AN EFFICIENT S-ALGORITHM FOR FINDING SINGLE SOURCE SHORTEST PATH PROBLEM IN GRAPH THEORY

Sakthi Kumar R#1, Diwakar Raja A#2, Harsha Vardhan Reddy D#3, Ezhilmaran D

School of Computer Science Engineering, VIT University, Vellore-632014, Tamil Nadu, India

ARTICLE INFO

Corresponding Author
Sakthi Kumar
School of Computer Science Engineering, VIT University, Vellore, Tamil Nadu, India

Keywords: Adjacent vertices-relaxing edges-linked-list-insertion and deletion.

ABSTRACT

The Single-Source Shortest Path problem (SSSP) an explanation for various networking based problems is in fact a basic concept of graph theory. The Dijkstra’s and Bellman-Ford are two well accomplished algorithms to solve the problem. Here in this paper, an algorithm called S-algorithm(S stands for Sakthi, meaning power) with enhanced performance will be discussed. The concept of linked-list will be implemented wherein the weights of the graph are stored in the nodes of the linked-lists. Valid data will be portrayed to prove that the proposed algorithm is more efficient than Dijkstra’s and Bellman-Ford’s algorithm.

©2013, AJCEM, All Right Reserved.

1. INTRODUCTION

The Single-source shortest path problem in graph theory is a solution to find the path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized. This is a fundamental concept of graph theory widely preached and practiced. It has proven to be a successful solution to problems regarding various fields such as communication, routing, supply chain management, road networks.

The more widely used algorithms for the Single-Source Shortest Path algorithm such as the Dijkstra’s employs a simple Breadth-First Search algorithm to traverse between the weights of the graph. The Bellman-Ford algorithm can additionally be applied to negative weights as well. Now an enhanced implementation of Linked-List data structure improves the performance when compared to these traditional algorithms. Each weight will be stored in a distinct node of the Linked-List from where they will be accessed. In this process the traversal among the nodes is improved which is of order O(1) and additionally further weights can be added or deleted efficiently making the problem solving more flexible. In this paper, the proposed S-algorithm and it’s features will be discussed.

In this paper a brief description about insertion and deletion of elements in a linked list is given. A new algorithm namely S-algorithm has been proposed to find the shortest path in the graph with a pseudo code.

2. Basic Concepts

Implementation of single source shortest path problem is done using singly-linked list. Let V be the set of vertices and E be the set of edges and G=(V,E) be the weighted and directed graph used in finding the shortest path. Let us consider a graph G as shown

Finding adjacent vertices is the key to finding shortest path. So here the graph has three vertices. Let the adjacent vertices of each vertex be represented using a table

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Adjacent vertex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2,3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

2.1 Initialise single source (G,s) [1]

1. for each vertex \( v \in V[G] \)
2. \( d[v]=\infty \)
3. \( d[s]=0 \)

Here \( d[v] \) is the distance of vertex from source vertex and the distance of source from source is set to zero.

2.2 Relaxing edges

The process when the distance of the vertex from source is changed from \( \infty \) to any finite value is relaxing and the edges are said to be relaxed.

RELAX \((u, v, w)\) [2]

1. if \( d[v] > d[u] + w(u, v) \)
2. then \( d[v] = d[u] + w(u, v) \)
3. \( \delta[v] = u \)

Here when the distance of a vertex is greater compared to the distance from source using some weighted specified edges leading to shortest path, we modify the distance of vertex to be the weights of collective specified edges.

3. LINKED LISTS

Use of singly-linked lists is carried out as implementation of single source shortest path problem.
Hence insertion and deletion algorithms of linked list is efficient for finding the shortest path.

### 3.1 Insertion algorithm of singly-linked list

Insertion is usually at the back of the singly-linked list.
1. A new node is present connected to null.
2. Now a node to be inserted is entered.
3. The list is traversed from start node till it reaches the final node in the list.
4. Once the final node is reached, the new node is inserted at the end.

**Insertion**
1. Traverse the list using the node as the starting point.
2. If list not empty, visit the list. Update list by list->next.

**Deletion**
1. Delete the node head.
2. Assign head node to temp.
3. Update head by head->next.
4. Free the memory. Now the node adjacent to it has the head pointer pointed at it.

### 3.2 Deletion of first node from the linked list

1. The head pointer is at the first node always.
2. The node to be removed is always the first node.
3. Adjust the head pointer to the next node and remove the original node that had the head pointer.
4. Free the memory. Now the node adjacent to it has the head pointer pointed at it.

**Deletion**
1. Delete the node head.
2. Assign head node to temp.
3. Update head by head->next.
4. Free the memory. Now the node adjacent to it has the head pointer pointed at it.

### 4.1 ALGORITHM FOR SINGLE SOURCE SHORTEST PATH PROBLEM

This section will have a new algorithm namely L-Linked list algorithm for finding the minimum weights of edges from a source vertex to all the other vertices in the graph.

#### 4.1 Description

A graph with six vertices is taken up as example for illustration of the new algorithm. Let the vertices be name v1, v2, v3, v4, v5, v6 respectively. We assume that the initial weights of all the vertices from the source vertex (v1 in this case) is taken to be infinity. We are applying the rules of new algorithm to the graph with six vertices.

A linked list is maintained along with the set of vertices in a tabulated form. The initialization of linked list L to contain just the start or source vertex, which is v1 in this case. Now the process is repeated till the linked list is null. A vertex k is inserted in to the linked list, during the time of removal of k from linked list the adjacent vertices are searched and that are reachable are checked. Let the adjacent vertex of k be m, then if the value d[m] from the source is greater than d[k]+w(k,m), where w(k,m) is the weight between the adjacent vertices k and m, then 0 > d[k] + w(k,m). So the value in this case will be changed to d[k] + w(k,m).

Then we insert m in to the linked list and find its adjacent vertices and same procedure is followed until the list become null.

#### 4.2 ILLUSTRATION

Procedure for L-algorithm is as follows. Let us consider a graph with five vertices as shown in Fig. 2.

1. Initially d[] for all the vertices are taken to be ∞ except the source vertex from where the distance to all the other vertices are found. So it is 0.

![Graph with five vertices](image)

### Fig. 2. Graph with five vertices

<table>
<thead>
<tr>
<th></th>
<th>V1</th>
<th>V2</th>
<th>V3</th>
<th>V4</th>
<th>V5</th>
</tr>
</thead>
<tbody>
<tr>
<td>D[v1]</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

**Linked list:**

L:

2. Vertex v1 is removed from the linked list L. Vertex adjacent to v1 is v2. So the value of d[v1] is checked such as d[v2] = 0 > d[v1]+w(v1,v2) which is 7. Hence the value of v2 is changed from ∞ to 7. Now the process of relaxation is done to the edges (v1, v2). Now vertex v2 is inserted back into the linked list.

<table>
<thead>
<tr>
<th></th>
<th>V1</th>
<th>V2</th>
<th>V3</th>
<th>V4</th>
<th>V5</th>
</tr>
</thead>
<tbody>
<tr>
<td>D[v1]</td>
<td>0</td>
<td>7</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

**Linked list:**

L:

3. Vertex v2 is now removed from the linked list. Two vertices are adjacent to v2 namely v5 and v4. So the values are changed once they are checked with d[v5] = 0 > d[v2]+w(v2,v5), where the total value is 67. And the same procedure for vertex v4 d[v4] = 0 > d[v2]+w(v2,v4), 19 is the value.

<table>
<thead>
<tr>
<th></th>
<th>V1</th>
<th>V2</th>
<th>V3</th>
<th>V4</th>
<th>V5</th>
</tr>
</thead>
<tbody>
<tr>
<td>D[v1]</td>
<td>0</td>
<td>7</td>
<td>∞</td>
<td>19</td>
<td>67</td>
</tr>
</tbody>
</table>

**Linked list:**

L:

4. Vertex v4 is now removed from the linked list. Two vertices are adjacent to v4 namely v5 and v3. So the values are changed once they are checked with d[v5] = 67 > d[v4]+w(v4,v5), here the value of d[v5] is changed from 67 to 51 as it has minimum path weight compared to the other. And the same procedure for vertex v3 d[v3] = 0 > d[v4]+w(v4,v3), 39 is the value. Now the edges (v4, v5) and (v4, v3) are relaxed.

<table>
<thead>
<tr>
<th></th>
<th>V1</th>
<th>V2</th>
<th>V3</th>
<th>V4</th>
<th>V5</th>
</tr>
</thead>
<tbody>
<tr>
<td>D[v1]</td>
<td>0</td>
<td>7</td>
<td>39</td>
<td>19</td>
<td>51</td>
</tr>
</tbody>
</table>
L:  

4.3 Pseudo code for S-algorithm:  
The pseudocode of our algorithm is depicted as follows.  
1.  INITIALIZE-SINGLE-SOURCE(G, s)  
   //Initialize the Graph G and the source vertex s  
2.  INITIALIZE-LIST(L)  //Initialize the linked list L  
3.  INSERTLIST(L, s)  //Insert the source vertex s in the LIST L  
4.  While Not EMPTY(L) 5.  Do DELETELIST(L)  //The DELETE operation on the LIST L where the first  
element in the link is deleted.  
5.  For each edge (u, v) in E[G]  
   //E[G] – egde set of G  
6.  Do tmp←[v]  
7.  RELAX(u, v, w)  
   //The process of relaxing the edge (u, v)  
8.  If (d[v] < tmp) and (v is not in L)  
9.  INSERTLIST(L, v)  //Insert the vertex v in the LIST L  

5. REMARKS:  

The pseudo code has a complexity of O(1) or O(n) for insertion. If it has to find the tail element for insertion it is  
O(n) and if the position is known it takes O(1) at the back  
and for deletion it takes O(1). The time complexity is very less for deletion and hence proves to be very useful while  
dealing with large set of data. Linked list has an average wastage of space of order O(n).  

6. EFFICIENCY OF LINKED LISTS:  

Linked lists allow constant time insertions and deletions. We can walk the lists front and back. Linked lists  
are memory efficient and allow faster access to the nodes.  
In a linked list, insertions and deletions can be handled efficiently without fixing the size of the memory in  
advance. It will be very useful to compute the result faster. Our algorithm mainly seeks to decrease the time taken to  
solve the SSSP and improve the efficiency of the existing algorithms. it mainly improves performance .  

6.1 Efficiency of linked lists over array:[5]  

As shown the linked list has more efficiency compared to arrays. So insertion, deletion and traversing  
in a linked list is much faster compared to that array implementation.  

6.2 Advantages of linked list over queues:  

Adding elements and deleting is much faster linked list when compared to arrays which greatly affects the  
computational time. Queues have a certain order in which the elements has to be inserted but linked lists do not.  
Elements can be inserted in any location. Linked lists also reduce the complexity of the problem when compared to  
using queue.  

CONCLUSION  

Thus this paper is dealt with a newer algorithm which has better time and space complexity compared to  
Dijkstra’s and bellman-ford algorithm. Implementation of single source shortest path using linked list is more  
efficient and processing time is faster. Deletion of head pointer has a complexity of O(1) which is very faster  
compared to the other algorithms.  

REFERENCES  

[1] Tianrui Li, Luole Qi, Da Ruan, An Efficient Algorithm  
for the Single-Source Shortest Path Problem in Graph  
theory, IEEE, Proceedings of 2008 3rd International  
Conference on Intelligent System and Knowledge  
Engineering,2008  
F453/3_3_5/data_structures/miniweb/pg6.htm  
to-use-linkedlist-over-arraylist