Implementation Of Shortest Path Algorithm Using In C

P. Manikandan* , S.Yuvarani*
*-Asst.Professor in Department of Computer Application, Thanthai Hans Roever College, Perambalur-621 212

ABSTRACT

The main purpose of this study is to evaluate the computational efficiency of optimized shortest path algorithms. Our study establishes the relative order of the shortest path algorithms with respect to their known computational efficiencies is not significantly modified when the algorithms are adequately coded .The complexity of Dijkstra's algorithm depends heavily on the complexity of the priority queue *Q*. If this queue is implemented naively, the algorithm performs in O(*n*2), where *n* is the number of nodes in the graph. With a real priority queue kept ordered at all times, as we implemented it, the complexity averages O(*n* log *m*). The logarithm function stems from the collections class, a red-black tree implementation which performs in *O*(*log* (m)). The algorithms are implemented in C Language.

Keywords: Shortest path; Dijkstr's algorithms; undirected Graph;

I.INTRODUCTION

1.1 GRAPH

A linear graph (or simply a graph) G=(V,E) consists of a set of objects V={v_1, v_2, ….. } called vertices, and another set E={e_1,e_2,……}, whose elements are called edges, such that each edge e_k, is identified with an unordered pair(v_i,v_j) of vertices. The vertices v_i, v_j associated with edge e_k are called the end vertices of e_k. The most common representation of a graph is by means of a diagram, in which the vertices are represented as points and each edge as a line segment joining its end vertices. Often this diagram itself is referred to as the graph. The object shown in Fig. 1-1, for instance, is a graph.

Observe that this definition permits an edge to be associated with a vertex pair (v_i, v_j). Such an edge having the same vertex as both its end vertices is called a self-loop or simply a loop. Edge e1 in Fig. 1-1 is a self-loop.

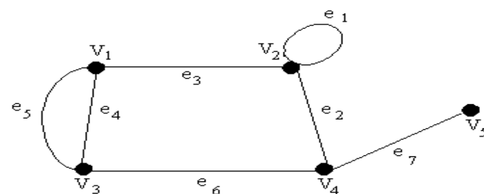

Fig. 1-1 A Graph

More than one edge associated with a given pair of vertices, for example, edges e_4 and e_5 in Fig 1-1. Such edges are referred to as parallel edges. A graph that has does not self-loop nor are parallel edges called a simple graph. Some authors use the term general graph to emphasize that parallel edges and self-loops are allowed.

A graph is also called a linear complex, a 1-complex, or a one-dimensional complex. A vertex is also referred to as a node, a junction, a point, 0-cell, or an 0-simple, Other terms used for an edge are a branch, a line, an element, a 1-cell, an arc, and a 1-simplex.

II.REVIEW OF LITERATURE

As mentioned before, graph theory was born in 1736 with Euler's paper in which he solved the Konigsberg bridge problem. For the next 100 years nothing more was done in the field.

In 1847, G. R. Kirchhoff (1824-1887) developed the theory of trees for their applications in electrical networks. Ten years later, A. Cayley (1821-1895) discovered trees

IJERT

while he was trying to enumerate the isomers of saturated hydrocarbons CnH2n+2.

The other milestone is due to Sir W. R. Hamilton (1805-1865). In the year 1859 he invented a puzzle and sold it for 25 guineas to a game manufacturer in Dublin. The puzzle consisted of a wooden, regular dodecahedron (a polyhedron with 12 faces and 20 corners, each face being a regular pentagon and three edges meeting at each corner. The corners were marked with the names of 20 important cities: London, New York, Delhi, Paris, and so on. The object in the puzzle was to find a route along the edges of the dodecahedron, passing through each of the 20 cities exactly once.

Although the solution of this specific problem is easy to obtain, to date no one has found a necessary and sufficient condition for the existence of such a route (called Hamiltonian Circuit) in an arbitrary graph.

This fertile period was followed by half a century of relative inactivity. Then a resurgence of interest in graphs started during the 1920s. One of the pioneers in this period was D. Konig. He organized the work of other mathematicians and his own and wrote the first book on the subject, which was published in 1936.

The past 30 years has been a period of intense activity in graph theory both pure and applied. A great deal of research has been done and is being done in this area. Thousands of papers have been published and more than a dozen books written during the past decade. Among the current leaders in the field are Claude Berge, Oystein Ore (recently deceased), Paul Erdos, William Tutte, and Frank Harary.

III. COMPUTER REPRESENTATION OF A GRAPH

An algorithm has some inputs – the data with which the algorithm begins (just as a recipe for a dish calls for raw ingredients). Naturally, the input for our algorithms here will be one or more graphs (or digraphs). A graph is generally presented to and is stored in a digital computer in one of the following five forms. Each has advantages and disadvantages. The choice depends on the graph, the problem, the language, the type of machine, and whether or not the graph is modified during the course of the computation.

A) Adjacency Matrix: The most popular form in which a graph or digraph is fed to computer is its adjacency matrix. After assigning a distinct number to each of the n vertices of the given graph (or digraph) G, the n by n binary matrix X(G) is used for representing G during input, storage, and output.

Since each of the n^2 entries is either a 0 or 1, the adjacency matrix requires n^2 bits of computer memory. Bits can be packed into words. Let w be the word length and n be the number of vertices in the graph. Then each row of the adjacency matrix may be written as a sequence of n bits in [n/w] machine words. ([x] Denotes the smallest integer not less than x.) The number of words required to store the adjacency matrix is, therefore, n[n/w].

B) Incidence Matrix: Occasionally, an incidence matrix is also used for storing and manipulation of a graph. An incidence matrix requires n. e bits of storage, which might be more than the n^2 bits needed for an adjacency matrix, because the number of edges e is usually greater than the number of vertices n.

On rare occasions it may be advantageous to use the incidence matrix rather than the adjacency matrix, in spite of the increased requirements in storage. Incidence matrices are particularly favored for electrical networks and switching networks.

IJERT

C) Edge Listing: Another representation often used is to list all edges of the graph as vertex pairs, having numbered the n vertices in some arbitrary order. This graph been undirected, we would simply ignore the ordering in each vertex pair.

Edge listing is a very convenient form for inputting a graph into the computer, but the storage, retrieval, and manipulation of the graph within the computer become quite difficult.

D) Two Linear Arrays: A slight variation of edge listing is to represent the graph by two linear arrays, say $F=(f_1,f_2,\ldots.f_e)$ and $H=(h_1,h_2,\ldots..h_e)$.

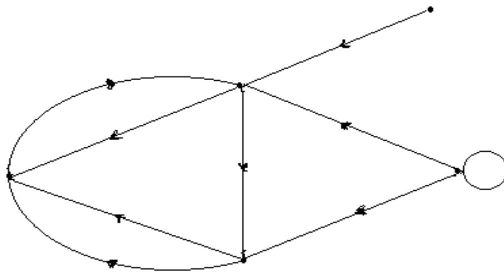

Fig. 3-1 A digraph.

IJERT

Each entry in these arrays is a vertex label. The ith edge e_i is from vertex f_i to vertex h_i if G is a digraph.

E) Successor Listing: Another efficient method used frequently for graphs in which the ratio e/n is not large is by means of n linear arrays.

After assigning the vertices, in any order, the numbers 1, 2, ….n, we represent each vertex k by a linear array, whose first element is k and whose remaining elements are the vertices that are immediate successors of k, that is, the vertices which have a directed path of length one from k.

IV.SHORTEST-PATH ALGORITHMS

A large number of optimization problems are mathematically equivalent to finding shortest paths in a graph. Some of these algorithms are better than others, some are more suited for a particular structure than others, and some are only minor variations of earlier algorithms.

There are different types of shortest-path problems. Most frequently encountered among these are the following five, of which we shall solve the first three:

1. Shortest path between two specified vertices.
2. Shortest paths between all pairs of vertices.
3. Shortest paths from a specified vertex to all others.
4. Shortest path between specified vertices that passes through specified vertices.
5. The second, third, and so on, shortest path.

Algorithm 1: Shortest Path from a Specified Vertex to another Specified Vertex

The problem of finding the shortest path from a specified vertex s to another specified vertex t, can be stated as follows:

A simple weighted digraph G of n vertices is described by an n by n matrix $D=[d_{ij}]$,where D_{ij}=length(or distance or weight) of the directed edge from vertex I to vertex j, $d_{ij}\geq 0$, $d_{ii}=0$, $d_{ij}=\infty$, if there is no edge from i to j (in carrying out a program ∞ is replaced by a large number, say 9999999).

In general, $dij\neq dji$, and the triangle inequality need not be satisfied. That is, $d_{ij}+d_{jk}$ may be less than d_{ik}. [In fact, if the triangle inequality is satisfied, for every i, j, and k, the problem would be trivial because the direct edge(x,y) would be the shortest path from vertex x to vertex y.] The distance of a directed path P is defined to be the sum of the lengths of the edges in P. The problem is to find the shortest possible path and its length from a starting vertex s to a terminal vertex t.

Description of the Algorithm: Dijkstra's algorithm labels the vertices of the given digraph. At each stage in the algorithm some vertices have permanent labels and others temporary labels. The algorithm begins by assigning a permanent label 0 to the starting vertex s, and a temporary label ∞ to the remaining n-1 vertices. From then on, in each iteration another vertex gets a permanent label, according to the following rules:

1. Every vertex j that is not yet permanently labeled gets a new temporary label whose value is given by Min [old label of j, (old label of i+d_{ij})], Where i is the latest vertex permanently labeled, in the previous iteration, and d_{ij} is the direct distance between vertices i and j. If i and j are not joined by and edge, then $d_{ij}=\infty$.
2. The smallest value among all the temporary labels is found, and this becomes the permanent label of the corresponding vertex. In case of a tie, select any one of the candidates for permanent labeling.

Steps 1 and 2 are repeated alternately until the destination vertex t gets a permanent label.

The first vertex to be permanently labeled is at a distance of zero from s. The second vertex to get a permanent label (out of the remaining n-1 vertices) is the vertex closest to s. Form the remaining n-2 verices, the next one to be permanently labeled is the second closest vertex to s. And so on. The permanent label of each vertex is the shortest distance of that vertex from s. As an illustration of Dijkstr's procedure, let us find the distance form vertex B to G in the digraph shown in Fig.4-1.

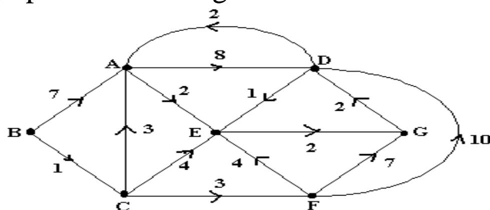

Fig.4-1 Simple weighted sub graph

IJERT

All steps are easily programmed except for the job of distinguishing the permanently labeled vertices from the temporarily labeled ones, which is slightly tricky. An efficient method of accomplishing this is to associate indices 1,2,…,n with the vertices, and keep a binary vector VECT of order n. When the ith vertex becomes permanently labeled, the ith element in this binary vector changes from 0 to 1.

A flow chart of this algorithm is given in Fig.4-3. The algorithm described does not actually list the shortest path from the starting vertex to the terminal vertex; it only gives the shortest distance. The shortest path can be easily constructed by working backward from the terminal vertex such that we go to that predecessor whose label differs exactly by the length of the connecting edge. (A tie indicates more than one shortest path).

Alternatively, the shortest path can be determined by keeping a record of the vertices from which each vertex was labeled permanently. This record can be maintained by another linear array of length n, such that whenever a new permanent label is assigned to vertex j, the vertex from which j is directly reached is recorded in the jth position of this array.

Remarks

1. In this algorithm, had we continued the labeling until every vertex got a permanent label (rather than stopping at the permanent labeling of the destination vertex t), we would have gotten an algorithm for the shortest paths from starting vertex s to all other vertices.
2. If we take a shortest path from the starting vertex s to each of the other vertices(which are accessible from s), then the union of these paths will be an arborescence T rooted at vertex s. Every path in T from s is the (unique) shortest path in the digraph (or graph, as the case may be). Such a tree is called the shortest-distance

arborescence. For example, the shortest-distance arborescence of Fig.4.1 is given in Fig.4.2

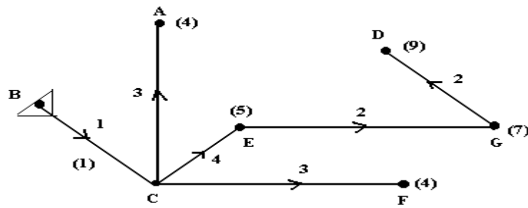

Fig. 4-2. Shortest-distance arborescence of Fig.4-1.

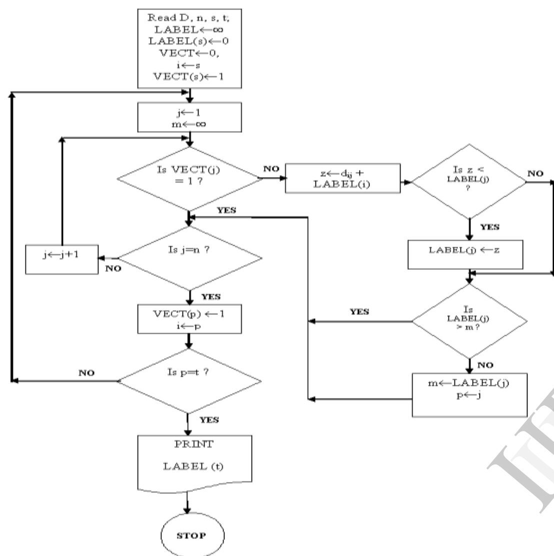

Fig. 4-3 shortest distance from s to t.

3. In this algorithm, as more vertices acquire permanent labels the number of additions and comparisons needed to modify the temporary labels continues to decrease. In the case where every vertex gets permanently labeled, we need n(n-1)/2 additions and 2n(n-1) comparisons. Thus the computation time is proportional to n^2.
4. Notice that for a given n the computation time is independent of the number of edges the digraph may have. Another random graph with 80 vertices but only 1000 edges also took 36/60 second for the same computation.
5. If the digraph is sparse [i.e., the number of edges e is much smaller than n(n-1)], it is possible to reduce the time of computation. This can be achieved by incorporating another test which alters the temporary labels of only those vertices that are successors of the most recent permanently labeled vertex
6. If the given digraph G is not weighted, every edge in G has a weight of one, and matrix D is the same as the adjacency matrix. Then the problem is simpler. We perform logical operations rather than real arithmetic.
7. We have assumed the distances d_{ij} are all nonnegative numbers. If some of the distances are negative, Algorithm 2 will not work.(Negative distances in a network may represent costs and the positive ones profits.) The reason for the failure of Algorithm 2 is that once a vertex is permanently labeled its label cannot be altered. Shortest-path algorithms have, however, been proposed that will solve this problem, provided the sum of all d_{ij} around every directed circuit is positive. The computation time of the existing algorithms that can handle negative d_{ij} is n3 and not n^2.
8. It was suggested by T.A.J Nicholson that carrying the shortest-path algorithm simultaneously from both ends s and t would improve the speed. Dreyfus has, however, shown that the double-ended procedure would improve the efficiency only in certain types of digraphs. In the case where nearly all n vertices must be permanently labeled from either one and or the other, the double-ended procedure is actually less efficient than Dijkstra's one-ended procedure.

```
/* TO FIND THE SHORTEST PATH OF A
GRAPH USING DJIKSTRA ALGORITHM
*/
#include<stdio.h>
#include<limits.h>
/* Maximum Number of Nodes in a Graph */
#define MAXNODE 10
#define PERM 1
#define TENT 2
#define infinity INT_MAX
typedef struct NODELABEL
```

IJERT

```
{
int predecessor;
int length; /* Optimal distance from Source
*/
int label; /* label is tentative or permanent */
}NODELABEL;
```

/* **Function**: Short path Prototype:
int Short Path(a, n, s, t, path, dist)
Input: a-Adjacency Matrix describing the graph n-Number of Nodes in the graph
s-Source Nodet-Target Node or Sink Node
Output: Path - list of optimal path from source to sink dist - Minimum Distance between source and sink
Returns: 0 - if there is no path count indicating the number of nodes along the optimal path, otherwise */

```
int Short Path( a, n, s, t, path, dist )
int  a[MAXNODE][MAXNODE], n, s, t,
path [MAXNODE], *dist;
{
NODELABEL state[MAXNODE];
int i, k, min, count;
int r Path[MAXNODE];
*dist=0;
/* Initialize all nodes as tentative nodes */
for(i=1; i<=n; i++)
{
state[i].predecessor=0;
state[i].length=infinity;
state[i].label=TENT;
}
/* Make source Node as Permanent */
state[s].predecessor=0;
state[s].length=0;
state[s].label=PERM;
/* Start from source node */
k=s;
do
{
/* Check all the paths from Kth node and find
their distance
form K node */
for(i=1; i<=n; i++)
{
/* -ve if no direct path, 0 if to the same
otherwise direct
path */
if(a[k][i]>0 && state[i].label==TENT)
{
if(state[k].length+a[k][i]<state[i].length)
{
state[i].predecessor=k;
state[i].length=state[k].length+a[k][i];
}
}
}
/* Find the tentatively labeled node with
smaller cost */
min=infinity;
k=0;
for(i=1; i<=n; i++)
{
if(state[i].label==TENT &&
state[i].length<min)
{
min=state[i].length;
k=i;
}
}
/* Is Source Or Sink Node is Isolated */
if(k==0)
return(0);
state[k].label=PERM;
}while(k!=t);
/* Store Optimal Path */
k=t;
count=0;
do
{
count = count + 1;
rPath[count]=k;
k=state[k].predecessor;
}while(k!=0);
/* Reverse nodes since algorithm stores path
in reverse
direction */
for(i=1; i<=count; i++)
path[i]=rPath[count-i+1];

for(i=1; i<count; i++)
```

IJERT

```
*dist+=a[path[i]][path[i+1]];
return(count);
}
void main()
{
int a[MAXNODE][MAXNODE], i, j;
int path[MAXNODE];
int from, to, dist,count, n;
printf("\nHow many Nodes?");
scanf("%d", &n);
printf("%2d", n);
for(i=1; i<=n; i++)
{
printf("\n Enter Node %d Connectivity: ",i);
for(j=1; j<=n; j++)
{
scanf("%d", &a[i][j]);
printf("%2d", a[i][j]);
}
}
printf("\n From to Where?");
scanf("%d %d", &from, &to);
printf("%d %d", from,to);
count = ShortPath(a,n,from, to, path, &dist);
if(dist)
{
printf("\nShortest Path:");
printf("%d",path[1]);
for(i=2;i<=count;i++)
printf("->%d",path[i]);
printf("\n Minimum Distance = %d\n", dist);
}
else
printf("\n Path does not exist \n");
}
```

OUTPUT

```
How many Nodes?     3
                    3
Enter Node 1 Connectivity:
 0      0      7
 0      0      7
Enter Node 2 Connectivity:
0       0      0
0       0      0
Enter Node 3 Connectivity:
3       0      3
3       0      3
From to Where?
        1      3
        1      3
Shortest Path:  1→3
Minimum Distance = 7

How many Nodes?     5
        5
Enter Node 1 Connectivity:
 0      85     80     20     0
 0      85     80     20     0
Enter Node 2 Connectivity    :
0       0      20     0      95
0       0      20     0      95
Enter Node 3 Connectivity:
70      20     0      80     0
70      20     0      80     0
Enter Node 4 Connectivity:
0       75     0      0      75
0       75     0      0      75
Enter Node 5 Connectivity:
70      10     20     80     0
70      10     20     80     0
From to Where?       1      5
        1      5
Shortest Path:  1→4→5
Minimum Distance = 95
```

IJERT

V. CONCLUSION

This study is the first to use C programming language to evaluate the efficiency of shortest path algorithms, and it yields several interesting conclusions.

To the first question that prompted our research effort, we provided empirical evidence that the relative order of his shortest computational efficiency is not modified when the algorithms are coded in C. Algorithms that use thresholds are still, in general, the fastest. We also have shown here that C implementations (using pointers) of shortest path algorithms are significantly

more efficient than traditional ones (using arrays).

The improvement in computing times is essentially due to the fact that in C-implementations, the forward star of each node may be scanned without performing any multiplication. The capabilities that explain this property have a second important effect as the data structure used to keep the scan eligible list Q may also by managed (search, insertions, deletions etc.) by using only additions and no multiplications. Consequently, by using C implementations with pointers, one may expect to speed up factor may reach 30%.

We have also shown that the level of difficulty required to implement shortest path algorithms in C by using pointers is not greater than that required by traditional implementations that use arrays. Thus, the significant efficiency gains reported in this paper are due to the choice of the proper use of its data structure manipulation capabilities. We expect similar result to be achieved for other network as well.

VI. REFERENCE

1. Mobius, A. F. (1828), "Kann von zwei dreiseitigen Pyramiden eine jede in Bezug auf die andere um- und eingeschrieben zugleich heissen?", *J. Reine Angew. Math.* **3**: 273–278. In *Gesammelte Werke* (1886), vol. 1, pp. 439–446

2. Cayley, A. (1875), "Ueber die Analytischen Figuren, welche in der Mathematik Bäume genannt werden und ihre Anwendung auf die Theorie chemischer Verbindungen", *Berichte der deutschen Chemischen Gesellschaft* **8**: 1056–1059,

3 D. Konig, "Theorie der endlichen und unendlichen Graphen" , Teubner, reprint (1986)

4. G. Kirchhoff, *Poggendorff Annalen*, **72** (1847) pp. 497–508

5. A. Cayley, "On the theory of the analytical forms called trees", *Collected mathematical papers*, **3**, Cambridge Univ. Press (1854) pp. 242–26

VII. AUTHOR INFORMATION

Mr.P.Manikandan recived his M.Phil in computer science from Bharathidasan University, Tiruchirappalli, India in 2005. Pursuing Ph.D in computer science at Bharathiar University, Coimbatore, India. Currently he is working as Asst.Professor in the Department of Computer Application, Thanthai Hans Roever College, Perambalur, India. His current research interests include Data structure algorithms, Image mining, Nano Computing, software metrics. He has published 3 papers in international Conference, 2 papers in international Journals and 6 papers in national Conferences.

Mrs.S.Yuvarani recived her M.Phil in computer science from Allagappa University, karaikudi, India in 2005. Pursuing Ph.D in computer science at Bharathiar University, Coimbatore, India. Currently she is working as Asst.Professor in the Department of Computer Application, Thanthai Hans Roever College, Perambalur, India. Her current research interests include Data structure algorithms, Image mining. She has published 3 papers in international Conference, 2 papers in international Journals and 4 papers in national Conferences.

IJERT