Complete Development of An Algorithm

A Complete Example

Example

The task:

Decision of whether to establish a computer network at some different sites;

Factors:

- Computing resources available at each site;
- Anticipate cite usage levels;
- Peak demands on the system;
- Possible system degradation of the major facility;
- ……
- The cost of the proposed network.

Example

The task:

Decision of whether to establish a computer network at some different sites;

Factors:

- Computing resources available at each site;
- Anticipate cite usage levels;
- Peak demands on the system;
- Possible system degradation of the major facility;
- ……
- The cost of the proposed network.

The Cost of the Proposed Network

This cost includes:

- Equipment purchases ;
- Establishment of communication links ;
- Systems maintenance;
- The costs of running a job of a given type at a given site.

The Cost of the Proposed Network

This cost includes:

- Equipment purchases ;
- Establishment of communication links ;
- Systems maintenance;
- The costs of running a job of a given type at a given site.

Analyze the Problem

Leased line costs:

- – Geographical distance between the sites;
- – The desired transmission rate;
- – The desired transmission capacity on a line.

After discussion, have

C_{ij} between sites i and j.

----a symmetric cost matrix

Complete Development of an Algo.

- 1 Statement of the problem
- 2 Development of a Model
- 3 Design of the algorithm
- 4 Correctness of the algorithm
- 5 Implementation
- 6 Analysis and complexity of the algorithm
- 7 Program testing
- 8 Documentation

1. Statement of the problem

- To establish a minimum cost communication network,

  in which any site can communicate with each other,

  while the cost of building a link between any two cites are given.

1. Statement of the problem

已知网络中任意两点间建立链路的花费，

需建立一个最小费用的通讯网络,

其中任意节点间可以相互通讯。

Complete Development of an Algo.

- 1 Statement of the problem
- 2 Development of a Model
- 3 Design of the algorithm
- 4 Correctness of the algorithm
- 5 Implementation
- 6 Analysis and complexity of the algorithm
- 7 Program testing
- 8 Documentation

Modeling

- What constitutes a solution to the problem?
  - Must decide which specific links should be established and which ones should not, that is:

    for every site pair *i* and *j*, whether or not establish a link at cost C_{ij} .

A solution will consist of a modified matrix C' of the original cost matrix C in which

- the (i, j) and (j, i) entries equal ∞ if we do not establish the link;
- the entries equal C_{ij} if we do.

Modeling

- Also, since every site is supposed to be able to communicate with every other site in the final network, each row and each column of C′ will have at lease one finite entry.

- Is this a correct model?
  - Will any modified matrix satisfying this condition on the rows and columns be a solution?

Look at white board Pls.

More Constraints

- What this illustrates is that a network corresponding to a solution to the problem will:
  - Must be connected
  - Must not contain a cycle of communication links.

If a solution were to contain a cycle, then a cheaper solution could be found by removing the most costly link in the cycle. Therefore, a solution to the problem will consist of a cheapest sub-network which is connected, contains no cycles, and contains every vertex.

The solution is a spanning tree!

Mathematically

- Our original problem can now be stated mathematically as follows:

  Given a weighted, connected network G, find a minimum-cost (or weight) spanning tree of G.

  ----MCST, MWST, MST

Model

Let

G=(V,E) be a connected, undirected graph;

w(u,v) is the cost for connecting *u,v*$\in$*V*;

find an acyclic subset T $\subseteq$ E and connects all vertices in *V*, such that

$$w(T) = \sum_{(u,v)\in T} w(u,v)$$ is minimized

Complete Development of an Algo.

- 1 Statement of the problem
- 2 Development of a Model
- 3 Design of the algorithm
- 4 Correctness of the algorithm
- 5 Implementation
- 6 Analysis and complexity of the algorithm
- 7 Program testing
- 8 Documentation

Designing an Algorithm,

- Perhaps the most natural thing to do is to try a greedy algorithm—

  select the cheapest edge first, then the next cheapest edge, and so forth.

  But in selecting edges we must keep in mind our three requirements:

  (1) the final subnetwork must contain all vertices and must be connected;

  (2) the final network must not contain any cycles;

  (3) the final network must have the minimum possible weight.

- **Algorithm A** To find a minimum-weight spanning tree T in a weighted, connected network G with N vertices and M edges.

```
Step 0.   [Initialize] **Set** T ← a network consisting of N
                    vertices and no edges;
                    set   H ← G.
Step 1.   [Iterate] While T is not a connected network
                    do through step 3 od; STOP.
Step 2.   [Pick a lightest edge] Let (U,V) be a lightest
          (cheapest) edge in H;
          if T + (U,V)  has no cycles
          then [add (U,V) to T] set T ← T + (U,V) fi.
Step 3.   [Delete (U, V) from H] Set H ← H – (U, V).
```

Does it work?

- There are several questions we should ask about this algorithm;

1. Does it always STOP?
2. When it STOPs, is T always a spanning tree of G?
3. Is T guaranteed to be a minimum-weight spanning tree?
4. Is it self-contained (or does it contain hidden, or implicit, sub-algorithms)?
5. Is it efficient?

Question 4&5

- Step 1 requires a sub-algorithm to determine if a network is connected, and
- step 2 requires a sub-algorithm to decide if a network has a cycle.
- These two steps can make the algorithm ineffective in that these sub-algorithms might be time-consuming. (how?)

More Advanced

- This suggests that we might improve Algorithm A if we could discover a method of constructing a spanning tree which guarantees, and

  without having to check,

  that a connected network with no cycles is created.

How?

- Edges are added to the partial solution one by one.

Want no checking?

May observe when adding!

- When adding an edge:
  - Guarantee connected
  - Guarantee no cycle

How?

- Edges are added to the partial solution one by one.

Want no checking?

May observe when adding!

- When adding an edge:
  - Guarantee connected---one end in the partial solution
  - Guarantee no cycle---the other end not in the partial solution

- **Algorithm B** To find a minimum-weight spanning tree T in a weighted, connected network G with N vertices and M edges.

```
*Step 0.*  [Initialize] Label all vertices “unchosen”;
           **set** T $\leftarrow$ a network with N vertices and
           no edges; choose an arbitrary vertex
           and label it “chosen”.
*Step 1.*  [Iterate] **While** there is an unchosen
                 vertex **do** step 2 **od**; STOP.
*Step 2.*  [Pick a lightest edge]
   Let (U, V) be a lightest edge between any
   chosen vertex U and any unchosen vertex V;
   label V as “chosen”; **and set** T $\leftarrow$ T + (U, V).
```

Now

- Algorithm B is self-contained;
- It is assumed to be more efficient than Algorithm A.

Does it work?

- There are several questions we should ask about this algorithm;

1. Does it always STOP?
2. When it STOPs, is T always a spanning tree of G?
3. Is T guaranteed to be a minimum-weight spanning tree?
4. Is it self-contained (or does it contain hidden, or implicit, sub-algorithms)?
5. Is it efficient?

Complete Development of an Algo.

- 1 Statement of the problem
- 2 Development of a Model
- 3 Design of the algorithm
- 4 Correctness of the algorithm
- 5 Implementation
- 6 Analysis and complexity of the algorithm
- 7 Program testing
- 8 Documentation

Correctness of Algorithm B

- Proof of the correctness of Algorithm B consists of the following sequence of theorems:

Theorem 1

At the completion of every execution of step 2 of Algorithm B,

the edges currently in T form a tree among the set of currently chosen vertices.

Proof *We proceed by induction on the number of times K that step 2 has been executed.*

- if K = 1, then there will be two chosen vertices and one edge in T between them.
- Assume that at the end of K executions of step 2, the edges currently in T form a tree, call it Tc, among the currently chosen vertices.
- Consider the (K + 1)st execution of step 2:
- This essentially involves adding one new vertex and one new edge to Tc.
- Since this new edge is the only edge between the new vertex and Tc,

  the addition of this vertex and edge cannot create a cycle in Tc and leaves the new Tc connected.
- Thus Tc is still a tree.

证明 按*step 2* 执行的次数K来推导

- 如果 K = 1, 那么会有两个已选择节点和T中连接它们的一条边。
- 设 在step 2 第K次执行结束时, 当前T中的边构成的在当前已选节点上的树记为Tc。
- 考虑step 2的第(K + 1)次执行:
- 这将是往Tc添加一个新节点和一条新边。 由于这条新边是新节点和Tc之间唯一的边,

  该节点和边的添加不会在 Tc 中产生回路, 且能使这个新 Tc 连通.
- 这样, Tc 仍旧是一棵树.

- **Algorithm B** To find a minimum-weight spanning tree T in a weighted, connected network G with N vertices and M edges.

Step 0. [Initialize] Label all vertices "unchosen"; **set** T $\leftarrow$ a network with N vertices and no edges; choose an arbitrary vertex and label it "chosen".

Step 1. [Iterate] **While** there is an unchosen vertex **do** step 2 **od**; STOP.

Step 2. [Pick a lightest edge]

Let (U, V) be a lightest edge between any chosen vertex U and any unchosen vertex V; label V as "chosen"; **and set** T $\leftarrow$ T + (U, V).

Theorem 2

At the end of Algorithm B, T is a spanning tree of G.

Proof This now follows immediately from Theorem 1,

which assures us that T is a tree,

and the fact that T contains every vertex of G----that is,

the algorithm stops only when all vertices are chosen.

Theorem 3

Let T be a sub-tree of a network G and let e be a lightest edge between a vertex in T and a vertex not in T.

Then there is a spanning tree T′ in G which contains T and e such that

if T″ is any spanning tree of G that properly contains T,

then W(T′) ≤ W(T″).

定理 3

记 T 是网络 G的一个 子树，且令 e 为T中一个节点和不在T中的一个节点之间的最轻的边。

则存在G中包含 T 和 e的生成树 T′，

使得，

如果 T″ 是G的任意一个包含T的生成树，那么有 $W(T') \le W(T'')$.

Proof Let T'' be a spanning tree of G which contains T and has minimum weight among all spanning trees which contain T.

Suppose further that T'' does not contain *e*.

Consider the network $T'' \cup \{e\}$ and the unique cycle C of $T'' \cup \{e\}$ which contains *e*.

Now C must contain an edge $f \neq e$ which joins a vertex of T to a vertex not in T.

By our assumptions, however, we know that $w(e) \leq w(f)$.

Therefore, the tree obtained from T'' by deleting *f* and adding *e*,

$$(T'' - \{f\}) \cup \{e\} = T'$$

Is a spanning tree of G which contains T and *e* and satisfies $W(T') \leq W(T'')$.

证明　　　令 T″ 是G的任意一个包含T的生成树， 且是所有包含T的生成树中权值最小的。
进一步假设 T″ 不包括*e*.

考虑网络 T″ ∪ {*e*} 和T″ ∪ {*e*}中唯一的环 C （C 包含 *e* ）

C 一定包括了一条边 $f \neq e$ ， 且*f* 连接了T中一个节点和不在T中的一个节点。

然而，通过我们的假设, 有$w(e) \leq w(f)$.

因此, 通过删除*f* 和添加e 从T″ 得到的树

$$(T'' - \{f\}) \cup \{e\} = T'$$

是 G的一个包含T和*e* 且满足 W(T′) ≤ W(T″)的生成树。

Why Theorem 3?

- Theorem 3 essentially provides the justification for step 2 of Algorithm B;

- It asserts that the basis for choosing a new edge in step 2 is sufficient to guarantee that there will exist a minimum-weight spanning tree of G which contains the chosen edge.

Theorem 4 Let G = (V, E) be a weighted, connected network, and let $e = (v, w)$ be a lightest edge incident to a vertex v. Then there exists an MST T which contains e.

Proof Let T be an MST of G which does not contain e. Consider the network $T \cup \{e\}$.

$T \cup \{e\}$ contains exactly one cycle. This cycle contains two edges incident to vertex v, including $e = (v, w)$ and another edge, say, $f = (u, v)$.

By the hypothesis, $w(e) \leq w(f)$, and thus

$(T - \{f\}) \cup \{e\}$ is an MST of G which contains e.

***

Theorem 5 Algorithm B finds an MST T in a weighted, connected network G with N vertices.

Proof

Theorem 2 ----- Algorithm B finds a spanning tree.

Theorem 4 ---- there exists an MST T1, which contains the first edge (called *e*1)chosen by Algorithm B.

Theorem 3 ---- there is a spanning tree T2 which contains the first edge *e*1 and second edge *e*2 chosen by Algorithm B. T2 also has a minimum weight among all spanning trees which contain *e*1; that is, W(T2) = W(T1) since T1 is our MST.

By iterating this process, Theorem 3 guarantees the existence of an MST which contains the first *k* edges chosen by Algorithm B, for all *k* = 1, 2,…, N – 1.

定理 5　　算法 B 得到一个有权重的、连通的、含N个节点的网络G的 MST T.

证明

定理 2 -----算法 B 得到一个生成树。

定理 4 ---- 存在一个 G的MST T1, 包含算法B选择的第一条边 (称为 *e*1).

定理 3 ---- 存在一个G的生成树 T2 包含算法 B选择的第一条边 *e*1 和第二条边 *e*2. T2 也是所有包含 *e*1的生成树中权值最小的; 即, W(T2) = W(T1), 因 T1 就是G的MST.

通过重复上述过程, 定理 3 保证了包含算法B选择的前k条边的 MST的存在，　*k* = 1, 2,…, N – 1.

$W(T_{N-1}) = W(T_1)$

Complete Development of an Algo.

- 1 Statement of the problem
- 2 Development of a Model
- 3 Design of the algorithm
- 4 Correctness of the algorithm
- 5 Implementation
- 6 Analysis and complexity of the algorithm
- 7 Program testing
- 8 Documentation

Implementation of Algorithm B

- The first question that might be asked is:

  What should be the input and output parameters for this subroutine?

  (Data Structures)

  Input: a weighted, connected network G with N vertices and M edges.

  NxN array C (cost matrix C[i,j]).

  Output: FROM(i),TO(i),COST(i). i^{th} edge

Other Data Structures

- Step 0 and Step 1 need a flag for each vertex .
- Step 2 may need two arrays CHOSEN and UNCHOSEN.
  - See any problem?

- **Algorithm B** To find a minimum-weight spanning tree T in a weighted, connected network G with N vertices and M edges.

Step 0. [Initialize] Label all vertices "unchosen"; **set** T $\leftarrow$ a network with N vertices and no edges; choose an arbitrary vertex and label it "chosen".

Step 1. [Iterate] **While** there is an unchosen vertex **do** step 2 **od**; STOP.

Step 2. [Pick a lightest edge]

Let (U, V) be a lightest edge between any chosen vertex U and any unchosen vertex V; label V as "chosen"; **and set** T $\leftarrow$ T + (U, V).

The problem is

- Step 2 may be too time consuming
- See the following chart

Flow Chart of Algorithm B

1. Start
2. Initialization
3. **IF** there is no unchosen vertex left **Then** Stop
4. Pick a lightest edge e=(u,v) between a chosen vertex U and an unchosen vertex V;
5. Label vertex V as chosen;
6. Add edge e to T;
7. **GOTO** 3.

How would you estimate the complexity of 4?

A Proposed Technique

A more efficient implementation of step 2 is obtained by:

- keeping only a list of un-chosen vertices:
  - UNCHSN (I), which initially contains NUMUN = N – 1 vertices,
- creating two additional arrays:
  - LIGHT(I), which equals the value of the lightest edge between the i^{th} unchosen vertex and a chosen vertex, and
  - VERTEX(I), which contains the chosen vertex on this edge.

step 2的一个效率更高的实现

保存 未选择节点的一个列表:
- UNCHSN (I), 初始包括 NUMUN = N – 1 个节点,

- 创建两个额外的数组:
  - LIGHT(I), 等于i*th* 未选择节点 和已选节点之间的最轻的边的费用值
  - VERTEX(I), 该条边上的已选择节点

- With these arrays step 2 of Algorithm B can be implemented as follows:

(1) search through the array LIGHT for the lightest edge, say, LIGHT(K);

(2) delete the K^th element from the UNCHSN list (by placing the last unchosen vertex UNCHSN(NUMUN) in UNCHSN(K));

(3) decrease the value of NUMUN= NUMUN-1;

(4) record the newly chosen LIGHT edge; and

(5) use old UNCHSN(K) as the newly chosen vertex, update the values of the remaining unchosen vertices by comparing LIGHT(I) with C(I, UNCHSN(K)).

- 用这些数组，算法 B 的step 2 可按如下实现

(1) 在数组 LIGHT里搜索最轻的边, 比如, LIGHT(K);

(2) 从UNCHSN 列表里删除第k个元素。 (通过把最后一个未选择节点UNCHSN(NUMUN) 放到UNCHSN(K) 来实现);

(3) NUMUN= NUMUN-1;

(4) 记录新选择的 LIGHT 边; 且

(5) 用原来的 UNCHSN(K) 作为新选择的节点, 通过比较 LIGHT(I) 和 C(I, UNCHSN(K))，更新剩下的未选择节点的LIGHT值.

Flow Chart of Algorithm C

1. Start
2. Initialization
3. Update lightest edge from each unchosen vertex to a chosen vertex
4. Pick a lightest edge e from an unchosen vertex to a chosen vertex
5. Add edge e to MST
6. Delete newly chosen vertex from unchosen list
7. IF No unchosen vertices left THEN Stop
8. GOTO 3

Programming----Variables

C (I, J)	THE WEIGHT OF THE EDGE BETWEEN VERITICES I AND J.
N	THE VERITICES OF G ARE NUMBERED 1, 2, …, N.
FROM (I)	THE I th EDGE IN THE MST IS FROM VERTEX
TO (I)	FROM (I) TO VERTEX TO (I) OF COST COST (I) .
COST (I)	
WEIGHT	TOTAL WEIGHT OF THE MST.
EDGES	NUMBERS OF EDGES IN MST SO FAR.
NEXT	NEXT VERTEX TO BE ADDED TO MST.
NUMUN	NUMBER OF UNCHOSEN VERTICES.
UNCHSN (I)	ARRAY OF UNCHOSEN VERTICES.
VERTEX (I)	VERTEX OF PARTIAL MST CLOSEST TO VERTEX UNCHSN (I).
LIGHT (I)	WEIGHT OF LIGHTEST EDGE BETWEEN THE Ith UNCHOSEN VERTEX AND A CHOSEN VERTEX. (weight of edge from VERTEX (I) to UNCHSN (I).)

Programming

Coding

Complete Development of an Algo.

- 1 Statement of the problem
- 2 Development of a Model
- 3 Design of the algorithm
- 4 Correctness of the algorithm
- 5 Implementation
- 6 Analysis and complexity of the algorithm
- 7 Program testing
- 8 Documentation

Analysis of Algorithm C

- Correctness
- Time complexity
  - A rough analysis

Flow Chart of Algorithm C

1. Start
2. Initialization
3. Update lightest edge from each unchosen vertex to a chosen vertex（a single vertex）
4. Pick a lightest edge e from an unchosen vertex to a chosen vertex
5. Add edge e to MST
6. Delete newly chosen vertex from unchosen list
7. IF No unchosen vertices left THEN Stop
8. GOTO 3

Analysis of Algorithm C

- Correctness
- Time complexity
  - A rough analysis----$O(N^2)$
  - A detailed analysis

Programming

Coding

Analysis of Algorithm C

- Correctness
- Time complexity
  - A rough analysis----$O(N^2)$
  - A detailed analysis
- Simplicity
- Optimality

Prim

- Algorithm B (C) is actually

  Prim’s Algorithm

Complete Development of an Algo.

- 1 Statement of the problem
- 2 Development of a Model
- 3 Design of the algorithm
- 4 Correctness of the algorithm
- 5 Implementation
- 6 Analysis and complexity of the algorithm
- 7 Program testing
- 8 Documentation

Program Testing

(1) for correctness,
(2) for implementation efficiency, and
(3) for computational complexity.

Taken together, these tests try to provide experimental answers to the questions:

- Does the algorithm work?
- How well does it work?

Complete Development of an Algo.

- 1 Statement of the problem
- 2 Development of a Model
- 3 Design of the algorithm
- 4 Correctness of the algorithm
- 5 Implementation
- 6 Analysis and complexity of the algorithm
- 7 Program testing
- 8 Documentation

During your implementation of

Algorithm C for MST

Something Happened

- You found another program runs faster than yours.

- You can’t believe it!
- Prim’s Algo is optimal!

- You are given the algorithm:

```
1.  $A \leftarrow \phi$
2.  for each vertex $v \in V[G]$
3.      do MAKE-SET ($v$)
4.  sort the edges of $E$ by nondecreasing weight $w$
5.  for each edge $(u, v) \in E$, in order by
    nondecreasing weight
6.      do if FIND-SET ($u$) $\neq$ FIND-SET ($v$)
7.              then $A \leftarrow A \cup \{(u, v)\}$
8.                   UNION($u$, $v$)
9.  Return $A$
```

End of Section