International Journal of Computer Science & Applications
Vol. II, No. II, pp. 22 - 37

COMPUTATIONAL EFFICIENCY OF OPTIMIZED SHORTEST PATH ALGORITHMS

P. Biswas, P. K. Mishra and N. C. Mahanti
Department of Applied Mathematics,
Birla Institute of Technology, Mesra (India)
Email : pkmishra@bitmesra.ac.in, pkmishra@ieee.org

1. 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. An important contribution of this study is the inclusion of the re-distributive heap algorithm. In spite of having its best theoretical efficiency, the re-distributive heap algorithm does not emerge as the best method in terms of computational efficiency. The complexity of Dijkstra's algorithm depends heavily on the complexity of the priority queue *Q*. If this queue is implemented naively (i.e. it is re-ordered at every iteration to find the minimum node), 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 $O(log\ (m))$.

2. INTRODUCTION

Dijkstra's Algorithm, introduced in 1959 provides one the most efficient algorithms for solving the shortest-path problem. In a network, it is frequently desired to find the shortest path between two nodes. The weights attached to the edges can be used to represent quantities such as distances, costs or times. In general, if we wish to find the minimum distance from one given node of a network, called the source node or start node, to all the nodes of the network, Dijkstra's algorithm is one of the most efficient techniques to implement. In general, the distance along a path is the sum of the weights of that path. The minimum distance from node a to b is the minimum of the distance of any path from node a to b.

Dijkstra's algorithm is probably the best-known and thus most implemented shortest path algorithm. It is simple, easy to understand and implement, yet impressively efficient. By getting familiar with such a sharp tool, a developer can solve efficiently and elegantly problems that would be considered impossibly hard otherwise. As we explore a possible implementation of Dijkstra's shortest path algorithm in C. Let $N = (V,A,L)$ denote a network over a directed graphs [1] $G = (V,A)$ with $n = |V|$ node and $m = |A|$ arcs, and a set of function L defined on the graph. For the problem considered in this paper, the set L consists of a length function $l:A \rightarrow Z^+$, where Z^+ is the set of positive integers. If $U = (i,j) \in A$, then the number L_u will be called the weight of length of arc U. We also use l_{ij} to denote the length of arc (i,j). We shall denote the maximum arc weight by l_{max} in the network. The set of all arcs emanating from node i is called the forward star of i, and is denoted by L_i and (r,s) path is a sequence of arcs of the form (r,i_1) , (i_1,i_2), (i_k,s). Dijkstra's algorithm, when applied to a graph, quickly finds the shortest path from a chosen source to a given destination. In fact, the algorithm is so powerful that it finds all shortest paths from the source to all destinations. This is known as the *single-source* shortest paths problem. In the process of finding all shortest paths to all destinations, Dijkstra's algorithm will also compute, as a side-effect if we will, a *spanning tree* for the graph. While an interesting result in itself, the spanning tree for a graph can be found using lighter (more efficient) methods than Dijkstra's.

3. ALGORITHMS AND MANIPULATION STRATEGIES

Algorithms are very useful in real life applications. The real-world performance of any software system depends on only two things: (1) the algorithms chosen and (2) the suitability and efficiency of the various layers of implementation. Good algorithm design is therefore crucial for the performance of all software systems. Moreover, the study of algorithms provides insight into the intrinsic nature of the problem as well as possible solution techniques independent of programming language, programming paradigm, computer hardware, or any other implementation aspect. An important part of computing is the ability to select algorithms appropriate to particular purposes and to apply them, recognizing the possibility that no suitable algorithm may exist. This facility relies on understanding the range of algorithms that address an important set of well-defined problems, recognizing their strengths and weaknesses, and their suitability in particular contexts. Efficiency is a pervasive theme throughout this area.

First let's start by defining the entities we use. The graph is made of *vertices* (or nodes, we will use both words interchangeably), and *edges* which link vertices together. Edges are directed and have an associated *distance*, sometimes called the weight or the cost. The distance between the vertex *u* and the vertex *v* is noted [*u, v*] and is always positive.

Dijkstra's algorithm partitions vertices in two distinct sets, the set of *unsettled* vertices and the set of *settled* vertices. Initially all vertices are unsettled, and the algorithm ends once all vertices are in the settled set. A vertex is considered settled, and moved from the unsettled set to the settled set, once its shortest distance from the source has been found. We all know that *algorithm + data structures = programs*, in the famous words of Niklaus Wirth. The following data structures are used for this algorithm.

d	Stores the best estimate of the shortest distance from the source to each vertex .
p	Stores the predecessor of each vertex on the shortest path from the source.
S	The set of settled vertices, the vertices whose shortest distances from the source have been found.
Q	The set of unsettled vertices.

With those definitions in place, a high-level description of the algorithm is deceptively simple. With *s* as the source vertex:

```
// initialize d to infinity, p and Q to empty
d = ( 8 )
p = ()
S = Q = ()
add s to Q
d(s) = 0
while Q is not empty
{
u = extract-minimum(Q)
add u to S
relax-neighbors(u)
}
```

The two procedures called from the main loop are defined below:

```
relax-neighbors(u)

{
   for each vertex v adjacent to u, v not in S
   {
      if d(v) > d(u) + [u,v]    // a shorter distance exists
```

```
        {
            d(v) = d(u) + [u,v]
            p(v) = u
            add v to Q
        }
    }
}

extract-minimum(Q)
{
   find the smallest (as defined by d) vertex in Q
   remove it from Q
   return it
}
```

The simplest way to keep the nodes in the scan eligible set Q is to use a list. Queues are used to implement the well known shortest path method credited to Moore [18] and Bellman[3]. This method, which we refer to as LQUEUE processes nodes in a "*first in first out*" manner, that is, the graphs are explored level by level. In fact, all shortest paths of one arc are first computed, followed by those made up of two arcs, and so on. Since the number of arcs in shortest path is bounded by n, the computational complexity of LQUEUE is $O(mn)$.
A deque may be viewed as a stack and a queue connected in series in such a way that the tail of the stack points to the head of the queue. Deques are used to implement the two way criterion idea attributed to Pape [19]. This method at the head of Q if it has already appeared on the list before appends the node at its tail otherwise nodes are always removed form the head of the scan eligible node list. When all nodes have a finite label, the method behaves like an algorithm that uses a stack. The computational complexity of LDEQUE is then $O(n2^n)$.

Glover et al [9, 10] introduced a new criterion to control the insertion of nodes into the scan eligible node list. This method, called *partitioning shortest path algorithm*, divides the list Q into two distinct lists, NOW and NEXT. A threshold value t is computed to determine which nodes currently in NEXT may be moved to NOW. At each insertion, a node i is chosen in the NOW list in a "last-in first-out" way. After the NOW list is exhausted, a new value for t is computed and qualifying nodes in NEXT are transferred to NOW. The process is repeated until NEXT is empty. Glover et al [10] showed that any partitioning shortest path procedure is polynomially bounded. Based on this approach, different algorithms may be obtained according to the formula used to compute the threshold value. The method, we have implemented, called LTHRS, is based on the procedure THRESH-*X2* due to Glover et al [10] and an implementation strategy developed by Gallo and Pallottino [8] insert these nodes directly at the tail of NOW leaving a copy in NEXT (when NEXT is scanned, copies are deleted). In practice, however, this does not seem to effect the performance of algorithm. In order to implement each of these four methods properly and efficiently linked list have to be used.

$$Q[I] = J\begin{cases} 0 & \text{if } i \text{ is not in } Q. \\ & \text{if } i \text{ precedes } j \text{ in } Q. \\ N+1 & \text{if } i \text{ the last element of } Q. \end{cases}$$

and $Q[N+1]$ = the head of $Q[\ ]$

Fig-1 Linked list implementation by means of an array *Q* [ ]

```
Struct node {
Struct are *a
Struct node *P;
Struct node *q;
};
```

Fig-2 Node structure for linked list implementation

Addition at the head

```
j → q = n1;                    /*Q[J] = Q[N+1]*/
n1 → q = j;                    /*Q[N+1] = J */
if ( last == n1) last = j;     /* IF (LAST.EQ.N+) LAST = J */
```

Addition at the tail

```
last q = j;                    /*Q[LAST) = J* /
j → q = n1;                    /*Q[J] = N+1*/
last = j;                      /*LAST = J*/
```

Addition at the position pointed to by pntr (only for L2QUE)

```
j → q = pntr → q;              /*Q[J]=Q [PNTR]* /
pntr → q = j;                  /*Q [PNTR] = J*/
if ( last == pntr) last = j;   /* IF (LAST.EQ.PNTR) LAST = J*/
pntr = j;                      /* pntr = J*/
```

Deletion (always at the head)

```
i = n1                         /*I = Q(N+1)*/
n1 → q = i → q;                /*Q[N+1]=Q[I]*/
i → q = nil;                   /*Q(I) = 0 */
if (last == i) last = n1       /* IF (LAST.EQ.I) PNTR = N + 1 */
if (pntr  == i ) pntr = n1     /* only for L2QIE procedure */
```

Fig- 3 Linked list manipulation with pointers.

Several operations are possible on a priority queue. These operations are used by label setting algorithms:

INSERT (i,Q)	:	insert item *i* into priority *Q* not previously containing *i*.
REINSERT (i,Q)	:	Reinsert item *i* into priority queue *Q* after the label of *i* has been modified.
DELETEMIN (Q)	:	Delete and return an item of minimum (or maximum) value label from priority queue *Q*.

The idea of using a priority queue in order to compute shortest path is credited to Dijkstra [6]. In terms of the operations described above, the algorithms may be stated as follows:

DIJKSTRA'S ALOGORITHM

Edsger W. Dijkstra is a Dutch computer scientist and mathematician who has made a number of remarkable contributions to computer science. Among the many algorithms he originated, the shortest path algorithm is probably the best known. The basic reasoning behind the algorithm is as follows:

- Like BFS, the algorithm gradually creates a tree from the vertices and some of the edges in the graph; the root of this tree is the source of the search, vertex *s*, as specified at the outset of the algorithm. Thus for each vertex *u*, we maintain a **parent pointer** $p[u]$ and a **distance** field $d[u]$ (the distance, of course, is measured from the source.)

- Like BFS, the algorithm maintains a queue in which it keeps the vertices it has not yet examined. However, this is a priority queue, not a simple first-in-first-out queue. This priority queue is sorted using the *d*[] field of the vertices.
- Unlike BFS, Dijkstra's algorithm maintains a **solution set** *S* which contains all the vertices examined so far *whose minimal distance to the source has been determined*. As the algorithm progresses and examines more and more vertices, this set grows.
- At the outset, we set all parent pointers to NIL and all distance fields to infinity, except that of the source which we set to zero. Likewise, we set the solution set to empty.

This reflects the fact that we have accumulated no information about any of the vertices. The priority queue, on the other hand, is filled with every vertex in the graph -- since none of them have been dealt with yet. However, since all the *d*[] fields are initially infinite, they will not be in any particular order -- *except* the source, which will therefore be the first to be extracted. This is important.

- Each vertex in the priority queue is extracted in sequence (remember, the priority queue is ordered based on the *d*[] field of the vertices).
- We add the vertex extracted, vertex *u*, to the solution set *as is*.
- For each neighbor *v* of this new vertex, we attempt to tighten down the *d*[] field -- that is, attempt to make a better estimate than the one we already have. How can we do this? We take our current estimate, and compare it to the sum of *u*'s distance from the root (that is, *d*[*u*]) and the weight of the edge between *u* and *v* (that is, *w*[*u*, *v*)]. If that sum is less than our current estimate, than our current estimate gets replaced by it. This process is called **relaxing** the edge which links the two vertices.

The result is that at any given time during execution, the *d*[] field of vertices in the solution set is permanently locked in and represents the shortest path length from that vertex to the source. For vertices *outside* the solution set (except for those set to infinity, sitting at the bottom of the priority queue), the *d*[] field is a *worst-case guess* – a path length we already know is achievable, by following some edge leading to one of its neighbors in the solution set. There may exist a shorter path which does not lead directly into the current solution set, in which case we will discover it at some later time when the rest of the nodes along that better path have been worked out.

Note that if *v*'s distance does indeed change, it will be necessary to adjust its position in the priority queue by "bubbling up" -- basically the same mechanism used when inserting items into a priority queue.

One can picture the solution set as a growing island, gradually assimilating mo re and more vertices which are floating about around it. At each iteration of the loop, we need to build a "bridge" from the "island" to one of the outlying vertices. The rule we use to pick the vertices to connect with this bridge (viz., chosen with the priority queue) stems from a very convenient property of this algorithm: *By the time a vertex in the priority queue reaches the top position, its best bridge is the one which leads to the vertex on the island with the shortest path to the source.*

- That's it! If we don't want to explore the whole graph but rather measure the distance to a specific destination, all we have to do is stop when the destination vertex enters *S* and follow the parent pointers back up to the source. If we do explore the entire gra ph, we'll get the shortest path to *every point* from the single source

Here is the formal notation of the algorithm:

Step 1. for each u e G:
Step 1.1. d[u] ¬ infinity;
Step 1.2. p[u] ¬ NIL;
Step 2. d[s] ¬ 0;
Step 3. S ¬ Ø;

```
Step 4. Q ¬ V;
Step 5. while NotEmpty(Q):
Step 5.1.    u ¬ DeleteTop(Q);
Step 5.2.    S ¬ S union { u };
Step 5.3.       for each v adjacent to u:
Step 5.3.1.          if d[v] > d[v] + w[u, v]:
Step 5.3.1.1.             d[v] = d[u] + w[u, v];
Step 5.3.1.2.             p[v] = u;
Step 5.3.2.3.             DecreaseKey[v, Q].
```

Dijkstra(*G*,*s*) finds all shortest paths from *s* to each other vertex in the graph, and shortest path (*G*,*s*,*t*) uses Dijkstra to find the shortest path from *s* to *t*. Uses the priority dictionary data structure to keep track of estimated distances to each vertex.

Find shortest paths from the start vertex to all vertices nearer than or equal to the end. The input graph *G* is assumed to have the following representation: A vertex can be any object that can be used as an index into a dictionary. *G* is a dictionary, indexed by vertices. For any vertex *v*, *G*[*v*] is itself a dictionary, indexed by the neighbors of *v*. For any edge *v*®*w*, *G*[*v*][*w*] is the length of the edge. Guido van Rossum suggests representing graphs as dictionaries mapping vertices to lists of neighbors, however dictionaries of edges have many advantages over lists: they can store extra information (here, the lengths), they support fast existence tests, and they allow easy modification of the graph by edge insertion and removal. Such modifications are not needed here but are important in other graph algorithms. Since dictionaries obey iterator protocol, a graph represented as described here could be handed without modification to an algorithm using Guido's representation. Of course, *G* and *G*[*v*] need not be Python dict objects; they can be any other object that obeys dict protocol, for instance a wrapper in which vertices are URLs and a call to *G*[*v*] loads the web page and finds its links. The output is a pair (*D*,*P*) where *D*[*v*] is the distance from start to *v* and *P*[*v*] is the predecessor of *v* along the shortest path from *s* to *v*. Dijkstra's algorithm is only guaranteed to work correctly when all edge lengths are positive. This code does not verify this property for all edges (only the edges seen before the end vertex is reached), but will correctly compute shortest paths even for some graphs with negative edges, and will raise an exception if it discovers that a negative edge has caused it to make a mistake.

```
D = {}      # dictionary of final distances
P = {}      # dictionary of predecessors
Q = priorityDictionary()  # est.dist. of non-final vert.
Q[start] = 0
            for v in Q:
            D[v] = Q[v]
            if v == end: break
                        for w in G[v]:
                        vwLength = D[v] + G[v][w]
                        if w in D:
                                    if vwLength < D[w]:
                                                raise ValueError, \
"Dijkstra: found better path to already-final vertex"
                        else if w not in Q or vwLength < Q[w]:
                                    Q[w] = vwLength
                                    P[w] = v
            return (D,P)
```

shortestPath(G,start,end).

Find a single shortest path from the given start vertex to the given end vertex. The input has the same conventions as Dijkstra(). The output is a list of the vertices in order along the shortest path.

```
D,P = Dijkstra(G,start,end)
Path = []
while 1:
        Path.append(end)
        if end == start: break
        end = P[end]
Path.reverse()
return Path.
```

The *C* implementation is quite close to the high-level description we just walked through. For the purpose of this article, my *C* implementation of Dijkstra's shortest path finds shortest routes.

We have listed above the data structures used by the algorithm, let's now decide how we are going to implement them in *C*. It can easily shown that, if all arc cost are non-negative, then Dijkstra's algorithm performs exactly n DELETMIN operations. The simplest way to manage a priority queue is to use an ordered linked list that contains all the nodes in the priority queue. In this case, the DELETMIN operation consists of scanning the list in order to find the minimum label node. A variant uses an unordered linked list in the literature, these methods have been proven to perform poorly on almost every graph topology [8, 16, 18]. For this reason, this paper does not cover the linked list implementations of priority queues.

In fact, both ordered and unordered linked list display the property that the number of operations needed to insert or delete nodes is a priority queue. This implies an $O(n^2)$ theoretical complexity. A significantly more efficient implementation of priority queue is obtained with either heap or bucket structures. A d-heap is a heap ordered tree for which each node has at most d-children and nodes are added in breadth-first order. This time for inserting an item in a d-heap of k-items is $\log_d k$; for deleting an item, it is $d \log_d k$. Thus, the total time needed by an implementation of the Dijkstras algorithm that makes use of a – heap is $nd\log_d n + m \log_d n$. The value of d that minimizes the total bound for the algorithm is a solution of the equation $d(\log_e d - 1) = m/n$, and good approximations are either $d = max\{2, m/n\}$ or $d = 2 + m/n$ [17, 20] on the other hand Johnson [14] provides empirical evidence that by setting $d = 2$ provides the fastest heap structure for sparse problems. This may be explained by the fact that the computational bound indicated above does not take into account the constants that are involved in heap operation. Also, it is worthwhile to round the value of d to the nearest power of 2 so that multiplications and divisions may be achieved via binary shifts.

For these reasons, the code used in this study makes use of binary heap.

A binary heap is efficiently implemented by Mishra [16] on a digital computer by using a pair of arrays HP[ ] and Q[ ] of dimensions n. The cell HP [k] contains the code number placed in position k of the heap. The array Q[ ] plays the role of the heap directory.

$$Q[I] = \begin{cases} k & \text{if } HP[k] = I \\ 0 & \text{if } i \text{ is not in } Q. \end{cases}$$

Thus, the children of heap node k are the cell number $2k$ and $2k+1$. The parent of a node k is simply $\lfloor k/2 \rfloor$. Finally, the variable NHP is used to keep track of the heap size (the number of nodes currently in the heap).

Pointer multiplication and pointer division are not allowed in the *C*-programming language [15]. For this reason, there is no real need for the elements in Q[ ] to be pointers. However, an efficient implementations is achieved by using the variable shown in fig. 3. The procedures used to manipulate binary heap structures in *C* are displayed in fig. 4. The main advantage of this approach over traditional implementation is due to the fact that heap manipulation often need to access the value D[HP[k]]. In *C*, this information is obtained more rapidly via pointers. It is worth mentioning that Fredman and Tarjan [7] have recently proposed a new heap structure for implementing priority queues. This structure, called Fibonacci heap produces shortest path algorithms with very good computational complexity. However, it requires a very sophisticated implementation and considerable memory space (7 arrays of dimension n). Besides, as reported by Hung and Divoky [13], it is disappointingly slow. Consequently, we have not considered it in our study.

For a time, the algorithm developed by Dial [5] was considered as the fastest priority queue implementation for the Dijkstra algorithm. Then different studies have shown that Dial's method is dominated by other algorithms from Dial [4], Glover et al [9] and Gondran et al [12]. However, all studies were based on FORTRAN implementations of the algorithms. On one hand, the poor performance of the method is attributed to the scanning of a long array (in order to find the next first non empty bucket) while, on the other hand, this operation is efficiently performed in C. This code will be referred to as SDIAL.

Recently, Ahuja et al [2] suggested a very simple and elegant use of buckets called re-distributive heaps. The shortest path algorithms using re-distributive heaps have the best known computational complexity. In order to verify their computational efficiency, we implemented the one-level re-distributive heap. The program will be referred to as SRHEAP.

Removal of the minimum label node from Q

```
i = hP[1]; i → q = 0; min = i → d;
x = hP[nhP]; dummy = x→ d; nhP--;
if (nhP0 {
hi = 1
do {
h2 = (h1<<1);
if (h2 £ nhP) {
if (h2 £ nhP) && (hP[h2+1] → d < hP[h2] → d)) h2++;
j = hP[h2];
if (j → d < dummy) {
hP[h1] = j;
j → q = h1;
h1 = h2;
}
}
} while (h1 == h2);
hP[h1] = x; x →q = h1;
```

Insert– reinsert

```
if (j → q) {
nhP++; hP[nhP] = j; j → q = nhP;
}
h1 = j → q; h2 = (h1 >> 1);
while ((h2) && (j → d < (x = hP[h2], x → d)) {
hp[h1] = x;
x → q = h1;
h1 = h2; h2 = (h1 >> 1);
}
hP[h1] = j; j →q = h1;
```

Fig-4 Binary heap manipulations in C.

$$Q[PNTR] = \begin{cases} J & \text{if } j \text{ is the first element of bucket PNTR.} \\ N+1 & \text{if bucket PNTR is empty.} \end{cases}$$

$$UP[J] = \begin{cases} 0 & \text{if } j \text{ is not in the priority queue.} \\ J & \text{if } i \text{ precedes } j \text{ in the same list.} \\ PNTR & \text{if } j \text{ is the first element of bucket PNTR.} \end{cases}$$

$$\begin{cases} 0 & \text{if } j \text{ is not in the priority queue.} \end{cases}$$

DOWN[J] =	*J*	*if i follows j in the same list.*
	N + 1	*if j is the last element of a bucket.*

Fig-5 Buckets – array implementation

```
Struct node {
Struct arc *a
Struct node *P, *down;
Union {
Struct node *up
Struct bucket *x,
} uval;
int flag;
int d;
};
Struct bucket {
Struct node *head;
};
```

Fig-6 Buckets – Pointer implementation

4. COMPUTATIONAL RESULTS

In this section, we describe the numerical experiments that have been conducted on networks based on graphs belonging to three distinct topological groups:

(i) One set consists of complete networks.
(ii) The second set of problems are random networks.
(iii) The last topological distinct group of problems consists of grid networks.

Complete Networks:

Since complete networks require an enormous amount of memory space, we restricted this study to small networks with 25, 75, 125 and 175 nodes, respectively. Results are summarized in Table-1.

Note that, for complete networks, Glover et al (1988) have not suggested any specific value for the parameter *P* used in their algorithm THRESH – *X2*. However, Gallo and Pallottino [8] tested different values and they concluded that by using *P* = 0.25, the algorithm LTHRS achieves its best performance. Gallao and Pallottino [8] use the variable *X2* to implement parameters *P*. Tests with complete graphs which comprise up to 1000 nodes (and 999,000 arcs) on a more powerful computer. We only report, the times achieved by using the pointer implementation since the relative order is the same with the two types of programs. Results are shown in Table-2 and they are illustrated in Fig. 10. Clearly, for very large complete networks, label setting algorithms SDIAL, SHEAP and SRHEAP exhibit the same performances.

This is surprising since SRHEAP has the best asymptotic computational complexity. Note that SBCKT behaves like the traditional implementation of Dijkstra's algorithm as the size of the network increases.

Random Networks:

Four classes of random network test problems were considered:

class A,	*n = 1000,*	*m = 10,000*
class B,	*n = 1000,*	*m = 30,000*
class C,	*n = 3000,*	*m = 10,000*
class D,	*n = 3000,*	*m = 30,000*

Table-3 shows the results we obtained on these random networks. We observe that

- LTHRS is the fastest code for all problems.
- With respect to the other methods shortest path algorithms generally dominate list search method. We therefore used this value in our tests. Regarding the performances of the two implementation of the algorithms on complete networks the following observations may be made:
- All list search algorithms (except LTHRS, which behaves as a label setting method) are dominated by shortest path algorithms. This is credited to the fact that very dense graphs have a high number of alternative paths between any two nodes. Furthermore, since, the arc lengths are integer values, the number of different lengths of these alternative paths increases with l_{max}.
- For small graphs, LTHRS is the fastest code dominating the large graphs.
- In all cases, the pointer implementation of an algorithm outperforms the array implementation of the same algorithm but efficiency is preserved by using the pointer implementation. The significance of the gain seems to increase with the size of the networks. Fig. 7 illustrates these results for the largest problem size we tested. Similar results have been observed for all other problems.
- SRHEAP is generally not the best algorithm. However, as the size of the network increases, its efficiency becomes comparable to the fastest ones.

  In order to verify the efficiency of the program SRHEAP for very large networks, we can take a second set of the range of the arc lengths affects the performance of the SBCKT algorithm. For limited length ranges that the algorithm performs poorly since, due to the high number significant of nodes in each bucket, it behaves very much like a classical implementation of Dijkstra's method.

- The fastest label setting algorithm is SDIAL and, remarkably, it is less affected by the value of l_{max} for denser random network problems than reported in the literature. This may be explained by the fact that we used a C implementation to mimic FORTRAN.
- SHEAP is faster than SRHEAP when the arc lengths take a wide range of values, SRHEAP is faster in the other case. Note that SRHEAP is never the fastest algorithm all of them.
- In all cases, the pointer implementation of an algorithm outperforms the array implementation of the same algorithm. Fig. 9 illustrates these results for the denser 1000 node networks.

Grid Networks:

We generated grid networks with 2500 nodes, with rectangularity's of 50×50, 25×100, 10×250 and 5×500. For grid networks the value of P is set at 1.5, as computed by Glover et al [10].

The results are shown in Table-4. Based on these results, the main conclusions concerning the relative efficiency of shortest path algorithms on grid networks are as follows:

- Three algorithms, L2QUE, LDEQUE and LTHRS, consistently display the best performance on this network topology class. This confirms the results of Gallo and Pallottino [8]. Note that, since the "two-way" criterion has been specifically designed for grid networks, it is not surprising to find L2QUE and LDEQUE among the most efficient methods.
- Once again, one finds that LTHRS is one of the most efficient procedures. It is worth noting that, at least for this network topology, this method behave rather similarly to SBCKT and SDIAL.
- The range of the arc length significantly affects the performance of the SBCKT algorithm. For limited length ranges the algorithm performs poorly due to the high number of nodes in each bucket. It also behaves similarly to the classical implementation of Dijkstra's method.
- SDIAL is also significantly affected by the range of the arc length. However, its behavior is exactly the opposite to that of SBCKT. Consequently, according to the arc lengths range, either SDIAL or SBCKT is the fastest label – setting algorithm. This may suggest the building principle for an improved label-setting procedure using buckets.
- SHEAP is always faster than SRHEAP and the gap increases when the maximum arc lengths increases.

- In all cases, the pointer implementation of an algorithm outperforms the array implementation of the same algorithm. Fig. 8 illustrates these results for the square 50×50 grid networks.

5. SUMMARY OF THE FINDINGS

We summarize the conclusions of our study for each algorithm as follows: SRHEAP employs buckets but, unlike SDIAL, the number of buckets used is very small for network with l_{max} = 10,000. For dense problems, its efficiency becomes comparable to that of the fastest procedures as the size of the network increases. However, even if it has the best theoretical efficiency, SRHEAP behaves poorly in practice for sparse network with large value of l_{max}. This behavior may be due to node redistributions, from bucket to bucket, required by the algorithm. This also suggests that implementation of algorithms which has two level re-distributive heaps at two levels may not perform any better, since they also require a considerable number of nodes redistributions from one sub-bucket to another.

It is now widely accepted that the threshold algorithms are dominant among shortest path procedures. In this study, we have also found that the LTHRS procedures is the fastest code over all network configurations. For this algorithm, we estimate that pointers accelerate it by a factor of at least 25%.

Algorithms LDEQUE and L2QUE display the same performance. For grid networks, they are to be preferred to LTHRS since they are fast but easier to implement. The versions coded in C by using pointers are about 25% faster than traditional implementations. For real problems, however L2QUE should be used, rather than LDEQUE, since it is polynomially bounded. Finally in dense problems, LQUEUE is faster than both LDEQUE and L2QUE, however, it is consistently dominated by a label setting algorithm.

Algorithm SHEAP is the one that takes the greatest advantage of the pointer implementations. This is explained by the relatively high number of array accesses needed to keep the heap property in SHEAP that is the faster label setting procedure and for the case, the speed up factor is about 33%.

6. CONCLUSIONS

This study is the first to use C programming language to evaluate the efficiency of shortest path algorithms, and it yields several interesting conclusions. It is also one of the first to perform an empirical study of the practical computational efficiency of the class of shortest path algorithms based on re-distributive heaps.

To the first question that prompted our research effort, we provided empirical evidence that the relative order of the shortest computational efficiency is not modified when the algorithms are coded in C. Additionally, the study documents that the re-distributive heap does not emerge as the best method in terms of actual computation efficiency in spite of the fact that it displays the best theoretical efficiency. 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%.

Finally, we believe that the implementations described in this paper, and the gains in algorithm efficiency that they generate, may not be obtained with any other commonly used high level programming language, even if they offer dynamic data structure and pointers (such as Pascal) since C provides unique models to efficiently manipulate computer memory addresses. 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.

7. ACKNOWLEDGEMENTS

The authors thanks anonymous referees for his useful comments and suggestion for improving the paper. The author is also thankful to Prof . H. C. Pande, Vice Chancellor Emeritus, Prof. S. K. Mukherjee, Vice-Chancellor Birla Institute of Technology, Mesra for encouragement and support.

TABLE-1 COMPUTATIONAL RESULTS – COMPLETE NETWORKS

Algorithms	1_{max} = 100				1_{max} = 10,000			
	n=25	*n*=75	*n*=125	*n*=175	*n*=25	*n*=75	*n*=125	*n*=175
L2QUE	10	79	274	497	12	100	378	748
	7	68	216	391	8	73	292	570
LDEQUE	9	86	297	517	11	105	389	779
	7	63	223	408	8	82	313	590
LQUEUE	7	64	190	378	11	78	251	480
	7	49	149	293	7	63	192	396
LTHRS	6	40	108	204	6	43	134	270
	5	32	89	160	3	29	86	215
SBCKT	7	57	153	279	7	49	124	267
	7	47	126	236	6	41	104	215
SDIAL	8	47	123	232	13	50	126	236
	6	36	91	171	13	39	96	177
SHEAP	8	50	132	249	8	53	134	250
	6	34	90	168	6	36	90	173
SRHEAP	9	39	123	228	12	56	126	245
	7	37	89	166	10	43	105	186

TABLE-2 EFFICIENCY IMPROVEMENTS POINTER IMPLEMENTATION (COMPLETE GRAPHS *n* = 175)

Algorithms	l_{max} = 100				l_{max} = 10,000			
	n=400	*n*=600	*n*=800	*n*=1000	*n*=400	*n*=600	*n*=800	*n*=1000
LQUEUE	685	1506	2584	3989	1093	2715	4998	7849
LTHRS	407	959	1674	2615	835	2146	3963	6325
SBCKT	483	1083	1915	3003	449	1052	1877	2954
SDIAL	386	866	1526	2385	390	870	1536	2401
SHEAP	398	877	1547	2400	400	896	1563	2432
SRHEAP	392	874	1542	2411	401	888	1559	2420

TABLE- 3 COMPUTATIONAL RESULT – RANDOM NETWORKS

Algorithms	I_{max}=100				I_{max} = 10,000			
	CLASS				*CLASS*			
	A	*B*	*C*	*D*	*A*	*B*	*C*	*D*
L2QUE	222	679	197	740	224	699	192	625
	138	496	167	537	152	567	187	556
LDEQUE	189	618	223	642	195	714	234	683
	130	473	146	482	147	572	168	555
LQUEUE	180	520	207	592	190	598	244	679
	135	397	167	451	142	429	159	466
LTHRS	115	242	197	333	114	258	211	374
	88	178	144	244	84	191	141	258
SBCKT	1794	2263	7356	14343	223	534	500	1188
	1410	1799	5776	11462	183	427	394	940
SDIAL	134	292	210	396	173	310	351	447
	107	212	173	316	151	242	329	368
SHEAP	238	396	559	746	237	393	571	744
	187	296	467	618	188	301	472	624
SRHEAP	183	316	425	581	304	426	831	914
	156	247	355	438	248	345	662	732

TABLE –4 COMPUTATIONAL RESULT GRID NETWORKS

Algorithms	l_{max} – 100				l_{max} = 10,000			
	p=50 *q*=50	*p*=25 *q*=100	*p*=10 *q*=250	*p*=5 *q*=500	*p*=50 *q*=50	*p*=25 *q*=100	*p*=10 *q*=250	*p*=5 *q*=500
L2QUE	176 165	169	157	172	184	171	162	
	128 125	128	120	129	137	121	122	
LDEQUE	318 592	1236	1216	357	615	1425	2211	
	241 450	958	1535	256	417	981	1584	
LTHRS	178 166	166	164	169	169	166	155	
	124 124	121	121	120	124	119	116	
SBCKT	913 590	359	287	234	233	225	208	
	717 502	271	229	192	176	182	184	
SDIAL	187 198	198	224	823	1070	2471	4144	
	163 165	169	191	771	1001	2372	3980	
	163 165	169	191	771	1001	2372	3980	
SHEAP	385 364	301	287	378	365	318	286	
	314 299	254	231	318	298	243	232	
SRHEAP	440 447	484	495	727	727	732	726	
	327 364	377	399	583	582	595	591	

REFERENCES

1. A. Aho, J. Hopecroft, and J. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, Mass (*1974*).

2. R. K Ahuja, K. Melhorn, J. B Orlin, and R. E. Tarjan, *Faster algorithms for the shortest path problem,* Technical Report No. 193, Operation Research Centre, M.I.T (*1988*).

3. R. Bellman, *On a routing problem.* Q Appl. Math 16, 88-90 (*1958*).

4. R.B. Dial, F. Glover, D. Karney, and D. Kilngman, *A computational analysis of alternative algorithms and labeling techniques for finding shortest path trees.* Networks, 9, 215-248 (*1979*).

5. R. B. Dial, *Algorithm 360 : Shortest path forest with topological ordering.* Communs. Ass. Comput. March. 12, 632-633 (*1969*).

6. E.W. Dijkstra, *A note of two problems in connection with graphs.* Num. Math. 1, 269-271 (*1959*).

7. M. L. Fredman, and R. E. Tarjan, *Fibonacci heaps and their uses in improve network optimization algorithms,* IEEE Found Comput. Sci. 338-357 (*1987*)

8. G. Gallo, and S. Pallottino: *Shortest path algorithms in FORTRAN codes for network optimization*. Ann Ops. Res. 13, 3-79 (*1988*).

9. F. Glover, D. Klingman, and N. Philips, *A new polynomials bounded shortest path algorithm,* Ops. Res. 33, 65-73 (*1985*).

10. F. Glover, D. Klingman and R. F. Schneider, *New Polynomial shortest path algorithms and their computational attributes.* Mgmt. Sci., 31, 1106-1128 (*1985*).

11. F. Glover, R. Glover, and D. Klingman, *Computational study of an improved shortest path algorithm,* Networks 14, 25-36 (*1984*).

12. M. Gondran, and M. Minoux, *Grapes et Algorithmes,* Editions Eyroless, Paris (*1985*).

13. M. S. Hung, and J. J. Dovoky, *A computational study of efficient shortest path algorithms,* Computers Ops. Res. 15, 567-576 (*1988*).

14. E. L. Johnson, *Shortest path on sorting,* In Proceedings of the 25th A.C.M. Annual Conference, pp. 510-517 (*1972*).

15. D. E, Knuth, *The Art of computer Programming Vol. I : Fundamental Algorithms,* 2nd Edn. Addison-Wesley, Reading Mass (*1990*).

16. P. K. Mishra, *A Study of data structure and algorithm analysis in C programming language,* M.Phil. thesis #1995, A.P.S. University, Rewa, INDIA, (*1995*).

17. J. F. Mondou, *Mice au point d'un prototype interact if- graphique pour la plannification tactique du transport des marchandises,* M.Sc. Thesis, publication #677, Centre de researche de recherche sar les transport, Universite de Montreal (*1989*).

18. E. F. Moore, *Shortest path through a maze,* In proceedings of the international symposium on theory of switching pp. 285-292, Harvard University Press, Cambridge, Mass (*1959*).

19. U. Pape, *Implementation and efficiency of Moore's algorithm for the shortest route problem,* Math Program. 7, 212 (*1974*).

20. P.E, Tarjan, *Data structure and network algorithms,* Society for Industrial and Applied Mathematics, Philadelphia, pa (*1983*).

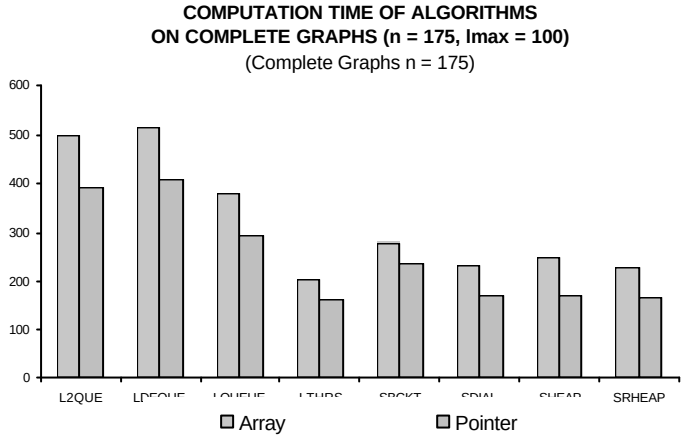

Figure-7

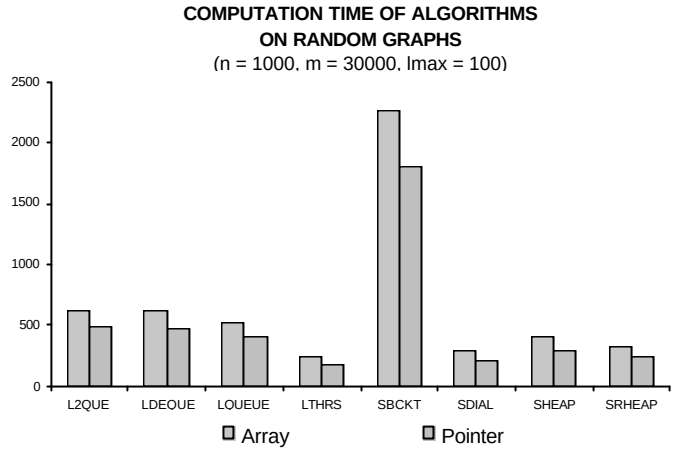

Figure-9

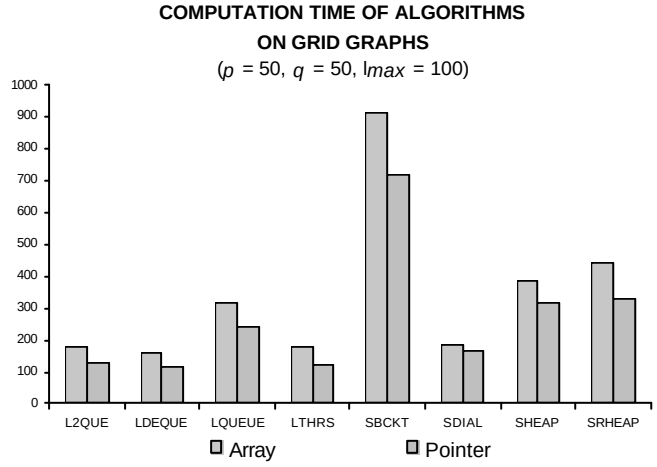

Figure-8

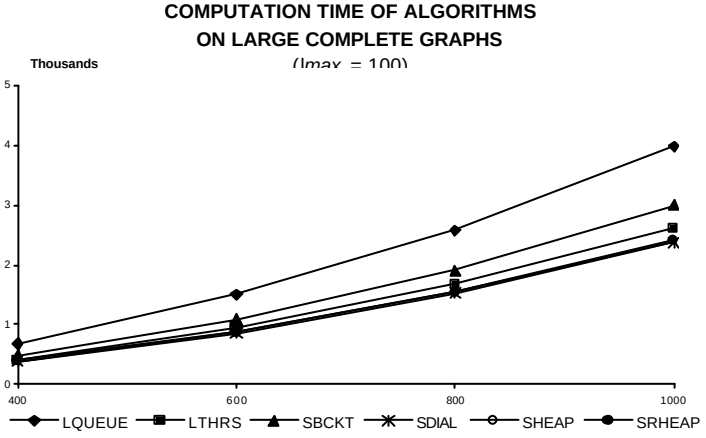

Figure-10