Finding Optimum Branchings

R. E. Tarjan

Stanford University
Stanford, California

ABSTRACT

Chu and Liu, Edmonds, and Bock have independently devised an efficient algorithm to find an optimum branching in a directed graph. We give an implementation of the algorithm which runs in $O(m \log n)$ time if the problem graph has n vertices and m edges. A modification for dense graphs gives a running time of $O(n^2)$. We also show that the unmodified algorithm runs in $O(n(\log n)^2 + m)$ time on an average graph, assuming a uniform probability distribution.

1. INTRODUCTION

A directed graph $G = (V, E)$ consists of a set of *vertices* V of size $|V| = n$ and a set of *edges* E of size $|E| = m$. Each edge is an ordered pair (v, w) of distinct vertices v, w, called the *endpoints* of the edge. A *semipath* joining x_1 and x_k in G is a sequence of vertices $x_1, x_2, \ldots, x_k$ such that $(x_i, x_{i+1}) \in E$ or $(x_{i+1}, x_i) \in E$ for each i, $1 \leq i \leq k$. A set of vertices $W \subseteq V$ is *weakly connected* if there is a semipath of vertices in W joining any pair of vertices in W. A maximal weakly connected set of vertices is a *weakly connected component* of G.

A *path* from x_1 to x_k in G is a sequence of edges $(x_1, x_2)(x_2, x_3) \ldots (x_{k-1}, x_k)$. The path is *simple* if $x_1, \ldots, x_{k-1}$ are distinct. The path is a *cycle* if $x_1 = x_k$. A set of vertices $S \subseteq V$ is *strongly connected* if there is a path whose edges have endpoints in S from any vertex in S to any other vertex in S. A maximal strongly connected set is a *strongly connected component* of G.

A *branching* B of G is a set of edges such that

(i)   if (x_1,y_1), (x_2,y_2) are distinct edges of B then $y_1 \neq y_2$;

(ii)   B does not contain a cycle.

Given a real value c(v,w) defined for each edge of G, we desire a branching B such that $\sum_{(v,w)\,\epsilon B} c(v,w)$ is maximum.   Such a set B is called an *optimum branching*.

   Chu and Liu [3], Edmonds [5], and Bock [1] have independently given efficient algorithms for finding an optimum branching in a graph G.   The Chu – Liu and Edmonds algorithms are virtually identical; the Bock algorithm is similar but stated as an algorithm on matrices rather than on graphs.   This paper gives an efficient implementation of the Chu – Liu – Edmonds algorithm. If G has n vertices and m edges the algorithm runs in $0(m \log n)$ time.   A modification for dense graphs gives an $0(n^2)$ running time.   Also, assuming a uniform probability measure, the algorithm runs in $0(n(\log n)^2 + m)$ time on the average.   These time bounds assume $m \geq kn$ for some constant k.   (This is not a serious restriction; any weakly connected graph has $m \geq n-1$.)

2.   AN ALGORITHM FOR OPTIMUM BRANCHINGS

   Let G = (V,E) be a weakly connected directed graph, having edge values c(v,w).   The Chu – Liu – Edmonds algorithm finds an optimum branching in G.   The algorithm constructs a set of edges H defining a subgraph G(H) = (V,H).   At all times during the construction, G(H) is such that for each strongly connected component S, there is at most one edge (v,w) ϵ H such that w ϵ S, v ϵ V-S.   If S is such that no edge (v,w) ϵ H satisfies w ϵ S, v ϵ V-S we call S a *root component* of G(H).

   Initially H = $\emptyset$; thus initially each vertex in V defines both a weakly connected and a strongly connected component of G(H).   Here is the algorithm for constructing the set H.

Algorithm BRANCH:

   First Step:

        F1:   Choose any vertex v with an edge (x,v) of value
              c(x,v) > 0.
        F2:   Select the edge (u,v) of largest value.
        F3:   Add (u,v) to H.

General Step:

G1: Choose any root component S of G(H) having an unexamined edge (x,v) with v ε S and c(x,v) > 0.

G2: Find the largest unexamined edge (u,v) such that v ε S.

G3: If u ε S, discard the edge and stop. Otherwise go to G4.

G4: u ∉ S. Let W be the weakly connected component of G(H) containing v. If u ∉ W add (u,v) to H and stop. Otherwise go to G5.

G5: u ∉ S, u ε W. Find the sequence $S_1, (x_1, y_1), S_2, (x_2, y_2), \ldots, S_k, (x_k, y_k)$ such that each S_i is a strongly connected component of G(H), (x_i, y_i) ε H, y_i ε S_i, and x_i ε S_{i+1} for all i, $S_k = S$, $(x_k, y_k) = (u,v)$, and x_k ε S_1.

G6: Find the edge (x_j, y_j) with minimum value among the (x_i, y_i)

G7: For each unexamined edge (x,y) with y ε S_i, modify the value of (x,y) as follows:

$$c(x,y) := c(x,y) - c(x_i, y_i) + c(x_j, y_j).$$

G8: Add (u,v) to H. (This combines $S_1, \ldots, S_k$ into a single strongly connected component which is a root component of G(H).)

Repeat the general step until there is no root component S of G(H) having an unexamined edge (u,v) with v ε S and c(u,v) > 0.

Suppose this algorithm is applied to a graph G. The following results are implicit in the work of Edmonds and Karp [5,8]. Let G(H) = (V,H).

Lemma 1: Each strongly connected component S of G(H) has at most one edge (u,v) ε H with v ε S, u ε V-S. Each weakly connected component W of G(H) contains exactly one root component.

Lemma 2: Let S be any root component of G(H), let W be the weakly connected component containing S, and let v ε S. Then for any w ε W there is a unique simple path in G(H) from v to w.

These lemmas follow easily by induction on the number of edges added to H.   When the algorithm is completed, H contains an optimum branching, which we can extract with the help of Lemma 2.   All we need to do is determine a unique vertex v in the root component of each weakly connected component of G(H); these vertices determine a branching consisting of the simple paths given by Lemma 2.   If we determine these vertices using the following algorithm, the resultant branching is optimum [5,8].

Algorithm ROOT:

   R1:   Find a root component R in G(H) containing more than
          one vertex.
   R2:   Find the sequence $S_1, (x_1,y_1) \ldots S_k, (x_k,y_k)$ determined
          in BRANCH, Step G5, such that edge (x_k,y_k) was added
          to H to form R.
   R3:   Find the edge (x_j,y_j) of minimum value among the
          (x_i,y_i).
   R4:   Delete (x_j,y_j) from H (this step makes S_j a root
          component).

Repeat Steps R1 – R4 until every root component consists of a single vertex.

If BRANCH keeps track of each minimum edge (x_j,y_j) found in Step G6 and of each strongly connected component formed in Step G8, then we do not need to actually carry out ROOT; BRANCH can explicitly keep track of the root vertex which would be returned by ROOT for any given root component.   Our implementation uses this observation.

3.   EFFICIENT IMPLEMENTATION

We need several bookkeeping mechanisms to implement BRANCH efficiently.   We must keep track of (a)   the weakly connected components of G(H), (b)   the strongly connected components of G(H), and (c)   the unexamined edges entering each strongly connected component.   For (a) and (b) we use a disjoint set union algorithm described in [7,13].   Given a collection of disjoint sets, the set union algorithm implements two operations:

(i)   FIND(x)   returns the name of the set containing element x;

(ii) UNION(A,B)   adds the elements of set A to set B, destroying set B.

The time required for $O(m)$ FINDs and $O(n)$ UNIONs is $O(n \log^* n+m)$ [13], where $\log^* n = \min\{i \mid \underbrace{\log \log \ldots \log n \leq 1}_{i \text{ times}}\}$. We use one collection of sets, manipulated by WFIND and WUNION, to represent the weakly connected components, and another collection of sets, manipulated by SFIND and SUNION, to represent the strongly connected components.

To keep track of the edges entering each strongly connected component, we use a priority queue mechanism described in [2,9]. Given a collection of elements, each with a value, and a collection of disjoint sets (called queues) of elements, the mechanism implements four operations:

(iii) QUNION(C,D)   adds the elements in queue D to queue C.
Time required: $O(\log |C| + \log |D|)$.

(iv)   MAX(C)   returns the largest element in queue C, deleting this element from the queue. (If C is empty, MAX(C) returns a dummy element with value $-\infty$.)
Time required: $O(\log |C|)$.

(v)   INIT(C,L)   initializes a queue C to contain all elements in the list L.
Time required: $O(|L|)$.

(vi)   ADD(a,C)   adds a constant a to the value of all elements in queue C.
Time required: $O(1)$.

Here is an implementation of the optimum branching algorithm, expressed in Algol-like notation. In the program, the set *roots* gives the root components of G(H) which may have entering edges of positive value. The set *rset* gives the root components of G(H) with no entering edges of positive value. For each i, *enter*(i) gives the unique edge in H entering strongly connected component i. (If there is no such entering edge, *enter*(i) = (0,0).)

The array *min*(i) gives, for each i, the root vertex which algorithm ROOT will select if applied to root component i. The variable *val* and *vertex* are used to select and record the minimum edge found in Step G6. The algorithm assumes that the graph G = (V,E) has vertex set V = {1,2,...,n}, and that, for each j, I(j) = {(i,j) ϵ E} is an incidence list for vertex j.

```
algorithm BRANCH:
  begin
    roots := ∅;
    for i := 1 until n do
        begin
            INIT(i,I(i));
            initialize an S-set named i containing i as its
            only element;
            initialize a W-set named i containing i as its
            only element;
            enter(i) := (0,0);
            roots := roots ∪ {i};
            min(i) := i;
        end;
    H := ∅;
    rset := ∅;
    while roots ≠ ∅ do
        begin
            delete some entry k from roots;
            (i,j) := MAX(k);
            if v(i,j) < 0 then rset := rset ∪ {k}
            else if SFIND(i) = k then roots := roots ∪ {k}
                else begin
                    H := H ∪ {(i,j)};
                    if WFIND(i) ≠ WFIND(j) then
                        begin
                            WUNION(WFIND(i),WFIND(j));
                            enter(k) := (i,j);
                        end
                    else
                        begin
                            val := ∞;
                            (x,y) := (i,j);
                            while (x,y) ≠ (0,0) do
                                begin
                                    if c(x,y) < val then
                                        begin
                                            val := c(x,y);
                                            vertex := SFIND(y);
                                        end;
                                    (x,y) := enter(SFIND(x));
                                end;
                            ADD(val - c(i,j),k);
                            min(k) := min(vertex);
                            (x,y) := enter(SFIND(i));
                            while (x,y) ≠ (0,0) do
```

$begin$
ADD(val - c(x,y),SFIND(y));
QUNION(k,SFIND(y));
SUNION(k,SFIND(y));
(x,y) := $enter$(SFIND(x));
end;
$roots$:= $roots$ $\cup$ {k};
$end\ end\ end\ end$ BRANCH;

After BRANCH is executed, the set of vertices
R = {min(i) | i ε $rset$} defines an optimum branching, which may
be found in O(n+m) time by using a search [12] to find the set
of simple paths from the vertices in R to all other vertices.
The operations in algorithm BRANCH are of three types:

(a)   priority queue operations;
(b)   disjoint set operations;
(c)   other operations.

The priority queue operations required are: O(n) INIT op-
erations, one for each vertex; O(m) MAX operations, O(n) QUNION
operations, and O(n) ADD operations. Thus the total time for the
priority queue operations is O(m log n). The disjoint set opera-
tions required are: O(m) WFINDs, O(m) SFINDs, O(n) WUNIONs, and
O(n) SUNIONs (the set H contains at most 2n-2 edges). Thus

the total time for the set operations is O(n log* n+m). The
total time for other operations is clearly O(n+m). Thus the
total time required to find an optimum branching is O(m log n)
(assuming m $\geq$ kn for some constant k). The space required is
O(m).

4.   A MODIFICATION FOR DENSE GRAPHS

The time bound given in Section 2 for the optimum branch-
ing algorithm is O(n^2 log n) if the graph G is dense; i.e., m
is $\Omega(n^2)$. However, by changing the priority queue representa-
tion, we can get the algorithm to run in O(n^2) time. This
modification is analogous to that used in a well-known O(n^2)
minimum spanning tree algorithm [4,10].
We use a different mechanism to represent the priority
queues. For each strongly connected component S, we have a
list of edges (i,j), at most one for each i, such that i $\notin$ S,
j ε S, and for all edges (i,k) such that k ε S, c(i,k) $\leq$ c(i,j).
This list is ordered on i, and takes the place of the priority
queue for component S. Associated with each edge (i,j) in the
list is its current value c(i,j).

Since such a list has length at most n, a MAX operation requires $O(n)$ time. Furthermore, since every edge (i,j) on such a list has $i \notin S$, algorithm BRANCH executes only $O(n)$ MAX operations, and $O(n^2)$ time total is required for the MAX operations.

A QUNION(C,D) operation requires $O(n)$ time plus time for $O(n)$ SFINDs. Thus the total time for the QUNION operations is $O(n^2)$. ($O(n^2)$ SFINDs plus $O(n)$ SUNIONs require $O(n^2)$ time [13].) An INIT($|L|$) operation requires $O(n)$ time since $|L| \leq n$ (the edges $(i,j) \in L$ may be sorted on the value of i in $O(|L| + n)$ time by using a radix sort [9]). Thus the total time for the INIT operations is $O(n^2)$, and the running time of the modified version of BRANCH is $O(n^2)$. The space required is still $O(m)$.

5. AVERAGE TIME ANALYSIS

In this section we show that the average running time of the unmodified version of BRANCH is $O(n(\log n)^2 + m)$ assuming a uniform probability measure. The analysis is analogous to that used in [11] to show the existence of an all-pairs shortest path algorithm with an average running time of $O((n \log n)^2)$.

Let G be selected according to some probability distribution from among the $\binom{n \cdot (n-1)}{m}$ labelled graphs having vertex set $\{1,2,\ldots,n\}$ and m edges. Assume that, for all j, the probability of the existence of an edge (i,j) in G is the same for each i. For each j, $1 \leq j \leq n$, let p_j be a real-valued probability distribution, and let the values of the edges (i,j) in G be independent variables with probability distribution p_j.

With this definition of a random graph, the probability that there exists an edge $(i,j) \in E$ and that (i,j) has maximum value among edges in the set $\{(k,j) \mid (k,j) \in E\}$ is the same for each i. Furthermore, at any time during the execution of BRANCH, the probability that there exists an edge $(i,j) \in E$ and that (i,j) has maximum value among unexamined edges in the set $\{(k,j) \mid (k,j) \in E\}$ is the same for each i such that (i,j) has not yet been returned by a call on MAX. This follows from the fact that the only information known about (i,j) is that if (i,j) is an edge in G then its value must be greater than that of all edges returned by previous MAX calls on queues corresponding to root components containing vertex j; and this information is independent of i.

We now derive an upper bound on the average number of MAX calls used by BRANCH and hence on the average running time of BRANCH. Suppose MAX(k) is executed for some root component k containing m vertices. Then the probability of MAX(k) returning an edge (i,j) with i outside component k is at least $\frac{n-m}{n}$. Thus an upper bound on $\bar{M}_k$, the average number of MAX(k) operations before a new edge is added to H, is

$$\bar{M}_k \leq \left(\frac{n-m}{n}\right) \sum_{i=1}^{\infty} i\left(\frac{m}{n}\right)^{i-1} = \frac{n}{n-m} .$$

The maximum number of edges which can be added to H after some root component reaches size m is 2(n-m). It follows that an upper bound on $\bar{M}$, the average total number of executions of Max, is

$$\bar{M} = \Sigma \bar{M}_k \leq \sum_{m=1}^{n-1} \frac{2n}{n-m} = O(n \log n)$$

and the average time required by all the MAX operations is $O(n(\log n)^2)$. Combining this with the worst case time bounds on the other parts of the algorithm gives an $O(n(\log n)^2 + m)$ bound on the average running time of the algorithm.

We also have

$$var(M_k) \leq \overline{2M_k^2} \leq \frac{n-m}{n} \sum_{i=1}^{\infty} i^2\left(\frac{n}{m}\right)^{i-1} \leq \frac{2n^2}{(n-m)^2} ,$$

and

$$var(M) \leq \sum_{m=1}^{n-1} \frac{4n^2}{(n-m)^2} = O(n^2).$$

Thus the standard deviation of M is $O(n)$, and the probability that the number of executions M of MAX is greater than $\bar{M} + kn$ is $O(1/k^2)$. Thus the probability that the running time of BRANCH is not $O(n(\log n)^2 + m)$ is $O\left(\frac{1}{(\log n)^2}\right)$.

6.   REMARKS

We have presented an implementation of Edmonds' optimum branching algorithm which has a worst-case running time of $O(m \log n)$, an average running time of $O(n (\log n)^2 + m)$, and can be modified to run in $O(n^2)$ time, which is an improvement if the graph is dense. One naturally wonders if these results are best possible. Recently discovered algorithms for finding minimum spanning trees in undirected graphs achieve a time bound of $O(m \log \log n)$ [2,14]; they can be used to find minimum spanning trees in $O(n \log \log n + m)$ time on the average, using results in [6]. These results suggest the following research questions:

Can optimum branchings be found in $o(m \log n)$ time?

If not, how can one prove that *any* optimum branching algorithm requires $\Omega(m \log n)$ time in the worst case?

REFERENCES

1.   Bock, F., "An Algorithm to Construct a Minimum Directed Spanning Tree in a Directed Network," *Developments in Operations Research*, Gordon and Breach, New York, 1971, pp. 29-44.

2.   Cheriton, D. and R. Tarjan, "Finding Minimum Spanning Trees," *SIAM J. Comput.*, 5, 1976, pp. 724-742.

3.   Chu, Y. J. and T. H. Liu, "On the Shortest Arborescence of a Directed Graph," *Sci. Sinica*, 14, 1965, pp. 1396-1400.

4.   Dijkstra, E. W., "A Note on Two Problems in Connection with Graphs," *Numerische Mathematik*, 1, 1959, pp. 269-271.

5.   Edmonds, J., "Optimum Branchings," *Jour. of Research of the National Bureau of Standards*, 71B, 1967, pp. 233-240.

6.   Erdös, P. and A. Rényi, "On the Evolution of Random Graphs," *Pub. of the Math. Inst. of the Hung. Acad. of Sciences*, 5, 1960, pp. 17-60.

7.   Hopcroft, J. and J. D. Ullman, "Set-Merging Algorithms," *SIAM J. Comput.*, 2, 1973, pp. 294-303.

8.   Karp, R. M., "A Simple Derivation of Edmonds' Algorithm for Optimum Branchings," *Networks*, 1, 1971, pp. 265-272.

9. Knuth, D., *The Art of Computer Programming, Vol. 3: Sorting and Searching,* Addison-Wesley, Reading, Massachusetts, 1973.

10. Prim, R. C., "Shortest Connection Networks and Some Generalizations," *Bell System Tech. J.,* 1957, pp. 1389-1401.

11. Spira, P., "A New Algorithm for Finding All Shortest Paths in a Graph of Positive Arcs in Average Time $O(n^2 \log^2 n)$," *SIAM J. Comput.,* 2, 1973, pp. 28-32.

12. Tarjan, R., "Depth-First Search and Linear Graph Algorithms," *SIAM J. Comput.,* 1, 1972, pp. 146-160.

13. Tarjan, R., "Efficiency of a Good but Not Linear Set Union Algorithm," *J. ACM,* 1975, pp. 215-225.

14. Yao, A., "An $O(|E| \log \log |V|)$ Algorithm for Finding Minimum Spanning Trees," *Info. Proc. Letters,* 4, 1975, pp. 21-23.