from s to all other vertices. Almost all developments concerning this problem have evolved around the famous Dijkstra's algorithm [2, 3], presented first in 1959. The original version of this algorithm ran in $O(n^2+m)$ time[3]. The complexity has since been reduced to $O(m+n\log_2 n)$ using Fibonacci heaps (Fredman and Tarjan, 1987)[4]. Various improvements have since been proposed [4, 5]. However, all these improvements have been limited to special classes of graphs. The current paper proposes to present a linear time algorithm for positively weighted directed graphs. While Thorup[6] developed a linear time algorithm for integral weights on undirected graphs only, our algorithm is valid for all real positive weights for directed graphs. This algorithm will work efficiently in the cases where the ratio between edge weights is bounded by a relatively small constant, and the graph is fairly sparse. We also show that though many algorithms that are linear in theory do not perform well in practice, our algorithm performs well in practice as well.

The assumptions of this algorithm are applicable to several real life problems such as:

- road networks
- Computer network i.e. LANs
- Internet
- Data communication networks
- Electronic Circuit Design

In these cases, no edge is unreasonably larger than the other edges, and therefore, the constant k is not very large. Also, in such graphs, the number of edges is not much greater than the number of vertices, so the space requirement is also low. So, the application of the proposed algorithm will produce the shortest path in linear time. This ensures the applicability of the present algorithm.

2. The Idea

In a directed graph having edges with equal weights, the shortest path can be found using the Breadth First Search (BFS) which runs in linear time. To simplify discussion in this paper by graph we will always mean a directed graph. We intend to transform any given graph G=(V,E) into a graph $G_A(\overline{V}, \overline{E})$ having equal weights on all possible desirable edges.

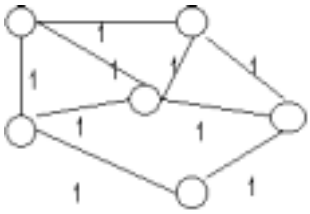

Figure 1: A Graph with Equal Weights

First, we find an edge having the smallest weight w_{min} in G=(V,A) Then we replace each of the other edges with a series of edges with weights w_{min} excepting possibly the last one. After this, we perform a BFS of the graph taking the source vertex as the starting vertex. In this way, we approach any vertex along the shortest path leading to that vertex from the source vertex.

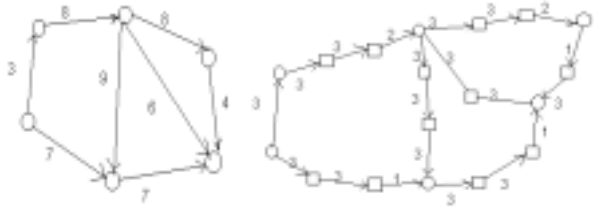

Figure 2: A graph G (left) and its augmented graph G_A (right)
(Square vertices represent the new augmented vertices created by division of edges)

One difficulty can arise in this approach. For example, let us examine the following instance:

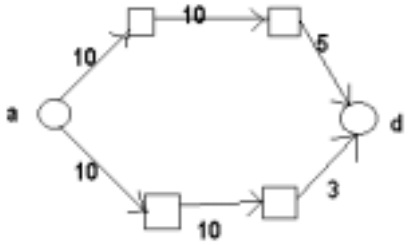

Figure 3: An instance of difference in the last edge weight

In figure 3, w_{min} = 10. But on calculating the shortest path from *a* to *d* in the transformed graph, the two paths differ only in their last edge. In the BFS of the transformed graph, if we traverse the edge with weight 5 first, then we come up with a wrong shortest path. However, the correct shortest path will be found in the limits of the same *distance order search*[1]. We finalize any shortest path at the end of the corresponding distance order search. This means after all vertices in the same level have been visited, i.e. removed from the queue, we will have the correct path length even though we might have had a different value before completing the level. Therefore, problems of this sort are tackled in each level and after completing a level, we can safely assert that all previously reached vertices have the correct shortest path.

Here one point can be noted. Although the augmented graph concept was used in this algorithm, it is not necessary to implement it physically. In that case, the space complexity due to the creation of the augmented graph is removed. In our implementation of this algorithm, we avoided the physical construction of the augmented graph by modifying the queue contents.

3. The algorithm

We describe the algorithm in the following.

```
Augmented_Shortest_Path (G,s)

1.   Find the minimum edge weight w_min in
     O(m) time from the adjacency list.
2.   for all (u,v)∈ E do
3.       inqueue[u,v] ← false
4.       edge[u,v] ← ∝
5.   enddo
6.   d[s]←0
7.   for all (s,u)∈ E do
8.      enqueue(u,s)
9.      inqueue[s,u] ← true
10.     edge[s,u] ← w[s,u]
11.  enddo
12.  while queue ≠ empty do
13.    v,p ← serve()
14.    w_v ← edge[p,v]-w_min
15.    if w_v <= 0 then
16.      if d[p]+w[p,v] <d[v] then
17.       d[v] ← d[p]+w[p,v]
18.       π[v] ← p
19.       for all (v,u)∈ E do
20.        if inqueue[v,u] = false then
21.           if d[v]+w[v,u] < d[u] then
22.               enqueue(u,v)
23.               inqueue[v,u] ← true
24.               edge[v,u] ← w[v,u]+w_v
25.           endif
26.       endif
27.       else
28.       edge[v,u]←min( edge[v,u],w[v,u]
29.        + w_v)
30.       endif
31.      enddo
32.     endif
33.   endif
34.   else
35.    if d[p]+w[p,v] <d[v] then
36.       enqueue(v,p)
37.       edge[p,v] ← w_v
38.    endif
39.   endif
40.  enddo
41.  Return the shortest path.
```

4. Analysis

Time complexity

The performance of the above algorithm heavily depends on the constant k, which is the ratio of w_{max} to w_{min}. The number of new edges is bounded above by m(k-1), where m = number of edges in G. Hence the total number of edges in the augmented graph = m(k-1) + m = mk – m + m = mk in the worst case.

In the average case, the number of edges in the augmented graph is m(k+1)/2.

The number of new vertices in the augmented graph is m(k-1). So the number of total vertices in the augmented graph = m(k-1) + n, where n = number of vertices in the original graph. So the size of the augmented graph G_A is *O(mk+n)*.

In line 1, the search for the smallest edge weight w_{min} and the constant k takes at most *O(m)* time. Lines 12 to 39 of the proposed algorithm take *O(mk/2 +n)* time, which is linear for small values of k. Therefore, this algorithm runs in linear time, provided that the edge weights in the original graph are spread over a reasonable range.

In cases where k is too large, it can be detected in the initial search taken in Line 1 of the algorithm. Then we can switch to Dijkstra's algorithm for finding the shortest path without increasing the asymptotic complexity.

Space Complexity

The graph is represented using the adjacency list implementation. For a graph with n vertices and m edges, the space requirement is O(m+n).

In the worst case, the space requirement for the Queue is equal to the maximum number of edges in the augmented graph, which is O(m).

5. Correctness Proof

We can observe that at any stage of execution of algorithm Augmented_Shortest_Path, there can be at most one entry in the queue corresponding to an edge. This is evident from statement no. 20 of the algorithm.

Let the set V of vertices be partitioned into $V_0, V_1, V_2, \ldots V_k$. where $(i-1) \times w_{min} < d[v] <= i \times w_{min}$ for all $v \in V_i$ for i=1,2,…,k. It may be observed that $V_0=\{s\}$.

Initially algorithm Augmented_Shortest_Path starts with entering all outgoing edges of source s into the queue. When in the while loop of statement 12, all these elements will be processed once, we say that first *distance order* search has been completed. Once i^{th} distance order search has been completed,

[1] Term defined in section 5(Correctness Proof) of this paper.

$(i+1)^{th}$ distance order search starts and is completed when eac element then existing in the queue has been processed exact once.

Now we present the following theorem for proving th correctness of the algorithm Augmented_Shortest_Path :

Theorem

The algorithm Augmented_Shortest_Path discovers all vertices $v \in V_i$ when i^{th} distance order search is complete.

Proof

For i=0, the statement holds true trivially. Let us assume that the statement holds true for i = k, that is, at k^{th} distance order search, we have discovered all vertices of V_k. Now we must prove that at $(k+1)^{th}$ distance order search, we shall discover all vertices of V_{k+1}.

Let us assume that vertex $x \in V_{k+1}$ could not be discovered during $(k+1)^{th}$ distance order search. Let p[s,x] be the shortest path from s to x. Then $\pi[x]$, predecessor of x in this path, must be an element of V_j, with $j \leq k$ and hence has already been discovered by the time k^{th} distance order search has been completed since $w[\pi[x],x] \geq w_{min}$. So, at k^{th} distance order search, edge($\pi[x]$,x) must have already been in the queue. Moreover, since $x \in V_{k+1}$, weight associated with edge($\pi[x]$,x) in the queue must have been lesser than or equal to w_{min} and in this distance order search, this weight must be reduced to a non-positive value indicating that x has been discovered. This contradicts the assumption that x could not be discovered at the $(k+1)^{th}$ distance order search. ■

6. Performance Test

For comparison between the existing Dijkstra's algorithm (using a binary heap) and the proposed Augmented_Shortest_Path(ASP) algorithm, the following simulation runs were performed in an AMD K6-II 400 MHz Computer running Windows 98 operating system. The program was written using C++ Programming Language and was compiled using Visual C++ Compiler.

In the following table, the times for 50 runs for each algorithm are presented.

No.	Ratio w_{max}/w_{min}, K	No. of Vertices n	No. of Edges m	Time for Dijkstra T_D (sec)	Time for ASP T_{ASP} (sec)	Ratio T_D/T_{ASP}
1	30	2000	10000	12.63	10.72	1.178
2	40	2000	10000	12.97	10.82	1.198
3	50	2000	10000	13.02	10.92	1.191
4	30	2000	20000	13.68	11.86	1.151
5	40	2000	20000	13.90	12.02	1.155
6	50	2000	20000	14.17	12.14	1.167
7	30	2000	40000	14.55	14.12	1.023
8	40	2000	40000	14.94	14.34	1.042
9	50	2000	40000	15.38	14.45	1.078
10	100	2000	40000	16.62	15.16	1.083

From the experimental values, it can be observed that the Augmented_Shortest_Path algorithm outperforms Dijkstra's algorithm, under the conditions specified above.

As an aside, researchers have shown that bucket-based algorithms often perform poorly in practical cases[1].

Reference

1. **Asano, Y., Imai H.:** *Practical Efficiency of the Linear Time Algorithm for the Single Source Shortest Path Problem* : .
2. **Cormen,T.H, Leiserson, C.E. and Rivest, R.L.:** *Introduction to Algorithms* : MIT Press, 1990.
3. **Dijkstra, E. W. :***A note on two problems in connection with graphs.* Numer. Math. 1, 269-271. 1959.
4. **Fredman, M.L. and Tarjan, R.E. :** *Fibonacci heaps and their uses in improved network optimization algorithms.* J.ACM 34, 3(July), 596-615. 1987.
5. **Moriya, E and Tsugane, K. :** *Optimally Fast Shortest Path Algorithms for Some Classes of Graphs.* Inter. J. Computer Math. Vol. 70, pp.297-317 1998.
6. **Thorup, Mikkel** : *Undirected Single Source Shortest Paths with Positive Integer Weights in Linear Time.* J.ACM 46, 3(May), 362-394, 1999.